UK News Outlets Search System

Group ID: 18

s2495270, s2442474, s2556955, s2482197, s2601534, and s2524320

Link to the live search engine: http://34.89.46.189/

Note : It might get changed due to the system reset, as we do not reserve static IP on the server.

Link to the Github Repository

Abstract

This study presents a dedicated search engine tailored for the UK news landscape, designed to address the challenges of excessive information and enhance news credibility. Utilizing methods like TF-IDF scoring, Boolean searches, and innovative features such as automated aggregation, concise summarization, and sentiment assessment, the platform ensures the delivery of pertinent and authentic information. This system onboarded 706k news documents from 4 trusted outlets and an additional $\pm 1.3k$ daily news of live indexing with about ± 600 tokens for each document. Diverging from traditional models, this system emphasizes the significance of content quality over mere viewership metrics, thereby facilitating more effective navigation through the extensive online news environment. Through the inclusion of features such as query enhancement and suggestion, this system provides substantial support to diverse groups such as media professionals, academic researchers, and the general populace, helping them to accurately identify and comprehend important topics.

1 Introduction

In the digital age, as the number of published news articles keeps increasing, it becomes a real challenge for people to digest the content effectively. The existing search engine model, primarily based on visited document metrics such as those described by PageRank[1], may pose risks. Specifically, it could inadvertently prioritize and promote high-traffic documents, including clickbait news articles with questionable credibility, as a result of efforts to optimize visibility based on traffic alone. This approach might overshadow the importance of content authenticity and reliability [5]. The vast volume of daily publications presents an opportunity to create solutions to aid individuals in obtaining relevant insights efficiently. To solve this challenge, we developed a search engine to perform comprehensive search news tailored for UK news content focused on relevancy with several additional features like sentiment analysis and summarization.

The designed system seeks to enhance user comprehension of pertinent news articles published in the UK. It achieves this through three primary functionalities: automated aggregation of news content, summarization of articles, and analysis of news sentiment. Additionally, to enhance user experience, the website offers several query-assistance tools, including query suggestions, a spell-checking feature, and query expansion capabilities. Ultimately, the system facilitates users in efficiently locating, extracting, and comprehending vital information, thereby enabling a more profound grasp of current events tailored to their interests.

The foundation of the proposed solution lies in the utilization of the TF-IDF score method [10]. This method quantifies the importance of terms within documents relative to the entire corpus, allowing the user to get sorted documents or ranking based on relevancy to the query. In addition to the TF-IDF method, the system incorporates a Boolean search feature, enabling users to locate documents containing specific terms exactly as per their queries within the corpus.

The relevance and utility of the UK News Outlet Search Engine System transcend various domains, providing an invaluable tool for journalists, researchers, policymakers, and the general populace. As the users find themselves with a large amount of information from UK news outlets daily, this solution enables them to effectively meet their information needs and insight from the sheer volume of available news content.

In summary, the team developed a system to help users effectively use the content of UK news by offering a sophisticated search engine equipped with automated aggregation, summarization, and sentiment analysis capabilities. Through the integration of these advanced technologies, we aim to empower individuals with the tools necessary to navigate the vast expanse of news media effectively and gain deeper insights into current affairs.

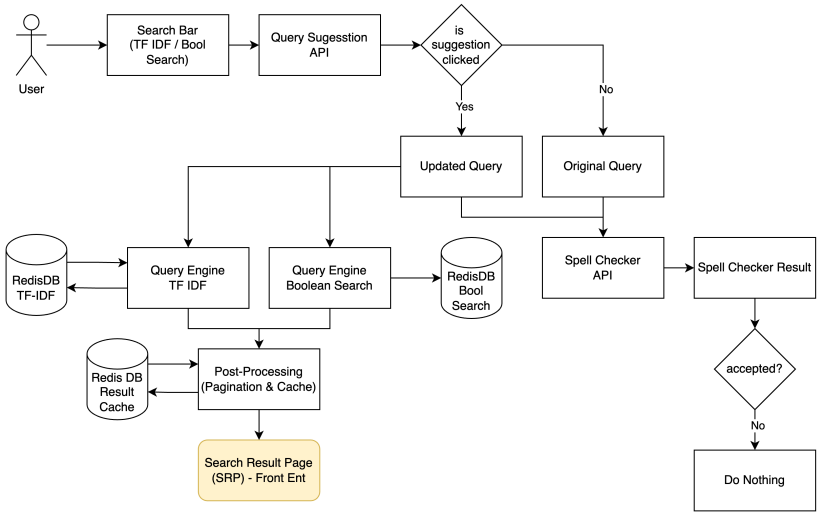

Figure 1: System Overview.

