PL04 - MPEI

Algoritmos Probabilísticos

Ano Letivo: 2023/2024

Turma: P8

Nuno Pinho nº108648

Guilherme Ferreira Santos nº113893

Índice

1 Introdução

A resolução desta secção consistiu no desenvolvimento de dois scripts principais, `loadFiles.m` e `main.m`.
O script `loadFiles.m` é executado uma única vez e tem como propósito ler os dois ficheiros de entrada fornecidos (`utilizadores.data` e `restaurantes.txt`) e guardar toda a sua informação relevante no *WorkSpace* do `MATLAB`.
O script `main.m` lê do *WorkSpace* todas as variáveis importantes para o seu desenvolvimento e depois apresenta ao utilizador todas as suas funcionalidades.
Ao longo da implementação de ambos os scripts foi necessário criar funções em ficheiros á parte que vão ser posteriormente analisadas.

2 Script loadFiles.m

Numa primeira etapa, o script lê os dados dos ficheiros de entrada. Primeiro, é guardado na célula `rest` a informação do ficheiro `restaurantes.txt`, de seguida, `udata` recebe todos os dados do ficheiro `utilizadores.data` e a variável `u` filtra esses dados pelas colunas 1, 2 e 4 pois são as únicas que nos interessam para a resolução da secção. Por fim, `users` pega em todos os utilizadores únicos da variável `u` pois, assim, será mais fácil, no futuro, de utilizar a informação.
Após os dados terem sido tratados é criada a *cell array* `Set` que, por cada *user* único, recebe os *ids* dos restaurantes aos quais deu rating.
Por fim, são geradas as assinaturas e o filtro de *Bloom* para o desenvolvimento das opções descritas nos comentários.

```
% Load data
rest = readcell('restaurantes.txt', 'Delimiter', '\t');
udata = load('utilizadores.data');
u = udata(1:end, [1, 2, 4]); clear udata;
users = unique(u(:,1));

Nu = length(users);
Set = cell(Nu, 1);
for n = 1:Nu
    Set{n} = [Set{n} u((u(:,1) == users(n)),2)];
end
userSignatures = minHash(Set); % for option 2
[dishesSignatures, hashFuns] = minHashDish(rest); %for option 3
restSignatures = minHashRestaurant(rest); %for option 4
[B, hashFunsBloom] = initBloomFilter(u); %for option 5
```

2.1 Função minHash

Esta função serve para gerar a matriz das assinaturas dos utilizadores para ajudar na implementação da opção 2. Em primeiro lugar, o valor de `k` é escolhido, nós optámos por 200 pois, após alguns testes, este foi o melhor valor na relação desempenho/resultado, ou seja, é um valor que contribui para obter resultados decentes sem levar demasiado tempo. De seguida, são inicializados os dados para serem geradas as diferentes funções de dispersão com o recurso á função `initHashFunctions`

```
function hashFuns = initHashFunctions(N, k)
    p = primes(N*2);
    p = p(p>=N);

    p = p(randperm(length(p), 1));

    hashFuns.p = p;
    hashFuns.M = N;
    hashFuns.a = randi([1, (p-1)], 1, k);
    hashFuns.b = randi([0, (p-1)], 1, k);
end
```

Como nós usámos o método de Carter e Wegman como função dispersão, há certos valores que precisam de ser gerados. Esta função começa por gerar números primos aleatórios entre 1 e `N*2`, filtra-os pois têm de ser maiores ou iguais a `N` e, por fim, permuta-os aleatoriamente e escolhe o primeiro. O valor de `M` é simplesmente `N` e `a` e `b` são *arrays* com tamanho `k` de valores aleatórios.

Numa segunda etapa, por cada utilizador `u`, vão se buscar todos os restaurantes a ele associados e por cada `j` de `k` funções dispersão é calculado o *minHash* do conjunto de restaurantes através a função `carterMinHashFunction` que aplica o método de Carter e Wegman anteriormente referido.

$$h(x) = ((ax + b) mod\ p) mod\ M$$

$$minHash = min(h(x)),\ x = conjunto\ de\ restaurantes$$

E, por fim, atribui á matriz das assinaturas `signatures` o valor obtido na posição (`j`, `u`)

```
function signatures = minHash(Set)
    k = 200;
    N = 1e6;
    hashFuns = initHashFunctions(N, k);

    Nu = length(Set);
    signatures = zeros(k, Nu);
    for u = 1:Nu
        restaurants = Set{u};
        for j = 1:k
            signatures(j, u) = carterMinHashFunction(hashFuns, j, restaurants);
        end
    end
end
```

