A Privacy-Preserving Framework for Keyword-Based Document Retrieval (Code)

Kerem Bayramoğlu
kerem.bayramoglu@bilkent.edu.tr
Bilkent University
Turkey

Ergün Batuhan Kaynak
batuhan.kaynak@bilkent.edu.tr
Bilkent University
Turkey

Melih Coşğun
melih.cosgun@bilkent.edu.tr
Bilkent University
Turkey

Kousar Kousar
kousar.kousar@bilkent.edu.tr
Bilkent University
Turkey

Ömer Kaan Gürbüz
kaan.gurbuz@bilkent.edu.tr
Bilkent University
Turkey

Abstract

Concerns about privacy have grown with the digitization of everyday life, especially as Internet services collect large amounts of personal data. If these records are exploited, search engines, which store user queries for individualized information retrieval, pose a serious privacy threat. Hiding search intent via query obfuscation has become a crucial approach in Private Web Search to protect user privacy. The lack of semantic meaning in the random searches produced by early obfuscation techniques reduced both privacy and retrieval value. Although recent developments such as proxy-based and scrambling techniques have improved privacy, they are still computationally expensive or vulnerable to data leakage.

This study presents a novel query obfuscation technique that utilizes Large Language Models(LLMs). The proposed method ensures robustness to adversarial attacks. To further enhance privacy, the method considers the user's consecutive queries. This work aims to provide a utility and privacy-preserving query obfuscation schema that is robust to adversarial attacks by addressing the shortcomings of earlier methods.

CCS Concepts

• **Security and privacy** → **Privacy-preserving protocols**.

Keywords

Private Information Retrieval, Query Obfuscation

1 General Architecture

This document outlines a modular architecture designed for an information retrieval system integrating modern technologies. The system includes a **frontend** hosted on Netlify and developed using Next.js, a **backend** implemented with Flask and deployed on Amazon EC2, a **machine learning inference layer** leveraging Hugging Face's BERT model and tokenizer, and an **information retrieval system** powered by SERPAPI for Google Search queries. Each component collaborates to provide a seamless user experience for retrieving relevant and contextual data efficiently. The overall system architecture can be seen in 1.

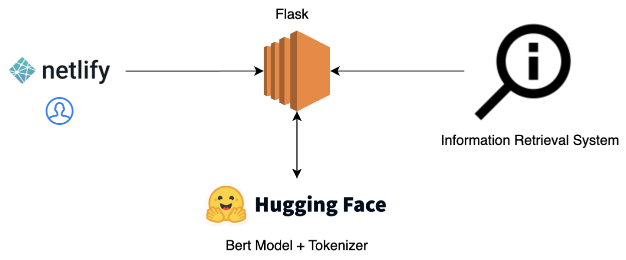

Figure 1: System Architecture

2 Frontend

The frontend of the application is built using **Next.js** and hosted on **Netlify**, ensuring a robust and efficient user interface. It consists of two key pages: the home page and the results page.

The home page features an interactive search interface, where users can input their search queries into the search bar. The home page is responsive and designed to guide users seamlessly to the results page upon query submission.

The results page dynamically fetches search results based on user input, using the API endpoint hosted at `cs533.hm-rdp.com`. It includes:

- A search bar at the top for refining or initiating new searches.
- Tabs for different query categories generated by the backend, allowing users to switch between dummy search contexts as well as the original search.
- A detailed list of search results for each category, with clickable links that open in new tabs for user convenience.

3 Backend

The backend is implemented using **Flask**, hosted on an **Amazon EC2** instance, and serves as the central processing unit for the system. It provides two key API endpoints for query generation tasks, leveraging the `DummyQueryGenerator` class for functionality.

Both endpoints accept JSON-formatted data and return responses in the same format.

3.1 API Endpoints

The backend exposes the following API endpoints:

- **generate-dummy-queries**
  - **Method:** POST
  - **Input:** JSON payload with:
    - `input_query` (required): The primary search query.
    - `num_queries` (optional): Number of dummy queries to generate (default: 2).
  - **Processing:** Uses the `generate_dummy_queries()` method to:
    - Generate dummy queries related to the `input_query`.
    - Determine the `input_category`.
    - Retrieve search results for the dummy queries.
  - **Output:** JSON object with:
    - `all_queries`: List of all generated queries.
    - `result`: Search results for the queries.
    - `input_category`: Category of the input query.
- **generate-consecutive-queries**
  - **Method:** POST
  - **Input:** JSON payload with:
    - `input_query` (required): The primary search query.
    - `dummy_queries` (required): List of dummy queries related to the `input_query`.
    - `input_category` (required): Category of the input query.
  - **Processing:** Uses the `generate_consecutive_queries()` method to:
    - Process the input query, dummy queries, and their category.
    - Generate consecutive queries and retrieve search results.
  - **Output:** JSON object with:
    - `all_queries`: List of all consecutive queries.
    - `result`: Search results for the queries.