2 Proposed System

2.1 System Overview

Figure 1 delineates the comprehensive architecture of the system deployed. Initially, users are greeted with a search interface comprising an input field and an engine selection option. Currently, the system supports two search engines: TF-IDF, which is set as the default, and Boolean Search. Upon query input, the system's Query Suggestion API, leveraging a tree search algorithm equipped with the monograms and bigrams of the corpus, dynamically provides query recommendations to minimize user input effort. Furthermore, as a secondary safeguard, any typographical errors within a submitted query activate a spell checker feature, presenting correct spelling suggestions derived from the same Word2vec model. The spell checker API will check every query, either the original query or the updated one. Users may either adopt the corrected query as proposed by the Spell Checker API to retrieve relevant results, or they may disregard the suggestion, in which case the system proceeds to generate results based on the original query.

The system employs Redis to maintain positional and frequency index, aligning with objectives for system scalability and real-time indexing. Concurrently, to further enhance scalability, a caching mechanism is implemented within Redis. This approach ensures that results for newly submitted queries are temporarily stored along with the results of 4 neighbouring pages for 5 minutes. Consequently, this reduces the computational load for identical queries received multiple times, optimizing the system's efficiency and response time for repeated inquiries.

Figure 2 illustrates the workflow for live indexing. Initially, the Redis-based Virtual Machine (VM) Index instigates the activation of the VM Daily Index, which is scheduled by a Cron's job to operate daily. Upon activation, a pipeline commences that is dedicated to gathering public data from various websites, adhering to the limitations set by their respective robots.txt files. Subsequently, the pipeline undertakes the construction of the index, incorporating metadata such as titles and dates. It also performs sentiment analysis predictions and content summarization. Upon completion of these tasks, the collated data are uploaded to the Redis database within the VM Index, where the index resides. Finally, the process concludes with the automatic shutdown of the VM Daily Index, ensuring resource efficiency through self-termination. See figure 3 for the full architecture.

2.2 Data

The data employed in the system are sourced through daily web crawling activities targeting specific news outlets: BBC, GBNews, The Independent, and The Telegraph. This is done in strict adherence to the robots.txt policies stipulated on their respective websites, ensuring the system's web scraping

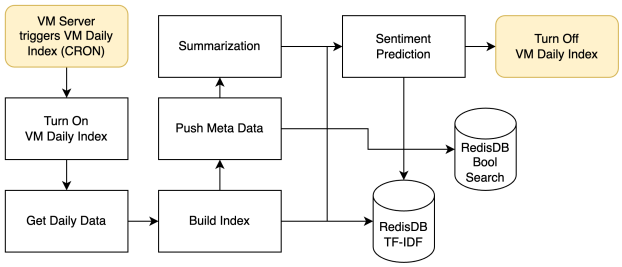

Figure 2: Live Daily Indexing.

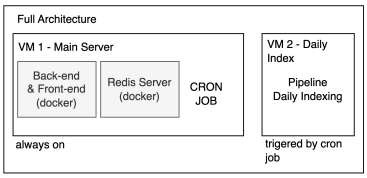

Figure 3: Infrastructure

activities comply with their established guidelines. Specifically, while BBC offers articles in multiple languages, the system is configured to exclusively retain English-language documents, filtering out all non-English content. Consequently, the system has accumulated approximately **706,844** past news articles to date. Additionally, owing to the live indexing feature, there is a consistent daily addition of up to **1,300** new articles, maintaining the database's growth and relevance. As a note, on average, each document has around **667.93** tokens to be processed.

The system is set up to collect only new URLs, avoiding duplicates in the database, which allows for more frequent updates and keeps the system updated with the latest news. Given that the daily volume of articles is manageable, we've scheduled the data collection to run daily, striking a balance between providing fresh content and maintaining resource efficiency.

2.3 Front End

The web application leverages React.js for the front end, utilizing Bootstrap and CSS to create a dynamic and visually appealing layout. This is integrated with a FastAPI backend as a static single page application. React.js offers a component-based architecture for efficient UI development, improving user interaction and performance. FastAPI was chosen for its efficiency and support for machine learning frameworks. It provides a smooth connection between the user interface and backend systems, which boosts platform responsiveness and functionality.

The system's GUI, built with React.js, is the interface for user interactions with various features like displaying search results, query expansion, and performing boolean and TF-IDF searches. Axios in the `api.js` function manages backend communication by calling endpoints in `search.py`, enhancing the application's responsiveness and data retrieval efficiency.

At the heart of the application, `App.js` functions as the homepage, utilizing React's `useState` to manage component state, thus facilitating the dynamic storage and updating of data. The `useEffect` hook plays a crucial role in executing side effects, such as `debouncing` [1] user input in the search query. This technique significantly enhances the user's search experience by efficiently managing keystroke registration and data retrieval, ensuring that users receive immediate and relevant feedback as they type. Navigation and user experience are thoughtfully designed to provide a consistent layout across different pages, with user-selected search options guiding navigation to corresponding result pages. For instance, a boolean search query like "Trump and Biden" triggers query suggestions, leading users to boolean component in the application for a detailed view of relevant results, including suggested expansions and the opportunity for further query refinement. The application has a dynamic sentiment analysis display, where a visually coded bar reflects the sentiment intensity of each article, offering users an immediate understanding of the prevailing emotional tone. Pagination at the bottom of the search results enables easy navigation through pages, improving accessibility to information.

