universidade de aveiro

Departamento de Eletrónica, Telecomunicações e Informática

Information Retrieval Engine

Implement a ranked retrieval method.
Tarefa 4

Curso	[8240] MI em Engenharia de Computadores e Telemática
Disciplina	[42596] Recuperação de Informação
Ano letivo	2016/2017

Alunos	[68021] Gabriel Vieira, gabriel.vieira@ua.pt
	[68779] Rui Oliveira, ruipedrooliveira@ua.pt
Grupo	5
Docente	Professor Sérgio Matos, aleixomatos@ua.pt

Aveiro, 21 de Dezembro de 2016

Conteúdo

1 Enquadramento

Pretende-se através deste relatório expor sob forma escrita, o nosso desempenho e objetivos alcançados na primeira tarefa do trabalho prático desenvolvido no âmbito da unidade curricular de Recuperação de Informação.

Neste trabalho pretende-se criar um mecanismo de recuperação de informação. Este será composto de vários módulos, são eles: *corpus reader* , *document processor*, *tokenizer*, *indexer*, *memory management*, *stop words*, *searcher* e *ranker*.

Na primeira iteração do projecto, prodeceu-se à sua modelação, definindo e descrevendo as classes, métodos (principais) e o fluxo de dados.

Na segunda iteração do projeto pôs-se em prática a execução do trabalho atendendo à modelação da primeira parte, explicando também com mais detalhe cada uma das classes e suas funções e todo o fluxo existente neste trabalho.

Na terceira iteração do projecto alterou-se o indexer baseado no modelo vector-space, usando o calculo do peso de um termo num documento: o **TF-IDF** e a estratégia **lnc.ltc** conforme está descrita nos slides da componente teórica.

Nesta quarta e última iteração do projecto prentede-se implementar um método de ranking de pesquisa. Ou seja, ordenar os resultados da pesquisa obtidos através da query introduzida pelo utilizador.

O fluxo de operações aplicado a um conjuntos de documentos apresenta-se no diagrama seguinte.

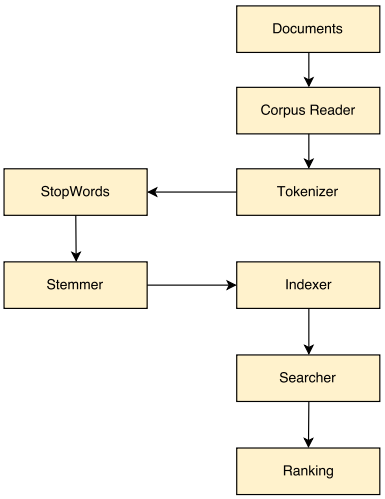

Figura 1: Diagrama geral

2 Arquitetura geral

Nesta secção iremos apresentar todos os módulos do nosso projeto e um esquema que representa o fluxo geral de execução do mesmo.

2.1 Componentes

2.1.1 *CorpusReader*

Este módulo trata da abertura, leitura e pré-processamento de cada um dos documentos presentes no conjunto de documentos a processar.

2.1.2 *Tokenizer*

Este módulo tem como função formar um conjunto de tokens (Array de tokens) com valor semântico recorrendo à sequência de caracteres que é fornecida pelo módulo *corpus reader*. Numa fase inicial, após o Tokenizer os termos serão imediatamente indexados através do Indexer sem sofrer qualquer processamento, apenas mais tarde e após garantirmos o total funcionamento (Tokenizer+Indexer), o processo passará pelos dois módulos que a seguir apresentamos.

2.1.3 *StopWords*

Esta classe é encarregue de ler um ficheiro de *stop words* fornecido e adiciona-as a uma lista. Esta lista é depois mais tarde usada para se comparar com um array de tokens e verificar se uma palavra é ignorada ou não.

2.1.4 *Stemmer*

Este módulo tem como objetivo tratar as variações morfológicas das palavras, permitindo que apenas a palavra principal seja indexada, pois na maioria dos casos têm significados semelhantes. A implementação do stemmer irá recorrer ao *The Porter stemming algorithm* disponível através do link `http://snowball.tartarus.org/algorithms/porter/stemmer.html`

2.1.5 *Indexer*

Consiste numa estrutura de dados que tem como objetivo aumentar o desempenho da pesquisa, recorrendo à indexação dos termos de um dado conjunto de documentos. Para esta parte do trabalho, aproveitando a indexação feita na parte anterior (Inverted index) era pedido que se adapta-se a indexação baseada no modelo vector-space, usando o esquema de cálculo

do peso de um termo num documento, o esquema **TF-IDF** e a estratégia lnc.ltc, de acordo com os slides da unidade curricular.

No indexer booleano do trabalho anterior eram usadas duas estruturas:

- **TokenFreqMap**: `HashMap<String, Integer>`

  Permitia armazenar a frequência com que o um determinado token aparecia em todos os documento.

- **TokenIDDocFreqMap**: `HashMap<String<HashMap<Integer, Integer>`

  Permitia armazenar a frequência com que o um determinado token aparecia num determinado documento.

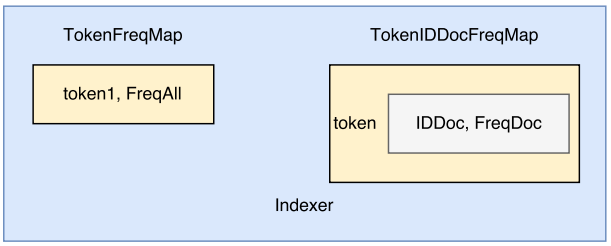

Figura 2: Estrutura de dados usado para o indexer booleano - tarefa anterior

Nesta fase do projeto são usadas três estruturas distintas, sendo que duas delas são consideradas auxiliares à estrutura final. São elas:

- `HashMap<String, Integer> tokenFreqDocMap`

  Este tipo de dados herda as características de uma *HashMap*, em que a *key* corresponde ao *token* e o *value* corresponde à frequência com que o token aparece num determinado documento.

  corresponde a uma outra *HashMap*. Nessa *HashMap* a *key* corresponde ao identificador do documento e o *value* ao número de ocorrências desse *token* no documento.

- `HashMap<String, String> tokenPosDocMap`

  Este tipo de dados herda as características de uma *HashMap*, em que a *key* corresponde ao *token* e o *value* corresponde a uma String que permite identificar as diferentes posições em que o token ocorre num determinado documento.

- `TokenPost extends HashMap<String, HashMap<Integer, String>>`

  Este tipo de dados herda as características de uma *HashMap*, em que a *key* corresponde ao *token* e o *value* a uma outra *HashMap*.

  Nessa *HashMap* a *key* corresponde ao identificador do documento e o *value* a uma String onde é possivel identificar o peso normalizado do token (será abordado mais à frente) no documento e as posições em que ele ocorre no determinado documento.

