section.5
section.5
Hajar BADRAOUI
Université Caen Normandie, France
M2 master Statistiques appliquées et analyse décisionnelle (SAAD)

• Compte rendu •

Réalisation d'un moteur de recherche

Abstract L'étude a pour but de réaliser un moteur de recherche basé sur l'utilisation de Spark, implémentant un modèle TF-IDF sur un corpus de documents. On décrit les manipulations effectuées, les commandes saisies, et les résultats obtenus pour chaque étape.

Introduction

Dans un monde où la quantité de données ne cesse de croître, l'extraction rapide et précise des informations est devenue difficile. Ce projet vise à réaliser un moteur de recherche basé sur Spark, utilisant le modèle TF-IDF pour identifier efficacement les documents pertinents dans un corpus.

Le modèle TF-IDF repose sur trois concepts fondamentaux pour évaluer l'importance d'un mot dans un document :

1. **Calcul du Term Frequency (TF)**
   La fréquence d'apparition d'un mot dans un document, exprimée comme suit :

$$TF(\text{mot, document}) = \text{nombre d'occurrences du mot dans le document}$$

2. **Calcul du Document Frequency (DF)**
   Le nombre de documents contenant le mot est défini comme suit :

$$DF(\text{mot}) = \text{nombre de documents où le mot apparaît}$$

3. **Application de la formule TF-IDF**
   Enfin, pour évaluer l'importance d'un mot dans un document par rapport à l'ensemble du corpus, on applique la formule suivante :

$$TF\text{-}IDF(\text{mot, document}) = TF(\text{mot, document}) \times \log\left(\frac{N}{DF(\text{mot})}\right)$$

   où N représente le nombre total de documents dans le corpus.

Email: hajar.badraoui@etu.unicaen.fr

1 Configuration de l'environnement et commandes initiales

Pour activer l'environnement virtuel Python pour travailler avec Spark , on utilise la commande suivante :

```
source /usr/bin/tp_python_virtualenv
```

Pour lancer l'interface PySpark :

```
pyspark
```

1.1 Chargement du corpus

Tout d'abord, on crée un fichier texte contenant le corpus de documents où chaque ligne représente un document comme suit :

```
cat > corpus.txt
il fait beau et chaud
il fait chaud et beau
chaud chaud chaud macao
chaud chaud chaud chocolat
```

On charge le fichier texte (corpus.txt) dans un RDD (corpus), c'est une structure de données distribuée de Spark, après on récupère toutes les données contenues dans le RDD et les ramène sous forme d'une liste Python (collect).

```
corpus = sc.textFile('corpus.txt')
corpus.collect()
```

Résultat :

```
['il fait beau et chaud', 'il fait chaud et beau',
 'chaud chaud chaud macao', 'chaud chaud chaud chocolat']
```

1.2 Transformation du corpus en listes des mots

On transforme chaque document en une liste de mots en appliquant la fonction split(' ') à chaque ligne.

```
documents = corpus.map(lambda line: line.split(' '))
```

Pour le code qui existe à l'intérieur des parenthèses (), si, comme on a défini une fonction qui divise chaque chaîne de caractères (ligne du corpus) en plusieurs mots, en utilisant l'espace (' ') comme séparateur dans Python comme suit :

def function(line) :
return line.splite(" ")

Pour bien comprendre ce passage de transformation , on prend l'**exemple** suivant :
exemple d'entrée (corpus.collect() avant transformation)
Résultat :

```
['il fait beau et chaud',
 'il fait chaud et beau',
 'chaud chaud chaud macao',
 'chaud chaud chaud chocolat']
```

Sortie (documents.collect() après transformation) :

```
documents.collect()
```

Résultat :

```
[['il', 'fait', 'beau', 'et', 'chaud'],
 ['il', 'fait', 'chaud', 'et', 'beau'],
 ['chaud', 'chaud', 'chaud', 'macao'],
 ['chaud', 'chaud', 'chaud', 'chocolat']]
```

On voit clairement que chaque ligne du corpus est transformée en une liste distincte de mots, ce qui facilite les manipulations suivantes : calcul des TF, DF et l'indexation.

