HiLabs™

Provider Data Quality Analytics & Dashboard

Team Name: beetrootRaita

Adesh Gupta, Aman Behera, Pratham Singla

The complete implementation and training pipeline are available at: GitHub

1 Introduction

Healthcare provider data often suffers from duplication, inconsistent formatting, missing identifiers, and invalid or expired licenses, all of which make reliable analysis and compliance tracking challenging. To address these issues, we developed a comprehensive data analytics and visualization platform that streamlines the entire workflow. The pipeline begins with the raw dataset, followed by entity resolution and deduplication to eliminate redundant records. Data quality assessment and standardization ensure consistency across key fields such as phone numbers, ZIP codes, and provider names. License validation and compliance tracking are performed against state-specific medical board databases, with tailored logic for California and New York. Beyond data cleaning and validation, the platform enables natural language queries through a small language model integrated with a SQL engine, allowing users to interact with the data intuitively. Finally, an interactive analytics dashboard with a robust frontend and backend architecture provides healthcare administrators with real-time insights, supporting better decision-making in credentialing and compliance management.

2 Table of Contents

The pipeline of the platform can be divided into the following parts:

3 Requirements

Requirements

1. Docker
2. Docker Compose
3. Docker Model Runner
4. NVIDIA-supported GPUs

4 Dataset

Overview of the dataset we used for this project:

1. **Provider Directory Dataset** (`provider_roster_with_errors.csv`)
   This dataset contains over 500 provider records and represents the main directory of healthcare providers. It includes various data quality issues such as duplicate entries, expired licenses, missing information, and formatting inconsistencies.

2. **NY State Medical License Database** (`ny_medical_license_database.csv`)
   This dataset contains official records of medical licenses issued by the New York State Medical Board. It includes information like license numbers, expiration dates, provider names, and specialties. We use it to cross-validate and check the validity of provider licenses listed in the main directory.

3. **CA State Medical License Database** (`ca_medical_license_database.csv`)
   Similar to the NY dataset, this file contains licensing records issued by the California State Medical Board. It serves the same purpose of validating provider licenses and detecting expired or mismatched records in the provider directory.

4. **Mock NPI Registry** (`mock_npi_registry.csv`)
   This is a simulated version of the National Provider Identifier (NPI) Registry, containing identifiers assigned to healthcare providers. It helps us validate provider NPI numbers, check for missing identifiers, and cross-check basic provider details like name and specialty.

P.S: The above dataset was provided to us by HiLabs, and was part of our problem statement.

5 Preprocessing the data

5.1 Provider Entity Resolution & Deduplication

The dataset contained multiple records referring to the same real-world healthcare provider, making it essential to identify and remove duplicates.

We initially implemented a naive duplicate detection algorithm with $\mathcal{O}(n^2)$ time complexity. For a dataset of $n = 500$ records, this required approximately

$$\frac{n(n-1)}{2} \approx 125{,}000$$

pairwise comparisons. As a result, the process was slow, taking 40-45 sec just to identify duplicates.

Instead of exhaustively comparing every provider record against every other record ($\mathcal{O}(N^2)$ complexity), we implemented a **blocking-based deduplication pipeline** that groups records into candidate sets using high-precision keys such as exact NPI, standardized phone number, zip3 + name key, and practice address. Comparisons are then restricted to records within the same block, drastically reducing the number of pairwise evaluations. Each candidate pair is scored using string similarity measures (normalized Levenshtein/Jaccard), address similarity, and specialty consistency, and only pairs above a tuned threshold are retained as duplicates.

This approach achieves near-linear runtime in practice by eliminating irrelevant comparisons while maintaining high accuracy, making it significantly faster than the naïve pairwise baseline.

