Github Repository: https://github.com/lucassandreu/irwa-search-engine
Tag Delivery 3: https://github.com/lucassandreu/irwa-search-engine/releases/tag/IRWA-2025-part-4

Universitat Pompeu Fabra Barcelona

IRWA Project Part 4: RAG, User Interface, and Web Analytics

Authors: Lucas Andreu, Pau Chaves, Pol Bonet, Joan Company

Course: Information Retrieval and Web Analytics (IRWA)

Date: November 29, 2025

1. Introduction

In Part 4 we turn the search engine from Parts 1–3 into a full web application with:

- A Flask-based UI for querying the product corpus and exploring individual products.
- A Retrieval-Augmented Generation (RAG) component that summarizes and recommends products using an external LLM (Groq API).
- A web-analytics layer that tracks how users interact with the system (requests, queries, clicks, dwell time, sessions, and user context) and visualizes this data in a dashboard.

All analytics are stored in memory (Python objects) for easy reproduction by the instructors.

2. User Interface & Search Integration

2.1 Search Page

The main search page is implemented in templates/index.html and served by the / route in web_app.py. It contains:

- A central search box (<input name="search-query">).
- A Search button submitting the form via POST to /search.

```HTML
<form class="d-flex" method="POST" onSubmit='return validate();'
action="/search">
    <input class="form-control me-2" name="search-query" type="search"
placeholder="Search" autofocus="autofocus">
    <button class="btn btn-primary" type="submit">Search</button>
</form>
```

The layout uses Bootstrap and a shared *base.html* template for header/footer.

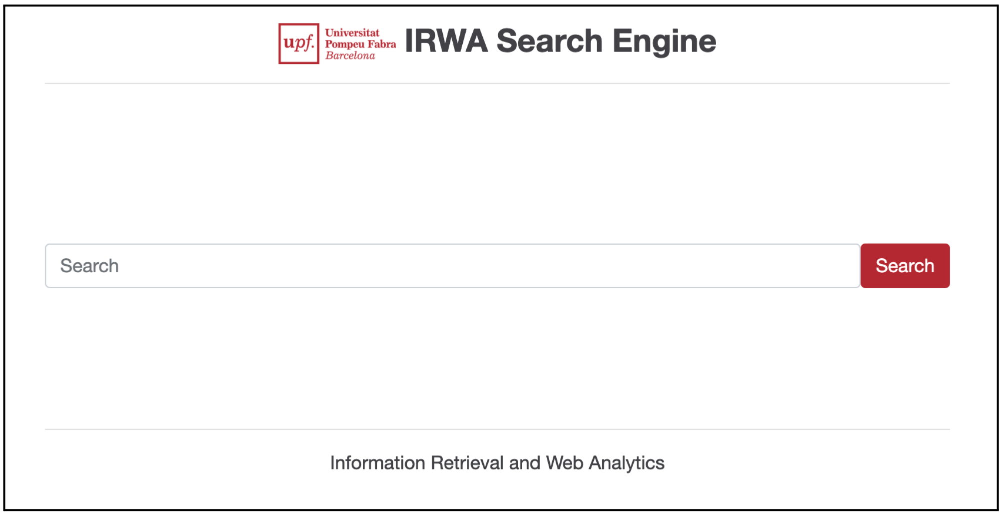

Figure 1 – Home / search page, before submitting any query

2.2 Search Action and Engine Integration

The /search route in web_app.py:

```python
@app.route('/search', methods=['POST'])
def search_form_post():
    search_query = request.form['search-query']
    ...
    search_id = analytics_data.save_query_terms(
        terms=search_query,
        ip=request.remote_addr,
        user_agent=request.headers.get("User-Agent", ""),
        browser=request.user_agent.browser,
        session_id=session.get("session_id")
    )
    ...
    results = search_engine.search(search_query, search_id, corpus)
```

Key points:

- The raw query string is passed to SearchEngine.search(search_query, search_id, corpus).
- A search_id is generated and stored; this links queries to later clicks (for analytics).
- Query metadata (IP, browser, session, terms, etc.) is registered via AnalyticsData.save_query_terms.

The **general search function** lives in myapp/search/search_engine.py:

```python
class SearchEngine:
    def search(self, search_query, search_id, corpus):
        results = search_in_corpus(
            query=search_query,
            search_id=search_id,
            corpus=corpus,
            method="bm25",
            k=20,
            use_and=True
        )
        return results
```

This function only needs a string query (plus search_id for logging) and can be easily reused from any UI.

2.3 Search Algorithms

The core retrieval logic is in myapp/search/algorithms.py:

- **Preprocessing and indexing**
  - We load an enriched Flipkart fashion dataset and a boolean inverted index (fashion_products_dataset_enriched.json, boolean_inverted_index.json).
  - Index fields: title_clean, description_clean, metadata_clean.

