The library linked automated medical bibliographic information tool (LLaMBIT): Code Manual

by

Erin S. King, BSN, RN, CPHQ

Submitted to the School of Public Health, School of Biomedical Informatics
in partial fulfillment of the requirements for the degree of

Master of Public Health

at the

THE UNIVERSITY OF TEXAS HEALTH SCIENCE CENTER

December 2023

© The University of Texas Health Science Center 2023. All rights reserved.

Author . . . .
School of Public Health, School of Biomedical Informatics
December 17, 2023

Under Advisement by . . . .
Dr. Baojiang Chen, PhD, MSE
Associate Professor
Practicum Supervisor

Under Advisement by . . . .
Dr. Sahiti Myneni, PhD, MSE
Associate Professor
Practicum Supervisor

Acknowledgments

Special thanks to Andrew King for assistance with code development and integration, Blake Harper for support and excellent facility preceptorship, and Travis Holder for valuable insight and direction.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

LLaMBIT: Library Linked Automated Medical Bibliographic Information Tool is an innovative software application designed to facilitate the automated gathering and processing of medical bibliographic data. Its primary function is to streamline the arduous task of extracting, preprocessing, and querying medical literature from various databases, notably PubMed, PubMed Central, and Web of Science. By integrating modern programming methodologies and data processing techniques, LLaMBIT significantly reduces the manual effort typically required in medical research and literature review.

1.1 Core Features

- **Automated Data Scraping**: Utilizing specialized scraper classes, such as *PubMedScraper*, *PubMedCentralScraper*, and *WoSJournalScraper*, LLaMBIT efficiently retrieves medical literature and associated data from multiple authoritative databases.

- **Data Preprocessing**: Through the *ArticlePreprocessor* class, the application processes the scraped data, making it conducive for analysis. This encompasses text cleaning, vectorization, and relevance scoring.

- **Advanced Querying**: LLaMBIT offers sophisticated querying functionalities, enabling users to find relevant articles based on specific search terms and defined thresholds.

- **User Interface**: The application is equipped with an intuitive user interface, comprising input fields for search queries, API keys, and user emails, complemented by engaging animations and informative loading messages.

- **Logging and Error Handling**: Built with robust error handling strategies and comprehensive logging, LLaMBIT ensures a smooth operation and facilitates effortless troubleshooting.

- **Configuration Management**: The tool allows for efficient loading and management of configurations, ensuring a tailored and efficient user experience.

1.2 Purpose and Scope of the Manual

This manual is intended as a detailed guide to the LLaMBIT application. It is designed to aid developers, researchers, and end-users in understanding the intricate functionalities of LLaMBIT, customizing it to suit individual needs, or employing it for medical literature research. The manual delineates LLaMBIT's architecture, features, and functionalities in an exhaustive manner. Additionally, it provides explicit instructions for installation, operation, and troubleshooting, making it an indispensable resource for both technical and non-technical stakeholders.

1.3 Document Structure

The subsequent sections of this manual delve into various aspects of LLaMBIT, each dedicated to a specific component or functionality:

1. **Application Architecture**: Detailed architecture and component description.

2. **User Interface**: Comprehensive overview and interaction guidelines.

3. **Scraper Functionality**: In-depth insights into scraper classes and methodologies.
4. **Article Preprocessing**: Detailed description of data preprocessing techniques.
5. **Query Parsing and Execution**: Explanation of query mechanisms and optimization strategies.
6. **Error Handling and Logging**: Best practices for error handling and application logging.
7. **Testing and Validation**: Guidelines for unit, functional, and performance testing.
8. **Deployment and Maintenance**: Procedures for deployment and ongoing maintenance.
9. **Appendices**: Supplementary information including code snippets and API references.

This structured approach ensures a comprehensive understanding of LLaMBIT's capabilities, fostering an environment conducive to efficient application use and development.

Chapter 2

Application Architecture

LLaMBIT's architecture is designed to efficiently handle the retrieval and processing of medical bibliographic data. This chapter delves into the technical architecture of the application, highlighting key components and their functionalities.

2.1 Main Components

The application is structured around several core components:

1. **User Interface (UI)**: Developed using PyQt5, it includes input fields, buttons, and dynamic animations for enhanced user interaction.

2. **Scraper Classes**: Responsible for fetching data from various medical databases.

3. **Article Preprocessor**: Processes the scraped data for analysis.

4. **Query System**: Allows users to search and filter the processed data.

2.2 Technical Implementation

2.2.1 User Interface Setup

```
# Set up animations
```

```
self.fade_in = QPropertyAnimation(self.opacity_effect, b"
    ↪ opacity")
self.fade_in.setDuration(1000)
self.fade_in.setStartValue(0)
self.fade_in.setEndValue(1)

self.fade_out = QPropertyAnimation(self.opacity_effect, b"
    ↪ opacity")
self.fade_out.setDuration(1000)
self.fade_out.setStartValue(1)
self.fade_out.setEndValue(0)

self.animation_group = QSequentialAnimationGroup()
self.animation_group.addAnimation(self.fade_in)
self.animation_group.addPause(2000)  # Display message for
    ↪  2 seconds
self.animation_group.addAnimation(self.fade_out)
```

2.2.2 Scraper Class Example

```
class WoSScraper(BaseScraper):
    def __init__(self, api_key, search_term, database_id='
        ↪ WOS'):
        super().__init__(search_term, api_key)
        logging.info(f"Initializing␣{self.__class__.
            ↪ __name__}␣with␣search␣term:␣{search_term}")
        self.configuration = self.configure_api(api_key)
        self.search_api_instance = self.init_search_api()
        self.database_id = database_id
```

2.2.3 Preprocessing and Querying

```
class ArticlePreprocessor:
    def __init__(self):
        self.vectorizer = TfidfVectorizer(stop_words='
           ↪ english')
        self.lemmatizer = WordNetLemmatizer()
        # Other initializations

    def preprocess(self, df):
        # Preprocessing steps
```

2.3 GUI Screenshots

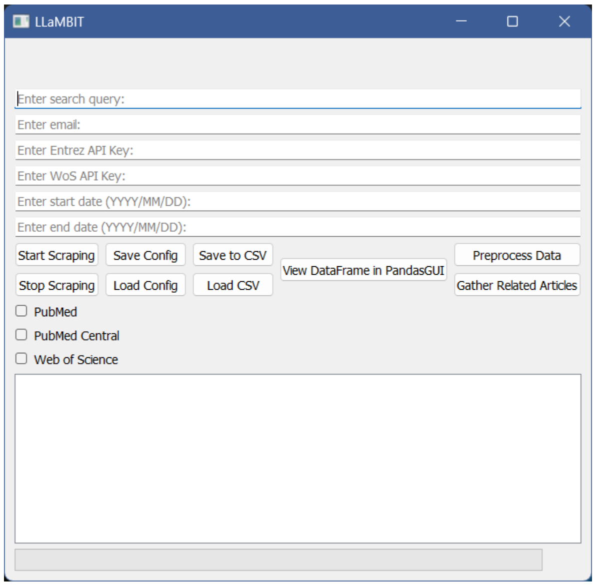


Figure 2-1: LLamBIT Main Screen

2.4 Application Diagram

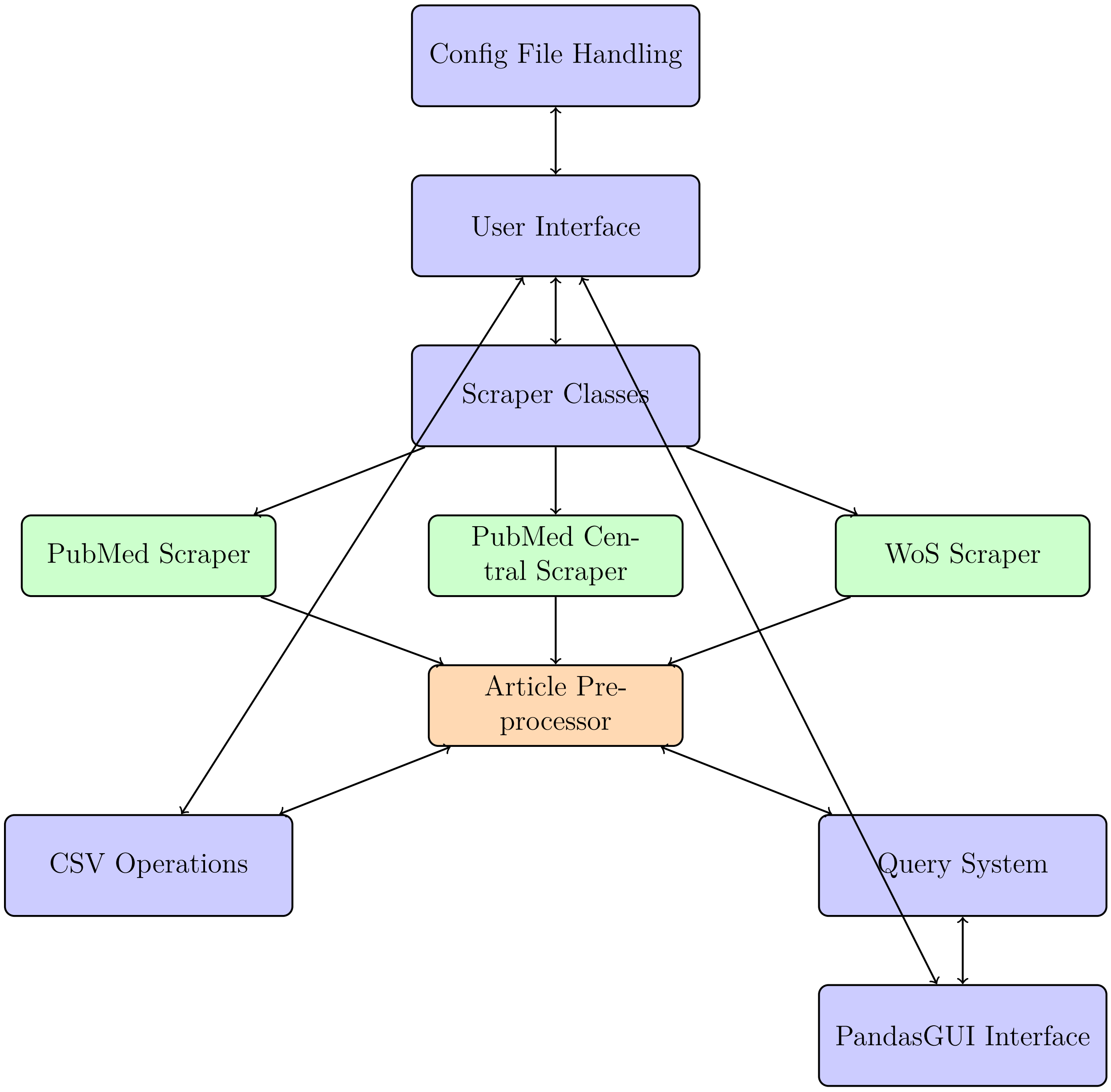


Figure 2-2: Detailed Architecture of LLaMBIT

Chapter 3

User Interface

LLaMBIT's User Interface, developed with PyQt5, provides an intuitive and interactive experience. This chapter describes the technical specifics of the UI components, including their design and functionalities.

