Technische Universität Berlin

Web-Service-Engineering

Fakultät IV

Einsteinufer 17

10587 Berlin

https://www.tu.berlin/ise

WSE Semester project

Group: Softwaregini

Marcel Heidebrecht, Alex Gaballa, Adham Gouda, Robert Hofmann, Dennis Enrique Gehrmann

Matriculation Number: 410505, 505329, 513513, 414151, 508060

23.02.2025

Course Leader

Prof. Dr. Timm Teubner

Contents

1 Introduction **1**

1.1 Challenge 1 Definiton 1

1.2 Challenge 2 Definition 2

2 Team and Project Organization **3**

3 Methodology & Results **5**

3.1 Challenge 1 5

3.1.1 Early Stages: 5

3.1.2 Pre-Processing Of Data 6

3.1.3 Matching Algorithm 8

3.1.4 Parameter Testing of Local matching approach 14

3.1.5 Results / Evaluation 17

3.2 Challenge 2 19

3.2.1 Data 19

3.2.2 Feature Extraction 19

3.2.3 Text Embedding 24

3.2.4 Clustering 26

3.2.5 Redundancy Detection 27

3.3 API 28

4 Discussion & Learnings **30**

4.1 Discussion/Learnings Challenge 1 30

4.2 Discussion/Learnings Challenge 2 31

4.2.1 Feature Extraction 31

4.2.2 Embeddings 32

4.2.3 Clustering and Redundancy 32
4.2.4 Conclusion & Outlook 32

Chapter 1

Introduction

Software tools provide significant value to organizations, but managing them can be challenging. Many organizations maintain a large and diverse inventory of tools, each with unique features and recurring costs. Tracking these tools manually can be time-consuming and prone to errors.

In cooperation with Softwaregini[1] this paper proposes a solution consisting of two tools. The first tool [1.1] automatically matches the tools in a customer's inventory with those in Softwaregini's database. This database consist of a collection of software tools and rich information related to it. The second tool extracts the features of a given tool in the database, compares them with others, and calculates a feature overlap [1.2]. This approach provides a comprehensive overview of the tool inventory, helping customers make informed decisions about which tools to retain, replace, or upgrade.

The two tools combined take a customer's inventory as input and provide a list of tools that are redundant or have overlapping features. This helps in identifying consolidation candidates and optimizing the tool inventory. The system will be evaluated based on its ability to identify consolidation candidates and its run-time performance.

1.1 Challenge 1 Definiton

The design and development process of the matching algorithm for the first phase of the project. This part will be responsible for processing software inventory data provided by the customer and matching it with records from Softwaregini's database. The goal

[1] `https://softwaregini.com/`

is to create a solution that accurately identifies and pairs entries based on software names, descriptions, and other relevant data. By leveraging advanced techniques in data cleaning, embedding generation, and semantic similarity, the algorithm will ensure that the matching process is both efficient and reliable. This part will focus on the key steps involved in transforming the customer's raw data into a structured, accurate report, providing insights into the matching process and its overall design.

1.2 Challenge 2 Definition

Part of the provided data consists of a collection of tools, each identified by a name, description, provider, and a list of related websites. This list includes not only the URLs but also the scraped markdown from each URL, stored as a single long string. The goal is to develop a metric for measuring feature overlap between tools. To quantify the percentage of overlapping features and systematically list them, the task has been divided into five subtasks:

1. Extract features from the markdown content of each tool and enrich this set by directly asking a LLM for further features. [3.2.2]
2. Create text embeddings to compare the extracted features. [3.2.3]
3. Cluster the features to identify commonalities and handle duplicate features arising from multiple sources (each tool has a list of URLs). [3.2.4]
4. Compute overlapping feature clusters between tools and calculate the percentage of overlap. [3.2.5]
5. Build an API to access the results. [3.3]

Chapter 2

Team and Project Organization

The project was divided into two main areas: **Challenge 1** and **Challenge 2**. Three team members—Alex Gaballa, Adham Gouda, and Dennis Enrique Gehrmann—focused on Challenge 1, while Marcel Heidebrech and Robert Hofmann worked on Challenge 2. This division enabled efficient parallel development, as both tasks could be implemented independently. Only the final integration required close collaboration between the two teams.

Contact Persons at Softwaregini

- Johannes Bock
- David Mente

To ensure a structured workflow, tasks were organized into work packages and tickets, which were managed in **Notion**[1]. Notion was also used for documenting meeting notes and key implementation decisions.

Communication with **Softwaregini** took place via **Slack**[2] and email, while meetings were conducted through **Google Meet**[3]. Internal team coordination was primarily handled via **WhatsApp**[4] for quick and informal discussions. Softwaregini provided data via **Google Drive**[5]. For collaborative development, a shared **GitHub**[6] repository was used, ensuring seamless version control and teamwork. The paper was written collaboratively

[1] https://www.notion.so/
[2] https://slack.com/
[3] https://meet.google.com/
[4] https://www.whatsapp.com/
[5] https://drive.google.com/
[6] https://github.com/

in **Overleaf**, allowing for an efficient and structured approach to document creation. Throughout the development process, **Jupyter Notebooks** played a central role, supporting all stages of the project. Using **Python**, the team conducted data analysis, implemented algorithms, and visualized results. This interactive approach facilitated efficient development and streamlined result sharing among team members.

Chapter 3

Methodology & Results

3.1 Challenge 1

3.1.1 Early Stages:

The team has been provided with specific requirements for the first phase of the project. The software to be developed will process data that is currently being handled manually, originating as a "dirty Excel sheet from customers. This sheet, which is typically filled out manually, represents the customer's software inventory and includes multiple data columns—two of which are particularly relevant: one for the name of the software and another for the description of the software. Additionally, the team has access to Softwareginis extensive database, which contains records to be matched with entries from the customers inventory. Matching can occur based on identical names or similar descriptions, or some descriptions can even include links to the softwares website.
However, several challenges must be addressed to ensure accuracy. Minimizing false positives is critical, as incorrect matches—such as applications linked solely through vague descriptions but not being the same— can lead to costly errors. Furthermore, not all entries in the customer's Excel sheet include descriptions. To address this, the software should have the capability to retrieve and analyze the applications online documentation during the matching process.
Other complexities include handling renamed applications due to rebranding, identifying in-house developed software, and accommodating scenarios where a single software entry from the customers inventory could match multiple entries in Softwareginis database. The solution must be robust, leveraging these considerations to enhance the reliability

and precision of the matching process.
With all these challenges in mind, the team brainstormed a plan for designing and developing a solution. The goal is to create a system that can take in the "dirty" Excel sheet from customers, clean and process the data, and then accurately match the entries to records in Softwaregini's database. Since software names might change due to rebranding and some descriptions could be vague or missing, the algorithm needs to be smart enough to handle these complexities while minimizing false matches. It will also need to recognize in-house software and deal with cases where one entry from the customer's list could match multiple records in the database. In the end, the system will generate a structured report in Excel, providing a clear and accurate analysis of the matched software.
Thus the team decided to split up the process into the following stages:

- Pre-Processing Of Data
- Matching Algorithm
- Matching Evaluation

3.1.2 Pre-Processing Of Data

For the pre-processing stage, the team focused on the data they had available: Softwaregini's database and a manually provided file, both containing columns for software names and descriptions. To ensure accurate and efficient matching, these files need to be cleaned and standardized first, addressing any inconsistencies or formatting issues before the actual comparison begins. Since the Softwaregini database was to be reused for the matching process, this meant that it only had to be pre-processed once, thus the following steps were applied once to it before the processing of the manual input file.

