universidade de aveiro deti departamento de electrónica, telecomunicações e informática

Assignment 3

Recuperação de Informação

Trabalho realizado por:

- Lucas Barros Nº Mec. 83895
- Pedro Cavadas Nº Mec. 85090

Introdução

O principal objetivo deste trabalho é o de criar um programa que processa vários documentos de texto e extraia os campos PMID, TI e AB. De seguida são extraídos tokens, ou seja, palavras chave de maneira a auxiliar a implementação de um mecanismo de busca. De seguida, recorrendo a ficheiros auxiliares, é feito um teste à eficiência do mecanismo de busca, produzindo resultados que vão ser explorados numa fase posterior deste relatório.

Este trabalho também ajuda a perceber as tarefas que estão por detrás dos melhores motores de busca da atualidade como o Google, sempre que se escreve uma Query e se pressiona ENTER.

A linguagem de Programação adotada foi Python 3 por ser simples, e por desempenhar as funções de processamento de texto (strings) com grande eficiência.

Conceitos Teóricos

Tokenization

Tokenization é o processo de converter uma sequência de caracteres de entrada em tokens de saída. Este processo tem normalmente um conjunto de subprocessos/regras associados que vão depender de Tokenizer para Tokenizer. Neste projeto, utilizamos 2 Tokenizers para fazer comparações.

Index

Index é uma estrutura de dados que permite que objetos estejam guardados como listas. Neste caso, o nosso Index pode ter um dos seguintes formatos:

1. “{ token : { doc_id : count, ... }, ... }”
2. “{ token : (idf, { doc_id : (weight, count), ... }, ... }”
3. “{ token : { doc_id : [index1, index2, ...], ... }, ... }”
4. “{ token : (idf, { doc_id : (weight, [index1, index2, ...]), ... }, ... }”

O primeiro e segundo formato, não contém o índice das posições onde o token aparece no dado documento, já o terceiro e o quarto contém. Por outro lado, no segundo e no quarto foi calculado o tfidf que permite atribuir um ranking a um documento para um dado token. Isto vai permitir fazer procuras na nossa estrutura de dados de forma rápida e estruturada.

Indexer

O Indexer é uma classe que processa documentos e armazena os resultados no Index. Esse processamento tem como objetivo dividir os documentos em tokens e com base nesses tokens criar “pesos” com base nesse token.

Ranker

O Ranker é uma classe que com recurso aos dados do Index avalia uma query e retorna documentos que avalie como relevantes em relação à query feita.

Passos para executar o Projeto

Utilização do nosso “engine”:

```
usage: engine.py [-h] [-s STOPWORDS] [-m MEMORY] [--store_positions] [--tfidf]
                 input output {tokenizer,simple_tokenizer}

positional arguments:
  input                 Filename or directory with files to index
  output                Filename of the file with the indexer result
  {tokenizer,simple_tokenizer}
                        Indicates which tokenizer the indexer must use

optional arguments:
  -h, --help            show this help message and exit
  -s STOPWORDS, --stopwords STOPWORDS
                        Filename of the stopwords list (ignored if tokenizer
                        is "simple_tokenizer")
  -m MEMORY, --memory MEMORY
                        Percentage of max memory used in the process
  --store_positions     Indicates if indexer stores positions of terms or not
  --tfidf               Indicates if program calculates tfidf or not
```

Execução do exercício Ranker:

```
usage: ranker.py [-h] [-q QUERY] [-f FILE] [-s STOPWORDS]
                 documents model {tokenizer,simple_tokenizer}

positional arguments:
  documents             Filename or directory with documents
  model                 Filename (without extension) of the index model
  {tokenizer,simple_tokenizer}
                        Indicates which tokenizer the ranker must use

optional arguments:
  -h, --help            show this help message and exit
  -q QUERY, --query QUERY
                        Query to rank
  -f FILE, --file FILE  File with queries to rank
  -s STOPWORDS, --stopwords STOPWORDS
                        Filename of the stopwords list (ignored if tokenizer
                        is "simple_tokenizer")
```

Execução do Measures: "*python(3) measures.py*"