3.1 UI Overview

The UI is designed to facilitate user interaction with various functionalities of LLaMBIT:

1. **Input Fields**: For entering search queries, email addresses, and API keys.
2. **Buttons**: To initiate scraping, stop ongoing processes, and handle data operations.
3. **Progress Indicators**: Visual feedback during lengthy operations.

3.2 Technical Implementation

3.2.1 Input Fields

```
# Input fields for search terms and email
self.search_term = QLineEdit(self)
self.search_term.setPlaceholderText("Enter␣search␣query:")
self.email_input = QLineEdit(self)
self.email_input.setPlaceholderText("Enter␣email:")
```

3.2.2 Buttons and Controls

```
# Buttons for starting and stopping scraping
self.start_scrape_button = QPushButton("Start␣Scraping",
    ↪ self)
self.start_scrape_button.clicked.connect(self.
    ↪ start_scraping)
self.stop_scrape_button = QPushButton("Stop␣Scraping",
    ↪ self)
self.stop_scrape_button.clicked.connect(self.stop_scraping
    ↪ )
```

3.2.3 Progress Bar and Animations

```
# Progress bar setup
self.progress_bar = QProgressBar(self)
self.progress_bar.setMaximum(100)

# Animations for loading messages
self.fade_in = QPropertyAnimation(self.opacity_effect, b"
    ↪ opacity")
self.fade_in.setDuration(1000)
self.fade_in.setStartValue(0)
self.fade_in.setEndValue(1)
```

3.3 GUI Screenshots

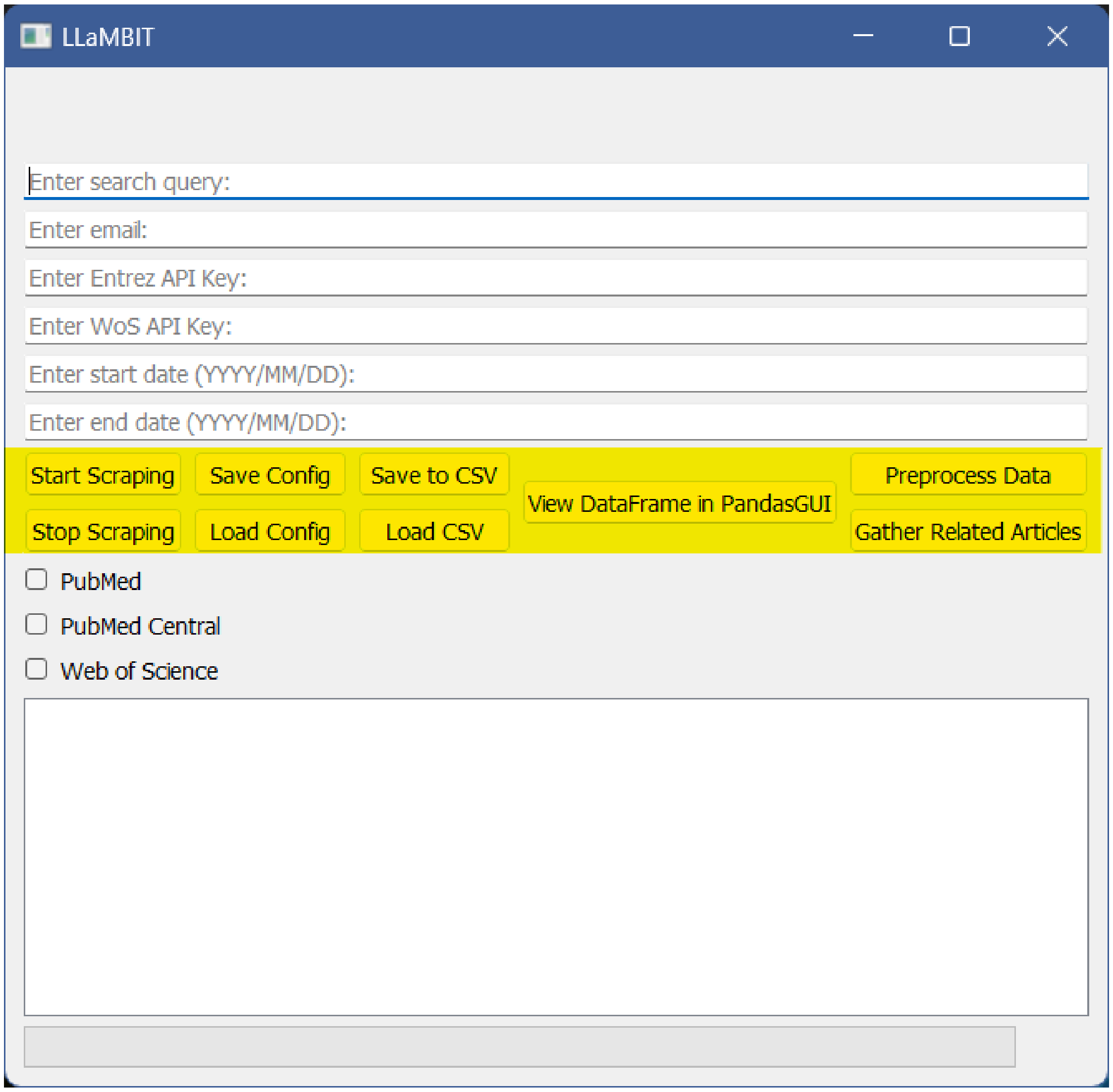


Figure 3-1: LLamBIT Button Options

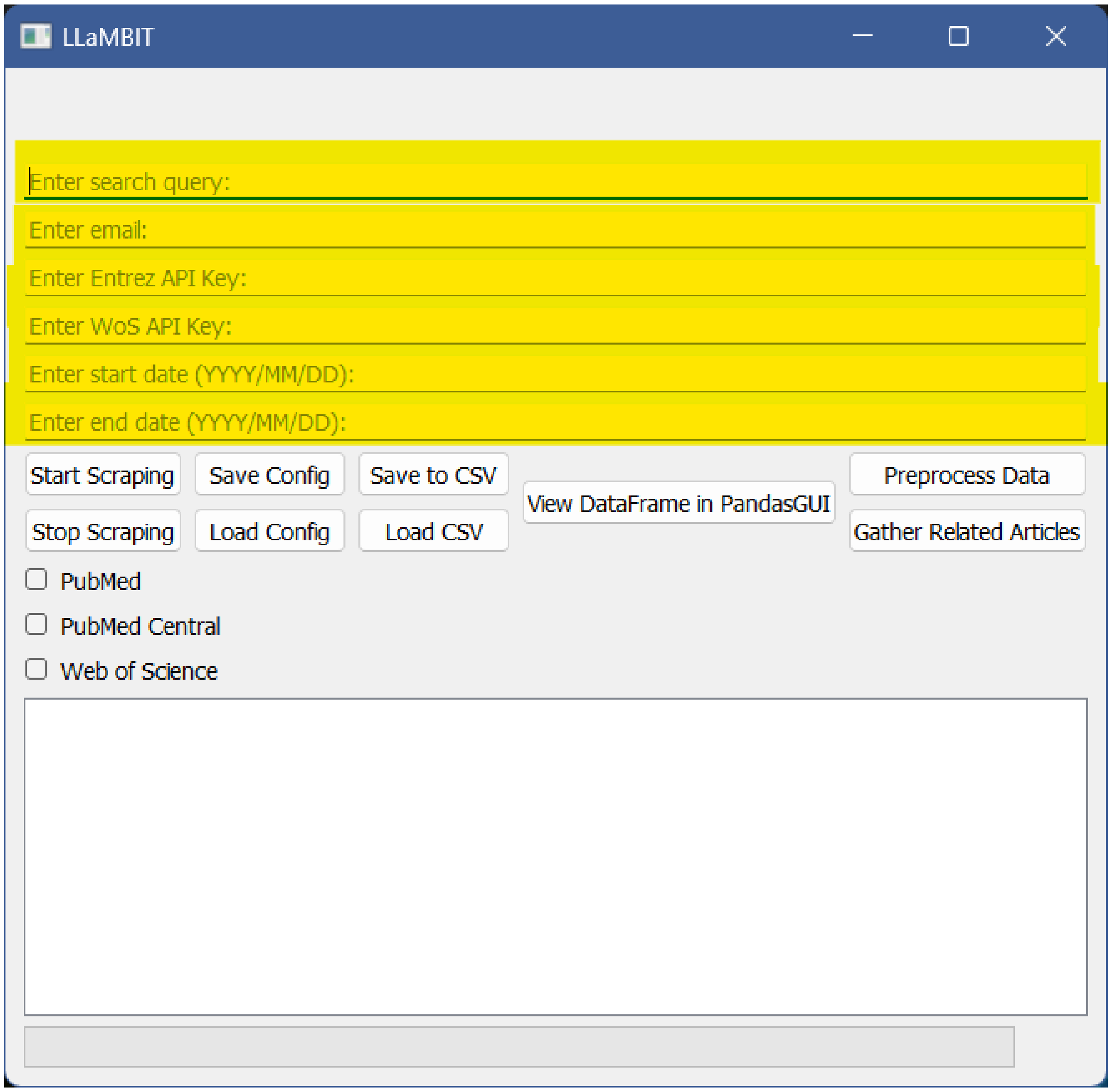


Figure 3-2: LLamBIT Input Field Options

Chapter 4

Configuration Management

Configuration management in LLaMBIT is critical for customizing and preserving user preferences and application settings. This chapter discusses the implementation details of configuration management, including code snippets for clarity.

