Leipzig University
Faculty of Mathematics and Computer Science
Institute of Computer Science

Evaluating Retrieval Methods to Find Related Articles on Wikipedia

Bachelor's Thesis

Leipzig, March 14, 2022

Submitted by:

Justus Stahlhut
Computer Science, B.Sc.

Supervised by:
Jun.-Prof. Dr. Martin Potthast
Wolfgang Kircheis

Acknowledgement

First and foremost, I want to thank my supervisor: Wolfgang Kircheis, for accompanying and helping me throughout the creation process of this work. I also want to thank my brother Max, and my best friend Conrad, for being supportive and having an open ear for ideas while helping me keep my eyes fixed on the goal.

And now that I've done it:
So what?

Declaration

I certify that I have prepared this thesis independently and only using the sources indicated; in particular, verbatim or analogous quotations are marked as such. I am aware that failure to comply with this requirement may result in the subsequent withdrawal of my degree.
I assure the electronic copy corresponds to the printed copies.

Leipzig, March 14, 2022

.....
Justus Stahlhut

Abstract

Article retrieval is a topic widely studied in computer science. Many unique approaches have been developed to retrieve articles for a given query. This work describes, implements, evaluates, and compares three different approaches to retrieve related articles on Wikipedia. The *keyquery* approach incorporates a search engine to find a query that will retrieve a set of documents. The *graph* approach creates a graph from Wikipedias' structure and finds similar articles based on its topology. The *text similarity* approach incorporates several measures comparing Wikipedias' articles by their bodies.

Table of Contents

1 Introduction

Wikipedia has grown to be one of the most widespread encyclopedias on the internet. With more than 6.4 million articles (regarding the English Wikipedia only), its knowledge spans many topics.
Because of this fact, Wikipedia is widely studied in computer science. Many algorithms have been created to cluster/analyze and- or compare Wikipedias articles.
Averagely 586 articles are added daily to the encyclopedia [1]. With this amount of (new) data, an inherent problem arises regarding the importance of its articles information.
The problem might be as simple as reading up on a topic for a school report. An excellent place to start researching is the article about the topic itself. However, where to go from there?
Depending on the information in the article, one can semi-randomly read through linked articles and hope to find all the relevant information. Still, relevant articles may not be found if pertinent information is buried deeper into Wikipedias link-tree. This problem is the core of this thesis and can be stated as follows:

Given one or more articles about a specific topic, find the most relevant articles related to this topic.

1.1 Motivation

Wikipedia offers a wide variety of topics. From history to physics - from mostly resolute- to highly innovative fields of research. This thesis' focus primarily lies on the latter.
Innovation in this sense is defined as: *"all material or symbolic artifacts, which are observed to be novel and an improvement relative to what is currently existing."* (Braun-Thürmann) [2]
CRISPR is an excellent example of innovation. It is a highly inventive topic, with lots of new research being published every year.
An innovation researcher might now want to read up on CRISPR. Instead of opening the Wikipedia article and browsing through links, they could be presented with a list of related articles. This list of articles would inform them about the fundamental concept of CRISPR with a certain percentage of accuracy.

1.1.1 Tracing Innovations on Wikipedia

Tracing Innovations on Wikipedia is a project led by the Webis Group in Weimar. With Wikipedia at the center of this project, the goal is to find out if Wikipedia is a valid medium to analyze an innovation's dynamics and profile/character.
This thesis concerns itself with the retrieval part of this project: *finding*, *implementing*, and *evaluating* ways to retrieve related articles on Wikipedia. The information collected on the success of these approaches can help determine which approach will be used to form a ground truth of articles that will be analyzed.

1.1.2 Retrieving Relevant Articles

Researching a given topic can be a demanding task. Starting at the topics main-article is an obvious choice. From there on out, most people would probably take advantage of Wikipedias link-tree and try gathering all the valuable information they find.
More avid users possibly consider the *See also* and *Category* section 'hidden' at the bottom of the page.
Besides that, there is no information available on which articles to read first- or even at all. Much time can be spent on finding the correct articles instead of collecting information.

1.2 Innovative Articles and Evaluation

When trying to retrieve related articles for innovative topics, it is sensible to evaluate these retrieval approaches on a set of innovative articles on Wikipedia. A list of twenty articles: L, spreading over a variety of fields-of-study [3] have been selected:

$L = \{$*African-American history, Animal language, Bitcoin, Brexit, Central solar heating, Climate change, Computational fluid dynamics, CRISPR, Garbage patch, Holocene extinction, Identity formation, Large Hadron Collider, Ocean acidification, Prosthesis, P versus NP problem, Social disorganization theory, Spread of Islam, Targeted therapy, Trap music, Universal basic income*$\}$

To achieve the best results for the keyquery approach: a ground truth of four articles, including the article itself, has been selected for every article $a \in L$.

Articles have been retrieved and evaluated for each of these topics in L. For each respective approach, the top-20 results have been extracted and labeled '1', '2' or '3' regarding the article a they have been retrieved from. '1', meaning that the retrieved article is semantically dissimilar- and '3' that the retrieved article is insightful towards a. Overall, 3058 unique article pairs have been retrieved.

When discussing the results of different evaluation measures, a question arose. The evaluation measures implemented score based on a binary choice: whether a document is to be regarded as relevant or not. Using three scores to evaluate: a lower bound regarding the accepted relevancy of a document must be set. Therefore, the question is: *Should documents labeled '2' be regarded as relevant?* In the context of this work, two answers to this question, therefore two tables will be shown.
Regarding the evaluation, it is also important to note that no specific metric was in place to determine which pair of articles should receive which score. Specifically labeling researchers (actors) who have contributed to a field-of-study was challenging. One 'metric' did eventually emerge regarding the evaluation of actors: the article-pair should receive a higher score (≥ 2) if the actor is mentioned in the body/references of the article.

2 Background

Document retrieval is a problem inherent to the age of internet technology. With the internet containing an inconceivable amount of structured- and unstructured data, finding relevant documents, articles or webpages is an ever-important task.
The first search engines came alive in the early 1990s, with *Yahoo! Search* being the first popular one [4].
In 1996 the first algorithm using hyperlinks to assert scores was introduced and patented by Robin Li [5]. This approach was superseded by Page et al. [6] two years later, creating the now most used search engine: Google.
Document retrieval today takes many forms, using many different approaches to the problem. The following describes an overview of the implemented approaches.

2.1 Keyquery Retrieval

Keyqueries have been introduced by Gollub et al. [7], focusing on the retrieval of a singular document in the top-k ranks. They have been proposed as document classifiers, returning the same set of documents when sending the same query repeatedly. This is built upon in [8]: using keyqueries to compress a set of documents; and [9]: generating keyqueries as labels for document classes.
The approach implemented in this work focuses on keyquery retrieval: using a set of documents to generate keyqueries that find these documents in the top-k ranks. Hagen et al. [10], establish the implemented approach, mainly using two algorithms to find related scholarly articles.

2.2 Graph Retrieval

Graph retrieval uses the graph's topology to create a pairwise similarity score. The internet, for example, can be regarded as a graph with every website being a vertex and every link being an edge. Page et al. [6] introduced PageRank. Using PageRank to score documents based on queries, Google became one of the most visited websites.
Ollivier et al. [11] introduce a method to retrieve articles building upon the use of PageRank. The paper also cites Kemeny et al. [12], using *Green functions* to calculate electric potential in a system. Using this method, calculating scores centered at a vertex u, can be thought of as creating a charge in an analog electrical system and observing the charge distribution.

2.3 Text Similarity Retrieval

Text similarity measures compare documents based on their textual information. TF-IDF, credited to the joint work of Hans Peter Luhn and Karen Spärck Jones [13, 14], is a well known method of scoring terms against their corpus with many retrieval methods implementing it for its success.
Mikolov et al. introduce Word2Vec [15], a model to encode words and phrases into a vector space so that similar words have close vectors. Based on Word2Vec, they introduced Doc2Vec [16], a model to encode texts into vectors.
Universal Sentence Encoder (USE) [17], for example, uses Word2Vec to embed words into vectors to train its encoding models [18, 19].

3 Handling the Wikipedia Corpus

Due to its size, it is inefficient to directly work with Wikipedias' website. Several APIs have been implemented to send requests to Wikipedia and retrieve articles. Still, too many requests would be needed for the implemented approaches to work.
To work with the entire Wikipedia corpus, Wikimedia, Wikipedia's parent company, provides XML dumps [20]. These XML dumps represent a monthly created snapshot of Wikipedia, with it being an approximately 19GB large compressed XML file when downloaded. It contains every article, link, redirect[1], etc., found on Wikipedia, including the metadata of every article.
When looking at the contents of the XML file, two things can be gathered:

(1) The article bodies grossly differ from Wikipedia's actual bodies and need to be formatted

(2) The formatting of the bodies is inconsistent

[1] A redirect page is an almost empty page, only displaying one link: the page it redirects to. The link 'Anarchist' found on a page might, for example, be redirected to 'Anarchism'. This can be done to preserve sentence structure.