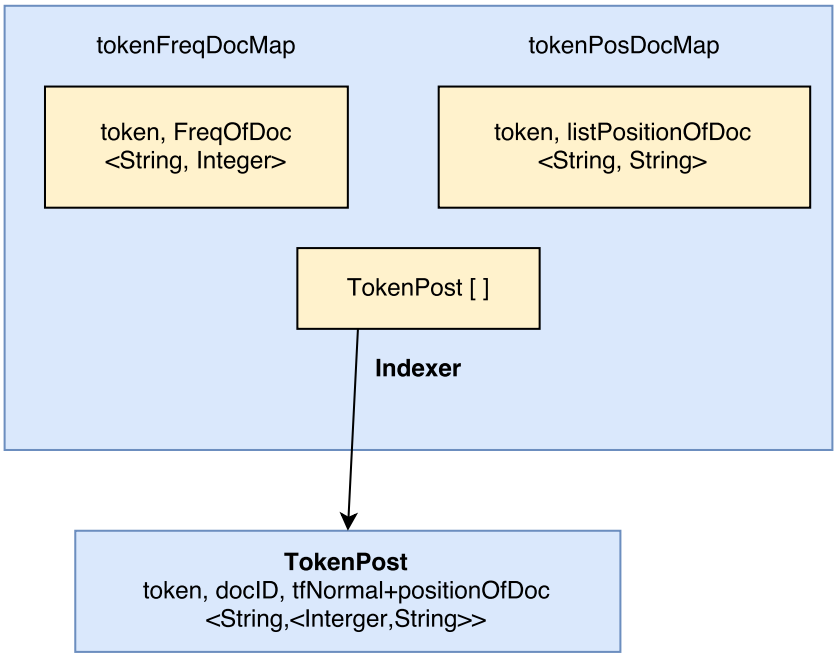

Figura 3: Estrutura de dados usado para o indexer TF-IDF

2.1.6 *MemoryManagement*

Este módulo permitirá fazer uma gestão da memória, sendo que quando a memoria virtual excede certo limite, a informação é armazenada em disco. Através deste módulo é possível aceder à quantidade de memoria virtual consumida pela JVM.

Ao contrário das fases anteriores do projeto, o mecanismo de gestão de memória já foi tida em conta.

2.1.7 *DocumentProcessor*

Este módulo tem como objetivo instanciar todo o fluxo associado à execução de um documento. Para além disso, o *document processor* irá iterar sob todos os documentos existentes num dado conjunto de documentos.

2.1.8 *SearchProcessor*

Este módulo tem como objectivo instanciar todo o fluxo associado à execução da pesquisa de um ou vários termos sobre o índice resultante, cujo foi escrito em disco no trabalho anterior. A pesquisa vai iterar sobre os ficheiros correspondentes à primeira letra do termo e vai retornar os resultados da pesquisa ordenados consoante o *score* obtido.

2.1.9 *IndexerResults*

Este módulo tem como objetivo fazer as pesquisas necessárias aos ficheiros resultantes da indexação, bem como filtrar o resultado das referidas pesquisas. Foram desenvolvidos na última parte do trabalho dois métodos de pesquisa: simples e por frase, ao qual serão explicados com mais detalhe na subsecção *Classes e métodos* deste relatório.

2.1.10 *Query*

Tipo de dados que define a criação de uma query, especificando o conteúdo da query, tipo e proximidade. Numa primeira fase apenas iremos implementar pesquisa simples por query.

2.1.11 *QueryProcessing*

Este módulo tem como objetivo guardar os dados referentes à progressão da pesquisa.

2.2 Diagrama geral

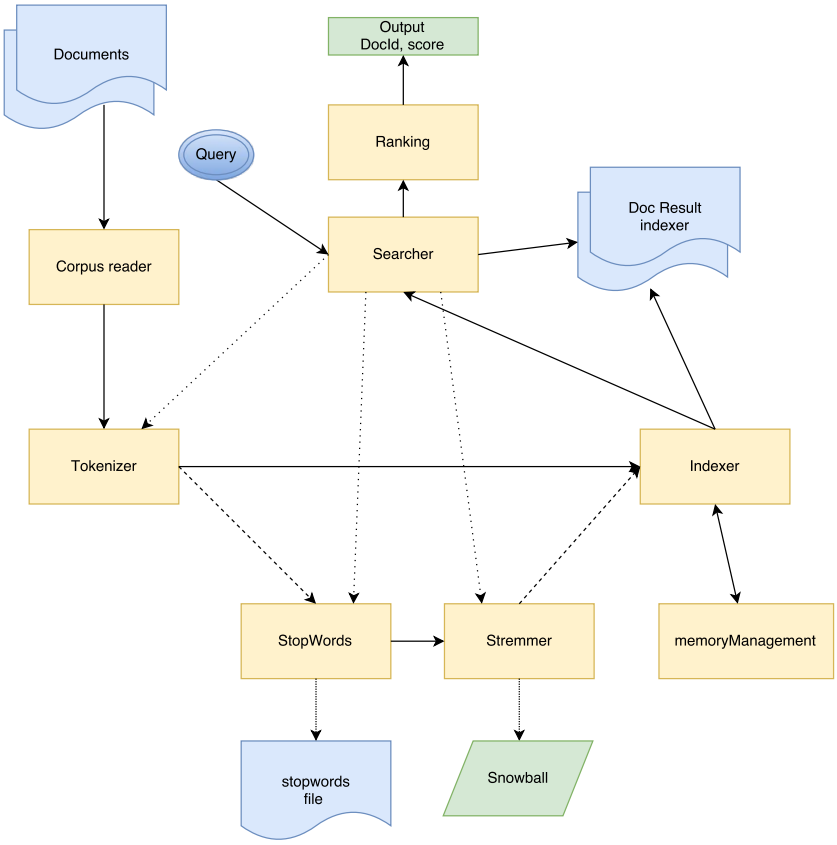

Figura 4: Fluxo geral de execução

A figura anterior mostra-nos de forma esquemática o fluxo de dados no nosso projeto.

Nesta fase do trabalho, o corpusReader irá encarregar-se de ler todos os ficheiros com o formato CSV que se encontram num determinado diretório e de efetuar o seu pré-processamento. O texto pré-processado é dirigido para o nosso Tokenizer que se encarrega de transformar uma String num conjunto de termos válidos (tokens).

Como foi efetuado na fase anterior do trabalho, após a leitura e filtragem do texto, pretendemos que todos os termos sejam válidos, de modo a garantirmos o funcionamento do indexer, sendo enviados todos os tokens existentes no array de tokens. Posteriormente, iremos usar técnicas de stemming e um filtro de stopwords (setas representadas a tracejado).

