AI506: DATA MINING AND SEARCH (SPRING 2023)

Homework 2: Distributed Programming with MapReduce

Release: March 29, 2023,
Due: April 14, 2023, 11:59pm

1 Introduction

This assignment will help you become familiar with distributed programming using MapReduce. In this assignment, you will develop a simple search engine that finds similar papers by their titles.

2 Basic Usage of Hadoop

You will use Hadoop, which is one of the most popular open-source implementations of MapReduce. To run your implementation, Hadoop needs to be installed on one or more machine(s). For your convenience, we will provide a Hadoop cluster during the homework period. Details for accessing the cluster will be provided later on Classum. You are encouraged to install Hadoop on your own device. An installation guide for Hadoop can be found at `https://hadoop.apache.org/docs/current/hadoop-project-dist/hadoop-common/SingleCluster.html`. Moreover, you should be able to access Hadoop DFS (HDFS) and run MapReduce applications.

2.1 How to Access HDFS

You can access HDFS by typing `hadoop fs`. To upload a file from your local file system to HDFS, use the command `hadoop fs -put`. To download a file from HDFS to the local file system, use the command `hadoop fs -get`. Also, you can use various bash commands. For example, `hadoop fs -cat` copies a file to stdout, `hadoop fs -mkdir` creates a directory at a given path, and `hadoop fs -rm` deletes a file at a given path. For more commands, see `https://hadoop.apache.org/docs/stable/hadoop-project-dist/hadoop-common/FileSystemShell.html`.

2.2 MapReduce Examples

2.2.1 Java Implementation of WordCount

Implementing a Map Function in Java. To implement a mapper function, you should extend the `org.apache.hadoop.mapreduce.Mapper` class. To count each word in a given document, each word should become a key, and the corresponding value should be set to 1. The context object is used to pass `<key, value>` pairs to reducers. We provide an example below.

```
public static class TokenizerMapper
    extends Mapper<Object, Text, Text, IntWritable>{
 // In wordcount mapper, each word becomes key and value is set to 1
 private final static IntWritable one = new IntWritable(1);
 private Text word = new Text();
 public void map(Object key, Text value, Context context)
        throws IOException, InterruptedException {
    StringTokenizer itr = new StringTokenizer(value.toString());
    while (itr.hasMoreTokens()) {
      word.set(itr.nextToken());
      // key, value pair is passed into context in Mapper function
      context.write(word, one);
    }
  }
}
```

Implementing a Reduce Function in Java. To aggregate the values associated with the same key, you should implement the `org.apache.hadoop.mapreduce.Reducer` class. For each key, the list of values is passed to the reducer, and in the example code below, the values are added. The reduced `<key, value>` pairs should be passed to the context object to be considered as final outputs.

```
public static class IntSumReducer
    extends Reducer<Text,IntWritable,Text,IntWritable> {
  private IntWritable result = new IntWritable();
  // values with same key is aggregated and passed to reducer object
  public void reduce(Text key, Iterable<IntWritable> values,
    Context context) throws IOException, InterruptedException {
    int sum = 0;
    // In wordcount example, adding all values is required for reducer
    for (IntWritable val : values) {
      sum += val.get();
    }
    result.set(sum);
    context.write(key, result);
  }
}
```

Compiling Java Classes. Java classes need to be compiled into a jar file to be run on Hadoop. A jar file can be created as follows:

```
    > hadoop com.sun.tools.javac.Main WordCount.java
    > jar cf wc.jar WordCount*.class
```

You can run a jar file on Hadoop by typing `yarn jar [path-of-the-jar-file] [arguments]`. For your convenience, sample source code written in Java is attached to this homework (see `wordcount_java` option in `Makefile`). For more information, see `https://hadoop.apache.org/docs/current/hadoop-mapreduce-client/hadoop-mapreduce-client-core/MapReduceTutorial.html`.

