IKEA Hacks: Information Retrieval Project Report

You can find the project repository on Github:

https://github.com/Dunkeshon/IRScrappers

Polad Bakhishzade, Andrii Ioffe

Abstract

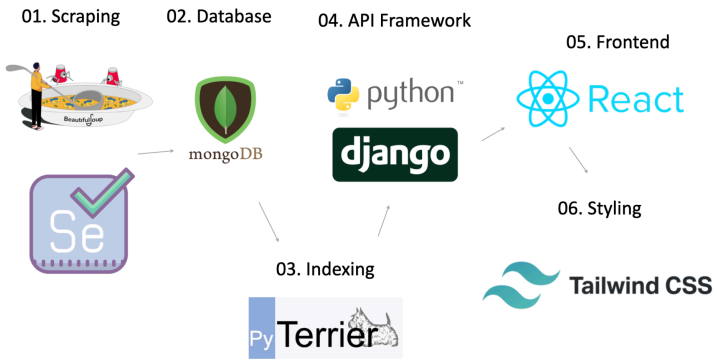
This report outlines the development of an information retrieval system focused on IKEA hacks. The system integrates web scraping, database management, preprocessing, indexing, a robust API framework, and a dynamic frontend. By leveraging modern tools and methodologies, this project offers insights into constructing an efficient and user-friendly search engine.

The IKEA Hacks project addresses a growing need for efficient retrieval of user-generated content and reviews concerning creative modifications of IKEA furniture. These modifications, often referred to as "IKEA hacks," inspire users to explore new ways of utilizing furniture components. By aggregating and processing data from multiple online sources, our system simplifies the search and discovery process for users, providing relevant results based on their queries.

The core objective was to build a robust pipeline capable of handling unstructured data from diverse sources and converting it into a structured format suitable for indexing and retrieval. This report details the project's architecture, methodologies, implementation challenges, and user evaluation results.

Simples Features: **Results Snippets** Complex Features: **User Feedback, Results Clustering**

System Architecture and Tech Stack



Pipeline Overview

The system pipeline was designed with distinct stages, ensuring modularity and efficiency. Each stage played a crucial role in processing data, from raw collection to delivering polished results to the user.

Scraping (Beautiful Soup & Selenium)

The pipeline began with the data scraping stage, where information was collected from three primary sources: IKEA Hackers, The Spruce, and Trustpilot. Beautiful Soup was used for parsing static web pages, while Selenium handled dynamic content scraping. Extracted fields included titles, detailed text, and source links. This data was structured and stored in MongoDB for further processing.

Preprocessing (Pyterrier)

In the preprocessing phase, raw data was refined to ensure uniformity and suitability for indexing. Tokenization was applied to split text into individual words. Stop word removal was conducted to eliminate common, non-informative words, such as "the" and "and." Finally, stemming reduced words to their root forms, minimizing redundancy and improving indexing efficiency. Preprocessed data was re-stored in MongoDB for the next phase.

Indexing (Pyterrier)

The indexing process was executed using PyTerrier, where preprocessed data was organized into optimized structures for retrieval. The BM25 relevance model was employed, leveraging term frequency and document length normalization to rank documents effectively. This phase ensured that search results were both accurate and contextually relevant.

Query Handling (Django)

Django managed the querying stage, acting as the intermediary between indexed data and the user interface. User queries submitted through the frontend were processed into API requests by Django. PyTerrier retrieved ranked documents in response, and Django applied dynamic relevance adjustments based on user feedback. Positive feedback increased relevance scores, while negative feedback decreased them, ensuring continuous system improvement.

Frontend Integration (React & Tailwind CSS)

The final stage involved frontend integration, where React dynamically displayed search results. Features such as pagination and clustering were implemented to improve navigation and user experience. Pagination divided results into manageable segments, while clustering grouped related articles and reviews, enabling intuitive exploration. This stage unified the pipeline components, delivering a seamless and engaging user interface.

