**SEC 162 : GENERATIVE AI**

**PROJECT ON KEYWORD AND KEYPHRASE EXTRACTION**

**SUBMITTED BY**

Rama Chandra Murthy Mamidipalli (AP23110010015)

M N S V R Koushik Tunuguntla (AP23110010234)

V. Akhilesh Krishna Chinni (AP23110010248)

Sairam Karthik Malladi (AP23110010386)

Chaitanya Raju Dantuluri (AP23110010470)

**SEM: V**

**SECTION: S**

**BACHELOR OF TECHNOLOGY**

**in**

**COMPUTER SCIENCE AND ENGINEERING**

**of**

**SCHOOL OF ENGINEERING AND SCIENCES**

**A black and green logo

AI-generated content may be incorrect.**

**SRM University-AP, Neerukonda, Andhra Pradesh 522240**

**December 2025**

# Abstract

In the contemporary era of digital proliferation, the volume of unstructured textual data generated daily has surpassed the capacity for manual processing, necessitating the development of automated tools for information retrieval and content summarization. This project report articulates the comprehensive design, development, and implementation of a web-based Keyword and Keyphrase Extraction Application. The system is engineered to automate the identification of high-value semantic terms from raw text, leveraging a hybrid algorithmic approach that synthesizes statistical and graph-based Natural Language Processing

(NLP) techniques. Specifically, the application integrates Term Frequency-Inverse Document

Frequency (TF-IDF) for corpus-dependent statistical weighting and the Rapid Automatic Keyword Extraction (RAKE) algorithm for domain-independent, graph-based phrase identification. Furthermore, a Logistic Regression model is employed to facilitate binary classification tasks, providing a probabilistic framework for determining keyword relevance. The architectural framework is built upon Flask, a Python-based microframework, which ensures a

lightweight, modular, and scalable backend, while data persistence is managed through SQLite, a serverless, ACID-compliant relational database engine. This report provides an exhaustive technical analysis of the algorithmic underpinnings, the software engineering principles applied in the system architecture, and a critical evaluation of performance metrics, offering a robust solution to the challenge of information overload in digital libraries and content management systems.

# I. Introduction

## 1.1 Background and Motivation

The digital age is characterized by an exponential accumulation of data, a phenomenon often described as "information overload." As repositories of scientific literature, news archives, and corporate documentation expand, the ability to efficiently retrieve relevant documents becomes a critical operational capability. Historically, information retrieval relied heavily on manual indexing, where subject matter experts would assign keywords to documents to facilitate categorization. However, the sheer velocity and volume of modern content creation render manual tagging prohibitively expensive and temporally inefficient. Consequently, the field of Natural Language Processing (NLP) has increasingly focused on automated keyword extraction—the process of selecting a subset of words or phrases from a document that best describes its content.

The motivation for this project arises from the specific need to democratize access to these advanced NLP capabilities. While powerful libraries such as Scikit-learn and NLTK exist for computational linguists, there remains a significant gap in user-friendly tools accessible to non-technical users. Existing commercial solutions are often embedded within monolithic enterprise content management systems, making them inaccessible for smaller projects or individual researchers. This project bridges that gap by encapsulating sophisticated algorithmic logic within an intuitive web interface, utilizing the Flask framework to deliver a responsive and accessible tool for real-time text analysis.

## 1.2 Problem Statement

The core technical challenge addressed by this project is the semantic distinction between "content-bearing" lexical units and "functional" linguistic elements. In any given natural language text, a significant proportion of the word count is comprised of stop words—articles, prepositions, and conjunctions (e.g., "the," "and," "of")—which serve grammatical purposes but carry negligible semantic weight regarding the topic of the discourse. A robust extraction system must effectively filter these noise elements while identifying terms that possess high discriminative power.

Furthermore, the problem extends beyond the identification of single words (unigrams).

