# 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

# Conteúdo

# Lista de Figuras

# Capítulo 1

# Introdução

Na área da informática, é, por exemplo, muitas vezes necessário saber se algo pertence a um certo conjunto de forma eficiente. A solução mais óbvia seria percorrer todo o conjunto e comparar, um a um, todos os elementos até encontrar aquele que se procurava inicialmente. No entanto, este método é deveras ineficiente especialmente se se estiver a trabalhar com conjuntos com milhões ou mais elementos. É aqui que surgem estruturas como *Bloom-filters*.

*Bloom-filters* usam geralmente várias funções de dispersão (ou *hashing*) para determinar a pertença de um dado elemento num conjunto (*set*). São muito utilizados em grandes conjuntos de dados e em diversas aplicações, como corretores ortográficos de computadores, *smartphones*, etc., análise textual, entre outras. Contudo, e sendo uma estrutura probabilística, acarreta um erro associado.

Outra técnica bastante relevante em situações onde é "impossível" guardar e trabalhar enormes conjuntos de dados é o de *MinHash*. Muitas vezes, a par com esta, aparece também o método de *Locality-sensitive hashing (LSH)*, com aplicações usuais em *clustering* de informação e procura de vizinhos próximos a um dado elemento (*nearest neighbor search*).

Neste trabalho, foi desenvolvido em MATLAB um *Bloom-filter*, um *MEX file* fazendo de interface para uma família de funções de *hash* escritas em C++ e ainda um módulo com capacidade de *nearest neighbor search* utilizando *MinHash* para gerar uma representação de diferentes documentos a serem tratados.

Neste contexto considere-se a filtragem de correio electrónico, algo largamente realizado por diversas empresas nos dias de hoje, tanto para o separar por temas como para evitar *spam* (i.e. lixo). Usando as técnicas acima, torna-se possível a implementação de um modelo probabilístico de categorização de correio eletrónico.

# Capítulo 2

# Bloom-filter

*Bloom-filters* são estruturas de dados probabilísticas que utilizam uma ou várias funções de *hash* para determinar a pertença de um dado elemento num *set*. Internamente, habitualmente utilizam um vetor de *bits*. Neste trabalho, utilizou-se um vetor de *bytes* visto que a linguagem em que o módulo foi escrito (MATLAB) não suporta o tipo *bit*. Para se tirar partido do espaço extra que se utilizou (*byte vs bit*) e visto ser-nos útil mais tarde na aplicação realizada, o módulo foi implementado como um *Counting filter*, onde, portanto, se mantém registo do número de elementos iguais que deverão ter sido adicionados ao filtro.

Desde que sejam utilizadas funções de *hash* eficientes, mesmo que se opere sobre um conjunto com milhões ou milhares de milhões de elementos, determinar a pertença ou não pertença de um valor nesse conjunto será sempre um processo significativamente mais rápido do que caso se iterasse sobre todo o *set*, elemento a elemento, à procura daquele em questão. Este processo depende do número e da qualidade das funções de *hash* utilizadas, bem como do tamanho do vetor interno.

Como estrutura probabilística que é, *Bloom-filters* têm um erro associado, neste caso manifestando-se apenas em falsos positivos, i.e., se um *Bloom-filter* indica que um elemento pertence a um conjunto, então provavelmente ele pertence mesmo. Por outro lado, se um *Bloom-filter* indica que um elemento não pertence a um conjunto, então definitivamente não pertence. Tendo isto em mente, habitualmente os *Bloom-filters* são dimensionados para um erro máximo admissível e para um determinado número de elementos que se tencione adicionar ao filtro.

Este erro depende do tamanho do vetor interno e do número de funções de *hash* a utilizar. Estas funções devem ser descorrelacionadas entre si e em número variável (em função dos parâmetros para os quais o filtro deve ser dimensionado). Uma solução para o número variável de funções que se deve utilizar é, em vez de se utilizarem `k` funções diferentes, utilizar-se uma família de funções que garanta `k` funções descorrelacionadas.

Neste trabalho, utilizou-se a família de funções de dispersão denominada *FarmHash*, desenvolvida pela Google, em que um dos argumentos, a *seed*, permite especificar cada uma das funções da família. A sua implementação é disponibilizada em C++ pela empresa que a desenvolveu, e, portanto, de forma a poder ser utilizada ao longo do projeto, foi escrito um ficheiro *MEX* que permite a interface entre código MATLAB e a função. Caso seja necessário, o ficheiro pode ser compilado executando (no directório onde os ficheiros *.h* e *.cpp* relativos à função se encontrem) `mex FarmHash.cpp`.

## 2.1 Atributos

Na implementação deste trabalho (ficheiro *BloomFilter.m*) desenvolveu-se uma estrutura 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 *hash* a utilizar.

**byteArray** Estrutura de dados interna do filtro, vetor de *bytes*.

**arraySize** Tamanho do vetor de *bytes*.

**amountAdded** Número total de elementos adicionados ao *array*.

**expectedMaxSize** Tamanho do conjunto que se pretende adicionar ao vetor.

## 2.2 Métodos

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, i.e., o erro admissível (`falsePositiveProbability`) e o tamanho do conjunto que se pretende adicionar ao vetor (`expectedMaxSize`).

Assumindo que a probabilidade de falsos positivos é $p$ e usando o tamanho do vetor de *bytes* $n$ e o tamanho do conjunto que queremos adicionar ao filtro $m$,

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

e

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

Para determinar o número $k$ ótimo de funções de *hash* que devem ser utilizadas, deduz-se

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

A partir das fórmulas acima encontradas, conclui-se também

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

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

**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 (apenas uma estimativa).

**Setters** Coleção de funções (utilizadas para testes ao módulo) que permitem a modificação dos atributos do *Bloom-filter*, caso o campo `debug` (argumento opcional passado ao construtor) esteja com o valor 1.

Existem ainda outros métodos que não foram aqui descriminados visto ou serem privados (e portanto não relevantes à interface do módulo) ou não terem sido testados adequadamente por não terem sido necessários ao longo do trabalho.

## 2.3 Testes

Para testar este módulo foram desenvolvidos diversos testes, de entre os quais uns para verificar qual seria o número ideal de funções de *hash* (`k`) a utilizar, outros para verificar o tamanho ideal do vetor de *bytes* (`n`), outros ainda para verificar que a família de funções de dispersão escolhida tinha um desempenho adequado.