Scraping

Data collected: ikea articles, ikea reviews

Websites scraped:

- Trustpilot
- IkeaHackers
- The Spruce Ikea

For web scraping we used python libraries “Beautiful soup” and “Selenium” .
Trustpilot and IkeaHackers were scraped with “Beautiful soup”.
TheSpruce had protection against scraping and used dynamically loaded javascript. This is why we used Selenium for it.

Scraper extracted "sourceSite", "articleLink", "articleTitle" and "articleText" and saved it to mongodb.


```
_id: ObjectId('675db634a8649a9469fd3bd7')
sourceSite : "https://www.thespruce.com/search?q=ikea"
articleLink : "https://www.thespruce.com/ikea-curtain-hack-7556707"
articleTitle : "IKEA Curtain Hack: How to Make IKEA Curtains Look Expensive"
articleText : "HOME DESIGN & DECORATING DECORATING DESIGN TIPS
               IKEA Curtain Hack: How…"
```


Pyterrier Preprocessing

The preprocessing phase plays a pivotal role in transforming raw text data into a consistent and structured format suitable for indexing. The process begins by retrieving documents from three MongoDB collections. These datasets, representing articles, hacks, and reviews are appended into a unified DataFrame to standardize processing.

Each document undergoes a series of operations to refine the title and text fields. Tokenization splits the content into discrete words, while stop word removal eliminates commonly used but semantically insignificant words like “the” and “and.” This step reduces noise in the dataset, focusing on terms critical to understanding document relevance. Stemming is then applied to reduce words to their base or root forms, ensuring uniformity across variations of a word (e.g., “running” and “runs” are reduced to “run”). This combination of tokenization, stop word removal, and stemming enhances the efficiency of subsequent indexing and retrieval.

The preprocessing pipeline retains the original fields for frontend use. Processed fields (title and text) are stored alongside raw fields (original title and text) in a dedicated MongoDB collection named processed documents. This setup provides flexibility for both backend indexing and frontend presentation. Additionally, the source link for each document is preserved to maintain traceability. By refining the

content while storing its raw versions, the preprocessing phase ensures both functionality and usability across the system.

Pyterrier Indexing

The indexing stage converts preprocessed data into an optimized format for retrieval. Documents stored in the processed documents collection are integrated into PyTerrier's indexing framework. This step organizes the data to facilitate efficient querying and ranking of search results.

The indexing process begins by extracting all relevant fields, including document numbers, processed titles and texts, raw titles and texts, and source links. These fields are indexed into a structured format using PyTerrier's IterDictIndexer. This ensures that both processed and raw versions of the data are available for retrieval, supporting use cases such as query handling and user feedback-driven reranking.

The BM25 relevance model is employed during this phase to assign scores to documents based on term frequency, document length, and term-specific characteristics. This scoring mechanism prioritizes documents most relevant to user queries. The index is stored at a designated location (ikea_index directory), ensuring it is readily accessible for querying.

By organizing the processed collection into a high-performance index, this stage bridges the preprocessing pipeline with the querying process. The indexing ensures that the system can retrieve, rank, and present results quickly and accurately, laying the foundation for an effective information retrieval experience.

Django Query Handling

The querying stage was managed by Django, which served as the intermediary between the indexed data and the user interface. Django provided the backbone for the API framework, enabling seamless communication between the frontend and backend. By defining API routes and views, the Django setup facilitated efficient query handling, retrieval, and dynamic relevance adjustments based on user feedback.

When a user submitted a query via the frontend, it was sent to the backend as an API request. The search function in Django processed this request, using PyTerrier to retrieve documents from the indexed dataset. The results were ranked based on their initial relevance scores calculated by the BM25 model. These results were further refined through dynamic relevance scoring, leveraging user feedback provided during the query session.

