Visual Search: A Keyword-Driven Hierarchical Document Navigation System

Dhruv Saligram
UIUC
Champaign, USA
dhruvks2@illinois.edu

Abstract

This paper presents a modular and scalable pipeline for organizing textual documents hierarchically using a keyword-based strategy. Implemented as a feature on the TextData [3] platform, this system addresses the limitations of the existing graph-based visualization by offering an intuitive semantic structure for navigating community-submitted documents. The method combines manual tagging with automated keyword extraction, primarily through Rakun, and applies a multi-stage merging process – utilizing both linguistic heuristics and large language models (LLMs) – to standardize terminology across documents. Once keywords are unified, a greedy set cover algorithm is applied recursively to build a multi-level tree that orders documents from general to specific themes. This hierarchy is then visualized via a React-based frontend that emulates a folder navigation system, making large-scale document exploration seamless and user-friendly. Qualitative evaluation on over 250 documents within the TextData environment demonstrates improved discoverability, reduced interface clutter, and enhanced navigation compared to the prior system. This helps motivate the overarching pipeline as one that could be applicable to any set of documents in a community across applications.

Keywords

Visual Search, Keywords, Keyword Extraction, Organization

ACM Reference Format:
Dhruv Saligram. 2025. Visual Search: A Keyword-Driven Hierarchical Document Navigation System. In *Proceedings of CS 510 (TextData Visual Search)*. ACM, New York, NY, USA, 6 pages.

1 Introduction & Motivation

The TextData platform fosters community collaboration by enabling individuals to create, explore, and interact with user submissions. In the context of our course, this platform has been used regularly as the central hub for collecting ideas, sharing resources, and brainstorming project topics. However, its current visualization tool – a sprawling graph layout – suffers from substantial usability shortcomings. While the visualization graph attempts to capture relationships between documents, its visual presentation lacks coherence and often becomes cluttered when scaled. As the number of documents increases, the visualization becomes overwhelming, providing little semantic guidance or intuitive browsing paths. Additionally, documents have connections drawn between other nodes representing their related questions or other mentioned documents. These connections crossing over other nodes only leads to more confusion and clutter. Finally, the organization of these documents leaves the user with many questions. Extremely similar documents can often be found on separate ends of the graph, with local clusterings seemingly arbitrary. For context, Figure 1 showcases the current visualization feature on TextData.

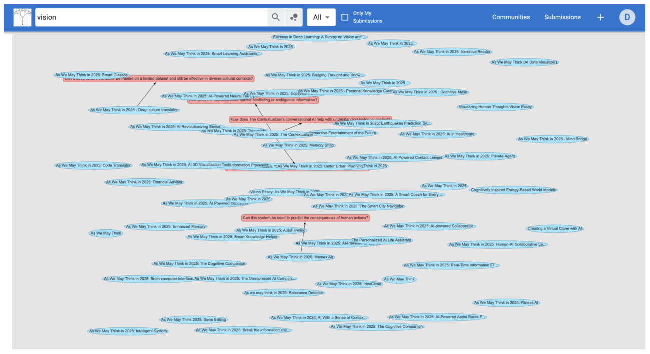

Figure 1: Current TextData visualization

Our class heavily relied on exploring submissions within TextData. From upvoting other students' vision essays to finding frontier topic videos to watch, almost every major component of the course relied on navigating TextData submissions in some form. Ultimately, this exploration process became tedious and inefficient. Other communities that gather many documents often suffer from the same problem. Motivated by these shortcomings, this project was conceived as a comprehensive restructuring of the way users interact with and understand document collections. The core objective was to design a new system capable of organizing documents hierarchically, based on their underlying content and thematic associations. Such a system should support both casual exploration and precise navigation while preserving semantic relationships between documents.

The concept eventually transformed into the Visual Search feature – a reimagining of document navigation based on keyword-driven hierarchies. The system aims to enhance user experience by enabling easier access to relevant information, facilitating discovery of emergent themes, and centralizing knowledge within a scalable and semantically structured interface. Beyond ease of exploration for individual users, the feature provides administrators with tools for content clustering, moderation, and boosting user engagement. The solution reflects a shift toward human-centered design in digital document ecosystems, merging text processing with intuitive interface design.