Translation

The Softwaregini database Excel sheet is already in english, so to ensure an accurate comparison for the matching algorithm, the "description" column of the manually provided file must also be translated into english. This is achieved by making asynchronous API calls to OpenAI's ChatGPT. The process begins by checking for and handling any missing values in the "description" column. Then, each text entry is sent to the ChatGPT API, allowing multiple translation requests to be processed in parallel for efficiency.

Once all descriptions have been translated, the updated data is incorporated back into the file, ensuring consistency before proceeding to the matching process.

Data Cleaning

The next step in the process is cleaning the manually provided file to ensure consistency and accuracy before matching. Once the "description" column has been translated into English, the dataset undergoes a cleaning procedure. The file, whether in CSV format or as a DataFrame, is loaded into the system. Then, all string-based columns are processed by converting text to lowercase and removing any unnecessary quotes. Special attention is given to the "description" column, where special characters are stripped to maintain a standardized format. This cleaning step ensures that the data is properly structured, reducing inconsistencies that could affect the matching algorithm's accuracy.

Embedding

Once the manual dataset is cleaned, the "description" column is prepared for the next crucial step—embedding. Embedding is the process of converting text into numerical representations that capture the underlying meaning of the descriptions, making it easier to compare them semantically. To start, any missing or null values in the "description" column are replaced with empty strings. This ensures that the column is fully populated, preventing any errors during the encoding process.

Next, the cleaned descriptions are converted into a list, which will be fed into a pre-trained SentenceTransformer model. This model is specifically designed to transform textual data into high-dimensional vectors, with each vector representing the semantic meaning of a given description. The model works by analyzing the context and structure of the text, ensuring that similar descriptions are mapped close to each other in the vector space.

To optimize the process and handle larger datasets more efficiently, the descriptions are encoded in batches rather than one at a time. This batch processing reduces the computational load and speeds up the overall process. Once the embeddings are generated, they are added back to the dataset as a new column, allowing each description to be paired with its corresponding vector.

These embeddings are essential for the next step: the matching process. They enable a more accurate comparison between the human descriptions and the preprocessed Softwaregini database, as the semantic meaning of the descriptions can now be analyzed, rather than relying on exact text matches. This ensures that the matching algorithm can identify similar tools based on semantics, not just on word choice. After embedding the Softwaregini software database, its encoded representation is saved as a Pickle (.pkl) file, allowing it to be imported in future matching runs without re-encoding the embeddings each time.

3.1.3 Matching Algorithm

First approach: Local matching

The local matching process aligns entries in a customer's software inventory with Softwaregini database records by combining semantic and string similarity using the cosine similarity calculation and fuzzy name matching.

Algorithm Overview

- **File name:**
  - `local_matching.py`
- **Input:**
  - CSV file containing columns for software names and descriptions.
  - Pickle file containing software tools and their precomputed embeddings.
- **Output:**
  - CSV file with matched pairs of Softwaregini database entries with customer tool entries and similarity scores
  - CSV file of software matches that nearly meet the similarity threshold but require review.

The algorithm first generates contextual embeddings for each software description from the customer's file using a pre-trained language model, and then compares these

embeddings to those precomputed in the Pickle file for the database entries. A "description similarity" score is returned as a real number ranging from 0 to 1.
Simultaneously, utilizing the `"fuzz.token_set_ratio"` fuzzy matching algorithm, the similarity of the name's are computed as a percentage. If an entry in the customer's inventory lacks a description, Levenshtein distance is used instead of fuzzy matching to check for an exact name match in the software database. If a direct match is found, it is recorded accordingly. Since Levenshtein distance penalizes differences more strictly than fuzzy matching, it helps reduce false positives and improve precision, in the case of an entry not having a description.
These two scores are combined, giving appropriate weight to both the description and the name, to produce an overall combined (similarity) score for each potential match.

$$combined\_score = (1 - weight) \times (description\_similarity \times 100) + weight \times name\_similarity$$

Matches that exceed a defined cutoff threshold are retained, while those that are close to the threshold are flagged as potential matches for further review. The conditions for potential matches include matches with a similarity score that falls below the cut-off threshold but not less than half of the cut-off threshold value. If a single customer's entry has multiple matches with scores above the threshold, only the match with the highest combined score is retained, while the rest are discarded.

This functionality is executed by the `"load_and_match()"` function, which returns both final confirmed and potential matches, which will be passed on to the Large Language Model in the second approach for subsequent analysis.

Second Approach: AI-Matching

This approach relies on a Large Language Model to interpret software titles and textual descriptions to determine whether a customer's software entry matches a record from the given Softwaregini database. Initially, the idea was to challenge the results of our local-matching approach and to test the capabilities of AI.

Algorithm Overview

- **File name:**
  - `langchain_matcher_async.py`
- **Input:**
  - Preprocessed CSV file containing software names and descriptions.
  - CSV-File of Softwaregini-database with tool records
- **Output:**
  - CSV-File with matched pairs of Softwaregini database entries with customer tool records and similarity scores
- **Assumptions:**
  - Input data is cleaned (lowercase, special characters removed) and translated into English
  - Softwaregini database entries are stored in a Pinecone vector-store index

Workflow Description

1. **Read Softwaregini database CSV-File:** The relevant columns are ‘name‘ and ‘description‘.
2. **Initialize Pinecone:** The Pinecone vector database is initialized by using a Pinecone API-key. An index named ‘tools-matcher‘ is accessed (previously created on https://www.pinecone.io/.
3. **Create Vectorstore (just once!):** The formatted texts and metadata are then used to create a vector store in Pinecone. This involves embedding the texts with the OpenAI embedding model and storing them in the specified Pinecone-index.
4. **Define Prompt Template:** A prompt template is defined for comparing two software tools and determining their similarity score. This template will be used to generate prompts for the language model.
5. **Initialize Language Model:** The language model (gpt-3.5-turbo) is initialized using the OpenAI-API. The LLM will be used to generate similarity scores based on the prompts.

