Bloom-filter + Locality-Sensitive Hashing

Métodos Probabilísticos para Engenharia Informática

Universidade de Aveiro

Pedro Martins 76551
Ricardo Jesus 76613

17 de Dezembro de 2015

Bloom-filter + Locality-Sensitive Hashing

Departamento de Electrónica, Telecomunicações e Informática

UNIVERSIDADE DE AVEIRO

Pedro Martins 76551, pbmartins@ua.pt
Ricardo Jesus 76613, ricardojesus@ua.pt

17 de Dezembro de 2015

Resumo

ABSTRACT

Conteúdo

Lista de Figuras

Capítulo 1

Introdução

Na área da informática, muitas vezes é necessário saber se algo pertence a conjunto de forma eficiente. Em muitas linguagens de programação, a maneira mais comum seria percorrer todo o conjunto e comparar, um a um, todos os elementos até encontrar o que se inicialmente andava à procura. No entanto, este método é deveras ineficiente, principalmente se se estiver a trabalhar com conjuntos com milhões ou mais elementos, e aí que surgem os *Bloom-filters*.

Este filtros usam uma ou várias funções de *hashing* para determinar a pertença de um elemento no *set*. São muito utilizados em grandes conjuntos de dados e em diversas aplicações como o corretor ortográfico dos computadores, *smartphones*, etc., análise textual, entre outras. Contudo, é um método probabilístico com um valor de erro associado.

Neste trabalho, foi desenvolvido em Matlab um *Bloom-filter* e desenhada uma interface para um função de *hashing* desenhada em C++ para atingir este objetivo.

Capítulo 2

Bloom-filter

Os *Bloom-filters* usam uma ou várias funções de *hashing* para determinar a pertença de um elemento no *set*. Deste modo, mesmo que se opere sobre um conjunto com milhões ou milhares de milhões de elementos, será sempre um processo muito mais eficiente do que se se iterasse sobre todo o *set* à procura do elemento em questão (dependendo também do número e da qualidade das funções de *hashing* utilizadas).

No entanto, é um método probabilístico, e tem um erro associado. Na criação de um *Bloom-filter* define-se sempre um erro máximo esperado, erro esse que apenas ocorre na "inclusão", isto é, um membro que à partida já se saiba que pertença ao conjunto nunca será considerado não pertencente, mas alguns membros que, à partida, se saiba que não estão no *set*, podem ser considerados como pertencentes ao conjunto.

Para diminuir esse erro, por norma, para além de aumentar o tamanho do *set*, usa-se um número variado de funções de *hashing* descorrelacionadas entre si, em vez de utilizar apenas uma. Isto permite que os resultados do *hashing* estejam mais dispersos por todo o vetor. Neste caso, em vez de utilizarmos k funções diferentes, utilizaremos a família de funções *FarmHash*, desenvolvida pela Google, em que um dos argumentos, a `seed`, especifica cada uma das funções.

Na implementação do filtro neste trabalho (*BloomFilter.m*) foi desenvolvida uma classe, baseada num *Counting Bloom-filter* (conta-se o número de vezes que cada elemento é adicionado ao filtro), com 5 atributos:

k Número de funções de *hashing* utilizadas.

byteArray Limite de vezes que o contador de cada posição é incrementado (255).

arraySize Tamanho do vetor de *bits*.

amountAdded Número total de elementos adicionados ao ⌊array⌋.

expectedMaxSize Tamanho do conjunto que se pretende adicionar do vetor.

No construtor da classe, são calculados os valores do tamanho do vetor (`arraySize`) e do número de funções de *hash* necessárias, consoante os valores passados como argumentos do próprio construtor, a probabilidade de falsos positivos, isto é, o erro esperado (`falsePositiveProbability`) e o tamanho do conjunto que se pretende adicionar ao vetor (`expectedMaxSize`).

