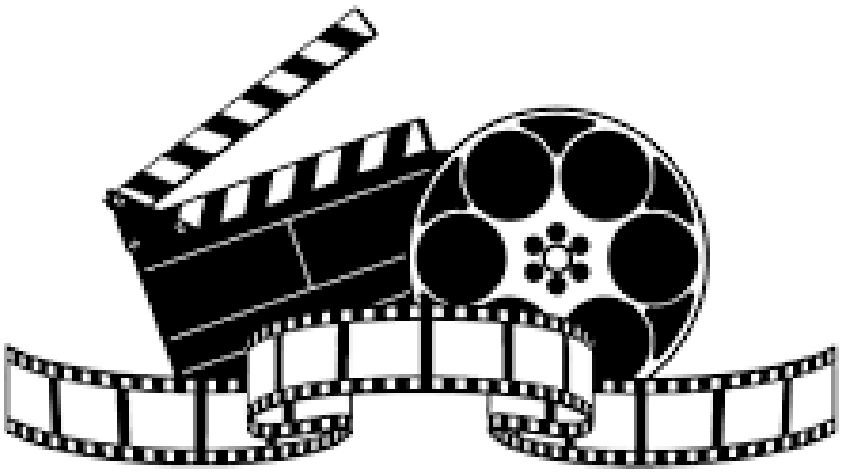
universidade de aveiro deti departamento de electrónica, telecomunicações e informática

Relatório Trabalho - Turma P01



Professor: Carlos Alberto da Costa Bastos

Trabalho realizado por:

André Oliveira, nº 107637

Vasco Faria, nº 107323

INTRODUÇÃO

No âmbito da disciplina de MPEI (Métodos Probabilísticos para Engenharia Informática), foi-nos apresentado este trabalho prático sobre algoritmos probabilísticos. O objetivo deste é o desenvolvimento de uma aplicação em Matlab, com algumas funcionalidades de um sistema online de disponibilização de filmes.

A aplicação consiste em pedir um número de utilizador (id) e através deste, podemos executar algumas tarefas que estarão no menu do programa como:

1-Listar os filmes do utilizador em questão

2- Listar filmes avaliados pelo utilizador mais "similar" ao utilizador atual

3-Listar filmes que, pelo título, sejam mais similares ao título que o utilizador introduz. Esta opção não depende do utilizador dado inicialmente.

4-Listar 3 filmes mais similares com o filme que o utilizador escolher.

5-Listar filmes de ano dado pelo utilizador

Ao longo do relatório serão apresentadas todas as etapas e respetivas descrições sobre a metodologia usada para o desenvolvimento deste trabalho.

MENU

Para elaboração do menu com todas as opções, utilizamos a função *menu()* disponibilizada já pelo *Matlab*. Esta função exibe uma caixa de diálogo, de múltipla escolha contendo o texto na mensagem. Cada uma das opções aparece como um botão. A função retorna o índice do botão selecionado (1 a 6) ou 0 se o utilizador clicar no botão "Fechar da janela".

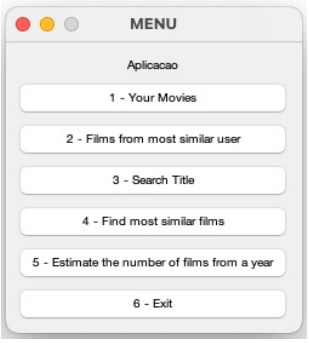


Figura 1 | Caixa do menu

OPÇÕES IMPLEMENTADAS

OPÇÃO 1

Na opção 1 é pedido para listar os filmes de um certo utilizador.

Para isso, começamos por carregar as informações dos ficheiros “u.data” e “films.txt”, para as variáveis *udata* e *dic* (usando *cell array*), respetivamente.

