Lab02 Assignment's Report

Le Anh Thu[1], Nguyen Nhat Truyen[1], Nguyen Minh Dat[1], and Huynh Duc Thien[1]

[1]Faculty of Information Technology, University of Science

1 Team's Result

After dedicating several weeks to this lab assignment, our team's work reveal the following percentages in this Table 1.

	Thu Le	Truyen Nguyen	Dat Nguyen	Thien Huynh	**Total**
Text Cleaning and Term Frequency	50%	-	50%	-	100%
Low-Frequency Term Elimination	-	100%	-	-	100%
Top 10 Most Frequent Words	-	-	-	100%	100%
TF-IDF	-	-	50%	50%	100%
Highest average tfidf	-	-	100%	-	100%
K-Means on 2D Data	100%	-	-	-	100%
K-Means on Preprocessed Data	-	100%	-	-	100%
Scalable K-Means++ Initialization	-	-	-	-	0%
Latex Report	50%	-	-	50%	100%
Contribution	**200%**	**200%**	**200%**	**200%**	**800%**

Table 1. The contribution of our team in each task

2 Data Description

The dataset is provided by the BBC news website, corresponding to stories on 5 different topics in life between the years 2004 and 2005. The explored topics include: business, entertainment, politics, sport, and tech.

Within each provided folder, we will have samples corresponding to that topic, with each sample being saved in a **txt** file named as a natural number. Each topic will have approximately 400 to 550 samples. This will be the dataset we will interact with and explore throughout this lab.

bbc
  |— politics
  |— business
  |— entertainment
  |— sport
  |— tech

Table 2. Dataset file struture

3 Data Preprocessing

In this section, we will perform preprocessing on the data, while also extracting some insights from the data. We will digitize the documents to facilitate the execution of algorithms in the following section.

3.1 Text Cleaning and Term Frequency

3.1.1 Task Information

I/O Description:

- **Input:** Path of the directory containing the dataset
- **Output:** An MTX file with three columns (termid, docid, frequency)

Task Challenges: We encounter two difficulties in this task.

- **ID Assignment:** We have to be able to assign a correct id to a term/document.
- **Nested Directory Structure:** We only provide a path to the largest directory (/bbc) as job input. Therefore, we have to be able to walk through the directory and access the inner files.

3.1.2 Implementation

We implement two MapReduce programs for this task, specifically:

- **The first MapReduce program:** Read the bbc.terms file as input and output the file (termid.mtx) where each line contains a term and its id. Similarly, read the bbc.docs file and output the file (docid.mtx) where each line contains a document and its id.
- **The second MapReduce program:** Read and tokenize the text data, remove stopwords and count the number of each term in a document.

Implementation of the first MapReduce program:
We write two MapReduce jobs in this program: the first job assigns id to term and the second job assigns id to document. Since the logic to implement each job is identical, we shall explain the second one - assign id to document, which is slightly more difficult than the first. Let's start by taking a look at a fraction of the input file (bbc.docs):

bbc.docs

```
business.001
business.002
...
business.510
entertainment.001
entertainment.002
...
```

Each line in the file represents a document, where the first part (before the dot) is the class name and the second part (after the dot) is the document file name (without extension). An important characteristic we can observe is that the file name within each class always starts from 001. However, we expect the document id to increment regardless of the class. As a result, we cannot rely on the file name to assign id to document. Instead, we take advantage of the fact that the id of a document is actually the line number of that document. Therefore, we can set the document id to the line number. Here is a fraction of the expected output file (docid.mtx), which has another column to represent the document id:

docid.mtx

```
business.001        1
business.002        2
...
business.510        510
entertainment.001   511
entertainment.002   512
...
```

To implement the map function, we initialize an integer (lineCount) to store the line at which we are processing. Each time we read a new line, we increment the lineCount variable by one. The output <key, value> is of type <Text, IntWritable>. The key is the complete textual content of the input line, and the value is an integer to store the line number, which is also the document id.

```
public static class DocIDMapper extends Mapper<Object, Text, Text, IntWritable> {
    private static int lineCount = 1;
    @Override
    public void map(Object key, Text value, Context context) throws IOException,
        InterruptedException {
        String classAndDocId = value.toString().trim();
        context.write(new Text(classAndDocId), new IntWritable(lineCount++));
    }
}
```

The implementation of the reduce function may be trivial since we simply write the <key, value> pairs to the output file.

```
public static class DocIDReducer extends Reducer<Text, IntWritable, Text, IntWritable> {
    @Override
    public void reduce(Text key, Iterable<IntWritable> values, Context context)
            throws IOException, InterruptedException {
        for (IntWritable value : values) {
            context.write(key, value);
        }
    }
}
```

Implementation of the second MapReduce program:
We need one MapReduce job for this program. The map function receives text data as input and output <term, 1> pairs. However, there are two important details we need to pay attention to:

- We consider the frequency of a term within a specific document, as a result, we also have to specify the document in the key. Therefore, the actual format of key-value pairs emitted by the map function are <term, doc, 1>.
- We do not provide a single text file as job input, but rather a path to a nested directory (/bbc). Therefore, we need to implement a setup() function to specify the necessary information (class name and doc id), and load data from the input files (stopwords.txt, termid.mtx and docid.mtx) into suitable data structures.

To tokenize the text input, we split text based on whitespace (space, tab, newline,...) and the underscore _ characters. We also lowercase all letters in each token, and throw away all but letters and digits. Then we check whether or not the resulting token is a stopwords. A valid token can be mapped into a term id, and combined with a doc id to form a key in the key-value pair (the value is 1).

```
public static class WordMapper extends Mapper<LongWritable, Text, Text, LongWritable> {
    private final static LongWritable one = new LongWritable(1);
    private String classAndDoc;
    private Set<String> stopWords = new HashSet<>();
    private static HashMap<String, Integer> wordIdMap = new HashMap<>();
    private static HashMap<String, Integer> docIdMap = new HashMap<>();

    @Override
    protected void setup(Context context) throws IOException, InterruptedException {
        String className = ((FileSplit)
            context.getInputSplit()).getPath().getParent().getName();
        String docName = ((FileSplit) context.getInputSplit()).getPath().getName();
        String[] docNameParts = docName.split("\\.");
        String docId = docNameParts[0];
        classAndDoc = className + "." + docId;

        Configuration conf = context.getConfiguration();
        Path[] cacheFiles = context.getLocalCacheFiles();
```

```
        String line;

        BufferedReader readerStopWords = new BufferedReader(new
            InputStreamReader(FileSystem.getLocal(conf).open(cacheFiles[0])));
        while ((line = readerStopWords.readLine()) != null) {
            stopWords.add(line.trim());
        }
        readerStopWords.close();

        BufferedReader readerWordIdMap = new BufferedReader(new
            InputStreamReader(FileSystem.getLocal(conf).open(cacheFiles[1])));
        while ((line = readerWordIdMap.readLine()) != null) {
            String[] parts = line.split("\\s+");
            wordIdMap.put(parts[0], Integer.parseInt(parts[1]));
        }
        readerWordIdMap.close();

        BufferedReader readerDocIdMap = new BufferedReader(new
            InputStreamReader(FileSystem.getLocal(conf).open(cacheFiles[2])));
        while ((line = readerDocIdMap.readLine()) != null) {
            String[] parts = line.split("\\s+");
            docIdMap.put(parts[0], Integer.parseInt(parts[1]));
        }
        readerDocIdMap.close();
    }

    @Override
    protected void map(LongWritable key, Text value, Context context)
            throws IOException, InterruptedException {

        StringTokenizer tokenizer = new StringTokenizer(value.toString(), " \\t\\n\\r\\f_");
        while (tokenizer.hasMoreTokens()) {
            String token = tokenizer.nextToken();
            String word = token.replaceAll("[^a-zA-Z0-9]", "").toLowerCase(); //

            if (!stopWords.contains(word) && wordIdMap.containsKey(word)) {
                int termId = wordIdMap.get(word);
                int docId = docIdMap.get(classAndDoc);
                Text keyText = new Text(termId + "\\t" + docId);
                context.write(keyText, one);
            }
        }
    }
}
```

The implementation of the reduce function is straightforward since we simply need to add all the 1s together to obtain the frequency of a term in a document. Note that the key already contains information about the term id and doc id!

```
public static class WordReducer extends Reducer<Text, LongWritable, Text, LongWritable> {
    @Override
    protected void reduce(Text key, Iterable<LongWritable> values, Context context)
            throws IOException, InterruptedException {
        long count = 0;
        for (LongWritable value : values) {
            count += value.get();
        }
        context.write(key, new LongWritable(count));
    }
}
```

Run on Hadoop

We provide two Java programs, including `IDAssignment.java` and `TermDocFreq.java`. To operate them on Hadoop, we follow these steps:

- **Step 1:** We create a directory named `IDAssignment`, and put two files `bbc.terms` and `bbc.docs` into it.

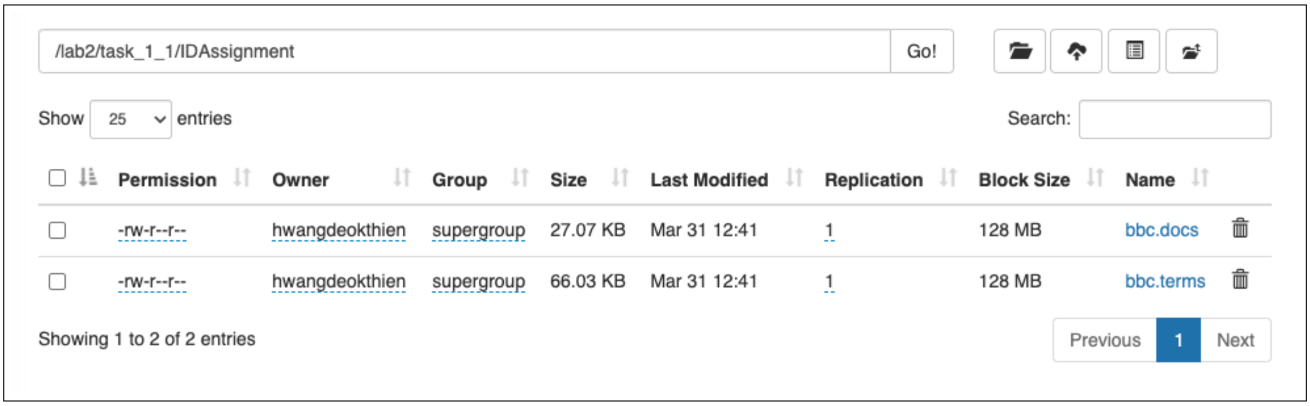


Fig. 1. Load files for Assigning ID job

- **Step 2:** We run the first program using this command

```
$ hadoop jar IDAssignment.jar IDAssignment
/lab2/task_1_1/IDAssignment/bbc.terms /lab2/task_1_1/IDAssignment/termOut
/lab2/task_1_1/IDAssignment/bbc.docs /lab2/task_1_1/IDAssignment/docOut
```

  Here, we've passed in four arguments, the first and the second are the input path and output path of the first job, the third and the fourth are for the second job in the program.

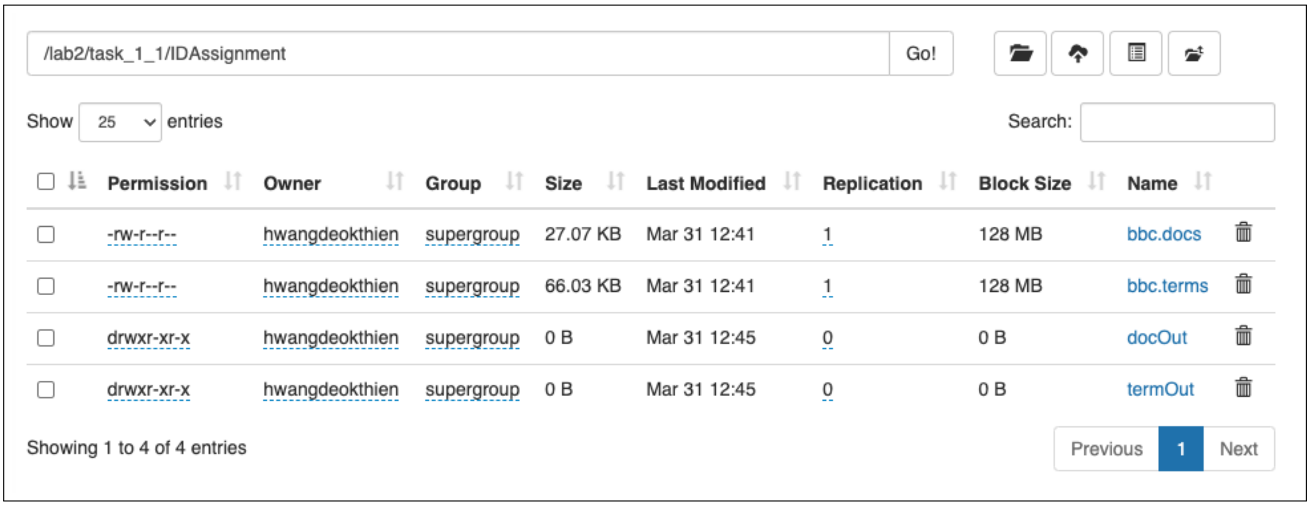


Fig. 2. Output folders of Assigning ID job

- **Step 3:** Copying the output files of first program, rename them into `termid.mtx` and `docid.mtx`. Put those files in `TermDocFreq` directory to run the second program. We also add the `bbc` dataset, and the `stopwords.txt` into the directory.

