Universidad Politécnica
de Madrid

Escuela Técnica Superior de Ingenieros Informáticos

Master in Data Science

Master Thesis

SONAD: An Approach for Automated Software Name Disambiguation in Scientific Publications

Author: Jelena Duric

Madrid, July, 2025

This Master Thesis has been deposited in ETSI Informáticos de la Universidad Politécnica de Madrid.

Master Thesis
Master in Data Science

Title: SONAD: An Approach for Automated Software Name Disambiguation in Scientific Publications
July, 2025

Author: Jelena Duric
Supervisor: Daniel Garijo Verdejo
Artificial Intelligence Department
ETSI Informáticos
Universidad Politécnica de Madrid

Summary

Accurate identification of software mentioned in academic papers and linking these references to their corresponding repositories is crucial to reproducibility, accurate citations, and efficient knowledge management. Unfortunately, tools are often referred to only by name, with no formal citation or additional metadata, making the task significantly more challenging. This lack of standardized referencing, combined with ambiguities in software naming conventions and frequent omission of repository links or version information, lowers the ability to reliably identify and connect software mentions to the correct repositories. This work addresses this challenge with a methodology to automatically disambiguate software mentions.

The research leveraged data extracted from scholarly publications and repositories available on GitHub, PyPI, and CRAN. The data included previously collected software mentions collected from a workshop from Chan Zuckerberg Initiative (CZI) as well as additional entries sampled from the CZI dataset. From scientific papers, relevant metadata such as the authors, keywords, software synonyms, and programming languages were extracted.

Candidate URLs were collected from two main package managers, PyPI and CRAN, as well as from GitHub, which served as the primary source. A variety of metadata fields and similarity techniques were then combined to train a supervised classification model that filters these candidates and predicts whether a candidate URL accurately corresponded to the given software mention.

Several models like LightGBM, XGBoost, Neural Nets, Logistic Regression, and Random Forest were tried, with many different similarity metrics to truly discover what metrics work best. Upon evaluation, best model yielded highly effective results, achieving a precision of 82%, a recall of 91%, and an F1 score of 86%, underscoring its ability to accurately resolve ambiguities in the software repository.

To contextualize these results further, comparative analyzes were performed using several large language models (LLMs), specifically, llama-3.1-8b-instant, qwen-qwq-32b, gemma2-9b-it and deepseek-r1-distill-llama-70b. Although LLM approaches showed high recall, they consistently delivered lower precision and F1 scores. In particular, the best-performing LLM, gemma2-9b-it, attained a precision of only 68%, a recall of 97%, and an F1 score of 80%. Thus, the explicit use of similarity metrics within a supervised learning framework outperformed precision of LLMs that rely purely on metadata without computed similarities.

The findings of this work highlight the effectiveness and importance of using explicit similarity calculations to resolve ambiguities in software mentions within academic texts. By outperforming contemporary LLM-based methods in precision, the

results demonstrate the strength of structured, metadata-driven supervised learning approaches. This methodology was successfully applied to correct and disambiguate 978 software mention names in the CZI dataset, showcasing its practical impact. Overall, the work contributes to more accurate software citations, improved reproducibility in scientific research, and better discoverability of software tools in scholarly ecosystems. The final model, along with all preprocessing steps, has been released as a Python package on PyPI, the full source code is available on GitHub[1], as well as on Zenodo[2].

[1] https://github.com/jelenadjuric01/Software-Disambiguation
[2] https://zenodo.org/records/15764860

Abstract

Accurate identification and linking of software mentioned in academic papers to their corresponding software repositories are critical for reproducibility, citation accuracy, and effective knowledge management. However, due to naming ambiguities and incomplete citations, accurately identifying the correct repository URL for software references can be challenging. This work addresses the problem of software mention disambiguation starting with experiments with different metadata that can be used in a context of software mentions, as software name, keywords, synonyms and authors of the paper of the mention, as well as different similarity measures, to at the end develop a supervised machine learning approach leveraging metadata extracted from scientific publications and candidate repositories from GitHub, PyPI, and CRAN. The approach involved collecting mentions of software from a previous CZI hackathon and additional sampling of CZI, extracting related metadata such as paper author names, keywords, synonyms of the software mention, and programming language from the surrounding paragraph, and retrieving candidate URLs from relevant repositories. Similarity measures between paper metadata and repository metadata—including string-based and embedding-based techniques—were computed and subsequently used as features in a supervised classification model to predict if the URL is the match. The evaluation of the model demonstrated a precision of 82%, a recall of 89%, and an F1 score of 85%, highlighting its effectiveness in accurately linking software mentions with the appropriate repositories. Comparative experiments with large language models (llama-3.1-8b-instant, qwen-qwq-32b, gemma2-9b-it and deepseek-r1-distill-llama-70b) that used raw metadata without similarity computations yielded lower precision and F1 scores, despite high recall. Specifically, the best-performing LLM (gemma2-9b-it) obtained a precision of 68%, recall of 97%, and an F1 score of 80%. Thus, the proposed similarity-based supervised approach outperformed all evaluated LLMs, emphasizing the effectiveness of leveraging explicit similarity metrics for accurate software disambiguation. These results demonstrate that metadata-based supervised approaches can effectively resolve software name ambiguities, improving scholarly communication and software discoverability in research contexts. The final product of this work can be found as a Python package PyPI, the full source code is available on GitHub[3], as well as on Zenodo[4].

[3] https://github.com/jelenadjuric01/Software-Disambiguation
[4] https://zenodo.org/records/15764860

Contents

Chapter 1

Introduction

Software has become a fundamental component in scientific research and significantly contributes to methodological rigor, reproducibility, and innovation [8]. Accurate identification and reference of software used in academic studies are essential to uphold these standards. Proper software citations facilitate reproducibility, enable accurate credit attribution, and support software discovery, thus strengthening the scientific ecosystem [8, 7].

However, accurately identifying software mentions within the scientific literature remains a complex challenge [7]. Issues such as naming ambiguities—multiple software packages sharing similar or identical names—and incomplete or inconsistent metadata in scholarly publications complicate the precise linkage of software mentions to the correct software repositories [1, 6]. This ambiguity leads to to incorrect associations, difficulty replicating studies [7], and reduced discoverability of essential research tools [8, 7].

The motivation for addressing this problem is clear: improving the reliability of software citation directly improves the quality and reproducibility of scientific research. Resolving ambiguities benefits researchers, publishers, software developers, and the broader scientific community, contributing to clearer communication, more efficient scientific workflows, and accurate recognition of software contributions.

Given this context, this work addresses the problem of software name disambiguation by introducing the following contributions:

- A comprehensive methodology for linking software mentions in scientific papers to their correct repository URLs.
- The creation of a gold standard corpus of manually validated mention–URL pairs.
- The development of SONAD, an end-to-end supervised pipeline that integrates metadata extraction, similarity computation, and classification. An extensive evaluation demonstrates the effectiveness of the pipeline in resolving ambiguities and improving the accuracy of software citation.

1.1 Objectives

The primary objective of this research is:

> Develop a methodology for disambiguating software names based on the context in which they appear, primarily the surrounding paragraph, by leveraging and analyzing a range of metadata and similarity strategies to accurately link mentions to their corresponding URLs.

Supporting objectives include:

- Develop a gold standard hand-validated corpus of software URL mappings to support research of this nature.
- Extract and process relevant metadata (authors, keywords, synonyms, programming languages) from academic papers.
- Calculate effective similarity metrics between publication-derived metadata and repository metadata.
- Evaluate the effectiveness of the proposed supervised approach in accurately linking software mentions, by comparing it to alternative existing methods.
- Design and implement an end-to-end pipeline that accurately disambiguates software mentions and links them to their corresponding repositories.

1.2 Work Structure

The remainder of this thesis is structured as follows:

Literature Review: Provides a comprehensive overview of existing methodologies and current state-of-the-art techniques in author name disambiguation, software entity disambiguation, and specifically software name disambiguation, highlighting both their strengths and limitations in addressing URL linking problems.

SONAD Methodology: Details the step-by-step methodological approach employed, including data collection, metadata extraction from scientific publications and repositories, candidate URL search procedures (GitHub, PyPI, CRAN), metadata retrieval from URLs, calculation of diverse similarity metrics (name, synonym, paragraph, keyword, programming language, and author similarities), supervised modeling strategies, and evaluation procedures.

Experiments and Results: Presents a sequence of experiments designed to incrementally enhance model performance. This section systematically explores baseline settings, the incorporation of synonyms, keyword extraction improvements, various similarity metric optimizations, candidate URL filtering strategies, class imbalance handling, and extensive hyperparameter tuning, culminating in the identification of the best-performing supervised model. Interpretative analyzes are included to illustrate model decisions using specific software mention examples, offering insights into how metadata and similarity measures influence URL disambiguation outcomes, as well as using SONAD to predict URLs from additional 806 CZI examples. Finally, the effectiveness of the developed supervised approach is evaluated through a comparative analysis against several existing methods, including contemporary large language

models (LLMs), with results demonstrating superior performance in terms of precision, recall, and F1 score.

Deployment and Package Release: Discusses the implementation details regarding the packaging, deployment, and distribution of the final model as an accessible Python package available on PyPI and GitHub.

Conclusions and Future Work: Summarizes the key findings, identifies limitations encountered, and proposes potential directions for future research, including dataset expansion, methodological refinements, integration of advanced embedding models, exploration of deep learning architectures, and expansion to additional repository platforms.

Bibliography: Lists all referenced sources, ensuring comprehensive coverage and acknowledgment of related work and supporting materials.

Chapter 2

Literature Review

Many variants of name disambiguation exist—covering legal records, social media identities, product names, places, and more—but our focus is concentrated on author name disambiguation and software name disambiguation [12, 13, 6, 11]. Software name disambiguation typically encompasses a multi-stage process that begins with software name extraction from text, followed by processing and potential disambiguation, and concludes with linking the software to its corresponding URL. The literature review presented below examines these comprehensive systems; however, our research objective is specifically focused on the linking phase—connecting already extracted software names to their URLs. Since we did not identify studies that isolate this particular approach in the existing literature, we have chosen to review it within the context of the broader disambiguation process.

2.1 Author Name Disambiguation

Recent advances in author name disambiguation include methods like WhoIs developed by Boukhers et al., which uses character-level name embeddings (Char2Vec) to capture orthographic similarities. These embeddings are combined with aggregated co-author network embeddings and topical features. A feed-forward neural network predicts if two records belong to the same author, followed by agglomerative clustering to form distinct author profiles. Evaluated on the large AMiner-WhoIsWho benchmark, this method achieved approximately 87–88% F1-score, significantly outperforming previous unsupervised methods [12]. Similarly, Liu et al. proposed ARCC, a model that starts with a heuristic coauthorship graph, refined through iterative pruning of low-confidence connections. A contrastive embedding model trained on the refined graph produced discriminative paper embeddings, leading to clustering in the embedding space. ARCC surpassed prior approaches, including WhoIs and S2AND, achieving an increase in F1-score across multiple benchmarks such as WhoIsWho and OAG [13].

2.2 Software Name Disambiguation

The approach presented by Istrate et al., employs a SciBERT-based Named Entity Recognition (NER) model fine-tuned on the SoftCite dataset. By processing millions

of articles from PubMed Central's Open Access subset and additional publisher corpora, the model extracts plain-text mentions of software with a high F1-score of 92%, thereby yielding millions of candidate software mentions [6]. Following extraction, Istrate et al. then construct a synonym graph to cluster together variant mentions. In this pipeline, domain-specific heuristics are used to generate candidate synonyms from sources such as SciCrunch. String similarity is computed via metrics including the Jaro–Winkler distance [10] and the algorithm's clustering step relies on DBSCAN [9] to partition the mentions into clusters, with each cluster ideally corresponding to one canonical software entity. The resulting clusters – numbering approximately 97,600 across the corpus – represent distinct software tools that have been "disambiguated" from the raw mention variants [6]. Once clusters are established, each one is linked to external repositories (e.g., GitHub, PyPI, CRAN, Bioconductor, and SciCrunch). This linking is accomplished by executing exact and fuzzy string searches against the repository endpoints. A second fuzzy layer utilizing Jaro–Winkler similarity "rescues" misspellings and acronymic variants; meanwhile, repository-specific keywords (such as "R package" or "Bioconductor") help ensure that queries are directed to the correct index. Returned URLs are then filtered heuristically by comparing repository metadata (e.g., declared language, maintainer name, license) with the mention context. If several repository candidates remain after filtering, a best-match technique is employed – for instance, selecting the URL that exactly matches the cluster label. While this unsupervised heuristic approach enables the pipeline to produce 185,427 mention-URL pairs that cover 55.8% of software-paper links, its overall correctness is only 54% when measured across all links; however,when considering the evaluation only of links retrieved through PyPI, Bioconductor, CRAN and SciCrunch, the accuracy improves considerably: 93.33% [6].