- **Candidate generation**
  - AND semantics by default: _candidate_docs_and(q_terms) intersects postings lists for all query terms.
  - Fallback to OR semantics if AND yields no candidates.

- **Ranking algorithms implemented**
  - **BM25** (default):
    - Standard Okapi BM25 with document length normalization and pre-computed idf_bm25.
  - **TF-IDF cosine similarity**:
    - tfidf_weights and doc_norms pre-computed for fast cosine scoring.
  - **Custom TF-IDF + business boost**:
    - _numeric_boost favors in-stock products with better ratings, higher discounts and reasonable prices.
    - scores = tfidf_score * _numeric_boost(record).

- **Result representation**
  - search_in_corpus(...) converts each hit into a ResultItem object (Pydantic model) with:

- title, description
- selling_price, actual_price, discount, average_rating, out_of_stock
- url → internal /doc_details?pid=...&search_id=...
- source_url → original Flipkart URL
- ranking → BM25 (or other) score, used later in analytics.

This structure is suitable for a web application: the search engine is pure Python (no Flask dependency) and returns serializable objects.

2.4 Results Page

The /search route renders templates/results.html with:

- The **number of results** found.
- An **AI-Generated Summary** (RAG, described in Section 3).
- A list of ranked results using the metadata specified in the assignment:

```html
<div class="doc-title">
  <a href="{{ item.url }}">{{ item.title }}</a>
</div>
<div class="doc-desc text-muted">{{ item.description }}</div>

<div class="mt-1">
  <span><strong>Price:</strong> ₹{{ item.selling_price }}</span>
  <span class="ml-2 text-muted"><del>₹{{ item.actual_price }}</del></span>
  <span class="ml-2 text-success"><strong>{{ item.discount }}%
OFF</strong></span>
</div>

<div class="mt-1">
  <span><strong>Rating:</strong> {{ item.average_rating }}/5 ⭐</span>
  {% if item.out_of_stock %}
    <span class="ml-2 text-danger"><strong>Out of stock</strong></span>
  {% else %}
    <span class="ml-2 text-success"><strong>In stock</strong></span>
  {% endif %}
</div>

<a href="{{ item.source_url }}" target="_blank" class="small">
  Open in original store ↗
</a>
```

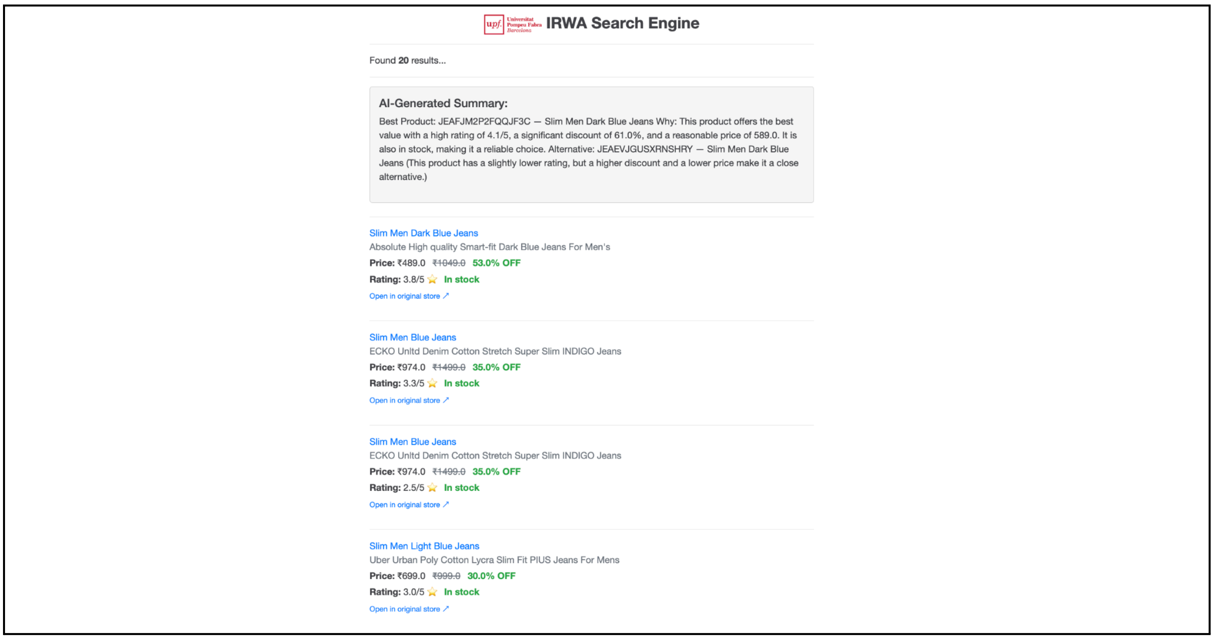

Figure 2 – Results Page Showing Ranked Product List (men slim blue jeans)

2.5 Document Details Page

The /doc_details route:

- Receives pid and search_id.
- Looks up the Document from the loaded corpus.
- Registers a click event (with ranking, query, and user context) in AnalyticsData.register_click.
- Stores last_click_time in the session so that the next search can compute dwell time.

templates/doc_details.html shows:

- Full title and description.
- Price, discount and rating.
- Brand, category & sub-category, seller.
- Stock status.
- A formatted list of product_details (technical specs).
- Product images if available.
- A link to the original Flipkart product.