O filtro de stopwords irá reduzir o array de tokens que anteriormente referimos, uma vez que irá comparar todos os candidatos a token com as

palavras da lista de stopwords[1]. Todas as palavras que não são stopwords são posteriormente passados por um Porter Stemmer que converte palavras morfologicamente distintas com o mesmo significado para um token comum.

Relativamente ao Indexer, este é o componente mais complexo do nosso projeto. Tal como descrito anteriormente, o indexer faz uso de três estruturas sendo duas delas auxiliares. A estrutura final permite armazenar os tokens, o identificador do documento em que este aparece e respectiva associação ao peso (TF) mais a(s) posição(ões) em que o token aparece no documento. Após a indexação, são gerados vários ficheiros com uma organização especifica.

Em relação à pesquisa, o seu pipeline é iniciado pelo módulo SearcherProcessor. Quando este inicia, é carregado para memória o conteúdo do ficheiro resultante do Pipeline de indexação que consiste no mapeamento, entre a identificação do documento utilizada na indexação e a real identificação do documento. Quando o Pipeline está preparado para receber as queries vindas do utilizador, este bloqueia e aguarda para que estas sejam recebidas. Assim que uma query é recebida, esta passa pelo Tokenizer, onde o seu conteúdo é tratado e resultando daí um conjunto de tokens. Este conjunto pode ou não passar pelo filtro de StopWords bem como pelo Porter Stemmer. Antes da query ser iniciada, é criado um objecto do tipo Query para guardar o tipo e conteúdo da query feita pelo utilizador, passando o fluxo do pipeline do tokenizer para Query. Depois de tudo isto, cada query é enviada para um HashMap que vai guardar todos os termos da Query a serem pesquisados. No final os resultados são ordenados de acordo com o score obtido na pesquisa.

[1]Palavra que é removida antes ou após o processamento de um texto em linguagem natural

3 Alterações à entrega anterior

Neste relatório final do trabalho, focamos-nos no processo de pesquisa. Com os termos já indexados e calculado o seu TF e por sua vez o indexer já escrito em disco (trabalho da entrega anterior), era pedido que se procedesse ao seu carregamento e se efectuasse uma pesquisa dos termos neste indexer.

Esta pesquisa teria que mostrar os resultados ordenados. Para estes resultados já virem ordenados, procedeu-se então a um método de ranking da pesquisa.

Foi então usada nesta parte do trabalho, conforme expresso no primeiro relatório do trabalho, um novo módulo para o pipeline de pesquisa de nome *SearchProcessor* ao qual carrega os dados já indexados, bem como a lista de stopwords e a memória máxima a ser usada no processo de pesquisa. No início deste processo é pedido ao utilizador que digite uma query com o termos ou os termos a pesquisar. Depois disso, os termos da query introduzida vão passar por um processo semelhante ao que os termos da parte de indexação do trabalho passaram, ou seja, é feita a sua tokenização, depois verificado se o termo é válido, seguindo-se a verificação com a lista de stopwords e por fim feito o seu stemming. Este processo tem de ser feito também à query para garantir que esta não retornar resultados inválidos, pois todos os termos indexados são resultantes do seu stemming. De referir que este módulo só termina a sua execução quando o utilizador escreve a query "*@exit*".

Depois disto são então adicionados os termos a um HashMap que guarda os termos a serem pesquisados. De seguida, de acordo com o tipo de query escolhida pelo utilizador (simples ou por frase) são pesquisados estes termos no respectivo ficheiro indexado de acordo com a primeira letra e é obtido o seu score. Score este que vai interessar na ordenação dos resultados da respectiva query.

4 Diagrama de classes

Na figura seguinte apresenta-se o diagrama de classes do nosso projeto.

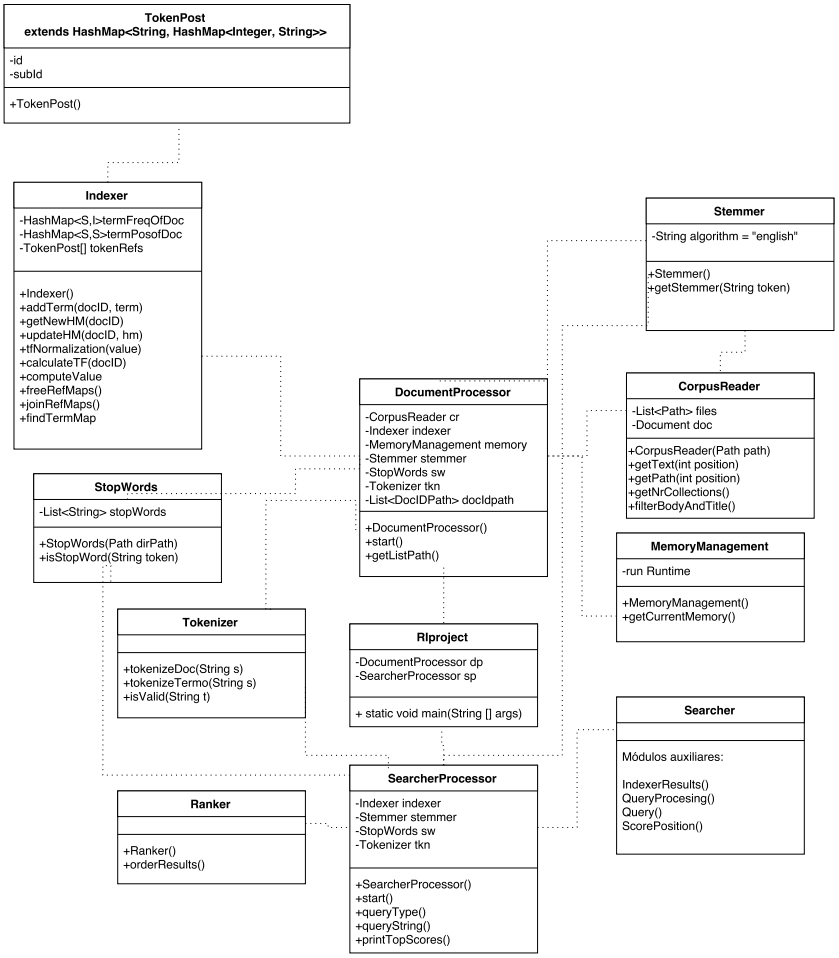

Figura 5: Diagrama de classes

4.1 Classes e métodos

4.1.1 RIproject (main)

Classe principal do projeto. É aqui onde é instanciada a classe SearchProcessor que iremos descrever de seguida. A função main recebe como argumento os seguintes parâmetros:

- Caminho para o indexer a ser carregado em memória.

- Caminho para o ficheiro de stopwords.
- Memoria máxima que é possível usar para o processamento.