### 2.3.1 Distribuição de funções de *hash*

Este módulo, *test_ hashFunction_ distribution.m*, tem como principal objetivo provar que várias funções de *hash* utilizadas têm uma distribuição uniforme para diversos valores de `k` (neste caso, irá variar entre 1 e 10).

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, terão tamanho fixo, caso contrário tamanho aleatório com máximo especificado pelo segundo argumento), e gerados diversos valores de *hash* para cada *string* e para a `seed` que estiver a ser avaliada em cada momento.

As distribuições para cada valor de `k` deverão ser o mais uniformes possíveis, de maneira a que quando utilizadas estas funções, estas não deem origem a valores concentrados numa pequena gama. Os resultados cumprem o pretendido, tal como evidenciado pelos histogramas da Figura 2.1.

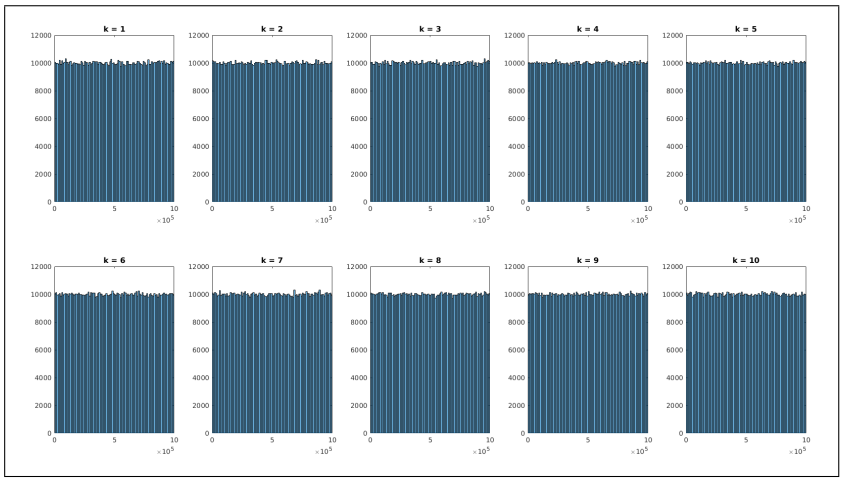


Figura 2.1: Resultados das distribuições das `k` funções de *hash*

### 2.3.2 Correlação das funções de *hash*

Para que se possam utilizar as várias funções de *hash* eficientemente (neste caso, isso significa sem aumento da taxa de falsos positivos), estas devem ser descorrelacionadas (i.e., o coeficiente de correlação respetivo deve ser 0). Neste trabalho considerar-se-ão valores de coeficiente de correlação inferiores a 0.01 como admissíveis, visto ser muito difícil que estes valores sejam verdadeiramente iguais a zero.

Este teste encontra-se no ficheiro *test_ hashFunction_ correlation.m* e, tal como noutros testes, gera-se um *set* de *strings* aleatórias e os respetivos valores de *hash* para diferentes valores de `k`, a variar entre 1 e 100. É depois criada uma matriz de dimensões `k` por `numTests` para guardar as *hashes* para os diferentes pares *k, string*. De seguida, calculam-se os coeficientes de correlação das funções duas a duas utilizando os resultados nas linhas da matriz anterior e a função `corrcoef` (calcula o coeficiente de correlação entre dois conjuntos, neste caso, duas linhas representativas de dois valores de `k` distintos).

Por fim, é gerado um gráfico recorrendo à função `surf` que permite ilustrar os valores de correlação resultantes. Verifica-se que para diferentes `k`s num par (i.e., comparando `k`s diferentes), os valores se situam geralmente bastante abaixo do erro admitido, havendo alguns picos onde efetivamente o coeficiente de correlação se aproxima deste valor. Conclui-se assim que as funções escolhidas são "suficientemente" descorrelacionadas, tal como se pretendia.

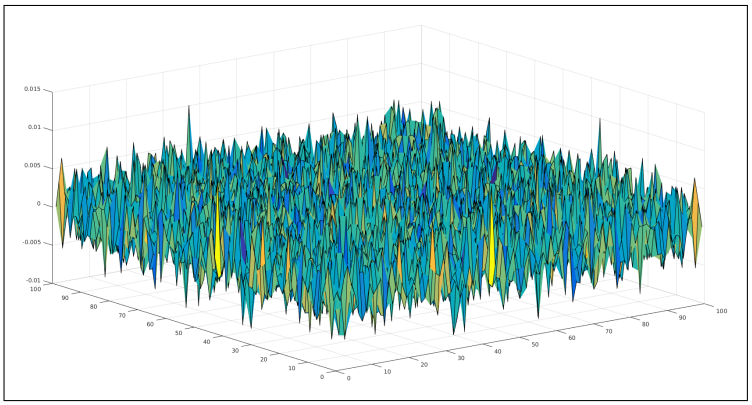


Figura 2.2: Resultados do teste de correlação de `k` funções de *hash*

### 2.3.3 Independência das funções de *hash*

Para além do teste de correlação, foi também efetuado um teste de independência das funções de *hash* (*test_ hashFunction_ independence.m*), adaptado de uma implementação disponibilizada pelo professor António Teixeira. Visto que independência implica descorrelação, este teste pode ser visto como uma versão mais forte do teste anterior. Analogamente a esse, estudou-se a independência das diferentes funções de *hash* duas a duas.

Primeiramente é criado um *set* de cem mil *strings* aleatórias. De seguida constrói-se uma matriz `n` (número de *strings* geradas) por `k` (número de funções de *hash* da família *FarmHash* que se pretende aplicar, neste trabalho, $k = 10$), onde cada elemento da matriz simboliza o *hash code* dum par (*string*, `k`). Seguidamente, é gerado um vetor de 10 elementos (`x`) linearmente espaçados entre 0 e `N` (valor ao qual são limitados os *hash codes*), e, para cada par de colunas, é criada uma matriz (`pmf`) de dimensão `length(x) - 1` por `length(x) - 1`. Guarda-se então em cada elemento da matriz o número de elementos das colunas de *hash codes* que se situam num mesmo intervalo do vetor `x` já criado.

Por fim, é calculada a matriz de probabilidade conjunta de duas colunas (*pmf - probability mass function*) dividindo cada elemento da matriz *pmf* anterior pelo número de *strings* que tinham sido criadas. A partir desta, são calculados os vetores de probabilidades individuais correspondentes às linhas

e colunas da matriz de probabilidades conjuntas. Finalmente, é calculado o resultado da multiplicação destes dois últimos vetores e verifica-se qual a diferença entre esses valores e os registados na matriz *pmf* (dois acontecimentos $A$ e $B$ são independentes se $P(A \cap B) = P(A) \times P(B)$).

