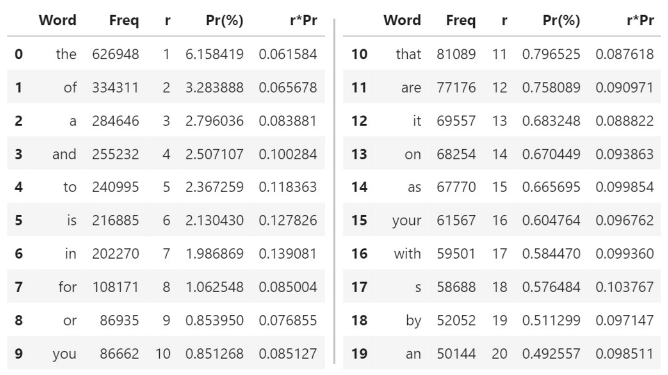
Information Retrieval and Data Mining COMP0084 Project Part 1

20031465
University College London
London, UK

1 TEXT STATISTICS

To find the word counts the text is first lowercased, then words are obtained by considering sequences of letters. Figure 1 depicts the 20 most common words with the corresponding statistics.



	Word	Freq	r	Pr(%)	r*Pr
0	the	626948	1	6.158419	0.061584
1	of	334311	2	3.283888	0.065678
2	a	284646	3	2.796036	0.083881
3	and	255232	4	2.507107	0.100284
4	to	240995	5	2.367259	0.118363
5	is	216885	6	2.130430	0.127826
6	in	202270	7	1.986869	0.139081
7	for	108171	8	1.062548	0.085004
8	or	86935	9	0.853950	0.076855
9	you	86662	10	0.851268	0.085127
10	that	81089	11	0.796525	0.087618
11	are	77176	12	0.758089	0.090971
12	it	69557	13	0.683248	0.088822
13	on	68254	14	0.670449	0.093863
14	as	67770	15	0.665695	0.099854
15	your	61567	16	0.604764	0.096762
16	with	59501	17	0.584470	0.099360
17	s	58688	18	0.576484	0.103767
18	by	52052	19	0.511299	0.097147
19	an	50144	20	0.492557	0.098511

Figure 1: 20 most common words.

The Zipf's law states:

$$r_i * f_i = k$$

Alternatively we can write:

$$r_i * Pr_i = \frac{k}{n} = c$$

Where r is the word rank in ascending order, f is a word frequency, k is some constant, Pr is a probability of word occurrence, and n is the number of words in the corpus. More generally, Zipf's law is a power law with $a = 1$:

$$Pr = c * r^{-a}$$

To check if the distribution follows a Zipf's law we will estimate the coefficients of a power law using maximum likelihood:

$$p(words|\theta) = \prod_i Pr_i^{f_i}$$

$$\log(p(words|\theta)) = \sum_i f_i \log(Pr_i)$$

$$Pr_i \propto r^{-a}$$

Then we can estimate parameter c as follows:

$$\sum_i Pr_i = 1 \wedge Pr_i = cr_i^{-a}$$

$$\sum_i Pr = \sum_i cr^{-a} = c\sum_i r^{-a} = 1$$

$$c = \frac{1}{\sum_i r_i^{-a}}$$

The negative log-likelihood was minimized using minize_scalar from sicpy.optimize [3]. Furthermore, we drop the words that occurred 3 or fewer times in the corpus under the assumption that these are misspellings or errors that would skew the distribution. The resulting maximum likelihood estimates:

$$a = 1.0156$$

$$c = 0.0953$$

Parameter a is close to 1. Thus, it does not disprove the hypothesis that the distribution follows a Zipf's law. We can also plot the data on a log-log scale to obtain qualitative insight, see Figure 2. Ideally the distribution would follow a line with slope -1:

$$\log(Pr_i) = \log(c * r_i^{-1}) = \log(c) - \log(r_i)$$

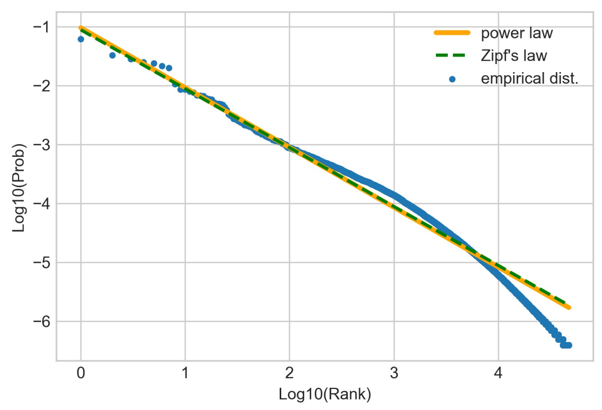


Figure 2: Zipf's law parameter estimation.