Error handling is carefully integrated to provide users with clear messages like 'No results found' in case of no query matches, ensuring transparency and engagement. The application also features TF-IDF search functionality from the homepage, similar in layout to boolean search but with the addition of displaying TF-IDF scores for ranking results by relevance. This enhances search experience and offers insights into article relevance, making the application a valuable resource for exploring large news article datasets.

[1] https://lodash.com/docs/4.17.15debounce

2.3.1 Features and Enhancements

In the attempt to develop a state-of-the-art news article search engine application, a series of strategic decisions and advanced functions were implemented, prioritizing speed, efficiency, and an intuitive user experience. Recognizing the unique needs of users navigating through vast datasets of news articles, the application incorporates filtering options, dynamic sentiment analysis, and efficient pagination, all designed to deliver a seamless and interactive platform.

The integration of filters by `year, sentiment, and source` is designed to enrich the user experience with customized search capabilities. The `year` filter is particularly beneficial for users interested in news articles from specific periods, enabling them to swiftly access temporally relevant information. The `sentiment` filter further streamlines the search experience, allowing users to segregate articles based on their emotional tone whether positive, neutral, or negative. This addition not only expedites the search process but also introduces an analytical dimension, facilitating users in discerning the emotional context surrounding the news content. Lastly, the `source` filter allows users to refine their searches to specific, preferred news outlets, enhancing the trustworthiness and pertinence of the search results, thereby ensuring that the information presented aligns with user preferences and credibility standards.

A standout feature of the application is its dynamic sentiment display, which creatively visualizes the emotional composition of news articles. This feature goes beyond traditional text-based presentations, employing color-coded badges to represent various sentiments. The visual representation of sentiment analysis within the search results is dynamically adjusted to convey the emotional weight of each article, with the color intensity serving as an intuitive indicator of sentiment strength. Specifically, a higher sentiment score within the positive spectrum results in a correspondingly deeper shade of the color, symbolizing a stronger positive impact. Conversely, a lower score in the same category is represented by a lighter color, denoting a milder positive sentiment. This nuanced approach allows users to quickly grasp the emotional significance of search results, enhancing the overall user experience by integrating both analytical depth and visual clarity.

Pagination plays a crucial role in the application's design, addressing the challenge of navigating through extensive search results. By dividing results into discrete pages, the application not only improves loading times but also enhances user navigation, preventing information overload. This thoughtful implementation of pagination underscores the application's dedication to maintaining a balance between comprehensive data access and user-centric design. In the search field, there's an added functionality of query spell checker and correction which can automatically correct misspelled queries or suggest corrections, ensuring that searches yield relevant results even when the user makes typographical errors. This feature is particularly useful for fast and efficient retrieval of information, as users may not always know the correct spelling of the terms they are searching for and this attention to detail can contribute to higher user satisfaction.

The application is a well-rounded solution that combines backend efficiency with frontend innovation for a fast and interactive user experience. Features like advanced filters, sentiment analysis visualization, and smart pagination are all designed to meet user needs. This holistic approach ensures not only the applications performance and usability but also its user-friendliness for navigating and making sense of extensive news content.

2.4 Back End

2.4.1 Database System

The system used 4 databases on Redis to store the inverted index, the inverted index for tf-idf, document metadata, and query cache. Redis is preferable among other NoSQL databases because it is highly optimized for retrieving the key-value data [4]; it loads the data into memory and uses a hashing mechanism to store the data. The computation to retrieve a single data is O(1). Thus, the computation complexity for a query becomes O(N+D) where N is the number of tokens and D is the number of retrieved documents. This database is deployed in the same VM along with the frontend & backend using docker images on the same network.

The system utilizes four distinct databases within Redis to manage the inverted indices, document metadata, and query cache. Redis is selected over other NoSQL databases due to its superior optimization for in-memory key-value data retrieval as highlighted in literature conducted by A. T

Kabakus et al.[4] It operates by loading data into memory and employing a hashing mechanism for data storage, which facilitates O(1) computational efficiency for accessing individual data points. Consequently, the overall computational complexity for processing a query is represented as O(N+D), where N corresponds to the number of tokens in the query and D denotes the number of documents retrieved in a page. To ensure retrieval performance, the Redis database is hosted on a the identical VM with a backend via a docker image within the same network.

2.4.2 Index

The Index is structured to contain five key components: document size, document ID list, URLs, and the **positional inverted index**, and **frequency inverted index for tf-idf**. The inverted index is particularly structured around each term, where the values consist of the document ID and the term's position within the document, utilizing delta encoding for efficiency. In comparison, the inverted index for tf-idf replaces the list of positions with the frequency of the term, as computing the frequency of the term on the fly has proven to be slower in the system. It reduces the computation of tf-idf method from O(NF+D) to O(N+D) where T is the number of query tokens, F is the average number of documents of term t, and D is the retrieved documents (it is similar to the computation of boolean search). The document ID serves as a unique identifier for each document, encompassing all sourced materials.