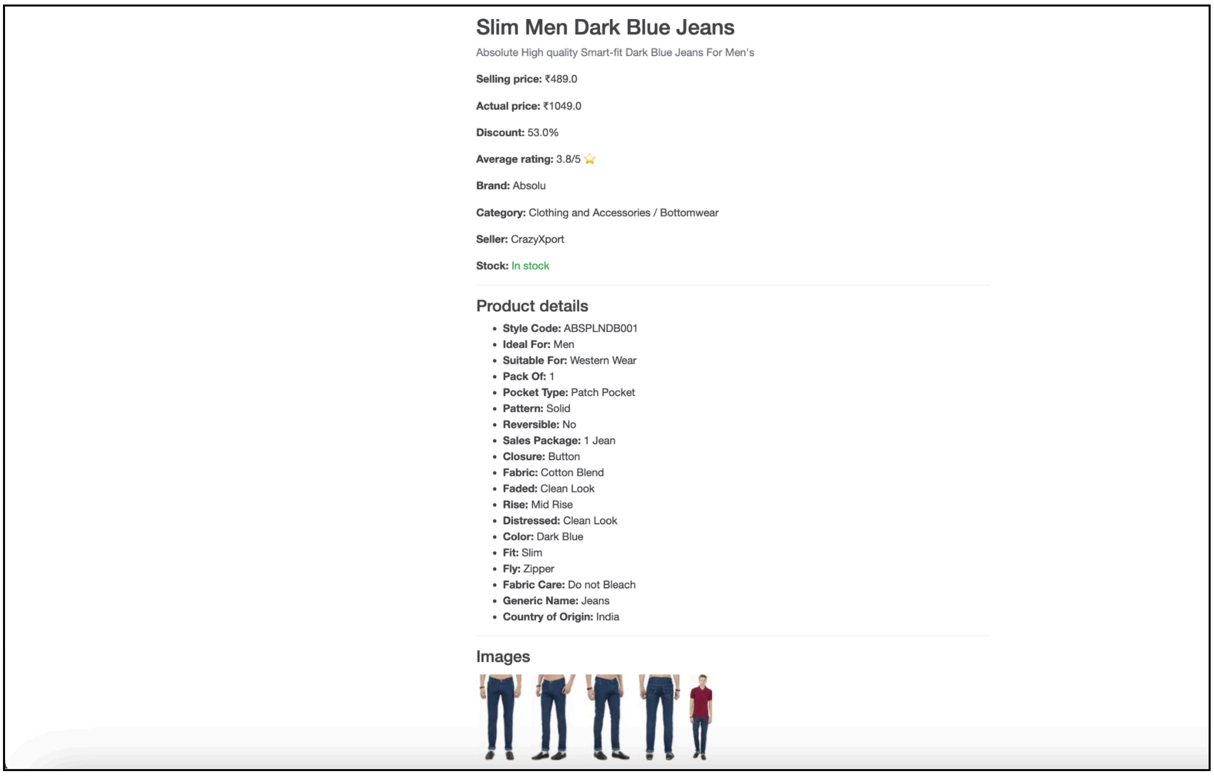

Figure 3 – All info details page (men slim blue jeans)

3. RAG: Retrieval-Augmented Generation

3.1 Baseline RAG and limitations

The instructor's baseline RAG pipeline:

- Passed only a list of product PIDs and titles to the Groq LLM.
- Used a generic prompt without emphasis on stock status or value.
- Let the model freely choose any product, which sometimes:
    - Ignored stock, discount, or rating.
    - Fabricated attributes.
    - Returned ambiguous recommendations.
    - Always chose a product, even if nothing matched the query.

3.2 Implemented improvements

The improved RAG logic is implemented in myapp/generation/rag.py (RAGGenerator).

1) Richer metadata + helper score

We pass a structured list of retrieved products to the LLM using _format_results:

```python
f"{i}. PID: {pid}\n"
f"   Title: {title}\n"
f"   Brand: {brand}\n"
f"   Category: {category} / {subcat}\n"
f"   Price: {price} | Actual: {actual_price} | Discount: {discount}%\n"
f"   Rating: {rating}/5 | In stock: {not bool(stock)}\n"
f"   retrieval_score: {retrieval_score} | helper_score: {helper}\n"
```

The helper_score is a deterministic value score:

```python
rating_norm   # normalized 0–1
discount_norm # normalized discount
price_norm    # cheaper is better
```

These fields guide the LLM toward products that are both **relevant and good value**, without hard-coding a ranking inside the model.

