Chapter 2
Information Retrieval

2.1 Introduction

The provision of appropriate navigation facilities is an essential requirement for any Website. These navigation facilities may include well-designed hyperlink structure, menus, and site maps. However, it is difficult to predict all possible navigational patterns, and provide corresponding navigation paths. The problem is even more complicated for a dynamically growing Website in which documents are deposited by several personnel. Any site of sufficient complexity should provide a search facility to simplify the navigation. There are many generic commercial and non-commercial search engines. Sometimes, installation of these generic search engines will satisfy the requirements of a Website. Other times, it may be necessary to develop a site-specific search engine. Such a search engine can be developed from scratch or assembled using various software components tuned to satisfy the needs of your Website. Whether one uses a generic or customized search engine, it is important to evaluate its retrieval performance on a regular basis. This chapter provides a detailed introduction to Information Retrieval (IR) that can be useful for the development, implementation, and evaluation of a search engine. The knowledge of information retrieval process may also help Web developers create innovative navigation tools, such as dynamically changing search engines.

For over 4000 years, humans have been designing tools to improve information storage and retrieval. Table of contents and the index table of a book are examples of

some of the early efforts. With the increasing collection of documents, various databases have been created for the summarization, searching, and indexing of these documents. The foundation of the modern information retrieval is credited to the 1945 paper "As We May Think" published by Vanover Bush, who visualized an imaginary IR machine called *Memex*. The SMART (System for the Manipulation and Retrieval of Text), conceived at Harvard University and flourished at Cornell University under the leadership of Gerald Salton, provided the first practical implementation of an IR system. The basic theoretical foundations of SMART still play a major role in today's IR systems. The four major components of an IR process are:

- document representation,
- query representation,
- ranking the documents by comparing them against a query using a retrieval model, and
- evaluation of the quality of retrieval.

This chapter will discuss the theoretical foundations of these four components, along with algorithms, their implementations, and descriptions of non-commercial information retrieval software.

2.2 Document Representation

The documents on the Web consist of a variety of different formats and the information may consist of text, graphics, audio, and video. This chapter deals with traditional text based information retrieval. Multimedia information retrieval is still in its infancy. Some initial attempts are discussed in Chapter 7. The IR process described here will be based

on text extracted from different types of documents. We will look at some of the tools that make it possible for us to retrieve text from various document formats. Traditional IR represents documents using keywords. In some cases, a keyword may consist of multiple words. Reducing text to a list of keywords is a simplistic representation of a document, as it ignores the semantics in the document. Chapter 3 will look at some of the early attempts of incorporating semantics in our searches.

Figure 2.1 shows the typical steps involved in transforming a document into a list of keywords with associated frequencies or weights. The first step in the transformation of a document is simply listing all the words in a document by removing spaces, tabs, new line characters, and other special characters such as a comma, period, exclamation mark, and parentheses. The second step is the removal of some of the most commonly occurring words. Words that appear in, for example, more than 80% of documents will not be very useful in discriminating documents. These words are usually referred to as *stopwords*. Natural candidates for stopwords are articles, prepositions, and conjunctions. They are filtered out from the list of potential keywords. Another advantage of eliminating stopwords is the reduction in the size of the document representation.

Figure 2.2 gives a partial list of typical stopwords in the English language. A more complete list of stopwords can be found on the CD-ROM under Chapter 2. Figure 2.3 shows an example of a document collection with four documents that will be used for illustrating the theory and implementation of information retrieval systems.

The CD-ROM contains Java classes that will help us create a simple information retrieval system. Clicking on the “Simple Information Retrieval System” link will reveal various links related to the IR system. One of the folders, “java”, contains Java classes

(source and class files) used for development of the system. Let us look at the first class called `TokenizedDoc`. The class is used for creating a list of words (also referred as terms, index terms, or tokens) from a file. The words are listed in alphabetical order and exclude stopwords.

> *Readers who are not familiar with Java can skip the highlighted paragraph below*.
>
> The link javadoc provides documentation for all the Java classes used in Chapter 2. Figure 2.4 shows the information about the constructor and methods of the `TokenizedDoc` class. The class only has one constructor, which takes three parameters. The first parameter is a document represented as a string, the second parameter is the list of characters in addition to the white spaces that may be used as word or token separators, and the third parameter is the name of the file containing stopwords. The function getTokens returns the vector of words/tokens in the document. The main function in the class shows an example of how to call the constructor. The main program can be used to make standalone use of the class.

Before testing the class, let us copy all the relevant files to a directory or folder called `IR`. We will assume basic understanding of using command line interface under windows and/or Linux. We will use the Linux conventions for directory specifications. Windows users should substitute / with \, whenever it is appropriate. At the command prompt, change your directory to `IR/fig2.3`. This directory contains electronic copies of four

documents from Figure 2.3. The first document from Figure 2.3 is called `d1.txt`. The class `TokenizedDoc` can be run for the file `d1.txt` using the following command:

```
java -cp ../java TokenizedDoc d1.txt "\!\?\[\].,;-" ../misc/stopwords.txt    (Command 2.1)
```

The command java is used to run a java class file.