4.1.2 DocumentProcessor

Principais métodos e sua descrição:

- **start()**: método responsável pela iniciação do pipeline de todo o processo. Este método é invocado pela main classe para se iniciar o processo de iteração na coleção de ficheiros existente.
- **getListPath()**: método que retorna a lista dos caminhos onde os ficheiros de extensão "csv"se situam.

Descrição: Neste trabalho, esta classe começa por ler os documentos que estão presentes no directório especificado. Percorre os documentos e com o auxílio da classe *CorpusReader* obtém o texto dos documentos presentes em cada ficheiro. Depois do *CorpusReader* ler todo o texto dos documentos com o formato CSV e fazer a sua filtragem, este módulo vai de seguida enviar todo o texto para o *Tokenizer* para este "tokenizar"todo o texto, guardando cada termo num array de strings. Depois de tokenizar o texto, o Document Processor vai percorrer a coleção dos ficheiros e obter o ID de cada documento, para melhor especificar quais os termos que pertencem a um dado documento. Após obter o ID de cada documento, este módulo vai "tokenizar"apenas os termos de cada documento (já filtrados) usando o método *tokenizeTermo* da classe *Tokenizer*, guarda cada um deles num array de strings e depois vai percorrer esse array. Ao percorrer esse array, vai verificar se cada token é valido, ou seja, no nosso caso para ser válido tem de ter mais de 2 caracteres e conter texto. Se o termo não for válido é descartado. Caso seja, então vai-se verificar se esse termo está contido na lista de stop words, usando o método *isStopWord* da classe *StopWord* (explicado mais à frente). Se esse termo constar na lista de stop words, então é descartado, caso contrário vai ser feito o seu stemming usando o módulo *Stemmer* que vai ser responsável por obter o stemmer especificado, que neste caso é o inglês. Depois de tudo isto, então o termo é indexado. É também neste método que à medida que cada termo é processado a memória utilizada é verificada, caso esteja dentro do parâmetro estipulado é invocada o método `freeRefMaps()`. No final da leitura de cada documento é calculado o peso do TF através do método `calculateTF()`. No final de todo o processamento é libertada a memória e invocado o método `freeRefMaps()`. Por fim, todos os ficheiros gerados do indexar são juntos.

4.1.3 CorpusReader

Principais métodos:

- **getText()**: Sempre que o DocumentProcessor inicia a leitura de um documento, solicita a este método o conteúdo de um novo documento. O novo ficheiro é lido, e é feita a filtragem do title e do body retornando para o DocumentProcessor o texto todo filtrado.
- **getPath(int position)**: método que é responsável por retornar o path de cada ficheiro de acordo com a posição que este se encontra no directório.
- **getNrCollections()**: método que retorna a número total de ficheiros a serem lidos.
- **filterBodyAndTitle():** recebe os título e o body do documento e faz a sua filtragem, como a remoção de tags HTML e todo o texto, excertos de código que se encontram dentro desta.

Descrição: O Corpus Reader vai ler cada documento e fazer a filtragem do texto linha a linha, usando o parser do JSoup que depois ajuda a eliminar todo o texto HTML e excertos de código que se encontram entre as tags `<pre></pre>` e passa depois somente o texto que importa à função `getText` e esta faz a filtragem do texto de acordo com a que foi feita na parte anterior do trabalho. Para além disto, o Corpus Reader tem um método que retorna o número de coleções (número de ficheiros CSV a serem lidos no directório), bem como um método que retorna uma string com o método.

4.1.4 Tokenizer

Principais métodos:

- **tokenizeTermo(String s)**: faz uso de uma expressão regular para trocar o conteúdo dos caracteres, depois faz-se a separação da String num array de Strings termo a termo.
- **isValid(String t)**: valida o conteúdo final dos tokens, andes da sua indexação.

Descrição: Este módulo faz a tokenização do texto e dos termos. Usando o método *tokenizeDoc* a tokenização do texto é feita linha a linha e depois guardado num array de Strings, já o método *tokenizeTermo* a tokenização

é feita termo a termo e cada termo é guardado num array de strings. Para além destas funções, este módulo tem ainda um método que verifica se um termo é ou não válido de nome *isValid* ao qual já foi explicada a sua função na descrição do Document Processor.

4.1.5 StopWords

Principais métodos:

- **isStopWord(String token)**: verifica se o token que irá ser indexado é ou não uma stop word. Se for uma stop word, então o termo é ignorado, caso contrário prossegue para a indexação.
- **getSize()**: retorna o tamanho da lista de stopwords.
- **getStopWords()**: retorna a lista de stop words.

Descrição: Este módulo inicialmente vai ler a lista de Stop Words, disponibilizada pelo docente. Este módulo depois contém um método booleano de nome *isStopWord* que vai verificar se o termo que é passado por argumento está contido na lista de stop words ou não. Caso não esteja contido informa, o Document Processor que pode avançar para a próxima etapa, caso contrário é descartado.

4.1.6 Stemmer

Principais métodos:

- **getStemmer(String token)**: pretende identifiar variações morfológicas de palavras e transforma-las numa mesma palavra comum. Este método recebe um token e caso seja possível de transformação essa operação é efetuada. É retornado o token transformado ou original (recebido como argumento).
- **getAlgorithm()**: retorna o algoritmo de stemming a ser usado. Neste caso é usado o "english".

Descrição: Este módulo vai obter o stemmer para os termos usando Port Stemmer da biblioteca *SnowBall*. No módulo, foi necessário especificar o algoritmo a usar, que neste caso foi o "englishStemmer"já com todas as regras de stemming implementadas, sendo apenas necessário saber onde e como usar. Para que esta biblioteca seja reconhecida, foi necessário incluir no projecto e nas dependências do Maven o jar do Snowball (`Snowball.jar`).

4.1.7 Indexer

Principais métodos:

- **addTerm()**: método usado para adicionar termos às estruturas de dados, caso estes não existiam. Se existirem a estrutura de dados é actualizada com a nova informação recolhida.
- **getNewHM(int docId)**: caso uma key não tenha um value, este é adicionado. Ou seja, se aparece um termo que ainda não tem nada na sua lista de postings, então é adicionada a esta lista o ID do documento que contém este token.
- **updateHM(int docId, hashmap<integer,integer> hm)**: caso um termo aparece mais que nos documentos, então é adicionado o ID do documento correspondente à lista de postings.
- **calculateTF()**: este método recebe a frequência com que um termo ocorre num dado documento e envia esse valor para a função *computeValue*
- **computeValue()**: método que recebe a frequência (ou número de vezes) com que um termo ocorre num documento e calcula o peso do termo no documento através de uma expressão que iremos apresentar mais à frente.
- **tfNormalization()**: este método recebe o peso do termo no documento calculado obtido no método *computeValue()* e faz a sua normalização. Este valor normalizado vai influenciar futuramente os resultados da pesquisa (parte seguinte do trabalho).
- **freeRefMaps()**: quando uma certa percentagem da memória atribuída inicialmente na execução do programa é atingida, a estrutura de dados do indexer é escrita em disco e consequentemente essa memoria é libertada.
- **mergeFilesRefs()**: junta num único ficheiro o resultado dos diferentes ficheiros gerado pela função *freeRefMaps()*.
- **findToken()**: recebe o termo e de acordo com a sua letra inicial vai indicar para qual dos 5 mapas (mencionados neste relatório como forma de organizar o indexer) onde o respectivo token vai ser indexado.