Diagrama de Classes

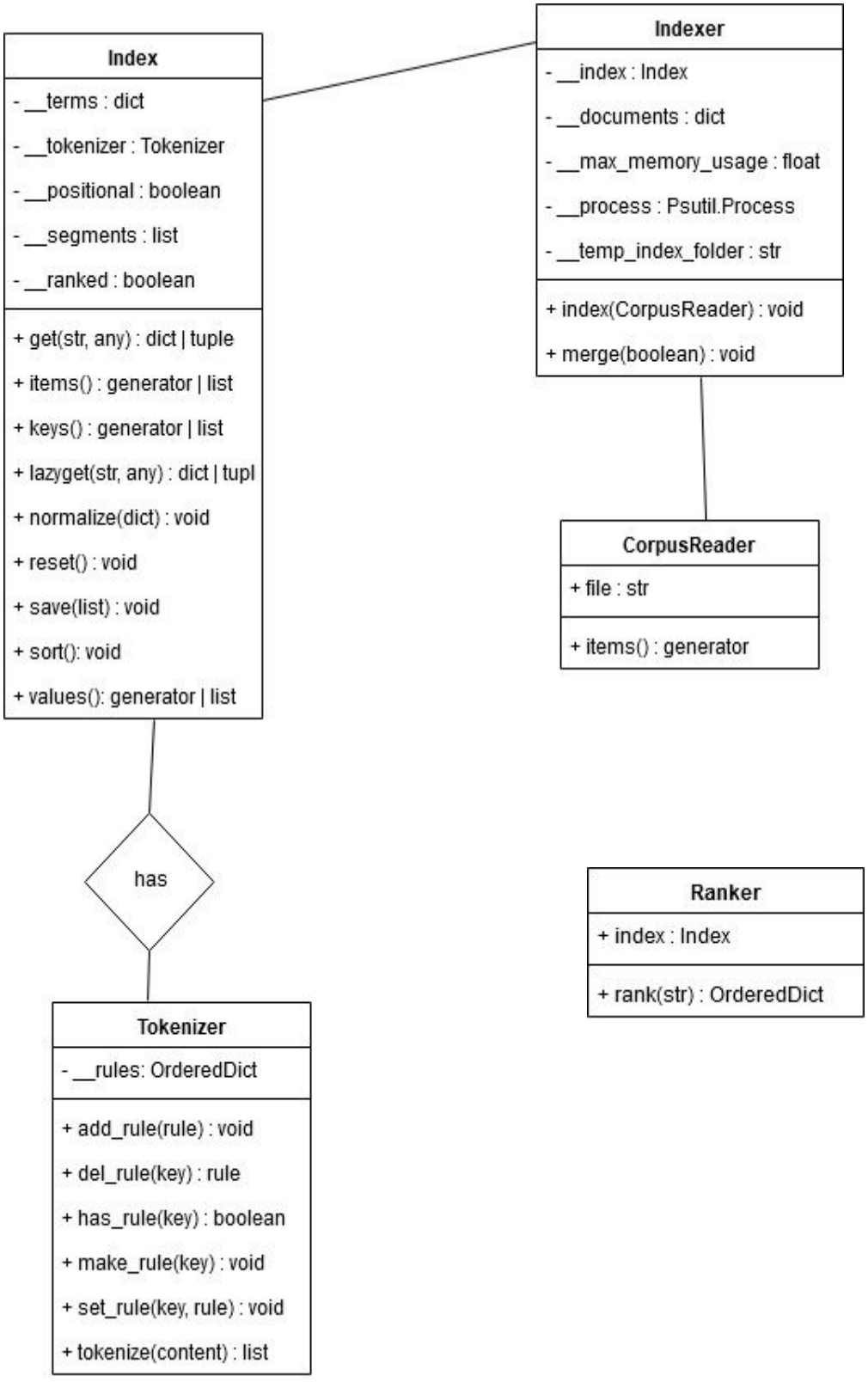
