universidade de aveiro

ASSIGNMENT 3 - RELATÓRIO

Weighted (tf-idf) Indexer and Ranked Retrieval System

Recuperação de Informação

Curso: Engenharia de Computadores e Telemática

Departamento de Eletrónica, Telecomunicações e Informática

20 de novembro de 2017

Professor:

Prof. Sérgio Matos

Autores:

Miguel Oliveira nº 72638

Tiago Henriques nº 73046

universidade de aveiro

Índice

universidade de aveiro

1 – Introdução

O Assignment 3 foi realizado tendo como base o Assignment 1 e o Assignment 2, onde foram feitas ligeiras alterações ao Indexer de modo a criar um Indexer por pesos (weighted indexer) utilizando o mecanismo tf-idf seguindo a estratégia lnc.ltc conforme está descrita nos slides da componente teórica. Ainda nesta iteração, pretende-se implementar um mecanismo de ranking de pesquisa, ou seja, tem-se como objetivo apresentar uma ordenação dos resultados da pesquisa obtida através de diversas queries.
Neste relatório, será feita uma descrição detalhada das classes e métodos usados, o data flow do funcionamento geral da execução, bem como as instruções e as estruturas usadas na elaboração do programa. Serão ainda referidas as alterações efetuadas sobre as iterações anteriores.
Por fim, serão expostos os resultados da execução do programa como o tempo e o total de memória utilizada.

2 – Alterações realizadas ao trabalho da primeira e segunda entregas

Nesta iteração, alteramos ligeiramente o processo de pesquisa usado na segunda iteração de modo a torná-lo mais otimizado e a retornar melhores tempos de performance dado que era pedido também para calcular as métricas de avaliação e eficiência para o trabalho realizado na segunda iteração. As alterações específicas da segunda iteração encontram-se na subseção imediatamente abaixo no relatório.
Mais no contexto propriamente dito desta iteração, focamo-nos no mecanismo de indexação tf-idf que vai ser devidamente explicado mais à frente no relatório. O formato do Indexer teve de ser ligeiramente mudado de modo a permitir o indexamento dos termos por peso.
Portanto, nesta iteração, o Indexer encontra-se com o seguinte formato: termo, docId: pesoNormalizado, ou seja:

- **termo:** termo ou palavra a ser indexada;
- **docId:** Identificador do documento do termo ou palavra indexado(a);
- **pesoNormalizado:** Peso normalizado tf-idf do termo no documento (explicado mais à frente).

Ainda no contexto da iteração, já com os termos indexados, após calculado o seu valor de tf-idf e correspondente indexer escrito em disco, era pretendido também o carregamento do indexer para memória de modo a efetuar uma pesquisa que apresentasse resultados ordenados. Para isso, procedeu-se a um método de ranking de pesquisa.

2.1 – Alterações específicas para o Assignment 2

Como foi referido, como era também objetivo desta iteração calcular as métricas de classificação e de eficiência relativamente à iteração anterior, foram efetuadas algumas mudanças ao nível das estruturas de dados usadas e implementados novos métodos de modo a obter melhores resultados. As alterações feitas são mínimas, mas resultam em tempos de performance significativamente melhores. O processo usado é semelhante ao desta iteração, e na nossa implementação, quando invocamos o método “process” no RankProcessor, passamos como argumento uma string “assignment”, que irá encaminhar o processo de cálculo do score e posteriormente, cálculo das métricas de classificação e de eficiência associadas a cada Assignment. Assim, foram criados novos métodos para o Assignment 2 como: getQueryIdDocIdScorei(), getQueryIdDocIdScoreii() e calculateScoreAssign2(...) na classe RankedRetrieval para adaptar as estruturas utilizadas no cálculo do score do Assignment 2.