4.1 Configuration File Structure

The application uses a JSON file (`config.json`) to store configuration settings:

- **API Keys**: Encrypted keys for accessing external data sources.
- **Search Parameters**: User-defined parameters for data retrieval.
- **Scraper Selection**: User choices for which scrapers to use.

4.2 Loading Configuration

Configuration is loaded at application start-up or upon user request:

```
def load_config(self, event):
    logging.debug("Entering␣load_config␣method.")
    try:
        with open('config.json', 'r') as f:
            config = json.load(f)
            # Load and decrypt API keys, set UI elements
            # Error handling for missing or corrupted
              ↪ config
    except FileNotFoundError:
        logging.warning("Config␣file␣was␣not␣found.")
        QMessageBox.warning(self, "Warning", "Config␣file␣
          ↪ not␣found!")
```

4.3 Saving Configuration

Users can save their current settings, which are then encrypted and stored:

```
def save_config(self):
    config = {
        'search_term': self.search_term.text(),
        'email': self.email_input.text(),
        'api_keys': {
            'entrez': self.encrypt_api_key(self.
              ↪ entrez_api_key_input.text()),
            'wos': self.encrypt_api_key(self.
              ↪ wos_api_key_input.text())
        },
        'scrapers': {
            'pubmed': self.pubmed_checkbox.isChecked(),
            'pubmed_central': self.pubmed_central_checkbox
              ↪ .isChecked(),
            'wos': self.wos_checkbox.isChecked()
        }
    }
    with open('config.json', 'w') as f:
        json.dump(config, f)
```

4.4 Encryption and Decryption Process

LLaMBIT uses symmetric encryption (Fernet) for securing API keys. The process includes:

4.4.1 Generating Salt and Hash

```
# Derive key from passphrase with salt
def derive_key_from_passphrase(self, passphrase, salt):
    kdf = PBKDF2HMAC(
        algorithm=hashes.SHA256(),
        length=32,
        salt=salt.encode('utf-8'),
        iterations=100000,
        backend=default_backend()
    )
    key = base64.urlsafe_b64encode(kdf.derive(passphrase.
      ↪ encode()))
    return key
```

4.4.2 Encrypting Configuration Data

```
# Encrypting API key
def encrypt_api_key(self, api_key, passphrase):
    salt = os.urandom(16)
    key = self.derive_key_from_passphrase(passphrase, salt
      ↪ )
    cipher = Fernet(key)
    encrypted_api_key = cipher.encrypt(api_key.encode())
    return encrypted_api_key
```

4.4.3 Decrypting Configuration Data

```
# Decrypting API key
def decrypt_api_key(self, encrypted_api_key, passphrase,
  ↪ salt):
    key = self.derive_key_from_passphrase(passphrase, salt
      ↪ )
    cipher = Fernet(key)
    decrypted_api_key = cipher.decrypt(encrypted_api_key).
      ↪ decode()
    return decrypted_api_key
```

4.5 Handling Configuration in the UI

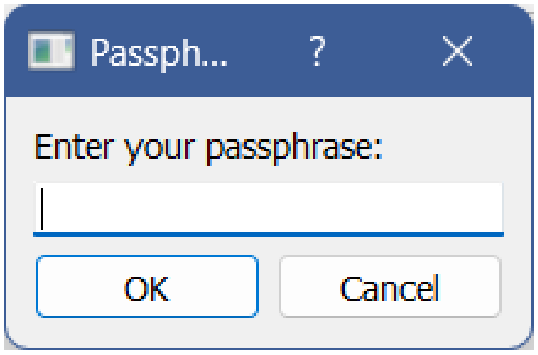
The UI provides options for entering a passphrase, saving encrypted data, and loading and decrypting configuration settings.

4.6 Security Considerations

- **Key Derivation**: Uses PBKDF2HMAC with SHA256 for robust key derivation.

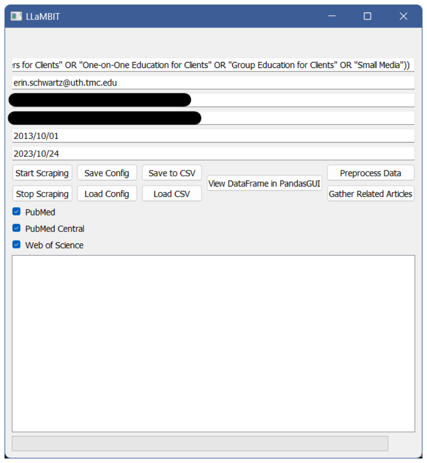
- **Data Integrity**: Ensures that only users with the correct passphrase can access encrypted data.

4.7 GUI Screenshots



Enter your passphrase:

Figure 4-1: On first startup, users will need to provide a one-time password to secure API keys and program settings.



rs for Clients" OR "One-on-One Education for Clients" OR "Group Education for Clients" OR "Small Media"))

erin.schwartz@uth.tmc.edu

[redacted]

[redacted]

2013/10/01

2023/10/24

✓ PubMed

✓ PubMed Central

✓ Web of Science

Figure 4-2: Users are able to provide custom queries for PubMed, PubMedCentral, and WoS. The config file will save these queries for use at a later date, along with private credentials like API keys (blacked out for security).

Chapter 5

Scraper Functionality

LLaMBIT employs specialized scrapers to automate the retrieval of medical bibliographic data from the Entrez (`bio` Python library) API and the Web of Science API. This chapter presents an in-depth look at the technical aspects and functionalities of these scraper classes.

5.1 Overview of Scraper Classes

LLaMBIT incorporates several scraper classes, each designed to interact with specific databases:

1. **PubMedScraper**: Fetches data from PubMed using the Entrez API.
2. **PubMedCentralScraper**: Retrieves articles from PubMed Central, also via the Entrez API.
3. **WoSJournalScraper**: Connects to the Web of Science API for data collection.

5.2 Interaction with Entrez API

5.2.1 Initializing PubMedScraper

The basis of the Entrez API, and both the PubMed and PubMedCentral scrapers, utilizes an XML interface to gather journal data. At present, this feature is hard-coded into the Python code, and is used preferentially to `HTML` or `JSON`, but is by no means required. The primary reason for this choice is to standardize data scraping for use with `BeautifulSoup`, along with ease of export into a `Pandas` dataframe.

```
class PubMedScraper(EntrezScraper):
    def __init__(self, email, api_key, search_term=None,
      ↪ config=None):
        super().__init__(search_term, email, api_key,
          ↪ dbase='pubmed', config=config)
        # Additional initializations
```

5.2.2 Data Retrieval Process

```
# Example of retrieving data from PubMed using the Entrez
   ↪ API
handle = Entrez.esearch(db="pubmed", term=search_term,
   ↪ retmax=10)
result = Entrez.read(handle)
handle.close()
```

5.2.3 Recursive Article Searching

Recursive article searching in the PubMedScraper is a crucial feature that enables the retrieval of not only directly relevant articles but also those that are related to them, expanding the scope of the search. This process leverages the Entrez API's `elink` functionality.

5.2.4 Fetching Related Articles

```
# Recursive function to fetch related articles using
    ↪ Entrez elink
def fetch_elink_articles(self, pubmed_id):
    handle = Entrez.elink(dbfrom="pubmed", id=pubmed_id,
        ↪ linkname="pubmed_pubmed")
    records = Entrez.read(handle)
    related_articles = [article for record in records for
        ↪ article in record['LinkSetDb'][0]['Link']]
    return related_articles
```

This function fetches related articles for a given PubMed ID. It uses the Entrez ‘elink‘ method to find articles linked to the specified ID. The returned list includes the IDs of articles that are deemed relevant in the context of the original search term.

5.2.5 Processing Related Articles

```
# Function to process related articles
def get_related_articles(self, pubmed_ids):
    for pubmed_id in pubmed_ids:
        related_articles = self.fetch_elink_articles(
            ↪ pubmed_id)
        # Additional processing of related articles
```

The `get_related_articles` function iterates over a list of PubMed IDs, calling `fetch_elink_articles` for each. It is designed to recursively expand the search scope by fetching articles related to each ID in the list. The resulting data offers a more comprehensive view of the research landscape around the initial search term.

5.2.6 Understanding Recursive Searching

Recursive article searching is a powerful method to expand the breadth of literature research. In the context of medical research, where interconnections between studies are often crucial, this method provides a way to uncover these links. The LLaMBIT application's PubMedScraper uses this approach to offer users a more exhaustive set of search results, which can be critical for comprehensive literature reviews or meta-analyses.

Mechanism of Action

The process begins with a set of base articles obtained through a primary search. For each article in this set, the PubMedScraper uses the Entrez 'elink' function to find other articles that are related to it. This is achieved by querying the PubMed database for articles that are linked in the context of citations, references, or thematic similarities. The IDs of these related articles are then collected.

Dataframe Structure for Related Articles

The collected IDs of related articles are then used to create or expand a dataframe, which typically includes columns like 'Article Title,' 'Authors,' 'Publication Date,' 'Abstract,' and 'DOI.' The structure of this dataframe is similar to the primary search results, allowing for seamless integration and comparison.

Challenges and Considerations

One of the challenges in recursive article searching is managing the depth and breadth of the search. Since each article can link to several others, there is a potential for exponential growth in the number of articles fetched. To manage this, LLaMBIT may implement limits on the depth of recursion or the total number of articles fetched.

Additionally, ensuring the relevance of fetched articles is crucial. The Entrez API provides a degree of filtering based on how articles are linked, but further filtering based on the original search criteria within LLaMBIT can enhance the precision of the results.

