Mestrado integrado em Engenharia de Computadores e Telemática

Departamento de Electrónica, Telecomunicações e informática

2015/2016

Recuperação de Informação

Implement a simple corpus reader, tokenizer, and Boolean indexer.

GRUPO 04

Autores:

- Bernardo Ferreira (67413)
- Bruno Silva (68535)

Responsável:

- Prof. Sérgio Matos

23/11/2016

Índice

1. Descrição das Classes

Nesta Secção do documento vai ser realizada a descrição das classes usadas na implementação do motor de indexação booleana.

1.1. Classes Principais

1.1.1. Corpus Reader

O corpus reader é a classe responsável por percorrer um diretório recursivamente e devolver a lista de todos ficheiros e sua extensão, existentes em todo o diretório e respectivos subdiretórios.

Esta classe conta conta com os seguintes atributos:

- String[] ignoreExtensions - array que contém extensões de documentos que queiramos ignorar no processo.
- ArrayList<CorpusFiles> files - Array list que contém cada ficheiro e respetiva extensão presente no diretório.
- boolean finishedReadDir - Flag que sinaliza que já foram encontrados todos os ficheiros.
- CorpusFile[] corpusIterator - Estrutura que serve como iterador para ir buscar os ficheiros um a um.

Os métodos para esta classe são os seguintes

- readDir(String path) - método que lê o diretório fornecido no argumento e cria uma lista de ficheiros (CorpusFiles) com todos nos diretórios e subdiretórios.
- ArrayList<CorpusFile> getFiles() - Retorna lista de todos os ficheiros.
- CorpusFile getNextFile() - Retorna o próximo ficheiro, ou null caso não existam mais.

1.1.2. Document Processor

O document processor é a classe responsável por processar o documento consoante a sua extensão, retirar tags específicas do formato. Para isto, é usado um processo que é especificado dependendo da extensão.

Esta classe conta conta com os seguintes atributos:

- Processor proc - Interface para processamento do ficheiro que é especificado dependendo da extensão.
- Buffer buffer - Estrutura adicional que permite ter o conteúdo de n documentos em memória.

Os métodos para esta classe são os seguintes

- processDocument(CorpusFile cfile) - método que vê a extensão do ficheiro recebido e envia para a função interna de processamento adequada à sua extensão.
- DocumentContent getNextDocument() - Função onde as threads vão buscar o proximo documento a ser processado.
- setFinish() - Função que sinaliza no Buffer que já não há mais documentos para processar.

1.1.3. Tokenizer

O tokenizer é a classe responsável por receber o conteúdo do Documento processado e aplicar transformações para se obter uma lista de palavras cujo o resultado será as formas simples de cada palavra(passar para letra minúscula, retirar pontuação, aplicar transformações linguísticas etc.). Esta classe é instanciada para cada documento processado, permitindo assim, a paralelização na tokenização de documentos. É permitido especificar o tipo de tokenização através de parametros.

Esta classe conta conta com os seguintes atributos:

- ArrayList<String> tokens - Lista de todos os token presentes num documento.
- englishStemmer stemmer - Stemmer de termos para os transformar na sua forma simples.
- String [] stopWordsList - Array de stop words recebido no construtor desta classe.

Os métodos para esta classe são os seguintes:

- tokenize(String content, Document doc) - método que recebe o conteúdo de um documento, separa em termos, e aplica transformações, chamando para cada um deles a função transform. Por fim, guarda cada palavra(token) numa Array List de tokens.
- String transform(String term) - método que converte uma palavra para minúsculas e aplica o steamming e stopword filtering etc.

1.1.4. Indexer

O Indexer é a classe responsável por indexar cada palavra presente nos documento fornecidos. Esta classe pode ser instanciada para cada documento de modo a paralelizar o processo de indexação.

Esta classe conta conta com os seguintes atributos:

- Index index - Estrutura de dados para suportar o Índice que vai ser gerado a partir dos documentos.
- int writeThreshold - Variável que indica o número de documentos que podem ser indexados antes de ter de ser efetuada uma escrita para disco. Este valor varia com a memória fornecida à JVM.