3 - Diagrama de Classes

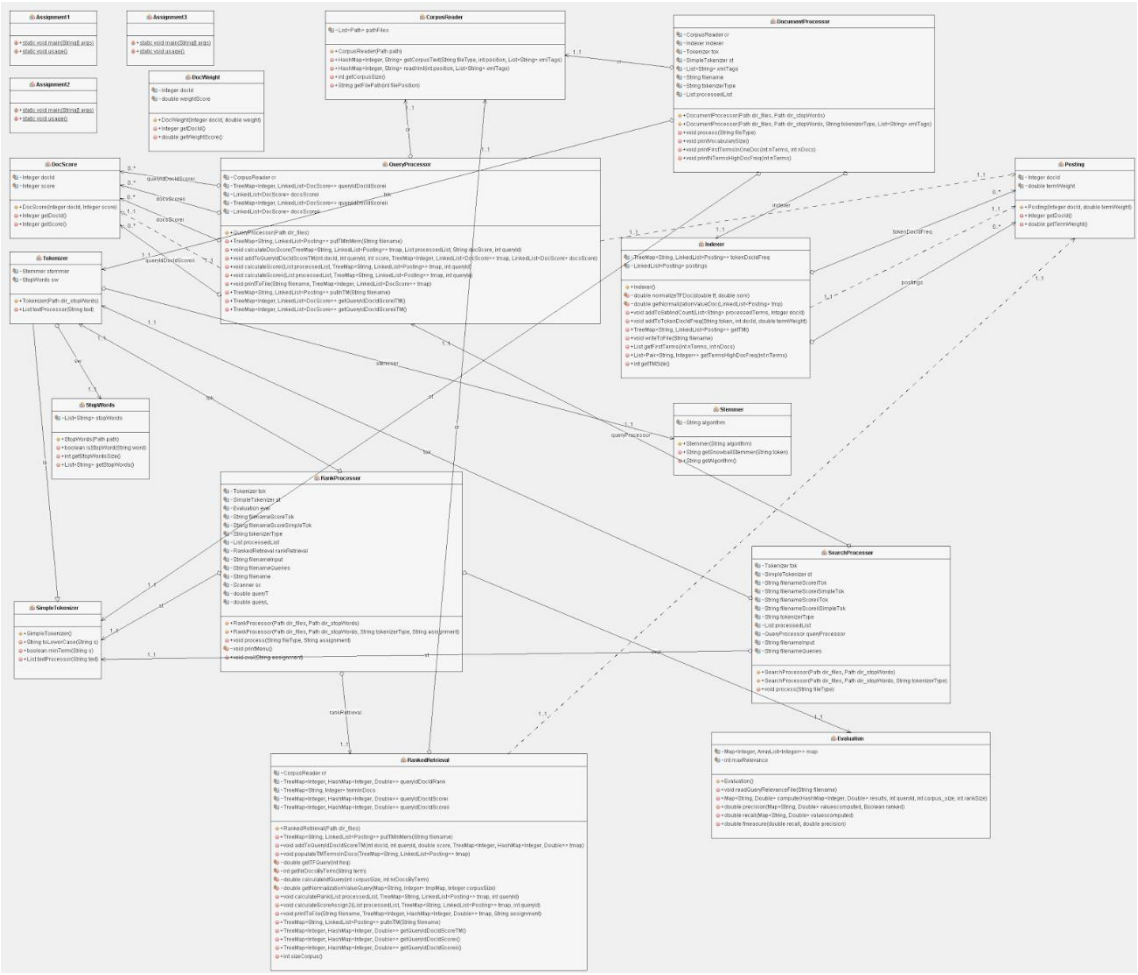


Figura 1 - Diagrama de Classes onde estão presentes as classes e os seus respetivos métodos

4 – Classes e Métodos

4.1 – Assignment3 (main)

Classe principal e executável do projeto. Aqui, é instanciado o DocumentProcessor ou o RankProcessor que iremos descrever mais abaixo.

4.2 – Assignment2 (main)

Classe principal e executável do projeto. Aqui, é instanciado o RankProcessor que iremos descrever mais abaixo.

4.3 – Indexer

Como foi referido na Introdução, foram feitas ligeiras alterações nesta classe de modo a criar um Indexer por peso (weighted-indexer) utilizando o mecanismo tf-idf seguindo a estratégia lnc.ltc conforme está descrita nos slides da componente teórica. Toda a lógica inerente a esta classe mantém-se a mesma que foi utilizada na iteração 1 do trabalho tendo sido acrescentados alguns métodos necessários para o cálculo do peso dos termos num dado documento.

Métodos adicionados e a sua descrição:

- **normalizeTFDoc(double tf, double som):**
  - Método responsável por proceder à divisão do valor de tf do termo do documento sobre o valor da normalização dos termos nesse documento.

- **getNormalizationValueDoc(LinkedList<Posting> tmp):**
  - Método que itera sobre os termos de um documento e retorna o valor da normalização dos termos nesse documento. O cálculo do valor da normalização dos termos de um documento encontra-se explicado mais abaixo no relatório.

- **addToSetAndCount(List<String> processedTerms, Integer docId):**
  - Este método mantém a mesma lógica da iteração 1, ou seja, a lista de termos processados é convertida num set de strings para se proceder à contagem da frequência de cada termo. No entanto, agora é necessário calcular o valor de tf do termo no documento dada a frequência de ocorrência (vai ser explicado mais à frente no relatório). O valor resultante é então normalizado e de seguida, é seguido o mesmo processo da primeira iteração. É então invocado, para cada termo, o método addToTokenDocIdFreq, que irá ser responsável por adicionar os termos e a valor de score à LinkedList e por fim, associar a LinkedList à TreeMap.

4.3 – Posting

Esta classe foi adaptada de iterações anteriores e tem como principal objetivo ser uma estrutura de dados de modo a ser usada no Indexer. Agora, esta classe é constituída pelo identificador do documento em que o termo aparece e o seu peso, representado por um valor double nesta iteração.

Métodos e a sua descrição:

- **getDocId():** Método que retorna o identificador do documento;
- **getTermWeight():** Método que retorna a peso desse termo no documento.

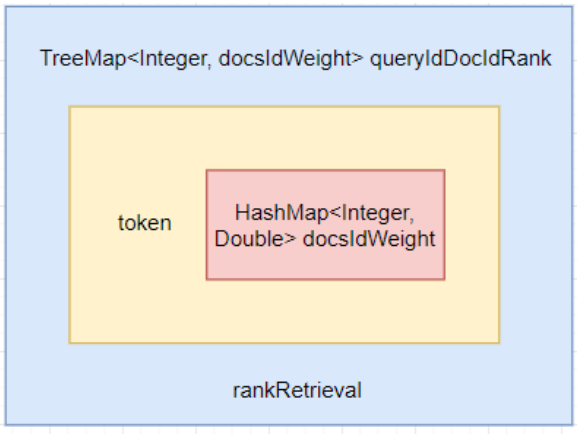
4.4 – RankedRetrieval

Estruturas de dados usadas:

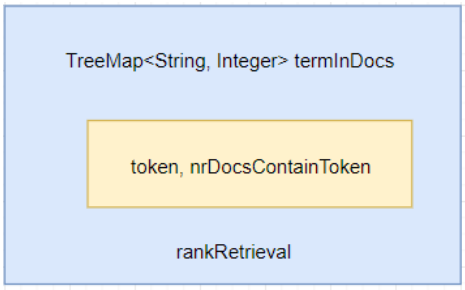
A estrutura de dados pensada e implementada tem como objetivo principal o aumento do desempenho da pesquisa e de ranking. Optámos por mudar as estruturas de dados a utilizar de modo a aumentar o processo de pesquisa usado na iteração anterior. Portanto, consideramos que a melhor estrutura de dados a adotar seria:

TreeMap<Integer, HashMap<Integer, Double>> queryIdDocIdRank: Este tipo de dados herda as características de uma TreeMap, em que a chave corresponde ao identificador da query e o valor corresponde a outra HashMap onde a chave é um identificador do documento (docId) e o valor é o respetivo peso (weight).

TreeMap referida anteriormente.



TreeMap<String, Integer> termInDocs: Esta estrutura é usada como suporte para o cálculo do idf das queries (explicado mais à frente) e herda as características de uma TreeMap, em que a chave corresponde ao termo e o valor corresponde ao número de documentos onde esse termo existe.