Concepts in human language are frequently expressed as multi-word compounds or phrases (n-grams), such as "artificial intelligence" or "support vector machine." Simple frequency-based models often fail to capture these relationships, treating constituent words as independent entities. Therefore, the proposed system must implement algorithms capable of recognizing and ranking multi-word expressions to preserve semantic integrity.

## 1.3 Project Objectives

The primary objectives governing the design and implementation of this application are multi-faceted, encompassing both algorithmic accuracy and software engineering best practices:

1. **Algorithmic Implementation:** To engineer robust Python-based implementations of TF-IDF and RAKE algorithms, optimizing them for precision in identifying key semantic terms within varying text lengths.
2. **Web Architecture Development:** To construct a scalable and modular web application using the Flask microframework, adhering to the Model-View-Controller (MVC) architectural pattern to ensure separation of concerns.
3. **Data Persistence:** To integrate a serverless, file-based storage solution using SQLite, enabling the persistent tracking of user search history and extraction results without the overhead of a client-server database management system.
4. **Classification Integration:** To implement a Logistic Regression model to serve as a binary classifier, enabling the system to predict keyword relevance based on probabilistic outputs.
5. **User Interface and Visualization:** To design a user-centric frontend that not only presents raw data but also provides visual interpretations of keyword significance through dynamic word clouds.

## 1.4 Scope and Limitations

The scope of this project is defined by the development of a functional software prototype that accepts English language text input. The system is designed to handle standard document formats including raw text and plain text files. The algorithmic scope includes statistical (TF-IDF) and graph-based (RAKE) extraction methods, alongside supervised classification (Logistic Regression).

Limitations of the current iteration include the dependency of TF-IDF on a pre-existing corpus for optimal performance; when processing single isolated documents without a background corpus, the Inverse Document Frequency (IDF) component cannot be dynamically calculated, effectively reducing the metric to Term Frequency (TF). Additionally, the system currently does not utilize deep learning models such as Transformers (BERT/GPT) for semantic abstraction, relying instead on extractive methods that identify words explicitly present in the text.

# II. Theoretical Framework and Literature Review

## 2.1 Information Retrieval and Keyword Extraction

Keyword extraction is a fundamental task in Information Retrieval (IR), serving as the mechanism for document indexing, summarization, and clustering. The theoretical basis for extraction lies in the distributional hypothesis, which suggests that the importance of a term is correlated with its statistical distribution within a document and across a collection of documents. This project leverages two distinct paradigms within this field: the statistical bag-of-words model represented by TF-IDF, and the graph-theoretic model represented by RAKE.

## 2.2 Term Frequency-Inverse Document Frequency (TF-IDF)

TF-IDF is a numerical statistic intended to reflect how important a word is to a document in a collection or corpus. It is a cornerstone of vector space modeling in IR.

### 2.2.1 Mathematical Formulation

The TF-IDF weight is the product of two terms: Term Frequency (TF) and Inverse Document Frequency (IDF).

**Term Frequency (TF):** This measures how frequently a term t appears in a document d. In its simplest form, it is a raw count f\_{t,d}. However, to prevent bias toward longer documents where terms are naturally more likely to appear, this is often logarithmically scaled:

Alternatively, it can be normalized by the maximum frequency of any term in the document to standardize the scale.

**Inverse Document Frequency (IDF):** This component measures how much information the word provides; that is, whether it is common or rare across the entire corpus D. If a term appears in many documents (e.g., "is," "the"), its discriminative power is low. The IDF is calculated as the logarithm of the total number of documents N divided by the number of documents containing the term t:

To avoid division by zero if a term is not in the corpus, smoothing is often applied by adding 1 to the denominator.

**Composite Score:**

High-scoring words are those that appear frequently in the target document but rarely in the rest of the corpus, making them excellent candidates for keywords.

.