O ficheiro “u.data”, tem na primeira coluna os *ID’s* dos diferentes utilizadores, na segunda coluna os *ID’s* dos filmes que os respetivos utilizadores da mesma linha viram, na terceira coluna a avaliação que os utilizadores deram ao respetivo filme da linha e na quarta coluna o *timestamp* no momento da avaliação do filme, pelo utilizador.

Começámos também por criar uma matriz apenas com a primeira e segunda coluna da variável *udata*. Através da primeira coluna desta, criámos um *array* (*users*) com os *ID’s* dos diferentes utilizadores.

De seguida, como o número de filmes vistos por cada utilizador não é igual para todos, criámos um *cell array* com uma coluna e número de linhas igual ao número de utilizadores diferentes. Cada linha **n** contém um *cell array* com os *ID*’s dos filmes que o utilizador com o *ID* **n** já viu. Para tal, fizemos um *for loop* a percorres os *ID’s* de todos os utilizadores, em que inicialmente descobrimos os índices do *array users*, onde o valor do *for loop* (*n*) era igual aos valores armazenados nesse *array*.

```
function films = getFilms(users,data)
    numUsers = length(users);
    films = cell(numUsers,1);

    for n = 1:numUsers
        ind = find(data(:,1) == users(n));
        films{n} = [films{n} data(ind,2)];
    end
end
```

Figura 2 | Função criadora de cell array com filmes visualizados por cada utilizador

Por fim, temos todos os filmes que cada utilizador viu, logo basta ir ao índice deste utilizador (*ID*), no *cell array films* e imprimir a informação dessa célula.

```
case 1
    watchedFilmes = films{current_user};
    numWatchedFilmes = height(watchedFilmes);

    fprintf("\n---------- MY FILMS ----------\n\n");

    for i= 1:numWatchedFilmes
        idFilm = watchedFilmes(i);
        nameFilm = dic(idFilm,1);
        fprintf("(ID: %d) %s\n", idFilm,nameFilm{1});
    end

    fprintf("\n------------------------------\n\n");
```

Figura 3 | Código para a execução da primeira opção

OPÇÃO 2

Na opção 2, é pedido para listar os filmes avaliados pelo utilizador mais "similar" ao utilizador atual.

Para isso, é necessário encontrar o *ID* do utilizador mais parecido com o utilizador atual com base nos filmes que ele viu e depois imprimir os filmes que o utilizador mais parecido viu. Para resolver isto, começámos por calcular o *minhash* dos utilizadores para 100 funções de dispersão, usando como base a função *DJB31MA* ajustando apenas os valores usados.

```
films = getFilms(users,data);
numHash = 100;
matrizMinHashUsers = minHashUsers(users,numHash,films);
distancesUsers = getDistancesMinHashUsers(numUsers,matrizMinHashUsers,numHash);
```

Figura 4 | Variáveis criadas

Para calcular os *MinHash*, começamos por inicializar a matriz *matrizMinHashUsers* com número de linhas igual ao número de utilizadores e o número de colunas igual ao número de funções de dispersão usadas (usou-se o valor infinito como *default*, uma vez que no preenchimento desta *HashTable* vão sendo escolhidos os menores valores, por isso, o objetivo é começar com os maiores valores possíveis).

Numa fase seguinte, num *for loop* é escolhido um conjunto de filmes associado a um utilizador e, dentro doutro *for loop* que itera sobre os títulos dos filmes calcula-se a chave que corresponde a cada carater do nome do filme. Com um último *for loop*, percorremos o número de funções de dispersão usadas e que adicionamos à chave o número de iteração. Dentro deste último *for loop*, é armazenado no vetor **h**, o valor correspondente à mapeação desta informação através de *MinHash*. No final deste *for loop* esse valor é aramazenado na matriz *matrizMinHashUsers*, apenas se for menor do que o lá guardado anteriormente.

```
function matrizMinHashUsers = minHashUsers(users,numHash,films)
    numUsers = length(users);
    matrizMinHashUsers = inf(numUsers, numHash);

    x = waitbar(0,'A calcular minHashUsers()...');
    for k= 1 : numUsers
        waitbar(k/numUsers,x);
        filmsUser = films{k};
        for j = 1:length(filmsUser)
            chave = char(filmsUser(j));
            for i = 1:numHash
                chave = [chave num2str(i)];
                h(i) = DJB31MA(chave, 127);
            end
            matrizMinHashUsers(k, :) = min([matrizMinHashUsers(k, :); h]);
        end
    end
    delete(x);
end
```

