tf-idf parallel implementation[⋆]

Roberta Conrad[1,2], Dora Ranilovic[1,2], and Zigfrid Zvezdin[1,2,3]

[1] Ecole Polytechnique, France
https://www.polytechnique.edu/
[2] HEC Paris, France
https://www.hec.edu/
[3] Moscow Institute of Physics and Technology, Russia
https://mipt.ru/

Abstract. tfidf, short for term frequencyinverse document frequency, is a statistic which allows to define the importance of a word relatively to a document corpus. We propose parallel implementations of algorithms to compute tf-idf score on Hadoop/HDFS and on Spark using Python.

Keywords: tf-idf · TFIDF · word frequency.

1 tf-idf Algorithm

1.1 Definition

tf-idf is a widely used statistic in the field of information retrieval, used to quantify the relevance of a query or key word to a document inside a corpus. It is commonly used in search engine optimization and text mining. The statistic is designed to assigns higher relevance to a term if it occurs often in the document, but penalize it if it occurs in many different documents, i.e. it is not unique or specific to one or few documents. The general formula to compute the tf-idf score for term t and document d in corpus D is:

$$tf.idf(t,d,D) = tf(t,d) \cdot idf(t,D)$$

where

$$tf(t,d) = \frac{n(t,d)}{N(d)} \text{ is the term frequency function}$$

$$idf(t,D) = \log \frac{|D|}{d(t)} \text{ is the inverse document frequency function}$$

$$n(t,d) \text{ is the number of times t occurs in d}$$

$$N(d) \text{ is the number of unique terms in d}$$

$$d(t) \text{ is the number of documents in D which contain t}$$

[⋆] Student project in Ecole Polytechnique

1.2 Baseline Algorithms

This section describes in detail the two baseline implementations of the tf-idf algorithm we developed and tested: a Python/Hadoop algorithm, and a Spark algorithm.

Python Hadoop The first implementation of tf-idf is a traditional MapReduce algorithm, designed to be run on a Hadoop cluster with only Python 2.7+ dependencies. It consists of consecutive Map and Reduce jobs, take a corpus of text documents as the input and compute the (nonzero) tf-idf scores for each word-document pair in the corpus. The following sections discuss the code of all three jobs, as well as the inputs and outputs. As this approach utilizes the traditional MapReduce framework, the inputs and outputs are always key-value pairs written in the format $< (key, value) >$. Additionally, the output of each *reduce* step, is the raw input of the *map* step of the following job.

Job 1: Map
Input: $(doc, contents)$
Output: $((doc, term), 1)$
Description: Read text document line by line, envoking the source document name. Split each line into terms (words), convert to lowercase and remove punctuation, and emit the document name and term, together with the digit 1. This choice of output was made so that after the shuffle and sort step, the documents and identical words in each document will be grouped together to be able to perform two types of counts.
Code:

```
import os
import sys

for line in sys.stdin:
    line = line.strip() # remove whitespace
    terms = line.split(" ") # create list of words
    path = os.environ['mapreduce_map_input_file'].split("/")
    # get full path of the text document
    docname = path[-1]
    for term in terms:
        term = term.strip('''!()-[]{};:'",<>./?@#$%^&*_~''').lower()
        print('%s\t%s' % (docname + '_' + term, 1))
```

Job 1: Reduce
Input: $((doc, term), 1)$
Output: $((term, doc), (N, n))$
Description: This reduce step is computationally the most intensive as it performs two important types of counts at the same time: unique term per document (N) and term count (n). These two numbers together form the term frequency

