Job Post Similarity Detection

Video Resource

YouTube Video: Job Post Similarity Detection

Objective

This project identifies duplicate or highly similar job postings from a given dataset using text embeddings and vector search techniques. It leverages Natural Language Processing (NLP) and demonstrates software engineering practices through containerization with Docker Compose.

Table of Contents

Project Structure

The project is organized as follows:

```
job-post-similarity
├─ .env
├─ .env.placeholder
├─ Dockerfile
├─ EDA_preprocess.ipynb
├─ LICENSE
├─ README.md
├─ analysis_files
│  ├─ EDA_proprocess.pdf
│  ├─ Embedding Model Choice Justification.pdf
│  ├─ Similarity Threshold Justification.pdf
│  ├─ qual_analysis_first_row.pdf
│  ├─ qual_analysis_last_row.pdf
│  ├─ qualitative_analysis_results.csv
│  └─ similarity_distribution.png
├─ app
│  ├─ fetech_jd.py
│  ├─ generate_embeddings.py
│  ├─ main.py
│  ├─ preprocess_data.py
│  └─ vector_search.py
├─ data
├─ docker-compose.yml
└─ requirements.txt
```

Data Exploration & Preprocessing

Data Exploration (`EDA_preprocess.ipynb`)

Initial analysis of the `jobs.csv` dataset (100k rows, 17 columns) revealed several key points:

- **Identifier:** `lid` column contains unique identifiers.
- **Content:** `jobDescRaw` contains job descriptions in HTML format. `jobTitle` and `companyName` provide basic job info. "Delivery Driver" and "DoorDash" were most frequent.
- **Missing Data:** Low percentage of missing values in location, company name, and date columns.
- **Duplicates:** No exact row duplicates, but ~4,700 duplicates found based on title, company, zipcode, and date combination. Some raw HTML descriptions were also duplicates.
- **Location Data:** Inconsistencies noted, like trailing commas in `finalState` and "remote" values in `finalZipcode`.
- **NLP Columns:** Columns like `nlpSkills`, `nlpBenefits` often contain empty lists.
- **Visualization:** Word clouds (using the `wordcloud` library) were used to visualize common terms in job descriptions.

Preprocessing (`app/preprocess_data.py`)

Based on the EDA, the following preprocessing steps are performed by the script when `main.py` is run:

1. **Load Data:** Reads the original `jobs.csv` from the `data/` folder.
2. **HTML Parsing:** Extracts clean text from `jobDescRaw` into `jobDescClean` using BeautifulSoup.
3. **Handle Missing Values:** Fills NaNs in categorical/location columns with 'Unknown'; drops rows with missing `correctDate`.
4. **Remove Duplicates:** Drops duplicate rows based on `['jobTitle', 'companyName', 'finalZipcode', 'correctDate']`.
5. **Clean Location:** Standardizes `finalState` (removes trailing commas), `finalZipcode` (handles "remote"), and `finalCity` (title cases).
6. **Clean Text:** Converts `jobDescClean` to lowercase and removes extra whitespace.
7. **Drop Columns:** Removes unused columns like `jobDescRaw`, `nlp*` columns, URLs, etc.
8. **Save:** Outputs the cleaned data to `jobs_processed.csv` (likely inside the container's `/app/data` directory during Docker execution).

Embedding Generation

Model Choice (`Embedding Model Choice Justification.pdf`)

- **Model:** `sentence-transformers` library with the `all-MiniLM-L6-v2` pre-trained model is used.
- **Justification:**
  - Optimized for semantic similarity of sentences/paragraphs, capturing context better than word embeddings (GloVe, CBOW).
  - The library provides a simple API for loading models and generating embeddings.
  - `all-MiniLM-L6-v2` offers a good balance of performance and efficiency for this task.
  - Transformer-based models handle job description nuances and terminology well.

  - Outperforms averaging word vectors or using general spaCy models for this specific similarity task.

Implementation (`app/generate_embeddings.py` logic within `main.py`)

- The `main.py` script loads the `sentence-transformers` model.
- It reads the `jobDescClean` column from the processed data.
- It encodes the descriptions into 384-dimensional vectors.
- The embeddings array (`job_embeddings.npy`) and corresponding `lid` array (`job_ids.npy`) are saved (likely inside the container's `/app/data` directory during Docker execution) or loaded if they already exist.

Hugging Face Authentication

- **Note:** While some Hugging Face models require authentication (using `huggingface-cli login` or setting the `HF_TOKEN` environment variable), standard models like `all-MiniLM-L6-v2` typically download automatically without needing explicit login or tokens. Authentication is usually only necessary for private or gated models.

Vector Search Implementation

Library Choice (`Vector Search Implementation Plan.pdf`)

- **Library:** `Faiss` (Facebook AI Similarity Search) is used.
- **Justification:**
  - Offers comprehensive and flexible indexing options (exact search like `IndexFlatL2`, ANN methods like HNSW, IVF).
  - Highly optimized for performance.
  - Mature library with good documentation and community support.
  - Fits project scope without the overhead of a full vector database.

Implementation (`app/vector_search.py`)

- A `VectorSearch` class encapsulates Faiss operations.
- **`__init__`**: Initializes the Faiss index using `faiss.index_factory` (using `IndexFlatL2` for exact L2 distance search) and an `id_map` list to link Faiss indices back to original `lid`s.
- **`train`**: Handles index training if required by the index type (not needed for `IndexFlatL2`).
- **`add`**: Adds batches of embeddings (converting to float32) and appends corresponding original string IDs to the `id_map`.
- **`remove`**: Not implemented efficiently for `IndexFlatL2`; raises `NotImplementedError`, recommending rebuilding the index if removal is needed.
- **`search`**: Finds k-nearest neighbors using `index.search`, returning L2 distances and the original string IDs (retrieved via `id_map`). Handles single vector queries and cases where k > number of indexed items.
- **`save`/`load`**: Methods to save the Faiss index (`faiss.write_index`) and the `id_map` (using pickle) to disk, and load them back.

Evaluation Techniques

Accuracy Check (`Evaluation Plan Summary.pdf`)

Since no ground truth is available, accuracy is assessed via:

1. **Qualitative Analysis:** Randomly sampling jobs, finding nearest neighbors using `VectorSearch.search`, and manually reviewing if the retrieved neighbors are actual duplicates or highly similar. (See `analysis_files/qual_analysis_first_row.pdf` and `analysis_files/qual_analysis_last_row.pdf` for examples of this analysis output).
2. **Similarity Score Distribution:** Plotting histograms of cosine similarity scores for (a) nearest neighbor pairs (likely duplicates) and (b) random pairs (likely non-duplicates) . This helps visualize if embeddings effectively separate similar/dissimilar items. The generated plot can be found in `analysis_files/similarity_distribution.png`.

Similarity Threshold (`Similarity Threshold Justification.pdf`)

- **Method:** The similarity score distributions (see `analysis_files/similarity_distribution.png`) are examined to find a threshold that separates likely duplicates from non-duplicates. Qualitative spot-checking refines the choice.
- **Chosen Threshold: 0.90** (Cosine Similarity).
- **Justification:** This threshold provides a good balance, lying in a region where random pair density is near zero (minimizing false positives) while still capturing the majority of the nearest neighbor distribution peak. It prioritizes precision, as confirmed by manual checks.