Os métodos para esta classe são os seguintes

- indexToken(int docId, HashMap<String,Double> weight, double doc_weight) - método para indexar todas as palavras(tokens) de um documento.
- writeIndex() - Método para escrever para um ficheiro o index ordenado.
- writeLast() - Método usado para escrever a última vez para disco depois de terem sido processados todos os documentos.
- mergeIndex() - Usado no final para agregar todos os índices parciais criados ao longo da indexação.

1.1.5. Searcher

Esta classe não está implementada para esta entrega.

1.2. Classes Auxiliares

1.2.1. CorpusFile

O CorpusFile é a classe responsável por guardar informação de cada ficheiro encontrado pelo CorpusReader.

Esta classe conta conta com os seguintes atributos:

- String path - Path do ficheiro encontrado.
- String extension - Extensão do ficheiro encontrado.

Esta classe contém getters para todos os seus atributos.

1.2.2. DocumentContent

Esta classe é usada para armazenar informação referente a um documento assim como o seu conteúdo. A quantidade deste tipo de objetos em memória é controlada pelo tamanho do buffer.

Esta classe conta conta com os seguintes atributos:

- int docId - Id do documento atribuído sequencialmente.
- String filePath - Nome do ficheiro onde se encontra o documento.
- int startLine - Linha do ficheiro onde se encontra o documento.
- String content - Conteúdo do documento.

Esta classe contém getters para todos os seus atributos.

1.2.3. Index

O Index é a classe responsável pela estrutura de dados usada para a indexação de palavras presentes nos documentos.

Esta classe conta conta com os seguintes atributos:

- HashMap<Integer, HashMap<Integer,Double>> dict - Estrutura usada como índice onde são guardadas as palavras, lista de documentos onde aparece e respetivo peso normalizado dessa palavra no documento
- BidiMap<String, Integer> words - Estrutura para guardar o vocabulário.
- HashMap<Integer, HashMap<Integer,Double>>[] alphabeticIndex - Estrutura final do índice organizada alfabeticamente.

Os métodos para esta classe são os seguintes:

- addTerm(String term, int doc) - Método para adicionar um termo.
- writeWords() - Método para escrever o vocabulário.
- writeIndex() - Método para escrever o indice atual para disco e inicializa um novo.
- mergeIndex() - Método usado para agregar todos os índices parciais usados na indexação.

1.2.4. Buffer

Classe que funciona como monitor com o propósito de guardar documentos em memória para futuro processamento.

Esta classe conta conta com os seguintes atributos:

- nDocs - Número máximo de documentos em memória.
- DocumentContent[] buffer - Array que contém o objeto representativo do documento.
- int nBuffer - Número de documentos em memória num dado instante.
- boolean finish - Flag que sinaliza se já não há mais documentos para processar.

Os métodos para esta classe são os seguintes:

- synchronized addItem(DocumentContent d) - Método para adicionar um documento para ser processado.
- synchronized getItem() - Método para ir buscar o próximo documento para ser processado.
- synchronized setFinish() - Método que notifica que já não existem mais documentos a serem processados.

1.2.5. CsvProcessor

Classe que processa o tipo de ficheiro ‘.csv, processando corretamente todas as tags específicas deste tipo de ficheiro.

Esta classe conta conta com os seguintes atributos:

- Buffer buffer - Instanciação do buffer onde serão colocados os documentos processados.

Os métodos para esta classe são os seguintes:

- String process(CorpusFile file) - Método para processar um Documento que teve origem num '.csv, retirando as tags específicas deste tipo de ficheiro. Este método coloca no buffer este documento processado. Bloqueia se estiver cheio.
- DocumentContent getDocument() - Função que retorna o próximo documento a ser processado. Bloqueia se o buffer estiver vazio.

1.2.6. ArffProcessor

Classe que processa o tipo de ficheiro '.arff', processando corretamente todas as tags específicas deste tipo de ficheiro.

