CSCI 493.71

Project 2

MapReduce Algorithm Design

Favian Flores

**Function 1: Computing TF-IDF matrix**

Step 1: importing and cleaning up data

We read in the data from the text file and save it in the RDD as one element per line in the file (line). Take count of the elements in this step to keep track of the total number of documents in the corpus (total\_docs).

Input: Project2\_demo.txt

Output: (line)

Then, we map each element by splitting the document using ‘ ’ as a delimiter, taking the first element (doc\_id), creating pairs with all the other words (term) in the element, and flattening the results.

Input: (line)

Output: (doc\_id, term)

We then clean up the elements by filtering out any blanks terms such as empty spaces, newline characters, etc.

After that, we run another filter removing any terms that don’t start with “dis\_” or “gene\_”, but we keep both RDDs since we are going to use each one for different steps. The data format remains the same after both filters.

Step 2: Computing the numerator for TF

For this step we take the second RDD created in step 1, filtered for both empty terms and terms that don’t start with “dis\_” or “gene”. We map it using the doc\_id and the term as a key and a 1 as the value.

Input: (doc\_id, term)

Output: ((doc\_id, term), 1)

We then combine and reduce using addition. The value produced is the number of times each term appears in each document (tf-numerator).

Input: ((doc\_id, term), 1)

Output: ((doc\_id, term), tf-numerator)

Although we have already calculated the numerator, we map the data with the doc\_id as the key one more time so that it will be easier to join with the denominator later on.

Input: ((doc\_id, term), tf-numerator)

Output: (doc\_id, (term, tf-numerator))

Step 3: Compute the denominator for TF

Since the denominator requires the number words per document, we will take the first RDD created in Step 1, which only filters for blanks term. We map each element using doc\_id as the key and a 1 as the value.

Input: (doc\_id, term)

Output: (doc\_id, 1)

Then we combine and reduce using addition. The value produced is the number of words per document (tf-denominator). Since doc\_id is already the key, we don’t need to map it again.

Input: (doc\_id, 1)

Output: (doc\_id, tf-denominator)

Step 4: Compute IDF

For this step we take the second RDD created in step 1, filtered for both empty terms and terms that don’t start with “dis\_” or “gene”. For the IDF function, we need the number of documents that each word appears in. Since each word can appear in a document more than once, we need to filter the RDD to only have distinct elements. Then we can map each term as the key and a 1 as the value.

Input: (doc\_id, term)

Output: (term, 1)

We then combine and reduce using addition. The value produced is the number of documents each word appears in. Using that number and the total\_docs number from step 1, we can map it using a function which will calculate the IDF value.

Input: (term, 1)

Output: (term, IDF)

Step 5: Join the RDDs and compute the TFIDF matrix

We start by joining the RDDs containing the TF numerator and TF denominator.

Input: (doc\_id, (term, tf-numerator)) and (doc\_id, tf-denominator)

Output: (doc\_id, ((term, tf-numerator), tf-denominator))

Then we map each element using the term as the key, so we can join it with the RDD containing the IDF value, and a (doc\_id, TF) pair by dividing the tf-numerator and tf-denominator.

Input: (doc\_id, ((term, tf-numerator), tf-denominator))

Output: (term, (doc\_id, TF))

Then we join this RDD with the RDD containing the IDF values from step 4.

Input: (term, (doc\_id, TF)) and (term, IDF)

Output: (term, ((doc\_id, TF), IDF))

We map each value with term as the key, and a (doc\_id, TFIDF) pair as the value by multiplying the TF and IDF values.

Input: (term, ((doc\_id, TF), IDF))

Output: (term, (doc\_id, TFIDF))

Then we combine by the key bringing all the values as a list of pairs and the TFIDF matrix is complete.

Input: (term, (doc\_id, TFIDF))

Output: (term, [(doc\_id1, TFIDF1), (doc\_id2, TFIDF2)…])

**Function 2: Computing Similarity**

Step 1: Check to see if query term exists

We start by checking to see if the term exists in the RDD matrix, if it doesn’t the function returns an empty list and the program will ask for another term.

Step 2: Filter out query term

Once we know that the query term exists, we filter out the query term from the matrix so that we don’t calculate and print it as a final result.

Step 2: Mapping query term matrix values to all the other terms

We can map the matrix values from that term into each of the other terms.

Input: (term, [(doc\_id1, TFIDF1), (doc\_id2, TFIDF2)…]) and (query\_term, [(doc\_id1, TFIDF1), (doc\_id2, TFIDF2)…])

Output: (term, ([(doc\_id1, TFIDF1), (doc\_id2, TFIDF2)…], [(query\_doc\_id1, query\_TFIDF1), (query\_doc\_id2, query\_TFIDF2)…])))

The resulting matrix has all the data needed to calculate all the similarity scores.

Step 3: Calculating similarity

We map matrix values once again using a custom function which will match TFIDF values from the query term and the other term using doc\_id values in order to calculate the final similarity scores.

Input: (term, ([(doc\_id1, TFIDF1), (doc\_id2, TFIDF2)…], [(query\_doc\_id1, query\_TFIDF1), (query\_doc\_id2, query\_TFIDF2)…])))

Output: (term, similarity)

Step 4: Sorting and taking the top 10

At this point, all the calculations are complete, we just use the sortBy and take functions to sort the data in descending order and then take the 10 highest values. The function returns it as a list and the interface prints it.

**Future Improvements**

A potential future improvement for the first function could be to reduce the number of child RDDs created. Creating a separate RDD for each part of the TFIDF function means that more joins need to be made at the end. We may be able to keep all the data and calculations in one RDD to improve computation time.

A potential future improvement for the second function could be to filter out any term term pairs with 0 similarity. The numerator always returns 0 when none of the terms doc\_id’s match. It would be possible to set a function that checks for this and removes those elements prior to the similarity calculation to improve computation time.