Descrição: Este módulo é responsável por fazer a indexação dos termos. A função `addTerm` é a responsável por fazer tal indexação, usando as estruturas de dados `HashMap<String, Integer> tokenFreqDocMap` e `HashMap<String, String> tokenPosDocMap`, no qual a primeira armazena o termo e a frequência com este ocorre num determinado documento, e a segunda armazena o termo e posição ou posições em que este ocorre num dado documento.

Depois de um documento ser lido é calculado o seu peso **TF** usando a função `calculateTF` e aqui envia para o método `computeValue` a frequência com que o termo ocorre no documento, obtida da estrutura `tokenFreqDocMap`. Depois deste processo é feita, como na fase anterior do trabalho, a verificação se este termo (key) tem um value correspondente. Caso não tenha, então usa a função `getNewHM()` para obter um value correspondente ao token (key), mas caso tenha já um value, esse value irá ser atualizado usando a função `updateHW()`.

Depois nos métodos `getNewHM()` e `updateHM()` (explicado já a sua função) vai armazenar no indexer o peso normalizado do termo no documento usando a função `tfNormalization()` e também armazenar as posições em que o termo ocorre no documento.

Desta forma, o indexer terá o seguinte formato (como já foi abordado):
termo, docID, pesoNormalizado, posição no documento

Exemplo:
`zoom - 21287632=0.19968858028443906-3!, 21288572=0.19514312914783194-6!, 21289012=0.14650919994471437-42!, 21289804=0.22054299692055124-24!25!, 21290243=0.162363572069557-4!46!47!`

Em que
`token - doID=pesotf-pos1!pos2!, doID=pesotf-pos1!pos2!,...`

Nota: A posição de um dado termo num documento não era pedido para esta fase do trabalho. Apenas optámos por fazer esta implementação para termos a certeza que frequência com este ocorria num determinado documento era certa.

4.1.8 Memory Management

Principais métodos:

- **getCurrentMemory()**: método responsável por obter a memória total a ser consumida

4.1.9 SearchProcessor

Principais métodos:

- **start()**: método responsável pela iniciação do fluxo de todo o processo de pesquisa. Este método é invocado pela main classe para se iniciar o processo de pesquisa.
- **queryString()**: método permite ao programa receber uma query de entrada inserida pelo utilizador.
- **printTopScores()**: método que retorna os resultados da pesquisa ordenados.
- **queryType()**: método que retorna o tipo de query escolhida pelo utilizador.

Descrição: Este módulo é responsável por instanciar todo o pipeline de pesquisa dos resultados já indexados em disco, efectuado na fase anterior do trabalho. Este módulo começa por pedir ao utilizador para que este introduza uma query a ser pesquisada. O utilizador introduz a query e de seguida é pedido o tipo de query, ao qual o utilizador escolhe se pretende que seja uma query simples ou por frase (a ser explicado mais à frente). Depois é perguntado ao utilizador se pretende retornar todos os resultados ordenados ou seja os X primeiros, em que este "X"é um número especificado pelo mesmo. O procedimento seguinte é semelhante ao processo de indexação efectuado nas primeiras fases do trabalho, ou seja, os termos da query são adicionados a um array de tokens, que depois são verificados se são válidos, de seguida verifica-se se são stopwords e por fim é obtido o seu stemming e adicionado a uma HashMap, como acontecia também com os termos na parte anterior do trabalho. O score é calculado com base no peso calculado para cada termo e feita a sua normalização. Depois este resultado é multiplicado pelo TF já efectuado no trabalho anterior. Por fim este módulo retorna o resultado da pesquisa ordenado de acordo com o valor de score final calculado na classe *QueryProcessing*.

4.1.10 Query

Principais métodos e construtores:

- **Query(int type, String contQuery)**: construtor do tipo de dados Query que permite aceder aos parâmetros necessários para o tipo de pesquisa. No caso de pesquisa simples e por frase apenas é necessário um variável que identifica o tipo de query e o seu conteúdo.

- **Query(...)**: poderão surgir outros tipos de construtores para query que necessitem da proximidade e/ou de campos.

- **Métodos get() e set()**: permitem aceder ou modificar os atributos desta classe.

Descrição: Esta classe vai auxiliar-nos a especificar as informações da query. Ou seja, retorna-nos o tipo de query e seu conteúdo.

4.1.11 IndexerResults

Principais métodos:

- **getPosting()**: método que recebe como argumentos uma lista de termos da query e o tipo de query a ser usado na pesquisa e encaminha essa lista de termos da query para a função respectiva que é selecionada de acordo com o tipo de query. Ou seja, se o utilizador decidir que é uma query simples, então este método chama a função que vai tratar de fazer a pesquisa e passa lhe a lista de termos da mesma.

- **getPostingNormal()**: método responsável por fazer a pesquisa com o tipo de query simples, ou seja, procura os ficheiros que têm guardado todas as palavras que comecem pela primeira letra de cada termo, procura pelo seu peso TF associado (calculado na iteração anterior) e são guardados num HashMap para depois se proceder ao calculado do peso tf-idf, seguido da sua normalização e por fim o cálculo do score da respectiva query.

- **findFile()**: método responsável por identificar qual a letra (a,b,c...z) ou numericos (0) que identifica o ficheiro onde consta a query que se pretende pesquisar.

- **getLine()**: método retorna os termos pesquisados nos respetivos ficheiros de indexação.

  Usados para pesquisa em frase (extra):

- **getPostingPhrase()**: método responsável por fazer a pesquisa por frase, sendo necessário o método *docsOccurrence* que retorna os documentos onde os termos da query de pesquisa se encontram para então ser possível efectuar a pesquisa por frase, tendo em conta também as posições onde estes se encontram nos respectivos documentos.

- **docsOccurrence()**: método responsável por obter os documentos onde os termos da query de pesquisa foram encontrados para então o score ser calculado por frase que fosse introduzida pelo utilizador.

- **getPostingByDistance()**: método responsável por obter os documentos filtrados pela distância máxima. Este método é usado para ajudar na pesquisa por frase.

- **getDocIDsDistance()**: método responsável por obter o posting de documentos verificando se a sua distância é válida. Também é usado este método para a pesquisa por frase.