User Feedback (Complex Feature)

The dynamic feedback mechanism allowed users to label specific documents as "relevant" or "irrelevant". Documents marked as relevant received a 50% boost to their relevance scores, while those marked as irrelevant were penalized by a 50% reduction in their scores. These adjustments were stored in a global feedback structure, ensuring the scores persisted across subsequent interactions with the same query. The

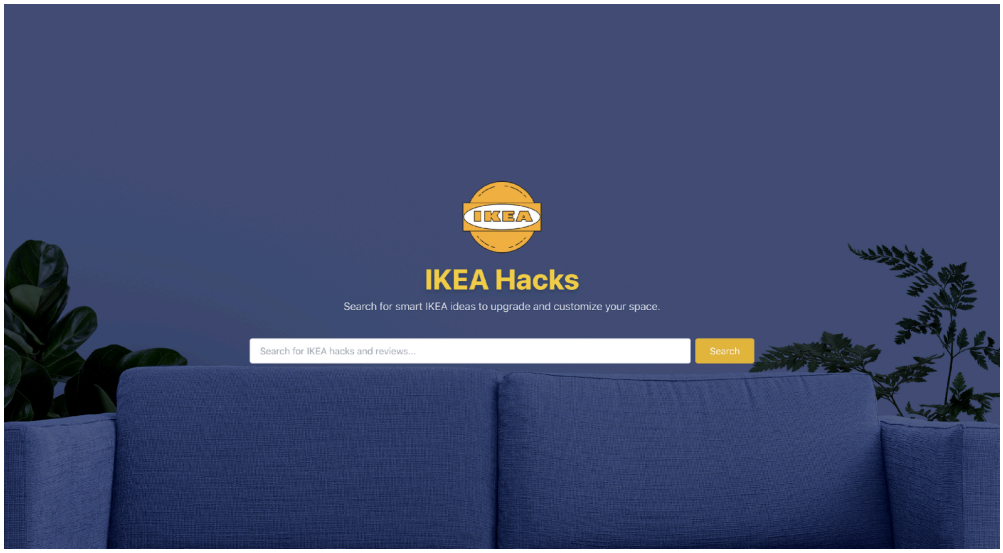
system then re-sorted the results in descending order of the updated relevance scores, ensuring that the most relevant documents appeared at the top of the list.

Django's querying logic not only handled the immediate retrieval of results but also dynamically adapted to user preferences, making the system highly responsive to feedback. This approach ensured that the search experience improved over time, tailoring results to individual user needs. Once the refined results were prepared, Django formatted the data into a structured response containing key document information, such as titles, text, scores, and source links. This response was then sent back to the React-based frontend, where it was displayed dynamically for the user.

By seamlessly connecting the PyTerrier indexing system to the frontend, Django acted as the central processing unit of the system. It provided robust query handling, real-time adaptability through feedback integration, and efficient communication between components, ensuring a smooth and responsive search experience for users. This architecture underscored Django's pivotal role in bridging the gap between data retrieval and user interaction.

Frontend Integration (React & Tailwind CSS)

The frontend, built with React and styled using Tailwind CSS, served as the primary interface for user interaction. It provided key functionalities such as query submission, dynamic result presentation, and real-time feedback mechanisms, ensuring an intuitive and engaging user experience.



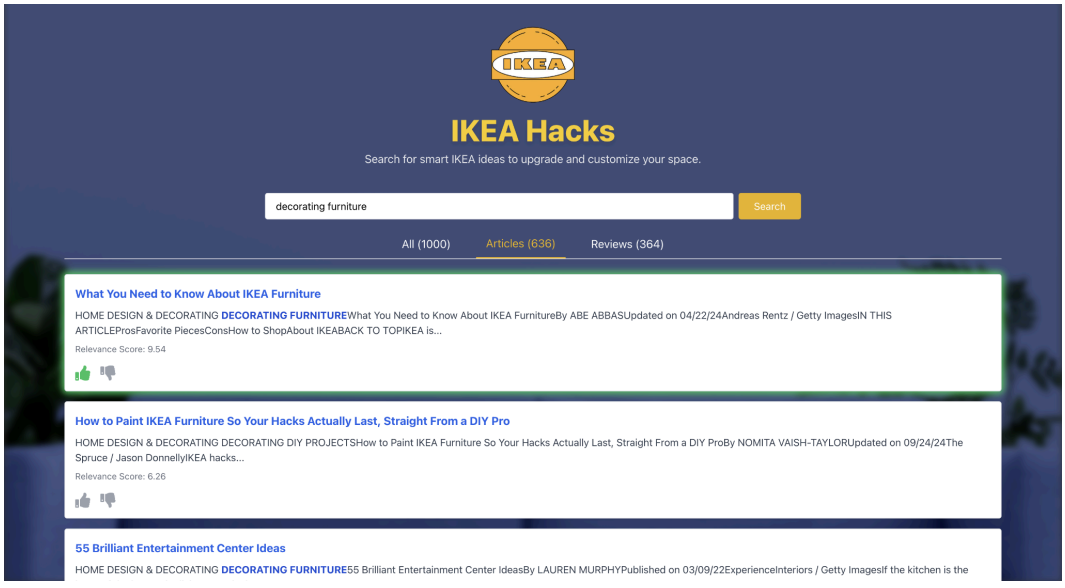
The search interface featured a simple yet effective feature: **results snippets**. For articles, concise summaries were presented, showing 2-3 lines of content with query terms highlighted, similar to Google-style search results. For reviews, the full text was displayed instead of snippets, enabling users to read complete reviews directly within the app. This approach encouraged users to stay on the platform