(tf) part of the tf-idf expression. They are computed using nested loops and local lists, and taking advantage of the sorted nature of the inputs. The term count is associated to each (term, document) pair and is computed first, in the inside loop: the memory of the one previous term is kept and compared to the current term; if they match the counter is incremented an we move to the next term; once they no longer match, the term and its count are saved in a list, holding all terms in the current document. Once the end of the document is reached, all terms in the list are emitted one-by-one, along with the document name, their respective counts in the document and the length of the final list, which represents the number of unique terms in the document.
Code:

```
import sys

current_doc = None #Initialize helper variables
current_term = None
term_list = []
current_term_count = 0

for line in sys.stdin:
    line = line.strip()
    pair, count = line.split('\t', 1)
    doc, term = pair.split('_', 1)

    try: # convert count (currently a string) to int
        count = int(count)
    except ValueError:
        continue
        # count was not a number, so silently ignore/discard this line

    if current_doc == doc:
    #outside loop, check if still in the same document
        if current_term == term:
            current_term_count += count
        else:
            term_list.append((current_term, current_term_count))
           #hold term and final count
            current_term = term
            current_term_count = count

    else:
        term_list.append((current_term, current_term_count))
        #end of doc; emit all terms and counts
        if current_doc:
            for t, ct in term_list:
                print('%s\t%s' % (t+'_'+current_doc,
                                    str(len(term_list))+'_'+(str(ct))))

        current_term_count = count #reset counters
        current_doc = doc
        current_term = term
        term_list = [] #reset list

if term_list: #emit all terms except last in last doc
    for t, ct in term_list:
        print('%s\t%s' % (t+'_'+current_doc,
                          str(len(term_list)+1)+'_'+(str(ct))))

print('%s\t%s' % (current_term+'_'+current_doc, #emit last term in last doc
                  str(len(term_list)+1)+'_'+str(current_term_count)))
```

Job 2: Map
Input: $((term, doc), (N, n))$
Output: $(term, (docname, N, n, 1))$
Description: The main purpose of this map step is to uncouple the key and pass only the term as the key to the next reducer, as we now want to calculate the document frequency of each term, and therefore need the terms grouped. Although we would achieve the same grouping by keeping the document name in the key (separated by an underscore, for example, from the term), we chose to perform the separation step in the mapper in order to take advantage of parallelization, and avoid performing the same task in the reducer.
Code:

```
import sys
for line in sys.stdin:
    pair, vals = line.split('\t')
    term, docname = pair.split('_', 1)
    N, n = vals.split('_')
    try:
        n = int(n)
        N = int(N)
    except ValueError:
        continue
    print('%s\t%s' % (term, docname+'_'+str(N)+'_'+str(n)+'_'+str(1)))
```

Job 2: Reduce
Input: $(term, (docname, N, n, 1))$
Output: $((term, docname), (N, n, d))$
Description: Much like a simple word count, count the occurrence of each term, which thanks to the previous reducer will occur once per each document it appears in. Since we need the rest of the information found in the values of

the key, we need to keep them in a local list until we reach the last occurrence of the term, at which point we emit all term-document pairs for the current term.
Code:

```
import sys
current_term = None
doc_list = []
current_count = 0
term = None
doc = None

for line in sys.stdin:
    term, rest = line.split('\t', 1)
    doc, N, n, count = rest.rsplit('_', 3)

    try: # convert count (currently a string) to int
        count = int(count)
    except ValueError:
        # count was not a number, so silently ignore/discard this line
        continue

    if current_term == term:
        doc_list.append((doc, N, n))
        current_count += count
    else:
        if current_term:
            for document, N, n in doc_list:
                print('%s\t%s' % (current_term + '_' + document,
                        str(N)+'_' + str(n) + '_' + str(current_count)))
            doc_list = []
            n_list = []
        current_count = count
        current_term = term
        doc_list.append((doc, N, n))
if current_term == term: #emit last term
    for document, N, n in doc_list:
        print('%s\t%s' % (current_term + '_' + document,
                         str(N)+'_' + str(n) + '_' + str(current_count))
```

Job 3: Map
Input: $((term, docname), (N, n, d))$
Output: $((term, docname), tfidf)$
Description: Having all necessary elements, compute the final tf-idf score for each *(term, document)* pair.
Code:

```
import sys
import math

for line in sys.stdin:

    pair, vals = line.split('\t', 1)
    N, n, d = vals.split('_', 2)
    try:
        N = int(N)
        n = int(n)
        d = int(d)
    except ValueError:
        continue
    tf = n/N
    idf = math.log(10000/(1+d)) # use 10,000 as max number of documents
    tfidf = tf * idf
    print('%s\t%s' % (pair, tfidf))
```

Job 3: Reduce
Input: $((term, docname), tfidf)$
Output: $((term, docname), tfidf)$
Description: Analogous to a pass task, simply emit the input without any additional treatment.
Code:

```
import sys

for line in sys.stdin:
    print(line)
```

