Faculty of Computer Science and Engineering Skopje

[Department of Intelligent Systems]

Technical Report

WebKnoGraph: Structuring Internal Links with Embeddings and Graph-Based Link Prediction

sponsored by data & testing partners

June 21, 2025

Author: Emilija Gjorgjevska

Contributors

- PhD Miroslav Mirchev
- PhD Georgina Mircheva

"Koji su dozvolili da sanjam, — "Who let me dream,
koji su omogucili da krenem, — who made it possible for me to go,
koji su virovali da cu stici, — who were waiting for me to arrive,
Vama." — To you."
- Denny Vrandečić

License

History

Version	Date	Comment
1.0	04/09/2024	Open-sourcing v1.0 of the technical solution under Apache 2.0 license
Version	Date	Comment
2.0	20/06/2025	Open-sourcing v2.0 of the technical solution under Apache 2.0 license

Sponsors: WordLift.io and Kalicube.com

Contents

Sponsors: WordLift.io and Kalicube.com

Sponsors: WordLift.io and Kalicube.com

1 Acknowledgements

This work would not have been possible without the support and contributions of several individuals and organizations to whom I am deeply grateful.

First and foremost, I would like to express my heartfelt thanks to Jason Barnard, Andrea Volpini, and their teams at Kalicube.com and WordLift.io for providing invaluable data and permission to work with their resources which served as the foundation of this research. Your expertise and willingness to share know-how and data significantly enhanced the scope and depth of this work.

I am also profoundly grateful to Dejan Petrovic (Dan Petrovic) for his insightful suggestions on crawler ideas. Dejan's innovative approaches and practical advice were instrumental in shaping the first technical aspects of this research.

Special thanks go to my esteemed professors, PhD Miroslav Mirchev and PhD Georgina Mircheva, whose guidance and support throughout this process were indispensable. Your ideas for the technical implementations helped bridge theory with practice, ensuring the academic rigor of this work.

To Sara Moccand Sayegh, your tireless efforts in brainstorming, collecting data, and managing technical tasks were invaluable. Our brainstorming discussions, your unwavering encouragement, and your belief in this project pushed it to new heights. I couldn't have done this without your partnership and friendship.

Finally, I would like to thank my family, friends, colleagues, and the international SEO community for their unwavering support and encouragement throughout this journey. Your reality checks kept me grounded and motivated when I needed it most. Your belief in my abilities gave me the strength to persevere and complete this project.

This work is not just mine—it belongs to all of you, and I am deeply appreciative of the unique and irreplaceable role each of you played in bringing it to life.

Sincerely,

Sponsors: WordLift.io and Kalicube.com

2 Introduction

In today's digital landscape, optimizing your website for search engines is essential for driving organic traffic and achieving online success. Internal linking (linking to relevant pages within your site) is a key aspect of SEO. However, traditional internal linking methods can be time-consuming and complex, especially as a website grows in size and depth, requiring more advanced strategies.

Recent advancements in Natural Language Processing (NLP) and Machine Learning (ML) offer new opportunities in SEO, particularly in content analysis and link prediction. These technologies allow SEO professionals to automate and optimize internal linking, ensuring each link has a strategic purpose. Graph-based models, such as GraphSAGE, predict optimal internal links by analyzing the structure and relationships between pages. These models utilize vector embeddings to represent content in a multidimensional space, capturing semantic similarities and relevance.

Our technical report examines the integration of data processing, vector embeddings, and GraphSAGE-like algorithms to refine internal linking strategies for SEO.

2.1 Aims and objectives

We aim to develop an advanced framework for optimizing internal linking strategies within websites to improve SEO performance, user navigation, and content discoverability. By integrating Natural Language Processing (NLP), Machine Learning (ML), and graph-based algorithms, the research addresses the limitations of traditional internal linking methods, particularly for large-scale websites. Some of our key objectives are provided below.

- Analyze the current limitations: examine existing internal linking practices and identify their shortcomings, particularly in handling complex website architectures with extensive content depth and breadth.
- Explore advanced technologies: investigate the application of NLP, ML, and graph-based models, such as GraphSAGE, for predicting and generating optimal internal links.
- Develop a framework: create a comprehensive framework that combines data processing, vector embeddings, and graph-based algorithms to automate the internal linking process.
- Evaluate effectiveness: apply the proposed framework to a real website.
- Provide strategic insights: offer practical recommendations for SEO professionals on implementing advanced internal linking strategies to enhance search engine visibility and improve user experience.

2.2 Target audience

This research is aimed at tech-savvy marketers and marketing engineers who possess a strong understanding of advanced data analytics and are familiar with the tools and techniques involved in data-driven marketing strategies. Our target audience consists of professionals who either have direct experience with Python and other programming languages or have access to development support within their teams. These individuals are not only proficient in interpreting and utilizing data but are also comfortable working with technical tools to optimize marketing efforts. Whether they are directly coding or collaborating with developers, these experts are well-versed in leveraging data and technology to enhance customer engagement, streamline marketing operations, and drive business growth.

3 Problem Statement

Large websites, especially those in enterprise settings, often contain millions of pages that need to be logically organized and interconnected from both a graph and semantic perspective. The main challenge is to implement intelligent internal linking strategies to achieve this. This is complicated by the fact that webmasters and SEO engineers often lack direct access to the front or backend systems, making large-scale changes difficult.

Effective scalability requires the ability to quickly adapt to new trends on a biweekly or monthly basis, with site architecture automatically adjusted based on SEO priorities set by professionals. We aim to develop a system that provides a robust foundation for addressing this challenge. By utilizing the website's graph structure and semantic data from each webpage, we seek to create a scalable, adaptive solution that improves the organization and connectivity of large-scale websites.

4 Literature Survey

The importance of internal linking in SEO has been extensively discussed in recent literature, highlighting the role of graph theory and dynamic linking strategies in enhancing website structure and user navigation. A study by Omio Engineering[1] outlines how graph theory can be applied to optimize internal linking by treating a website as a network of interconnected nodes, where pages are connected based on relevance and authority. This approach not only improves search engine crawlability but also enhances user experience by guiding visitors through a logical pathway of content that aligns with their search intent.

Another perspective emphasizes that the optimal internal linking structure varies depending on the business model. Kevin Indig[2] discusses how e-commerce sites, content-driven platforms, and service-based websites each require different linking strategies to maximize SEO benefits. For instance, e-commerce sites may benefit from

a hierarchical structure that connects product pages to category pages, while content-driven platforms might adopt a more mesh-like structure to promote related articles. This adaptability ensures that the linking strategy aligns with each business model's specific objectives and user behavior.

WordLift[3] introduces the concept of dynamic internal linking, which automatically generates and updates internal links based on real-time data and evolving SEO priorities. This method uses natural language processing (NLP) and machine learning (ML) to continuously assess the relevance and performance of content, dynamically adjusting links to reflect changes in user behavior and search trends. This approach is particularly effective in large-scale websites where manual linking is impractical and scalability is crucial.

Together, these studies provide a comprehensive view of current approaches to internal linking in SEO, showcasing the need for adaptable and scalable strategies to meet the diverse requirements of different website types and industry contexts.

4.1 Overview of website hierarchies

Website hierarchies [4] are essential for effective information architecture and user navigation on digital platforms. Different architectures—flat, hierarchical, matrix, database, and mixed offer unique ways to organize content, each suited to specific website types and sizes. Flat architectures maximize PageRank flow but struggle with larger sites, while hierarchical structures use parent-child relationships and taxonomies to create more organized, content-rich experiences. Matrix architectures focus on interconnectedness, offering dynamic exploration, and database-driven sites adapt content dynamically based on user input, providing flexibility and scalability.

A key insight here is that a website is a link network, represented as a Directed Cyclic Graph (DCG), where each page represents a node, and internal links between pages form the directed edges:

- Websites represent *directed graphs* because links have a specific direction from the source page to the destination page.
- Websites represent *cyclic graphs* since the graph has loops or cycles. For example, the second-level nodes (links) from the picture below link to each other (also the leaves back to the homepage).
- Website graphs are *weighted graphs* since links have different importance (weights or costs).