Descrição: Este módulo e restantes métodos auxiliares são vitais se proceder à obtenção aos respectivos tipos de pesquisa, pois são eles que vão verificar o número de occrrências de um termo nos documentos e respectivas posições para se efectuar a pesquisa simples e por frase. Também neste módulo existe então o método *findFile()* que especifica em que ficheiros indexados se deve proceder a procura pelos respectivos termos da query, bem como o método *getLine()* que retorna os termos pesquisados nos respectivos ficheiros de indexação.

4.1.12 QueryProcessing

Principais métodos:

- **addTerm()**: método responsável por, tal como era feito na indexação da parte anterior do trabalho, adicionar os termos a uma estrutura de dados HashMap que guardava como *key* o token e respetiva frequência com que o mesmo ocorria na query de pesquisa.

- **getQueryTerms()**: método responsável por obter os termos da query de pesquisa armazenados na estrutura de dados HashMap onde foi feita a sua inserção.

- **calculateScore()**: método responsável por calcular o score da query de pesquisa.

- **getNrDocuments()**: método responsável por obter o número total de documentos processados. Este valor é utilizado no cálculo do IDF de um termo.

Descrição: Este módulo é responsável por efectuar o cálculo do Score de pesquisa e retornar o seu resultado ao módulo *SearchProcessor()*. Depois

dos termos serem adicionados à estrutura de dados HashMap¡String, Integer¿ que guarda o termo e a frequência com que este ocorre na query de pesquisa, estes são obtidos através do método *getQueryTerms()* e assim é possível proceder-se ao cálculo do peso TF-IDF de cada termo. A fórmula deste cálculo é mostrada na secção seguinte deste relatório. Ao mesmo tempo que é feito cálculo do peso TF-IDF para cada termo, é feito ao mesmo tempo o somatório dos pesos dos termos da query, para depois mais tarde ser usado para calcular a normalização, como era feito no trabalho anterior. O termo e respectivo peso associado calculado são armazenados numa estrutura de dados `HashMap<String, Double>` e de seguida os pesos são então normalizados, da mesma forma que na iteração anterior do trabalho. Depois da normalização estar feita pode-se então proceder ao cálculo do score, ao qual basta multiplicar o resultado obtido da normalização pelo peso TF normalizado do trabalho anterior e que se encontra também no ficheiro de indexação. Depois de efectuado o cálculo do score, este é então retornado para o módulo *SearchProcessor* proceder à ordenação e "printagem"dos resultados.

4.1.13 ScorePosition

Principais métodos:

- **getScore()**: método responsável por obter o score da respectiva query calculado.
- **getPositions()**: método responsável por obter as posições dos tokens nos respectivos documentos, já contidas também nos ficheiros de indexação. As posições foram guardadas para serem usadas para poder obter o score de uma query do tipo "QueryPhrase"ou seja, query por frase.

Descrição: Esta classe auxiliar é usada para guardar as posições dos termos nos respectivos ficheiros resultantes da indexação e seu score. Esta classe instância também um construtor no qual tem como argumentos o score de um termo e uma estrutura de dados `ArrayList<Integer>` onde guarda todas as posições de um termo (posições estas que já estavam guardadas na iteração anterior do trabalho.

5 Implementação do indexer TF-IDF e Score

Nas figuras seguintes é mostrado com maior detalhe todo o processo de indexação que é efectuado:

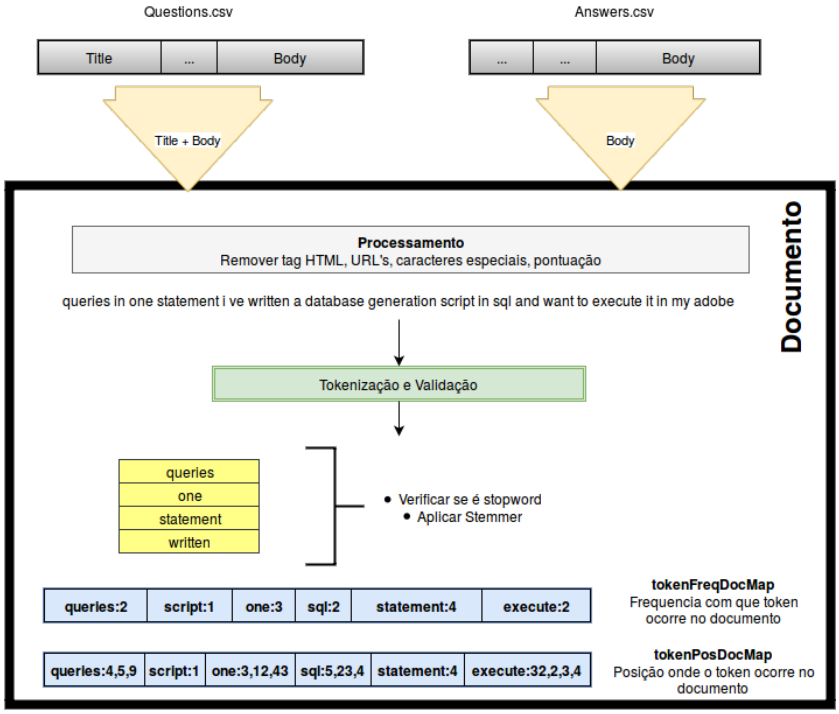

Figura 6: Processo de indexação - 1

Na figura acima, podemos ver que são lidos dois ficheiros CSV, *Questions.csv* e *Answers.csv* ao qual o primeiro contém Title e Body enquanto que o segundo contém apenas Body para ser tratado. O body e title dos respectivos ficheiros são processados e filtrados, ou seja, são removidas as tags HTML, URLs, caracteres especiais e pontuação para depois se poder proceder à sua tokenização e validação. É importante referir que depois disso um documento é o texto obtido do módulo *CorpusReader*. Após a tokenização e validação do documento, é percorrido cada um dos seus tokens e verifica-se se este é ou não uma stop word, se este for stop word é ignorado, caso contrário avança e é aplicado o seu stemming. Posteriormente este token é guardado numa estrutura temporária para cada documento e respetiva frequência de ocorrência. Existe também outra estrutura onde guarda o token e a(s) posição(ões) em que este ocorre (no documento). Estas duas estruturas servirão como auxiliares à estrutura final do indexer.