1.3 Other Designed Algorithms

Spark 1 This algorithm is good for big collection of small documents as it parallelizes the documents over machine with function. *wholeTextFiles()*.
Code:

```
import math

def word_freq(doc):
    """Function to compute frequencies of occurrences of words
    in a document.

    Args:
        doc: Document = string with lines separated by '\n'.
    Returns:
        List of pairs (word, frequency) corresponding to the doc.
    """
    lines = doc.lower().split('\n')
    words = []
    for line in lines:
        for word in line.split():
            clean_word = ''.join(x for x in word if x.isalpha())
            words.append(clean_word)
    occurrences = []
    for word in words:
        occurrences.append(words.count(word))
    N = len(words)
    freqs = [w/N for w in occurrences]
    wordfreq = set(zip(words, freqs))
    return wordfreq

def word_to_key(x):
    """Function to compute frequencies of occurrences of words
    in a document.

    Args:
        x: Pair (document_name,(word,frequency)).

    Returns:
        List of pairs (word,(document_name,frequency)).
    """
    file = x[0]
    l = []
    for pair in x[1]:
        l.append((pair[0], (file, pair[1])))
    return l


path = '/user/hadoop/tfidf/input/*.txt'
corpus = sc.wholeTextFiles(path)

# compute number of documents in the corpus
num_docs = corpus.count()
# compute tf
tf = corpus.map(lambda x: (x[0].split('/')[-1],
                           word_freq(x[1]))).flatMap(word_to_key)
# compute idf
idf = tf.map(lambda x: (x[0], 1)).reduceByKey(
    lambda x, y: x+y).map(lambda x: (x[0], math.log(num_docs/x[1])))
# compute tfidf
tfidf = tf.join(idf).map(lambda x: (
    x[0], (x[1][0][0], x[1][0][1]*x[1][1]))).collect()
```

Spark 2 This document is good for a small number of document of any size because it parallelizes the lines of documents over different machines but does in a loop for each document.

Code:

```
import os
import subprocess
import math


# The solution D = glob.glob("/user/hadoop/tfidf/input/*.txt")
# returns D = [],
# so we need to use the command line in order to get the list
# of files in the hdfs directory.
cmd = 'hdfs dfs -ls /user/hadoop/tfidf/input/'
files = subprocess.check_output(
    cmd, shell=True).decode('utf-8').strip().split('\n')
D = [x.split(' ')[-1] for x in files]
D = D[1:]

num_docs = len(D)

all_words_tf = sc.parallelize([])
all_words_idf = sc.parallelize([])

# for each document in the list keep unique words in all_words_idf and
# frequencies in all_words_tf
for doc in D:
    d = sc.textFile(doc)
    words = d.flatMap(lambda s: s.lower().split()).
    map(lambda w: ''.join(x for x in w if x.isalpha()))
        # x.isalpha() == True iff x is an alphabetic character.
    all_words_idf = all_words_idf.union(words.distinct())
    N = words.count()
    occurrences = words.map(lambda x: (x, 1)).reduceByKey(
        lambda x, y: x+y).map(lambda x: (x[0], (doc, x[1]/N)))
    all_words_tf = all_words_tf.union(occurrences)
```

```
# compute idf
idf = all_words_idf.map(lambda x: (x, 1)).reduceByKey(
    lambda x, y: x+y).map(lambda x: (x[0], math.log(num_docs/x[1])))

# compute tfidf
tfidf = all_words_tf.join(idf).map(lambda x: (
    x[0], (x[1][0][0], x[1][0][1]*x[1][1]))).collect()
```

---

2 Experimental analysis

In order to compare the performance of the three different implementations, we designed two different tests considering different structures of a text collection with increasing sizes. We differentiated between a collection of documents containing many short documents; and a collection containing few long documents:

- Collections with a fixed document size: we fixed the number of words at 100 words per document, changing the number of documents in (100,200,300,400,500,1000).
- Collections with a fixed number of documents: We ran our tf-idf algorithms on 100 documents containing (100,200,300,400,500,1000) words.

For easy reproducability this report contains all code and scripts that have been used to produce the following analysis and can be recreated on AWS Elastic Map Reduce cluster.

Limitations
In order to control the size of the documents used to test our algorithm, we decided to write a script creating documents for testing purposes. One document containing 100 words has a size of 11 KB when created by our script.

2.1 Comparison of the implementations

The results of experiments are here. See Fig. 1 and Fig. 2.

Spark 1, the one that uses wholeTextFiles() and puts many documents on machines is the best one in terms of clock time in both cases of longdoc and manydoc. In the terms of ”time” Spark 2 is only better for longdoc but is bad for manydoc because there were some lost tasks so the tfidf was not computed. The MapReduce implementation is the slowest option of all three and only better then the first Spark implementation as documents become longer then 300 Words (approx. 30 KB). So overall the most reliable algorithm is Spark 1.