Intelligent website structures enable efficient navigation and topical clustering[5], crucial for search engine optimization (SEO). Let's explore some website hierarchies:

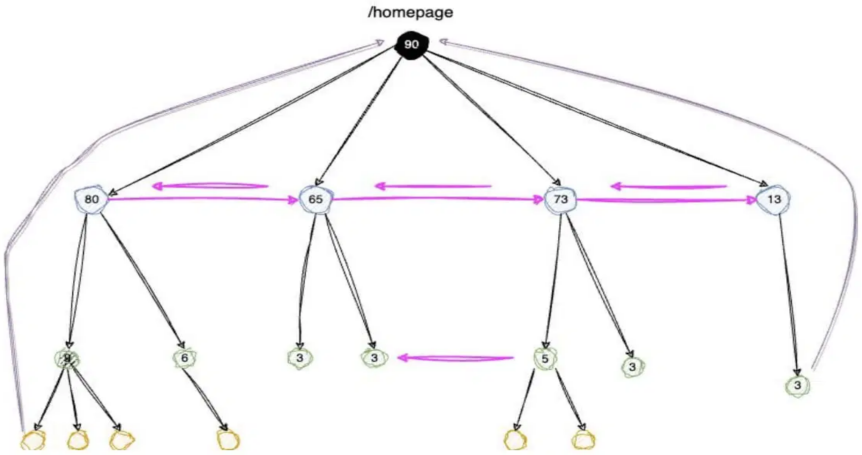

Figure 1: Website as a Directed Cyclic Graph (DCG)

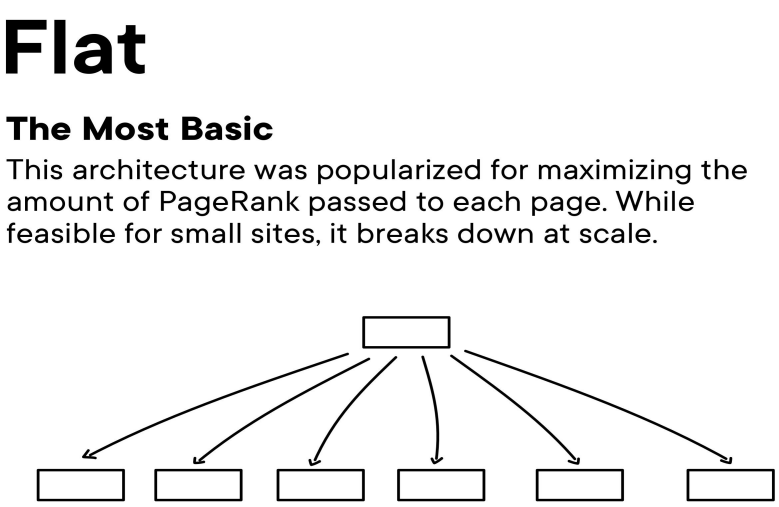

Figure 2: Flat hierarchy

Considering the diversity of website hierarchies influenced by factors such as industry, business model, and technical configuration, our focus is on identifying effective linking candidates in general, rather than prescribing specific connections. However, our research will also provide insights and suggestions on potential ways these links can be established.

Sponsors: WordLift.io and Kalicube.com

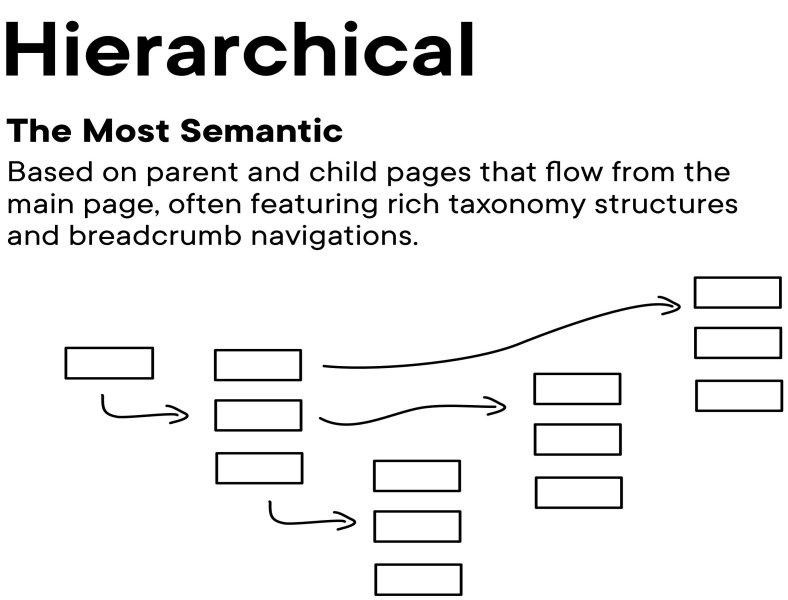

Figure 3: Hierarchical hierarchy

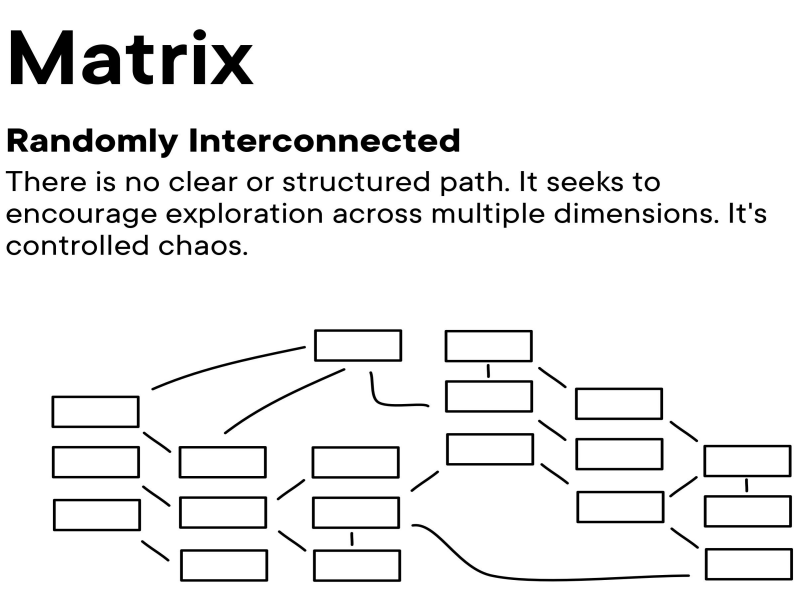

Figure 4: Matrix hierarchy

Database

Completely Dynamic

Think of sites like Pinterest or Google, and many types of marketplaces. Aside from the main navigation, content is generated based on input/output.

