MASTER IN COMPUTER SCIENCE

Recommending ordered sections with similar section filtering to help structuring Wikipedia articles

Master Thesis

Sergiy Goloviatinski

University of Neuchâtel

Supervisor
Prof. Dr. Philippe Cudré-Mauroux
eXascale Infolab, Department of Informatics, University of Fribourg

Co-Supervisor
Natalia Ostapuk
eXascale Infolab, Department of Informatics, University of Fribourg

February 2022

UNIVERSITÄT BERN

UNIVERSITÉ DE NEUCHÂTEL

UNIVERSITÉ DE FRIBOURG
UNIVERSITÄT FREIBURG

Abstract

Wikipedia articles which belong to the same category often have a similar layout, *i.e.,* a similar set of sections in a particular order. For example, an article about a country usually starts with those sections: “Etymology”, “History”, “Geography”, “Politics”. Usually new articles are structured based on the precedent of similar articles, and there is no universal official guideline regarding the layout. Therefore it is difficult for a new user to expand existing articles or create new ones, because the user would need to get inspiration from already existing articles.

This project consists in reproducing and improving an already existing Wikipedia section recommendation algorithm [1] which used the most frequent sections from similar categories to recommend sections ranked by their frequency to an already existing or new article.

We have firstly improved the algorithm by removing noise in the recommended sections, which increased precision@1 from 0.55 to 0.65 and recall@20 from 0.7 to 0.77.

We resolved two limitations of the above-mentioned algorithm: we filtered semantically redundant sections from recommendation lists *e.g.,* if ”Career” and ”Later career” appeared at the same time in a recommendation list, we kept only one of them. This gave us an improvement in terms of f1 score@20 from 0.235 to 0.29. We also introduced a more logical ordering of the sections in the recommendation lists, *e.g.,* ”Conclusion” could no more be before ”Introduction” even if ”Conclusion” was more frequent than ”Introduction”. We improved the average Kendall’s tau correlation coefficient (which was used to compare the ordering between recommendation lists and the ground truth) from 0.22 to 0.89.

Contents

List of Figures

List of Tables

List of Algorithms

1 Introduction

Wikipedia articles are structured with sections, which subdivide the article in parts. While creating a new article or expanding an already existing one, the only way for now is to take already existing articles as example because there are no official guidelines on the article's layout, only on the general structure[1] (*i.e.,* what to put in the first paragraph, what has to be in the appendices *etc.*). Some categories (*e.g.,* articles about Aviation[2], Firearms[3] *etc.*) have community-made guidelines for the article style and sometimes for the article structure[4]. There are very few such categories and for the vast majority of articles the layout is not predefined. Therefore, it is difficult for a new user to know how to structure Wikipedia articles that belong to categories which don't have community-made guidelines.

The number of sections in an article is a factor which differentiates articles between those marked as being of good quality[5] and all other articles (excluding those that are explicitly marked as being of poor quality, *i.e.,* stubs) as shown on Figure 1.1.

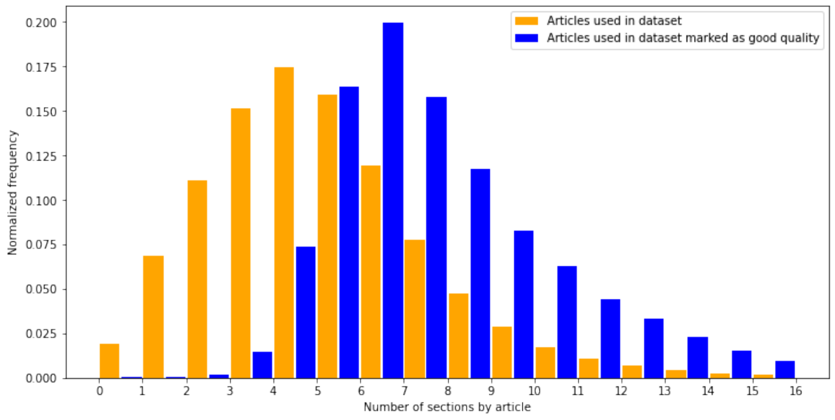


Figure 1.1: Distribution of number of sections, comparing all articles used in dataset (*i.e.,* excluding stubs) with those marked as being good quality

Articles which have no or only one section are called stubs[6] and therefore are too short to provide encyclopedic coverage of a subject, those articles account for 37% of all articles in March 2021. Articles are marked as stubs by Wikipedia users, therefore there are articles which are too short but aren't marked as stubs[7], which

[1] https://en.wikipedia.org/wiki/Wikipedia:Manual_of_Style/Layout
[2] https://en.wikipedia.org/wiki/Wikipedia:WikiProject_Aviation/Style_guide/Layout
[3] https://en.wikipedia.org/wiki/Wikipedia:WikiProject_Firearms#Structure
[4] https://en.wikipedia.org/wiki/Category:WikiProject_style_advice
[5] https://en.wikipedia.org/wiki/Wikipedia:Good_articles
[6] https://en.wikipedia.org/wiki/Wikipedia:Stub
[7] *e.g.,* https://en.wikipedia.org/w/index.php?title=Hasid&oldid=998746951

explains that on Figure 1.1 even if stubs are excluded from the distribution, there are still articles with 0 sections.

Only 0.5% of articles were marked as being of good quality in March 2021. Good quality articles tend to have more sections with an average of 7.32 sections compared to an average of 4.63 for all articles. Therefore, providing users with a section recommendation list could lead to longer articles which could increase the overall quality of articles.

Additionally, 87% of sections titles in March 2021 were used only in a single article, which shows that there are inconsistencies in terms of section usage (*e.g.,* instead of a section titled as ”Early life of Dionysius the Elder”[8], ”Early life” could be used, which would be more consistent with other articles).

This shows that there is a need for the standardization of the structure of Wikipedia articles (*i.e.,* avoid the usage of unique section titles which could be replaced with already existing ones) and that recommending a list of frequently used sections to users could help them getting inspired which could lead to more good quality articles. Therefore, an algorithm for recommending sections for new or already existing articles could respond to an existing need.

One of the recent efforts in this direction is [1]. The idea of this section recommendation algorithm is to automate the process of taking already existing articles as example and to recommend the most frequent sections from other articles of the same categories to which an already existing article belongs in order to expand it by adding new sections. The same principle can be used to build a list of recommended sections for a new article, after choosing a set of categories for which the user wants to create a new article. The authors used Wikipedia's category network describing child-parent relationships (*e.g.,* the category ”1893 births” is a subcategory of ”1890s births”) in order to enrich recommendations for a given category with sections from articles coming from this category's child categories.

The authors of this algorithm have proposed several ways to improve their algorithm, among which we have chosen two to work on:

- Filter semantically redundant sections, *e.g.,* if ”Career” and ”Later career” are both present in a recommendation list, only one of them should be retained and the other filtered out because they are semantically similar.
- Order sections in the recommendation list with their logical order, because in the initial algorithm the sections are ordered by their frequency. This means that the section ”Conclusion” can be ordered before ”Introduction” in the recommendation list if the first is more frequent than the second.

1.1 Contributions

The aim of this project is to reproduce the already existing section recommendation algorithm [1] and to improve its performance.

While reproducing the algorithm, we noticed that the recommended sections contained noise which lowered the algorithm's performance. This noise came from some articles from child categories that were used to enrich recommendations for a given category. We found which kind of articles contributed to the noise and filtered them out in order to avoid introducing noise in the recommendations, which increased the algorithm's performance.

We also resolved two limitations of this algorithm:

- We developed a method which filtered semantically redundant sections from the recommendation lists, *e.g.,* if a recommendation list contained at the same time sections ”Career” and ”Later career”, we are now able to detect that those two sections are semantically similar and we retain only one of them.
- We introduced the logical ordering of the recommended section lists. Initially, because the algorithm recommended the most frequent sections from a set of categories, the sections were ranked by their frequency, which could lead to cases where *e.g.,* the section ”Conclusion” could be ranked before ”Introduction” if the former was more frequent than the latter. With our solution, this case is no longer possible, the ”Introduction” section is guaranteed to be ranked before the ”Conclusion” section.

[8] `https://en.wikipedia.org/w/index.php?title=Dionysius_I_of_Syracuse&oldid=1007943734`

We also implemented a prototype in order to demonstrate how this algorithm could help a user to create new Wikipedia articles by suggesting a structure for the table of contents based on the choice of one or more categories. This prototype also generates a template in the Wikipedia markdown format ready to be used in the Wikipedia editor.

1.2 Project repository

The github repository of this project can be downloaded here: `https://github.com/SergiyGolov/wikipedia_section_recommendation`. The `README.md` file at the root of the repository contains details on how to reproduce the results with Wikipedia's 2021 or 2017 data or how to apply the method on up-to-date data. A link to download the results is also provided in the `README.md` file.

1.3 Outline

This thesis consists of 8 chapters:

1. Introduction
2. Background: we explain in detail the paper on which this thesis is based on in Section 2.1 as well as the theoretical background which is used throughout this thesis in Section 2.2 (semantic similarity), Section 2.3 (graph theory) and Section 2.4 (section ordering).
3. Reproduction of existing method and improvement: this chapter contains details on how to build the dataset (Section 3.1), how the paper was reproduced (Section 3.2) and how we found a way to improve the algorithm's performance (Section 3.3).
4. Detect and filter semantically similar sections: we describe our methodology of Wikipedia article text extraction in Section 4.1. Then we describe how we grouped together Wikipedia categories in order to be able to retrieve sets of sections that were likely to appear together in the same recommendation lists (Section 4.2). Finally we show how we compared those sections semantically among each other in order to detect which sections were semantically similar (Section 4.3), which allowed us to filter out semantically redundant sections in recommendation lists (Section 4.4).
5. Section ordering: in Chapter 5 we propose a method to order sections in the recommendation lists in a way which permitted to preserve the original order from the articles where those sections came from.
6. Experiments and Results: this chapter contains results showing the performance of different parts of the project. In Section 6.1 we remind the metrics that we use to evaluate our system. Section 6.2 contains the results of our reproduction of the initial algorithm, in Section 6.3 we show the increase in performance due to our improvement of the initial algorithm. We present in Section 6.4 the performance of the algorithm after applying our semantic similar section filtering method. Finally we show in Section 6.5 the performance of our section ordering method.
7. Prototype: in Chapter 7 we present in detail the developed prototype, we also provide a link to download it.
8. Conclusion and future work: Chapter 8 concludes this thesis and proposes options which could be worth to explore to improve specific parts of this project.

2 Background

In this chapter, we firstly present the initial paper and its Wikipedia section recommendation algorithm on which this project is based, then we explain this paper's limitations that we have chosen to resolve.

Then, we present the methods that we use to detect semantically similar sections and some graph theory concepts which we use to resolve the first limitation of the initial paper; *i.e.,* filtering out redundant sections produced by the initial paper's section recommendation algorithm.

Finally we talk about an idea of how to order sections of the recommendation list and how to evaluate our ordered recommended sections by comparing them with the ground truth from Wikipedia's articles. This was used to resolve the second limitation of the initial paper; *i.e.,* ordering the recommended sections in such a way that could be found in real Wikipedia articles.

2.1 Recommendation algorithm for Wikipedia sections

This project is based on the reproduction and resolution of the limitations of a paper which proposed a section recommendation algorithm that leveraged Wikipedia's category network[1]. Therefore, this paper has to be firstly introduced in order to understand the context of this project.

On Wikipedia, each article belongs to one or more categories, those categories are linked together with hierarchical relationships, each category can have several subcategories, *e.g.,* “1893 births” is a subcategory of “1890s births”.

The main idea of the recommendation algorithm is to use this category graph to recommend sections for a given article by counting the most common sections of other articles which belong to the same categories.

In order to leverage the potential of this category graph, an article is propagated in the category graph from his categories to the parent categories; *i.e.,* parent categories are enriched with sections from articles which are in their child categories.

The issue is that the Wikipedia category network does not contain only ontological relationships; *i.e.,* ”is a” relationships between categories (as shown on Figure 2.1), therefore in order to avoid propagating noisy sections in the category network, the edges of the network need to be somehow pruned in order to keep only ontological relationships before propagating articles.

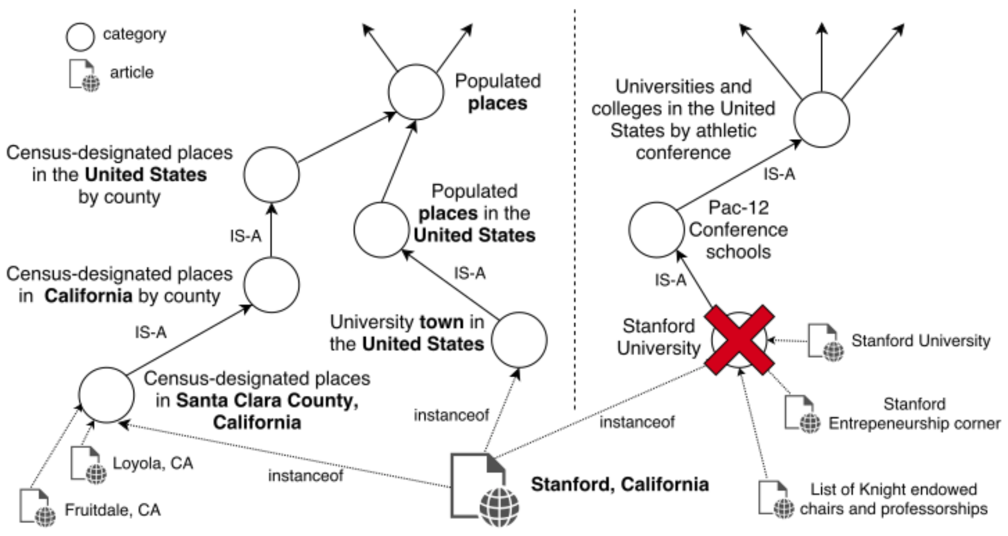

category
article
Populated places
Census-designated places in the United States by county
IS-A
Census-designated places in California by county
IS-A
Census-designated places in Santa Clara County, California
Loyola, CA
Fruitdale, CA
instanceof
Populated places in the United States
IS-A
University town in the United States
instanceof
Stanford, California
Universities and colleges in the United States by athletic conference
IS-A
Pac-12 Conference schools
IS-A
Stanford University
instanceof
Stanford University
Stanford Entrepeneurship corner
List of Knight endowed chairs and professorships

Figure 2.1: Example where a pruning would be required in order to be able to safely propagate articles for recommendations in the Wikipedia category network

Source: [1]

On the left side of the figure, every subcategory of “Populated places” contains articles about cities which means that all those articles could potentially have the same structure and sections. We could safely use information from any of those articles to recommend sections to articles which belong to the ”Populated places” category. On the right side of the figure, the category “Stanford University” contains articles such as “Stanford, California”, “Stanford University” and “List of Knight endowed chairs and professorships”, all those articles have different structures and sections. We cannot use information from the “Stanford, California” article to recommend sections for an article in the ”Pac-12 Conference schools” category.

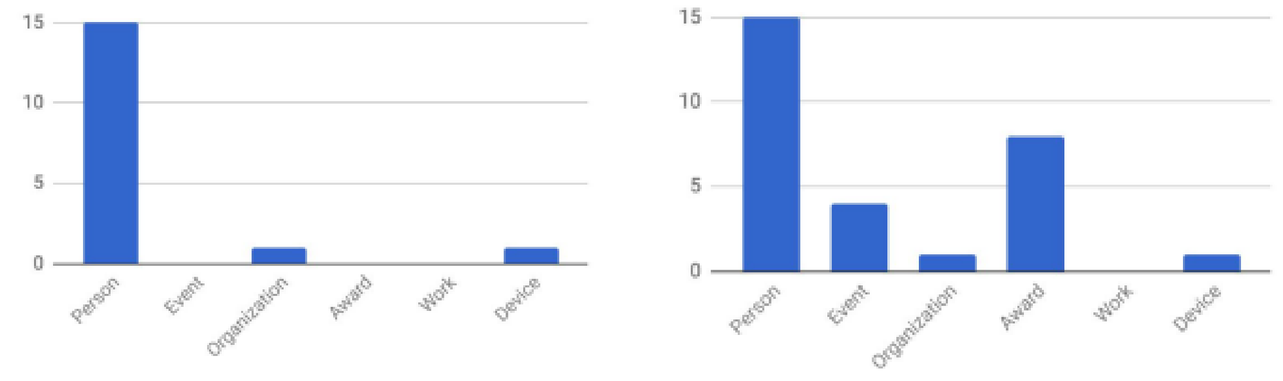
To know which edges of the network have to be pruned, we need a way to decide if a relationship between two categories is ontological, *i.e.,* we need to determine if there is an “is a” relation between two categories, which is not guaranteed by default. The relationships which are not ontological should be pruned.

Because the relationships between categories in Wikipedia’s category graph are not guaranteed to be ontological, we need an external source to know if articles from a subcategory can be safely propagated to its parent categories without violating the “is a” relation. For example: DBpedia [2] which lists entities that are often present as articles in Wikipedia, has a “type” attribute for those entities (*e.g.,* “Location”, “Person”, “Organization”). The idea is that if a child category and its parent category contain mainly articles of the same type, there is an ”is a” relationship between the child and parent category. *e.g.,* on the left side of Figure 2.1, all articles have the DBpedia type ”Location”, therefore relationships between their categories can be assumed as ”is a” relations.

The pruning algorithm implemented in a java program provided by the authors of the paper works as follows:

- Starting from the leaves of the category graph, an article type histogram (*i.e.,* a histogram counting how many articles of which type are present) for each category is made.
- Based on this histogram, the gini coefficient is computed
  - A high gini coefficient means that articles have mostly the same type, as shown on the left side of Figure 2.2.
  - A low gini coefficient means that there are articles of different types with no type that is enough predominant, as shown on the right side of Figure 2.2.
- If the gini coefficient of the type histogram of a given category is sufficiently high (defined by a minimal threshold), this category is marked as ontologically pure and all the articles from this category are propagated to the parent category.
- The algorithm further computes the gini coefficient of the parent category based on articles belonging to the parent category and the articles propagated from its child categories at the previous step.

- If the gini coefficient is lower than the threshold for a given category, this category is removed from the category network.



(a) Article type histogram of a category that is ontologically pure (b) Article type histogram of a category that is not ontologically pure

Figure 2.2: Example of category's article type histograms

Source: `https://upload.wikimedia.org/wikipedia/commons/c/cb/Using_Wikipedia_categories_for_research.pdf`

With this method, the section counts of categories (which will be used to recommend sections to articles from a given category) are enriched with the articles coming from their child categories, if the article type histogram of their child categories has a high enough gini coefficient.

The required input for the Java implementation of the algorithm consists of 3 files: an edge list describing the category network with a child - parent relation for each edge (it must be a directed acyclic graph), a list of categories to which each article belongs, and a list of article-type tuples.

This algorithm outputs for each category a list of articles that were either initially in this category, or that were propagated from sub-categories that were pure enough. Categories that were not pure enough are not present in the output of the program.

With the output of the category network pruning algorithm, the next step is to compute the section occurrence count for each category, by taking all the articles which were propagated by the pruning algorithm. The section occurrences are normalized by the number of articles in the category, this gives us the probability of occurrence of a given section in a given category. For each category, only the 30 most frequent sections are taken into account.

Additionally, the authors have removed a list of 13 sections that are considered as not useful information for recommendation and that are among the most frequent ones in the dataset (See also, Footnotes, Links, External links, References, Further reading, References and sources, Sources, References and notes, External sources, Notes, Bibliography, Notes and references). Sections that appeared only once in the whole dataset were removed for the same reason (*e.g.,* a section "Education of Napoleon Bonaparte" would be useless for recommendation because it appears only in one article in the whole dataset). Articles marked as stubs (*i.e.,* incomplete or of poor quality) were ignored to avoid using data explicitly marked as poor quality.

In order to recommend sections for a given article, the algorithm performs an unweighted sum of all the normalized section occurrence counts from each category to which a given article belongs and returns the k most common sections.

Only top-level sections are considered in this algorithm, *i.e.,* subsections are ignored.

The algorithm is summarized on Figure 2.3.