5.2.7 Error Handling and Logging

```
# Handling errors during data retrieval
try:
    # Data retrieval logic
except Exception as e:
    logging.error(f"Error␣during␣scraping:␣{e}")
    raise
```

5.3 Interaction with Web of Science API

5.3.1 Configuring WoSJournalScraper

```
class WoSJournalScraper(BaseScraper):
    def __init__(self, api_key, search_term, database_id='
      ↪ WOS'):
        super().__init__(search_term, api_key)
        self.configuration = self.configure_api(api_key)
        self.search_api_instance = self.init_search_api()
        # Additional configurations
```

The ‘WoSJournalScraper‘ is designed to interact with the Web of Science API, which differs significantly from the Entrez API in terms of query structure and data retrieval methods. The configuration of this scraper involves setting up the API key and initializing the search API instance.

5.3.2 API Request and Response Handling

```
# Example of making a request to the Web of Science API
```

```
response = self.search_api_instance.search(database_id,
    ↪ search_term)
if response.status_code == 200:
    # Process successful response
else:
    logging.error(f"API␣request␣failed␣with␣status:␣{
        ↪ response.status_code}")
```

This snippet illustrates the process of making a request to the WoS API. Unlike Entrez, which uses an XML-based interface, WoS often employs different data formats and query parameters.

5.3.3 Query System Differences

The WoS API's query system fundamentally differs from the Entrez-based queries in several ways:

- **Query Language and Structure**: WoS queries often require a different syntactical structure, focusing on specific fields and using different operators.
- **Data Format and Response Structure**: The response from WoS might be in formats other than XML, such as JSON, necessitating different parsing techniques.
- **Rate Limiting and API Usage**: The WoS API might have different rate limiting policies and usage restrictions, which must be accounted for in the scraper.

5.3.4 Adapting Query Input for WoS Scraper

To effectively utilize the WoS API, the query input system in LLaMBIT needs to be adaptable. This involves:

- **Dynamic Query Construction**: Building queries that are compatible with the WoS API's structure, including the correct use of fields and operators.

- **Flexible Data Handling**: Ensuring the scraper can process and normalize data from various formats provided by WoS.

- **User-Friendly Interface**: Providing a user interface that allows input of WoS-specific query parameters without overwhelming the user.

The `convert_WoS_query` method in LLaMBIT's main module is a key function that translates user-inputted search terms into a format compatible with the WoS API.

```python
def convert_WoS_query(self, query):
    # Code snippet from LLaMBIT
```

This function reformats the input query by breaking it down into individual components, such as phrases, logical operators, and standalone words, and then restructuring them to align with the WoS query syntax.

Detailed Explanation of convert_WoS_query

Formatting Individual Components : The method identifies different types of components in the query: - Phrases are enclosed in double quotes and reformatted as `ALL=({phrase})`. - Logical operators `AND`, `OR`, `NOT`) are retained as they are crucial for query logic. - Individual words are formatted as `(ALL=({word}))`.

Handling OR Statements : Special attention is given to OR statements. The method groups all terms following the last occurrence of `AND` in an `ALL` clause. This step ensures that the `OR` logic is correctly interpreted by the WoS API.

Compiling the Formatted Query : The formatted components are then combined to form the complete query, which is compatible with the WoS API's search syntax.

Significance of convert_WoS_query

This method is crucial for ensuring that the search queries inputted by users are accurately translated for the WoS API. Without this adaptation, the scraper would be unable to retrieve relevant results from WoS, as the query syntax and structure significantly differ from the Entrez API used for PubMed and PubMed Central.

Challenges and Considerations

One of the challenges in implementing `convert_WoS_query` is ensuring that all possible query formats and logical structures inputted by users are correctly translated. This requires a robust understanding of both the WoS API's query syntax and the diverse ways users might format their queries.

5.3.5 Data Extraction and Preprocessing

```
def extract_data(self, record):
    # Extract and preprocess data like authors, abstract,
      ↪ etc.
    # Return a structured data object
```

Data extraction in the WoS scraper involves parsing the specific data structure returned by the WoS API. This might include extracting fields like article titles, authors, and publication dates, which may be presented differently compared to Entrez responses.

5.3.6 Extracted Dataframe Structure

```
# Example of the structure of the extracted dataframe
dataframe_columns = ['Article␣Title', 'Authors', '
   ↪ Publication␣Date', 'Abstract', 'DOI']
extracted_dataframe = pd.DataFrame(columns=
   ↪ dataframe_columns)
```

```
# Data extraction and filling the dataframe
```

The structure of the extracted DataFrame for WoS data must align with the specific fields and formats provided by the WoS API. This structure ensures consistency and ease of use when analyzing the scraped data.

5.4 Integration with User Interface

The scrapers are integrated with the UI, providing real-time feedback:

- **Progress Updates**: Reflect the progress of scraping operations in the UI.
- **Visual Feedback**: Display animations and messages during data retrieval.
- **Error Alerts**: Communicate any scraping errors to the user through dialog boxes.

5.5 Future Development and Improvement Opportunities

The current implementation of LLaMBIT's scrapers, while effective, offers several opportunities for enhancement. The following offer potential areas for future development:

5.5.1 Implementing Advanced Rate Limiting

Rationale: To avoid overloading the external APIs and to comply with their usage policies, implementing sophisticated rate limiting is crucial.

Execution Strategy:

- Integrate a rate limiter that dynamically adjusts the frequency of requests based on the API's response (e.g., using HTTP headers to detect rate limits).
- Develop a queue system that can efficiently manage and schedule API requests.

5.5.2 Enhancing Data Extraction with NLP Techniques

Rationale: Natural Language Processing (NLP) can be used to extract more nuanced information from article abstracts and titles, such as sentiment, key phrases, or contextual relationships.

Execution Strategy:

- Integrate NLP libraries like NLTK or spaCy to analyze the text data.
- Apply NLP models to identify key themes, trends, or sentiment in the research articles.

5.5.3 Expanding Scraper Functionality to Additional Databases

Rationale: Adding scrapers for more databases would make LLaMBIT a more comprehensive tool for literature review.

Execution Strategy:

- Identify and integrate APIs of other relevant medical databases (e.g., Scopus, Cochrane Library).
- Develop additional scraper classes tailored to the specific formats and requirements of these new databases.

5.5.4 Implementing a More Robust Error Recovery System

Rationale: Enhancing the error recovery mechanism would make the scrapers more resilient to failures, reducing the risk of data loss or incomplete searches.

Execution Strategy:

- Develop a system to save the state of the scraping process periodically.
- Implement recovery logic to resume scraping from the last saved state in case of a failure.

5.5.5 User-Configurable Scraping Parameters

Rationale: Allowing users to configure scraping parameters (like depth of search, specific fields to extract) would make the tool more flexible and user-friendly.

Execution Strategy:

- Add UI elements for users to specify their scraping preferences.
- Modify scraper classes to adapt their behavior based on these user-defined parameters.

5.5.6 Integration with RStudio for Advanced Data Analysis

Rationale: Integrating with RStudio can provide advanced data analysis capabilities, such as employing Random Forest or Boruta algorithms for more sophisticated insights.

Execution Strategy:

- Develop an interface or plugin to connect LLaMBIT's data output directly with RStudio.
- Implement functions or scripts within LLaMBIT that can prepare and export data in a format readily usable in RStudio for complex analyses.

5.5.7 Tableau Integration for Enhanced Data Visualization

Rationale: Integrating with Tableau would allow users to perform more robust and interactive visualizations of the scraped data.

Execution Strategy:

- Create a data export feature in LLaMBIT that formats data specifically for Tableau compatibility.
- Develop tutorials or templates in Tableau that are tailored to the types of data extracted by LLaMBIT.

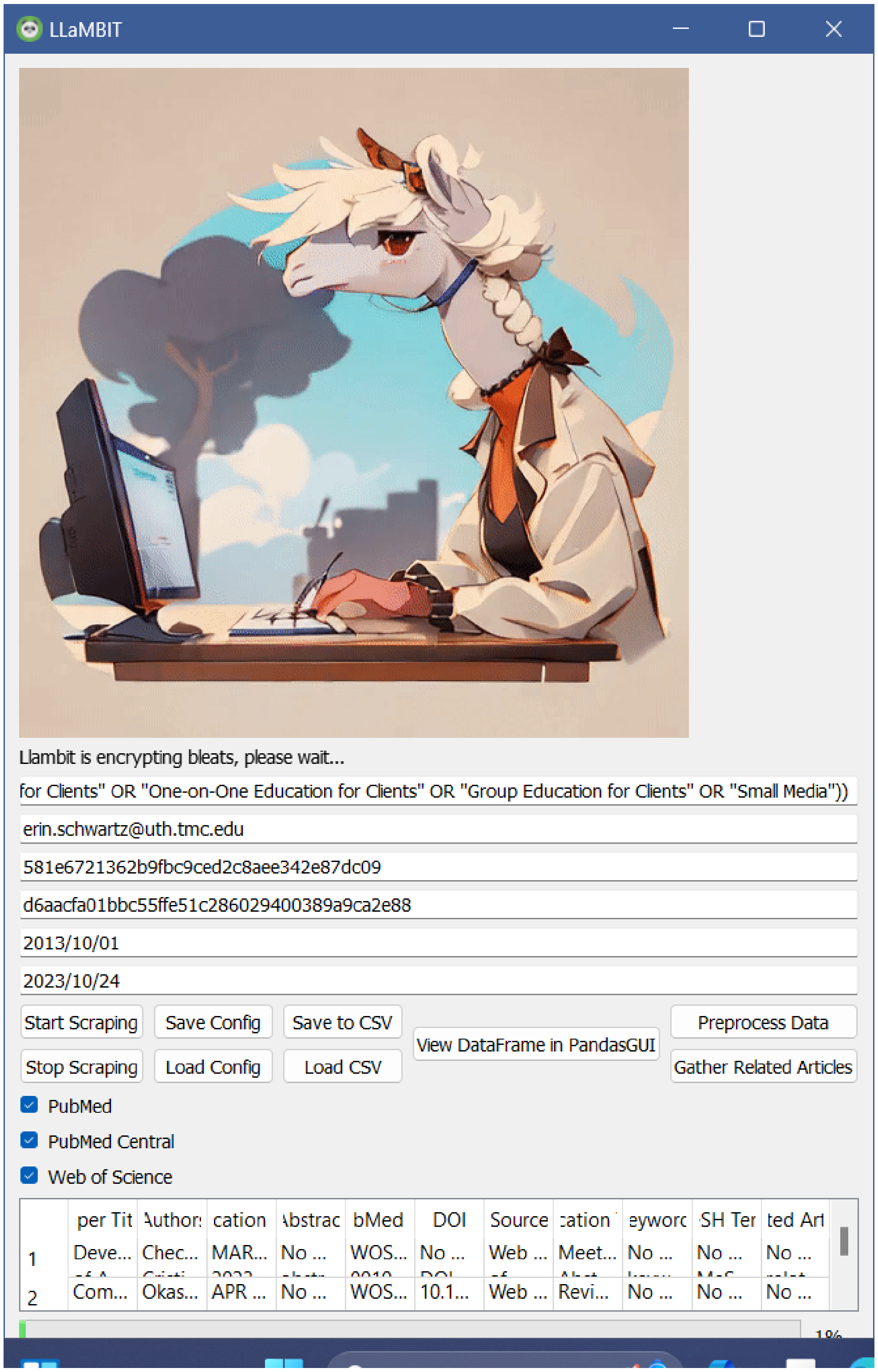
5.5.8 Cloud-based Operation for Scalability and Knowledge Database Building