Si on veut récupérer le premier élément du RDD, on compile la commande suivante :

```
documents.take(1)
```

Résultat :

```
[['il', 'fait', 'beau', 'et', 'chaud']]
```

1.3 Numérotation des documents

```
documents = corpus.zipWithIndex().map(lambda x: (x[1], x[0]))
```

La méthode Spark zipWithIndex() assigne un index unique à chaque élément de l'RDD. Par **exemple** : notre corpus contient les lignes suivantes

```
['il fait beau et chaud', 'il fait chaud et beau',
 'chaud chaud chaud macao', 'chaud chaud chaud chocolat']
```

Après zipWithIndex(), le résultat sera :

```
[('il fait beau et chaud', 0),
 ('il fait chaud et beau', 1),
 ('chaud chaud chaud macao', 2),
 ('chaud chaud chaud chocolat', 3)]
```

Chaque ligne du corpus est associée à un numéro unique.

```
map(lambda x: (x[1], x[0]))
```

On utilise une fonction lambda pour inverser l'ordre des paires (Valeur, Key) générées par zipWithIndex(), on trouve alors

```
[(0, 'il fait beau et chaud'),
 (1, 'il fait chaud et beau'),
 (2, 'chaud chaud chaud macao'),
 (3, 'chaud chaud chaud chocolat')]
```

Alors on a réorganisé l'index en passant du format (Valeur, Key) au format (Key, Valeur).

1.4 Compter les documents

Pour compter le nombre total de documents dans le corpus, on utilise count() :

```
nbOfDocuments = documents.count()
```

Résultat :

```
4
```

Pour compter le nombre de documents sur une valeur flottante :

```
nbOfDocuments = float(documents.count())
```

Résultat :

```
4.0
```

1.5 Extraction de la liste des mots

Pour trouver toute la liste des mots, on utilise documents.flatMap() et si on veut supprimer les doublons, on fait set(), alors la commande complète est

```
documents.flatMap(lambda x: set(x[1].split())).collect()
```

On commence par le code qui se trouve à l'intérieur de (), la fonction lambda x : set(x[1].split()) effectue les opérations suivantes :
- x[1] : extrait le texte brut.
- .split() : divise ce texte en une liste de mots.
- set() : supprime les doublons dans cette liste.

Par **exemple** pour la ligne "chaud chaud chaud chocolat", cela donne :

```
{"chaud", "chocolat"}
```

Par **exemple** si on prend : "il", "fait", "beau", "et", "chaud" "il", "fait", "chaud", "et", "beau"
flatMap combine les ensembles de mots ligne par ligne pour obtenir une séquence unique comme suit :

```
["il", "fait", "beau", "et", "chaud", "il", "fait", "chaud", "et", "beau"]
```

Pour la méthode collect, elle retourne les résultats sous forme d'une liste Python.

```
["il", "fait", "beau", "et", "chaud", "il", "fait", "chaud","et", "beau"]
```

2 Calcul du Term Frequency- TF

On compte le nombre d'occurrences (term frequency TF) de chaque terme dans le document, c'est-à-dire on associe à chaque mot (clé) la valeur 1, on groupe par clés et on récupère la longueur.

```
tfs= sc.parallelize(document.split(" ")).map(lambda word: (word, 1)).groupByKey().
    ↪ mapValues(len)
```

Par exemple, pour le document "il fait beau et chaud", le TF de chaque mot est
Résultat :

```
[('il', 1), ('fait', 1), ('beau', 1), ('et', 1), ('chaud', 1)]
```

3 Calcul du Document Frequency - DF

Le calcul les document frequency (DF) sert à indiquer pour chaque terme, le nombre de documents qui le contiennent.

```
dfs = corpus.map(lambda line: line.split(' '))
            .map(lambda x: set(x))
            .flatMap(lambda x: x)
            .map(lambda word: (word, 1))
            .groupByKey()
            .map(lambda x: (x[0], len(x[1])))
```

On détaille ce code sous forme les étapes suivantes :

3.1 Étape 1 : Division chaque ligne en une liste de mots

```
step1 = corpus.map(lambda line: line.split(' '))
```

Résultat :

```
[['il', 'fait', 'beau', 'et', 'chaud'],
 ['il', 'fait', 'chaud', 'et', 'beau'],
 ['chaud', 'chaud', 'chaud', 'macao'],
 ['chaud', 'chaud', 'chaud', 'chocolat']]
```