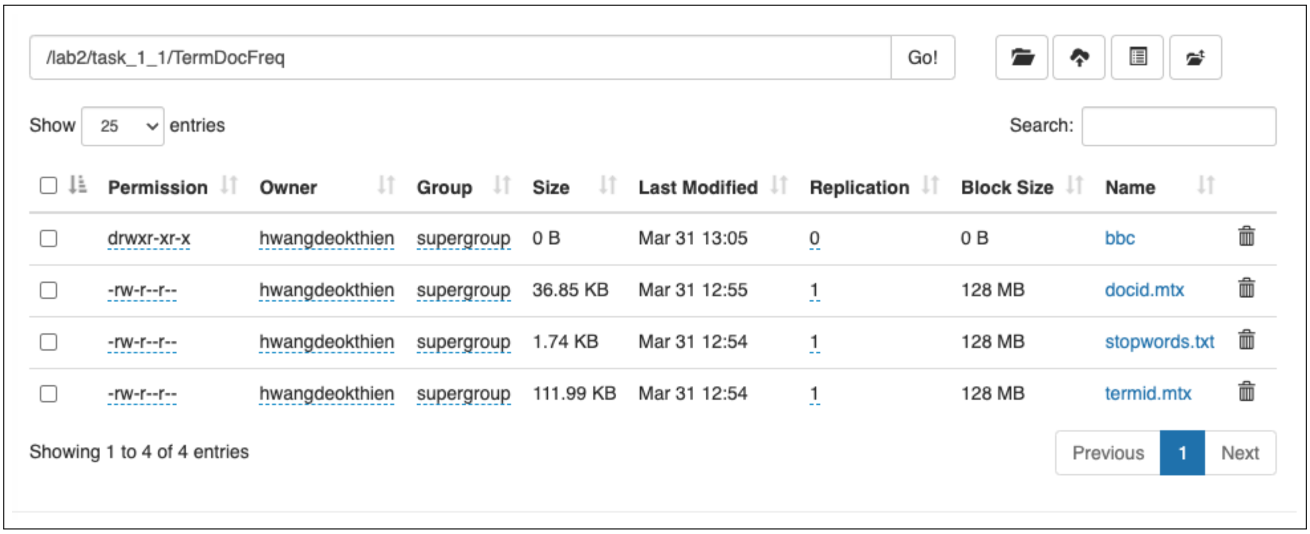


Fig. 3. Input for second program

- **Step 4:** Running the program using this command:

```
$ hadoop jar TermDocFreq.jar TermDocFreq
/lab2/task_1_1/TermDocFreq/bbc /lab2/task_1_1/TermDocFreq/output
/lab2/task_1_1/TermDocFreq/stopwords.txt
/lab2/task_1_1/TermDocFreq/termid.mtx
/lab2/task_1_1/TermDocFreq/docid.mtx
```

There are total 5 argument for each input, output files and folders for the program.

Output of task 1.1

```
1   1    1
1   11   1
1   1199 1
...
997 952  1
997 956  2
997 982  2
```

3.2 Low-Frequency Term Elimination

3.2.1 Task Information

I/O Description:

- **Input:** MTX file generated from Task 1.1
- **Output:** MTX file with terms having a frequency least than 3 removed

Task Challenge: With the premises established in task 1.1, the task 1.2 becomes relatively simple, as we only need to iterate through the terms and remove those with a frequency less than 3. However, the challenge here lies in finding the total frequency across all documents.

3.2.2 Implementation

Since we want to calculate the frequency among all documents; in the Map function, we will set the *termid* as the key, so that when the data going to the Reduce stage, all pair that have the same key (same *termid*) will be united together.

Mapper

```java
public static class TokenizerMapper
    extends Mapper<LongWritable, Text, LongWritable, Text> {
    private LongWritable outputKey = new LongWritable();
    private Text outputValue = new Text();

    public void map(LongWritable key, Text value, Context context)
        throws IOException, InterruptedException {

        StringTokenizer itr = new StringTokenizer(value.toString());
        int termid = Integer.parseInt(itr.nextToken());
        String docid = itr.nextToken();
        float frequency = Float.parseFloat(itr.nextToken());

        outputKey.set(termid);
        outputValue.set(docid + "\t" + frequency);
        context.write(outputKey, outputValue);
    }
}
```

At the Reduce phase, we will sum up all the frequency of each term from all documents, the comparison will then be operated, and the term that has least than 3 appearance will be removed.

Reducer

```java
public static class FloatSumReducer
    extends Reducer<LongWritable, Text, LongWritable, Text> {
    private Text outputValue = new Text();

    public void reduce(LongWritable key, Iterable<Text> values, Context context)
        throws IOException, InterruptedException {

        float sum = 0;
        List<String> listStr = new ArrayList<String>();
        for (Text val : values) {
            StringTokenizer itr = new StringTokenizer(val.toString(), "\t");
            String docid = itr.nextToken();
            float frequency = Float.parseFloat(itr.nextToken());
            sum += frequency;
            listStr.add(docid + "\t" + frequency);
        }
        if (sum >= 3) {
            for (String val : listStr) {
                outputValue.set(val);
                context.write(key, outputValue);
            }

        }
    }
}
```

Run on Hadoop

To run this on Hadoop, we need to put the **mtx** file from task 1.1 to the input folder, then using this command:

```
$ hadoop jar TermFrequencyFilter.jar TermFrequencyFilter /lab2/task_1_2/input
/lab2/task_1_2/output
```

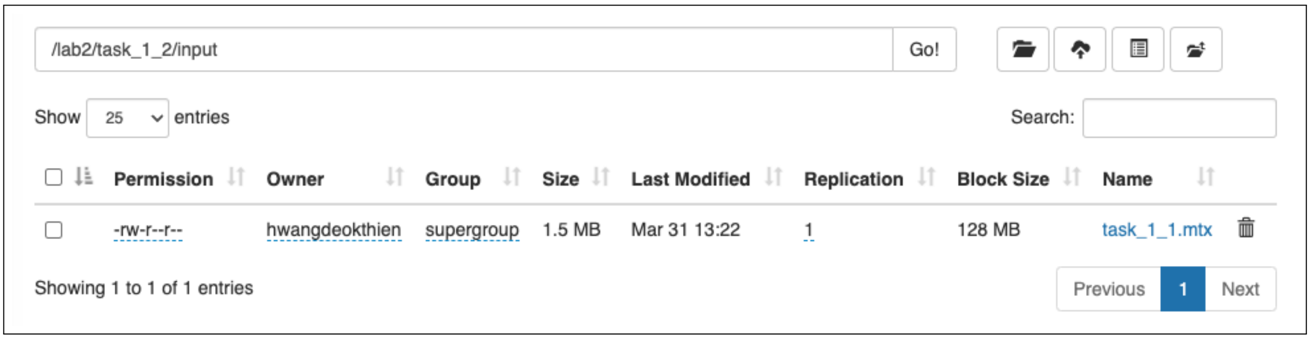


Fig. 4. Input file for Task 1.2

Output task 1.2

```
1   1962  1.0
1   2033  1.0
1   2134  1.0
...
9632  2215  2.0
9632  2194  1.0
9632  2110  2.0
```

The contents of the three columns in the output file respectively correspond to *termid*, *docid*, *frequency*. A detailed output file will be included with this report.

3.3 Top 10 Most Frequent Words

3.3.1 Task Information

I/O Description:

- **Input:** MTX file generated from Task 1.2
- **Output:** List of the top 10 most frequently occurring terms along with their frequencies.

Task Challenge: We encounter two difficulties in this task.

- **Total frequency:** to select out the top values, we first have to calculate the term frequency among documents, this job is pretty similar to which we do in the previous task.
- **Filter the top values:** chosing a suitable solution for this is not easy, since sorting over a very long list of values is not a good idea.

3.3.2 Implementation

To solving this task, we have to do 2 seperate MapReduce jobs in a single program. The first one is to calculate the total frequency of each term, the second job is to select out the top 10 term with highest frequency.

The first job is pretty similar to the previous task, but the difference here is after we have calculated the frequency, we'll save the result to a file for future usage with the role of input file for the second job.

Mapper of sencond job

```java
public static class AssignMapper
        extends Mapper<LongWritable, Text, Text, Text> {
    private Text outputKey = new Text("header");
    private Text outputValue = new Text();

    public void map(LongWritable key, Text value, Context context)
            throws IOException, InterruptedException {
```

```
        Text termFreq = new Text(Long.toString(key.get()) + " " + value.toString());

        outputValue.set(termFreq);
        context.write(outputKey, outputValue);
    }
}
```

In the Map phase of second job, we will extract the term and the frequency collected from the first job. To perform the sorting, filtering mechanism in all terms, it have to go to the same Reducer, which mean they have to have the same key.
Here I set all the key to the same String `'header'` (it does not matter the content, it just has to be the same among all) and the value will be a String containing *termid* and *frequency*. For example `'header 1 190.0'`.

Reducer of sencond job

```
public static class SortingReducer
        extends Reducer<Text, Text, Text, Text> {
    private Text outputKey = new Text();
    private Text outputValue = new Text();

    public void reduce(Text key, Iterable<Text> values, Context context)
            throws IOException, InterruptedException {

        TreeMap<Float, String> topValues = new TreeMap<>();

        for (Text val : values) {
            String[] token = val.toString().split("\\s+");

            String term = token[0];
            float freq = Float.parseFloat(token[1]);

            topValues.put(freq, term);

            if (topValues.size() > 10) {
                topValues.pollFirstEntry();
            }
        }

        for (Map.Entry<Float, String> entry : topValues.entrySet()) {
            Text term = new Text(entry.getValue());
            Text freq = new Text(Float.toString(entry.getKey()));

            outputKey.set(term);
            outputValue.set(freq);

            context.write(outputKey, outputValue);
        }
    }
}
```

In the Reduce phase, we use a data structure called **TreeMap**[1] to store the top 10 highest frequency term. The idea of this algorithm:

- **Step 1:** Adding every single pair of term and frequency to the tree until it have 10 elements.
- **Step 2:** Add the next pair into tree, the tree now contains 11 elements.
- **Step 3:** Sorting 11 elements (the tree has already do itself).

- **Step 4:** Take out the least frequent term among 11 elements
- **Step 5:** Repeat the process from step 2 until there's no value left in the list.

After run this, we will get the top 10 values as a result.

The advantage of this method, is that we don't have to perform sorting over a very long array, but instead a much smaller one, this will reduce the memory usage and time consumption during the process.

Run on Hadoop
To run this program on Hadoop, we need the input file **task_1_2.mtx** putted on HDFS, and using the following command:

```
$ hadoop jar MostFrequencyFilter.jar MostFrequencyFilter
/lab2/task_1_3/input /lab2/task_1_3/middle /lab2/task_1_3/output
```

There are three arguments in the command, represent the path of *input*, *middle*, *output* folders respectively. The *middle* folder will store the output of the first job.

