Universidade de Aveiro

Relatório PL04

Métodos Probabilísticos para Engenharia Informática

Departamento de Engenharia Eletrónica, Telecomunicações e Informática

Prof. Carlos Bastos

Ano Letivo 2022/2023

Turma P3

Rafael Pinto (rafaelpbpinto@ua.pt) nº 103379

Tiago Alves (tiagojba9@ua.pt) nº 104110

06/01/2023

Índice

Introdução

Com este relatório, temos como objetivo explicar as opções tomadas na implementação dos métodos probabilísticos, o funcionamento da nossa aplicação e como estão implementadas as interações com o utilizador.

Para o desenvolvimento da aplicação criou-se dois scripts, aos quais demos o nome de “data.m” e “app.m”. No “data.m” encontra-se o código relativo à leitura dos dois ficheiros de entrada (“u.data” e “films.txt”) e ao armazenamento em ficheiro (“data.mat”) de todas as estruturas de dados associadas aos utilizadores e aos filmes. No “app.m” encontra-se o código relativo à leitura dos dados previamente guardados pelo primeiro script e às interações com o utilizador.

Data

```
%% Load films.txt
dic = readcell('films.txt','Delimiter','\t');

%% Load udata
udata = load('u.data');
% obter as duas primeiras colunas
u= udata(1:end,1:2);
% obter segunda e a terceira coluna
avaliacoes = udata(2:end, 2:3);

clear udata;

%% Lista de utilizadores
users = unique(u(:,1)); % IDs dos utilizadores
Nu = length(users); % Numero de utilizadores

%% Lista de filmes para cada utilizador
films_by_user_set= cell(Nu,1);
for n = 1:Nu
    films = find(u(:,1) == users(n));
    films_by_user_set{n} = [films_by_user_set{n}
end(films,2)];
```

Figura 1 - Código de leitura e armazenamento de dados dos ficheiros

Neste script começou-se por guardar o nome e categorias de cada filme a partir do ficheiro *films.txt*. Guardou-se também os IDs de todos os utilizadores, os IDs dos filmes avaliados e as avaliações que cada utilizador deu aos filmes a partir do ficheiro *u.data*. Com essa informação criou-se uma lista com os filmes avaliados por cada utilizador, onde o índice do *array* guardado com os ids dos filmes corresponde ao id do utilizador.

Ao realizarmos o guião PL04 nas aulas, decidimos que a função *DJB31MA.m* seria a função mais indicada para realizar as funções de hash da nossa aplicação, já que uma vez comparada com outras funções de *hash*, esta foi a que deu melhores resultados. Estes resultados envolvem uniformidade, número de colisões e tempos de execução dos quatro tipos de funções disponibilizadas nas aulas.

```
%%
% Matriz de assinaturas com os vectores MinHash correspondente ao conjunto de
filmes
% avaliados por cada utilizador

num_hash_functs = 100;
MinHashSig_user_films = inf(length(users), num_hash_functs);
for i = 1:length(users)
    % conjunto de filmes avaliados pelo utilizador i
    set = films_by_user_set{i};

    for j = 1:length(set)
        key = set(j);
        hash = zeros(1, num_hash_functs);
        for k = 1:num_hash_functs
            key = [key num2str(k)];
            hash(k) = DJB31MA(key, 127);
        end
        MinHashSig_user_films(i, :) = min([MinHashSig_user_films(i, :); hash]);
    end
end
```

Figura 2 - Código para cálculo da matriz de assinaturas com os vetores MinHash correspondente ao conjunto de filmes avaliados por cada utilizador

Para suporte à opção 2 da aplicação, implementou-se uma matriz de assinaturas com os vetores *MinHash* correspondentes ao conjunto de filmes avaliados por cada utilizador.