6. **Batch Processing with Async Threading:** The algorithm processes the human-provided data in batches. For each batch, it searches for similar documents in the vector store, generates prompts, and uses the language model to get similarity scores. This is done using asynchronous threading to improve performance in the core method:

```
def process_batch(batch_rows, vectorstore, min_score, batch_size)
```

7. **Aggregate Results:** The results from all batches are aggregated into a single list. The aggregated results are converted into a pandas DataFrame.

8. **Sort and Save Results:** The results are sorted by similarity score in descending order and saved to a CSV-file.

Core Parameters:

`k=1`	Top matches retrieved from Pinecone
`Temperature=0`	Deterministic LLM outputs
`Batch_size=50`	API calls per batch
`Min_score=90`	Match cutoff threshold

Prompt Template:

```
prompt_template = PromptTemplate(
    input_variables=["human_tool", "human_desc", "matched_tool", "matched_desc"],
    template="""Compare these software tools and determine if they are the same product.

    Tool 1:
    Name: {human_tool}
    Description: {human_desc}

    Tool 2:
    Name: {matched_tool}
    Description: {matched_desc}

    Scoring rules:
    100: Exact same product (identical names or official variants)
```

```
    95: Same product with minor name differences (e.g., full name vs. short name)
    90: Same product in different editions/versions
    0: Different products, even if similar purpose or from same company

    DO NOT match:
    - Different products from same company
    - Free vs. Pro versions as different products
    - Similar tools with different core purposes
    - Platform vs. specific service variants

    Respond ONLY with a number between 0-100."""
)
```

Optimization Steps:

- Initial version: Sequential processing with `for` loops and Batching
- Enhanced version: Async via `concurrent.futures` library which imporoved performance by $\sim 70\%$

Validation Methods:

- `evaluate_matching()` calculates precision and recall
- Cross-check against `ground_truth.csv`
- Error analysis of false positives/negatives lists

Combined Approach: Local-Matching + AI-Matching

This approach integrates two methodologies: Local-Matching, which uses semantic embeddings and fuzzy string matching, and AI-Matching, which leverages a Large Language Model (LLM) to refine ambiguous matches. The goal is to improve precision and recall by combining the cheap local matching with the contextual understanding of LLMs.

Algorithm Overview

- **File name:**

  - `pipeline.py`

- **Input:**
  - Raw CSV file with customer software inventory (`organization_tools.csv`).
  - Softwaregini database entries
- **Output:**
  - `pipeline_final_matches.csv`: High-confidence matches from both approaches.
  - `pipeline_potential_matches.csv`: Matches requiring human review.
- **Assumptions:**
  - Input data is cleaned (lowercase, special characters removed) and translated to English (if necessary)
  - Preprocessed Softwaregini database entries are stored in `.pkl` file and in Pinecone vector-store-index as embeddings)
  - API keys for OpenAI and Pinecone are available in the environment.

Workflow Description

1. **Preprocessing:**
   - **Translation:** Non-English descriptions are translated using GPT-4 via `translator.py`.
   - **Data Cleaning:** Special characters, quotes, and casing are normalized using `clean_file()` from `local_matching.py`.
2. **Local Matching:**
   - **Embedding Generation:** Customer descriptions are converted into embeddings using `SentenceTransformer` (e.g., `all-MiniLM-L6-v2`).
   - **Fuzzy + Semantic Matching:** Combined similarity scores (`combined_score`) are computed using:
     - Semantic similarity (cosine similarity of embeddings).
     - Name similarity (fuzzy matching with `rapidfuzz.token_set_ratio`).

- Matches above a cutoff threshold (e.g., 75) are retained, while potential matches (50%–75% score) are flagged for LLM validation.

3. **AI-Matching Validation:**

   - **Vector Search:** Potential matches are embedded by `OpenAIEmbeddings` and searched in Pinecone's vector store index (`tools-matcher`) which contains previously stored softwaregini database entries.
   - **LLM Scoring:** GPT-4o evaluates pairs using a structured prompt to assign a similarity score (0–100). Matches scoring ≥95 are promoted to final results.

4. **Result Aggregation:**

   - Final matches (`df_final_async`) combine:
     - High-confidence local matches.
     - LLM-validated potential matches.
   - Duplicates are removed, and results are saved to CSV.

Core Parameters:

`Model (Local)`	`all-MiniLM-L6-v2`
`Model (AI)`	`gpt-4o` / `text-embedding-3-small`
`Similarity Cutoff (Local)`	`75`
`Similarity Cutoff (AI)`	`95`
`Batch Size (Local)`	`64` (embeddings)
`Batch Size (AI)`	`50` (API calls)
`Name Weight (Local)`	`0.55`
`Temperature (AI)`	`0` (deterministic outputs)

Prompt Template: same template as mentioned in AI matching approach

3.1.4 Parameter Testing of Local matching approach

To enhance the performance of the Local Matching approach, we conducted parameter testing to determine the optimal cutoff values for similarity scores. The testing was performed on Raw input data and Preprocessed Data. The goal was to analyze how different cutoff values affect the **F1-Score**, **Precision**, and **Recall** for both types of input data. The results are visualized in the following figures.

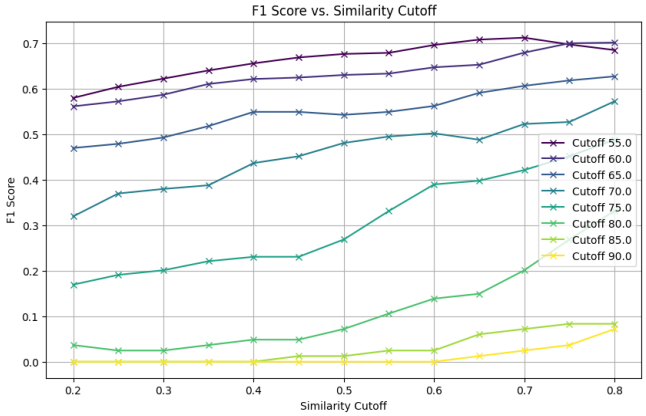

Figure 3.1: F1-Scores with different Cutoffs / without cleaning.

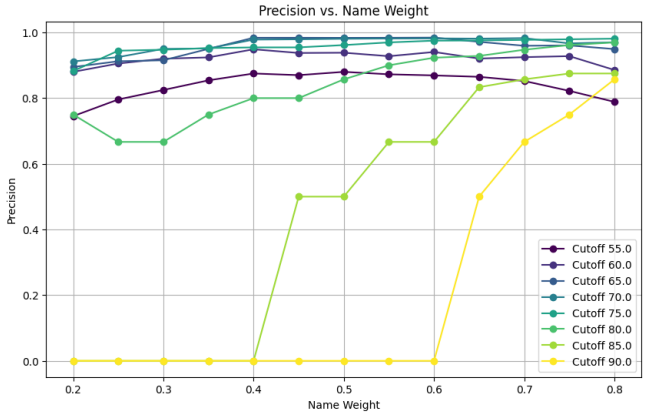

Figure 3.2: Precision-Scores with different Cutoffs / without cleaning.