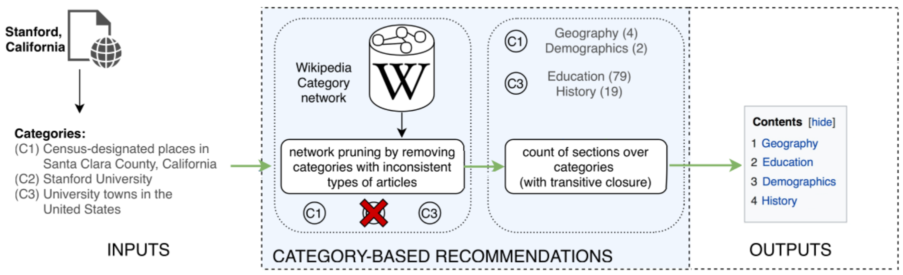

Stanford, California
Categories:
(C1) Census-designated places in Santa Clara County, California
(C2) Stanford University
(C3) University towns in the United States
INPUTS
Wikipedia Category network
network pruning by removing categories with inconsistent types of articles
C1
C3
Geography (4)
Demographics (2)
Education (79)
History (19)
count of sections over categories (with transitive closure)
CATEGORY-BASED RECOMMENDATIONS
Contents [hide]
1 Geography
2 Education
3 Demographics
4 History
OUTPUTS

Figure 2.3: Overview of the recommendation algorithm

Source: modified figure from [1]

The ”transitive closure” means that the count of sections in a given category are enriched with sections of its child categories.
The section counts need to be normalized before being ordered by descending normalized section count in the recommendation list. C1 has 8 articles and C3 has 184 articles. Therefore the normalized section counts are as follows: Geography $4/8 = 0.5$, Demographics $2/8 = 0.25$, Education $79/284 = 0.28$, History $19/284 = 0.07$

For the evaluation of the recommendation algorithm, the authors of the paper generated for each article of the test set 20 lists with size k=1 to 20 (if the list has a size of 1, it means that only the most common section given by the algorithm is recommended, if the list has a size of 2 it means that the 2 most common sections are recommended, *etc.*) because 20 seemed a reasonable maximum size for recommending a list of sections for a user.

The algorithm was evaluated by computing the precision and recall for each list size k, averaged over the whole test set.

The authors of the paper point to these two limitations of their method among others:

- Sometimes, semantically related sections are in the same recommendation list, *e.g.,* “Works” and “Discography” in a recommendation list for an article about a musician. The quality of the recommendations could be improved by grouping those related sections together and recommending only one of them.
- The recommendation list is not ordered, *i.e.,* “Conclusion” could be in the beginning of the recommendation list and “Introduction” at the end. This would be the case if the section “Conclusion” appears more frequently than “Introduction” for a given category. A future improvement could be to find a way to order the recommendation list.

The goal of this thesis is to resolve these two limitations.

2.2 Semantic similarity

We needed to be able to detect semantic similar sections in order to resolve the first limitation of the paper described at the end of Section 2.1, *i.e.,* detecting and filtering out semantically redundant sections in the recommended sections. Therefore we needed to firstly review existing solutions in order to decide which one to use in our project.

BERT

All the other methods presented in this section are based on BERT, therefore we have to introduce it first.

BERT stands for ”Bidirectional Encoder Representations from Transformers” [3] and was developed by Google.

The encoder used in the model is bidirectional in the sense that it takes into account the surroundings *i.e.,* the context around the words by taking as input whole sentences during the pre-training phase. This allowed word embeddings (*i.e.,* representation of words in vector space) which are learnt from different contexts. To achieve this, a new language model was designed, called ”MLM” (masked language model), with the goal to predict the initial values of words that were masked beforehand, based on the context of the masked words. With this pre-training task, the model was able to learn relationships between words.

BERT was also pre-trained with a ”next sentence prediction” task, which allowed it to learn relationships between sentences. To achieve this, the task consisted in predicting if in a pair of sentences A and B, B was either originally the next sentence after A or if it was a randomly picked sentence from the corpus.

BERT was pre-trained with a large corpus from Wikipedia articles and books in a self-supervised way (without explicit labeling by humans for a specific NLP task) in order to acquire representations of the human language.

The advantage of BERT is that the architecture and pre-trained model can be adapted for a wide range of NLP tasks by adding output layers to the network, allowing it to be used for Transfer Learning.

RoBERTa

RoBERTa (Robustly Optimized BERT Pretraining Approach) [4] was developed by Facebook AI and was released in July 2019.

After replicating the BERT method, the authors from Facebook AI have removed the next sentence prediction task and focused on the masked language modeling objective. This allowed them to improve the performance of the model for this task compared to the original BERT. Additionally, they have trained the model with an order of magnitude more unannotated data from news articles for a longer amount of time while using larger mini batches and learning rates.

With those changes, the achieved performance on GLUE, RACE and SQuAD benchmarks was higher than the performance of the original BERT model.

Sentence-BERT

Sentence-BERT also called SBERT [5] is a method that was released in August 2019 by researchers from the UKPLab at the Darmstadt University .

The problem of computing the semantic similarity between sentences using BERT is that BERT takes as input a pair of sentences and outputs a similarity score between those sentences. Therefore in order to compute the similarity score between a set of sentences, all possible sentence pairs have to be fed into the model which caused a massive computational overhead (*i.e.,* a given sentence has to be fed multiple times into the model) and made BERT unsuitable for semantic similarity search.

SBERT uses a modified architecture of BERT to produce a fixed-size embedding for each sentence, those sentence embeddings can afterwards be compared *e.g.,* by measuring the cosine similarity.

Thanks to their solution, the sentence embeddings can be generated once for each sentence and then be compared with each other instead of generating embedding before each comparison. This reduced the effort for finding the most similar pair of sentences among 10’000 sentences from 65 hours with BERT/RoBERTa to about 5 seconds with SBERT.

DistilBERT

The purpose of DistilBERT (which was introduced with the paper ”DistilBERT, a distilled version of BERT: smaller, faster, cheaper and lighter” [6] released in October 2019 by researchers from Hugging Face) was to make the usage of BERT-based models faster by using knowledge distillation (*i.e.,* a compression technique in which a compact model - the student - is trained to reproduce the behavior of a larger model - the teacher) during the pre-training phase in order to reduce the size of the BERT model by 40% by keeping 97% of its language understanding capabilities and being 60% faster.

Implementation

The authors of the SBERT method have developed a python module called sentence_transformers[1] which allows transforming sentences into their respective sentence embeddings in order to be able to compare them in the vector space using cosine similarity.

The sentence_transformers module proposes a method which finds a list of most similar sentence pairs inside a large corpus (*i.e.,* more than 10’000 sentences). This method is called paraphrase mining[2] which is well suited

[1]https://www.sbert.net

[2]https://www.sbert.net/examples/applications/paraphrase-mining/README.html

for our case where we need to compare large amounts of sections among each other.

The sentence_transformers module allows using various pre-trained models, the most interesting one among those suited for paraphrase search in terms of efficiency-performance[3] is paraphrase-distilroberta-base-v2[4].

Paraphrase-distilroberta-base-v2 is based on distilroberta-base[5] which is a distilled version of the RoBERTa model. Distilroberta-base was firstly pre-trained on OpenWebTextCorpus [7] with the same training procedure as DistilBERT with the goal to distillate the larger RoBERTa model.

Then paraphrase-distilroberta-base-v2 was trained with various datasets[6] containing paraphrases such as pairs of questions marked as duplicate from Quora and Stackexchange, the same sentence once from the english Wikipedia and once from the "simple english" Wikipedia *etc.*

Paraphrase-distilroberta-base-v2 was also trained on a NLI (Natural Language Inference) dataset which consists of sentence pairs which are labeled as entail (synonym), contradict (antonym) and neutral. The dataset used for training in this case was the The Stanford Natural Language Inference (SNLI) Corpus [8]. Training the model on a NLI dataset produces sentence embeddings suitable for semantic search [9].

Finally paraphrase-distilroberta-base-v2 was fine-tuned for the STS benchmark [10], which generated sentence embeddings that are especially suitable to measure the semantic similarity between sentence pairs.

2.3 Graph theory

In this section we will introduce some concepts from graph theory because they were used in the semantically similar section detection part of the project.

Connected components

A connected component (or simply "component"), is a set of vertices in an undirected graph where each vertex can reach every other vertex by a path. If there is no path between two vertices, they are not in the same connected component. Figure 2.4 contains an example of connected components.

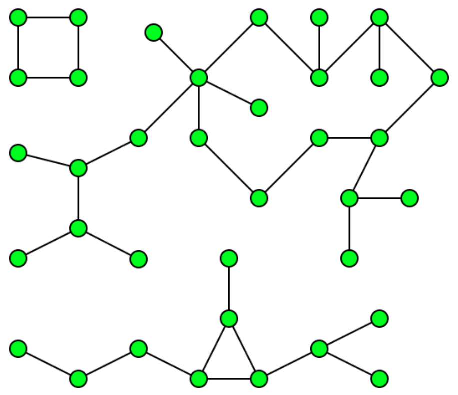


Figure 2.4: A graph with 3 connected components

Source: https://en.wikipedia.org/wiki/File:Pseudoforest.svg

Louvain method for community detection

Community detection is a way of grouping together nodes in a network. The goal is to detect sets of nodes (*i.e.,* communities) where the nodes are densely connected.

In our project we use the Louvain algorithm for community detection [11].

The idea of the Louvain method is to cluster a network into communities by maximizing the modularity of the network.

[3]https://www.sbert.net/_static/html/models_en_sentence_embeddings.html toggle "All models"

[4]https://huggingface.co/sentence-transformers/paraphrase-distilroberta-base-v2

[5]https://huggingface.co/distilroberta-base

[6]https://www.sbert.net/examples/training/paraphrases/README.html

The modularity is a measure of how densely the nodes inside a community are connected with each other compared to connections with nodes which are outside of the community, by summing the difference between the actual number of edges between each pair of nodes and the expected number of edges between each pair of nodes.

The modularity of a network is computed with the following formula:

$$Q = \frac{1}{(2m)} \sum_{vw} \left[A_{vw} - \frac{k_v k_w}{(2m)} \right] \delta(c_v, c_w) \tag{2.1}$$

Where:

m is the number of edges in the network

v, w represents any pair of different nodes in the network

k_x is the node degree of node x (*i.e.,* number of edges of a node)

$\delta(c_v, c_w)$ (δ is the kronecker delta symbol) is 1 if node v and w are in the same community, 0 otherwise

A_{vw} is the element in the graph adjacency matrix which is at the v-th row and w-th column, *i.e.,* 1 if v and w are connected with an edge, 0 otherwise (if the edges have weights, the value of the weight can be used instead of 1, in this case the weight represents the intensity of the connection between two nodes, *i.e.,* nodes which share edges with high weights will more likely be put in the same community)

The algorithm starts by putting each node of the network in a different community. Then the following two phases are repeated in passes until there are no more changes to be made and a maximum of modularity is achieved:

- The first phase is called modularity optimization. At each iteration of this phase, nodes are re-assigned to neighboring communities in a way that maximizes the modularity gain (difference between the modularity before and after the re-assignment), the algorithm iterates this phase until the modularity gain is less than a predefined threshold (*i.e.,* until convergence of the algorithm).
- The second phase is called community aggregation. The algorithm builds a new network where the nodes are communities detected in the first phase. Links between nodes inside the same community are now represented as self-loops. This new network is then used in the first phase of the next pass.

Those steps are visualized on Figure 2.5.

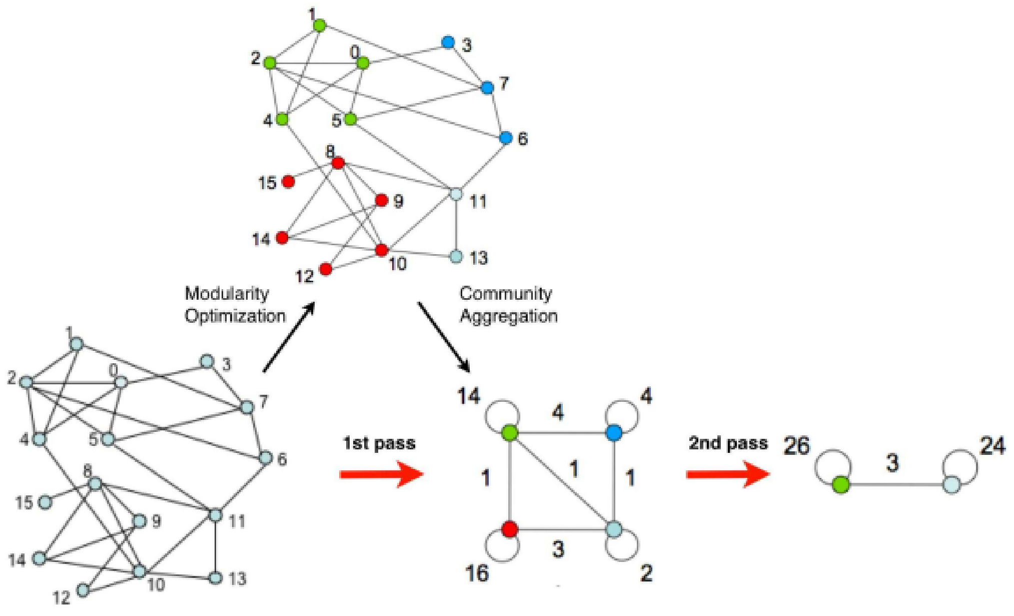


Figure 2.5: Visualization of the steps of the Louvain algorithm

Source: [11]

The expected resulting behavior is that the nodes are divided into communities in a way that the modularity of the network is maximized.

To detect communities, we used in this project the python-louvain module [12], compatible with the networkx module [13] which we used to handle graphs.

2.4 Section ordering

We present the initial idea of how to represent the order of sections in a document and the metric that we used to measure the difference in terms of ordering between two lists of elements (in our case, comparing the order of recommended sections with the order of sections from an existing Wikipedia article).

Idea of representing the order of the sections

The idea was to use a similar approach as described in [14], where the goal was to build a summary for multiple documents in which the sections of the summary had the same order as in the original documents from where the sections came from. To do that, each section had an order value that represents the relative position of a section in the table of contents, this value ranges from 0 to 1 (0 means that the section was originally in the beginning of a document, and 1 means that the section was at the end of a document) and then the sections of the summary were ordered by increasing order value.

But in our case, the sections of the recommendation list come from multiple articles in which a given section is not necessarily at the same position in the table of contents and the number of sections in each article is different. This means that we had to find a way to aggregate different order values coming from multiple articles for each section.

For example, a given section could be one time in the beginning of the table of contents which has a total number of 10 section, and another time it could be the only section in an article. If we use only the relative position where the section starts, in both cases its order value would be 0, which does not take in account the fact that in one case this section covered 1/10th of the beginning of an article and in the other case this section covered the entire article.

Kendall's tau ordering metric

We needed a metric in order to quantify how much our method improves the ordering of the recommendations by being able to compare orderings between the recommendations and the ground truth. A paper proposed a metric called "Kendall's tau" to automatically evaluate information ordering [15].

The Kendall rank correlation coefficient (also called Kendall's tau coefficient) measures how much the ordering between two lists differs [16]. This metric is applicable only if all the items in both lists are the same and if both lists have the same size, therefore only the order of items which are in both lists can be compared. Kendall's tau has a value of 1 if both lists have exactly the same order, a value of -1 if one list is exactly the inverse order as the other. Therefore for the evaluation with this metric, a bigger value is better.

We used in this project scipy's [17] implementation of kendall's tau, which is the tau-b variant that takes in account ties in the compared lists [18] (if some elements have the same rank in the same list, *e.g.,* if we order a list of integers where there is more than one occurrence of the same integer).

Kendall's tau-b is defined in scipy's implementation as follows[7] for comparing two ranking lists x and y:

$$\frac{P-Q}{\sqrt{(P+Q+T)(P+Q+U)}} \tag{2.2}$$

Where:

P is the number of concordant pairs

Q is the number of discordant pairs

T is the number of ties only in x

[7]https://docs.scipy.org/doc/scipy/reference/generated/scipy.stats.kendalltau.html

U is the number of ties only in y

But in our case, each section appears only once in each list which means that T and U will alway be 0, therefore we can rewrite the formula as:

$$\frac{P - Q}{P + Q} \tag{2.3}$$

The original kendall's tau which doesn't take in account ties is not implemented in scipy because in absence of ties the tau-b variant can be reduced to the original kendall's tau.

An example to illustrate how to count the concordant and discordant pairs is shown on Table 2.1.

x	**y**	**ranking of x**	**ranking of y**	**Concordant (P)**	**Discordant (Q)**
A	A	1	1	3	0
B	D	2	4	0	2
C	B	3	2	1	0
D	C	4	3	0	0

Table 2.1: Example to illustrate how to count the concordant and discordant pairs

The original list is x, therefore the elements ABCD define the ranking 1,2,3,4. We want to compare the ranking of y with x. y has the elements ADBC, therefore the elements of y have the ranks 1,4,2,3 compared to x.

Each row in the concordant column contains the number of elements of y below a given element that have a higher rank (*e.g.,* the first row has the value of 3 because the elements D,B,C have a higher rank than A in both lists), and the discordant column contains the number of elements below a given element of y which have a lower rank (*e.g.,* the second row has a value of 2 because the elements B and C have a lower rank than D in x).

Then, to get the total number of concordant pairs we sum up the elements of the Concordant column: $P = 3 + 0 + 1 + 0 = 4$

The same for the Discordant column: $Q = 0 + 2 + 0 + 0 = 2$

Finally, we can compute the kendall's tau value: $(P - Q)/(P + Q) = (4 - 2)/(4 + 2) = 2/6 = 1/3$

In the case where the ordering is exactly the same in both lists, $Q = 0$ because all pairs of items are concordant and no pair is discordant, therefore kendall's tau value is $P/P = 1$.

And in the case where the ordering of y is exactly the reverse of x, $P = 0$ because all pairs of items are discordant and no pair is concordant, therefore kendall's tau value is $-Q/Q = -1$

3 Reproduction of existing method and improvement

The first part of the project is to reproduce the existing Wikipedia section recommendation method based on its description in the initial paper and to check if the obtained results match those provided by the authors of the paper.

We first describe how to retrieve Wikipedia's data that is needed for the recommendation algorithm.

Then we present the problems that occurred while we were trying to achieve the same performance as the authors of the paper and how we solved them.

Finally we propose a method to increase the performance of the algorithm.

3.1 Data collection

The data published by the authors was retrieved in September 2017, therefore it was decided to retrieve up-to-date data (March 2021) from the Wikipedia dumps in order to build the dataset for this project.

The needed data for the section recommendation algorithm was the following:

- Wikipedia articles with their respective first-level sections (*i.e.,* subsections are not taken into account)
- Categories to which each article belongs
- The category network, with child-parent relationships between categories
- Corresponding DBpedia type for each article

This data can be retrieved with 4 files:

- A SQL file containing information about Wikipedia pages[1]
- A SQL file containing category links[2]
- A XML file containing the content of Wikipedia pages[3]
- A TTL file (*i.e.,* turtle syntax used to store RDF data, *i.e.,* a way of describing data) containing types for DBpedia entities[4]

[1] `https://dumps.wikimedia.org/enwiki/latest/enwiki-latest-page.sql.gz`
[2] `https://dumps.wikimedia.org/enwiki/latest/enwiki-latest-categorylinks.sql.gz`
[3] `https://dumps.wikimedia.org/enwiki/latest/enwiki-latest-pages-articles.xml.bz2`
[4] `https://databus.dbpedia.org/dbpedia/mappings/instance-types/` download the ”en, specific” file

First, the ids and titles of the Wikipedia articles needed to be extracted from the Wikipedia page SQL table[5]. This page table has a "page_namespace" field, the Wikipedia articles are in the namespace 0 (there are multiple namespaces, *e.g.,* for Wikipedia user pages, for discussions about articles, for categories, *etc.*[6]). This table has also a "page_is_redirect" field which should be equal to 0 in order to avoid to retrieve pages which are redirects (*e.g.,* there is a page "UK" which redirects to the actual Wikipedia article "United Kingdom"). The article ids are stored in the field "page_id" and the article titles in the field "page_title".

Then, the category ids and titles were extracted, again from the page SQL table. The category pages are in the Wikipedia namespace 14, the method to extract them was the same as for the articles.

