Sahitya Khoz: Hindi Search Engine for Document Retrieval

Anubhav Ujjawal, Anurag Gupta and Garvit Kataria
Indian Institute of Information Technology
Sri City, Andhra Pradesh, India
{anubhav.u16,anurag.g16,garvit.k16}@iiits.in

ABSTRACT

Stemming is the process of reducing words to their word stem, base or root form. It is usually sufficient for related words map to the same stem, even if this stem is not in itself a valid root. Different languages have different rules for stemming, and stemming algorithms differ between themselves in terms of performance and accuracy. In this project, we implement a stemmer for Hindi language and integrate it with a search engine which indexes Hindi Documents. We develop different stemming rules for the stemmer and evaluate those rules.

CCS CONCEPTS

• **Information systems** → **Information retrieval**.

KEYWORDS

Hindi Search Engine, Hindi Stemmer, Information Retrieval

1 INTRODUCTION

According to Wikipedia, Modern Standard Hindi is a standardized and Sanskritised register of the Hindustani language. Hindi written in the Devanagari script is one of the official languages of India, along with the English language. Hindi is written in the Devanagari script, an abugida. Devanagari consists of 11 vowels and 33 consonants and is written from left to right. Unlike for Sanskrit, Devanagari is not entirely phonetic for Hindi, especially failing to mark schwa dropping in spoken Standard Hindi. Hindi has naturally inherited a large portion of its vocabulary from Shauraseni language , in the form of tadbhava words. This process usually involves compensatory lengthening of vowels preceding consonant clusters in Prakrit. For developing a stemmer for Hindi, we try to formulate various stemming rules for Hindi by researching about Hindi language rules and implementing hit and trial on them trying to increase the *Score*, trying to find out which works best, and then evaluate it on our own test corpus containing about 20 documents with each document containing about 60 terms each. We plug our custom stemmer with lucene and then evaluate the results for various queries. The End product is a Hindi Search Engine, which utilizes our stemmer. To take care of other problems such as normalization, tokenisation, indexing etc, we use open sourced lucene packages. Our work has implications on how to create an effective stemmer for the Hindi language.

2 WORKING OF LUCENE BASED SEARCH ENGINE

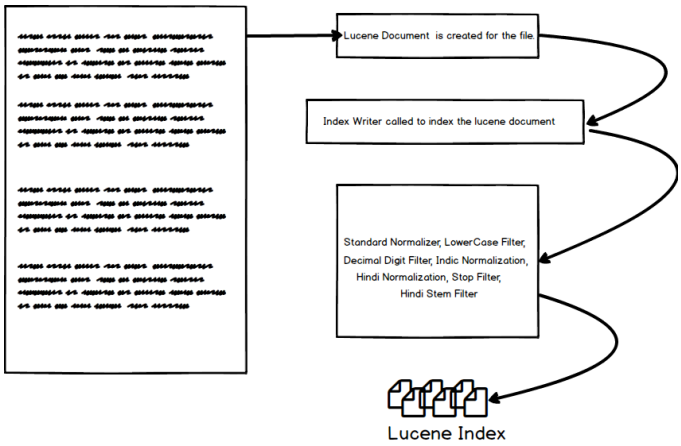

Figure 1: Lucene Index Creation.

- We create a `IndexWriterConfig` which stores the configurations to index the documents. We create a `HindiAnalyser` which uses our custom stemmer for stemming and other open sourced packages for normalization etc. We create the index in `ADD_OR_APPEND` mode as it is the most convenient one.
- We then initialize an `IndexWriter` object which takes the input corpus directory and `IndexWriterConfig` we created earlier as arguments.
- `indexDocs` function is then called which recursively iterates to all input files and directories.
- We then create a lucene `Document` for each file that we want to index. Each document contains 3 fields: path of the input file, last modified time and the actual contents of the file.
- We finally call the `IndexWriter` to further process the document and then write it into the index.
- Further processing on the documents is done by `StandardTokenizer`, `LowerCaseFilter`, `DecimalDigitFilter`, `IndicNormalization`, `HindiNormalization`, `StopFilter`, and finally `HindiStemFilter`.
- When we query the items, each query is processed in the same fashion as each document is processed above, and then the `TopDocs` are fetched. The search is done in the contents field of the `Documents` and the top hits are returned with their scores.