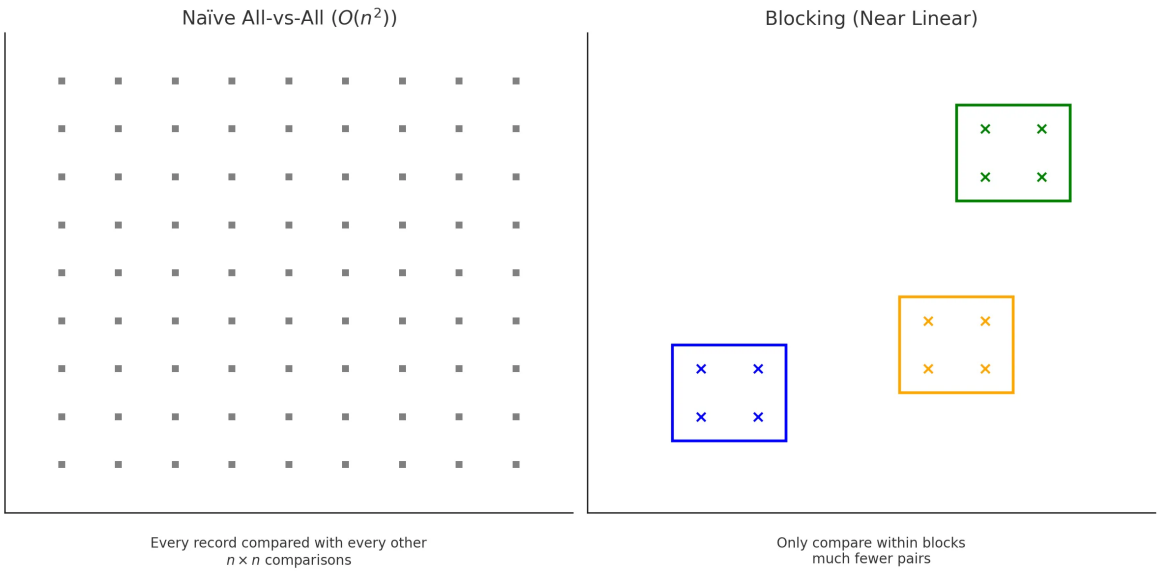

Figure 1: Diagram to represent the two approaches

Dataset Size (n)	Naive O(n²)	Blocking O(n+p)	Speedup Factor
1,000	499,500	~5,000	100×
10,000	49,995,000	~50,000	1,000×
100,000	4.99B	~500,000	10,000×

Figure 2: Performance Improvement Statistics

Our results show a speedup of more than 10x, with runtime for duplicate detection reduced from 45-50s (naive) to just 5-7s (blocking-based) on the same dataset. Here p = block grp size

5.2 Data Quality Assessment & Standardization

Raw provider data often contained inconsistencies in phone numbers, ZIP codes, names, and addresses, making cross-database matching unreliable. To address this, we designed a standardization pipeline that enforces consistent formatting before downstream processing:

- **Phone Numbers:** Only digits are retained, removing symbols and spaces.
  Example: `(123)-456-7890` → `1234567890`.
- **ZIP Codes:** Normalized with zero-padding and converted ZIP+4 codes into standard format.
  Examples: `123` → `00123`, `123456789` → `12345-6789`.
- **Name & Address Fields:** Converted to title case to eliminate formatting discrepancies.
  Example: `JOHN DOE` → `John Doe`.
- **Full Name Reconstruction:** Rebuilt `full_name` dynamically from `first_name`, `last_name`, and `credential` to enforce consistency across records.

5.3 License Validation & Compliance Tracking

For license validation, we needed to merge two tables, each having a different key structure. Providers are first filtered based on `license_state`:

- **California (CA):** System looks up the CA License Table using `license_number` as join key.
- **New York (NY):** The system queries the NY License Table using a composite join key of `license_number` plus `expiration_date` to identify the correct record.

This ensures that license status is retrieved accurately according to state-specific requirements.