2 Major Functions

The Visual Search pipeline is composed of four major functions: keyword extraction, keyword merging, hierarchical tree construction, and interactive visualization. These functions are executed sequentially but maintain modular independence, allowing for future enhancements, integration into other platforms, or usage of individual functions for separate tasks.

First, a set number of relevant keywords are extracted from documents, both through manual tagging and automated NLP techniques. These keywords are then cleaned and semantically merged to eliminate duplication and increase consistency. The resulting unified keywords are used to build a hierarchical tree via a recursive greedy set cover algorithm. Finally, the constructed hierarchy is rendered using a custom-built React frontend that mimics the structure of a traditional desktop file system.

The entire system was implemented with extensibility in mind. It handles edge cases and failure as gracefully as possible and can scale across document sets of varying sizes and topic distributions.

3 Implementation Details

Each of the major functions listed in the previous section corresponded to an individual stage of the implementation. The following sections break down the implementation of each stage in detail and offer examples of how they function.

3.1 Keyword Extraction

Effective keyword extraction is the most essential task for constructing a semantically coherent hierarchy. As such, ensuring high quality keyword extraction was the first primary goal of this project. A deep dive into keyword extractors was done to determine which should be evaluated specifically for the context of this project. After analyzing online resources, it was clear that KeyBERT and RAKE should be evaluated. [1] [7] Additionally, TextBlob was discussed by the creator of TextData as a lightweight method for extracting meaningful phrases from text, which prompted its inclusion in this evaluation. Finally, Rakun was found in a comment under a keyword extraction post in a Machine Learning forum mentioned as a quick extractor. This left KeyBERT [4], RAKE [6], TextBlob [5], and Rakun [2] as the keyword extractors to be rigorously evaluated for this project.

KeyBERT is a state-of-the-art keyword extractor that leverages transformer-based language models – specifically, BERT (Bidirectional Encoder Representations from Transformers) – to identify semantically significant phrases within a text. Unlike frequency-based methods, KeyBERT generates embeddings for both the document and candidate keywords, then calculates cosine similarity scores between them. The most semantically aligned phrases (i.e. those closest in embedding space to the full document representation) are ultimately selected as keywords. This semantic analysis and high accuracy has led to widespread adoption of KeyBERT in academic and industrial NLP applications. It is particularly popular for smaller document sets where precision is critical.

RAKE (Rapid Automatic Keyword Extraction) is a classic unsupervised keyword extraction algorithm designed for speed and simplicity. It works by splitting text into candidate phrases using stopword delimiters, then scoring each phrase based on the frequency of its constituent words and their co-occurrence degrees within the document. The underlying assumption is that important keywords tend to be less connected to stopwords and more connected to other content words. RAKE is rule-based and deterministic, requiring no model training or external data. RAKE is mainly popular for lightweight keyword extraction, and is often used as a benchmark against more complex methods.

TextBlob is a general-purpose Python NLP library that includes simple noun-phrase extraction. Using part-of-speech tagging, it identifies contiguous noun phrases as candidate keywords. Unlike RAKE or KeyBERT, TextBlob does not assign weights or confidence scores to extracted phrases. The algorithm is fast but relatively shallow, as it does not account for word semantics or context beyond basic syntactic grouping. TextBlob is widely used due to its ease of use, minimal setup, and broad functionality (e.g., sentiment analysis, translation, tokenization), but it is less commonly used for keyword extraction specifically.

Finally, Rakun is a hybrid keyword extraction tool that combines rule-based candidate generation with transformer-based scoring. The process begins with an enhanced version of RAKE to identify candidate phrases based on part-of-speech patterns, stopword delimiters, and co-occurrence statistics. These candidates are then re-ranked using neural embeddings, typically via Sentence-BERT, to evaluate their contextual fit within the document. Rakun outputs keywords along with confidence scores that reflect their semantic relevance. It is a relatively newer tool that prioritizes a balance between performance and speed.

In order to test these keyword extractors for this project, 264 documents were chosen from existing TextData communities as the test set. The extractors were then evaluated on runtime and output quality.