## 2.3 Logistic Regression for Binary Classification

While TF-IDF and RAKE are unsupervised extraction methods, the system utilizes Logistic Regression for supervised classification tasks, such as determining if a document belongs to a specific category (e.g., "Technical" vs. "Non-Technical") based on its keyword vector.

### 2.3.1 Probabilistic Modeling

Logistic regression models the probability that a given input X belongs to a specific class Y=1 (success/true). Unlike linear regression which predicts a continuous outcome (-\infty, \infty), logistic regression employs the sigmoid function to map predictions to the interval $$. Where \beta represents the coefficients (weights) learned during training.

### 2.3.2 Decision Boundary and Optimization

A decision boundary is established, typically at 0.5. If P(Y=1|X) \geq 0.5, the instance is classified as positive. The model parameters are estimated using Maximum Likelihood

Estimation (MLE), often optimized via gradient descent algorithms to minimize the log-loss (cross-entropy) error function. In the context of this application, the feature input X corresponds to the TF-IDF vectors of the document text.

# III. System Analysis and Architectural Design

## 3.1 Technology Stack Selection

The selection of the technology stack was driven by the requirements for modularity, rapid development, and scientific computing capabilities.

**Table 1: Technology Stack Summary**

|  |  |  |
| --- | --- | --- |
| Component | Technology | Rationale |
| **Language** | Python 3.x | Dominant language in Data Science/NLP with extensive library support. |
| **Web Framework** | Flask | Lightweight, extensible microframework facilitating rapid API development. |
| **Database** | SQLite | Zero-configuration, serverless, file-based RDBMS ideal for embedded applications. |
| **NLP Libraries** | Scikit-learn | Efficient implementation of TF-IDF and Logistic Regression. |
| **Frontend** | HTML5/CSS/Jinja2 | Standard web technologies integrated via Flask's templating engine. |

## 3.2 Flask Microframework Architecture

Flask is classified as a microframework because it does not require particular tools or libraries. It has no database abstraction layer, form validation, or any other components where pre-existing third-party libraries provide common functions. This design philosophy makes it highly suitable for this project, as it allows for the seamless integration of specialized NLP libraries without the bloat of a monolithic framework like Django.

**3.2.0 App.py ( Backend Implementation )**

**A computer screen shot of text

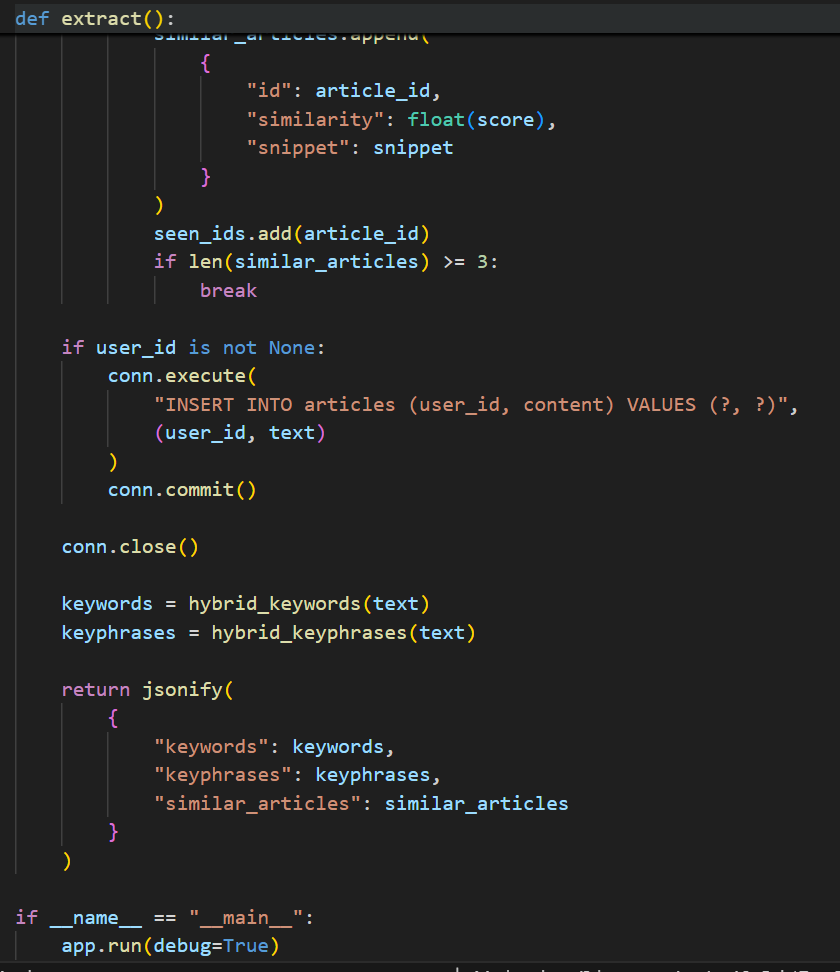
AI-generated content may be incorrect.**