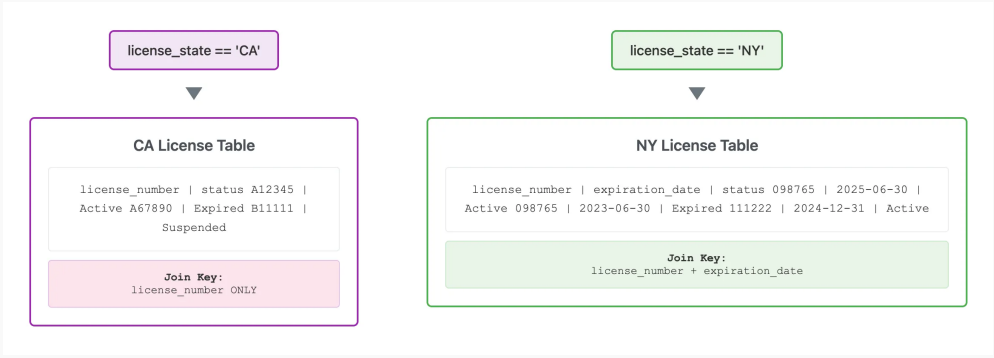

Figure 3: License validation table merge schematic diagram

5.4 Compliance Score Calculation

The compliance score was calculated using a combination of data quality metrics: completeness, validity, consistency, uniqueness, accuracy, missing NPI records & expired licenses. Each metric was measured independently, and the overall score was computed as the average of these dimensions.

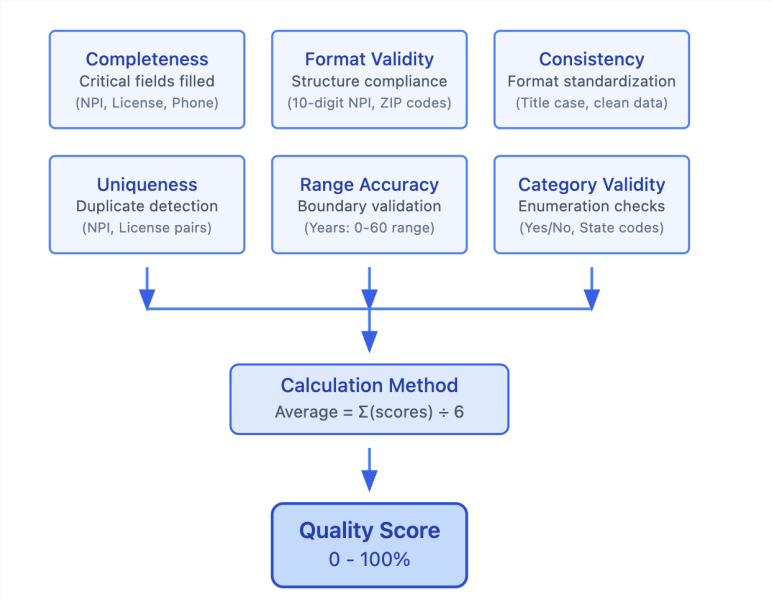

Figure 4: Compliance Score calculation pipeline

5.5 NPI Matching (Present/Not Present)

NPI was used as the primary identifier for providers. For missing NPIs, fuzzy matching using names, phone numbers, license numbers, addresses, and city-state combinations ensured reliable linkage. The merged dataset includes a boolean `npi_present` flag indicating whether the provider's NPI exists in the reference NPI roster, helping identify missing or unmatched records.