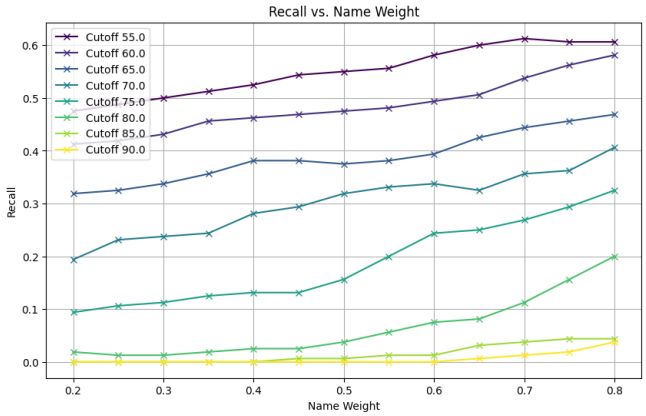

Figure 3.3: Recall-Scores with different Cutoffs / without cleaning.

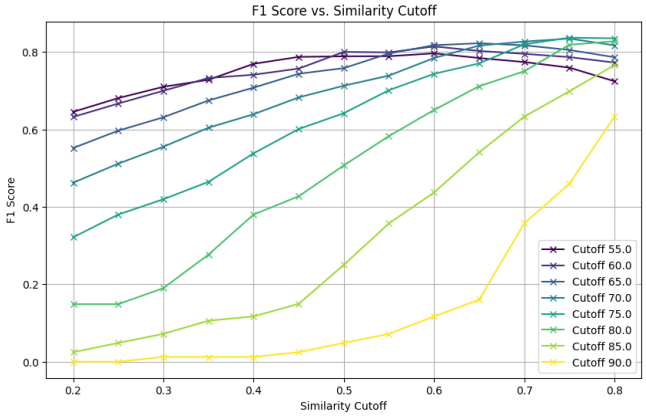

Figure 3.4: F1-Scores with different Cutoffs / with normalization and translation.

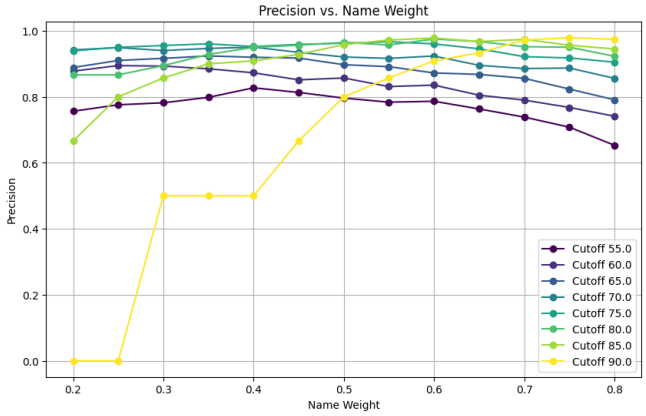

Figure 3.5: Precision-Scores with different Cutoffs / with normalization and translation.

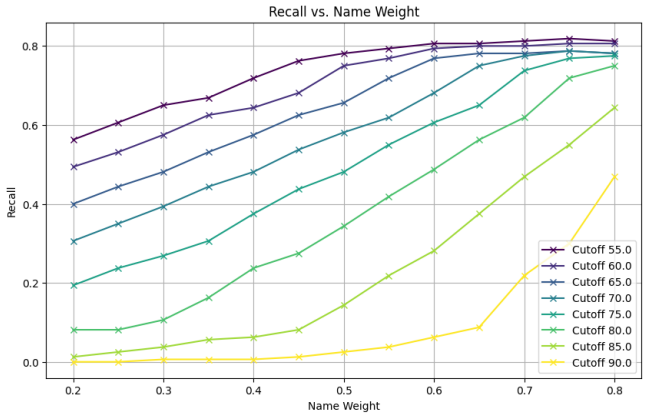

Figure 3.6: Recall-Scores with different Cutoffs / with normalization and translation.

Analysis of Results

- **Unpreprocessed Data**:
  - **F1-Score**: The F1-Score peaked at a cutoff value of **0.75**, indicating the optimal balance between precision and recall for raw data.
  - **Precision**: Precision increased steadily with higher cutoff values, reaching its highest value at a cutoff of **0.95**. However, this came at the cost of significantly lower recall, as stricter cutoffs led to fewer matches being identified.
  - **Recall**: Recall was highest at lower cutoff values (e.g., **0.55**), but it decreased sharply as the cutoff increased. This indicates that while lower cutoffs capture more true matches, they also introduce more false positives.
- **Preprocessed Data**:
  - **F1-Score**: The F1-Score showed a significant improvement compared to unpreprocessed data, with the highest score at a cutoff value of **0.80**.
  - **Precision**: Precision remained consistently high across cutoff values, even at lower thresholds. This indicates that preprocessing reduced false positives, as the cleaned and translated data provided more accurate matches.
  - **Recall**: Recall also improved, especially at mid-range cutoff values (e.g., **0.70** to **0.85**). Preprocessing helped identify more true matches without loosing precision.

Based on the analysis, we went for a cutoff value of **0.75**, as it achieves a good F1-Score while it maintains a good precision and a good recall. To enhance the precision of our Local Matching results, we integrated the AI Matching approach, leveraging its capabilities to maximize accuracy and minimize false positives.

3.1.5 Results / Evaluation

To evaluate the effectiveness of our implementation, we conducted tests using two distinct datasets: `organization_tools.csv` (326 records) and `organization_tools_2.csv` (110 records).

Evaluation Process

- **Initial Testing with** `organization_tools.csv`:
  - The first dataset, `organization_tools.csv`, was used to initially test and refine our matching algorithms. After achieving satisfactory results, we proceeded to test our implementation on a second dataset to evaluate its generalizability.
- **Out-of-Sample Testing with** `organization_tools_2.csv`:
  - The second dataset, `organization_tools_2.csv`, was used as an "out-of-sample" dataset to assess whether our implementation could maintain good performance on unseen data. This step was important to ensure that our approach is robust and not overfitted to the initial dataset.

Evaluation Methodology

To evaluate the accuracy of our matching algorithms, we created two ground truth files ground_truth.csv for organization_tools.csv and ground_truth_2.csv for organization_tools_2.csv. The evaluation process involved cross-checking the matches generated by our algorithms against the corresponding ground truths. The key metrics **precision**, **recall**, **F1-score**, **true positives**, **false positives**, and **false negatives** were calculated to assess the performance of our implementation.