3.1 Formatting Articles

Naturally, the contents of the XML dump are formatted (1) to represent the actual article on the Wikipedia website. For example, the introduction of the article *'Anarchism'*:

```
'''Anarchism''' is a [[political philosophy]] and [[Political movement|movement]] that is skeptical of [[authority]] and rejects all involuntary, coercive forms of [[hierarchy]]. Anarchism calls for the abolition of the [[State (polity)|state]] which it holds to be undesirable, unnecessary and harmful.
```

Figure 3.1: Excerpt from Wikimedia Dump

The single quotes alter the boldness of the word while the words/phrases in square brackets are links to other articles, with the words appearing after '|' being shown on Wikipedia. Those are easily formatted. But looking at a citation that might occur in the text,

```
{{sfnm|1a1=Merriam-Webster|1y=2019|...|3a1=Sylvan|3y=2007|3p=260}}
```

Figure 3.2: Citation found in Wikimedia Dump

it is harder to extract useful information, especially because of inconsistencies (2) between those citations.
Overall, inconsistencies have been an issue while formatting the dump, for example, some brackets are opened and never closed, making perfect, automated formatting not possible.

After formatting the dump, every article can be split into its elements: *id*, *title*, *links*, *categories*, *sources* (citations), and *urls*.
The articles are stored in JSON files with their respective *id* as a key.

4 Retrieving Articles with Keyqueries

Keyqueries are a concept introduced by Gollub et al. [7]. Given a search engine, they are defined as minimal queries that return a document in the top-k ranks.
Adjusting the definition slightly to fit the problem: a keyquery for a set of documents is defined as a query, consisting of a set of *keyphrases*, that retrieves these documents in the top-k results. Keyphrases here are defined as words or phrases extracted from a document that retrieve that document in the top-k results.
The concept of keyquery retrieval lies in the idea that if the queried documents are retrieved in the top-k results, the other documents retrieved in this range might be related to the initial documents.

4.1 Elasticsearch as a Search Engine

Elasticsearch [21] is a search engine based on the Lucene library. It allows its users to perform requests on the indexed data. Elasticsearch uses the JSON format to index data: every article can be mapped to an Elasticsearch-field, which then can be queried.

4.1.1 Ranking Documents

The Elasticsearch library has several similarity measures already implemented. In this work, three of Elasticsearch's prewritten measures have been tested.

BM25 [22] is the similyrity measure Elasticsearch uses by default. It retrieves scores for documents based on a terms *frequency* and its *inverse document frequency*.

$$BM25(q,d) = \sum_{i=1}^{n} IDF(q_i) \cdot \frac{freq(q_i,d) \cdot (k_1+1)}{freq(q_i,d) + k_1 \cdot (1 - b + b \cdot \frac{|d|}{avgdl})}, \tag{4.1}$$

where d is the document that is to be scored regarding the query q consisting of keywords $\{q_1, ..., q_n\}$. $freq(q_i, d)$ is the number of occurrences of q_i in d. $|d|$ is the length of document d and $avgdl$ the average document length in the corpus. k_1 and b are parameters chosen for optimization: here $k_1 = 1.2$ and $b = 0.75$ by default.

$IDF(q_i)$ is the inverse document frequency.

$$IDF(q_i) = ln\Big(\frac{|\mathcal{C}| - n(\mathcal{C}, q_i) + 0.5}{n(\mathcal{C}, q_i) + 0.5} + 1\Big), \tag{4.2}$$

where $|\mathcal{C}|$ is the total amount of documents in the corpus $\mathcal{C}$ and $n(\mathcal{C}, q_i)$ is the number of documents in $\mathcal{C}$ containing q_i.

DFI (Divergence from independence) [23] measures the dependence/independence of a term from a document. Dependence means that the term contributes more information to a document. It does so by calculating the expected word frequency under independence and measuring the divergence from that value.

$$DFI(q, d) = \sum_{i=1}^{n} DFI(q_i, d), \tag{4.3}$$

$$DFI(q_i, d) = \frac{freq(q_i, d) - e(q_i, d)}{e(q_i, d)}, \tag{4.4}$$

$e(q_i, d)$ being the expected frequency of q_i in d:

$$e(q_i, d) = \frac{n(\mathcal{C}, q_i) \cdot |d|}{|\mathcal{C}_w|} \tag{4.5}$$

where $|\mathcal{C}_w|$ is the vocabulary size of $\mathcal{C}$

LMDirichlet [24] is a language modeling approach using Dirichlet smoothing (LMDirichlet). It is assumed that a query q is generated by a probabilistic model from a document d and $p(q|d)$ estimates the probability that q is generated by d. The smoothed model can be stated as:

$$p(q|d) = \prod_{i=1}^{n} p(q_i|d), \text{ with} \tag{4.6}$$

$$p(q_i|d) = \begin{cases} p_\mu(q_i|d) & \text{, if } d \text{ contains } q_i \\ \alpha_d \; p(q_i|\mathcal{C}) & \text{, otherwise.} \end{cases} \tag{4.7}$$

$p(q_i|\mathcal{C})$ being the portion of documents in $\mathcal{C}$ containing q_i and

$$p_\mu(q_i|d) = \frac{freq(q_i, d) + \mu p(w|\mathcal{C})}{|d| + \mu}, \tag{4.8}$$

$$\alpha_d = \frac{\mu}{|d| + \mu} \tag{4.9}$$

$\mu = 2000$ by default.

Articles can be retrieved using these similarity measures by passing the article's title to an Elasticsearch query. This approach will be called the *Keyphrase* approach.

4.2 Finding Keyqueries

Keyqueries $q = \{q_1, ..., q_n\}$ will be generated by concatenating keyphrases extracted from the set of documents selected as the ground truth for every $a \in L$. There are several ways to find keyphrases for a document. In this work, three keyphrase-extractors have been tested.

4.2.1 Keyphrase Extraction

RAKE (Rapid Automatic Keyword Extraction) [25] applies scores to candidate keyphrases by calculating the degree and frequency of their occurrence. Given a document d, a list of stopwords and delimiters: d can be split into candidate keyphrases $Q = \{q_1, ...q_n\}$. Every $q_i \in Q$ ($|q_i| \geq 1$) then gets scored based on their *frequency* and *degree* in d.
The degree of q_i is the total number of occurrences of q_i alone and combined with other words $w = w_1...w_n$ so that $q_i w$ or $w q_i$ is not interrupted by a delimiter. The score for q_i is then calculated as:

$$score_R(q_i) = \sum_{w \in q_i} deg(w)/freq(w) \tag{4.10}$$

YAKE! (Yet Another Keyword Extractor!) [26] combines statistical- and contextual information to calculate a score for keywords.
First, YAKE! starts preprocessing the text, and word tokenization is applied. Then, a set of five features is calculated for each term. The features considered are *Casing* (W_{Case}), *Word Position* (W_{Pos}), *Word Frequency* (W_{Freq}), *Word Relatedness to Context* (W_{Rel}), and *Word DifSentence* ($W_{DifSentence}$).

W_{Rel} scores based on the words surrounding the phrase to be scored, and $Word_{DifSentence}$ quantifies the amount a word appears in different sentences. After these features have been extracted, each candidate word w can be scored by combining these heuristics.

$$score_Y(w, d) = \frac{w_{Rel} \cdot w_{Position}}{w_{Case} + \frac{w_{Freq}}{w_{Rel}} + \frac{w_{DifSentence}}{w_{Rel}}} \tag{4.11}$$

Lastly, candidate keyphrases q can be generated. Given a sliding window of size k, candidates of size $1, 2, ..., k$ can be scored by:

$$S(q) = \frac{\prod_{w \in q} S(w)}{freq(q,d) * (1 + \sum_{w \in q} S(w))} \tag{4.12}$$

KP-Miner [27] extracts keyphrases from documents in three steps: *Candidate Keyphrase Selection*, *Candidate Keyphrase Weighting*, and *Keyphrase Refinement*. First, the text is split by punctuation and a list of stopwords, generating a list of phrases. To filter out candidate keywords: a phrase must have appeared at least n times in the document. By default, $n = 3$.
A cut-off constant limits the field in which keyphrases are to be searched for. In the second step, the candidate keyphrases are weighted using TF-IDF. Additionally, a boosting factor B_d is applied when scoring candidates.

$$B_d = \frac{|N_d|}{|P_d| \cdot \propto}, \tag{4.13}$$
$$if\ B_d > \sigma\ then\ B_d = \sigma,$$

$|N_d|$ is the number of candidate terms in document d, $|P_d|$ is the number of candidate terms whose length exceeds one and $\propto$ and σ are weight adjustment constants. By default $\propto = 2.3$ and $\sigma = 3$.
Finally the weight for candidate keyphrases q can be calculated:

$$score_{KP}(q,d) = freq(q,d) \cdot idf(q,d) \cdot B_d \tag{4.14}$$

RAKE has been implemented using the *rake-nltk* library [28]. YAKE! and KP-Miner have both been implemented using the *python keyphrase extraction toolkit* (PKE) [29].

4.2.2 Generating Keyqueries

Given the keyphrase extraction- and Elasticsearch's similarity algorithms, keyqueries for the ground truth $\mathcal{I}$ can be generated. Hagen et al. [10] introduce the retrieval approach using mainly two algorithms: Related Work Search and Keyquery Cover, a subroutine of the first algorithm. Additionally to $\mathcal{I}$, the keyquery approach has three parameters: k, l, and r, which will be explained in context.