We discuss each of the processing steps in the below subsections.

2.1 Standard Tokenizer

Tokenisation is the task of converting documents unit into pieces, known as *tokens* while throwing away some characters

such as spaces or punctuation marks. `StandardTokenizer` implements word break rules from the **Unicode Text Segmentation** algorithm [2]. This tokenizer was good enough to meet our requirements. `StandardTokenizer` converts the content of our documents into tokens.

2.2 Lowercase Filter

`LowerCaseFilter` normalizes token text to lowercase. There is no such thing as lower case in Hindi, but it's better to use this to convert Latin text such as time, document name etc. present in the document to lowercase.

2.3 Decimal Digit Filter

`DecimalTextFilter` looks for digits outside of basic latin and replace it with basic equivalent latin digit. So, this converts Hindi digits into their corresponding Latin equivalents.

2.4 Indic Normalization

`IndicNormalization` normalizes the Unicode representation of text in Indian languages. It follows guidelines from *Unicode 5.2, chapter 6, South Asian Scripts* and *graphical decomposition from* ***Indian scripts and Unicode***[6].

2.5 Hindi Normalization

`HindiNormalization` normalizes the hindi text to remove differences in spelling variations. It implements Hindi language specific algorithm specified in **Word Normalization in Indian Languages**[4] along with certain additions from **Hindi CLIR in Thirty Days**[5]. For ex: ँ is normalized to *anusvara*, which is represented with a dot (bindu) above the letter (e.g. मं). *virama* (्) is deleted.

2.6 Stop Filter

`StopFilter` is the list of stop words containing 200+ words from the Hindi language. Words present in Stop Filter list are ignored while creating the index. The words list is taken from **IR Multilingual resources at UniNE**[3].

2.7 Hindi Stem Filter

`HindiStemFilter` is created by us, mostly by using the stemming criteria specified in *A Lightweight Stemmer for Hindi*[1].

We discuss the rules of stemming in short below.

2.7.1 When the length of reduction needs to be 5. We remove the following 5 letter suffixes from our Hindi terms:
एंगी , एंगे , ऊंगी , ऊंगा , इयाँ , इयों , इयां and return the stem. For ex: बनाएंगी stem form is बन .

2.7.2 When the length of reduction needs to be 4. We remove the following 4 letter suffixes from our Hindi terms:
एगी , एगा , ाओगी , ाओगे , एंगी , ेंगी , एंगे , ेंगे , ूंगी , ूंगा , तीं , नाओं , नाएं , ताओं , ताएं , यिाँ , यिों , यिां

2.7.3 When the length of reduction needs to be 3. We remove the following 3 letter suffixes from our Hindi terms:
ाकर , ाइए , ाईं , ाया , ेगी , ेगा , ोगी , ोगे , ाने , ाना , ाते , ाती , ाता , ाती , ाओं , ाएं , ुओं , ुएं , ुआं

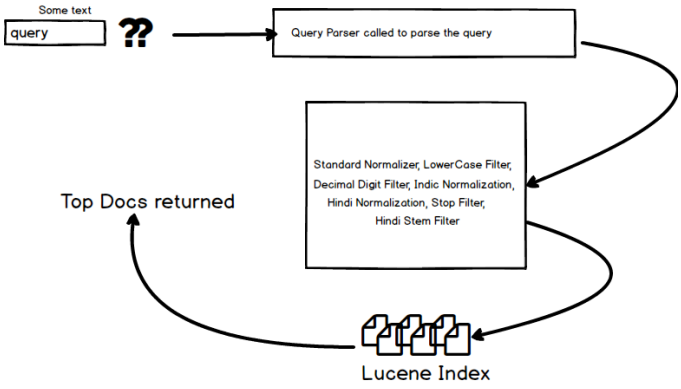

Figure 2: Lucene Query Parsing and Result Retrieval.