Effect in practice
For a query like men slim blue jeans, the summary typically selects a **high-discount, good-rating, in-stock** pair of jeans and explicitly references price, discount and rating in the explanation.

2) Pre-filter obviously bad candidates

Before calling the LLM we filter out out-of-stock products if there are enough remaining hits:

```python
in_stock = [r for r in retrieved_results if not getattr(r, "out_of_stock",
False)]
if len(in_stock) >= 3:
    retrieved_results = in_stock
```

This prevents recommending unavailable products and simplifies the LLM's job by focusing on viable options.

Effect in practice
For queries with both in-stock and out-of-stock items (e.g., women solid cotton kurta beige), the LLM now recommends an in-stock product with good discount and rating instead of a cheaper but unavailable alternative.

3) Safer prompting, low temperature, and “no good products” handling

We add a restrictive system prompt:

```python
SYSTEM_PROMPT = (
    "You are a careful, non-hallucinating product recommender. "
    "You must only use the provided retrieved products. "
    "If the retrieved products do not fit, say so explicitly."
)
```

The user prompt instructs the model to:

- Use the metadata to justify the recommendation.
- Prefer in-stock items.
- Output a fixed format:
    - Best Product: <PID> — <Title>
    - Why: ...
    - Alternative (optional): ...
- If nothing fits, output exactly:
  "There are no good products that fit the request based on the retrieved results."

We also reduce creativity with:

```python
temperature=0.1,
max_tokens=300
```

and add a simple fallback if the model returns an empty string.

Effect in practice

- For a realistic query like winter boots waterproof women, the answer is:
  “Best Product: SHOF97KQWHH7FDTG — Boots For Women (Beige)
  Why: This product is the best match for the user's request as it is a pair of boots for women, which aligns with the query intent. It is also waterproof …”
- For intentionally mismatched queries such as neon green tuxedo or kids shoes, the system now correctly returns:
  “There are no good products that fit the request based on the retrieved results.”

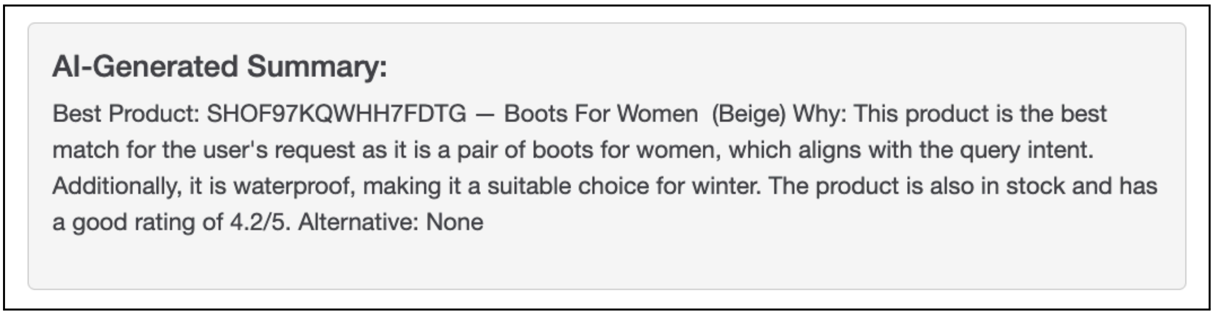

Figure 4 – RAG Summary for a Well-Matched Query ("winter boots waterproof women")

Figure 5 – RAG Output for a Query with No Suitable Products ("neon green tuxedo")

4. Web Analytics

Analytics are implemented in myapp/analytics/analytics_data.py and integrated via web_app.py.

All data is stored in memory as Python lists/dicts so instructors can reproduce results simply by running the app and interacting with it.

4.1 Data collection

We collect four main event types.

4.1.1 HTTP requests

@app.before_request in web_app.py logs every request:

```python
analytics_data.register_request(
    path=request.path,
    method=request.method,
    user_agent=request.headers.get("User-Agent", ""),
    ip=request.remote_addr,
    session_id=session.get("session_id"),
    ts=time.time()
)
```

Stored fields (table requests):

- ts – timestamp.

- path – URL path ("/", "/search", "/doc_details", etc.).
- method – HTTP method (GET/POST).
- user_agent – full user agent string.
- ip – remote IP address.
- session_id – synthetic ID stored in the Flask session cookie (used to group user journeys).