Figura 5 | Construção da matriz com minHashs

Por sua vez, calculamos as distâncias de *Jaccard entre todos os utilizadores e guardamos na variável distancesUsers.* Para este efeito, usamos um *for loop* que começa em 1 e dentro dele, outro *for loop* que está sempre uma unidade à frente do anterior. Também é inicializada uma matriz quadrada com a dimensão do número total de utilizadores. A distância de *Jaccard* entre dois utilizadores é calculada, somando a quantidade de linhas da matriz *matrizMinHashUsers* correspondentes a cada utilizador que são iguais e, por fim, dividindo pelo número de funções de dispersão usadas.

```
function distances = getDistancesMinHashUsers(numUsers,matrizMinHash,numHash)
    distances = zeros(numUsers,numUsers);
    for n1= 1:numUsers
        for n2= n1+1:numUsers
            distances(n1,n2) = sum(matrizMinHash(n1,:)==matrizMinHash(n2,:))/numHash;
        end
    end
end
```

Figura 6 | Cálculo das distâncias entre os utilizadores

Com as distâncias de *Jaccard* já calculadas, através de um *for loop*, percorremos na variável *distancesUser*, as distâncias entre o utilizador atual e os outros todos, achando o mínimo dessas distâncias, pois pretendemos os filmes do utilizador mais similar.

Por fim, é usado o mesmo método usado na opção 1 para imprimir os filmes do utilizador mais similar.

```
case 2
    min = 1;
    u2 = 0;
    for u2 = current_user+1:numUsers
        if distancesUsers(current_user,u2) < min
            id = u2;
            min = distancesUsers(current_user,u2);
        end
    end

    watchedFilmes = films{u2};
    numWatchedFilmes = height(watchedFilmes);

    fprintf("\n---------- WATCHED FILMS FOR MOST SIMILAR USER ----------\n\n"

    for i= 1:numWatchedFilmes
        idFilm = watchedFilmes(i);
        nameFilm = dic(idFilm,1);
        fprintf("(ID: %d) %s\n", idFilm,nameFilm{1});
    end

    fprintf("\n--------------------------------------------------------\n\n"
```

Figura 7 | Código para a execução da segunda opção

OPÇÃO 3

Na opção 3, à semelhança do que fizemos na opção 2, começamos por calcular os *MinHash*, desta vez, dos títulos dos filmes, usando a matriz matrizMinHashTitles para os registar. Para isso, definimos o número de funções de dispersão igual a 100 e o tamanho dos *shingles* igual a 3. O raciocínio para serem calculados foi semelhante ao da alínea anterior.

```
titles = dic(:,1);
numTitles = length(titles);
numHash = 100;
shingleSize = 3;
matrizMinHashTitles = minHashTitles(titles,numHash,shingleSize);
distancesTitles = getDistancesMinHashTitles(numTitles,matrizMinHashTitles,numHash);
```

Figura 8 | Variáveis criadas

```
function matrizMinHashTitles = minHashTitles(titles,numHash,shingleSize)
    numTitles = length(titles);
    matrizMinHashTitles = inf(numTitles, numHash);

    x = waitbar(0,'A calcular minHashTitles()...');
    for k= 1 : numTitles
        waitbar(k/numTitles,x);
        movie = titles{k};
        for j = 1 : (length(movie) - shingleSize + 1)
            shingle = lower(char(movie(j:(j+shingleSize-1))));
            h = zeros(1, numHash);
            for i = 1 : numHash
                shingle = [shingle num2str(i)];
                h(i) = DJB31MA(shingle, 127);
            end
        matrizMinHashTitles(k, :) = min([matrizMinHashTitles(k, :); h]);
        end
    end
    delete(x);
end
```