Rationale: Moving to a cloud-based platform could enable LLaMBIT to run more extensive scraping operations over longer durations, support multiple users, and accumulate a knowledge database for enhanced NLP and data analysis.

Execution Strategy:

- Adapt the LLaMBIT architecture to operate in a cloud environment, such as AWS or Azure.
- Implement features for multi-user support and concurrent scraping operations.
- Develop a system for accumulating and storing scraped data in a cloud-based knowledge database, which can be continually enriched and utilized for advanced NLP applications.
- Explore integration with platforms like GitHub for code operation and version control.

5.6 GUI Screenshots



Llambit is encrypting bleats, please wait...

for Clients" OR "One-on-One Education for Clients" OR "Group Education for Clients" OR "Small Media"))

erin.schwartz@uth.tmc.edu

581e6721362b9fbc9ced2c8aee342e87dc09

d6aacfa01bbc55ffe51c286029400389a9ca2e88

2013/10/01

2023/10/24

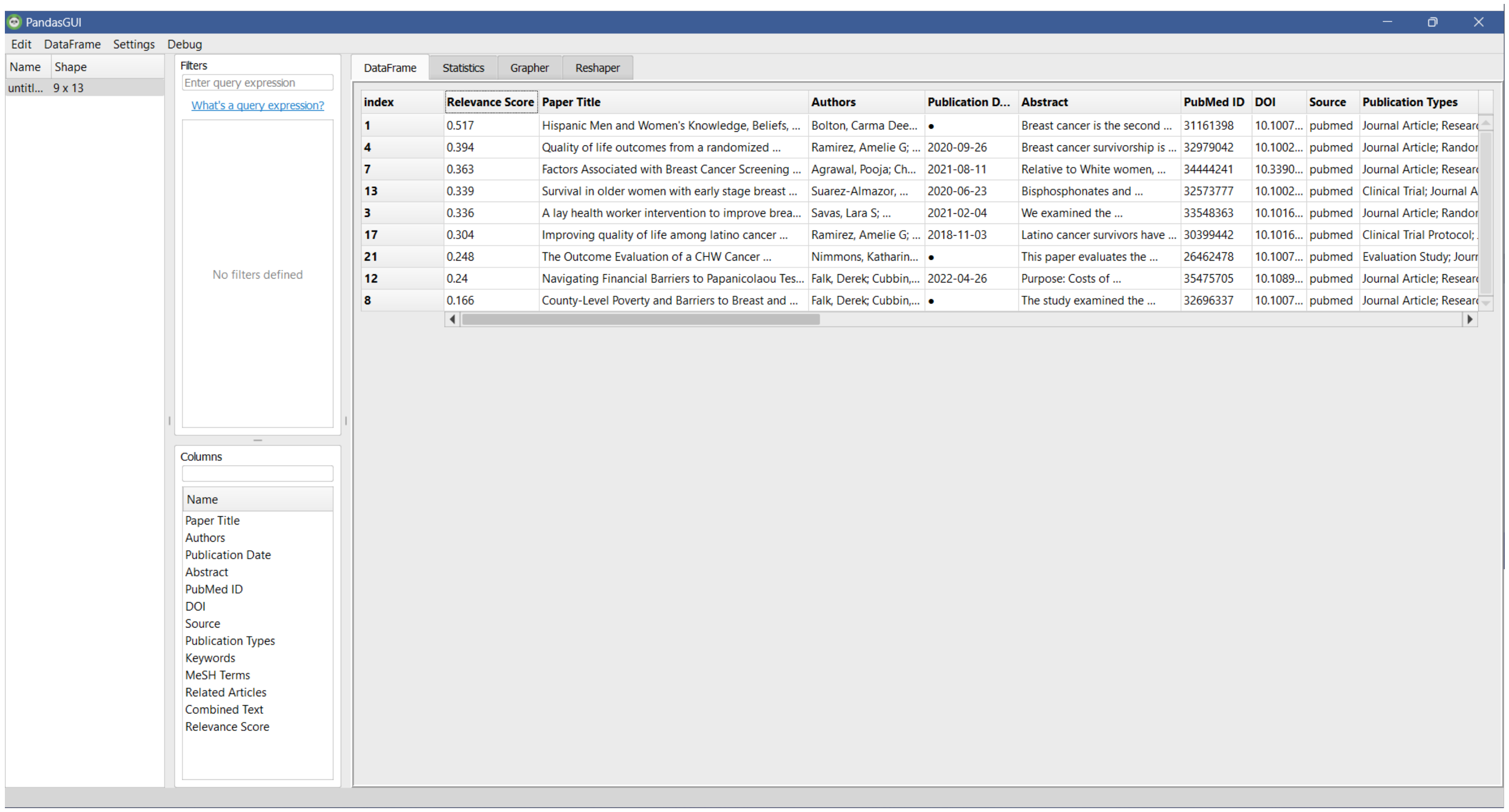
✓ PubMed

✓ PubMed Central

✓ Web of Science

	per Tit	Author:	cation	Abstrac	bMed	DOI	Source	cation	eyword	SH Ter	ted Art
1	Deve...	Chec...	MAR...	No ...	WOS...	No ...	Web ...	Meet...	No ...	No ...	No ...
2	Com...	Okas...	APR ...	No ...	WOS...	10.1...	Web ...	Revi...	No ...	No ...	No ...

Figure 5-1: LLaMBIT is reticulating splines, please wait... This sample GUI output shows the LLaMBIT tool in action, interfacing with three databases behind the GUI and scraping data into a Pandas dataframe. The data can later be pre-processed and visualized in the PandasGUI interface.



index	Relevance Score	Paper Title	Authors	Publication D...	Abstract	PubMed ID	DOI	Source	Publication Types
1	0.517	Hispanic Men and Women's Knowledge, Beliefs, ...	Bolton, Carma Dee...	●	Breast cancer is the second ...	31161398	10.1007...	pubmed	Journal Article; Researc
4	0.394	Quality of life outcomes from a randomized ...	Ramirez, Amelie G; ...	2020-09-26	Breast cancer survivorship is ...	32979042	10.1002...	pubmed	Journal Article; Randor
7	0.363	Factors Associated with Breast Cancer Screening ...	Agrawal, Pooja; Ch...	2021-08-11	Relative to White women, ...	34444241	10.3390...	pubmed	Journal Article; Researc
13	0.339	Survival in older women with early stage breast ...	Suarez-Almazor, ...	2020-06-23	Bisphosphonates and ...	32573777	10.1002...	pubmed	Clinical Trial; Journal A
3	0.336	A lay health worker intervention to improve brea...	Savas, Lara S; ...	2021-02-04	We examined the ...	33548363	10.1016...	pubmed	Journal Article; Randor
17	0.304	Improving quality of life among latino cancer ...	Ramirez, Amelie G; ...	2018-11-03	Latino cancer survivors have ...	30399442	10.1016...	pubmed	Clinical Trial Protocol;
21	0.248	The Outcome Evaluation of a CHW Cancer ...	Nimmons, Katharin...	●	This paper evaluates the ...	26462478	10.1007...	pubmed	Evaluation Study; Journ
12	0.24	Navigating Financial Barriers to Papanicolaou Tes...	Falk, Derek; Cubbin,...	2022-04-26	Purpose: Costs of ...	35475705	10.1089...	pubmed	Journal Article; Researc
8	0.166	County-Level Poverty and Barriers to Breast and ...	Falk, Derek; Cubbin,...	●	The study examined the ...	32696337	10.1007...	pubmed	Journal Article; Researc

Figure 5-2: The PandasGUI interface becomes available after initial data scraping, after gathering related articles, and after pre-processing results to generate filtered data and relevance scores. In this example, results below scores of 0.15 are filtered and only the remaining results are shown. A number of fields are available to the researcher, including both PubMed ID and DOI, which make locating the resource significantly easier.

5.7 Example Data Output

The following table is an example output from LLaMBIT, via the PandasGUI integration after pre-processing data. The following PubMed query was used to generate the original results, along with gathering related articles. Filtering was applied at a relevance threshold of 0.15, with articles scoring below this threshold being filtered.

