Algoritmo MinHash em PySpark
Projeto Apresentado à Disciplina de Inteligência na Web e Big Data

Andréia Cristina dos Santos Gusmão[1]

[1]Centro de Matemática, Computação e Cognição – CMCC
Universidade Federal do ABC (UFABC) – Santo André – SP – Brasil

andreia.gusmao@ufabc.edu.br

Resumo. *Com a abundância dos dados, é cada vez mais necessário tratar da grande massa de dados com agilidade e precisão. Muitas vezes, os dados não se encontram padronizados e é preciso uma normalização dos dados, antes de aplicar alguma métrica. Existem várias métricas de cálculo de similaridade. Esse projeto tem como finalidade explicar o algoritmo de* minhash *e a similaridade de Jaccard com o* minhash*, utilizando PySpark.*

1. Introdução

Uma forma de verificar a similaridade entre conjuntos de textos é através do índice de Jaccard, que é calculada pela razão dos tamanhos da interseção dos conjuntos e da união:

$$J(d1, d2) = \frac{|d1 \cap d2|}{|d1 \cup d2|}. \tag{1}$$

Porém, se o número de termos da união dos dois documentos for muito grande e sendo necessário avaliar muitos pares de documentos, essa medida de similaridade se torna computacionalmente custosa.

Uma forma de simplificar esse processo, de forma aproximada, é utilizando funções *hash* lineares para gerar uma permutação. Dado que os termos possam ser representados de forma numérica, ao aplicar para cada elemento de um documento a seguinte função:

$$f(x) = a \cdot x + b \cdot \mod P, \tag{2}$$

com a e b sendo números escolhidos aleatoriamente na faixa $[0, P]$ e P um número primo suficientemente grande e x sendo um valor *hash* inteiro que identifica o *token*(2).

Esse procedimento gera uma permutação aleatória dos atributos com valores variando de $[0, P]$. Considere $d1$ e $d2$ como dois vetores de documentos. Basta então comparar o menor valor da função obtido na aplicação no vetor de atributos de $d1$ com o menor valor obtido em $d2$, se forem iguais, contabilize em M. Essa técnica é conhecida como $minhashing$ [Broder 1997] e permite verificar rotações aleatórias com complexidade linear.

Na próxima seção, explicamos o algoritmo *minHash* desenvolvido.

2. O Algoritmo

Descrevemos nessa seção o algoritmo *minHash* e todos os códigos apresentados, foram desenvolvidos utilizando PySpark.

A entrada do algoritmo *minHash* consiste em um arquivo texto onde cada linha representa um documento. Cada documento pode ter um tamanho variável e representamos esses documentos como listas de palavras. Mapeamos cada linha desse arquivo e salvamos em uma estrutura RDD (Figura 1).

```
import os
filename = os.path.join(FILE_PATH)
fileRDD = (sc
           .textFile(filename,8)
           .map(tokenize)
          )
```

Figura 1. Entrada do algoritmo *minHash*.

A função *"tokenize"* é responsável pelo pré-processamento do texto. A linha então é dividida em *"tokens"*, ou seja, palavras, onde o delimitador é o espaço entre elas.

Para fins de normalização dos dados, cada *token* é convertido para minúsculo, e deve ter mais de dois caracteres, caso contrário, será descartado. É removido espaços em branco e sinal de pontuação. Também é verificado se o *token* é uma palavra válida, ou seja, se não está contida em lista de *"stop words"* (lista de palavras que devem ser descartadas para análise dos textos, pois não agregam conhecimento para medir a similaridade entre textos, por exemplo, artigos, pronomes, etc).

A Figura 2 apresenta as funções utilizadas para normalização dos dados.

```
#padronização dos dados: remover espaço em branco, pontuação, eliminar palavras stopwords, palavras com menos de 3 caracteres
import re
import os

split_regex = r'\W+'

def simpleTokenize(texto):
    novaPalavra = re.sub(r'[^A-Za-z0-9 ]', '', texto).strip()
    return filter(lambda novaPalavra: novaPalavra != '', re.split(split_regex, texto.lower()))


stopWords = set(sc.textFile(stopFile).collect())

def tokenize(string):
    return filter(lambda tokens: len(tokens)>2 and tokens not in stopWords,  simpleTokenize(string))
```

Figura 2. Funções para normalização dos dados.