while engaging deeply with the content.

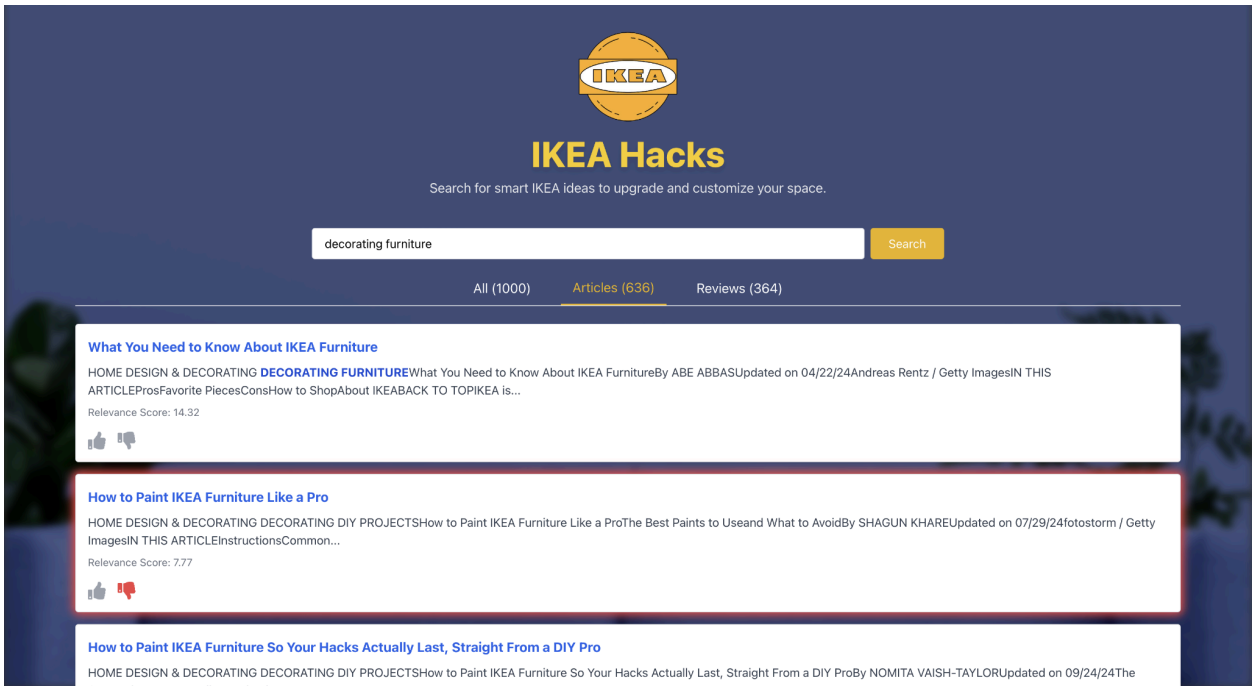
Dynamic Feedback and Relevance Updates

Interactive feedback buttons allowed users to mark results as “relevant” or “irrelevant.” This feedback was sent to the backend, where relevance scores were dynamically adjusted (+50% for relevant, -50% for irrelevant) and the results were resorted based on updated relevance scores. The updated results were returned to the frontend, which automatically re-ranked and reordered the displayed results in real time based on the backend response. This feedback loop improved result relevance continuously, ensuring a highly tailored user experience.