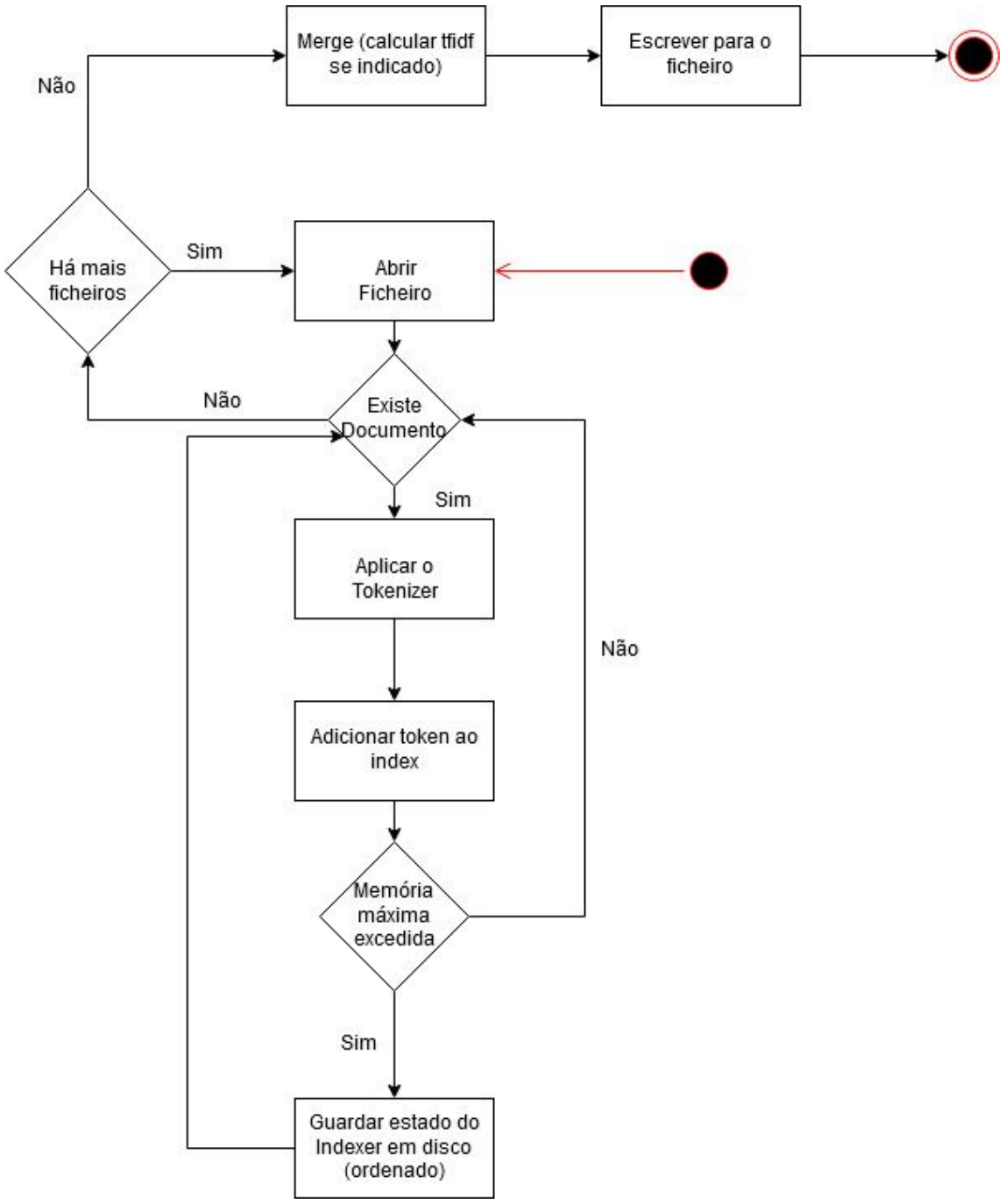


