Lloyd Massiah

Gael Blanchard

Big Data Project 2: Documentation

TFIDF:

Term Frequency:

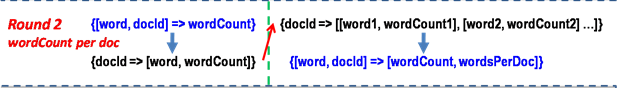


* The first map transformation splits each line of the RDD, into a list of all its elements
* The next transformation is flatMap, which calls the function Mapper

1. Mapper takes each element of the RDD and outputs a key-value pair
2. First it filters out all terms not starting or ending with “gene\_” and “\_gene” and “disease\_” and “\_disease”
3. It takes the first element of the RDD element(DocId), makes a key-value pair for every term where DocId is the key, and the term is the value
4. Output ---> a list of [(DocId, term1), (DocId, term2….]
5. flatMap finally splits that list into elements that can be individually acted upon by RDD transformations and actions

* the next map transformation is called and takes each element and creates a key-value pair where the element is the key and the value is 1, i.e. ((docId, term), 1)
* reduceByKey is called to count all instances of unique terms in each document

Word Count per Document:



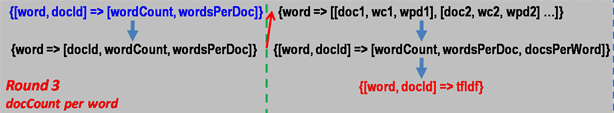
* The first map transformation calls the SecondMapper function which outputs key-value pair in the form (DocId, (Word, WordCount))

1. SecondMapper simply creates a new tuple by making the DocId the key and the Term and its WordCount, in its document, the value, i.e. (docid, (Word, WordCount))

* ReduceByKey is called to aggregate all the terms with their wordcounts into one value, using their respective documents as the key
* ReduceByKey undoes the tuple structure from the SecondMapper output, therefore CreateTuple is called to restructure the list of terms and their counts into individual tuples
* flatMap is used and calls WordCountPerDoc

1. WordCountPerDoc adds the WordCount of each key’s(DocId) value(word, wordCount) to form the total number of words in the document
2. Then it creates a list and appends a tuple of the form ((word, DocId), (wordCount, WordsPerDoc))
3. Finally the flatMap transformation splits the list into individual elements in the RDD

Document Count per Word:



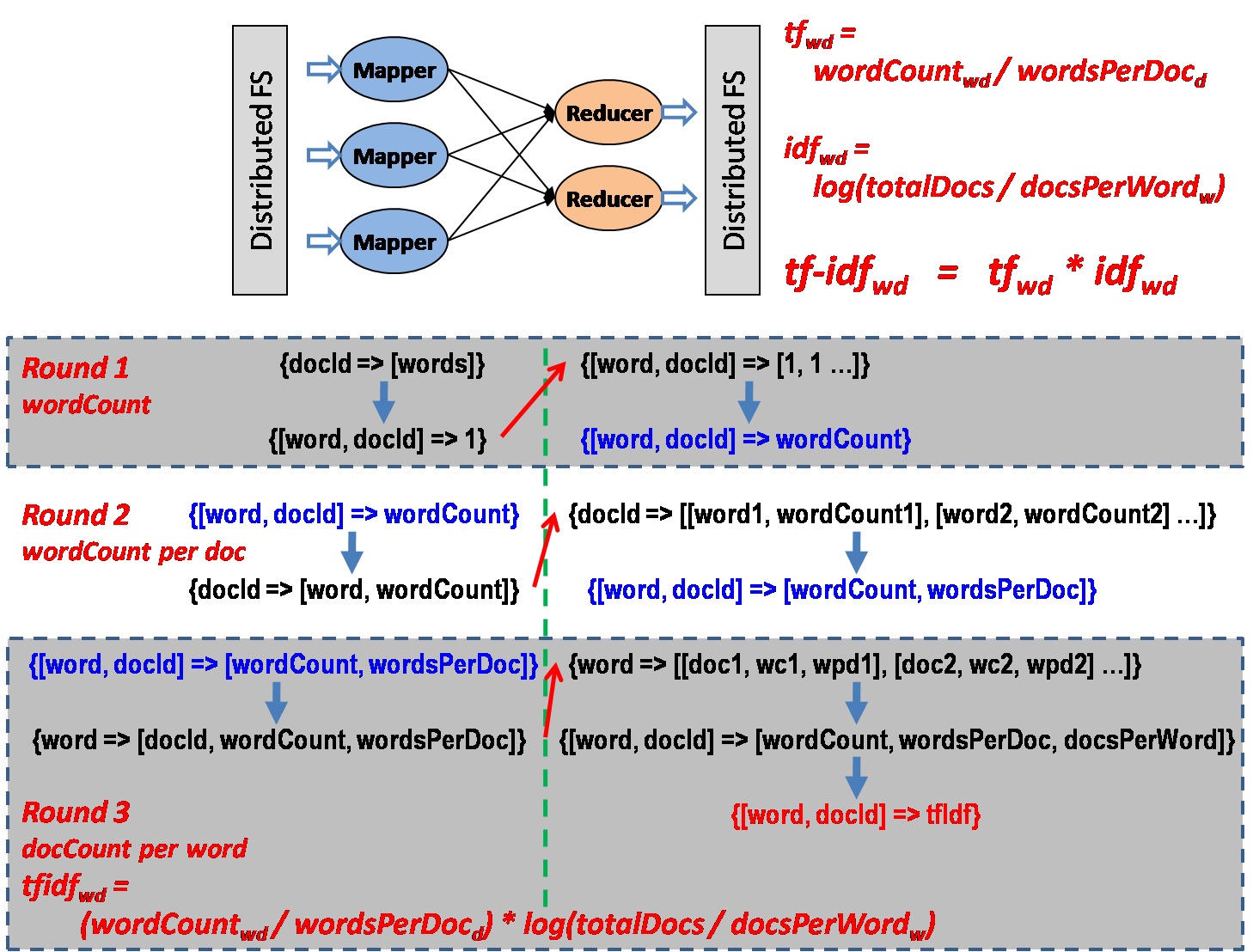
* The first map transformation calls the ThirdMapper function
* ThirdMapper outputs a tuple of the form (word, (docId, WordCount, WordsPerDoc))
* ReduceByKey is called to aggregate all docs, wordCounts, and WordsPerDoc, where the key is the word
* CreateSecondTuple is called to regroup the docId, wordCount, WordsPerDoc, because ReduceByKey undoes the tuple structure and makes it one list
* flatMap is used and it calls CountDocsPerWord which calculates the number of documents the word appears in
* CountDocsPerWord accomplishes this by counting each tuple in the value part of the key-value pair and outputs a new key-value pair of the form ((word, docId), (wordCount, wordsPerDoc, docsPerWord))

TF-IDF:



* The tf-idf is calculated in a new RDD of the same name
* A map transformation is used where the input is the output from the previous step
* The number of documents can be passed as a constant value. I opted to directly write the number of documents for this instance
* Tf-idf = (wordCount/wordsPerDoc)\*log(totalNuqmberofDocs/docsPerWord)

Diagram for TFIDF Algorithm:



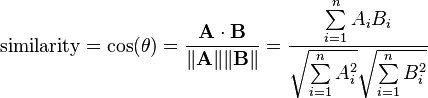
Creating Term Vectors:

* After calling the TfIdf function, the output becomes (word, tfidf)
* In order to create the term vector a GroupByKey transformation is used to bring all tfidfs using the word as the key
* The map transformation turns the result of the groupByKey into an iterable list
* Output is (term, [tfidf1, tfidf2,…,tfidfN])

Query Term Vector:

* An RDD is created specifically to hold the query term vector
* It’s obtained from the dataset by using the filter transformation to return it if it’s in the dataset

Semantic Similarity:



* After filtering for the query vector, another filter is called on the original dataset with all the term vectors, to return an RDD that doesn’t contain the query term vector. This will be necessary for the next step
* After filtering, the Cartesian transformation is called which makes all possible pairs out of all elements of an RDD with another dataset. In this instance Cartesian was called on all the term vectors, and the query vector was used as the second dataset. This in turn creates a tuple of the form (Query term vector, B-vector1), (Query term vector, B-vector2)… and so on
* Using these pairs, the SemanticSimilarity function can be called, in a map transformation, which will find the relevance of the second vector with respect to the query vector
* Output will be: ((query term, term), semantic similarity)

Sorting Semantic Similarity:

* In order to sort the semantic similarity, a map transformation is called that swaps the positions of the key and value from Semantic Similarity output
* Doing that allows the RDD to use the transformation, SortByKey to sort all the semantic similarities in descending order.
* Afterwards, another map transformation is used to re-swap the key and value to become the final output, sorted by value

Running the Code:

* the program requires the file path for the input file and the query term as command line arguments
* the code is run, after entering spark’s bin directory through the command, by spark-submit file\_path\_for\_source\_code file\_path\_for\_project\_data query\_term