Assumindo que a probabilidade de falsos positivos p é

$$p = \left(1 - e^{-\frac{km}{n}}\right)^k$$

e, usando o tamanho do vetor de *bits* n e o tamanho do conjunto que queremos adicionar ao filtro m,

$$a = \left(1 - \frac{1}{n}\right)^m$$

para determinar o número k ótimo de funções de *hashing* que se devem utilizar, deduz-se que

$$\ln p = k * \ln\left(1 - a^k\right) \Leftrightarrow k = \frac{n * \ln 2}{m}$$

A partir das fórmulas acima encontradas, também se deduz que

$$n = \frac{m * \ln\left(\frac{1}{p}\right)}{(\ln 2)^2}$$

Para além do construtor, existem também métodos para adicionar e verificar a existência de elementos no filtro.

getIndexes Devolve os vários índices para os quais a função de *hashing* aplicada à *string* passada como argumento aponta.

add Adiciona um elemento ao filtro.

contains Verifica se o elemento passado como argumento existe no filtro (poderá haver ocorrência de falsos positivos).

count Devolve o número de vezes que um dado elemento foi adicionado ao vetor.

remove Remove do filtro, caso exista, o elemento passado como argumento.

maxCount Devolve o número máximo de vezes que um elemento foi adicionado ao vetor.

minCount O inverso da função `maxCount`.

Setters Coleção de funções utilizadas para modificar os atributos do *Bloom-filter*, caso o campo `debug` (argumento passado ao construtor) esteja com o valor 1.

2.1 Testes

Para testar este módulo, foram desenvolvidos diversos, de entre os quais uns para verificar qual seria o número ideal de funções de *hashing* (`k`) e outro para verificar o tamanho ideal do vetor de *bits* (`n`).

No entanto, também foram realizados outros testes relativamente às funções de *hashing* utilizadas pelo *Bloom-filter*.

2.1.1 Distribuição de funções de *hashing*

Este módulo (*test_hashFunction_distribution.m*) tem como principalmente objetivo provar que as funções de *hashing* têm uma distribuição uniforme, para diversos valores de `k` (neste caso, irá variar entre 1 e 10).

A família de funções de *hashing* usada tanto no filtro como nos testes é designada por *FarmHash*, desenvolvida pela Google. Foi apenas criada uma interface para que pudesse ser usada em Matlab.

Neste teste, foi gerado um conjunto de *strings* aleatórias, usando a função `generateStrings`, que aceita como argumentos o tamanho do conjunto que devolverá e o tamanho máximo das *strings* que irá gerar (caso o terceiro argumento seja 0, elas terão tamanho fixo, caso contrário, será definido como tamanho máximo), e gerados diversos valores de *hashing*, consoante os diversos elementos do conjunto e da `seed` correspondente.

As distribuições para cada valor de `k` deverão ser o mais uniformes possíveis, e os resultados comprovam-no, tal como se pode ver nos histogramas da Figura 2.1.

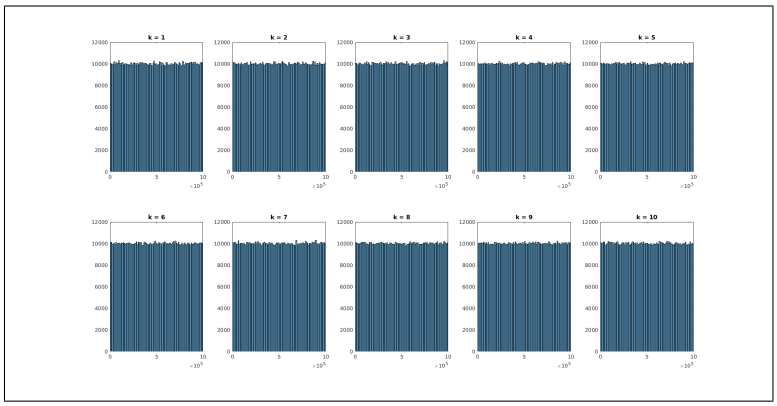

Figura 2.1: Resultados das distribuições das `k` funções de *hashing*

2.1.2 Correlação das funções de *hashing*

Para que se possam ser utilizar as várias funções de *hashing*, é necessário que as mesmas sejam descorrelacionadas, isto é, que o coeficiente de correlação seja 0. No entanto, neste caso, considerar-se-á um erro de 0.01 (é quase impossível que os valores sejam iguais a zero, portanto, aceita-se esta ligeira variação).

Este módulo encontra-se no ficheiro *test_hashFunction_correlation.m* e, tal como noutros testes, gerar-se-á um *set* de *strings* aleatórias e os respetivos valores de *hashing* para diferentes valores de `k`, a variar entre 1 e 10. Foi também criada uma matriz de dimensões `k` por `numTests`, isto é, cada linha terá diferentes valores de *hashing* para um mesmo valor de `k`. De seguida, calcular-se-ão os coeficientes de correlação entre de cada conjunto de *hash codes* (linha) desses mesmos valores com o auxílio da função `corrcoef` (calcula o coeficiente de correlação entre dois conjuntos, neste caso, duas linhas distintas).