- The option `-cp` is used to specify the class path, i.e., where the java class files can be found. In our case, the class path of `../java` (second argument) is a subdirectory `java` under the main directory `IR`.
- The third argument `TokenizedDoc` specifies the name of the class.
- The fourth argument gives the name of the document to be `d1.txt`.
- The fifth argument specifies the additional (non-white) characters that should be used as word separators. A backslash, used before !, [, and], ensures that these characters are not treated as special characters.
- The sixth argument, `../misc/stopwords.txt`, mentions that the file containing stopwords is called `stopwords.txt` and is located in the subdirectory `misc` under the main directory `IR`.

The list of documents using (Command 2.1) will be printed on the screen. If you wish to store the documents in a file, you should redirect the output as shown:

```
command > output-file-name
```

For example, appending `> Tokenized-d1.txt` to the (Command 2.1) as:

```
java -cp …… > Tokenized-d1.txt
```

will allow us to store the list of tokens from `d1.txt` into a file called `Tokenized-d1.txt`. The file `Tokenized-d1.txt` is in the `IR/fig2.3`

directory on the CD-ROM and is also shown in Figure 2.5. The list is alphabetically sorted to make it easy to count the frequency of each word. It also underscores the fact that keyword based retrieval tends to ignore the semantic structure of documents. The files `d2.txt, d3.txt, d4.txt` under the subdirectory `fig2.3` can be used to create a similar list of words for the rest of the documents from Figure 2.3 (see Exercise section).

2.2.1 Stemming

A given word may occur in a variety of syntactic forms, such as plurals, past tense or gerund forms (a noun derived from a verb). For example, the word "connect", may appear as "connector", "connection", "connections", "connected", "connecting", "connects", "pre-connection", and "post-connection". A *stem* is what is left after its affixes (prefixes and suffixes) are removed. In our example, "ed", "s", "or", "ed", "ing", and "ion" are prefixes, while "pre-" and "post-" are prefixes that will be removed to form the stem "connect". It can be argued that the use of stems will improve retrieval performance. The users rarely specify the exact forms of the word they are looking for. Moreover, it seems reasonable to retrieve documents that contain a word similar to the one included in a user request. For example, a document containing the word "connection" may be relevant to a user request that includes the word "connect". Reducing words to stems also reduces the storage required for a document representation by reducing the number of distinct index terms.

Researchers have conflicting opinions about the value of stemming in an information retrieval process. Some of the search engines do not use stemming in their

document representations. Nevertheless, stemming is an important part of document preparation in information retrieval, and hence, we should study it in more detail.

There are several stemming strategies. For example, one can simply maintain a table of all the words and corresponding stems. Stemming in that case will involve a simple table lookup. This strategy will require significant storage, assuming that data on every word in the language is available. Other strategies include successor variety based on structural linguistics, or N-grams based on term clustering. These strategies can be relatively complex. Affix removal is one of the simplest stemming strategies that is intuitive and can be easily implemented. We will study affix removal in greater detail. It may be desirable to combine affix removal with table lookup for those words that cannot be easily stemmed.

While affixes mean prefixes and suffixes, suffixes appear more frequently than prefixes. There are a few suffix removal algorithms. The most popular algorithm was proposed by Martin Porter (Porter, 1980). The Porter algorithm is known for its simplicity and elegance. Even though the algorithm is simple, the stemming results from the Porter algorithm compare favourably to more sophisticated algorithms. The following is a detailed description of the Porter algorithm.

2.2.1.1 Porter's Stemming Algorithm

Martin Porter maintains an official page for the algorithm at: http://www.tartarus.org/~martin/PorterStemmer/index.html. The information in this section includes verbatim description of the five steps of the algorithm (Figures 2.6-2.10) from Porter's website. The explanation used here is also provided from the website.

In order to understand the algorithm, we need to define a few terms. Letters A, E, I, O or U are *vowels*. A consonant in a word is a letter other than A, E, I, O or U, with the exception of Y. Letter Y is a vowel if it is preceded by a consonant otherwise it is a consonant. For example, Y in "SYNOPSIS" is a vowel, while in "TOY", it is a consonant. A consonant in the algorithm description is denoted by `c`, and a vowel by `v`. A list `ccc...` of length greater than 0 will be denoted by `C`, and a list `vvv...` of length greater than 0 will be denoted by `V`. Any word, or part of a word, therefore has one of the four forms:

```
CVCV ... C
CVCV ... V                                    (2.1)
VCVC ... C
VCVC ... V
```

Square brackets are used to denote optional presence of a sequence. Therefore, the four forms shown in Eq. (2.1) can be represented by the single form:

```
[C]VCVC ... [V]                               (2.2)
```

