Assignment 2: Deduplication and Crawling

Prepared by
Kushagra Gupta
(2025909001)

Under the guidance of
Prof. Anil Nelakanti

Submitted in
partial fulfillment of the requirements
for the course of
CS4.406: Information Retrieval and Extraction

INTERNATIONAL INSTITUTE OF INFORMATION TECHNOLOGY
HYDERABAD

(November 2025)

Activity 2.1: Deduplication

Goal: To find and group all records belonging to the same person from a noisy dataset of 5,000 entries.

My Approach: Followed a data-cleaning and linkage pipeline:

1. **Exploratory Data Analysis (EDA):** I first inspected the data and found many problems, including missing values, incorrect data types (like date_of_birth as a number), and typos in text fields (like 'nsw' vs. 'nws').
2. **Data Preprocessing:** I cleaned the entire dataset. I fixed all typos, converted dates and postcodes to standard string formats, and filled in missing values.
3. **Blocking:** To avoid comparing 12.5 million pairs, I used a "blocking" strategy. I grouped records by postcode, assuming matches would be in the same postal area.
4. **Pairwise Comparison:** Within these blocks, I used the recordlinkage library to compare pairs. I scored similarity on names, addresses, and date of birth using the Jaro-Winkler method.
5. **Decision & Clustering:** I set a rule that any pair with a total similarity score of 4.0 (out of 5.0) was a "match." I then used the networkx library to find all connected groups, or "clusters."

Key Libraries Used:

- pandas (for data loading and cleaning)
- recordlinkage & jellyfish (for blocking and comparing)
- networkx (for clustering the matched pairs)
- sklearn (for evaluation)

Results & Insights:

- **Insight:** The soc_sec_id column was the "answer key." The 5,000 records only had 2,291 unique soc_sec_ids. I used this to check my work.
- **Result 1 (Blocking**): My blocking strategy was very effective, reducing the number of comparisons by 99.87% (from 12.5 million to just 16,115).
- **Result 2 (Accuracy):** My algorithm found 1,059 clusters. When I compared this to the 2,291 "true" clusters, I got an Adjusted Rand Score (ARS) of 0.7307. This is a great result, showing my model's groups were very similar to the real ones. The lower cluster count means my model tended to "over-cluster," sometimes grouping two different people who just had very similar information.

Activity 2.2: Crawling

Goal: To crawl a local web server, calculate the PageRank of its pages, and create a smart strategy to check for updates efficiently.

My Approach:

1. **Discovery & Parsing:** I first discovered that the server did not have a JSON API. Instead, it only sent HTML. I used the BeautifulSoup library to parse the HTML and extract the page_id, node_id, and all outgoing links from the page.
2. **Crawling:** I wrote a simple crawler that used a queue (a Breadth-First Search) to visit every page on the site, starting from the root. I kept a "visited" set to avoid loops. This process discovered 14 unique pages.
3. **PageRank Calculation:** I used the networkx library to build a directed graph of the 14 pages and their links. I then ran the PageRank algorithm on this graph to calculate an "importance" score for every page.
4. **Efficient Update Strategy:** To minimize visits, I analyzed the history of node ID changes for each page. I calculated the average time (in seconds) between changes. I combined this "change frequency" score with the "importance" (PageRank) score to create a final revisit_priority score for each page.

Key Libraries Used:

- requests (to fetch the web pages)
- beautifulsoup4 (to parse the HTML)
- networkx (to build the graph and run PageRank)
- pandas (to help with timestamp calculations)

Results & Insights:

- **Insight:** The key was realizing I had to be a scraper, not just an API consumer. The real data was hidden in the HTML.
- **Result:** I successfully crawled the entire site and calculated the PageRange for all 14 pages.
- **Final Solution:** The final result was a prioritized table of all pages. Pages that were both important (high PageRank) and changed often (low average time between updates) received the highest priority. Pages that were unimportant or rarely changed were at the bottom. This list provides the exact, efficient strategy needed to keep the node IDs updated while minimizing server visits.