universidade de aveiro

Departamento de Eletrónica, Telecomunicações e Informática

Information Retrieval Engine

Modelling: classes and main methods definition.
Tarefa 1

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, 5 de Outubro 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* e *searcher*.

Nesta primeira iteração do projeto pretende-se chegar a uma conclusão relativamente à modelação do mesmo. Considera-se também definir e descrever as classes, métodos (principais) e o fluxo de dados aqui presente. Para além do descrito anteriormente, um dos principal objetivo deste projeto consiste em compreender os conceitos fundamentais de armazenamento e recuperação de informação.

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

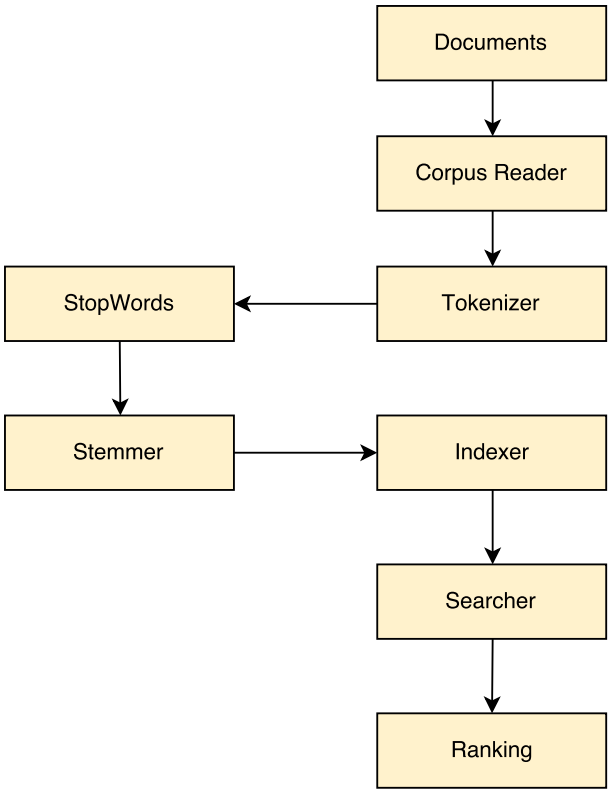

Figura 1: Diagrama geral

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 lê o 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 `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. Neste caso, a estratégia utilizada será o Inverted Indexer que consiste em, para cada termo, guardar uma referência com a

identificação do documento em que está presente, bem como o seu número de ocorrências nesse documento.

Nesta primeira fase do projeto, pensamos que a melhor estrutura de dados a adotadar seria:

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

  Este tipo de dados herda as características de uma *HashMap*, em que a *key* corresponde ao *token* e o *value* à frequência com que o *token* aparece em todos os documentos existentes.

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

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

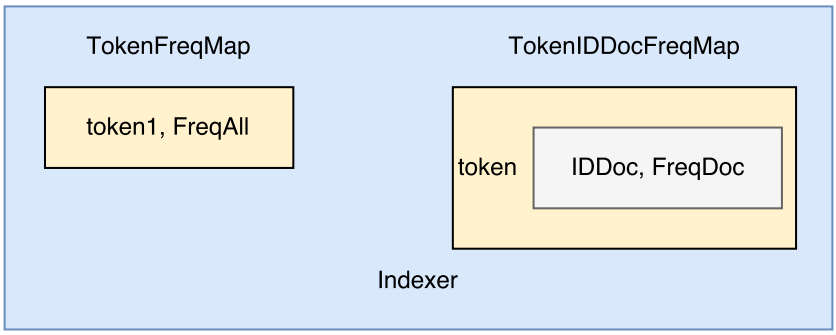

Figura 2: Fluxo geral de execução

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.

2.1.7 *Searcher*

Este módulo tem como objetivo fazer as pesquisas necessárias aos ficheiros resultantes da indexação, bem como filtrar o resultado das referidas pesquisas.

2.1.8 *Ranker*

Este módulo tem como objetivo ordenar os resultados provenientes do módulo de *Searcher*

2.1.9 *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 numa dada conjunto de documentos .

2.1.10 *SearcherProcessor*

Este módulo tem como objetivo instanciar todo o fluxo associado à pesquisa, isto é, aguarda pela receção de queries realizadas pelo utilizador, solicita ao Tokenizer a tokenização do conteúdo do documento e envia os vários tokens para um objeto do tipo Query.