Diagrama de Workflow



Como se pode observar pelos diagramas, o texto original (documentos) são processados pela classe "CorpusReader". Esta classe possui apenas um construtor e um método que lê o ficheiro e retorna um gerador que vai retornando um tuplo (pmid : TI) com o ID do documento (PMID) e com TI correspondente numa string. Isto significa que os dados vão sendo lidos à medida que vão sendo processados.

Os TI's são então processados pela classe Tokenizer que vai aplicar uma série de regras. Estão implementados 2 Tokenizers, o "simple tokenizer" e o "tokenizer". O "simple tokenizer" substitui todos os caracteres não alfabéticos por espaços, substitui todos os caracteres pelo minúsculo correspondente. A divisão dos tokens é efetuada por espaço e são ignorados todos os tokens com menos de 3 caracteres.

Já o "tokenizer" substitui todos os caracteres pelo minúsculo correspondente, substitui apenas certos sinais de pontuação por espaços, sendo eles: ".", ",", "-", "!", "?", ";", "*", "/", "=", "(", ")", "[", "]", ":", "'", "`", "\", "\n", com a exceção do "." quando este é precedido de um único caratere alfabético seguido e no máximo precedido de um caratere alfabético seguido, transformando assim uma sequência deste tipo de pontos num único token, por exemplo, "u.s.a." ou "u.s.a" são ambos convertidos para "usa". Além do ponto, também o "-" não é substituído quando é antecedido e precedido por uma palavra, por exemplo, "algodão-doce". Além disto filtra todas as palavras com menos de 2 carateres, além de utilizar uma *stopword list* para filtrar certas palavras específicas, bem como aplica stemming a todas as palavras. Os tokens, tal como anteriormente são separados utilizando os espaços.

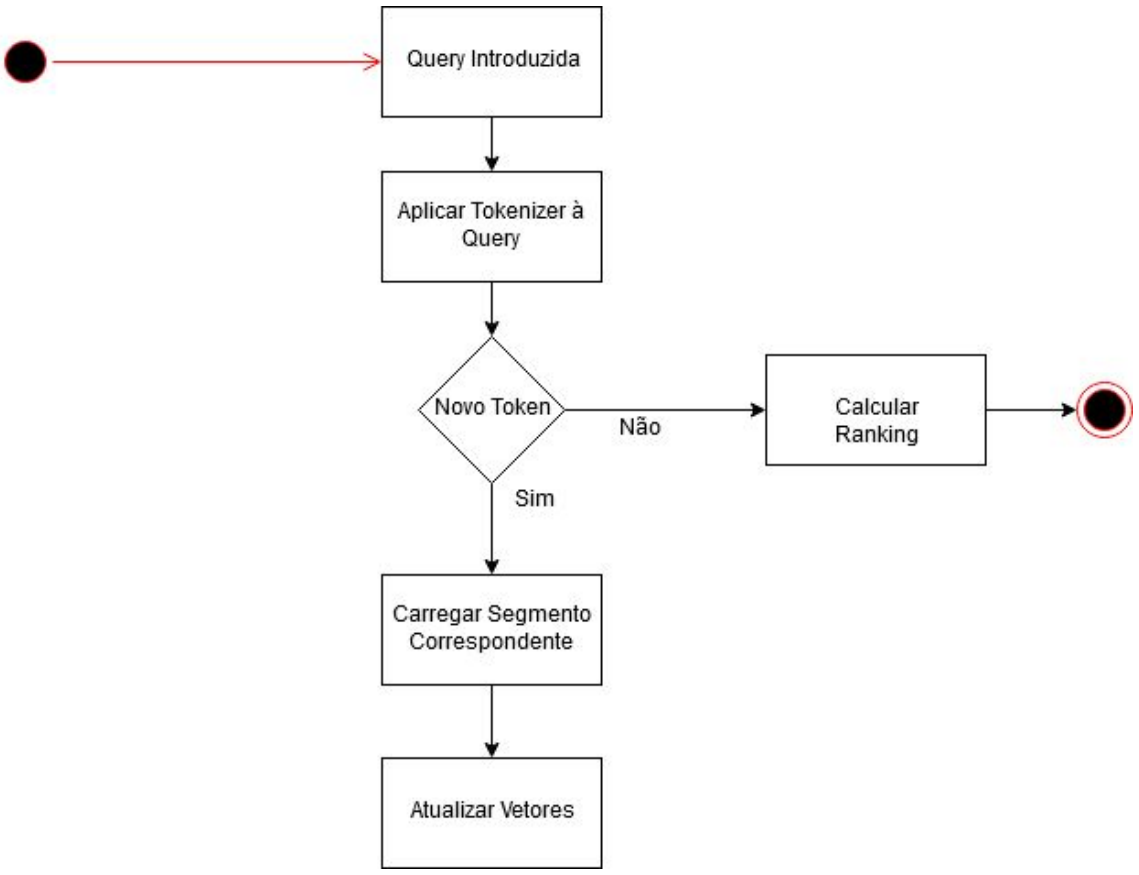
De seguida os tokens são indexados pela classe Indexer que possui um dicionário "terms" que tem a seguinte estrutura:

{token : {doc_id1 : [index1, index2], doc_id2 : [index3]}}

em que token como o próprio nome indica é o token, o doc_id é o PMID do documento, e os index são as posições no texto em que o token apareceu. Esta classe possui o método "update(doc_id, terms)" que sempre que é processado um documento, indexa os tokens desse documento.