Como foi referido anteriormente, foram feitas alterações de modo a melhorar os tempos de pesquisa da segunda iteração. As estruturas de dados usadas como suporte da pesquisa da segunda iteração foram também mudadas e de momento são as seguintes:

TreeMap<Integer, HashMap<Integer, Double>> queryIdDocIdScorei: Este tipo de dados herda as características de uma TreeMap, em que a chave corresponde ao identificador da query e o valor corresponde a outra HashMap onde a chave é um identificador do documento (docId) e o valor é o respetivo score (score do tipo 1).

TreeMap<Integer, HashMap<Integer, Double>> queryIdDocIdScoreii: Este tipo de dados herda as características de uma TreeMap, em que a chave corresponde ao identificador da query e o valor corresponde a outra HashMap onde a chave é um identificador do documento (docId) e o valor é o respetivo score (score do tipo 2).

Esta classe é instanciada na classe RankProcessor. Esta classe tem como objetivo implementar os métodos associados ao carregamento do Indexer em disco e ao cálculo do score, ou seja, é nesta classe que primeiramente são carregados os termos existentes no Indexer criado na iteração 1 e ligeiramente adaptado nesta iteração para permitir a indexação por pesos. O score pode agora ser de três tipos (2 tipos de score da iteração 2 (score tipo 1 e score tipo 2) e o score dos pesos (doc+query) utilizado nesta iteração).
Nesta classe são então implementados métodos auxiliares para o cálculo dos três tipos de scores e permite também, através da função printToFile(...), armazenar os resultados de cada tipo de scoring em disco através da escrita destes em ficheiros distintos.

Métodos e a sua descrição:

- **putTMInMem(String filename)**:
  - Método que instancia o método putInTM(String filename), onde é definido o caminho do ficheiro de onde vai ser lido o indexer do disco. Na nossa implementação, este ficheiro está armazenado na pasta '*outputs*' e relativamente a esta iteração, é designado por "fileout_tok.txt" caso se queira indexar em memória o ficheiro previamente gerado pelo tokenizer complexo.

Caso se queira indexar em memória o ficheiro previamente gerado pelo tokenizer simples, este ficheiro terá o nome “fileout_simple.txt”. Relativamente à segunda iteração, os ficheiros a serem lidos são designados “fileout_tok_assign2.txt” e “fileout_simple_assign2.txt” respetivamente. O motivo para que estes ficheiros tenham nomes diferentes prende-se com o facto de na segunda iteração, o formato de indexação ser ligeiramente diferente, ou seja, indexava-se por frequência de ocorrência de termos e não por peso.
    - Este método retorna uma Treemap que contém os termos e a lista de Postings carregados do ficheiro previamente gerado pelo indexer.

- **addToQueryIdDocIdScoreTM(int docId, int queryId, double score, TreeMap<Integer, HashMap<Integer,Double>> tmap):**
    - Este método é chamado no fim da função calculateRank(...) após ser calculado o valor de score para cada documento dada uma determinada query. Este método recebe como argumento o identificador do documento, o identificador da query, o valor de score e o respetivo treemap que contém o hashmap de scores. Esta função tem um comportamento similar à função addToTokenDocIdFreq(...) da primeira iteração. Caso a treemap não contenha a queryId como chave, é criada uma nova hashmap com o docId como chave e o score como valor. No entanto, caso a queryId exista como chave, é necessário retornar a hashmap associada. Se a hashmap retornada contiver como chave o docId passado como argumento, é necessário somar o valor de score anterior com o valor de score passado como argumento e substituir o score na hasmap para esse docId. Por outro lado, caso a hashmap retornada não contenha como chave o docId passado como argumento, é apenas adicionada à hashmap esse valor de score para esse docId.
    - Por fim, é associada a hashmap retornada à treemap referida anteriormente.

- **populateTMTermsInDocs(TreeMap<String, LinkedList<Posting>> tmap):**
    - Método responsável por popular a TreeMap de suporte (termInDocs) dos termos e correspondentes número de documentos onde esse termo existe.

- **getTFQuery(int freq):**
    - Método que recebe a frequência que um termo existente na query e calcula o peso do termo (valor TF) através de uma expressão que iremos apresentar mais à frente.

- **getNrDocsByTerm(String term):**
    - Método que retorna o número de documentos onde o termo passado por argumento existe.

- **calculateIdfQuery(int corpusSize, int nrDocsByTerm):**
    - Método que retorna o valor idf dado o tamanho do corpus (corpusSize) e o número de documentos onde existe um dado termo. O cálculo do idf nas queries vai ser explicado mais à frente no relatório.

- **getNormalizationValueQuery(Map<String, Integer> tmpMap, Integer corpusSize):**
    - Método que calcula e retorna o valor normalizado dos termos para uma determinada query. Uma vez mais, a fórmula deste processo de normalização vai ser explicado mais à frente no relatório.

- **calculateRank(List processedList, TreeMap<String, LinkedList<Posting>> tmap, int queryId):**
    - Método similar ao calculateScorei(...) utilizado na segunda iteração do trabalho na classe QueryProcessor, no entanto, aqui é melhorado o processo de pesquisa. Este método recebe o identificador da query e os termos processados da query, ou seja, recebe os termos da query após serem tratados com o Tokenizer usado no Indexer. Este método itera sobre toda a coleção do corpus e calcula o cálculo do score final (documento + query) (vai ser explicado mais à frente). No fim, se o score resultante do cálculo anterior for diferente de zero, é adicionado à estrutura de dados que permite agrupar queries, documentos e scores, através da invocação do método addToQueryIdDocIdScoreTM(...).

- **calculateScoreAssign2(List processedList, TreeMap<String, LinkedList<Posting>> tmap, int queryId):**
    - Este método tem um funcionamento bastante similar ao método calculateRank(...), e foi um método acrescentado para melhorar a performance da pesquisa na iteração 2.
    - Invoca o método addToQueryIdDocIdScoreTM(...) para cada tipo de score.

- **printToFile(String filename, TreeMap<Integer, HashMap<Integer,Double>> tmap, String assignment):**
    - Função que trata de escrever num ficheiro o scoring resultante. A estrutura da escrita no ficheiro obedece à seguinte forma: Qi doc_id doc_score.
    - O argumento assignment trata o diferente formato de impressão quer se trate do executável do Assignment2 ou do Assignment3.

