Métodos Probabilísticos para Engenharia Informática

Docentes:
Prof. António Teixeira e Prof. Amaro de Sousa

Deteção de Itens Similares & Deteção de Pertença a um Conjunto

Tema 11 - Padrões Climáticos

Carolina Araújo, 93248
Rui Fernandes, 92952

DETI
Universidade de Aveiro
12-2019

---

1 Introdução

Este trabalho prático foi motivado pela necessidade de criar um programa na linguagem *Java* que facilitasse a deteção de itens similares entre si, assim como a deteção de pertença de um ou mais itens a um dado conjunto.

Como tal, com base em conhecimentos adquiridos por ambos os alunos na cadeira de **Programação Orientada a Objetos** e com os conhecimentos aprendidos este semestre em **Métodos Probabilísticos para Engenharia Informática**, foi-nos possível desenvolver uma solução para o trabalho em questão.

O objetivo deste relatório é, então, explicar a utilização deste mesmo programa e evidenciar pontos fulcrais da nossa implementação.

2 Preparação

Antes de avançar com o desenvolvimento de código, foram realizadas e estudadas atentivamente as aulas práticas PL05, PL06 e PL07, uma vez que estas incidiam sobre as ferramentas importantes que iriamos ter de implementar: **Funções de Dispersão**, **Bloom Filter** e **MinHash**. Seguidamente, o grupo começou a fazer *brainstorming* de possíveis ideias daquilo que gostaríamos que uma aplicação relacionada com padrões climáticos fosse capaz de fazer, tendo em conta que teríamos de utilizar as funções acima referidas.

3 Funções de Dispersão/Hash Functions

É uma função que seja capaz de transformar um elemento *(key)* de um dado *Data Type* e retornar um valor inteiro que possa ser usado para aceder a uma estrutura de dados onde, na posição retornada, podem estar armazenadas as mais diversas informações relativas a esse tal elemento inicialmente passado, quando associadas a *Hash Tables*. São usadas para facilitar o armazenamento e o acesso a *data*, ocupando muito menos espaço do que guardando os elementos por si só.

Considera-se muito boa uma função que evite colisões, isto é, duas *keys* diferentes cujos valores retornados após o *hashing* seja igual, sendo, portanto, dirigidos à mesma posição na *Hash Table*.

4 Bloom Filter

Criado em 1970 por **Burton Howard Bloom**, é uma ferramenta usada para testar a presença de um elemento num determinado *set*, sendo bastante útil para pré-processamento, uma vez que usa uma quantidade de memória limitada. Com recurso a funções de dispersão, todos os elementos pertencentes a um determinado conjunto são passados por estas funções (uma ou mais), das quais resulta um número inteiro usado para aceder a um vetor - o filtro. O filtro, inicializado a zeros, coloca a 1 todas as posições devolvidas pelas funções de dispersão.

Assim, quando se quer verificar a presença de um elemento num dado conjunto, basta passar esse mesmo pelas diferentes funções de dispersão e verificar se **todos** os valores do array nos índices retornados estão a 1. No entanto, **falsos positivos** são algo que deve ser tido em conta, uma vez que quantos mais elementos tem o conjunto, mais facilmente se preenche o vetor com 1's e pode retornar resultados adversos. Não existem **falsos negativos**.

4.1 Calcular a probabilidade de falsos positivos

Seja $\underline{\text{k}}$ o número de funções de dispersão utilizadas pelo filtro e $\underline{\text{n}}$ o tamanho do mesmo, a probabilidade de obter falsos negativos após adicionar $\underline{\text{m}}$ elementos é dada por $\mathrm{P}_{fp} = \left[1 - \left(1 - \frac{1}{n}\right)^{mk}\right]^k$. Sabendo que $lim_{n\to\infty}\left(1 - \frac{1}{n}\right)^n = e$, pode-se, por fim, concluir que $\mathrm{P}_{fp} \approx \left(1 - e^{-\frac{km}{n}}\right)^k$.