The braces {} are used to represent repetition. For example, `(VC){m}` means `VC` repeated `m` times. Therefore, Eq. (2.1) or (2.2) can also be written as:

```
[C](VC){m}[V]
```
(2.3)

Here, `m` will be called the *measure* of any word or word part. For a *null* word `m = 0`. The following are some of the examples of various values of the measure:

```
m = 0    TR, EE, TREE, Y, BY.
m = 1    TROUBLE, OATS, TREES, IVY.
m = 2    TROUBLES, PRIVATE, OATEN, ORRERY.
```

The strings for `m = 0`, match `[C][V]`. The strings for `m = 1`, have only one occurrence of `(VC)` between `[C][V]`, and so on. The rules for removing a suffix in Figures 2.6-2.10 are given in the form:

```
(condition) S1 -> S2
```

The `condition` is usually given in terms of `m`. If the stem before `S1` satisfies the condition, then replace `S1` by `S2`. For example, in the rule

```
(m > 1) EMENT ->
```

`S1` is "`EMENT`" and `S2` is null. The above mentioned rule would, for example, map "REPLACEMENT" to "REPLAC", since "REPLAC" is a word part for which m = 2. The condition part may also contain the following expressions:

```
*S  - the stem ends with S (and similarly for the other letters).
*v* - the stem contains a vowel.
*d  - the stem ends with a double consonant (e.g., -TT, -SS).
*o  - the stem ends cvc, where the second c is not W, X or Y
      (e.g., -WIL, -HOP).
```

Finally, the condition part may also contain expressions with logical operators `and`, `or`, and `not`. For example,

```
(m > 1 and (*S or *T))
```

matches a stem with `m > 1` ending in `S` or `T`. Similarly, in the condition,

```
(*d and not (*L or *S or *Z))
```

a stem will end with a double consonant other than `L`, `S` or `Z`.

In a set of rules that follow each other, only the one with the longest matching `S1` for the given word is obeyed. For example, consider the following sequence of rules (with null conditions):

```
SSES -> SS

IES  -> I

SS   -> SS

S    ->
```

The word "CARESSES" stems to "CARESS", since "SSES" is the longest match for `S1`. Similarly, "CARESS" stems to itself (`S1`= "SS") and "CARES" to "CARE" (`S1`= "S").

Now we are ready to look at the five steps of the Porter algorithm. Figures 2.6-2.10 show the rules used in the Porter algorithm in every step. Applications of the rules are given on the right in lower case. Step 1 given in Figure 2.6 deals with plurals and past participles. The subsequent steps 2-4, given in Figures 2.7 to 2.9, show the relatively straightforward stripping of suffixes. Step 5 (Figure 2.10) is used for tidying up. Complex suffixes are removed in several stages. For example, GENERALIZATIONS is stripped as follows:

Step 1: GENERALIZATION

Step 2: GENERALIZE

Step 3: GENERAL

Step 4: GENER

Similarly, OSCILLATORS is stripped as follows:

Step 1: OSCILLATOR

Step 2: OSCILLATE

Step 4: OSCILL

Step 5: OSCIL

The algorithm does not remove a suffix when the length of the stem, given by its measure (m), is small. For example, consider the following two lists:

list A	list B
RELATE	DERIVATE
PROBATE	ACTIVATE
CONFLATE	DEMONSTRATE
PIRATE	NECESSITATE
PRELATE	RENOVATE

The words from List A have small measures, hence -ATE is not removed. However -ATE is removed from the words from List B, which have larger measures.

In an experiment reported on Porter's site, a vocabulary of 10,000 words reduced the words in various steps, as shown in Figure 2.11. After reduction, there were 6370 stems left in the list. That is, the suffix stripping process using Porter's algorithm reduced the size of the vocabulary by 36%. Figure 2.16 (details of the figure will be discussed a little later) shows the list of stems left after eliminating the stopwords and stripping the words from document d_1 in Figure 2.3. The original document contained 184 words. After the elimination of stopwords and duplicate stems, there were 73 distinct words found in Figure 2.3. The aforementioned document was a summary course description. A longer

document would have resulted in additional reductions. The two numbers after each stem in Figure 2.16 represent the frequency and normalized frequency of the stem in the document (discussed later).