The category network is stored in the categorylinks SQL table [7]. The field "cl_from" contains the page id (either of an article or a category) of the page that belongs to a given category. The field "cl_to" contains the name of the category to which a given page belongs. With these two fields, we were able to extract the categories to which each article belongs, and the child-parent category relationships for each category. Moreover, at this step we could detect articles which are explicitly tagged as poor quality articles (*i.e.,* marked as "stubs", which are articles deemed too short to provide encyclopedic coverage of a subject [8]), by filtering out articles which belonged to at least one category that has the word "stubs" in it (*e.g.,* "British journalist stubs", "Tasmania geography stubs" *etc.*).

Once the category network was built, we had to transform in intro a direct acyclic graph with a modified version[9] of a java program provided by the authors of the paper[10] which implemented [19].

Additionally, after retrieving the child-parent relationships for the categories, we were able to retain only categories that were children of the category "Main Topic Classification" to avoid keeping categories used only for internal Wikipedia maintenance which were guaranteed to be not ontological (*e.g.,* "Articles with French-language external links"), as precised in the original paper.

After filtering articles that would not be used for the recommendation algorithm (*i.e.,* stubs), we could then retrieve the top-level sections for each remaining article. The pages' content is stored in the "pages-articles" XML file, which is compressed with bz2. The compressed file takes about 18 GB of space but its uncompressed version expands to over 78 GB, therefore we had to decompress and process the file's content line by line with a tool called "wikiextractor" [20] (we used a modified version where we fixed a bug that occurred while extracting articles containing lists and outputting in HTML format[11]).

Wikiextractor allowed us to extract the content of Wikipedia articles in HTML format instead of Wikipedia's markdown. From the HTML format we could easily get the top-level sections by retrieving h2 HTML tags. Additionally in order to have only sections easily readable for the end user, we removed from those section titles potential HTML tags (*e.g.,* "`Track listing <small>(with Kottke's notes)</small>`" which became "Track listing (with Kottke's notes)") and Wikipedia templates (*e.g.,* "`{{flagicon|Canada}} Canada`" which became "Canada").

Once we had the sections for each article, we counted the occurrence of each section across the dataset and removed sections that appeared only once. Additionally, sections that were considered as useless for recommendation ("References", "External links", "See also", "Notes", "Further reading", "Bibliography", "Sources", "Footnotes", "Notes and references", "References and notes", "External sources", "Links","References and sources") were also removed, as described in the original paper.

Finally, in order to retrieve DBpedia types for Wikipedia articles, we had to parse the TTL file which contained the specific types for DBpedia entities. If a DBpedia entity exists for a given Wikipedia article, it has exactly the same title as the a Wikipedia article which it represents[12] (with underscores replacing the spaces). The types in this file are specific in the sense that each DBpedia entity is defined by the DBpedia type that is the most specific possible, *i.e.,* those types are the leaves of the type hierarchy[13]. But we needed to assign each article to one type among the 55 top-level types to be consistent with the paper's authors (those top-level types

[5]`https://www.mediawiki.org/wiki/Manual:Page_table`

[6]`https://www.mediawiki.org/wiki/Manual:Namespace`

[7]`https://www.mediawiki.org/wiki/Manual:Categorylinks_table`

[8]`https://en.wikipedia.org/wiki/Wikipedia:Stub`

[9]`https://github.com/SergiyGolov/GraphCyclesRemoval` we corrected a bug which prevented the program to produce an output file

[10]`https://github.com/epfl-dlab/GraphCyclesRemoval`

[11]`https://github.com/SergiyGolov/wikiextractor`

[12]`http://downloads.dbpedia.org/wiki-archive/data-set-38.html` section 3: Denoting or Naming "things"

[13]`https://databus.dbpedia.org/dbpedia/mappings/instance-types/2021.09.01`

are those that are direct children of the ”Thing” or ”Agent” types in the hierarchy). Therefore, we needed to build a mapping from specific types to their equivalent top-level types. This data was retrieved with the DBpedia API which had a SPARQL endpoint[14], by retrieving the entities having a ”rdfs:subClassOf” attribute. This attribute was present in DBpedia entities which described DBpedia types, and it contained the name of the DBpedia type that was the parent in the type hierarchy of a given DBpedia type. After retrieving the child-parent relations for all DBpedia types, we could use it to translate the specific types from the TTL file to the top-level types needed for the recommendation algorithm.

Once this dataset was built, we were able to use it as input to the Java program provided by the authors of the paper[15]. The program outputted the list of categories that were marked as ontologically pure, as well as articles belonging to the categories which are descendants of those ontologically pure categories.

3.2 Achieving the same results

After retrieving the data from 2021 and applying the initial paper's method on it, the results didn't match those from the paper (we had precision@1 at 0.55 and recall@20 at 0.7 instead of precision@1 at 0.57 and recall@20 at 0.8).

We decided to reproduce the result with the data from 2017 provided by the authors of the paper in order to see if our implementation of their algorithm was correct.

The paper's authors provided the following files useful for recommendation:

- Categories to which each article belongs.[16]
- The output of the category pruning algorithm which contained ontologically pure categories with the articles belonging to those categories as well as articles belonging to the children of those categories.[17]
- The normalized section counts for those ontologically pure categories.[18]

The authors of the paper have not provided data on which article contained which sections. Therefore we had to use the provided section counts for categories that contained the normalized section counts of all articles from each category. We used those category section counts to recommend sections for the same test set as the one used in the provided results. This allowed us to achieve the same results as the paper's authors, which meant that our reproduction of their algorithm is correct up to the computation of the category section counts.

Then, we decided to parse a Wikipedia dump from 2017[19] ourselves in order to retrieve the article's sections. We computed the category section counts for the same test set and compared our results with theirs (in particular for the smallest categories, which were easily comparable). We noticed that our category section counts contained fewer section than theirs. We found that the reason for this difference in performance was that the authors of the paper have included the test set into the category section counts used to predict the sections for articles from the test set. This explained the high values in their results (particularly in terms of recall@20).

This time, we got precision@1 at 0.53 and recall@20 at 0.68.

Here is an example from the results provided by the authors of the paper which demonstrates how the only way that the recommendation list contained the correct sections would be to include the ground truth in the training (the sections in bold are only contained in the article of the test set):

Recommendations for article id 2429464 (Babur (cruise missile))[20], with corresponding normalized section counts:

[14]`https://dbpedia.org/sparql/?help=intro`

[15]`https://github.com/epfl-dlab/WCNPruning`

[16]`https://figshare.com/articles/dataset/WCNPruning_input_set/6157445` ”`article_categories_sept17.tsv`” file

[17]`https://figshare.com/articles/dataset/WCNPruning_input_set/6157445` ”`articles_scores_gini_coefficient0.966.json`” file

[18]`https://figshare.com/articles/dataset/Structuring_Wikipedia_Articles_with_Section_Recommendations/6157583` ”`recs_probabilistic_by_category_top30.json`” file

[19]`http://itorrents.org/torrent/D567CE8E2EC4792A99197FB61DEAEBD70ADD97C0.torrent`

[20]`https://figshare.com/articles/dataset/Structuring_Wikipedia_Articles_with_Section_Recommendations/6157583` ”`recs_probabilistic_by_article.json`” file, line 114885

1. Operational history: 1.0
2. **Origin: 0.5**
3. **Features: 0.5**
4. **Guidance: 0.5**
5. Development and design: 0.5
6. **Design: 0.5**
7. Variants: 0.5

Normalized section counts for the category ”Cruise missiles of Pakistan”[21]:

1. Operational history: 1.0
2. Variants: 0.5
3. **Origin: 0.5**
4. **Features: 0.5**
5. **Guidance: 0.5**
6. **Design: 0.5**
7. Development and design: 0.5

Here are the only two articles in category ”Cruise missiles of Pakistan”[22] with their sections:

- article id 31637697 (Hatf-VIII (Ra'ad))[23]
    1. Variants
    2. Development and design
    3. Operational history
- article id 2429464 (Babur (cruise missile))[24]
    1. **Origin**
    2. **Design**
    3. **Guidance**
    4. **Features**
    5. Operational history

By looking at the section counts of the recommendations, we can see that they are exactly the same as for the category-section counts for the category ”Cruise missiles of Pakistan”, and the fact that the only section count values are 1 and 0.5 means that there are only two articles in this category, one of which is from the test set.

Therefore, the only way to predict correctly those sections was to include in the category-section counts the article for which the predictions are made. If the test set was excluded from the category section counts, the recommendations for this article should be:

1. Operational history: 1
2. Development and design: 1
3. Variants: 1

[21] `https://figshare.com/articles/dataset/Structuring_Wikipedia_Articles_with_Section_Recommendations/6157583` ”`recs_probabilistic_by_category_top30.json`” file, line 11185

[22] `https://figshare.com/articles/dataset/WCNPruning_input_set/6157445` ”`articles_scores_gini_coefficient0.966.json`” file, line 404115

[23] `https://en.wikipedia.org/w/index.php?title=Hatf-VIII_(Ra%27ad)&oldid=774362743`

[24] `https://en.wikipedia.org/w/index.php?title=Babur_(cruise_missile)&oldid=797024462`

The normalized category sections counts became all 1 because there was only one article in the category left after removing the article which was in the test set.

The precision@1 could remain the same (depending on the way that ties in sorted lists are handled, because all 3 recommended sections have the same normalized section count), but the recall@20 would be 0.2 instead of 1 (only "Operational history" was in both articles of this category out of 5 sections for the article for which the recommendations were made).

Then we decided to include articles from the test set in our train set and we achieved a slightly better performance as theirs (*i.e.,* precision@1 at 0.59 and recall@20 at 0.84). This time the section counts for the category "Cruise missiles of Pakistan" were exactly the same as those provided by the paper's authors[25].

For the next parts of the project, instead of using the results published by the paper's authors as baseline, we used the reproduction with 2021 data without mixing the test and train set (*i.e.,* precision@1 at 0.55 and recall@20 at 0.7).

3.3 Improvement of the method

We analyzed the source code of the WCNPruning program[26] which was used to filter out non ontological categories and to propagate articles in the category hierarchy (*i.e.,* assigning articles to the categories which are higher in the category hierarchy, as long as those categories are ontologically pure enough). We found out that articles which had no DBpedia type were not included in the category's type histogram on which the gini coefficient was computed; *i.e.,* articles with no defined DBpedia type were not considered as belonging to a category for its gini coefficient computation. But those articles were still propagated to the parent category if the type histogram was pure enough among the remaining articles which had a DBpedia type.

In terms of detection of ontological categories this doesn't matter, but for the section counting for categories it does. Some articles with no DBpedia type add noise to the category section counts after being propagated from child categories because we have no guarantee that they would keep the category ontological. For example, we could have categories where half of the articles are of type "Person" but the other half of articles have no defined DBpedia type, but if their type would be defined it could be *e.g.,* "Location" which would mean that the sections from those articles with no defined DBpedia type would not be relevant for recommending sections for articles of type "Person".

Moreover, 48% of articles had no DBpedia type in the dataset used by the authors, whereas in the 2021 dataset 42% of articles had no DBpedia type. Their potential impact on adding noise in the category section counts was significant. The decrease of the proportion of articles which have no defined DBpedia type could be explained by the fact that in 4 years since the publication of the paper, people have contributed to the annotation of new DBpedia entities. This could explain that with Wikipedia data from 2021 the results were slightly better than with Wikipedia data from 2017 because less articles with unknown DBpedia types were included in the category section counts, which inducted less noise compared to the data from 2017.

Therefore, the adopted solution was simply to filter out and ignore articles having no DBpedia types *i.e.,* those articles were neither included in the test set nor in the train set. This could be done either before applying the pruning algorithm, or after. In both cases the categories detected as ontologically pure would be the same. The amount of articles in the dataset was reduced from 3'643'496 to 2'048'191 *i.e.,* 56% articles remained because they had a defined DBpedia type.

This solution resulted in an increase of precision@1 from 0.55 to 0.63 and an increase of recall@20 from 0.7 to 0.76.

Another way to improve the results was to change the gini coefficient threshold value above which a category is considered as ontologically pure. By empirical testing it was found that a gini coefficient threshold value of 0.985 instead of 0.966 increased the precision@1 from 0.55 to 0.58 and the recall@20 from 0.7 to 0.72. We modified the orignal WCNPruning program to allow specifying the gini coefficient threshold as a command line parameter[27]. Combined with the filtering of articles having no DBpedia types, the best achieved result was a

[25]See our published results, `results/epfl_reproduction-category_section_counts_with_test_set/recs_by_category_top30.json` file, line 386712

[26]`https://github.com/epfl-dlab/WCNPruning/blob/master/src/main/java/ch/epfl/dlab/wikipedia/wcnpruning/GraphPruner.java#L161`

[27]`https://github.com/SergiyGolov/WCNPruning`

precision@1 at 0.65 and a recall@20 at 0.77.

The various results presented throughout this chapter are shown on Table 3.1.

	Precision@1	**Recall@20**
Results provided by paper's authors	0.57	0.8
Reproduction with 2017 data, sections from test set in section counts	0.59	0.84
Reproduction with 2017 data	0.53	0.68
Reproduction with 2021 data	0.55	0.7
Improvement: increasing gini threshold	0.58	0.72
Improvement: filter out articles with unknown type	0.63	0.76
Improvement: filter unknown type articles and increase gini threshold	0.65	0.77

Table 3.1: Summary of the results from this chapter

For the next parts of the project, we used the dataset which contained only articles having a defined DBpedia type and categories which were ontologically pure with a gini coefficient threshold set at 0.985 *i.e.,* the configuration which gave us the best results.

4
Detect and filter semantically similar sections

This part of the project proposes a way to resolve the first cited limitation of the paper on which this thesis is based on. The goal was to find sections that are semantically similar in order to group those related sections and to recommend only one of them to the user.

Firstly, we describe the methodology of text extraction from Wikipedia's dump and some difficulties we encountered.

Then, we explain how and why we grouped categories together instead of using single categories in order to retrieve sets of sections that were likely to appear together in recommendation lists.

We measured the semantic similarity between section contents in the same context (*i.e.,* sections that were likely to appear together in recommendation lists) with the paraphrase-distilroberta-base-v2 model[1] from the sentence_transformers python module[2].

Finally once that we detected semantically similar sections in the dataset we could use them to filter recommendation lists in order to remove redundant sections.

4.1 Extracting section contents

By "section content" we refer to the text that is in a given Wikipedia article under a given section.

Section's contents were extracted at the same time as section titles, with the Wikiextractor tool. The purpose of using an external tool was its ability to convert wikipedia markdown in HTML, which made the task of parsing articles content easier.

For each article having a DBpedia type, we extracted the content of sections which remained after filtering (*i.e.,* sections that are not unique in the dataset and not in the list of 13 ignored sections mentioned in Section 2.1).

List elements encountered in the text were transformed into sentences by adding a point at the end of each list element.

Unfortunately the parsing by the tool was not perfect because some articles' markdown contained errors made by Wikipedia users (*e.g.,* on Figure 4.1), therefore we had to filter out remaining malformed Wikipedia markdown or http links from malformed references.

[1]`https://huggingface.co/sentence-transformers/paraphrase-distilroberta-base-v2`
[2]`https://www.sbert.net`

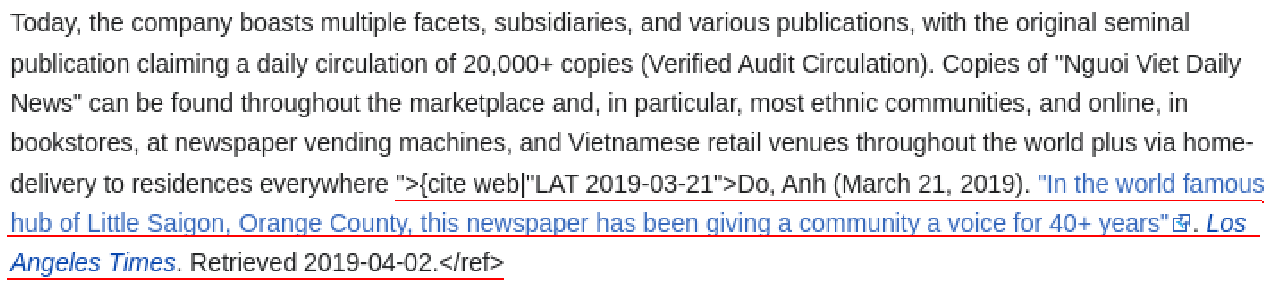

Today, the company boasts multiple facets, subsidiaries, and various publications, with the original seminal publication claiming a daily circulation of 20,000+ copies (Verified Audit Circulation). Copies of "Nguoi Viet Daily News" can be found throughout the marketplace and, in particular, most ethnic communities, and online, in bookstores, at newspaper vending machines, and Vietnamese retail venues throughout the world plus via home-delivery to residences everywhere ">{cite web|"LAT 2019-03-21">Do, Anh (March 21, 2019). "In the world famous hub of Little Saigon, Orange County, this newspaper has been giving a community a voice for 40+ years". *Los Angeles Times*. Retrieved 2019-04-02.</ref>

Figure 4.1: Example of a syntactic error which prevented the correct parsing of the articles

Source: `https://en.wikipedia.org/w/index.php?title=Nguoi_Viet_Daily_News&oldid=1004692631`

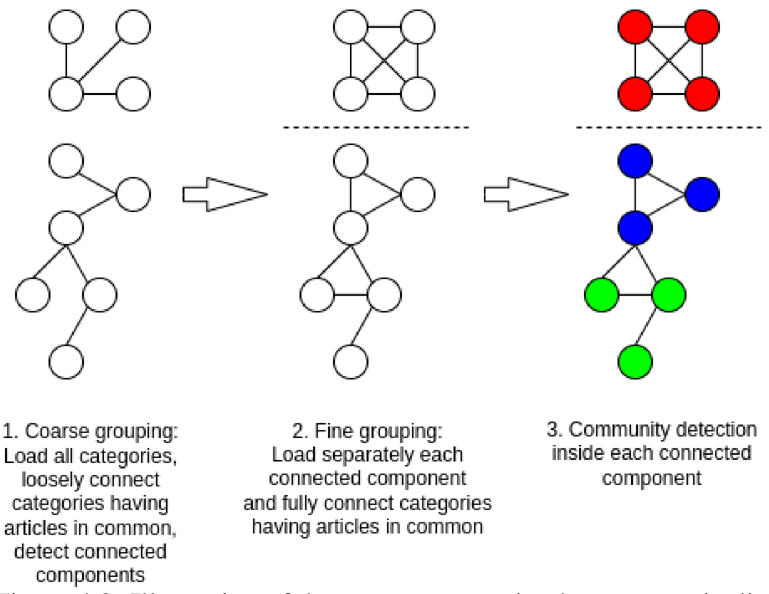
In order to normalize our corpus, we transformed diacritics to their ASCII counterpart, *e.g.,* ”é” became ”e”. Remaining characters which were not present in the English language (*e.g.,* cyrillic, arabic *etc.*.) were also removed because the model which we used to transform sentences to their embeddings was trained only on English corpora.

We have split the section's contents into sentences with the sent_tokenize method[3] from the nltk python module [21]. Then we counted the number of tokens in those sentences by tokenizing them with the model that we wanted to use for sentence to embeddings conversion. The idea was to obtain section contents formed of whole sentences where the sum of the token counts for each section content was under the token limit used by the model. Because the model ignores tokens after its limit (*i.e.,* 256 in our case), we wanted to avoid to cut sentences in the middle in order to preserve their structure and meaning. Therefore we grouped together the first sentences in each section's content, as long as the number of tokens didn't exceeded 256.

Taking only the beginning of each section's content (*i.e.,* the first 256 tokens) is a naive but acceptable way of conveying semantical information for the sections even if some information is lost as shown in [22].

Finally, we had for each section in each article their section content consisting of its whole first sentences which totalized up to 256 tokens.