The SoMeSci project [11] represents a comprehensive effort to create the most detailed knowledge graph of software mentions in scientific literature, addressing a critical gap in understanding how software is cited and used in research. The authors manually annotated 3,756 software mentions across 1,367 PubMed Central articles, creating a gold standard dataset that goes far beyond simple software name identification. Their annotation scheme captures software types (applications, plugins, operating systems, programming environments), mention types (usage, creation, allusion, deposition), and crucially, associated metadata including versions, developers, citations, and URLs.To support Named Entity Recognition (NER), Relation Extraction (RE), and Entity Disambiguation (ED), they employed a Bi-LSTM-CRF model for NER and a Random Forest classifier for RE, achieving high precision (F1=0.83 for NER, 0.88 for RE). For disambiguation, they used hierarchical clustering with features like Levenshtein distance and substring matching, achieving F1=0.97 for linking mentions to unique entities[11]. After the initial textual annotation, the team performed manual entity disambiguation where software, developer, citation, and license entities were linked to unique identifiers by four annotators working in parallel. For URL linking specifically, they established a clear priority hierarchy: WikiData first, then package repositories (CRAN, PyPI), GitHub, and finally official websites. When initial inter-rater reliability for disambiguation proved insufficient at only 68% agreement, they implemented a rigorous quality control process where two annotators independently worked on the entire dataset, with results then merged to ensure canonical link formats. This meticulous approach successfully linked 615 out of 637 unique software entities to external URLs, with only 22 remaining unlinked due to challenges like resource locations changing over time, paywalled information, or

obsolete software lacking persistent identifiers.

2.3 State-of-the-art Summary

The current state-of-the-art in software name disambiguation reveals significant limitations that constrain both scalability and accuracy in real-world applications. Existing approaches predominantly rely on unsupervised heuristic methods rather than sophisticated machine learning techniques, resulting in fundamental bottlenecks for automated processing of large-scale scientific literature.

The most prominent limitation lies in the heavy dependence on manual processes and rule-based systems. While Istrate et al. approach demonstrates reasonable coverage by producing 185,427 mention-URL pairs, its overall correctness remains low at 54% across all links, highlighting the inadequacy of purely heuristic matching strategies. Their system relies extensively on hand-crafted rules, exact string matching, and basic fuzzy search techniques that fail to capture the nuanced relationships between software mentions and their corresponding repositories. The improved accuracy of 93.33% achieved on specific repositories (PyPI, Bioconductor, CRAN, and SciCrunch) suggests that repository-specific heuristics can work well in constrained domains, but this approach does not generalize effectively across the diverse landscape of software repositories and mention contexts.

More critically, the SoMeSci project, despite achieving impressive disambiguation results (F1=0.97), fundamentally relies on intensive manual annotation and human verification processes that are inherently non-scalable. Their approach required four annotators working in parallel with multiple quality control iterations, and even then, they could only process 3,756 software mentions across 1,367 articles. This manual bottleneck makes it practically impossible to extend such methods to the millions of scientific articles published annually, severely limiting the real-world applicability of their otherwise high-quality approach.

The absence of trained and supervised models for the ranking of candidate URLs represents another critical gap in current methodologies. Existing systems lack the ability to automatically learn from historical linking patterns, contextual metadata, or semantic relationships between software mentions and repositories. Instead, they rely on simplistic best-match techniques and threshold-based filtering that cannot adapt to the evolving nature of software ecosystems or handle ambiguous cases effectively.

These existing approaches highlight critical gaps and opportunities for advancement. In particular, integrating supervised learning methodologies to automatically classify candidate URLs based on diverse metadata and contextual information could significantly improve both precision and recall. Applying supervised learning and embedding-based techniques, as seen in successful author name disambiguation methods, can further enhance the capability of software disambiguation systems. Thus, developing robust supervised frameworks that leverage detailed metadata embeddings to link software mentions accurately to their corresponding URLs represents the key area for future research and methodological improvement in software name disambiguation.

Chapter 3

SONAD

3.1 Methodology

The methodology applied in this work involved collecting software mentions from datasets previously compiled from Chan Zuckerberg Initiative (CZI)[2] hackathon, augmented with additional data from CZI. Relevant metadata was systematically extracted from the textual context surrounding each software mention. Candidate URLs from the GitHub, PyPI and CRAN repositories were searched with both exact and fuzzy search, and various similarity metrics, both string-based and semantic embedding-based, were computed. Subsequently, these metrics served as input features for training, tuning, and evaluating a supervised classification model. The evaluation of the model led to new experiments exploring alternative metadata sources, candidate URL retrieval methods, and similarity calculation approaches (see Figure 3.1). The final pipeline was then deployed and published on PyPI[3]. The entire research and the package itself were coded in Python and are open-source on GitHub [4], as well as on Zenodo [5].

3.1.1 Relevant Data

The training corpus for our model was constructed using software mentions extracted from a diverse set of scientific publications. Initially, we identified several key pieces of information as essential for each mention: the software name as it appears verbatim in the paper, the DOI of the publication, and the paragraph in which the software is mentioned. These core elements served as the foundation for enriching the dataset with additional metadata relevant to the task of software name disambiguation.

From the surrounding paragraph, we extracted keywords to better understand the context in which the software is used—this helps distinguish between tools with similar names but different functions. We also gathered software synonyms, which are particularly useful in cases where a tool is referred to using alternative names or abbreviations, a common source of ambiguity. Furthermore, the programming language associated with the software was included, as it helps narrow down the correct platform (e.g., PyPI for Python, CRAN for R). Finally, we considered the authors of the paper. Although not always applicable, in cases where the developers of the software are also the authors of the publication, this information provides an additional signal that can support correct identification and linking.

3.1.2 Data Collection

Dedicated corpus of software mentions and their contextual metadata was constructed through a multistage process:

1. **Source aggregation**
   A previously curated benchmark dataset developed for the CZI organized hackathon [1] served as the foundation. Additional examples were drawn from the CZI linked dataset [2], which consolidates references from GitHub, PyPI and CRAN. From both sources, the main columns collected for each software mention were the software name, the DOI of the corresponding paper, and the paragraph in which the mention appeared. This information was already directly available in the hackathon dataset [1]. In contrast, the CZI linked dataset [2] provided software mentions in separate files, categorized by the type of platform they were linked to: GitHub, PyPI, or CRAN. Due to this structure, and considering the manual effort required to verify the correctness of each URL, it was not feasible to utilize the entire dataset. Therefore, a sampling strategy was applied to select a manageable subset of mentions that could be manually verified, while still ensuring diversity across platforms and preserving the overall relevance of the dataset for training and evaluation.

2. **Sampling strategy**
   To ensure both breadth and practical relevance, two complementary sampling approaches were employed. Random sampling captured a statistically unbiased cross section of software mentions, reducing the risk of over representing any particular technology domain. Popularity-aware sampling specifically targeted the most widely used Python and R packages, as determined by download statistics and repository activity. Although these packages are well-known within their respective communities, their popularity does not eliminate ambiguity. On the contrary, widely used tools are often referred to in various ways—abbreviated, mentioned without version numbers, or described in different contextual phrasings—which can make disambiguation nontrivial. Furthermore, tools with common names (e.g., `pandas` or `flask`) may appear in unrelated contexts or be confused with similarly named software. Including these high-visibility tools ensures that the corpus captures the kinds of real-world challenges models will face when applied to scholarly literature. For Python, we fetched mentions of the following popular packages: `numpy`, `tensorflow`, `scikit-learn`, `pandas`, `matplotlib`, `requests`, `beautifulsoup4`, `flask`, `django`, and `pytorch`. For R, we fetched mentions of: `ggplot2`, `dplyr`, `data.table`, `tidyr`, `readr`, `stringr`, `lubridate`, `shiny`, `rmarkdown`, and `knitr`. Including these high-visibility tools ensures that the corpus reflects the software researchers most frequently cite, thus improving the applicability of the model to the real-world literature.

3. **Corpus assembly and cleaning**
   Each record in the corpus includes the software name (as it appears in the source text), the DOI of the citing paper, and the paragraph surrounding the mention. All records were manually reviewed to filter out false positives—instances where the mention did not actually refer to a software package. For example, while the word “text” which exists as a package, was often used in a generic sense (e.g., referring to a group of letters) rather than indicating a specific soft-

ware tool. Such ambiguous cases were especially common in the CZI GitHub dataset, highlighting the need for careful manual verification to ensure the quality and accuracy of the training data.

4. **Ground-truth enrichment**
   For every valid mention, the corresponding authoritative URL was manually located and verified. This step standardized references to canonical project pages, archives, or documentation sites.

The final corpus contains 682 unique software mentions that span various research domains and usage contexts, and is now ready for downstream preprocessing and model training. It can be found on Zenodo [5].

3.1.3 Extracting Software Metadata

For our experiments, we extracted multiple layers of metadata linking each software mention to its source paper and the specific paragraph context:

1. **Author Extraction**
   We retrieved author names by querying the OpenAlex API with the DOI of each paper. OpenAlex is a comprehensive, open-source index of scholarly publications, authors, institutions, and related entities, designed to serve as a modern replacement for Microsoft Academic Graph [15]. It provides structured metadata and supports flexible API queries. For each DOI, we sent a GET request to:

   `https://api.openalex.org/works/https://doi.org/{doi}`

   and parsed the `authorships` array to collect the `display_name`, defaulting to an empty list if no authorship data was returned.

2. **Keyword Extraction**
   We generated two complementary keyword sets per mention:

   - *Paper-level keywords*: fetched via OpenAlex and filtered to include only those with a relevance score > 0.5, ensuring we captured the most significant descriptors.
   - *Paragraph-level keywords*: extracted directly from the paragraph containing the software mention using the RAKE algorithm, limiting to the top five phrases by keyword score. RAKE (Rapid Automatic Keyword Extraction) is an unsupervised, domain-independent algorithm that identifies key phrases in a body of text by analyzing word frequency and co-occurrence patterns, then scoring candidate phrases based on the degree to which words appear together versus separately [16].

3. **Synonym Aggregation**
   To account for divergent naming conventions (e.g. “scikit-learn” vs. “sklearn”), we ingested a CZI [2] provided DataFrame of manually curated synonym pairs. For each lowercase software key, we searched all rows where `software_mention` matched that key and collected the corresponding `synonym` entries in a `set`. Once populated, this dictionary of mention→synonym sets was serialized to JSON, allowing rapid cache reads during large-scale batch processing.

4. **Programming Language Identification**
   When possible, we inferred the programming language from the text surrounding the mention. We maintain a list of tokens in common language (e.g. 'Python', 'R', 'Java') and scan the paragraph for these terms, recording their character spans. We then computed the midpoint of the software mention and selected the language token whose midpoint lay closest to it. If no language token was detected, the field remained empty.

3.1.4 Searching for Candidate URLs

To build a ranked list of potential landing pages for each software mention, we drive three search strategies, one per platform, each tuned to the conventions and metadata available on that service. The three streams of results are then merged and globally re-ranked before light post-processing.

3.1.4.1 GitHub Repository Search

We query the GitHub search API for repositories whose names contain the mention string. By restricting the search field to 'name' and sorting by star count, we first surface the most popular projects, under the assumption that higher-starred repos are more likely to correspond to widely used tools. We limit the returned page to the top N repositories (in our case 5), paginating as needed, and automatically back off and retry if we hit rate limit responses.

3.1.4.2 PyPI Lookup and Fuzzy Search

Our first attempt on PyPI is a direct JSON lookup of the mention as a package name; if found, the canonical project URL is taken immediately. Should that fail, we fall back to an XML-RPC name search, which returns a list of candidate names scored by text match. We fetch the top hits (up to 5), filter out exact-name duplicates, and retrieve each package's project URL via the JSON API. Within this fuzzy-search stream we rank first by reported relevance score, then by download statistics when available, so that more commonly downloaded packages bubble up.

3.1.4.3 CRAN Search

For R packages, we download the master CRAN PACKAGES index and parse all registered names. We then perform three matching passes in order: exact name, substring containment, and finally close string similarity (using a Jaro-Winkler cutoff of 0.6). Each stage contributes up to N candidates, ordered by match type (exact > substring > fuzzy) and within fuzzy by similarity score. Drawing from the same up-to-date index for each run, we ensure comprehensive coverage of both newly released and long-standing R packages.

3.1.4.4 Final Processing

Once the top candidates of each platform have been retrieved and internally ordered according to their native popularity or similarity metrics (stars for GitHub, relevance / downloads for PyPI, distance between matches for CRAN), we merge the three lists into one global slate.

Finally, we apply a brief postprocessing pass to normalize URL schemes and hosts, collapse duplicates, and discard any non-HTML or purely documentation endpoints (via lightweight HTTP HEAD checks). The curated, ordered URLs that remain form the candidate set forwarded to downstream metadata extraction and similarity scoring.

3.1.4.5 Additional Candidates

In the final experiments, we refined the candidate URL set by incorporating GitHub links referenced in the CRAN and PyPI metadata and by appending each CRAN-derived URL with its corresponding GitHub repository from the CRAN-to-GitHub mirror maintained by CRAN. To reduce noise in the candidate set, we discarded PyPI packages with descriptions shorter than 300–400 characters and GitHub repositories with an empty README file, except when the repository owner was "cran". We considered such URLs to be unlikely to represent the correct software, as legitimate and widely used packages typically include meaningful descriptions on PyPI and provide at least minimal documentation in the form of a README file. Their absence was interpreted as a strong signal that the repository or package was inactive, incomplete, or unrelated to the actual mention of the software.

3.1.5 Fetching Corresponding Metadata from URLs

For each candidate URL, we invoke a domain-specific extractor that produces a unified metadata dictionary entry, which is then persisted in a JSON store to enable efficient downstream processing. Based on our experimental findings, we evaluated multiple retrieval strategies for certain repositories to optimize metadata coverage and quality. The following sections provide an overview for all approaches.