2.2 Diagrama geral

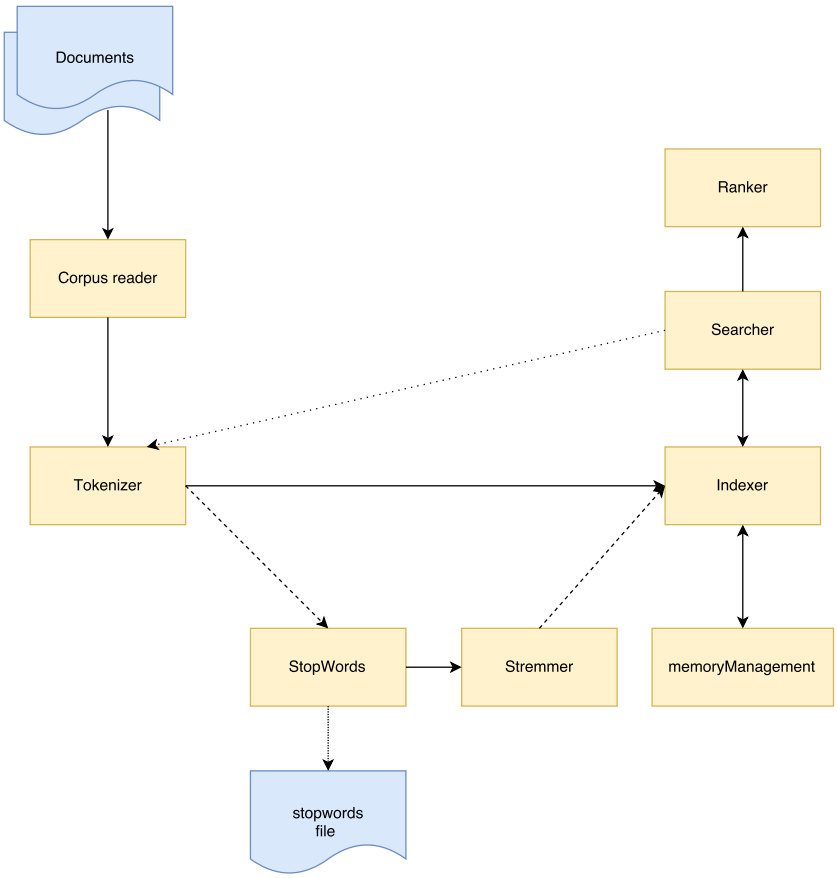

Figura 3: Fluxo geral de execução

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

O corpusReader irá encarregar-se de ler todos os ficheiros que se encontram num determinado diretório e de efetuar o seu pre-processamento. O texto pre-processado é dirigido para o nosso Tokenizer que se encarrega de transformar uma String num conjunto de termos válidos (tokens).

Numa primeira fase 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 pa-

lavras da lista de stopwords[1]. Todas as palavras não relevantes (stopwords) e posteriormente passado 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. Faz dois tipos de indexação: um mapa de frequência e um mapa de referências. O primeiro consiste numa estrutura de dados que faz o mapeamento entre um termo e o seu número total de ocorrências em todo o projeto. O segundo faz o mapeamento entre número de ocorrências de um termo em cada um dos ficheiros lidos. No final da indexação todos os termos são guardados em dois ficheiros, um para a frequência e outro para as referências.

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 o módulo Searcher que faz uso do método getTerms() que obtém a lista de termos a serem pesquisados.

Por fim temos o Ranker (ou Ranking) que vai ser responsável por, tendo em conta a lista de termos a serem pesquisados, ordenar o resultado da pesquisa efetuada no módulo Searcher.

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

3 Diagrama de classes

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

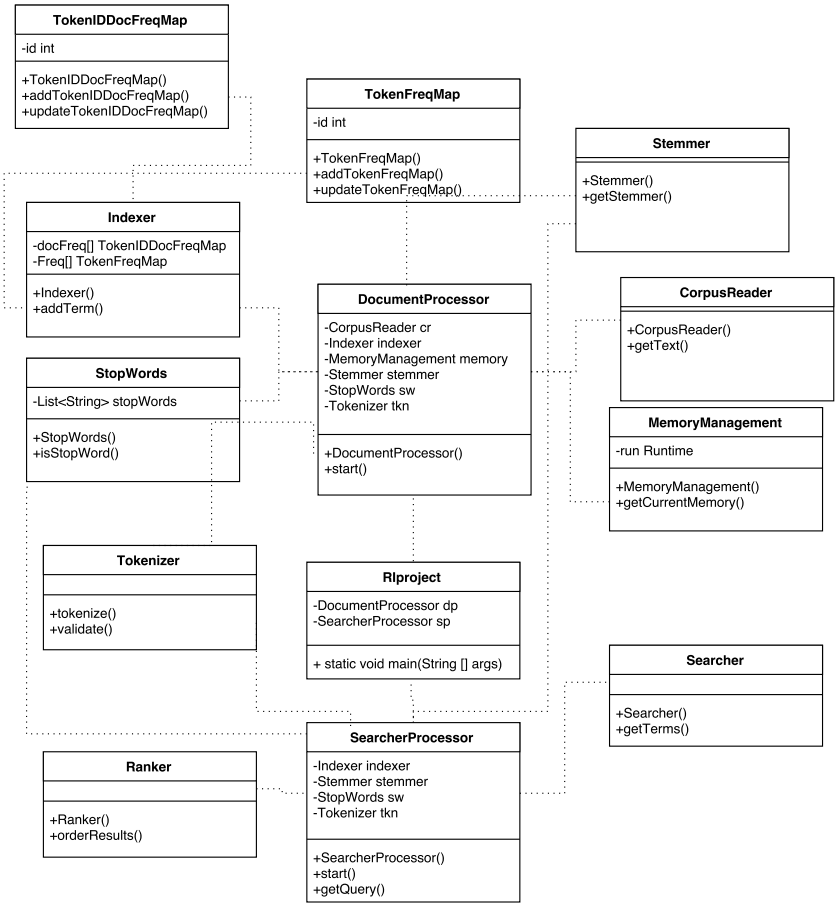

Figura 4: Diagrama de classes

3.1 Classes e métodos

3.1.1 RIproject (main)

Classe principal do projeto. É aqui onde são instanciadas as classes DocumentProcessor e SearcherProcessor que iremos descrever de seguida.