In the context of live indexing, the process involves retrieving the term's document data from the index, appending this with data from newly added documents, and subsequently updating the term entry in the Redis database. This procedure is executed within a dedicated VM, initiated by a Google Cloud (gcloud) command from the main VM Index (Redis). This operation is scheduled via a Cron Job to occur daily at 5 AM (denoted by the Cron schedule "0 5 * * *"). Following the completion of the indexing update, the VM designated for Live Indexing is automatically shut down to optimize resource usage and minimize operational costs.

The last version of the index contains 469,030 terms. Along with the metadata, all together consumes 5GB. It might be get updated if the system finds a new term in a new document (due to daily live indexing). Initially, we had also employed delta encoding for document IDs, and for each term, its document IDs and Positions, which had been saving 25-27% of the size of the full index. However, due to substantial slowdown in decoding the encoded index for search retrieval, in the final version, we only encode the Positions of terms, which saves around 3% of the weight.

2.4.3 Query Engine

The system incorporates two distinct types of query engines: TF-IDF and Boolean. The TF-IDF engine operates in accordance with the methodology outlined in "Introduction to Information Retrieval" by Manning et al.[7], mirroring the implementation utilized in Coursework 1. This approach employs the standard TF-IDF formula to evaluate the importance of a term within a document relative to a collection of documents.

Conversely, the **Boolean engine** facilitates the **Boolean search** and **Proximity search** with syntax: $\#dist(term1, term2)$, consistent with the structure established in Coursework 1. To incorporate complex queries involving multiple logical connectors (OR and AND) and brackets, we convert the infix expression to postfix expression with precedence using stack operators. We process each operand (i.e. phrase, word or proximity) separately as each part of the computation for operands is independent of each other, and execute the logical operators last to optimize the speed of code execution. Within this engine, users can conduct fixed term searches using logical operators such as OR and AND. Additionally, the engine supports Phrase Search, which allows for the retrieval of documents containing an exact sequence of words, and Proximity Search, which finds documents where the specified terms are within a certain distance from one another. This multifaceted approach enables users to perform a range of precise searches tailored to their specific information needs.

To optimize speed, two separate indices are used for two search methods. Database operations are consolidated into a single request pipeline to reduce latency when accessing the database.

2.5 Query Expansion

Our initial query expansion using Pseudo Relevance Feedback with TF-IDF ($\mathrm{tf}. \log \frac{N}{\mathrm{df}}$) revealed limitations due to its reliance on terms from retrieved documents, failing to capture a wider range of semantic relationships - it becomes evident that the terms added through the initial method are confined to those present in the retrieved documents, resulting in the omission of synonyms or related concepts not directly mentioned. Transitioning to distributional semantics, we experimented with pre-trained BERT embeddings [13] for deeper query contextualization, supplemented by the Facebook AI Similarity Search library [3] for term addition through cosine similarity. However, the high computational load of deep model embeddings, such as DistilBERT [12] and DistilRoBERTa [6], proved challenging for real-time processing, resulting in times incompatible with the application requirements, highlighting the need for a more efficient approach without sacrificing the capacity for direct model fine-tuning on our dataset. Instead, we adopted a feasible alternative as delineated by Diaz et al.[2] and Rattinger et al.[11], developing our own local Word2Vec embeddings[8] tailored to the complete dataset, from which stopwords have been excised yet without applying stemming. Here, a small neural network is employed to capture word associations within a d-dimensional space, and words with similar meaning are positioned closer together. For a given word w, Word2Vec maximizes the objective $\frac{1}{T}\sum_{t=1}^{T}\sum -c \leq j \leq c, j \neq 0 \log p(w_{t+j}|w_t))$, where T is the corpus size, c is the context window, and $p(w_{t+j}|w_t)$ represents the probability of observing a context word w_{t+j} given the current word w_t. Upon entry of a query, the system augments it by adding n_t terms for each non-stopword term in the query, post-verification that it is not already present in the query post-stemming. Albeit we had created our own implementation of Word2vec with `PyTorch`, we find that even with a GPU we cannot match the speed of the C++-optimized `Gensim` implementation needed for re-training of the vectors quickly, which is desirable for live indexing. We train word vectors with $d = \{100, 200\}$ and $c = \{5, 10\}$ finding that the higher-dimensional vectors with a larger window bring about the best results, while the query expansion times, even for a query with 5 terms, is still in the tens of milliseconds. The vocabulary size is around 400 thousand. This implementation is contained in `backend/utils/query_expander.py`.

2.6 Sentiment Analysis

Initial attempts with sentiment analysis were made with deep Bi-LSTM with Batch Normalisation, Dropout and pre-trained Glove embeddings [9] (created for Coursework 2) fine-tuned on Kaggle Financial News Dataset[2] for negative, neutral and positive categories, however, it was quickly discovered that the F1-dev macro saturates at about 0.53. We find a reasonable solution in using a DistillRoBERTa[3] from `HuggingFace`, fine-tuned on the financial-phrasebank[4] dataset to achieve a 97.8 % validation accuracy while being within our computational budget - on our GPU, inference of a single news article takes 0.02 seconds, while on a VM CPU, it takes 0.5 seconds, meaning it is compatible with live-data collection, and gives back reasonable results for our data. This implementation is contained in `backend/utils/sentiment_analyzer.py`.