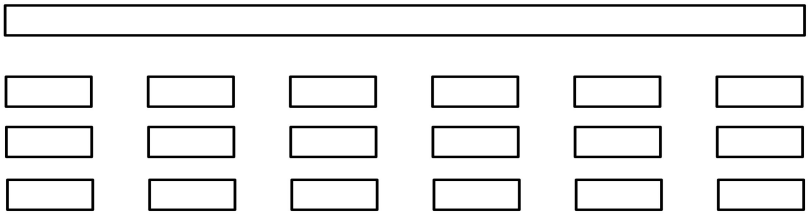

Figure 5: Database hierarchy

Flat + Matrix

Ex.) A blog or news website

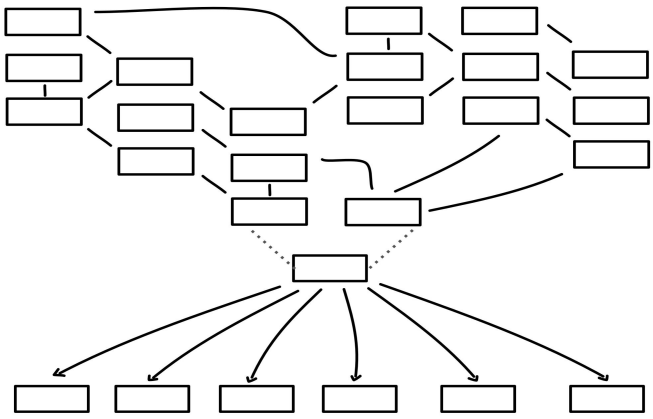

Figure 6: Flat + matrix hierarchy

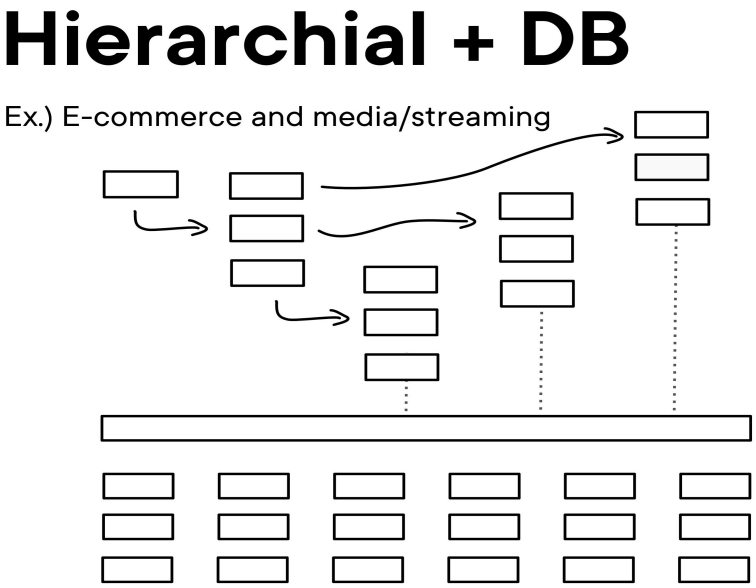

Figure 7: Hierarchical + DB hierarchy

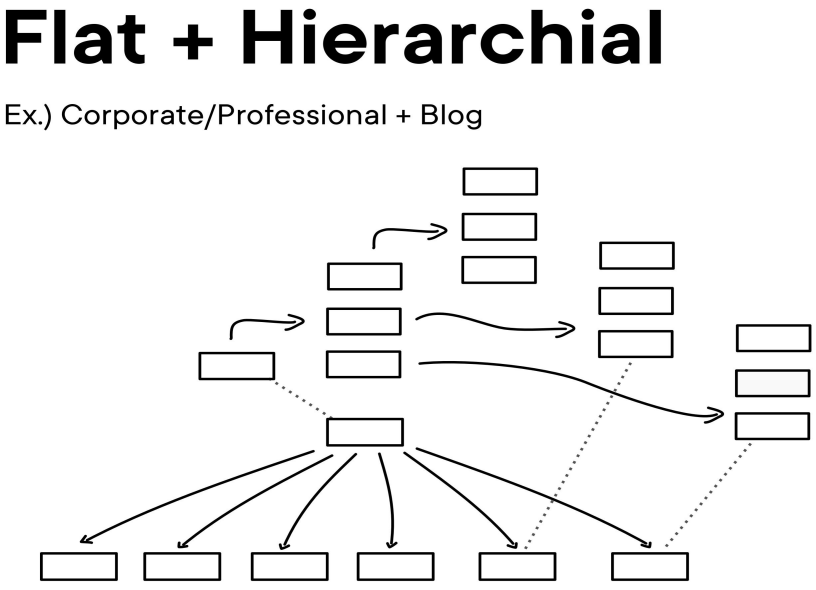

Figure 8: Flat + Hierarchical hierarchy

Sponsors: WordLift.io and Kalicube.com

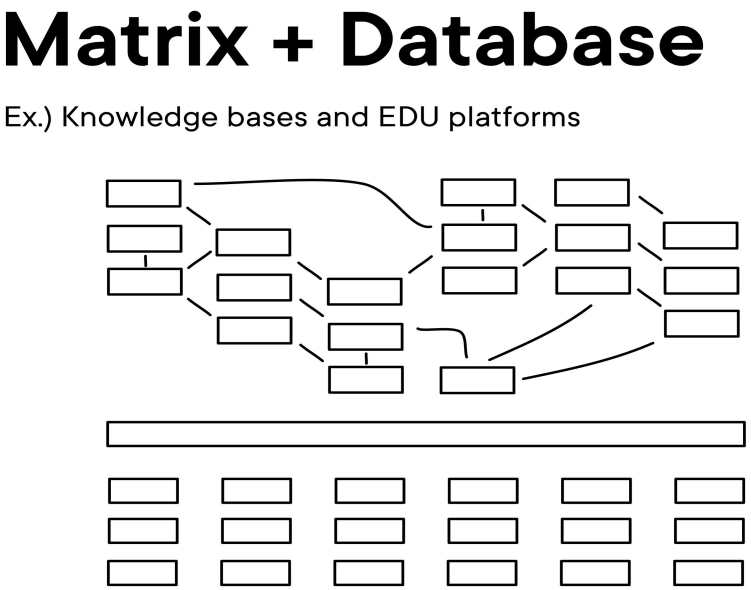

Figure 9: Matrix + DB hierarchy

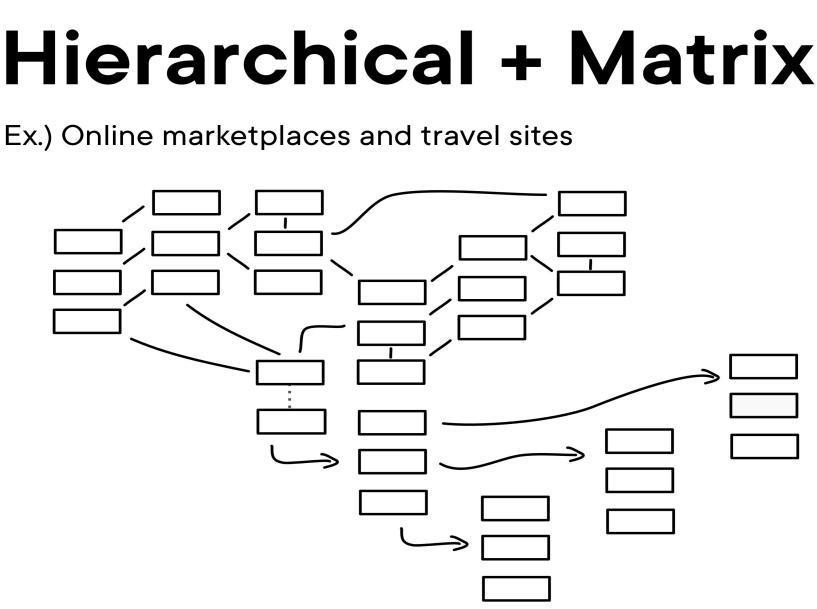

Figure 10: Hierarchical + Matrix hierarchy

Sponsors: WordLift.io and Kalicube.com

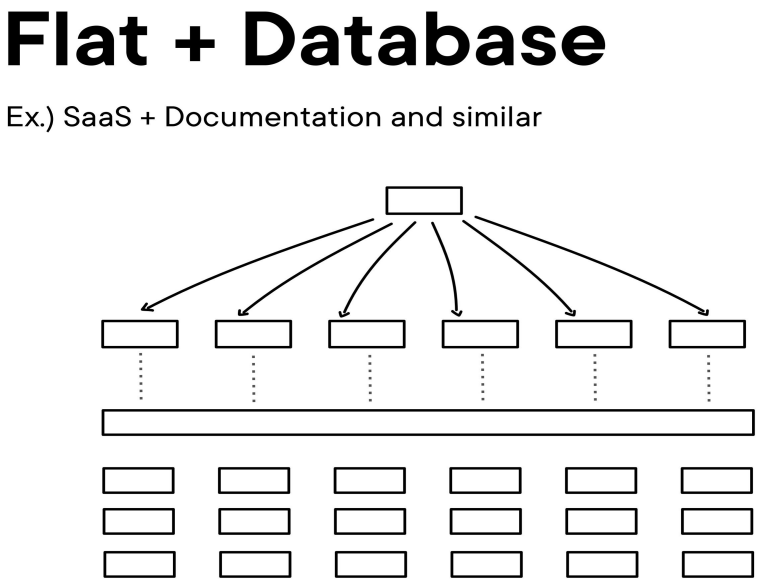

Figure 11: Flat + DB hierarchy

5 The Dataset

The WebKnoGraph project relies on a meticulously structured data pipeline, where various modules generate and transform datasets to facilitate web graph analysis and link prediction. All processed and generated datasets are stored persistently on Google Drive, allowing for modular execution and reproducibility.