3.1.5.1 GitHub

All GitHub URLs are first normalized to the form `https://github.com/{owner}/{repo}`. The metadata is then retrieved by invoking the SOMEF CLI[14] through:

```
run somef describe
```

This command is executed twice, first normally and in failure with the `-kt` temporary directory option, to produce JSON output, with a threshold of 0.93 (used to classify the text). From this output, we extract:

- **Name** and **description**
- **Keywords**: comma-split list, with an optional RAKE-based fallback applied to the description, if keywords were empty
- **Authors**: repo owner's display name (queried via the GitHub Users API)
- **Language**: determined by the largest code-size language entry

An alternate extractor variant also retrieves the README URL and flags whether the README is empty.

3.1.5.2 PyPI

The package name is parsed from the URL path and then queried from the PyPI JSON API. The basic extractor captures the following:

- **Name** and **description**
- **Trove classifiers**, filtered into keywords
- **Authors**: “author”/“maintainer” fields, split on commas, “and”, or semicolons
- **Language**: set to “Python”

To enhance keyword coverage, three RAKE-based augmentation strategies that were applied on the description were evaluated:

1. *RAKE-only*: always apply RAKE when JSON keywords are missing
2. *RAKE+Classifier fallback*: attempt JSON keywords, then RAKE, then Trove classifiers
3. *RAKE-after*: apply RAKE only if both JSON keywords and classifiers are absent

3.1.5.3 CRAN

For CRAN packages, the extractor identifies the package name from either the CRANDB API, where we extract:

- **Name**, **description**, and basic **author information**
- **Language:** set to "R"
- **Keywords**: sourced from the DESCRIPTION file's `Keywords` field; if none are declared, RAKE is applied to the long description to extract up to five representative phrases

3.1.6 Calculating Similarities Between Paper and Candidate URL Metadata

To quantify the degree of match between each software mention in the paper and its corresponding candidate metadata, we define six complementary similarity metrics: name, synonym, paragraph–description, keyword, programming language, and author similarity. These metrics were chosen to reflect both the lexical and semantic relationships between the mention context and metadata attributes. Additional reasons for the choice of each metric were explained in respective subsections (see Subsections 3.1.6.1, 3.1.6.2, 3.1.6.6, 3.1.6.4, 3.1.6.5, 3.1.6.3). For each type of similarity, we considered multiple possible approaches - balancing precision, robustness to incomplete data, and computational efficiency - but ultimately limited our implementation to one or two techniques per type due to time and resource constraints.

For example, name and synonym similarities may have been calculated using a wider range of string distance measures such as cosine similarity over character n-grams or token-based methods like SoftTF-IDF. Paragraph–description and keyword similarities could have leveraged more complex transformer models or domain-adapted embeddings, while author comparison could have involved named entity matching or ORCID disambiguation. However, after initial testing, we prioritized widely used and interpretable methods such as Levenshtein distance, Jaro–Winkler similarity, and SBERT-based embeddings, which provided a practical trade-off between performance and scalability for our setting.

3.1.6.1 Name similarity

Both normalized Levenshtein [17] distance and Jaro–Winkler [10] string similarity were implemented on 'cleaned' names (lowercased, stripped of version tokens, punctuation and common affixes). The Levenshtein-based score (distance-to-similarity conversion) was chosen for its ability to capture fine-grained edit differences between software names, such as insertions, deletions, and substitutions. This is particularly useful when names are similar but contain minor spelling errors or variations, which is common in software naming conventions (e.g., "pandas" vs. "panda"). On the other hand, Jaro–Winkler similarity was selected for its sensitivity to transposed or aliased tokens, which can often occur in software names, especially when abbreviations or common synonyms are used (e.g., "tensorflow" vs. "tf"). The Jaro–Winkler metric is more forgiving of small changes in word order and is better suited for cases where similar software names may be used interchangeably or where the characters are in slightly different positions, which are frequent occurrences in large software repositories and academic references. Although Jaro–Winkler was previously used for linking software mentions with URLs in approach applied by Istrate et al.[6], it was applied solely in the context of exact and fuzzy matching using predefined thresholds. The key difference in our approach is that we integrate Jaro–Winkler as a feature within a supervised machine learning model, allowing it to contribute to a learned decision function rather than relying on fixed similarity cutoffs.

3.1.6.2 Synonym similarity

To take advantage of known alias sets, we computed the average pairwise similarity between the normalized repository name and each synonym entry. Two variants were tested: Levenshtein-based normalized similarity [17] and Jaro–Winkler averaging [10] for the same reasons they were used for name similarity.

3.1.6.3 Paragraph–description similarity

We measured the semantic closeness between the paper paragraph and the candidate's description using transformer-based sentence embeddings. Initially, we experimented with the RoBERTa-large model (`all-roberta-large-v1`) due to its strong performance in semantic similarity tasks and its ability to capture rich contextual relationships between words. However, we found that its high computational cost made it impractical for large-scale evaluation. As a result, we adopted a more lightweight alternative—SBERT (`all-MiniLM-L6-v2`)—which produces competitive semantic embeddings at a fraction of runtime. Although RoBERTa may offer slightly higher accuracy in specific cases, SBERT provided a better balance between performance and efficiency, making it more suitable for our pipeline, where many comparisons needed to be computed quickly.

3.1.6.4 Keyword similarity

For metadata keyword fields, we compared comma-separated keyword lists extracted from the paper and from the candidate site using two main strategies. The primary method involved embedding both keyword lists using a lightweight BERT-based model and computing cosine similarity between them. This approach was selected because keywords typically consist of short, discrete phrases where sentence-level

embeddings are less effective, and a token-level model like BERT can capture important term-level distinctions efficiently.

In cases where the candidate metadata did not include keywords—a common issue with certain repositories—we fell back to comparing the paper's keyword list against the full textual description of the candidate using a more robust transformer model, RoBERTa. This fallback ensured that similarity could still be estimated, even in the absence of structured metadata.

As an alternative approach, we also evaluated SBERT-based embeddings. While SBERT is primarily designed for sentence-level similarity, we considered it as a flexible option capable of capturing deeper semantic relations even in sparse or loosely structured keyword inputs. Its inclusion allowed us to explore how well sentence-level models generalize to this more compact form of data and to compare results across different embedding paradigms.

3.1.6.5 Programming-language similarity

We normalize language names (lowercase, trimmed) and applied Jaro–Winkler similarity. This simple string-based approach proved sufficient to distinguish identical versus differing language mentions.

3.1.6.6 Author similarity

The author fields—when present—are normalized by converting names to a consistent first-last format, expanding initials, and removing non-letter characters. After normalization, we compare author lists using Jaro–Winkler similarity. This metric was chosen because it is particularly effective for short string comparisons and is sensitive to differences near the beginning of the strings, which is important in author names where the first and last names carry the most distinguishing information. Additionally, Jaro–Winkler can handle common variations such as reordered names, abbreviated initials, or minor spelling differences, making it a practical choice for detecting potential author overlaps between the paper and the candidate repository metadata.

Each metric produces a numeric score: cosine-based and transformer embedding similarities lie in $[-1, 1]$, while Levenshtein and Jaro-Winkler measures lie in $[0, 1]$; any missing inputs yield NaN. Together, these six scores form the set of characteristics for downstream classification and classification.

3.1.7 Modeling

We frame the mention–URL matching task as a binary classification problem: Given the similarity metrics for each candidate, predict whether it is a 'match' or 'non-match'. In our initial experiments, the dataset exhibited a 3:1 imbalance in favor of the non-match class; after augmenting with additional candidate URLs, this skew increased to roughly 6:1. To assess the impact of imbalance mitigation, we trained models both with and without class-weighting (or equivalent rescaling) strategies and used stratified cross validation. To explore the trade-offs between interpretability, robustness, and predictive power, we trained and compared five supervised classifiers.

- **Logistic Regression[18]:** Used as a simple, interpretable baseline for binary classification. We used the `liblinear` solver with an ℓ_2 penalty, which is effective for small to medium datasets. Both unweighted and class-balanced versions (`class_weight = 'balanced'`) were tested to assess the robustness to class imbalance.

- **Random Forest[19]:** A tree-based ensemble model that captures non-linear relationships and feature interactions, making it a strong candidate for structured tabular data. It consists of 100 decision trees with unrestricted depth, leveraging all CPU cores for parallelism. We included both standard and `class_weight='balanced'` configurations to account for the skew in class distribution. Its ability to handle missing values and resist overfitting on small datasets also made it suitable for our problem.

- **XGBoost[20]:** Gradient-boosted decision trees optimized for tabular data, selected for their strong performance on many classification benchmarks. XGBoost handles missing values internally and offers fine-grained control over the complexity of the model. We used 100 estimators with logistic loss for binary classification, and handled class imbalance by tuning `scale_pos_weight` based on the ratio of negative to positive samples. This makes XGBoost particularly suitable when positive examples are underrepresented.

- **LightGBM[21]:** A highly efficient gradient boosting framework designed for fast computation and low memory usage. LightGBM is well-suited for numeric features and is capable of handling large datasets and imbalanced classes efficiently. We used 100 estimators with full parallelization, and computed class weights using `compute_class_weight` to reflect the true distribution of positive vs. negative samples. LightGBM was included as a competitive alternative to XGBoost, offering comparable accuracy with shorter training times.

- **Neural Network (MLP)[22]:** A multi-layer perceptron with hidden layers of sizes (50, 30, 10), designed to capture complex, non-linear interactions among similarity features. ReLU activations and the Adam optimizer were used, with training capped at 200 iterations. Neural networks are especially flexible for modeling abstract feature spaces and can generalize well given enough examples. To address class imbalance, we applied per-class weighting in the binary cross-entropy loss, allowing the model to assign higher penalty to misclassified positive examples.

3.1.7.1 Feature Correlation Analysis

Prior to model fitting, we computed a dense correlation matrix over all six similarity metrics (name, synonym, keywords, paragraph, language, author) and the true-match label (see Figure 3.2). Most pairwise correlations among the core metrics remain modest (below 0.40), indicating that each captures largely distinct information. Two notable exceptions are the name–synonym pair (0.75), reflecting that exact name variants and curated aliases often coincide, although experiments showed that both are necessary, and the keywords–paragraph pair (0.65), since both derive from overlapping textual context. All other metric–metric correlations lie below 0.40, and the label itself shows its strongest association with paragraph similarity (0.40) and language similarity (0.41). These results justify feeding the raw feature set into

tree-based models without aggressive dimensionality reduction, while also suggesting that in a lightweight linear model one might combine or omit one of each highly correlated pair (e.g. name/synonym or keywords/paragraph) to simplify the feature space. Considering that metadata was fetched using different approaches and that many similarity measures were utilized, the matrix was calculated several times, but the results were very similar every time.

3.1.7.2 Preprocessing

For tree-based models (XGBoost and LightGBM), we feed raw metric values directly, since these algorithms are invariant to monotonic feature transformations. For Logistic Regression, Random Forest and Neural Network, we apply a preprocessing pipeline that (i) imputes missing values with column medians and adds binary "_missing" indicators, then (ii) standardizes each numeric feature to zero mean and unit variance.

3.1.7.3 Training and Evaluation

We split our dataset stratified by the true-match label into an 80% train+validation set and a 20% held-out test set. This percentages were chosen because when our corpus was expanded to mention-URL pairs, we had about 6500 rows, which is a moderate amount, appropriate for 80/20 split. Within the train+validation portion, we perform 5-fold stratified cross-validation to tune each model and estimate out-of-fold performance, reporting precision, recall, and F1-score. Finally, we retrain each classifier on the full 80% and evaluate on the unseen 20%, ensuring that our results reflect generalization to novel mention–URL pairs.

3.1.7.4 Feature Selection

To assess the value of more compact feature subsets, we conducted two types of feature selection: (i) univariate selection using ANOVA F-scores and (ii) multivariate selection using feature importances from a Random Forest classifier.

In the univariate approach, each feature is scored independently using an F-statistic that measures how well it distinguishes between the two classes. Higher F-scores indicate stronger individual predictive power, meaning that features with higher values are more relevant for classification.In the multivariate approach, Random Forests rank features based on how frequently and effectively they are used to split decision trees across the ensemble. Higher importance values suggest that the feature plays a more significant role in the model's decision-making, especially when interacting with other features. Both methods aim to identify a minimal set of features that retains most of the predictive information, which can be useful for building faster or more interpretable models. In either case, features with higher scores are preferred, while those with near-zero scores can often be removed with minimal loss in performance.

3.1.7.5 Fine Tuning

A systematic hyperparameter tuning procedure was implemented to rigorously optimize each classification model. The full dataset of feature–label pairs was first partitioned using stratified splitting, ensuring that both the training and testing sets

preserved the original class distribution. In this case, the final testing set was balanced, allowing for an unbiased evaluation of each model's performance across both classes (i.e., correct and incorrect URLs).

Within the training portion (referred to as the selection set), we performed grid search over a range of algorithm-specific hyperparameters, including tree depths, regularization strengths, and learning rates. To avoid introducing bias and to ensure that each model was exposed to a representative class distribution during tuning, we applied 5-fold stratified cross-validation. This approach maintained consistent class ratios across all folds, which is particularly important given the moderate class imbalance present in the original dataset.

Model performance was assessed using three key metrics: precision, recall, and F_1 score. Precision measures the proportion of predicted "correct" URLs that are actually correct—critical when false positives (e.g., linking to the wrong repository) could mislead users or systems. Recall captures how many of the truly correct URLs were successfully identified—important when omission of valid links could result in broken references or lost functionality. Since models that focus too much on one of these metrics may perform poorly on the other, the F_1 score was used as a balanced metric, reflecting the harmonic mean of precision and recall.