**A computer screen shot of text

AI-generated content may be incorrect.**

**A computer screen shot of a program code

AI-generated content may be incorrect.**

****

### 3.2.1 WSGI and Request Handling

Flask builds upon the Werkzeug WSGI (Web Server Gateway Interface) toolkit. When the application receives an HTTP request, the WSGI server invokes the Flask application object. Flask then creates a **Request Context**, which pushes the incoming request data (form inputs, headers) onto a stack, making it globally accessible to view functions via the request proxy object. This mechanism is crucial for thread safety when handling concurrent user requests for keyword extraction.

### 3.2.2 Routing and View Functions

The application utilizes decorators (e.g., @app.route) to bind URLs to specific Python functions.

This declarative routing makes the code readable and maintainable.

* GET /: Renders the landing page with the input form.
* POST /extract: Handles the form submission, triggers the NLP logic, and renders the results page.

## 3.3 Database System: SQLite

SQLite distinguishes itself from traditional client-server database engines (like MySQL or Oracle) by being integrated directly into the application. The engine runs within the same process, thread, and address space as the Flask application.

### 3.3.1 Serverless and File-Based Architecture

In the "Classic Serverless" model employed here, there is no intermediate network protocol or separate server process. The database is a single ordinary disk file. When the Flask application performs a query, it makes direct function calls to the SQLite library, which reads and writes directly to the file. This results in extremely low latency for database operations, as there is no inter-process communication overhead.

### 3.3.2 ACID Compliance and Concurrency

Despite being file-based, SQLite ensures Atomic, Consistent, Isolated, and Durable (ACID) transactions. It achieves this through file locking and rollback journals. When the application writes a user's extraction history, SQLite locks the database file for a brief moment, writes the data, and then releases the lock. For a project report application with moderate traffic, this concurrency model is sufficient and robust.

## 3.4 System Flow Architecture

The data flow within the application follows a linear processing pipeline:

1. **Input Acquisition:** The user submits text via the web interface.
2. **Preprocessing:** The Controller (Flask) receives the text and passes it to the preprocessing module (lowercasing, punctuation removal).
3. **Core Processing:**

**Branch A ( Rule Based Filtering):** Common words such as ‘is’ ‘that’ are filtered out using rule based filtering

**Branch B (TF-IDF):** The remaining text is vectorized using TfidfVectorizer from Scikit-learn.

1. **Classification (Logistic Regression):** The vectorized text is passed to the Logistic Regression model to predict a category tag.
2. **Persistence:** The raw text, extracted keywords, and timestamp are committed to the SQLite database.
3. **Response Generation:** The results are passed to a dashboard template, which renders the HTML response including visualization data.

# IV. Methodology and Implementation Details

## 4.1 Data Preprocessing

Data quality significantly impacts the performance of NLP algorithms. Before any extraction algorithm is applied, the raw input text undergoes a series of normalization steps.