Por fim, é gerado um gráfico recorrendo à função `surf` para mostrar os valores de correlação resultantes. Como verificamos, todos os valores situam-se abaixo do erro assumido (à exceção dos valores em que assumimos o mesmo valor de `k`, isto é, compararmos a mesma linha, sendo que, nesse caso, o valor será 1), daí se poder considerar que as funções de `hashing` são descorrelacionadas, tal como se pretendia.

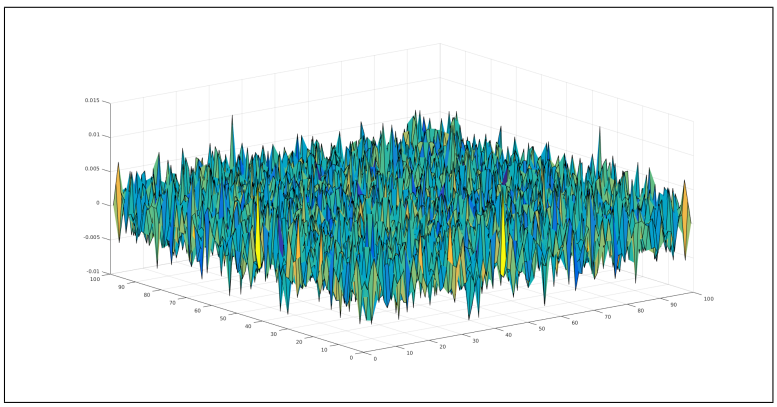

Figura 2.2: Resultados do teste de correlação de `k` funções de *hashing*

2.1.3 Independência das funções de *hashing*

Para além do teste de correlação, foi também efetuado um teste de independência das funções de *hashing* (*test_hashFunction_independence.m*), adaptado da implementação do professor António Teixeira. A independência implica descorrelação, e com isto, pode-se provar com qualquer um dos testes que as funções são descorrelacionadas. O objetivo é provar que cada

par de colunas é independente entre si.

Primeiramente, é criado um *set* de cem mil *strings* aleatórias e, seguidamente, outra matriz dos respetivos *hash codes* com `k` (neste teste, `k = 10`) diferentes *seeds*, utilizando a família de funções *FarmHash*, de tamanho `k` (número de funções de *hashing* que se pretende aplicar) por `N` (número de testes que se pretendem executar). De seguida, é gerado um vetor de 10 elementos linearmente espaçados (`x`) entre 0 e `N`, e, para cada par de colunas, é criada uma matriz (`pmf`) de dimensão `length(x) - 1 / length(x) - 1`. Depois, guarda-se em cada elemento da matriz o número de elementos das colunas de *hash codes* que se situam num mesmo intervalo (esse intervalo é definido pelo vetor `x` já criado).

Por fim, é calculada a matriz de probabilidade conjunta de duas colunas (*PMF - Probability Mass Function*), e, a partir da mesma, são calculados os vetores de probabilidade individuais de cada uma das colunas. Finalmente, é calculado o resultado da multiplicação destes dois últimos vetores (dois acontecimentos A e B são independentes se $P(A\&B) = P(A) * P(B)$), e compara-se com a matriz de probabilidade conjunta.