3.2 Étape 2 : Suppression des doublons dans chaque document

Rappelons que pour transformer chaque document (liste de termes) en un ensemble, ce qui élimine les doublons, on utilise la fonction set().

```
step2 = step1.map(lambda x: set(x))
```

Résultat :

```
[{'il', 'fait', 'beau', 'et', 'chaud'},
 {'il', 'fait', 'chaud', 'et', 'beau'},
 {'chaud', 'macao'},
 {'chaud', 'chocolat'}]
```

3.3 Étape 3 : Aplatissement des termes

On aplatit la structure pour obtenir une liste unique contenant tous les termes des documents à l'aide de la transformation flatMap().

```
step3 = step2.flatMap(lambda x: x)
```

Résultat :

```
['il', 'fait', 'beau', 'et', 'chaud',
 'il', 'fait', 'chaud', 'et', 'beau',
 'chaud', 'macao',
 'chaud', 'chocolat']
```

3.4 Étape 4 : Association des termes avec une valeur (Key, Valeur)

Chaque terme est considéré comme une clé, et on lui associe la valeur 1.

```
step4 = step3.map(lambda word: (word, 1))
```

Résultat :

```
[('il', 1), ('fait', 1), ('beau', 1), ('et', 1), ('chaud', 1),
 ('il', 1), ('fait', 1), ('chaud', 1), ('et', 1), ('beau', 1),
  ('chaud', 1), ('macao', 1),
 ('chaud', 1), ('chocolat', 1)]
```

3.5 Étape 5 : Regroupement par clé

On utilise groupByKey() pour grouper selon les clés et fusionner les valeurs.

```
step5 = step4.groupByKey()
```

Résultat :

```
[('fait', <pyspark.resultiterable.ResultIterable object at 0x7fe01b48ebc0>),
 ('macao', <pyspark.resultiterable.ResultIterable object at 0x7fe01b48fe50>),
  ('il', <pyspark.resultiterable.ResultIterable object at 0x7fe01b48dde0>),
   ('beau', <pyspark.resultiterable.ResultIterable object at 0x7fe01b48fd00>),
  ('et', <pyspark.resultiterable.ResultIterable object at 0x7fe01b48e200>),
  ('chaud', <pyspark.resultiterable.ResultIterable object at 0x7fe01b48e8f0>),
  ('chocolat', <pyspark.resultiterable.ResultIterable object at 0x7fe01b48ec80>)]
```

3.6 Étape 6 : Calcul de la fréquence

On mappe une lambda qui calcule la longueur len(...) de l'itérable.

```
step6 = step5.map(lambda x: (x[0], len(x[1])))
```

```
[('fait', 2),
 ('macao', 1),
 ('il', 2),
 ('beau', 2),
 ('et', 2),
 ('chaud', 4),
 ('chocolat', 1)]
```

4 Calcul des TF-iDF

4.1 Étape 1 : Extraction d'un document

Pour commencer, on travaille sur un document spécifique :

```
step1 = documents.map(lambda x: x[1]).collect()[3]
```

Résultat :

```
"chaud chaud chaud chocolat"
```

4.2 Étape 2 : Transformation en RDD

On transforme le document sélectionné en un RDD :

```
step2 = sc.parallelize(step1.split())
```

Résultat :

```
['chaud', 'chaud', 'chaud', 'chocolat']
```

4.3 Étape 3 : Calcul du Term Frequency (TF)

```
step3 = step2.map(lambda word: (word, 1)) \
             .groupByKey() \
             .map(lambda x: (x[0], len(x[1])))
```

Résultat :

```
[('chaud', 3), ('chocolat', 1)]
```

4.4 Étape 4 : Jointure avec le Document Frequency (DF)

On effectue une jointure entre les TF calculés et les DF pour obtenir une association $(mot, (tf, df))$:

```
step4 = step3.join(dfs)
```

Résultat :

```
[('chaud', (3, 4)), ('chocolat', (1, 1))]
```

4.5 Étape 5 : Calcul des scores TF-IDF

Enfin, on applique la formule TF-IDF pour chaque mot :

$$TF\text{-}IDF(mot, document) = TF(mot, document) \times \log\left(\frac{N}{DF(mot)}\right)$$

où N est le nombre total de documents dans le corpus. Le calcul est effectué comme suit :

```
import math
step5 = step4.mapValues(lambda x: x[0] * math.log(nbOfDocuments / x[1]))
```