The Zip's law line in Figure 2 was obtained setting $a = 1$ and estimating parameter c as before to get:

$$c = 0.0881$$

In Figure 2, we can see that the empirical distribution approximately follows Zipf's law. The worst fit occurs in the tails of the distribution, especially for less common words. Additionally, Figure 3 shows the distribution of top 60 words against the estimated Zipf's law.

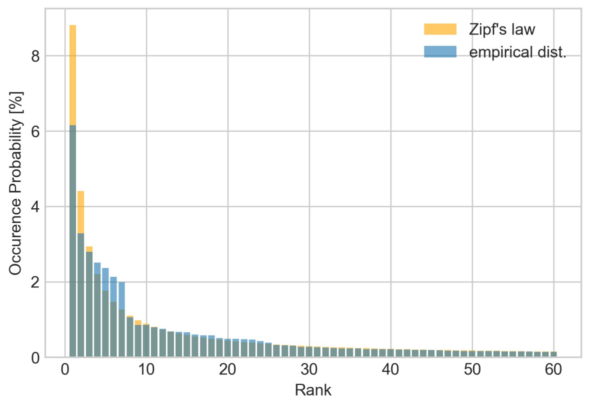


Figure 3: Word's distribution.

2 INVERTED INDEX

Text pre-processing:

(1) lower case,
(2) tokenize - get sequences of letters and numbers (findall with regex pattern $r"[\backslash w]+")$),
(3) remove stop words - stop words obtained from nltk.corpus.stopwords [2],
(4) stem the tokens - using nltk.stem.PorterStemmer [1].

We need to lower case the text to make the search invariant of letter captioning. The tokens include both letters and numbers to be able to find the number or date-specific information (e.g. Porche 911 vs Porche 930s). We also remove stop words such as "and", "a", "the" because they do not provide much information and would slow down the ranking. Finally, we stem the tokens to treat words like "sample", "samples", "sampling" as the same token. It makes the ranking more robust and further reduces the vocabulary size.

The inverted index stores the following information for each token:

- list of passage ids where the token occurred in,
- number of passages the token occurred in,
- count of occurrence in the collection.

The passage ids are used as pointers to the passage specific information such as:

- number of counts for each token - sparse bag of words representation using Python dictionary data structure,
- passage length.

The inverted index also stores global information:

- vocabulary size,
- number of passages in the collection,
- total number of tokens in the collection,
- average passage length.

The above information is necessary for the ranking algorithms presented in this paper. Pre-computing this statistics reduces ranking time.

3 RETRIEVAL MODELS

3.1 Vector Space Model

The vector space model represents queries and passages as tf-idf vectors, and later compares them using cosine similarity.

$$\text{score}(D, Q) = \frac{\mathbf{d} \cdot \mathbf{q}}{||\mathbf{d}||_2 ||\mathbf{q}||_2}$$

Where $\mathbf{d}$,$\mathbf{q}$ are the tf-idf vectors corresponding to document D and query Q. The tf weighting is responsible for capturing the common tokens between the query and the passage. However, tokens are not equally important. Therefore, the idf term captures token's importance based on the number of passages it appeared in. The fewer passages it occurred in the more relevant it is. The employed tf-idf scheme [4]:

$$\text{(log) tf} = 1 + \log(f_t)$$
$$\text{(smooth) idf} = 1 + \log\left(\frac{N}{n_t}\right)$$
$$\text{tf-idf} = \text{tf} * \text{idf}$$

Where f_t is the token count, N is the number of passages in the collection, and n_t is the number of passages including that token. The resulting weight vectors are then normalized to compute cosine similarity. Using logarithms dampens the values. Furthermore, adding one to the logarithms prevents the weights to be zero when the term inside the logarithm is one. It is the main reason to use smooth idf as in our application that is sometimes the case (e.g. qid = 1103528). That way, we can still rank based on tf and prevent having zero scores for all passages.

The tf-idf vectors are represented as sparse vectors using the Python dictionary data structure, where tokens are the keys and tf-idf weights are the values. We can then perform vector operations considering only non-zero elements - keys.
Dot product:

$$\mathbf{a} \cdot \mathbf{b} = \sum_{i \in a_{keys} \wedge b_{keys}} a_i b_i$$

Norm:

$$||\mathbf{a}||_2 = \sqrt{\sum_{i \in a_{keys}} a_i^2}$$

This representation allows for efficient storing and fast operations as the vectors are very sparse. The model obtains all necessary statistics for document tf-idf calculations from the inverted index. It only pre-processes the raw query text and drops tokens not present in the vocabulary. Then, the vector space model calculates the query tf-idf normalized vector. It is reused for the similarity computation with each of the relevant documents (at least one query term appeared) obtained from the inverted index.