2.2 Função minHashDish

Esta função serve para gerar a matriz das assinaturas que vai ajudar na implementação da opção 3. Diferente da função anterior, esta usa o conceito de *shingles* uma vez que a matriz retornada vai servir para a comparação entre vetores de caracteres. Assim como em `minHash`, foram precisos serem escolhidos valores para `k` mas desta vez também é preciso escolher o valor para o `shingleSize` e ambos foram escolhidos pelas mesmas razões da função anterior (a relação resultados/desempenho foram razoáveis para estes valores).
Esta função, gera a estrutura `hashFuns` e, por cada restaurante, vai buscar o vetor de caracteres associado ao prato (*dish*). De seguida, o vetor de caracteres é unificado pois, da maneira que foi implementada, os espaços dividiam os *shingles* antes de atingirem o tamanho `shingleSize` desejado. Após isto, são gerados os *shingles* e são-lhes aplicadas as `k` funções dispersão e atribuído o *minHash* na matriz das assinaturas por cada iteração. Viu-se necessário o retorno da estrutura `hashFuns` uma vez que, no script `main.m` será necessária para a resolução da opção 3

```
function [signatures, hashFuns] = minHashDish(rest)
    shingleSize = 5;
    k = 200;
    Nres = height(rest);
    signatures = zeros(k, Nres);
    N = 1e6;
    hashFuns = initHashFunctions(N, k);

    for restaurantID=1:Nres
        dish = lower(rest{restaurantID, 6});
        if ismissing(dish)
            signatures(:,restaurantID) = -1; %if restaurant doesnt have dish ignores
        else
            dish = strrep(dish, ' ', '');
            shingles = {};
            if length(dish) < shingleSize
                shingles{1} = dish;
            else
                for i=1:length(dish)-shingleSize+1
                    shingles{i} = [dish(i:i+shingleSize-1) ' '] ;
                end
            end
            shingles = [shingles{:}];
            for j=1:k
                signatures(j, restaurantID) = carterMinHashFunction(hashFuns, j, shingles);
            end

        end
    end
end
```

2.3 Função minHashRestaurant

Esta função serve para gerar a matriz das assinaturas que vai ajudar na implementação da opção 4. Como na função anterior, esta usa o conceito de *shingles*, no entanto, em vez da matriz retornada servir para a comparação entre vetores de caracteres vai ser entre conjuntos de vetores de caracteres. Assim como em `minHashDish`, foram precisos serem escolhidos valores para o `shingleSize` e para `k` e ambos se mantiveram iguais pelas mesmas razões.
Esta função, gera a estrutura `hashFunsrest` e, por cada restaurante, vai buscar os vetores de caracteres associados aos diferentes campos do restaurante. De seguida, é feito um vetor de caracteres que resulta da junção dos vetores anteriores e este é também unificado. Após isto, são gerados os *shingles* e são-lhes aplicadas as `k` funções dispersão e atribuído o *minHash* na matriz das assinaturas por cada iteração.

```
function signatures = minHashRestaurant(rest)
    shingleSize = 5;
    k = 200;
    Nres = size(rest, 1);
    signatures = zeros(k, Nres);
    N = 1e6;
    hashFunsrest = initHashFunctions(N, k);

    for restaurantID = 1:Nres
        local = lower(rest{restaurantID, 3}); % Local in the 2nd column
        district = lower(rest{restaurantID, 4}); % Districts in the 4th column
        restaurantType = lower(rest{restaurantID, 5}); % Types in the 5th column
        dish = lower(rest{restaurantID, 6}); % Dishes in the 6th column
        if ismissing(dish)
            dish = '';
        end
        closed = lower(rest{restaurantID, 7}); % Closed time in the 7th column
        if ismissing(closed)
            closed = '';
        end
        combinedText = [local, ' ' , district, ' ', restaurantType, ' ', dish, ' ', closed]; %
    Combine all char vectors
        combinedText = strrep(combinedText, ' ', '');
        % Split combinedText into shingles
        shingles = {};
        for i = 1:length(combinedText) - shingleSize + 1
            shingles{i} = [combinedText(i:i + shingleSize - 1) ' '];
        end
        shingles = [shingles{:}];

        % Generate signatures using shingles
        for j = 1:k
            signatures(j, restaurantID) = carterMinHashFunction(hashFunsrest, j, shingles);
        end
    end
end
```

2.4 Função initBloomFilter

Aqui será gerado o *Bloom Filter* que vai ajudar na implementação da opção 5. Nesta função também foi necessária a escolha de certos valores, no entanto, ao contrário das outras funções, esta baseia-se numa escolha teórica.