2.2.2 Python Implementation of WordCount

You can run a Hadoop job using Hadoop Streaming, a utility for various programming languages, including Python. In this assignment, only Python is allowed if you use Hadoop Streaming. For running MapReduce using the utility, you need to provide a mapper file and a reducer file written in Python. While executed, mappers get inputs from `sys.stdin` and pass `<Key, Value>` pairs to `sys.stdout`. Then, reducers receive `<Key, Value>` pairs from `sys.stdin` in key-sorted order and emit reduced outputs to `sys.stdout`. Note that reducers receive consecutive `<Key, Value>` pairs with the same key instead of a list of values for each key. Here is an example Python code for Hadoop Streaming:

```python
# (1) mapper.py
# Tokenize each line and generates <Word, 1> pairs

import sys
def write(key, value):
    sys.stdout.write(f"{key}\t{value}\n")
for line in sys.stdin:
    words = line.strip().split()
    for w in words:
        write(w, "1")

# (2) reducer.py
# Tokenize each <Word, 1> pairs and summing up until a new word comes

import sys
def write(key, value):
    sys.stdout.write(f"{key}\t{value}\n")
curr_word, curr_count = None, 0
for line in sys.stdin:
    key_from_map, value_from_map = line.strip().split()
    if curr_word != key_from_map:
        if curr_count:
            write(curr_word, curr_count)
        curr_word = key_from_map
        curr_count = 0
    curr_count += int(value_from_map)
```

```
if curr_count:
    write(curr_word, curr_count)
```

Sample source code written in Python is attached to this assignment (see `wordcount_py` option in `Makefile`). For more information, see `https://hadoop.apache.org/docs/stable/hadoop-streaming/HadoopStreaming.html`.

3 Implementation (75 points)

In this assignment, your job is implementing a simple search algorithm that finds similar papers by their titles. Specifically, you will implement Locality Sensitive Hashing (LSH) using the MapReduce framework. For implementation, **you can use one between Python and Java.**

Your job can be briefly divided into the following steps: given a list of paper titles, you will (1) represent each title as a list of k-shingles, (2) generate the signature matrix using Min-hashing, and then (3) identify candidate pairs based on their signatures.

3.1 Subtask 1: Shingling (25 points)

Your first job is implementing the k-shingling algorithm, which converts the paper titles into a list of shingles. Specifically, your task is to create a dictionary that associates each title with a list of 3-shingles (also known as 3-grams). k-Shingling is a method for creating document representations. Each k-shingle of a document is a consecutive substring of length k found in the document. The definition of substring may vary, but in this assignment, you will use a word as a token for your shingle, not a character. Descriptions of the inputs and outputs, as well as pseudocode, are provided below.

(Input) Pairs of a paper id and a paper title.

- Example: (1, "Generative Adversarial Nets"), (2, "Attention Is All You Need"), ...

(Output) Pairs of a paper id and a 3-shingle (**duplicates are allowed**).

- Example: (1, ("Generative", "Adversarial", "Nets")), (2, ("Attention", "Is", "All")), (2, ("Is", "All", "You")), (2, ("All", "You", "Need")), ...

(Pesudocode)

```
def ngrams(word, n=3):
    return [tuple(words[i:i+n]) for i in range(len(words)-n+1)]
```

To implement k-Shingling using the MapReduce programming model, you should implement the following classes:

- **Subtask1Mapper/Subtask1Reducer** for obtaining 3-shingles of the given titles.

3.2 Subtask 2: Min-hashing (25 points)

Your second job is to Min-hashing. Let U be a universal set, and f be a hash function that maps the members of U to distinct integers. For any subset S of U, we can define a Min-hashing function $h_f(S)$ as the minimum among hashed elements as follows:

$$h_f(S) = \min_{s \in S} f(s). \tag{1}$$

We call $h_f(S)$ the (Min-hashing) signature obtained using the hash function f. Given papers, each of which is represented as a set of shingles, you should compute 10 (Min-hashing) signatures of each paper using 10 given hash functions. Descriptions of the inputs and outputs, as well as pseudocode, are provided below.

(Input) Pairs of a paper id and a 3-shingle.

- Example: (1, (“Generative”, “Adversarial”, “Nets”)), (2, (“Attention”, “Is”, “All”)), (2, (“Is”, “All”, “You”)), (2, (“All”, “You”, “Need”)), ...