Paper Title	Authors	Publication Date	Abstract	PubMed ID	DOI	Source	Publication Types	Keywords	MeSH Terms	Related Articles	Combined Text	Relevance Score
Hispanic Men and Women's Knowledge, Beliefs, Perceived Susceptibility, and Barriers...	Bolton, Carma Deem; Sunil, T S; Hurd, Thelma; Guerra, Hector	...	Breast cancer is the second leading cause...	31161398	10.1007/s10900-019-00682-1	pubmed	Journal Article; Research Support...	Barriers; Breast cancer knowledge...	Adult; Age Factors; Aged; Breast Neoplasms...	['35731462', '25628831',...	hispanic men woman knowledge belief...	0.5168
Quality of life outcomes from a randomized...	Ramirez, Amelie G; Muñoz, Edgar...	2020-09-26	Breast cancer survivorship is a life-long...	32979042	10.1002/cam4.3272	pubmed	Journal Article; Randomized...	behavioral science; breast cancer...	Adult; Aged; Breast Neoplasms...	['17695343', '28935442',...	quality life outcome randomized controlled...	0.3938
Factors Associated with Breast Cancer Screening...	Agrawal, Pooja; Chen, Tzuan A; McNeill...	2021-08-11	Relative to White women, African...	34444241	10.3390/ijerph18168494	pubmed	Journal Article; Research Support...	African American women; Andersen...	Black or African American; Aged...	['37423346', '35731462',...	factor associated breast cancer...	0.3633
Survival in older women with early stage breast cancer...	Suarez-Almazor, Maria E; Herrera, Raul...	2020-06-23	Bisphosphonates and denosumab, as adjuvant...	32573777	10.1002/cncr.33035	pubmed	Clinical Trial; Journal Article...	adjuvant therapy; bisphosphonates...	Aged; Aged, 80 and over; Antineoplastic...	['36408625', '28727345',...	survival older woman early stage breast...	0.3386
A lay health worker intervention to improve breast...	Savas, Lara S; Atkinson, John S; Figueroa-Solis...	2021-02-04	We examined the effectiveness of a lay...	33548363	10.1016/j-ypmed-2021-106446	pubmed	Journal Article; Randomized Controlled...	Breast and cervical cancer screening...	Adult; Breast Neoplasms; Early Detection...	['27513995', '35954824',...	lay health worker intervention improve...	0.3362

Table 5.1: LLaMBIT Sample Output: Breast Cancer in Texas

Chapter 6

Article Preprocessing

Article preprocessing in LLaMBIT is a critical step in preparing the scraped data for efficient querying and analysis. This chapter discusses the methods and techniques used in the preprocessing phase.

6.1 Preprocessing Steps

The `ArticlePreprocessor` class in LLaMBIT performs several key steps to preprocess the data:

1. **Text Cleaning**: The extracted text data, including abstracts and titles, undergoes cleaning to remove special characters and stopwords. This step ensures the text is in a consistent format for analysis.

2. **Text Combination**: Titles, abstracts, and keywords are combined to provide a comprehensive context for each article.

3. **TF-IDF Vectorization**: The cleaned and combined text is transformed into TF-IDF vectors. This process converts the textual data into a numerical format suitable for machine learning and similarity computations.

4. **Relevance Scoring**: Articles are scored based on their relevance to the search terms. This score is used to filter and sort articles in subsequent steps.

6.2 Overview

The `ArticlePreprocessor` class in LLaMBIT handles the preprocessing of articles. It involves cleaning text, handling NaN values, and converting data into a machine-readable format using techniques like TF-IDF vectorization.

6.2.1 Initialization

```
class ArticlePreprocessor:
    def __init__(self):
        self.vectorizer = TfidfVectorizer(stop_words='
            ↪ english')
        self.lemmatizer = WordNetLemmatizer()
        self.articles_df = None
        self.tfidf_matrix = None
```

This code snippet shows the initialization of the ArticlePreprocessor class, which sets up the necessary tools for text processing, including a TF-IDF vectorizer and a WordNet lemmatizer.

6.2.2 Cleaning Text

```
def _clean_text(self, text):
    text = re.sub(r'[^a-zA-Z\s]', '', text, re.I | re.A)
    text = text.lower()
    tokens = text.split()
    tokens = [self.lemmatizer.lemmatize(token) for token
        ↪ in tokens]
    tokens = [token for token in tokens if token not in
        ↪ self.stop_words]
    return '␣'.join(tokens)
```

This function cleans the text data by removing special characters, converting it to lowercase, tokenizing, lemmatizing, and removing stopwords. It's a crucial step in ensuring the quality of the data for analysis.

6.2.3 Preprocessing Dataframe

```
def preprocess(self, df):
    logging.info("Starting␣preprocessing␣of␣data")
    self.articles_df = df.copy()
    df['Relevance␣Score'] = 0.0
    self.articles_df['Combined␣Text'] = self.articles_df['
      ↪ Paper␣Title'] + '␣' + self.articles_df['Keywords'
      ↪ ]
    try:
        self.articles_df['Combined␣Text'] += '␣' + self.
          ↪ articles_df['MeSH␣Terms']
    except KeyError:
        pass
    self.articles_df['Combined␣Text'] = self.articles_df['
      ↪ Combined␣Text'].apply(self._clean_text)
    self.tfidf_matrix = self.vectorizer.fit_transform(self
      ↪ .articles_df['Combined␣Text'])
    return self.articles_df
```

In this function, the DataFrame is preprocessed by combining the 'Paper Title' and 'Keywords' columns, cleaning the text, and converting it into TF-IDF vectors. This step transforms the raw data into a format suitable for further analysis, like querying and relevance scoring.

6.2.4 TF-IDF Vectorization

TF-IDF (Term Frequency-Inverse Document Frequency) vectorization is a critical part of preprocessing. It converts the text data into numerical vectors, which can be used to measure the relevance of articles based on the user's search terms.

6.2.5 Future Development

Looking ahead, there are several areas where article preprocessing in LLaMBIT could be further improved:

- **Advanced NLP Techniques**: Integrating more sophisticated NLP techniques for better context understanding and sentiment analysis.
- **Language Support**: Adding support for multiple languages to cater to a broader range of research articles.
- **Automated Data Cleaning Enhancements**: Implementing more advanced algorithms for automated data cleaning and normalization.
- **Scalability for Large Datasets**: Optimizing the preprocessing steps for handling larger datasets more efficiently.
- **Customizable Preprocessing Steps**: Allowing users to configure and customize the preprocessing steps based on their specific needs.

6.3 Error Handling and Logging

The preprocessing phase includes robust error handling and logging:

- **Logging**: Key events in the preprocessing steps are logged for monitoring and debugging purposes.
- **Exception Management**: Exceptions and errors encountered during preprocessing are handled gracefully, with appropriate user notifications.

Chapter 7

Query Parsing and Execution

Query Parsing and Execution is a fundamental aspect of LLaMBIT, enabling users to search and retrieve relevant medical bibliographic data efficiently. This chapter describes the mechanisms and algorithms behind query parsing and execution in the application, along with specific technical details and code snippets.

7.1 Query Parsing Process

The query parsing process in LLaMBIT involves the following steps:

1. **Input Query Parsing**: The user's input query is parsed into individual search terms. This parsing considers logical operators like `AND` and `OR` to correctly interpret the intended search criteria.

2. **Text Cleaning**: Each term in the query is cleaned of any special characters and stopwords to ensure consistency with the preprocessing format.

3. **Term Vectorization**: The cleaned terms are then vectorized using the same TF-IDF approach applied in the data preprocessing phase.

7.1.1 Parsing Strategy in LLaMBIT

```
def _parse_query(self, query):
    # Code snippet demonstrating the parsing of the query
```

This function demonstrates LLaMBIT's approach to parsing user queries. It breaks down the query into its constituent terms and logical operators, preparing them for effective searching.

7.2 Execution of Queries

Once the query is parsed and vectorized, the application executes the search:

- **Cosine Similarity Computation**: The application calculates the cosine similarity between the query vector and the TF-IDF vectors of the preprocessed articles.
- **Relevance Scoring**: Articles are scored based on their cosine similarity to the query, with higher scores indicating greater relevance.
- **Threshold Filtering**: Articles with relevance scores above a certain threshold are selected as relevant to the query.
- **Results Presentation**: The relevant articles, sorted by their relevance scores, are displayed to the user in the application's UI.

7.2.1 Executing Searches in LLaMBIT

```
def query_articles(self, query, threshold=0.15):
    # Code for executing searches based on the parsed
      ↪ query
```

This method exemplifies how the parsed and vectorized query is used to search within the preprocessed articles database, employing cosine similarity for determining the relevance of each article.

7.3 Integration with the User Interface

The querying process is closely integrated with the UI:

- **Search Term Input**: Users input their search queries through a dedicated text field in the UI.
- **Feedback on Query Progress**: The application provides real-time feedback on the query execution progress.
- **Display of Results**: The results of the query are presented in a readable and interactive format, allowing users to explore the retrieved articles.

7.4 Error Handling and Logging

- **Exception Management**: Errors during the query execution are handled gracefully, with appropriate messages displayed to the user.
- **Logging**: Key events during the query parsing and execution are logged for monitoring and troubleshooting.

Chapter 8

Animation and Feedback Mechanisms

Animation and feedback mechanisms in LLaMBIT play a critical role in enhancing user experience, especially during data processing and scraping operations. This chapter provides a detailed overview of these mechanisms, explaining their technical implementation and integration into the application's user interface.

8.1 Implementation of Animation

LLaMBIT uses Qt's animation framework to implement engaging and informative animations. These animations are designed to provide visual feedback to users during various operations like data scraping and processing.

8.1.1 Setting Up Animations

```
self.fade_in = QPropertyAnimation(self.opacity_effect, b"
    ↪ opacity")
self.fade_in.setDuration(1000)
self.fade_in.setStartValue(0)
self.fade_in.setEndValue(1)
self.fade_out = QPropertyAnimation(self.opacity_effect, b"
    ↪ opacity")
self.fade_out.setDuration(1000)
self.fade_out.setStartValue(1)
self.fade_out.setEndValue(0)

self.animation_group = QSequentialAnimationGroup()
self.animation_group.addAnimation(self.fade_in)
self.animation_group.addPause(2000)
self.animation_group.addAnimation(self.fade_out)
```

In this implementation, fade-in and fade-out animations are created using `QPropertyAnimation`. These animations are then grouped sequentially, providing a smooth transition effect for the loading messages.