Os métodos para esta classe são os seguintes:

- ArrayList<Document> identify(CorpusFile file) - Método que para cada CorpusFile, separa o ficheiro em documentos.
- String process(Document doc) - Método para processar um Documento que teve origem num '.arff', retirando as tags específicas deste tipo de ficheiro.

1.2.7 DocumentList

Classe que contém a lista de documentos e respectiva localização(ficheiro onde se encontram e linha). Esta classe é usada ainda para a escrita desta lista para disco.

Esta classe conta conta com os seguintes atributos:

- HashMap<String,Integer> filesMapping - Lista de ficheiros.
- HashMap<Integer, Pair<Integer,Integer>> documents - Lista de documentos e respetiva localização.

Os métodos para esta classe são os seguintes:

- addDocument(String filePath, int docID, int line) - método que adiciona um novo documento à lista.
- writeDocuments() - método que escreve a lista de documentos para disco e inicializa uma nova.

1.3. Threads

1.3.1. DP_Worker

Esta thread será responsável por processar os documentos presentes num ficheiro. As threads deste tipo ficam a escuta de novos CorpusFiles identificados pelo CorpusReader para identificar os documentos presentes no mesmo.

1.3.2. TI_Manager

Thread responsável por gerir as TI_Worker, instanciando-as, fornecendo-lhes o documento e garantindo que não existem mais do que o número estabelecido.

1.3.3. TI_Worker

Esta thread será responsável por tokenizar e indexar cada palavra presente num documento. As threads deste tipo recebem um documento, processam-no, indexam, e de seguida, morrem. Só podem existir um número limitado deste tipo de threads em simultâneo.

2. Processo de Tokenização

O processo de Tokenização, que escolhemos para a nossa implementação, assenta, primeiramente, nas seguintes regras aplicadas ao conteúdo todo:

- Os símbolos '-', '(' e ')' são substituídos por espaços, devido à ocorrência frequente neste corpus de várias palavras ligadas por hífens (Ex: Eu-anthracene-9-carboxylic). A utilização de parênteses sem espaços antes ou depois, faria com que a palavra dentro de parêntesis fosse concatenada à palavra imediatamente antes(ou depois).
- Fazer split a um ou mais espaços consecutivos, de modo a obter um array com todas as palavras do documento(tokens).

De seguida são aplicadas as seguintes regras a cada palavra(token), antes de ser indexada:

- Passagem da palavra para minúsculas.
- Retirar todos os símbolos alfanuméricos (Ex: U.S.A -> USA).
- Retirar palavras de apenas um caracter.
- Retirar palavras presentes na lista de stop-words.
- Fazer o stemming das palavras.

O nosso processo de tokenização, indexa números do tipo '0.0001%' como 00001, de modo a que se esta percentagem for pesquisada, o resultado apontar apenas para os documentos que contéem exatamente '0.0001%' e não para todos os que contéem o número 1(Que seria o caso se guardássemos '0.0001%' como '1' no índice). No caso de um número ser seguido de unidades (Ex. '0.01L/s'), este é indexado como 001ls.

3. Data Flow

O nosso Information Retrieval Engine, contém uma main que é responsável por instanciar as classes principais referidas acima.

Para a execução deste programa são utilizadas threads de três tipos, DP_Workers, responsáveis por processar os ficheiros presentes no diretório do corpus, uma TI_Manager responsável por gerir o terceiro tipo de threads que são as TI_Workers, responsáveis por tokenizar e indexar cada documento.

No nosso caso o programa corre com 1 thread DP_Worker, 1 thread TI_Manager e 100 threads TI_Workers. Na nossa análise concluímos que estes números eram os mais indicados para correr o programa e o que obtinha melhores resultados.