The configuration that achieved the highest average F_1 score during cross-validation was selected, and the final model was retrained on the entire selection set using those optimal parameters. Performance was then evaluated on the independent, balanced hold-out test set, providing an unbiased estimate of the model's ability to generalize. All results were recorded and compared to assess the overall effectiveness of each classification approach.

3.2 Experiments and Results

A total of nineteen experiments were conducted, each informed by the results of previous ones. Successive trials introduced new hypotheses regarding the incorporation of specific metadata types or similarity metrics, with the goal of incrementally improving model performance. All test-set results are summarized in the tables. Across all models and feature configurations, the gap between training and test performance never exceeded 3%, indicating minimal overfitting. Unless otherwise specified, the reported results use the full feature set (including missing-value indicators where applicable). Although we tracked univariate and multivariate subsets during experimentation, those configurations did not yield additional insights beyond what is already presented and thus are omitted here for clarity, but can be seen in the results folder available on Zenodo [5].

3.2.1 Baseline

In our baseline experiment, we paired the features extracted from the document, namely the software name, the keywords of the paper, the surrounding paragraph, any detected tokens of the programming language and the authors of the document, with the corresponding metadata recovered from each candidate URL (name, keywords, description, authorship). For name, language, and author similarity we employed the Jaro–Winkler metric as detailed in the Methodology (see Section 3.1). Key-

word similarity was calculated by first encoding both keyword sets with a BERT-based model; whenever URL-side keywords were unavailable, we instead compared the paper's keywords against the full URL description using a RoBERTa encoder. The similarity between the paragraph and the description also relied on the RoBERTa embeddings. All models used class weighted approach. This configuration yielded strong initial performance, validating our multimodal approach to feature matching (see Table 3.1).

A deeper analysis of the contribution of the characteristics reveals that the name similarity metric is the strongest predictor, accounting for roughly half of the total importance of the model and achieving the highest univariate score (972.5, $p \ll 0$). Although name similarity provides the most decisive signal overall, particularly when the names differ slightly, it becomes less useful in cases where the names are identical, which is often the true nature of disambiguation. In such instances, the model must rely on other features, primarily the contextual paragraph and the associated programming language, to differentiate between candidate URLs. The paragraph similarity metric emerges as the second most powerful signal, although its share of importance drops from a univariate score of 923.4 ($p \ll 0$) to about 16.8% in the multivariate setting when conditioned on name similarity. Keyword and language similarities contribute more modestly, approximately 10.8% and 5.7% respectively, together representing approximately 16.5% of the explanatory power, indicating partial overlap with the dominant name and paragraph signals. Author similarity, while barely significant in univariate testing (score 12.2, $p \approx 4.9 \times 10^{-4}$), still contributes about 8.5% in the multivariate model; notably, the absence of author metadata (missing-author flag) proves even more informative at roughly 2.9%. Finally, missing value indicators for paragraph, keywords, and language, despite their high univariate significance, each account for only 1–3% of overall importance. These results underscore that, although our multimodal design leverages diverse signals, the software name remains the dominant predictor, and in cases where the name alone is insufficient, such as when multiple candidates share it, the paragraph and language signals are most critical for accurate disambiguation.

Table 3.1: Test Precision, Recall, and F_1 Scores for Baseline Experiment

Classifier	**Precision %**	**Recall %**	**F_1 score %**
Logistic Regression	63	92	75
Random Forest	86	88	87
XGBoost	84	92	88
LightGBM	81	94	87
Neural Network	84	87	86

3.2.2 Adding Synonyms

In the next phase of our analysis, we incorporated a synonym similarity feature to assess whether semantic expansions could further improve matching accuracy. Synonyms were recovered from the Chan Zuckerberg Initiative (CZI) and pairwise similarity between the paper and URL terms was measured using the Jaro–Winkler distance. Adding the synonyms yielded a consistent performance boost across all candidate models—with the sole exception of the neural network—despite the fact that this feature did not rank highly in either the univariate or multivariate importance analyses

(see Table 3.2). Considering that it did give better results, synonyms were included from this point into the next experiments.

Table 3.2: Test Precision, Recall, and F_1 Scores for Added Synonyms Experiment

Classifier	Precision %	Recall %	F_1 score %
Logistic Regression	64	93	76
Random Forest	85	89	87
XGBoost	84	94	89
LightGBM	84	94	89
Neural Network	83	88	85

3.2.3 Paper Keywords Obtained with RAKE

The hypothesis was that keywords carry vital information for a disambiguation process and give a lot of space for experimenting, therefore the next few experiments were keywords focused. In this set of keyword-focused experiments, we posited that paper-level keywords alone may be poorly aligned with software semantics—*e.g.*, a biology paper using a matrix library will carry biology-centric keywords that do not aid in disambiguating URLs featuring matrix-related terms. To address this, we extracted more contextually relevant keywords directly from the surrounding paragraph using the RAKE algorithm. In fact, this adjustment produced measurable gains: the univariate score for `keywords_metric` rose dramatically to 904.96 (versus 351.67 at our baseline; $p \approx 2.3 \times 10^{-183}$), elevating it to the second highest feature after `name_metric`. In the multivariate model, its importance increased from 10.8% to 13.3%, whereas `paragraph_metric`—though still critical—fell slightly from 16.8% to 13.1%. Meanwhile, `name_metric` remained dominant but its share dipped modestly (from 49.3% to 47.8%), reflecting a more balanced contribution from keywords. In general, our RAKE-based keyword extraction substantially narrowed the gap between name and keyword signals, validating the hypothesis that locally extracted keywords better discriminate software mentions and the performance results of all models continued to increase (see Table 3.3), which is why software mention keywords were fetched using RAKE in the upcoming experiments.

Table 3.3: Test Precision, Recall, and F_1 Scores for Keywords from RAKE Experiment

Classifier	Precision %	Recall %	F_1 score %
Logistic Regression	66	93	77
Random Forest	85	92	88
XGBoost	84	95	89
LightGBM	84	95	89
Neural Network	84	95	89

3.2.4 Missing GitHub Keywords Obtained with RAKE

In our analysis, we observed that GitHub and PyPI entries frequently lack keyword metadata. To address this gap, we imputed missing GitHub keywords by applying the RAKE algorithm to each repository's description. This enhancement yielded further performance gains across all models except Logistic Regression and the Neural Network (see Table 3.4). These and previous experiment results suggest that tree-based

models are particularly well suited to our problem, therefore to maintain clarity, from the next experiment, results will be shown only for tree models. Consequently, we adopted RAKE-based keyword imputation for all missing GitHub entries in subsequent experiments.

Table 3.4: Test Precision, Recall, and F_1 Scores for GitHub Missing Keywords from RAKE Experiment

Classifier	**Precision %**	**Recall %**	**F_1 score %**
Logistic Regression	64	92	76
Random Forest	87	92	90
XGBoost	86	94	90
LightGBM	87	96	91
Neural Network	82	92	86

3.2.5 Missing PyPI keywords

To enrich keyword metadata, we evaluated three distinct imputation approaches, as detailed in the Methodology (see Section 3.1).

3.2.5.1 RAKE-only

When repository keywords were missing, we applied RAKE to the repository description, which further improved overall model performance (see Table 3.5).

Table 3.5: Test Precision, Recall, and F_1 Scores for PyPI Missing Keywords from RAKE Experiment

Classifier	**Precision %**	**Recall %**	**F_1 score %**
Random Forest	89	92	90
XGBoost	87	95	91
LightGBM	86	96	91

3.2.5.2 Rake and Classifier fallback

When RAKE did not produce keywords, we used classifier-based imputation for those cases, which did not materially alter the model performance.

3.2.5.3 Rake-after

In the next phase, we reversed the imputation sequence: missing keywords were first inferred via classifiers, then supplemented with RAKE extraction from repository descriptions. This adjustment improved performance for the Random Forest and XGBoost models, while LightGBM experienced a slight decline (see Table 3.6). As these results represented the best overall performance to date, we adopted this two-stage imputation strategy for handling missing PyPI keywords in subsequent experiments.

3.2.5.4 Feature Analysis

Across the three imputation strategies for missing PyPI keywords—RAKE-only, RAKE with classifier fallback, and classifier-first then RAKE—we observe a progressive at-

Table 3.6: Test Precision, Recall, and F_1 Scores for PyPI Missing Keywords from RAKE After Experiment

Classifier	Precision %	Recall %	F_1 score %
Random Forest	89	94	91
XGBoost	89	95	92
LightGBM	85	96	90

tenuation of the explicit keyword signal as more complex imputation is applied. In the RAKE-only version, the `keywords_metric` achieves a univariate score of 646.00 ($p \approx 2.1 \times 10^{-134}$) and a multivariate importance of 9.80%, ranking it third after `name_metric` and `paragraph_metric`. Introducing classifier-based fallback reduces the univariate score to 629.52 ($p \approx 3.3 \times 10^{-131}$) and importance to 9.60%. Finally, reversing the sequence further lowers the univariate score to 580.95 ($p \approx 9.6 \times 10^{-122}$) and importance to 9.50%. Concurrently, the model shifts weight toward `synonym_metric` (increasing from 13.56% to 13.97%) and `paragraph_metric` (rising from 13.31% to 13.72%). These results suggest that while multi-stage imputation broadens semantic coverage, it dilutes the precision of direct keyword matches; RAKE-only best preserves keyword fidelity, whereas the two-stage methods offer a more balanced reliance on semantic proxies. Considering that overall best results were obtained with the last results, here the idea that keywords are not vital for disambiguation started to arise.

3.2.6 SBERT Keyword Similarity Calculation

In the final keyword experiment, we replaced our BERT-based similarity computation with Sentence-BERT (SBERT), a substantially more lightweight model. This substitution yielded a significant reduction in calculation time, although we observed a modest decline in matching performance (see Table 3.7). Considering that this lowered results, next experiments returned to using BERT for calculating similarity for keywords.

Table 3.7: Test Precision, Recall, and F_1 Scores for SBERT Keywords Calculation Experiment

Classifier	Precision %	Recall %	F_1 score %
Random Forest	88	93	90
XGBoost	87	95	91
LightGBM	85	95	90

3.2.6.1 Removing Keywords

As noted earlier, when keyword features exhibited diminished impact in our feature analysis, model performance often improved. To explore this further, we performed an ablation study by removing all keyword-based features. Surprisingly, overall results remained largely unchanged or even improved—except for a modest drop in XGBoost precision (see Table 3.8). This outcome was both shocking and counterintuitive, as we initially expected keywords to play a critical role in disambiguation. In next experiments, we evaluated both keyword-inclusive and keyword-

omitted configurations. In our most comprehensive test—training and evaluating every possible feature subset—we discovered that omitting keywords consistently yielded equal or superior performance. Because of this conclusion, the results presented for the rest of experiments will be the ones achieved without keywords. Additionally, across every possible feature subset, the `name_metric` proved absolutely indispensable: on its own it yielded very high recall (90–98%) but only moderate precision ($\approx 60\%$), for an F_1 around 75%. A `synonym_metric`–only model behaved similarly, with recall above 90% but precision near 50%. Pairing `name_metric` and `synonym_metric` dramatically improved the balance—precision climbed to nearly 70% and recall exceeded 95%, giving $F_1 \approx 80\%$. Similar gains were observed for `name_metric+keywords_metric` and `name_metric+paragraph_metric` in some classifiers, although Random Forest, LightGBM, and the neural network showed more variability. In contrast, `name_metric+language_metric` delivered consistently strong results across all models (precision >70%, recall $\sim 97\%$, F_1 >80%). Combinations lacking `name_metric` consistently gave poor results, underscoring its critical role. More complex sets further boosted performance: `name_metric+language_metric+synonym_metric` achieved precision $\approx 80\%$, recall >90%, $F_1 \approx 85\%$, and adding `author_metric` raised precision to $\approx 85\%$, recall >90%, and F_1 to nearly 90%. Substituting keywords or paragraph for authors yielded comparable but slightly lower metrics. The best overall results came from LightGBM and XGBoost with name, synonyms, authors, language, and paragraph. These findings confirm that while `name_metric` is the foundation, carefully chosen multi-feature combinations, particularly those including language, synonyms, paragraph, and authors, maximize the performance of our models.

Table 3.8: Test Precision, Recall, and F_1 Scores for Removing Keywords Experiment

Classifier	**Precision %**	**Recall %**	**F_1 score %**
Random Forest	89	93	91
XGBoost	87	95	91
LightGBM	86	96	91

3.2.7 Levenshtein Name Similarity Calculation

In the next phase, we experimented with alternative similarity measures for name-based matching, replacing the Jaro–Winkler metric with the Levenshtein distance in light of the name feature's consistently dominant importance. We observed that feature analysis under Levenshtein yielded marginally higher univariate and multivariate scores for `name_metric`, indicating an even stronger isolated signal. The overall model performance stayed consistent for every metric and model, except for Random Forest recall which dropped by 1% (see Table 3.9). Consequently, we reverted to the Jaro–Winkler metric for name similarity in all upcoming experiments.

Table 3.9: Test Precision, Recall, and F_1 Scores for Levenshtein Name Similarity Calculation Experiment

Classifier	Precision %	Recall %	F_1 score %
Random Forest	89	92	91
XGBoost	87	95	91
LightGBM	86	96	91

3.2.8 Levenshtein Synonyms Similarity Calculation

Next, we evaluated the effect of using Levenshtein distance in place of Jaro–Winkler for the synonyms. Interestingly, this modification obtained improved performance for the LightGBM. XGBoost obtained the best result for the model itself so far, while precision and recall declined for Random Forest (see Table 3.10). Feature analysis confirmed that the `synonym_metric` achieved higher univariate and multivariate importance under Levenshtein, suggesting that this measure captures semantic variation more directly. Looking at the mixed nature of these results, the next step was combining Levenshtein for both name and synonyms similarity.