`m` é o número de dados a serem introduzidos no filtro;
`b` é o número de bits de capacidade de cada elemento, que foi escolhido de maneira a não haverem problemas de *overflow* mas também sem "alocar"demasiado espaço na memória;
`probFakePositive` é a probabilidade máxima aceite para os falsos positivos.
Agora para valores importantes: o `n` é escolhido entre 1e5 e 1e7, de seguida, é calculado o valor ótimo de `k` baseado na expressão $\frac{0.693n}{m}$, após ter sido obtido o `k` é calculada a probabilidade de falsos positivos com estes valores usando a expressão $(1 - e^{(\frac{-km}{n})})^k$. Se a probabilidade calculada for menor ou igual á variável `probFakePositive` os valores de `n` e de `k` são aceites, se não. é atribuído a `n` outro valor e assim sucessivamente até a condição for verdadeira. Com estas condições, os valores gerados foram:

$$n = 900000,\ k = 11,\ \textit{False Positive Probability} = 0.000654$$

Com estes valores, a função gera a estrutura `hashFunsBloom` para gerar as funções de dispersão e é inicializado o *Bloom Filter* a zeros. De seguida, por cada `userID` de `u`, são calculados os índices do filtro a partir das `k` funções dispersão e, em cada índice, é incrementado o valor. Viu-se necessário o retorno da estrutura `hashFunsBloom` uma vez que, no script `main.m` será necessária para a resolução da opção 5

```
function [B, hashFunsBloom] = initBloomFilter(u)
    m = length(u);
    b = 10; % number of max bits per element
    % Choosing the best k and n values
    probFakePositive = 0.001; % 0.1% of false positives in maximum
    for i = 1e5:1e5:1e7
        n = i;
        k = round((0.693*n)/m); % best k in theory
        prob = (1 - exp(-k*m/n))^k;
        if prob <= probFakePositive
            break;
        end
    end
    N = 1e6;
    hashFunsBloom = initHashFunctions(N, k);
    % Initializing the bloom filter
    B = zeros(n, 1);
    % Inserting the elements
    for i = 1:m
        userID = u(i, 1);
        for j=1:k
            hash = mod(mod(mod(hashFunsBloom.a(j).*userID + hashFunsBloom.b(j), hashFunsBloom.p),
    hashFunsBloom.M), n)+1;
            if any(B(hash) == 2^b-1) % If the counter is full
                break;
            end
            B(hash) = B(hash)+1; % Incrementing the counter
        end
    end
end
```