Figura 9 | Construção da matriz com minHashs

Após o cálculo dos *MinHash* dos filmes, temos de calcular os *MinHash* da *string* introduzida pelo utilizador. Isto processa-se de maneira semelhante ao que fizemos para calcular os *MinHash*, tanto dos filmes, como dos utilizadores.

```
function searchTitle(search, matrizMinHashTitles, numHash, titles, shingleSize)
    minHashSearch = inf(1, numHash);
    for j = 1 : (length(search) - shingleSize + 1)
        shingle = char(search(j:(j+shingleSize-1)));
        h = zeros(1, numHash);
        for i = 1 : numHash
            shingle = [shingle num2str(i)];
            h(i) = DJB31MA(shingle, 127);
        end
        minHashSearch(1, :) = min([minHashSearch(1, :); h]);
    end

    threshold = 0.99;
    [similarTitles,distancesTitles,k] = filterSimilar(threshold,titles,matrizMinHashTitles,minHashSearch,numHash);

    if (k == 0)
        disp('No results found');
    elseif (k > 5)
        k = 5;
    end

    distances = cell2mat(distancesTitles);
    [distances, index] = sort(distances);

    for h = 1 : k
        fprintf('%s - Distância: %.3f\n', similarTitles{index(h)}, distances(h));
    end
end
```

Figura 10 | Função que imprime no máximo 5 títulos mais parecidos com o input do utilizador

Esta função é a função de pesquisa. Ela primeiro calcula os valores *minHash* como referido anteriormente, e em seguida, usa a função *filterSimilar* para encontrar títulos na matriz de valores *minHash* que tenham uma alta similaridade com o termo de pesquisa.

Finalmente, exibe os **k** títulos similares mais importantes juntamente com suas distâncias por ordem crescente.

Aqui está uma explicação dos parâmetros de entrada:

- ***Search* ->** Este é o termo de pesquisa, que deve ser um array de caracteres;
- ***matrizMinHashTitles* ->** Esta é uma matriz de valores *minHash* para um conjunto de títulos. Cada linha representa um título e cada coluna representa uma função de dispersão;
- ***numHash* ->** Este é o número de funções de dispersão usadas para calcular os valores *minHash*;
- ***Titles* ->** Este é um *array* de títulos, onde cada célula representa um título;
- ***shingleSize* ->** Este é o tamanho do *shingle* usado para calcular os valores *minHash* para o termo de pesquisa.

```
case 3
    search = lower(input('Write a string: ', 's'));

    while (length(search) < shingleSize)
        fprintf('Write a minimum of %d characters\n', shingleSize);
        search = lower(input('Write a string: ', 's'));
    end

    fprintf("\n---------- MOST SIMILAR TITLES ----------\n\n");

    searchTitle(search, matrizMinHashTitles, numHash, titles, shingleSize);

    fprintf("\n-----------------------------------------\n\n");
```

Figura 11 | Código para a execução da terceira opção

OPÇÃO 4

Na opção 4, a aplicação lista os filmes vistos pelo utilizador atual e desses filmes o utilizador escolhe um deles. A aplicação deverá imprimir os 3 filmes mais similares relativamente aos seus géneros e em caso de "empate" compara os valores médios das duas avaliações.

```
numHash = 100;
genres = getGenres(dic);
numGenres = length(genres);
matrizAssGenres = matrizAss(dic,genres);
matrizMinHashGenres = minHash(matrizAssGenres,numHash);
distancesGenres = getDistancesMinHashGenres(numFilms,matrizMinHashGenres,numHash);
```