Table 3.10: Test Precision, Recall, and F_1 Scores for Levenshtein Synonyms Similarity Calculation Experiment

Classifier	Precision %	Recall %	F_1 score %
Random Forest	87	93	90
XGBoost	88	96	92
LightGBM	86	97	91

3.2.9 Levenshtein Name and Synonyms Similarity Calculation

At this stage, classifier performance exhibited mixed trade-offs: certain models achieved higher precision at the expense of recall, while others showed the opposite pattern (see Table 3.11), though overall results remained strong. In determining our next steps and identifying the most promising candidates, we chose to prioritize precision and F_1 score—even if this meant accepting a reduction in recall, because when linking software mentions to their actual URLs, precision outweighs recall because a single incorrect link can be far more damaging than a missed one. A false positive—misclassifying a URL as the correct target when it isn't—can send users to the wrong project, outdated documentation, or even a malicious site, undermining confidence in your system and potentially causing security risks. In contrast, a false negative—failing to identify a valid URL—merely means the user has to search a bit further, which is usually less harmful.

Table 3.11: Test Precision, Recall, and F_1 Scores for Levenshtein Name and Synonyms Similarity Calculation Experiment

Classifier	Precision %	Recall %	F_1 score %
Random Forest	87	92	89
XGBoost	86	95	90
LightGBM	87	97	92

3.2.9.1 SBERT Keywords Similarity Calculation

Although our primary configuration omitted keyword features, we performed an auxiliary experiment combining Levenshtein-based name and synonym similarities with SBERT-derived keyword similarity to assess the full design space. This setup uniformly degraded performance across all classifiers, indicating that the inclusion of SBERT-based keywords did not yield any benefit.

3.2.10 SBERT Paragraph Similarity Calculation

Since paragraph similarity was our next most important feature, we decided to swap out the RoBERTa embeddings for SBERT in the `paragraph_metric` and return to computing name and synonyms metrics with Jaro–Winkler. As you can see in Table 3.12, when we prioritized precision, Random Forest and LightGBM both reached their best performance with SBERT, but XGBoost still did best with the earlier Levenshtein-synonym approach. Feature analysis reveals that the univariate score for `paragraph_metric` increased from 909.87 ($p \approx 2.8 \times 10^{-184}$) when using RoBERTa to 935.75 ($p \approx 4.7 \times 10^{-189}$) now, while its multivariate importance remained effectively constant (13.72% vs. 13.56%). Concurrently, the importance of `name_metric` rose modestly from 44.32% to 44.81%, whereas `synonym_metric` declined slightly (from 13.97% to 13.53% and 9.50% to 9.19%, respectively). This suggests that SBERT embeddings capture richer contextual nuances in paragraphs, boosting the standalone strength of the paragraph feature, without fundamentally altering the overall weighting of the model's characteristic. In other words, SBERT helps our tree models eke out extra precision from paragraph context, even though the relative importance of name and synonym metrics remains stable. Therefore, this approach will be used in the next experiments.

Table 3.12: Test Precision, Recall, and F_1 Scores for SBERT Paragraph Similarity Calculation Experiment

Classifier	**Precision %**	**Recall %**	**F_1 score %**
Random Forest	90	92	91
XGBoost	88	93	91
LightGBM	88	95	91

3.2.11 Discarding GitHub and PyPI URLs

For the next phase, we sampled an entirely new data set from CZI, one not previously used, and applied our best-performing models to it. During error analysis, we observed that many false positives stemmed from 'noisy' URLs. On GitHub, these noise URLs almost universally lacked a `README` file (with the exception of repositories authored by `cran`), while on PyPI the problematic entries featured overly short package descriptions without linked GitHub project. To mitigate this, we implemented a simple heuristic filter that discards any GitHub URL without a `README` and any PyPI entry whose description is below a length threshold or lacks a GitHub link.

3.2.11.1 GitHub

When we discarded GitHub URLs without a README file, the model performance stayed basically the same (see Table 3.13). Since filtering didn't hurt our results, we decided to keep using this approach.

Table 3.13: Test Precision, Recall, and F_1 Scores for Discarding GitHub Experiment

Classifier	**Precision %**	**Recall %**	**F_1 score %**
Random Forest	90	92	91
XGBoost	88	93	91
LightGBM	88	95	91

3.2.11.2 PyPI

We tested two length thresholds for PyPI descriptions, 400 and 300 characters, and found that each one alone led to a -2, 3% drop in performance. When we combined the 400 character limit with our GitHub README filter, overall metrics still decreased slightly (see Table 3.14), while combined with the 300 limit made it decrease drastically. Also, if the PyPI has GitHub linked, it was added to the candidates. The performance drop can be attributed to a small subset of valid packages whose repositories were mistakenly filtered out because they did not meet the imposed conditions. For example, `https://pypi.org/project/pyphysio/` and `https://pypi.org/project/pyqi/` are valid PyPI packages that were excluded due to short descriptions, despite being legitimate resources. However, a larger factor is that many of the removed repositories shared names with valid CRAN packages. In those cases, although the PyPI URL is not the correct match for the mention and lacks a detailed description, it still contains minimal metadata that could help the model learn the difference between similarly named packages across platforms.

Despite this, we chose to retain the 400-character description cut-off, as we believe it effectively filters out low-quality or placeholder URLs. In the long term, this threshold helps improve the overall clarity of the candidate pool and supports better model performance by reducing noise during training.

Table 3.14: Test Precision, Recall, and F_1 Scores for Discarding GitHub and PyPI Experiment

Classifier	**Precision %**	**Recall %**	**F_1 score %**
Random Forest	89	90	89
XGBoost	87	93	90
LightGBM	85	94	90

3.2.12 Removing Class Weighted Approach

We also tried the experiment without using class weights, even though there is still a moderate imbalance between the classes. Surprisingly, this tweak increased precision while only causing a negligible drop in recall (see Table 3.15). At this point, the best result we had after discarding and adding new URL candidates was LightGBM in this experiment.

Table 3.15: Test Precision, Recall, and F_1 Scores without Class Weighted Approach Experiment

Classifier	Precision %	Recall %	F_1 score %
Random Forest	88	90	89
XGBoost	89	90	89
LightGBM	90	92	91

3.2.13 Adding Additional URL Candidates

Following our PyPI approach, we decided to include every GitHub URL linked from CRAN candidate packages. We also realized that prior to this change, GitHub repositories authored by `cran` were only sometimes captured by our GitHub search; we therefore ensured that each CRAN candidate URL explicitly includes its associated GitHub repo. This revision nearly doubled our mention–URL corpus from 6500 rows to 10 000, exacerbating the class imbalance. As a result, model performance increased (see Table 3.16), which was expected, considering we view the larger, more comprehensive dataset as beneficial for long-term robustness.

Table 3.16: Test Precision, Recall, and F_1 Scores for Adding Additional URLs Experiment

Classifier	Precision %	Recall %	F_1 score %
Random Forest	92	90	91
XGBoost	90	95	92
LightGBM	87	95	91

We repeated the last experiment without applying class weights, aiming to increase performance. As before, this adjustment led to a noticeable increase in precision but incurred a drop in recall (see Table 3.17). This result indicates that while removing class weighting can sharpen the model's positive predictions, it may further exacerbate recall losses in highly imbalanced, expanded datasets.

Table 3.17: Test Precision, Recall, and F_1 Scores for Adding Additional URLs without Class Weights Experiment

Classifier	Precision %	Recall %	F_1 score %
Random Forest	90	90	90
XGBoost	92	92	92
LightGBM	92	93	92

3.2.14 Fine Tuning

We decided to do fine tuning for the best models on the expanded training corpus, which were XGBoost and LightGBM. For XGBoost, we defined a grid over the number of trees (100, 200, 500), maximum tree depth (3, 6, 9), learning rate (0.01, 0.1, 0.2), subsampling ratio (0.6, 0.8, 1.0), and column sampling per tree (0.6, 0.8, 1.0). For LightGBM, we explored variations in the number of iterations (100, 200, 500), number of leaves (31, 50, 100), learning rate (0.01, 0.1, 0.2), and minimum data in leaf (20, 50, 100). Each combination was evaluated via five-fold cross-validation

to identify the optimal configuration. The best configurations turned out to be the starting ones, described in Modeling (see Section 3.1.7).

3.2.15 Final Approach

Finally, we chose the two best performing models, XGBoost and LightGBM from the final experiment (Section 3.2.13), which means that the similarities approaches chosen were Jaro-Winkler for the name, synonyms, author and language metadata, and SBERT for paragraph, while keywords were discarded for reasons previously discussed. We were interested in how these models would perform on an independent CZI sample of 142 additional samples. As shown in Table 3.18, the XGBoost model of the final experiment outperformed slightly on new data. The relatively small size of the CZI sample, combined with a match-to-nonmatch ratio that may differ from the original split, means that each error has a disproportionately large effect on measured precision. Furthermore, the presence of niche or domain-specific software names - absent from the training data - weakens the signal based on the embedding of the model. Together, these two factors help explain the approximate 10% drop in precision. After finalizing the choice of the best performing model, we used it to generate predictions for an additional 806 mentions sampled from the CZI dataset. These mentions were not manually cleaned or included in the evaluation process, but the resulting prediction file - intended to support further analysis or future annotation - can be found on Zenodo [5].

Table 3.18: Test Precision, Recall, and F_1 Scores for Final Model Experiment

Classifier	**Prec. %**	**Rec. %**	**F_1 %**	**Prec. (CZI) %**	**Rec. (CZI) %**	**F_1 (CZI) %**
XGBoost	92	92	92	82	91	86
LightGBM	92	93	92	81	91	85

3.2.16 LLM Comparative Evaluation

In addition to our usual tests on held-out sets, we became curious about how large language models would tackle software disambiguation. To directly compare our pipeline with language models, we evaluated four pre-trained LLMs:`llama-3.1-8b-instant`, `qwen-qwq-32b`, `gemma2-9b-it`, and `deepseek-r1-distill-llama-70b` on the same 20% held-out test split used for our classifiers. For each pair of mention-paragraph and its candidate URLs, we constructed a single prompt containing the software name, DOI, surrounding text, authors, detected language, synonyms and respective URL metadata, as seen in the example bellow. Since most of our work focused on experimenting with different similarity metrics and identifying which ones truly contribute to URL disambiguation, we excluded similarity scores that were precomputed in other approaches. This ensured a fair evaluation, as we wanted to assess the effectiveness of our own similarity-based features using the same raw metadata that were available before any similarity calculations.

Prompt

You are given information about a software mention in a scientific paper and a candidate URL.

Software: BeautifulSoup

DOI: 10.3389/fvets.2021.674730
Context: The HTML and XML document parser, BeautifulSoup, from Python module BS4 was implemented to parse the source page; multimodal texts of Web page sources were extracted as BeatifulSoup object
Paper authors: Majid Jaberi-Douraki, Soudabeh Taghian Dinani, Nuwan Indika Millagaha Gedara, Xuan Xu, E. Richards, Fiona Maunsell, Nader Zad, Lisa A. Tell
Programming language from context: Python
Synonyms: "Beautiful Soup", beautiful-soup, BeautifulSoup Python library, Beautiful, BeautifulSoup Python Package, Beautiful soup, Beautiful Soup, BeatifulSoup, BeautifulSoup4, beautifulsoup4, BeautifulSoup9, BeautifulSoup 4 (PyPi)

Candidate URL: `https://github.com/DeronW/beautifulsoup`
→ **URL name:** beautifulsoup
→ **URL authors:** delong.wang
→ **URL description:** Beautifulsoup docs in Chinese
→ **URL programming language:** [not specified]

Task:
Determine if the candidate URL refers to the same software mentioned in the paper. Return only a single digit:

- **1** if the URL corresponds to the software
- **0** if it does not

The data was then submitted through the Groq API [26]. Each model returned a binary decision (1=match, 0=no match), which we collected and compared with the ground truth labels. Finally, we computed precision, recall and F_1 scores, allowing a direct comparison between LLM performance and our models.

It was somewhat expected that these large language models - trained on far more data than our specialized pipeline - would achieve high overall scores, but their precision proved to be lower (see Table 3.19), even with an extremely high recall. This outcome highlights the importance of trying various data sources and similarity measures when tackling software disambiguation. It also demonstrates that a tailored model, optimized specifically for this task, can outperform more general approaches by focusing on the right features and similarity calculations, although, it is important to note that agentic-based architectures may offer improved performance, but often come with higher computational costs or may require paid access.

Table 3.19: Test Precision, Recall, and F_1 Scores for LLMs Evaluation

Classifier	**Precision %**	**Recall %**	**F_1 score %**
llama-3.1-8b-instant	45	88	80
qwen-qwq-32b	37	99	59
gemma2-9b-it	68	97	80
deepseek-r1-distill-llama-70b	40	98	57

3.2.17 Qualitative Examples

It is crucial to develop an intuitive understanding of our model's behavior. To that end, we examine two representative cases: one drawn from the additional CZI sample

used in the final evaluation of our top three models, and another selected from the pipeline's internal test set. By analyzing how each model handles these concrete examples, we can gain insight into the patterns and edge cases that drive its decisions.