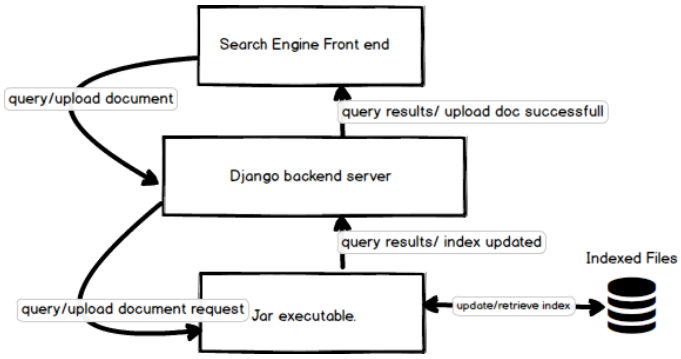

Figure 3: Server Architecture.

2.7.4 When the length of reduction needs to be 2. We remove the following 2 letter suffixes from our Hindi terms:
कर , ाओ , िए , ाई , ाए , ने , नी , ना , ते , ी , ती , ता , ाँ , ां , ों , ें

2.7.5 When the length of reduction needs to be 1. We remove the following 1 letter suffixes from our Hindi terms:
ो , े , ू , ु , ी , ि , ा

The rules of stemming are kept in `if-else`, with the rules of highest suffix being first. And only one stemming rule is applied on each term. So the rules of length 5 suffix are matched first, followed by rules of suffixes of length 4, 3, 2, 1 respectively. After Stemming, the stem form of the term is returned.

3 HOSTING THE SEARCH ENGINE

We compile our search engine into a executable `.jar` file, and use it to fetch the results of the query. The `.jar` file is run from a python script, which is called from inside of a django-based server. The response from `.jar` file is shown to the user. User can upload documents, on which, `.jar` is called to update its index.

4 TESTING AND RESULTS

For Testing, we used different single word and multi-word queries and checked the scores of the output with and without

Query	With Hindi Stemmer	Without Stemmer
प्यास के कुआ	2.3951	0
आधे गाने	5.2542	5.7001
काला	4.032	0

Table 1: Scores with and without Hindi Stemmer

Hindi Stemmer. Some of the outputs are shown in Table 1. We can see that using a stemmer vastly affects performance of the search engine. However, The stemmer can't relate words which have different forms because of grammatical constructs, such as words which have totally different constructs in comparative or superlative forms. Stemmers however, have their own advantages, such as drastic improve in size of index and high speed of stemming process.

5 CONCLUSION

Thus, we have successfully implemented and integrated a Hindi stemmer with a Lucene based Search Engine, and are able to retrieve results on the basis of their scores, as well as are able to add documents to our index. We found that stemming depends very much on the language's grammatical as well as its script. We also found out that Hindi has a lot more stop words than the English language. We also found out how to convert ISCII to unicode. [6].

ACKNOWLEDGMENTS

The authors would like to thank Dr. Venkatesh Vinayakarao for providing immense support and encouragement to create Sahitya Khoz, Lucene Based Search Engine for Hindi Language.

REFERENCES

[1] Ananthakrishnan Ramanathan, Durgesh D Rao *A Lightweight Stemmer for Hindi.*
[2] Unicode Text Segmentation, `http://unicode.org/reports/tr29/`
[3] IR MultiLingual resources at UniNe `http://members.unine.ch/jacques.savoy/clef/index.html`
[4] Prasad Pingali, Vasudeva Varma *Word Normalizatiion in Indian Languages*
[5] LEAH S. LARKEY, MARGARET E. CONNELL, AND NASREEN ABDULJALEEL *Hindi CLIR in Thirty Days*
[6] Indian scripts and Unicode `http://ldc.upenn.edu/myl/IndianScriptsUnicode.html`