5.1 Overview of Data Flow

The data journey within WebKnoGraph progresses through several key stages:

- Raw web content acquisition: a dedicated web crawler captures HTML content from a target website.
- Semantic feature extraction: this HTML content is transformed into structured text and then into numerical vector embeddings, representing the semantic meaning of each page.
- Structural link graph generation: a separate crawler extracts the internal linking relationships, forming the raw graph structure.
- Graph analysis & feature augmentation: the link graph is analyzed for centrality measures (PageRank, HITS), and additional features like folder depth are computed.

Sponsors: WordLift.io and Kalicube.com

- Graph Neural Network (GNN) training: the augmented link graph and semantic embeddings are combined to train a GNN model.
- Prediction artifacts: the trained GNN model and related mappings are saved for future link prediction tasks.

5.2 Dataset Details

The dataset comprises information gathered through web crawling of English texts from the learning space directory of the Kalicube website. The primary elements of this dataset include URLs, raw HTML content, HTTP response codes, crawl depth, and inlinks associated with each URL, used to create the final website graph. All generated datasets are stored in the `/content/drive/My Drive/WebKnoGraph/data/` directory by default, ensuring modularity and persistent storage across Colab sessions.

1. **Uniform Resource Locators (URLs)**: fundamental elements of the dataset, serving as the unique identifiers for each web page crawled. Each URL represents a specific web resource. The project ensures that only links representing web pages (article-based content) are included, excluding other types of online content. URLs are stored as strings and include components such as protocol, domain name, path, query parameters, and fragment identifiers. These components allow for the precise identification and retrieval of content from the web.
    - Data type: String.
    - Definition: The string uniquely identifies a resource on the web.
2. **Raw HTML**: the full content of the HTML source code retrieved from each URL. This data encapsulates the structure and content of a webpage, including tags, metadata, text content, links, and embedded media. Raw HTML is critical for further analysis, such as parsing for specific elements, extracting metadata, and understanding the semantic structure of the webpage.
    - Data type: String.
    - Definition: The complete HTML rendered source code of a webpage as retrieved by the web crawler.
3. **HTTP Response Code**: a numerical status code sent by a web server in response to a request made by the web crawler. This code provides insight into the outcome of the HTTP request, indicating whether it was successful, resulted in a redirection, or encountered an error (e.g., 200 (OK), 301 (Moved Permanently), 404 (Not Found), and 500 (Internal Server Error). These codes are integral to understanding the accessibility and status of each URL.
    - Data type: Integer.

- Definition: A three-digit code that indicates the outcome of the HTTP request for a specific URL.

4. **Folder Depth**: the level of distance from the root URL to the specific URL in question. It represents how many "clicks" away a particular page is from the homepage or the initially targeted URL. A depth of 0 indicates the root URL, a depth of 1 refers to a direct link from the root, and so on. Folder depth is recorded as an integer in the dataset.

   - Data type: Integer.
   - Definition: The number of links or levels between the root URL and the specific URL.

5. **Inlinks or inbound links**: hyperlinks from other URLs that point to a particular webpage within the dataset. The count and source of inlinks are significant for assessing the importance and authority of a webpage, as pages with a higher number of inlinks are typically considered more valuable for SEO purposes. In the dataset, inlinks are often represented as a list of URLs that link to the specific page, providing context for the page's position within the web structure.

   - Data type: List of strings.
   - Definition: URLs of web pages that contain links pointing to the specific URL within the dataset.

6. **URL Embeddings**

   - Dataset Name: url_embeddings
   - Source Module: Embeddings Pipeline (notebooks/embeddings_ui.ipynb)
   - Input Data: crawled_data_parquet
   - Format: Parquet files (e.g. /data/url_embeddings/*.parquet)
   - Key Columns: URL, Embedding (dense vector).
   - Purpose: Provides semantic features for each crawled URL. These embeddings are crucial inputs for the GNN Link Prediction model, allowing it to understand the content similarity between pages.

7. **Link Graph Edges**

   - Dataset Name: `link_graph_edges.csv`
   - Source Module: Link Graph Extractor (`notebooks/link_crawler_ui.ipynb`)
   - Format: CSV file
   - Key Columns: FROM, TO (representing a directed hyperlink from one URL to another).

- Purpose: Captures the structural relationships (internal links) between web pages. It forms the core graph structure for both PageRank/HITS analysis and the GNN Link Prediction model.

8. **URL Analysis Results**

   - Dataset Name: `url_analysis_results.csv`
   - Source Module: PageRank & HITS Analysis (`notebooks/pagerank_ui.ipynb`)
   - Input Data: `link_graph_edges.csv`
   - Format: CSV file
   - Key Columns: URL, Folder_Depth, PageRank (score), Hub Score (for HITS analysis), Authority Score (for HITS analysis).
   - Purpose: Provides derived structural features and centrality measures for each URL. PageRank and Folder Depth are used as node features in the GNN, and all scores are used for manual analysis and filtering.

9. **GNN Prediction Model Artifacts**

   - Dataset Name: `prediction_model/`
   - Source Module: GNN Link Prediction & Recommendation Engine (notebooks/link_prediction_ui.ipynb)
   - Input Data: link_graph_edges.csv, url_embeddings/
   - Format: Various files (e.g., .pth for model state, .pt for tensors, .json for metadata)
   - Key Contents:
     - graphsage_link_predictor.pth: Trained weights of the GraphSAGE model.
     - final_node_embeddings.pt: Final learned node embeddings after GNN training.
     - model_metadata.json: Contains mappings (e.g., url_to_idx), and model architecture parameters (in_channels, hidden_channels, out_channels).
     - edge_index.pt: The edge index tensor used during training.
   - Purpose: These artifacts are the persistent output of the GNN training process, enabling the Recommendation Engine to load the trained model and generate new link predictions without retraining.

5.3 On the Use of First-Party Data Sources in SEO

When optimizing internal linking architecture for SEO, leveraging first-party data sources is crucial to understanding how users and search engines interact with a website. First-party data includes information collected directly from the website, such

as crawl data, Google Search Console (GSC) metrics, Google Analytics (GA) reports, and log file data. However, the utility and applicability of these data sources vary significantly depending on the context and the technical expertise required to manage them.

5.3.1 Crawl Data

Crawl data is one of the most valuable sources for SEO analysis because it provides a comprehensive view of a website's structure and how search engines navigate it. This data includes all the pages that the crawler has visited, the links between them, and metadata such as response status codes. By analyzing crawl data, issues like broken links, orphaned pages, and inefficient internal linking structures that might hinder SEO performance can be identified. Crawl data is prioritized in this project due to:

- **Direct control**: SEO professionals have direct control over crawl data, making it a reliable and actionable data source.
- **Comprehensive view**: Crawl data offers a full picture of the website's internal structure, which is essential for optimizing internal links. **No sampling issues**: Unlike some first-party and third-party tools, crawl data is typically not sampled, providing a complete dataset for analysis.

Given these advantages, crawl data is prioritized in this project as it is accessible, actionable, and relevant for all users.