2.7 Document Summarization

For this section, our objective was to devise a rapid method for enhancing user experience by appending a descriptive phrase to the title when an article is retrieved, a task made challenging due to the lack of a separate `div` field for such information in most news outlets' formats. Given the prohibitive computational demands of employing a large language model for summarization, as explored in Subsection 2.5, we opted for a simpler, more feasible approach suited to real-time indexing. This involved utilizing a TF-IDF vectorizer to identify the sentence within the article content that most closely aligns with the title. We removed stopwords from both the title and the content to reduce noise, yet refrained from stemming to preserve semantic coherence. The TF is normalized by the total word count in the text, while the IDF is calculated across the entire corpus to diminish the impact of commonly recurring terms. The most pertinent sentence is then determined based on cosine similarity, comparing the TF-IDF vector of the title against those of individual sentences

[2] kaggle.com/datasets/hoshi7/news-sentiment-dataset

[3] huggingface.co/mrm8488/distilroberta-finetuned-financial-news-sentiment-analysis

[4] huggingface.co/datasets/financial_phrasebank

within the content, that is $\cos(\theta) = \frac{\sum_{i=1}^{n} \text{TF-IDF}_{\text{title},i} \times \text{TF-IDF}_{\text{content},i}}{\sqrt{\sum_{i=1}^{n} (\text{TF-IDF}_{\text{title},i})^2} \times \sqrt{\sum i=1^{n} (\text{TF-IDF}_{\text{content},i})^2}}$. This functionality is encapsulated in `utils/backend/module_summarizer.py`.

2.8 Query Spell Checker

Initial attempts for query spell checking had been made with the Norvig spell checking algorithm[5], generating candidate spellings within an edit distance of one or two from the potentially misspelt words, then selecting the most probable correction based on $P(c|w) = P(w|c) \cdot P(c)/P(w)$, where $P(c|w)$ is the probability of correction c given word w, $P(w|c)$ is the likelihood of typing w when c was intended, $P(c)$ is the frequency of c in language, and $P(w)$ is the probability of w appearing in the input. However, even having pre-calculated and stored the frequencies of unigrams in a subcorpus (330 thousand documents), this algorithm took well over a minute to correct a simple query, e.g. "donuld tromp". The only solution we find to be compatible with the deployment of the system with a corpus this large is the C++ optimized version of famously fast `SymSpell`[6] algorithm paired with pre-calculating and storing frequencies of unigrams (choosing a threshold of appearing 100 times). Specifically, SymSpell retrieves candidate corrections C for a misspelt word W by calculating $C = \bigcap_{d \in D(W)} M(d)$, where $D(W)$ is the set of all delete permutations of W within a specified *edit distance*, and $M(d)$ maps each delete permutation d to its corresponding original words in the dictionary. The final correction is then selected based on the minimum Damerau-Lavenshetein distance W and the highest frequency f of candidates in the corpus, formalized as $\min_{c \in C}(\text{distance}(W, c) + \log(1/f(c)))$. This implementation takes tens of milliseconds to correct the query, and is contained in `utils/backend/spell_checker.py`.

2.9 Query Suggestion

Query Suggestion is designed to enhance user experience by providing context-aware suggestions as users type their queries. Initially, our system harnessed the capabilities of a pre-trained RoBERTa to generate sophisticated query suggestions. Despite this, similarly as in previous sections, we had no way to fine-tune it, hence resorted to building our own corpus search. We found the best solution in leveraging a directed acyclic word graph (DAWG) structure for efficient lookup and Least Frequently Used (LFU) cache, taking a lot of inspiration from the Fast Autocomplete library[7] library. We use a similar approach on our created dataset of monogram and bigram frequencies for our corpus, limiting the frequency to 100. This dictionary is loaded into memory, and the words are reduced to their *valid character sets* - stripping of irrelevant characters and limiting their length to ensure computational efficiency (similar process is applied to the query as it is being typed). DAWG organizes words in a compact form that allows for the rapid traversal and retrieval of potential suggestions based on the user's input. As users type, the system dynamically adjusts its suggestions, prioritizing relevance and frequency of use. The suggestions are further refined through the application of an LFU cache, which stores the most recently and frequently accessed queries to expedite future retrievals. We find that suggesting 5 dynamically generated outputs based on their frequency and proximity to the last two tokens of the user's input is the best at the tradeoff of useful suggestions and waiting times, taking tens of milliseconds. This implementation is in `backend/utils/query_suggestion.py`. In the frontend, this is further improved by employing debouncing - the system waits for a brief pause in typing before sending a request, effectively minimizing the number of API calls to the backend.