KeyBERT had excellent output with keywords that almost perfectly captured each document's overall meaning and themes. Additionally, specifying diversity and a range for the number of words in each keyword led to even better results. Unfortunately, for the 264 documents, KeyBERT required over 50 seconds to complete its analysis.

RAKE ran almost instantaneously, able to analyze all the documents in under 0.2 seconds. However, RAKE's output was primarily phrases such as "authors use prompt engineering", "amazon beauty dataset using", "five different recommendation scenarios", "different recommendation tasks without", and "also include human evaluations." While these phrases often contained relevant information, they were too long for keywords and frequently contained stop words that harmed their overall impact.

TextBlob experienced a similar runtime to RAKE but extracted noun-phrases that were far more meaningful. However, there were two key issues with TextBlob. First, since each document in this pipeline can have only 1 to N keywords, the number of noun-phrases extracted from TextBlob had to be limited. However, TextBlob has no system for assigning relevance scores to noun-phrases, which meant that important noun-phrases were often getting sacrificed for less relevant ones. Second, TextBlob had a habit of extracting “noun-phrases” that were actually URLs, names, or random words with punctuation (characterized by its insistence that “ ’s i” was a noun phrase in one document).

Rakun analyzed all documents again in under 0.2 seconds and had meaningful extractions similar to TextBlob. While it also sometimes contained keywords that were not as relevant, on the whole, it produced high-quality output. Additionally, its inclusion of relevance scores helped quantifiably reduce the number of irrelevant keywords included. Ultimately, its speed, accuracy, and inclusion of scores led to it being chosen as the final keyword extractor for this project.

Upon further analysis, it became apparent that there were ways to improve the quality of keywords by using Rakun in combination with more manual extraction. Hashtags were often included in TextData documents – since these were specifically provided by the user, it stood to reason that these would be the most representative of the document itself. Beyond this, after extracting hashtags, they were removed from the document before Rakun analysis to prevent Rakun from identifying the same hashtags as keywords and duplicating them. Moreover, to optimize Rakun’s outputs, documents underwent a preprocessing phase wherein URLs were removed to reduce malformed keywords. This was due to a common failure case encountered during testing, where Rakun would identify sections of URLs as keywords. Rakun was then applied on the cleaned text, and only keywords exceeding a 0.15 confidence threshold were retained. If no keyword surpassed this threshold, the most confident non-hashtag keyword was included as a fallback. Next, any keywords duplicating existing hashtags were discarded to preserve keyword diversity. Finally, the list of keywords for each document was limited to 4. This choice was taken as a balance between offering documents a wide breadth of keywords and limiting the depth users would have to traverse in order to find documents.

This process ensured each document was assigned a small set of high-quality and contextually relevant keywords. The overall workflow for keyword extraction on each document is illustrated in Figure 2.

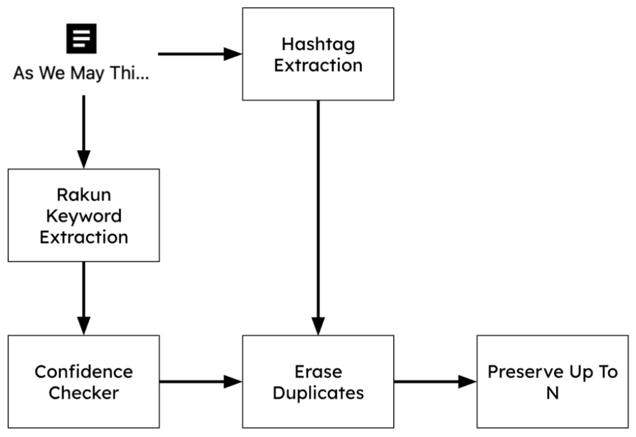

Figure 2: Keyword Extraction Workflow

3.2 Keyword Merging

The diversity in user expression – ranging from acronyms to synonyms to suffixes – led to inconsistencies among keyword sets. For instance, two users talking about the same topic included “#design” versus “#designer”. Similarly, occurrences of “#ai” and, separately, “#artificialintelligence” were found. These keywords, while clearly representing the same topics, were represented distinctly. A key challenge with extracting and then visualizing keywords is the clutter – when communities grow to encompass hundreds of documents, and especially as the number of distinct users uploading documents increases, the number of represented keywords tends to grow exponentially. The best way to combat this while still faithfully representing each document’s meaning was decided to be a two-phase keyword merging strategy. The first phase was executed automatically in Python based on linguistic heuristics, while the second leveraged the power of LLMs for semantic clustering.