Positive Feedback: Green Highlight (relevance score increased by backend +50%)



Negative Feedback: Red Highlight (relevance score penalized by backend -50%)

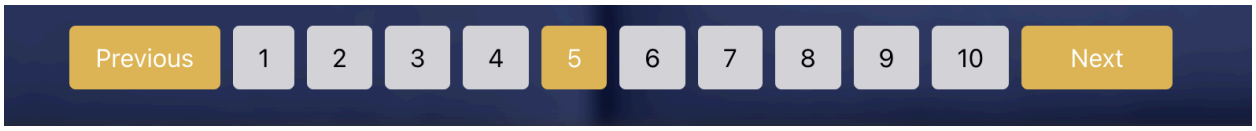


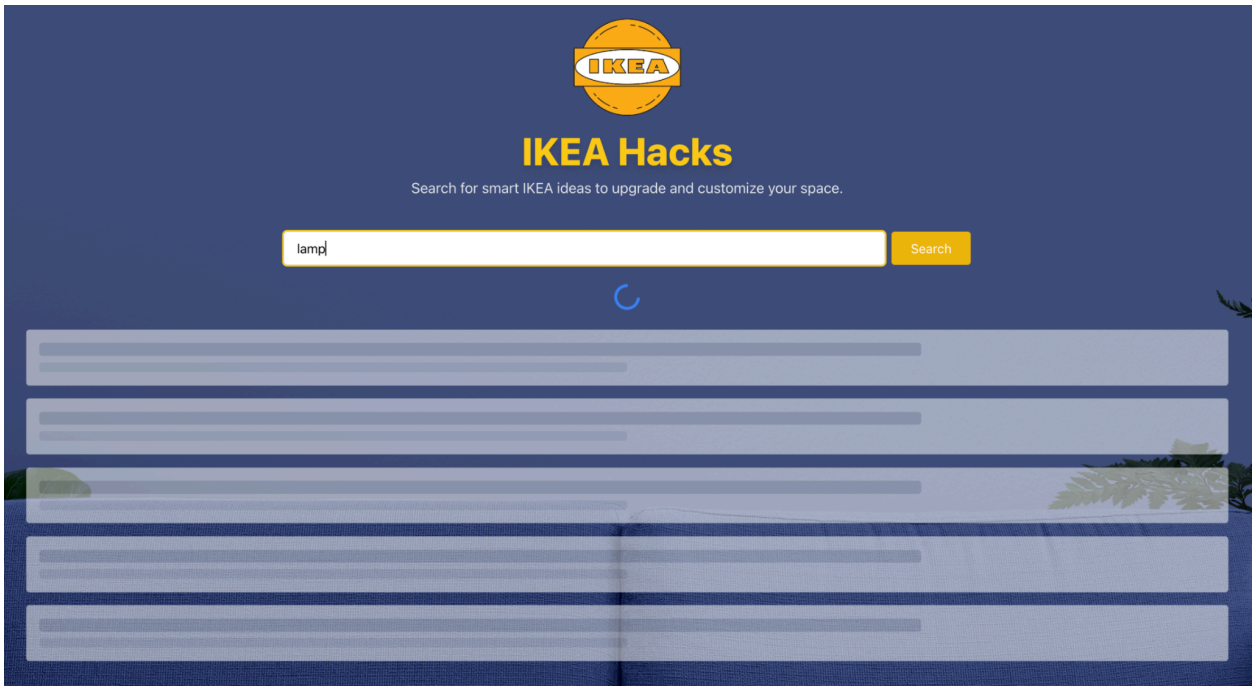
Results Clustering and Navigation

The system grouped results into clusters, such as “All”, “Articles”, and “Reviews”, sorted in descending order of the number of results they contained. Articles were displayed as concise snippets for quick scanning, while reviews, such as those from Trustpilot, were shown in full text for comprehensive reading. This clustering system provided an intuitive structure, making it easy for users to browse and analyze information efficiently based on content type.