Para decidir o número de funções de dispersão a usar verificámos os tempos de cálculo do *MinHashSig_user_films* para 50, 100 e 200 funções de dispersão, tal como sugerido no exercício 4 do guião PL04, tendo obtido tempos de 46, 87 e 249 segundos, respetivamente. Usando um maior número de funções de dispersão obtemos valores de distância de *Jaccard* mais próximos da realidade, mas é necessário despender mais tempo para fazer os cálculos quanto maior o número de funções de dispersão usadas. Desta forma achámos que 100 funções de dispersão seria o ideal.

```
%%
% Matriz de assinaturas com os vectores MinHash correspondente ao conjunto de
géneros
% cinematográficos de cada filme

num_hash_functs = 100;
MinHashSig_films_genre = inf(length(dic), num_hash_functs);
for i = 1:length(dic)
    % conjunto de géneros cinematográficos do filme i
    set = {length(dic(i,:))};
    for j = 2:length(dic(i,:))
        set{j-1} = lower(dic{i,j});
    end

    for j = 1:length(set)
        if ~isa(set{j},"missing")
            key = char(set{j});
            hash = zeros(1, num_hash_functs);
            for k = 1:num_hash_functs
                key = [key num2str(k)];
                hash(k) = DJB31MA(key, 127);
            end
            MinHashSig_films_genre(i, :) = min([MinHashSig_films_genre(i, :);
hash]);
        end
    end
end
```

Figura 3 - Código para cálculo da matriz de assinaturas com os vetores MinHash correspondente ao conjunto de géneros cinematográficos de cada filme

Para suporte à opção 3 da aplicação, implementou-se uma matriz de assinaturas com os vetores *MinHash* correspondentes ao conjunto de géneros cinematográficos de cada filme.

Para decidir o número de funções de dispersão a usar tivemos um raciocínio idêntico ao anterior pelo que escolhemos também 100, uma vez que os tempos obtidos foram idênticos aos que obtivemos anteriormente.

```
%%
% Matriz de assinaturas com os vectores MinHash associados aos titulos dos
filmes

shingle_size = 3;
num_hash_functs = 150;
MinHashSig_films_title = inf(length(dic), num_hash_functs);

for i = 1:length(dic)
    set = lower(dic{i,1});
    shingles = {length(set)};
    % Criação de shingle para cada filme
    for j = 1:length(set) - shingle_size + 1
        shingles{j} = set(j:j+shingle_size-1);
    end

    for j = 1:length(shingles)
        key = char(shingles{j});
        hash = zeros(1, num_hash_functs);
        for k = 1:num_hash_functs
            key = [key num2str(k)];
            hash(k) = DJB31MA(key, 127);
        end
        MinHashSig_films_title(i, :) = min([MinHashSig_films_title(i, :);
hash]);
    end

end
```

Figura 4 - Código para cálculo da matriz de assinaturas com os vetores MinHash associados aos títulos dos filmes

Para suporte à opção 4 da aplicação, implementou-se uma matriz de assinaturas com vetores *MinHash* associados aos títulos dos filmes.

Para decidir o número de funções de dispersão o raciocínio foi o mesmo que o anterior, sendo que decidimos calcular tempos para 50,100,150 e 200 funções de dispersão, tendo obtido para o cálculo da *MinHashSig_films_title* tempos de 14, 34, 59 e 88 segundos, respetivamente. Como o número de *shingles* será maior que o número títulos de filmes, decidimos aumentar o número de funções de dispersão para 150 em vez de 100.

Para o tamanho das *shingles* decidimos utilizar comprimento 3. Isto porque um tamanho de 5 seria muito específico podendo não encontrar diferentes filmes similares e um tamanho mais pequeno como 2, já poderia, pelo contrário, ser muito abrangente para a pesquisa pretendida.

```
% Counting Bloom Filter
n = 200000;
m = length(dic);
k = 6;

% calcular
BF = inicializar(n);

for i = 1:length(avaliacoes(:,2))   % percorrer todas as avaliações
    if avaliacoes(i,2) >= 3
        BF = adicionarElemento(BF,avaliacoes(i,1),k);
    end
end

% Funções Bloom Filter %
function BF = inicializar(n)
    BF = zeros(1,n);
end

function BF = adicionarElemento(BF,elemento,k)
    n = length(BF);
    for i=1:k
        elemento = [elemento num2str(i)];
        h = DJB31MA(elemento,127);
        h = mod(h,n) + 1;
        BF(h) = BF(h) + 1;
    end
end
```

Figura 5 - Código relativo ao Bloom Filter

Também foi necessário implementar um *Counting Bloom Filter* para contar o número de vezes que os filmes encontrados foram avaliados com uma avaliação igual ou superior a três.

