Dorjee Gyaltsen December 12, 2020

Nusrat Chowdhury

CSCI 496

Professor Lie Xie

**Summary**

This program utilizes MapReduce design with a common web analysis algorithm, TF-IDF which is the Term frequency inverse document. Essentially it is a statistical measure that evaluates how relevant a word is to a document in a collection of documents. In short we multiplied the term frequency of a specific term in that document with the inverse document frequency which is the logarithm of the n number of documents divided by the count of the term in each document.

Text, letter

Description automatically generated[[1]](#footnote-1)

This results to the TF-IDF score which shows us the relevance of that term, allowing us to retrieve terms that are similar to the query term we have inputted.

**Sources and Requirements:**

* TF-IDF
* For more info on TF-IDF, <https://monkeylearn.com/blog/what-is-tf-idf/>
* Apache Spark
* For this program we are using PyCharm, which we imported pyspark by downloading packages.
* If run through terminal download Apache Spark and follow the documentation, <https://spark.apache.org/docs/latest/quick-start.html>
* Python 3.9 (PyCharm)

**Running the program**

* The programmed is run through PyCharm
* To run this the path to the text file in the same directory if in a different directory add a path to it.

**Input**

*spark-submit file\_name.py txt\_file query\_term*

*spark-submit main.py project2\_demo.txt dis\_block\_glioma\_dis*

**Output**

* Makes an output file consisting the results.

The term you entered is: dis\_block\_glioma\_dis  
  
Term tf-idf   
gene\_plc\_gamma\_gene 1.0  
gene\_u87\_gene 0.48061081041475057  
dis\_c6\_glioma\_dis 0.15868695839711647  
dis\_gbm\_dis 0.05911747370309243  
dis\_glioma\_dis 0.03496778326010016  
dis\_glioblastoma\_dis 0.015593126967805874  
dis\_tumor\_dis 0.009207788492463485  
gene\_epidermal\_growth\_factor\_gene 0.0  
gene\_alphavbeta3\_ligand\_gene 0.0  
gene\_vitronectin\_gene 0.0

**Launching Spark on YARN**

Ensure that HADOOP\_CONF\_DIR or YARN\_CONF\_DIR points to the directory which contains the (client side) configuration files for the Hadoop cluster. These configs are used to write to HDFS and connect to the YARN ResourceManager. The configuration contained in this directory will be distributed to the YARN cluster so that all containers used by the application use the same configuration. If the configuration references Java system properties or environment variables not managed by YARN, they should also be set in the Spark application’s configuration (driver, executors, and the AM when running in client mode).

There are two deploy modes that can be used to launch Spark applications on YARN. In cluster mode, the Spark driver runs inside an application master process which is managed by YARN on the cluster, and the client can go away after initiating the application. In clientmode, the driver runs in the client process, and the application master is only used for requesting resources from YARN.

**File does not exist: Spark runs ok in local mode but can’t find file when running in YARN**

Again as in #2, all the necessary files/ jars should be located somewhere accessible to all of the components of your cluster, e.g. an FTP server or a common mounted drive.

# Fix for log4j WARN No appenders could be found for logger, Please initialize the log4j system properly

log4j:WARN No appenders could be found for logger log4j:WARN Please initialize the log4j system properly. log4j:WARN See https://logging.apache.org/log4j/1.2/faq.html#noconfig for more info.

Please Check this website for more info on the error

“https://www.journaldev.com/10721/log4j-warn-no-appenders-could-be-found-for-logger-please-initialize-the-log4j-system-properly”

**Breakdown of the Project**

***record reader***

The record reader translates an input split generated by input format into records. The purpose of the record reader is to parse the data into records, but not parse the record itself. It passes the data to the mapper in the form of a key/value pair. Usually, the key in this context is positional information and the value is the chunk of data that composes a record. Customized record readers are outside the scope of this book. We generally assume you have an appropriate record reader for your data.

def txt\_to\_doc(txt):  
 splitted = txt.split()  
 # returns doc id and the word  
 return splitted[0], [w for w in splitted[1:] if DIS\_REGEX.match(w) or w == QUERY]  
  
def doc\_to\_words(doc):  
 words = doc[1]  
 count\_words = len(words)  
 res = []  
 for word in words:  
 res.append(((doc[0], count\_words, word), 1)) #append helps us combine the lists  
 return res

***map***

In the mapper, user-provided code is executed on each key/value pair from the record reader to produce zero or more new key/value pairs, called the intermediate pairs. The decision of what is the key and value here is not arbitrary and is very important to what the MapReduce job is accomplishing. The key is what the data will be grouped on and the value is the information pertinent to the analysis in the reducer. One major differentiator between MapReduce design patterns is the semantics of this pair.

doc\_count = docs.count()  
#output.write(f'Document frequency: {doc\_count}\n')  
# flatmap allows us to create a 1 to many relationships  
doc\_words = docs.flatMap(doc\_to\_words) \  
 .reduceByKey(lambda a, b: a + b) # term count per doc  
tf = doc\_words.map(lambda word: (word[0][2], [(word[0][0], word[1]/word[0][1])])) \  
 .reduceByKey(lambda a, b: a + b)  
  
# log function to find the idf and tf by dividing with the total length.  
tf\_idf = tf.map(lambda word: (word[0],  
 (math.log(doc\_count / len(word[1]), 10), word[1])))  
  
# tf-idf computed by multiplying  
join\_tf\_idf = tf\_idf.map(lambda word: (word[0], {i[0]: word[1][0] \* i[1] for i in word[1][1]}))

# inputting RDD through the function, with the result of the function  
# being the new value of each element in the resulting RDD.  
txt = sc.textFile(filename)  
docs = txt.map(txt\_to\_doc)

Similar to map, it returns a new RDD by applying a function to each element of the RDD, but output is flattened.  
Also, function in flatMap can return a list of elements (0 or more)

Next is the mapper code that parses and prepares the text. Once some of the punctuation and random text is cleaned up, the text string is split up into a list of words.

A Combiner, also known as a **semi-reducer,** is an optional class that operates by accepting the inputs from the Map class and thereafter passing the output key-value pairs to the Reducer class.

The main function of a Combiner is to summarize the map output records with the same key. The output (key-value collection) of the combiner will be sent over the network to the actual Reducer task as input.

# flatmap allows us to create a 1 to many relationships  
doc\_words = docs.flatMap(doc\_to\_words) \  
 .reduceByKey(lambda a, b: a + b) # term count per doc  
tf = doc\_words.map(lambda word: (word[0][2], [(word[0][0], word[1]/word[0][1])])) \  
 .reduceByKey(lambda a, b: a + b)

Combiner is optional yet it helps segregating data into multiple groups for Reduce phase, which makes it easier to process.

output.writelines([f'{word} {item}\n' for (word, item) in terms if word != QUERY])  
output.close()

Key-Value pair output.

1. Formula of the term frequency and Inverse Document frequency. [↑](#footnote-ref-1)