Boolean Retrieval Model

Dulio Paolo Caggiano Amand
dulio.caggianoad@udlap.mx

Oscar Iván de Alva Martínez
oscar.dealvamz@udlap.mx

Universidad de las Américas Puebla — September 6, 2019

Introduction

In this practice, research and development focused on the *Boolean Information Retrieval Model* was carried out to define the basic concepts of a retrieval model. For the development of this project, a dictionary and a posting list were used in order to implement an Inverted iIdex based on the Cranfield collection.

For the implementation of this project, the programming language *Scala* was used as it provides a concise notation that grants the creation of a *Domain Specific Language* (DSL) for the ability to model and create a solution for the problem. Also, *Scala* has a tight compatibility and interoperability with *Java*, that allows to take advantage of both the *Java Virtual Machine* (JVM) and the libraries of *Java*, and at the same time having its own libraries. Furthermore, its strict type system has the advantage of allowing the creation of specific types that model the solution of this problem.

1 Boolean Retrieval Model

The Cranfield collection is based on three main components, the first one is a collection of documents which contains information from different investigations. The second component is the one that contains a set of specific queries for the aforementioned data collection and the last component contains the relevance judgments which are all the factors that are taken in consideration to categorize a document as relevant.

The documents contained in the Cranfield collection generally maintain a structure, where the information contained in the document is specified by means of headings, for example, the usage of the header *.I* to describe the documents ID or *.A* to list the authors of the document.

1.1 Parsing the collection

To transform the words found in the documents, a dedicated parser was created to perform the tokenization of these. This parser has the functionality of creating tokens from the headers established within each document, which allows establishing metadata, in the form of classes, for the words found.

Also, specific classes were created for the documentation of words and metadata, and a class was established to store the content of a document. These classes represent the different types of header found within the document, these are: document number, author, title, affiliation, and abstract text. The established classes are the following:

```
sealed trait Header
case class Id(id: Int) extends Header
case class Title(text: String) extends Header
case class Author(fullName: String) extends Header
case class Adscription(adscription: String) extends Header
case class Abstract(text: String) extends Header
case class Document(tags: List[Header])
```

On the other hand, the internal structure of a document was established as a List of parsed headers. This decision was made because not all of the documents within the collection have a defined structure, which makes it difficult to generalize and represent them with only one.

In order to parse the words, the **cran.all.1400** file was first divided into its documents, which are specified at the beginning with the *.I* header and followed by the document number. This was done in order to allow an easier identification of the words inside each document.

Then, each document was divided into the sections previously mentioned, which are identified by the headings: *.A*, *.T*, *.B* and *.W*, along with the document number where they are located. Finally, these elements were stored within a **Document** class to indicate that a document from the collection was parsed along with its contents.

1.2 Inverted Index

The *Inverted Index* was implemented by using a *Dictionary* which gave mainly the advantage of fast access time once you search for a word. The keys used in the dictionary were all the different words found in the documents and the values are stored inside a *Tuple*, with the first element representing the number of occurrences that the word has in all the processed documents and the second element is a *Set* which contains the IDs of the documents in which the word is found. The *Set* data structure is used to avoid the repetition of IDs, taking advantage of this characteristic.

Once the whole file is parsed and the words are tokenized, the list of tokens is processed element by element, adding them to the *Inverted Index*. When adding an element to the *Inverted index* and it is already stored inside of it, then the ocurrence counter is increased and the document id where it appeared is added to the *Set*.

1.3 Parsing the query

To parse the query that the user enters in the program, a free context grammar was used, since the structure of a query was already defined. This grammar is based upon the grammar used to solve basic algebraic operations. This decision was made because the operators dealing with the *Boolean Retrievl Model* work the same as the basic mathematical operators of **+**, * and ***- of a scalar***. The grammar for a query is defined as follows:

$$
\begin{aligned}
Expr &::= Term(\text{'OR'}Term)* \\
Term &::= Factor(\text{'AND'}Factor)* \\
Factor &::= [\text{'NOT'}](Word|\text{'('}Expr\text{')'}) \\
Word &::= [a-zA-Z]
\end{aligned}
$$

This definition allows the parser to be able to analyze a query through the precedence of the **AND**, **OR** and **NOT** operators, as well as the parentheses. To make the parsing of the query, the Scala Parser Combinators library was used, since it allows to represent a grammar free of context in code and define the procedurements while parsing. This library uses its own DSL to both model the grammar and the Succes case if the expression is a match. The following code represents the way to express the *Expr* case of the grammar:

```
def expr: Parser[BinaryArray] = term ~ rep("or" ~> term) ^^ {
    case expr ~ list =>
      list.foldLeft(expr) {
        case (x, y) => x | y // Make the OR operator of the terms
      }
  }
```

The *CranParser* is in charge of identifying the specified grammar cases and asking the *Boolean Retrieval Model* to carry out a specified operation. When the parser finds a word in query, it calls the **convertToBinaryArray(word)** method of the *Boolean Retrieval Model* in order to tokenize the word based upon the following method:

```
private def word: Parser[BinaryArray] =
    """[a-zA-z]+""".r ^^ (brm.convertToBinaryArray(_))
```

1.4 Boolean Retrieval Model

When the words are parsed inside the query, they are transformed into a *BinaryArray* class, which is an *Array* of *Shorts* of the size of the number of processed documents, being 1400 for the Cranfield documents. The *BinaryArray* limits the data that can be used in the array to 0 and 1 in order to represent the ocurrence of a word inside the different documents in a binary way. This representation consists of a 1 in the position **i** if the element is in the **document i** and if it is not found inside **document i** then will be represented with a 0. The following method is in charge to convert a word to its *BinaryArray* representation:

```
def convertToBinaryArray(word: String) = {
    index.get(word) match {
      case Some(value) =>
        value match {
          case (_, documents) =>
            documents
              .map(_.id)
              .foldLeft(BinaryArray.fill(index.numberOfDocs)(0)) {
                case (acc, index) => acc.update(index - 1, 1)
              }
        }

      case None => BinaryArray.fill(index.numberOfDocs)(0)
    }
  }
```

The **convertToBinaryArray(word)** method is in charge to convert a word to a *BinaryArray* making the following steps:

(i) Get the word from the *Inverted Index*.

(ii) If the word is a key in the index:

  (a) Get its documents where it appears, its *Set[Id]*.

  (b) Generate a new *BinaryArray* that is empty (all its indexes are 0).

  (c) Start filling this array with 1's in the index where a word appears.

(iii) If its not in the index: return an empty *BinaryArray*, since its not in any document.

After each word is represented as a *BinaryArray*, if the query contains the key words *AND*, *OR* or *NOT*, then the corresponding binary operation is performed between the two arrays, based on the precedence of operations previously defined in the grammar of the *CranParser* that tokenizes the query.

The resulting array from performing the binary operations, will be the result of the query. The positions that are marked with a 1 are the number of the documents that are the result of the user's query, being the indexes that are considered relevant for the search.

2 Conclusions

Through out the project, the steps for the creation of a retrieval model were identified and developed. Starting from the parsing of the collections, to the implementation of both an *Inverted Index* and a *Boolean Retrieval Model*.

In the same way, the impact of the data structures and the importance that they have within the development of a retrieval model were observed. Especially in both effectiveness and efficiency of a retrieval model.

Finally, the methods to parse both the documents of the *Cranfield* collection and the user's queries were analyzed. This was in order to process and obtain information from the sources of the Information Retrieval Model, along with analyzing the implications that they entail.