2.2 Practical Instructions to run tests on the cluster

1. Create text documents on hadoop home directory
   - create the input directory on /user/hadoop local file system

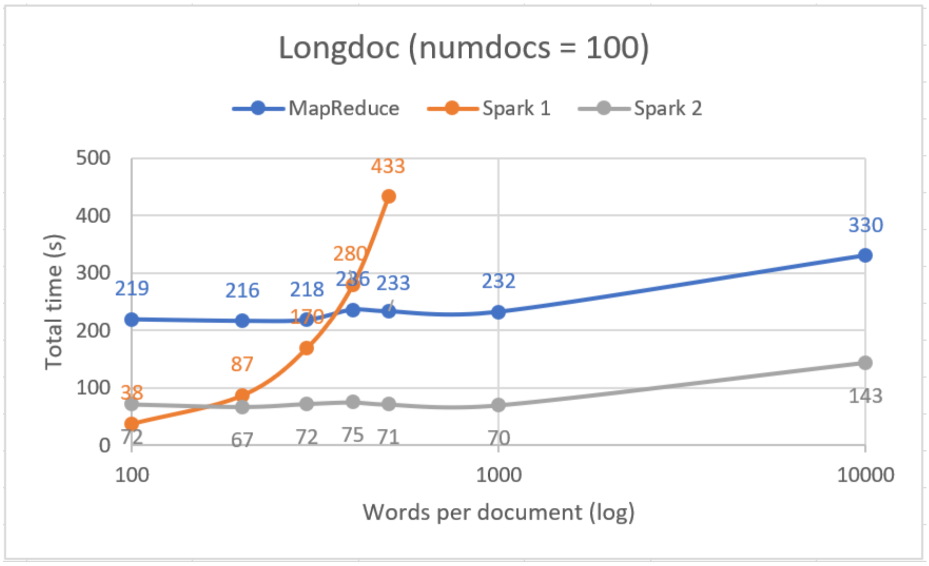

Fig. 1. No special problems for longdoc.

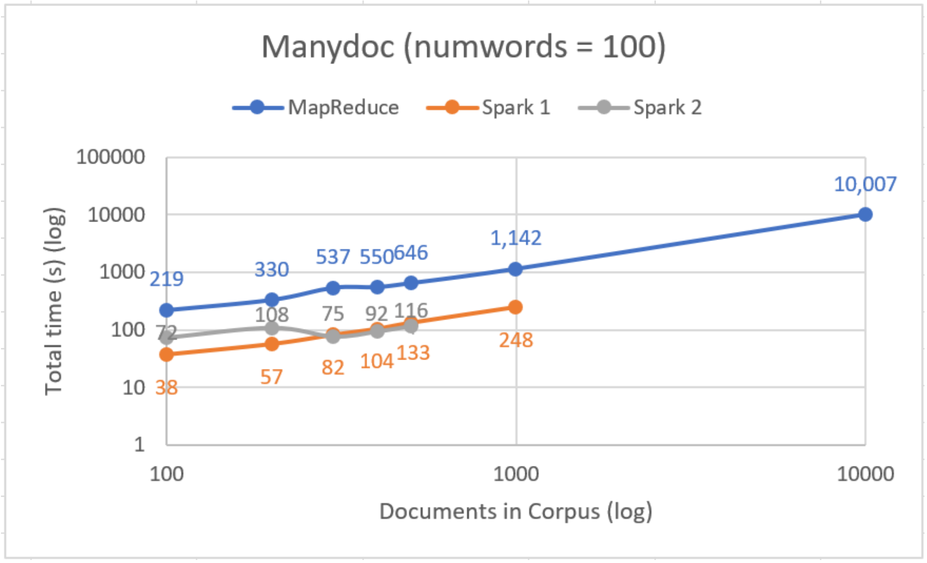

Fig. 2. Note that three lasts points for Spark 2 correspond to **failed** tf-idf computation since some tasks were lost.