Como os valores da variação dos resultados são mínimos (como se observa na Figura 2.3, onde todos os valores se encontram na casa dos $10^{-4}$), podemos considerar que a família de funções é independente dois a dois (pelo menos até ao valor de `k` considerado), o que permite a concluir que estas funções são descorrelacionadas, tal como verificado anteriormente.

Como trabalhar a dimensões superiores (3 a 3, 4 a 4, etc.) tornaria o processo mais complexo e demorado e, usualmente, a prova de independência é feita *pairwise*, 2 a 2, assume-se o resultado do teste como satisfatório.

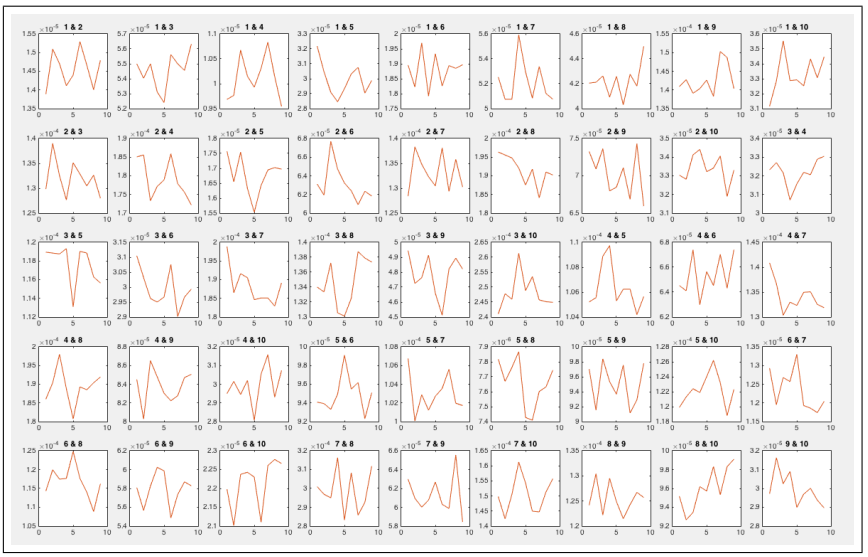

Figura 2.3: Resultados do teste de independência de `k` funções de *hash*

### 2.3.4 Número ideal de funções de *hash*

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

Começa-se por se criar uma instância da classe de *Bloom-filter* (módulo anteriormente desenvolvido), com o campo `debug` inicializado a 1, para que se possam utilizar as funções de atribuição de valores. Geram-se então dois conjuntos (um dos quais será adicionado ao filtro, outro não) distintos, com cem mil (valor que pode ser redefinido) *strings* cada um. *Strings* repetidas são removidas para que se obtenham verdadeiros *sets*, e portanto o tamanho dos dois conjuntos poderá (e deverá) ser sempre ligeiramente inferior ao definido inicialmente. De seguida, define-se o tamanho do vetor de *bytes*

do filtro como oito vezes o tamanho do conjunto de *strings* gerado que se pretende adicionar.

Por fim, itera-se sobre um vetor de `k` que se definira anteriormente (neste caso, é um vetor com valores de 1 a 15, incrementado unidade a unidade) e, para cada ciclo, define-se o `k` respetivo no objeto *Bloom-filter* que está a ser utilizado. Adicionam-se então os elementos do vetor de *strings* incialmente gerado e verifica-se se algum dos elementos do outro conjunto não adicionado de *strings* pertence ou não ao filtro. Caso pertença, é incrementado um contador, para que, no final do ciclo, possa ser calculada a probabilidade de falsos positivos.

Utiliza-se também a fórmula

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

para determinar a probabilidade de falsos positivos (teórica) para os diferentes valores de `k` considerados. Este seria o valor utilizado por omissão.

Usando os valores do teste acima, verificamos (Figura 2.4) que as diferenças entre o valor teórico e o observado são mínimas, o que significa não só que é válido utilizar a expressão teórica para definir o valor de `k` na instanciação de *Bloom-filters*, como também que as funções de *hash* usadas devolvem excelentes resultados (o que vem de acordo com os testes anteriores). Nas condições iniciais deste teste vem que o valor de `k` ideal obtido experimentalmente é 6, o que concorda com o valor que teria sido utilizado por definição aquando da instanciação do *Bloom-filter* através da dedução teórica.

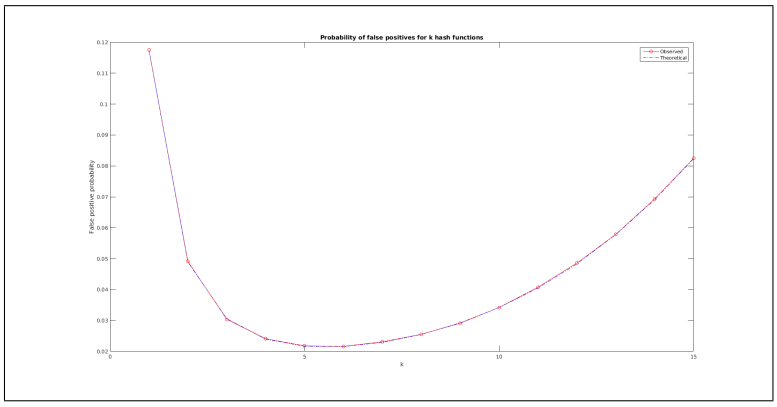


Figura 2.4: Resultados do teste do número ideal de funções de *hash* (`k`)

### 2.3.5 Tamanho ideal do vetor de *bytes*

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

Tal como no teste da Subseção 2.3.4, 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 do conjunto de *strings* a adicionar até 10 vezes esse valor, com um incremento de metade do tamanho do conjunto entre cada.

De seguida, itera-se sobre os valores deste último vetor, criando-se uma instância de um *Bloom-filter* com os valores de `n` (atributo `arraySize` do módulo) e `k` (que 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, de maneira semelhante ao que foi feito no teste anterior. Vai sendo também calculado, para cada valor de `n`, o valor teórico correspondente da probabilidade de falsos positivos e guardando estes valores num segundo vetor.