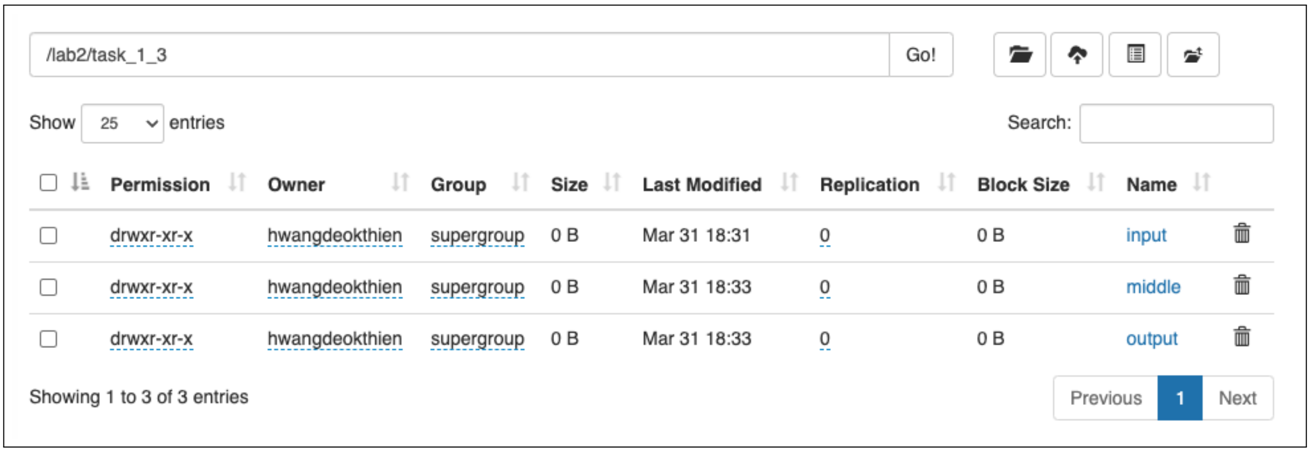


Fig. 5. File structure of Task 1.3

Top 10 highest frequent terms among all documents

1009167	1845217.0
1009182	1845231.0
1009197	1845245.0
1009212	1845259.0
1009227	1845273.0
1009242	1845287.0
1009257	1845301.0
1009272	1845315.0
1009287	1845329.0
1009315	1845343.0

3.4 TF-IDF

3.4.1 Task Information

I/O Description:

- **Input:** MTX file generated from Task 1.2
- **Output:** Calculate the TF-IDF score for each term-document pair.

3.4.2 Implementation

We implement two MapReduce jobs for this task, specifically:

- **The first MapReduce job:** Calculate the TF score for each term-document pair.
- **The second MapReduce job:** Calculate the IDF score for each term across all documents, and combine with the TF computed in the first job to calculate the final TF-IDF score.

Implementation of the first MapReduce job:

The map function reads as input the MTX file from Task 1.2, where each line contains three numbers: term id, doc id, and frequency. Since our goal is to compute the TF score for a term in a document, we need to know two things:

- **The total number of occurrences of a term in a document:** Fortunately, this quantity is already provided in the input file!
- **The total number of terms in a document:** This is what we need to compute. In the map function, we set the key to be the doc id, and the value contains the term id and frequency.

```java
public static class FirstMapper extends Mapper<Object, Text, Text, Text> {
    private Text documentId = new Text();
    private Text termIdAndFrequency = new Text();

    public void map(Object key, Text value, Context context)
          throws IOException, InterruptedException {
        String[] parts = value.toString().split("\\s+");
        String termId = parts[0];
        String docId = parts[1];
        String freq = parts[2];

        documentId.set(docId);
        termIdAndFrequency.set(termId + " " + freq);
        context.write(documentId, termIdAndFrequency);
    }
}
```

The reduce function sums up the frequencies of all the terms in a document to compute the total number of terms that the document has. Then it loops through each term and compute the TF score for the term-document pair by dividing the term frequency by the total number of terms. The key is set to the combination of term id and doc id, while the value is the TF score of that term-doc pair. There are two things worth mentioning in this reduce phase:

- We can perform only one loop through the input Iterable<Text> values. Since we use that loop to compute the total number of terms, we have to define a data structure to store values for a second loop. In this case, we use the List<Text> valueList.
- We can take advantage of the fact that each reduce task has a doc id as an input key. If we let each reduce task add its doc id to a set, then we can obtain a set of all the doc ids once all the reduce tasks are completed. This set is valuable since we can easily compute the total number of documents by getting the size of the set. That is the reason why we implement a cleanup() function, where we store the set size into a shared variable (numDocs), which will be used by the second MapReduce job to compute IDF scores.

```java
public static class FirstReducer extends Reducer<Text, Text, Text, DoubleWritable> {
    private Text new_key = new Text();
    private DoubleWritable result = new DoubleWritable();
    private Set<String> uniqueDocIDs = new HashSet<>();
```

```
    public void reduce(Text key, Iterable<Text> values, Context context)
            throws IOException, InterruptedException {

        uniqueDocIDs.add(key.toString());
        double totalWords = 0;
        List<Text> valueList = new ArrayList<>();

        for (Text val : values) {
            String[] parts = val.toString().split(" ");
            double freq = Double.parseDouble(parts[1]);
            totalWords += freq;
            valueList.add(new Text(val));
        }

        for (Text val : valueList) {
            String[] parts = val.toString().split(" ");
            String termId = parts[0];
            double freq = Double.parseDouble(parts[1]);
            double tf = freq / totalWords;

            new_key.set(termId + "@" + key);
            result.set(tf);
            context.write(new_key, result);
        }
    }

    @Override
    protected void cleanup(Context context) throws IOException, InterruptedException {
        numDocs = uniqueDocIDs.size();
        context.getCounter("CustomCounters", "NUM_DOCS").setValue(numDocs);
    }
}
```

Implementation of the second MapReduce job:
After computing the TF score for each term-doc pair, our goal is to compute the IDF score for each term across all documents. Therefore, we need to know two things:

- **The total number of documents:** This quantity is already computed by the first job!
- **The total number of documents containing the term:** This is what we need to compute. In the map function, we set the key to be the term id, while the value contains the doc id and TF score.

```
public static class SecondMapper extends Mapper<Object, Text, Text, Text> {
    public void map(Object key, Text value, Context context)
            throws IOException, InterruptedException {
        String[] parts = value.toString().split("\\s+");
        String termIdAndDocId = parts[0];
        String tf = parts[1];
        String[] termIdAndDocIdParts = termIdAndDocId.split("@");
        String termId = termIdAndDocIdParts[0];
        String docId = termIdAndDocIdParts[1];

        Text termIdText = new Text(termId);
        Text docIdAndTF = new Text(docId + " " + tf);
        context.write(termIdText, docIdAndTF);
    }
}
```

The reduce function aggregates all the documents for a term, therefore obtaining the total number of documents that the term occurs in. The real challenge lies in sharing the total number of documents (stored in the numDocs variable) across two jobs. In the first job, the numDocs variable is updated in the cleanup() method. This numDocs value is then used to set a custom counter using context.getCounter() method. This counter is retrieved later during the second job to obtain the total number of documents. The reduce function in the second job can then compute the IDF score for each term across all documents, and combine with the TF score calculated by the first job to output the TF-IDF score for each term-doc pair.

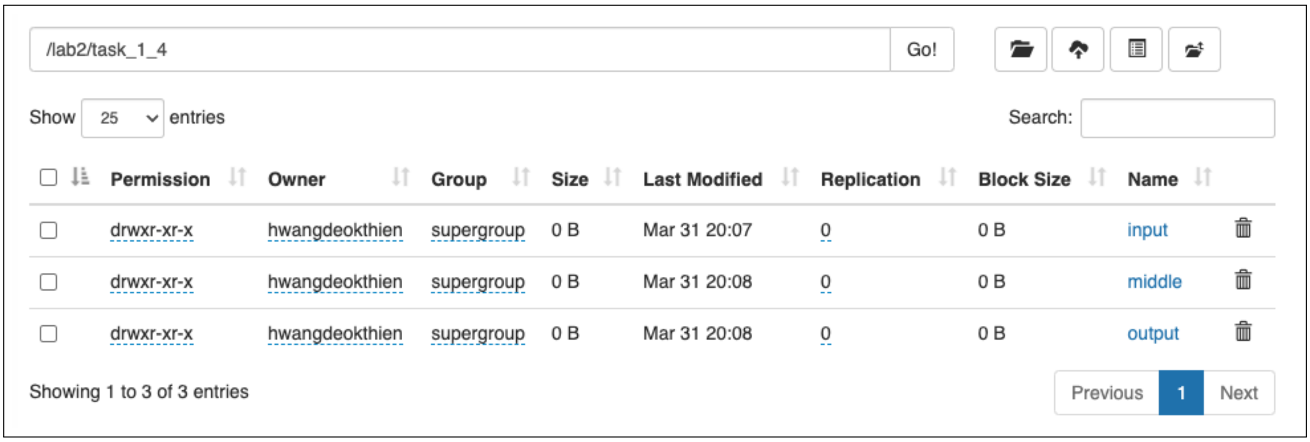
```java
public static class SecondReducer extends Reducer<Text, Text, Text, DoubleWritable> {
    private Text new_key = new Text();

    @Override
    protected void setup(Context context) throws IOException, InterruptedException {
        numDocs = context.getConfiguration().getInt("numDocs", 0);
    }

    public void reduce(Text key, Iterable<Text> values, Context context
                ) throws IOException, InterruptedException {
        int numDocOccurrences = 0;
        List<String> valueList = new ArrayList<>();

        for (Text val : values) {
            numDocOccurrences++;
            valueList.add(val.toString());
        }

        for (String val : valueList) {
            String[] parts = val.split(" ");
            String docId = parts[0];
            double tf = Double.parseDouble(parts[1]);
            double idf = Math.log((double) numDocs / numDocOccurrences);
            DoubleWritable tf_idf = new DoubleWritable(tf * idf);

            new_key.set(key + " " + docId);
            context.write(new_key, tf_idf);
        }
    }
}
```

Run on Hadoop

For this task, we reuse `task_1_2.mtx` from the previous task for input file. With the provided program `TFTDF.java`, we compile it to `jar` file and run on hadoop using this command:

```
$ hadoop jar TFIDF.jar TFIDF /lab2/task_1_4/input
/lab2/task_1_4/middle /lab2/task_1_4/output
```

Similar to the previous task, this command also take three arguments withc `middle` folder for storing file transacting between two jobs.



	Permission	Owner	Group	Size	Last Modified	Replication	Block Size	Name
	drwxr-xr-x	hwangdeokthien	supergroup	0 B	Mar 31 20:07	0	0 B	input
	drwxr-xr-x	hwangdeokthien	supergroup	0 B	Mar 31 20:08	0	0 B	middle
	drwxr-xr-x	hwangdeokthien	supergroup	0 B	Mar 31 20:08	0	0 B	output

Fig. 6. Folders of task 1.4

Output of task 1.4

```
1    1833     0.01735772214035987
1    1        0.04873514293254887
1    693      0.05569730620862728
...
997  170      0.06889232324783914
997  923      0.02855405503035438
997  1276     0.006507071011415092
```

3.5 Highest average tfidf

3.5.1 Task Information

I/O Description:

- **Input:** MTX file from Task 1.4.
- **Output:** Write 5 terms with the highest average TF-IDF score for each class.

3.5.2 Implementation

We implement two MapReduce jobs for this task, specifically:

- **The first MapReduce job:** Read the MTX file from Task 1.4 as input and output the average TF-IDF score for each term-class pair.
- **The second MapReduce job:** Take the output from the first job and write 5 terms with the highest average TF-IDF score for each class.