4.2 Grouping categories by context


1. Coarse grouping: Load all categories, loosely connect categories having articles in common, detect connected components
2. Fine grouping: Load separately each connected component and fully connect categories having articles in common
3. Community detection inside each connected component
Figure 4.2: Illustration of the category grouping by context pipeline

The idea was to compare section contents from the same context in order to detect which sections were semantically similar. By section contents from the same context we mean section contents from articles

[3]`https://www.nltk.org/api/nltk.tokenize.html#nltk.tokenize.sent_tokenize`

belonging to categories which are likely to have common articles, *e.g.,* among articles belonging to categories ”Detroit red wings arenas” and ”Indoor ice hockey venues in Detroit”, with both categories having 4 articles each, 3 articles appear in both categories and therefore we consider these two categories as belonging to the same context. Comparing section contents from the same context is important because for example a section titled “History” in an article about a battle in World War 2 will contain different words than the “History” section from an article about a city.

Section contents coming from a single category could have been compared among them, but only about 5% of articles belong to a single category and 12% of articles belong to three categories as shown on Figure 4.3. This means that if we compare section contents from single categories, the same section content from a given article would be used multiple times in different categories to find similar sections, which seemed redundant in terms of computations because the same pairs of section contents would be compared multiple times. The idea was to compute the semantic similarity between sections which would more likely appear together in recommendation lists, by grouping categories together if they are likely to be used to provide sections in the same recommendation lists. In other words, we wanted to group together categories in such a way that articles would in majority belong to one single category group *i.e.,* context.

This allowed us to reduce the number of times that the same section contents would be compared. Additionally, we had a bigger set of section contents to be compared with each other, especially for categories containing few articles because those categories would be grouped with bigger ones. The whole category grouping pipeline is shown on Figure 4.2, the different steps are explained in this section.

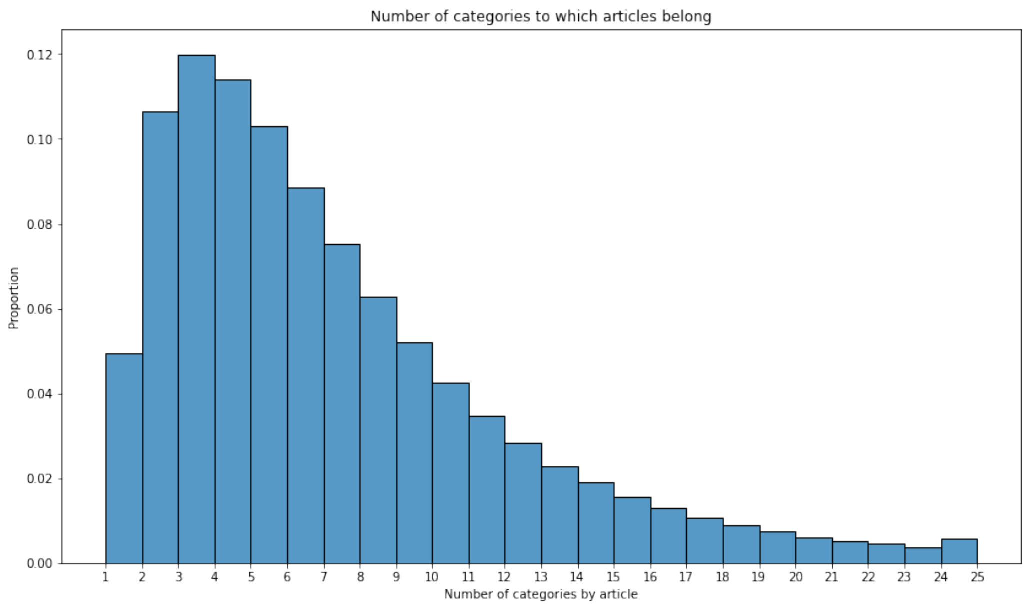
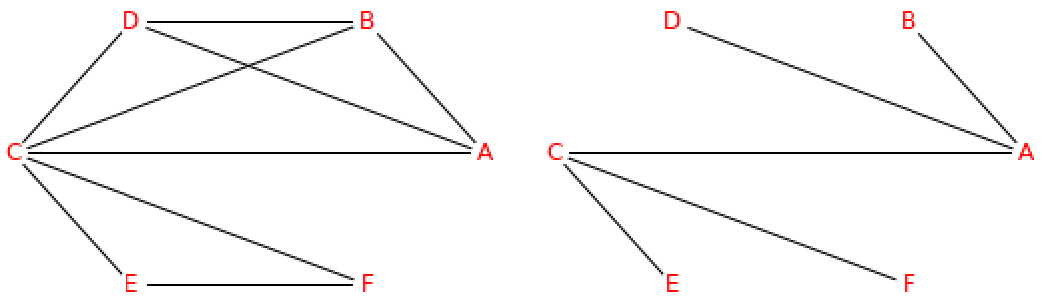


Figure 4.3: Histogram of number of categories to which articles belong

The chosen method for grouping together categories by context was the Louvain method, a network community detection algorithm. The idea was to build a graph where categories were nodes, and the edges between two categories had a weight which represented the likelihood that those categories shared the same articles.

To be able to fit this graph into memory, we had to make a first coarse grouping of the categories. In order to achieve this, we built a first graph where for each article, we connected together categories to which this article belongs. Instead of connecting all categories between them and computing the weights of the edges, we grouped all categories to which an article belongs to a single category as shown on the right side on Figure 4.4. The idea was to firstly split the graph into parts by detecting connected components and then load only one single connected component at the same time and apply community detection on each component separately.



(a) How those categories should be connected before applying community detection

(b) How we connected them to detect connected components beforehand to save memory

Figure 4.4: If an article belongs to categories A,B,C,D and another to C,E,F instead of fully connecting all the categories to which an article belongs, we connected them to one single category in order to make a first coarse grouping

We found 24'061 connected components in this graph, this allowed us to make a first coarse grouping where categories which were inside the same connected component could have articles in common. Categories that were in different connected components could not have articles in common, therefore we were able to avoid wasting memory by loading in the same graph categories which were guaranteed to not be grouped together.

With those connected components, we have made a first coarse grouping and we were able to connect categories inside each connected component as shown on the left side on Figure 4.4 and computing the edge's weights based on the number of articles belonging to both categories connected with a given edge.

In order to compute the semantic similarity between section contents, we first needed to precompute on the GPU the embeddings of those section contents. Therefore, an element of which we had to take care of was the memory limitation of the GPU on which we had to transform section contents into embeddings. We wanted that the set of section contents of the biggest category community would fit into the GPU's memory. To achieve this, we defined an upper limit in terms of unique section contents that a category community should have. To define this limit, we have looked at the categories which had the biggest number of section contents in the dataset (*i.e.,* those that have the most articles contributing to their section counts) and have manually grouped together categories[4] which seemed to have likely the same context. Then we have counted the number of unique section contents in those categories that we have manually grouped together (*i.e.,* unique in the sense that some articles belong to multiple categories among those that we have grouped together, therefore we counted each article's section contents only once for the whole category group), and obtained 433'150 unique section contents. If after detecting a category community, the number of unique section contents was above this limit, we re-applied community detection inside this category community in order to further split it in smaller communities, but connected categories together if they had common articles which contributed to the categories' section counts instead of using common articles which belong to both categories. This way, after applying the community detection, each article would be more likely contributing to section counts of a single category community. At first, we were not able to connect categories which had common articles contributing to their section counts because the graph didn't fit into the memory, therefore we used this way of connecting categories in the graph only to further split a category community which had too many unique section contents.

The method of category grouping is presented in detail on Algorithm 1.

[4]American compositions and recordings, Albums by American artists, Rock albums by American artists, Rock albums, 21st-century albums, 21st-century songs, Pop songs, American songs, Rock songs, 20th-century songs, Jazz albums

Algorithm 1: Category grouping by context algorithm

```
Result: Categories grouped by context
$category\ communities \leftarrow \{\}$
for each connected component do
    build graph $G$ with categories from connected component as nodes, connected together if there are
      articles belonging to both categories
    for each edge between categories $x$ and $y$ in graph $G$ do
        $Articles(x) \leftarrow$ articles belonging to category $x$
        $Articles(y) \leftarrow$ articles belonging to category $y$
        $edge\ weight \leftarrow \frac{|Articles(x) \cap Articles(y)|}{|Articles(x) \cup Articles(y)|}$
    end
    $detected\ communities \leftarrow Louvain(G)$
    for community $C$ in detected communities do
        $nb\ unique\ section\ contents \leftarrow Count\ unique\ section\ contents(C)$
        if $nb\ unique\ section\ contents \leq 433150$ then
            $category\ communities \leftarrow category\ communities \cup \{C\}$
        else
            build graph $G_2$ containing only categories from $C$, connected together if there are articles
              contributing to section counts of both categories
            for each edge between categories $x$ and $y$ in graph $G_2$ do
                $Articles(x) \leftarrow$ articles contributing to section counts of category $x$
                $Articles(y) \leftarrow$ articles contributing to section counts of category $y$
                $edge\ weight \leftarrow \frac{|Articles(x) \cap Articles(y)|}{|Articles(x) \cup Articles(y)|}$
            end
            $detected\ communities_2 \leftarrow Louvain(G_2)$
            for community $C_2$ in $detected\ communities_2$ do
                $category\ communities \leftarrow category\ communities \cup \{C_2\}$
            end
        end
    end
end
return $category\ communities$
```

An example of a detected category community is shown on Figure 4.5.

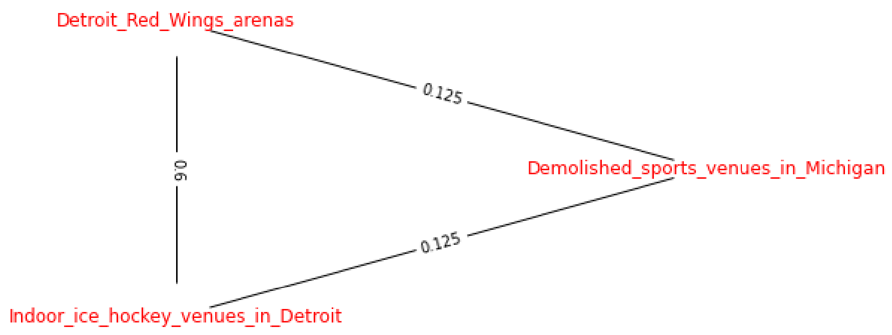

Detroit_Red_Wings_arenas
0.125
0.6
Demolished_sports_venues_in_Michigan
0.125
Indoor_ice_hockey_venues_in_Detroit

Figure 4.5: Example of a subset inside a detected category community

The edge weight between ”Detroit red wings arenas” and ”Indoor ice hockey venues in Detroit” is 0.6 because both categories have 4 articles, 3 of them are in common between both categories, and 1 article is unique to each category, therefore there are 5 different articles in total with 3 of them that are common, the computed weight is 3/5=0.6

After grouping together categories into category communities, about 60% of articles belonged to a single category community, and about 20% to two category communities as shown on Figure 4.6. Thanks to this, we have reduced the number of times that the same pairs of section contents would be compared, *i.e.,* reduced computational overhead. Moreover categories containing few articles have benefited from the grouping, because it allowed us to compare a bigger set of section contents among them while maintaining coherent contexts.

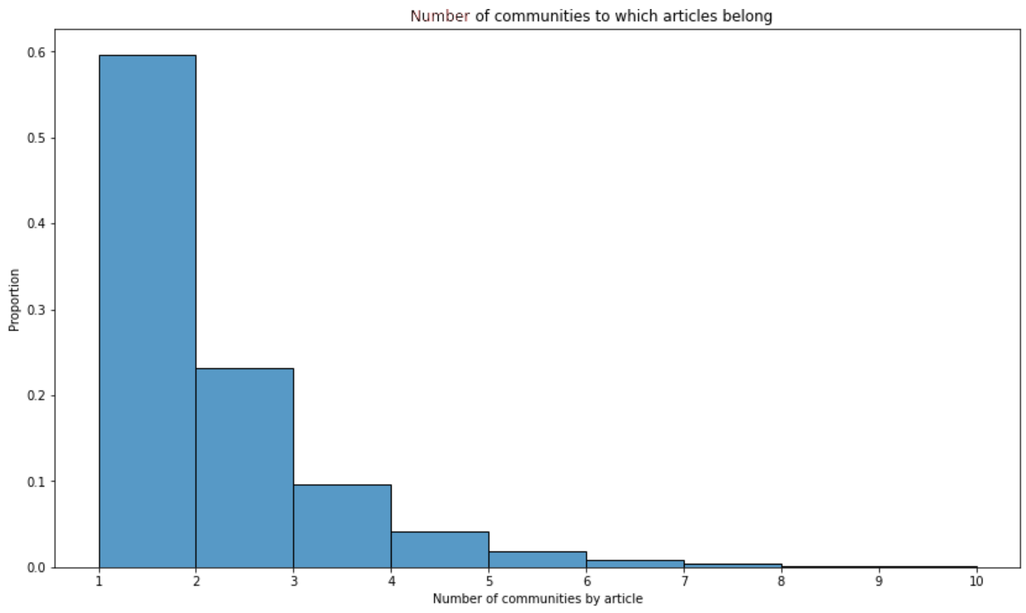


Figure 4.6: Histogram of number of category communities to which each article belongs

4.3 Detect semantically similar sections

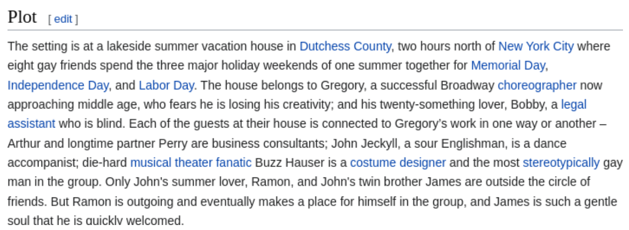
After grouping categories into communities and extracting the section contents, we needed to compare section contents appearing in the same category communities in order to detect semantically similar sections.

For each category we computed the set formed by sections that could appear in recommendation lists; *i.e.,* the 30 most frequent sections as defined by the initial recommendation algorithm. We also retrieved for each category the set of articles which contributed to those 30 most frequent sections.

Then for each category community, we computed the union between the sets of sections of all categories inside each category community. We also computed the union between the sets of articles which belong to categories inside each category community. This allowed us to retrieve for each category community from all articles, all the contents of sections which could appear in recommendations.

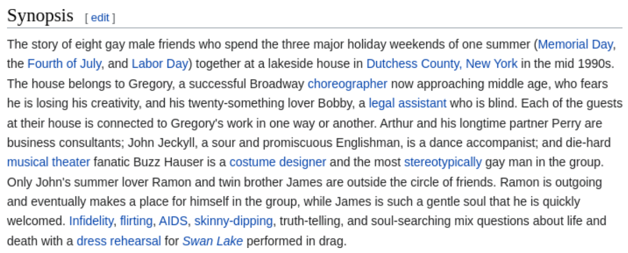
For each category community, we retrieved all its section contents and then transformed them into their sentence embeddings by using the ”paraphrase-distilroberta-base-v2” model from the sentence_transformers module.

To find semantically similar section content pairs, we used a slightly modified version of the paraphrase_mining method provided by the sentence_transformers module. We made sure that only pairs where both section contents came from different articles and from different sections were retained. This allowed us to avoid detecting section content pairs which were obviously similar because they came from the same section but from different articles or from the same article but from different sections. Additionally, we decided to retain only section content pairs where the cosine similarity score was over 0.5 because our goal was to find only the most similar section content pairs. An example of a section content pair detected as semantically similar is shown on Figure 4.7.


Plot [edit]

The setting is at a lakeside summer vacation house in Dutchess County, two hours north of New York City where eight gay friends spend the three major holiday weekends of one summer together for Memorial Day, Independence Day, and Labor Day. The house belongs to Gregory, a successful Broadway choreographer now approaching middle age, who fears he is losing his creativity; and his twenty-something lover, Bobby, a legal assistant who is blind. Each of the guests at their house is connected to Gregory's work in one way or another – Arthur and longtime partner Perry are business consultants; John Jeckyll, a sour Englishman, is a dance accompanist; die-hard musical theater fanatic Buzz Hauser is a costume designer and the most stereotypically gay man in the group. Only John's summer lover, Ramon, and John's twin brother James are outside the circle of friends. But Ramon is outgoing and eventually makes a place for himself in the group, and James is such a gentle soul that he is quickly welcomed.

(a) "Plot" section from an article about a play


Synopsis [edit]

The story of eight gay male friends who spend the three major holiday weekends of one summer (Memorial Day, the Fourth of July, and Labor Day) together at a lakeside house in Dutchess County, New York in the mid 1990s. The house belongs to Gregory, a successful Broadway choreographer now approaching middle age, who fears he is losing his creativity, and his twenty-something lover Bobby, a legal assistant who is blind. Each of the guests at their house is connected to Gregory's work in one way or another. Arthur and his longtime partner Perry are business consultants; John Jeckyll, a sour and promiscuous Englishman, is a dance accompanist; and die-hard musical theater fanatic Buzz Hauser is a costume designer and the most stereotypically gay man in the group. Only John's summer lover Ramon and twin brother James are outside the circle of friends. Ramon is outgoing and eventually makes a place for himself in the group, while James is such a gentle soul that he is quickly welcomed. Infidelity, flirting, AIDS, skinny-dipping, truth-telling, and soul-searching mix questions about life and death with a dress rehearsal for *Swan Lake* performed in drag.

(b) "Synopsis" section from an article about the film adaptation of the same play

Figure 4.7: Example of a detected semantically similar section content pair, with a cosine similarity of 0.95

Source: `https://en.wikipedia.org/w/index.php?title=Love!_Valour!_Compassion!&oldid=994865300` (left), `https://en.wikipedia.org/w/index.php?title=Love!_Valour!_Compassion!_(film)&oldid=998558517` (right)

The same pair of sections could have multiple section contents which were detected as semantically similar. Therefore we averaged the cosine similarity scores and retained the number of section contents which were detected as similar between those two different sections. Section contents which were not detected as semantically similar were not taken in account.

4.4 Filter semantically similar sections

Once we had pairs of semantically similar sections inside each category community, we could use them to filter out similar sections which appeared inside recommendation lists.

Because a recommendation list could have groups of more than 2 semantically similar sections, we decided to represent pairs of semantically similar sections as edges in a graph. Therefore if we had two pairs of sections such as (A, B) and (B, C), the sections A and C could also be grouped together as shown on Figure 4.8.

We decided to try out different thresholds for the semantic similarity above which we considered that two sections were similar. This way the user could choose the granularity with which the filtering was applied. Lowering the minimum cosine similarity threshold above which a section pair is considered as semantically similar would mean that more sections would be detected as semantically similar and potentially more sections would be filtered out. Increasing this threshold would mean that less sections would be detected as semantically similar and potentially less sections would be filtered out.

Because the distribution of cosine similarity thresholds for similar section pairs differed from one category community to another, we defined 4 semantic similarity levels which corresponded to different cosine similarity threshold in each category communities instead of predefined threshold values across all category communities.

Level 0 meant that no semantic filtering would be applied. For levels 1 to 3, we have split the distribution of cosine similarities of detected section pairs in each category community into 3-quantiles. Semantic filtering level 3 contained all detected similar pairs, level 2 contained the top 2/3 most similar pairs and level 1 contained the top 1/3 of most similar pairs.