- **putInTM(String filename):**
    - Método que tem como principal objectivo funcionar como index reader, ou seja, pretende carregar o indexer do disco em memória. Para isso, irá ler o ficheiro definido nos parâmetros de entrada linha a linha. Numa primeira fase, é feito o split por vírgulas, para aceder aos elementos de cada linha. De seguida, é realizado o split por dois pontos (":") de forma a aceder ao termo do indexer e à lista ligada de Postings associados (docId e a peso). No fim, é utilizada a estrutura de dados TreeMap<String, LinkedList<Posting>>, onde a chave será o termo do indexer lido e atualizada a LinkedList de Postings correspondentes.

- **getQueryIdDocIdScoreTM():**
    - Método que retorna a TreeMap queryIdDocIdRank.

- **getQueryIdDocIdScorei():**
    - Método que retorna a TreeMap queryIdDocIdScorei, da segunda iteração.

- **getQueryIdDocIdScoreii():**
  - Método que retorna a TreeMap queryIdDocIdScoreii, da segunda iteração.

- **sizeCorpus():**
  - Método que retorna o tamanho do corpus

4.5 – Evaluation

Esta classe é responsável por calcular as métricas de avaliação e eficiência de forma a poder inferir se os resultados foram ou não positivos e se o sistema é ou não eficiente. Como pedido foram implementadas as seguintes métricas:

- **Precision;**
- **Recall;**
- **F-measure;**
- **Mean Average Precision;**
- **Mean Precision at Rank 10;**
- **Mean Reciprocal Rank**

Quanto a métricas de teste de performance e eficiência foram implementadas:

- **Query throughput;**
- **Query latency;**

Para podermos proceder ao cálculo das métricas referidas anteriormente, é lido o ficheiro "cranfield.query.relevance.txt", onde está especificado o grau de relevância associado às diferentes queries para os documentos considerados relevantes. Na nossa implementação, foram considerados como documentos relevantes para o cálculo das métricas de avaliação todos aqueles que têm um valor de relevância inferior a 5.

Métodos e a sua descrição:

- **readQueryRelevanceFile(String filename):**
  - Método onde é processado o ficheiro dos documentos relevantes "cranfield.query.relevance.txt". É feito o split por espaços (" "), de forma a obter as 3 colunas (queryId, docId, relevância) à medida que é lida cada linha do ficheiro. Os elementos lidos são então armazenados num Map<Integer, ArrayList <Integer> >, onde a chave é o identificador da query e valor é uma ArrayList dos identificadores dos documentos existentes para essa query.

- **compute(HashMap<Integer, Double> results, int queryId, int corpus_size, int rankSize):**
    - Método onde decorre o processamento dos elementos necessários para efetuar o cálculo das métricas como os True Positives, False Positives, False Negatives e True Negatives. No início, para a queryId introduzida por argumento, são encontrados e armazenados numa ArrayList todos os documentos relevantes associados a essa query. De seguida, é verificado, percorrendo todos os docIds presentes na ArrayList, se a ArrayList dos documentos relevantes, possui o docId presente na estrutura de dados HashMap, passada como argumento. Caso contenha, é dito que é um True Positive, e a variável associada a este elemento é incrementada. Caso contrário, é um False Positive. Para o cálculo dos False Negatives, basta verificar na ArrayList dos docIds que não existem docIds relevantes. Quanto aos True Negatives, é a diferença entre o tamanho do corpus (corpus_size), True Positives, False Positives e False Negatives, ou seja, False Negatives = corpus_size - True Positives - False Positives - False Negatives.
    - Portanto, torna-se agora possível calcular a Precision, Recall e a F-measure. O cálculo da métrica Mean Precision at Rank 10, é o cálculo da média das precisões, mas para apenas 10 documentos. Quanto ao cálculo da Mean Average Precision, é feito a contagem de todos os docIds da ArrayList, e por cada True Positive encontrado é deduzida a Average Precision, somando todos os True Positives encontrados sobre o número de documentos já percorridos, sendo estes documentos relevantes e não relevantes. Depois este somatório, é dividido pelo número de documentos relevantes para essa queryId. Por fim a média é posteriormente calculada na classe RankProcessor, onde é feita a divisão pelo número total de queries. A métrica Mean Reciprocal Rank, consiste em verificar para a queryId passada, a ocorrência do primeiro True Positive sobre o número de documentos já percorridos. A média é calculada, à semelhança da Mean Average Precision, no RankProcessor.
    - Por fim, este valores são guardados numa Hasmap< String, Double>, onde a key é uma string representativa do nome do elemento ou métrica, e o value é o valor do elemento ou métrica. É retornada essa HashMap.

- **precision(Map<String, Double> valuescomputed, Boolean ranked):**
    - Neste método, são calculadas as métricas Precision e Precision at Rank 10 consoante a variável boolean passada nos argumentos de entrada. Para estas métricas são utilizados os True Positives e os False Positives da seguinte forma:

$$\text{Precision} = \frac{tp}{tp + fp}$$

- **recall(Map<String, Double> valuescomputed):**
  - Neste método, é calculado a Recall. Para estas métricas são utilizados os True Positives e os False Negatives da seguinte forma:

$$\text{Recall} = \frac{tp}{tp + fn}$$

- **fmeasure(double recall, double precision):**
  - Neste método, é calculada a métrica F-Measure que utiliza a Precision e a Recall segundo a seguinte fórmula:

$$\text{F1} = 2 * \frac{\text{precision} * \text{recall}}{\text{precision} + \text{recall}}$$

4.6 – RankProcessor

Esta classe é instanciada no programa que é executado nesta iteração (Assignment3) quando se pretende o fluxo relativo ao cálculo de pesquisa por ranking e posterior cálculo das métricas de avaliação e também no programa de execução da Assignment 2 (melhorado). Esta classe instancia o RankedRetrieval que é responsável por inicialmente carregar o indexer previamente gerado em memória, e posteriormente permitir uma pesquisa ordenada por ranking, ou seja, processa os termos de cada query e aplica o cálculo do score para cada documento tendo por base a query. Aqui, também é instanciada a classe responsável pela classificação dos resultados, a classe Evaluation. Durante a execução do programa, é definido um menu de interação com o utilizador para que este possa escolher que tipo de métrica deseja ser apresentada. Na nossa implementação, em relação a Precision, Recall e F- Measure, que são métricas processadas para cada query, optamos por apresentar um valor médio destas métricas, pois consideramos que não faz sentido apresentar 225 valores de cada métrica para cada query.