8.1.2 Triggering Animations

```
self.animation_group.finished.connect(self.change_message)
self.messageChanged.connect(self.start_animation)
```

Animations are triggered by specific events in the application. The `finished` signal of the animation group is connected to the `change_message` method, allowing for a new message to be displayed once an animation cycle completes.

8.2 Feedback Mechanisms

Feedback mechanisms in LLaMBIT are crucial for keeping users informed about the application's state and progress.

8.2.1 Loading Messages

```
self.loading_messages = [
    "scraping␣the␣web", "reticulating␣splines", "parsing␣
       ↪ results", ...
]
```

A variety of loading messages are used to provide a dynamic and engaging way to inform users about ongoing processes. These messages add an element of interactivity and humor to the user experience.

8.2.2 GIF Integration for Visual Feedback

```
self.movie = QMovie("resources/scrapingPleaseWait.gif")
self.gif_label.setMovie(self.movie)
self.movie.start()
```

A GIF animation is integrated to provide visual feedback during lengthy operations like data scraping. This approach helps in maintaining user engagement and effectively communicates the ongoing process.

8.3 Error Handling and Logging

Proper error handling and logging are essential for a robust application. LLaMBIT implements these mechanisms to ensure any issues are gracefully managed and recorded.

8.3.1 Exception Management

```
except Exception as e:
    QMessageBox.critical(self, "Error", f"Failed␣to␣load␣
       ↪ data.␣Error:␣{e}")
```

Errors encountered during operations are caught and displayed to the user through message boxes. This approach helps in making the application user-friendly and transparent about its state.

8.3.2 Logging for Monitoring

```
logging.info("Starting␣scraping...")
logging.error(f"API␣request␣failed␣with␣status:␣{response.
    ↪ status_code}")
```

Key events, especially during query execution and data processing, are logged for monitoring and troubleshooting purposes. Logging aids in maintaining the application's health and assists in debugging.

Chapter 9

Error Handling and Logging

Error handling and logging are crucial aspects of LLaMBIT, ensuring the application's stability and providing insights into its operations. This chapter explores the strategies and implementations for handling errors and logging activities in LLaMBIT.

9.1 Error Handling Mechanisms

LLaMBIT implements several mechanisms to handle errors gracefully:

- **Exception Handling**: The application uses `try-catch` blocks to manage exceptions that occur during operations like data loading, scraping, and preprocessing.
- **User Alerts**: In case of errors, LLaMBIT provides immediate feedback to users through `QMessageBox` dialogs, displaying critical errors or warnings as needed.
- **Fail-Safe Operations**: The application includes fail-safe checks, such as verifying file existence and integrity, to prevent operations from causing crashes or data loss.

9.1.1 Exception Handling Strategy

```
except Exception as e:
    QMessageBox.critical(self, "Error", f"Failed␣to␣load␣
      ↪ data.␣Error:␣{e}")
```

This code snippet demonstrates how LLaMBIT handles unexpected errors. When an exception is caught, the application displays an error message to the user, detailing what went wrong.

9.1.2 Specific Error Handling Cases

```
if not salt:
    logging.warning(f"{self.__class__.__name__}:␣Salt␣file
      ↪ ␣is␣NONE.")
    return None
```

In cases like missing or invalid data (e.g., a missing salt file for encryption), the application logs a warning and takes appropriate action, such as returning a `None` value, to prevent further issues.

9.2 Logging Activities

LLaMBIT employs Python's logging module to record various activities:

- **Logging Levels**: Different logging levels (INFO, WARNING, ERROR) are used to categorize the significance of logged messages.
- **Log Messages**: The application logs key events, such as the start and completion of data scraping, configuration loading, and any encountered errors.
- **Debugging Aid**: Logging aids in debugging by providing detailed information about the application's execution flow and any issues that arise.

9.2.1 Logging Setup and Levels

```
logging.basicConfig(level=logging.WARNING)
```

LLaMBIT configures its logging to capture warnings and above (i.e., errors). This setup ensures that significant events and issues are logged for later review.

9.2.2 Logging Examples in LLaMBIT

```
logging.info("Starting␣scraping...")
logging.error(f"API␣request␣failed␣with␣status:␣{response.
   ↪ status_code}")
```

These examples show how LLaMBIT logs different types of events. Informational logs track the start of processes like scraping, while error logs capture issues such as failed API requests.

9.3 Integration with Application Components

Error handling and logging are integrated throughout LLaMBIT's components:

- **Scraper Classes**: Each scraper class includes logging for events and error handling for issues during the scraping process.
- **UI Interactions**: User interface components are equipped to handle errors and log user interactions for enhanced usability and troubleshooting.
- **Preprocessing and Query Execution**: The preprocessing and query execution phases include comprehensive logging and error management to ensure data integrity and provide insights into the operations.

Chapter 10

Testing and Validation

Testing and Validation are integral to the development of LLaMBIT, ensuring that the application functions correctly and meets its intended requirements. This chapter describes the processes and practices implemented for testing and validation in LLaMBIT, along with specific technical examples.

10.1 Testing Strategies in LLaMBIT

LLaMBIT employs various testing strategies to ensure the application's robustness and reliability:

1. **Unit Testing**: Testing individual components or units of the application for correctness.

2. **Integration Testing**: Verifying that different modules of the application work together seamlessly.

3. **Functional Testing**: Assessing the application's functionality against its defined requirements.

10.1.1 Unit Testing Example

```
def test_extract_paper_title(soup):
    title = extract_paper_title(soup)
    assert title is not None
```

This example demonstrates a unit test for the `extract_paper_title` function, ensuring that it correctly extracts titles from provided HTML content.

10.1.2 Integration Testing Approach

```
def test_search_integration(search_term, expected_result):
    articles = perform_search(search_term)
    assert expected_result in articles
```

This snippet illustrates an integration test where the search functionality is tested with a specific search term, validating the integration between the querying mechanism and the database.

10.2 Validation Methods

LLaMBIT's validation methods involve ensuring that the application meets user expectations and functional specifications:

- **User Acceptance Testing (UAT)**: Involves end-users testing the application in a real-world scenario to validate its usability and functionality.
- **Regression Testing**: Ensures that new changes or updates do not adversely affect existing functionalities.

10.3 Error Handling and Logging in Testing

Proper error handling and logging are crucial during the testing phase:

```
try:
    # Testing code...
except Exception as e:
    logging.error(f"Test␣failed␣due␣to␣{e}")
```

This code pattern is used in tests to catch and log errors, facilitating troubleshooting and ensuring that issues are properly addressed.

10.4 Continuous Integration and Automated Testing

LLaMBIT employs Continuous Integration (CI) practices, automating the testing process to ensure code quality:

- Automated tests are run on every code commit to detect issues early.
- CI tools like Jenkins or Travis CI can be integrated for automated build and test cycles.

Chapter 11

Deployment and Maintenance

Deployment and Maintenance are critical aspects of LLaMBIT's lifecycle, ensuring the application is reliably available for users and kept up-to-date with the latest features and fixes. This chapter outlines the deployment strategies, maintenance practices, and specific code implementations.

11.1 Anaconda-Based Deployment

Utilizing Anaconda as the development and deployment environment ensures compatibility and ease of setup for LLaMBIT. Python 3.9.7 is specified to maintain support for all required libraries.

11.1.1 Automated Dependency Management

Upon the first execution of LLaMBIT, an automated check is performed to ensure all necessary libraries are installed. This is achieved through a batch script (`firstStart.bat`) that also sets up a dedicated Anaconda virtual environment (`llambit_env`). The script handles the installation of all dependencies via pip.

```
# Contents of firstStart.bat
@echo off
SET venv_name=llambit_env
```

```
REM Check if the virtual environment already exists using
   ↪ Conda
conda env list | findstr /C:"%venv_name%"
IF %ERRORLEVEL% == 0 (
    echo Virtual environment "%venv_name%" already exists.
) ELSE (
    echo Creating virtual environment "%venv_name%"...
    conda create --name %venv_name% python=3.9.7 --yes
    IF %ERRORLEVEL% NEQ 0 GOTO Error
)

REM Activate the virtual environment
call conda activate %venv_name%

REM Check if pip is already installed
conda list pip | findstr pip
IF %ERRORLEVEL% NEQ 0 (
    conda install -c conda-forge pip
    IF %ERRORLEVEL% NEQ 0 GOTO Error
)

REM Check if pipreqs is already installed
conda list pipreqs | findstr pipreqs
IF %ERRORLEVEL% NEQ 0 (
    conda install -c conda-forge pipreqs
    IF %ERRORLEVEL% NEQ 0 GOTO Error
)

pipreqs
```

```
IF %ERRORLEVEL% NEQ 0 GOTO Error

pip install -r requirements.txt
IF %ERRORLEVEL% NEQ 0 GOTO Error

pip install git+https://github.com/Clarivate-SAR/
   ↪ woslite_py_client.git
IF %ERRORLEVEL% NEQ 0 GOTO Error

(echo Success) > install_success.txt
GOTO End

:Error
(echo Failure) > install_success.txt

:End
```

11.1.2 Creating an "Exe-like" Shortcut for Non-Technical Users

To facilitate ease of use, `LLaMBIT.bat` is created as a shortcut to run the program via the Anaconda Command Prompt. This batch file activates the `llambit_env` environment and runs `main.py`, providing a user-friendly way to start the application without needing to execute Python commands manually.

```
# Contents of LLaMBIT.bat
@echo off
SET "AnacondaPath=%USERPROFILE%\anaconda3"
CALL "%AnacondaPath%\Scripts\activate.bat" %AnacondaPath%
CALL conda activate llambit_env
python main.py
```

11.2 Development Environment for Code Modifications

Spyder, or any other Python IDE, is recommended for developing bug fixes, feature additions, or other modifications to the LLaMBIT codebase. This approach allows contributors to leverage robust development tools while working on the application.