We show on Algorithm 2 how we filter semantically similar sections inside a list of recommended sections for a given article:

Algorithm 2: Semantically similar section filtering algorithm

```
Result: Filtered recommended sections
$recs \leftarrow$ recommended sections
$semantic\_filtering\_level \leftarrow$ chosen semantic filtering level
if $semantic\_filtering\_level = 0$ OR $len(recs) < 2$ then
    return $recs$
end
$categories \leftarrow$ categories to which article belongs
$communities \leftarrow \{\}$
for $category$ in $categories$ do
    $communities \leftarrow communities \cup \{$ category community of $category\}$
end
$G \leftarrow$ empty graph
$edge\_weights \leftarrow$ init dict with pair of sections as key, empty list of edge weights as value
$global\_nb\_sections \leftarrow$ init dict with section as key, number of sections init to 0 as value
for $community$ in $communities$ do
    $cosine\_threshold \leftarrow$ threshold corresponding to $semantic\_filtering\_level$ in $community$
    $pairs \leftarrow$ get similar section pairs detected in $community$
    for $pair$ in $pairs$ do
        $section\_A \leftarrow$ first section of pair
        $section\_B \leftarrow$ second section of pair
        if $cosine\_similarity(pair) > cosine\_threshold$ AND $section\_A$ in $recs$ AND $section\_B$ in
          $recs$ then
            $nb\_sections\_A \leftarrow$ get section count of $section\_A$ in $community$
            $nb\_sections\_B \leftarrow$ get section count of $section\_B$ in $community$
            $global\_nb\_sections[section\_A] \leftarrow global\_nb\_sections[section\_A] + nb\_sections\_A$
            $global\_nb\_sections[section\_B] \leftarrow global\_nb\_sections[section\_B] + nb\_sections\_B$
            $nb\_similar\_section\_contents \leftarrow$ get number of similar section contents between
              $section\_A$ and $section\_B$ in $community$
            if $nb\_similar\_section\_contents > 1$ OR $min(nb\_sections\_A, nb\_sections\_B) = 1$ then
              // Consider pairs with 1 similar section content as noise
                add edge in $G$ between $section\_A$ and $section\_B$
                $w \leftarrow \frac{nb\_similar\_section\_contents}{nb\_sections\_A \cdot nb\_sections\_B}$
                $edge\_weights[(section\_A, section\_B)].append(w)$
            end
        end
    end
end
for each edge beetween $section\_A$ and $section\_B$ in $G$ do
    $edge\ weight \leftarrow$
      $sum(edge\_weights[(section\_A, section\_B)])/len(edge\_weights[(section\_A, section\_B)])$
end
$section\_relevance \leftarrow$ init dictionary with section as key, value init to 0 which will decide which
  section to keep in each group of similar sections
for each section represented as node in $G$ do
    $neighbors \leftarrow$ get neighbors of $section$ in $G$
    $average\_edge\_weight \leftarrow$ average edge weight among edges between $section$ and its $neighbors$
    $section\_relevance[section] \leftarrow average\_edge\_weight \cdot global\_nb\_sections[section]$
end
```

```
$groups\_of\_similar\_sections \leftarrow Louvain(G)$
**for** *each group in groups_of_similar_sections* **do**
    $sections\_in\_group \leftarrow$ get sections in $group$
    $section\_to\_keep \leftarrow$ get $section$ that has the highest value according to
      $section\_relevance[section]$ among $sections\_in\_group$
    $sections\_to\_remove \leftarrow$ all sections from $sections\_in\_group$ except $section\_to\_keep$
    remove all sections from $recs$ which are in $sections\_to\_remove$
**end**
**return** $recs$
```

The intuition behind the usage of Louvain's algorithm to detect groups of similar sections was that we wanted to group together sections that had the highest normalized number of similar section contents inside each group (the normalized number of similar section contents is $\frac{nb\_similar\_section\_contents}{nb\_sections\_A \cdot nb\_sections\_B}$ shown on Algorithm 2, which has a value of 1 if all possible pairs of section contents between two sections were detected as semantically similar). An example of two detected groups of similar sections is shown on Figure 4.8.

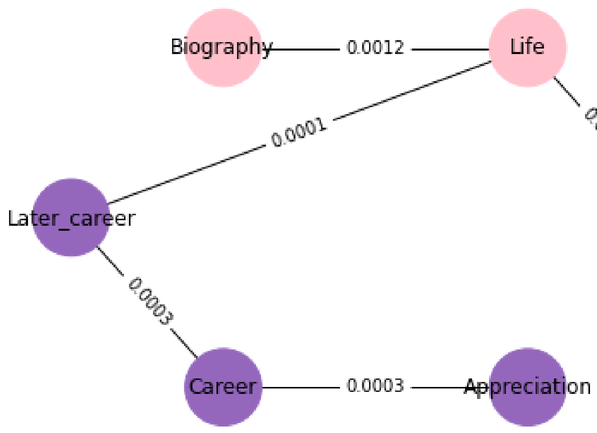


Figure 4.8: Example of 2 detected communities of semantically similar sections inside a recommendation list, "Career" and "Biography" would be kept and the remaining sections filtered. The edge weights are the normalized number of similar section contents.

Initially too many sections were grouped together, *e.g.,* on the example on Figure 4.8 all the sections were grouped together instead of being separated into two groups. In order to fix that, we used the first pass of the Louvain algorithm's partition instead of the last one, which gave us smaller communities.

Another way to reduce the size of detected communities was to ignore (*i.e.,* not consider as similar) pairs of sections which appeared in the same articles in the dataset. The intuition behind this was that redundant sections in the recommendation lists came from different articles of the same category, *e.g.,* normally in the same article there should not be "Life" and "Biography" at the same time, if a recommendation list contains both of them this means that "Life" came from one article and "Biography" from another. But the dataset contained some articles where redundant sections were in the same article, *e.g.,* an article contained both the section "Life" and the section "Biography" because the "Life" section was a summary of the person's biography whereas the section "Biography" was about a biographical book written on the life of this person[5]. Therefore we could not use in a binary way the fact that a pair of similar sections appeared in the same articles in order to ignore those pairs of similar sections in the similar section graph. We came up with a way of quantifying the number of times where two sections appeared in the same articles which allowed us to ignore some noise like the example with "Life" and "Biography" sections which appeared in the same article.

Therefore, we computed for each pair of similar sections how many times those two sections would appear in the same article. In order to do that, in each category community we retrieved for each section the set of articles in which this section appears. Then for each retrieved similar sections pair (x, y) we computed the Jaccard similarity between the sets of articles where those sections appeared, as shown in the following equation:

[5] `https://en.wikipedia.org/w/index.php?title=George_Smith_(Scottish_artist)&oldid=981246048`

$$same\ article\ appearance(x, y) = \frac{|Articles(x) \cap Articles(y)|}{|Articles(x) \cup Articles(y)|} \tag{4.1}$$

Where:

$Articles(x)$ is the set of articles in which section x appears.

$Articles(y)$ is the set of articles in which section y appears.

We found empirically that a maximum threshold of 0.001 was enough to ignore similar section pairs (*i.e.,* not adding them to the similar sections graph if their same article appearance is above the threshold) by considering that pairs having a same article appearance value under this threshold came from noise like the above-mentioned example of "Life" and "Biography" sections which appeared in the same article.

If this maximum same article appearance threshold would be set lower, less section pairs (*i.e.,* those appearing more often in the same articles than the threshold) would be considered as similar, whereas if this threshold would be set higher, more section pairs would be considered as similar.

This maximum threshold of 0.001 was used with the semantic filtering levels 1 and 2. For semantic filtering level 3 we used a threshold of 0.005 because it allowed to group more semantic similar sections, which was a desired behavior when the semantic filtering level was increased. With semantic filtering level 3 *i.e.,* the threshold set to 0.005, in a tested example the sections "Works" and "Filmography" were grouped together because this threshold was increased, otherwise those sections appeared too often in the same articles in this category community to be grouped together with the threshold at 0.001.

In the example on Figure 4.9 which is from a recommendation list for an article about a city, all edges in the similar section graph which are shown were removed after filtering out pairs of sections which appeared too many times in the same articles. After introducing this same article appearance threshold, sections "Media", "Sports", "Economy" and "Demographics" were not longer considered as semantically similar.

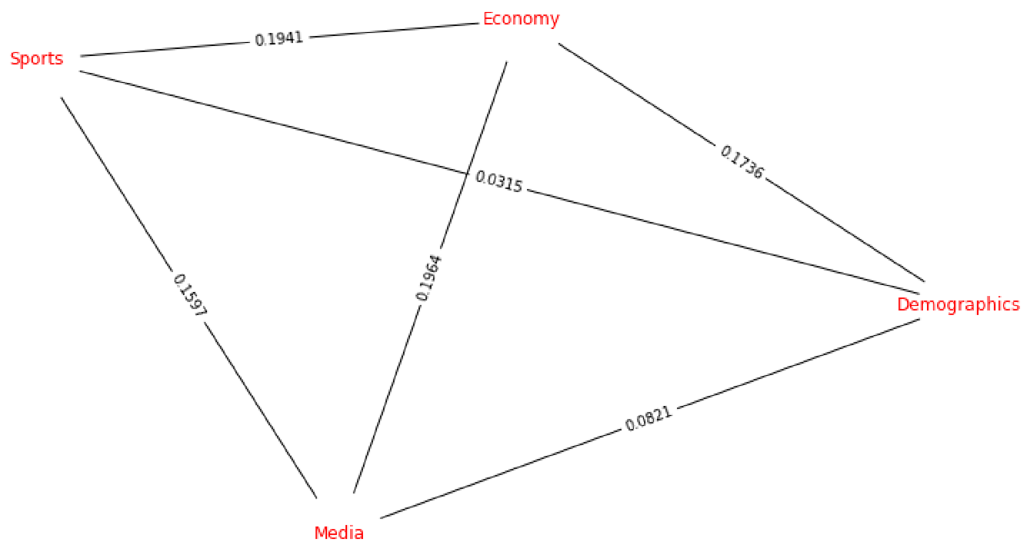
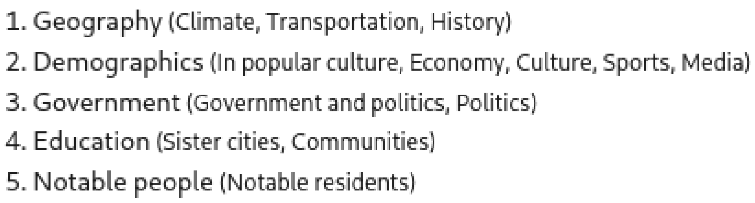


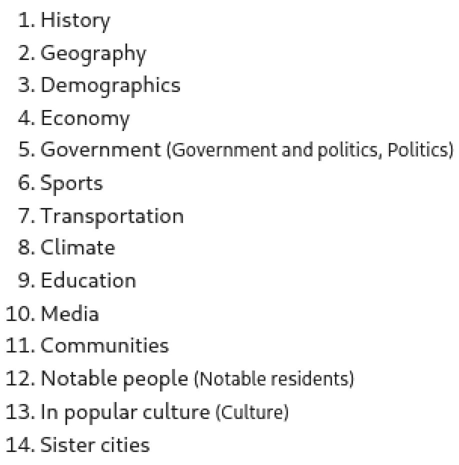
Figure 4.9: A subset of a similar section graph where the edge weights are the Jaccard similarity between the sets of articles in which each section appears (*i.e.,* the same article appearance), before ignoring edges which had weights above 0.005.

Figure 4.10 illustrates the effect on the whole recommendation list after removing section pairs from Figure 4.9. The sections between parentheses are those that were detected as semantically similar and were filtered out, *e.g.,* "Government (Politics, Government and politics)" means that those 3 sections were detected as a group of similar sections and only "Government" was retained and the others filtered out.



1. Geography (Climate, Transportation, History)
2. Demographics (In popular culture, Economy, Culture, Sports, Media)
3. Government (Government and politics, Politics)
4. Education (Sister cities, Communities)
5. Notable people (Notable residents)

(a) Recommendation list before ignoring section pairs that appeared too many times in same articles.



1. History
2. Geography
3. Demographics
4. Economy
5. Government (Government and politics, Politics)
6. Sports
7. Transportation
8. Climate
9. Education
10. Media
11. Communities
12. Notable people (Notable residents)
13. In popular culture (Culture)
14. Sister cities

(b) Recommendation list after ignoring section pairs that appeared too many times in same articles.

Figure 4.10: Example a recommendation list for an article about a city, before and after filtering similar section pairs which appeared too often in same articles.

One final encountered problem was that after filtering out pairs of similar sections that appeared too many times in the same articles, in some category communities, the sections ”Works” and ”Work” were no longer considered as similar. To fix that, we have decided to keep pairs of similar sections in the similar sections graph if there was an overlap between sets of stemmed words from both sections, *e.g.,* pairs consisting of ”Early Life” and ”Life” or ”Works” and ”Work” were kept even if they appeared more often in the same articles than the defined threshold.

5 Section ordering

This part of the project proposes a way to solve the second cited limitation of the original paper.

Because the recommendation algorithm builds the recommendation list using the most frequent sections in a given category, the first section in the list is the most frequent and the k-th section in the list is the k-th most frequent. The original ordering of the sections inside the articles of the training set is not taken into account (*e.g.,* “Conclusion” could be at the beginning of the recommendation list and “Introduction” at the end if “Conclusion” is more frequent than “Introduction”).

The goal of this part of the project was to resolve this issue by finding a method to order the sections in the recommendation list in a way that would preserve the original ordering from the articles from which those sections come from.

As introduced in Section 2.4, the idea was to use the relative position of the sections in the table of contents *i.e.,* the order value, which ranges between 0 and 1 (0 meaning that the section is at the beginning of the document, and 1 meaning that the section is at the end). We had to find a way to aggregate the order values for the same section from different articles, and to take into account the possibly different coverage of articles for the same section (*e.g.,* if in one case a section covers the whole article and in another only 1/10th at the beginning of the article).

The solution for this problem was to compute the relative position where a section begins (which we call beginning order value), and the relative position where a section ends (which we call end order value) as shown on Figure 5.1.

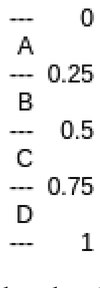


Figure 5.1: Illustration of beginning and end order values for an article with 4 sections

In this example, where an article contains sections A,B,C,D, section A has an order value of 0 for the beginning and 0.25 for the end, B has 0.25 for the beginning and 0.5 for the end, *etc.*

For each section the beginning and end order values were computed by measuring the relative position of those sections inside the table of contents of the articles in which the sections appeared as shown on Figure 5.1. Only the most frequent sections in each category were used, in order to avoid to make computations for sections that will not appear in the recommendation lists, which takes the 30 most frequent sections in each category.

Another potential issue to take care of was the fact that the same section could have a different relative position in different categories (*i.e.,* context). The solution to this was to compute section order values separately for each different category. Then, when sections were ordered, only the order values which came from the categories to which a given article belongs were used and aggregated.

We summed up the order values for each section in each category and retained these sums alongside the section's occurrence number. This allowed us to later aggregate order values from different categories when we ordered the sections of the recommendation lists as shown on Algorithm 3.

Finally, once the beginning and end order values were computed for each section in each category, we could use those values to order the recommendation lists as shown on Algorithm 3.

Algorithm 3: Section ordering algorithm

```
Result: Ordered sections
$order\_value\_by\_section \leftarrow$ Init dict with section as key and order value init to 0 as value
for each section to order do
    $nb\ values \leftarrow 0$ ;                                /* Number of order values */
    $sum\ values\ beginning \leftarrow 0$ ;                  /* Sum of beginning order values */
    $sum\ values\ end \leftarrow 0$ ;                        /* Sum of end order values */
    for each category to which article belongs do
        $nb\ values \leftarrow nb\ values + number\ of\ order\ values\ of\ section\ in\ category$
        $sum\ values\ beginning \leftarrow$
          $sum\ values\ beginning + sum\ values\ beginning\ of\ section\ in\ category$
        $sum\ values\ end \leftarrow sum\ values\ end + sum\ values\ end\ of\ section\ in\ category$
    end
    $beginning\ order\ value \leftarrow \frac{sum\ values\ beginning}{nb\ values}$
    $end\ order\ value \leftarrow \frac{sum\ values\ end}{nb\ values}$
    $order\_value\_by\_section[section] \leftarrow \frac{beginning\ order\ value + end\ order\ value}{2}$
end
Ascending sort sections by their $order\_value\_by\_section[section]$ value
return sorted sections
```

With this method, the average kendall's tau metric which compared orderings between recommendation lists with the ordering of the ground truth, increased from 0.22 for unordered recommendation lists to 0.89 after the recommendation lists were ordered.

An example of a recommendation list for an article[1] before and after applying the ordering algorithm is shown on Table 5.1.

Ground truth	Recommendations before ordering	Recommendations after ordering
Calendar	First round	Calendar
Format	Second round	Format
First round	Third round	First round
Second round	Final	Second round
Third round	Fourth round	Third round
Fourth round	Quarter-finals	Fourth round
Quarter-finals	Semi-finals	Fifth round
Semi-finals	Statistics	Quarter-finals
Final	Fifth round	Semi-finals
	Calendar	Final
	Format	Statistics

Table 5.1: Section ordering example

The kendall's tau coefficient comparing the recommended sections with the ground truth for the example of Table 5.1 increased from 0.05 before ordering to 1 after ordering. To compute kendall's tau coefficient, we used sections that were in common between the ground truth and the recommendations, *i.e.,* we ignored for the kendall's tau computation the "Fifth round" and "Statistics" sections in the recommendation list.

[1]Category "2015–16_in_Scottish_football_cups", semantic filtering level set to 0, recommendation list length set to 11

6 Experiments and Results

In this chapter, the results are presented and commented in details.

We begin with the explanation of the metrics that we used to evaluate our system.

As the first result, we present the problem that we had while reproducing the recommendation algorithm from the paper on which this project is based. We weren't able to obtain the same performance at the beginning of the project, the reasons were already explained in Section 3.2. Here we show the steps which allowed us to be sure that our reproduction is nevertheless correct.

Secondly, we show in details how much the performance of the original recommendation algorithm increased with our proposed method improvements described in Section 3.3.

Then, the way we evaluated the semantic similar section filtering part is presented (this part is explained in detail in Chapter 4), alongside with plots comparing different filtering configurations with the baseline before filtering.

Finally, the evaluation method and results for the section ordering part of the project (described in Chapter 5) are presented.

6.1 Used metrics

The initial algorithm's authors published their results as precision@k and recall@k for k ranging from 1 to 20 (*i.e., k* is the length of the section recommendation lists), therefore we decided to compare our result to those published by the initial algorithm's authors by using the same metrics. We used particularly precision@1 and recall@20 because the precision was the highest at $k = 1$ and the recall was the highest at $k = 20$ for all the results before the semantic filtering part.

As a reminder, the precision is defined as follows:

$$\text{precision} = \frac{|\{\text{ground truth sections}\} \cap \{\text{recommended sections}\}|}{|\{\text{recommended sections}\}|} \tag{6.1}$$

And the recall is defined as follows:

$$\text{recall} = \frac{|\{\text{ground truth sections}\} \cap \{\text{recommended sections}\}|}{|\{\text{ground truth sections}\}|} \tag{6.2}$$

Precision@k or recall@k means that we use only the k first recommended sections from the recommendation list to compute the precision or recall.

For the semantic similar section filtering part we also use the F1 score as metric, which is the harmonic mean of precision and recall and is defined as follows:

$$F_1 = 2 \cdot \frac{\text{precision} \cdot \text{recall}}{\text{precision} + \text{recall}} \tag{6.3}$$

F1 score@k is the harmonic mean of precision@k and recall@k.

All those above-mentioned metrics were averaged over all articles from the test set for k ranging from 1 to 20.

Finally, we used for the section ordering evaluation kendall's tau correlation coefficient which is already described in Section 2.4. To summarize, this value is used to compare how close the ordering is between two ranked lists *i.e.,* in our case we compare the ordering of the sections from our recommendation lists with the ordering of sections from the ground truth. This value can range from -1 (exactly inverse order) to 1 (exactly the same order).

6.2 Paper reproduction

First, we used data from the March 2021 Wikipedia dump to reproduce the method, by building ourselves the dataset needed for the recommendation algorithm.

However, the results didn't matched those provided in the paper (we had precision@1 at 0.55 and recall@20 at 0.7 instead of precision@1 at 0.57 and recall@20 at 0.8), therefore we decided to reproduce the method with the same data as the authors, *i.e.,* Wikipedia's 2017 dump.

The recommendation algorithm was applied with the provided data by the authors, with the list of categories to which each article belongs and by using the category section counts which were precomputed by the authors to predict sections for each article from the test set (*i.e.,* articles present in the results file).

This time, the results were the same (precision@1 at 0.57 and recall@20 at 0.8), which means that those precomputed category section counts containing sections exclusively present in the test set (as shown in Section 3.2) were indeed used for predicting the sections for the test set. Additionally, it proved that our method reproduction was right up to the category section counts computation.

The next step was to be sure that our way of computing the category section counts was correct. We reproduced the method by computing ourselves the category section counts instead of using the provided ones. This was done by using as before the categories to which each article from the test set belongs and this time we also used the list of ontologically pure categories which contained articles that belong to those categories alongside articles that belong to children of those categories. The sections for each article of the train set were not provided, therefore we had to extract them ourselves from a 2017 Wikipedia dump. We voluntary included articles from the test set in the category section counts to see if it would match the provided results.

