Big Data

Implémention distribuée de l'algorithme TF-IDF sur Hadoop Streaming et Spark.

DJILANI Amira et OUAZZANI CHAHDI Nizar

Table des matières

1 Présentation de l'algorithme TF-IDF

1.1 Définition

Le TF-IDF (term frequency-inverse document frequency) est une méthode de pondération utilisée en recherche d'information et en particulier dans la fouille de textes. Cette mesure statistique permet d'évaluer l'importance d'un terme contenu dans un document, relativement à une collection ou un corpus. Le poids augmente proportionnellement au nombre d'occurrences du mot dans le document. Il varie également en fonction de la fréquence du mot dans le corpus. Des variantes de la formule originale sont souvent utilisées dans des moteurs de recherche pour apprécier la pertinence d'un document en fonction des critères de recherche de l'utilisateur.

1.2 Formule

Terme Frequency du terme$_j$ dans le document$_i$ est donnée par le ratio occurrence de j dans i sur total des termes dans i :
$\mathrm{TF}_{i,j} = \frac{n_{i,j}}{\sum_i n_{i,j}}$

Inverse Docuemnt Frequency du terme$_j$ dans le document$_i$ est donnée par :
$\mathrm{IDF}_i = \log \frac{|D|}{|\{d_j : t_i \in d_j\}|}$

où :

— |D| : nombre total de documents dans le corpus

— $|\{d_j : t_i \in \mathrm{d}_j\}|$: nombre de documents où le terme t_i apparaît (c'est-à-dire $n_{i,j} \neq 0$).

1.3 Exemple numérique

Pour illustrer ces propos nous utilisons les deux documents suivants :

— Document 1 : Les données des utilisateurs sont importantes.

— Document 2 : Les données utilisateurs doivent restées confidentielles.

Par exemple pour le mot "confidentielles" le calcul du score TF-IDF se fait comme suit :
$\mathrm{TF}_{doc2,confidentielles} = \frac{1}{6}$
$\mathrm{IDF}_{confidentielles} = \log \frac{2}{1}$

```
["confidentielles","1"] 0.0833333333
["des","1"]     0.0833333333
["doivent","1"] 0.0833333333
["donn\u00e9es","1"]    0.1666666667
["importantes","1"]     0.0833333333
["les","1"]     0.1666666667
["rest\u00e9es","1"]    0.0833333333
["sont","1"]    0.0833333333
["utilisateurs","1"]    0.1666666667
```

Figure 1 – Résultat des scores TF-IDF pour chacun des termes.

2 Description de la solution adoptée

Requirement :

— Detailed description of the adopted solution.

— Designed algorithms plus related comments/description to main fragments of code.

Dans cette section nous allons présenter les deux solutions que nous avons choisi pour l'implémentation de la méthode de calcul du score TF-IDF.

2.1 En utilisant Hadoop Streaming via MRJob Python

Afin d'implémenter le Streaming Hadoop, nous avons utilisé la librairie Python MRJob [1]. Cette librairie permet d'écrire plus simplement des jobs MapReduce : On commence par définir des fonctions Map et Reduce qui seront injectées dans un MRJob prenant en charge le lancement de jobs en utilisant les entrées et sorties en ligne de commande. On peut également utiliser un chaînage de plusieurs étapes MapReduce, c'est ce que nous avons utilisé dans notre implémentation.

2.1.1 Structure des documents utilisés

Le fichier d'entrée du streaming est un fichier JSON sous le format suivant :

```
[{"docId": "1",
 "content": "contenu du corpus 1"},
{"docId": "2",
 "content": "contenu du corpus 2"}]
```

Il contient plusieurs documents, chaque document est défini par son "docId" et comporte un corpus textuel dans son "content".

