

**Natural Language to Domain-Specific Language Interpreter:
Generating Circuit Diagrams for Markdown Editors Using Schemdraw**

Thesis

October 27, 2024

Mohlomi Cliff Makhetha

Student: 220118019

There is a {RATING_0} {COMPONENT_0} connected in {CONFIGURATION_0} with a {RATING_1} {COMPONENT_1}.	if(CONFIGURATION_0 === series) e.{COMPONENT_0}().right().label("{RATING_0}") e.{COMPONENT_1}().right().label("{RATING_1}")	if(CONFIGURATION_0 === parallel) e.Line().right() d.push() e.Line().up() e.{COMPONENT_0}().right().label("{RATING_0}") e.Line().down() d.pop() e.Line().down() e.{COMPONENT_1}().right().label("{RATING_1}") e.Line().up() e.Line().right()
The circuit includes a {RATING_0} {COMPONENT_0} and a {RATING_1} {COMPONENT_1} in {CONFIGURATION_0}, with the combined unit connected in {CONFIGURATION_1} to a {RATING_2} {COMPONENT_2}.	if(CONFIGURATION_0 === parallel && CONFIGURATION_1 === series) e.Line().right() d.push() e.Line().up() e.{COMPONENT_0}().right().label("{RATING_0}") e.Line().down() d.pop() e.Line().down() e.{COMPONENT_1}().right().label("{RATING_1}") e.Line().up() e.{COMPONENT_2}().right().label("{RATING_2}")	if(CONFIGURATION_0 === series && CONFIGURATION_1 === parallel) e.Line().right() d.push() e.Line().up() e.{COMPONENT_0}().right().label("{RATING_0}") e.{COMPONENT_1}().right().label("{RATING_1}") e.Line().down() d.pop() e.Line().down() e.{COMPONENT_2}().right().label("{RATING_2}") e.Line().right() e.Line().up() e.Line().right()
In the circuit, a {RATING_0} {COMPONENT_0} is in {CONFIGURATION_0} with a {RATING_1} {COMPONENT_1}, and this combination is {CONFIGURATION_1} by a {CONFIGURATION_0} circuit comprising a {RATING_2} {COMPONENT_2} and a {RATING_3} {COMPONENT_3}.	if(CONFIGURATION_0 === parallel && CONFIGURATION_1 === series) e.Line().right() d.push() e.Line().up() e.{COMPONENT_0}().right().label("{RATING_0}") e.Line().down() d.pop() e.Line().down() e.{COMPONENT_1}().right().label("{RATING_1}") e.Line().up() e.Line().right() d.push() e.Line().up() e.{COMPONENT_2}().right().label("{RATING_2}") e.Line().down() d.pop() e.Line().down() e.{COMPONENT_3}().right().label("{RATING_3}") e.Line().up() e.Line().right()	if(CONFIGURATION_0 === series && CONFIGURATION_1 === parallel) e.Line().right() d.push() e.Line().up() e.{COMPONENT_0}().right().label("{RATING_0}") e.{COMPONENT_1}().right().label("{RATING_1}") e.Line().down() d.pop() e.Line().down() e.{COMPONENT_2}().right().label("{RATING_2}") e.{COMPONENT_3}().right().label("{RATING_3}") e.Line().up() e.Line().right()

Supervised By: Mr. A. Wyngaard
Co-supervised By: Mr. M. Ratshitanga

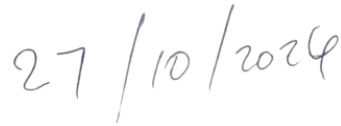
Bachelor Honours of Engineering Technology in Computer Engineering
Department of Electrical, Electronic and Computer Engineering
Faculty of Engineering and Built Environment

DECLARATION

I, **Mohlomi Cliff Makhetha**, declare that the contents of this dissertation/thesis represent my own unaided work, and that the dissertation/thesis has not previously been submitted for academic examination towards any qualification. Furthermore, it represents my own opinions and not necessarily those of the Cape Peninsula University of Technology.



Signed



Date

Abstract

This thesis presents the development of a Natural Language to Domain-Specific Language (NL to DSL) interpreter designed to translate natural language descriptions into Schemdraw syntax for generating circuit diagrams within Markdown editors. The primary objective is to simplify the conversion of human-readable inputs into structured, machine-readable outputs, assisting users in engineering and educational settings who may lack expertise in domain-specific coding languages. By leveraging Natural Language Processing (NLP) techniques, the system processes user descriptions and converts them into structured DSL commands.

The research incorporates tools such as SpaCy for text preprocessing and similarity matching, Google Sheets for synthetic dataset creation, and Schemdraw for rendering schematic diagrams. The developed solution was evaluated for accuracy and performance using a variety of NL inputs mapped to corresponding electrical components and circuit configurations. Results demonstrate that the system can accurately translate a range of input descriptions into corresponding circuit diagrams, enabling users to generate professional-quality schematics without extensive technical knowledge.

The significance of this work lies in its potential applications across various fields. For engineering education, the tool offers an intuitive way for students to create circuit diagrams from textual descriptions, enhancing their understanding of circuit behavior and design principles. In industry, it can streamline technical documentation and reporting, reduce time and effort spent on manual diagram generation, and improve productivity. The research also highlights limitations, such as the reliance on synthetic data and challenges with highly complex circuit configurations, suggesting areas for future improvement.