We obtained a slight difference in performance (precision@1 at 0.59 and recall@20 at 0.85) compared to the results provided by the authors of the paper (precision@1 at 0.57 and recall@20 at 0.8). This could be explained by potential differences in the way we extracted sections from the dump. Moreover, sections that appeared only once in small categories were sometimes in the 30 most common sections used for recommendation. The handling of ties in the sorting algorithm used by Python Counter class for those sections unique in their categories could be different compared to the handling of ties used by the paper authors, which could introduce differences because of that.

Nevertheless, the category section counts which we computed for the category "Cruise missiles of Pakistan" (which we used as example in Section 3.2 to show that the paper authors have included the test set in their training set) were the same[1] as those in the provided results by the authors[2].

Finally, we evaluated the system with Wikipedia's 2017 data by excluding the articles of the test set from the category section counts.

This time, the category section counts for the category "Cruise missiles of Pakistan" were different from those provided by the paper authors because sections which were exclusively present in the test set were no more included[3].

Those results were similar to the first reproduction of the method with Wikipedia data from 2021, with precision@1 at 0.53 and recall@20 at 0.68, which turned out to be slightly lower than the first obtained results with data from 2021 (the precision@1 was at 0.55 and recall@20 at 0.7). This difference could be explained by

[1]See our published results, `results/epfl_reproduction-category_section_counts_with_test_set/recs_by_category_top30.json` file, line 386712

[2]`https://figshare.com/articles/dataset/Structuring_Wikipedia_Articles_with_Section_Recommendations/6157583` "`recs_probabilistic_by_category_top30.json`" file, line 11185

[3]See our published results, `results/epfl_reproduction/recs_by_category_top30.json` file, line 153979

the fact that the proportion of articles with no defined DBpedia type dropped from 48% to 42% between 2017 and 2021, which decreased the inducted noise coming from those articles in the category section counts.

The comparison in terms of precision between experiments described in this section are shown on Figure 6.1 and the comparison in terms of recall on Figure 6.2. We haven't included on the plot the results of the reproduction with the precomputed category section counts which were provided by the paper's authors (see Table A.3) because they are the same as the results provided by the paper's authors (see Table A.2).

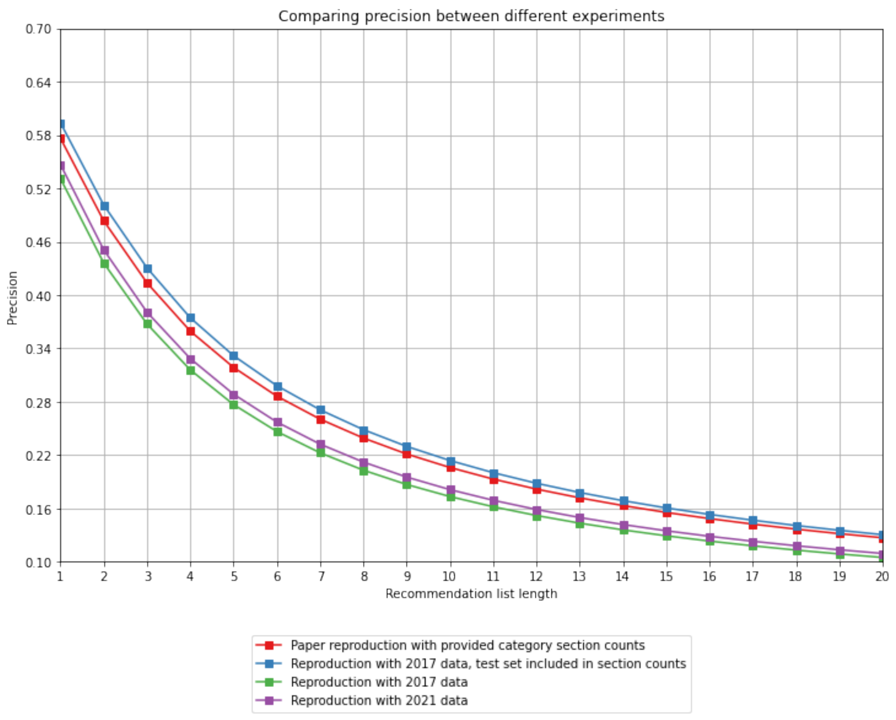


Figure 6.1: Comparation of precision between different reproduction experiments, values are shown in detail in Section A.1.

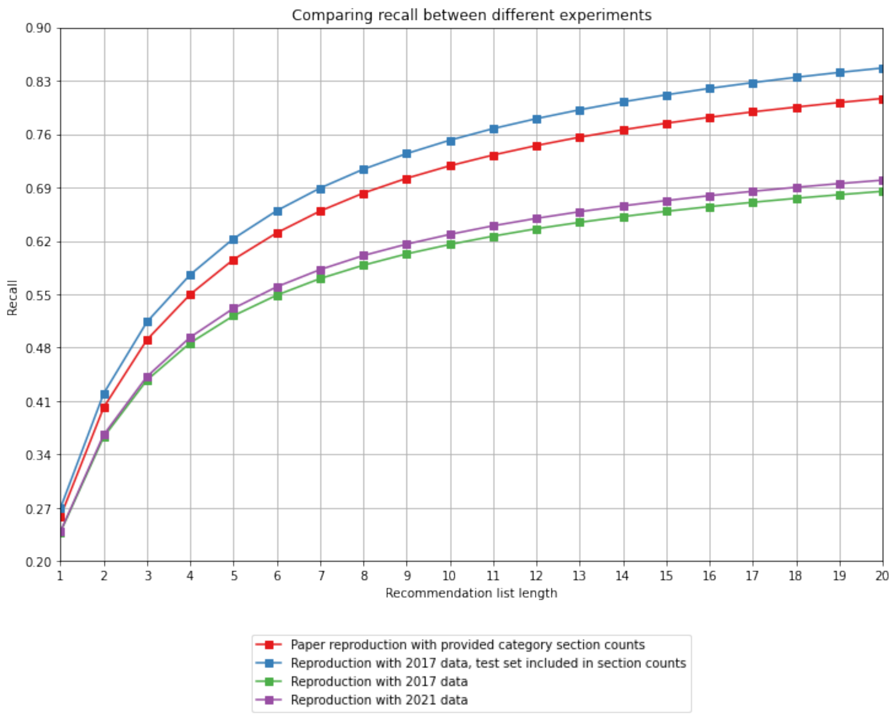


Figure 6.2: Comparation of recall between different reproduction experiments, values are shown in detail in Section A.1.

The results of this section are summarized in Table 6.1.

	Precision@1	**Recall@20**
Results provided by paper's authors	0.57	0.8
Reproduction with 2017 data, sections from test set in section counts	0.59	0.84
Reproduction with 2017 data	0.53	0.68
Reproduction with 2021 data	0.55	0.7

Table 6.1: Summary of the results from this section

The results with 2021 data were used as baseline to be compared with results from the method improvements presented in Section 6.3.

6.3 Method improvements

We noticed that the initial category pruning algorithm included sections from articles with no defined DBpedia type belonging to child categories in the section counts of parent categories. To avoid that, we decided to filter out those articles in order to prevent potential propagation of noise in the category section counts. We used Wikipedia data from March 2021.

After filtering out articles which had no defined DBpedia type, the amount of articles decreased from 3'643'496 to 2'048'191 (56% of articles were retained). Articles which had no defined DBpedia type were no more used in the test set nor in the train set.

Precision@1 increased from 0.55 to 0.63 and recall@20 increased from 0.7 to 0.76 compared to the baseline. This was the improvement that has given the most significant performance increase. This means that indeed, articles with an unknown DBpedia type added noise by being included the section counts of categories which were higher in the category hierarchy, without being sure that they would keep the categories ontologically pure.

Then we tried to change the gini coefficient threshold under which a category is considered as ontologically impure, and after some empirical testing we found that changing the threshold from 0.966 to 0.985 gave the biggest performance improvement. We computed the results without filtering out articles with unknown types to see the increase compared to the baseline.

Precision@1 increased from 0.55 to 0.58 and recall@20 increased from 0.7 to 0.72 compared to the baseline.

Finally, we combined both improvements (raising the gini threshold and filtering out articles which had no known type) to see how much the performance will increase compared to the baseline.

Precision@1 increased from 0.55 to 0.65 and recall@20 increased from 0.7 to 0.77 compared to the baseline. The precision was even higher than the results provided by the original paper's authors where the test set was included in the category section counts, the recall was almost as high.

The comparison in terms of precision between experiments described in this section and the chosen baseline from Section 6.2 is shown on Figure 6.3 and the comparison in terms of recall on Figure 6.4.

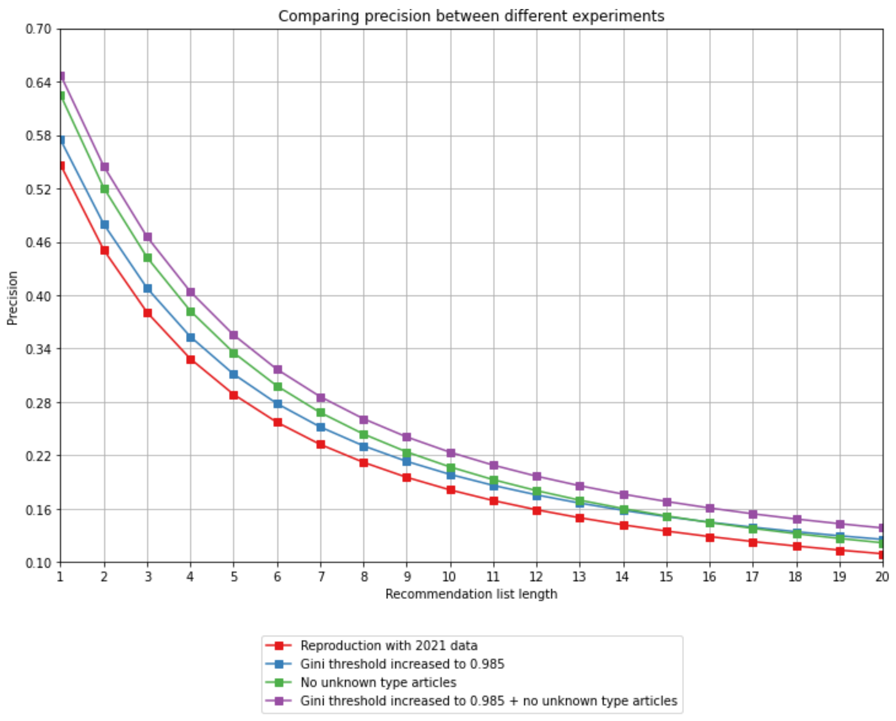


Figure 6.3: Comparation of precision between different method improvement experiments, values are shown in detail in Section A.2.

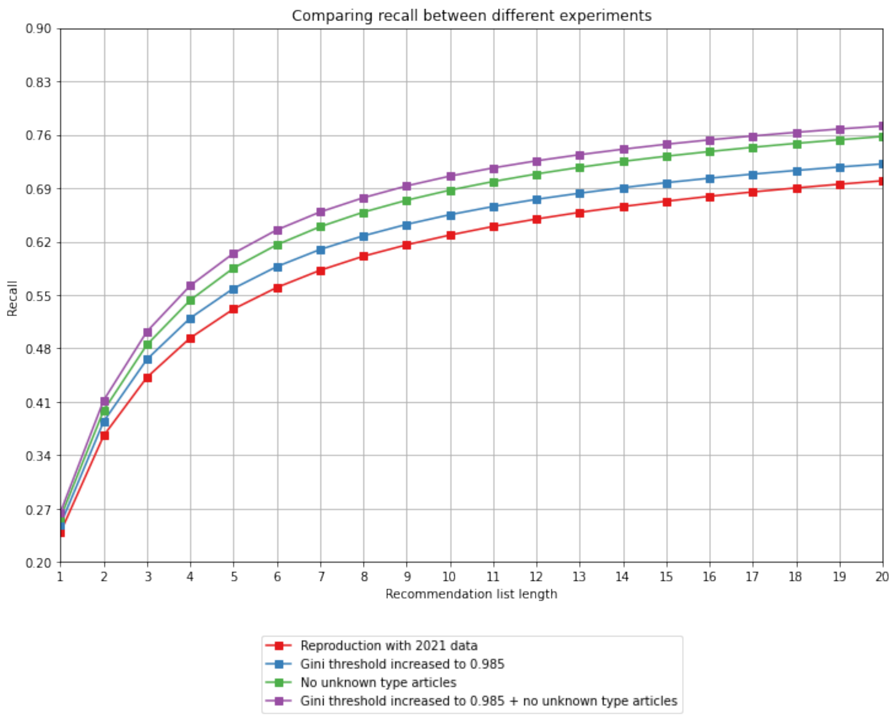


Figure 6.4: Comparation of recall between different method improvement experiments, values are shown in detail in Section A.2.

The results of this section as well as the baseline from Section 6.2 are summarized in Table 6.2.

	Precision@1	**Recall@20**
Reproduction with 2021 data	0.55	0.7
Improvement: increasing gini threshold	0.58	0.72
Improvement: filter out articles with unknown type	0.63	0.76
Improvement: filter unknown type articles and increase gini threshold	0.65	0.77

Table 6.2: Summary of the results from this chapter

6.4 Filtering out semantically similar sections

The similar section filtering was applied on the best results obtained at the previous step, *i.e.,* with data from Wikipedia's 2021 dump, after filtering out articles which had no known type, with gini coefficient threshold set to 0.985.

We tested 3 different semantic similarity filtering levels (the relationships between the cosine similarity threshold above which two sections are considered as semantically similar and semantic filtering levels is explained in Section 4.4). The test set was the same between different semantic filtering levels.

The performance of those filtering levels were compared with the performance before filtering (*i.e.,* semantic similarity filtering level 0, which is equivalent to the best improved method used in Section 6.3).

We applied the semantic similar section filtering on all articles of the test set for recommendation list sizes k=2 to 20. We omitted recommendation lists of size 1 because the goal was to filter out semantically redundant sections. Therefore we needed at least a pair of sections in the recommendation list to be able to decide if they are semantically similar.

Because our algorithm filtered out sections, the size of the recommendation lists decreased after the filtering. Therefore to be able to compare the performance after the filtering part among different filtering levels and the baseline, we used the recommendation list size after the filtering, *e.g.,* recall@x represents the mean recall after applying the filtering for the recommendation lists which had the size of x after the filtering.

Some articles of the test set could have multiple times the same set of recommended sections after filtering for different recommendation list lengths before filtering, as shown with the following example of a recommendation list before filtering:

1. Geography
2. Demographics
3. Education
4. Healthcare
5. **Civic administration**
6. Transport
7. Infrastructure
8. Culture
9. Sports
10. Notable people
11. **Civic Administration**

The recommendation list shown above was the same after filtering for the initial recommendation list sizes 10 and 11. Because the section ”Civic Administration” (at position 11) is semantically similar to ”Civic administration” (at position 5), only one of both was retained, therefore this article would contribute two times with the same value in the averages for the metrics@10.

In order to avoid such behavior, we decided to use only once each set of recommended sections after filtering for each semantic similarity filtering level for each article of the test set as shown on Algorithm 4.

Algorithm 4: Semantic similar section filtering evaluation algorithm

```
for each article to evaluate do
    for $lvl$=0 to 3 do
        $already\_evaluated\_recommendation\_lists \leftarrow \{\}$
        for $k$=2 to 20 do
            $recs\_after\_filtering \leftarrow$ get recommendations after filtering from result file, with semantic
              similarity filtering level set to $lvl$, recommendation list size before filtering set to $k$
            if $recs\_after\_filtering$ not in $already\_evaluated\_recommendation\_lists$ then
                $k\_after \leftarrow len(recs\_after\_filtering)$
                compute recall and precision between $recs\_after\_filtering$ and ground truth
                append recall and precision to recall@$k\_after$ and precision@$k\_after$
            end
            $already\_evaluated\_recommendation\_lists \leftarrow$
              $already\_evaluated\_recommendation\_lists \cup \{recs\_after\_filtering\}$
        end
    end
end
average precision and recall for $k$=1 to 20
compute f1 score based on average precision and recall for $k$=1 to 20
```

The number of values that were used to compute the average for the metrics were different depending on the semantic similarity filtering level and the recommendation list length after filtering (because some articles had multiple times the same set of recommended section lists after filtering that were not taken into account as shown on Algorithm 4, and some articles had different recommended section lists after filtering but with the same size), therefore they are shown on Table A.14 in the Appendix.

First, we compared the precision among different filtering levels as shown on Figure 6.5. Detecting and filtering out more sections as semantically similar (*i.e.,* increasing the filtering level) increased the precision, especially for longer recommendation lists (*i.e.,* with 10 or more sections) where the precision was higher than the baseline. This was predictable because filtering out more sections meant that potential noise in the recommendations was reduced.

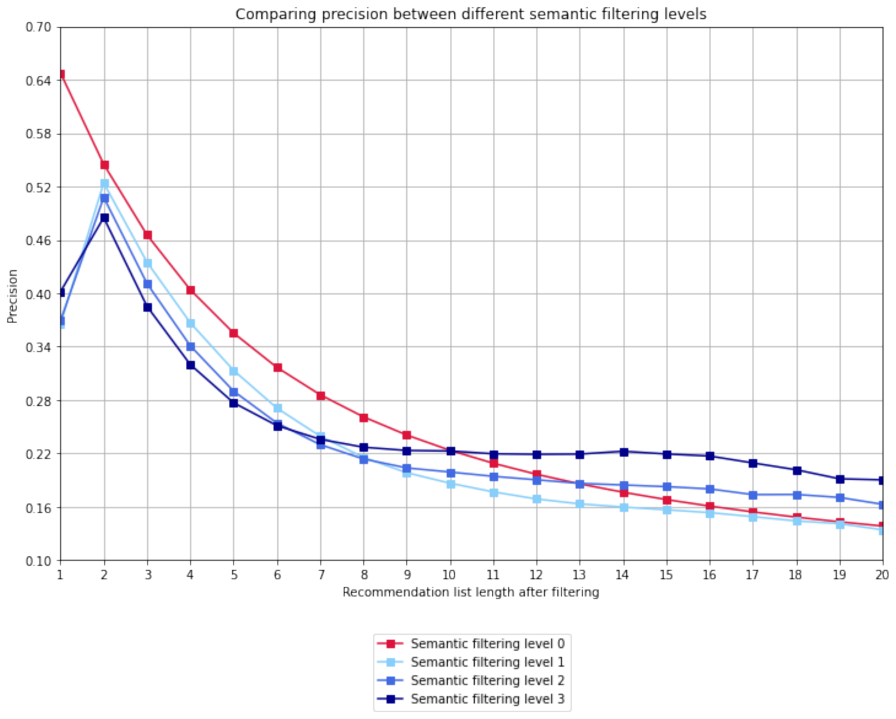


Figure 6.5: Comparison of precision among semantic similar levels, values are shown in detail in Section A.3

Then we compared the recall as shown on Figure 6.6. The recall was lower than the baseline in all cases, because detecting sections as similar and filtering them out meant that potentially sections from the ground truth would be filtered out.

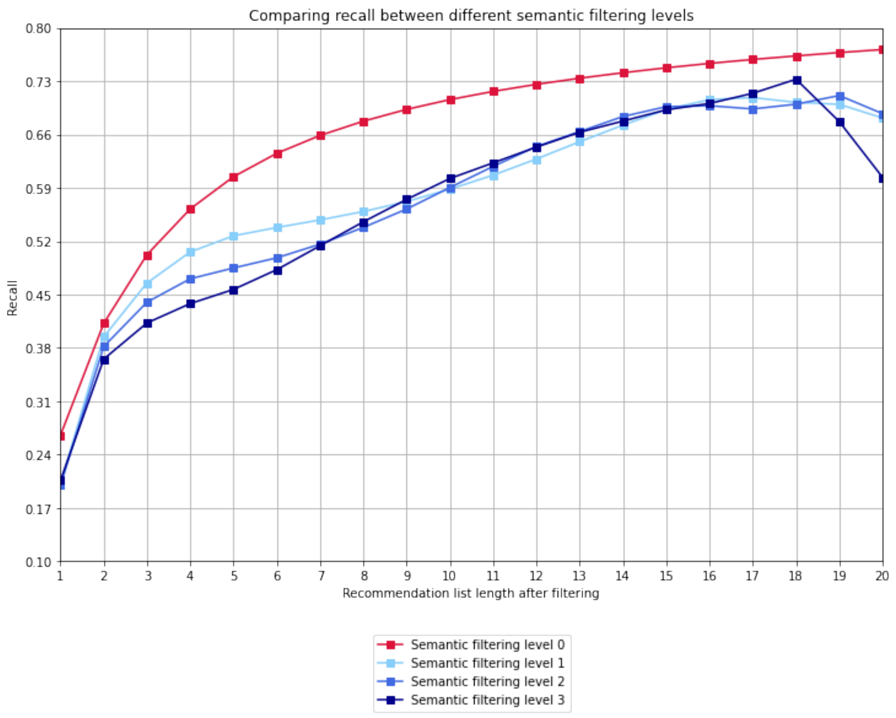


Figure 6.6: Comparison of recall among semantic similar levels, values are shown in detail in Section A.3

Finally by looking at the F1-score as shown on Figure 6.7 we could measure the accuracy of the system after applying our filtering algorithm.