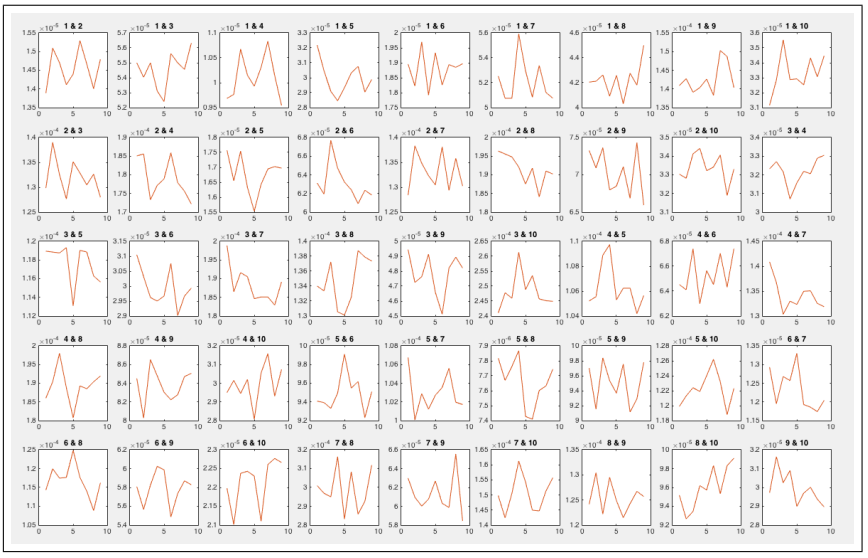

Figura 2.3: Resultados do teste de independência de `k` funções de *hashing*

Como os valores da variação de resultados são mínimos (como se observa na Figura 2.3 todos os valores rondam os $10^{-}4$), podemos considerar que a família de funções é independente, o que nos leva a concluir que são descorrelacionadas.

2.1.4 Número ideal de funções de *hashing*

O programa de teste para o valor de `k` ideal encontra-se com o nome *test_ optimalK.m.*

Começa-se por criar uma instância da classe de *Bloom-filter* anteriormente desenvolvida (com o campo `debug` inicializado a 1, para que se possam utilizar as funções de alteração de valores) e gerar dois conjuntos (um dos quais será adicionado ao filtro, outro não) distintos de cem mil (valor que pode ser alterado) *strings* (como poderá haver *strings* iguais nos *sets*, o tamanho será sempre ligeiramente inferior ao definido anteriormente). De seguida, define-se o tamanho do vetor de *bits* do filtro como oito vezes maior do que o tamanho dos conjuntos de *strings* gerados.

Por fim, itera-se sobre um vetor de `k` que se definira anteriormente (neste caso, é um vetor com valores de 1 a 15) e, cada ciclo, define-se um `k` no *Bloom-filter*, adiciona-se os elementos do vetor incialmente escolhido como aquele que se iria adicionar ao filtro e verifica-se se algum dos elementos do outro conjunto de *strings* pertence ou não ao filtro. Caso pertença, é incrementado um contador, para, no final do ciclo, ser calculada a probabilidade de falsos positivos.

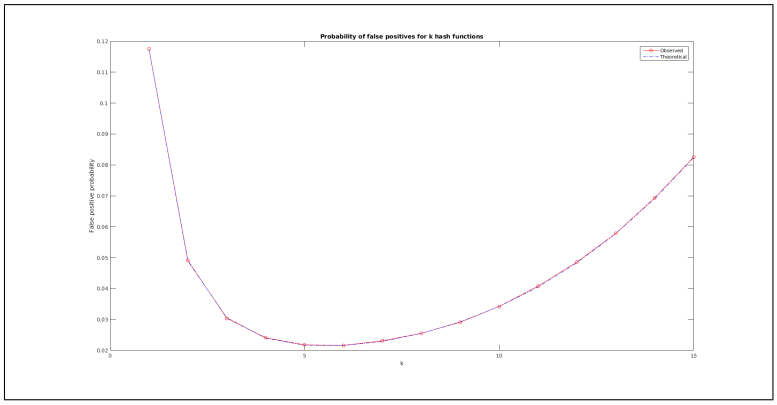

Figura 2.4: Resultados do teste do valor ideal de funções de *hashing* (`k`)

De acordo com o gráfico da Figura 2.4, usando a fórmula

$$p = \left(1 - e^{-\frac{km}{n}}\right)^k$$

para determinar a probabilidade de falsos positivos (teórica) para diferentes valores de `k` e usando os valores do teste acima, verificamos que as diferenças entre o valor teórico e observado são mínimas (o que significa que a função de *hashing* devolve excelentes resultados, mas isso será abordado mais abaixo), e que o valor ideal é 6, para a função de *hashing* em questão. Para uma função diferente, os valores poderão diferir.

2.1.5 Número ideal do vetor de *bits*

O módulo para o teste do valor ideal do tamanho do vetor `n` encontra-se no ficheiro *test_ optimalN.m*.

Tal como no teste do Subseção 2.1.4, também são criados dois conjuntos de `strings` aleatórias com o mesmo propósito, um para ser adicionado ao filtro e o outro não. É também criado um vetor com diferentes valores de `n`, que vão desde o tamanho dos conjuntos de *strings* até 10 vezes esse valor, com uma diferença de metade do mesmo valor entre cada.