Após a normalização dos dados, precisamos converter cada *token* em um número inteiro, através de uma função *hash*. Esse número inteiro será utilizado como índice no cálculo do *minhash*. Mapeamos a RDD que contém a lista de lista de *tokens* já normalizada, para que cada *token* seja convertido em inteiro, porém, filtramos para que não exista nenhuma lista vazia na RDD. E gravamos esses novos valores em outra RDD (Figura 3).

Para gerar as nF funções *minhashes* para cada documento são necessários 3 valores inteiros: a, b e P (2). Mapeamos a RDD que contém os *tokens* já convertidos em número e chamamos a função *minHash* para cada posição da RDD.

Na Figura 4 mostramos duas funções: a primeira para gerar as nF funções *minhashes* e a segunda, o cálculo de *minhash*.

```
#converte os tokens para número inteiro
numberTokensRDD = (fileRDD
                   .filter(lambda x: len(x)>0)
                   .map(lambda x: list(map(lambda y: string2numeric_hash(y), x)))
                  )
```

Figura 3. Converte RDD de strings em número.

Para cada valor de nF geramos aleatoriamente os valores para a e b, e esses valores são utilizados para todas as listas. Após gerarmos todas as nF funções *minhashes*, escolhemos a de menor valor para representar cada nF para cada documento (Figura 4).

No final, a saída será uma lista de listas, onde cada lista filha terá o tamanho igual ao número de nF, ou seja, essa lista armazenará o menor valor de cada função *minhash* para cada documento.

Para exemplificação considere nF=5 e a lista filha de posição 0 (documento 1), ou seja, o primeiro item da lista pai. A lista filha terá 5 valores, onde o primeiro valor corresponda ao menor valor da função *minhash* $nF = 1$ para o documento 1, o segundo corresponda ao menor valor da função *minhash* $nF = 2$ para o documento 1, e assim para todos os valores de nF.

Após gerar os *minhashes*, já temos as listas de *minhash* para cada documento e vamos então, calcular a similaridade de Jaccard. O Jaccard é calculado para todos os pares de documentos e então, utilizamos uma função para gerar os pares de documentos distintos (Figura 5).

Após gerar os pares de documentos, calculamos a similaridade de Jaccard com *minhash*. Nessa etapa não será verificado a ocorrência de palavras entre cada par de documentos, mas sim, a lista de *minhash* de cada documento. A Figura 6(a) mostra o algoritmo utilizado para esse cálculo.

Verifica-se posição a posição se os valores são iguais. Sendo assim, a similaridade de Jaccard com *minhash* será o total de valores iguais na mesma posição dividido pelo tamanho do vetor, ou seja, nF que é o total de funções de *minhashes* geradas.

Na figura 6(b), mostramos o código completo da chamada da função de Jaccard para cada posição da RDD. Depois, calculamos o maior e o menor valor da similaridade de Jaccard encontrados, assim como a quantidade de pares de documentos em que ocorreu esses valores.

Após a explicação do algoritmo, mostramos os resultados encontrados.

3. Resultados Experimentais

O algoritmo foi implementado em PySpark e executado com Jupyter Notebook.

Para esse experimento, utilizamos uma base de dados de *twitter* intitulada **Twitter US Airline Sentiment**, disponível em https://www.kaggle.com/crowdflower/twitter-airlinesentiment/data. Filtramos essa base de dados, e utilizamos somente as primeiras 5.000 linhas e um único atributo no arquivo, que representa o texto do *twitter*.

Executamos o algoritmo *minHash* para $nF = 5, 50, 100$, onde nF representa o número de funções de *minhashes*. Quanto maior o número de nF, melhores os resultados,

ou seja, a maior quantidade de pares de documentos com jaccard com *minhash* mais próximo de 1. Para esses testes, $nF = 100$ foi o que apresentou o maior valor encontrado como mínimo (Tabela 1).

Algoritmo *minHash*		
nF	**Máximo - Qtde**	**Mínimo - Qtde**
5	1,00 Q=2436	0,003 Q=63
50	1,00 Q=3566	0,045 Q=17
100	1,00 Q=4219	0,670 Q=5

Tabela 1. Resultado do Algoritmo *minHash*.