4.1.2 Queries

When /search is called, we generate a search_id and save the query via:

```python
search_id = analytics_data.save_query_terms(
    terms=search_query,
    ip=request.remote_addr,
    user_agent=request.headers.get("User-Agent", ""),
    browser=request.user_agent.browser,
    session_id=session.get("session_id")
)
```

Under the hood, register_query records:

- ts, search_id, query
- n_terms and tokenized terms
- User context:
    - ip
    - user_agent
    - browser (parsed from Flask's request.user_agent.browser)
    - session_id

This satisfies the requirement to collect *number of terms*, *order*, and *user context* (IP, browser, time of day via ts).

4.1.3 Clicks and ranking

In /doc_details we link each click to the query that produced it:

```python
analytics_data.register_click(
    pid=pid,
    search_id=int(search_id) if search_id else -1,
    rank=getattr(doc_obj, "ranking", None),
    query=session.get("last_search_query"),
    ip=request.remote_addr,
    user_agent=request.headers.get("User-Agent", ""),
    session_id=session.get("session_id")
```

```
)
```

Stored fields (clicks table):

- ts, pid, search_id
- rank – BM25/TF-IDF score of the clicked document.
- query – last search query from the same session.
- ip, user_agent, session_id

We also maintain fact_clicks[pid] for quick "top clicked products" statistics.

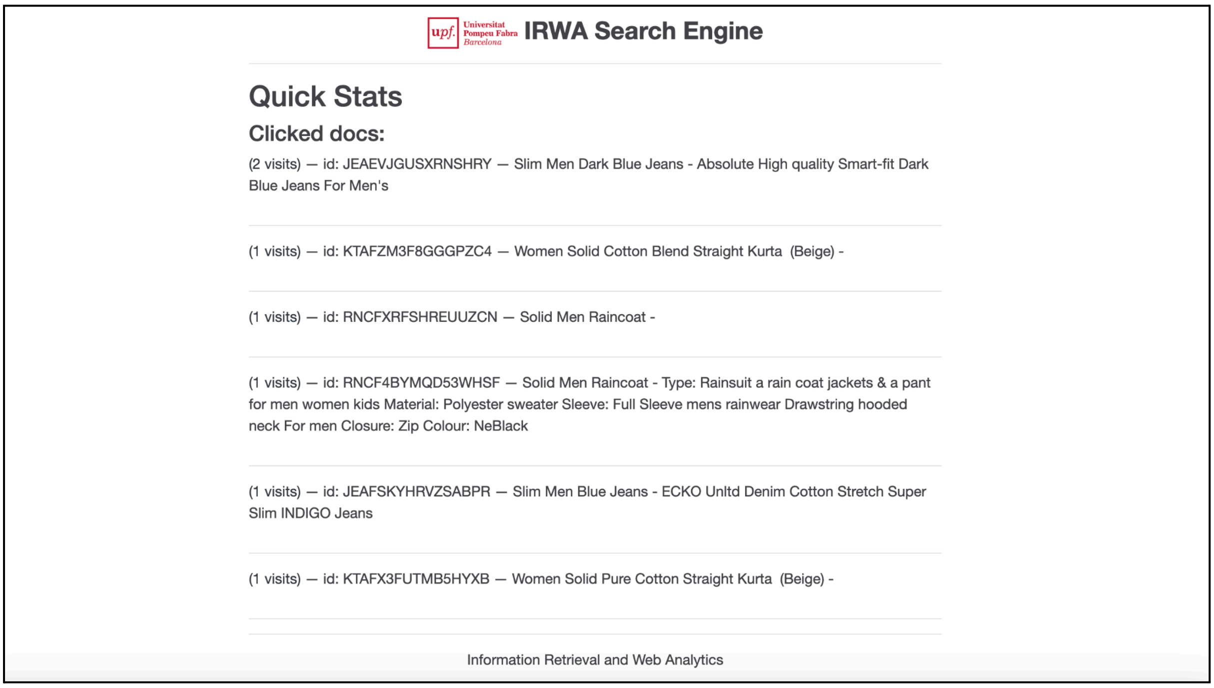

Figure 5 – Top Clicked Products Page (Stats)

4.1.4 Dwell time

We measure *dwell time* as the time between clicking a result and issuing the next search:

- On click (/doc_details), we save:

```python
session["last_click_time"] = time.time()
session["last_clicked_pid"] = pid
session["last_search_id"] = int(search_id) if search_id else -1
```

- On the next /search, before processing the new query:

```python
dwell = time.time() - session["last_click_time"]
analytics_data.register_dwell(
    pid=session["last_clicked_pid"],
    search_id=session.get("last_search_id", -1),
    dwell_seconds=dwell
)
```

Stored fields (dwell_times table):

- ts, pid, search_id, dwell_seconds.

This completes query→click→dwell linkage for later funnel analysis.