3 Script main.m

Este módulo representa o sistema interativo que permite aos usuários consultar e explorar informações relativamente aos restaurantes avaliados. Primeiramente, o script começa por ler do *WorkSpace* todos os dados que foram gerados pelo script `loadFiles.m` e, de seguida, será pedido para inserir um `userID` válido. Depois um menu de opções será mostrado, cada uma destas opções representa uma funcionalidade que poderá, ou não, depender do `userID` inserido. O utilizador deverá escolher uma destas opções, se a escolhida não for a opção 6-"exit"de terminar todas as operações, o menu voltará a aparecer e o processo irá se repetir. A opção 1 consiste em simplesmente mostrar os restaurantes que o `userID` avaliou com recurso á função `printEvaluatedRestaurants`. A opção 2 , usa a função `findSimilar` com as assinaturas `userSignatures` e recolhe o *id* do utilizador mais similar ao inserido, de seguida, recorrendo á função referida anteriormente, mostra todos os restaurantes aos quais o utilizador com *id* = `similarUserID` avaliou. A opção 3, diferente das anteriores, não depende do `userID` inserido mas pede uma *string* ao utilizador que posteriormente vai ser usada para encontrar, no máximo, cinco restaurantes com pratos similares á *string* que inseriu com a utilização das assinaturas `dishesSignatures`. Na opção 4, são mostrados os restaurantes que o `userID` avaliou e é pedido que o utilizador escolha um dos restaurantes por *id*, após a escolha, com o uso da função `seatchSimilarRestaurants` e utilização das assinaturas `restSignatures`, são mostrados, no máximo, 3 restaurantes mais similares ao escolhido.Por fim, a opção 5 pede ao utilizador um novo *userID* que é validado, ou seja, tanto tem de pertencer ao conjunto `users` como não pode ser o mesmo que foi inserido antes, e com recurso á função `calcNumberOfRatings` mostra o número estimado de avaliações do `newUserID`.

```
% Input the User ID
while 1
    userID = input('Insert User ID (1 to ??):');

    % Only breaks if valid
    if userID >= 1 && userID <= length(users)
        break;
    end
end

while 1
    option = input('1 - Restaurants evaluated by you\n' + ...
        '2 - Set of restaurants evaluated by the most similar user\n' + ...
        '3 - Search special dish\n' + ...
        '4 - Find most similar restaurants\n' + ...
        '5 - Estimate the number of evaluations for each tourist\n' + ...
        '6 - Exit\n' + ...
        'Select choice:');

    switch option
        case 1
            fprintf(0-----------Evaluated Restaurants-----------\n0);
            printEvaluatedRestaurants(userID, Set, rest);
            fprintf('----------------------------------------\n');
        case 2
            similarUserID = findSimilar(userID, userSignatures, Set);
            fprintf('-----------Evaluated Restaurants By Similar-----------\n');
            printEvaluatedRestaurants(similarUserID, Set, rest);
            fprintf('----------------------------------------------------\n');
        case 3
            dishInput = lower(input('Write a dish:', 's'));
            searchSimilarDishes(dishInput, rest, hashFuns, dishesSignatures)
        case 4
            printEvaluatedRestaurants(userID, Set, rest)
            while 1
                restaurantID = input('\nChoose a restaurant by ID:');
                evaluatedRestaurants = Set{userID};
                if any(evaluatedRestaurants == restaurantID)
                    break
                else
                    fprintf('RestaurantID Not Found\n');
                end
            end
            searchSimilarRestaurants(restaurantID, rest, restSignatures, u);

        case 5
            while 1
                newUserID = input('Insert Another User ID:');
                % Only breaks if valid
                if newUserID >= 1 && newUserID <= length(users) && newUserID ~= userID
                    break;
                end
            end
            nRatings = calcNumberOfRatings(newUserID, B, hashFunsBloom);
            fprintf('UserID: %d ESTIMATE RATINGS NUMBER: %d\n', newUserID, nRatings);
        case 6
            break
        otherwise
            fprintf('Invalid Option\n');
    end
end
```

3.1 Função printEvaluatedRestaurants

Esta função tem a finalidade de mostrar os detalhes dos restaurantes que foram avaliados pelo usuário inserido. Através do `userID` e procurando no `Set`, será possível recolher e guardar todos os restaurantes avaliados pelo user. De seguida, a função vai iterar por cada restaurante encontrado e, a partir do seu *id*, com acesso á *cell array* `rest`, vai imprimir os seus detalhes como o ID, Nome e Concelho.

```
function printEvaluatedRestaurants(userID, Set, rest)
    evaluatedRestaurants = unique(Set{userID});

    for i=1:length(evaluatedRestaurants)
        restaurantID = evaluatedRestaurants(i);
        restaurantNAME = rest{restaurantID, 2};
        restaurantCONCELHO = rest{restaurantID, 4};
        fprintf('ID:%3d NAME:%s DISTRICT:%s\n', restaurantID, restaurantNAME, restaurantCONCELHO);
    end
end
```

3.2 Função findSimilar

Com o objetivo de encontrar o user mais similar em termos de restaurantes visitados, a função `findSimilar` percorre todos os `users` e calcula a distância entre o `userID` e todos os outros `users`, ignorando a distância entre si mesmo, com base na matriz de assinaturas `signatures` recebida como argumento. No fim, será retornado o ID do usuário mais similar (o que possui a menor distância) em relação ao userID. Neste caso, a distância de Jaccard é calculada com base nos valores que são iguais na matriz das assinaturas `signatures`.

```
function similarUserID = findSimilar(userID, signatures, Set)
    distances = zeros(1,length(Set));
    for u=1:length(Set)
        if u~=userID
            c1 = signatures(:, u);
            c2 = signatures(:, userID);

            distances(u)= 1 - sum(c1 == c2)/height(signatures);
        else
            distances(u) = Inf; %ignores the distance between himself
        end
    end
    [~,similarUserID] = min(distances);
end
```