De seguida, itera-se sobre os valores deste último vetor de valores e vai-se criando uma instância de um *Bloom-filter* com os valores de `n` (`arraySize`) e `k` (depende do tamanho do vetor) a cada passagem. Adiciona-se um dos conjuntos de *strings*, verifica-se a existência dos elementos do outro que não foi adicionado e, por fim, calcula-se a probabilidade de falsos positivos, tal como no teste anterior.

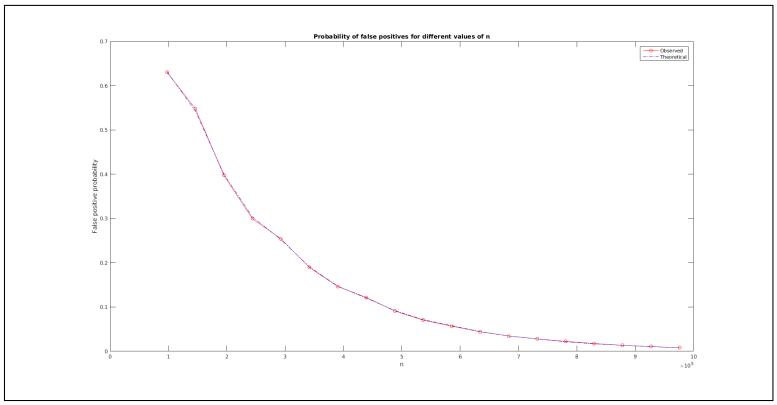

Figura 2.5: Resultados o teste do tamanho ideal do vetor de *bits* (`arraySize / n`)

Como se verifica no gráfico da Figura 2.5, à medida que o tamanho do *array* aumenta, a probabilidade de falsos positivos diminui, isto é, são inversamente proporcionais.

Os valores ideias são os presentes na Figura 2.6

2.1.6 Testes práticos do *Bloom-filter*

Foram desenvolvidos dois módulos semelhantes para um teste mais direcionado ao uso real de um *Bloom-filter*. Estão em dois ficheiros distintos, *test_big_BloomFilter.m* e *test_small_BloomFilter.m*.

O primeiro segue o modelo dos testes anteriores, em que se gera dois *sets* de um certo tamanho (neste caso, cem mil elementos) de *strings* aleatórias, cria-se uma instância de um *Bloom-filter* com uma probabilidade de falsos positivos de 0.0001%, adiciona-se um dos conjuntos ao filtro e verifica-se a existência de falsos positivos. Por fim, compara-se o resultado observado da probabilidade de falsos positivos com a que se definiu aquando a criação do filtro. O objetivo é que os resultados sejam o mais próximos possíveis, algo que se conseguiu atingir em ambos os testes.

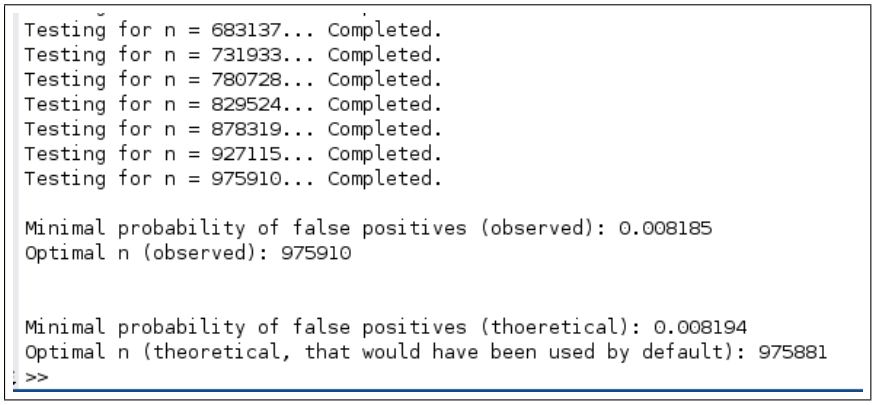

Figura 2.6: Resultados ideias do tamanho ideal do vetor de *bits* (`arraySize / n`)

No segundo teste (*small*), contrariamente ao primeiro em que se geram os conjuntos, definem-se conjuntos muito pequenos inicialmente e, depois, executa-se o mesmo processo que o anterior.

```
Generated strings with maximum random length: 50
Probability of false positive: 0.000100
Length of the set to add (m): 97661

97661 randomly generated strings added to the filter.

97661 strings that were previously added are probably in the set.
0 strings that were previously added are not in the set.

12 strings that were not added are probably in the set.
96860 strings that were not added are not in the set.
Probability of false positives (observed): 0.000124
```