(Output) Pairs of a paper id and the corresponding signature values (**duplicates are NOT allowed**).

- Example: (1, (1, 5, 2, 0, 3, 0, 1, 4, 5, 0)), (2, (8, 5, 2, 10, 7, 3, 5, 9, 5, 2)), ...

Note that the signature values in the above example can be different from those obtained in your implementation.

(Pseudocode)

```
def minHash(paper_idx_to_shingles, hash_functions):
    C = len(paper_idx_to_shingles) # Total number of titles
    M = len(hash_functions) # Total number of hash_functions
    # Each signature value is initialized to a very large constant
    signatures = [[(10 ** 100) for j in range(C)] for i in range(M)]

    for paper_idx, shingles in enumerate(paper_idx_to_shingles.items()):
        # paper_idx: Index of a paper
        # shingle_idxes: shingles of the paper
        for hash_idx, hash_func in enumerate(hash_functions):
            # hash_idx: Index of a hash function H
            # hash_func: hash function H
            # Obtain sig, which is the minHash value of title X,
            # using hash function H
            sig = min([hash_func(shingle) for shingle in shingles])
            signatures[hash_idx, paper_idx] = sig

    return signatures
```

Please note that for computational efficiency and simplicity, your implementation should use hash functions instead of random permutations. We have provided the hash functions with a simple usage example in the attached skeleton code for your convenience.

To implement Min-hashing using the MapReduce programming model, you should implement the following classes:

- **Subtask2Mapper/Subtask2Reducer** for obtaining signature values for each paper.

3.3 Subtask 3: Bucketing (25 points)

Your last job is finding candidate pairs based on the signature matrix, where each i-th row is the 10 signature value of the i-th paper. Given a signature matrix M, we divide it into 5 bands (i.e., disjoint groups) of 2 consecutive rows. Any two papers become a candidate pair if and only if all their signature values are exactly the same in at least one band. Descriptions of the inputs and outputs, as well as pseudocode, are provided below.

(Input) Pairs of a paper id and the corresponding signatures.

- Example: (1, (1, 5, 2, 0, 3, 0, 1, 4, 5, 0)), (2, (8, 5, 2, 10, 7, 3, 5, 9, 5, 2)), ...

(Output) List of all candidate pairs (**duplicates are allowed**).

- Example: (1, 3), (2, 3), ...

In the above example, note that (1, 2) cannot be a candidate pair, even though 3 signatures are matched.

(Pseudocode)

```
def lsh(signatures, b, r):
    M = signatures.shape[0]  # The number of min-hash functions
    C = signatures.shape[1]  # The number of titles

    assert M == b * r

    candidatePairs = set()

    for i in range(b): # Iterate through each band
        buckets = defaultdict(list)
        for d in range(C): # Iterate through each paper
            key = tuple(signatures[i * r + j][d] for j in range(r))
            buckets[key].append(d)

        for bucket in buckets.values():
            for paper1, paper2 in itertools.combinations(bucket, 2):
                candidatePairs.add((paper1, paper2))

    return candidatePairs
```

To implement bucketing using the MapReduce programming model, you should implement the following classes:

- **Subtask3Mapper/Subtask3Reducer** for obtaining candidate pairs.

3.4 Input Format

We provide a single file in the Hadoop cluster: `/hadoop/inputs/titles.txt`. In the file, each line contains a paper id x and a title s separated by a tab, which implies that the title of the paper whose id is x is s. You can test your implementation with `small.txt` in the attachment files, which has the same formats as `titles.txt`.

3.5 Output Format

- For subtask 1, each line of your output should consist of a paper id and a 3-shingle. The paper id and the 3-shingle should be separated by a tab, and the words contained in the 3-shingle should be separated by a comma.
- For subtask 2, each line of your output should consist of a paper id and the corresponding (Min-Hash) signature values. The paper id and the signature values should be separated by a tab, and the signature values should be separated by a comma.
- For subtask 3, each line of your output should consist of an unordered pair of paper ids. The paper ids should be separated by a tab.

