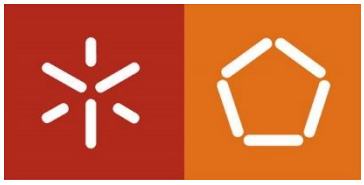

Mestrado em Engenharia Biomédica – Informática Médica

1º Ano | 2022/2023 | 2º Semestre

Processamento e Recuperação de Informação

Serviço de Pesquisa Simples

Ciarán McEvoy A87240

Gonçalo Carvalho PG50392

Hugo Silva PG50416

Tomás Lima PG50788

Docentes:

António Costa

Introdução

O acesso à informação é algo imprescindível e, em 2020, a quantidade de informação disponível na internet já tinha ultrapassado os 64 ZB ($64x10^{12}$ GB), com previsão de atingir os 175 ZB em 2025.[1] Assim, são necessários serviços de pesquisa sofisticados, como os desenvolvidos por empresas como a Google e a Microsoft, que são capazes de recolher e armazenar informações de *websites*, para conseguirem indicar quais destes são os melhores perante as procuras efetuadas pelos seus utilizadores.

Assim, este trabalho foca-se no desenvolvimento de um serviço semelhante aos indicados anteriormente que, apesar de ser mais simplista, contém os módulos necessários para a visualização do funcionamento de um sistema deste tipo.

Motor de Pesquisa

O motor de pesquisa é utilizado para retirar documentos de uma determinada página web que contenham uma determinada informação que se pretenda retirar.

Para isto são criados 3 módulos internos que servem para retirar abrir os *links* presentes numa dada página, ver se têm um termo que se pretende retirar, caso isto se verifique fazer o *download* do conteúdo destas em formato HTML e, por fim, limpá-las, sendo estes 3 módulos denominados *HtmlDownloader, LinkExtractor e Cleaner*.

O módulo *HtmlDownloader* recebe um dado *link* escolhido pelo utilizador (sendo este o link que nos vai servir como *link* base). Vai ser estabelecida uma conexão com esta página *web*, e, de seguida, caso a ligação tenha sucesso procede-se ao passo seguinte. O conteúdo da página vai ser passado para um *buffer* e depois armazenado numa variável que, de seguida, iremos passar para o módulo *LinkExtractor*.

No módulo *LinkExtractor*, procedemos a utilizar dar ao utilizador a possibilidade de escolher que termo quer procurar, e utilizamos uma expressão regular para encontrar todos os *URL*s presentes no conteúdo que lhe foi passado pelo módulo *HtmlDownloader*. Vamos iterar por estes *URLs* tentando estabelecer ligações com estes, e caso seja possível, iremos proceder a fazer *parse* do conteúdo da página HTML ficando apenas com as porções que contém a *tag* correspondente a parágrafos. Se encontrarmos aqui o termo que

estamos a procurar, e se este *URL* não estiver presente na nossa lista de *URLs*, ir-se-á proceder a avisar o utilizador que o conteúdo da página do *URL* em questão contem o termo e guardamos o *URL* em questão numa lista. No fim desta operação esperamos 50 milissegundos para assegurar que não bombardeamos o servidor com pedidos. Quando tivermos percorridos todos os *URLs* no conteúdo do *URL* base, procedemos a retornar a lista com todos os *URLs* que cumprem as condições anteriores.

Esta lista é então devolvida para o módulo *HtmlDownloader* e procedemos a establecer uma conexão com cada um destes *URL*s e caso o pedido tenha sucesso, a fazer download da página em questão. Para evitar problemas, é utilizada uma expressão regular para assegurar que determinados caracteres são substituídos por *underscores* quando os guardamos na máquina em questão. Por fim avisamos o utilizador que o ficheiro foi guardado com sucesso.

O último módulo, *Cleaner* acede à pasta aonde guardamos os ficheiros, e procede a fazer parse dos mesmos, limpando-os para serem utilizados pelos outros módulos. Nesta limpeza apenas considerámos a *tag* de parágrafos tal como anteriormente. Este módulo apenas está separado dos anteriores para permitir a limpeza de qualquer documento que descarreguemos em vez de apenas documentos de que fizemos download.