Algorithm 4.1: Solving Keyquery Cover

Input : Set D of documents and W of keyphrases, keyquery generality parameters k and l
Output: Set Q of keyqueries covering W
foreach $w \in W$ **do**
  if w returns less than l results **then** $W \leftarrow W \backslash w$
$q \leftarrow$ ""
foreach $w \in W$ **do**
  $q \leftarrow q \circ w$
  if q is keyquery **then** $Q \leftarrow Q \cup \{q\},\ q \leftarrow$ ""
if $q \neq$ "" **then**
  foreach $w \in W$ **do**
    if $\nexists q' \in Q : q' \subset q \circ w$ **then**
      $q \leftarrow q \circ w$
      if q is keyquery **then** $Q \leftarrow Q \cup \{q\}$, **break**
return Q

First, keyphrases that retrieve less than l documents are removed. l, therefore, is the lower bound of documents that need to be retrieved for every query.
Then, keyphrases are concatenated to an at first empty query q. After each concatenation, it is checked if q is a keyquery; that is, q retrieves every document in D in its top-k ranks, and q retrieves more than l documents. k, therefore, is the upper bound rank of documents regarded as relevant.
If q is not empty after the first iteration step, a phrase w is tried to be concatenated to q so that none of the already existing keyqueries $q' \in Q$ is contained in $q \circ w$.

Algorithm 4.2: Related Work Search

Input : Set $\mathcal{I}$ of documents, keyquery generality parameters k and l, number r of desired related documents
Output: Set $\mathcal{R}$ of candidate articles
for $i \leftarrow |\mathcal{I}|$ down to 1 **do**
  foreach $D : D \in 2^{\mathcal{I}}, |D| = i$ **do**
    $W \leftarrow$ combine keyphrrases of documents in D
    $Q \leftarrow$ Keyquery Cover(D, W, k, l)
    $R \leftarrow$ combined top-l results of each $q \in Q$
    $\mathcal{R} \leftarrow \mathcal{R} \cup R$
    if $|\mathcal{R}| \geq r$ **then break**
return $\mathcal{C}$

While the documents in $\mathcal{I}$ are expected to be found, the number of found documents, after keyqueries for $\mathcal{I}$ have been generated, might be $< r = 100 \cdot |\mathcal{I}|$. Therefore, Related Work Search tries to retrieve documents for the subsets of the power set of $\mathcal{I}$ while choosing the next smaller subset with each iteration so that r articles are found.
To extract keyphrases, for every document in D, an exact-match query is sent to Elasticsearch using the document's title. The exact-match query only retrieves one document with the exact same title.
Keyphrases are extracted from the document's body using one of the keyphrase extractors discussed. Every extracted keyphrase is then ranked against the document it originates from using Elasticsearch.
After keyphrases are combined into W, Keyquery Cover is executed, and the list of keyqueries is stored in Q. Each $q \in Q$ is then queried using Elasticsearch, saving the top-l results.

Elasticsearch allows boosting search results using the indexed fields. The boosted fields[2] are the *title*, *links*, *categories*, and *redirects* for every indexed article. If, for example, the query $q = CRISPR$ is passed to the query-template, the document with $d_{title} = CRISPR$ will be boosted in its rank.

[2]The values by which documents did get boosted were mostly set based on intuition and a vague explanation in the paper.

5 Retrieving Articles with Wikipedias Graph

Wikipedia can be regarded as a directed graph when considering its articles and their linkage. In this case, every article would be a vertex and every link a directed edge.
Numerous graph-based algorithms already exist, ranking nodes based on their connectedness. PageRank [6] is probably the best-known candidate.
The appeal of PageRank and similar implementations, including the implemented approach here, is that they may simulate the behavior of a person (surfer) surfing the web. Given a starting point: these algorithms visit the links similarly to a surfer looking through Wikipedia. The similarity between two vertices u and v might then, for example, be regarded as the probability of a person starting on vertex u and ending up on v after n steps.
Ollivier et al. [11] introduce the implemented approach to find similar articles using *Green Measures* [12].

5.1 Creating the Wikipedia Graph

The graph is implemented using the graph-tool library [30]. Many graph-based algorithms, including PageRank, have already been implemented in graph-tool. Additionally, users can assign attributes (*PropertyMaps*) to vertices, edges, and the graph itself.
The terms *article* and *vertex* will be used interchangeably. While 'vertex' will always correspond to a vertex in a graph including all its properties and 'article' corresponds to a Wikipedia article with its title, body, etc., both terms represent the same concept.

The graph can be created in two iterations using the JSON files from the Wikimedia dump.
In the first iteration, the vertices are created. Every vertex gets either three or five properties assigned.
Each vertex v has an *id*, *title*, and a boolean value: *hanging*, which, if true, indicates that the vertex has no outgoing edges. If the vertex is created from a redirect page, it has two additional properties: *redirect* and *redto*. *redirect* indicates that the vertex is a redirect and *redto* saves the target of the redirect.
Furthermore, graph-tool assigns its own ID to every vertex: id_v. It uses this ID to index vertices in a JSON format. v_{id} stems from the Wikimedia dump and is equal to the ID given to each article by Wikipedia.[3] v_{title} is the lowered

[3]It is important to distinguish id_v and v_{id}. id_v is the ID assigned by graph-tool, which is assigned incrementally for every vertex. v_{id} is a property assigned to v.

title of the Wikipedia page. The title is lowered because links extracted from article bodies have no consistent casing. The same article may be linked in different ways. Lowering the title solves these inconsistencies but creates a new problem: If articles on Wikipedia can only be differentiated by their titles' casing: these articles will collide using this method. The calculated similarities may not reflect the accurate results.
The portion of these titles is so tiny that they can be neglected though. ($\sim$ 6.100 of $\sim$ 6.2 million[4]).
In the second iteration, vertices are connected using the articles links, and possible redirects are resolved.
Iterating through the JSON files, for every article u an edge is created to every outgoing link v. This is done by getting the respective ids id_u and id_v from the graph using u_{title} and v_{title}. It is checked if v is a redirect. If it is not, an edge from id_u to id_v is created, and $u_{hanging}$ is set to $false$. If v is a redirect, an edge from id_u to $id(v_{redto})$ is created, and $u_{hanging}$ is set to $false$. Doing this for every article in the file: the created graph represents Wikipedias' complete linkage.

5.2 Preprocessing the Graph

Two conditions must be met for the algorithm to work: the graph needs to be *irreducible* and *aperiodic*.
A graph is said to be irreducible if it is strongly connected, meaning that every vertex v can be reached by every other vertex and vice versa.

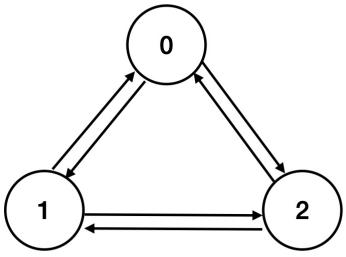


Figure 5.1: Irreducible Graph

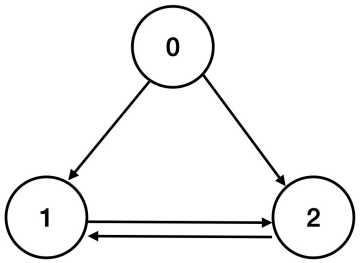


Figure 5.2: Non-Irreducible Graph

[4]6.2 million articles are less than the 6.4 million articles mentioned before. Not every article is extracted from the dump. To be precise: namespaces are assigned to articles and only articles with namespace-id= 0 (main articles, lists, redirects,...) are extracted from the XML dump.

A graph is aperiodic if the greatest common divisor of every cycle in the graph is equal to one. This condition is assumed to be true due to the size of the graph.
Before the irreducibility condition is met, a depth filter is applied to the graph. Centered at the vertex u: the depth filter will only keep vertices $n = 2$ iterations of links deep into the graph and accept all ingoing edges to u. This is done because working with the entire graph is computationally not feasible.
The implemented depth filter makes sure that the resulting graph has more than 300 vertices, incrementing n if this condition is not met. Therefore, it is also assumed that articles related to u are in the vicinity of u.

Kosaraju's algorithm [31] is applied to find a strongly connected subgraph of the Wikipedia graph.
The key component of this algorithm is that a DFS search is started from vertex s, similar articles are supposed to be retrieved from, and every visited vertex is prepended to a list L_K if it was unvisited before. This implies that if there is a forward path $u \to v$, for $u, v \in g$, u appears later in L_K than v. The second step is the iteration through L_K, so that for every $u \in L_K$ a recursive subroutine Assign(u, u) is called:

Algorithm 5.1: Assign

```
Input  : Vertex u, vertex root
if u_root is empty then
    u_root ← root
    for v ∈ in-edges(u) do
        Assign(v, root)
```

Now every vertex is assigned to a root component. To extract the strongly connected subgraph for s: every vertex not having s_{root} as its $root$ will be deleted from the graph. Graph-tool makes this easy by applying a filter using its PropertyMaps, so that every vertex whose $root$ differs from s_{root} will be filtered out.