Decidimos dar um tamanho de 200 000 ao *Bloom Filter* já que no exercício seria necessário percorrer todas as avaliações de filmes e introduzir os IDs de todos os filmes, com avaliações iguais ou superiores a 3, no *Bloom Filter*. Com este tamanho evitamos um grande número de colisões de *hash* de filmes diferentes para posições iguais do filtro e reduzimos o número de falsos positivos.

Utilizámos k = 6 funções de dispersão já que no guião PL04 tinha sido o número para qual obtemos menos falsos positivos.

Ao preencher o *Bloom Filter,* quando o filme já se encontrava presente no mesmo, as posições para que este foi mapeado eram incrementadas permitindo assim contar o número de avaliações pretendido.

App

```
load data;
user_id = 0;

while user_id < 1 || user_id > 943
    user_id = input("Insert User ID (1 to 943): ");
    if user_id < 1 || user_id > 943
        fprintf("Invalid User ID\n");
    end
end

while true
    fprintf("1 - Your movies\n");
    fprintf("2 - Suggestion of movies based on other users\n");
    fprintf("3 - Suggestion of movies based on already evaluated movies\n");
    fprintf("4 - Movies feedback based on popularity\n");
    fprintf("5 - Exit\n");
    option = input('Select choice: ');

    switch option
        case 1
            movies = films_by_user_set(user_id);
            movies = movies{1,1};
            for i = 1:length(movies)
                fprintf('%d - %s\n',movies(i),dic{movies(i),1});
            end
        case 2
            similar_users = getSimilarUsers(user_id, Nu, MinHashSig_user_films);
            movies_user = films_by_user_set(user_id);
            movies_sim_user1 = films_by_user_set(similar_users{1});
            movies_sim_user2 = films_by_user_set(similar_users{2});
            suggestions = getSuggestions(movies_sim_user1, movies_sim_user2,
movies_user);
            for i = 1:length(suggestions)
                fprintf('%d - %s\n',suggestions(i),dic{suggestions(i),1});
            end
            fprintf('\n');
        case 3
            movies_user = films_by_user_set(user_id);
            id_mostCommon_movies =
getTwoMostCommon(movies_user,MinHashSig_films_genre, dic);
            for i = 1:2
                fprintf('%s\n',dic{id_mostCommon_movies{i},1});
            end
            fprintf('\n');
        case 4
            str = lower(input('Insert movie name: ', 's'));
            while length(str) < 4
                fprintf("Invalid input! Must have 4 or more characters.\n")
                str = lower(input('Insert movie name: ', 's'));
            end
            sim_names = getSimilarNames(str, dic, MinHashSig_films_title);
            for i = 1:length(sim_names)
                countAvaliacoes = getAvCount(sim_names{i},BF);
                fprintf('%d - %s -
%d\n',sim_names{i},dic{sim_names{i},1},countAvaliacoes);
            end
        case 5
            break

        otherwise
            fprintf('Invalid option\n');
    end
end
```

Figura 6 - Código do menu da aplicação

Neste script, implementou-se o essencial para a aplicação funcionar de forma correta. A aplicação inicia com um *loop* a pedir, como *input*, o id do utilizador e só avança quando o utilizador introduzir um id válido, se o utilizador introduzir um id inválido será apresentada uma mensagem de erro. Após a introdução do ID do *user*, será apresentado um menu com as opções que o utilizador pode executar, este será apresentado num ciclo infinito até que seja selecionada a opção 5.