2.2.1.2 Stemmer Software

The Porter algorithm has been implemented in a variety of programming languages. The original implementation of the algorithm was in BCPL, an ancestor of C. The official site of the Porter Stemming Algorithm (http://www.tartarus.org/~martin/PorterStemmer/) provides links to its implementation in C, Java, Perl, PHP, C#, Python, Common Lisp, Visual Basic, Ruby, and Javascript. The links to these versions can be found under Chapter 2 on the CD-ROM. A Java version (Stemmer.java) and its class file (Stemmer.class) are also available on the CD-ROM. The compilation and execution of the stemmer can be done at the command prompt (Windows, UNIX, or MacOS). First, change the directory to `IR/porter`. The Java file can then be compiled using the command:

```
javac Stemmer.java
```

This command will produce the class file that can be run using the command:

```
java Stemmer input.txt
```

It is assumed in the command above that you have a file with a list of words. The program will go through the list and output the stemmed versions of the word. The file `voc.txt` contains a sample of vocabulary from the official site of the Porter algorithm. The corresponding output is given in `vocoutput.txt`. The files `Stemmer.java`, `Stemmer.class`, `voc.txt`, and `vocoutput.txt` are stored under a subdirectory

called `porter` in the `IR` directory. Duplicate copies of files `Stemmer.java` and `Stemmer.class` are also stored in the `java` subdirectory. Readers should see the effects of running the Stemmer on `Tokenized-d1.txt` from the subdirectory `fig2.3` by typing the following command:

```
java -cp ../java Stemmer Tokenized-d1.txt                    (Command 2.2)
```

The results of redirecting the output from (Command 2.2) appear in the file `Stemmed-d1.txt` in the subdirectory `fig2.3`.

2.2.2 Term Document Matrix

Term document matrix (TDM) is a two dimensional representation of a document collection. The rows of the matrix represent various documents, and the columns correspond to various index terms. The values in the matrix can be either the frequency or weight of the index term (identified by the column) in the document (identified by the row). Figure 2.12 shows an abstract representation of a document collection. It is assumed that the document collection contains 7 documents: D0 to D6. These documents are represented by 7 keywords: K0…K6. We use the C-array convention for numbering lists. For a list of *n* items, the numbering starts at 0 and ends at *n*-1. The cell values denote the frequency of the keywords in the documents. For example, in Figure 2.12, the keyword K2 appears in document D0 5 times. Similarly, the keyword K3 appears 7 times in document D3.

Usually, the number of keywords is large. For example, in our document collection from Figure 2.3 with four very short documents, the number of keywords is 172. Most of the documents do not contain every one of these keywords. Hence, the

term-document matrix is usually sparse, i.e., many of its entries are zeroes. In order to save storage, usually only non-zero entries are stored. One of the popular representations of a sparse matrix uses triplets (row, column, value) for non-zero entries. For example, the term-document matrix from Figure 2.12 will be represented using the triplet as shown in Figure 2.13. There were 49 entries in Table 2.12. Only 24 of those entries need to be stored in table 2.13. In this example, since we are storing three values for each non-zero entries, we end up storing 72 numbers. If we had created separate storage for each document, we could have eliminated the need to store the first number corresponding to the document number. Each non-zero entry in this case will be represented using a pair (column,value). We can use a line character to distinguish the rows/documents. This will give us the representation of our document collection as shown in Figure 2.14. In this case, we store 48 numbers to represent 49 values from our original matrix. Our abstract example does not illustrate the reduction in storage very well. For the document set from the four short documents in Figure 2.3, with 173 keywords, we will require 172*4=688 entries in the term-frequency matrix. However, there are only 195 non-zero entries. If we use a triplet (row,column,value), we will need 195*3=585 numbers. The savings correspond to 103 integers or 14.97%. If we use a pair, we will need 195*2=390 number. In this case, the savings will be 298 numbers or 43.31%. The savings are even more noticeable in larger and more diverse document collections.

Usually, the raw frequency values are not useful for a retrieval model. Many retrieval models prefer normalized weights, usually between 0 and 1, for each term in a document. Dividing all the keyword frequencies by the largest frequency in the document

is a simple method of normalization. Mathematically, we can write the equation for calculating the weight of a term in a document as:

$$w_{ij} = \frac{freq_{ij}}{\underset{k=1}{\overset{m}{\text{MAX}}} freq_{ik}}, \tag{2.4}$$

where w_{ij} is the weight , and $freq_{ij}$ is the frequency of the j^{th} keyword in i^{th} document. It is assumed that there are *m* terms in a document collection. That is, the number of columns in the TDM is *m*. Figure 2.15 shows the normalized version of the term document matrix from Figure 2.12. Whenever necessary, we will qualify the term document matrix with either frequency or weight. For example, the matrix given in Figure 2.12 is the term document frequency matrix, while the one in Figure 2.15 is the term document weight matrix.