5.3.2 Google Search Console (GSC)

Google Search Console is a valuable tool that provides insights into how Google's search engine interacts with a website, offering metrics such as click-through rate (CTR), impressions, clicks, and queries. However, GSC data is often sampled, which can introduce inaccuracies in metrics like CTR and impressions, making them less reliable for detailed SEO analysis. GSC metrics are excluded from the core analysis in this project to:

- **Avoid sampling issues**: GSC data is sampled, so it may not provide a complete picture, especially for websites with large volumes of traffic or numerous pages.
- **Maintain data integrity**: for a project focused on precise internal linking optimization, relying on unsampled, complete data is essential. Sampled data could lead to misleading conclusions.
- **Enable applicability**: to keep the project applicable to a semi-technical audience, complexities and potential pitfalls of sampled data are avoided, focusing instead on more reliable sources.

5.3.3 Google Analytics (GA)

Google Analytics offers valuable insights into user behavior, but similar to GSC, its data can be sampled, especially for high-traffic sites, affecting metrics like session duration, bounce rate, and pageviews. For reasons analogous to those for GSC, metrics from Google Analytics are also excluded from the core analysis of this project to maintain data integrity and applicability.

5.3.4 Log File Data

Log file data provides a granular view of every request made to a server, including those by search engine bots. This data can offer insights into how search engines crawl a site, which pages are prioritized, and where potential issues like crawl errors might occur. However, accessing and analyzing log files can be challenging due to:

- **Technical complexity**: managing and analyzing large log files requires significant technical expertise.
- **Security and protocols**: log files contain sensitive information, and accessing them often involves strict security protocols.
- **Accessibility**: This data source is less accessible to us as SEO professionals.
- **Applicability**: by focusing on data sources more easily accessible and understandable to a broader audience, the project remains practical and applicable, even for semi-technical users.

5.3.5 Prioritization and Practicality

The decision to prioritize certain data sources, such as crawl data, over others, like sampled GSC or GA metrics and complex log file data, is driven by the goal of making the project more accessible and hands-on to a broader audience. For semi-technical users, understanding and implementing complex data-driven strategies can be challenging. By focusing on data that is reliable, unsampled, and easier to manage, the project remains practical for SEO professionals at all levels of technical expertise. This approach not only simplifies the process of internal linking optimization but also ensures that the insights derived are based on accurate and actionable data, enabling users to implement effective SEO strategies without requiring deep technical knowledge or access to sophisticated data processing tools.

6 System Design and Requirements Engineering

A crucial aspect of implementing advanced SEO strategies is the ability to process and analyze large volumes of data efficiently. Developing a scalable, cost-effective crawler is essential for handling the complexities of large-scale internal linking optimization.

A well-designed crawler must be capable of processing thousands of pages, to extract content and link data necessary for building a comprehensive internal linking strategy. To make this feasible on a single machine or a cloud-based service like Google Colab Pro, several factors need to be considered, translated into specific technical implementations, to meet the demands of large-scale web data processing.

7 Architectural Principles and Design Patterns

The architecture of the WebKnoGraph project is founded on robust software engineering principles to ensure maintainability, extensibility, and clarity across its diverse modules.

7.1 SOLID Principles

These principles guide the design of object-oriented systems towards more understandable, flexible, and maintainable code.

- **Single Responsibility Principle (SRP)**: each class or module is designed to have one and only one reason to change. For example, *CSVLoader* is solely responsible for loading CSV data, *URLProcessor* for URL specific calculations, and *HttpClient* for managing HTTP requests. This separation prevents a single change in requirements from affecting multiple functionalities.
- **Open/Closed Principle (OCP)**: software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification. The *CrawlingStrategy* abstract base class exemplifies this: new crawling strategies (e.g., a custom priority queue based strategy) can be added by extending *CrawlingStrategy* and implementing its abstract methods, without altering the existing *BFSCrawlingStrategy* or *DFSCrawlingStrategy* implementations or the *EdgeCrawler* that utilizes them.
- **Liskov Substitution Principle (LSP)**: subtypes must be substitutable for their base types without altering the correctness of the program. In WebKnoGraph, *BFSCrawlingStrategy* and *DFSCrawlingStrategy* (subclasses of *CrawlingStrategy*) can be used interchangeably by the crawler services (*WebCrawler*, *EdgeCrawler*), as they adhere to the interface defined by *CrawlingStrategy*.
- **Interface Segregation Principle (ISP)**: clients should not be forced to depend on interfaces they do not use. The *ILogger* interface defines a minimal set of logging methods (*info*, *error*, *exception*, *debug*, *warning*), ensuring that any logging component only implements the necessary functions.
- **Dependency Inversion Principle (DIP)**: high-level modules should not depend on low-level modules; both should depend on abstractions. Abstractions

should not depend on details; details should depend on abstractions. Core services like *EmbeddingPipeline* and *PageRankService* depend on abstract interfaces (like *ILogger*) or abstract base classes (*CrawlingStrategy*), rather than directly on concrete implementations. This makes the system flexible and easier to test.

7.2 Object-Oriented Programming (OOP) Principles

Core OOP concepts are applied throughout the design:

- **Encapsulation**: data and the methods that operate on that data are bundled within classes, and internal states are often hidden. Examples include *HttpClient* encapsulating session management details, and *StateManager* managing database interactions for crawl frontiers.
- **Inheritance**: used to establish relationships between classes, allowing subclasses to inherit properties and behaviors from a base class. *BFSCrawlingStrategy* and *DFSCrawlingStrategy* inheriting from *CrawlingStrategy* is a prime example.
- **Polymorphism**: the ability of different classes to respond to the same method call in their own specific ways. The *WebCrawler* can operate with either *BFSCrawlingStrategy* or *DFSCrawlingStrategy*, as both implement the *CrawlingStrategy* methods polymorphically.

7.3 Good Design Patterns

Specific design patterns are implicitly or explicitly used to solve common architectural problems:

- **Strategy Pattern**: explicitly implemented through the *CrawlingStrategy* hierarchy, allowing the crawling algorithm (BFS or DFS) to be chosen and swapped at runtime without changing the *WebCrawler*'s core logic.
- **Repository Pattern**: *StateManager*, *LinkGraphStateManager*, and *EmbeddingStateManager* act as specialized repositories, abstracting away the details of data persistence (SQLite, Parquet files) for different types of application state.
- **Orchestration/Facade Pattern**: high-level service classes like *WebCrawler*, *EmbeddingPipeline*, *EdgeCrawler*, and *PageRankService* act as orchestrators or facades, providing a simplified interface to a complex subsystem of interacting components.

7.4 Resumable Operations

A critical design requirement for long-running processes in potentially unstable cloud environments or sudden interrupts for whatever reason.

The implementation of resumable crawls and embedding generation workflows (via *StateManager*, *LinkGraphStateManager*, *EmbeddingStateManager*) explicitly saves the state of operations (e.g., URLs in the frontier, already processed URLs). This prevents redundant processing and saves significant computing time and cost by allowing the process to continue from where it left off after an interruption. This is achieved by periodically persisting application state to SQLite databases and scanning previously generated output files.

- **Efficiency in Data Acquisition and Processing**: The system is optimized to minimize resource usage while maximizing data extraction speed.
  - *HTTP Client Optimization*: the HttpClient component utilizes requests.Session with HTTPAdapter and urllib3.util.retry.Retry for connection pooling, efficient session management, automatic retries on transient errors, and configurable timeouts. This enhances robustness and reduces connection overhead.
  - *Politeness and Anti-Detection Measures*: configuration parameters such as min_request_delay and max_request_delay CrawlerConfig, LinkCrawlerConfig) implement request throttling. User agent rotation from a diverse list (user_agents) helps avoid detection by anti-bot systems. max_redirects prevents infinite redirection loops.
  - *HTML Parsing*: both LinkExtractor and LinkExtractorForEdges leverage BeautifulSoup4 with the lxml parser for fast and efficient parsing of HTML content.
  - *Text Extraction*: the TextExtractor component (within the Embeddings Pipeline) employs trafilatura for robust and clean text extraction from raw HTML, essential for quality embeddings.
- **Data Processing and Normalization**: post-crawl data processing is streamlined to quickly parse and analyze content, generating the necessary embeddings and structured data for further analysis. This step involves rigorous cleaning and normalization.
  - *URL Normalization*: the LinkExtractor for content crawling incorporates *allowed_query_params* to retain only relevant URL parameters, ensuring canonical representation. The LinkExtractorForEdges for link graph extraction performs a more aggressive normalization, stripping all query parameters and fragment identifiers to focus purely on page paths.
  - *Feature Imputation*: in the GNN pipeline, the GraphDataProcessor manages missing embedding features by imputing them using neighboring node features, ensuring a complete dataset for model training.