Résultat :

```
[('chaud', 0.0), ('chocolat', 1.3862943611198906)]
```

4.6 Calcul des TF-IDF pour tout le corpus

On applique ces étapes à chaque document du corpus pour calculer les scores TF-IDF globaux :

```
tfidfs = [
    sc.parallelize(x.split())
      .map(lambda word: (word, 1))
      .groupByKey()
      .map(lambda x: (x[0], len(x[1])))
      .join(dfs)
      .mapValues(lambda x: x[0] * math.log(nbOfDocuments / x[1]))
    for x in documents.map(lambda x: x[1]).collect()
]
```

Structure finale :

```
[PythonRDD[290] at RDD at PythonRDD.scala:53,
 PythonRDD[291] at RDD at PythonRDD.scala:53,
 PythonRDD[292] at RDD at PythonRDD.scala:53,
 PythonRDD[293] at RDD at PythonRDD.scala:53]
```

Pour afficher les TF-IDF d'un document spécifique, par exemple le premier document :

```
tfidfs[0].collect()
```

Résultat :

```
[('il', 0.6931471805599453), ('beau', 0.6931471805599453),
 ('chaud', 0.0), ('fait', 0.6931471805599453), ('et', 0.6931471805599453)]
```

5 Requêtes

Pour répondre à une requête, on a besoin d'un index qui associe chaque terme du corpus à l'ensemble des identifiants des documents dans lesquels il apparaît.

5.1 Étape 1 : Création de l'index

L'index est construit en associant chaque mot à son identifiant de document, puis en regroupant les identifiants par mot.

```
index = documents.flatMap(lambda x: [(word, x[0]) for word in x[1]])
              .groupByKey()
              .map(lambda x: (x[0], set(x[1])))
```

Résultat :

```
[('fait', {0, 1}),
 ('macao', {2}),
 ('il', {0, 1}),
 ('beau', {0, 1}),
 ('et', {0, 1}),
 ('chaud', {0, 1, 2, 3}),
 ('chocolat', {3})]
```

5.2 Étape 2 : Définir une requête

La requête est définie comme une liste de mots à rechercher dans le corpus.

```
query = [u'il', u'chaud']
```

5.3 Étape 3 : Insérer la requête dans Spark

On transforme la requête en un RDD sous la forme (Key,Valeur).

```
query_rdd = sc.parallelize(query).map(lambda x: (x, {}))
```

Résultat :

```
[('il', {}), ('chaud', {})]
```

5.4 Étape 4 : Joindre l'index à la requête

On effectue une jointure entre les mots de la requête et l'index pour récupérer les identifiants des documents correspondants.

```
joined = query_rdd.join(index).mapValues(lambda x: x[1])
```

Résultat :

```
[('il', {0, 1}), ('chaud', {0, 1, 2, 3})]
```

5.5 Étape 5 : Calcul de l'intersection des documents

Pour identifier les documents contenant tous les mots de la requête, on réduit la collection avec une fonction lambda qui calcule l'intersection des ensembles.

```
found = joined.reduce(lambda x, y: x[1].intersection(y[1]))
```

Résultat :

```
{0, 1}
```

5.6 Étape 6 : Calculer les coordonnées du vecteur de la requête

On associe à chaque mot de la requête son **Document Frequency (DF)** pour obtenir les coordonnées du vecteur de la requête.

```
query_vect = sc.parallelize(query).map(lambda x: (x, 0))
               .join(dfs)
               .mapValues(lambda x: math.log(nbOfDocuments / x[1]))
```