Neste processo, caso uma dada quantidade de memória seja excedida, o indexer guarda os tokens atuais (por ordem alfabética) em disco, limpa a memória e continua o processo. No final, ocorre o processo de merge onde o indexer realiza um merge dos vários ficheiros em disco.

Este merge consiste em juntar os dados associados a um dado token que podem estar espalhados pelos vários documentos. Isto é feito de forma ordenada, primeiro escolhe-se o token mais pequeno encontrado em todos os ficheiros. De seguida, volta-se a escolher o token mais pequeno (o anterior já não conta para esta operação) e se esse novo token for igual ao anterior realiza-se o merge, caso contrário guarda-se outro numa lista temporário. Repete-se o processo até que nenhum dos ficheiros gerados pelo processo de indexing contenha tokens. No meio disto, a lista temporária pode ser escrita para disco (caso exceda um dado limite de memória) e guarda-se apenas o bloco (ficheiro) em memória associados ao primeiro e último token dessa lista (isto será útil para carregar o bloco para memória quando necessário). Além disto pode também ser realizado o cálculo do tfidf. Este é feito quando o token vai para ser escrito na lista temporária.



Depois do indexer calculado e guardado em disco, o programa já está preparado para processar querys e retornar os documentos mais importantes. Para tal, sempre que uma query é introduzida, é-lhe aplicada o Tokenizer para remover "stop words" e aplicar todas as outras regras necessárias. De seguida, para cada token, é carregado o segmento de index (cada segmento tem no máximo 100 MB de dados) que contém o dado token e para cada doc_id associado a esse token, é guardado o *tf-idf*. No final, é calculado o produto interno entre os vetores para dar finalmente o Ranking final.

Análise de Resultados

Com o intuito de analisar o nosso Ranker calculamos algumas métricas sobre o top 10, top 20 e top 50 documentos. Este processo foi feito para cada query e no final uma média de cada métrica. Para facilitar a análise dos resultados teremos em conta apenas a média de cada uma das métricas e como cada uma varia à medida que aumentamos o número de documentos.

Top-10 - Average

Precision	Recall	F-Measure	Avg. Precision	NDCG
0.292	0.065	0.086	0.337	0.101

Top-20 - Average

Precision	Recall	F-Measure	Avg. Precision	NDCG
0.257	0.081	0.105	0.303	0.120

Top-50 - Average

Precision	Recall	F-Measure	Avg. Precision	NDCG
0.214	0.116	0.131	0.258	0.150

Aqui pode-se verificar que à medida que se aumenta o número de documentos a precisão e a precisão média caem, sendo que todas as outra métricas aumentam. Este resultado era o esperado já que a precisão é uma métrica que avalia a relação entre o número de documentos relevantes e o número total de documentos. Ao aumentar o número total de documentos e tendo em conta que estes estão ordenados de forma decrescente pelo ranking, é de esperar que o número de documentos relevantes aumenta numa proporção menor ao aumento do número total de documentos, o que faz com que o valor final da métrica caia.

Já a “recall” é uma métrica que relaciona o número de documentos relevantes retornados pelo Ranker e os relevantes não retornados. Ainda que o aumento de documentos relevantes aumente numa proporção inferior ao número total de documentos, a métrica deverá crescer já que o número total de documentos relevantes (retornados e não retornados) mantém-se.

Pudemos também verificar que a métrica “f-measure” aumenta com um número maior de documentos, isto significa que a relação entre a “recall” e a “precision” melhora.

Finalmente, o NDCG também aumenta, isto deve-se ao facto de o número de documentos relevantes aumentarem.

Conclusão

Com este trabalho aprendemos mais sobre processamento de texto e também sobre as várias tarefas desempenhadas pelos motores de busca como a Google.

Aprendemos também sobre várias medidas para testar o desempenho destes motores de forma a melhorar a qualidade destes.

Podemos concluir também que os resultados obtidos são muito satisfatórios e fazem sentido de acordo com o que está a ser processado.

Referências

https://medium.com/@datamonsters/text-processing-in-python-step-tools-and-examples-bf025f872908

https://www.stackoverflow.com

https://medium.com/towards-artificial-intelligence/text-mining-in-python-steps-and-examples-78b3f8fd913b