Approach	**Prec.**	**Rec.**	**F1**	**TP**	**TN**	**FP**	**FN**
Local	0.912	0.775	0.838	124	154	12	36
AI	0.915	0.744	0.821	119	156	11	41
Local + AI	0.947	0.787	0.860	126	160	7	34

Table 3.1: Test Results for `organization_tools.csv`

Approach	**Prec.**	**Rec.**	**F1**	**TP**	**TN**	**FP**	**FN**
Local	0.988	0.872	0.927	82	15	1	12
AI	1.000	0.840	0.913	79	16	0	15
Local + AI	0.988	0.872	0.927	82	15	1	12

Table 3.2: Test Results for `organization_tools_2.csv`

Summary

The **Local + AI Matching** approach provided the best balance between precision and recall, especially for the larger dataset (`organization_tools.csv`). For the smaller dataset (`organization_tools_2.csv`), both **Local Matching** and **Local + AI Matching** performed equally well, while **AI-Matching** achieved perfect precision but had slightly lower recall.

3.2 Challenge 2

3.2.1 Data

The data provided for this challenge consists of multiple files, each containing information about a software tool or related to it. These files are connected through IDs, allowing all correlated information for a specific tool to be retrieved using its tool ID. To gain a better understanding of the data, the following tables and graphs will provide a comprehensive overview of the data files, including their contents and the relationships between them.

- **Tools → Website Contents/Tools Documents** (1:n) Each tool can have multiple website contents.
- **Tools → Providers** (n:1) Each tool has one provider. But a provider can have multiple tools.
- **Tools → Categories** (n:n) A Tool can belong to multiple categories and a category can have multiple tools.

3.2.2 Feature Extraction

The feature extraction process involved several systematic steps to identify and collect functional features from real-world software tools. Our methodology focused on creating a robust and maintainable pipeline for feature extraction and storage.

Initially, we identified the crucial data files for feature extraction, primarily focusing on the Tools, Website Contents, and Tools Documents tables, as these contained the most relevant information about tool functionality. We then conducted a thorough analysis

Data File	**Description**
Tools	Contains the ID, name, description, provider ID, and main website of the software tool.
Website Contents/Tools Documents	Contains the markdown content of a tool's website, the tool ID, and the website URL.
Providers	Contains the ID, name, location, website, and social media links of the software provider.
Categories	Contains the name and description of the software categories.

Table 3.3: Overview of Data Files

	id	provider_id	name	description	website
count	774	774	774	774	774
unique	774	537	773	774	774
freq	1	45	2	1	1

Table 3.4: Tools

	tool_id	url	markdown
count	15253	15253	15253
unique	727	13440	12902
freq	100	36	83

Table 3.5: Web Contents

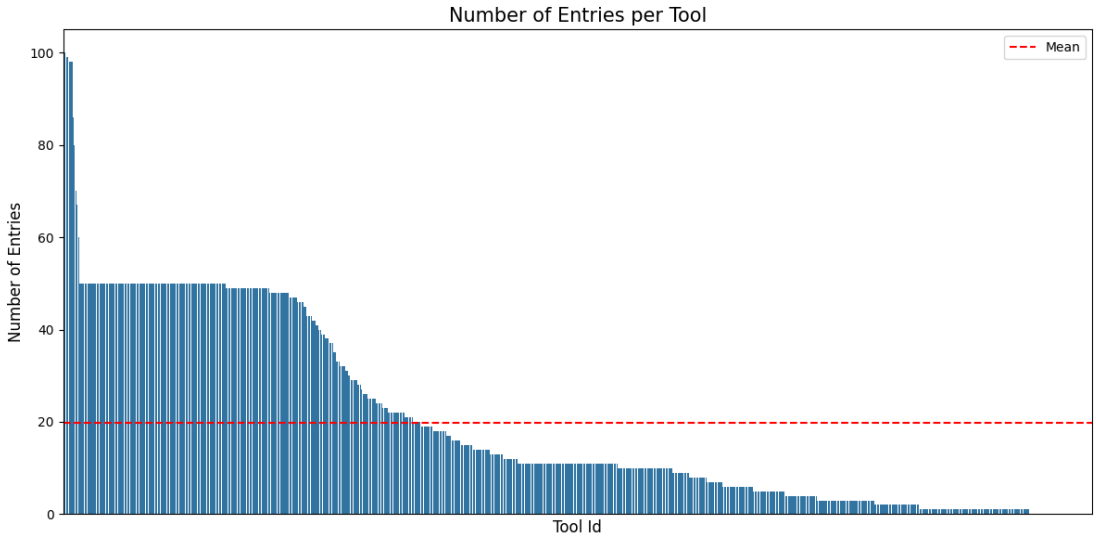

Figure 3.7: Number of Website Contents per Tool

of the file formats and structures to understand the data organization and relationships between different files.

The data preparation phase involved multiple steps:

	id	name	city	country	website	twitter	linkedin	year	employees	...
count	569	569	377	410	7	416	440	364	294	
unique	533	533	206	39	4	381	406	NaN	NaN	
freq	3	3	19	207	3	3	3	NaN	NaN	

Table 3.6: Providers

	id	name	description	archived
count	1722	1722	1722	1722
unique	1722	1722	1722	2
freq	1	1	1	1580

Table 3.7: Categories

id	tool_id	category_id
81001	81001	81001
NaN	46746	1601
NaN	28	458

Table 3.8: Categories to Tools Links

1. Loading the raw data from various sources
2. Cleaning and standardizing the data formats
3. Merging related datasets based on tool IDs
4. Implementing validation checks to ensure data integrity and completeness

To maintain code quality and facilitate future modifications, we structured our implementation in modular Jupyter notebook cells, with each cell handling a specific task in the pipeline. This approach significantly improved maintainability and made it easier to introduce changes or debug issues when necessary.

For the actual feature extraction, we employed OpenAI's GPT-4o-mini model using two distinct approaches:

1. Website Content Analysis: We provided the model with scraped website contents from each tool, prompting it to identify and extract functional features.
2. Direct Feature Identification: We asked the model to independently identify features based on the tool's description and available metadata.

The features extracted from both approaches were merged into a comprehensive dataset and exported to CSV files for further processing. We deliberately postponed feature cleaning and deduplication, as these tasks would be handled implicitly during the subsequent clustering phase, which would group similar features based on their semantic

meaning.