Métodos e a sua descrição:

- **process(String fileType, String assignment):**
  - Este método é responsável pelo processo do cálculo do score, onde começa por carregar o indexer previamente gerado em memória, e posteriormente processa os termos de cada query e aplica o cálculo do score para cada documento tendo por base a query e o tokenizer aplicado.

- **printMenu():**
  - Método responsável por imprimir o menu das opções que vão do 1 ao 9 (de acordo com as métricas especificadas).

- **eval(String assignment):**
  - Neste método é processado o fluxo do processo de classificação dos resultados, ou seja, é instanciada a classe Evaluation e invocados os seus métodos para as diversas métricas. Assim, o método começa por carregar a estrutura TreeMap<Integer, HashMap<Integer,Double>> resultante do cálculo do score. De seguida, é lido o ficheiro "cranfield.query.relevance.txt" para definir os documentos relevantes para cada query. Depois para cada query da estrutura,TreeMap<Integer, HashMap<Integer, Double>>, anteriormente referenciada, é processado o cálculo das métricas: Precision, Recall, F-Measure, Mean Average Precision, Mean Precision at Rank 10 e Mean Reciprocal Rank. Neste método apenas são calculadas as médias para todas estas métricas de classificação.

universidade de aveiro

5 - Diagrama geral de fluxos

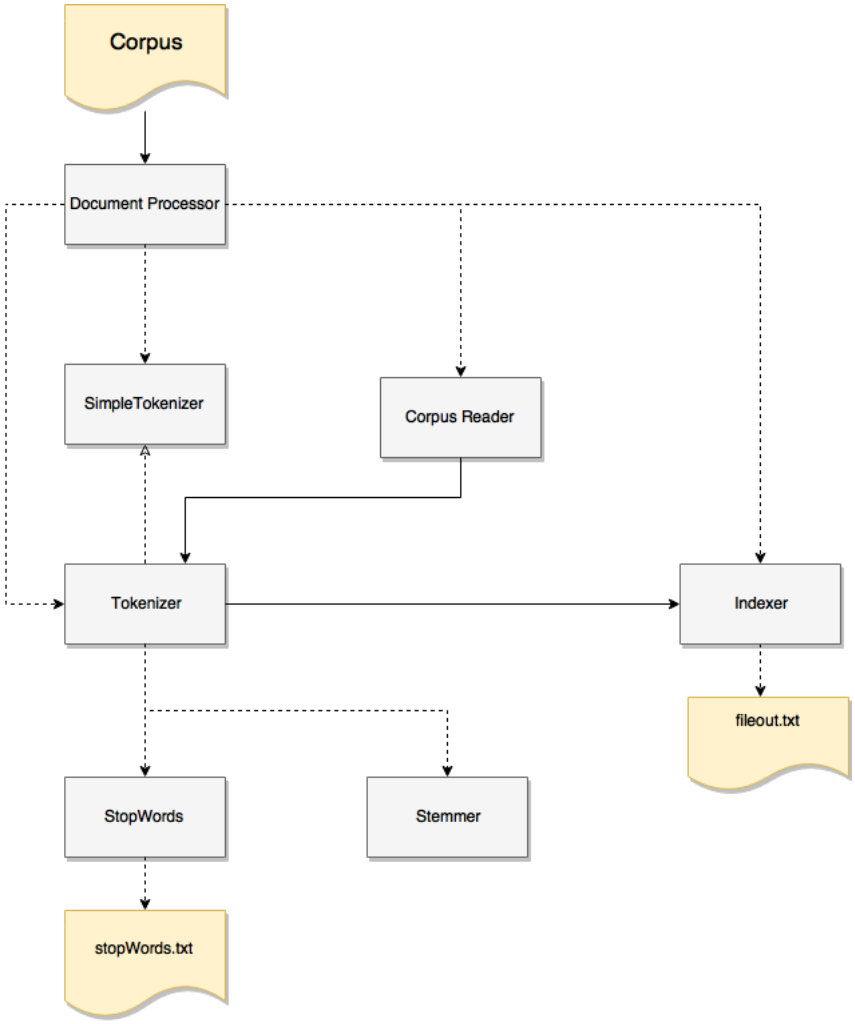


Figura 4 - Imagem que representa o fluxo de execução do programa através do processo de indexação por pesos

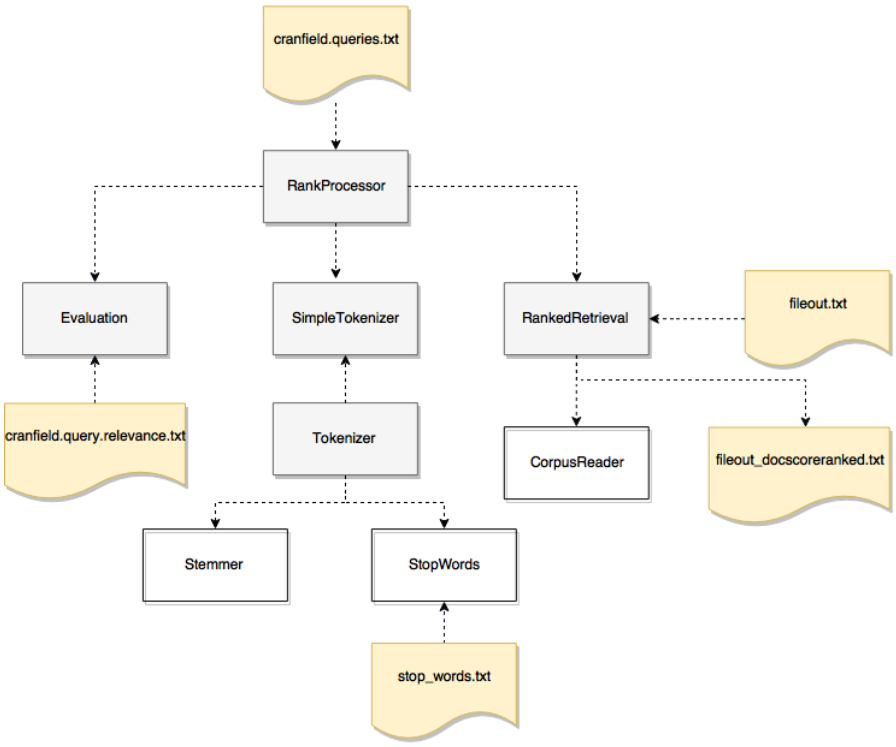


Figura 5 - Imagem que representa o fluxo de execução do programa através do processo de pesquisa por ranking