Implementation of the first MapReduce job:

Each line in the input file (task_1_4.mtx) only contains three information: term id, doc id and TF-IDF score. To know the class information, we rely on the docid.mtx file obtained from section 3.1. As a result, we implement a setup() function to read the docid.mtx file, and store data in a HashMap which maps doc id into corresponding class name. The map function can then use this HashMap to retrieve a correct class name given a doc id, and set the emitted key to be the combination of term id and class name, while the value contains doc id and TF-IDF score. An important detail related to implementation is that we keep the class names the same without converting them into corresponding class ids for the highly readable final output.

```
public static class FirstMapper extends Mapper<Object, Text, Text, Text>{
    private Text termIdAndClassName = new Text();
    private Text docIdAndTFIDF = new Text();
    private static HashMap<Integer, String> docIdToClassName = new HashMap<>();
```

```java
    @Override
    protected void setup(Context context) throws IOException, InterruptedException {
        Configuration conf = context.getConfiguration();
        Path[] cacheFiles = context.getLocalCacheFiles();
        String line;

        BufferedReader reader = new BufferedReader(new
            InputStreamReader(FileSystem.getLocal(conf).open(cacheFiles[0])));
        while ((line = reader.readLine()) != null) {
            String[] parts = line.split("\\s+");
            String className = parts[0].split("\\.")[0];
            int docId = Integer.parseInt(parts[1]);
            docIdToClassName.put(docId, className);
        }
        reader.close();
    }

    public void map(Object key, Text value, Context context) throws IOException,
        InterruptedException {

        String[] parts = value.toString().split("\\s+");
        String termId = parts[0];
        int docId = Integer.parseInt(parts[1]);
        String tf_idf = parts[2];
        String className = docIdToClassName.get(docId);

        termIdAndClassName.set(termId + " " + className);
        docIdAndTFIDF.set(docId + " " + tf_idf);
        context.write(termIdAndClassName, docIdAndTFIDF);
    }
}
```

Once the map phase is completed, each reduce function receives a single pair of <term id - class name> as a key, and a list of <doc id - TF-IDF score> as values. Notice that at this point, we no longer need the doc id information, since we are only interested in computing the average TF-IDF score for the <term id - class name> pair. The reduce function can then compute the average TF-IDF score, and emit the key as the <term id - class name> pair, while the value is the average TF-IDF score.

```java
public static class FirstReducer extends Reducer<Text, Text, Text, DoubleWritable> {
    public void reduce(Text key, Iterable<Text> values, Context context) throws
        IOException, InterruptedException {

        double sumTFIDF = 0;
        int numTFIDF = 0;

        for (Text val : values) {
            String[] parts = val.toString().split(" ");
            String docId = parts[0];
            String tf_idf = parts[1];

            sumTFIDF += Double.parseDouble(tf_idf);
            numTFIDF++;
        }

        DoubleWritable averageTFIDF = new DoubleWritable(sumTFIDF / numTFIDF);
        context.write(key, averageTFIDF);
    }
}
```

Implementation of the second MapReduce job:

The second job reads as input the output from the first job, where each line represents the term id, class name, and the average TF-IDF scores. Our goal is to find the top 5 terms with the highest average TF-IDF score for each class. Therefore, the map function should set the key as the class name, while the value is the term id and TF-IDF score.

```java
public static class SecondMapper extends Mapper<Object, Text, Text, Text>{
    private Text classNameText = new Text();
    private Text termIdAndAvgTFIDF = new Text();

    public void map(Object key, Text value, Context context) throws IOException,
        InterruptedException {

        String[] parts = value.toString().split("\\s+");
        String termId = parts[0];
        String className = parts[1];
        String avgTFIDF = parts[2];

        classNameText.set(className);
        termIdAndAvgTFIDF.set(termId + " " + avgTFIDF);
        context.write(classNameText, termIdAndAvgTFIDF);
    }
}
```

The reduce phase receives a single class name as a key, and a list of <term id - average TF-IDF score> as values. To find the top 5 terms with the highest average TF-IDF scores, we leverage the TreeMap data structure, where we only keep five elements within the TreeMap at a time. There is a minor detail related to implementation of a setup() function. Since we want to output the term itself instead of the term id, while we only receive the term id as input, we have to read the termid.mtx file and load data into a HashMap which maps a term id into a correct term.

```java
public static class SecondReducer extends Reducer<Text, Text, Text, Text> {
    private static HashMap<Integer, String> termIdToTerm = new HashMap<>();

    @Override
    protected void setup(Context context) throws IOException, InterruptedException {
        Configuration conf = context.getConfiguration();
        Path[] cacheFiles = context.getLocalCacheFiles();
        String line;

        BufferedReader reader = new BufferedReader(new
            InputStreamReader(FileSystem.getLocal(conf).open(cacheFiles[0])));
        while ((line = reader.readLine()) != null) {
            String[] parts = line.split("\\s+");
            String term = parts[0]; // yeah
            int termId = Integer.parseInt(parts[1]);
            termIdToTerm.put(termId, term);
        }
        reader.close();
    }

    public void reduce(Text key, Iterable<Text> values, Context context
                ) throws IOException, InterruptedException {

        TreeMap<Double, Text> top5 = new TreeMap<>();

        for (Text val : values) {
            String[] parts = val.toString().split(" ");
            int termId = Integer.parseInt(parts[0]);
            double avgTFIDF = Double.parseDouble(parts[1]);
            String term = termIdToTerm.get(termId);
```

```java
            top5.put(avgTFIDF, new Text(term + ": " + String.format("%.3f", avgTFIDF)));
            if (top5.size() > 5) {
                top5.remove(top5.firstKey());
            }
        }

        StringBuilder entryValues = new StringBuilder();
        for (Map.Entry<Double, Text> entry : top5.descendingMap().entrySet()) {
            if (entryValues.length() > 0) {
                entryValues.append(", ");
            }
            entryValues.append(entry.getValue());
        }
        Text newKey = new Text(key.toString() + ": ");
        context.write(newKey, new Text(entryValues.toString()));
    }
}
```

Run on Hadoop

In this task, we use the output files `task_1_4.mtx` of task 1.4, `termid.mtx` and `docid.mtx` from task 1.1.

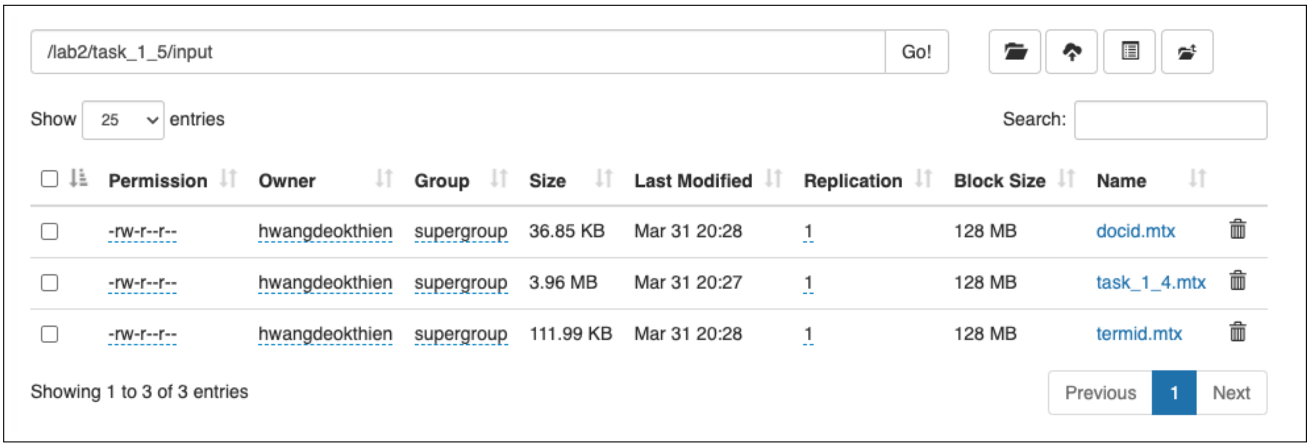


Fig. 7. Input files for task 1.5

Next, run the `HighestAverageTFIDF.java` program using the command:

```
$ hadoop jar HighestAverageTFIDF.jar HighestAverageTFIDF
/lab2/task_1_5/input/task_1_4.mtx /lab2/task_1_5/middle /lab2/task_1_5/output
/lab2/task_1_5/input/docid.mtx /lab2/task_1_5/input/termid.mtx
```

We pass in path for input files, middle folder and output folder as arguments.

Output of task 1.5

```
business:  nortel: 0.964, fsa: 0.754, circuit: 0.706, titan: 0.656, gun: 0.643
entertainment:  walmart: 0.956, carson: 0.922, aguilera: 0.640, austria: 0.610, bach: 0.575
politics:  foster: 0.640, swansea: 0.592, mock: 0.578, archer: 0.575, roma: 0.518
sport:  mutu: 0.744, hamm: 0.597, mansfield: 0.526, glazer: 0.518, mcclaren: 0.483
tech:   p2p: 0.873, rfid: 0.666, cabir: 0.558, printer: 0.524, wong: 0.511
```

4 K-Means Algorithm

We will apply K Means clustering algorithms to the 2D data and the processed data. More specifically, this will be elaborated on below.

4.1 K-Means on 2D Data

4.1.1 Task Information

I/O Description:

- **Input:** A sample file containing 2D data points (one point per line, separated by spaces)
- **Output:** Cluster assignments for each data point

4.1.2 Implementation

Generally, we will follow these requires in this section:

- Define the desired number of clusters k.
- Implement the K-Means algorithm:
  - Initialize k centroids randomly.
  - Assign each data point to the closest centroid based on Euclidean distance.
  - Recompute centroids as the mean of points assigned to each cluster.
  - Repeat steps 2.2 and 2.3 until convergence (centroids no longer change significantly).
- Run the K-Means with $K = 3$ for 20 iterations. Output the clusters centers to `task_2_1.clusters` and cluster assignment for each data point to `task_2_1.classes` in text format.

Now, we will go into detail:

- We need to preserve the results of each previous run in order to apply the KMeans algorithm. The `Task_2_1.clusters` file serves as storage for the centroid values of clusters calculated in the previous iteration. For the first time, when the `Task_2_1.clusters` file hasn't been initialized yet, the `initializeRandomCentroids()` function is called to randomly generate initial clusters. For subsequent runs, the `readCentroidsFromFile()` function is invoked to read the `Task_2_1.clusters` file, along with `ensureEnoughCentroids()` to ensure that there are enough centroids for the later stage.

setup function for collect centroids information

```
protected void setup(Context context) throws IOException, InterruptedException
    {
    Configuration conf = context.getConfiguration();
    FileSystem fs = FileSystem.get(conf);
    Path centroidPath = new Path("task_2_2.clusters");

    if (!fs.exists(centroidPath)) {
        initializeRandomCentroids();
    } else {
        readCentroidsFromFile(fs, centroidPath);
        ensureEnoughCentroids();
    }
}
```

- Map: With the key-value pair being `cluster_id` and a string containing the coordinates (x, y) of a point, the map function facilitates assigning a new cluster for a point based on Euclidean distance.

Map function

```
protected void map(Object key, Text value, Context context) throws
    IOException, InterruptedException {
    if (value.toString().startsWith("class")) {
        return;
```

```java
        }
        double[] vectorTf_Idf_Double = new double[VECTOR_SIZE];
        Arrays.fill(vectorTf_Idf_Double, 0.0);

        String[] parts = value.toString().split("\\s+");
        int docIdInt = Integer.parseInt(parts[0]);
        String vectorTf_Idf = parts[1];
        String[] vector_string = parts[1].split(",");

        for (String termStr : vector_string) {
            String[] term = termStr.split(":");
            int term_id = Integer.parseInt(term[0]);
            double tf_idf = Double.parseDouble(term[1]);
            vectorTf_Idf_Double[term_id] = tf_idf;
        }
        double minDistance = Double.MAX_VALUE;
        int closestCentroidIndex = -1;

        for (int i = 0; i < K; i++) {
            double[] centroid = centroids.get(i);
            double distance = cosineSimilarity(vectorTf_Idf_Double, centroid);
            if (distance < minDistance) {
                minDistance = distance;
                closestCentroidIndex = i;
            }
        }

        context.write(new Text(String.valueOf(closestCentroidIndex)), value);
    }
```

- Reduce: From the key-value pairs mapped above, recalculate the centroids of each cluster by averaging the x, y values relative to the centroid of the cluster containing that point.