4.2 Calcular o $\underline{\text{k}}$ ótimo

Um $\underline{\text{k}}$ considerado ótimo será aquele que diminui a probabilidade de falsos positivos acima referida (P_{fp}).

Como é que este $\underline{\text{k}}$ dito ótimo pode ser calculado? Ao minimizar $P_{fp} \to ln\left(P_{fp}\right)$ podemos afirmar que $ln\left(P_{fp}\right) = k\ ln\left(1 - \left(1 - \frac{1}{n}\right)^{km}\right)$. Calculando os zeros da derivada de $ln\left(P_{fp}\right)$, obtêm-se que $\left(1 - \frac{1}{n}\right)^{mk} = \frac{1}{2}$.

Concluindo, $k_{ótimo} = \frac{ln\left(\frac{1}{2}\right)}{m * ln\left(1 - \frac{1}{n}\right)} \approx \frac{0.693n}{m}$, sendo que se utilizará o inteiro mais próximo da solução final.

5 MinHash

Esta técnica, inventada por **Andrei Broder** em 1997, é utilizada para rapidamente estimar o quão similar são dois conjuntos de data. De grande importância no ramo da Informática, servindo para, por exemplo, prevenir casos de plágio, quer seja em código, páginas Web ou até na deteção de imagens praticamente idênticas.

O funcionamento básico desta ferramenta passa por pegar em cada *set* de informação e passar cada um dos seus elementos por uma função de dispersão. Seguidamente, para cada *set* verifica-se qual dos valores obtidos após

o *hashing* é o mais pequeno - este será o valor que é colocado na matriz *Signatures* na linha correspondente à hash function utilizada. Este processo é então repetido para um determinado número de hash functions completando assim a matriz.

5.1 Distância e Similaridade de Jaccard

A **Similaridade de Jaccard** de dois conjuntos é dada pelo quociente entre a dimensão da interseção dos conjuntos com a dimensão da união, ou seja, $similaridade_{(set1,set2)} = \frac{\#(set1 \cap set2)}{\#(set1 \cup set2)}$.

A **Distância de Jaccard** entre dois conjuntos é dada por 1 -$similaridade_{(set1,set2)}$.

6 App

6.1 Tipo de dados utilizado

O *data set* escolhido, que é o mesmo para a *App* e para uma parte dos testes, contém a informação, hora a hora, da temperatura de diferentes cidades do mundo, contendo também a data com dia-mês-ano. Decidimos usar apenas as informações desde 2013 até 2017 de 30 estados americanos distintos para o uso da aplicação se tornar mais simples ao utilizador.

6.2 Entender melhor a App

A aplicação conjunta desenvolvida visa utilizar os dois módulos desenvolvidos (**Bloom Filter** e **MinHash**) e aplicá-los a um caso real, no nosso caso a dados metereológicos (temperaturas). O display é simples, efetuado na linha de comandos, tendo por base um menu inicial que leva a outros sub menus.

Ao iniciar a App o utilizador depara-se com o menu inicial, podendo escolher utilizar as funcionalidades do **Bloom Filter** ou procurar por **Similaridades**.

Nas funcionalidades do **Bloom Filter** o utilizador pode procurar pela existência de pares de temperaturas mínimas e máximas ou temperaturas médias na base de dados, podendo também remover uma entrada da mesma.

Quanto às **Similaridades**, o utilizador pode procurar dias, meses ou anos com conjuntos de temperaturas semelhantes, sendo devolvidas aquelas que têm distância de Jaccard bastante pequena.

Em ambos os ramos da *App* o utilizador escolhe o volume de dados a utilizar, os de 1 cidade (à escolha) ou todas as cidades da base de dados, isto através de um menu dedicado a tal.