The extracted features and their distribution across tools are summarized in Tables 3.9 and 3.10, which provide statistical information about the feature datasets. Figure 3.8 visualizes the distribution of features across individual tools, while Figure 3.9 illustrates the correlation between the number of website content entries and the number of extracted features per tool.

	feature_id	feature
count	92625	92625
unique	92625	90284
freq	1	16

Table 3.9: Features

	feature_id	tool_id
count	92625	92625
unique	92625	774
freq	1	863

Table 3.10: Features to Tools Links

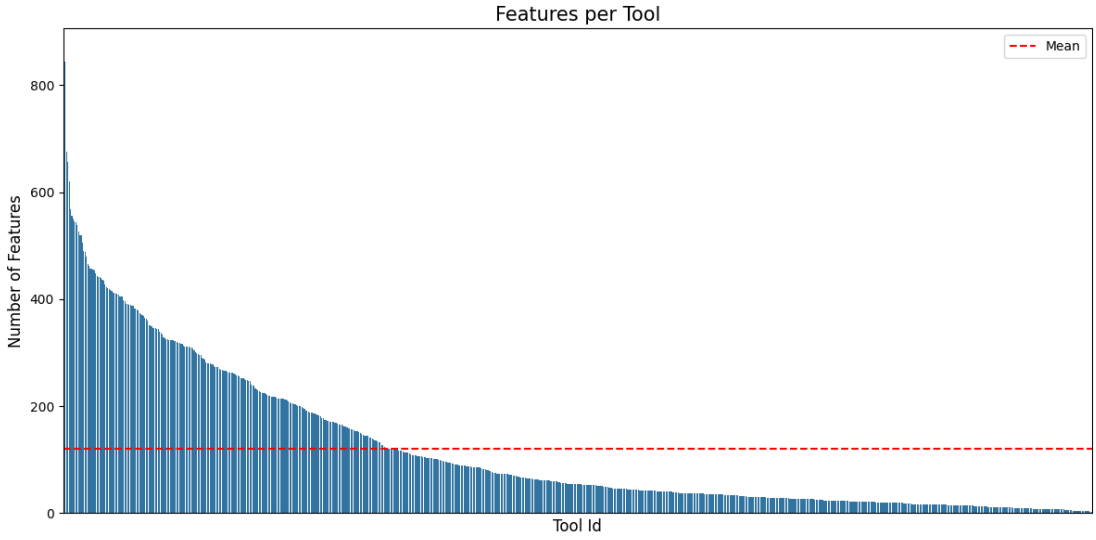

Figure 3.8: Number of Features per Tool

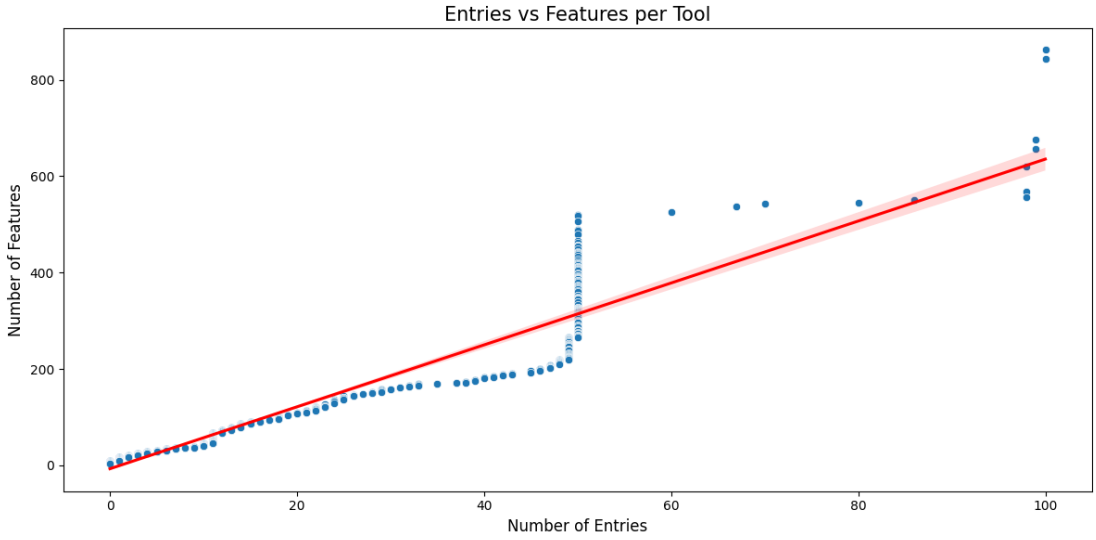

Figure 3.9: Website Contents Entries per Tool vs. Features per Tool Correlation

3.2.3 Text Embedding

With a comprehensive feature list exceeding 90,000 unique entries, the subsequent task involved enabling meaningful feature comparison. An initial approach employing a similarity matching algorithm with RapidFuzz, analogous to the methodology used in Challenge 1, was implemented. However, this strategy yielded unsatisfactory results. As illustrated in table 3.11, features with the same meaning often varied significantly in phrasing and length due to multiple sources and synonymous language. This variance made it impossible for RapidFuzz to reliably identify similar features based on direct string comparison.

	feature
-	Automatic background updates to keep the browser secure and up to date with the latest features.
-	Automatic updates for the browser.

Table 3.11: Example of Feature Variance

Another approach considered was leveraging the Chat API of OpenAI. This method allows for the inclusion of substantial context, such as the tool to which a feature belongs and other relevant details. However, there are key limitations. Firstly, the Chat API can only generate responses in text or text-like structures, which precludes the use of mathematically comparable values. Secondly, the approach would require an initial comparison of all features within a tool to create a cleaner shortlist of features for each tool. Subsequently, all features in the shortlist would need to be compared to one another (within their respective categories). Considering the number of tools and features this process would result in significant complexity, leading to high costs. Furthermore, the lack of repeatability is a concern, as the Chat API's responses may vary, and there is no numerical output to ensure consistency.

Therefore, to address the limitations of both direct string comparison and complete contextual analysis, we opted for a solution based on text embeddings. Initially, the Linq-Embed-Mistral [1] model, a locally hosted embedding model that had demonstrated strong performance on the MTEB[2] (Massive Text Embedding Benchmark) leaderboard, was explored. However, the computational demands required to run Linq-Embed-Mistral

[1] https://huggingface.co/Linq-AI-Research/Linq-Embed-Mistral
[2] https://arxiv.org/abs/2210.07316

proved to be beyond the capabilities of the infrastructure. As a result, OpenAI's text-embedding-3-large API was utilized. This solution, implemented as described in the embeddings.ipynb notebook, involved transforming each extracted feature into a high-dimensional vector representation that encapsulates its semantic meaning. As shown table 3.12 , each feature is represented by a unique feature_id along with its corresponding vector of floating-point numbers. These embeddings enabled the quantification of feature similarity through distance metrics in vector space, providing a more efficient and scalable method for comparing tool features.

feature_id	**embedding**
f440b8...	`[0.024352225, -0.0447857, -0.029604664000000003, ...]`
f2fc40...	`[-0.02192216, -0.02224342, -0.029923059000000002, ...]`

Table 3.12: Feature Embeddings

By default the returned vectors have a length of 3072, but OpenAI claims the vectors can be reduced to a length of 256 without significant loss of information. This reduction in vector length would allow for more efficient processing and storage of the embeddings. However, the impact of this reduction on the quality of the embeddings would need to be evaluated to ensure that the shortened vectors retain the necessary information for accurate feature comparison. For further development, both the full-length and reduced-length embeddings were stored in separate files.