2.1.2 Algorithme

L'algorithme consiste en 3 étapes MapReduce et une dernière étape Map. ‘

— Etape 1 : Le 1er Mapper prend en paramètre le document JSON et en sortie on récupère la paire clé-valeur suivante : (terme, docId, nbDocs), 1 pour chaque terme dans les différents documents. Le 1er Reducer de cette étape a pour objectif de sommer les occurrences de chaque mot dans chacun des documents (on aura calculer le $n_{i,j}$ de la formule du $\mathrm{TF}_{i,j}$.
La sortie sera de la forme `(terme, docId, nbDocs), sum(occurence par doc)`

— Etape 2 : Le 2ème Mapper va réarranger la sortie du 1er Reducer sous le format suivant : `(docId, [terme, occurence par doc, nbDocs])`.
Le 2ème Reducer permettra de calculer le nombre total de mots dans chacun des termes, la sortie est de la forme suivante :
`([terme, docId, nbDocs], [occ_terme_docId, nb_tot_termes_docId])`

— Etape 3 : Le 3ème Mapper permettra de réarranger la sortie précedante sous le format suivant : `(terme, [docId, occurence_docId, nb_tot_termes_docId, nbDocs, 1])` Le 3ème Reducer calculera la composante manquante au score : le nombre de documents où un terme appraît, sa sortie sera comme suit : `([terme,docId,nbDocs],[occ_terme_docId,nb_tot_termes_docId,nb_docs_terme_app])`.

— Etape 4 : Le 4ème Mapper calculera le score final TF-IDF.

En annexe, nous avons rajouté le code de cette partie avec les commentaires explicatifs.

2.2 En utilisant PySpark

N'ayant pas pu installer la distribution d'Hadoop sur nos machines locales (OS Windows 10), nous avons eu recours à la plateforme Google Colab (un service Cloud de Google basé sur Jupyter Notebook supportant Python et mis a disposition gratuitement pour développer des modèles de Machine Learning). On peut y installer notre environnement Spark et démarrer des sessions PySpark en toute simplicité.

2.2.1 Algorithme

Notre algorithme effectue le facul du score en 6 étapes.

— On calcule le nombre total de documents.

— On compte le nombre de termes contenu dans chacun des documents.

— On extrait les termes de chacun des documents en les formattant.

— On calcule le nombre de termes dans chaque document.

— On compte le nombre de documents où un terme apparaît.

— On utilise une fonction définie pour calculer le score TF-IDF en utilisant les différentes étapes précedentes.

3 Analyse expérimentale

3.1 Données utilisées pour l'expérience

Nous avons choisi d'utiliser plusieurs dataset pour illuster la performance des deux algorithmes.

— 1er dataset : une centaine de commentaires sur des vidéos youtube.

— 2ème dataset : une centaine d'annonces sur leboncoin.

— 3ème dataset : une centaine d'articles de presse trouvés BBC.

— 4ème dataset : une centaine d'allocutions officielles d'hommes politiques.

— 5ème dataset : Collection des oeuvres de Shakespeare.

3.2 Résultats de l'expérience

Typologie des documents	**Nombre de corpus**	**MRJob Streaming**	**PySpark**
Commentaire youtube	40 (moyenne 15 mots)	2.72sc	1.70sc
Annonces	40 (moyenne 70 mots)	2.95sc	1.60sc
Articles de presse	40 (moyenne 460 mots)	2.85sc	1.55sc
Allocutions	40 (moyenne 1500 mots)	4.95sc	1.65sc
Commentaire youtube	400 (moyenne 15 mots)	3.52sc	8.63sc
Annonces	400 (moyenne 70 mots)	2.40sc	13.09sc
Articles de presse	400 (moyenne 500 mots)	14.65sc	16.22sc
Allocutions	400 (moyenne 1500 mots)	30.95sc	17.48sc
Travaux de Shakespeare	2 (904100 mots environ)	39.48sc	20.14sc

TABLE 1 – Résultats des tests.

3.3 Commentaires

Pour notre expérience, nous avons considéré 5 typologies de documents avec des tailles croissantes et avec deux nombres de documents (40 et 400) afin de pouvoir comprendre le comportement des algorithmes développés dans les différents cas de figures.

Nous remarquons d'abord que : D'une manière globale, l'algorithme développé sur PySpark sur-performe celui de Hadoop Streaming. En particulier, PySpark réalisent de bons résultats quand il s'agit de documents de petites tailles. Nous estimons que cette différence est dûe à plusieurs facteurs :

— D'abord, au fait que la libraire MRJob fait appel à des services de Google pour exécuter les jobs MapReduce. Cette communication supplémentaire entraîne inévitablement un décalage temporaire.