4.2 Data model (in-memory star schema)

Conceptually we follow a small **star schema**:

- **Fact tables**
  - queries: one row per issued query (with search_id).
  - clicks: one row per click on a result.
  - dwell_times: one row per interaction with dwell measurement.
  - requests: one row per HTTP request to any endpoint.
- **Dimensions / keys**
  - search_id links queries, clicks and dwell records.
  - pid links clicks/dwell to the **Document** (product) in the corpus.
  - session_id groups multiple queries and clicks into a physical session (user journey).
  - ts (timestamp) enables temporal analysis such as time-of-day.

This schema is implemented purely in Python lists/dicts but could be easily translated to relational tables if a database were used.

4.3 Key metrics and reports

4.3.1 Summary statistics

AnalyticsData.summary_stats() computes:

- total_searches – number of queries.
- total_clicks – number of clicks.
- ctr – click-through-rate = clicks / searches.
- unique_queries – distinct query strings.
- unique_terms – distinct terms across all queries.
- avg_dwell – average dwell time in seconds.

These are shown as KPI cards at the top of dashboard.html.

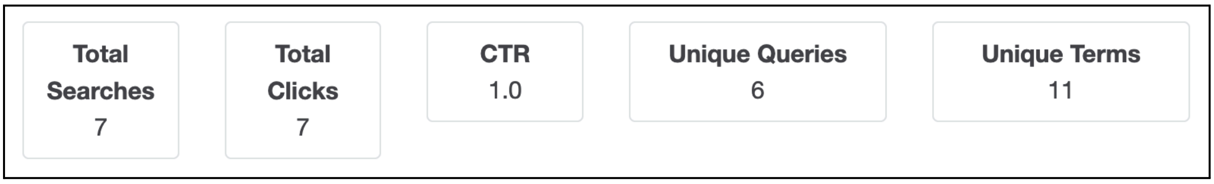

Figure 6 – Dashboard High-Level KPIs

4.3.2 Funnel metrics (search → click → dwell)

AnalyticsData.funnel_metrics() further computes:

- searches, clicks
- dwell_over_5s – number of interactions where dwell_seconds ≥ 5.
- ctr – as above.
- engagement_rate – fraction of searches that led to a click with dwell ≥ 5s.

These KPIs highlight how many searches turn into engaged product views (not just accidental clicks).

Figure 7 – Dashboard KPIs (further)

4.3.3 Top clicked products

plot_number_of_views():

- Uses fact_clicks to plot a bar chart "Number of Views per Document".
- Embedded via an <iframe> /plot_number_of_views in the dashboard.

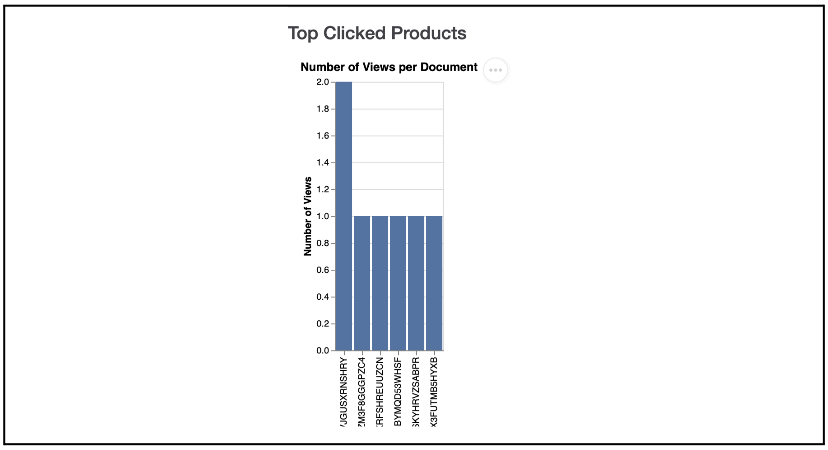

Figure 8 – Bar Chart Number of Views

4.3.4 Query-level insights

- plot_top_queries() – bar chart of most frequent full queries.
- plot_top_terms() – bar chart of most frequent individual terms.

Both are embedded as iframes in dashboard.html.