Reduce function

```java
    protected void reduce(Text key, Iterable<Text> values, Context context) throws
        IOException, InterruptedException {

        double[][] sum = new double[VECTOR_SIZE][K];
        int[] count = new int[K];
       double[] vectorTf_Idf_Double = new double[VECTOR_SIZE];
        for (Text value : values) {
            int centroidIndex = Integer.parseInt(key.toString());
            Arrays.fill(vectorTf_Idf_Double, 0.0);

            String[] parts = value.toString().split("\\s+");
            String vectorTf_Idf = parts[1];
            String[] vector_string = vectorTf_Idf.split(",");

            for (String termStr : vector_string) {
                String[] term = termStr.split(":");
                int term_id = Integer.parseInt(term[0]);
                double tf_idf = Double.parseDouble(term[1]);
                vectorTf_Idf_Double[term_id] = tf_idf;
                sum[term_id][centroidIndex] += tf_idf;
            }
            count[centroidIndex]++;
        }

        for (int i = 0; i < K; i++) {
```

```
            if (count[i] > 0) {
                String str = "";
                double[] newCentroid = new double[VECTOR_SIZE];
                for (int j = 0; j < VECTOR_SIZE; j++) {
                    newCentroid[j] = sum[j][i] / count[i];
                    str = str + "," + newCentroid[j];
                }
        double loss = calculateLoss(newCentroid, vectorTf_Idf_Double);
        totalLoss += loss;
        context.getCounter(LossCounter.TOTAL_LOSS).increment((long) (loss * 1000));
                context.write(key, new Text(str));
            }
        }

    }
```

4.1.3 Output

- The file `task_2_1.clusters` can be found in the local directory on your computer after running the `KMeansMapReduce.java` program.

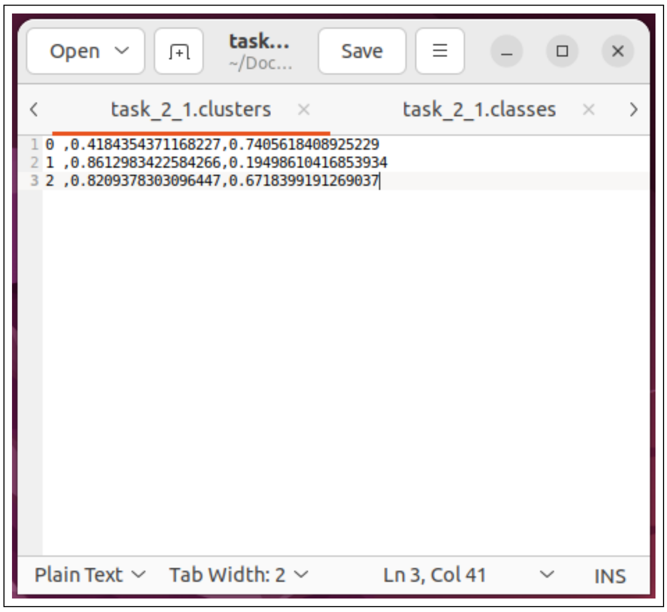


Fig. 8. Clusters information

- The file `task_2_1.classes` is located on the HDFS in the `output` directory with the name `part-r-00000` after running `AssignCluster.java`. The result after running `AssignCluster.java` provides us with the output where data points have been assigned to new clusters.

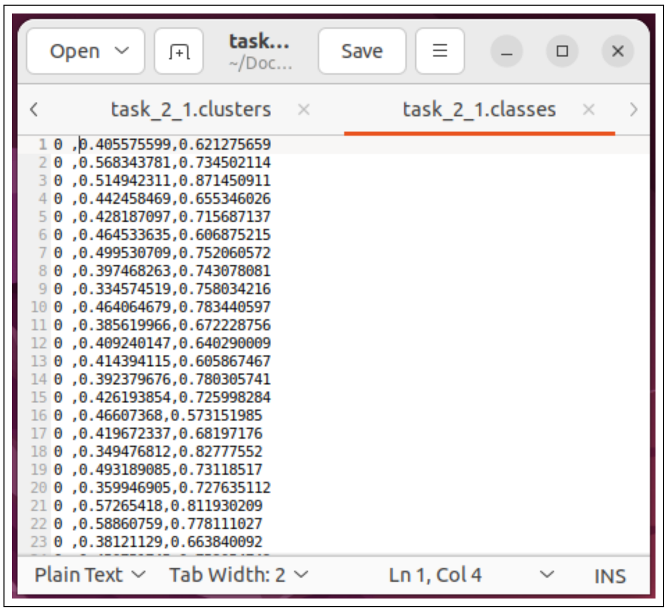


Fig. 9. Classes information

- Run the K-Means with $K = 3$ for 20 iterations.

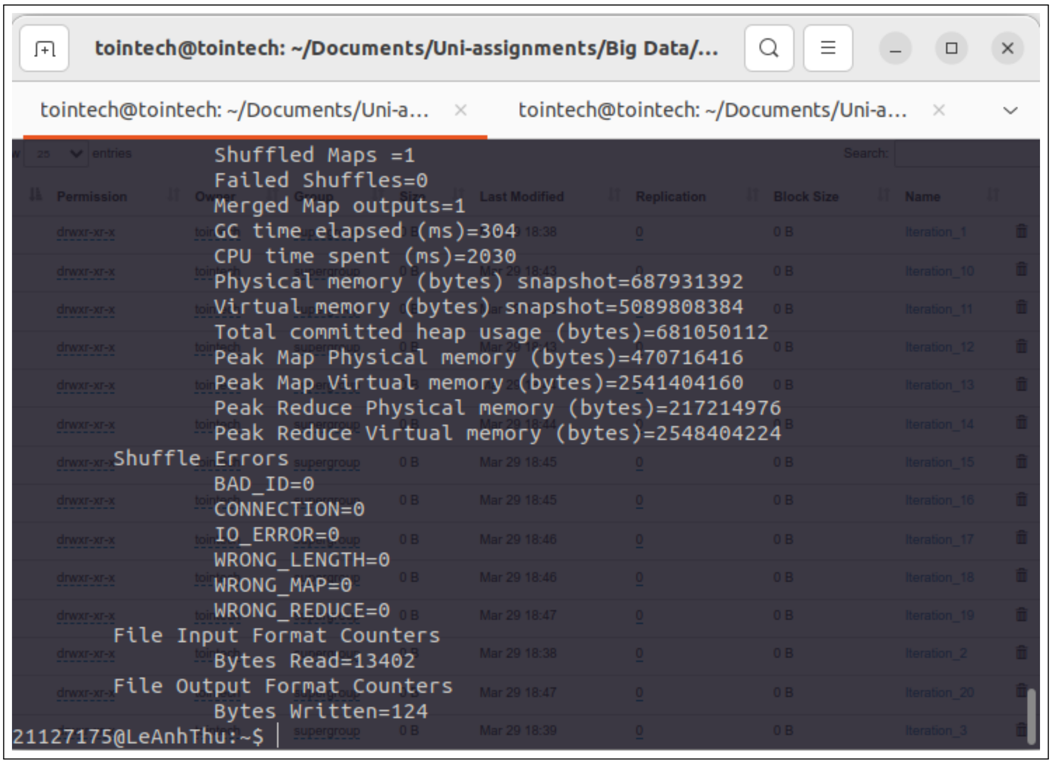
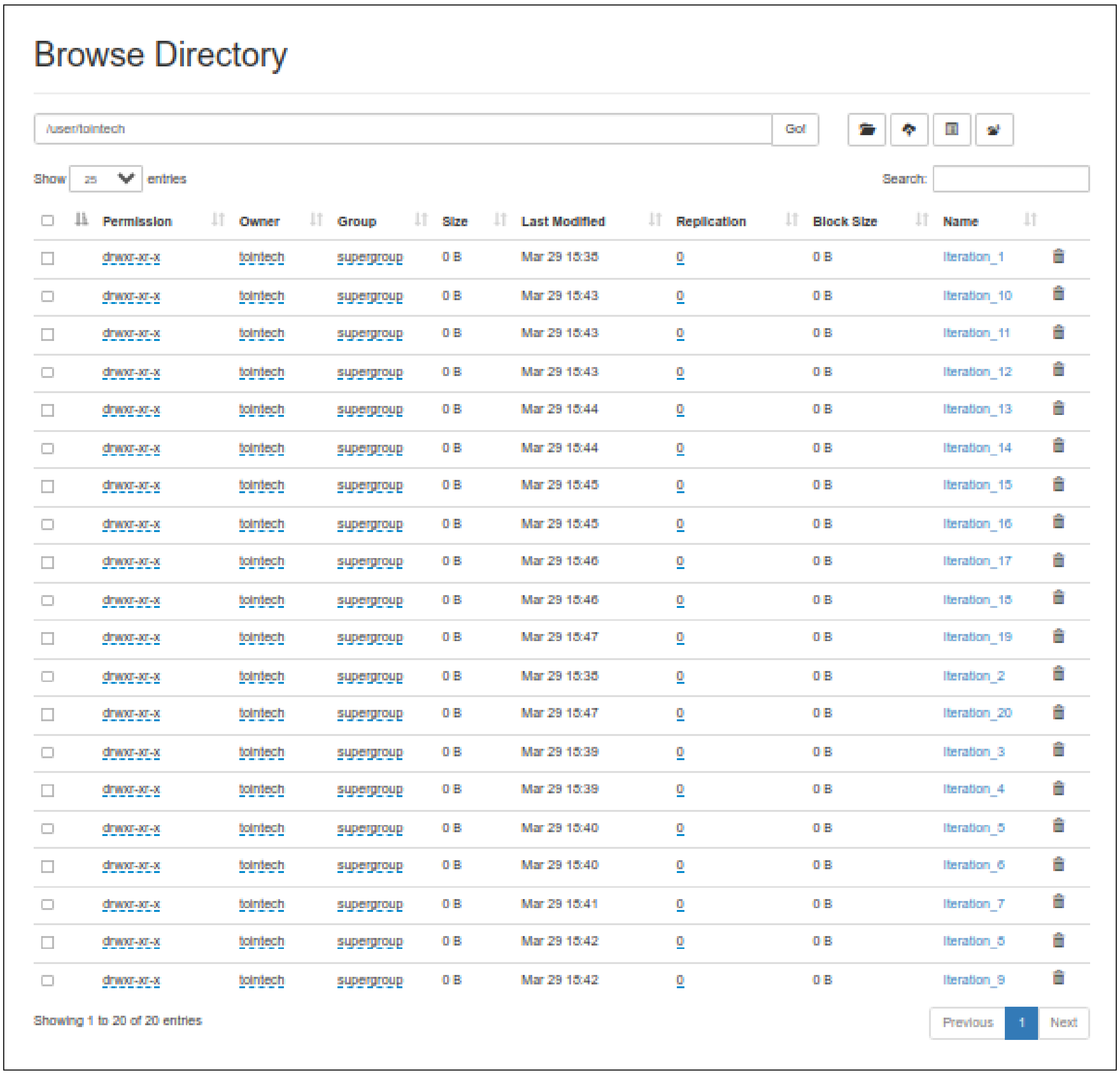


Fig. 10. Run K-Means algorithm



	Permission	Owner	Group	Size	Last Modified	Replication	Block Size	Name
	drwxr-xr-x	tointech	supergroup	0 B	Mar 29 15:35	0	0 B	iteration_1
	drwxr-xr-x	tointech	supergroup	0 B	Mar 29 15:43	0	0 B	iteration_10
	drwxr-xr-x	tointech	supergroup	0 B	Mar 29 15:43	0	0 B	iteration_11
	drwxr-xr-x	tointech	supergroup	0 B	Mar 29 15:43	0	0 B	iteration_12
	drwxr-xr-x	tointech	supergroup	0 B	Mar 29 15:44	0	0 B	iteration_13
	drwxr-xr-x	tointech	supergroup	0 B	Mar 29 15:44	0	0 B	iteration_14
	drwxr-xr-x	tointech	supergroup	0 B	Mar 29 15:45	0	0 B	iteration_15
	drwxr-xr-x	tointech	supergroup	0 B	Mar 29 15:45	0	0 B	iteration_16
	drwxr-xr-x	tointech	supergroup	0 B	Mar 29 15:46	0	0 B	iteration_17
	drwxr-xr-x	tointech	supergroup	0 B	Mar 29 15:46	0	0 B	iteration_18
	drwxr-xr-x	tointech	supergroup	0 B	Mar 29 15:47	0	0 B	iteration_19
	drwxr-xr-x	tointech	supergroup	0 B	Mar 29 15:35	0	0 B	iteration_2
	drwxr-xr-x	tointech	supergroup	0 B	Mar 29 15:47	0	0 B	iteration_20
	drwxr-xr-x	tointech	supergroup	0 B	Mar 29 15:39	0	0 B	iteration_3
	drwxr-xr-x	tointech	supergroup	0 B	Mar 29 15:39	0	0 B	iteration_4
	drwxr-xr-x	tointech	supergroup	0 B	Mar 29 15:40	0	0 B	iteration_5
	drwxr-xr-x	tointech	supergroup	0 B	Mar 29 15:40	0	0 B	iteration_6
	drwxr-xr-x	tointech	supergroup	0 B	Mar 29 15:41	0	0 B	iteration_7
	drwxr-xr-x	tointech	supergroup	0 B	Mar 29 15:42	0	0 B	iteration_8
	drwxr-xr-x	tointech	supergroup	0 B	Mar 29 15:42	0	0 B	iteration_9