3.1.2 DocumentProcessor

Principais métodos:

- **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.

3.1.3 SearcherProcessor

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.
- **getQuery()**: método permite ao programa receber uma query de entrada inserida pelo utilizador.

3.1.4 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, recorrendo a um Stream que permite fazer o pré-processamento de cada linha.

  uma das linhas;

3.1.5 Tokenizer

Principais métodos:

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

3.1.6 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.

3.1.7 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).

3.1.8 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.

3.1.9 TokenFreqMap e TokenIDDocFreqMap

As estruturas de dados criadas para o armazenamento do indexer terá pelo menos os seguintes métodos:

- **addTokenFreqMap()**: é executado sempre que um novo token é adicionado ao indexer.
- **updateTokenFreqMap()**: é executado sempre que um token já existente no indexer aparece novamente.
- **addTokenIDDocFreqMap()**: é executado juntamente com o método addTokenFreqMap()
- **updateTokenIDDocFreqMap()**: é executado juntamente com o método updateTokenFreqMap()

3.1.10 Searcher

Principais métodos:

- **getTerms()**: método para obter a lista dos termos a serem pesquisados

3.1.11 Ranker

Principais métodos:

- **orderResults()**: este método é responsável por ordenar o resultado da pesquisa, vinda do módulo Searcher.

4 Bibliotecas externas

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

5 Estrutura do código

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

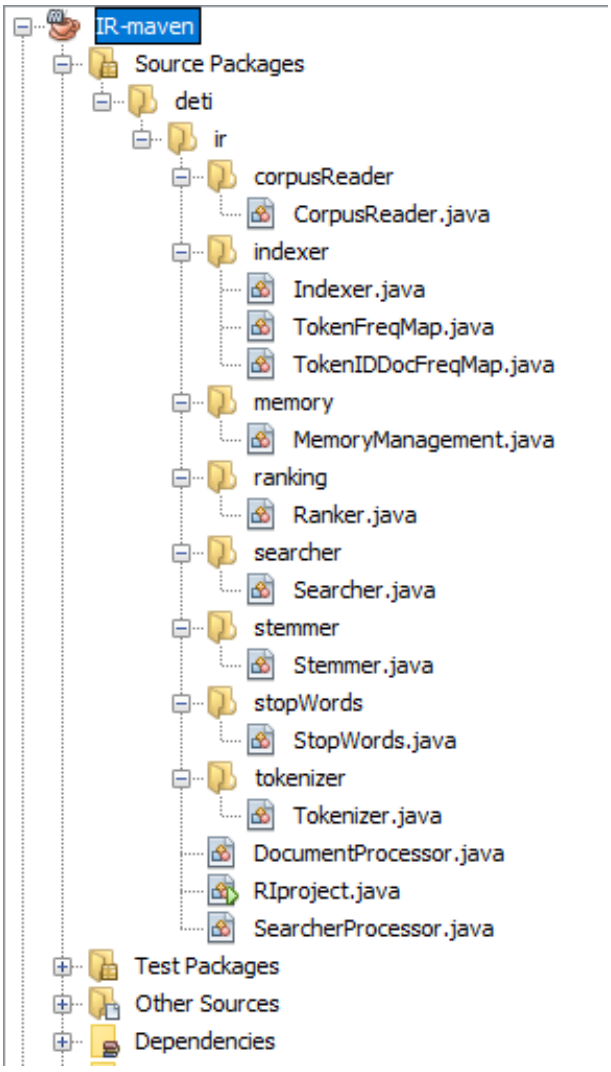

Figura 5: Estrutura do código

done

6 Repositório de desenvolvimento

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

7 Conclusões

Neste relatório foi apresentada uma proposta de modulação de um mecanismo de recuperação de informação, tendo como base conteúdos abordados nas aulas de Recuperação de Informação. Nesta primeira fase do trabalho, focámo-nos essencialmente na modulação de todo o processo, desde o *corpus reader*, passando ao *tokenizer*, indexação e por fim ao mecanismo de pesquisa.

Neste documento apresentamos as escolhas que fizemos para modular este projeto, sendo que achamos que se adequa aos requisitos apresentados na proposta de trabalho e nos conteúdos das aulas teóricas. Além disso, sabemos que poderão haver falhas sendo que apenas serão descobertas quando passarmos à implementação i.e criação de novos métodos, métodos auxiliares, novos atributos, ect.

Com esta primeira fase do trabalho ficámos a saber mais a nível teórico, mais especificamente dos vários módulos existentes neste processo e pretendemos futuramente aplicar estes conhecimentos a um nível mais prático.