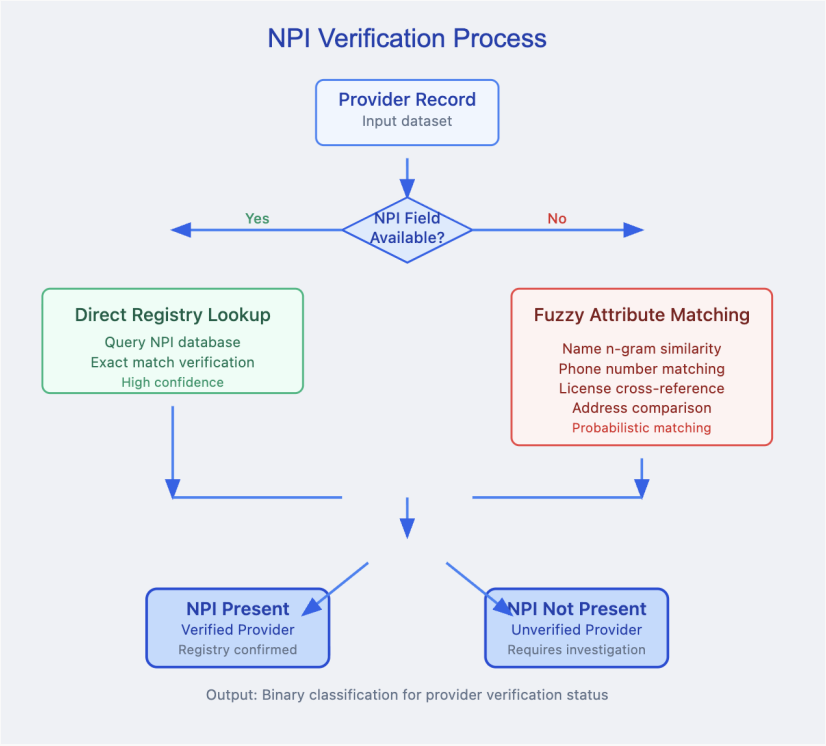

Figure 5: NPI matching methodology

5.6 Outlier Removal

Outliers and invalid entries were filtered from the dataset. Specifically, records with unrealistic values in `years_in_practice` (outside 0–60) were removed. Additionally, missing or malformed entries in `practice_phone`, `practice_zip`, and other key columns were standardized or cleaned to ensure data consistency.

5.7 Final preprocessed data format & Summary Format

This shows how our data looks after preprocessing, along with the summary generated after cleaning and restructuring the dataset.

#	Column	Non-Null Count	Dtype
0	provider_id	500 non-null	object
1	npi	500 non-null	int64
2	first_name	500 non-null	object
3	last_name	500 non-null	object
4	credential	500 non-null	object
5	full_name	500 non-null	object
6	primary_specialty	500 non-null	object
7	practice_address_line1	500 non-null	object
8	practice_address_line2	159 non-null	object

9	practice_city	500 non-null	object
10	practice_state	500 non-null	object
11	practice_zip	500 non-null	object
12	practice_phone	500 non-null	object
13	mailing_address_line1	500 non-null	object
14	mailing_address_line2	154 non-null	object
15	mailing_city	500 non-null	object
16	mailing_state	500 non-null	object
17	mailing_zip	500 non-null	object
18	license_number	500 non-null	object
19	license_state	500 non-null	object
20	license_expiration	500 non-null	object
21	accepting_new_patients	500 non-null	object
22	board_certified	500 non-null	bool
23	years_in_practice	500 non-null	int64
24	medical_school	500 non-null	object
25	residency_program	500 non-null	object
26	last_updated	500 non-null	object
27	taxonomy_code	500 non-null	object
28	status	500 non-null	object
29	npi_present	500 non-null	bool

Table 1: Provider dataset schema after preprocessing

As shown in the table, the dataset contains 30 columns: 28 from the original provider dataset, 1 indicating the license status (from the combined CA and NY datasets), and 1 indicating whether the NPI is present or not.

Why do we only store the relevant columns?

We limit the schema to relevant columns because the data will be passed to the small language model (SLM). Including hundreds of columns can easily exceed the SLM's context window, which may lead to incoherent responses or even no response at all.

While preprocessing, we also generate a structured summary, which is passed to the frontend for displaying the analysis. We make sure a schema is followed so that the data stays consistent and easily interpretable

Example of Summary Passed to the Frontend

```
Summary {
  "total_records": 524,
  "candidate_pairs": 46229,
  "duplicate_pairs": 28,
  "unique_involved": 44, "clusters": 20,
  "outliers_removed": 0, "final_records": 500,
  "expired_licenses": 459, "missing_npi": 0,
  "providers_available": 165, "ca_state": 188,
  "ny_state": 312, "formatting_issues": 59,
  "compliance_rate": 8.2, "data_quality_score": 87.73
}
```

6 Catering Natural Language Queries

6.0.1 Small Language Model

We hosted a locally optimized small language model (`gemma2-it-GGUF`) using the `llama.cpp` engine via a Docker model runner service for easy integration. It takes the Natural Language Query and generates the SQL query which is fed into the SQL Engine, which runs the commands and returns the fetched results.