3.2 DummyQueryGenerator Class

The `DummyQueryGenerator` class is the central component of the project, designed to handle the generation of dummy queries, the identification of query categories, and the retrieval of search results. Upon initialization, the class loads a dataset of Google Trends data from a CSV file, which serves as the basis for its query categorization capabilities. Each category in the dataset includes a precomputed embedding, which is processed into a numerical format to facilitate similarity computations.

One of the key functionalities of this class is its ability to generate embeddings for a given query. This is achieved by making a POST request to the Hugging Face API endpoint, which processes the query and returns its embedding. These embeddings are used in conjunction with the precomputed category embeddings to identify the most relevant category for a given query. The class uses cosine similarity to calculate the closeness between the query and category embeddings, ensuring accurate categorizations.

To enhance diversity in query generation, the DummyQueryGenerator class employs a clustering technique using K-means. By partitioning the embedding space into distinct clusters, it can select categories from diverse regions of the space. This ensures that the generated queries are varied and not overly concentrated around a single topic. The class also analyzes the input query's stylistic features, such as length, capitalization, and the presence of question marks, to ensure that generated queries match the input query's style.

The class provides two main methods for query generation. The first, `generate_dummy_queries`, creates queries that are unrelated to the input query but conform to its stylistic attributes. The second, `generate_consecutive_queries`, generates follow-up queries for a given list of dummy queries, maintaining consistency with the style of the original query. Both methods rely on external APIs, including SerpAPI for retrieving search results for the generated queries.

Overall, the `DummyQueryGenerator` class is a robust and flexible utility that underpins the backend's query generation functionality, enabling it to deliver diverse and high-quality results tailored to the needs of the system.

3.3 ML-Inference Server

ML inference server is normally a part of the backend, however, for the sake of cost reduction, ML model is not ran in an EC2 instance. Instead, the BERT model and BERT tokenizer are moved to the free inference server.

4 Information Retrieval

An **information retrieval** layer is integrated using **SERPAPI** to fetch real-time results from the **Google Search engine**. once the dummy queries are generated, said queries are then inputted to google search engine for actual results, enhancing the system's practicality.

5 Privacy Attacks using `attack_with_random` and `attack_with_gpt`

The `attack_with_random` and `attack_with_gpt` functions are designed to perform attacks on outputs generated via our methodology. The functions load a CSV file containing consecutive queries and shuffle the rows to create query flows with random order (consecutive queries keep their sequence). The `attack_with_random` function shuffles the query rows and selects the top 10 based on a random ranking. It returns a JSON object of ordered ranks and their indices. The `attack_with_gpt` function formats the shuffled query dataset into a prompt for GPT, which is then used to identify the most probable human-generated query. The attack can be enhanced with a "whitebox" prompt, which includes predefined categories to assist the model in distinguishing between human and AI-generated queries. It again returns a JSON object of ordered ranks and their indices.

The `evaluate_results` is a helper function that compares the predicted rankings from the attacks with the original query index. It outputs the rank at which the original query was identified. The `attack_and_eval` function uses this helper function and computes average prediction rank, and tracks the hit-rate. It can handle multiple attack types, either random or GPT-based, and works across different datasets to evaluate the robustness of query identification strategies under adversarial conditions.

Overall, the `attack_with_random` and `attack_with_gpt` functions offer a comprehensive framework for testing the resilience of our query systems to adversarial attacks, helping to assess the impact of query shuffling and AI-based analysis on detecting the original query.

6 Conclusion

In conclusion, the proposed query obfuscation method, supported by advanced LLM-based techniques and adversarial attack evaluations, offers a robust solution to privacy challenges in Private Web Search. It strikes a balance between protecting user privacy and maintaining the utility of search results, providing a more effective and resilient approach to query obfuscation compared to earlier methods.

References