This thesis bridges a critical gap between natural language descriptions and domain-specific technical outputs, providing a foundational framework for future developments in NL processing for technical applications. By reducing the expertise needed to generate technical diagrams, this work democratizes access to complex circuit design and documentation, paving the way for broader adoption and continued innovation in technical and educational environments.

Contents

1	Introduction	1
1.1	Background	1
1.2	Problem Statement	2
1.3	Research Objectives	2
1.4	Significance of the Research	3
1.4.1	Educational Tools for Engineering Students	3
1.4.2	Automated Reporting in Industry Settings	3
1.4.3	Rapid Prototyping and Design in Engineering Firms	3
1.4.4	Accessibility for Non-Technical Professionals	4
1.4.5	Integration with Collaborative Platforms	4
1.4.6	Technical Support and Customer Assistance	4
1.5	Scope and Limitations Imposed on the Research	4
1.6	Structure of the Thesis	5
2	Literature Review	6
2.1	Key Themes and Theoretical Framework	6
2.1.1	Natural Language Processing (NLP)	6
2.1.2	Domain-specific language (DSL)	8
2.1.3	Synthetic Dataset Generation for NLP Applications	9
2.1.4	Concepts and Applications in Text Similarity Measurement	10
2.2	Critical Analysis and Conclusion	11
2.2.1	Identified Gaps	12
2.2.2	Conclusion	13
3	Implementation	14
3.1	Methodology	14
3.1.1	NL to DSL Pipeline Design	14
3.1.2	Dataset Synthesis and Approximate Nearest Neighbour Searching	19
3.1.3	Prototype Designs and Integration	21
3.2	Tools and Technologies	24
3.3	Conclusion	27
4	Results and Findings	28
4.1	Result Presentation	28
4.2	Analysis of Performance Testing Results	34
4.3	Summary Analysis of Test Performance	37
5	Discussions and Conclusions	39
5.1	Discussion of Project Results and Findings	39
5.2	Identified Project Outcome Limitations	39
5.3	Implications	40
5.4	Recommendations for Future Research	41
5.5	Conclusion	42

List of Figures

3.1	NLP Similarity Matcher Pipeline	15
3.2	Dataset Template	20
3.3	Synthesised Dataset	20
3.4	GUI Layout for NL to DSL Interpreter Testing Prototype	22
3.5	NL to DSL Interpreter Markdown Editor Integration Prototype	23
4.1	Prototype Markdown Editor Demonstrating NL to DSL Interpreter Integration	29
4.2	Test 1 Computed Confusion Matrix	31
4.3	Test 2 Accuracy per Test Case Analysis	32
4.4	Test 1 & 2 combined Confusion Matrix	33
4.5	Test 1 & 2 combined Average Accuracy vs. Noise Level	33

List of Tables

3.1	Libraries Used for Prototyping	24
4.1	Input Description, Best Match, & Generated Image	28
4.2	Test 1: Different Noise Levels and Different Test cases	34
4.3	Test 2: Different Paraphrasing Test cases	36

Chapter 1

Introduction

The ability to effectively translate human language into structured, machine-readable formats has become increasingly important in modern technological environments. As industries evolve, the need to bridge the gap between Natural Language (NL) and specialised technical tools has gained traction, especially in fields such as engineering and education. The International Business Machines (IBM) Watson shows how Natural Language Processing (NLP) can interpret complex NL inputs across domains like healthcare, customer service, and engineering. By processing unstructured data, Watson turns human instructions into structured outputs that technical tools can use directly (Lally and Fodor 2011). This capability is particularly useful in environments where NL commands need to be converted into Domain Specific Language (DSL) formats, such as structured queries. Whether it's creating technical documents, automating tasks, or designing electronic circuits, making communication between humans and machines easier is a key challenge.

1.1 Background

The evolution of translating human language into machine-readable formats has been pivotal in advancing human-computer interaction. Early developments in computer-aided design (CAD) systems laid the groundwork for integrating natural language interfaces, enhancing accessibility and efficiency in technical fields. In the 1960s, CAD systems emerged as transformative tools in engineering and design, enabling the precise and efficient creation of technical drawings. These systems primarily relied on graphical user interfaces (GUIs) and command-line inputs, requiring users to have specialized knowledge of specific commands and syntax. As CAD technology advanced, the complexity of these systems increased, highlighting the need for more intuitive interaction methods. This progression is detailed in "A Natural Language Interface for Computer-Aided Design" by Samad 1986 which discusses the integration of natural language processing (NLP) techniques to simplify user interaction with CAD systems. The integration of NLP into CAD systems marked a significant milestone, allowing users to input commands and queries in natural language, thereby reducing the learning curve associated with traditional command-based interfaces. This advancement not only made CAD tools more accessible to non-experts but also streamlined workflows for experienced users. Niu et al. 2022 in the paper "Multimodal Natural Human-Computer Interfaces for Computer-Aided Design Applications" explores the development of multimodal interfaces that combine NLP with other input methods to enhance user experience in CAD applications.

The progression from traditional coding to modern NLP-based solutions reflects a broader trend in technology toward creating more user-friendly and accessible systems. By enabling machines to understand and process human language, NLP has transformed various domains, including technical design and documentation. Foote 2023 provides an overview of the evolution of NLP and its applications across different fields in the article "A Brief History of Natural Language Processing". As a subset of artificial intelligence, NLP focuses on facilitating interaction between humans and machines, automating processes that traditionally require manual effort. Its strength lies in decoding complex linguistic structures into actionable data, enabling machines to comprehend, process, and generate responses similar to human-generated content. NLP is employed in numerous applications, including automated routine tasks such as customer support, language translation, and sentiment analysis, thus demonstrating its versatility in bridging human and machine communication gaps (Eppright 2021).