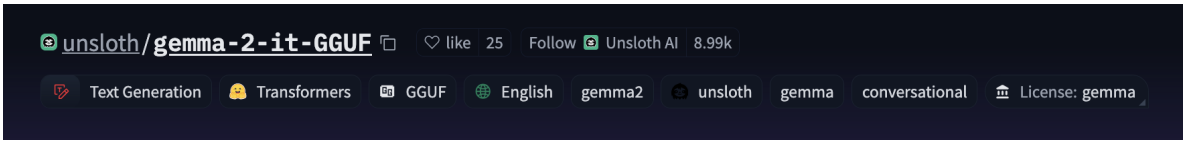

Figure 6: unsloth model card

6.0.2 SQL Engine

A Dockerized MySQL service stores both the original and preprocessed data for the model to query. The model converts natural language queries into SQL statements, which are executed by the SQL engine. The results are then interpreted by the small language model and presented to the user.

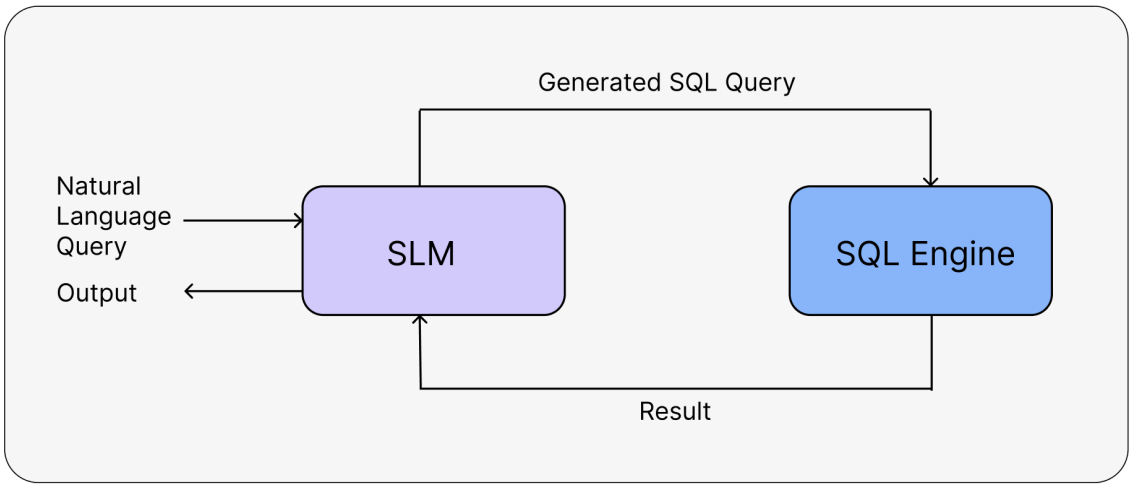

Figure 7: Diagram representing the flow between the components

7 Frontend

The frontend consists of an analytics dashboard , duplicate analysis page, provider directory page, AI Assistant built using `Next.js` and `shadcn`. It includes high-quality plots for deduplication, duplicate analysis, anomaly detection, ensuring a smooth user experience and in-depth insights.