The `java` subdirectory under the `IR` directory contains a class called `DocVector` that can be used to create each row of a term document matrix. Each element of the row is represented as a triplet (word, frequency, normalized frequency). It should be noted that the class creates a representation that takes a little more space than an efficient implementation requires. For example, a word could be stored by its index similar to Figure 2.14, and only frequency or normalized frequency needs to be stored for a given information retrieval model. In fact, the Boolean retrieval model requires neither the frequency nor the normalized frequency, only the presence or absence of a term, which can lead to a very efficient storage. However, the document representation created by `DocVector` is more readable and easy to use in simple implementations of a variety of

information retrieval models. The class `DocVector` is run essentially the same way as the class `TokenizedDoc`.

```
java -cp ../java DocVector d1.txt "\!\?\[\].,;-" ../misc/stopwords.txt     (Command 2.3)
```

The redirected output from (Command 2.3) is stored in the file `Vector-d1.txt` also shown in Figure 2.16. Readers are encouraged to run the same command for the other three text files from Figure 2.3 (see the Exercise section). Unlike Figure 2.14, the triplets from a document vector appear as one per line. In order to store the entire term document matrix, we need separators. Following the convention from the SMART collections, we will use a line starting with '.I' followed by document ID to identify the beginning of the file. In fact, the file `Vector-d1.txt` begins with the line `.I d1.txt`. The vector representations of the remaining three documents from Figure 2.3 can be created similarly using a variation of (Command 2.3). The concatenation of resulting files can then be used as a term document matrix (see the Exercise section).

Readers who are not familiar with Java can skip the following highlighted paragraph.

Figure 2.17 shows information about the constructor and methods of the `DocVector` class. The class has two constructors. One of them is a default constructor that does not initialize any fields. The other constructor takes two parameters: `Vector` of the words in the document, and document's ID. The constructor then uses two private functions to

- stem the word,
- eliminate duplicates, and
- calculate frequency and normalized frequency of the stemmed words.

One member function, `getVector()`, returns the vector representation of the documents as a `Vector` of objects of type `Term`, a triplet (word, frequency, normalized frequency). The class `Term` is discussed at the end of this paragraph. The other member function `getID()` returns the ID of the document. The functions starting with "`set`" make it possible to set the values of the two fields in the class. The constructor expects the file processing is done elsewhere (`TokenizedDoc` class). The main function in the class shows an example of how to use the `TokenizedDoc` class before calling the constructor of the `DocVector` class. The main program can be used to make standalone use of the class. Figure 2.18 shows relevant portions of the main function. The program segment between the `try` and `catch` block reads a document as a string. The string representing a document is then passed along with another string containing separators, and the name of the stopwords file to construct an object of the type `TokenizedDoc`. The `Vector` of words/tokens from the `TokenizedDoc` is then used to construct the vector representation of documents using the triplet (word, frequency,

normalized frequency). The triplet is represented using a Java class called `Term`. It consists of three fields:

- `word`: A string representing the actual term.
- `freq`: The number of times the term appears in the document.
- `normalizedFreq`: Given by Eq. (2.4).

Figure 2.19 shows the summary of constructors and various methods to access the fields of the class `Term`. The constructor needs the value of `word` in order to construct an object of type `Term`. The functions beginning with "`get`" are used to get the value of a field. Similarly, functions beginning with "`set`" are used to change the value of a field.

2.2.3 Standard Document Collections

The experiments described so far are based on a document set with four very short documents given in Figure 2.3. For any significant experimentation, one needs to work with a larger number of documents with more significant contents. This section describes some of the standard document collections that are available for experimentation. These collections are widely used by IR researchers and therefore also provide a good means of conducting comparative studies. Many of these document collections are too large to be provided on the CD-ROM accompanying this textbook. Therefore, the URLs for these collections appear both on the CD-ROM, as well as in this section. Additionally, the CD-ROM contains smaller document collections that can be used for preliminary experimentation.

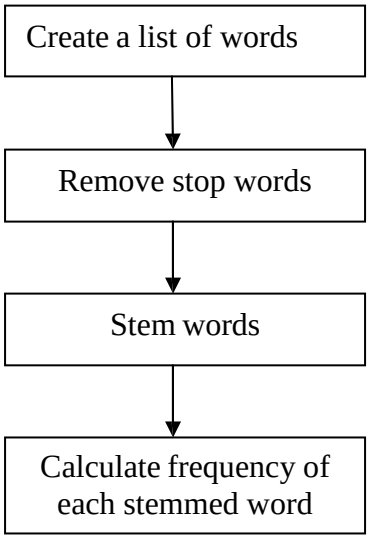

Figure 2.1 Transforming text document to a weighted list of keywords

I a	how	was
about	in	were
an	is	what
are	it	when
as	la	where
at	of	who
be	on	why
by	or	will
com	that	won’t
de	the	with
en	their	within
for	there	without
from	this	und
	to	www

Figure 2.2 A partial list of stopwords

Data Mining has emerged as one of the most exciting and dynamic fields in computing science. The driving force for data mining is the presence of petabyte-scale online archives that potentially contain valuable bits of information hidden in them. Commercial enterprises have been quick to recognize the value of this concept; consequently, within the span of a few years, the software market itself for data mining is expected to be in excess of $10 billion. Data mining refers to a family of techniques used to detect *interesting* nuggets of relationships/knowledge in data. While the theoretical underpinnings of the field have been around for quite some time (in the form of pattern recognition, statistics, data analysis and machine learning), the practice and use of these techniques have been largely ad- hoc. With the availability of large databases to store, manage and assimilate data, the new thrust of data mining lies at the intersection of database systems, artificial intelligence and algorithms that efficiently analyze data. The distributed nature of several databases, their size and the high complexity of many techniques present interesting computational challenges.