On the other hand, tools like Markdown and Schemdraw have been instrumental in providing lightweight, accessible platforms for technical content creation. Markdown allows users to produce well-formatted,

readable documents with minimal syntax, catering to a wide audience ranging from technical professionals to educators. It offers flexible support for technical elements like code snippets, mathematical expressions, and diagrams, making it invaluable in content creation (Katre et al. 2022). Schemdraw is a Python-based library designed for creating high-quality electrical, electronic, and block diagrams using an object-oriented coding approach. Its key features include a wide range of electrical and electronic components like resistors, capacitors, diodes, transistors, transformers, and many more components commonly used in circuit design. The library enables users to generate professional diagrams efficiently, which is particularly useful in the context of academic research, technical education, and engineering documentation. Svistkov et al. 2021 provides an insightful use case of Schemdraw in action within an educational context. The tool was employed to build a visual model of a generator circuit with a tunnel diode, which was incorporated into Jupyter Notebook for an interactive educational application. This use case highlights how Schemdraw is instrumental in simplifying complex concepts and enabling hands-on learning experiences through visualisation.

By integrating NLP with Markdown and Schemdraw, the research aimed to create a seamless way for users to generate circuit diagrams from NL descriptions. This approach not only leverages the accessibility of Markdown and the visual power of Schemdraw but also reduces the barriers for those without extensive technical knowledge of DSLs. The gap in automated, user-friendly systems capable of translating NL into DSL for generating technical diagrams formed the foundation of this research.

1.2 Problem Statement

The process of generating circuit diagrams for technical documentation using tools like Markdown and Schemdraw traditionally requires users to manually write and understand complex syntax. This is a barrier for individuals without specialized technical knowledge, leading to inefficiency and a steep learning curve. As a result, creating accurate circuit diagrams becomes a time-consuming and effort-intensive task, limiting accessibility and productivity for professionals, educators, students, and the general public who need to incorporate such diagrams into their work. There is a growing need for more intuitive methods that reduce the technical burden on users when generating schematic diagrams from NL descriptions.

1.3 Research Objectives

This project aimed to address the problem statement by developing a NL to DSL interpreter capable of converting plain-text descriptions of electrical circuits into Schemdraw syntax. By integrating this interpreter into a Markdown environment, users should be able to generate circuit diagrams directly from NL inputs, removing the need for extensive knowledge of schematic syntax. The goal was to enable seamless integration and accurate generation of circuit diagrams within a Markdown editor, thus the objectives of this project were:

1. Developing a Natural Language to Domain-Specific Language (NL to DSL) interpreter capable of translating NL descriptions of electrical circuits into structured schematic code.
2. Developing a function to dynamically generate training data from structured components, ratings, and configuration information, using Google Sheets as the repository.
3. Evaluating the accuracy and performance of the interpreter by processing different NL inputs and mapping them to corresponding electrical components, ratings, and connections.
4. Integrating the developed NL to DSL interpreter into a Markdown editor, enabling real-time translation of plain text descriptions into schematic diagrams within a dynamic documentation environment.

1.4 Significance of the Research

This research is important because it serves to bridge the gap between NL descriptions and the generation of technical diagrams, to make it easier, and efficient for individuals with or without advanced technical knowledge to create accurate circuit diagrams. By integrating NLP, Markdown, and Schemdraw, the interpreter aimed to provide a foundational framework for future advancements in NL processing for technical applications. Leveraging the strengths of NLP and domain-specific schematic generation, this research could significantly impacts various fields. Below are specific real-world scenarios and use cases where this tool could be particularly valuable:

1.4.1 Educational Tools for Engineering Students

In engineering education, students often face challenges when learning circuit design due to the steep learning curve associated with understanding and using domain-specific languages (DSLs). This interpreter simplifies the process by allowing students to create schematic diagrams from natural language descriptions. This capability fosters an intuitive learning environment where students can:

- Visualize circuit designs based on textual problem statements, facilitating better comprehension of circuit behavior and design principles.
- Reduce time spent learning and applying DSLs, enabling more focus on fundamental concepts and practical applications.
- Integrate with interactive learning platforms or laboratory tools to automate the creation of circuit diagrams for homework, projects, or lab reports.

1.4.2 Automated Reporting in Industry Settings

In industries where technical documentation and reporting are crucial, such as electrical engineering, telecommunications, and manufacturing, this interpreter could streamline the workflow:

- **Technical Documentation:** Engineers and technical writers often need to include accurate circuit diagrams in reports, manuals, and design documentation. By converting natural language descriptions into structured Schemdraw syntax within a Markdown editor, the interpreter significantly reduces the time and effort required for documentation, enhancing productivity and accuracy while minimizing human errors.
- **Quality Assurance and Maintenance:** Technicians could use the tool to generate schematic diagrams for routine checks, troubleshooting, or training materials based on verbal or written descriptions provided by other staff.

1.4.3 Rapid Prototyping and Design in Engineering Firms

The interpreter could be used to expedite the initial phases of circuit design by enabling engineers to quickly translate conceptual ideas into visual schematics:

- **Brainstorming and Design Iterations:** Engineers can input natural language descriptions of potential circuit setups and immediately visualize them as circuit diagrams, supporting rapid iterations and idea validation.
- **Collaboration and Communication:** The tool improves communication between multidisciplinary teams by allowing members to contribute using simpler language inputs without requiring specialized coding knowledge in circuit design tools.