Fig. 11. Browse directory

4.2 K-Means on Preprocessed Data

4.2.1 Task Information

I/O Description:

- **Input:** TF-IDF MTX file from Task 1.4. The data is converted to `tfidf.txt`: each row is in the form of `docid|termid1:tfidf1,termid2:tfidf2,:` : : . We can look at example in figure 8

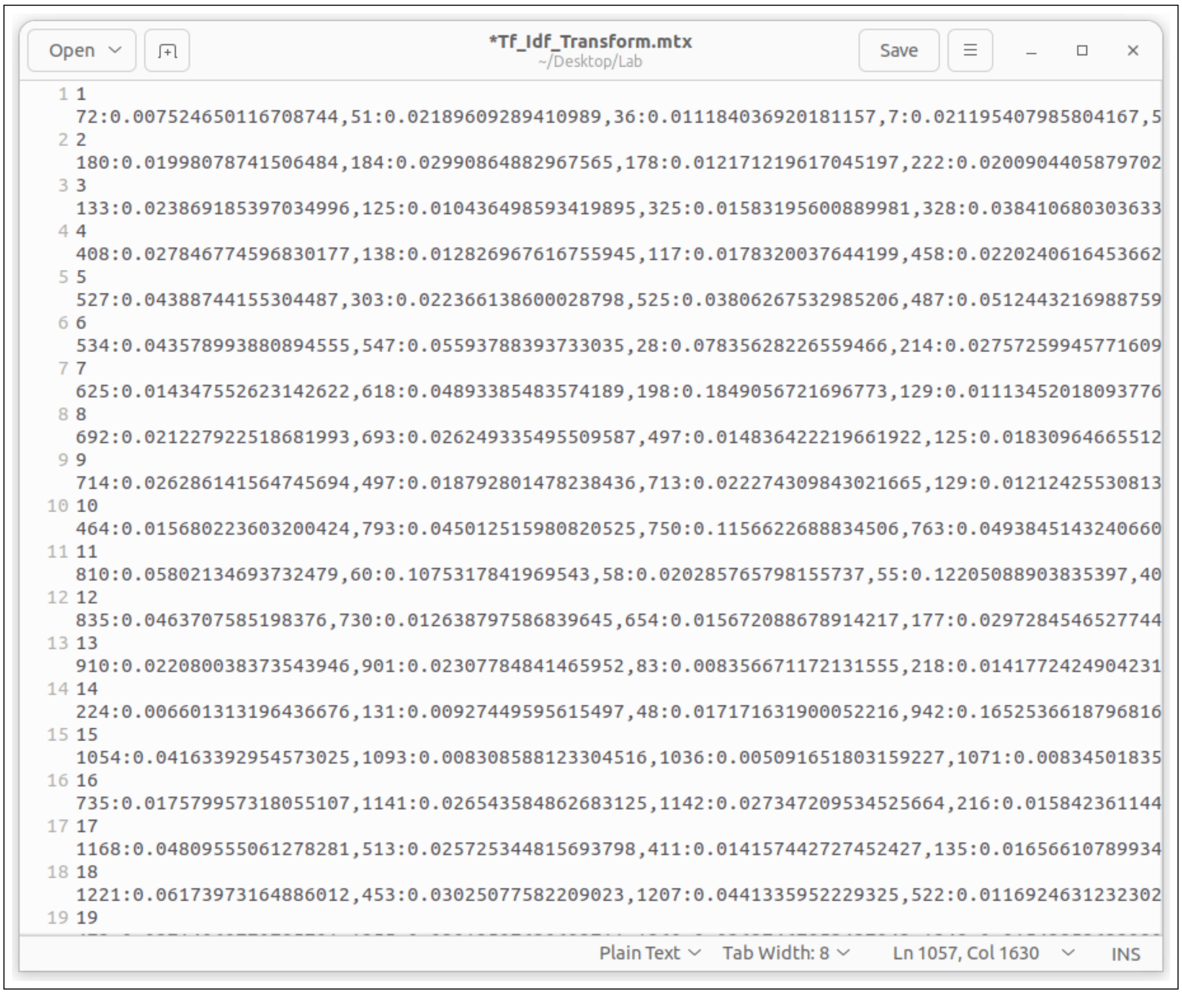


Fig. 12. Transformed TF-IDF

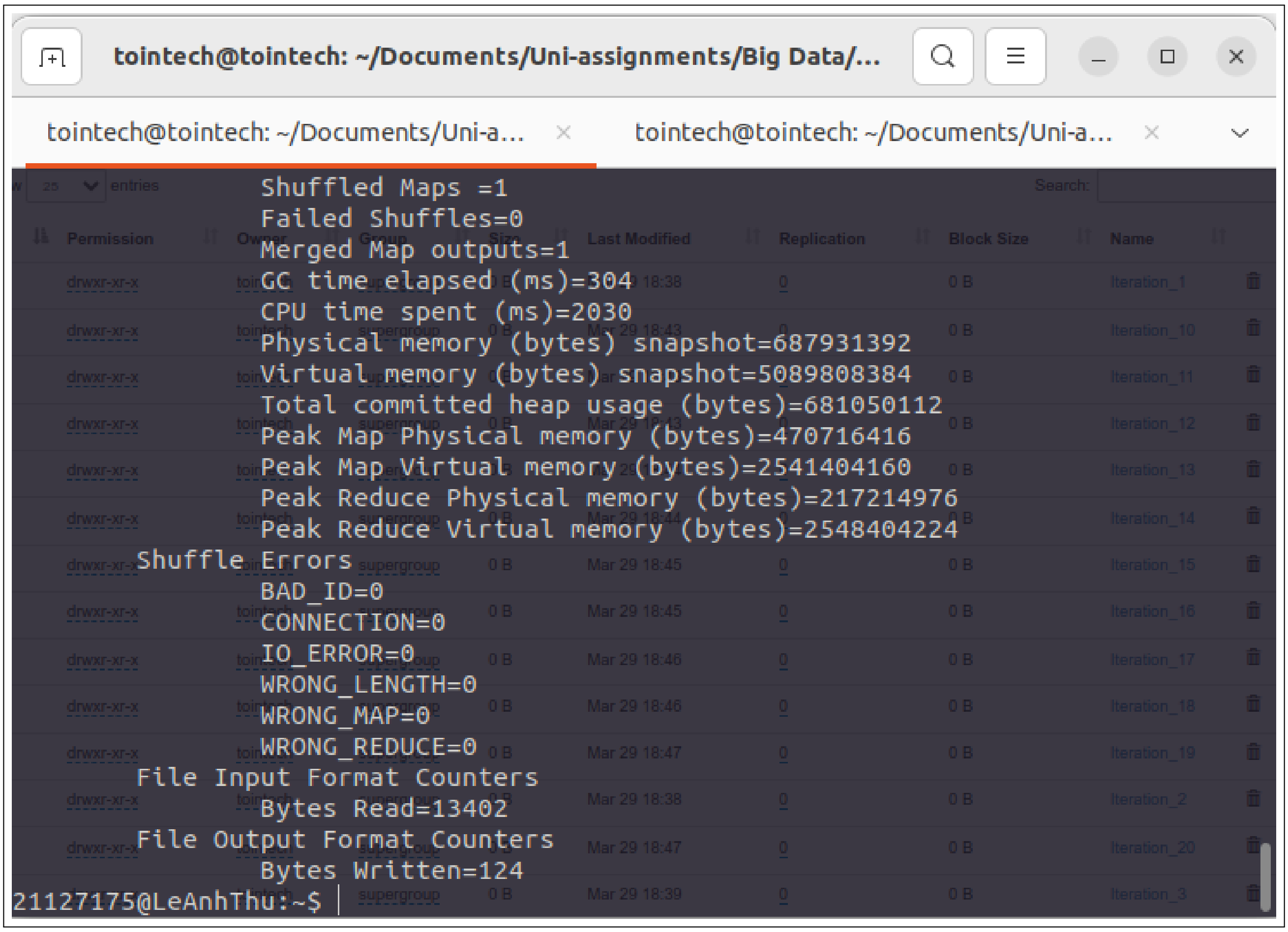


Fig. 13. Run K-Means algorithm

- **Output:** Cluster assignments for each document represented in the MTX file and mean of each clusters

4.2.2 Implementation

Generally, for this task, we will do following steps:

- Define the desired number of clusters `k`
- Implement the K-Means algorithm (similar to Task 2.1) but calculate distances using a suitable distance metric for TF-IDF vectors (e.g., cosine similarity).
- Run k-means with K = 5 for 10 iterations. Output the clusters centers to `task_2_2.clusters` and cluster assignment for each data point to `task_2_2.classes` in text format. For each iteration, report mean (top 10 words in tfidf) of each clusters to `task_2_2.txt` and objective function value to `task_2_2.loss` in text format.

Now we'll have a deeper look:

- Because the nature of Hadoop allows only one-time execution, we need to preserve the results of each previous run in order to apply the KMeans algorithm. The `Task_2_2.clusters` file serves as storage for the centroid values of clusters calculated in the previous iteration. For the first time, when the `Task_2_2.clusters` file hasn't been initialized yet, the `initializeRandomCentroids()` function is called to randomly generate initial clusters.

  For subsequent runs, the `readCentroidsFromFile()` function is invoked to read the clusters file, along with `ensureEnoughCentroids()` to ensure that there are enough K clusters.

read centroids information

```java
protected void setup(Context context) throws IOException, InterruptedException
    {
    Configuration conf = context.getConfiguration();
    FileSystem fs = FileSystem.get(conf);
    Path centroidPath = new Path("task_2_2.clusters");

    if (!fs.exists(centroidPath)) {
        initializeRandomCentroids();
    } else {
        readCentroidsFromFile(fs, centroidPath);
        ensureEnoughCentroids();
    }
}
```

- Map: From the file `Tf_Idf_Transform.mtx`, we will create a mapping of key-value pairs, where the key represents the `cluster_id` and the value consists of `term_id:tf_idf` pairs. The `cluster_id` is determined based on cosine similarity values to distribute documents into their nearest clusters.

Map function

```java
protected void map(Object key, Text value, Context context) throws
    IOException, InterruptedException {
    if (value.toString().startsWith("class")) {
        return;
    }
    double[] vectorTf_Idf_Double = new double[VECTOR_SIZE];
    Arrays.fill(vectorTf_Idf_Double, 0.0);

    String[] parts = value.toString().split("\\s+");
    int docIdInt = Integer.parseInt(parts[0]);
    String vectorTf_Idf = parts[1];
    String[] vector_string = parts[1].split(",");

    for (String termStr : vector_string) {
        String[] term = termStr.split(":");
        int term_id = Integer.parseInt(term[0]);
        double tf_idf = Double.parseDouble(term[1]);
        vectorTf_Idf_Double[term_id] = tf_idf;
    }
    double minDistance = Double.MAX_VALUE;
    int closestCentroidIndex = -1;

    for (int i = 0; i < K; i++) {
        double[] centroid = centroids.get(i);
        double distance = cosineSimilarity(vectorTf_Idf_Double, centroid);
        if (distance < minDistance) {
            minDistance = distance;
            closestCentroidIndex = i;
        }
    }

    context.write(new Text(String.valueOf(closestCentroidIndex)), value);
}
```

- cosineSimilarity: Calculated according to the formula.

$$similarity(A, B) = cos(\theta) = \frac{A.B}{\|A\| \, \|B\|} = \frac{\sum_{i=1}^{n} A_i B_i}{\sqrt{\sum_{i=1}^{n} A_i^2 \sum_{i=1}^{n} B_i^2}}$$

Cosine similarity function

```
private double cosineSimilarity(double[] vector1, double[] vector2) {
    double dotProduct = 0.0;
    double magnitude1 = 0.0;
    double magnitude2 = 0.0;

    for (int i = 0; i < vector1.length; i++) {
        dotProduct += vector1[i] * vector2[i];
        magnitude1 += Math.pow(vector1[i], 2);
        magnitude2 += Math.pow(vector2[i], 2);
    }

    magnitude1 = Math.sqrt(magnitude1);
    magnitude2 = Math.sqrt(magnitude2);

    if (magnitude1 == 0 || magnitude2 == 0) {
        return 0.0; // Handle zero vectors
    } else {
        return dotProduct / (magnitude1 * magnitude2);
    }
}
```

- Reduce: From the key-value pairs mapped earlier, we recalculate the centroids of each cluster by averaging the `tf_idf` values of the documents assigned to that cluster. Simultaneously, we calculate the loss after updating the centroid of each cluster.