In the first phase, keywords were normalized using a union-find algorithm in Python. Several transformations were applied systematically:

(1) Whitespace differences were normalized – for example, if the user included “#machinelearning” and Rakun determined “machine learning” as a keyword, these would get merged
(2) The same terms with only an ‘s’ difference at the end were merged to account for plurals (i.e. “sport” and “sports”)
(3) A WordNet lemmatizer was used to merge words based on lemmatization (merging words like “running” and “run”)
(4) An exhaustive list of suffixes was applied to further merge on lemmatization (including suffixes such as ’ion’, ’ment’, ’ness’, ’ity’, ’ty’, ’able’, and many more)
(5) Synonyms were merged based on WordNet to combine keywords that are semantically identical

This led to improved results – our initial keyword extraction with hashtags and Rakun on the 256 test documents generated 365 unique keywords. After the merges listed above, they were combined into a total of 317 unique keywords. However, despite the good performance of these merges, certain relationships – such as acronyms, paraphrases, or more complex synonyms – were not captured. Because of this, the second phase of merging was employed. This final merge revolved around using the enhanced reasoning capabilities of LLMs to cluster the remaining keywords into semantically equivalent groups. For this project, due to ease of use and gratuity, Google’s Gemini LLM was used.

The following prompt was provided to Gemini along with the unique list of keywords after the initial merge:

"""You are a keyword normalizer. Given the following list of keywords:

INSERT LIST OF UNIQUE KEYWORDS HERE

Group any keywords that are redundant by semantic meaning / synonym.

For each group, choose one canonical form. The goal is to logically pool keywords. For example: "llm", "llms", "large language", "language models", and "large language models" should all be grouped together. "ai", "ai-powered", "artificial intelligence", and "artificial intelligence systems" should also be grouped.

Make sure to match synonyms together along with acronyms that are written out.

Do this only for groups that would actually contribute to pooling keywords together (i.e. you should ignore any rare keywords and only focus on grouping ones together that are quite common, such as "ai" and "llm").

Return **only** a Python-style dict that maps a group's single canonical form to a list of all keywords in that group. For example:

input: ["llm", "personalized", "llms", "personalization", "language model", "intelligentagent", "personalize", "intelligent agent", "agent", "sports"] output: "llm": ["llm", "llms", "language model"], "personalized": ["personalized", "personalization", "personalize"], "intelligent agent": ["intelligentagent", "intelligent agent", "agent"]

"sports" is ignored as a key in the output dict since it has only one keyword and does not create any pooling. Order the output dict by the number of keywords that it pools.

Do not output any additional text or explanation - just the dict. """

An example output dict from the LLM is:

```
{
  'llm': ['llm', 'llama'],
  'ai': ['ai', 'intelligence'],
 'agent': ['intelligentagent', 'agents', 'agentchat'],
  'optimization': ['optimization', 'tuning'],
  'embedding': ['embedding', 'vector'],
  'bias': ['bias', 'fairness'],
  'technique': ['techniques', 'approaches'],
  'search': ['search', 'retrieval', 'relevance'],
  'data': ['data', 'datasets'],
  'health': ['health', 'healthcare'],
  'representation': ['representation', 'structure'],
 'communication': ['communication', 'conversations'],
  'ranking': ['ranking', 'reranking'],
  'generation': ['generation', 'generative']
}
```

From the LLM's output dict, every keyword in a given "cluster" was matched to the most commonly occurring keyword in said cluster (effectively ignoring the LLM dict's key value). Additionally, the LLM's output was constrained to 500 tokens. This was mainly done to help reduce the runtime of the pipeline, but testing also showed that the best clustering results were often achieved within the first 500 tokens. Furthermore, the LLM had a tendency to sometimes go off the rails and generate garbage after a certain point. By limiting the output tokens, this was protected against as well. If no semblance of a dict could be found in the LLM's response, then the merged keywords after the first phase were preserved and the LLM's output was completely ignored.