We allow the duplicated lines in the outputs for subtask 1 and subtask 3 for your convenience. Here are the expected outputs when we use `/hadoop/inputs/titles.txt` as input:

```
# for subtask 1
1       i,have,a
1       have,a,pen
...
6       pineapple,apple,pen

# for subtask 2
1       38434503,36408736,...,113292102
2       38434503,33081067,...,113292102
...
6       214311585,18264843,...,350524841

# for subtask 3
2       4
3       6
```

3.6 Hints

- We provide `WordCount` examples and skeleton codes for this assignment. For Java users, we provide `ExampleMapper`, `ExampleReducer`, and example `main` functions in `LSH.java`. For Python users, we provide `ExampleMapper.py`, `ExampleReducer.py`, and `run.py` for running `ExampleMapper` and `ExampleReducer`.
- You can freely set the number of reducers for all MapReduce jobs. However, since the computational resources of the provided cluster are limited, please set the number of reducers to at most 5. However, your implementation should produce the correct outputs regardless of the number of mappers and reducers. In other words, you should not assume a specific number of mappers and reducers.
- **(Optional)** Note that the reducer sorts keys in ascending order and then processes them sequentially in the default setting. It sorts the keys with a type `Text` in lexicographical order and the keys with a type `IntWritable` in numerical order. If you want to sort keys in numerical order with Hadoop Streaming, you need to add following options.

  - `-D mapred.output.key.comparator.class=org.apache.hadoop.mapred.lib .KeyFieldBasedComparator`
  - `-D mapred.text.key.comparator.options=-n`

- In HDFS, you are only allowed to use a `/user/[YOUR_STUDENT_ID]` directory (See `Makefile`). You should not manipulate other people's directories.

4 Analysis (25 points)

- (Q1) Provide any three candidate pairs with their titles. Feel free to refer to `/hadoop/inputs/titles_raw.txt`, which contains each paper's id and title.
- (Q2) Describe your design of map/reduce functions for solving each subtask. For example, our functions for WordCount can be described as follows:
  - Mapper: For every line, mappers split the line and then emit $\langle w, 1\rangle$ for each word w in the line.
  - Reducer: For each word w, reducers take a list L then emit $\left\langle w, \sum_{e\in L} e\right\rangle$.

5 Test

You should submit `Makefile` for compiling and running your implementations. Your Makefile should support `all` option to execute the implementation and output the final execution result. For the `all` option, you can choose one between the two built-in options, `all_java` and `all_py`, depending on which language you have used for your implementation. Again, you do NOT need to implement both in Python and Java.

For evaluation, we will test your implementation on our Hadoop clusters. During the homework period, you can access the cluster via SSH. More information for accessing the cluster will be provided soon. Your implementation should not take more than 30 minutes for the given dataset on our cluster.

6 Notes

- You may encounter some subtleties when it comes to implementation. Come up with your own design and/or contact Kyuhan Lee (kyuhan.lee@kaist.ac.kr) and Jihoon Ko (jihoonko@kaist.ac.kr) for discussion. Any idea can be taken into consideration when grading if they are written in the *readme* file.
- For this assignment, you are allowed to reference official Hadoop MapReduce tutorial (`https://hadoop.apache.org/docs/current/hadoop-mapreduce-client/hadoop-mapreduce-client-core/MapReduceTutorial.html`), and Hadoop Streaming tutorial (`https://hadoop.apache.org/docs/current/hadoop-streaming/HadoopStreaming.html`).

7 How to submit your assignment

1. Create hw2-[your student id].tar.gz, which should contain the following files:
   - **Makefile and source code**: these should contain your implementation.

- **report.pdf**: this should contain your answers for Section 4 (Analysis).
- **readme.txt**: this file should contain the names of any individuals from whom you received help, and the nature of the help that you received. That includes help from friends, classmates, lab TAs, course staff members, etc. In this file, you are also welcome to write any comments that can help us grade your assignment better, your evaluation of this assignment, and your ideas.

2. Make sure that no other files are included in the tar.gz file.
3. Submit the tar.gz file at KLMS (`http://klms.kaist.ac.kr`).