- **Memory Management**: given the scale of data inherent in web crawling, efficient memory management is critical to prevent system resource exhaustion, particularly in shared cloud environments like Google Colab.
  - *Incremental Processing*: the primary content crawler (WebCrawler) utilizes an data_buffer that accumulates crawled pages. Data is saved incrementally to Parquet files at configurable intervals (save_interval_pages), and memory is explicitly managed using Python's garbage collection (gc.collect()) after each save.
  - *Data Streaming and Batching*: the *DataLoader* within the Embeddings Pipeline employs duckdb.fetch_record_batch to stream data from large Parquet datasets in manageable batches (batch_size), minimizing peak memory usage. Similarly, embeddings are generated and saved in batches.
- **Cost Optimization**: a key consideration for many businesses and SEO professionals is the cost of implementing such a system.
  - *Google Colab Pro*: offers a cost-effective solution by providing access to powerful GPUs and extended computing resources, reducing infrastructure overhead.
  - *Resource Management*: careful management of crawling parameters (request_timeout, max_pages_to_crawl) and batch sizes helps avoid exceeding usage limits and optimizes resource consumption.
  - *Resumable Workflows*: the implementation of resumable crawls and embedding generation workflows (via StateManager, LinkGraphStateManager, EmbeddingStateManager) prevents redundant processing and saves computing time and cost.
- **Data Storage and Persistence**: the choice of data formats and storage mechanisms is vital for both efficiency and accessibility.
  - *Parquet for Content and Embeddings*: crawled HTML content and generated URL embeddings are stored in Parquet files. Parquet is a columnar, compressed format highly optimized for analytical queries (e.g., via DuckDB) and efficient storage.
  - *CSV for Link Graphs*: link graph edges and analysis results (PageRank, HITS) are stored in CSV files, providing a simple, human-readable, and widely compatible format for graph analysis and visualization tools.
  - *SQLite for State Management*: the crawl frontiers (*crawl_frontier*) for both the content crawler and the link graph extractor are managed using SQLite databases (*.db* files). SQLite offers lightweight, file-based persistence for managing the state of ongoing operations, enabling seamless resumption.

  - *Model Artifacts*: trained GNN models and associated metadata are saved in specialized formats (*.pth*, *.pt*, *.json*) to ensure quick loading for inference.

- **Robustness and Error Handling**: the system is designed to operate reliably against common web and data processing challenges.
  - *Comprehensive Logging*: a generalized *ILogger* interface and *ConsoleAndGradioLogger* provide detailed logging to both the console and the Gradio UI, crucial for monitoring progress, identifying issues, and debugging.
  - *Graceful Error Management*: extensive *try-except* blocks are implemented across components (e.g., *HttpClient*, *StateManager*, *DataLoader*, *EmbeddingPipeline*, *PageRankService*) to handle exceptions like *FileNotFoundError*, *IOError*, *ValueError*, and network errors, ensuring the pipeline either recovers or fails gracefully with informative messages.
  - *Defensive Data Loading*: components like *PageRankService* include defensive reloads and data validation checks to ensure expected data columns and formats are present, preventing crashes from malformed or missing input files.

- **Modularity and Maintainability**: the entire project is structured into distinct, single-responsibility components and services.
  - *Separation of Concerns*: components are grouped into logical directories such as *config*, *data*, *graph*, *models*, *services*, and *utils*, enhancing clarity and testability.
  - *Reusability*: generic utilities like *HttpClient*, *strategies.py* (for crawling algorithms), and *url_processing.py* are designed to be reused across different modules.
  - *Dedicated Logic*: distinct functionalities, such as URL filtering for content vs. link graphs (*url.py* vs. *link_url.py*), are kept in separate files to avoid complex conditional logic and improve readability.

7.5 Limitations in JavaScript Handling

Many modern websites utilize JavaScript for dynamic content rendering. Traditional crawlers, which primarily parse static HTML, may fail to capture this dynamic content. To address this comprehensively, future enhancements would require integrating headless Browse tools such as Puppeteer or Selenium. This would enable the crawler to execute JavaScript and fully render pages as a user's browser would, ensuring all content, including dynamic elements, is extracted. Effective JavaScript handling also involves implementing smart wait strategies to manage page load times and avoid detection by anti-bot systems, thereby enhancing the accuracy of internal linking strategies. This feature is not currently implemented but is recognized as a key area for future development.

Sponsors: WordLift.io and Kalicube.com

8 Technical Stack and Setup

WebKnoGraph leverages a comprehensive and specialized technical stack to efficiently manage web crawling, data storage, and advanced machine learning tasks. This section outlines the tools and technologies employed, detailing their roles and how they are integrated into the overall workflow, focusing exclusively on what has been developed and utilized within the Python scripts.

8.1 Python on Google Colab Pro

Python is the core programming language for the entire WebKnoGraph project, chosen for its versatility, rich ecosystem of libraries, and suitability for data science and web development. All Python scripts are designed for execution on Google Colab Pro. Colab Pro is selected for its enhanced computational power, including access to GPUs and extended runtime environments, which are crucial for handling large-scale data processing tasks and training machine learning models without requiring local hardware investment.

Python is used to automate web crawling processes (via custom crawlers utilizing the *requests* and *BeautifulSoup4* libraries), perform complex data transformations, manage data persistence, and implement machine learning models. The modularization of the Python codebase into backend services, data components, and utility functions within the *src* directory (*src/backend/config*, *src/backend/data*, *src/backend/graph*, *src/backend/models*, *src/backend/services*, *src/backend/utils*, *src/shared*) facilitates organized development, testing, and execution within the Colab environment.

8.2 SQLite for Data Storage

SQLite serves as a lightweight, file-based database used for managing specific aspects of the crawl state. Its simplicity and ability to handle complex queries directly from disk make it an ideal choice for persistent storage in a cloud environment like Colab, where traditional database servers might be impractical. The *StateManager* (for the Content Crawler) and *LinkGraphStateManager* (for the Link Graph Extractor) components utilize SQLite databases (*.db* files) to store the *crawl_frontier* (the queue or stack of URLs to be visited). This setup is critical for enabling resumable crawls: if a Colab session disconnects or the processing is interrupted, the crawler can restart from its last known state, preventing redundant work and improving operational efficiency.

8.3 DuckDB for Data Analysis and Embeddings Management

DuckDB is employed as an in-process SQL OLAP (Online Analytical Processing) database, specifically for its high-performance analytical capabilities when working with large datasets, particularly those stored in Parquet format. It is integrated into the WebKnoGraph pipeline for several key roles:

- **Efficient Parquet Querying**: DuckDB allows for direct, high-speed SQL queries on Parquet files (e.g., in *crawled_data_parquet/* and *url_embeddings/*). This is critical for reading and filtering large volumes of semi-structured data without loading it entirely into memory.
- **Vector Embeddings Management**: it facilitates the efficient storage and retrieval of dense numerical vector embeddings. The Embeddings Pipeline, for instance, generates embeddings that are stored in Parquet, and DuckDB is used to read and process these for subsequent analysis or GNN model training.
- **Batch Processing**: the *DataLoader* component leverages *duckdb.fetch_record_batch* to stream data from large Parquet datasets in manageable batches, minimizing peak memory usage during intensive processing tasks.