Reduce function

```
protected void reduce(Text key, Iterable<Text> values, Context context) throws
    IOException, InterruptedException {

    double[][] sum = new double[VECTOR_SIZE][K];
    int[] count = new int[K];
   double[] vectorTf_Idf_Double = new double[VECTOR_SIZE];
    for (Text value : values) {
        int centroidIndex = Integer.parseInt(key.toString());
        Arrays.fill(vectorTf_Idf_Double, 0.0);

        String[] parts = value.toString().split("\\s+");
        String vectorTf_Idf = parts[1];
        String[] vector_string = vectorTf_Idf.split(",");

        for (String termStr : vector_string) {
            String[] term = termStr.split(":");
            int term_id = Integer.parseInt(term[0]);
            double tf_idf = Double.parseDouble(term[1]);
            vectorTf_Idf_Double[term_id] = tf_idf;
            sum[term_id][centroidIndex] += tf_idf;
        }
        count[centroidIndex]++;
    }

    for (int i = 0; i < K; i++) {
```

```java
            if (count[i] > 0) {
                String str = "";
                double[] newCentroid = new double[VECTOR_SIZE];
                for (int j = 0; j < VECTOR_SIZE; j++) {
                    newCentroid[j] = sum[j][i] / count[i];
                    str = str + "," + newCentroid[j];
                }
      double loss = calculateLoss(newCentroid, vectorTf_Idf_Double);
      totalLoss += loss;
      context.getCounter(LossCounter.TOTAL_LOSS).increment((long) (loss * 1000));
                context.write(key, new Text(str));
            }
        }

    }
```

- calculateLoss: Calculated according to the formula

$$WSSC = \sum_{i=1}^{k} \sum_{x \in C_i} \|x - \mu_i\|^2$$

Where:

- k is the number of clusters.
- C_i is the set of points in the i^{th}.
- x is a data point in cluster C_i.
- μ_i is the centroid of the i^{th} cluster.
- $\|x - \mu_i\|^2$ is the squared Euclidean distance between a data point x and a centroid μ_i

We can utilize the reduce process to compute the loss.

Calculating loss function

```java
    private double calculateLoss(double[] newCentroid, double[]
        vectorTf_Idf_Double) {
            double loss = 0.0;
            for (int i = 0; i < VECTOR_SIZE; i++) {
                if (vectorTf_Idf_Double[i] != 0) {
                loss += Math.pow(vectorTf_Idf_Double[i] - newCentroid[i], 2);
                }
        }
        return loss;
     }
```

- To obtain the file `task_2_2.txt`, as mentioned earlier, `task_2_2.clusters` stores the results of the previous run. We can access `task_2_2.clusters` after the reduce process is completed. Using those values, we sort them in descending order and take the top 10 values to generate `task_2_2.txt`.

outputClusterMeans function

```java
    private static void outputClusterMeans(Job job, Configuration conf, Path
        centroidOutputPath, int iteration) throws IOException {
        try (BufferedReader br = new BufferedReader(new
            FileReader(centroidOutputPath.toString()))) {
            String line;
            double[] tf_idf_mean = new double[VECTOR_SIZE];
            List<String> means = new ArrayList<>();
```

```java
            while ((line = br.readLine()) != null) {
                String[] parts = line.split(",");
                StringBuilder strBuilder = new StringBuilder(parts[0]);
                for (int i = 1; i < parts.length; i++) {
                    tf_idf_mean[i - 1] = Double.parseDouble(parts[i]);
                }
                Integer[] indices = new Integer[VECTOR_SIZE];
                for (int i = 0; i < VECTOR_SIZE; i++) {
                    indices[i] = i;
                }
                Arrays.sort(indices, Comparator.comparingDouble((Integer i) ->
                    tf_idf_mean[i]).reversed());
                for (int i = 0; i < 10; i++) {
                    int index = indices[i];
                    double value = tf_idf_mean[index];
                    strBuilder.append(",").append(indices[i]).append(":").append(value);
                }
                means.add(strBuilder.toString());
            }

            FileSystem fs = FileSystem.get(conf);
            Path outputPath = new Path("Iteration_" + iteration + "/task_2_2.txt");
            try (BufferedWriter writer = new BufferedWriter(new
                OutputStreamWriter(fs.create(outputPath)))) {
                for (String mean : means) {
                    writer.write(mean);
                    writer.newLine();
                }
            }
        }
    }
```

4.2.3 Output

- Regarding the output, to clearly observe the issues, we have three files: `part-r-00000` containing the centroid values of clusters, `task_2_2.loss`, and `task_2_2.txt`. These three files can be found when running the program `KMeansMapReduce.java` in the directory `/user/.../Iteration_/` on the HDFS.

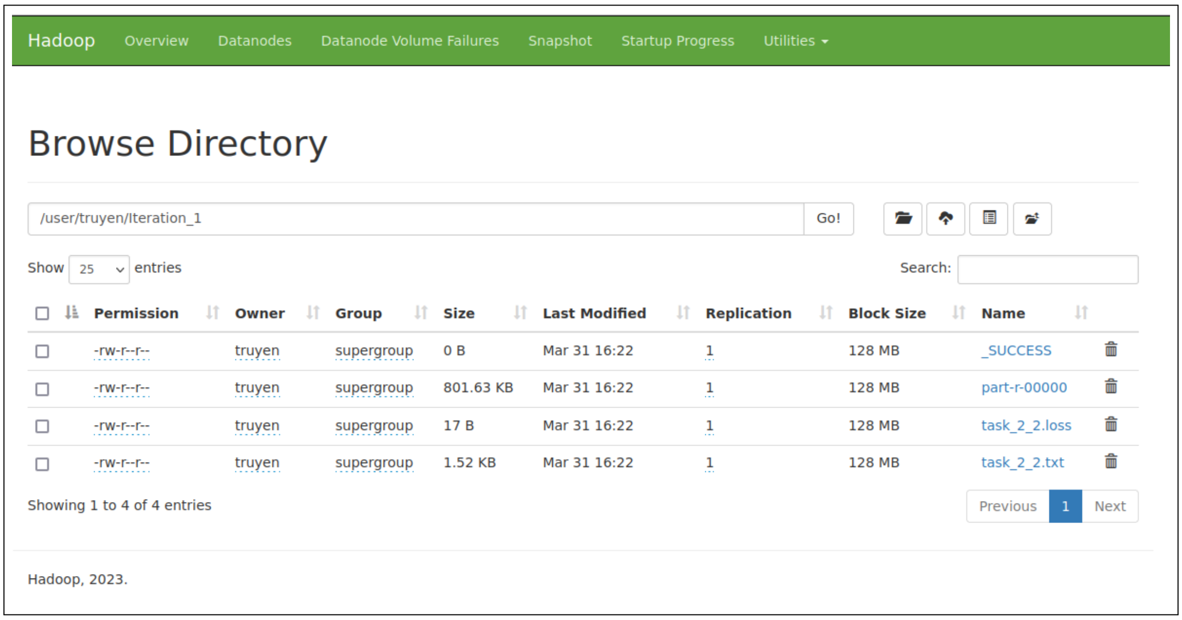


Fig. 14. Output of KMeansMapReduce.java

- The file `task_2_2.clusters` can be found in the local directory on your computer after running the `KMeansMapReduce.java` program.
- The file `task_2_2.classes` is located on the HDFS in the `output` directory with the name `part-r-00000` after running `AssignCluster.java`. The result after running `AssignCluster.java` provides us with the output where data points have been assigned to new clusters.