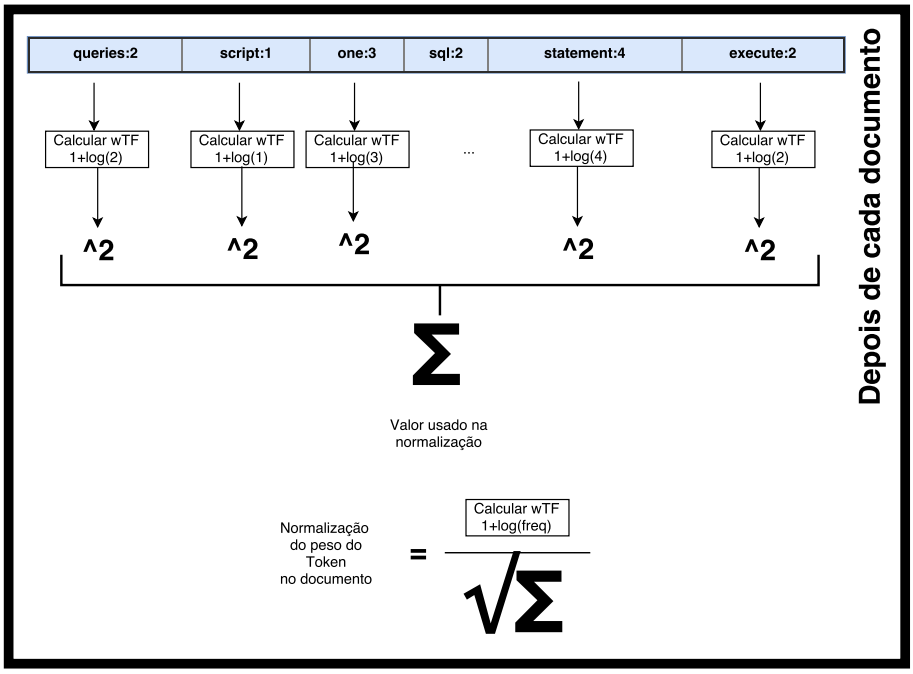

Figura 7: Processo de indexação - 2

No final do processamento de cada documento é então calculado o peso TF para cada token existente. Para tal, é necessário recorrer à estrutura que nos permite aceder à frequência com que cada token aparece no documento.

Conforme está especificado abaixo, o número de ocorrências de um termo t num documento d é representado por tf_t,d

Desta forma, o peso de um termo t em um documento d pode ser calculado recorrendo ao seguinte sistema de equações:

$$w_{t,d} = \begin{cases} 1 + log_{10}(tf_{t,d}) & tf_{t,d} > 0 \\ 0 & otherwise \end{cases}$$

Depois de calculado o peso de um termo t num documento d, é necessário proceder à normalização do seu valor.

A normalização desse valor é representada de acordo com a seguinte expressão:

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

Em que **x** é p valor obtido pelo sistema de equações anterior (peso do termo num documento). O valor indexado na estrutura do indexer será o

peso normalizado para cada token num determinado documento.

A frequência inversa do documento é obtida através da seguinte fórmula:

$$idf_t = log_{10}(\frac{N}{df_t})$$

Em que N é o número total de documentos da colecção.

Para calcular o peso de TF-IDF é usada a seguinte expressão e seguidamente proceder à normalização:

$$w_{t,d} = 1 + log_{10}tf_{t,d} * log_{10}(\frac{N}{df_t})$$

Finalmente, para obter o ranking da pequisa é usada a seguinte expressão para calculo do score.

$$Score(q,d) = \sum t \in q, ttf_{t,d} * idf_t$$

6 Bibliotecas externas

- *The Porter stemming algorithm*: http://snowball.tartarus.org/
- *Apache Commons CSV*: https://commons.apache.org/proper/commons-csv/
- *Apache Commons IO*: http://commons.apache.org/proper/commons-io/
- *Jsoup HTML parser library*: https://jsoup.org/

7 Estrutura do código

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

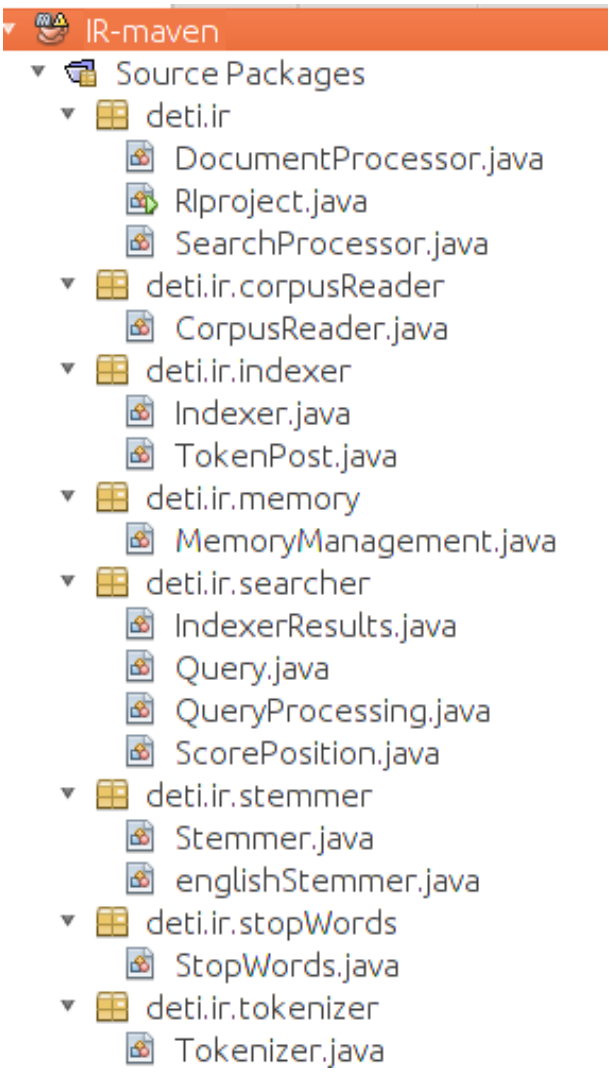

Figura 8: Estrutura do código

Na raiz do projeto Maven é possível encontrar três diretórios:

- `files-data`: dentro desta mesma pasta existe uma outra de nome *stacksample* onde se encontra o corpus a ser tratado. (Como os ficheiros a serem tratados tinham no total mais de 3G, decidiu-se não adicioná-los à entrega do trabalho, pois estes só se encontram mesmo

localmente nos nossos pcs onde o trabalho foi desenvolvido). De referir também que deste directório (files-data) que se encontra o ficheiro `stopwords_en.txt` que contém a lista de stopwords.

- `outputs`: Nesta pasta são colocados os ficheiros resultantes do processo de indexação. No fim de fazer a junção dos ficheiros pelas letras e números estes encontram-se neste directório
- `jars`: Neste directório estão colocados as bibliotecas compiladas necessárias para este projeto (Para este caso foi usado o jar do Snowball).

