Algoritmo MinHash e Bigram 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. Verificar a similaridade entre textos não é uma tarefa fácil, visto que, na maioria das vezes, os dados não se encontram padronizados. 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 $[0, P]$. 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.

O *ngram* é uma sequencia de n itens de uma determinada sequência de texto. Um modelo de *ngram* é um tipo de modelo de linguagem probabilística para prever o

próximo item em uma sequência dessa forma na forma de um modelo de Markov $(n-1)$. Para $n = 1, 2, 3$, nos referimos como "unigram", "bigram" e "trigrama", respectivamente. Existem diversas aplicações, como por exemplo, para contar sequência em um DNA.

Na próxima seção, é mostrado os algoritmos desenvolvidos para as técnicas apresentadas.

2. Os Algoritmos

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 2).

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

Figura 1. Entrada do algoritmo *minhash* em PySpark

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. São palavras que não agregam conhecimento e não têm importância para medir a similaridade entre textos.

A Figura 2 apresenta o algoritmo em PySpark utilizado para normalização dos dados.

```
import re

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)>3 and tokens not in stopwords,  simpleTokenize(string))
```

Figura 2. Funções para normalização dos dados em PySpark

Para o cálculo do *minhash*, precisamos converter cada *token* em um número inteiro, através de uma função *hash*. Para gerar as nF funções *minhashes* para cada documento são necessários 3 valores inteiros: a, b e P (2).

Após gerarmos todas as nF funções *minhashes*, escolhemos a de menor valor para representar cada nF para cada documento (Figura 2).

```
# 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 3. Função para cálculo de *minhash* em PySpark

Nessa etapa, calculamos a similaridade de Jaccard com *minhash*, ou seja, não será verificado a ocorrência de palavras entre cada par de documentos, mas sim, o vetor de *minhash* de cada documento. A Figura 2 mostra o algoritmo utilizado para esse cálculo.

```
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
```

Figura 4. Função para cálculo de jaccard sobre minhash em PySpark

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.

No final, para cada documento, mostramos o valor do jaccard para o par que apresentou melhor resultado, ou seja, mais próximo de 1. Nesse caso, se temos 10.000 documentos, teremos 10.000 linhas com o melhor resultado.

Finalizando o algoritmo do *minHash*, mostramos um algoritmo que também pode ser bastante útil na fase de normalização de dados quando se se trata de similaridade de textos, o "bigram", utilizando as função *map* e *reduce* do PySpark. Ele pode ser usado na fase do pré-processamento do texto, para combinar palavras, dependendo da necessidade (Figura 2).

```
textoRDD = sc.parallelize(listaTexto)
bigramsRDD = (texto
              .map(tokenize)
             .flatMap(lambda s: [((s[i],s[i+1]),1) for i in range (0, len(s)-1)])
             .reduceByKey(lambda x, y : x+y)
             )

print bigramsRDD.collect()
```

Figura 5. Algoritmo *bigram* em PySpark

Criamos uma RDD a partir de uma lista que contém o texto. Mapeamos cada posição normalizando os dados, com os mesmos critérios já apresentados no algoritmo do *minhash*. Depois geramos um conjunto de palavras duas a duas, acrescentando o valor 1 em cada uma delas. Para finalizar, utilizamos o *reduce* para reduzir esse conjunto, agrupando as chaves iguais e somando a quantidade encontrada.

Após a explicação dos algoritmos, mostramos os resultados encontrados.

3. Resultados Experimentais

Os algoritmos foram implementados PySpark e executados 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 sobre *minhash* mais próximo de 1. Para esses testes, $nF = 100$ foi o que apresentou os melhores resultados.

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 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?"

O algoritmo *bigram* não foi utilizado nessa implementação do algoritmo *minhash*, apenas para testes para possíveis alterações no algoritmo.

Na próxima seção, apresentamos a conclusão.

4. Conclusão

Para finalizar, concluímos que calcular a similaridade entre textos não é tarefa fácil, e quanto mais padronizado seja os dados, mas eficiente pode ser o cálculo da similaridade de jaccard com *minhash*.

O cálculo de jaccard com *minhash* é bem mais eficiente em termos de velocidade computacional que o cálculo de jaccard puro sobre os textos.

Os algoritmos foram implementados de forma bem simples, mas esses poderão serem alterados aproveitando mais dos recursos *map* e *reduce* do PySpark para melhor desempenho dos mesmos.

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.