6.3 Vantagens e limitações da solução apresentada

Uma das limitações encontradas foi a implementação de *Locality Sensitive Hashing*, uma vez que por falta de tempo não foi possível garantir o funcionamento da mesma a 100%, pelo que foi preferível retirar por completo esta funcionalidade e focar mais em bom testes e numa boa aplicação.

Vemos qualquer uma das funcionalidade apresentadas na aplicação como uma vantagem, uma vez que as implementamos sempre com o intuito de que as mesmas usufruíssem o máximo possível do **Bloom Filter** e da **MinHash**, sendo também algo que um possível utilizador tivesse real interesse. Por exemplo, uma das ideias que acabou por ser retirada era a passagem de uma sequência de 24 temperaturas, visando saber se havia algum dia com essa mesma sequência presente na *data base*. Ora, claramente em termos práticos e reais, esta não seria uma boa funcionalidade.

7 Testes

O objetivo dos testes passa por verificar o bom funcionamento das nossas implementações do **Bloom Filter** e da **MinHash**, tanto para com o nosso *data set* como para, no caso do **Bloom Filter**, valores aleatórios.

Ambos os testes se encontram no ficheiro *Testes.java* e basta colocar a correr, uma vez que o output é explícito sobre o que está a ser feito e quais os resultados.

7.1 Bloom Filter Tests

Escolhemos utilizar a informação diária de uma das cidades durante os diferentes anos para correr estes testes, verificando que tudo se encontra a funcionar de acordo com o previsto e o necessário. Foram também utilizados valores obtidos aleatoriamente para o segundo tipo de testes do **Bloom Filter** uma vez que, sendo quase impossível gerar dois números inteiros idênticos num *range* grande, foi usado $[1 : 99999999]$, caso um elemento do set2 pertença ao **Bloom Filter** do set1, é muito provável que seja falso positivo.

7.1.1 Resultados obtidos nos testes do Bloom Filter

Para um melhor desempenho do **Bloom Filter**, iniciamos o mesmo com 10 vezes o número de elementos a inserir, adicionamos todos as temperaturas acima referidas e inicia-se o teste. Passa por verificar se o **Bloom Filter** reconhece, depois da inserção, que não está vazio, a partir da função *isEmpty()*. Seguidamente vamos confirmar, com recurso a *isElement(int element)* que elementos que não se encontram no *data set* retornam *false* e outros, temperaturas que realmente ocorreram, retornam *true*, informando também sobre o número de vezes que cada uma delas se deu.

Passa-se então a uma parte crucial de testes do **Bloom Filter**: testar a existência de falsos positivos, calcular o k ótimo e as probabilidades, tanto teórica como prática, da ocorrência de falsos positivos.

Esta é a parte do teste é realizada com valores aleatórios e não com o *data set*.

Descobre-se que o melhor valor para k é 7, sendo possível comparar o valor das duas probabilidades de falsos positivos consoante k. O Limite Inferior da Probabilidade Teórica de Falsos Positivos pode ser calculada como $2^{-k_{ótimo}}$ que neste caso é $2^{-7} = 0.0078$. Como apresentado no *output* dos testes, o valor teórico calculado para $k = 7$ é 0.0082, ou seja, não difere muito do limite.

São ainda apresentados o número de falsos positivos resultantes de cada k, que tende a crescer com o aumento sucessivo do valor de k, o que é confirmado também a partir do gráfico, é possível verificar que a partir do $k_{ótimo}$ ambas as probabilidades de falsos positivos tendem a crescer. Também é possível entender o valor teórico de probabilidade para k's pequenos, uma vez que poucas funções de dispersão a lidar com *data sets* tão grandes vai claramente resultar em bastantes colisões que, por sua vez, resultam em falsos positivos.

7.2 MinHash Tests

Novamente usando a informação diária de uma cidade, nestes testes visa-se apenas confirmar o bom funcionamento do cálculo da similaridade entre pares, e, por sua vez, da distância de Jaccard.