3.3 Função searchSimilarDishes

Esta função recebe a string inserida na opção 3, bem como as assinaturas dos pratos existentes de cada restaurante. Através da utilização de *shingles* e da `minHash` respetiva, será criada a assinatura da *string* que se deseja comparar. Para comparação, é calculada a distância entre a assinatura da *string* inserida e as assinaturas dos pratos dos restaurantes utilizando a distância de Jaccard. Isto é feito comparando os elementos das assinaturas e calculando a proporção de elementos coincidentes. As distâncias calculadas serão ordenadas para encontrar os pratos dos restaurantes mais similares à string inserida selecionando, no máximo, 5 `similarRestaurants`. Por fim, serão mostrados os restaurantes que contêm os pratos mais similares e os seus detalhes.

```
function searchSimilarDishes(string, rest, hashFuns, dishesSignatures)
    string = strrep(string, ' ', '');
    % Creating signatures for inputed string
    shingleSize = 5;
    k = 200;
    shingles = {};
    if length(string) < shingleSize
        shingles{1} = string;
    else
        for i=1:length(string)-shingleSize+1
            shingles{i} = [string(i:i+shingleSize-1) ' '];
        end
    end
    shingles = [shingles{:}];

    stringSignature = zeros(k, 1);
    for j=1:k
        stringSignature(j, 1) = carterMinHashFunction(hashFuns, j, shingles);
    end

    % Find similars
    Nres = height(rest);
    distances = zeros(1, Nres);
    for restaurantID=1:Nres
        if ~any(dishesSignatures(:, restaurantID) == -1)
            c1 = stringSignature(:, 1);
            c2 = dishesSignatures(:, restaurantID);

            distances(restaurantID)= 1 - sum(c1 == c2)/k;
        else
            distances(restaurantID) = Inf; %Distance from missing dish ignored
        end
    end

    [sortedDistances, sortedIndices] = sort(distances);
    similarRestaurants = [];

    for i = 1:min(5, sum(sortedDistances <= 0.99))
        similarRestaurants = [similarRestaurants, sortedIndices(i)];
    end

    % Display the result
    if isempty(similarRestaurants)
        fprintf('Restaurant NOT FOUND\n');
    else
        for i=1:length(similarRestaurants)
            restaurantID = similarRestaurants(i);
            restaurantNAME = rest{restaurantID, 2};
            restaurantLOCAL = rest{restaurantID, 3};
            restaurantDISH = rest{restaurantID, 6};
            distance = distances(restaurantID);
            fprintf('NAME: %s LOCAL: %s DISH: %s JACCARD_DISTANCE:%.2f\n', restaurantNAME,
    restaurantLOCAL, restaurantDISH, distance);
        end
    end
end
```