Figura 12 | Variáveis criadas

número de linhas igual ao número de géneros diferentes e número de colunas igual ao número de filmes. Foi criado um *array* com os géneros dos filmes, através de dois *for loops* que iteram a variável *dic* por todas as linhas e todas as colunas a partir da segunda, captando todas as células que tenham valor diferente de *missing*.ß

```
function genres = getGenres(dic)
    genres = {};
    k = 1;

    for i= 1:height(dic)
        for j= 2:7
            if ~anymissing(dic{i,j}) && ~strcmp(dic{i,j},'unknown')
                genres{k} = dic{i,j};
                k = k+1;
            end
        end
    end

    genres = unique(genres);
end
```

Figura 13 | Função criadora do array dos diferentes géneros de filmes

Para a criação da matriz assinatura, colocámos o valor 1 nos géneros que o filme apresenta, através de 3 *for loops*. Os primeiros dois, iteram a matriz assinatura, para ir atribuindo os valores 1 e o terceiro itera a variável *dic* para saber que géneros pertencem ao filme.

```
function matrizAss = matrizAss(dic,genres)
    numFilms = height(dic);
    numGenres = length(genres);
    matrizAss = zeros(numGenres,height(dic));

    for i= 1:numGenres
        for n= 1:numFilms
            for k= 2:7
                if ~anymissing(dic{n,k})
                    if strcmp(genres(i),dic{n,k})
                        matrizAss(i,n) = 1;
                    end
                end
            end
        end
    end
end
```

Figura 14 | Função criadora da matriz assinatura dos géneros

De seguida, a função *minHash* tem como objetivo calcular o valor de *minhash* para uma matriz de dados de gêneros de filmes e um número de funções de dispersão a usar.

Ela faz isso selecionando ***numHashFunc*** números primos aleatórios e, em seguida, calcula o módulo do índice de cada elemento igual a 1 na matriz de dados pelo número primo atual. O valor mínimo desses módulos é armazenado na matriz *matrizMinHashGenres* e a função retorna essa matriz como resultado. Esta é usada para calcular a similaridade entre dois conjuntos de elementos de maneira eficiente.

```
function matrizMinHashGenres = minHash(matrizAss,numHashFunc)
    p = primes(10000);
    matrizMinHashGenres = zeros(numHashFunc,width(matrizAss));
    kList = p(randperm(length(p),numHashFunc));

    for func= 1:length(kList)
        for d= 1:width(matrizAss)
            matrizMinHashGenres(func,d) = min(mod(find(matrizAss(:,d)==1),kList(func)));
        end
    end

end
```

Figura 15 | Função criadora da matriz minHash dos géneros

Após este processo todo, foi calculada a matriz das distâncias de *Jaccard* com um método já usado duas vezes anteriormente.

```
function distances = getDistancesMinHashGenres(numFilms,matrizMinHash,numHash)
    distances = zeros(numFilms,numFilms);
    for n1= 1:numFilms
        for n2= n1+1:numFilms
            distances(n1,n2) = sum(matrizMinHash(:,n1)==matrizMinHash(:,n2))/numHash;
        end
    end
end
```

Figura 16 | Função que calcula as distâncias de Jaccard da matriz minhash dos géneros

Por fim, apesar de não termos conseguido implementar toda a opção 5, foi implementada parte dela, como a impressão dos filmes vistos pelo utilizador atual e a validação da introdução do *ID* do filme já visto pelo mesmo.

```
watchedFilms = films{current_user};
numWatchedFilmes = height(watchedFilms);

fprintf("\n---------- MY FILMS ----------\n\n");

for i= 1:numWatchedFilmes
    idFilm = watchedFilms(i);
    nameFilm = dic(idFilm,1);
    fprintf("(ID: %d) %s\n", idFilm,nameFilm{1});
end

fprintf("\n------------------------------\n\n");

verificador = false;
while verificador == false
    verificador = false;
    filmIdInput = input('Choose a film ID: ');
    for i= 1:numWatchedFilmes
        idFilm = watchedFilms(i);
        if filmIdInput == idFilm
            verificador = true;
        end
    end

    if verificador == false
        fprintf("ERROR: Enter a valid film ID\n");
    end
end

fprintf("\n------------------------------\n\n");
```