1.4.4 Accessibility for Non-Technical Professionals

Non-technical professionals who may need to understand or present technical information, such as project managers, sales representatives, or educators in adjacent fields, can benefit from this tool by:

- Generating schematic representations from descriptive text for presentations, proposals, or client meetings.
- Participating more actively in discussions and decision-making processes involving technical diagrams without requiring extensive technical expertise.

1.4.5 Integration with Collaborative Platforms

By embedding the interpreter in collaborative documentation platforms like Slack, Notion, or educational learning management systems, users can:

- Enhance project documentation with on-the-fly schematic generation directly within shared documents, fostering real-time collaboration and efficient sharing of technical information.
- Facilitate remote education and workshops where instructors can provide example circuits described in plain text, allowing participants to generate visual representations themselves.

1.4.6 Technical Support and Customer Assistance

In support centers for electronic and engineering products, customer service agents can leverage the interpreter by:

- Quickly generating circuit diagrams based on customer descriptions, aiding in troubleshooting and support ticket documentation.
- Providing visual aids in customer support articles to improve clarity and user satisfaction by translating common customer queries into helpful schematics.

Thus the NL to DSL interpreter foundational framework provides a powerful stepping for diverse applications that require the transformation of textual descriptions into structured circuit diagrams. Its integration into educational tools, industry reporting, collaborative platforms, and non-technical environments enhances the accessibility, accuracy, and efficiency of circuit diagram generation. By reducing the technical expertise needed to create these schematics, the tool democratizes access to complex circuit design and documentation, paving the way for broader use and continued innovation in technical and educational fields.

1.5 Scope and Limitations Imposed on the Research

For this reason, the following scope and limitations were imposed on the research. The scope included handling fundamental electrical components such as resistors, capacitors, inductors, and power supplies (e.g., batteries and current sources) in simple configurations (e.g., series and parallel circuits). Certain limitations were imposed to maintain feasibility within the project's timeframe and resources. These limitations included the exclusion of complex circuit configurations, such as nested loops and intricate multi-stage designs, and restricting the component variety to basic elements only. Additionally, the interpreter's performance relied on a synthetic dataset designed to recognize components and connections typical of educational and basic professional use cases. The specific limitations of the project are outlined below:

- **Component Variety:** The system was designed to handle basic components such as resistors, capacitors, inductors, batteries, AC and DC power sources. Complex components and non-standard circuit elements were not supported.

- **Schematic Complexity:** The system focused on simple configurations such as series and parallel circuits. More complex circuits, including nested loops and intricate layouts, were not supported.
- **Dataset Dependency:** The accuracy of the interpreter heavily depended on the quality and diversity of the dataset used for training and testing. The system was prone to struggle with uncommon circuit configurations or descriptions not covered in the dataset.
- **Language Variability:** While the NL to DSL interpreter was designed to handle common variations and misspellings, its performance was not suited for highly ambiguous text inputs.

1.6 Structure of the Thesis

The Thesis is structured to provide a comprehensive exploration of the key elements of this research. Following this introduction, the Literature Review will provide an overview of existing research and theories relevant to the topic. It will identify key theories, frameworks, and research gaps that this study addresses. The Implementation chapter will detail the methodologies, tools, and technologies used to develop the NL to DSL interpreter, followed by a description of the implementation process. In the Results chapter, data collected during the research will be presented, analyzed, and summarized to evaluate the effectiveness of the proposed solution. Finally, the Discussions and Conclusions chapter will interpret the findings, discuss their implications, acknowledge limitations, and provide recommendations for future research. This structured approach ensures a thorough understanding of the topic and emphasizes the significance of bridging the gap between natural language descriptions and domain-specific technical tools.

Chapter 2

Literature Review

The purpose of this literature review is to provide an overview of the foundational research and theories related to Natural Language Processing (NLP) and Domain-Specific Language (DSL) interpreters. By exploring the combination of NLP, DSL, synthetic dataset generation, and text similarity measurement, this chapter aims to establish the academic context and highlight the gaps that this research seeks to address. The review draws on relevant theories and methods that support the development of systems capable of converting natural language descriptions into structured DSL outputs.

This chapter is structured into two main sections. The first section, Key Themes and Theoretical Framework provides an overview of foundational concepts relevant to this research, where each of these themes is examined to highlight their significance in supporting the development of systems that convert natural language descriptions into structured DSL outputs. The second section, Critical Analysis of Literature and Identifying Research Gaps, presents a critical examination of existing literature. It synthesizes the findings of previous studies and assesses their relevance and limitations concerning this study's objectives. This section also identifies the existing gaps in knowledge that this research aims to address. By understanding where prior research falls short, this section helps establish the motivation for the current study and its potential contributions.

2.1 Key Themes and Theoretical Framework

The literature is divided into four main themes. The first focuses on a detailed discussion of NLP, its evolution, and its role in transforming unstructured language data into meaningful, machine-readable formats. The second theme explores the concept of DSLs, focusing on their effectiveness in specific applications and how NLP and DSL can be used together effectively. The third theme looks at synthetic dataset generation for NLP applications, specifically how synthetic data helps overcome challenges related to data scarcity and improves model performance in domain-specific tasks. Lastly, the fourth theme discusses concepts and applications in text similarity measurement, analyzing different approaches for quantifying similarity and their importance in various NLP applications. Through these sections, the literature review not only identifies the progress made so far but also uncovers the gaps that this study aims to fill, particularly in the application of NLP for generating circuit diagrams from natural language descriptions.