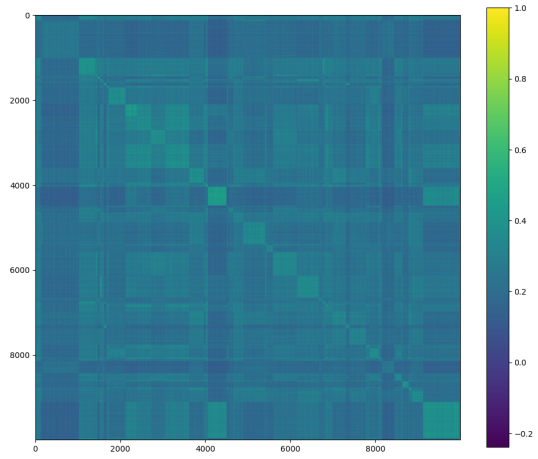

Figure 3.10: Similarity Matrix for Features

To gain an initial understanding of the embeddings, a similarity matrix for the first 10,000 features was plotted, sorted by tool. Cosine similarity was used to calculate the feature similarity, which was visualized as a heatmap. In the matrix, the x-axis and y-axis represent feature indices, while color intensity indicates similarity. The diagonal, representing self-similarity (always 1.0), was set to 0 for bet-

ter visualization. This matrix helps identify clusters of similar features within the dataset.

The similarity matrix in Figure 3.10 provides evidence that the embeddings are functioning as expected. Since each tool has multiple input sources, it is likely that many features within a tool convey the same meaning. This is reflected in the matrix by the larger clusters along the diagonal, which appear because features are sorted by tool. Similarities outside the diagonal indicate relationships between features from different tools.

3.2.4 Clustering

After generating text embeddings, the next step was to group similar features together to identify those that describe the same concept. However, two main challenges arose: (1) the optimal number of clusters was unknown, and (2) the high dimensionality of the embeddings made visualization difficult.

To address these challenges, DBSCAN (Density-Based Spatial Clustering of Applications with Noise) was chosen as the clustering algorithm. Unlike K-means, DBSCAN does not require specifying the number of clusters in advance. Instead, it groups densely packed points while marking points in low-density regions as outliers. The algorithm relies on three key parameters:

- **Epsilon (ε)**: Defines the radius within which points are considered neighbors.
- **min_samples**: Specifies the minimum number of points required to form a dense region.
- **Metric**: Determines the distance metric used to calculate the similarity between points.

For measuring similarity, cosine distance metric was chosen due to its suitability for high-dimensional data and its ability to effectively capture semantic relationships between features. Furthermore, its use is recommended in the OpenAI embeddings API documentation[3]. Finding the right parameters for ε and `min_samples` and was challenging. A high ε value led to a dominant cluster encompassing most features, while a low ε resulted in too many small clusters. Similarly, setting `min_samples` too low caused nearly every feature to form its own cluster, whereas a high value classified most features as outliers.

[3] `https://platform.openai.com/docs/guides/embeddings`

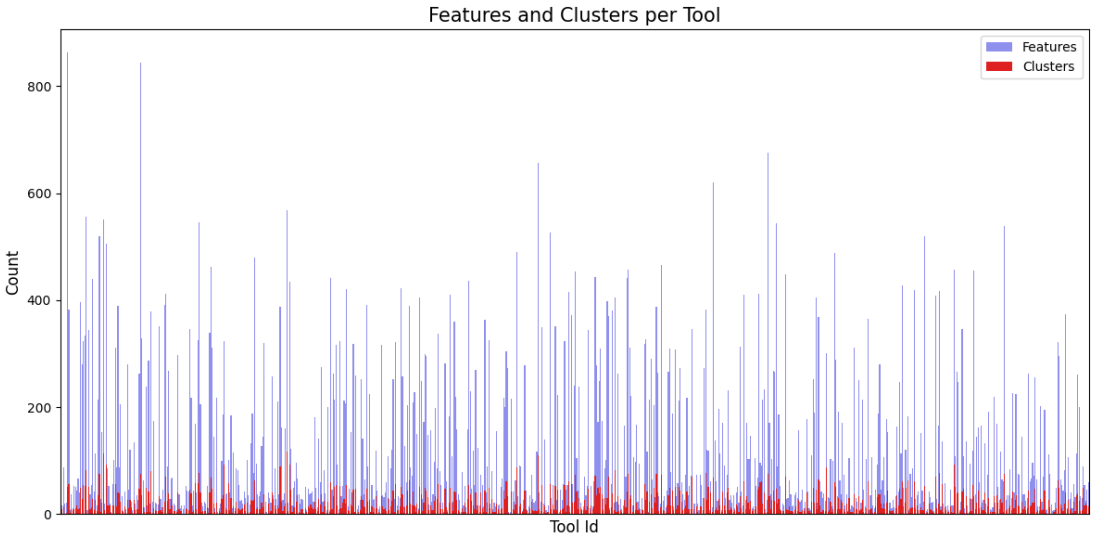

Figure 3.11: Number of Features per Tool vs. Number of Clusters per Tool

After several iterations, $\varepsilon = 0.15$ and $\mathbf{min\_samples} = \mathbf{2}$ were selected, resulting in a total of 8,332 clusters. If a tool contains a feature within a cluster, the cluster is then assigned to that tool. This process was repeated for all tools. The results are presented in figure 3.11, which compares the number of features per tool with the number of clusters per tool.

3.2.5 Redundancy Detection

Finally, the capabilities of the tools can be compared. The approach involves constructing a similarity matrix, not based on feature-to-feature comparison, but rather tool-to-tool, utilizing a similarity score derived from the number of shared clusters. The similarity score for each pair of tools I and J is calculated as follows:

$$similarity\_matrix[i,j] = \frac{clusters_j \cap clusters_j}{clusters_j} \tag{3.1}$$

This results in the statement that Tool I overlaps with Tool J on $\delta\%$ of their clusters. This metric serves as a strong indicator of redundancy and can be used not only to identify redundant tools but also to determine which tools a particular tool is redundant to. The similarity matrix is visualized in figure 3.12.