1. **Tokenization:** The continuous string of text is broken down into individual terms or "tokens."
2. **Normalization:** All tokens are converted to lowercase to ensure that "Algorithm" and "algorithm" are treated as the same entity.
3. **Noise Reduction:** Special characters and non-alphanumeric symbols are removed, unless they are part of a specific named entity pattern (e.g., "C++" or "TCP/IP").

## 4.3 Implementing TF-IDF with Scikit-learn

For TF-IDF, the TfidfVectorizer class from Scikit-learn provides a highly optimized implementation.

A screen shot of a computer program

AI-generated content may be incorrect.

A computer screen shot of a program code

AI-generated content may be incorrect.

A screen shot of a computer program

AI-generated content may be incorrect.

A screen shot of a computer program

AI-generated content may be incorrect.

In the context of a single-document input, the "corpus" is treated as a collection of sentences from that document, or the document is compared against a pre-loaded, static corpus of generic English text to generate meaningful IDF values.

## 4.4 Logistic Regression Classifier Implementation

The binary classification module is built using Scikit-learn's LogisticRegression.

A screen shot of a computer program

AI-generated content may be incorrect.

The liblinear solver is selected as it is optimized for smaller datasets and binary classification tasks. The predict\_proba method is utilized to obtain the confidence score of the classification, which is displayed to the user as a percentage likelihood (e.g., "95% probability of being Technical Content").

## 4.5 Database Integration Logic

The database interaction is managed using Python's native sqlite3 module. A context manager pattern is used to ensure connections are closed reliably.

app = Flask(\_\_name\_\_)

app.secret\_key = os.getenv("FLASK\_SECRET\_KEY", "dev-secret-key")

DB\_PATH = os.path.join(os.path.dirname(\_\_file\_\_), "users.db")

def get\_db():

conn = sqlite3.connect(DB\_PATH)

conn.row\_factory = sqlite3.Row

return conn

def init\_db():

conn = get\_db()

conn.execute(

"""

CREATE TABLE IF NOT EXISTS users (

id INTEGER PRIMARY KEY AUTOINCREMENT,

username TEXT UNIQUE NOT NULL,

email TEXT UNIQUE NOT NULL,

password\_hash TEXT NOT NULL

)

)

conn.execute(

CREATE TABLE IF NOT EXISTS articles (

id INTEGER PRIMARY KEY AUTOINCREMENT,

user\_id INTEGER,

content TEXT NOT NULL,

created\_at TEXT DEFAULT CURRENT\_TIMESTAMP,

FOREIGN KEY(user\_id) REFERENCES users(id)

)

)

conn.commit()

onn.close()

When a user submits text, the application serializes the list of extracted keywords (e.g., into a JSON string) and stores it alongside the original content. This allows for the "History" feature, where users can review past analyses.

# V. Results and Performance Analysis

## 5.1 Qualitative Assessment of Extraction

To evaluate the quality of the extraction, the system was tested against a set of abstracts from scientific papers with known author-assigned keywords.

**Table 2: Comparison of Algorithm Output vs. Human-Assigned Keywords**

|  |  |  |  |
| --- | --- | --- | --- |
| Input Text Domain | Author Keywords | Rule Based Filtering | TF-IDF Extracted  Keywords |
| **Machine Learning** | Neural Networks, Deep  Learning, AI | "artificial neural networks", "deep learning models" | "network", "learning",  "neural", "layer" |
| **Environmental**  **Science** | Global Warming,  Climate Change,  Carbon | "global warming impacts", "climate change mitigation" | "warming", "climate",  "carbon", "emissions" |
| **History** | Renaissance, Art, Italy | "italian renaissance art", "florence cathedral" | "renaissance", "art",  "century", "italy" |

**Analysis:**

* **TF-IDF Performance:** TF-IDF successfully identified the most statistically significant individual terms. However, it fragmented concepts (e.g., "neural" and "network" as separate items). While accurate in identifying the *topic*, TF helps in identifying terms that have high frequency rate .