Como se verifica no gráfico da Figura 2.5, à medida que o tamanho do *array* aumenta, a probabilidade de falsos positivos diminui, i.e., são inversamente proporcionais, o que já era esperado visto que com um vetor maior haverá mais espaço para os possíveis *hash codes* gerados pela função de *hash*. No entanto, o mais importante com este teste é verificar que os valores experimentais são semelhantes aos teóricos, já que o importante será para um dado valor de erro admissível, dimensionar o *array* interno eficientemente (pois caso contrário ou se vai ocupar memória desnecessária, ou o número de falsos positivos real será acima do admissível). Como os valores obtidos (experimentais e teóricos) são bastante semelhantes, conclui-se que a dedução teórica que é utilizada no construtor do filtro permite um bom dimensionamento da estrutura de dados interna deste.

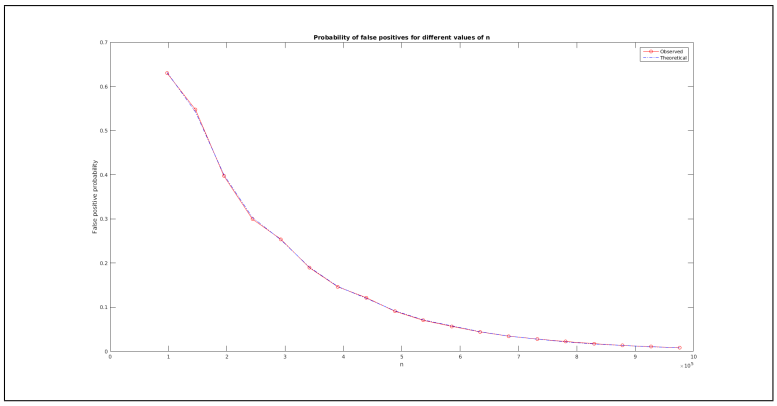


Figura 2.5: Resultados do teste do tamanho ideal do vetor de *bytes* (`arraySize` ou `n`)

Os valores ideias são os presentes na Figura 2.6.

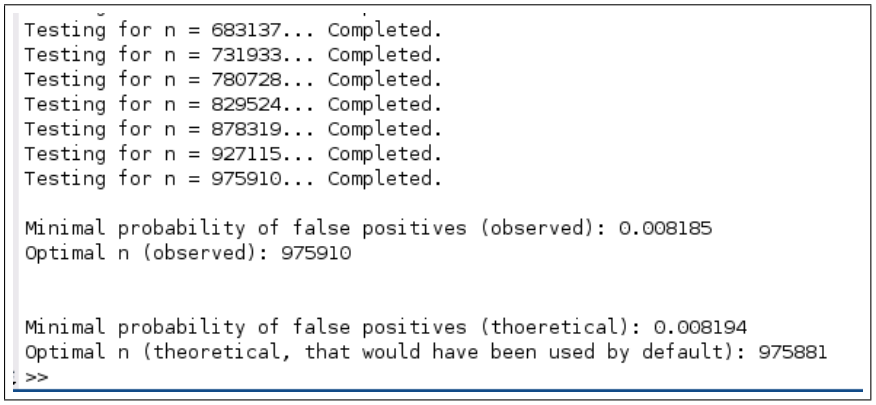


Figura 2.6: Resultados ideias do tamanho ideal do vetor de *bytes* (`arraySize` ou `n`)

### 2.3.6 Testes práticos do *Bloom-filter*

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

O primeiro (*big*) segue o modelo dos testes anteriores, em que se geram 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 ocorrência de falsos positivos com o outro. 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.