Figura 2.7: Resultados da execução de *test_ big_ BloomFilter.m*

Capítulo 3

Locality-Sensitive Hashing

O *Locality-Sensitive Hashing* é particularmente útil quando se têm grandes conjuntos de documentos e se pretende achar semelhantes entre si, como, por exemplo, achar notícias semelhantes em diversos *websites*. No entanto, pequenos trechos dos documentos podem aparecer noutros não necessariamente na mesma ordem. Para além do mais, se tivermos *sets* na ordem dos milhões e milhares de milhão, são tantas as comparações que são necessárias fazer que excedem a memória disponível.

Portanto, para podermos comparar documentos, são necessários 3 passos:

Shingling Converter documentos para conjuntos.

Min-Hashing Converter grandes conjuntos para pequenas assinaturas (por norma, de inteiros), preservando a similariedade.

Locality-Sensitive Hashing Forcar-se apenas nos pares de assinaturas que possivelmente pertençam a documentos semelhantes e, mais tarde, testar realmente a sua similariedade.

O módulo de *LSH* está no ficheiro *LSH.m* e dispõe dos seguintes métodos:

shingleWords Aceita um *cell array* com o conteúdo de um documento e devolve *shingles* relativos ao mesmo.

signature Converte um vetor de *shingles* num vetor de assinaturas, através do processo de *min-hashing*.

candidates Devolve a lista de documentos candidatos a serem semelhantes a outros, consoante a matrix de assinaturas e o *threshold* passados como argumentos.

candidates_to Executa o mesmo processo que a função `candidates`, no entanto, em vez de aceitar apenas uma matriz de assinaturas, aceita também a assinatura de um documento, o qual pretendemos comparar com os restantes.

similars Verifica quais são os documentos realmente semelhantes e a sua Similariedade de Jaccard, consoante a matriz de candidatos, assinaturas e o *threshold* passados como argumentos.

similiars_to Tem a mesma função que `similars`, no entanto, aplica-se ao caso em que pretendemos comparar um documento com os restantes.

O único construtor do módulo aceita apenas um argumento, que é o erro esperado (`expectedError`). Este erro será utilizado para calcular o número `k` de *hash functions* necessárias, que é também o único atributo que o módulo possui, segundo a fórmula:

$$k = 1/expectedError^2$$

3.0.1 Shingling

Uma *k-shingle* de um documento é uma sequência de k *tokens* (palavras, caracteres, etc.) que aparecem no documento. Documentos que tenham muitas *shingles* em comum, possuem texto parecido, mesmo que o mesmo não apareça na mesma ordem.

A similariedade entre dois documentos $(D1, D2)$ pode ser calculada através da Similariedade de Jaccard:

$$sim(D1, D2) = \frac{|C1 \cap C2|}{|C1 \cup C2|}$$

sendo $C1$ e $C2$ os conjuntos de *shingles* de $D1$ e $D2$, respetivamente.

A Distância de Jaccard, por outro lado, pode ser dada por:

$$d(D1, D2) = 1 - sim(D1, D2)$$

Para gerar um *set* de *shingles* é utilizada a função `shingleWords`, que recebe como argumento um *cell array*, neste caso, de *strings* (considere-se o mesmo `Doc`). Para gerar esse conjunto, é necessário selecionar quais as sequências de palavras mais importantes de `Doc`. Para tal, é necessária a criação de um *set* de *stop words*, logo, foram guardadas num *cell array* `stop_words` as cem palavras mais comuns da língua inglesa. Quando o programa encontra uma *stop word*, ele guarda a sequência *stop_word next_word1 next_word2* num outro *cell array*, que será o retorno da função.

3.0.2 Min-Hashing

Para contornar as demoras que as comparações de excertos de textos provocariam, os *shingles* são convertidos em assinaturas de inteiros, no entanto, a sua similariedade deve ser conservada.

Para a criação das respetivas assinaturas é usada a função `signature`; começa-se por se criar um vetor em que o seu valor inicial é o valor máximo do tipo *double* (auxilia no cálculo dos valores mínimos) e, de seguida, para cada valor de `k` (número de *hash functions*), é gerado um vetor das assinaturas do *cell array* de *shingles* que é passado como argumento, dos quais é extraído o valor mínimo e guardado no *array* mais tarde devolvido pela função.