Para o par de documentos $D1820, D2084$ a similaridade de Jaccard com *minHash* é igual a 1, pois ambos os textos são idênticos.

$D1820$ = "@AmericanAir ok thank you!"
$D2084$ = "@AmericanAir ok thank you!"

Já para o par de documentos $D4285, D4290$, a similaridade de Jaccard com emph-minhash é igual $0,67$. Podemos verificar que os textos são bem similares, apenas diferindo no final do texto, a url.

$D4285$ = "@JetBlue: Our fleet's on fleek. http://t.co/cRFrwpc1Sx"
$D4290$ = "@JetBlue: Our fleet's on fleek. http://t.co/g97HAbyeP5?"

Na próxima seção, apresentamos a conclusão.

4. Conclusão

Para finalizar, concluímos que calcular a similaridade entre textos e obter um resultado satisfatório, é preciso que os dados estejam normalizados.

Embora não foi apresentado nesse projeto o resultado do cálculo de Jaccard puro sobre os textos, o cálculo de Jaccard com *minhash* é bem mais eficiente em termos de velocidade computacional, o que nos motiva para continuar utilizando *minhash* para novos resultados.

O algoritmo foi implementado de forma bem simples, mas será modificado aproveitando-se mais dos recursos *map* e *reduce* do PySpark para melhor desempenho dos mesmos e também, para uma comparação com o mesmo algoritmo implementado em C/C++.

Referências

Broder, A. (1997). On the resemblance and containment of documents. In *Proceedings of the Compression and Complexity of Sequences 1997*, SEQUENCES '97, pages 21–, Washington, DC, USA. IEEE Computer Society.

```python
from random import*

def gerarMinHash(nF):
    list1 = []
    list2 = []
    listAux1 = []
    listAux2 = []

    for x in range(nF):
        a = randint(1,p-1)
        b = randint(1,p-1)
        novaRDD = numberTokensRDD.map(lambda x: minhash(x, a, b, p))
        for x in novaRDD.collect():
            list1.append(x)

        list2.append(list1)
        list1 = []

    for i in range(len(list2[0])): #i até qtde de itens que tem em cada posição da lista
        for c in range(len(list2)):#qtde de listas
                    listAux1.append(list2[c][i])

        listAux2.append(listAux1)
        listAux1 = []

        return gerarPares(listAux2)


#cria novo RDD com o resultado dos minHashes
nF = 10 #qtde de funções minHashes
minHashesRDD = sc.parallelize(gerarMinHash(nF))
```

```python
# Map the indices to hash values and take the minimum.
import numpy as np

def minhash(v, a, b, p):

    indices = np.array(v)

    return  np.array((((a * indices) + b) % p)).min()
```

Figura 4. Funções para gerar nF *minhashes*.

```python
#gerarPares: usado para similaridade de jaccard
def gerarPares(x):
    #print 'meu x: ', x
    i = 0
    while i < len(x)-1:
        c = i+1
        l = []
        while c < len(x):
            l.append(x[i])
            l.append(x[c])
            lista.append(l)
            l = []
            c = c+1

        i = i+1
    return lista
```

Figura 5. Função gerar pares de documentos.

```python
def jaccard(d1, d2):

    equals = 0
    n = len(d1)

    for i in range(n):
        if d1[i] == d2[i]:
            equals +=1

    result = float(equals) / (n)

    return result
```

(a)

```python
#Similaridade de jaccard entre 2 documentos
jaccRDD = minHashesRDD.map(lambda x: (jaccard(x[0],x[1])))

#valor mínimo e máximo e suas respectivas qtde, da similaridade de jaccard
maxJ = jaccRDD.max()
minJ = jaccRDD.min()

countMin = jaccRDD.filter(lambda x: x == minJ ).count()
countMax = jaccRDD.filter(lambda x: x == maxJ ).count()

print "\t\tMin\tMax"
print "Jaccard:\t{:.2f}\t{:.2f}".format( minJ, maxJ)
print "Quantid:\t{0:2d}\t{1:3d}".format( countMin, countMax )
```

(b)

Figura 6. Resultado do cálculo de Jaccard sobre *minhash*.