Figura 17 | Código para a execução da quarta opção

OPÇÃO 5

Na opção 5, é pedido, através de um filtro de *Bloom* com contagem, que seja apresentada uma estimativa da quantidade de filmes criados no ano que o utilizador pretende saber.

```
filterSize = 1000;
filtro = inicializarFiltro(filterSize);
years = getAllYears(dic);
numYears = length(years);
uniqueYears = unique(years);
numUniqueYears = length(uniqueYears);
numHashFunc = round(filterSize*log(2)/numUniqueYears);
adicionarElementosFiltro(numYears,years,filtro,numHashFunc,filterSize);
```

Figura 18 | Variáveis criadas

Primeiro temos a função *inicializarFiltro* que inicializa um *array* de zeros com o tamanho especificado pelo parâmetro de entrada **n**. Isto criará o filtro de *Bloom* com **n** posições.

```
function filtro = inicializarFiltro(n)
    filtro = zeros(n,1);
end
```

Figura 19 | Função inicializadora do filtro de Bloom

Depois temos a função *adicionarElemento*. Esta função é usada para adicionar um elemento ao filtro de *Bloom* e este é implementado como um *array* de inteiros. Quando um elemento é adicionado ao filtro, o valor das posições correspondentes é incrementado em um.

Através de um *for* loop, este itera pelo número de funções de dispersão especificado pelo parâmetro numHashFunc. Em cada iteração, o *for loop* gera um novo valor de *hash* para a chave e incrementa-os nas posições do filtro correspondentes.

O parâmetro *tablesize* é usado para garantir que o valor de *hash* gerado pelas funções de dispersão não exceda o tamanho do filtro.

```
function filtro = adicionarElemento(filtro,chave,numHashFunc,tablesize)
    for i= 1:numHashFunc
        chave1 = [chave num2str(i)];
        code = mod(string2hash(chave1),tablesize)+1;
        filtro(code) = filtro(code)+1;
    end
end
```

Figura 20 | Função que adiciona as chaves ao filtro de Bloom

A função *adicionarElementosFiltro* é usada para adicionar vários elementos a um filtro Bloom. Esta adiciona os elementos ao filtro através de um *for loop* que itera por todos os anos guardados na *cell array years*. Em cada iteração, este *for loop* adiciona o ano correspondente ao filtro de Bloom.

Para adicionar o ano ao filtro, a função *adicionarElemento* é chamada. Esta é descrita acima e é usada para adicionar um elemento ao filtro de Bloom.

```
function adicionarElementosFiltro(numYears,years,filtro,numHashFunc,filterSize)
    for i= 1:numYears
        yearstr = "";
        year1 = years{i};
        yearstr = yearstr+year1;

        if yearstr ~= "know" && yearstr ~= ") (V"
            filtro = adicionarElemento(filtro,yearstr,numHashFunc,filterSize);
        end
    end
end
```

Figura 21 | Função que adiciona todos os anos dos filmes ao filtro de Bloom

A função *pertenceConjunto* é usada para verificar se um elemento pertence a um filtro bloom.

Através de um *for loop* que itera pelo número de funções de dispersão especificado pelo parâmetro numHashFunc, cada *loop* gera um novo valor de hash para a chave e incrementando as posições correspondentes no filtro. Os valores são armazenados em um vetor *res*.

Os valores de hash são usados para acessar às posições no filtro de Bloom.