3.2.17.1 Scikit-learn Example

The paper titled "Circulating bacterial signature is linked to metabolic disease and shifts with metabolic alleviation after bariatric surgery" [23] investigates the presence and role of bacterial DNA in human blood and its association with obesity, type 2 diabetes (T2D), and metabolic improvements after bariatric surgery. The `scikit-learn` library was used to build a naive Bayes classifier trained on the SILVA database, enabling taxonomic classification of bacterial DNA sequences obtained from 16S rRNA gene sequencing. The software mention found has the following metadata:

- **name:** `scikit-learn`
- **paragraph:** For taxonomic classification, a scikit-learn [44] naive Bayes classifier was created against the taxonomic classification from ARB-SILVA [45] 132 release (99% OTU data set), which was trained for the used primers.
- **ground truth URLs:**
  - `https://pypi.org/project/scikit-learn/`
  - `https://github.com/scikit-learn/scikit-learn`
- **DOI:** `10.1186/s13073-021-00919-6`
- **synonyms:** sciki-learn, SciKit-learn, sckit-learn, Python scikit-learn, scikit-learn Developers, Python scikit-learn package, SciKit learn, etc.
- **language:** (none)
- **authors of the paper:** Rima Chakaroun, Lucas Massier, Anna Heintz-Buschart, Nedal Said, Jörg Fallmann, Alyce Crane, Tatjana Schütz, Arne Dietrich, Matthias Blüher, Michael Stümvoll, Niculina Musat, Péter Kovács
- **URLs SONAD identified as correct:**
  - `https://pypi.org/project/scikit-learn/`
  - `https://github.com/scikit-learn/scikit-learn`
- **URLs SONAD identified as incorrect:**
  - `https://cran.r-project.org/package=hmclearn`
  - `https://github.com/cran/catlearn`
  - `https://cran.r-project.org/package=catlearn`
  - `https://github.com/justmarkham/scikit-learn-videos`
  - `https://cran.r-project.org/package=asciiruler`
  - `https://cran.r-project.org/package=specklestar`
  - `https://github.com/ajwills72/catlearn`
  - `https://github.com/scikit-learn-contrib/sklearn-pandas`

- https://cran.r-project.org/package=clustlearn
- https://github.com/drastega/specklestar
- https://github.com/cran/asciiruler
- https://github.com/cran/clustlearn
- https://github.com/cran/hmclearn
- https://github.com/cran/specklestar
- https://github.com/scikit-learn-contrib/hdbscan
- https://github.com/Ediu3095/clustlearn
- https://github.com/scikit-learn-contrib/imbalanced-learn

Knowing that name similarity is the most influential feature, we can conclude why URLs https://cran.r-project.org/package=hmclearn, https://github.com/cran/catlearn, https://cran.r-project.org/package=catlearn, https://cran.r-project.org/package=asciiruler, https://cran.r-project.org/package=specklestar, https://github.com/ajwills72/catlearn, https://cran.r-project.org/package=clustlearn, https://github.com/drastega/specklestar, https://github.com/cran/asciiruler, https://github.com/cran/clustlearn, https://github.com/cran/hmclearn, https://github.com/cran/specklestar, and https://github.com/Ediu3095/clustlearn are dismissed. We see that name similarity is very low; even with the synonyms included, it is clear that these are not the URLs that the software is referring to.

Next, we encounter two URLs—https://github.com/scikit-learn-contrib/hdbscan and https://github.com/scikit-learn-contrib/imbalanced-learn—that superficially appear to be related to scikit-learn. However, in both cases, the 'scikit-learn' functions merely as part of the author or organization name rather than the actual software being mentioned. Their repository descriptions focus on clustering and imbalance learning topics that have little semantic overlap with the content of the surrounding paragraphs. This mismatch in both naming context and topical relevance clearly explains why our model correctly excludes them as valid scikit-learn links.

Finally, we consider the last two relevant URLs,https://github.com/justmarkham/scikit-learn-videos and https://github.com/scikit-learn-contrib/sklearn-pandas. Both repositories contain 'scikit-learn' in their names and feature content closely related to the library, so it is not immediately obvious why they were rejected. However, our model correctly discards them, demonstrating its ability to detect nuanced contextual cues beyond simple string matches. In these instances, the surrounding paragraph refers to the core scikit-learn package itself, not video tutorials or auxiliary tooling, and the descriptions of these repos emphasize supplementary resources rather than the primary software. This subtle distinction highlights the model's strength in distinguishing near-match URLs that would otherwise be tempting false positives.

In correctly classified cases, URL metadata aligns closely with the text mention on multiple fronts: the software name achieves a very high name similarity score, its synonyms register strong matches, and both the surrounding paragraph and the

URL's description exhibit high contextual similarity. Together, these complementary signals drive the model to mark these URLs as correct.

3.2.17.2 Widoco Example

The paper "Onto4CITY: An Ontology for Smart City Data Integration and Knowledge Management" [24] presents Onto4CITY, an ontology designed to model and integrate diverse urban data related to mobility, environment, and infrastructure in smart cities. The authors used Widoco (Wizard for Documenting Ontologies) [25] to generate human-readable documentation of the ontology, producing structured HTML pages that describe its classes, properties, and structure to facilitate reuse and understanding by others. The example was used while testing SONAD package; it is not present in CZI, therefore, it has no synonyms.

- **name:** Widoco
- **paragraph:** Documentation and publication: Creating HTML documentation using WIDOCO. Multiple practical examples were elaborated on. An example data set, which instantiates the ontology, was openly published under the CC-BY 4.0 license.
- **ground truth URLs:**
  - `https://github.com/dgarijo/Widoco`
- **DOI:** `10.3390/buildings12101522`
- **synonyms:** (none)
- **language:** HTML
- **authors of the paper:** Alex Donkers, Dujuan Yang, Bauke de Vries, Nico Baken
- **URLs SONAD identified as correct:**
  - `https://github.com/dgarijo/Widoco`
- **URLs SONAD identified as incorrect:**
  - `https://cran.r-project.org/package=midoc`
  - `https://github.com/cran/RWildbook`
  - `https://github.com/cran/doconv`
  - `https://cran.r-project.org/package=docopt`
  - `https://github.com/IATA-Cargo/one-record-ontologies-widoco`
  - `https://cran.r-project.org/package=RWildbook`
  - `https://github.com/Salva5297/WidocoServer`
  - `https://github.com/soilwise-he/widoco-action`
  - `https://github.com/docopt/docopt.R`
  - `https://github.com/cran/midoc`
  - `https://github.com/ardata-fr/doconv`

- https://cran.r-project.org/package=doconv
- https://github.com/cran/docore
- https://cran.r-project.org/package=docore
- https://github.com/cran/docopt
- https://github.com/ec-europa/widoco

In most of the misclassified cases, the rejection of the model is primarily based on a low name similarity score: When neither the raw name nor any known synonym aligns with the mention, that signal alone is sufficient to flag the URL as incorrect. For the few URLs that contain 'widoco' in their path -such as https://github.com/IATA-Cargo/one-record-ontologies-widoco, https://github.com/Salva5297/WidocoServer, https://github.com/ec-europa/widoco, and https://github.com/soilwise-he/widoco-action - mere presence of the term 'widoco' was insufficient to conclusively identify the correct software repository. Upon examining these repositories, it became clear they were distinct projects with diverse objectives and usage contexts, each reflecting significant differences in repository descriptions and primary programming languages. For instance, one project targets airline industry ontology documentation, another is a server implementation, a third is a general EU documentation initiative, and the last is designed as a GitHub action for documentation workflows. Thus, the model correctly leveraged additional contextual information from the paragraphs in academic papers, along with language-specific metrics and repository descriptions, to accurately determine that these similarly named URLs did not correspond to the intended software reference.

In the case of correctly classified URLs, the model easily chooses the true repository—https://github.com/dgarijo/Widoco—thanks to the tight alignment of its name with the mention and the strong semantic match between the paragraph and the description of the repository, as well as the language matching. This shows that, though some data may be missing, the pipeline and the model still provide correct URL disambiguation.

3.3 SONAD Application Deployment and Package Release

The SONAD tool (see Figure 3.3) is packaged and distributed using Poetry, which centralizes both dependency management and build metadata in a single `pyproject.toml`. In that file, we declare the package name (`sonad`), version, author, license, classifiers (including the requirement for Python 3.10), and a set of runtime dependencies:

- **Python requirement:** `==3.10`
- **Core libraries:** `rdflib`, `somef`, `click`, `cloudpickle`, `xgboost`, `lightgbm`, `sentence-transformers`, `textdistance`, `beautifulsoup4`, and `SPARQLWrapper`.

These dependencies are automatically installed when the user runs:

```
pip install sonad
```

which pulls the latest version from PyPI, along with all required packages.

A command line entry point (`sonad`) is defined in the `[tool.poetry.scripts]` section, so after installation the user can invoke:

```
sonad process --input mentions.csv --output results.csv [--temp temp/]
```

Here, `mentions.csv` must include at least:

- `name`: the software mention as it appears in the text,
- `doi`: the paper's DOI,
- `paragraph`: the surrounding sentence or paragraph containing the mention,
- `candidate_urls` (optional): a comma-separated list of URLs user wants to be included into disambiguation.

The result is written to `results.csv`, containing the original input columns plus:

- `synonyms`, `language`, `authors` — the inferred metadata,
- `urls` — a semicolon-separated list of URLs classified as correct,
- `not_urls` — the URLs evaluated but rejected.

Consider an example to illustrate how the SONAD works in practice. Suppose we encounter the software mention `scikit-learn` in the following sentence: *"For taxonomic classification, a scikit-learn naive Bayes classifier was created against the taxonomic classification from ARB-SILVA 132 release (99% OTU data set), which was trained for the used primers"* in the paper titled "Circulating bacterial signature is linked to metabolic disease and shifts with metabolic alleviation after bariatric surgery" [23], which has DOI `10.1186/s13073-021-00919-6`. To process this mention, the following row would be included in the input CSV file:

```
name,doi,paragraph
scikit-learn,10.1186/s13073-021-00919-6,"For taxonomic classification,
a scikit-learn naive Bayes classifier..."
```

Running the tool will result in a row in the output file with the format:

```
name,paragraph,doi,synonyms,language,authors,urls,not_urls
scikit-learn,"For taxonomic classification, a scikit-learn naive Bayes
classifier...",10.1186/s13073-021-00919-6,"scikit learn,sklearn,
Scikit-learn Python package,scikit-learn API,SciKit-Learn",Python,"Ri-
ma Chakaroun,Lucas Massier,Anna Heintz-Buschart, ...","https://github
.com/scikit-learn/scikit-learn, https://pypi.org/project/scikit-le
arn/","https://github.com/scikit-learn-contrib/sklearn-pandas,https:
//github.com/cran/catlearn, ..."
```

Under the hood, SONAD's function orchestrates the full pipeline: it reads the input, loads the bundled synonym matrix, infers languages and authors, fetches candidate URLs from GitHub, PyPI, and CRAN (with caching to avoid repeated network calls), extracts metadata for each URL (via SOMEF, PyPI JSON APIs, or CRANDB), computes similarity scores (Jaro–Winkler on names, synonyms, languages and authors; SBERT embeddings on paragraphs), applies a pre-trained machine-learning model (loaded with Cloudpickle), and finally aggregates the predictions back to one row per mention.

All intermediate data structures (candidate caches, metadata caches, exploded pair tables, similarity matrices) are optional and only persisted if the user passes a –temp;

otherwise SONAD cleans up on exit. This design keeps the tool both reproducible (all resources versioned in the package) and flexible for large-scale batch processing.

Currently, SONAD requires Python 3.10 to match requirements of SOMEF, but future releases will expand support to newer Python versions as soon as SOMEF allows it.