A nossa implementação da **MinHash** apenas permite comparar similaridades e distâncias entre dias da mesma cidade ou entre todas as cidades disponíveis, sendo que, para testar, decidimos usar apenas a comparação entre a mesma cidade. Consideramos que ter apenas estas duas funcionalidades não é uma desvantagem, visto que acrescentar outras tornar-se-ia redundante.

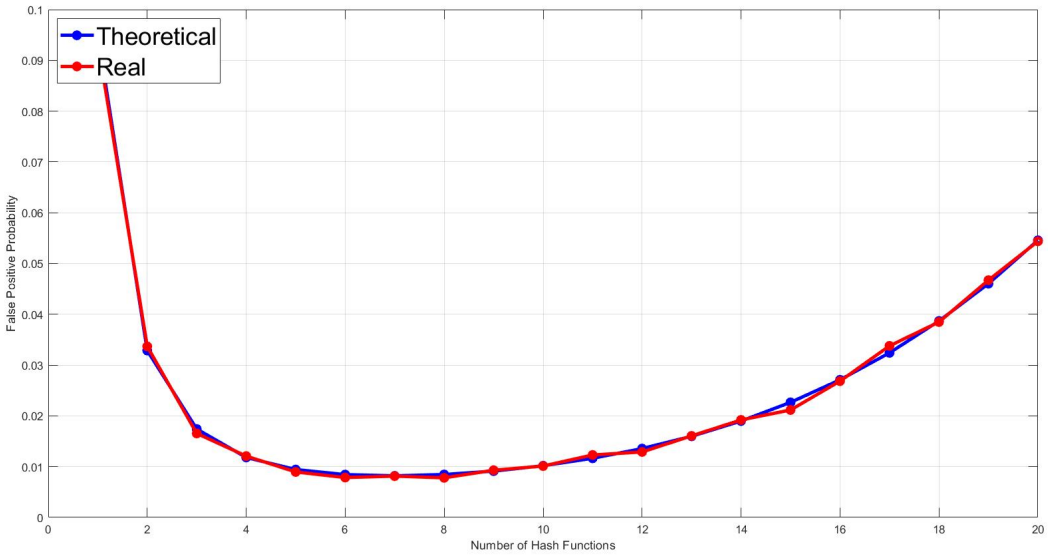

Figure 1: Variação das Probabilidades de Falsos Positivos (Teórica e Prática) com base no valor de k

7.2.1 Resultados obtidos nos testes da MinHash

Como esperado, com um *threshold* default tão pequeno (0.05), poucos serão os dias com uma similaridade tão alta (0.95), pelo que preferimos apresentar apenas estes dias para i *output* do teste não se tornar demasiado extenso.

Caso seja desejado, é possível mudar no ficheiro *Testes.java* o valor do *threshold* para ver mais valores, com maiores distâncias entre si.

8 Conclusão

Por fim, conclui-se que os objetivos propostos pelos docentes da cadeira foram devidamente alcançados. Embora tenham surgido algumas advertências aquando do desenvolvimento da solução final, considera-se que a realização do trabalho prático foi um sucesso. A título de exemplo, os percalços que foram aparecendo resumem-se a exceções do tipo *NullPointer* e *ArrayIndexOutOfBounds*, embora também tenha havido momentos de maior tensão, onde a procura de uma solução para as barreiras que iamos encontrando se tornou complicada e exigiu algumas horas de pesquisa e reestruturação do código.

Em suma, foi também concluído que a implementação de funções como **MinHash** e **Bloom Filter** facilitam bastante a pesquisa pelas mais diversas estruturas de dados, evitando a resolução de eventuais problemas com recurso a algoritmos do tipo **Brute Force**.

9 Bibliografia

[1] https://en.wikipedia.org/wiki/Hash_function

[2] https://en.wikipedia.org/wiki/Bloom_filter

[3] https://en.wikipedia.org/wiki/MinHash