8.4 Embeddings Creation

The embeddings capture the semantic content and structural properties of web pages, which are essential for tasks like link prediction and clustering. The project utilizes the *sentence-transformers* Python library for embedding generation, allowing for easy access and deployment of pre-trained models.

- **Model Selection**: by default, the project wil enable the *jinaai/jina-embeddings-v2-base-en* model from Hugging Face. This choice is based on its efficiency and ability to generate high-quality sentence embeddings for English texts. The model is well-suited for processing long-form content, as it generates semantically rich representations.
- **Computational Efficiency**: the selected JinaAI embedding model is open-source, computationally efficient, and proven effective in real-world scenarios, offering a cost-effective solution for large-scale text vectorization.

8.5 CSV Files for Intermediate Data Storage

Intermediate results from various processing and analysis steps are strategically stored in CSV (Comma Separated Values) files. This approach is chosen for its simplicity, wide compatibility, and ease of human readability and inspection. CSV files provide a portable format for storing and exchanging tabular data, making them ideal for intermediate stages in the pipeline before data is consumed by other modules or used for final reporting. Examples include *link_graph_edges.csv* (the output of the Link Graph Extractor) and *url_analysis_results.csv* (the output of the PageRank & HITS Analysis).

8.6 GraphSAGE for Link Prediction

For the central task of link prediction, the project utilizes GraphSAGE, a powerful graph neural network (GNN) framework. GraphSAGE is implemented using the *PyTorch Geometric* library, which provides an efficient and flexible framework for GNN development in Python. GraphSAGE is particularly effective in generating node embeddings by sampling and aggregating features from a node's local neighborhood. This mechanism is crucial for predicting missing links or suggesting new linking opportunities within the web graph, as it captures both the structural context and semantic content of web pages.

8.7 Computer Machine Details (CPU, GPU, Memory)

WebKnoGraph is designed to run efficiently within the resource constraints typical of cloud computing environments like Google Colab. While specific hardware details (CPU type, exact GPU model, total RAM) can vary in Colab's dynamic environment, the Python code is optimized to leverage available resources effectively. The project anticipates the need for sufficient Google Drive storage (e.g., at least 50 MB for 1,000 long-form articles) for persisting large datasets and model artifacts.

8.8 Integration and Workflow

The project's workflow is a multi-stage pipeline designed for seamless data flow and analysis, orchestrated by Python scripts executed in Colab notebooks:

- **Data Collection**: initiated by the Content Crawler and Link Graph Extractor, which acquire raw HTML and internal link structures.
- **Initial Processing**: data is processed through various Python modules for cleaning, normalization, and initial storage (e.g., SQLite for crawl state, Parquet for raw HTML).
- **Embedding Generation**: the Embeddings Pipeline transforms processed HTML into semantic vector embeddings using the *sentence-transformers* library and JinaAI models, with DuckDB aiding in data handling. **Graph Analysis**: the Link Graph Extractor's output is analyzed by the PageRank & HITS Analysis module to compute centrality measures and folder depths.
- **GNN Application**: The GNN Link Prediction & Recommendation Engine integrates the link graph and URL embeddings to train a GraphSAGE model, producing prediction artifacts.
- **Intermediate Storage**: CSV files are used extensively for intermediate results, ensuring data portability between stages.

- **User Interfaces**: each major module has a dedicated Gradio UI notebook (*crawler_ui.ipynb*, *embeddings_ui.ipynb*, *link_crawler_ui.ipynb*, *link_prediction_ui.ipynb*, *pagerank_ui.ipynb*) that orchestrates the execution of backend services and displays results interactively.

8.9 Data Collection and Processing Workflow Tips and Tricks

The project embraces best practices that are essential for building reliable and efficient data-intensive workflows, even if not always stated explicitly.

One fundamental concern is how to responsibly interact with target websites during crawling. To avoid overwhelming servers or triggering IP bans, care is taken to distribute the load across different domains when testing. This mitigates the risk of server strain or unintentional downtime, a consideration often overlooked in automated systems.

Patience is another recurring theme. Data collection and processing are inherently time-consuming phases, and rushing them can compromise the quality and completeness of results. It is advisable to allocate sufficient time for each step, rather than pushing for quick outputs. Regular monitoring ensures that processes are running as expected logging mechanisms and alerting systems play a vital role in catching issues early. Similarly, breaking tasks into smaller batches improves manageability and reduces the likelihood of processing failures or memory issues.

Another key design habit is the preservation of raw HTML files. Saving full HTML snapshots allows for more flexible reprocessing, retrospective debugging, and the ability to extract new data types that might become relevant later. These files serve as a durable fallback, especially when working with evolving data schemas or refining extraction techniques. Versioned storage and systematic file organization are encouraged to maintain data provenance.

Lastly, practical attention is given to infrastructure constraints. The project underscores the need for adequate cloud storage, particularly when working with large datasets over time. Estimating storage requirements in advance and setting up automated backup routines, especially within environments like Google Drive, are important safeguards to ensure data persistence and recovery in case of interruptions.

8.9.1 Subnet verification and IP whitelisting

Effective crawling depends on mutual understanding and coordination between website owners and those conducting the crawl. Two essential mechanisms that underpin this collaboration are subnet verification and IP whitelisting. Both contribute to the security, efficiency, and stability of large-scale data collection—particularly in contexts such as advanced SEO analysis and internal link optimization.

Subnet verification involves confirming that crawler traffic originates from authorized IP ranges. This step helps website owners distinguish legitimate crawlers from potentially harmful bots, reducing the risk of unauthorized access and scraping. From

the crawler's perspective, subnet verification establishes credibility and prevents avoidable disruptions, such as blocking or throttling.

Technically, the crawler provides a list of operational IP ranges, which the website owner validates against expected patterns or known affiliations. In some cases, this process is reinforced through digital certificates or authentication protocols. Verified subnets can then be granted differentiated access rights, improving the reliability of ongoing crawling activities.

Complementing subnet verification, IP whitelisting allows website owners to explicitly permit traffic from specific addresses. This control mechanism protects server resources and user data by filtering out unknown or suspicious sources. For authorized crawlers, being whitelisted ensures uninterrupted access, bypassing common security measures that might otherwise misclassify their requests.

Whitelisting reduces false positives in firewall or intrusion prevention systems, supporting more consistent and scalable data acquisition. It also reassures both parties that the crawling operation is recognized and accounted for within the site's access policies.

The success of both subnet verification and whitelisting hinges on transparent and ongoing communication. Crawlers should clearly disclose their objectives, scope, and infrastructure, while website owners should provide information about access policies, rate limits, and server constraints. Establishing dedicated contact channels for sharing updates—such as changes in IP ranges or temporary crawling restrictions—fosters trust and operational continuity.

By aligning technical configurations and expectations, both sides can protect infrastructure, minimize disruptions, and enable efficient, respectful crawling workflows.

The WebKnoGraph project currently employs sentence embeddings for capturing the semantic information of web pages. These embeddings are crucial for tasks such as information retrieval, clustering, sentence similarity, and ultimately for the Graph Neural Network (GNN) link prediction.

Currently, we utilize the *all-MiniLM-L6-v2* model from the *sentence-transformers* library. This model serves as an efficient encoder for sentences and short paragraphs, outputting a vector that encapsulates semantic information. By default, input text longer than 256 word pieces is truncated by this model.

9 GraphSAGE for Link Prediction

Link prediction in a graph involves predicting whether a link (edge) between two nodes should exist. Using GraphSAGE for link prediction is explained in the following subsections.

9.1 Node embedding generation

In this section, we outline a comprehensive approach for implementing link prediction using GraphSAGE (Graph Sample and Aggregated) solution for link prediction. GraphSAGE helps identify new linking opportunities within a graph network by leveraging both the structural and semantic information of nodes. Each node represents a web page or entity and is characterized by several features, such as a vector of its content, crawl depth, and response code. Here's an overview of the implementation:

- **Feature aggregation**: GraphSAGE learns node embeddings by aggregating the features of each node's neighbors. In this case, each node has features like content vectors (representing the textual or contextual information of the web page), crawl depth (indicating how far the node is from the starting point of the crawl), and response codes (showing the status of the web page). GraphSAGE incorporates these features and those of neighboring nodes to generate an embedding for each node that captures both local structure and node-specific features.