  - in /input, import the python script and execute the text_generator() function to create documents needed, specify params as described below
  - move only the .txt to the hdfs hadoop input directory
2. Download and run the MapReduce algorithms
  - give permission to access all documents
  - execute all jobs through bash script runjobs.sh
  - after retrieving the time information, clear input directories with clear.sh

```
python -m nltk.downloader brown
mkdir input
cd input
wget https://raw.githubusercontent.com/ZiggerZZ/DB-tf-idf/
master/text_docs/text_generator.py
python -c 'from text_generator import create_docs;
create_docs(number_of_words_per_doc = 200, num_doc = 10,startnr = 23)'
cd
hdfs dfs -mkdir /user/hadoop/tfidf/input
hdfs dfs -put input/*.txt /user/hadoop/tfidf/input

wget https://raw.githubusercontent.com/ZiggerZZ/DB-tf-idf/master/mapper1.py
wget https://raw.githubusercontent.com/ZiggerZZ/DB-tf-idf/master/mapper2.py
wget https://raw.githubusercontent.com/ZiggerZZ/DB-tf-idf/master/mapper3.py
wget https://raw.githubusercontent.com/ZiggerZZ/DB-tf-idf/master/reducer1.py
wget https://raw.githubusercontent.com/ZiggerZZ/DB-tf-idf/master/reducer2.py
wget https://raw.githubusercontent.com/ZiggerZZ/DB-tf-idf/master/reducer3.py
wget https://raw.githubusercontent.com/ZiggerZZ/DB-tf-idf/master/runjobs.sh
chmod +x *.py
chmod +x *.sh
time sh runjobs.sh
sh clear.sh
```

3 Code Appendix

3.1 text_generator.py

Requirements: Requires installing the nlkt library brown.
Functionality: The function create_docs takes as input parameters the number of words per document, number of documents, and start number of the document to be taken into account when naming files. The function creates the specified number of documents at the specified length by taking samples of the brown corpus. It randomly samples lines of twenty words at the same time. The Brown University Standard Corpus of Present-Day American English (or just Brown Corpus) was compiled in the 1960s as a general corpus. It contains 500 samples of English-language text, totaling roughly one million words, compiled from works published in the United States in 1961. It is a suitable library to sample from in

order to test our algorithm as it represents how language is used in reality.
Code:

```
from nltk.corpus import brown
import random
corpus_length = len(brown.words())
hardcopy = brown.words()


def create_docs(number_of_words_per_doc=200, num_doc=10, startnr=0):
    # control number of words per doc
    # and number of documents
    # we fix the line length at 20
    line_length = 20
    number_of_lines = int(number_of_words_per_doc / line_length)

    for i in range(0, num_doc):
        # create new file with writing + permission
        new_file = open("textdoc"+str(number_of_words_per_doc) +
                        "words" + str(i+startnr)+".txt", "w+")
        for line in range(0, number_of_lines):
            words = list(map(
                lambda x: hardcopy[x:x+line_length],
                random.sample(range(corpus_length), line_length)))
            sentences = list(
                map(lambda x: ' '.join(word for word in x), words))
            text = ''.join(map(str, sentences))
            new_file.write(text + "\n")
        new_file.close()
        if (i % 10 == 0):
            print("You created "+str(i)+" files! "+str(num_doc-i)+" left")
```

3.2 runjobs.sh

Code:

```
#!/usr/bin/env bash
hadoop jar /usr/lib/hadoop-mapreduce/hadoop-streaming.jar \
-input /user/hadoop/tfidf/input \
-output /user/hadoop/tfidf/output1 \
-file /home/hadoop/mapper1.py \
-mapper /home/hadoop/mapper1.py \
-file /home/hadoop/reducer1.py \
-reducer /home/hadoop/reducer1.py
hadoop jar /usr/lib/hadoop-mapreduce/hadoop-streaming.jar \
-input /user/hadoop/tfidf/output1 \
-output /user/hadoop/tfidf/output2 \
-file /home/hadoop/mapper2.py \
-mapper /home/hadoop/mapper2.py \
-file /home/hadoop/reducer2.py \
-reducer /home/hadoop/reducer2.py
hadoop jar /usr/lib/hadoop-mapreduce/hadoop-streaming.jar \
-input /user/hadoop/tfidf/output2 \
-output /user/hadoop/tfidf/output \
-file /home/hadoop/mapper3.py \
-mapper /home/hadoop/mapper3.py \
-file /home/hadoop/reducer3.py \
-reducer /home/hadoop/reducer3.py
```

3.3 clear.sh

Code:

```
#!/usr/bin/env bash
hdfs dfs -rm -r /user/hadoop/tfidf/input
hdfs dfs -rm -r /user/hadoop/tfidf/output
hdfs dfs -rm -r /user/hadoop/tfidf/output1
hdfs dfs -rm -r /user/hadoop/tfidf/output2
hdfs dfs -mkdir /user/hadoop/tfidf/input
```