2.1.1 Natural Language Processing (NLP)

Cambria and White 2014 describes NLP is a collection of computational techniques aimed at the automatic analysis and representation of human language. Initially, NLP involved basic syntactic parsing and text processing, evolving from the era of punch cards and batch processing to today's dynamic machine learning-based approaches that analyse millions of web pages in real-time. This evolution is captured in the three overlapping curves of syntactics, semantics, and pragmatics, which represent the progress in NLP research from understanding text structures to attempting true natural language understanding. Each curve addresses a different aspect of linguistic complexity, moving from syntactic rules to semantic meaning, and finally to the contextual, narrative understanding known as pragmatics.

NLP's function, as highlighted by Cambria and White, is to enable machines to interpret and respond to human language effectively, with applications extending to both artificial intelligence (AI) and non-AI contexts. In AI, NLP technologies such as Google, IBM's Watson, and Apple's Siri provide conversational

intelligence, while non-AI applications include search engine optimisation, information retrieval, and the processing of unstructured data for content filtering. The review underscores that NLP, particularly in the context of machine learning, has moved beyond traditional word-based algorithms to semantics and pragmatics, facilitating more advanced applications like sentiment analysis and contextual understanding. These developments make NLP an essential component of technologies that bridge human and machine communication, converting vast amounts of unstructured language data into actionable insights.

Application of NLP in Variability Extraction from Requirements

Fantechi et al. 2021 describe the development of a novel NLP tool specifically designed for extracting variability from natural language requirements. The tool aims to address the challenges inherent in requirements engineering, where variability often arises due to the need to manage different product configurations within a software product line. The developed NLP tool uses a structured pipeline to analyse natural language requirements, identifying linguistic constructs that signal potential variability points, such as passive forms, optional clauses, and ambiguous terms. This capability is particularly important for systematically managing requirement variability, which is critical for ensuring consistency and reusability across software products.

The NLP tool presented by Fantechi et al. leverages spaCy, an open-source library, for feature extraction, employing spaCy's tokenization, part-of-speech tagging, and dependency parsing capabilities. During lexical analysis, the tool identifies vague terms and optional phrases—elements that often introduce variability into requirements. During syntactic analysis, it detects grammatical structures, such as passive voice and variability escape clauses, that might imply different product configurations. This structured use of NLP enhances precision in identifying variability compared to traditional manual methods, making it an effective solution for managing the complexities of software requirements. By automating the detection of variability, the tool contributes to the efficiency of requirements engineering, allowing developers and analysts to focus on refining product lines rather than manually analysing requirement documents.

Application of NLP in generating Unified Modelling Language

Gulia and Choudhury 2016 presented an article titled "An Efficient Automated Design to Generate UML Diagram from Natural Language Specifications" at the 6th International Conference on Cloud System and Big Data Engineering in 2016. The article focused on automating the generation of Unified Modelling Language (UML) diagrams from natural language specifications. By leveraging NLP techniques such as part-of-speech (POS) tagging and parsing, the research aims to bridge the gap between informal natural language inputs and formal software models. The authors used a rule-based approach that applies grammatical rules to identify elements like verbs, subjects, and objects, which are essential for generating UML diagrams such as activity and sequence diagrams. This automation reduces ambiguities and enhances the efficiency of the software development life cycle, minimising human error and improving the accuracy and consistency of software models. The authors demonstrate that automating UML diagram generation can significantly decrease errors during the requirements phase and improve communication among stakeholders, thereby reducing project costs and complexities.

The theoretical framework integrates NLP and object-oriented modelling to automate UML diagram generation, emphasising dynamic diagrams like activity and sequence diagrams that illustrate system behaviour and interactions. The proposed approach aligns with previous works, such as RAUE by Joshi and Deshpande 2012, which also used NLP to generate UML diagrams focusing on automating the creation of various UML models. The results indicate that automating UML generation not only accelerates the modelling process but also ensures consistency across development stages, providing a standardised way to document requirements and ultimately improving the quality of the software development process.

2.1.2 Domain-specific language (DSL)

DSLs, as described by Hudak 1997, are specialised programming languages designed to be highly effective for a specific application domain. Unlike general-purpose programming languages that aim to solve a broad range of problems, DSLs are tailored to address particular needs, providing natural ways to express solutions for problems in specific domains. This specificity allows users to develop application programs more concisely, quickly, and effectively. Hudak argues that an ideally designed DSL should capture precisely the semantics of its application domain, making it easier for experts in that domain, who may not be professional programmers, to use the language to accomplish their goals. DSLs also find usage in creating graphical representations for domains such as engineering, education, and data visualisation, where visual clarity is vital in communicating complex concepts effectively.

A significant aspect of DSL design discussed by Hudak is the concept of modular interpreters. Modular interpreters allow the separation of language features into reusable components, providing a flexible framework for language extension. This modular approach means that interpreters can evolve incrementally new features can be added without disrupting the existing functionality. For example, different language constructs, such as arithmetic operations or function calls, can be integrated as independent modules, which are then composed to form a complete language. This flexibility makes modular interpreters particularly valuable for evolving DSLs to meet changing requirements or expanding them to accommodate additional features. By designing interpreters in a modular fashion, DSLs can maintain a high degree of adaptability while ensuring that the core semantics remain intact, thus providing an efficient and scalable approach to DSL implementation.