5.3 Retrieving Similar Articles with Green Measures

To retrieve similar articles: the graph g, will be regarded as a *Markov chain*. A Markov chain is *"a stochastic model describing a sequence of possible events in which the probability of each event depends only on the state attained in the previous event"* [32].
Therefore the graph and its transitions are converted into a stochastic entity. This is done by assigning transitional probabilities p_{uv} to edges. p_{uv} is the probability to transition from vertex $u \in g$ to vertex $v \in g$ while holding that for each u, $\sum_v p_{uv} = 1$.
This is achieved by assigning probabilities to edges in the following order:

$$p_{uv} = \begin{cases} 1/d_u & \text{, if there is an edge from } u \text{ to } v \\ 0 & \text{, otherwise,} \end{cases} \tag{5.1}$$

where d_u is the number of outgoing edges of vertex u.

Having assigned transitional probabilities to every edge in g: the graph can be transformed into a *transition matrix* M. M is an $n \times n$ matrix, with n being the number of vertices in g. M_{uv} is then equal to p_{uv}.
A simple random walk can be started by multiplying M with itself. M^2 accords to two steps of the random walk, meaning that M^2_{uv} would be the probability of beginning a walk at vertex u and ending up on vertex v after two steps. Due to the condition $\sum_v p_{uv} = 1$, the total sum of probabilities in the matrix is preserved.
A row vector (probability measure) μ so that $\sum \mu_i = 1$ is introduced. Furthermore, *forward propagation* μM is then defined by $(\mu M)_u := \sum_u \mu_u p_{uv}$, meaning that each vertex u sends a part of its mass (corresponding to μ_u) to vertex v.
Given all these definitions and the preprocessed graph, the Markov chain M of g has a unique probability measure ν with $\nu M = M$. This measure is called *equilibrium measure*. For any measure μ with $\sum \mu_u = 1$, μM^n converges to ν faster than M^n as $n \to \infty$.
Looking at the definition of ν, one can see the resemblance to PageRank. ν can be thought of as PageRank without random jumps. As stated before, graph-tool has several algorithms already implemented, including PageRank. PageRank is executed on g, and the values are stored in ν.

To score every article in g regarding u, *Green Measures* are introduced. The *Green Matrix* of a finite Markov chain is defined as:

$$G := \sum_{t=0}^{\infty} (M^t - M^{\infty}) \tag{5.2}$$

where M^t is the t-th power of the matrix M, corresponding to t steps of the random walk. M^{∞} corresponds to ν.
Because calculating the score for $n \times n$ vertices is computationally too expensive, the calculation is centered at the vertex u. G_u is defined as the *Green measure centered at u*. G_u corresponds to the row u of the Green matrix G. Let δ_u be the *Dirac measure* centered at u, so that:

$$\delta_{uv} = \begin{cases} 1 & \text{, if } u = v \\ 0 & \text{, otherwise} \end{cases} \tag{5.3}$$

By definition $G_u = \delta_u G$ holds. The equation above can then be rewritten as:

$$G_{uv} := \sum_{t=0}^{\infty} ((\delta_u M^t)_v - \nu_v), \tag{5.4}$$

where $(\delta_u M^t)_v$ is the probability of starting a random walk at vertex u and being at vertex v after t steps.

With every value for G_u now being calculable, the final equation S, calculating the similarity between two vertices u and v, is introduced.

$$S^u(v) := G_{uv} \log(1/\nu_v) \tag{5.5}$$

G_{uv} is multiplied with the logarithm of ν_v to favor uncommon vertices.

6 Retrieving Articles using Text Similarity

While the keyquery method of retrieving articles uses text similarity measures to rank keyphrases and queries against documents, the respective similarity of two documents in the corpus is unknown.
The graph-based method does generate a similarity score for a pair of documents. Still, it does so only by regarding the graph's topology, disregarding the articles' textual information.
For this next section of *text similarity measures*, seven different methods of scoring documents have been implemented.
The general gist of retrieval for these models is the same: given an article a: generate a score for every article b in the corpus. Once every score is generated, the articles are sorted by their scores to rank similar articles first.

6.1 TF-IDF Retrieval

TF-IDF [33] is probably one of the most commonly used text similarity measures. As the name (term frequency-inverse document frequency) suggests, TF-IDF calculates values for terms in a document through *"the inverse proportion of the frequency of the word (term) in a particular document to the percentage of documents the word appears in."* [33] TF-IDF scores are calculated using the following equation.

$$TF - IDF(t, d, D) = TF(t, d) * IDF(t, D), \tag{6.1}$$

with

$$TF(t, d) = \frac{freq(t, d)}{|d|} \tag{6.2}$$

$$IDF(t, D) = log\Big(\frac{|D|}{n(\mathcal{C}, t)}\Big) \tag{6.3}$$

TF-IDF has been implemented using the *sklearn library* [34]. In this implementation, the IDF aspect is limited to two documents. For each article $a \in L$, only a itself is indexed and therefore part of the document collection D. For every article b that is to be scored, b is transformed into a TF-IDF-vector based on the terms present both in a and b.
TF-IDF itself only scores terms against documents. Using sklearn's TfidfVectorizer, sklearn uses a pipeline to convert a given document into a CountVector representing the occurrences of each term in the document

and then creating a TF-IDF-vector based on the CountVector.
Having TF-IDF implemented this way allows one to only ever compare two documents directly instead of comparing two documents relative to the entire corpus. This saves time by not indexing the whole Wikipedia corpus and then creating the vectors which would need to calculate their values based on every document indexed ($\sim$ 6.2 million).
To create scores for every $a \in L$: a vector for a is created and then for every article in the corpus. The cosine similarity is being calculated between both vectors, and the result is saved. After a score has been created for every document regarding a, the scores are ranked by their cosine similarity.

6.2 Doc2Vec Retrieval

Doc2Vec has been introduced by Mikolov et al. [16] and infers vectors from sentences, paragraphs or documents. Doc2Vec is based on Word2Vec [15] which infers vectors from words. Vectors generated by a trained Word2Vec model even support some form of algebra, so that:
$vec(\text{"}King\text{"}) - vec(\text{"}man\text{"}) + vec(\text{"}woman\text{"})$ is closest to $vec(\text{"}Queen\text{"})$.
This approach is implemented using the Gensim library [35] and its Doc2Vec module.

6.2.1 Training the Word2Vec Model

Every word gets a unique vector assigned to it with every vector being represented by a column in a matrix W of all words. The column is indexed based on the position of the words' first appearance in the corpus. To train the neural network using W: two models have been introduced.

The continuous bag of words model acts like a sliding window iterating over the corpus. When the sliding window is centered at the t-th word $w(t)$, the neural network tries to predict $w(t)$ using the words surrounding it in a specific range, r. So that if $r = 2$, the input neurons are given the vectors of $w(t-2), w(t-1), w(t+1), w(t+2)$ and the network tries to predict $w(t)$.
The continuous skip-gram model acts analogously by using the vector of $w(t)$ to predict the words surrounding $w(t)$ in a range, r.

Doc2Vec uses the *continuous bag of words* model to train its word vectors, and by default, $r = 5$.

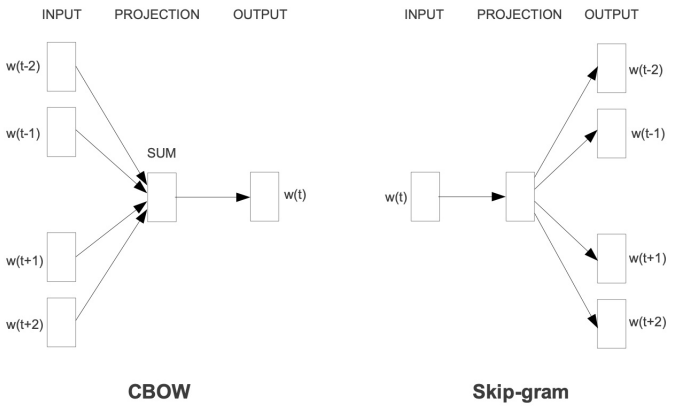


Figure 6.1: Word2Vec Training Models

6.2.2 Training the Doc2Vec Model

The paragraph (document) vectors contribute to a prediction task similar to Word2Vec.
First, documents are assigned a unique vector so that every document is represented as a column in a matrix D. The column is indexed based on the documents' first appearance in the corpus.
The network is then trained by creating a sliding window of fixed length over every document. For each emerging context, the documents vector is concatenated by a range of vectors of the following words from that context. Every emerging vector is fed into the neural network and predicts the next word.

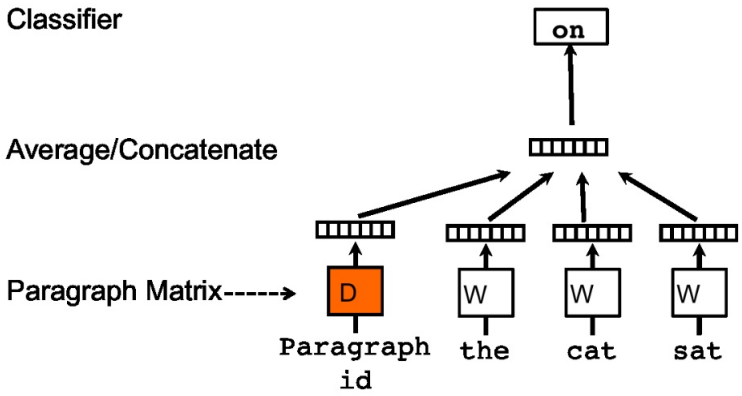


Figure 6.2: Dov2Vec Training

6.3 Universal Sentence Encoder Retrieval

Universal Sentence Encoder (USE) [17] is a collection of encoding models transforming text into vectors. USE is part of the *TensorFlow library* [36] and mainly incorporates two different encoding models using a *transformer*-based- [18] and *deep averaging network* (DAN) [19] based sentence encoding model. The transformer-based model promises better results in a tradeoff for computational complexity and longer training time. The DAN model trades less training for a less complex model.
Both models have been pre-trained by Google and are available to download.