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 anteriormente.

```
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

*Locality-sensitive hashing (LSH)* é particularmente útil quando se têm, por exemplo, grandes conjuntos de documentos e se pretende encontrar aqueles semelhantes entre si, como, por exemplo, encontrar notícias semelhantes (em conteúdo) em diversos *websites*. Muitas das vezes, com conjuntos com tamanho na ordem dos milhões, são tantas as comparações necessárias a fazer e a memória necessária para guardar os documentos, que se torna inviável a utilização de métodos "clássicos", por exemplo, comparando os documentos no seu todo um a um.

Com o objetivo de implementar um módulo capaz de efetuar a procura pelos vizinhos mais próximos (*nearest neighbor search*), guardando representações de documentos mais facilmente comparáveis e que ocupam geralmente menos espaço, seguiram-se três passos:

**Shingling** Converter documentos para conjuntos de elementos (habitualmente pequenas *strings*) denominados *shingles*, o que permite uma melhor representação do conteúdo de um documento. Neste trabalho escolheu-se uma técnica de extração de *shingles* sugerida em [1], onde se procura por *stopping words* e, para cada palavra encontrada, forma-se um *shingle* constituído pela palavra encontrada e pelas duas seguintes.

**Minhashing** Converter grandes conjuntos (usualmente de *shingles*) para uma pequena assinatura (por norma, vetor de *hash codes*), preservando o conceito de similariedade entre diferentes documentos (neste caso, conjuntos de *shingles*).

**Locality-sensitive hashing** Reduzir o conjunto de elementos a candidatos de forma eficiente, testando mais tarde a sua "real" similariedade.

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

**shingleWords** Aceita um *cell array* com as palavras (*strings*) que constituem um documento e devolve um conjunto de *shingles* relativos ao mesmo, construído tendo em conta um conjunto de *stopping words*.

**signature** Converte um conjunto (mais precisamente *cell array*) de *shingles* numa assinatura (vetor de inteiros), recorrendo a um processo de *minhashing*.

**candidates** Devolve a lista de documentos candidatos a formarem conjuntos semelhantes, consoante a matriz de assinaturas e o *threshold* passados como argumentos.

**candidates_to** Executa um processo semelhante ao da função `candidates`, no entanto em vez de formar conjuntos de candidatos para uma dada matriz de assinaturas, forma uma lista de candidatos (representados numa matriz de assinaturas), face a um documento específico (representado pela sua assinatura e passada como argumento).

**similars** Verifica quais são os documentos de um conjunto de candidatos realmente semelhantes, consoante a matriz de candidatos, assinaturas e o *threshold* passados como argumentos. Esta comparação é feita utilizando as assinaturas dos documentos.

**similiars_to** Semelhante à função `similars`, no entanto, aplica-se ao caso em que pretendemos comparar os candidatos a similares entre um único documento e outro conjunto de documentos (sendo os documentos tal como habitualmente representados pelas suas assinaturas).

O construtor do módulo aceita apenas um argumento (`expectedError`), que é o erro esperado quando se calcula a similaridade entre dois conjuntos representados pelas suas assinaturas, obtidas segundo o método de *Minhash*, face à real similaridade de Jaccard entre esses conjuntos. Este valor é utilizado para calcular o tamanho `k` das assinaturas (e, portanto, o número de funções de dispersão necessárias). `k` é o único atributo que o módulo possui (não considerando o atributo `debug`, utilizado em *debugging*), e é calculado utilizando a fórmula

$$k = \frac{1}{expectedError^2}$$

Este resultado é obtido pelos limites de Chernoff (*Chernoff bounds*), concluindo-se que o erro esperado para o cálculo da similaridade de dois conjuntos segundo o método de *Minhash* é $\mathcal{O}\left(\frac{1}{\sqrt{k}}\right)$.

### 3.0.1 Shingling

Um *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 este não apareça na mesma ordem. Este facto é importante pois permite determinar a semelhança entre diferentes documentos tendo em conta a ordem segundo a qual diferentes palavras aparecem (considere-se que facilmente se escrevem dois textos com aproximadamente as mesmas palavras, mas em que, dada a ordem com que estas aparecem, o conteúdo destes documentos é bastante diferente).

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` do módulo escrito, que recebe como argumento um *cell array*, neste caso, de *strings* que contêm as palavras do documento (considere-se `Doc`) para o qual se quer gerar o conjunto de *shingles*. Para gerar esse conjunto, é necessário selecionar quais as sequências de palavras mais importantes de `Doc`. Para tal, e seguindo o algoritmo sugerido em [1], é necessária a criação de um *set* de *stop words*. Estas palavras devem ser palavras que apareçam com grande frequência na língua considerada e que geralmente aparecem próximas de palavras importantes no contexto de um documento. Assim, e para este efeito, foram guardadas num *cell array* as cem palavras mais comuns da língua inglesa. Quando o programa encontra uma *stopping word*, guarda a sequência *stop_ word next_ word1 next_ word2* num segundo *cell array*, que será o retorno da função.

### 3.0.2 Minhashing

Para contornar as demoras que as comparações de excertos de textos provocariam e a memória eventualmente necessária para os ter em RAM, os *shingles* são convertidos em assinaturas (vetores de inteiros). No entanto, a similariedade entre documentos deve manifestar-se também nestas assinaturas. Para isto utilizou-se uma técnica de *MinHash*, *Many Hash*, que consiste em considerar *k hash functions* ($k$ será também o tamanho da assiantura). Para cada $i$ entre 1 e `k`, calcular $\text{hash}_i$ de todos os elementos do conjunto

para o qual está a ser gerada a assiantura. O elemento $i$ da assinatura será o mínimo de todos os $\text{hash}_i$ gerados anteriormente.

Com este objetivo foi criada a função `signature`. Inicialmente é criado um vetor de tamanho `k`, tal como definido no construtor do módulo. De seguida, para cada *seed* entre 1 e `k` é gerado um vetor de *hash codes* para o *cell array* de *shingles* passado como argumento, e para esse valor de *seed*. É depois calculado o mínimo desse vetor, guardando-se o resultado na posição *seed* do vetor assinatura que será no final devolvido pela função.

### 3.0.3 Locality-sensitive hashing

Por fim, depois de geradas as assinaturas dos documentos, estas podem ser comparadas para se determinar os documentos que mais se assemelham.

Para isso, inicialmente começa-se por se definir uma lista de candidatos possíveis (onde é "provável" a ocorrência de falsos positivos), para, mais tarde, se verificar quais aqueles que são realmente semelhantes. É por isso necessário definir um valor de *threshold*, que deverá indicar quão semelhantes dois documentos devem ser para poderem ser considerados similares, no contexto do problema considerado.

A função `candidates` definida no módulo `LSH` serve para calcular a lista de candidatos possíveis.

Consideremos a matriz de assinaturas $S$, onde cada coluna corresponde à assinatura de um documento (i.e., o resultado do processo de *minhashing* dos *shingles* obtidos para esse documento).