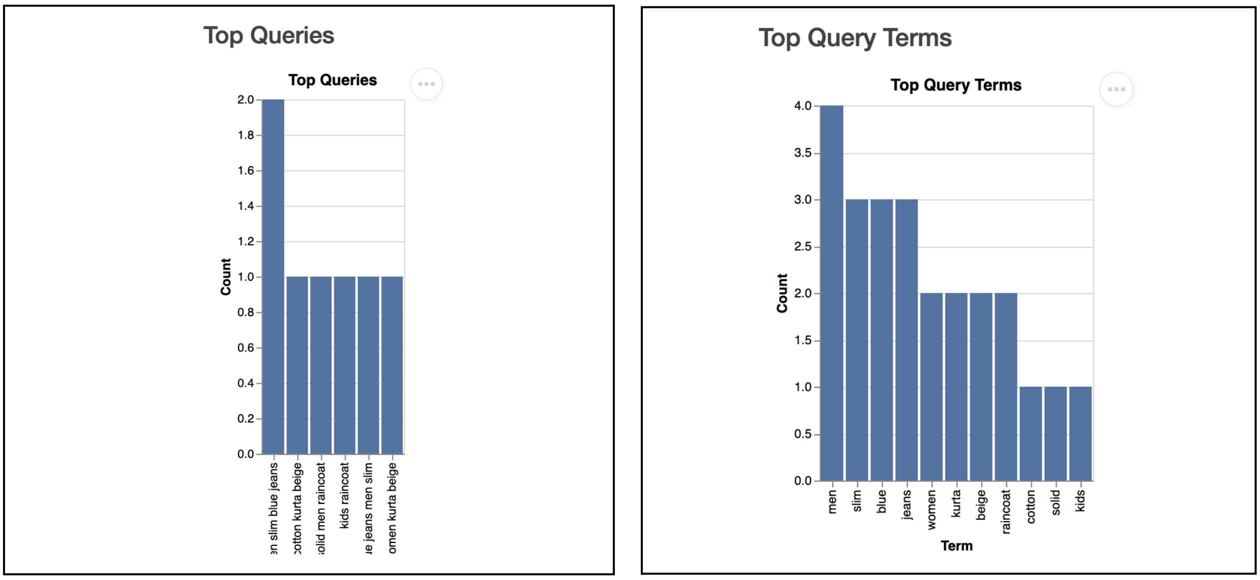

Figure 9 – Bar Chart Top Queries and Top Query Terms

4.3.5 Temporal analysis: searches per hour

plot_searches_per_hour():

- Converts query timestamps ts to hour-of-day ("00"–"23").
- Groups by hour and plots a line chart "Searches per Hour of Day".

This would show, for a real deployment, when the search engine is most used.

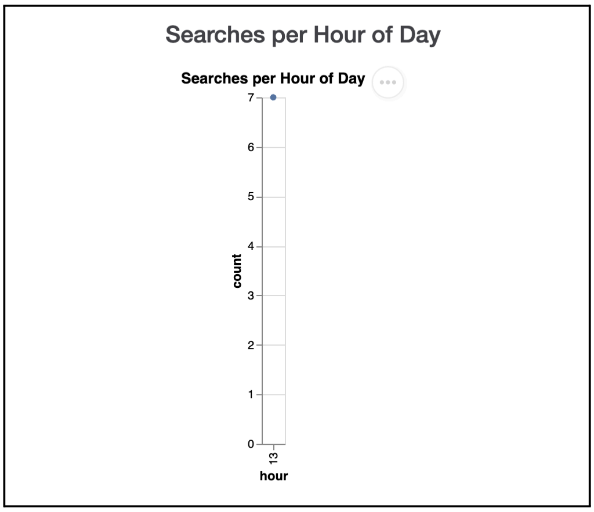

Figure 10 – Searches per hour (all queries were searched at 1pm)

4.3.6 Term co-occurrence heatmap (bonus)

plot_term_heatmap(k=20):

- Finds top-k most frequent terms across queries.
- Builds a co-occurrence matrix over unique terms in each query.
- Visualizes as an Altair heatmap (term vs term, colored by co-occurrence count).

This reveals which terms tend to be queried together (e.g., men + slim + jeans).

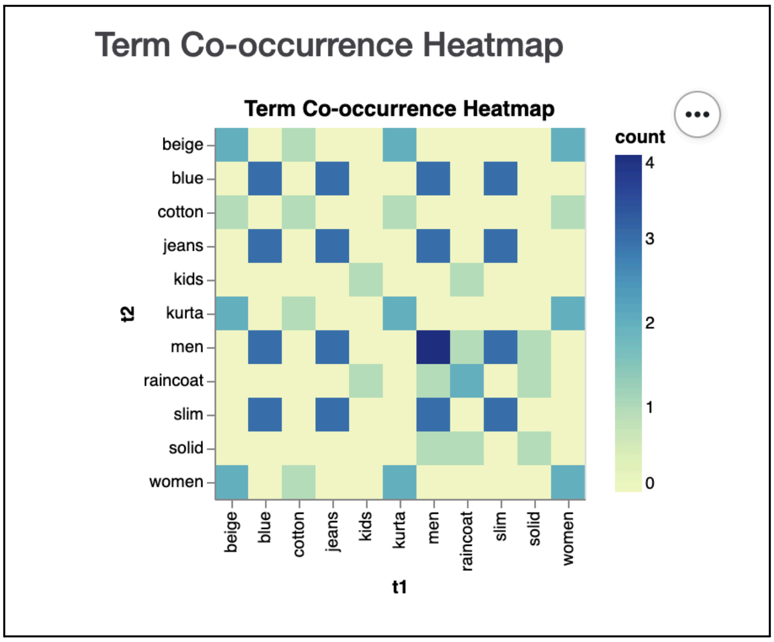

Figure 11 – Term Co-occurrence Heatmap

4.3.7 Session paths (user journeys) – bonus

session_paths():

- Groups requests by session_id.
- For each session outputs the sequence of paths visited (e.g., / → /search → /doc_details → /search).