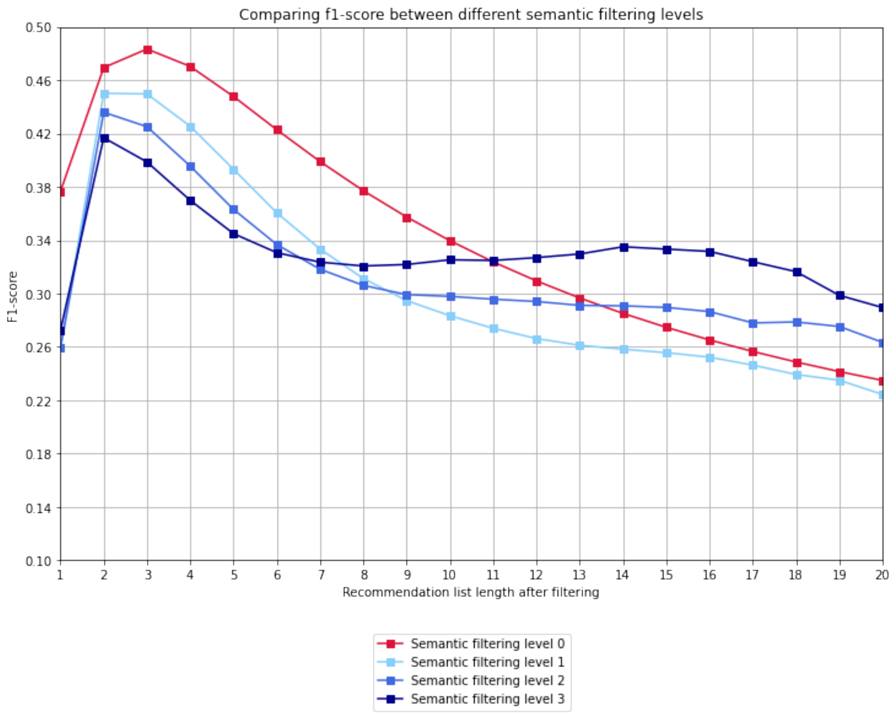


Figure 6.7: Comparison of F1 score among semantic similar levels, values are shown in detail in Section A.3

For recommendation list sizes below 11 the F1-score was better without filtering, but from recommendation list sizes 11 to 20 the F1-score was higher with the filtering than without. This means that the performance of the recommendation algorithm is improved by the filtering for longer (*i.e.,* at least of size 11) recommendation lists. It means that 11 is in average the minimum number of recommended sections where there is enough noise that is detected and removed by our algorithm.

The semantic filtering level 3 had the worst F1-score for short recommendation lists (*i.e.,* 6 and under) among different semantic filtering levels but it became the best in terms of F1-score for longer recommendation lists (*i.e.,* 11 and above). Semantic filtering level 2 becomes better than the baseline for recommendation list sizes 14 to 20 but has a lower performance than semantic filtering level 3. Semantic filtering level 1 performs never better than the baseline.

We can conclude that in terms of performance it is worth to filter out semantically similar sections if the recommendation list has a size of at least 11, but for shorter recommendation lists the system is more accurate without semantical similar section filtering.

6.5 Section ordering

Kendall's tau values were averaged over the test set after comparing the ordering between the ground truth and the recommendations.

Because Kendall's tau coefficient can only be applied on lists having the same size and same elements, it was applied on the sections of the recommendation list that appeared in the ground truth (*e.g.,* if the ground truth had the sections ABCD and the recommendation list BCEF, only the ordering of sections BC were compared between both lists).

If a recommendation list had not at least 2 sections in common with the ground truth, it wasn't included in the result because no ordering comparison was possible.

Additionally, it was decided to evaluate the ordering of recommendation lists which had a recommendation list size of 20 before filtering, in order to have more chances that the recommendation list had at least 2 sections in common with the ground truth.

The ordering quality was compared between 4 configurations, among 4 semantic filtering levels ranging from 0 to 3 (the 0-th level meaning that no filtering was applied). Those 4 configurations were as follows:

- No ordering applied.
- Representing the relative position of a section with the mean between its beginning order value and its end order value (see Figure 5.1 for an example for those order values). These order values were different for each section in each separate category.
- Representing the relative position of a section only with its beginning order value, these order values were still different for each section in each separate category.
- Representing the relative position of a section with the mean between its beginning order value and its end order value, but the context was ignored which means that those order values were the same for each section across the dataset as if all articles were in the same category.

The comparison is shown on Figure 6.8.

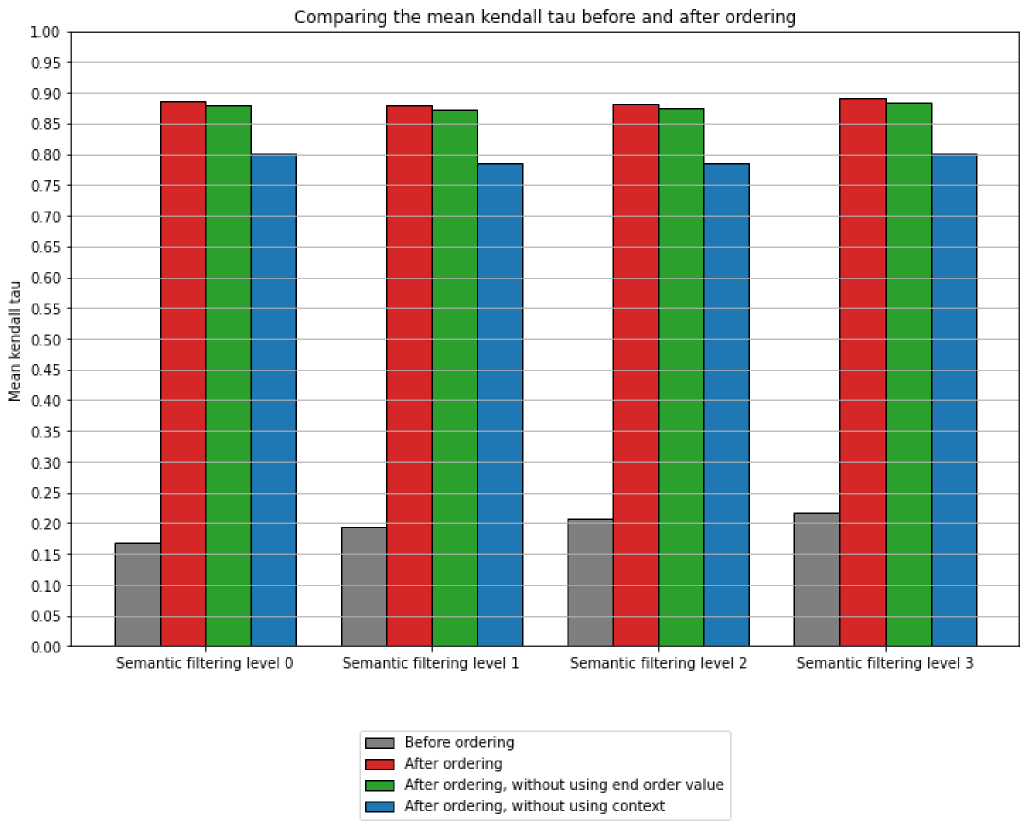


Figure 6.8: Comparison of mean kendall's tau between tested configurations for each semantic similarity filtering level, values are shown in detail in Section A.4

The differences in terms of average kendall's tau between different semantic filtering levels were small, but in general the semantic filtering level 3 had the best results and level 1 had the worst.

The average Kendall's tau value for the recommendation list before the ordering ranged from 0.169 for semantic filtering level 0, to 0.22 for semantic filtering level 3.

With the proposed ordering method, the average Kendall's tau value ranged from 0.88 for semantic filtering level 1, to 0.89 for semantic filtering level 3.

By sorting the ordered list only by the average beginning order value, the average kendall's tau ranged from 0.87 for semantic filtering level 1 to 0.88 for semantic filtering level 3. This shows that the first potential problem that was formulated, *i.e.,* that the table of contents could have varying sizes with the same section occupying different proportions of the table of content in different articles, has little effect on the quality of the ordering.

If we ignore the context of the articles and do not aggregate the ordering values by category, the mean tau ranged from 0.78 for semantic filtering level 1 to 0.8 for semantic filtering level 3. Which means that the orderings for the sections depended on the context and the same set of sections in different contexts had not necessary the same order.

When the semantic filtering level increased, there were less recommendation lists which had at least 2 sections in common with the ground truth, therefore the number of kendall tau values that were averaged decreased. The number of values that were used to compute the average kendall tau are shown on Table 6.3.

Semantic filtering level	**Number of values used to compute the average kendall tau**
0	215'660
1	166'053
2	148'275
3	133'129

Table 6.3: Number of values used to compute the average kendall tau, by semantic filtering level

7 Prototype

A prototype was developed in the form of a web application, in order to demonstrate how this project could be helpful in practice for an end user.

It can be downloaded at this address: https://drive.switch.ch/index.php/s/tIsib3HvAVVTsub . The installation instructions are described in the README.md file which is in the root folder. The archive includes the web application which can be deployed with docker compose and the mongodb dumps needed for the application.

On the starting page is located a list of already setup examples as shown on Figure 7.1. Their purpose is to provide a coherent combination of categories that could be found in the dataset. Whereas the ”Choose categories” button brings the user to the category selection page.

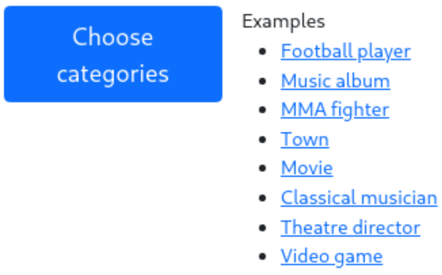


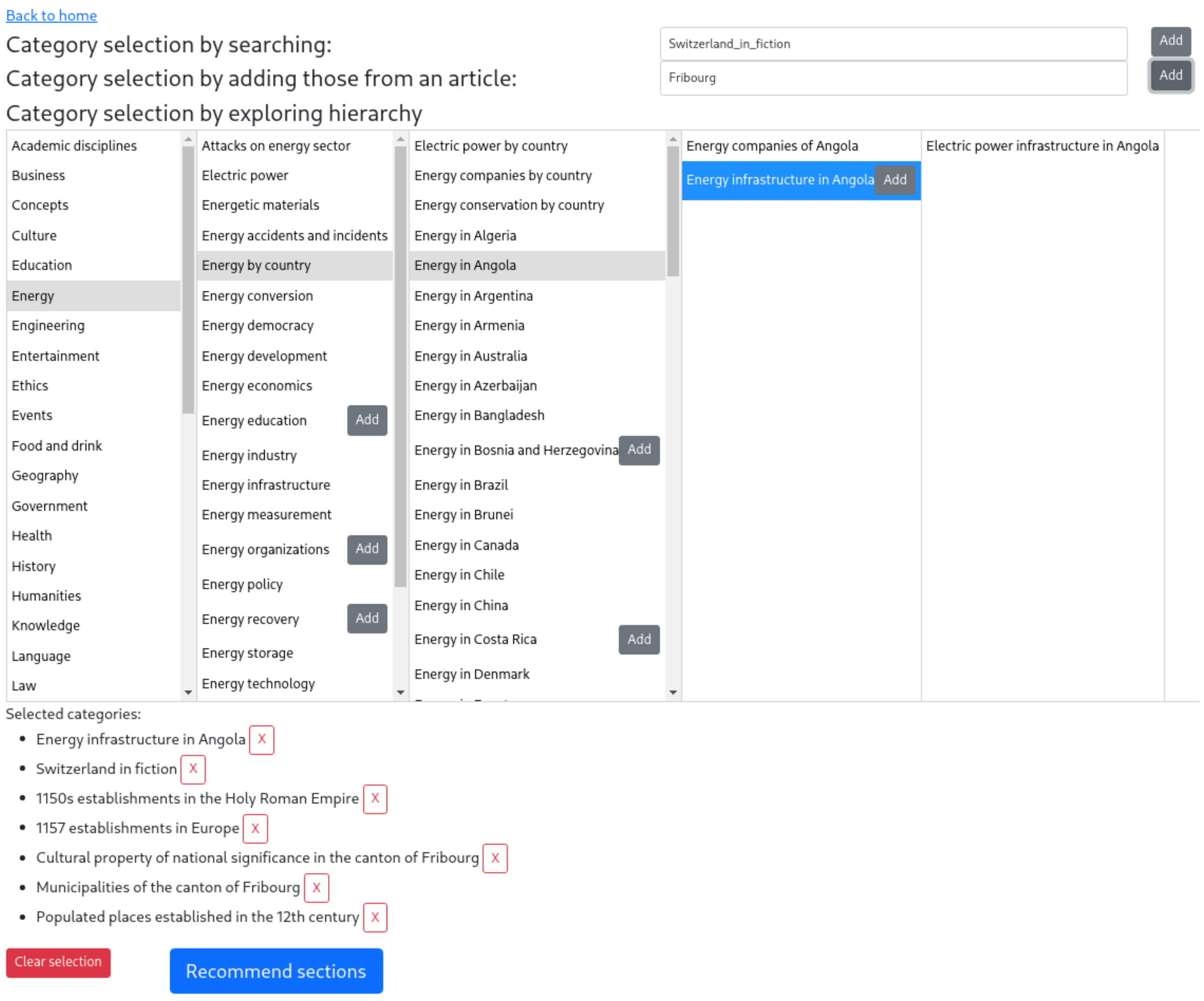
Figure 7.1: Home page

The category selection page allows the user to select a list of categories for which the recommendations are wanted as shown on Figure 7.2. Only categories which have section counts can be added (*i.e.,* those which were ontologically pure enough after the pruning algorithm).

There are three ways to add categories to the list:

- At the top of the page, search for a category by its name (the query term must be a full word which is present in a category name having section counts, and must be between delimiters such as spaces, *e.g.,* ”switzerl” will have no results, you have to type ”switzerland” in order to be able to select ”Telecommunications in Switzerland”), click on the ”Add” button on the right of the search field to add the category to the selected category list.
- Just below, there is an option to search for an existing article from the dataset. With this option, the categories to which the selected article belongs are added. The constraints are the same as for the category search option, *i.e.,* the query must contain full words which could be found in an article title.
- Navigate in the category hierarchy at the center of the page, clicking on a category will open a new column on the right with its subcategories, click on the ”Add” button located to the right of a category name to add the desired category to the selected category list.

Selected categories can be removed from the list on the bottom of the page by clicking on the red cross near the category name or with the ”Clear selection” button which will remove all the selected categories from the list.



Selected categories:

- Energy infrastructure in Angola X
- Switzerland in fiction X
- 1150s establishments in the Holy Roman Empire X
- 1157 establishments in Europe X
- Cultural property of national significance in the canton of Fribourg X
- Municipalities of the canton of Fribourg X
- Populated places established in the 12th century X

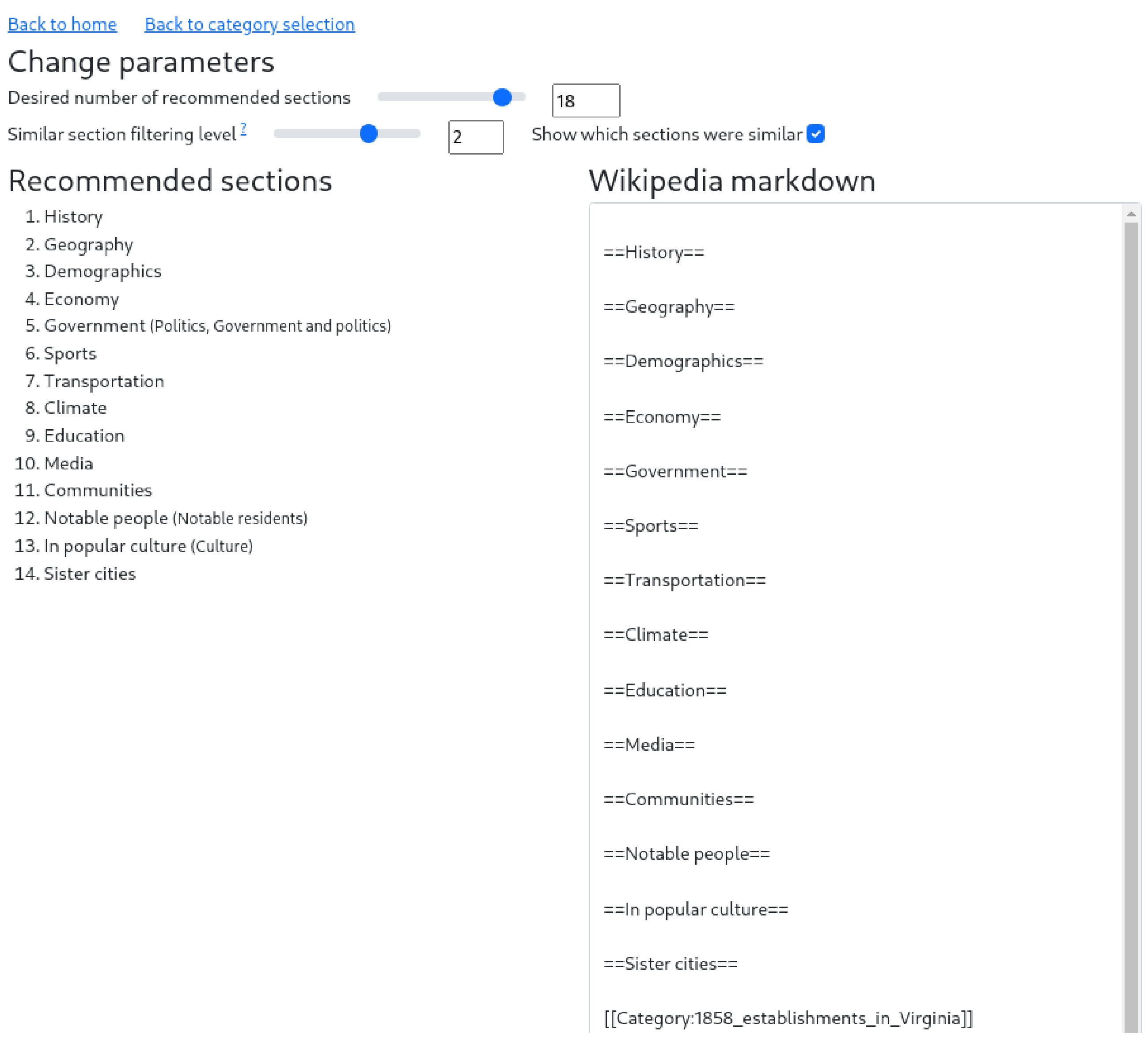
Figure 7.2: Category selection page, ”Energy infrastructure in Angola” was selected by navigating the category hierarchy, ”Switzerland in fiction” was selected by searching the category name, the remaining categories were selected by searching the ”Fribourg” Wikipedia article.

The recommendation page (shown on Figure 7.3) contains the list of recommended sections for the selected categories on the left part of the page. By default after the category selection, the size of the recommendation list is the most common article length among the selected categories. On the right part of the page is located a text area which contains Wikipedia markdown that can be copied and pasted into the Wikipedia editor in order to create an article containing the recommended sections and the selected categories.