Opção 1

Quando o utilizador seleciona a opção 1 a aplicação deve listar os títulos dos filmes que o utilizador atual viu, cada linha deve mostrar o ID e o título do filme. Para tal, através da lista com os filmes avaliados por cada utilizador, criada anteriormente em *data.m*, onde o índice da lista corresponde ao *cell array* com os filmes do utilizador com ID igual a esse índice, seleciona-se o *cell array* armazenado no índice igual ao ID do utilizador atual. Em seguida, percorre-se esse *cell array* e faz-se *print* de cada filme.

Opção 2

Quando o utilizador seleciona a opção 2 a aplicação deve determinar os 2 utilizadores mais similares ao utilizador atual e apresentar os títulos dos filmes que foram avaliados por pelo menos um dos 2 utilizadores e que ainda não foram avaliados pelo utilizador atual. Para isso, criou-se uma função que determina os 2 utilizadores mais similares chamada *getSimilarUsers*.

```
function similar_users = getSimilarUsers(user_id, Nu, MinHashSig_user_films)
    num_hash_functs = 100; % numero de hash functions usadas em data.m
    J = ones(1, Nu); % vetor de similaridade entre o usuario e os outros
    h = waitbar(0, 'Calculating similarity...');
    for i = 1:Nu
        waitbar(i/Nu, h);
        if i ~= user_id
            J(i) = sum(MinHashSig_user_films(i,:) ~=
MinHashSig_user_films(user_id,:))/num_hash_functs; % calcula a distancia de
Jaccard entre o utilizador e os outros
        end
    end
    delete(h);

    similar_users = {2};
    for i=1:2
        [val, id] = min(J);
        similar_users{i} = id;
        J(id) = [];
    end
end
```

Figura 7 - Código da função getSimilarUsers

A função *getSimilarUsers* recebe como argumentos o ID do utilizador atual (*user_id*), o número de utilizadores (*Nu*) e a matriz de assinaturas calculada anteriormente em *data.m* (*MinHashSig_user_films*).

Com os valores das *MinHash* calcularam-se as distâncias de *Jaccard* entre o utilizador e os outros utilizadores. Para encontrar os utilizadores mais similares guardaram-se os dois utilizadores com os menores valores calculados nas distâncias de *Jaccard*. O *cell array* que guarda os utilizadores similares é retornado no final.

Após determinar os utilizadores similares, seleciona-se os filmes desses utilizadores e do utilizador atual de forma semelhante à opção 1. Para encontrar os filmes a listar criou-se a função *getSuggestions*.

```
function suggestions = getSuggestions(movies_sim_user1, movies_sim_user2,
movies_user)
    suggestions = union(cell2mat(movies_sim_user1(:)),
cell2mat(movies_sim_user2(:)), "rows");
    suggestions = setdiff(suggestions,cell2mat(movies_user(:)), 'rows');
end
```

Figura 8 - código da função getSuggestions

Nessa função são passados como argumentos os *cell array* dos filmes do utilizador atual e dos similares. É feita a união dos filmes dos utilizadores similares e em seguida a diferença entre essa união e os filmes do utilizador atual e é retornado esse conjunto. Após determinar os filmes, estes são listados no ecrã.