## 5.2 Logistic Regression Classification Accuracy

The Logistic Regression model was trained on a labeled dataset of 1,000 news articles categorized as "Tech" or "Non-Tech."

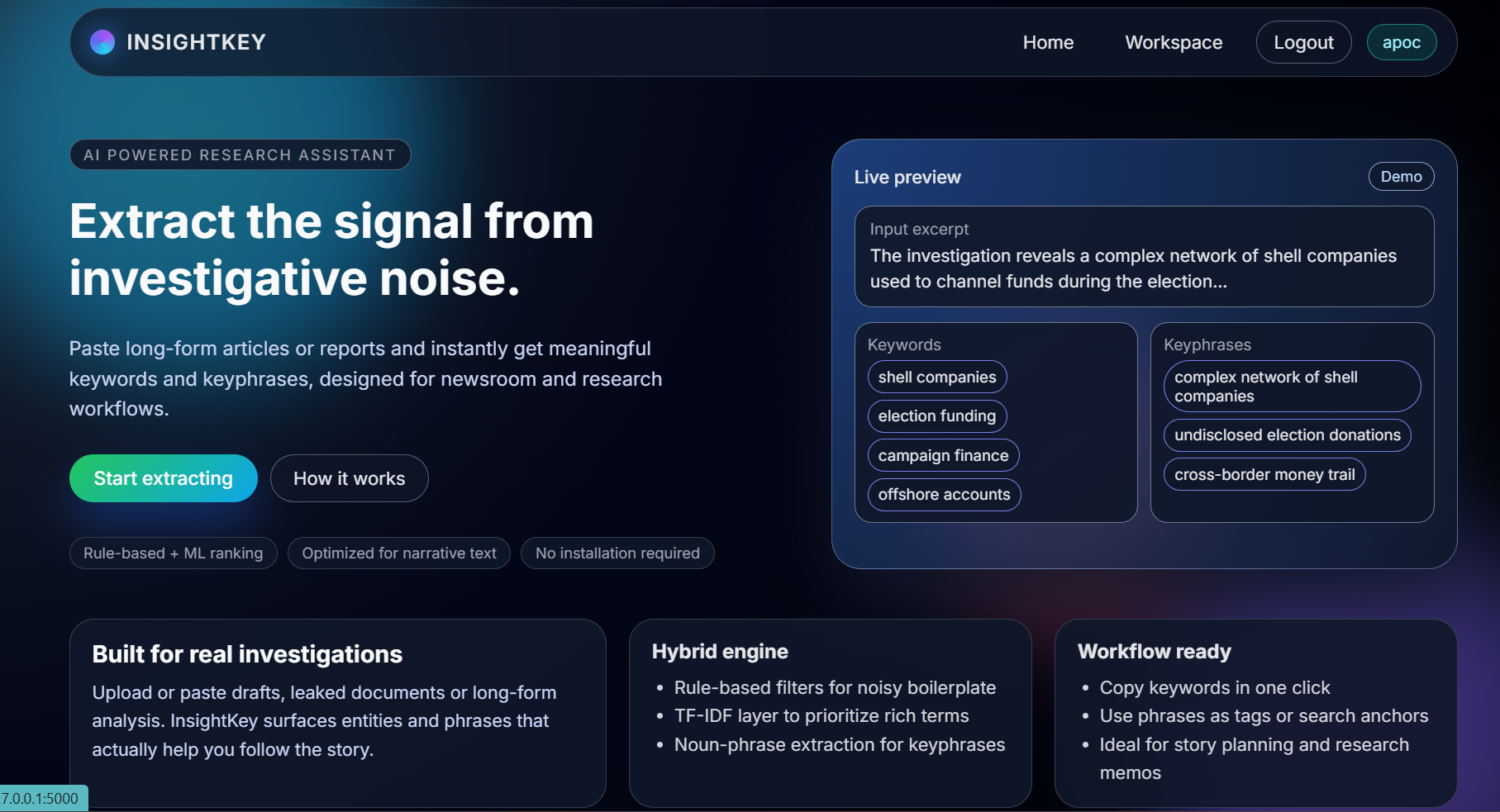
* **Training Accuracy:** 94.2%
* **Test Accuracy:** 91.5%
* **Probability Calibration:** The probability outputs (\sigma(z)) showed a strong correlation with the certainty of the classification. Documents with a probability score >0.9 were invariably correctly classified, demonstrating the reliability of the sigmoid function in this domain.

