Building Unigram Postings and Positional Postings in Python

William Scott
Jan 27, 2019 · 3 min read

Let's build Unigram and Positional Postings in Python from scratch on real world dataset.

> *This is the second post on Information Retrieval series.*

Click here to checkout the git repo

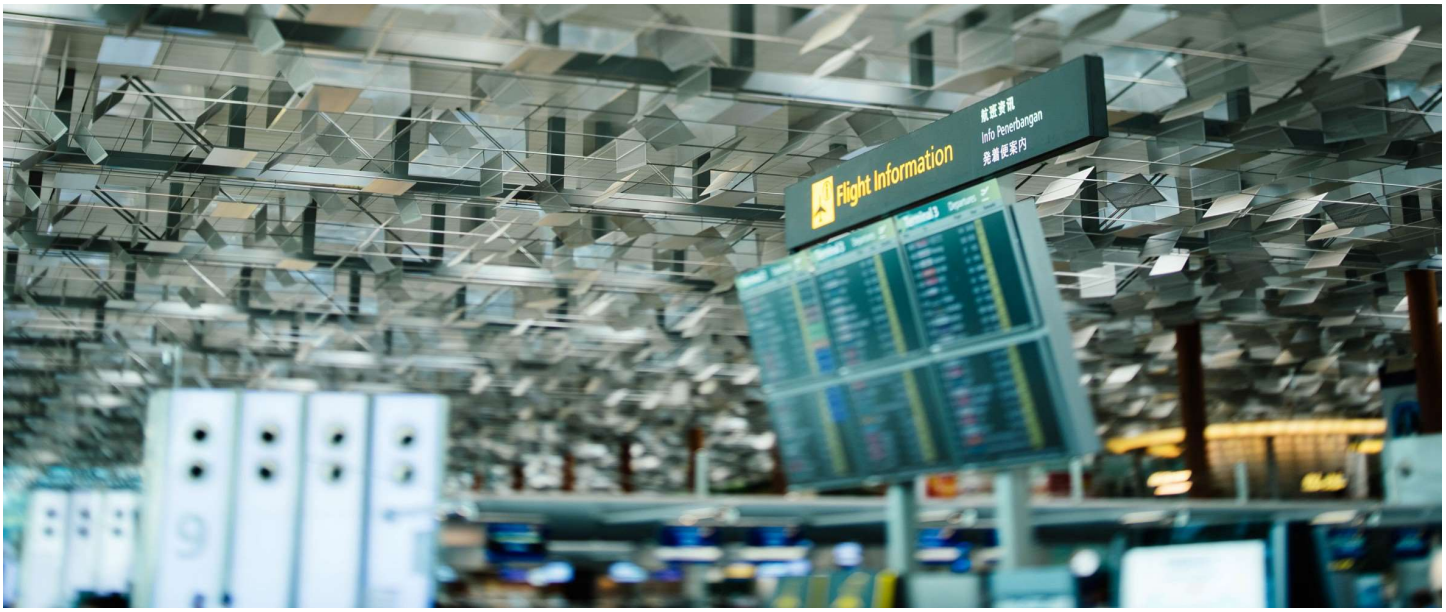
Information Retrieval Series:

- 1. Introduction
- 2. Unigram Indexing & Positional Indexing
- 3. TF-IDF
- More to come…

. . .

Table of Contents:

- Generate Unigram Inverted Index and run complex Boolean queries
- Generate Positional Indexes and search for phrases.



Introduction

The above two methods are basic document (text files) retrieval techniques in which we try to extract the documents by giving some kind of query.

To extract the data appropriately, first we need index the data.

Term Document Incidence Matrix:

In this method, we collect all the unique words and form a matrix of Unique Words & Documents. Mark 1 in the cell in which a word exists in that corresponding document.

Problem:

- Extremely sparse matrix
- Very large size matrix when stored to disk

Tools used:

- NLTK
- Pandas
- NumPy
- Pickle

Preprocessing Used:

- Removing stop words

- Removing punctuation
- Convert to lowercase
- Stemming
- Converting numbers to its equivalent words
- Removing header

Unigram Inverted Index:

To overcome the problems of Term Documenting, we use a technique called unigram inverted index.

in this technique we store all the list of documents that contain a word. The collection of these lists for every word is called postings list.

The words that we use here are called dictionary. And each of the document list for a word is called postings.

Steps:

- Use a document ID for all the documents (usually the index during iteration)
- Preprocess the text
- Generate tokens
- For every token in the document, add a documentID.
- Repeat this for every document.
- If the encountered token is a new token which is not in the dictionary, add the token to the dictionary.
- The document IDs should be sorted for easy extraction (but depends on the implementation)
- Also keep a track of the number of documents that each word consists (will be used for smart Boolean query extraction)
- Handle the wrong/nonexistent tokens

Complex Query Extraction:

- Separate the commands and the tokens
- Preprocess the tokens
- First implement the not command on the respective words, and replace the tokens list with the postings list of that particular word
- Then from left to right apply the “and” “or” commands
- Return the final merges

Positional Indexing:

Positional Indexing is almost similar to Unigram Inverted indexing, in positional indexing we additionally keep the position of all occurrences of token related to the documentID’s

This helps to better search for queries without the Boolean searching.

One important difference is that in this we will not remove the stop words.

Steps:

- Use a document ID for each document (usually the indexed during iteration)
- Preprocess the text
- Generate tokens
- For every token in the document, add a (documentID, positionID) pair.
- Repeat this for every document.
- If the encountered token is a new token which is not in the dictionary, add the token to the dictionary.
- The document IDs should be sorted for easy extraction (but depends on the implementation)
- Also keep a track of the number of documents that each word consists (will be used for smart Boolean query extraction)
- Handle the wrong/nonexistent tokens

Query Extraction:

- Take the query string
- Preprocess the whole query
- Convert to tokens
- Take the (document, position) pair for the first token
- for each pair:
- for all the next tokens
- match if the next token exists in (document at position++)
- if the next token doesn't exist, just break the loop

Observations:

- Unigram inverted index is faster to build but is not much useful due to the requirement of Boolean queries
- Positional Indexing is helpful to give phrase searches, but is little time consuming to build.
- Memory: Unigram < Positional
- Build Speed: Unigram > Positional
- Query Speed: Unigram > Positional

Assumptions:

- Header doesn't contain anything important
- And footer is not needed to be removed

Statistics:

	Unigram	Positional
Removed Stop Words	Yes	No
Documents Used	20,000	1000
Dictionary Size	90027	11388
Generation of Postings	32Min 42Sec	1min 16sec
Postings Retrieval	3.11ms	9.66ms
Query Run Time	30.2ms	111ms

Click here to checkout the git repo

Information Retrieval Series:

- 1. Introduction
- 2. Unigram Indexing & Positional Indexing
- 3. TF-IDF
- More to come…

Machine Learning Information Retrieval NLP Nltk