- **Node embeddings for similarity**: the embeddings generated by GraphSAGE represent nodes in a high-dimensional space where similar nodes (both in terms of graph structure and semantic content) are positioned closer to one another. This allows the model to identify nodes (web pages) that share similar characteristics, making them suitable candidates for linking. For example, a node with similar content and a crawl depth close to another could be a potential linking opportunity.

- **Link prediction**: once nodes are embedded, a link prediction algorithm uses these embeddings to determine the likelihood of a link (hyperlink or connection) between two nodes. The model assesses both the graph-based proximity (how close the nodes are in the network structure) and the semantic similarity (content similarity based on the vector representation) to predict new links. This enables the discovery of linking opportunities that make sense from a structural perspective (e.g., nodes that should be connected in a website hierarchy) and from a semantic viewpoint (e.g., content-relevant linking).

- **Finding relevant candidates**: by evaluating the graph's structure and the content of each node, GraphSAGE suggests linking opportunities that align with the website's or the graph's intent. For instance, a web page about "machine learning" may find linking opportunities with other pages that also cover similar content but are currently not connected. Additionally, the model considers other features like crawl depth and response codes to ensure the new links are practical and won't point to irrelevant or inaccessible pages.

10 The Final User Interface of the Solution

The final user interface (UI) is designed to simplify the discovery of new linking candidates based on a website's semantic and structural analysis. Users input a sample link, which the system analyzes for content relevance and position within the site's graph to suggest optimal linking opportunities.

Key UI elements include input fields for the sample link, adjustable parameters, and visual indicators for the analysis status. The results section will present proposed links intuitively for easy review and selection.

The design prioritizes a clean, user-friendly experience, helping users quickly establish meaningful internal links to improve navigation and SEO. This enables webmasters and SEO professionals to choose their preferred hierarchy while using our solution to easily find good linking candidates.

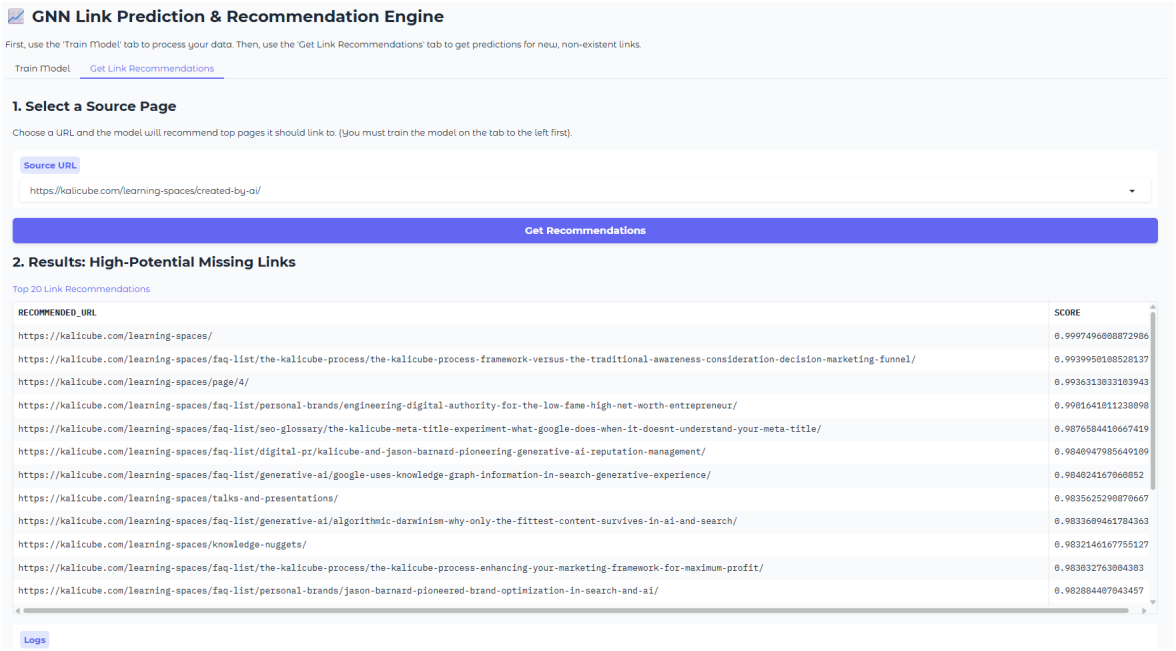

Figure 12: The link recommendation solution

11 Testing and evaluating results

At this stage, we are conducting rigorous testing together with WordLift to ensure the robustness of the solution. The testing phase involves implementing the solution in a real-world environment, but in a controlled manner, allowing us to carefully monitor performance, identify potential issues, and refine the approach. This process is crucial

to validating the effectiveness of the solution in diverse scenarios and ensuring its scalability and adaptability to various use cases.

By systematically experimenting with different configurations and settings, we aim to optimize both the technical and functional aspects of the solution. Feedback from real-world use cases will help us fine-tune the algorithms, improve user experience, and ensure that the solution meets the practical needs of its intended audience.

As this project is dynamic and evolving, we will share comprehensive results once it's fully complete. We anticipate that the insights gained during this testing phase will lead to further refinements, and we will publish detailed evaluations and findings shortly as part of our ongoing efforts to enhance the solution's impact.

In the meantime, we invite the technical marketing community to try it and share feedback. This is crucial to our product-driven, agile approach, as collaborative knowledge sharing is key to our team's success.

12 Final Words

Thank you for being with us so far, dear reader.
It's your turn to act now. This research has been quite a journey, and I sincerely hope it was worthwhile, providing you with new insights and valuable perspectives. Conducting this study was no easy task, but with the support of an innovative, research-driven team, *anything is possible*. Your engagement with this work means the world to us, and I hope it has sparked ideas, questions, or perhaps even a new path forward for your own endeavors.

As we close this chapter, I encourage you to take action—whether that's applying the findings in your own projects, pushing the boundaries of the research presented, or simply reflecting on what you've learned. Progress, after all, comes from curiosity and the courage to explore uncharted territory.

This research is part of a larger, ongoing conversation, and I would love for you to stay connected as we continue to innovate and grow. Your thoughts, feedback, and collaboration are always welcome, as they help shape the future of this dynamic field. Let's continue this journey together—because, in the end, it's the exchange of ideas and shared passion that makes real impact. Thank you once again for your time, your curiosity, and your commitment to learning.

Sponsors: WordLift.io and Kalicube.com

References

[1] P. Suzart, "Nailing internal linking with graph theory — by paulo suzart — omio engineering — medium." https://medium.com/omio-engineering/nailing-seo-internal-linking-with-graph-theory-2c45544a024d, August 2020. (Accessed on 09/04/2024).

[2] K. Indig, "The best internal linking structure depends on your business model." https://www.growth-memo.com/p/the-best-internal-linking-structure-depends-on-your-business-model, February 2019. (Accessed on 09/04/2024).

[3] E. Gjorgjevska and V. Izzo, "Dynamic internal links in seo: your superhero in the generative ai era." https://wordlift.io/blog/en/dynamic-internal-links-in-seo/, July 2024. (Accessed on 09/04/2024).

[4] M. Ciffone, "Linkedin post." https://www.linkedin.com/posts/mike-ciffone_common-website-architectures-and-powerful-ugcPost-7235710529627963392-FCSn/, September 2024. (Accessed on 09/05/2024).

[5] E. Gjorgjevska, "Mastering topic clusters in seo - wordlift blog." https://wordlift.io/blog/en/master-topical-seo/, January 2024. (Accessed on 09/05/2024).

Sponsors: WordLift.io and Kalicube.com