The use of DSL in program synthesis

Sarthi et al. 2021 presents an approach that uses program synthesis to create explicit string transformation rules for natural language processing (NLP) tasks. ProLinguist employs the Stateful Noisy Disjunctive Synthesis (Stateful-NDSyn) algorithm to generate interpretable rules from both small and large datasets. Experts can control the abstraction level of inferred rules through a domain-specific language (DSL) called FlashMeta a framework for inductive program synthesis, which specifies the transformation operations available to the synthesis algorithm. By embedding linguistic knowledge directly into the system, ProLinguist generates linguistically relevant rules efficiently. This approach is particularly effective for phonological tasks like grapheme-to-phoneme (G2P) conversion for Hindi and Tamil. G2P conversion transforms written text (graphemes) into phonetic representations (phonemes), crucial for applications like text to speech. ProLinguist models these transformations accurately, even in noisy conditions, making it a valuable tool for low-resource languages.

The paper highlights several advantages of using program synthesis techniques over traditional neural network approaches for NLP. Unlike sequence-to-sequence models, which require vast amounts of training data and produce opaque models, ProLinguist provides a framework for learning interpretable rules even with limited examples. The results of the study show that ProLinguist can achieve a comparable word error rate (WER) and phoneme error rate (PER) to state of the art models while requiring fewer training examples. Similarly, FlashMeta, as presented by Polozov and Gulwani 2015, is a framework designed to simplify the development of inductive program synthesizers by leveraging a data-driven domain-specific deduction (D4) methodology. FlashMeta enables efficient synthesis by separating domain-specific insights from the synthesis algorithm itself. This methodology has been successfully applied in industrial applications like Microsoft PowerShell and Azure Operational Management Suite. The combination of interpretability, domain-specific adaptability, and efficiency positions both ProLinguist and FlashMeta as valuable tools for NLP tasks requiring transparent rule-based transformations.

The Use of DSL in Facilitating Network Security Policy Verification

Shi et al. 2021 introduced the Network Policy Conversation Engine (NPCE), which allowed network operators to verify network security policies through natural language queries. The NPCE system integrated a custom Domain-Specific Language (DSL) with Natural Language Processing (NLP) to translate user-friendly questions into structured queries, enabling network operators to determine if specific network policies had been violated. The DSL acted as an intermediary between natural language inputs and low-level network traffic data queries, simplifying the translation of high-level directives into actionable network checks. The NPCE system architecture consisted of an NLP module for entity extraction, a query generation module, and a data storage component using Elasticsearch to store and analyze network traffic data. This design allowed both technical and non-technical users to interact easily with the system without needing to understand complex query languages or network configurations.

The evaluation of NPCE was conducted using real-world network policies gathered from university websites, demonstrating its effectiveness in practical scenarios like detecting prohibited services, identifying port scanning activity, and monitoring the use of insecure protocols such as File Transfer Protocol (FTP) and Telnet. These protocols, which transmit data in clear text, pose security risks and their use can indicate policy violations. The system translated questions like "Is there any FTP or Telnet traffic in the network?" into Elasticsearch queries that examined network traffic data collected by tools such as NetFlow and Tcpdump. NPCE utilized a rule-based model, which relied on a mapping layer that categorized and translated user inputs into specific technical details, ensuring that the generated queries accurately reflected the intended policy checks. This approach resulted in an effective, automated solution for monitoring network compliance, reducing manual errors, and making policy verification more accessible to a broader range of users.

2.1.3 Synthetic Dataset Generation for NLP Applications

Synthetic datasets, termed "synthsets," have emerged as a pivotal tool in computer vision and machine learning, addressing the significant challenge of collecting sufficient annotated data for supervised training. The review by Paulin and Ivasic-Kos 2023 explores the evolution of synthetic dataset generation techniques from early low-resolution generators to the latest generative adversarial networks (GANs) that produce highly realistic data. The authors note that synthetic datasets were first used in the 1980s for specific computer vision applications such as optical flow analysis and autonomous driving, significantly reducing the dependency on real-world data and offering a cost-effective solution to data scarcity. These datasets facilitate the generation of balanced, varied, and annotated data that is not always feasible to acquire from real-world environments, making them invaluable in complex, dynamic scenarios such as autonomous navigation, surveillance, and sports analysis.

The study identifies nine major generation methods, including game engines, commercial computer games, and physics-based simulations, and organizes these methods into a framework of 17 processes. These processes range from object rendering to noise addition and automatic annotation. The authors emphasize that the utility of synthetic data lies not only in the scalability it offers but also in the precision it brings to the annotation process, overcoming the limitations posed by human errors in manual labeling. With advancements in GANs and domain adaptation techniques, synthetic datasets can effectively minimize domain gaps, enhancing the generalization ability of models trained on synthetic data. This dynamic adaptability is crucial for keeping models updated with evolving real-world scenarios, especially in domains where data privacy, rarity, or cost are constraints.

A Framework for Synthetic Dynamic Dataset Generation

The SynDy framework, by Shliselberg et al. 2024, introduces an approach to synthetic dataset generation aimed at misinformation-related tasks. SynDy leverages Large Language Models (LLMs) to create

automatically labelled training datasets, thus reducing the dependency on costly, time-consuming, and potentially harmful manual annotations. The framework focuses on three key misinformation mitigation tasks: Claim Matching, Topical Clustering, and Claim Relationship Classification. Using prompt engineering and LLMs, SynDy generates synthetic labels that can be used to train models capable of improving the accuracy of misinformation-related tools. The authors emphasize the capability of synthetic datasets generated by SynDy to match the performance of models trained on human labelled data, with only a slight reduction in accuracy. These synthetic datasets thus provide an effective alternative in scenarios where annotated data is difficult to procure or human involvement could cause ethical concerns, such as misinformation targeting diaspora communities.