3 Quantitative Evaluation

In this segment, our objective was to evaluate the effectiveness of the ranking system by the tf-idf. The experiment design compares this model against boolean search results as the baseline. Initially, boolean searching generated 5 candidates on 5 dedicated queries. Later, the experiment compared the random order from those candidates against the ranked candidates by using the tf-idf score. As the ranking label, 5 members rated the usefulness on a 5-point Likert scale, the score 5 indicates high utility and 1 signifies minimal relevance. Table 1 shows the result of the NDCG scores from our samples of a query with the comparison with a randomly ordered document. From Table 1 it could be

[5] https://norvig.com/spell-correct.html
[6] https://github.com/wolfgarbe/SymSpell
[7] github.com/seperman/fast-autocomplete

Table 1: Comparison of NDCG scores

Query	Random	TFIDF	Uplift (%)
Healthy breakfast	0.7678	0.9098	
Small business	0.8203	0.9644	
Manchester United	0.8149	0.8277	
Latest technology	0.785	0.8387	
Travel destination	0.7409	0.821	
Average	0.78444	**0.87232**	**11.20%**

seen that our system can effectively rank documents according to their relevance compared to baseline random document order. The uplift our system provided compared to the baseline performance is 11.20%.

4 Scalability

The main server is hosted in 4 CPUs & 24 GB RAM in the London Region ($140.41 per month). The backend, frontend, and the redis server consume about 8GB of memory. As the daily pipeline index adds $\pm 1.3k$ news daily, it will only add est. 10.97MB of RAM per day. Hence, by keeping the system as it is, the system can handle the new data up until the next **6 years**.

5 Limitations and Challenges

The UK News Outlets Search System faces certain limitations and challenges primarily due to restrictions imposed by the data sources. Currently, we can only aggregate news articles from BBC, The Independent, The Telegraph, and GBNews. This limitation arises because other news outlets do not permit public access to their past news archives, as indicated by their policies, suggesting a restriction against the public crawling of their historical articles. Additionally, the system is presently unable to display and process image data. This aspect is earmarked for enhancement in subsequent application updates.

When updating the positional inverted index, the size of the positional index for a term gradually increases with the corpus. This limits the number of term positional indexes that can be updated simultaneously. Eventually, the term index will exceed the memory capacity that a computer can handle. Additionally, the bottlenecks for TFIDF searching mainly arise from calculation and sorting. To optimize this for a billion-scale, the computation and storage should be distributed. For instance, storing the child indices in a distributed database system and performing distributed computation i.e. map-reduce for calculating the scores. Furthermore, due to limited time and resources, we are unable to implement advanced search techniques such as word embeddings. However, the use of vector searching to explore the embeddings of the full or summarized news content could significantly enhance the retrieved results as it captures the semantics of the query and documents.

6 Conclusion and Future Work

The development of the UK News Outlets Search System is an attempt to help users navigating the vast landscape of UK news content. Through the implementation of algorithm such as TF-IDF and Boolean search, coupled with features like automated aggregation, summarization, sentiment analysis and advanced natural language processing models like Word2Vec and RoBERTa, the system offers a tailored search experience that prioritizes relevance, sentiment analysis, and efficient content summarization. Our system aims to help users in finding relevant documents tailored to their interests. The qualitative analysis conducted through NDCG scores showcases the system's ability to effectively rank documents based on their relevance compared to a random baseline, indicating 11.20% uplift in performance. This underscores the system's capability in enhancing user engagement by ensuring that the retrieved information is both relevant and reliable. However, the system does face limitations, notably in data source accessibility and image data handling, which could be improved in future updates.