Ultimately, this second LLM-based phase of clustering was very successful as well. From 317 unique keywords, the total number became 272. Through this two-phase merging strategy, over 25% of the initial keywords were merged to make each keyword more unique and significant. After incorporating these merges, the overall architecture for the Visual Search is represented by Figure 3.

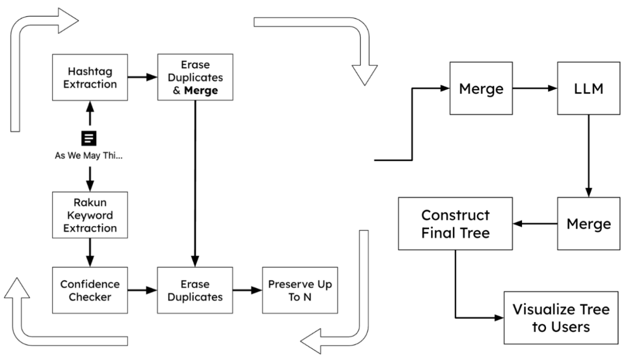

Figure 3: Visual Search Workflow

3.3 Hierarchical Tree Construction

With a refined and merged document to keywords mapping, the next step for the system was organizing the documents hierarchically by keyword. To accomplish this, a greedy set cover algorithm was used recursively to build a tree structure. The pseudocode for generating this tree is provided in Algorithms 1 and 2.

In plain terms, at each level of the tree, the algorithm:

(1) Calculated the keyword clusters for that level which covered all documents through greedy set cover
(2) Assigned each keyword cluster every one of the initially uncovered document that contained it (not only the ones that the keyword covered in the set cover problem)
(3) Determined whether to add a "miscellaneous" folder
(4) Constructed the nodes at the given level and then began recursion on their children

This design ensured that documents could appear along all semantically relevant branches and were not isolated under a single path. By having documents appear under multiple applicable paths, users would be able to more easily find and interact with them. Additionally, the "miscellaneous" folder was introduced to handle cases of excessive branching. When documents become too spread apart at a level and many of them have their own unique keyword, clustering is impossible and the level gets very cluttered. As such, at any level where more than 30 nodes were present and at least 2 documents had only one keyword remaining, all of these "unique" documents were grouped under the catch-all "misc." folder.

Algorithm 1 Greedy Set Cover

```
Require: Set of documents $\mathcal{D}$, mapping doc2keywords
 1: Initialize exclude ← $\emptyset$, uncovered ← $\mathcal{D}$
 2: Build mapping kw2docs[$k$] ← documents in $\mathcal{D}$ containing $k$
 3: picks ← [ ]
 4: while uncovered $\neq \emptyset$ do
 5:     candidates ← keywords not in exclude that cover any
        uncovered docs and do not cover all $\mathcal{D}$
 6:     if candidates is empty then
 7:         break
 8:     end if
 9:     $(k^*, d^*)$ ← keyword and covered set in candidates with
        max $|d^*|$, break ties by global doc count for $k^*$
10:     Append $(k^*,$ kw2docs[$k^*$]$)$ to picks
11:     exclude ← exclude $\cup \{k^*\}$
12:     uncovered ← uncovered $\setminus d^*$
13: end while
14: for all $d \in$ uncovered do
15:     Append $(-1, \{d\})$ to picks
16: end for
17: return picks
```

Algorithm 2 Tree Construction

```
Require: Set of documents $\mathcal{D}$, mapping doc2keywords
 1: if $|\mathcal{D}| \leq 1$ then
 2:     return leaf node with docs
 3: end if
 4: picks ← GreedySetCover($\mathcal{D}$, doc2keywords)
 5: if picks is empty then
 6:     return leaf node with docs
 7: end if
 8: Sort picks by descending size of covered documents
 9: Count number of keywords covering exactly one document
    (cnt_one) and those covering more than one (cnt_two_plus)

10: if cnt_one $> 1$ and cnt_two_plus $> 0$ and |picks| $> 30$
    then
11:     Move all singleton keyword picks to misc. folder
12: end if
13: nodes ← [ ]
14: for all $(k, D_k) \in$ picks do
15:     if $k = -1$ then
16:         Append leaf node with docs $= D_k$ to nodes
17:     else
18:         children ← BuildTreeMinCover($D_k$, doc2keywords)
19:         Append node with name $= k$, docs $= D_k$, and children
            to nodes
20:     end if
21: end for
22: return nodes
```