The user can vary the desired size of the recommendation list.

The user can also vary the similar section filtering level, which ranges from 0 to 3. If it is set to 0, no similar section filtering is done. If this value is increased, more sections will be detected as similar and filtered out.

The ”show which sections were similar” checkbox allows to see which sections were detected as semantically similar. If this option is checked, sections that were filtered appear alongside the retained sections on the left part of the page (they are not included in the Wikipedia markdown), *e.g.,* ”(Notable residents)” appear alongside ”Notable people” because both sections were detected as being semantically similar, but the ”Notable residents” section was filtered out.



Back to home Back to category selection

Change parameters

Desired number of recommended sections 18

Similar section filtering level ? 2 Show which sections were similar ✓

Recommended sections

1. History
2. Geography
3. Demographics
4. Economy
5. Government (Politics, Government and politics)
6. Sports
7. Transportation
8. Climate
9. Education
10. Media
11. Communities
12. Notable people (Notable residents)
13. In popular culture (Culture)
14. Sister cities

Wikipedia markdown

```
==History==

==Geography==

==Demographics==

==Economy==

==Government==

==Sports==

==Transportation==

==Climate==

==Education==

==Media==

==Communities==

==Notable people==

==In popular culture==

==Sister cities==

[[Category:1858_establishments_in_Virginia]]
```

Figure 7.3: Recommendations page, ”Town” example

8 Conclusion and future work

This project is based on an already existing Wikipedia section recommendation algorithm [1] which used sections from other articles of the same categories in order to recommend sections to a given article. This algorithm leverages Wikipedia's category network which contains child-parent relationships between categories, by enriching recommendations for a given category with sections from articles coming from this category's child categories.

While reproducing the algorithm, we noticed that some articles from child categories that were used to enrich recommendations for articles belonging to parent categories added noise to the sections recommendations. By filtering out those articles, we increased precision@1 from 0.55 to 0.65 and recall@20 from 0.7 to 0.77.

We also implemented two improvements suggested by the algorithm's authors:

- We developed a method that filters semantically redundant sections from the recommendation lists, *e.g.,* if sections ”Career” and ”Later career” are both present in a section recommendation list, they are detected as being semantically similar and only one of them is retained. The performance of the algorithm improved in terms of f1 score@20 from 0.235 to 0.29.
- We introduced a logical ordering to the section recommendation lists. In the original algorithm, sections are ranked in the recommendation lists by their frequency because the algorithm recommends the most frequent sections that appear in a set of categories, which means that *e.g.,* the section ”Conclusion” could be ranked before ”Introduction” if the first appeared more frequently than the second. Thanks to our introduction of logical ordering, the ”Introduction” section would be ranked before the ”Conclusion” section in the recommendation list. We improved the average Kendall's tau correlation coefficient (which was used to compare the ordering between recommendation lists and the ground truth) from 0.22 to 0.89.

Finally, we implemented a prototype to demonstrate how this algorithm could help an user to create new Wikipedia articles. This prototype allows the user to select one or multiple categories and based on this set of categories a structure for the table of contents is recommended. This prototype also outputs a template containing the recommended sections ready to be used in the Wikipedia editor.

8.1 Future work

We propose some potential improvements to our project and things to explore in the future:

- A dataset containing groups of sections labeled as semantically similar would be helpful to quantify how well our method performs in terms of semantically similar section detection. This would also allow to fine tune some hyperparameters used in our filtering method.

- We used the Jaccard similarity in some parts of this project, *e.g.,* to quantify the number of articles in common between categories before grouping them. Other ways of computing the similarity in such cases could be used.
- Other strategies could be used to measure the semantic similarity between sections. Instead of using the beginning of section contents to measure the similarity, the whole section's content could be divided into multiple parts before comparing those parts with parts from other sections. This would require a method to aggregate the similarities between section parts from different sections. More semantically similar sections could be detected this way.
- The semantically similar sections detection part could also be extended to other languages than English, by using models specific to a given language or multilingual models.

A
Data used for plots

This chapter of the appendix contains tables with values that were not shown in detail beforehand but were shown as plots. This data comes from the `data/results` folder in the project repository.

A.1 Paper reproduction

Values from this section are used in Figure 6.1 (comparing precision), Figure 6.2 (comparing recall).

Table A.1 shows in which result folder are contained values shown in the tables of this section. In each folder, the recommendations are in the `recs_by_article.json` file.

Table	file
Table A.2	Provided by initial paper's authors, a link is provided in our github repo's `README.md`
Table A.3	`epfl_reproduction_provided_category_section_counts`
Table A.4	`epfl_reproduction-category_section_counts_with_test_set`
Table A.5	`epfl_reproduction`
Table A.6	`gini_threshold-0966`

Table A.1: Result folder where values from this section are from

k	Mean precision	Mean recall
1	0.574	0.257
2	0.482	0.400
3	0.412	0.488
4	0.358	0.547
5	0.317	0.593
6	0.285	0.628
7	0.259	0.656
8	0.238	0.680
9	0.220	0.699
10	0.205	0.716
11	0.192	0.730
12	0.181	0.742
13	0.171	0.753
14	0.162	0.763
15	0.155	0.771
16	0.148	0.779
17	0.142	0.786
18	0.136	0.792
19	0.131	0.798
20	0.126	0.804

Table A.2: Results provided by authors of the paper

k	Mean precision	Mean recall
1	0.577	0.259
2	0.483	0.401
3	0.413	0.489
4	0.359	0.548
5	0.318	0.594
6	0.286	0.629
7	0.26	0.657
8	0.239	0.681
9	0.221	0.7
10	0.206	0.717
11	0.193	0.731
12	0.182	0.743
13	0.172	0.754
14	0.163	0.764
15	0.155	0.773
16	0.149	0.78
17	0.142	0.787
18	0.137	0.794
19	0.132	0.8
20	0.127	0.805

Table A.3: Reproduction of the results using precomputed category section counts provided by paper’s authors

k	Mean precision	Mean recall
1	0.594	0.27
2	0.501	0.42
3	0.431	0.514
4	0.374	0.576
5	0.332	0.623
6	0.298	0.66
7	0.271	0.689
8	0.249	0.714
9	0.23	0.734
10	0.214	0.752
11	0.2	0.767
12	0.188	0.78
13	0.178	0.792
14	0.169	0.802
15	0.161	0.811
16	0.153	0.82
17	0.147	0.827
18	0.141	0.834
19	0.136	0.841
20	0.131	0.847

Table A.4: Reproduction of the method with 2017 data, with test set included in train set

k	Mean precision	Mean recall
1	0.530	0.237
2	0.434	0.360
3	0.366	0.433
4	0.315	0.483
5	0.276	0.518
6	0.246	0.545
7	0.222	0.566
8	0.203	0.584
9	0.187	0.598
10	0.173	0.611
11	0.162	0.621
12	0.152	0.631
13	0.143	0.639
14	0.136	0.647
15	0.129	0.654
16	0.124	0.660
17	0.118	0.665
18	0.114	0.671
19	0.109	0.675
20	0.105	0.680

Table A.5: Reproduction of the method with 2017 data, without test set included in train set

k	Mean precision	Mean recall
1	0.547	0.239
2	0.451	0.366
3	0.381	0.442
4	0.329	0.494
5	0.289	0.532
6	0.257	0.56
7	0.232	0.582
8	0.212	0.601
9	0.195	0.616
10	0.18	0.628
11	0.16	0.64
12	0.15	0.65
13	0.15	0.658
14	0.14	0.666
15	0.13	0.673
16	0.12	0.679
17	0.12	0.685
18	0.11	0.69
19	0.11	0.695
20	0.11	0.7

Table A.6: Reproduction of the method with 2021 Wikipedia data

A.2 Method improvements

Values from this section are used in Figure 6.3 (comparing precision), Figure 6.4 (comparing recall). Those figures also contain values from the baseline (*i.e.,* reproduction with 2021 data), which are shown on Table A.6.

Table A.7 shows in which result folder are contained values shown in the tables of this section. In each folder, the recommendations are in the `recs_by_article.json` file.

Table	file
Table A.8	`gini_threshold-0966_no_unknown_types`
Table A.9	`gini_threshold-0985`
Table A.10	`gini_threshold-0985_no_unknown_types`

Table A.7: Result folder where values from this section are from

k	Mean precision	Mean recall
1	0.625	0.258
2	0.521	0.399
3	0.443	0.485
4	0.382	0.543
5	0.336	0.585
6	0.298	0.616
7	0.268	0.639
8	0.244	0.658
9	0.224	0.674
10	0.207	0.687
11	0.193	0.699
12	0.18	0.709
13	0.17	0.717
14	0.16	0.725
15	0.152	0.732
16	0.145	0.738
17	0.138	0.743
18	0.132	0.749
19	0.127	0.753
20	0.122	0.758

Table A.8: Reproduction of the method with 2021 Wikipedia data, after filtering out articles which had an unknown type

k	Mean precision	Mean recall
1	0.579	0.250
2	0.484	0.386
3	0.412	0.468
4	0.356	0.522
5	0.314	0.561
6	0.280	0.590
7	0.254	0.612
8	0.232	0.630
9	0.215	0.645
10	0.200	0.658
11	0.188	0.669
12	0.177	0.678
13	0.167	0.686
14	0.159	0.693
15	0.152	0.700
16	0.146	0.706
17	0.140	0.711
18	0.135	0.716
19	0.130	0.720
20	0.126	0.724

Table A.9: Reproduction of the method with 2021 Wikipedia data, with gini coefficient threshold set to 0.985

k	Mean precision	Mean recall	Mean F1 score
1	0.649	0.264	0.376
2	0.547	0.411	0.469
3	0.468	0.502	0.483
4	0.406	0.562	0.47
5	0.357	0.604	0.448
6	0.318	0.635	0.423
7	0.287	0.659	0.399
8	0.262	0.678	0.377
9	0.242	0.694	0.357
10	0.224	0.707	0.34
11	0.210	0.718	0.324
12	0.197	0.727	0.31
13	0.186	0.735	0.297
14	0.177	0.742	0.285
15	0.168	0.749	0.275
16	0.161	0.755	0.265
17	0.154	0.760	0.257
18	0.149	0.764	0.249
19	0.143	0.769	0.242
20	0.138	0.773	0.235

Table A.10: Reproduction of the method with 2021 Wikipedia data, after filtering out articles which had an unknown type, with gini coefficient threshold set to 0.985

A.3 Detect and filter semantically similar sections

Values from this section are used in Figure 6.5 (comparing precision), Figure 6.6 (comparing recall) and Figure 6.7 (comparing f1 score).

Values for semantic filtering level 0 are those from Table A.10.

Results from this section are in the `gini_threshold-0985_no_unknown_types/filtered_recs_by_article.json` file from the results folder.

k	Mean precision	Mean recall	Mean F1 score
1	0.365	0.201	0.259
2	0.524	0.395	0.45
3	0.435	0.465	0.45
4	0.367	0.506	0.425
5	0.314	0.527	0.393
6	0.271	0.538	0.361
7	0.24	0.548	0.333
8	0.216	0.559	0.311
9	0.199	0.572	0.295
10	0.187	0.588	0.283
11	0.177	0.607	0.274
12	0.169	0.628	0.266
13	0.163	0.65	0.261
14	0.16	0.672	0.258
15	0.157	0.693	0.256
16	0.154	0.706	0.252
17	0.149	0.708	0.246
18	0.144	0.702	0.239
19	0.141	0.699	0.235
20	0.134	0.682	0.224

Table A.11: Performance of the system after applying semantic filtering, with semantical similarity filtering level 1

k	Mean precision	Mean recall	Mean F1 score
1	0.369	0.2	0.259
2	0.508	0.382	0.436
3	0.411	0.44	0.425
4	0.341	0.471	0.396
5	0.29	0.485	0.363
6	0.254	0.498	0.337
7	0.23	0.516	0.318
8	0.214	0.538	0.306
9	0.204	0.562	0.299
10	0.199	0.59	0.298
11	0.194	0.618	0.296
12	0.191	0.644	0.294
13	0.186	0.664	0.291
14	0.185	0.683	0.291
15	0.183	0.696	0.29
16	0.18	0.698	0.287
17	0.174	0.693	0.278
18	0.174	0.7	0.279
19	0.171	0.711	0.275
20	0.163	0.687	0.263

Table A.12: Performance of the system after applying semantic filtering, with semantical similarity filtering level 2

k	Mean precision	Mean recall	Mean F1 score
1	0.402	0.206	0.272
2	0.485	0.365	0.417
3	0.385	0.413	0.399
4	0.32	0.438	0.37
5	0.277	0.457	0.345
6	0.251	0.483	0.331
7	0.236	0.514	0.324
8	0.227	0.545	0.321
9	0.223	0.575	0.322
10	0.223	0.602	0.325
11	0.22	0.623	0.325
12	0.219	0.643	0.327
13	0.219	0.663	0.33
14	0.223	0.678	0.335
15	0.22	0.692	0.333
16	0.217	0.701	0.332
17	0.209	0.714	0.324
18	0.202	0.732	0.316
19	0.192	0.677	0.299
20	0.19	0.604	0.29

Table A.13: Performance of the system after applying semantic filtering, with semantical similarity filtering level 3

k	level 0	level 1	level 2	level 3
1	343083	18058	32062	55041
2	343083	366083	387897	429152
3	343083	392135	444187	521283
4	343083	405875	486534	581700
5	343083	421429	516820	593617
6	343083	431290	512204	516443
7	343083	430828	461700	390392
8	343083	414692	386518	273552
9	343083	385098	304486	191186
10	343083	348253	235951	133139
11	343083	301056	176865	84410
12	343083	248993	129434	50880
13	343083	200418	91583	29125
14	343083	157549	62190	16115
15	343083	120069	38278	8351
16	343083	84539	20571	4230
17	343083	51835	9557	2144
18	343083	26250	4125	1067
19	343083	10999	1604	337
20	343083	2863	359	115

Table A.14: Number of values used to compute average of metrics as shown in Algorithm 4, for each recommendation list length after filtering, by semantic similarity filtering level

A.4 Section ordering

Values from this section are used in Figure 6.8.

Table A.15 shows in which file are contained values shown in the tables of this section. All those files are in the `gini_threshold-0985_no_unknown_types` folder from the results.

Table	file
Table A.16	All 3 files below (the values were all the same)
Table A.17	`ordered_recs_by_article.json`
Table A.18	`ordered_recs_by_article_ignore_context.json`
Table A.19	`ordered_recs_by_article_ignore_end_order_value.json`

Table A.15: Result folder where values from this section are from

Semantic similarity filtering level	Mean tau before ordering
0	0.169
1	0.195
2	0.208
3	0.217

Table A.16: Mean kendall's tau, before ordering, by semantic similarity filtering level

Semantic similarity filtering level	Mean tau after ordering
0	0.886
1	0.880
2	0.882
3	0.892

Table A.17: Mean kendall's tau, after ordering by mean between beginning and end order values different by category for each section, by semantic similarity filtering level

Semantic similarity filtering level	Mean tau after ordering
0	0.801
1	0.786
2	0.786
3	0.800

Table A.18: Mean kendall's tau, after ordering by mean between beginning and end order values for each section without grouping them by category, by semantic similarity filtering level

Semantic similarity filtering level	Mean tau after ordering
0	0.880
1	0.872
2	0.875
3	0.884

Table A.19: Mean kendall's tau, after ordering with only beginning order values different by category for each section, by semantic similarity filtering level

Bibliography

[1] Tiziano Piccardi, Michele Catasta, Leila Zia, and Robert West. Structuring wikipedia articles with section recommendations. In *The 41st International ACM SIGIR Conference on Research & Development in Information Retrieval*, pages 665–674, 2018.

[2] Sören Auer, Christian Bizer, Georgi Kobilarov, Jens Lehmann, Richard Cyganiak, and Zachary Ives. Dbpedia: A nucleus for a web of open data. In *The semantic web*, pages 722–735. Springer, 2007.

[3] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.

[4] Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. Roberta: A robustly optimized bert pretraining approach, 2019.

[5] Nils Reimers and Iryna Gurevych. Sentence-bert: Sentence embeddings using siamese bert-networks. *CoRR*, abs/1908.10084, 2019.

[6] Victor Sanh, Lysandre Debut, Julien Chaumond, and Thomas Wolf. Distilbert, a distilled version of bert: smaller, faster, cheaper and lighter. *ArXiv*, abs/1910.01108, 2019.

[7] Aaron Gokaslan and Vanya Cohen. Openwebtext corpus. `http://Skylion007.github.io/OpenWebTextCorpus`, 2019.

[8] Samuel R. Bowman, Gabor Angeli, Christopher Potts, and Christopher D. Manning. A large annotated corpus for learning natural language inference. In *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing (EMNLP)*. Association for Computational Linguistics, 2015.

[9] Alexis Conneau, Douwe Kiela, Holger Schwenk, Loic Barrault, and Antoine Bordes. Supervised learning of universal sentence representations from natural language inference data. *arXiv preprint arXiv:1705.02364*, 2017.

[10] Daniel Cer, Mona Diab, Eneko Agirre, Inigo Lopez-Gazpio, and Lucia Specia. Semeval-2017 task 1: Semantic textual similarity-multilingual and cross-lingual focused evaluation. *arXiv preprint arXiv:1708.00055*, 2017.

[11] Vincent D Blondel, Jean-Loup Guillaume, Renaud Lambiotte, and Etienne Lefebvre. Fast unfolding of communities in large networks. *Journal of statistical mechanics: theory and experiment*, 2008(10):P10008, 2008.

[12] Thomas Aynaud. python-louvain x.y: Louvain algorithm for community detection. `https://github.com/taynaud/python-louvain`, 2020.

[13] Aric A. Hagberg, Daniel A. Schult, and Pieter J. Swart. Exploring network structure, dynamics, and function using networkx. In Gaël Varoquaux, Travis Vaught, and Jarrod Millman, editors, *Proceedings of the 7th Python in Science Conference*, pages 11 – 15, Pasadena, CA USA, 2008.

[14] Aria Haghighi and Lucy Vanderwende. Exploring content models for multi-document summarization. In *Proceedings of human language technologies: The 2009 annual conference of the North American Chapter of the Association for Computational Linguistics*, pages 362–370, 2009.

[15] Mirella Lapata. Automatic evaluation of information ordering: Kendall's tau. *Computational Linguistics*, 32(4):471–484, 2006.

[16] Maurice G Kendall. A new measure of rank correlation. *Biometrika*, 30(1/2):81–93, 1938.

[17] Pauli Virtanen, Ralf Gommers, Travis E. Oliphant, Matt Haberland, Tyler Reddy, David Cournapeau, Evgeni Burovski, Pearu Peterson, Warren Weckesser, Jonathan Bright, Stéfan J. van der Walt, Matthew Brett, Joshua Wilson, K. Jarrod Millman, Nikolay Mayorov, Andrew R. J. Nelson, Eric Jones, Robert Kern, Eric Larson, C J Carey, İlhan Polat, Yu Feng, Eric W. Moore, Jake VanderPlas, Denis Laxalde, Josef Perktold, Robert Cimrman, Ian Henriksen, E. A. Quintero, Charles R. Harris, Anne M. Archibald, Antônio H. Ribeiro, Fabian Pedregosa, Paul van Mulbregt, and SciPy 1.0 Contributors. SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python. *Nature Methods*, 17:261–272, 2020.

[18] Maurice G Kendall. The treatment of ties in ranking problems. *Biometrika*, 33(3):239–251, 1945.

[19] Peter Eades, Xuemin Lin, and William F Smyth. A fast and effective heuristic for the feedback arc set problem. *Information Processing Letters*, 47(6):319–323, 1993.

[20] Giuseppe Attardi. Wikiextractor. `https://github.com/attardi/wikiextractor`, 2015.

[21] Steven Bird, Ewan Klein, and Edward Loper. *Natural language processing with Python: analyzing text with the natural language toolkit.* " O'Reilly Media, Inc.", 2009.

[22] Krzysztof Fiok, Waldemar Karwowski, Edgar Gutierrez-Franco, Mohammad Reza Davahli, Maciej Wilamowski, Tareq Ahram, Awad Al-Juaid, and Jozef Zurada. Text guide: Improving the quality of long text classification by a text selection method based on feature importance. *IEEE Access*, 9:105439–105450, 2021.