A figura anterior mostra-nos de forma esquemática o fluxo de dados no nosso projeto. Caso seja desejada a indexação dos termos, o fluxo do programa será o esquematizado em (4), tendo por base o DocumentProcessor devidamente detalhado na iteração 1. Caso seja desejado a pesquisa por ranking, o fluxo do programa será o esquematizado em (5), encontrando-se detalhado na descrição da classe RankProcessor.

6 – Implementação do cálculo tf-idf

Cálculo do score nos documentos:

No final do processamento de cada documento, é então calculado o peso tf-idf para cada termo existente. Durante a indexação, é usada a estratégia **lnc.ltc** que consiste em assumir que o idf é 1, portanto, para o cálculo de peso apenas vai ser considerado o valor do tf normalizado. Para isso, é necessário aceder à estrutura de dados que nos permite aceder à frequência com que cada termo aparece no documento. Dado que o número de ocorrências de um termo t num dado documento d, é representado por tf(t,d), o peso de um termo t num dado documento d, pode ser calculado com base no sistema de equações imediatamente abaixo:

$$w_{t,d} = \begin{cases} 1 + \log_{10} \mathrm{tf}_{t,d}, & \text{if } \mathrm{tf}_{t,d} > 0 \\ 0, & \text{otherwise} \end{cases}$$

Figura 6 – Fórmula usada para calcular o valor de tf nos documentos

Após calcular o peso de um termo t num dado documento t, é necessário proceder à normalização do seu valor para posteriormente indexação. A valor de normalização é obtido a partir da seguinte fórmula, também vista nas aulas teóricas:

$$\|\vec{x}\|_2 = \sqrt{\sum_i x_i^2}$$

Figura 7 – Fórmula usada para calcular o valor normalizado

Em que x representa o valor obtido pelo sistema de equações em (6) para cada termo t num dado documento d. O valor da normalização é obtido através da raiz quadrada da soma de todos os termos t do documento d ao quadrado.

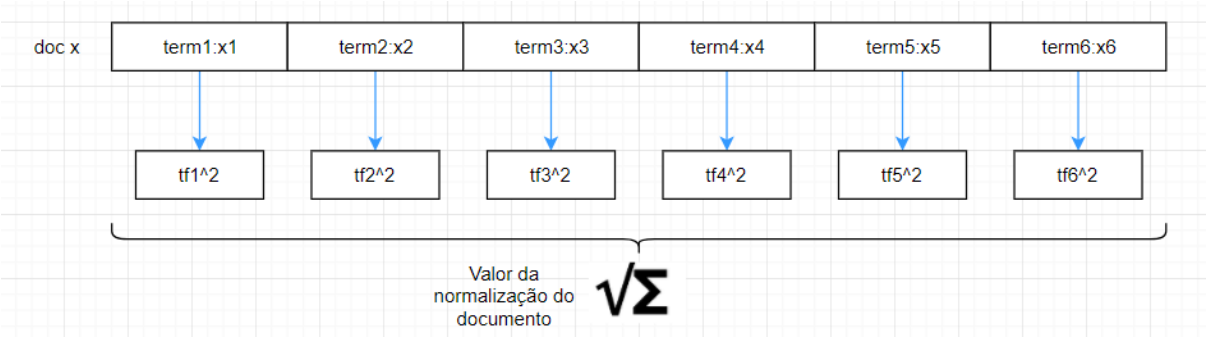


Figura 8 – Representação ilustrativa do cálculo do valor da normalização do documento

Cálculo do score nas queries:

Durante o processamento das queries, já não é possível assumir o valor de idf a 1, portanto, para calcular o peso tf-idf da query, é usada a seguinte expressão:

$$\mathrm{w}_{t,d} = (1 + \log \mathrm{tf}_{t,d}) \times \log_{10}(N / \mathrm{df}_t)$$

Figura 9 – Fórmula usada para o cálculo do peso tf-idf nas queries

onde o valor de tf da query corresponde à frequência do termo t na query q, o N corresponde ao tamanho do corpus utilizado, 1400 no nosso caso, e df corresponde ao número de documentos onde o termo t da query q existe.

Seguidamente, o valor do peso tf-idf (score) para cada termo da query será dividido pelo valor da normalização da query. O processo para obtenção do valor de normalização da query encontra-se ilustrado imediatamente abaixo:

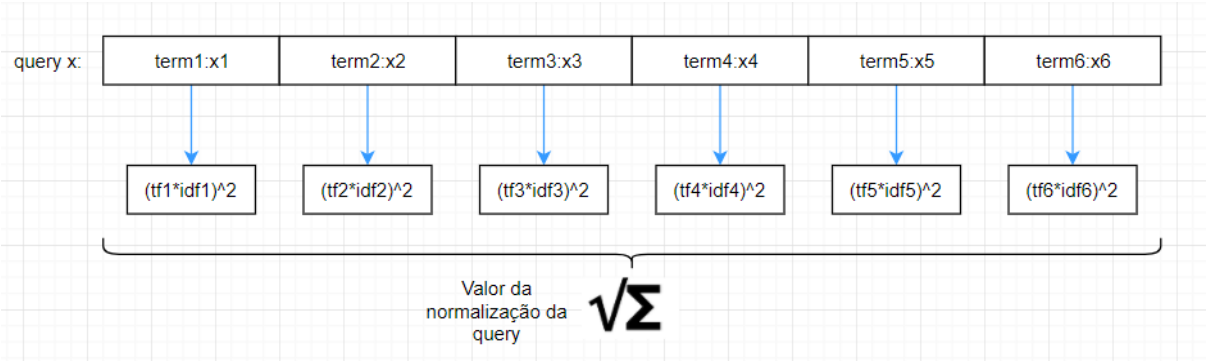


Figura 10 – Representação ilustrativa do cálculo do valor da normalização da query

O valor de normalização da query corresponderá à raiz quadrada da soma do valor dos scores ao quadrado de cada termo da query.