3.4 Frontend Visualization & Final Results

While the previous pipeline could be applied to any community of documents, this introductory test specifically focused on integration into the TextData website. As such, this frontend visualization was made specifically for TextData and may not be immediately applicable to other systems.

The frontend interface was developed using React and designed to resemble a hierarchical file browser. Upon loading, users see the top-level keywords as folders (Figure 4).

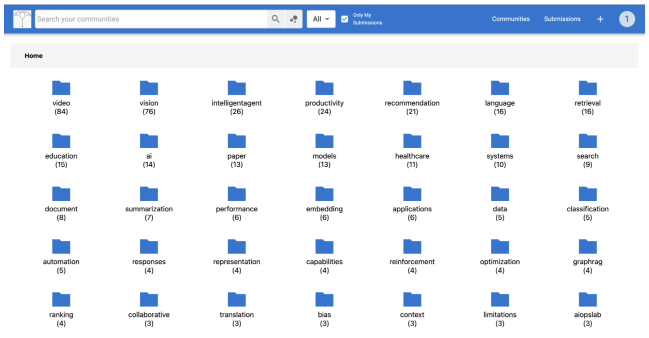

Figure 4: Top-level Visualization

Clicking on a folder transitions the interface to reveal the next level of sub-keywords and documents. A breadcrumb trail persists across all views, allowing users to track their current location within the hierarchy and seamlessly move between levels (Figure 5). Figure 5 also highlights the "misc." folder – multiple items were present at the level which contained only 1 keyword, so they were merged into one folder to reduce clutter.

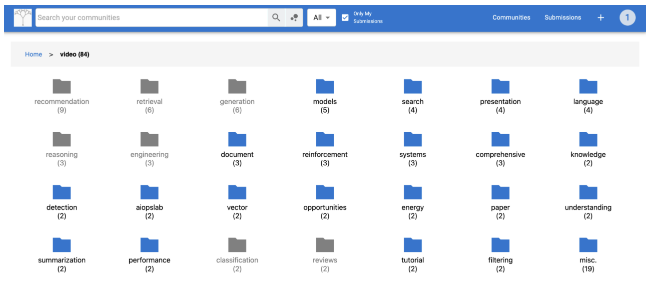

Figure 5: Deeper-level Visualization

Figure 6 showcases what a miscellaneous folder looks like, solely containing other folders with 1 associated document.

In contrast, Figure 7 shows a level with multiple folders with only 1 associated document, but no misc. folder since the total number of items at the level does not reach the 30+ threshold.

Additionally, Figure 8 highlights what coming across an individual document looks like.

To enhance usability, the interface also records interaction history. Paths that the user has previously visited are visually grayed out, providing orientation during browsing (Figure 5). Additionally, any document opened by the user is marked as visited across all its appearances in the hierarchy, allowing the user to understand when they have already come across a document in a previous path.