6.3.1 Deep Averaging Network Model

The DAN model omits sentence structure to achieve faster training and embedding. The training is extended using word dropout which prevents overfitting. Given a sentence: the model averages the word-embeddings and feeds the resulting vector through a multi-layer neural network.

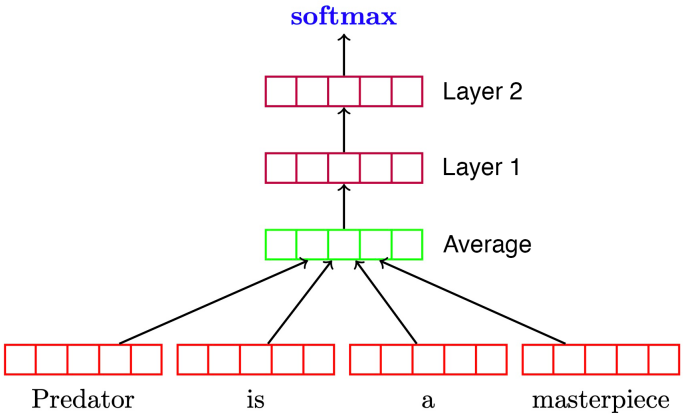


Figure 6.3: Deep Averaging Network

While training the network, words from the token sequence X are dropped from the sequence with a probability of p. This is done to avoid overfitting the network to the training data. Given a sequence X: $2^{|X|}$ possible sequences can be generated using this method.

6.4 Failed Models

Some of the implemented models didn't retrieve articles as expected. This didn't necessarily mean that the model was unfit for this task.
The following models have been implemented and checked if they do indeed retrieve articles. These models have not been evaluated in the same sense the others have.

6.4.1 Univeral Sentence Encoder - Transformer Model

Universal Sentence Encoders transformer model is modeled after the encoding sub-graph of the transformer architecture in *Attention Is All You Need* [18]. The transformer described in this paper is meant to be a *sequence transduction* model tasked to translate sentences. The architecture is based on an encoder-decoder model, with the encoder being connected to its decoder through an *attention* mechanism.
The idea is that an English sentence is encoded into a vector, and the decoder interprets (decodes) that vector and outputs a translated, french, sentence.

The model is designed to be as general-purpose as possible. So the same encoding model was trained using varying training tasks. The model was trained using a *skip-thought* task using running text, *input-response* tasks using conversational data, and a *classification* task on supervised data.

- the skip-thought model uses sentences from continuous text to predict the following sentence
- the input-response model uses questions from eMails to try to predict appropriate answers

6.4.2 Simhash/MinHash

While *Simhash* [37] and *MinHash* [38] are two different models, they are based on the same concept: hashing documents (creating fingerprints) and comparing these fingerprints to find near-duplicates.

SIMHASH tries to solve a dimensionality problem. Similar to earlier methods, it creates a vector from the document's body. The document is split into k-shingles. Where a *shingle* is a string consisting of k characters or words. The word *"hippopotamus"*, for example, could be divided into:
"hippo", "ippop", "ppopo", "popot"... with $k = 5$.

Hashing these shingles to a 64-bit integer value creates a unique set of (binary) hashes H_d for every document d. An empty *document-hash* h_d (consisting of zeros) is created. For each $h \in H_d$, h is added to h_d so that a bit in h_d is incremented if that same bit in h has the value 1 and decremented if the value is 0.
h_d is a vector of positive and negative values. If h_d is now altered so that all numbers greater than 0 are 1 and every other number is changed to 0, every document has a unique binary fingerprint.

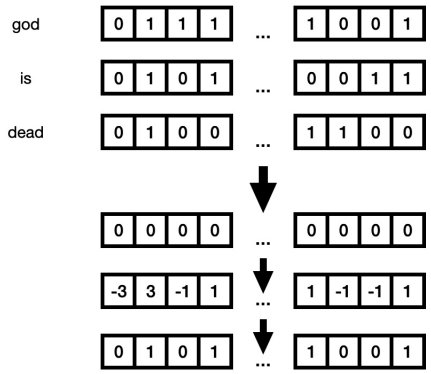


Figure 6.4: Simhash Document Vector

If a document d' has the same shingles it has the same document-hash. If d' has a few minor changes, $h_{d'}$ may be similar but not the same as h_d.
To calculate the similarity between these hashes: the hamming distance is calculated.
MinHash, much like Simhash is also based on hashing shingles. After hashing every shingle for the document d, a set of these hashed shingles: S_d has been created. To calculate the resemblance between two documents d and d': the following equation is calculated.

$$r(d, d') = \frac{S_d \cap S_{d'}}{S_d \cup S_{d'}}. \tag{6.4}$$

6.4.3 Measuring the Semantic Similarity of Articles

Courley et. al. [39] introduce a knowledge-based approach to find semantically similar texts using *WordNet*. In WordNet, words are grouped into sets of cognitive synonyms (synsets). Each synset expresses a concept and has conceptual relations to other synsets in the database. Relations between synsets are used to calculate similarities between words and ultimately between documents. This approach has been implemented using *Gensim* [35].

Several word-to-word-similarity metrics have been described in the paper and are implemented in the WordNet library. The *Wu and Palmer* similarity [40] has been used in this implementation.
To calculate text similarity for a given pair of texts: sets of *nouns, verbs, adjectives, adverbs*, and *cardinals* for both texts are created using POS-tags. Similar words between the sets of the same open class are then being determined. The similarity is calculated using the *Wu and Palmer* similarity for nouns and verbs. For each noun/verb, the noun/verb with the highest semantic similarity $maxSim$ in the other text is identified and saved. For every other set, similar words are determined by lexical matching: the semantic similarity is equal to 1 if the adjective (adverb, cardinal) is contained in the other document, 0 otherwise. The similarity between the documents is then calculated as:

$$sim(d_1, d_2)_{d_1} = \frac{\sum_{\mathcal{C} \in POS} (\sum_{w \in \mathcal{C}} maxSim(w) * specificity(w))}{\sum_{\mathcal{C} \in POS} \sum_{w \in \mathcal{C}} specificity(w)} \tag{6.5}$$

where $specificity$[5] is the normalized $depth$.
The way the similarity between two documents is being calculated here is unidirectional meaning the scores of $sim(d_1, d_2)_{d_1}$ and $sim(d_2, d_1)_{d_2}$ differ. To get a bidirectional score, the values are calculated and averaged:

$$sim(d1, d2) = \frac{sim(d_1, d_2)_{d_1} + sim(d_2, d_1)_{d_2}}{2} \tag{6.6}$$

[5]WordNet has no entries for some scientific terminology like 'CRISPR' or 'Cas9'. While POS-tagging might correctly classify them as nouns, not being able to calculate the score for crucial terminology would corrupt the results. Because of this reason, the highest specificity is assigned to words which have not been found in the WordNet Database.

6.4.4 Word Movers Distance

Word Movers Distance (WMD) is a similarity measure introduced by Kusner et al. [41], which is an optimization problem based on the *Earth Mover's Distance* (EMD). In mathematics, EMD is an optimization problem that minimizes the cost of turning one landmass into another. Therefore, WMD tries to calculate the minimum cost to transform one document into another. This approach makes use of Word2Vec [15], which embeds words into vectors so that vectors of semantically similar words are close to each other.

Given a trained Word2Vec model on n words, the vectors are embedded in an $\mathbb{R}^{m \times n}$ matrix so that the i^{th} column represents $x_i \in \mathbb{R}^m$, the i^{th} word in m-dimensional space.
Text documents can be represented by a vector $v_d \in \mathbb{R}^n$ with:

$$v_d^{(i)} = \frac{freq(i,d)}{\sum_{j \in d} freq(j,d)}. \tag{6.7}$$

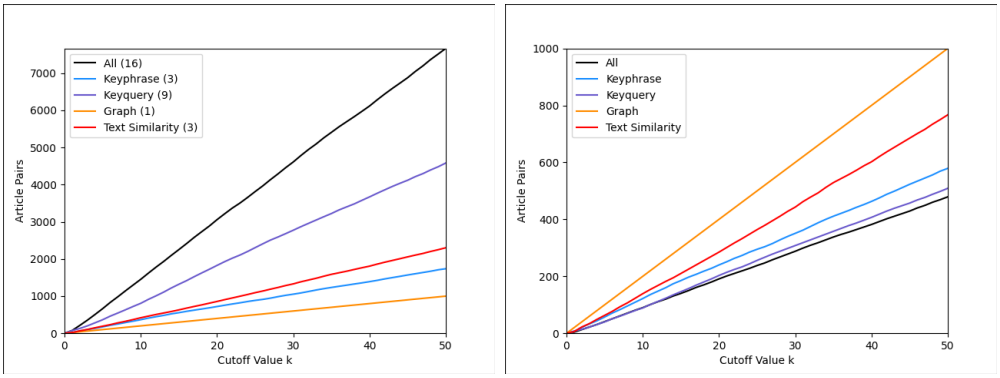
Given this representation of documents, the vectors of semantically similar documents will still be far apart if they don't share the same words. To incorporate the semantics of the documents, the *word travel cost*: $c(i,j)$, is introduced:

$$c(i,j) = \|x_i - x_j\|_2, \tag{6.8}$$

This builds upon the fact, that two word embeddings $x_i, x_j \in \mathbb{R}^{m \times n}$ are close if they share meaning. To transform a document vector v_d into another document vector v'_d: every word $i \in d$ is allowed to transform into any word $j \in d'$ to any extent. To keep track of these transformations, $T \in \mathbb{R}^{n \times n}$ is introduced so that T_{ij} is the amount word i transforms into word j.
To ensure the complete transformation, T is subjected to two conditions regarding its minimization.