O motor de pesquisa é inicializado pelo método main da classe *HtmlDownloader*.

Estrutura dos dados

Foi necessário desenvolver duas estruturas de dados para a aplicação, uma para guardar os dados em si (o índice invertido), e outro para guardar a relação entre o ID dos documentos e o seu nome (ou URL).

O índice invertido tem uma estrutura complexa, com diversas “camadas”. Esta estrutura é:

```
HashMap<String,Pair<Integer, HashMap<Integer,ArrayList<Integer>>>>
```

Ou seja, o índice invertido é um HashMap, cujas keys são Strings (os termos), e os seus valores são Pairs (tuplos), em que o primeiro elemento são Integers (o número de documentos em que o termo aparece – document frequency) e o segundo valor é outro HashMap, que corresponde à posting list, que guarda nas keys Integers (os IDs dos

documentos em que aparece) e nos valores ArrayLists (que guarda as posições em que este aparece no documento - position list).

Já segunda estrutura é mais simples, conseguida através de um simples HashMap, em que as keys do mesmo dizem respeito aos IDs dos documentos e os valores são os nomes (ou URLs) dos documentos que lhes dizem respeito.

Indexação

A indexação tem como objetivo a criação do índice invertido, a partir dos documentos devolvidos pelo motor de pesquisa. Estes são guardados na pasta *Documentos* da base de dados. O motor de indexação irá então percorrer todos os documentos nesta pasta, criando primeiro um *hashCode* a partir do nome do ficheiro (que servirá como docID) e guardando esta relação na estrutura HashMapDocs, sendo a chave o docID e o valor o nome por extenso do documento.

Após o cálculo do *hashCode*, verifica-se se esta chave já existe no ficheiro *hashmapdocs.json*. Se sim, não se fará de novo a indexação. Caso contrário, procede-se então à indexação do novo documento. A sua leitura devolve um String única, a partir da qual se cria uma lista de termos, dividindo-a em todos os espaços ou vírgulas. Cada termo sofre então uma pequena normalização, removendo-se todas as capitalizações, acentuações e removendo os adereços, como “”, (), {}, [], !, ?, etc.

Inicia-se uma variável inteira para a posição e à medida que se vai percorrendo a lista de termos, vai-se incrementando a posição em uma unidade. Verifica-se se o termo não equivale a nenhuma *stopword*, definidas anteriormente e caso não o seja, vai-se adicionar o termo, associado à sua posição, a uma lista de pares com recurso à função *indexTerm()*. Deste modo, ter-se-á no fim uma lista completa dos termos e respetiva posição no documento.

Esta lista será então percorrida, de forma a adicionar a nova informação ao Dicionário. Existem 3 situações possíveis: o termo ainda não existe no dicionário; o termo já existe no dicionário, mas a *posting list* para este termo-documento não; e, por fim, a já existência no Dicionário, tanto do termo como da *posting list* para este termo-documento, sendo que só se terá de adicionar a nova posição neste caso.

De referir que, no fim da indexação de cada documento, tanto o Dicionário como o *HashMapDocs* são atualizados nos respetivos ficheiros *json* na base de dados. Isto garante assim a manutenção da informação em disco, e não apenas em memória. Permite ainda que, caso a indexação de um dos documentos falhe, a indexação dos que já tinham sido processados dentro da pasta não seja perdida.

Engenho de Pesquisa

A interface que o utilizador vai usar para fazer a interrogação e receber os documentos é o terminal. Este é acedido através do método main da classe *QueryProcura*. Os documentos vão ser enviados de acordo com a relevância com a query do utilizador. Sendo que foi possível a obtenção dos dados necessário como modelo de classificação em "rank" escolheu-se usar o Modelo Vetorial. Para a criação do Modelo Vetorial foram criados vários métodos que tratam de diferentes partes do modelo.