Esta matriz é dividida em `b` bandas (*bands*) de `r` linhas (*rows*) cada. Então, para cada banda verificam-se quais as colunas (da banda) iguais - estes elementos serão considerados candidatos e serão, portanto, adicionados ao *cell array* de candidados que será devolvido.

Tendo em conta que

$$br = k,\ k \text{ tamanho de cada assinatura}$$

e

$$t \approx \left(\frac{1}{b}\right)^{\frac{1}{r}}$$

onde $t$ é o *threshold* escolhido, pode aproximar-se `b` ou `r` e em função do valor calculado, obter o outro.

A estrutura da matriz de retorno (*cell array* de candidatos) é tal que cada linha $i$ corresponde ao documento $i$ e cada elemento dessa linha corresponde ao documento $j$, significando que esse par *(Docsi, Docsj)* constitui um par de candidatos (ou seja, o documento $i$ poderá ser semelhante ao documento $j$). $i$ e $j$ são os índices da matriz de assianturas recebida como argumento para os quais foi detetada alguma semelhança.

No final da execução da função, deverá obter-se um *cell array* com todos os pares de candidatos a serem semelhantes. É de notar que, na abordagem seguida, tenta-se eliminar entradas irrelevantes, ou seja, se a matriz de candidatos indica que poderá haver semelhança entre *Docsi* e *Docsj*, então não deverá indicar o equivalente *Docsj* "semelhante a" *Docsi*.

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

Para isso, itera-se sobre a matriz de candidatos e compara-se, um a um, se a similariedade de Jaccard entre as assinaturas é maior ou igual ao *threshold* definido (usando a função `intersect`, que devolve os elementos em comum entre dois conjuntos). Se a condição se verificar, adiciona-se ao vetor de retorno os índices na matriz de assinaturas dos documentos em questão e o valor da sua similariedade, segundo um modelo *DocN DocZ sim(DocN, DocZ)*.

### 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 obrigatoriamente de ser o mesmo, apenas variando 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` dado que cada conjunto de filmes define por si só cada utilizador. Após terem sido construídos estes vetores, os seus elementos são convertidos para *strings* para que possam ser utilizados diretamente pela função encarregue de gerar assinaturas de conjuntos, sendo este passo realizado utilizando a função `signature`.

Inicialmente é calculada a similariedade de Jaccard esperada (teórica), simplesmente iterando sobre o conjunto de assinaturas e utilizando a expressão

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

e as funções de MATLAB `intersect` (devolve os elementos que dois conjuntos têm em comum) e `union` (devolve a união de dois conjuntos) (é ainda utilizada a função `length` pois apenas importa o tamanho dos conjuntos que as duas funções anteriores retornam). Por fim, verifica-se quais destes valores

acima do *threshold* definido (nestes testes, este valor foi de 0.6). Os resultados mostram que existem apenas 3 documentos com 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 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 pequena variação dos valores de similariedade (à volta de 0.02). Note-se que a instância de *LSH* utilizada foi inicializada para um erro de 0.05, e portanto como $0.02 < 0.05$ os resultados estão de acordo com o esperado e dentro dos limites admissíveis.

| | | |
|---|---|---|
| 328 | 788 | 0.6525 |
| 408 | 898 | 0.8125 |
| 489 | 587 | 0.6475 |

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

No teste com um milhão de utilizadores, usou-se também um *threshold* de 0.6, no entanto, na construção da matriz de candidatos (utilizando a função `candidates`), utilizou-se um valor de 0.55, com o objetivo de "apanhar" valores muito próximos do limiar considerado, já que foi definido um erro admissível de 0.05.

Depois de observados os resultados do teste (Figura 3.3), verificamos que estes são bastante satisfatórios e de acordo com o esperado. Devido ao erro assumido de 0.05, alguns resultados muito próximos do limiar podem ou não ser apresentados. Como verificamos na Figura 3.3, a similariedade de duas assinaturas de listas de filmes de utilizadores é muito próxima do limiar definido (0.6), no entanto visto ser inferior a este não foi incluida na lista de pares similares calculada. O valor teórico para a similaridade deste par de utilizadores é na verdade 0.6012 (quando consideradas as listas de filmes de cada um, e não as suas assinaturas), e portanto este par deveria ter sido dado como par similar. Ainda assim, é um resultado mais uma vez

admissível visto ter-se utilizado um erro de 0.05 para a precisão do cálculo da similaridade de dois elementos através do uso das suas assinaturas.

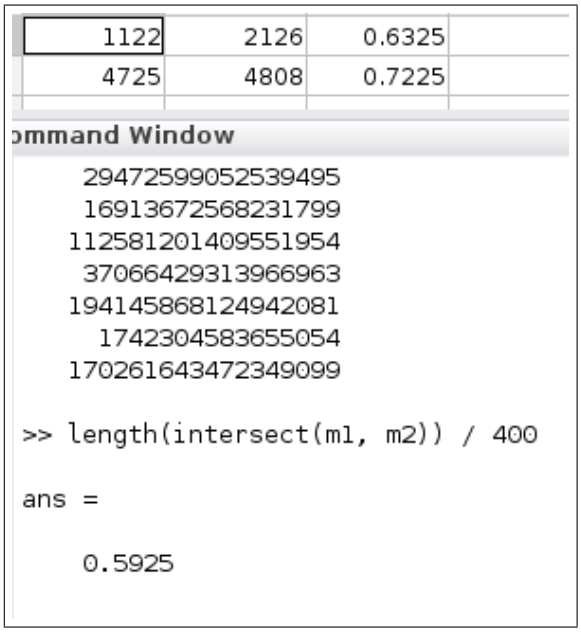


Figura 3.3: Resultados (valores observados) do teste com 1 milhão de utilizadores

# Capítulo 4

# SPAM Filter

A filtragem de *email* é uma maneira de o organizar segundo um dado critério, por exemplo detetando *spam* evitando assim ter de ler mensagens sem interesse. Neste trabalho, considera-se *spam* como sendo mensagens que, para um dado utilizador, não têm interesse (ou seja, o que é *spam* para uma pessoa poderá não o ser para outra), e portanto poderá ser visto como apenas mais uma forma de categorizar *emails*.

Esta funcionalidade é geralmente oferecida por empresas como a Google ou Microsoft, que habitualmente utilizam algoritmos que detetam, por exemplo, picos de atividade de determinados endereços (especialmente se até então desconhecidos), bem como *feedback* dos seus utilizadores que marcam ou não como *spam* os *emails* que recebem. A par disto, muitas vezes as mensagens que levantam mais dúvidas são verificadas por equipas com essa tarefa.

Outra forma de filtrar *spam* de uma caixa de entrada poderá ser comparar o conteúdo de *emails* previamente consideramos como tal e ver se são semelhantes à mensagem que se acabou de receber - se sim, colocar a mensagem na pasta certa (que pode ser diretamente o caixote do lixo), se não, então nada fazer. Outro método é verificar se o endereço de *email* da nova mensagem já enviara algum *email* de *spam* anteriormente.

Neste trabalho pretende-se implementar um simples filtro de *spam* (ou qualquer outro tipo de mensagens) de acordo com o exposto no último paragrafo, utilizando para isso um *Bloom-filter* (de forma a filtrar facilmente por endereço) e técnicas de *Locality-sensitive hashing* (para se filtrar por conteúdo).

O *Bloom-filter* servirá como armazenamento de endereços de *email* que previamente tinham enviado lixo, constituindo assim uma *blacklist* de mensagens que deverão ser imediatamente filtradas. Já *Locality-sensitive hashing* servirá para comparar o conteúdo de novas mensagens com antigas mensagens marcadas como *spam*, permitindo assim inferir novas mensagens como sendo de interesse ou não. Algumas das principais vantagens desta abor-

dagem são a facilidade da sua implementação, bem como a eficiência de filtragem que demonstra não só em termos de velocidade de execução, mas também permitindo, por exemplo, que cada utilizador tenha a sua própria lista de *emails* sem interesse. Visto ser um modelo que utiliza técnicas probabilísticas, deverá também ser altamente escalável.

Os passos necessários para a filtragem segundo este modelo serão:

**Aprendizagem**

Esta fase deverá ser executada uma vez. Aqui o programa deve aprender a que se assemelha *spam*, permitindo mais tarde filtrar mensagens semelhantes. Para isso, deverão ser fornecidos ao programa *emails* considerados como *spam*, e o programa em troca deve não só armazenar os remetentes destas mensagens num *Bloom-filter* para referência futura, como também gerar e guardar uma assinatura segundo a técnica de *Minhash* para o corpo da mensagem. Este último passo é executado conforme indicado anteriormente no que toca à criação de um conjunto de *shingles* para um documento, bem como à posterior geração da assinatura desse conjunto. Estas assinaturas vão sendo guardadas numa matriz onde cada coluna é referente à assinatura de um *email* de *spam*.

**Verificação 1**

Neste momento, o modelo utilizado já conhece o que é *spam*, e portanto está apto a filtrar este tipo de mensagens. Nesta primeira fase de filtragem, deverá ser verificado se o endereço do remetente é ou não um endereço *blacklisted*. Note-se que, visto que o *Bloom-filter* deste trabalho foi implementado como um *Counting filter*, poderá ser mais proveitoso não apenas verificar se um endereço pertence ou não ao filtro, mas antes verificar se a contagem para esse endereço é superior a um determinado valor. Neste sentido foi assumido que um remetente tem de enviar pelo menos dois *emails* considerados *spam* para o seu endereço ser filtrado.

**Verificação 2**

Caso a mensagem passe na verificação acima, então passa-se a uma fase onde o conteúdo da mensagem é analisado contra o conhecimento que o modelo tem do que é *spam*. Também esta verificação é dividida em vários passos:

1. Começa-se por se gerar o conjunto de *shingles* referentes ao *email*. Caso este forme um conjunto vazio, então imediatamente conclui-se que a mensagem é *spam*, já que nenhuma informação relevante foi obtida. Caso contrário, calcula-se a assinatura do *email* e avança-se para a fase seguinte.
   A construção do conjunto de *shingles* e da assinatura do *email* é conforme o já referido anteriormente.

2. Conhecendo a assinatura da mensagem e possuindo a matriz de assinaturas de *emails* de *spam*, calcula-se a lista de candidatos da matriz de assinaturas a serem semelhantes à assinatura da mensagem a avaliar atualmente. Isto é possível recorrendo ao método `candidates_to` do módulo *LSH*, que funciona tal como a função homóloga `candidates`, com a diferença de que a primeira calcula os candidatos face a um determinado documento (e não todos os conjuntos de candidatos possíveis como a segunda).
   Caso seja retornado um conjunto vazio, então conclui-se que a mensagem atual não se assemelha a *spam* e portanto a verificação termina indicando que a mensagem é de interesse. Caso contrário, avança-se para a fase seguinte.
3. Por fim, a partir da lista de candidatos gerada na fase anterior, calculam-se as verdadeiras semelhanças entre as diferentes assinaturas dos candidatos, a partir da função `similars_to` que faz as vezes da função `similars`, com a diferença de que a primeira é também específica para calcular assinaturas similares a uma só assinatura, ou contrário da segunda que calcula os pares similares para todos os elementos da matriz de candidatos. Caso seja devolvida uma matriz vazia, o passo anterior apenas devolveu falsos positivos e portanto a mensagem não é considerada *spam*. Caso contrário, o conteúdo da mensagem assemelhou-se a algum outro doutra mensagem que tinha sido aprendida (como *spam*), e portanto conclui-se que esta deve também ser uma mensagem sem interesse.

Note-se que nos dois pontos da descrição anterior, as duas funções utilizadas utilizam um valor de *threshold*, passado como argumento, que deve ser definido tendo em conta dois fatores: um valor muito elevado garantirá que muito poucas mensagens que não são *spam* sejam detetadas como tal, por outro lado, poderá estar-se a não filtrar mais mensagens que são efetivamente *spam*. O contrário acontece para valores muito baixos de *threshold*, onde quase todos os *emails* de *spam* serão filtrados, mas também mais mensagens com interesse podem ser filtradas.

Assim, primeiro é necessário que o programa "aprenda" que *emails* deve filtrar. Para tal, foi utilizado um *dataset* do grupo *Enron*[1]. A função `create_generic` permite a aprendizagem, aceitando um diretório onde vai procurar pelos ficheiros da extensão que também lhe é passada como argumento, para guardar num ficheiro (o caminho também é passado como argumento) as estruturas *Bloom-filter* com os endereços de *email* que enviaram *spam* adicionados, as assinaturas do conteúdo dos mesmos e o módulo de *LSH* usado (para salvaguardar o número de funções de *hash* utilizadas).

[1] `http://csmining.org/index.php/enron-spam-datasets.html`

Por outro lado, a função `test_generic` testa o filtro de *spam* com base no ficheiro previamente guardado. Os dados desse mesmo ficheiro são carregados para o programa e todos os *emails* contidos na pasta que lhe é passada como argumento são analisados.

É de notar que o método descrito acima para filtragem de *spam* pode ser facilmente utilizado para filtrar por tema, permitindo, por exemplo, categorizar *emails* com três *labels*, *Primary*, *Social* e *Promotions* à semelhança do que o *Gmail* da Google faz. Basta para isso fornecer um conjunto de aprendizagem adequado. O programa pode também aprender com a experiência a melhorar os seus resultados, bastando para isso que, quando falha a filtragem de uma mensagem e o utilizador o assinale, seja adicionado o endereço do remetente ao filtro de *Bloom* em uso e a assinatura da mensagem à matriz de assinaturas que já possui.

## 4.1 Teste

O teste levados a cabo nesta fase foi o meno claro de todo o projeto, dada a falta de dados para testar eficientemente o programa.

Para este teste foi utilizado o *dataset* do grupo *Enron*, que continha cerca de 17 mil *emails* de *spam* e 16 mil *non-spam* (*ham*). Antes de se executarem os testes, deve-se correr o programa de aprendizagem da aplicação para este *dataset* (*enron_ training.m*).

No ficheiro *test_ thresold.m* foram executados vários testes de deteção de *spam* para diferentes valores de *threshold* (entre 0.1 e 1, com espaçamento de 0.1 entre eles). Conclui-se que o melhor compromisso entre falsos positivos e falsos negativos é para um valor de *threshold* de 0.1, o que assegura uma taxa de deteção de *spam* de 83% e uma de deteção de *non-spam* (portanto falsos positivos) de 3%, dos quais mais de 2% se devem a mensagens a partir das quais não foi possível formar quaisquer *shingles*.

O teste realizado com o valor de *threshold* encontrado no teste anterior encontra-se no ficheiro *test_ enron.m* que, consoante os resultados obtidos da aprendizagem, chama a função `test_generic` que devolverá as mensagens consideradas como *spam* e o número de *emails* analisados. Os resultados são os da Figura 4.1 e da Figura 4.2.

Os resultados da análise a *emails* com *spam* resultou numa percentagem de deteção de "lixo eletrónico" de cerca de 83%, o que se pode explicar por se ter tido de separar o *dataset* em dois conjuntos (um para aprendizagem e outro para testes); isto leva a que a base de aprendizagem não seja tão grande e que alguns dos *emails* com conteúdo de *spam* não sejam detetados.

Os resultados da filtragem de *emails ham* resultaram numa deteção de 3% de *spam*. No entanto, como se verifica, em apenas uma das mensagens o programa conseguiu gerar *shingles* (depois de analisar esta mensagem em concreto, concluiu-se que efetivamente esta se assemelhava a *spam*). As