The results were subsequently stored in a file, where each tool is associated with a

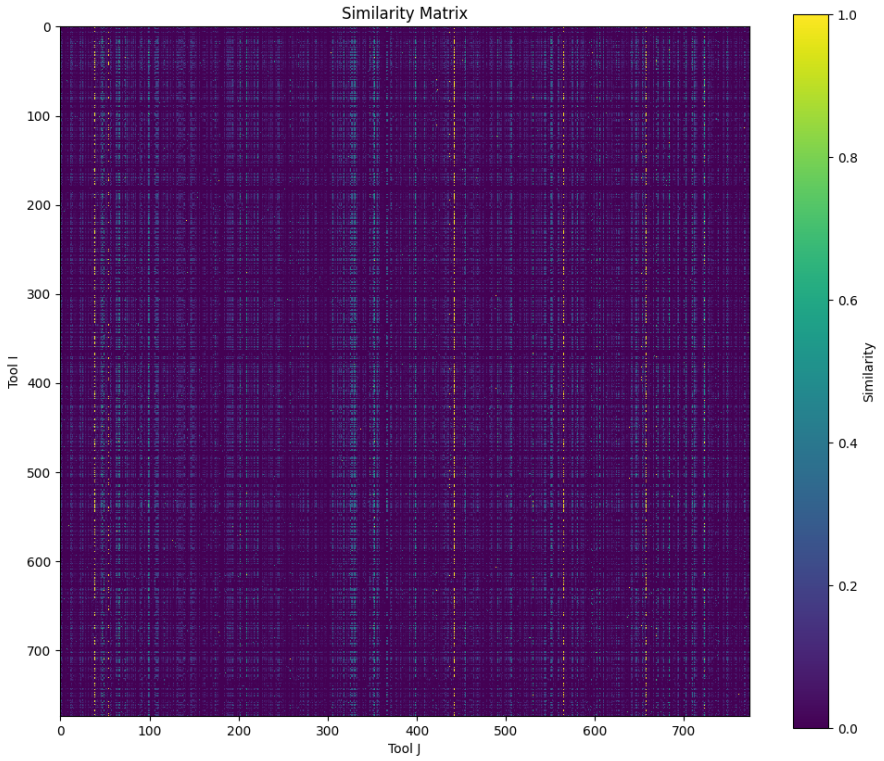

Figure 3.12: Similarity Matrix for Tools

list of similar tools that exceed a defined similarity threshold. This list includes the tool ID, the corresponding similarity score, and the IDs of the shared clusters.

3.3 API

To demonstrate the practical application of our tool redundancy detection system, we developed a lightweight REST API using FastAPI. This implementation serves as a reference for Softwaregini to integrate the algorithms into their software system.
The API follows a minimalist design with two core components: data models for organization tools and similar tools, and a single endpoint for redundancy analysis. The data models capture essential tool information including identifiers, names, descriptions, and

in-house status flags. The primary endpoint (/organization-tools) accepts lists of tools and returns potential redundancies. The workflow consists of two main steps: (1) tool matching against the database, and (2) similarity analysis using pre-computed scores. For demonstration purposes, similarity results are stored in JSON format, though production environments would typically use a database solution.
Organizations can analyze their software inventory by submitting their tool list through a POST request, receiving a response containing similarity scores and feature clusters for potentially redundant tools.

Chapter 4

Discussion & Learnings

4.1 Discussion/Learnings Challenge 1

As the matching algorithm progresses, several areas have been identified that could benefit from further improvements and refinements. The first issue relates to the presence of URLs in the manually provided input file. In some cases, the descriptions in the input data include links to the software's official website, which could potentially be leveraged to enhance the accuracy of the matching process. Unfortunately, these links were not incorporated into the current matching approach, which means an opportunity to gather more contextual information about the software was missed. In future iterations of the algorithm, it would be valuable to include functionality that allows the system to extract key information from these external links—such as version details, company names, or feature lists—which could help solidify or refine matches, particularly when there are vague descriptions in the input file.

Another consideration involves the presence of in-house developed software in the customer's inventory. In-house software, being proprietary and not publicly available, is often difficult to match using external databases like Softwaregini's, especially when there are no publicly accessible records or references to these tools. Currently, the algorithm lacks a mechanism to identify and handle such software. This limitation means that these tools are not matched properly and may remain unaccounted for.

Another key area for improvement is the overall testing and validation of the algorithm.

While the initial runs show promising results, more extensive testing is necessary to assess the robustness and quality of the matching process. Testing with a wider variety of datasets, including edge cases and unusual input, will help identify potential flaws and ensure that the algorithm performs consistently across different scenarios. Testing the system's ability to handle missing data, ambiguous descriptions, and various formats in both the customer's inventory and Softwaregini's database will be crucial. Furthermore, the incorporation of a detailed error reporting mechanism could help catch issues early in the process, leading to quicker resolution and better long-term performance.

4.2 Discussion/Learnings Challenge 2

The goal was to enable tool comparison with low computational overhead during runtime, which was successfully achieved. However, while the theoretical objective was achieved, it is difficult to assess the effectiveness of the results without manually labeling similar tools. This limitation also influenced the optimization of each step, as there was uncertainty regarding the expected outcomes. While it was possible to select a few example tools for comparison, this approach did not adequately reflect the success of the system as a whole. In the following sections, the specific challenges and lessons learned of/for each step are discussed.

4.2.1 Feature Extraction

A significant amount of time was spent refining the prompts used to extract tool features. The process was initially tested on a small set of hand-picked tools, and efforts were made to optimize the prompts accordingly. However, this approach proved to be time-consuming, and ultimately, it was difficult to know the full range of information sources for each tool. Given the large number of tools and diverse sources of information, it was not feasible to reliably adjust the prompts to fit the characteristics of "most" tools. This made it challenging to ensure the accuracy of feature extraction across the entire dataset.

Furthermore, the information sources themselves, primarily markdown strings, are not always structured and often contain a considerable amount of irrelevant content. Despite the capabilities of ChatGPT in processing text, this remained a difficult task. Issues were also encountered with website information, which included elements such as cookie

consent banners—something that was attempted to be filtered out using the prompts, though it is likely that other issues were not anticipated.

Additionally, each tool also recieved features provided by ChatGPT when based solely on its training data, without incorporating the markdown strings directly, which raises concerns about the reliability of the information. Although the implementation of a reliability score for each feature was considered, it proved to be too complex, particularly since each feature was provided by ChatGPT with varying contextual information(with or without markdown string). As a result, the process remains somewhat of a "black box," and the reliability of the features extracted is uncertain. Because this is the first step in the process, any inaccuracies or inconsistencies here will propagate through the rest of the steps.

4.2.2 Embeddings

Although embeddings are an excellent method for measuring semantic and grammatical similarity between texts, incorporating additional context, such as the specific tool to which a feature belongs, proved challenging without introducing unintended effects on the embeddings. Consequently, this approach was not implemented.

4.2.3 Clustering and Redundancy

As mentioned in 3.2.4, tuning the parameters for DBSCAN proved to be difficult. Furthermore, due to the uncertainty regarding the expected results, objectively assessing the effectiveness of the clustering was not feasible. Since the final redundancy detection depends on all preceding steps, particularly the clustering process and the assignment of clusters to each tool, evaluating the overall effectiveness of redundancy detection remains similarly challenging.

4.2.4 Conclusion & Outlook

To improve the results, an in-depth cluster analysis is needed. Examining which features belong to each cluster, their source tools, and categories. This could help refine feature extraction or clustering, potentially by first forming general clusters and then subclusters based on new insights.