[Recitation 7]

Paahuni Khandelwal
Email: cs435@cs.colostate.edu

11th October, 2019
Submission

- Submission Deadline for PA2 - 24th October (by 5 pm)
- Worth 10 points (10% of your grades)
- Dataset - same as for PA1
How to design Jobs for PA2

• There are few documents (articles) without article body. (For example: HelloWorld<====>12345<====>)

• Put a checker before you proceed. Also, ignore such articles while calculating \( N \) (total number of articles in the corpus).

• Divide the PA2 into two profiles (A and B). From Part A, get the TF-IDF value for each unigram in each document.

• From profile B, you should get article summary for any given article (provided during demo).

• To calculate \( N \) and pass its value to a different job, use Hadoop's **Counters** class
Profile A Job 1

MAP REDUCE JOB 1

- Splitting input value by <====>
- Preprocess unigrams ie remove non-alphanumeric characters from articles
- Find the frequency of each unigram in each article
- Similar to Profile2 in PA1
- Output will be in the form: <docID, (unigram frequency)>

NOTE: Make your application scalable by having multiple reducer
Profile A Job 2

- Calculate \( TF \) of each unigram \( i \) in each article \( j \) using:
  - \( TF_{ij} = 0.5 + 0.5 \left( \frac{f_{ij}}{\max_k f_{kj}} \right) \)

  where, \( \max_k f_{kj} \) is the highest frequency of a unigram in the article/document \( j \).

- **Mapper** output will be in the form: \(<K,V> -> <docID, (unigram frequency)>\)

- **Reducer**:
  - First iterate through all the values and find \( \max_k f_{kj} \)
  - Next, calculate TF for each unigram
  - Output will be in the form: \(<docID, (unigram TFvalue)>\)
In this job, you can calculate total number of documents in entire corpus

Calculate \( n_i \), total number of documents in the whole corpus, where unigram \( i \) occurred at least once

Mapper output will be in the form: \(<K,V> \rightarrow \text{<unigram, (docID unigram TFvalue)>}\)

Reducer:
- In setup() : Fetch total number of documents counter value from previous map job
- Iterate through all the input values and find \( n_i \).
- Calculate idf value as well, finally push \( tf \times idf \) as one of the outputs
- Output will be in the form: \(<\text{docID, (unigram Tfidf)}>\)
Profile A Job 3 (Continued)

- Calculate TF-IDF of each unigram $i$ in each article $j$ using: $\text{IDF}_i = \log_{10}(N/n_i)$
- $\text{TF-IDF}_{ij} = \text{TF}_{ij} \times \text{IDF}_{ij}$
- Calculate IDF for each unigram.
- Calculate TF-IDF for each unigram.
- Output will be in the form: $<\text{docID}, (\text{unigram TF-IDF value})>$

**Note:** To calculate IDF, you need to pass $N$ that you calculate in previous map job.
Counter Class - Different jobs

- Creating global structure:
  
  ```java
  public static enum DocumentsCount{
    NUMDOCS
  }
  ```

- Updating the global count in the mapper:
  
  ```java
  context.getCounter(DriverClass.DocumentsCount.NUMDOCS).increment(1);
  ```

- Retrieving the count in driver class before using it in next job
  
  ```java
  Counter documentCount = firstJob.getCounters().findCounter(DocumentsCount.NUMDOCS);
  ```

- Declare use of counters in the driver:
  
  ```java
  IDFJob.getConfiguration().setLong(DocumentCount.getDisplayName(), DocumentCount.getValue());
  ```

- Access the count value in the mapper/reduce inside the setup() method:
  
  ```java
  N = context.getConfiguration().getLong(DriverClass.DocumentCount.NUMDOCS.name(), 0 );
  ```
Counter Class - Same Jobs

- Creating global structure:
  ```java
  public static enum DocumentsCount{
      NUMDOCS
  };
  ```

- Updating the global count in the IDF mapper:
  ```java
  context.getCounter(DocumentsCount.NUMDOCS).increment(1);
  ```

- Access the count value in the reducer inside the setup() method:
  ```java
  private long mapperCounter;

  @Override
  public void setup(Context context) throws IOException, InterruptedException{
    Configuration conf = context.getConfiguration();
    JobClient client = new JobClient((JobConf) conf);
    RunningJob parentJob =
      client.getJob((org.apache.hadoop.mapred.JobID) JobID.forName( conf.get("mapred.job.id") ));
    mapperCounter = parentJob.getCounters().getCounter(DocumentsCount.NUMDOCS);
  }
  ```
Profile B Overview

- Profile A and Profile B are separate jars which can run independently
- Try to scale your application by having multiple reducers and partitioner
- In profile B, we generate article summary for any given article (Document).
Profile B - Approach

- Split by a period followed by single space (.) to fetch sentences
- Then only preprocess unigrams i.e., removing non-alphanumeric characters
- Multiple inputs required: Output of previous job (Profile A - Job 3) and given article.
- You can implement this in either of the two ways:

  1. Use `MultipleInputs.addInputPath()` and join them on keys. (Reduce Side Join)
  2. Persist output from previous job in Hadoop’s DistributedCache.
Reduce side join - Job1

**Mapper:**
- Two Mappers each passing corresponding input file with docID as key
- Append value with some character to identify which mapper parsed that input data

**Reducer:**
- Inside reduce() - Remove the characters added in values from previous mapper
  - Load TD-IDF data into lookup table (HashMap: key - DocumentID+unigram, value: TF-IDF value)
  - Set Document content from Mapper2 outputs
- Split the given article on a period (.) followed by whitespace ( ) to get individual sentences
- Assign index to each sentence based on its order of occurrence
- Use the lookup table to calculate sentence TF-IDF (Use top 5 unigrams from each sentence)
  - Maintain TreeMap to make sorted list
  - Clean up unigrams
  - Fetch TF-IDF value from previous hashmap for each unigram in document
  - As the size grows to 5, remove lowest unigram with lowest tf-idf value
- Output top 3 sentences preserving its order of occurrence in original article

```java
MultipleInputs.addInputPath(<job>, <Path>, <InputFormat.class>, <TaggingMapper.class>);
```
Hadoop Distributed Cache - Job 1

• **Mapper:**
  • Single Mapper
  • Load TF - IDF data from distributed cache into lookup table
  • Same steps in previous slide’s reducer

• **Reducer:**
  • Identity reducer - No reducer required

In Driver:

```java
DistributedCache.addCacheFile(new Path(args[0].toUri()), job.getConfiguration());
```
Combiner Class

- Make sure that you use Combiner correctly
- The **input of combiner class** should match with the **output of Mapper Class**
- The **output of combiner class** should match **input of reducer class**
- Combiner functionality must be : Commutative and associative
- Hadoop doesn’t guarantee on how many times a combiner will be called for each output key
- Do NOT use Reducer class as your combiner unless sure!
References

Questions?