Cálculo do score documento + query:

Finalmente, para obter o ranking de pesquisa é usada a seguinte expressão para cálculo do score:

$$Scores[d] += w_{t,d} \times w_{t,q}$$

Figura 11 – Fórmula usada para cálculo do score (doc+query)

onde o W(t,d) corresponde ao valor do score do documento (explicado anteriormente) e W(t,q) corresponde ao valor do score na query (explicado anteriormente).

7 - Estrutura do código

A estrutura do nosso projeto *Maven* encontra-se organizada da seguinte forma:

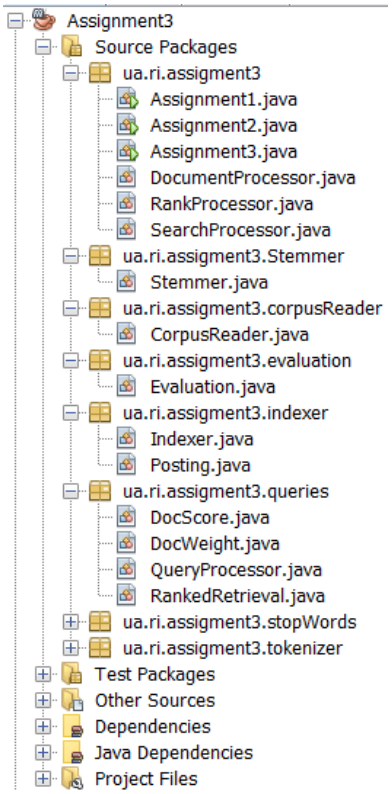


Figura 12 - Descrição da estrutura do nosso projeto Maven

No diretório raiz do nosso projeto Maven é possível encontrar três diretórios principais:

• **data:** Este diretório, tal como na segunda iteração do trabalho, está dividido em subdiretórios. No subdiretório "cranfield", encontra-se a coleção de corpus utilizados. Para esta fase do projeto apenas foi utilizado o cranfield. No subdiretório "queries", encontra-se o ficheiro das queries considerado para pesquisa, "cranfield.queries.txt". Ainda neste subdiretório, encontra-se agora também o ficheiro "cranfield.query.relevance.txt", relevante para o processo de cálculo das métricas de avaliação.
Na raiz do diretório, existe um ficheiro stopWords.txt que contém todas as stopwords consideradas.

• **output:** Este diretório vai conter o(s) ficheiro(s) gerado(s) no processo de indexação. Este(s) ficheiro(s) terá(ão) o nome "fileout_simple.txt" caso tenha sido utilizado o simple tokenizer e o nome "fileout_tok.txt" no caso de ter sido utilizado o tokenizer complexo, tal como nas iterações anteriores. Os ficheiros gerados nesta terceira iteração vão também ser colocados neste diretório. É gerado um novo ficheiro para cada tipo de tokenizer utilizado. O ficheiro terá o nome "fileout_docscoreranked_tok_3.txt" no caso de ser lido o ficheiro indexado pelo

tokenizer complexo. No caso de ser lido o ficheiro indexado pelo tokenizer simples, o ficheiro terá o nome “fileout_docscoreranked_simple_3.txt”. Relativamente às melhorias do Assignment2, os ficheiros previamente gerados no processo de indexação dessa iteração, terão o nome “fileout_simple_assign2.txt” e “fileout_tok_assign2.txt”. Os ficheiros gerados no processo de pesquisa terão o nome “fileout_docscoreranked_tok_2_i.txt” para o tipo de score 1, e o nome “fileout_docscoreranked_tok_2_ii.txt”, para o tipo de score 2.

• **Dependencies:** Neste diretório estão colocados as bibliotecas compiladas necessárias para este projeto (Para este caso foi usado o jar do Snowball). (Relativo à primeira iteração do trabalho).

8- Bibliotecas Externas

The Porter stemming algorithm: http://snowball.tartarus.org/.

De referir, que esta biblioteca externa apenas foi utilizada na primeira iteração do projeto.

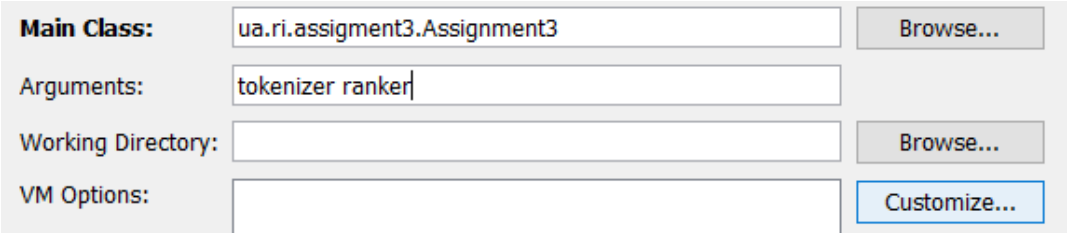
9- Execução

Para executar o nosso projeto é necessário a seguinte abordagem:

- Compilar e executar o nosso projeto recorrendo ao Netbeans.
- Podem ser passados, por argumento, o tipo de tokenizer que se pretende correr como 1º argumento e o modo de fluxo do programa como 2º argumento, ou seja, se se pretende efetuar indexação ou se se pretende efetuar pesquisa por ranking e cálculo das métricas associadas. Modo de uso**: java Assignment3 <tokenizerType> <flowType>**. No entanto, por defeito, é considerado o tipo de Tokenizer complexo e processo de indexação como modo de fluxo do programa, por isso, apenas é necessário correr o comando: **java Assignment3**. De notar que o tipo de tokenizer apenas aceita a string <tokenizer> que corresponde ao Tokenizer complexo ou a string <simple> que corresponde ao Simple Tokenizer. Caso seja introduzida uma outra string, nenhum tipo de tokenizer será reconhecido e será apresentada uma mensagem no terminal com o formato que deve ser utilizado para correr o programa. O tipo de fluxo do programa tem um comportamento similar, ou seja, o tipo de fluxo do programa apenas aceita a string <indexer> que corresponde ao processo de indexação ou a string <ranker> que corresponde ao processo de pesquisa por ranking. Caso seja introduzida uma outra string, nenhum tipo de fluxo de programa será

reconhecido e será apresentada uma mensagem no terminal com o formato que deve ser utilizado para correr o programa.