Opção 3

Quando o utilizador seleciona a opção 3 a aplicação procura os filmes mais próximos aos que o utilizador já viu por comparação dos géneros. A comparação é feita por cada filme já visto pelo utilizador, sendo que no final são escolhidos os dois filmes, ainda não vistos, mais similares a todos os que o utilizador já viu. Para determinar estes dois filmes foi realizada a função *getTwoMostCommon*.

```
function mostCommonThroughGenres = getTwoMostCommon(movies_user, MinHashSig_films_genre, dic)
    user_movies = movies_user{1,1};
    length_user_movies = length(user_movies);
    length_totalMovies = length(dic);
    J = ones(length_totalMovies);

    movies_not_seen = [];
    for m2=1:length_totalMovies
        flag_not_seen = true;
        for m1=1:length_user_movies
            if user_movies(m1) == m2
                flag_not_seen = false;
            end
        end
        if flag_not_seen
            movies_not_seen = [movies_not_seen m2];
        end
    end
    length_not_seen = length(movies_not_seen);

    for m1=1:length_user_movies
        for m2=1:length_not_seen
            J(user_movies(m1),movies_not_seen(m2)) = sum(MinHashSig_films_genre(user_movies(m1),:) ~=
MinHashSig_films_genre(movies_not_seen(m2),:))/100;
        end
    end
    setM = {};
    threshold = 0.8;
    % guardar em set os filmes que tenham dist menor ao m1
    for m1= 1:length_user_movies
        setM{m1} = [];
        for m2= 1:length_not_seen
            % se a distancia for menor que o limiar adiciona ao conjunto
            if J(user_movies(m1),movies_not_seen(m2)) < threshold
                setM{m1} = [setM{m1} movies_not_seen(m2)];
            end
        end
    end

    % Contagem para obter os filmes que aparecem em mais conjuntos
    count = zeros(1,length_totalMovies);
    for i = 1:length(setM)
        for j = 1:length(setM{i})
            % incrementa a cada id de filme do conjunto
            count(setM{i}(j)) = count(setM{i}(j)) + 1;
        end
    end
    %disp(count)
    mostCommonThroughGenres = {2};
    for i=1:2
        [val, id] = max(count);
        mostCommonThroughGenres{i} = id;
        count(id) = [];
    end
end
```

Figura 9 - Código da função getTwoMostCommon

A função recebe como argumentos os filmes vistos pelo utilizador atual (*movies_user*), a matriz de assinaturas calculada anteriormente em *data.m* (*MinHash_Sig_films_genre*) e a lista onde estão presentes todos os filmes e os géneros correspondentes (*dic*).

Primeiro são guardados numa lista todos os filmes que ainda não foram vistos que serão usados na comparação.

Após obter esta lista, com os valores da *MinHash*, vamos obter as distâncias de *Jaccard* entre cada filme que o utilizador já viu e os filmes que ainda não foram vistos. Depois de calcular as distâncias vamos guardar em vários conjuntos, um por cada filme já visto pelo utilizador, os filmes que estão mais próximos desse mesmo filme, devendo estes terem uma distância de *Jaccard* inferior a 0,8.

Finalmente é feita a contagem de quantas vezes cada filme aparece nos diversos conjuntos e são escolhidos os dois filmes que estão presentes num maior número de conjuntos, ou seja que têm mais em comum por género aos filmes já vistos pelo utilizador. Após determinar os dois filmes, estes são listados no ecrã.

Opção 4

Quando o utilizador seleciona a opção 4 a aplicação deve pedir ao utilizador uma *string* com o nome de um filme ou parte do um nome e, em seguida, deve devolver os 5 nomes de filmes com os títulos mais similares à *string* introduzida e, para cada filme, o número de vezes que foi avaliado com uma nota superior ou igual a 3. O utilizador deve introduzir uma *string* que contenha mais de 3 caracteres uma vez que o tamanho da *shingle* que foi usada para calcular as *MinHash* dos títulos dos filmes eram de 3 caracteres, para garantir isso, é criado um *loop* que acaba quando o utilizador insere um número válido de caracteres e mostra uma mensagem de erro se o utilizador não cumprir com o requisito. Após o utilizador inserir uma *string* é necessário encontrar os 5 nomes de filmes mais similares à *string*, para isso criou-se a função *getSimilarNames*.

```
function similar_names = getSimilarNames(str, dic, MinHashSig_films_title)
    num_hash_functs = 150; % numero de hash functions usadas em data.m
    shingle_size = 3;

    % shingles da string inserida
    shingles_string = {};
    for i=1:length(str)-shingle_size+1
        shingles_string{i} = str(i:i+shingle_size-1);
    end

    % Minhash da string inserida
    MinHashSig_string = inf(1,num_hash_functs);
    for j=1:length(shingles_string)
        key = char(shingles_string{j});
        hash = zeros(1,num_hash_functs);
        for k=1:num_hash_functs
            key = [key num2str(k)];
            hash(k) = DJB31MA(key, 127);
        end
        MinHashSig_string(1,:) = min([MinHashSig_string(1,:); hash]);
    end

    % calcula a distancia de Jaccard entre a string inserida e os outros filmes
    J = ones(1, size(dic,1));
    h = waitbar(0, 'Calculating similarity...');
    for i=1:size(dic,1)
        waitbar(i/size(dic,1), h);
        J(i) = sum(MinHashSig_films_title(i,:) ~=
MinHashSig_string)/num_hash_functs;
    end
    delete(h);

    % guarda os 5 filmes mais similares à string inserida
    similar_names = {5};
    for i=1:5
        [val, id] = min(J);
        similar_names{i} = id;
        J(id) = [];
    end
end
```