3.0.3 Locality-Sensitive Hashing

Por fim, depois de gerados os vetores de assinaturas dos documentos, basta compará-los e ver quais aqueles que mais se assemelham.

Primeiramente, começa-se por definir uma lista de candidatos possíveis (onde é provável a ocorrência de falsos positivos), para, por fim, verificar quais são mesmo os documentos que são realmente semelhantes (segundo um coeficiente de Similariedade - *threshold*).

A função `candidates`, definida no módulo `LSH`, serve para definir a lista de candidatos possíveis. Consideremos a matriz de assinaturas M, em que cada coluna corresponde à assinatura de um documento, isto é, o resultante do processo de *min-hashing* dos *shingles* obtidos.

Cria-se também um *cell array* onde serão armazenados os pares candidatos a serem semelhantes, numa estrutura em que a linha i corresponde ao documento i e em que cada coluna corresponde ao índice j de um documento possível candidato semelhante ao documento i.

De seguida, itera-se sobre cada uma das bandas que se extraem da matriz de assinaturas e compara-se com todas as outras bandas existentes (verica-se se a soma do número de assinaturas que as bandas têm em comum é igual ao número de *rows*).

Por fim, a função `similars` trata de finalizar o processo e devolver os documentos que são realmente semelhantes e o seu coeficiente de similariedade.

Itera-se sobre a matriz de candidatos e compara-se, uma a uma, se a Similariedade de Jaccard entre as assinaturas é maior que o *threshold* definido (usando a função `intersect`, que devolve os elementos em comum entre dois conjuntos). Se a condição se verifica, adiciona-se ao vetor de retorno os índices na matriz de assinaturas dos documentos em questão e o valor da sua similariedade.

3.0.4 Testes

Para os testes deste módulo, foram adaptados os *scripts* do Guião Prático-Laboratorial 07 da disciplina, mas, para além de ser usado o *dataset* de cem mil utilizadores da MovieLens, foi também utilizado o que contém um milhão de utilizadores.

A única diferença entre os testes é apenas o *dataset* utilizado, pois o processo tem de obrigatoriamente ser o mesmo, apenas se muda o tamanho do conjunto de dados.

Inicia-se o teste obtendo um vetor com todos os utilizadores a partir do *dataset*. De seguida, para cada utilizador, constrói-se o conjunto de *shingles* (neste caso, não é usada a função `shingleWords` do módulo `LSH`, visto que não é documento de texto, mas sim um vetor de de índices de filmes que cada utilizador avaliou) e respetivas assinaturas, usando a função `signature`.

Primeiro, vai ser calculada a Similariedade de Jaccard esperada (teórica), simplesmente iterando sobre o conjunto de assinaturas e, usando a fórmula

$$sim(D1, D2) = \frac{|C1 \cap C2|}{|C1 \cup C2|}$$

e as funções de Matlab `intersection` (devolve os elementos que dois conjuntos têm em comum) e `union` (devolve a união de dois conjuntos). Por fim, verifica-se, quais destes valores estão a acima do *threshold* assumido (nestes teste, assumiu-se 0.6). Os resultados mostram que existem apenas 3 documentos com uma similariedade acima dos 0.6, como se verifica na Figura 3.1.

328	788	0.6730
408	898	0.8387
489	587	0.6299

Figura 3.1: Resultados (valores esperados) do teste com 100 mil utilizadores

Finalmente, para calcular a Similariedade de Jaccard observada (experimental), gera-se a lista de candidatos usando a função `candidates` do módulo `LSH` e a matriz com os documentos similares usando a função `similars`, também do mesmo módulo.

A Figura 3.2 mostra que os resultados de ambos os testes foram os mesmos, havendo apenas uma pequeníssima variação dos valores de similariedade (à volta de 0.2).

328	788	0.6525
408	898	0.8125
489	587	0.6475

Figura 3.2: Resultados (valores observados) do teste com 100 mil utilizadores

Capítulo 4

SPAM Filter

SPAM Filter.

Capítulo 5

Conclusões

Referências

[1] *Hammond organ*, http://en.wikipedia.org/wiki/Hammond_organ, 2015.

[2] *Synthesizing hammond organ effecss*, http://www.soundonsound.com/sos/jan04/articles/synthsecrets.htm, 2015.

[3] *Synthesizing tonewheel organs*, http://www.soundonsound.com/sos/nov03/articles/synthsecrets.htm, 2015.