— En suite, le fait que PySpark utilise de la mémoire vive pour effectuer les calculs sur les fichiers des corpus accélère le processus.

— Finalement, le fait de ne pas avoir utiliser des Combiner intermédiaires dans nos différentes étapes MapReduce Hadoop Streaming peut réduire la performance de calcul.

Par ailleurs, des irrégularités peuvent être remarquées par rapport à cette tendance globale, c'est le cas pour le 5ème et 6ème tests où Hadoop Streaming sur-performe largement PySpark. Nous pensons que l'origine de cet écart provient de la disponibilité du service de Google Colab en matière de force de calcul de de mémoire RAM.

4 Annexe

4.1 Code MapReduce Streaming

```
from mrjob.job import MRJob
from mrjob.protocol import JSONValueProtocol
from mrjob.step import MRStep
from math import log
import re
import json
import time


regex = re.compile(r"[\w']+")

class MapReduceTFIDF(MRJob):
  # On pr cise qu'il s'agit du protocole JSON    utiliser dans la lecture
  INPUT_PROTOCOL = JSONValueProtocol
  # Sortie => (terme, docId, nbDocs), 1
  def mapper1(self, _, json_doc):
    #Longueur du document JSON ie nombre de documents/corpus
    D = len(json_doc)

    for i in range(D):
      corpus = json_doc[i]

      for terme in regex.findall(corpus['content']):
        yield (terme.lower(),corpus['docId'],D), 1

    # Sortie : (terme, docId, nbDocs), sum(occurence par doc)
  def reducer1(self, termeInfos,occurence):
    yield (termeInfos[0],termeInfos[1], termeInfos[2]), sum(occurence)

  # Sortie : (docId, [terme, occurence par doc, nbDocs])
  def mapper2(self, docInfo, n):
    yield docInfo[1], (docInfo[0],n, docInfo[2])
  # Sortie : ([terme, docId, nbDocs], [occurence terme dans docId, nombre total
  def reducer2(self,docId,termeInfo):
    total=0
    n=[]
    terme=[]
    D=[]
    for valeur in termeInfo:
      total+=valeur[1]
      n.append(valeur[1])
      terme.append(valeur[0])
      D.append(valeur[2])
```

```
44      N=[total]*len(terme)
45
46      for valeur in range(len(terme)):
47        yield (terme[valeur],docId,D[valeur]),(n[valeur],N[valeur])
48
49    # Sortie : (terme, [docId, occurence dans doc, nb total termes dans doc, nbDoc
50    def mapper3(self, termeInfo, compteTermes):
51      yield termeInfo[0], (termeInfo[1], compteTermes[0], compteTermes[1],termeInf
52
53    # Sortie : ([terme docId, nbDocs], [occurence dans doc, nb total termes dans d
54    def reducer3(self, terme, wordInfoComptes):
55      total = 0
56      docId = []
57      n = []
58      N = []
59      D = []
60      for valeur in wordInfoComptes:
61        total += 1
62        # id du document j
63        docId.append(valeur[0])
64        # nombre d'occurence du terme dans le document j
65        n.append(valeur[1])
66        # nombre total de termes dans le document j
67        N.append(valeur[2])
68        # nombre total de documents dans le corpus
69        D.append(valeur[3])
70          # nombre de documents o   le terme ti appara t
71      m = [total]* len(n)
72      for valeur in range(len(m)):
73        yield (terme, docId[valeur], D[valeur]), (n[valeur], N[valeur], m[valeur])
74
75    def mapper4(self,termeInfo,termeMetrics):
76      tfidf=(termeMetrics[0]/termeMetrics[1])*log(termeInfo[2]/termeMetrics[2], 2)
77      yield (termeInfo[0],termeInfo[1]),tfidf
78
79    def steps(self):
80      return [
81        MRStep(mapper=self.mapper1,
82          reducer=self.reducer1),
83        MRStep(mapper=self.mapper2,
84          reducer=self.reducer2),
85        MRStep(mapper=self.mapper3,
86          reducer=self.reducer3),
87        MRStep(mapper=self.mapper4)]
88
89  if __name__ == '__main__':
90    start_time = time.time()
91    MapReduceTFIDF.run()
```

```
92   print("--- %s seconds ---" % (time.time() - start_time))
```