SynDy’s integration into Meedan’s misinformation mitigation tools, such as the chatbot tiplines, demonstrates its potential for real-world application. The framework allows small, low-resourced organizations to build specialized language models for misinformation tasks using synthetic data, thereby scaling up fact-checking previously labor-intensive efforts. The experimental results showed that SynDy-trained models perform comparably to human labelled models across various misinformation tasks, indicating that synthetic datasets could play a crucial role in the broader information ecosystem. Overall, SynDy’s contributions effectively generate synthetic training datasets at a fraction of the cost and time required for human annotations, making them suitable for misinformation mitigation, particularly in low-resource contexts.

A Framework for Synthetic Smart Home Data

The SynSys framework, as described by Dahmen and Cook 2019, provides an advanced approach to synthetic data generation for healthcare applications, specifically focused on smart home activity data. SynSys employs a two-level generative mechanism using Hidden Markov Models (HMMs) to generate activity sequences and sensor events, these sequences emulate real-life smart home data patterns. The first-level HMM generates a realistic sequence of activities, which is then expanded by second-level HMMs to create corresponding sequences of sensor events within each activity. This nested sequence structure captures the hierarchical nature of sensor-driven human activity data, which makes SynSys particularly effective in reflecting the complexities inherent in real-world behavioural patterns.

A significant contribution of SynSys is its integration with semi-supervised learning techniques to augment limited labelled datasets, thereby enhancing the performance of machine learning models for activity recognition. SynSys utilizes regression learners to generate realistic timestamps for sensor events, capturing the temporal characteristics of daily activities, while enforcing day-like structures through periodic resets. In comparative analysis using Dynamic Time Warping (DTW) and Euclidean distance measures, SynSys-generated data demonstrated greater similarity to real datasets than those generated using simpler HMM-based methods or random permutations. This high fidelity makes SynSys-generated synthetic datasets a valuable tool for initial testing and validation, especially in scenarios where real labelled sensor data is scarce or costly to obtain.

2.1.4 Concepts and Applications in Text Similarity Measurement

The concept of text similarity measurement has been pivotal in the progression of NLP technologies, it quantifies how similar two pieces of text are. This is crucial for text analysis tasks as they underpin various essential applications such as information retrieval, machine translation, question-answering, and document matching. Wang and Dong 2020 extensively surveys existing text similarity measurement methods, exploring both traditional and emerging approaches. The authors categorize these techniques into two main approaches: text distance and text representation. Text distance methods focus on measuring the semantic proximity between text segments through metrics like Euclidean distance, cosine distance, and distribution-based measures. In contrast, text representation methods include corpus-based

and semantic-based models. These models represent texts numerically to facilitate similarity measurement, ranging from simple string-based approaches to sophisticated neural network-based techniques. Their analysis serves as a critical reference point for research in this domain, highlighting the need to address the computational challenges of modern text representation methods, such as the graph-structure approach.

Concepts on Similarity Measures, Techniques, and Algorithms

Vijaymeena and Kavitha 2016 categorize these measures into three main types: String-based, Corpus-based, and Knowledge-based approaches. String-based measures, such as the Longest Common Substring (LCS) and Damerau-Levenshtein distance, compare character sequences to determine similarity at a granular level, useful for tasks like spelling correction. Corpus-based measures, including Latent Semantic Analysis (LSA), rely on large text collections to assess semantic similarity between words, enabling systems to capture deeper relationships, such as identifying synonyms. Knowledge-based measures utilize structured semantic resources, such as WordNet, to evaluate similarity based on shared conceptual information, which is particularly useful in understanding the meanings and relationships of words beyond their textual context.

Their paper provided a comprehensive survey of these similarity measures used in text mining and NLP, highlighting their importance in various applications like clustering and information retrieval. They discussed different algorithms and metrics for each type, providing examples to illustrate their effectiveness, such as the Needleman-Wunsch algorithm for String-based similarity, Hyperspace Analogue to Language (HAL) for Corpus-based similarity, and the Wu-Palmer metric for Knowledge-based similarity. The paper also reviewed various real-world applications of these similarity measures, such as their role in question answering, document clustering, short answer scoring, and machine translation. Additionally, it introduced advanced concepts, including the use of hybrid similarity measures that combine different approaches to improve performance in specific tasks.

2.2 Critical Analysis and Conclusion

The literature review presents a comprehensive overview of four primary themes: Natural Language Processing (NLP), Domain-Specific Languages (DSLs), synthetic dataset generation for NLP applications, and text similarity measurement. Each theme highlights significant advancements while also revealing limitations that underscore the necessity for further research, particularly in the application of NLP for generating circuit diagrams from natural language descriptions.

Natural Language Processing (NLP):

NLP has evolved remarkably from basic syntactic parsing to sophisticated machine learning-based approaches capable of real-time analysis of vast amounts of data Cambria and White 2014. Applications such as variability extraction Fantechi et al. 2021 and automated UML diagram generation Gulia and Choudhury 2016 showcase NLP's potential in specialized domains. However, these applications primarily address structured and semi-structured tasks. The complexity involved in translating nuanced and highly technical natural language descriptions into precise circuit diagrams remains a challenging frontier. Existing NLP tools often lack the deep semantic understanding and contextual interpretation needed for accurate technical diagram generation, emphasizing the need for further advancement in this area to meet the precision required for such tasks.