Résultat :

```
[('il', 0.6931471805599453), ('chaud', 0.0)]
```

5.7 Étape 7 : Calcul du produit scalaire et de la similarité cosinus

Il est nécessaire de déterminer, parmi les documents qui contiennent la requête, ceux qui ont la meilleure similarité avec la requête. La requête va donc être considérée comme un document, à comparer aux autres. Un document est représenté par le vecteur des TF-iDF des mots de la requête. La similarité entre deux vecteurs est donnée par le cosinus 1 de l'angle entre les deux vecteurs, qui fournit une valeur entre -1 et 1.

```
def cosinus(tuple):
    return tuple[0] / math.sqrt(tuple[1] * tuple[2])
```

```
for found in sc.parallelize(query).map(lambda x: (x, {})).join(index).mapValues(lambda
    ↪  x: x[1]).reduce(lambda x,y: x[1].intersection(y[1])): print(found, cosinus(
    ↪ tfidfs[found].join(query_vect).map(lambda x: (x[1][0] * x[1][1],  x[1][0] * x
    ↪ [1][0], x[1][1] * x[1][1])).reduce(lambda x,y: (x[0] + y[0], x[1] + y[1], x[2]
    ↪ + y[2])))))
```

Résultat final

On trouve les résultats suivants :

```
0 0.4804530139182014
1 0.4804530139182014
```

On sait que les scores de similarité cosinus indiquent la pertinence des documents par rapport à la requête. On voit clairement que les documents **0** et **1** ont une similarité de `0.4804530139182014`, cela indique que les documents contiennent une proportion significative des termes de la requête.

6 Test sur un gros corpus

Maintenant, nous travaillons avec un corpus plus volumineux et utilisons le fichier 00000000.txt.gz. Le code reste inchangé, mais nous allons simplement charger ce nouveau corpus.

```
corpus = sc.textFile('00000000.txt')
```

6.1 Vérification du contenu

Pour vérifier le contenu du corpus, nous utilisons la commande :

```
corpus.collect()
```

Résultat :

```
['r gun and rashad brown <br> by the 1660s the status of african servants had hardened into permanent
servitude based on skin color <br> now there were slaves in virginia and labor starved tobacco
planters wanted more <br> the plot exposed during this decade was conspiracy of slaves and
indentured servants who planned to destroy their masters and afterwards to set up for themselves....']
```

```
nbOfDocuments = float(documents.count())
```

Résultat :

```
990.0
```

6.2 Calcul du Document Frequency (DF)

Résultat :

```
[('needless', 3),('stubbornness', 1), ('sensibility', 4),
 ('30pm', 7), ('merton', 2), ('roses', 6), ('memorized', 3),
 ('thoroughly', 16), ...]
```

6.3 Calcul des TF-IDF

```
tfidfs[0].take(5)
```

Résultat :

```
[('of', 2.354513009278734), ('an', 0.6011184824839684), ('stuck', 3.6788291182604347),
('two', 1.7292374426596682), ('knees', 3.6788291182604347)]
```

6.4 Création de l'index

```
index = documents.flatMap(lambda x: [(word, x[0]) for word in x[1]]) \
              .groupByKey() \
              .map(lambda x : (x[0], set(x[1])))
index.collect()
```

Résultat :

```
[('vehicular', {771, 395, 415}),  ('tallies', {395}), ('crri', {395}),
 ('uneducated', {395, 927}), ('ibanez', {395}), ...]
```

6.5 Vecteur de la requête

```
query = [u'china', u'love']
query_vect = sc.parallelize(query)
              .map(lambda x: (x, 0))
              .join(dfs)
              .mapValues(lambda x: math.log(nbOfdocuments / x[1]))
query_vect.collect()
```

Résultat :

```
[('china', 2.7233176732329984), ('love', 1.927891643552635)]
```

6.6 Calcul du produit scalaire et de la similarité cosinus

Pour chaque document contenant les mots de la requête, on a calculé les scores de la similarité cosinus :

```
1 0.8166010248642187
644 1.0
135 1.0
136 0.9428091108177831
782 0.8395165656288697
272 0.777929315649231
152 1.0
416 1.0
928 0.777929315649231
930 0.9428091108177831
675 1.0
938 0.9043869167393386
50 0.9622012792496836
690 0.777929315649231
947 0.9771125001645176
567 1.0
580 1.0
453 1.0
842 0.9622012792496836
207 0.9428091108177831
```

```
489 1.0
620 1.0
634 0.9428091108177831
```

Les documents qui ont le score 1 : c'est la meilleure similarité.

Conclusion

En conclusion, ce travail a permis de répondre à la question suivante : Comment retrouver rapidement les documents les plus pertinents dans un large corpus ? Ce travail explore l'utilisation de Spark et du modèle TF-IDF pour construire un moteur de recherche performant et efficace.