Para começar, o modelo recebe o dicionário da Indexação no formato:

HashMap<String,Pair<Integer,PostingList>>

A partir desta estrutura foram extraídos o term frequency da query e do documento, o document frequency de cada termo e o número de documentos total. O código foi dividido em dois métodos principais, `queryScore()`, `scoresDocs()`, `similiaridadeFinal()`.

Termo	Query						Documento				Produto
	tf	tf-wt	df	idf	wt	normalized	tf	tf-wt	wt	normalized	
auto	0	0	5000	2.3	0	0	1	1	1	0.52	0
best	1	1	50000	1.3	1.3	0.34	0	0	0	0	0
car	1	1	10000	2.0	2.0	0.52	1	1	1	0.52	0.27
insurance	1	1	1000	3.0	3.0	0.78	2	1.3	1.3	0.68	0.53

queryScore(`HashMap<String, Integer> t_freq`**)**

Esta função trata de calcular os valores normalizados de wt para cada palavra que aparece na query. Recebe como parâmetro t_freq que é um HashMap de cada palavra que aparece na query, e o respetivo term frequency. Se uma palavra aparecer no Documento e não na query é devolvido um valor normalizado de 0 para essa palavra. O return final é

um HashMap<String, Double>, sendo o String, o termo, e o Double, o valor normalizado de esse termo.

```
scoresDocs(String query)
```

Esta função trata de calcular os valores normalizados de wt para cada palavra que aparece na query e no respetivo documento. Recebe como parâmetro a query que usa para comparar a query e o documento de modo a não ter que calcular os dados de todas as palavras no documento. O return final desta função é um HashMap<Integer, HashMap<String, Double>>, cada *Integer* docID, tem um HashMap com os *String* termos e o respetivo *Double* score normalizado.

Por fim a função similiaridadeFinal(HashMap<Integer, HashMap<String, Double>> dscores, HashMap<String, Double> qscores) recebe os returns do queryscore e scoresDocs e efetua o cálculo do produto e do score final entre o documento e a query, de modo a devolver os documentos por "rank", sendo que quanto maior o score melhor o "rank"

Conclusão e melhorias

O programa desenvolvido cumpre os requisitos necessários, permitindo-nos a procura de documentos de acordo com um termo, uma organização dos documentos tendo em conta a sua relevância para uma dada procura, e até o *download* de documentos da *web* tendo em conta isto. No entanto, existem ainda falhas no *software* criado.

No motor de pesquisa, determinados sites podem recusar pedidos. Isto acontece porque o nosso motor de pesquisa não tem nenhum tipo de autenticação. Através da utilização de bibliotecas e protocolos como o SSL (*Secured Socket Layer)* ou o TLS (*Transport Layer Security)* estes problemas poderiam ser resolvidos. No entanto, o funcionamento atual deste módulo serve como *proof-of-concept*.

Outro problema presente neste motor de pesquisa é que determinados *URL*s diferentes acabam muitas vezes por ser praticamente idênticos, por exemplo, no caso de algumas páginas da Wikipédia, a única diferença está nos *URL*s contendo estes a mesma informação, mas pertencendo a secções diferentes da mesma página.

Já que a estrutura de Dicionário criada guarda também as posições dos termos nos documentos, seria possível a criação de um método de procura booleano. Tendo os dois tipos de procuras disponíveis (vetorial e booleano), seria interessante comparar os desempenhos de ambos para uma mesma query.

O processo de tokenização feito é também de muito baixo nível. Um processo mais refinado permitiria um melhor desempenho do modelo, já que no Dicionário construído, por exemplo, “intromento” e “instruments” são dois termos distintos.

Por fim, o último ponto de melhoria a apontar será o facto de o ponto de interação com o utilizador ser muito rudimentar. Mais ainda, não existe um ponto de comunicação entre o motor de busca na web e o engenho de procura no dicionário.

Referências

[1] “How big is the internet, and how do we measure it?", Health IT, acedido a 21/05/23 https://healthit.com.au/how-big-is-the-internet-and-how-do-we-measure-it/