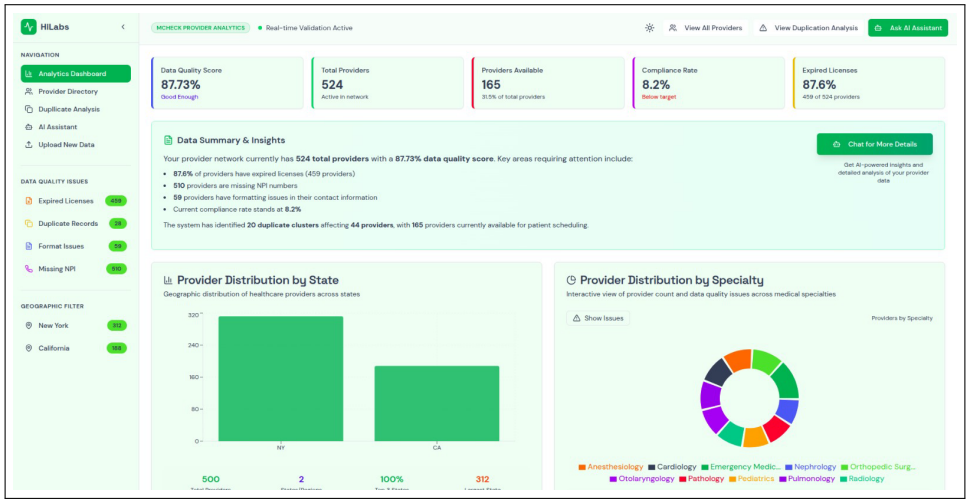

Figure 8: Analytics Dashboard Page

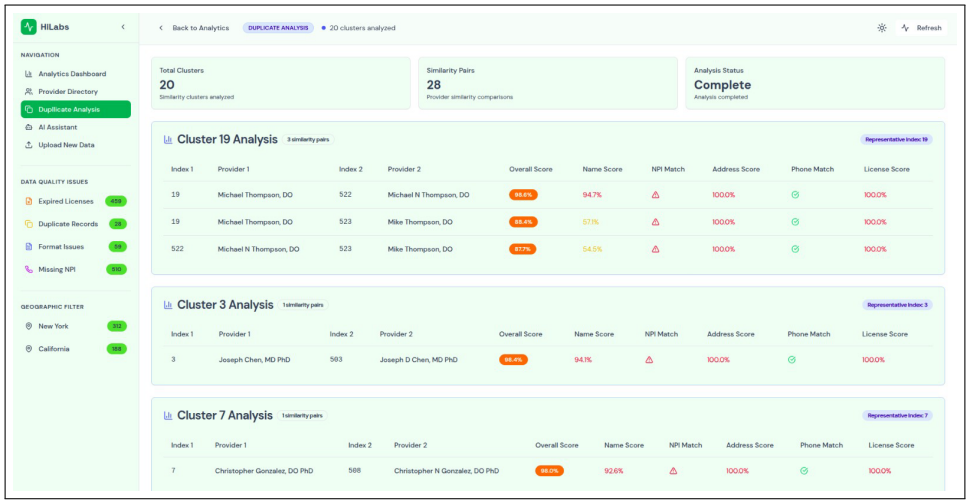

Figure 9: Duplicate Analysis Page

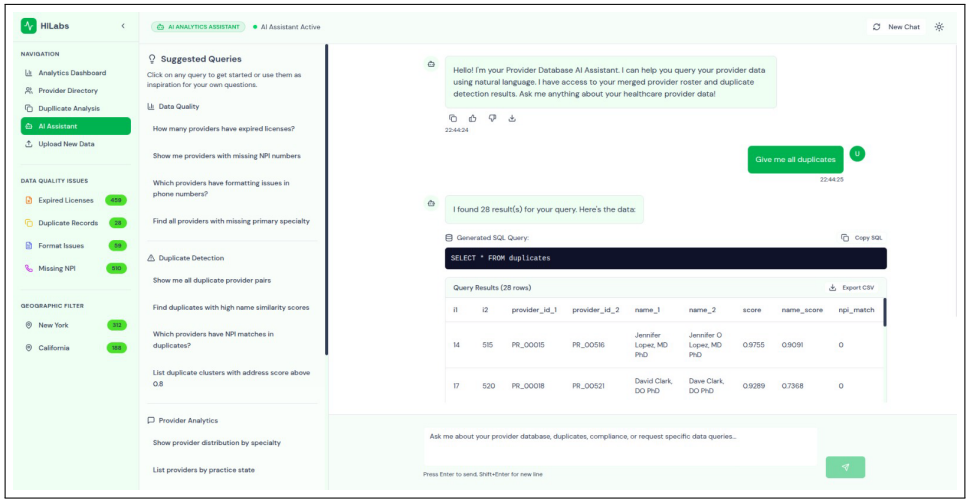

Figure 10: AI assistant

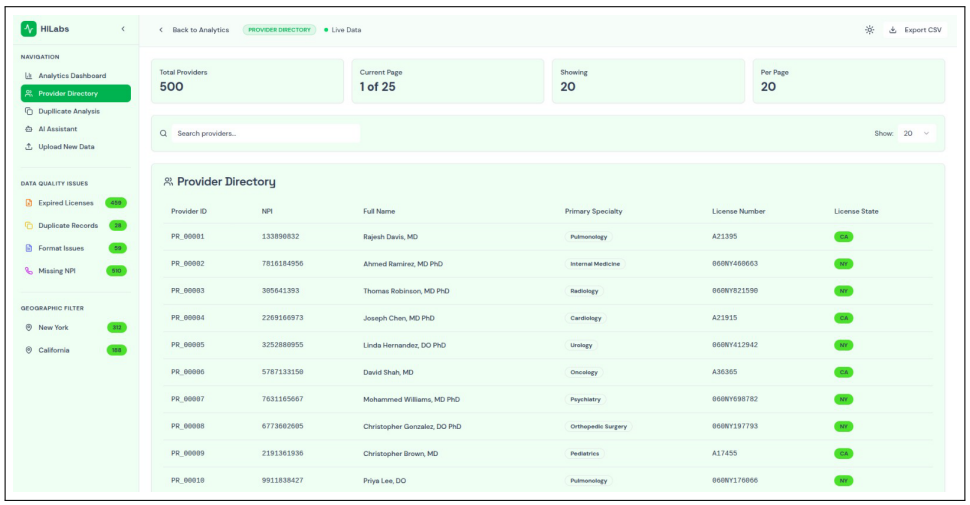

Figure 11: Provider Directory Page

7.1 Plots for insights

Several plots have been included to illustrate the results of the data analysis.These visualizations help highlight key trends and patterns observed in the dataset. Examples are shown below:

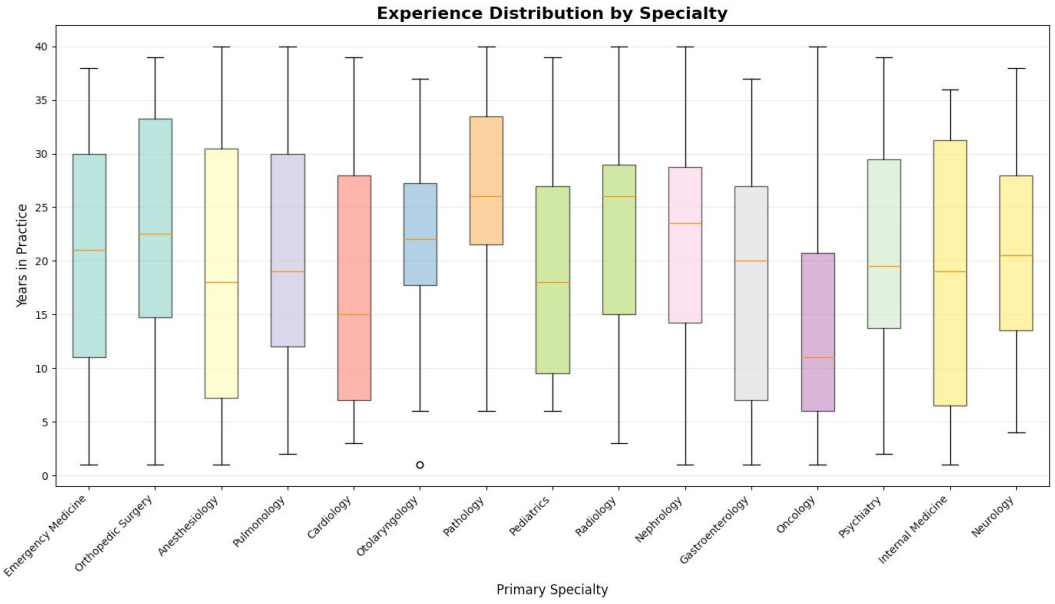

Figure 12: Experience distribution by speciality

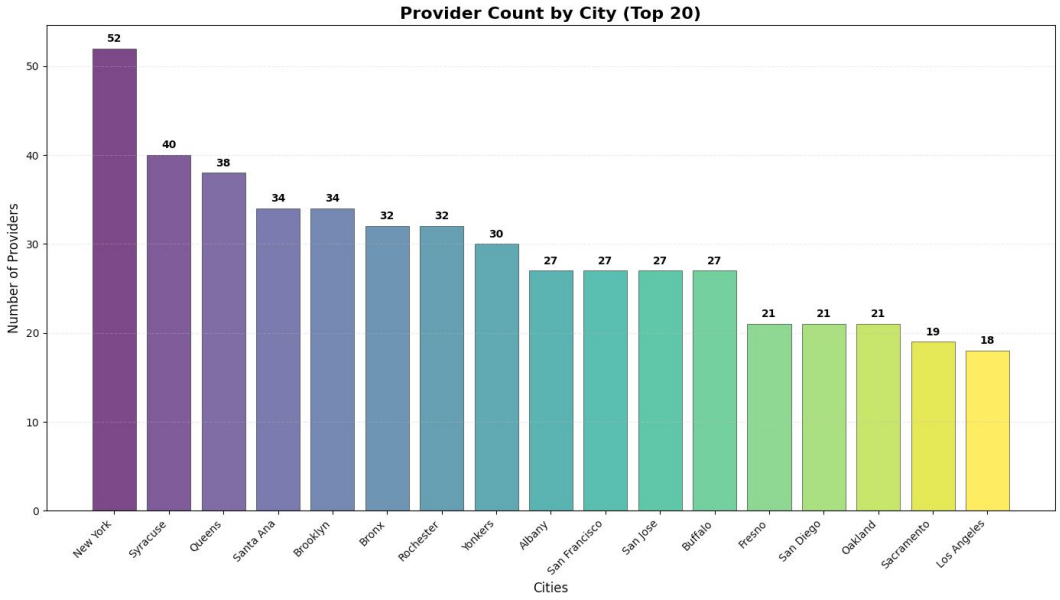

Figure 13: Provider count by city (Top 20)

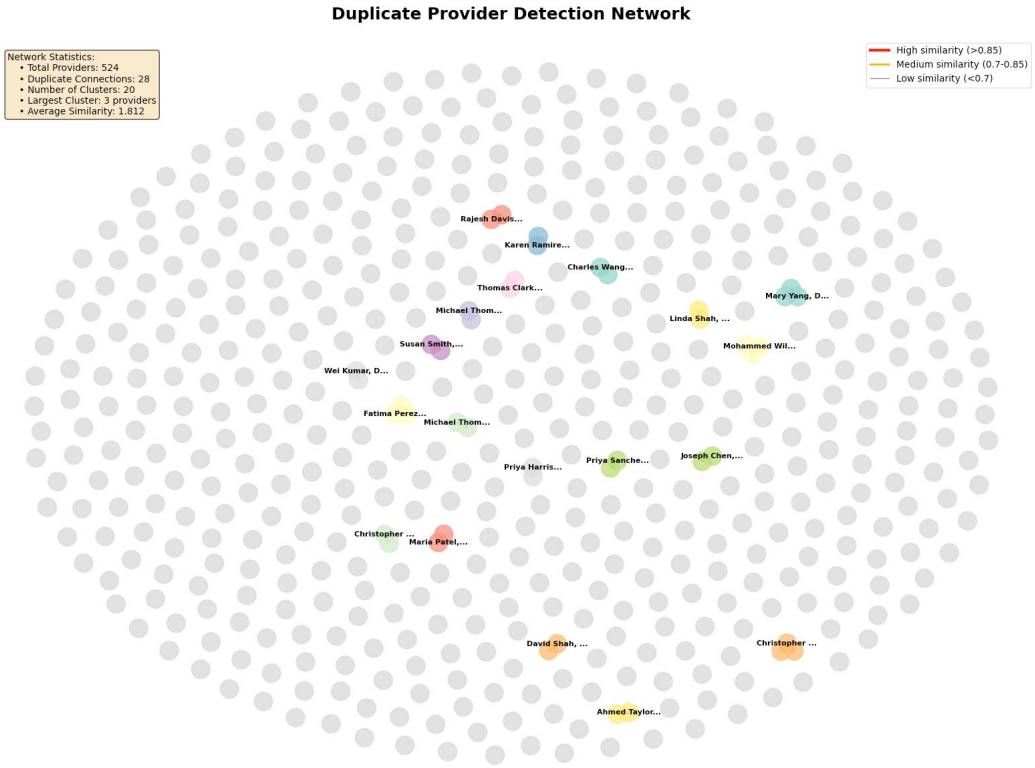

Figure 14: Duplicate provider detection network

8 Backend

The backend is implemented with `FastAPI` and handles the preprocessing pipeline as well as SQL connections, providing APIs for the dashboard and the natural language query engine.