7 Discussion

The following section discusses the experimental results of each retrieval approach. Every approach has retrieved the top-20 results for every topic in L. The evaluation was limited to the top-20 results because of the article-pool size: 3058 of hypothetical 6400 (16 retrieval methods * 20 retrieved articles per method * 20 topics)[6] article pairs have been retrieved.
Against expectations, the number of retrieved article pairs seems to be linearly correlated with the cutoff value k of regarded ranks. Intuitively, the approaches should retrieve similar articles in their first few ranks and scatter in their similarity as k increases, resulting in a parabola-like shape.



(a) Total Retrieved Article-Pairs

(b) Average Retrieved Article-Pairs per Method

Figure 7.1: Retrieved Article Pairs

As Figure 7.1(a) suggests, the retrieved article pairs are disjoint for every group of measures, resulting in a linear shape. Though 7.1(b) indicates that the keyquery- and keyphrase approach retrieve more similar articles regarding their respective retrieval methods, resulting in fewer new articles per regarded rank k. This can probably be attributed to the fact that although different similarity measures and keyphrase extractors were used, the retrieval method is the same. Contrarily, the implemented text similarity approaches work so differently that they averagely retrieve the second to most new articles per regarded rank.

[6]The actual value is probably closer to 5560 articles because the topic itself is expected to be retrieved in the top-20; additionally, the four given ground truth articles given to the keyquery approach are also expected to be retrieved for every combination of similarity measure and extractor.

This, however, does not give any information about the quality of the results. It only shows that these methods of retrieval work in different ways and do not necessarily retrieve the same set of articles.

When evaluating these approaches, it became apparent that the 'standard practice' evaluation measure *recall* did not contribute meaningful information.
Because so many approaches were implemented, a variety of unique articles (Figure 7.1) were retrieved, and therefore the ground truth for every topic in L was rather large. Counting documents labeled ≥ 2 as relevant, the ground truth averagely contained 91 articles, and 39 articles, when counting only articles labeled '3' as relevant.
Therefore, retrieving only the top-20 articles can receive a maximum recall score of ~ 0.22 or ~ 0.51, respectively. This is also why counting only documents labeled '3' as relevant, the stricter evaluation method, continuously received a better recall score.
The *mean Average Precision* (mAP)- [42] and *nDCG*- [43] scores were calculated to compare results. Both measures only take the retrieved articles ranking regarding the ground truth into account. mAP does so by calculating the precision at every depth $(1, ..., k)$ and dividing the result by the number of ranks regarded $(= AP)$. Doing this for every query (topic in L) and calculating the mean yields the mean Average Precision.

$$AP@k = \frac{\sum_{i=i}^{k} rel_i * prec@i}{min(k, |\text{ground truth}|)}, \tag{7.1}$$

where rel_i is the binary value indicating whether the document on rank i is relevant and $prec@i$ is the precision regarding the first i ranks.
nDCG calculates the *Discounted Cumulative Gain* (DCG) for a query and divides that score by the *ideal DCG* (iDCG), which corresponds to the DCG of the ranked ground truth.

$$DCG = \sum_{i=1}^{k} \frac{rel_i}{log_2(i+1)} \tag{7.2}$$

mAP and nDCG can be regarded as the quality of retrieved information and ranking by the retrieval methods.

7.1 Elasticsearch Retrieval

7.1.1 Keyphrase Approach

The keyphrase approach uses the title of every article found in L and passes it to *query-template* as used in the keyquery approach. Tables 7.1 and 7.2 contain the experimental results of the keyphrase approach.

Relevant Documents ≥ 2				
Retrieval Method	mAP	nDCG@5	nDCG@10	nDCG@20
BM25	0.575	0.785	0.715	0.655
DFI	**0.661**	**0.826**	**0.777**	**0.709**
LMDirichlet	0.659	0.821	0.754	0.693

Table 7.1: Keyphrase Results for Relevant Documents Labeled ≥ 2

Relevant Documents $= 3$				
Retrieval Method	mAP	nDCG@5	nDCG@10	nDCG@20
BM25	0.306	0.695	0.593	0.501
DFI	**0.373**	**0.735**	**0.666**	**0.559**
LMDirichlet	0.329	0.688	0.606	0.519

Table 7.2: Keyphrase Results for Relevant Documents Labeled '3'

Regarding both evaluations, DFI scores the best results.
Disregarding the time it took to index the corpus: the keyphrase approach retrieved its results the fastest. Querying Elasticsearch usually retrieves documents in 3-10 seconds, taking about 90 seconds per measure. This approach can also be parallelized, querying the BM25-, DFI- and LMDirichlet index simultaneously.
On the other hand, indexing the corpus was more time-intensive, taking about three days per measure, with all three similarity measures being indexed simultaneously.

7.1.2 Keyquery Approach

The keyquery approach was implemented as described. Given a list of relevant articles: related articles are found.
This is the only approach that works with a set of predetermined relevant articles for every topic in L. This has two noticable side effects: (1) a list of articles deemed relevant must be found, (2) the results are based on these articles.
While (1) can probably be achieved quite easily, one might argue that this goes against the principle of article retrieval. *'If four articles have been deemed relevant, I can read up on those articles.'* - Of course, different articles will be found using the approach, as (2) suggests though, better results might be achieved given more articles.
How many articles should the ground truth contain? This implementation chose a ground truth of four articles, including the topics main-article. Adding more articles to the ground truth should yield better results though. Where should the line be drawn? If the ground truth contains, for example, ten articles, the best scores might be achieved, but at that point, relevant documents have been found. Not all, but maybe enough to read up on a given topic.
Additionally, (2) comprises a problem every approach shares: the retrieved results are based on a number of parameters that can be optimized. While the keyquery approach has these parameters present in the similarity measures, keyphrase extractors, and the keyquery generality parameters k and l: defining which query will be deemed a keyquery, the retrieved articles for this approach are based on $\mathcal{I}$, as discussed. A list of relevant articles for a topic can be created with little effort, but creating a list that will yield the best results is almost impossible. Using this approach, a list of the most relevant articles for a topic might not deliver the best results. There might always be a set of articles (maybe narrowly related to the topic) that will yield better results.

Tables 7.3 and 7.4 contain the experimental results using a ground truth of four articles for every topic, pairing each similarity measure with each keyphrase extractor.

Regarding the two tables, YAKE! yields the best results using LMDirichlet scoring, producing the third-highest overall score.
Articles have been retrieved using multiprocessing so that each process has been given a ground truth and retrieved articles for every similarity measure using one keyphrase extractor. Using this method: articles for one extractor,

Relevant Documents ≥ 2					
Retrieval Method		mAP	nDCG@5	nDCG@10	nDCG@20
BM25	RAKE	**0.658**	**0.840**	**0.792**	**0.693**
	YAKE!	0.572	0.750	0.708	0.632
	KP-Miner	0.508	0.775	0.676	0.588
DFI	RAKE	0.586	0.830	0.757	0.656
	YAKE!	**0.675**	**0.875**	**0.795**	**0.712**
	KP-Miner	0.570	0.824	0.750	0.648
LMDirichlet	RAKE	0.529	0.850	0.740	0.620
	YAKE!	**0.705**	**0.859**	**0.825**	**0.730**
	KP-Miner	0.625	0.854	0.779	0.679

Table 7.3: Keyquery Results for Relevant Documents Labeled ≥ 2

Relevant Documents = 3					
Retrieval Method		mAP	nDCG@5	nDCG@10	nDCG@20
BM25	RAKE	**0.339**	**0.733**	**0.652**	**0.521**
	YAKE!	0.291	0.672	0.575	0.472
	KP-Miner	0.230	0.645	0.527	0.417
DFI	RAKE	0.322	0.723	0.616	0.502
	YAKE!	**0.356**	**0.752**	**0.640**	**0.531**
	KP-Miner	0.302	0.682	0.604	0.496
LMDirichlet	RAKE	0.284	0.761	0.616	0.475
	YAKE!	**0.384**	**0.791**	**0.691**	**0.560**
	KP-Miner	0.313	0.742	0.615	0.504

Table 7.4: Keyquery Results for Relevant Documents Labeled '3'

paired with all the similarity measures, could be retrieved in about sixteen hours. This was done three times - for each extractor. More parallelism can probably be achieved, but Elasticsearch timed out when sending too many queries in earlier tests.