4.2.4 Evaluate

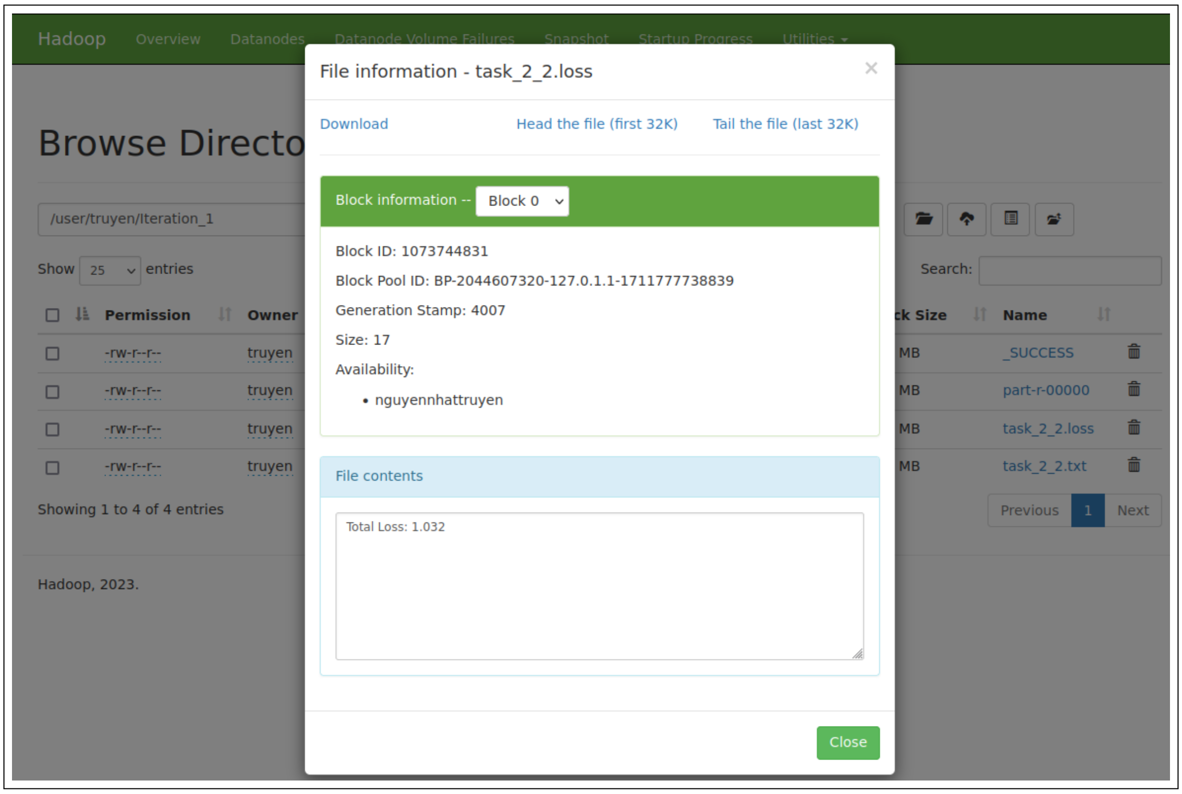


Fig. 15. Loss at iteration 1

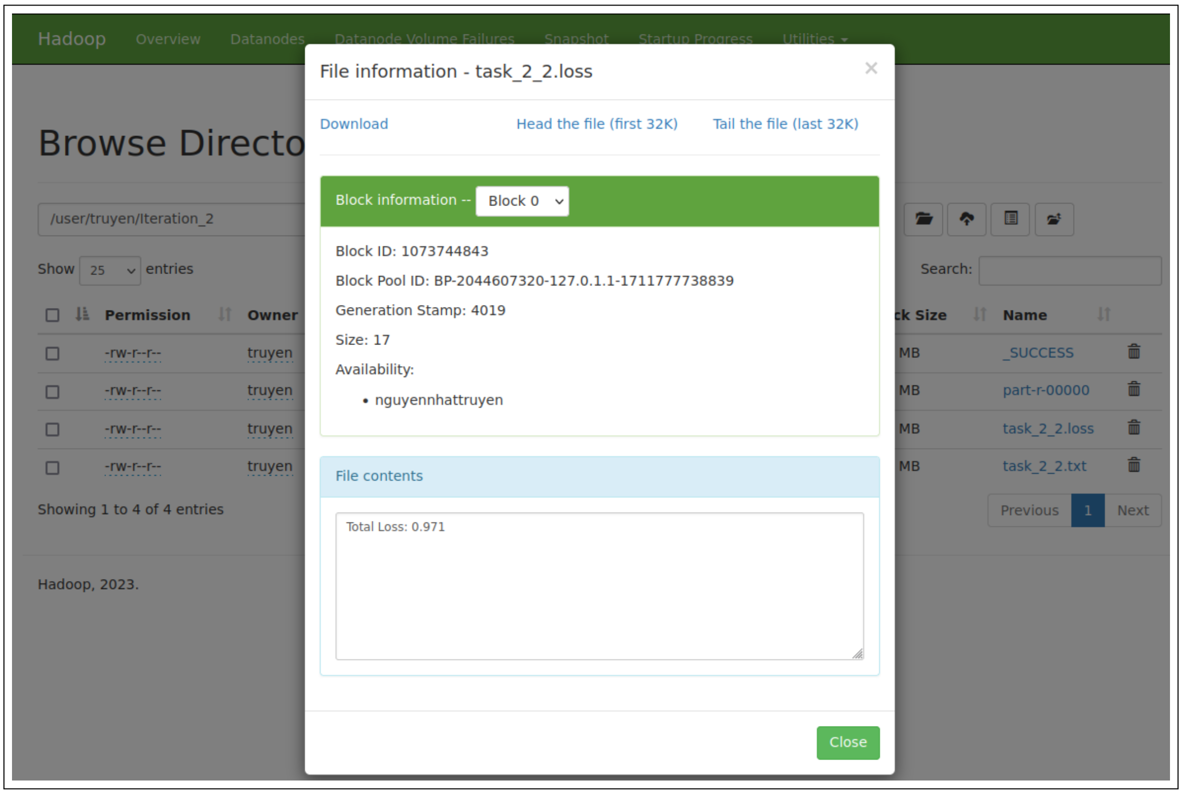


Fig. 16. Loss at iteration 2

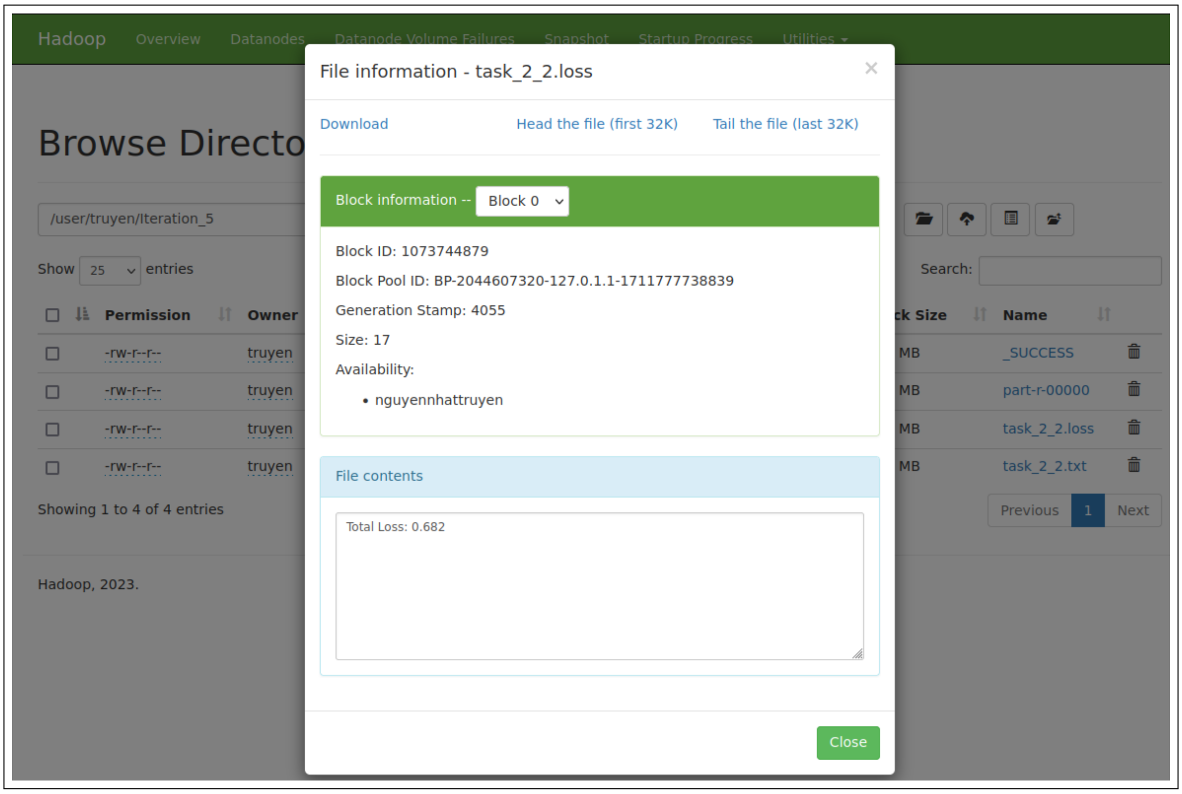


Fig. 17. Loss at iteration 5

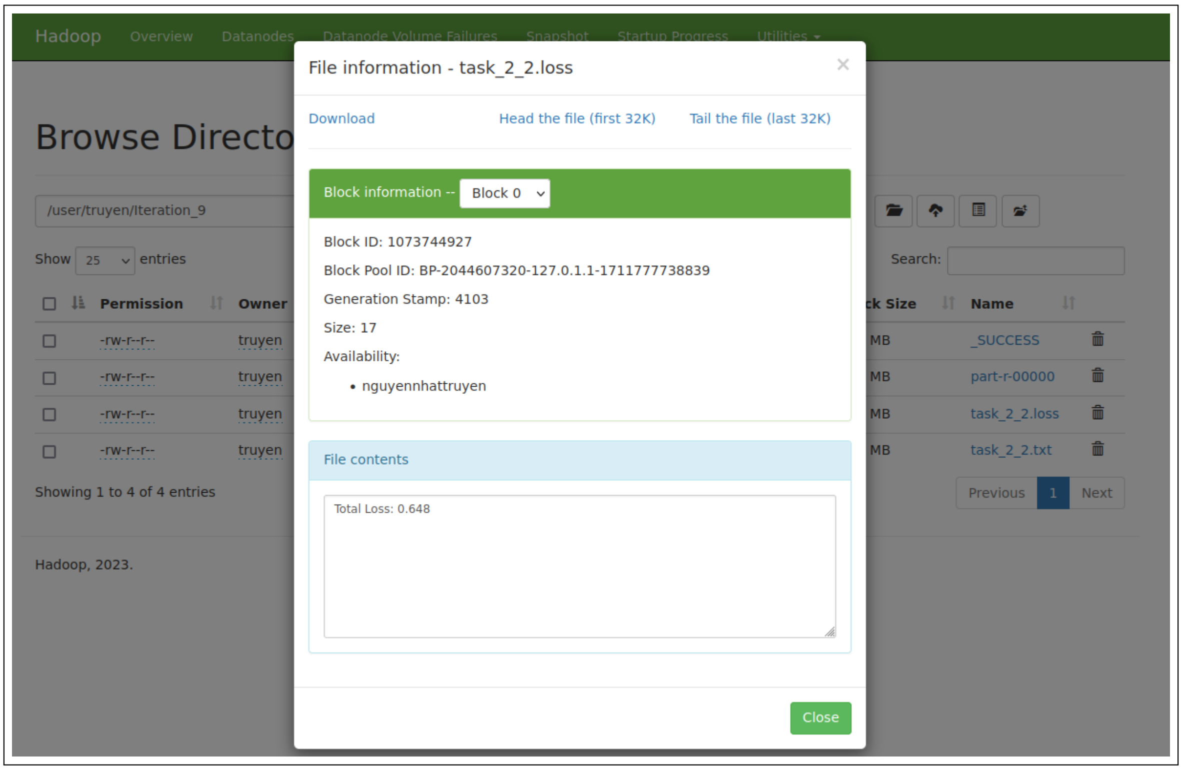


Fig. 18. Loss at iteration 9

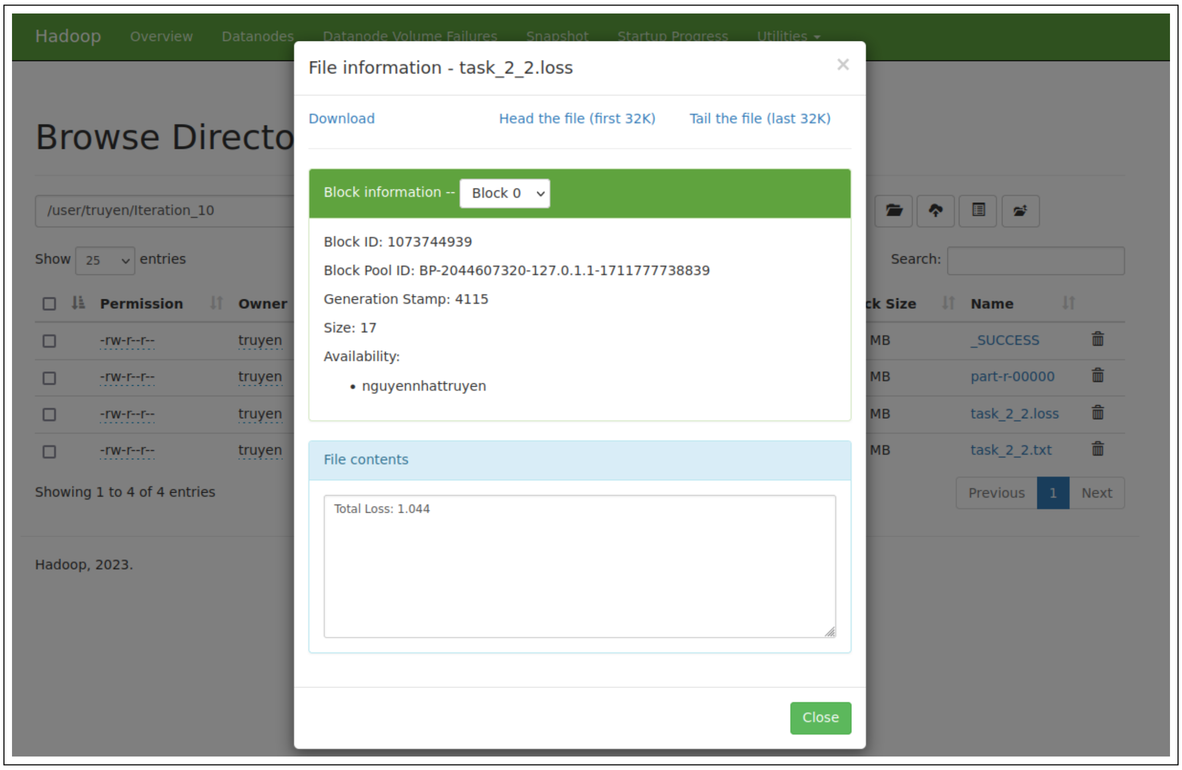


Fig. 19. Loss at iteration 10

The loss function generally decreases as the iteration increases. However, there is still a possibility that increasing the iteration may not result in a decrease in the loss function. In the case mentioned above, after iteration 10, the loss suddenly increased significantly, even higher than the initial value.

4.3 Scalable K-Means++ Initialization

Due to time constraints and the complexity of the task, our team has not been able to complete this section on time.

5 Conclusion

We gain valuable experience and knowledge during this project, specifically:

- For me (Minh-Dat), personally, this is my first experience with Linux. Duc-Thien helped me install a virtual machine running on Linux, and familiarize with the system. Since then, I have learnt to edit code using nano, execute commands, manage the file system, and some other things. This is a short but mesmerizing adventure for me!

- As a team, we learn to code Java and realize that it is pretty similar to C++. We also revise fundamental knowledge in OOP and DSA when building classes and implementing data structures such as HashMap and HashSet.

- Of course, designing and implementing MapReduce programs is the heart and soul of this project. We have to learn to think in a MapReduce way. Specifically, we start from building simple map and reduce functions, then learn to chain multiple MapReduce jobs together, rewrite the setup and cleanup methods, add cache files, and so on. By the way, we all find it interesting to implement familiar algorithm like K-Means in MapReduce. It is way more difficult than we could imagine!

References

[1] Treemap class documentation. Accessed: March 28, 2024.