```
Got a similarity of 1.000000 TO A SPAM email. Filtered.
Got 5142 of 6675 SPAM messages. P = 0.770337
No shingles could be made. Filtered.
Got 8 candidates. Analyzing... Done.
This was considered NOT TO BE SPAM. Skipped.
No shingles could be made. Filtered.
```

Figura 4.1: Resultados do teste ao *dataset* do grupo *Enron*, fornecendo apenas mensagens de *spam*

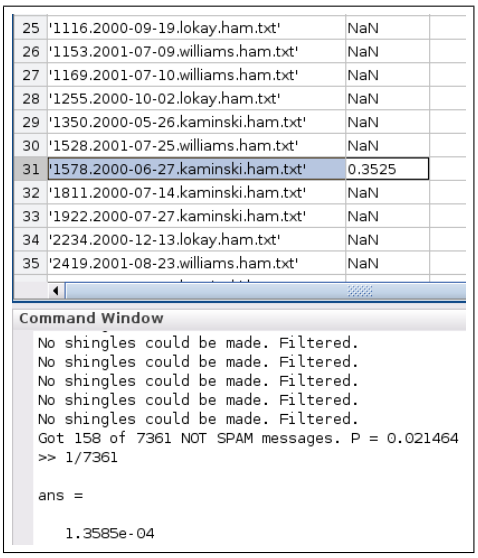


Figura 4.2: Resultados do teste ao *dataset* do grupo *Enron*, fornecendo apenas mensagens de *non-spam*

restantes foram consideradas "lixo" pois não foi possível gerar *shingles* das mesmas, provavelmente porque se tratavam de *emails* com agradecimentos ou pequenas expressões, como por exemplo, "Ok", "Thank you", "Done" como foi verificado após algumas das mensagens terem sido verificadas à mão.

## 4.2 Programa *demo*

Foi desenvolvido ainda um programa *demo* (*proof_of_concept.m*) que pretende funcionar como *proof of concept* do modelo de filtragem apresentado. Para isso, foram testados *emails* sem interesse que recorrentemente são recebidos pelos autores deste projeto. Com um *use case* específico pretende-se mostrar como funciona efetivamente o programa.

Assim, começou-se por se juntar um conjunto de cerca de 40 *emails* que

são, por nós, considerados como lixo. No entanto, são mensagens de fontes fidedignas e possivelmente com interesse para outras pessoas e, portanto, não são filtradas por definição por serviços de *email* como o *Gmail*. Guardou-se depois um pequeno número de *emails* que são consideradas com interesse e verificou-se se alguma destas mensagens era filtrada como *spam* pelo programa (dando origem a um falso positivo). Posto isto e para efeitos demonstrativos, o programa entra num *loop* onde permite a verificação de outras mensagens sem que estas tenham de ser previamente pré-processadas. O resultado é o ilustrado na Figura 4.3.

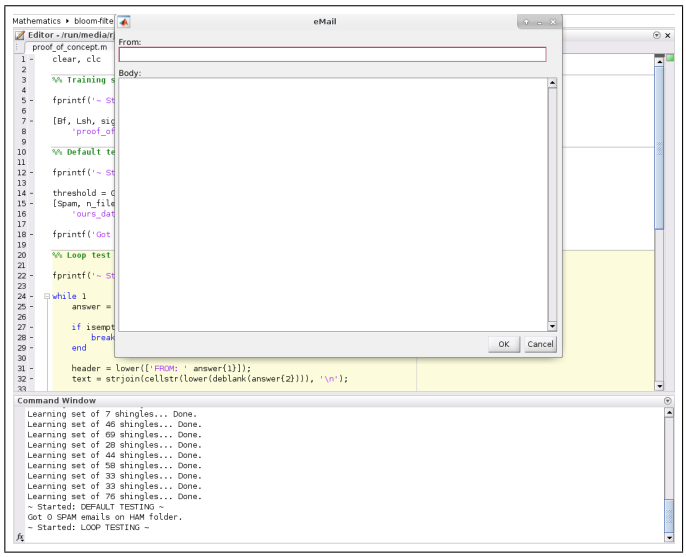


Figura 4.3: Resultados da execução do programa *demo* + janela para teste de novos *emails*

Neste teste conseguiu-se filtrar as mensagens sem interesse, ao mesmo tempo que as relevantes não eram afetadas.

# Capítulo 5

# Conclusões

Tendo em conta a eficiência demonstrada por filtros de *Bloom*, conclui-se que estes são efetivamente ótimas estruturas quando se pretende confirmar a existência de um elemento num conjunto muito grande e se admite um pequeno erro (de falsos positivos); por outro lado, e face aos resultados que apresenta, *Locality-sensitive hashing* (aliado a técnicas de *shingling* e *minhashing*) poderá ser uma ótima solução em algoritmos de *nearest-neighbour search* e *clustering*. Estas conclusões são baseados nos resultados bastante satisfatórios de testes relativos ao módulo de *Bloom-filter* e de *Locality-sensitive hashing*, resultando em desvios mínimos em relação aos valores teóricos.

Juntando estes dois métodos, foi possível criar um filtro de *spam* (ou de outros conteúdos) em que se obtiveram resultados satisfatórios tendo em conta os *datasets* utilizados.

Por fim, é de notar que qualquer modelo probabilístico é isso mesmo, apenas probabilístico, e assim sendo por mais eficiente que um filtro de *Bloom*, por exemplo, possa ser, só faz sentido ser utilizado em situações onde é admissível cometer algum erro. Isto pode ser um factor limitante em algumas situações, por outro lado, e sendo o mundo um lugar probabilístico a, vasta maioria dos casos monstra-nos que é possível trabalhar com alguma margem para erro, desde que esta seja suficientemente pequena. Assim, adotar aproximações probabilísticas a vários problemas pode contribuir para dar origem a soluções não só mais eficientes como até mais simples. E com o crescente aumento da informação disponível no mundo, cada vez mais soluções destas deverão ser precisas.

# Referências

[1] J. U. Jure Leskovec Anand Rajaraman, *Mining of Massive Datasets*.

[2] *Powerpoints de mpei*, 2015.