11.3 Contributing to LLaMBIT via GitHub

New users and contributors can access the LLaMBIT repository on GitHub to submit pull requests, report bugs, or request features. A clear workflow is outlined for contributing, including guidelines for pull requests, commit comments, and issue reporting.

- GitHub Repository: `https://github.com/ElectronPhenomenon/MDA_Practicum`
- Pull Request Process: Detailed instructions on how to fork the repository, create feature branches, and submit pull requests for review.
- Commit Comment Style: Guidelines for writing clear and informative commit messages.
- Feature Requests and Bug Reporting: Procedures for submitting feature requests and reporting bugs via GitHub Issues.

11.4 Deployment Strategies

LLaMBIT's deployment involves several key strategies to ensure smooth operation:

- **Initial Deployment**: Setting up the application on a suitable platform or server.
- **Continuous Deployment**: Automating the deployment process for new updates and features.

11.4.1 Deployment Example

```
# Example of deployment script or command
deploy_to_server(application_package)
```

This snippet illustrates a simplified version of a deployment command or script used to deploy LLaMBIT to a server or cloud platform.

11.5 Maintenance Practices

Maintaining LLaMBIT involves regular updates, bug fixes, and performance monitoring:

1. **Regular Updates**: Implementing new features and updates to keep the application current.
2. **Bug Fixes**: Addressing and resolving reported issues and errors.
3. **Performance Monitoring**: Monitoring the applications performance and making necessary optimizations.

11.5.1 Update and Bug Fix Implementation

```
def update_application():
    # Code for updating application
    logging.info("Updating␣LLaMBIT...")
```

This function might be used to update the application, including downloading and installing new features or bug fixes.

11.6 Error Handling in Deployment

Error handling during deployment is crucial to prevent downtime and other issues:

```
try:
    deploy_to_server(application_package)
except DeploymentError as e:
    logging.error(f"Deployment␣failed:␣{e}")
```

This pattern shows how deployment errors are caught and logged, ensuring that any issues during deployment are promptly addressed.

11.7 Maintenance Scheduling

Regular maintenance schedules are established to ensure LLaMBIT remains functional and efficient:

```
schedule_maintenance(downtime_window, maintenance_tasks)
```

This command sets a scheduled maintenance window and defines the tasks to be performed, such as updates or performance optimizations.

Chapter 12

Appendices

The appendices section of this manual serves as a resource for additional information, code examples, and detailed explanations that complement the main chapters. It provides users and developers with valuable insights and practical examples.

12.1 Appendix A: Code Snippets and Technical Details

12.1.1 Query Parsing and Execution

```
def _parse_query(self, query):
    """
    Parses the query into individual terms based on AND
       ↪ and OR.
    """
    and_terms = [term.strip() for term in query.split("AND
       ↪ ")]
    or_terms = []
    for term in and_terms:
        or_terms.extend([t.strip() for t in term.split("OR
           ↪ ")])
    or_terms = [re.sub(r'[\\"\\(\\)]', '', term) for term
        ↪ in or_terms]
    return or_terms

def query_articles(self, query, threshold=0.15):
    """
    Queries the articles based on the provided search
        ↪ query and returns
    the relevant articles.
    """
    logging.info(f"Querying␣articles␣with:␣{query}")
    or_terms = self._parse_query(query)
    combined_scores = np.zeros(self.tfidf_matrix.shape[0])

    for term in or_terms:
        term_cleaned = self._clean_text(term)
        term_vector = self.vectorizer.transform([
            ↪ term_cleaned])
        cosine_similarities = cosine_similarity(
            ↪ term_vector, self.tfidf_matrix).flatten()
        combined_scores = np.maximum(combined_scores,
            ↪ cosine_similarities)

    # Add the Relevance Score to the dataframe
    self.articles_df['Relevance␣Score'] = combined_scores

    # Filter articles based on the threshold
    relevant_articles = self.articles_df[self.articles_df[
        ↪ 'Relevance␣Score'] > threshold]
    # Sort the dataframe based on the Relevance Score
    ...
```

12.1.2 WoS Journal Scraper Initialization

```
def __init__(self, api_key, search_term, database_id='WOS'
  ↪ ):
    super().__init__(search_term, api_key)
    logging.info(f"Initializing␣{self.__class__.__name__}␣
      ↪ with␣search␣term:␣{search_term}")
    self.configuration = self.configure_api(api_key)
    self.search_api_instance = self.init_search_api()
    self.database_id = database_id
    self.search_term = search_term
    ...
```

12.1.3 Data Extraction and Preprocessing

```
def extract_data(self, record):
    # Extract and preprocess data like authors, abstract,
      ↪ etc.
    # Return a structured data object
    ...
```

12.1.4 Article Preprocessing

```
class ArticlePreprocessor:
    """
    A class to preprocess and query articles.
    """
    def __init__(self):
        """
        Initializes the ArticlePreprocessor with necessary
            ↪  tools.
        """
        self.vectorizer = TfidfVectorizer(stop_words='
            ↪ english')
        self.lemmatizer = WordNetLemmatizer()
        self.articles_df = None
        self.tfidf_matrix = None
        ...
```

12.2 Appendix B: Configuration File Structure

Details on the structure and format of the ‘config.json‘ file used in LLaMBIT:

```
{
    "search_term": "example␣term",
    "email": "user@example.com",
    "api_keys": {
        "entrez": "your_entrez_api_key",
        "wos": "your_wos_api_key"
    },
    "scrapers": {
        "pubmed": true,
        "pubmed_central": false,
        "wos": true
    },
    "date_range": {
        "mindate": "2021-01-01",
        "maxdate": "2021-12-31"
    }
}
```

12.3 Appendix C: Additional Resources

This appendix provides a list of additional resources for further reading and reference, which are invaluable for understanding and utilizing the technologies and methodologies employed in LLaMBIT.

- **Python Documentation**: Comprehensive documentation for Python. `https://docs.python.org/3/`
- **PyQt5 Documentation**: Official documentation for PyQt5. `https://www.riverbankcomputing.com/static/Docs/PyQt5/`
- **Python Logging Module**: Guide to Python's logging module. `https://docs.python.org/3/library/logging.html`
- **Pandas Documentation**: Documentation for Pandas. `https://pandas.pydata.org/pandas-docs/stable/`
- **Scikit-Learn Documentation**: Guide on Scikit-Learn. `https://scikit-learn.org/stable/documentation.html`
- **Beautiful Soup Documentation**: Official documentation for Beautiful Soup. `https://www.crummy.com/software/BeautifulSoup/bs4/doc/`
- **Anaconda and Conda Documentation**: Guides on using Anaconda and Conda. `https://docs.anaconda.com/`

- **Entrez Programming Utilities (E-utilities)**: Documentation for NCBI's Entrez API, used for PubMed and PubMed Central data retrieval. `https://www.ncbi.nlm.nih.gov/books/NBK25501/`

- **Biopython Documentation**: Essential for interacting with the Entrez API. `http://biopython.org/DIST/docs/tutorial/Tutorial.html`

- **WoS (Web of Science) API Documentation**: Detailed information for the WoS API, used for WoS data retrieval. `https://developer.clarivate.com/apis/wos`

- **Git and GitHub Guides**: Resources for Git and GitHub.

  – GitHub Docs: `https://docs.github.com/en`

  – Atlassian Git Tutorial: `https://www.atlassian.com/git/tutorials`

- **Web Scraping Tutorials**:

  – Real Python Tutorial: `https://realpython.com/`

  – DataCamp Tutorial: `https://www.datacamp.com/community/tutorials/web-scraping-using-python`

- **NLP and Text Processing Tutorials**:

  – NLTK Book: `https://www.nltk.org/book/`

  – Text Processing Tutorial: `https://kavita-ganesan.com/text-preprocessing-tutorial/#.Yl38Fi2z2Ul`

These resources provide a deep dive into the various aspects of software development relevant to LLaMBIT, from programming in Python and GUI development to data processing, web scraping, API integration, and version control.

Chapter 13

User Feedback and Community Contributions

User feedback and community contributions are vital for the continual improvement and evolution of LLaMBIT. This chapter outlines the mechanisms for gathering user feedback and the process for community members to contribute to the project.

13.1 User Feedback Mechanisms

LLaMBIT incorporates several features for users to provide feedback and report issues:

- **Interactive GUI Feedback**: The application's GUI includes elements where users can easily report bugs or suggest improvements.
- **Logging and Error Reporting**: LLaMBIT utilizes Python's logging module to capture user interactions and errors, which can be reviewed for improvements.

13.1.1 Logging and Error Reporting Implementation

```
# In main.py
logging.basicConfig(level=logging.WARNING)
logging.info("Starting␣LLaMBIT...")


# Error handling in application
try:
    # Application logic
except Exception as e:
    logging.error(f"Encountered␣an␣error:␣{e}")
```

13.2 Community Contributions

LLaMBIT encourages contributions from the community, including code enhancements, feature suggestions, and bug fixes.

- **GitHub Repository**: The primary platform for code management and contributions. `https://github.com/ElectronPhenomenon/MDA_Practicum`
- **Contribution Guidelines**: Detailed instructions for submitting pull requests, coding standards, and commit message formats.

13.2.1 Contributing to LLaMBIT

Contributors are encouraged to follow these steps:

```
# Fork the repository
# Clone the forked repository
git clone https://github.com/[your-username]/MDA_Practicum
   ↪ .git


# Create a new branch for your feature or bug fix
git checkout -b feature/your-new-feature


# Make changes and commit them
```

```
git commit -am "Add␣a␣new␣feature"

# Push to your fork
git push origin feature/your-new-feature

# Create a pull request from your fork to the main LLaMBIT
   ↪  repository
```

13.3 Acknowledgments

Special thanks to all contributors and users who have provided valuable feedback and suggestions, helping to shape LLaMBIT into a more robust and user-friendly tool.

- Andrew King, MSE. Automation and Controls Engineer, CelLink Corporation.
- Blake Harper, MPH. Program Director, Cancer Prevention and Control Platform, The University of Texas MD Anderson Cancer Center.
- Travis Holder, MSLS, MA. Librarian, Research Medical Library, The University of Texas MD Anderson Cancer Center.