8 Execução

- Compilar e executar o nosso projeto recorrendo ao Netbeans.
- Adicionar argumentos ao programa
  - 1º Argumento: Caminho para directorio com ficheiros do corpus
  - 2º Argumento: caminho para o ficheiro das stopwords
  - 3º Argumento: memória máxima

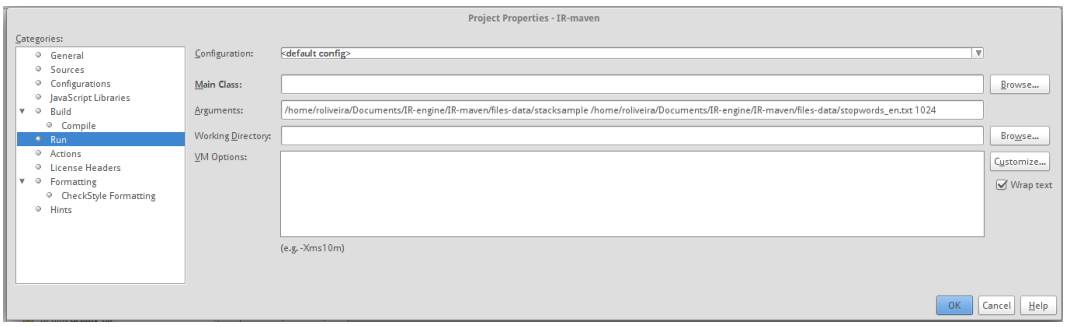

Figura 9: Adicionar argumentos no netbeans

- **OU**
- Executar ficheiro JAR através do terminal

  Exemplo:

  ```
  roliveira@roliveira-K55VJ: /Documents/IR-engine/IR-maven/target$
  java -jar IR-maven.jar  ../files-data/stacksample/
  ../files-data/stopwords_en.txt 512
  ```

9 Resultados

Para obtermos os tempos que demorou o processo de indexeção e respectiva escrita em ficheiro, foi utilizado o seguinte hardware:

- **Processador**: Intel(R) Core(TM) i7-3630QM CPU @ 2.40GHz, 2401 Mhz, 4 Núcleo(s), 8 Processador(es) Lógico(s)
- **Memória**: 8.00 GB

9.1 Indexação

Para o corpus fornecido `sample_corpus.zip` obtivemos os seguintes tempos para os diferentes limites de memória (MB). Os tempos médios e desvio padrão também são apresentados na figura seguinte.

Memória (MB)	Tempo 1	Tempo 2	Tempo 3	Tempo 4	Tempo médio	Desvio padrão
512	25:07,789	26:17,342	26:33,754	25:18,555	25:49,360	00:36,847
1024	18:36,833	18:53,899	18:28,036	19:01,709	18:45,119	00:13,349

Figura 10: Tempos medidos

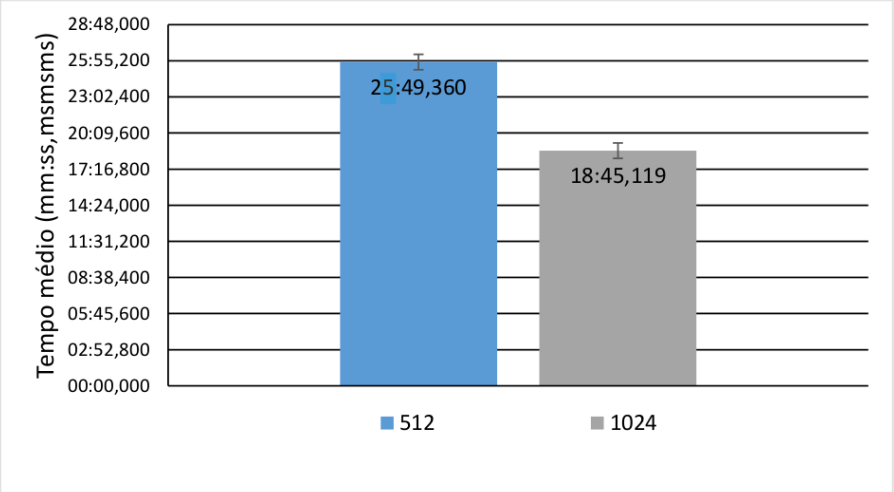

Figura 11: Gráfico dos valores médios

O corpus `sample_corpus` ocupa 3.5 GB em disco e o indexer escrito em disco ocupa aproximadamente 2.4GB.

Os ficheiro de *output* onde é possível observar o resultado da indexação tem o seguinte formato:

`token - DocID= peso_normalizado_do_token-posicao_no_doc!posicao_no_doc!...`

Um exemplo apresenta-se a seguir:
`arithmet - 2985101=0.19500697247805526-2!13!,`
`2985448=0.26151089264437677-6!15!18!, 2987045=0.19866170756091162-11!, 2987847=0.1409299442727274-2!`

9.2 Pesquisa

Os tempos para pesquisa simples foram os seguintes:

query insert	type	test1 (ms)	test2 (ms)	test3 (ms)	test4 (ms)	avg (ms)	nº docs
name	all results	6876	5472	5263	5348	5739.75	242886
result	all results	7223	5180	5985	5985	6093.25	204965
stop	all results	5690	1689	1637	1738	2688.5	48991

Figura 12: Resultados pesquisa simples

query insert	type	test1 (ms)	test2 (ms)	test3 (ms)	test4 (ms)	avg (ms)	nº docs
query insert hello	show all results	6130	5061	4682	4631	5126	213246
time hour	show all results	9047	7060	7179	6790	7519	314006

Figura 13: Resultados pesquisa multi query simples

10 Repositório de desenvolvimento

https://github.com/ruipoliveira/IR-engine

11 Conclusões

Chegado ao final deste relatório, é nossa intenção efetuar uma retrospetiva da evolução do mesmo, tendo em conta os problemas com que nos deparámos, e principais conclusões retiradas.

Em primeiro de tudo foi necessário efectuar alguns ajustes no que toca aos ficheiros resultantes da indexação, pois devido a problemas de junção foi necessário alterar o código de forma a estabelecer que os ficheiros com nome *tokenRef X1* referiam-se a todos os termos começados pela letra X do ficheiro Answers.csv e os ficheiros *tokenRef X2* referiam-se ao ficheiro

Questions.csv. Os termos que começassem por um digíto, eram guardadas nos ficheiros *tokenRef 01* e *tokenRef 02*.

Reparámos que, como foi visto, foi necessário adicionar classes auxiliares para o cálculo do final do score ordenado, pois teria-se que aproveitar o que estava ja indexado em disco e aceder aos respectivos pesos TF normalizados e suas posições em que o termo ocorria (decisão nossa em armazenar), assim como instanciar um tipo de dados Query para armazenar o conteúdo e o tipo de query desejada pelo utilizador. Infelizmente não tivemos tempo em implementar pesquisa por proximidade e por campo.

Finalizando este trabalho, podemos concluir que ficámos com uma ideia muito mais vasta sobre como realmente funcionam muitos motores de pesquisa, conscientes também que este tipo de trabalho pode ser sempre explorado mais afincadamente para outro tipo de projectos.