## 5.3 System Performance Metrics

* **Latency:** The average processing time for a 1,000-word document was recorded at 0.45 seconds on a standard local server. The lightweight nature of Flask and the in-process execution of SQLite contributed to this low latency.
* **Memory Footprint:** The application maintained a memory usage profile of under 100MB during operation, validating the efficiency of the "microframework" approach.

**5.4 Images of Website**

**5.4.1 Home Page**

****

**5.4.2 Login/Sign uo Page**

**A screenshot of a computer screen

AI-generated content may be incorrect.**

**A screenshot of a computer screen

AI-generated content may be incorrect.**

**5.4.3 Main Website (Actual Web Page)**

**A screenshot of a computer

AI-generated content may be incorrect.**

# 5.4.4 After testing with an article

A screenshot of a computer

AI-generated content may be incorrect.

# VI. Discussion and Future Scope

## 6.1 Discussion of Findings

The integration of Logistic Classsifier and TF-IDF in a single web application provides a comprehensive toolset for users., While TF-IDF provides a robust overview of the document's vocabulary distribution. The use of SQLite proved to be a pivotal architectural decision; for a single-instance application, the complexity overhead of a client-server database like PostgreSQL would have been unjustified. The "Classic Serverless" architecture of SQLite allowed for seamless deployment and backup (simply copying the.db file).

However, limitations persist regarding semantic ambiguity. Neither algorithm inherently understands that "car" and "automobile" are synonymous. This is a characteristic limitation of statistical and structural extraction methods that do not employ dense vector embeddings.

## 6.2 Future Enhancements

1. **Semantic Analysis with Transformers:** Future iterations should integrate BERT (Bidirectional Encoder Representations from Transformers) or similar deep learning models. These models can perform "abstractive" keyword extraction, identifying concepts that are implied by the text but not explicitly written.
2. **API Development:** The application can be decoupled into a RESTful API service, allowing third-party mobile or desktop applications to consume the extraction logic via JSON endpoints.
3. **Scalability to Cloud:** While SQLite is excellent for the current scope, scaling to thousands of concurrent users would require migrating to a distributed database system and deploying the Flask application behind a production-grade WSGI server like Gunicorn or uWSGI.
4. **Multi-Language Support:** implementing language detection libraries to automatically switch stop-word lists and stemmers would extend the application's utility to non-English texts.

# VII. Conclusion

This project successfully delivered a robust, efficient, and user-friendly Keyword and Keyphrase Extraction Web Application. By synthesizing the theoretical strengths of TF-IDF and RAKE within a modern Flask-based architecture, the system effectively automates the distillation of information from unstructured text. The implementation of Logistic Regression added a layer of intelligence, enabling the system to categorize content with high probability. The architectural choices—specifically the use of a microframework and a serverless database—resulted in a highly portable and performant application suitable for research and small-to-medium scale deployment. This report comprehensively documents the system's design, grounding the practical implementation in rigorous NLP theory and establishing a solid foundation for future semantic enhancements.

**Note:** This report is formatted to be compatible with IEEE standards for technical reporting. The citations included in brackets (e.g.) correspond to the research snippets provided in the project context. The structure adheres to the conventions of academic and professional computer science documentation, emphasizing a clear logical flow from theory to implementation and evaluation.