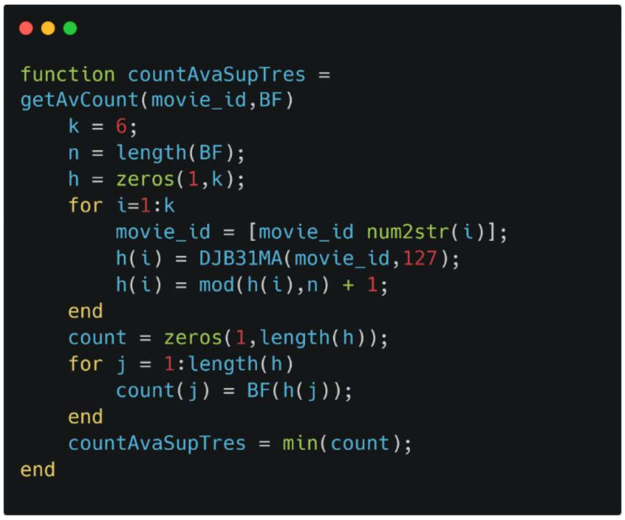
Figura 10 - Código da função getSimilarNames

Nessa função são passados como argumentos a *string* introduzida pelo utilizador (*str*), o *cell array* com o nome dos filmes (*dic*), e a matriz de assinaturas calculada anteriormente em *data.m* (*MinHashSig_films_title*).

A função começa por calcular os *shingles* da *string* inserida e a matriz de assinaturas com vetores *MinHash* associados às *shingles* da *string*. De seguida, com os valores das MinHash calcula-se a distância de *Jaccard* entre a *string* inserida e os outros filmes.

Para encontrar os nomes de filmes com os títulos mais similares à *string* introduzida, guardaram-se os 5 filmes com menor distância de *Jaccard* com a *string* introduzida. O *cell array* que guarda os nomes similares é retornado.

Após determinar os nomes de filmes mais similares à *string* introduzida, é necessário contar o número de vezes que cada um desses filmes foi avaliado com uma nota igual ou superior a 3, para isso criou-se a função *getAvCount*.



```
function countAvaSupTres =
getAvCount(movie_id,BF)
    k = 6;
    n = length(BF);
    h = zeros(1,k);
    for i=1:k
        movie_id = [movie_id num2str(i)];
        h(i) = DJB31MA(movie_id,127);
        h(i) = mod(h(i),n) + 1;
    end
    count = zeros(1,length(h));
    for j = 1:length(h)
        count(j) = BF(h(j));
    end
    countAvaSupTres = min(count);
end
```

Figura 11 - Código da função getAvCount

Nesta função são passados como argumentos um ID de um filme (*movie_id*) e um *Bloom Filter* que foi calculado no *data.m* (*BF*).

São calculados os *hash* do ID do filme, depois com esses *hash* seleciona-se ao *Bloom Filter* a contagem das avaliações com nota igual ou superior a 3 desse filme e no fim é retornada a menor das contagens, uma vez que é a que tem menor número de colisões.

Uma vez descoberto o número de avaliações com nota igual ou superior a 3 desse filme, é listado o ID, nome do filme e número de avaliações.