This course will expose the students to research in applied sciences. The objectives of the course will be achieved through various active learning sessions that involve critical review of papers from scholarly journals and conferences. The activities will also include the preparation and presentation of annotated bibliographies, literature reviews, thesis abstracts, and research project outlines. Students are also required to provide feedback to their colleagues. It is hoped that students will refine their own presentation skills through critically reviewing other presentations. In addition to the regular class activities, students must attend and submit written reports for a total of six (three per semester) external seminars.

~~This course is designed to extend the student's knowledge of, and~~ provide additional hands-on experience with, the programming language encountered in CSC
226, in the context of the structured data types provided by that language, and within the larger contexts of abstract data types and more complex problem-solving situations. Techniques for managing file input and output in the current language will also be studied. A number of classical algorithms and data structures for the storage and manipulation of information of various kinds in a computer's internal memory will be studied. The student will acquire the knowledge that comes from actually implementing a non-trivial abstract data type and the experience that comes from having to make use, as a client programmer, of an abstract type that has already been implemented.

This course consists of a study of general language design and evaluation. The course will include examination of the design issues of various language constructs, design choices for these constructs in various languages, and comparison of design alternatives. Students will program in a variety of programming languages (FORTRAN, Pascal, C, C++, Java, C#, Lisp, and Prolog) to gain a better understanding of the theoretical discussion. In addition, students will get exposure to other languages ranging from Algol-60 to Ada-95.

Figure 2.3 A sample document collection = {d_1, d_2, d_3, d_4}

Constructor Detail

TokenizedDoc

```
public TokenizedDoc(java.lang.String doc,
                    java.lang.String separators,
                    java.lang.String stopWordFile)
```

Parameters:

`doc` - represented as a single string
`separators` - String of characters used as separators, e.g. "!?[](),;.?"
`stopWordFile` - name of the file containing stopwords, e.g. "stopwords.txt"

Method Detail

getTokens

```
public java.util.Vector getTokens()
```

Returns:

A vector of words/tokens.

---

main

```
public static void main(java.lang.String[] args)
```

Parameters:

`args` - An array of Strings, e.g. d1.txt "\!\[\](),;.?" stopwords.txt

---

Figure 2.4 Methods for `TokenizedDoc` class
(Suitable for Java programmers)

```
ad              Data            field           mining          relationships
algorithms      data            fields          mining          scale
analysis        data            force           mining          science
analyze         data            form            mining          size
archives        data            hidden          mining          software
artificial      database        high            nature          span
assimilate      databases       hoc             nuggets         statistics
availability    databases       information     online          store
billion         detect          intelligence    pattern         systems
bits            distributed     interesting     petabyte        techniques
challenges      driving         interesting     potentially     techniques
commercial      dynamic         intersection    practice        techniques
complexity      efficiently     large           presence        thrust
computational   emerged         largely         present         time
computing       enterprises     learning        quick           underpinnings
concept         excess          lies            recognition     valuable
data            exciting        machine         recognize       years
data            expected        manage          refers
data            family          market
```

Figure 2.5 List of words in d_1 after deleting stopwords

```
Step 1a

    SSES -> SS                          caresses  ->  caress
    IES  -> I                           ponies    ->  poni
                                        ties      ->  ti
    SS   -> SS                          caress    ->  caress
    S    ->                             cats      ->  cat

Step 1b

    (m>0) EED -> EE                     feed      ->  feed
                                        agreed    ->  agree
    (*v*) ED  ->                        plastered ->  plaster
                                        bled      ->  bled
    (*v*) ING ->                        motoring  ->  motor
                                        sing      ->  sing

If the second or third of the rules in Step 1b is successful, the following is
done:

    AT -> ATE                       conflat(ed)  ->  conflate
    BL -> BLE                       troubl(ed)   ->  trouble
    IZ -> IZE                       siz(ed)      ->  size
    (*d and not (*L or *S or *Z))
       -> single letter
                                    hopp(ing)    ->  hop
                                    tann(ed)     ->  tan
                                    fall(ing)    ->  fall
                                    hiss(ing)    ->  hiss
                                    fizz(ed)     ->  fizz
    (m=1 and *o) -> E               fail(ing)    ->  fail
                                    fil(ing)     ->  file

The rule to map to a single letter causes the removal of one of the double
letter pair. The -E is put back on -AT, -BL and -IZ, so that the suffixes
-ATE, -BLE and -IZE can be recognised later. This E may be removed in step
4.

Step 1c

    (*v*) Y -> I                    happy        ->   happi
                                    sky          ->   sky
```