O programa inicia-se com uso do CorpusReader para a identificação dos vários ficheiros presentes no diretório (CorpusFile's) que, conforme são identificados, vão sendo processados pelas threads DP_Worker, ou seja, a thread vai processar o próximo ficheiro e identificar todos os documentos nele presentes. De seguida cria um DocumentContent para cada um e coloca-o no Buffer enquanto houver espaço, caso não haja este bloqueia lá à espera que os documentos sejam consumidos. Ao mesmo tempo que a thread DP_Worker está a realizar o seu trabalho a thread TI_Manager está no buffer bloqueada à espera que tenha documentos disponíveis para processar. Quando esta thread adquire um documento ela lança imediatamente um thread TI_Worker e volta ao buffer para ir buscar o próximo documento, até um limite máximo de 100(com as nossas settings) threads TI_Worker ponto que quando é atingido ele espera que estas acabem todas a execução antes de lançar um novo conjunto. Isto é importante para garantir que não estam sempre a ser lançadas threads novas e estas começam a atropelar-se e piorar a performance não só em termos de tempo mas também em termos de memória. Dentro de cada thread TI_Worker o documento é tokenizado, de seguida são calculados os pesos para cada termo e é efetuada a indexação. Consoante a memória disponível para a JVM após um certo número de documentos processados o index actual é escrito para disco e um novo é inicializado. Além da estrutura do index é também guardada uma estrutura com o vocabulário que apenas é escrita para disco no final da indexação e também uma estrutura que contém referências para onde os documentos estão localizados fisicamente no disco.

Após esta fase de indexação é começada uma nova fase de agregação dos índices parciais que foram criados durante a fase anterior. Para isto são identificados todos os ficheiros parciais que foram criados e depois um a um é separado em 27 índices diferentes correspondentes cada um a uma letra do alfabeto. Estes índices pequenos são depois juntos aos índices finais em disco uma letra de cada vez por forma a minimizar o gasto de memória.

No final da execução do programa obtemos 27 ficheiros de índices correspondentes a cada letra do alfabeto mais um de números. Isto na pesquisa vai ser muito vantajoso pois um índice muito mais pequeno vai ter de ser carregado em memória. Além destes ficheiros existe ainda também um ficheiro com o vocabulário, um com a lista de documentos e a sua localização em termos de ficheiro + linha original e ainda um ficheiro que mapeia o path dos

ficheiros originais com um id desta forma não tem de ser guardado um path para cada documento mas sim apenas um id.

4. Análise de resultados

RAM	512MB	1024MB	4096MB
Tempo de Indexação	388seg (6min 27seg)	403seg (6min 42seg)	588seg (9min 48seg)
Tempo de criação do índice alfabético			445seg (7min 24seg)

Na tabela acima é possível observar os resultados obtidos para a execução do programa com diferentes níveis de RAM dadas à JVM.

Ao contrário do que seria esperado a execução com menos RAM conseguiu um tempo menor isto foi devido a dois fatores. O primeiro é que foi utilizada uma biblioteca de serialização sem ser a default do java o que acelerou muito a escrita e por isso tornou a penalização por escrever no disco muito mais baixa do que seria. O segundo fator é que como ia mais vezes a disco o index era não se tornava tão grande o que torna a inserção mais rápida. Em contrário a execução com 4gb de ram tornava o índice em memória maior por precisar de escrever menos vezes e consequentemente a inserção de novas itens ao índice mais lenta.

Contudo, e após concluída a fase de indexação, na fase de agregar os índices apenas a execução nos 4gb de memória consegue acabar. Este é um problema que nos surgiu e que ainda não conseguimos resolver, mas nas implementações com menos memória o programa não consegue terminar de agrupar todos os índices. Este problema vai continuar a ser trabalhado esperando estar resolvido para a próxima entrega.

Todos estes testes foram realizados num Macbook Pro 15 (mid 2012) e podem variar consoante o computador onde foram executados.

5. Execução do programa

Para executar corretamente o programa, é necessário, da primeira vez, fazer clean and build e só depois executar. Na main, existem ainda vários parâmetros configuráveis.

6. Anexos

Por motivos de visualização, o diagrama de classes encontra-se num ficheiro à parte.