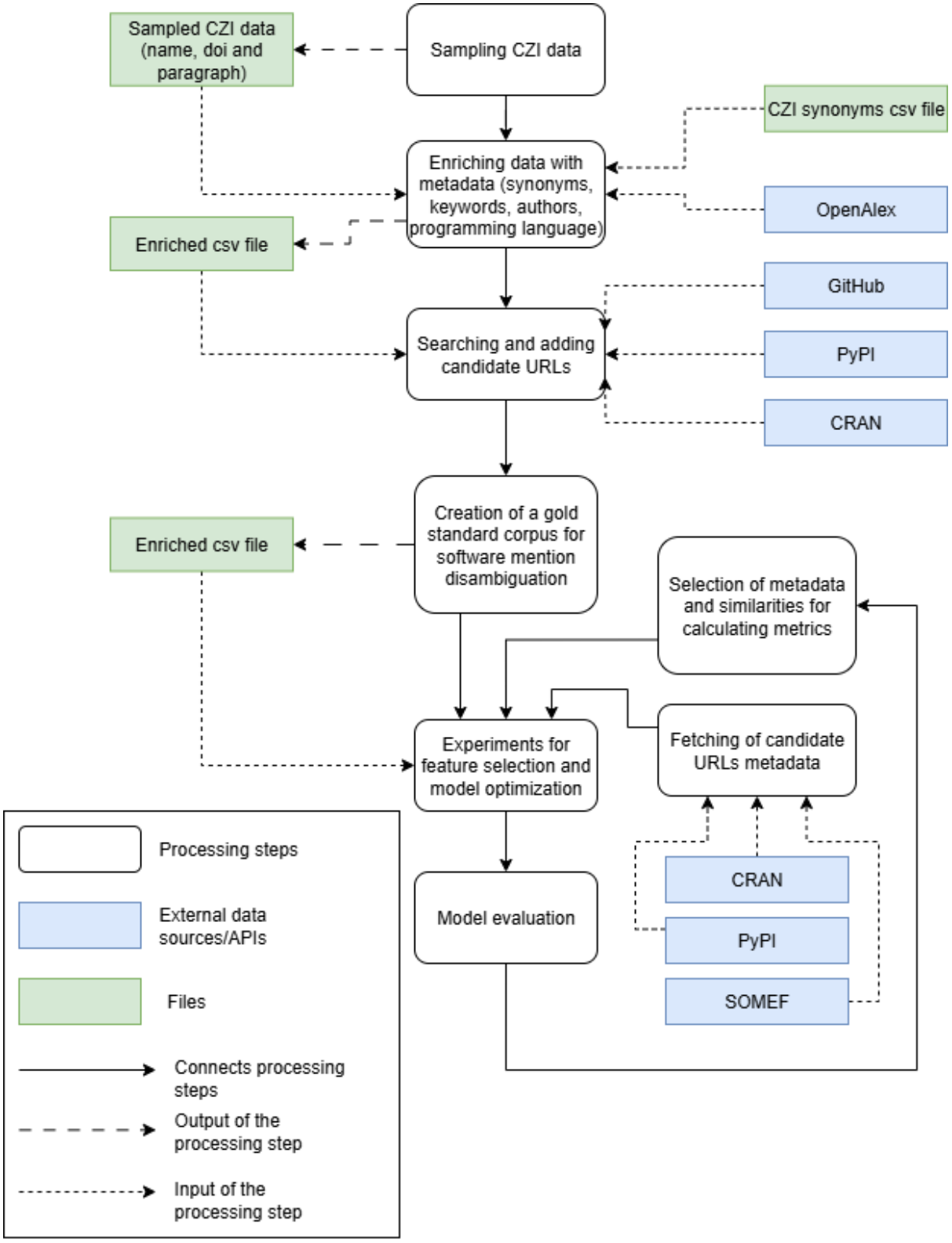

Figure 3.1: Overview of the SONAD methodology

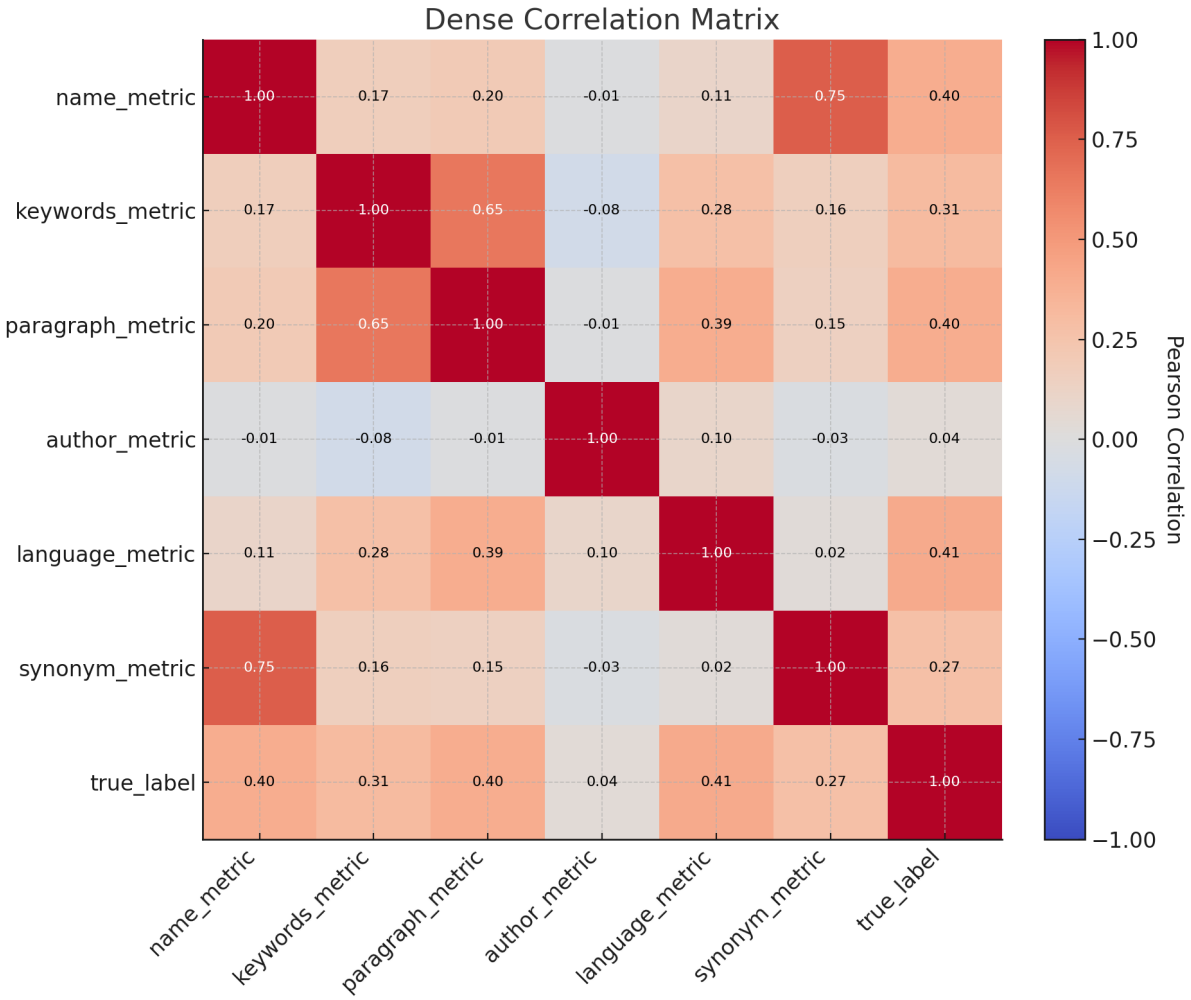

Figure 3.2: Dense correlation matrix showing pairwise Pearson correlations among the six similarity metrics and the ground-truth label. Strong positive correlations (warm colors) and negative correlations (cool colors) are annotated with their numeric values.

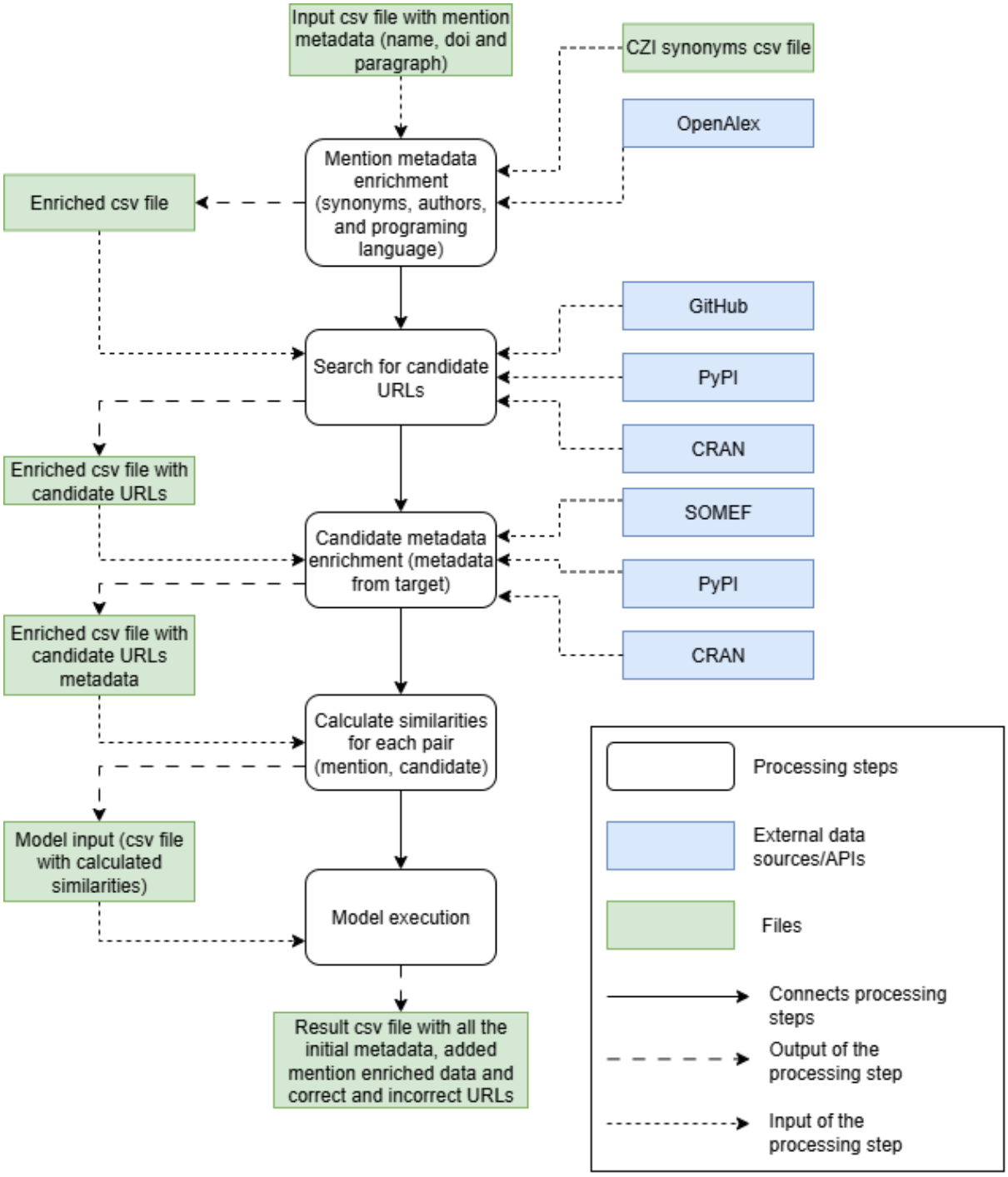

Figure 3.3: Overview of the SONAD package pipeline

Chapter 4

Conclusions and Future Work

4.1 Discussion

We examine the primary insights concluded from our experiments into three focused areas. First, we examine the relative importance of various data signals for disambiguation. Next, we analyze how different similarity metrics and embedding strategies influence performance. Finally, we compare the performance of several modeling approaches, highlighting their strengths and limitations.

4.1.1 Data

From our extensive research, we conclude that the software name itself is by far the most powerful disambiguation signal—no other feature approaches its ability to single out the correct repository. The surrounding paragraph then provides the essential semantic context that connects those names to the right URLs, rescuing cases where a name alone might refer to multiple unrelated projects. Synonyms offer valuable support, especially when authors employ alternate spellings or abbreviations, but their contribution remains secondary to the raw name and paragraph embeddings. Inferring the programming language proves crucial whenever identically named packages exist across ecosystems (for example, in both Python and R), since it deterministically anchors each mention to its proper platform. Perhaps most surprisingly, author overlap—though relatively rare—emerged as a useful precision booster, indicating that matching paper authors to repository maintainers can resolve borderline cases. Finally, contrary to our initial intuition, keywords provided little independent value: paper-derived keywords often misalign with software context, and keywords extracted from the paragraph merely duplicate information already captured by paragraph embeddings. In fact, including keywords sometimes degraded performance by introducing redundancy, underscoring the efficacy of focusing on name, context, synonyms, language, and authorship for robust software mention disambiguation.

4.1.2 Similarity Calculations

Overall, we found that Jaro–Winkler yielded the best end-to-end model performance for string features, likely because its prefix-bias and transposition handling closely match the kinds of minor typos and abbreviation variations common in software names. The Levenshtein distance, in contrast, produces a purer edit count that

amplifies differences and thus shows up as a stronger isolated signal in feature importance analyzes - even though it does not translate into higher overall accuracy.

On the paragraph side, SBERT embeddings proved both faster and more accurate than raw RoBERTa embeddings. This makes sense because SBERT is specifically fine-tuned with a pooling head and contrastive objectives to produce single-vector sentence representations optimized for semantic similarity. In contrast, RoBERTa produces per token embeddings that require additional pooling and have not been directly trained on sentence-level similarity tasks, both slowing inference and yielding less discriminative paragraph-level vectors in our context.

These findings underscore the value of choosing similarity measures and embedding models that are tailored to the specific noise patterns and granularity of the disambiguation task, rather than relying on off-the-shelf architectures alone.

Across our experiments, the tree-based gradient boosters XGBoost and LightGBM consistently outperformed all other methods. This is unsurprising given their ability to handle heterogeneous feature types, automatically model high-order interactions, and apply sophisticated regularization to prevent overfitting.

Random Forest came in a close second; Because it aggregates many decorrelated decision trees, it reduces variance effectively, making it robust on noisy data even though it does not optimize sequentially like gradient boosters.

Logistic regression fared poorly overall, as its linear decision boundary could not capture the complex nonlinear relationships present in our features without extensive manual feature engineering.

The neural network sometimes matched or even slightly beat the simpler models, suggesting that with deeper architectures, better hyperparameter tuning (e.g., learning rate schedules, dropout) and more training data, it could surpass the tree-based methods.

4.1.3 Challenges and Limitations

There are several limitations to our approach. First, although the necessary data is available, constructing a training corpus requires considerable time to manually verify each software mention and identify its correct URL. This manual process not only slows model development but also introduces the risk of human error: Misidentifying or overlooking URLs can lead the model to learn incorrect associations.