Most time is spent trying to find keyqueries. Choosing k more lenient - approving keyqueries if, for example, the relevant documents only appear in the top-50 ranks would improve the speed while probably scoring worse. Vice versa, choosing stricter parameters could yield better results with more time being spent finding keyqueries.

Interestingly, the keyquery approach is the only method that didn't retrieve its main article on its first rank consistently. This is the case because it is only supposed to retrieve its given ground truth in the top-k ranks.

7.2 Graph Retrieval

The graph approach of article retrieval suffered under Wikipedias size. The corpus had to be scaled down massively to yield any results. It is hard to tell if using the entire Wikipedia graph would have yielded much better results but filtering the depth of the graph definitely distorted the results to some extent. The sizes of the resulting subgraphs after Kosaraju's algorithm is applied vary from 215- to 11003 vertices. The second one has 14 vertices after the first depth filter is applied, 81 after the depth is incremented, and then 11051 after the depth is incremented further.
When comparing the results for a few examples, counting documents labeled '3' as relevant,

Title	mAP	nDCG@5	nDCG@10	nDCG@20	Graph Size
CRISPR	0.236	0.869	0.630	0.444	215
Bitcoin	0.056	0.339	0.220	0.176	10412
Central Solar Heating	0.150	0.485	0.445	0.369	11003

Figure 7.2: Comparing Graph Sizes

it is apparent, that a graph with fewer vertices can yield better results than a larger graph. While a comparison of this size is nowhere near enough to make any meaningful suggestions, it shows that the quality of result is not restricted by the number of vertices a graph has.

Tables 7.5 and 7.6 contain the results of the graph approach.

Overall, this approach yielded the worst results. This may be attributed to many things.
To avoid computational complexity: the graph was limited in its depth. When increasing the depth, the size of the graph grew exponentially, with many of these new vertices probably not being relevant.
Some sort of *filter* could be applied to select the outgoing links to follow in every article.
Additionally, choosing varying transitional probabilities may yield better results. TF-IDF could be used to score the links against their document.

Relevant Documents ≥ 2				
Retrieval Method	mAP	nDCG@5	nDCG@10	nDCG@20
Graph	0.445	0.674	0.597	0.546

Table 7.5: Graph Results for Relevant Documents Labeled ≥ 2

Relevant Documents $= 3$				
Retrieval Method	mAP	nDCG@5	nDCG@10	nDCG@20
Graph	0.205	0.569	0.461	0.381

Table 7.6: Graph Results for Relevant Documents Labeled '3'

Normalizing the results would then yield transitional probabilities. Using this method: the depth may also be increased by choosing only the first n links by their scores for every document, where n could vary depending on the number of links per document.

The Wikipedia graph was created in about three hours. It was then saved for later use.
With the graph created, the results were calculated in $\sim$ 12.5 hours. Most time was spent calculating the G_{uv} values for the largest graphs. This is to be expected because the graph matrix M is multiplied by itself many times. When working with matrices of this size (with many 0), it was beneficial to use CSR matrices. CSR matrices only store values not equal to zero in three different arrays. This helped with memory management but had the disadvantage of slower matrix multiplication.
Per topic, about twenty minutes were spent loading the existing Wikipedia graph back into memory, totaling about 400 minutes. This may also be avoided by keeping the graph in memory but was of no concern on this scale.

7.3 Text Similarity

The three text similarity measures that have been evaluated shared a similar implementation style and retrieval method. For every topic in L, a vector is created by the method. This vector is then compared with the vectors of every other article in the corpus using their cosine similarity.
Of the three implementations, the Doc2Vec model was the only one receiving extra training. Trained models are available but Doc2Vec/Word2Vec only creates vectors reliably on already seen data. If there, for example, is a trained Wikipedia model trained one or two years ago, some of Wikipedias' articles might have changed drastically, distorting the results yielded by the model. While training Word2Vec on Wikipedia is a time-intensive step, it is a necessary one.
On the other hand, Universal Sentence Encoder has several trained models available for its two architectures: the deep averaging network and transformer model. These models have been trained on various tasks and should yield good results without any further training.

Tables 7.7 and 7.8 contain the three text similarity measures results with Universal Sentence Encoder using the deep averaging network to create vectors.

Relevant Documents ≥ 2				
Retrieval Method	mAP	nDCG@5	nDCG@10	nDCG@20
TF-IDF	**0.796**	**0.911**	**0.851**	**0.799**
Doc2Vec	0.665	0.835	0.769	0.713
USE	0.751	0.865	0.821	0.768

Table 7.7: Text Similarity Results for Relevant Documents Labeled ≥ 2

Relevant Documents = 3				
Retrieval Method	mAP	nDCG@5	nDCG@10	nDCG@20
TF-IDF	**0.450**	**0.816**	**0.716**	**0.632**
Doc2Vec	0.398	0.761	0.668	0.578
USE	0.419	0.786	0.692	0.604

Table 7.8: Text Similarity Results for Relevant Documents Labeled '3'

Surprisingly, TF-IDF, which conceptually was the easiest model, received the best overall scores, showing that complexity does not always merit better results.
Like every approach, these models could promise better results if their parameters were optimized. Especially Doc2Vec would benefit from longer training, and if USE's transformer model is supposed to generate better results, USE could outperform the TF-IDF approach.
Like the graph approach, the Universal Sentence Encoders transformer model might benefit from filtering information.
When passing entire articles to the model, 100 vectors are created in about 60 seconds. Assumed the performance will not vary, the entire corpus is vectorized in 43 days. Two solutions arise: (1) - letting the program run on 43 different machines, giving each machine some part of the corpus, and merging the results after each machine is finished. (2) - filtering out the first n words found in each article, creating vectors for each article's introduction, and reducing computational complexity.
(2) supports the belief that the most crucial information in each article is found in its introduction, similarly to an abstract in a scientific paper.
Implementing one of these approaches, using the transformer model for USE might score better results than TF-IDF.
Using the deep averaging network model, Universal Sentence Encoder retrieved its results the fastest in an *astounding* 10.5 hours.

The Doc2Vec model was trained in about 64 hours, training for eight epochs, a little less than the default of ten epochs.
After training, the retrieval step took about 28 more hours, totaling 92, including training. Doc2Vec's vectors were inferred sequentially. Using multiprocessing, this progress can be sped up by being mindful of the models' memory consumption ($>$ 1GB) and loading the JSON files (300MB-2GB).
Receiving the fourth-best overall score: the question arises if much better results would have been yielded if more training had been done. Compared to TF-IDF, which received no training, finished after 26 hours, and received the best overall results, the Doc2Vec model might (even with more training) not hold up regarding the task of article retrieval.

7.3.1 Failed Models

While Simhash and MinHash had no problem with scalability, it became apparent that these models were not designed for article retrieval. Both methods are designed to find near-duplicates in a large set of documents. Creating document hashes based on shingles does not capture any kind of similarity between two documents that describe the same topic in different words. This makes Simhash and MinHash unfit for article retrieval.

Using a knowledge-based approach (WordNet) to calculate semantic similarities between documents would seem to yield good results. Unfortunately, this approach does not scale. Averagely, this implementation yields a score every 22 seconds using a look-up table keeping track of already seen similarities. Not limiting the table will improve the speed over time, but the number of word-pair-similarities would need a table too big to keep in memory. Assuming that scores will be produced at that rate continuously, creating document similarities with the entire corpus for every topic in L is unfeasible.

Word Movers Distance also struggles at scale, averagely producing 100 scores every 25 seconds to retrieve one topic in approximately 18 days. This would need to be done 20 times for every topic found in L.

8 Conclusion

Three approaches to retrieve articles from Wikipedia have been implemented and evaluated. The evaluation shows that the TF-IDF text similarity approach scores the best overall results demonstrating that model complexity does not necessarily determine the quality of results that the model retrieves. However, Universal Sentence Encoder shows promising results using the DAN model yielding its results the fastest with a speedup of almost 2.5, while scoring only slightly worse than TF-IDF. With USE's transformer model promising better results than the DAN model, TF-IDF might be outperformed, trading TF-IDF's speed for computational complexity.
The keyphrase approach, using Elasticsearch as a search engine, querying only the articles' title, retrieves its results the fastest, assuming that the corpus is already indexed. Providing the right query template to boost the articles according to the query yields results not far from the more time-intensive keyquery approach. The keyquery approach, while receiving better scores, has drawbacks regarding the predetermination of a list of articles it needs to act as a ground truth. If a list of articles has already been determined to be relevant, generating keyphrases for these articles does yield better results than the keyphrase approach. If this list of articles must be determined first, the question might arise if the retrieval is still useful. Additionally, a perfect ground truth may never be found.
Given that the keyphrase approach retrieves its articles with arguable reliance, and much faster, keyqueries may not necessarily be used for article retrieval.
The graph approaches results suggest that - besides being impaired by the size of the Wikipedia corpus, there is a lack of information needed to retrieve better results. Following Wikipedia's link-tree does not produce any information about the article's contents but rather shows how many connections there are from one article to another.