3.4 Função searchSimilarRestaurants

Esta função será necessária para encontrar as similaridades nos vários campos que representam cada restaurante. Utilizando as assinaturas dos restaurantes recebidas como argumento `restSignatures`, será comparada a assinatura do restaurante de entrada com todas as outras assinaturas dos restaurantes utilizando a distância de Jaccard. Estas distâncias serão armazenadas e classificadas para encontrar os restaurantes que são mais similares ao restaurante de entrada considerando a condição de similaridade (distância menor ou igual a 0.99). No entanto, como só podem ser exibidos, no máximo, três restaurantes, esta função aplica um método de desempate, que consiste em pegar na *array* de distâncias ordenadas `sortedDistances` e vai percorrendo-a começando pelo início e, caso não hajam valores iguais ao da `currentDistance`, não é preciso desempatar e move-se para a frente, no entanto, caso sejam encontrados valores iguais, ou seja, `length(tiedIndices)` > 1, vão-se buscar os *ids* dos restaurantes com distâncias iguais `tiedIDS` e faz-se uma lista da média de avaliações de cada restaurante, ganha aquele que tiver maior média e anda-se para a frente na lista, o tamanho dos restaurantes com a mesma distância, isto repete-se até, ou a lista das distâncias `similarDistances` ter sido toda percorrida, ou que o tamanho de `similarRestaurants` seja 3. No fim, exibe os detalhes de, no máximo, três restaurantes mais similares ao restaurante de entrada, incluindo a distância de Jaccard calculada.

```
function searchSimilarRestaurants(restaurantID, rest, restSignatures, u)
    Nres = width(restSignatures);
    distances = zeros(1, Nres);
    for restID = 1:Nres
        % Skip comparing the input restaurant with itself
        if restID == restaurantID
            distances(restID) = Inf;
            continue;
        end

        c1 = restSignatures(:, restaurantID);
        c2 = restSignatures(:, restID);

        distances(restID) = 1 - sum(c1 == c2) / height(restSignatures);
    end

    % Sort distances and find three most similar restaurants
    [sortedDistances, sortedIndices] = sort(distances);
    similarDistances = sortedIndices(1:sum(sortedDistances <= 0.99));

    similarRestaurants = [];

    i = 1;
    idx = 1;
    while i <=length(similarDistances) && length(similarRestaurants) < 3
        currentDistance = sortedDistances(i);
        tiedIndices = find(sortedDistances == currentDistance);
        if length(tiedIndices) == 1
            similarRestaurants(idx) = sortedIndices(tiedIndices);
            i = i+1;
            idx = idx+1;
        else
            tiedIDS = sortedIndices(tiedIndices);
            meanRatings = zeros(1, length(tiedIDS));
            for j=1:length(tiedIDS)
                restID = tiedIDS(j);
                restIndex = u(:, 2) == restID;
                meanRatings(j) = mean(u(restIndex, 3));
            end
            winner = tiedIDS(find(max(meanRatings)));
            similarRestaurants(idx) = winner;
            idx = idx+1;
            i = i+length(tiedIDS);
        end

    end
    [name, local, district, restaurantType, dish, ~] = getRestInfo(rest, restaurantID);
    % Display provided restaurant and three most similar restaurants
    fprintf('Comparing Restaurant:\nNAME: %s, LOCAL: %s, DISTRICT: %s, TYPE: %s, DISH: %s\n', name,
    local, district, restaurantType, dish);
    if isempty(similarRestaurants)
        fprintf('Similar restaurants NOT FOUND\n');
    else
        fprintf('Three Most Similar Restaurants:\n');
        for i = 1:length(similarRestaurants)
            restID = similarRestaurants(i);
            [restNAME, restLOCAL, restDISTRICT, restTYPE, restDISH, ~] = getRestInfo(rest, restID);
            fprintf('NAME: %s, LOCAL: %s, DISTRICT: %s, TYPE: %s, DISH: %s, JACCARD_DISTANCE: %.3f\n
    ', restNAME, restLOCAL, restDISTRICT, restTYPE, restDISH, distances(restID));
        end
    end
end
```

3.5 Função getRestInfo

Devido á necessidade de, repetidamente, recolher informação relativamente a um restaurante `restaurantID`, foi criada esta função como auxílio á função anterior.

```
function [restNAME, restLOCAL, restDISTRICT, restTYPE, restDISH, restCLOSED] = getRestInfo(rest,
    restaurantID)
    restNAME = rest{restaurantID, 2};
    restLOCAL = rest{restaurantID, 3};
    restDISTRICT = rest{restaurantID, 4};
    restTYPE = rest{restaurantID, 5};
    restDISH = rest{restaurantID, 6};
    if ismissing(restDISH)
        restDISH = '';
    end
    restCLOSED = rest{restaurantID, 7};
    if ismissing(restCLOSED)
        restCLOSED = '';
    end
end
```

3.6 Função calcNumberOfRatings

Esta função recebe o *id* de um utilizador `newUser`, o *Bloom Filter* `B` e a estrutura para as funções de dispersão `hashFunsBloom`. Com isto, é inicializada a lista das avaliações do utilizador inserido, de seguida, por cada `i` função dispersão, é encontrado o *hashcode* associado ao utilizador e é atribuído, á lista criada anteriormente, o valor do filtro na posição do *hashcode*. Por fim, o número estimado de avaliações do utilizador `newUser` é o menor valor da lista `nRatingsArray`.

```
function nRatings = calcNumberOfRatings(newUser, B, hashFunsBloom)
    n = length(B);
    k = length(hashFunsBloom.a); %Number of hash functions

    %Getting the estimate number of ratings
    nRatingsArray = zeros(1, k);
    for i=1:k
        hash = mod(mod(mod(hashFunsBloom.a(i).*newUser + hashFunsBloom.b(i), hashFunsBloom.p),
    hashFunsBloom.M), n)+1;
        nRatingsArray(i) = B(hash);
    end

    nRatings = min(nRatingsArray);
end
```