Por fim, a função retorna o valor mínimo de *res*. Se todos os valores de *res* são diferentes de 0, então há grande probabilidade de haver filmes criados naquele ano e a quantidade estimada é igual ao mínimo desses valores. Caso contrário, não há nenhum filme criado nesse ano.

```
function resultado = pertenceConjunto(filtro,chave,numHashFunc,tablesize)
    res = zeros(numHashFunc,1);
    for i= 1:numHashFunc
        chave1 = [chave num2str(i)];
        code = mod(string2hash(chave1),tablesize)+1;
        res(i) = filtro(code);
    end
    resultado = min(res);
end
```

Figura 22 | Função que retorna a estimativa da quantidade de filmes de um determinado ano

Temos também a função auxiliar *getAllyears* que permite percorrer através de um *for loop* todos os títulos dos filmes da variável *dic* e extrair apenas o ano de criação do filme do respetivo título. Estes anos são adicionados a um *cell array years*.

```
function years = getAllYears(dic)
    years = {};
    for i= 1:height(dic)
        titulo = dic{i,1};
        year = extractBetween(titulo, length(titulo)-4, length(titulo)-1);
        years{end+1} = year{1};
    end
end
```

Figura 23 | Função que retorna cell array com os diferentes anos de criação dos filmes

Por fim, com tudo implementado, temos o código que nos permite a concretização da opção 5. Inicialmente, o utilizador tem de inserir o ano que pretende saber quantos filmes foram criados, sendo este valor validado.

```
case 5
    year = 0;
    while(year<=0 || year>2023)
        year = input('Insert year (0 to current year): ');
        if (year<1 || year>2023)
            fprintf("ERROR: Enter a valid year\n");
        end
    end

    yearUserStr = ""+year;

    quantidade = pertenceConjunto(filtro,yearUserStr,numHashFunc,filterSize)

    fprintf("There are %d movies from this year\n",quantidade);

    fprintf("\n-----------------------------------------\n\n");
```

Figura 24 | Código para a execução da quinta opção

ESCOLHA DE NÚMEROS

Experimentamos inicialmente com 200 funções de dispersão, no entanto, o código demorava demasiado tempo a correr. De seguida, reduzimos para 150 funções, mas o problema persistiu. Decidimos, assim, utilizar 100 funções de dispersão para as questões 2, 3 e 4, exclusivamente por uma questão de otimização, uma vez que para este valor, o tempo de espera não era tão elevado. Para além disto, 100 funções de dispersãofornece já uma boa precisão para a distância de Jaccard.

Quanto ao tamanho dos *shingles*, 3 pareceu-nos um tamanho adequado, porque quanto maior for o tamanho do *shingle,* menor vai ser a probabilidade de haver correspondência com outros *shingles*, isto é, é mais improvável obtermos dois *shingles* iguais (daí excluirmos logo o tamanho igual a 2). Após fazermos algumas experiências com o tamanho dos *shingles* igual a 3 e igual a 4, concluímos que havia mais correspondências com 3, pelo que decidimos utilizar este valor.

Por fim, quanto ao tamanho do filtro de *Bloom* a ser utilizado, após várias experiências, verificámos que um tamanho de 1000 é um tamanho equilibrado para o nosso propósito, não sendo demasiado grande nem demasiado pequeno. Verificámos os valores estimados, através de *Ctrl+F* do ficheiro “films.txt”.

CONCLUSÃO

Ao longo deste trabalho, a nível teórico, consolidámos os nossos conhecimentos sobre a matéria em geral do guião 4 (*hash functions*, filtros de *Bloom* , similaridade, etc).

Em suma, quanto ao nível prático, melhorámos não só os nossos conhecimentos sobre Matlab, como também as nossas capacidades de produzir algoritmos probabilísticos ou até mesmo, as nossas capacidades de trabalhar enquanto equipa.

Acreditamos ter alcançado grande parte dos objetivos dados pelo professor, apesar da incompleta implementação da opção 4.