After reviewing these measures, it became apparent that there is an inherent bias towards measures inferring vectors from documents. Creating a vector for every article on Wikipedia compresses all that information into a list of $|\mathcal{C}|$ vectors. This makes article retrieval easy by simply measuring the cosine similarity between two document-vectors.
Document-to-document similarities depend on calculating that similarity based on the given documents which tends to be more time-intensive.
Having a vector for every article on Wikipedia makes it more feasible to calculate the pairwise similarities for a more extensive set of articles.

References

[1] Wikipedia:statistics. https://en.wikipedia.org/wiki/Wikipedia:Statistics. Accessed: 2022-03-12.

[2] Holger Braun. *Innovation*. 2005.

[3] Dfg classification of scientific disciplines, research areas, review boards and subject areas (2016-2019). https://www.dfg.de/download/pdf/dfg_im_profil/gremien/fachkollegien/amtsperiode_2016_2019/fachsystematik_2016-2019_en_grafik.pdf. Accessed: 2022-03-12.

[4] Search engines. https://en.wikipedia.org/wiki/Search_engine. Accessed: 2022-03-12.

[5] Hypertext document retrieval system and method. https://patents.google.com/patent/US5920859A/en. Accessed: 2022-03-12.

[6] L. Page, S. Brin, R. Motwani, and T. Winograd. The pagerank citation ranking: Bringing order to the web. In *Proceedings of the 7th International World Wide Web Conference*, pages 161–172, Brisbane, Australia, 1998.

[7] Tim Gollub, Matthias Hagen, Maximilian Michel, and Benno Stein. From keywords to keyqueries: Content descriptors for the web. In *Proceedings of the 36th International ACM SIGIR Conference on Research and Development in Information Retrieval*, SIGIR '13, page 981–984, New York, NY, USA, 2013. Association for Computing Machinery.

[8] Matthias Hagen, Maximilian Michel, and Benno Stein. What Was the Query? Automatically Generating Queries for Document Sets with Applications in Cluster Labeling. In Elisabeth Métais, Mathieu Roche, and Maguelonne Tesseire, editors, *19th International Conference on Applications of Natural Language to Information Systems (NLDB 2015)*, volume 9103 of *Lecture Notes in Computer Science*, pages 124–133, Berlin Heidelberg New York, June 2015. Springer.

[9] Tim Gollub, Matthias Busse, Benno Stein, and Matthias Hagen. Keyqueries for Clustering and Labeling. In Yi Chang, Zhicheng Dou, Yiqun Liu, Shaoping Ma, Ji-Rong Wen, Min Zhang, and Xin Zhao, editors, *12th Asia Information Retrieval Societies Conference (AIRS 2016)*, pages 42–55, Berlin Heidelberg New York, November 2016. Springer.

[10] Matthias Hagen, Anna Beyer, Tim Gollub, Kristof Komlossy, and Benno Stein. Supporting Scholarly Search with Keyqueries. In Nicola Ferro, Fabio Crestani, Marie-Francine Moens, Josiane Mothe, Fabrizio Silvestri, Giorgio Maria Di Nunzio, Claudia Hauff, and Gianmaria Silvello, editors, *Advances in Information Retrieval. 38th European Conference on IR Research (ECIR 2016)*, volume 9626 of *Lecture Notes in Computer Science*, pages 507–520, Berlin Heidelberg New York, March 2016. Springer.

[11] Y. Ollivier and P. Senellart. Finding related pages using green measures: An illustration with wikipedia. In *AAAI*, 2007.

[12] John G. Kemeny, James Laurie Snell, and Anthony W. Knapp. Denumerable markov chains. 1969.

[13] H. P. Luhn. A statistical approach to mechanized encoding and searching of literary information. *IBM Journal of Research and Development*, 1(4):309–317, 1957.

[14] Karen Spärck Jones. A statistical interpretation of term specificity and its application in retrieval. *Journal of Documentation*, 28:11–21, 1972.

[15] Tomas Mikolov, Kai Chen, G.s Corrado, and Jeffrey Dean. Efficient estimation of word representations in vector space. *Proceedings of Workshop at ICLR*, 2013, 01 2013.

[16] Quoc Le and Tomas Mikolov. Distributed representations of sentences and documents. In Eric P. Xing and Tony Jebara, editors, *Proceedings of the 31st International Conference on Machine Learning*, volume 32 of *Proceedings of Machine Learning Research*, pages 1188–1196, Bejing, China, 22–24 Jun 2014. PMLR.

[17] Daniel Cer, Yinfei Yang, Sheng-yi Kong, Nan Hua, Nicole Limtiaco, Rhomni St. John, Noah Constant, Mario Guajardo-Cespedes, Steve Yuan, Chris Tar, Yun-Hsuan Sung, Brian Strope, and Ray Kurzweil. Universal sentence encoder. *CoRR*, abs/1803.11175, 2018.

[18] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. *CoRR*, abs/1706.03762, 2017.

[19] Mohit Iyyer, Varun Manjunatha, Jordan Boyd-Graber, and Hal Daumé III. Deep unordered composition rivals syntactic methods for

text classification. In *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, pages 1681–1691, Beijing, China, July 2015. Association for Computational Linguistics.

[20] Wikimedia downloads. `https://dumps.wikimedia.org/backup-index.html`. Accessed: 2022-03-12.

[21] Elasticsearch framework. `https://github.com/elastic/elasticsearch`. Accessed: 2022-03-12.

[22] Okapi bm25. `https://en.wikipedia.org/wiki/Okapi_BM25`. Accessed: 2022-03-12.

[23] Bekir Dincer, Ilker Kocabas, and Bahar Karaoğlan. Irra at trec 2010: Index term weighting by divergence from independence model. 01 2010.

[24] Chengxiang Zhai and John Lafferty. A study of smoothing methods for language models applied to ad hoc information retrieval. In *Proceedings of the 24th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, SIGIR '01, page 334–342, New York, NY, USA, 2001. Association for Computing Machinery.

[25] Stuart Rose, Dave Engel, Nick Cramer, and Wendy Cowley. *Automatic Keyword Extraction from Individual Documents*, pages 1 – 20. 03 2010.

[26] Ricardo Campos, Vítor Mangaravite, Arian Pasquali, A. Jorge, C. Nunes, and A. Jatowt. A text feature based automatic keyword extraction method for single documents. In *ECIR*, 2018.

[27] Samhaa R. El-Beltagy and Ahmed Rafea. KP-miner: Participation in SemEval-2. In *Proceedings of the 5th International Workshop on Semantic Evaluation*, pages 190–193, Uppsala, Sweden, July 2010. Association for Computational Linguistics.

[28] rake-nltk library. `https://github.com/csurfer/rake-nltk`. Accessed: 2022-03-12.

[29] Florian Boudin. pke: an open source python-based keyphrase extraction toolkit. In *Proceedings of COLING 2016, the 26th International Conference on Computational Linguistics: System Demonstrations*, pages 69–73, Osaka, Japan, December 2016.

[30] graph-tool library. https://git.skewed.de/count0/graph-tool. Accessed: 2022-03-12.

[31] Kosaraju's algorithm. https://en.wikipedia.org/wiki/Kosaraju\%27s_algorithm. Accessed: 2022-03-12.

[32] Markov chain. https://en.wikipedia.org/wiki/Markov_chain. Accessed: 2022-03-12.

[33] tf–idf. https://en.wikipedia.org/wiki/Tfidf. Accessed: 2022-03-12.

[34] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[35] Radim Řehůřek and Petr Sojka. Software Framework for Topic Modelling with Large Corpora. In *Proceedings of the LREC 2010 Workshop on New Challenges for NLP Frameworks*, pages 45–50, Valletta, Malta, May 2010. ELRA.

[36] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[37] Gurmeet Singh Manku, Arvind Jain, and Anish Das Sarma. Detecting near-duplicates for web crawling. In *Proceedings of the 16th International Conference on World Wide Web*, WWW '07, page 141–150, New York, NY, USA, 2007. Association for Computing Machinery.

[38] Andrei Z. Broder. Identifying and filtering near-duplicate documents. In Raffaele Giancarlo and David Sankoff, editors, *Combinatorial Pattern Matching*, pages 1–10, Berlin, Heidelberg, 2000. Springer Berlin Heidelberg.

[39] Courtney D Corley and Rada Mihalcea. Measuring the semantic similarity of texts. In *Proceedings of the ACL workshop on empirical modeling of semantic equivalence and entailment*, pages 13–18, 2005.

[40] Zhibiao Wu and Martha Palmer. Verbs semantics and lexical selection. In *Proceedings of the 32nd Annual Meeting on Association for Computational Linguistics*, ACL '94, page 133–138, USA, 1994. Association for Computational Linguistics.

[41] Matt Kusner, Yu Sun, Nicholas Kolkin, and Kilian Weinberger. From word embeddings to document distances. In Francis Bach and David Blei, editors, *Proceedings of the 32nd International Conference on Machine Learning*, volume 37 of *Proceedings of Machine Learning Research*, pages 957–966, Lille, France, 07–09 Jul 2015. PMLR.

[42] Effectiveness measures. `https://webis.de/downloads/lecturenotes/information-retrieval/unit-en-ir-effectiveness-measures.pdf`. Accessed: 2022-03-12.

[43] Discounted cumulative gain. `https://en.wikipedia.org/wiki/Discounted_cumulative_gain`. Accessed: 2022-03-12.