4.2 Code PySpark

```
1 from pyspark import SparkContext
2 import re
3 import numpy as np
4 from __future__ import division
5 import time
6
7
8 sc = SparkContext.getOrCreate()
9
10 start_time = time.time()
11 #On lit tous les fichiers dans le r pertoire courant
12 docs = sc.wholeTextFiles("*.txt").map(lambda t: (t[0].replace('file:/content/','
13 count_terms = sc.wholeTextFiles("*.txt").map(lambda t: (t[0].replace('file:/cont
14
15 #On compte le nombre de documents
16 count_words = count_terms.collect()
17
18 #On tokenize le document d'entr e : on transforme le document en un ensemble de
19 tokenized_docs = docs.map(lambda t : (t[0], re.split("\\W+", t[1].lower()) ) )
20
21 #On compte le nombre de termes dans chaque document
22 term_frequency = tokenized_docs.flatMapValues(lambda x: x).countByValue()
23
24 #On compte le nombre de documents o   un terme appara t
25 document_frequency = tokenized_docs.flatMapValues(lambda x: x).distinct()\
26                         .filter(lambda x: x[1] != '')\
27                         .map(lambda t: (t[1],t[0])).countByKey()
28
29 #Fonction de calcul du score
30 def tf_idf(N, ct, tf, df):
31     result = []
32
33     for key, value in tf.items():
34         #id du document
35         doc = key[0]
36         #le terme pour lequel on calcule le score
37         term = key[1]
38         #pour le document en question, on r cup re le nombre total de termes
39         count = ct[ct[0]==doc][1]
40         #pour le terme en quesiton, on r cup re le nombre de documents
41         #o   il apppara t
42         df = document_frequency[term]
43         if (df>0):
44           tf_idf = float(value/count)*np.log(number_of_docs/df)
```

```
45
46          result.append({"doc":doc, "term":term, "score":tf_idf})
47      return result
48 tf_idf_output = tf_idf(number_of_docs,count_words, term_frequency, document_freq
49
50 print("--- %s seconds ---" % (time.time() - start_time))
```

4.3 Script pour la génération de fichiers .txt et .json

```
1 import json
2 import os
3 import re
4 import json
5
6 #Cr ation du fichier JSON    partir des fichiers textes
7 WORD_RE = re.compile(r"[\w']+")
8 directory = 'C:/Users/bbbbb/Desktop/bigdat'
9
10 data = []
11 words_doc=[]
12 for filename in os.listdir(directory):
13     if filename.endswith(".txt"):
14         file = open(directory+'/'+filename)
15         with open(directory+'/'+filename, 'r') as f:
16             num_words=0
17             for line in f:
18                 words = line.split()
19                 num_words += len(words)
20         words_doc.append(num_words)
21         data.append({
22         'docId': filename.replace(directory,''),
23         'content': " ".join(WORD_RE.findall(file.read()))
24         })
25         continue
26     else:
27         continue
28
29 print(float(sum(words_doc)/len(words_doc)))
30 with open('data1.json', 'w') as outfile:
31     json.dump(data, outfile)
32
33 #Cr ation des fichiers textes    partir de la base de donn es Excels
34 import xlrd
35
36 loc = ("C:/Users/bbbbb/Desktop/bigdat/index.xlsx")
37
38 wb = xlrd.open_workbook(loc)
39 sheet = wb.sheet_by_index(1)
```

```
40
41 # For row 0 and column 0
42 sheet.cell_value(0, 0)
43
44 print(sheet.ncols)
45
46 for j in range(400):
47     try:
48         f = open("C:/Users/bbbbb/Desktop/bigdat/"+str(j)+'.txt','w')
49         f.write(sheet.cell_value(j,7))
50         f.close()
51         pass
52     except Exception as e:
53         continue
```

Références

[1] Documentation MRJob Python.
`https://pythonhosted.org/mrjob/guides/quickstart.html`

[2] Calcul de TF-IDF en utilisant le paradigme MapReduce.
`https://github.com/guillaume6pl/mr`$_t fidf$

[3] Fichier du jeu de données "Vélib : Disponibilité en temps réel" utilisé.
`https://dzone.com/articles/calculating-tf-idf-with-apache-spark`