Pagination and Loading States

To manage large datasets, pagination divided results into manageable pages with intuitive navigation controls. While waiting for results to load, the frontend displayed **loading states**, including a **spinner animation** and **skeleton loaders**. Skeletons provided a visual placeholder, simulating the structure of search results and improving the user experience during fetch operations. This approach ensured seamless transitions and minimized perceived wait times, maintaining engagement even during loading phases.





Styling and Usability
Tailwind CSS provided a clean, modern, and fully responsive design. Key components, such as the search bar, feedback buttons, results clustering, and loading indicators, were designed for both functionality and aesthetics. These design elements ensured consistent usability across different devices, contributing to a polished and professional interface.

User Evaluation

The system was evaluated using the **System Usability Scale (SUS)**, where three users were asked to perform tasks such as submitting queries, exploring clustered results, and interacting with the feedback system. The system achieved an average SUS score of **91**, indicating excellent usability and strong user satisfaction.

Users highlighted the intuitive interface, especially the ability to search for articles tailored to their needs related to IKEA. They found it highly convenient to browse through clustered results and quickly access external articles for detailed information through clear links provided in our system. The inclusion of Google-style snippets for articles allowed users to assess relevance at a glance, while full-text displays for reviews enabled comprehensive reading without leaving the platform.

The dynamic feedback system, which adjusted relevance scores and reordered results in real time, was noted as a standout feature, improving search relevance with minimal effort. Users also appreciated the logical clustering of results into categories like “Articles” and “Reviews,” which simplified navigation and made exploring content types straightforward. Pagination provided seamless browsing for large result sets, further enhancing the user experience.

Overall, participants were able to complete tasks successfully and efficiently, confirming that the system meets expectations for usability, adaptability, and relevance. The evaluation validated its design as both functional and user-centric, making it a robust tool for information retrieval.

Install and Run Instructions

1. Installation

To set up the system, follow these steps to install the necessary components:

1. **PyTerrier, pandas, and NLTK**
   Install the required Python libraries for indexing, retrieval, and text processing if you have not installed them yet. These libraries form the backbone of the system's backend processes.

   pip install python-terrier pandas nltk

2. **Django**
   Install Django and its dependencies. Django is the backend framework that handles API requests and server management.

   pip install django djangorestframework django-cors-headers

3. **MongoDB**
   Set up MongoDB for database management. Ensure the MongoDB service is running to manage any necessary collections or data.

---

2. Run Instructions

1. **Backend Setup**
   Navigate to the backend directory and set up the database schema by running migrations. The ikea_index directory already contains preprocessed and indexed docs, so you can skip the MongoDB setup. After making migrations once, the backend server can then be started using Django's runserver command.

   cd backend

   python manage.py makemigrations

   python manage.py migrate

   python manage.py runserver

2. **Frontend Setup**
   Navigate to the frontend directory (ikea_search) and install all the required dependencies. Once

the dependencies installation is complete, start the React development server to make the user interface accessible.

cd ikea_search

npm install

npm start

---

3. Final Steps

Run the system as follows:

- Start the Django backend server on one terminal.
- Start the React frontend server on another terminal.

Once both servers are running, the application will be accessible through the frontend at *http://localhost:3000.*

Conclusion

The IKEA Hacks information retrieval system integrates modern technologies to provide an efficient, user-centric search experience. With a robust backend powered by Django and PyTerrier, and an intuitive React and Tailwind CSS frontend, the system enables users to seamlessly explore and access relevant IKEA-related content. The implementation includes a basic feature: **Results Snippets**, and two complex features: **User Feedback with dynamic relevance updates** and **Results Clustering** for intuitive navigation. This approach demonstrates practicality, scalability, and high usability, effectively meeting the project's objectives.