Figure 2.6. First step in the Porter algorithm (http://www.tartarus.org/~martin/)

```
Step 2

     (m>0) ATIONAL ->  ATE           relational     ->  relate
     (m>0) TIONAL  ->  TION          conditional    ->  condition
                                     rational       ->  rational
     (m>0) ENCI    ->  ENCE          valenci        ->  valence
     (m>0) ANCI    ->  ANCE          hesitanci      ->  hesitance
     (m>0) IZER    ->  IZE           digitizer      ->  digitize
     (m>0) ABLI    ->  ABLE          conformabli    ->  conformable
     (m>0) ALLI    ->  AL            radicalli      ->  radical
     (m>0) ENTLI   ->  ENT           differentli    ->  different
     (m>0) ELI     ->  E             vileli        - >  vile
     (m>0) OUSLI   ->  OUS           analogousli    ->  analogous
     (m>0) IZATION ->  IZE           vietnamization ->  vietnamize
     (m>0) ATION   ->  ATE           predication    ->  predicate
     (m>0) ATOR    ->  ATE           operator       ->  operate
     (m>0) ALISM   ->  AL            feudalism      ->  feudal
     (m>0) IVENESS ->  IVE           decisiveness   ->  decisive
     (m>0) FULNESS ->  FUL           hopefulness    ->  hopeful
     (m>0) OUSNESS ->  OUS           callousness    ->  callous
     (m>0) ALITI   ->  AL            formaliti      ->  formal
     (m>0) IVITI   ->  IVE           sensitiviti    ->  sensitive
     (m>0) BILITI  ->  BLE           sensibiliti    ->  sensible
```

Figure 2.7. Second step in the Porter algorithm (http://www.tartarus.org/~martin/)

```
Step 3

    (m>0) ICATE ->  IC              triplicate     ->  triplic
    (m>0) ATIVE ->                  formative      ->  form
    (m>0) ALIZE ->  AL              formalize      ->  formal
    (m>0) ICITI ->  IC              electriciti    ->  electric
    (m>0) ICAL  ->  IC              electrical     ->  electric
    (m>0) FUL   ->                  hopeful        ->  hope
    (m>0) NESS  ->                  goodness       ->  good
```

Figure 2.8. Third step in the Porter algorithm (http://www.tartarus.org/~martin/)

```
Step 4

    (m>1) AL    ->                  revival        ->  reviv
    (m>1) ANCE  ->                  allowance      ->  allow
    (m>1) ENCE  ->                  inference      ->  infer
    (m>1) ER    ->                  airliner       ->  airlin
    (m>1) IC    ->                  gyroscopic     ->  gyroscop
    (m>1) ABLE  ->                  adjustable     ->  adjust
    (m>1) IBLE  ->                  defensible     ->  defens
    (m>1) ANT   ->                  irritant       ->  irrit
    (m>1) EMENT ->                  replacement    ->  replac
    (m>1) MENT  ->                  adjustment     ->  adjust
    (m>1) ENT   ->                  dependent      ->  depend
    (m>1 and (*S or *T)) ION ->     adoption       ->  adopt
    (m>1) OU    ->                  homologou      ->  homolog
    (m>1) ISM   ->                  communism      ->  commun
    (m>1) ATE   ->                  activate       ->  activ
    (m>1) ITI   ->                  angulariti     ->  angular
    (m>1) OUS   ->                  homologous     ->  homolog
    (m>1) IVE   ->                  effective      ->  effect
    (m>1) IZE   ->                  bowdlerize     ->  bowdler
```

Figure 2.9. Fourth step in the Porter algorithm (http://www.tartarus.org/~martin/)

```
Step 5a

    (m>1) E     ->                  probate         ->  probat
                                    rate            ->  rate
    (m=1 and not *o) E ->           cease           ->  ceas

Step 5b

    (m > 1 and *d and *L) -> single letter
                                    controll        ->  control
                                    roll            ->  roll
```

Figure 2.10. Fifth step in the Porter algorithm (http://www.tartarus.org/~martin/)

```
Number of words reduced in step 1:   3597
                "                2:    766
                "                3:    327
                "                4:   2424
                "                5:   1373
Number of words not reduced:         3650
```

Figure 2.11. Suffix stripping of a vocabulary of 10,000 words
(http://www.tartarus.org/~martin/)

	K0	K1	K2	K3	K4	K5	K6
D0	0	0	5	2	0	1	2
D1	4	1	1	0	0	0	0
D2	0	3	0	0	4	2	7
D3	2	2	0	7	1	0	0
D4	0	0	0	5	2	1	1
D5	7	0	2	0	0	0	3
D6	0	0	0	0	2	3	0

Figure 2.12 An abstract term-document matrix

```
(0,2,5)          (2,4,4)          (4,4,2)
(0,3,2)          (2,5,2)          (4,5,1)
(0,5,1)          (2,6,7)          (4,6,1)
(0,6,2)          (3,0,2)          (5,0,7)
(1,0,4)          (3,1,2)          (5,2,2)
(1,1,1)          (3,3,7)          (5,6,3)
(1,2,1)          (3,4,1)          (6,4,2)
(2,1,3)          (4,3,5)          (6,5,3)
```