Displayed as a list under "User Journeys (Session Paths)" in dashboard.html. This approximates **time-based sessions**.

Figure 12 – User Journeys

4.3.8 Intent clusters (missions) – bonus

intent_clusters():

- For each query, normalizes its set of terms into a sorted string key.
- Counts how many times each unique term set appears.
- Sorted by descending frequency.

Displayed as "Query Intent Clusters" with entries like:

3 searches — *blue jeans men slim*
2 searches — *cotton kurta women beige*

This approximates **logical missions**: repeated attempts to satisfy the same information need with slightly different queries.

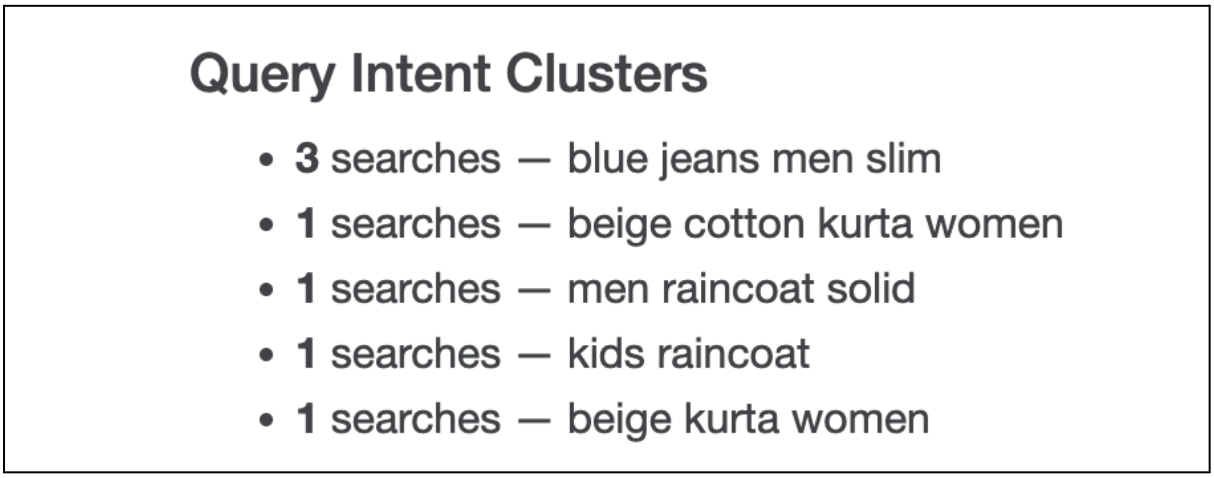

Figure 13 – Query Intent Clusters

4.4 Dashboard UI

The /dashboard route in web_app.py renders templates/dashboard.html, passing:

- stats – from summary_stats().
- funnel – from funnel_metrics().
- paths – from session_paths().
- intents – from intent_clusters().

The HTML shows:

1. KPI cards for basic stats and funnel metrics.
2. Embedded Altair charts in iframes:
    - Top Clicked Products
    - Top Queries
    - Top Query Terms
    - Searches per Hour
    - Term Co-occurrence Heatmap
3. Lists of session paths and intent clusters.

We have seen all the screenshots above that show how we are seeing all those metrics.

5. How to Run and Reproduce

1. Install dependencies (Flask, pydantic, pandas, altair, groq, httpagentparser, python-dotenv, etc.).
2. Create a .env file (not committed to GitHub) with:

```Shell
SECRET_KEY=your_flask_secret
SESSION_COOKIE_NAME=irwa_session
DATA_FILE_PATH=../data/fashion_products_dataset_enriched.json
GROQ_API_KEY=your_groq_key
GROQ_MODEL=llama-3.1-8b-instant
DEBUG=True
```

3. Start the web app:

```Shell
python web_app.py
```

4. Open http://127.0.0.1:8088 in a browser and issue several queries as described earlier, clicking on some products and waiting a few seconds before starting new searches.
5. Visit /stats to see top clicked products and /dashboard to see analytics.

6. Use of AI Assistance

For Part 4, we used **ChatGPT (OpenAI GPT-5.1 Thinking)** as an assistant to:

- Discuss design options for the RAG improvements and analytics data model.
- Suggest concrete code snippets (especially for prompt engineering, helper scoring, and analytics plots).
- Help draft and structure this written report.

All suggestions were manually reviewed, adapted and tested by us. Any mistakes or omissions in the final code and report remain our responsibility.