Second, our current implementation only supports three platforms, GitHub, PyPI, and CRAN. Consequently, users who seek software hosted elsewhere will not receive URLs. Although the tool can report when no valid URLs are found - indicating that alternative platforms should be consulted - it does not yet provide direct links for those cases.

A further dependency on Python 3.10 presents another constraint, although we anticipate that this issue will be resolved in the near future. Additionally, some metadata may be incomplete: Certain software entries lack synonyms in CZI, programming languages may not be mentioned in the accompanying paragraph, and author information from OpenAlex may be unavailable. Although missing metadata could complicate URL disambiguation, our model relies primarily on name and paragraph

similarity, features that remain robust even when other information is absent, so we expect to achieve strong performance despite these gaps.

However, it is important to acknowledge that, in some cases, disambiguation may simply not be possible. If multiple candidates share very similar names and no distinctive context is provided, such as a missing or vague paragraph, absent metadata, or overlapping repository names across platforms, even the best similarity model may not be able to determine the correct URL with high confidence. Furthermore, strict filtering rules (e.g., requiring a minimum description length or a non-empty README) may exclude valid repositories that appear minimal but are still correct. This introduces a trade-off between precision and recall that must be carefully managed depending on the intended application of the system.

4.2 Conclusions

Due to the prevalence of ambiguous naming conventions and inconsistent metadata, correctly identifying software references in academic texts presents a significant barrier to reproducibility and effective knowledge management. To overcome this, a robust supervised learning pipeline was developed, using extensive metadata extracted from scientific publications and repositories such as GitHub, PyPI, and CRAN. Diverse similarity measures, including string-based and embedding-based metrics, were systematically evaluated to discern their effectiveness. Among several models tested, including Logistic Regression, Random Forest, XGBoost, LightGBM, and Neural Networks, the best-performing model achieved precision, recall, and F_1 scores of 82%, 91%, and 86%, respectively. Comparative analyses with contemporary large language models underscored the superior performance of this structured, similarity-based supervised approach. Additionally, the creation of a manually validated gold-standard corpus further strengthened the validity of these results.

4.3 Broader Impact and Future Work

The broader impact of the SONAD project goes beyond the technical scope of software name disambiguation. By improving the precision and reliability of linking software mentions in scholarly publications to their correct repositories, SONAD significantly contributes to open science and the reproducibility of research. Accurate software citation practices facilitated by SONAD enhance the transparency and integrity of scientific communication, ensuring that researchers can easily locate and reuse software tools, thereby advancing collective scientific progress.

Additionally, the methodological framework developed through SONAD offers valuable insights into best practices in metadata utilization and similarity metrics within scholarly ecosystems. These insights can guide future research in refining and standardizing software citation protocols across diverse scientific communities. The project's explicit comparison against large language models (LLMs) highlights the continued necessity and effectiveness of structured, metadata-driven supervised learning approaches, emphasizing their role in achieving high precision and recall.

To build on the results obtained in this work, several approaches of investigation deserve further exploration. Each of the following areas promises to strengthen the

robustness, generality, and precision of the proposed software-mention disambiguation framework.

4.3.1 Comprehensive Candidate URL Inclusion

For the each experiment, generate and evaluate the *entire* set of candidate URLs for a given software mention—rather than limiting candidates to those returned by initial name-based searches. This includes URLs discovered via package-registry links (e.g., PyPI's `project_urls` or CRAN's `URL` field) and mirrored GitHub repositories.

4.3.2 Corpus Expansion

Increase both the size and diversity of the training corpus by incorporating additional papers, disciplines, and registry sources. A larger and more heterogeneous corpus will expose the model to various naming conventions, documentation styles, and domain-specific terminology. This breadth enhances the generalizability of learned similarity thresholds and feature importances, particularly for less common or interdisciplinary software tools.

4.3.3 Similarity Strategies

Due to time and resource constraints, we limited ourselves to testing only one or two similarity methods per feature type. A significant portion of our effort was initially focused on experimenting with keyword-based similarity, which ultimately proved to be of limited value. This focus explains why fewer variations were explored for other metrics. However, it could be useful to additionally evaluate alternative string- and token-based similarity measures beyond Jaro–Winkler and Levenshtein for string metadata types:

- *Name matching:* Cosine similarity on character n-gram TF–IDF vectors, Monge–Elkan, Smith–Waterman, or SoftTFIDF.
- *Language mentions:* Token-based Jaccard or Sørensen–Dice coefficients on normalized language keywords, or embedding-based similarity using fastText or BPEmb.
- *Author names:* Hybrid approaches combining phonetic algorithms (Soundex, Metaphone) with edit distances, or author-specific embedding comparisons (e.g., name2vec).

Different metrics capture distinct aspects of string similarity—character overlap, phonetic likeness, or semantic proximity. By tailoring the choice of metric to the structure of each metadata field (names, programming languages, personal names), we can improve disambiguation accuracy, especially in cases of abbreviations, transliterations, or minor typographical variations. In this work, we dedicated a lot of time to experimenting with keywords, but we strongly believe that experimenting with the rest of the metadata may increase the performance.

4.3.4 Exploration of Additional Embedding Models

Build on the promising performance of SBERT by exploring:

- *Domain-adapted SBERT:* Fine-tune existing SBERT models on a corpus of software documentation and research abstracts to better capture technical vocabulary.
- *Alternative sentence encoders:* Evaluate other lightweight transformers such as MPNet, Universal Sentence Encoder (USE), or MiniLM for trade-offs between inference speed and semantic fidelity.
- *Contrastive fine-tuning:* Apply contrastive learning (e.g., SimCSE) on software-mention pairs to sharpen distinctions between correct and incorrect URL candidates.

Although initial experiments with RoBERTa-large underperformed relative to SBERT, customizing SBERT to the software-mention domain and testing other compact encoders may yield further gains in capturing nuanced technical descriptions—while maintaining efficient inference times.

4.3.5 Deep Learning and End-to-End Architectures

Investigate deep learning models that jointly learn representation and classification, such as:

- Multimodal neural networks combining textual embeddings with structured metadata inputs,
- Graph neural networks that model relationships among software mentions, authorship, and package dependencies.

End-to-end architectures can automatically weight and integrate heterogeneous features, reducing reliance on hand-crafted similarity metrics. Graph-based methods, in particular, may exploit link structures (e.g., citations or dependency graphs) to improve disambiguation in tightly connected software ecosystems.

4.3.6 Multi-Platform Support

Extend the tool's coverage to additional software hosting platforms and registries, including Bioconductor (R), RubyGems (Ruby), crates.io (Rust),Maven Central (Java), and overall Google search that would include websites. Limiting support to GitHub, PyPI, and CRAN leaves gaps in applicability for users of other ecosystems. By integrating platform-specific search APIs and metadata extractors, the tool can deliver a unified disambiguation solution across diverse programming communities.

While SONAD supports one major code repository (GitHub) and two mayor package repositories (PyPI and CRAN), its contributions may help reduce inconsistencies in software citation. By automating part of the citation and linking process, it contributes towards improving findability in scientific research.

Bibliography

[1] SoftwareUnderstanding, “Software Disambiguation Benchmark: A benchmark for software name disambiguation and reconciliation,” GitHub, Jun. 2025. [Online]. Available: https://github.com/SoftwareUnderstanding/SoftwareDisambiguationBenchmark. Accessed: Mar. 2025.

[2] A.-M. Istrate et al., "CZ Software Mentions: A large dataset of software mentions in the biomedical literature," Dryad, 2022. [Dataset]. [Online]. Available: https://doi.org/10.5061/dryad.6wwpzgn2c. Accessed: Apr. 2025.

[3] J. Duric, “sonad: Software Name Disambiguation,” PyPI, version 0.2.6, Jun. 11, 2025. [Online]. Available: https://pypi.org/project/sonad/. Accessed: Jun. 2025.

[4] J. Duric, “Software-Disambiguation: Tackling the problem of software disambiguation in texts using Python and supervised machine learning,” GitHub, Jun. 2025. [Online]. Available: https://github.com/jelenadjuric01/Software-Disambiguation. Accessed: Jun. 2025.

[5] J. Duric, “Software Name Disambiguation Tool: SONAD,” *Zenodo*, Jul. 2025, doi: 10.5281/zenodo.15764860.

[6] A.-M. Istrate et al., "A large dataset of software mentions in the biomedical literature," *arXiv preprint* arXiv:2209.00693, Sep. 2022. [Online]. Available: https://arxiv.org/abs/2209.00693. Accessed: Mar. 2025.

[7] S. Druskat et al., "Don’t mention it: Challenges in using software mentions for citation and discoverability research," *arXiv preprint* arXiv:2402.14602, Feb. 2024. [Online]. Available: https://arxiv.org/abs/2402.14602. Accessed: Mar. 2025.

[8] A. M. Smith, D. S. Katz, K. E. Niemeyer, and N. P. Chue Hong, "Software citation principles," *PeerJ Computer Science*, vol. 2, art. no. e86, Sep. 2016, doi: 10.7717/peerj-cs.86.

[9] M. Ester, H.-P. Kriegel, J. Sander, and X. Xu, “A density-based algorithm for discovering clusters in large spatial databases with noise,” in *Proc. 2nd Int. Conf. Knowledge Discovery and Data Mining (KDD)*, Portland, OR, USA, Aug. 19–21, 1996, pp. 226–231.

[10] M. A. Jaro, “Advances in record-linkage methodology as applied to matching the 1985 census of Tampa, Florida,” *J. Amer. Stat. Assoc.*, vol. 84, no. 406, pp. 414–420, 1989.

[11] D. Schindler, F. Bensmann, S. Dietze, and F. Krüger, "SoMeSci—A 5 Star Open Data Gold Standard Knowledge Graph of Software Mentions in Scientific Articles," in *Proc. 30th ACM Int. Conf. Information and Knowledge Management (CIKM)*, Virtual Event, QLD, Australia, Nov. 1–5, 2021, pp. 4574–4583, doi: 10.1145/3459637.3482017.

[12] Z. Boukhers, "Whois? Deep Author Name Disambiguation using Bibliographic Data," arXiv preprint arXiv:2207.04772, Jul. 2022. [Online]. Available: https://arxiv.org/abs/2207.04772. Accessed: March 2025.

[13] D. Liu et al., "Author Name Disambiguation via Paper Association Refinement and Compositional Contrastive Embedding," in *Proc. ACM Web Conf.*, Singapore, May 13–17, 2024, pp. 2193–2203, doi: 10.1145/3589334.3645596.

[14] A. Kelley and D. Garijo, "A Framework for Creating Knowledge Graphs of Scientific Software Metadata," *Quant. Sci. Stud.*, vol. 2, no. 4, pp. 1423–1446, Dec. 2021, doi: 10.1162/qss_a_00167.

[15] J. Priem, H. Piwowar, and R. Orr, "OpenAlex: A fully-open index of scholarly works, authors, venues, institutions, and concepts," *arXiv preprint arXiv:2205.01833*, May 2022. [Online]. Available: `https://arxiv.org/abs/2205.01833`

[16] S. Rose, D. Engel, N. Cramer, and W. Cowley, "Automatic Keyword Extraction from Individual Documents," in *Text Mining: Applications and Theory*, M. W. Berry and J. Kogan, Eds. Chichester, UK: John Wiley & Sons, 2010, pp. 1–20, doi: 10.1002/9780470689646.ch1.

[17] V. I. Levenshtein, "Binary codes capable of correcting deletions, insertions, and reversals," *Soviet Physics Doklady*, vol. 10, no. 8, pp. 707–710, 1966.

[18] D. R. Cox, "The regression analysis of binary sequences," *Journal of the Royal Statistical Society: Series B (Methodological)*, vol. 20, no. 2, pp. 215–232, 1958.

[19] L. Breiman, "Random forests," *Machine Learning*, vol. 45, no. 1, pp. 5–32, Oct. 2001.

[20] T. Chen and C. Guestrin, "XGBoost: A scalable tree boosting system," in *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, San Francisco, CA, USA, Aug. 2016, pp. 785–794, doi: 10.1145/2939672.2939785.

[21] G. Ke *et al.*, "LightGBM: A highly efficient gradient boosting decision tree," in *Advances in Neural Information Processing Systems 30 (NIPS 2017)*, Long Beach, CA, USA, Dec. 2017, pp. 3146–3154.

[22] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning representations by back-propagating errors," *Nature*, vol. 323, no. 6088, pp. 533–536, Oct. 1986.

[23] R. M. Chakaroun *et al.*, "Circulating bacterial signature is linked to metabolic disease and shifts with metabolic alleviation after bariatric surgery," *Genome Medicine*, vol. 13, no. 1, Art. no. 105, Jun. 2021, doi: 10.1186/s13073-021-00919-6.

[24] A. Donkers, D. Yang, B. de Vries, and N. Baken, "Semantic Web Technologies for Indoor Environmental Quality: A Review and Ontology Design," *Buildings*, vol. 12, no. 10, Art. no. 1522, Oct. 2022, doi: 10.3390/buildings12101522.

[25] D. Garijo, "WIDOCO: A Wizard for Documenting Ontologies," in *The Semantic Web – ISWC 2017*, C. d'Amato *et al.*, Eds. Cham, Switzerland: Springer, 2017, pp. 94–102, doi: 10.1007/978-3-319-68204-4_9.

[26] "GroqCloud Developer Platform," Groq Inc., Mountain View, CA, USA, Feb. 2024. [Online]. Available: `https://console.groq.com/`. [Accessed: Jun. 2025].