Exemplo de modo de execução do programa:



Para correr o programa da segunda iteração:

- Podem ser passados, por argumento, o tipo de tokenizer que se pretende correr como 1º argumento. Modo de uso: **java Assignment2 <tokenizerType>.** No entanto, por defeito, é considerado o tipo de Tokenizer complexo, por isso, apenas é necessário correr o comando: **java Assignment2**. De notar que o tipo de tokenizer apenas aceita a string <tokenizer> que corresponde ao Tokenizer complexo ou a string <simple> que corresponde ao Simple Tokenizer. Caso seja introduzida uma outra string, nenhum tipo de tokenizer será reconhecido e será apresentada uma mensagem no terminal com o formato que deve ser utilizado para correr o programa.

10- Resultados

Para obtermos os tempos que do processo de indexação e respetiva geração e escrita em ficheiro, foi utilizado o seguinte hardware:

• **Processador**: Intel(R) Core(TM) i7-4700MQ CPU @ 2.40GHz

• **Memória**: 8.00 GB

• **Sistema Operativo**: Windows 10, 64bits

Para o corpus fornecido sample cranfield.zip e cranfield.queries.txt, foram obtidos os seguintes tempos:

Simulação	Tempo de execução (s)
1	1.284
2	1.367
3	1.369
4	1.369

O tempo médio de execução foi: 1.3465 segundos.

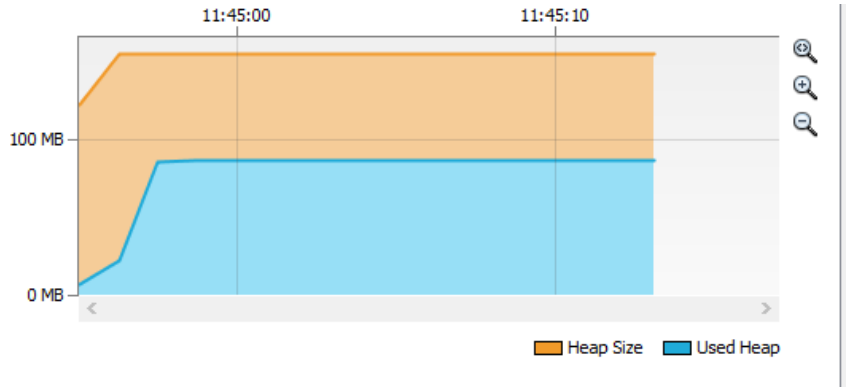


Figura 13 - Descrição de medidas de desempenho: CPU, Memória e Garbage Collection

11- Resultados das métricas de classificação e de eficiência

Como foi referido anteriormente, para efetuar os cálculos das métricas de classificação e de eficiência foram considerados para cada query, documentos em que o seu grau de relevância fossem menor do que 5. Serão apresentados apenas os valores médios e não os resultados para cada uma das 225 queries. De seguida, serão apresentados os resultados tanto do Assignment 2 como do Assignment 3 usando o Tokenizer complexo.

Assignment2

Métricas de Classificação	
Mean Precision	0,008311
Mean Recall	0,959918
Mean F-measure	1,785054
Mean Average Precision	0,019062
Mean Precision at Rank 10	0,230000
Mean Reciprocal Rank	0,032326

Métricas de Eficiência	
Query throughput (queries/s)	1275
Median query latency (s)	784329

Assignment3

Métricas de Classificação	
Mean Precision	0,008718
Mean Recall	0,955038
Mean F-measure	1,895241
Mean Average Precision	0,019615
Mean Precision at Rank 10	0,240000
Mean Reciprocal Rank	0,033202

Métricas de Eficiência	
Query throughput (queries/s)	1202
Median query latency (s)	831701

12 - Repositório de desenvolvimento

Para o desenvolvimento deste trabalho foi utilizado a plataforma CodeUA de forma a potenciar o trabalho de equipa. O repositório é o seguinte: http://code.ua.pt/projects/ir/repository.

13 - Conclusão

No final da elaboração do Assignment 3 e do respetivo relatório, é hora de fazer uma introspeção do que foi feito até ao momento. Como já foi referido, houve uma ligeira alteração no Indexer de modo a agora utilizar o mecanismo tf-idf seguindo a estratégia lnc.ltc. Outro aspeto que foi verificado foi a relevância das estruturas de dados escolhidas nos tempos de performance quando do cálculo dos resultados das métricas de eficiência. Inicialmente, usámos na nossa implementação a seguinte estrutura: TreeMap<Integer, LinkedList<DocWeight>> para pesquisa, mas verificamos que a utilização de uma LinkedList era ineficiente quando utilizada no RankProcessor para o cálculo do score, uma vez para cada query, tinham de se percorrer todos os elementos da LinkedList para atualizarmos o valor do score. Assim, optámos por usar a estrutura de dados HashMap<Integer, Double> na pesquisa, onde tal processo de iteração sobre todos os valores não se verifica pois com uma estrutura deste tipo conseguimos aceder aos elementos que queremos diretamente. Os resultados foram significativamente melhores, processando mais queries por segundo e diminuindo a latência por query. Assim, o cálculo das métricas de classificação e de eficiência teve um papel determinante no modo como elaboramos a nossa solução.
Ainda outro ponto que é importante referir, é o facto de duas métricas de classificação, Mean Average Precision e Mean Reciprocal Rank, darem valores muito mais baixos do que seria expectável. Apesar desses valores serem muitos baixos, achamos que estamos a proceder ao seu cálculo de forma correta, o que os leva a considerar que tais resultados podem ser devidos ao processo de tokenização que estamos a utilizar.
Por fim, é relevante referir que durante a elaboração do Assignment 3, optámos sempre que possível por uma solução modular e achámos por bem melhorar o trabalho anterior realizado na segunda iteração de modo a conseguirmos melhores resultados.