Individual Contributions

1. **S2601534** worked on optimising indexing and Δ-encoding, and researched, built implementations/APIs and wrote the respective sections of the report for query expansion, suggestion and spell checking, document sentiment analysis and summarization functionalities.
2. **S2495270** worked in both phases of the application development. In the backend, s2495270 have worked on document summarising feature. This functionality, implemented in `backend/ai/doc_sum/doc_summary.py`, leverages cosine similarity for assessing the semantic similarity of sentences within articles. A graph representation of sentence relationships is then constructed, upon which the PageRank algorithm is implemented via the `NetworkX library`, applied to rank sentences by their significance and interconnectivity. However, upon testing, it was discerned that the computational complexity $O(N^2)$ and memory intensity of this method, especially for lengthy documents, rendered it impractical for our dataset, leading to its non-adoption. s2495270 have also implemented Query Expansion using the pre-trained `BERT` model described in the section 2.6. This feature preprocesses user queries, generates embeddings, and employs cosine similarity to identify and suggest articles most relevant to the user's search intent, as encapsulated in `backend/ai/QE_Bert.py`. Furthermore, the development of the Query Suggestion feature, was undertaken by s2495270, employing the `RoBERTa` model for its superior performance in efficiently resolving ambiguous queries. s2495270 handled the frontend implementation for this feature using `debouncedFetchSuggestions` explained in section 2.4. On the frontend, s2495270's contributions involves the creation of React components for `Boolean` and Tfidf features and integration of backend with the frontend. This includs integrating advanced features such as filtering, sentiment analysis, and pagination, which are elaborated upon in section 2.4 and subsection 2.4.1. The frontend implementations is contained in `frontend/src`. s2495270 worked on drafting the project report as well, particularly the sections detailing frontend development, features, enhancements and query suggestions.
3. **S2442474** have been working on several tasks such as data collection, live indexing, database infrastructure, and assessing code for deployment. On data collection, at the initial phase, S2442474 examined 10 sources of outlets in the UK. After policy checking and examination, only four of them are eligible for data scrapping. Later, S2442474 built a general module to scrape four outlets using Beautifulsoup4 in Python. To make the process faster, S2442474 implemented async requests on two VMs. In a day we retrieved 300k documents. Later, S2442474 deployed the Redis database using docker on a VM with a startup script to run the docker automatically. Later, S2442474 built a script to push all of the metadata into Redis (asynchronously). Then, S2442474 built a pipeline which covered scripts to scrape the daily data, build an index, generate summary and sentiment, push data into Redis, and set up the cron job. S2442474 also help with the code integration for deployments such as spell checker and query expansion implementation. S2442474 also work on code optimization like optimizing the query retrieval and pagination on the backend.
4. **s2482197** In the development of our React application, S482197 worked on designing an intuitive front-end and incorporated comprehensive error handling to ensure smooth user interactions. S482197 responsibilities extended to contributing to the Quantitative Evaluation section of our project report along with other aspects. On the API side, S482197 was responsible for devising functions in 'api.js' and linking them to the front-end to facilitate boolean and TF-IDF searches, prioritizing efficient network requests and the uniform presentation of search results. This work aimed to provide a reliable and easy-to-navigate search experience. Additionally, S482197 focused on refining various front-end elements to enhance the overall usability and ensure the functionality of different interface components.
5. **S2524320** worked on data processing, backend development, frontend development, database utilities, data structures, deployment, data indexing, query engine, code optimisation and system architecture.
6. **S2556955** In the project, one of S2556955 contributions primarily focused on writing the project report especially to create introduction, overall system overview, main document layout, limitation, conclusion and abstract. S2556955 also manage the report to be compliant to the coursework requirement. S2556955 also designed and implement qualitative measurement framework for the system. S2556955 meticulously designed a comprehensive

set of qualitative metrics tailored to assess the system's performance and user satisfaction. Our team system's performance was assessed through the lens of Normalized Discounted Cumulative Gain (NDCG) scores, comparing our query results with baseline random document order. The benchmark for relevance in our experiments was established based on a consensus score from participants, derived from averaging their individual assessments. Additionally, S2556955 also developed front-end section of the topic feature. s2556955 also involved in attempt to implement pagination on our front-end system. Furthermore, S2556955 also involved in data collection of our team. S2556955 collected the news data collection from past researches to enrich the project's informational resources and also as a benchmark for data quality in our system. Through these contributions, S2556955 aimed to elevate the project's overall quality and impact.

The harmonious synergy amongst the members of our joint initiative was genuinely astonishing, leading to a smooth and successful endeavour. Each member of the team gave their full and enthusiastic contribution towards the project's goals. We all planned and spoke on the approach we will take to the project. All six equally contributed to different phases of the system development. Each of the members contributed equally to the completion of the report, which allowed us to share knowledge about the unique components that each person had implemented. We used tools like Jira, discord to log our efforts and set up work and issues within a fixed time frame. In addition to improving the calibre of our work, this egalitarian approach fostered a sense of collective ownership and pride in the finished product. The project's success is evidence of the power of our teamwork and emphasises the importance of each member's contribution to attaining our goals.

References

[1] Monica Bianchini, Marco Gori, and Franco Scarselli. Inside pagerank. *ACM Transactions on Internet Technology (TOIT)*, 5(1):92–128, 2005.

[2] Fernando Diaz, Bhaskar Mitra, and Nick Craswell. Query expansion with locally-trained word embeddings, 2016.

[3] Jeff Johnson, Matthijs Douze, and Hervé Jégou. Billion-scale similarity search with gpus, 2017.

[4] Abdullah Talha Kabakus and Resul Kara. A performance evaluation of in-memory databases. *Journal of King Saud University-Computer and Information Sciences*, 29(4):520–525, 2017.

[5] Vivek Kaushal and Kavita Vemuri. Clickbaittrust and credibility of digital news. *IEEE Transactions on Technology and Society*, 2(3):146–154, 2021.

[6] 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.

[7] Christopher D Manning. *Introduction to information retrieval*. Syngress Publishing,, 2008.

[8] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient estimation of word representations in vector space, 2013.

[9] Jeffrey Pennington, Richard Socher, and Christopher Manning. GloVe: Global vectors for word representation. In Alessandro Moschitti, Bo Pang, and Walter Daelemans, editors, *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 1532–1543, Doha, Qatar, October 2014. Association for Computational Linguistics.