Advantages and limitations of Vector Space Model:

- simple and fast,
- continuous degree of similarity between queries and documents,
- allows partial matching,
- assumes that terms are independent - looses structure,
- poor representation of large documents.

3.2 BM25

The BM25 is based on the binary independence model. It follows a probabilistic retrieval framework. Moreover, BM25 allows to incorporate document relevance to the scoring function. Here the relevance is set to zero and ommited in the equations as we do not have this additional information. The implementation follows the lecture:

$$\text{score}(D, Q) = \sum_{t \in Q} \log\Big(\frac{N - n_t + 0.5}{n_t - 0.5}\Big) * \frac{(k_1 + 1) f_t}{k_1(1 - b + b * \frac{dl}{avgdl}) + f_t} * \frac{(k_2 + 1) qf_t}{k_2 + qf_t}$$

Where f_t is the number of times query term t occurs in document D, and qf_t is the count of term t in query Q. The BM25 model obtains all relevant document statistics from the inverted index. It is worth to mention that the first and last terms in the summation depend only on the query. Therefore, we can pre-compute and reuse them to score each document. The constants were chosen as suggested in the lecture slides:

$$k_1 = 1.2$$
$$k_2 = 100$$
$$b = 0.75$$

The BM25 became popular due to its efficiency. It is considered state-of-the-art tf-idf retrieval model. It accounts for document length. Hence, performs much better than the Vector Space Model on longer documents. However, it employs numerous heuristics making it less general.

4 LANGUAGE MODELLING RETRIEVAL MODELS

Language modelling is based on probabilistic distributions and has strong theoretical foundations. Thus, it is easy to extend this framework. The basic query-likelihood model:

$$p(Q|M_D) = \prod_{t \in Q} p(t|M_D) = \prod_{t \in Q} \frac{f_t}{|D|}$$

The problem with the above is that the probability would be zero if not all query terms are present in the passage. It also only considers the current passage and does not incorporate information about the collection. Another downfall is that it can not benefit from passage relevance information. However, we can fix some of the issues with smoothing. Due to numerical stability reasons the implementations of the algorithms consider log-probability:

$$\log p(Q|M_D) = \sum_{t \in Q} \log p(t|M_D) = \sum_{t \in Q} \log \frac{f_t}{|D|}$$

As previously, all the necessary statistics are obtained from the inverted index described in section 2.

4.1 Laplace Smoothing

$$p(t|M_D) = \frac{f_t + 1}{|D| + |V|}$$

Adding one to f_q in the nominator prevents from $p(Q|M_D) = 0$. We normalize this by adding $|V|$ - vocabulary size. That results in having a uniform prior over words. However, Laplace smoothing assigns too much weight to the unseen terms. To counter that, we introduce discounting.

4.2 Lidstone Smoothing

$$p(t|M_D) = \frac{f_t + \epsilon}{|D| + \epsilon|V|}$$

Lidstone smoothing discounts by $\epsilon \in [0, 1]$, which allows decreasing the weight of unseen terms. The disadvantage of Laplace and Lidstone smoothing is that they treat the terms equally.

4.3 Dirichlet Smoothing

$$p(t|M_D) = \frac{|D|}{|D| + \mu} p(t|D) + \frac{\mu}{|D| + \mu} p(t|C) = \frac{|D|}{|D| + \mu} \frac{f_t}{|D|} + \frac{\mu}{|D| + \mu} \frac{cf_t}{|C|}$$

Where $|C|$ is the number of tokens in the collection, cf_t is the count of the term in the collection, μ is a constant. Dirichlet smoothing accounts for token importance. Therefore, it is expected to perform better than Laplace or Lidstone smoothing.

REFERENCES

[1] [n.d.]. NLTK stemmer. https://www.nltk.org/api/nltk.stem.html.
[2] [n.d.]. NLTK stop words. https://www.nltk.org/book/ch02.html.
[3] [n.d.]. Scipy minimize. https://docs.scipy.org/doc/scipy/reference/generated/scipy.optimize.minimize_scalar.html.
[4] [n.d.]. tf-idf. https://en.wikipedia.org/wiki/Tf-idf.