Figure 2.13 Representation of sparse term-document matrix using triplets

```
(2,5)(3,2)(5,1)(6,2)
(0,4)(1,1)(2,1)
(1,3)(4,4)(5,2)(6,7)
(0,2)(1,2)(3,7)(4,1)
(3,5)(4,2)(5,1)(6,1)
(0,7)(2,2)(6,3)
(4,2)(5,3)
```

Figure 2.14 Representation of sparse term document matrix using pairs

	K0	K1	K2	K3	K4	K5	K6
D0	0	0	1	0.4	0	0.2	0.4
D1	1	0.25	0.25	0	0	0	0
D2	0	0.43	0	0	0.57	0.29	1
D3	0.29	0.29	0	1	0.14	0	0
D4	0	0	0	1	0.4	0.2	0.2
D5	1	0	0.29	0	0	0	0.43
D6	0	0	0	0	0.67	1	0

Figure 2.15 Normalized term document weight matrix

```
ad 1 0.125              drive 1 0.125          larg 2 0.25            recognit 1 0.125
algorithm 1 0.125       dynam 1 0.125          learn 1 0.125          refer 1 0.125
analysi 1 0.125         effici 1 0.125         li 1 0.125             relationship 1 0.125
analyz 1 0.125          emerg 1 0.125          machin 1 0.125         scale 1 0.125
archiv 1 0.125          enterpris 1 0.125      manag 1 0.125          scienc 1 0.125
artifici 1 0.125        excess 1 0.125         market 1 0.125         size 1 0.125
assimil 1 0.125         excit 1 0.125          mine 5 0.625           softwar 1 0.125
avail 1 0.125           expect 1 0.125         natur 1 0.125          span 1 0.125
billion 1 0.125         famili 1 0.125         nugget 1 0.125         statist 1 0.125
bit 1 0.125             field 2 0.25           onlin 1 0.125          store 1 0.125
challeng 1 0.125        forc 1 0.125           pattern 1 0.125        system 1 0.125
commerci 1 0.125        form 1 0.125           petabyt 1 0.125        techniqu 3 0.375
complex 1 0.125         hidden 1 0.125         potenti 1 0.125        thrust 1 0.125
comput 2 0.25           high 1 0.125           practic 1 0.125        time 1 0.125
concept 1 0.125         hoc 1 0.125            presenc 1 0.125        underpin 1 0.125
data 8 1.0              inform 1 0.125         present 1 0.125        valuabl 1 0.125
databas 3 0.375         intellig 1 0.125       quick 1 0.125          year 1 0.125
detect 1 0.125          interest 2 0.25        recogn 1 0.125
distribut 1 0.125       intersect 1 0.125
```

Figure 2.16 Vector representation of document d_1

Constructor Detail

DocVector

`public DocVector()`

Default Constructor

DocVector

```
public DocVector(java.util.Vector wordVector,
                 java.lang.String initID)
```

Creates a vector of Term using the words/tokens from the wordVector.

Parameters:

`wordVector` - A Vector of words

`initID` - A string used to identify the document

See Also:

`Term`

Method Detail

getVector

`public java.util.Vector getVector()`

Returns:

A vector of Term: a triplet consisting of stemmed words, frequency, and normalized frequency.

See Also:

`Term`

---

getID

`public java.lang.String getID()`

Returns:

A string identifying the document

---

setVector

`public void setVector(java.util.Vector initVector)`

See Also:

`Term`

---

setID

`public void setID(java.lang.String initID)`

Parameters:

`initID` - A string identifying the document

---

main

`public static void main(java.lang.String[] args)`

Parameters:

`args` - An array of Strings, e.g. d1.txt "\!\[\](),;.?" stopwords.txt

---

Figure 2.17 Methods for `DocVector` class
(Suitable for Java programmers)

```
try
{
                    ......
     String s = in.readLine();
     while(s != null)
     {
         test += s + "\n";
         s = in.readLine();
     }
}catch(IOException e){}

Vector wordVector = new TokenizedDoc
                        (test, args[1], args[2]).getTokens();

DocVector documentVector = new DocVector(wordVector, args[0]);

Vector cdv = documentVector.getVector();

System.out.println(".I "+documentVector.getID());

for(int i = 0; i < cdv.size(); i++)
   System.out.println(cdv.get(i));
```

Figure 2.18 Code snippet from the main of `DocVector` class
(Suitable for Java programmers)

Constructor Summary
Term(java.lang.String initWord)

Method Summary	
int	**getFreq**()
double	**getNormalizedFreq**()
java.lang.String	**getWord**()
void	**setFreq**(int f)
void	**setNormalizedFreq**(int n)
void	**setWord**(java.lang.String w)
java.lang.String	**toString**()

Figure 2.19 Methods for `Term` class
(Suitable for Java programmers)