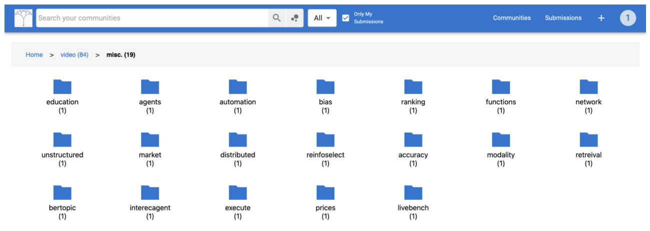

Figure 6: Misc. Visualization Example

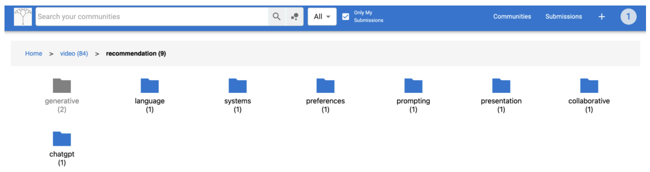

Figure 7: No Misc. Folder Example

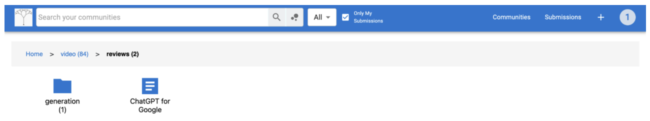

Figure 8: Document Visualization

This visual paradigm offers a substantial improvement over the previous graph-based interface by enabling intuitive and semantically meaningful exploration.

4 Intended Users & Impacts

The intended users for this feature are all members of communities that compile documents. While this feature solely exists on TextData, however, the intended users are all TextData users. Regardless of the scope of adoption, there are two primary user groups that stand to benefit from this system.

4.1 Community Members

For community members, the Visual Search feature enables thematic discovery without requiring users to know specifically what they are looking for in advance. Additionally, if a user does have a targeted search, they can easily explore all of the subtopics that make up what their initial search was. It also encourages exploration by presenting users with coherent pathways through the document space, facilitating both broad and deep dives into community topics. Finally, users can immediately find closely related documents. If there is a certain path that a user finds particularly interesting, they are able to immediately find all other documents in their communities that cover similar topics.

4.2 Community Administrators

For community administrators, this system can offer helpful insights into content distribution which would allow for more effective moderation, tagging, and organization. Moreover, since users could more easily explore and search for topics, communities could be more inclusive of a wide range of topics – in the specific context of our course, there would be no need for separate “Frontier Topic” and “Base” communities.

This dual benefit underscores the system’s utility not just as a navigational tool, but as a framework for understanding and managing large-scale discourse.

5 Use

This feature has currently been added as a branch to the TextData repository and fully functions locally. Until the system is pushed to production, however, the feature is not publicly accessible online. The source code for this feature can be viewed at: https://github.com/dhruvKS7/TextDataVisualSearch

This repository is intended to highlight individual contributions and will not function in isolation without access to TextData’s internal infrastructure.

In order to fully run this feature locally, visit https://github.com/kevinros/textdata. Clone the repository, enter the “new-visual-search” branch, and follow the local setup instructions provided.

6 Conclusion

By combining keyword extraction, merging, and efficient clustering algorithms with an intuitive frontend, the Visual Search feature brings meaningful order to unstructured document sets.

It also reflects broader trends in NLP and HCI, where automation and interface design combine to offer more intelligent content exploration tools. The modular nature of the pipeline makes it extensible to other platforms beyond TextData, and its benefits to both end-users and administrators help position it as a valuable addition to any document-based community platform.

References

[1] Andrea D’Agostino. 2021. Keyword Extraction - A Benchmark of 7 Algorithms in Python. https://towardsdatascience.com/keyword-extraction-a-benchmark-of-7-algorithms-in-python-8a905326d93f/.
[2] Blaz Skrlj. [n. d.]. rakun2: Rapid Automatic Keyword Extraction in Python. https://github.com/SkBlaz/rakun2.
[3] Kevin Ros. [n. d.]. TextData. https://textdata.org/about.
[4] Maarten Grootendorst. 2020. KeyBERT. https://maartengr.github.io/KeyBERT/.
[5] Steven Loria. 2013. TextBlob: Simplified Text Processing in Python. https://textblob.readthedocs.io/en/dev/.
[6] Vishwas B Sharma. [n. d.]. rake-nltk: Rapid Automatic Keyword Extraction for NLTK. https://github.com/csurfer/rake-nltk.
[7] Zaira Hassan Amur, Yew Kwang Hooi, Gul Muhammad Soomro, Hina Bhanbhro, Said Karyem, and Najamudin Sohu. 2023. Unlocking the Potential of Keyword Extraction: The Need for Access to High-Quality Datasets. *Applied Sciences* 13, 12 (2023), 7228. doi:10.3390/app13127228