[10] Shahzad Qaiser and Ramsha Ali. Text mining: use of tf-idf to examine the relevance of words to documents. *International Journal of Computer Applications*, 181(1):25–29, 2018.

[11] André Rattinger, Jean-Marie Le Goff, and Christian Gütl. Local word embeddings for query expansion based on co-authorship and citations. In *BIR@ECIR*, 2018.

[12] Victor Sanh, Lysandre Debut, Julien Chaumond, and Thomas Wolf. Distilbert, a distilled version of bert: smaller, faster, cheaper and lighter, 2020.

[13] Deepak Vishwakarma and Suresh Kumar. A contextual query expansion model using bert based deep neural embeddings. In *2023 6th International Conference on Information Systems and Computer Networks (ISCON)*, pages 1–6, 2023.

Appendix A Demo

Link to the demo application: https://www.youtube.com/watch?v=7AMeuwQL4LU

Appendix B Snapshot

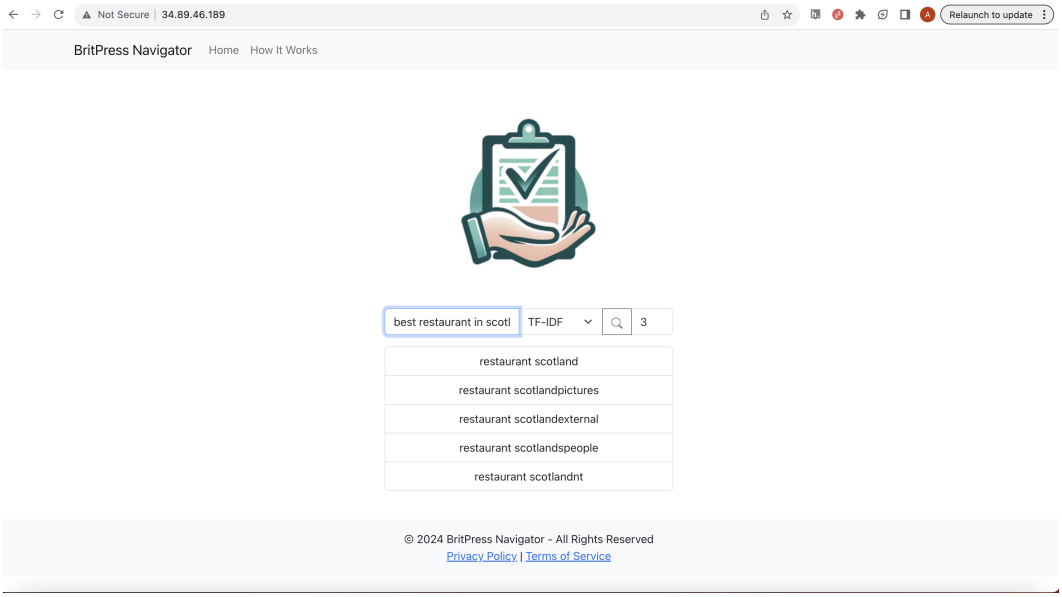

Figure 4: Homepage

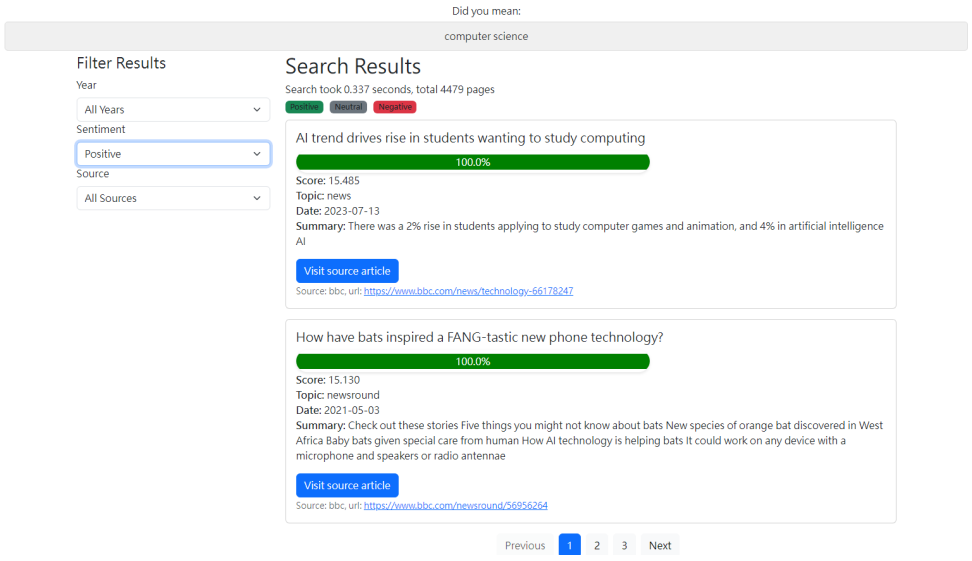

Figure 5: Search Result Page