Domain-Specific Languages (DSLs):

DSLs provide tailored solutions that enhance efficiency within specific application domains Hudak 1997. Examples include program synthesis for string transformation tasks Sarthi et al. 2021 and network secu-

rity policy verification Shi et al. 2021. These applications illustrate the effectiveness of DSLs in creating interpretable and adaptable systems. However, developing DSLs demands significant domain expertise and their specificity can be limiting. While advantageous for targeted applications, this specialization poses challenges in scalability and adaptability to more complex or varied tasks, such as circuit diagram generation. Addressing this gap requires a unique integration of linguistic and technical capabilities to bridge the limitations in current DSLs.

Synthetic Dataset Generation for NLP Applications:

Synthetic datasets have proven invaluable in addressing data scarcity and enhancing model performance in various domains Paulin and Ivasic-Kos 2023. Frameworks like SynDy Shliselberg et al. 2024 and SynSys Dahmen and Cook 2019 demonstrate the capability of synthetic data to mimic real-world complexities and improve model training efficiency. However, generating high-fidelity synthetic datasets tailored to technical fields, such as circuit design, remains an under explored area. Existing synthetic dataset frameworks primarily focus on general or non-technical applications, highlighting a significant gap in methodologies capable of accurately representing technical descriptions necessary for precise circuit diagram generation. This project contributes by focusing on the creation of domain-specific synthetic datasets to support technical language translation for circuit schematics.

Text Similarity Measurement:

Text similarity measures are fundamental to numerous NLP tasks, facilitating functions such as information retrieval and machine translation Wang and Dong 2020. Surveys categorize these measures into string-based, corpus-based, and knowledge-based approaches Vijaymeena and Kavitha 2016. While effective in general contexts, these existing measures fall short when applied to highly technical language that requires capturing detailed semantics and contextual nuances. This limitation can hinder accurate interpretation and conversion of technical descriptions into circuit diagrams. To address this, the project incorporates enhanced text similarity metrics tailored to the specific requirements of technical language, improving the mapping accuracy between natural language input and structured circuit diagram outputs.

2.2.1 Identified Gaps

1. **Limited Integration of NLP and DSLs for Detailed Technical Diagrams:** While the existing body of research includes examples where NLP aids in generating block diagrams or UML representations, these approaches fall short when applied to detailed technical diagrams involving wiring and complex connections. There is a distinct gap in research that examines how NLP and DSLs can be integrated effectively to support the translation of complex circuit language into comprehensive schematic diagrams. This project uniquely addresses the need for an end-to-end system capable of translating nuanced natural language descriptions into structured DSL outputs specifically designed for circuit schematics.
2. **Scarcity of Synthetic Datasets for Technical Use Cases:** The review highlights the use of synthetic datasets in NLP applications to overcome data scarcity; however, there is limited application in technical fields like circuit design. Most synthetic datasets are not tailored to capture the granularity and specificities required for training interpreters handling circuit schematics. Addressing this gap, this research emphasizes creating a synthetic dataset that encapsulates essential circuit components, configurations, and variations, enabling more precise and context-aware outputs.
3. **Adequacy of Text Similarity Measures in Technical Contexts:** Standard similarity measures such as cosine similarity and string-based comparisons have proven effective in general NLP tasks but may not fully address the detailed, context-sensitive needs of technical language. Existing metrics may struggle to handle the intricate relationships between components, ratings, and circuit con-

nections described in natural language. This project identifies the need for refined or combined similarity approaches that ensure accurate mapping from user input to technical representations.

2.2.2 Conclusion

While the literature review provided a robust overview of existing research, linking the findings more explicitly to the identified research gaps can further emphasize the originality and relevance of this work. The review highlighted the evolution of NLP techniques and their applications in automating technical diagram generation. However, as detailed in Section 2.2.1, the integration of NLP and DSLs specifically for generating complex circuit diagrams remains underexplored. This research builds on the strengths of prior NLP applications while addressing the need for a specialized approach capable of translating detailed circuit descriptions into structured schematic representations.

The significance of synthetic data in overcoming data scarcity was evident from studies such as Paulin and Ivasic-Kos 2023, which underscored the utility of synthetic datasets in model training. However, these methods were mostly applied to non-technical fields, pointing to a gap in high-fidelity synthetic datasets for circuit design tasks. By focusing on the development of a tailored synthetic dataset, this thesis contributes a foundational resource for NLP applications in technical education and industry.

Moreover, the review of text similarity measures by Vijaymeena and Kavitha 2016 showed that while traditional and corpus-based methods are effective for general NLP tasks, they may not capture the nuanced semantics required in technical contexts. This research addresses this limitation by incorporating a hybrid approach combining cosine similarity, schematic similarity, and Jaccard-based chunking similarity to enhance the precision of matches between user inputs and dataset entries. This integration not only supports the research gaps identified but also sets a precedent for future studies aiming to bridge NLP and domain-specific applications.

Chapter 3 will delve into the implementation details of the NL to DSL interpreter, including the design of the NLP pipeline, the generation of synthetic datasets, and the integration of approximate nearest neighbour (ANN) search techniques. This chapter will provide an in-depth view of the methodologies and tools employed to bridge the gap between natural language descriptions and domain-specific technical outputs.

Chapter 3

Implementation

The development and implementation of the NL to DSL interpreter serve as the core of this research, transforming natural language descriptions into structured schematics, primarily focusing on basic electrical circuits. This chapter outlines the technical methodologies, tools, and design choices that underpin the successful realization of the interpreter. The key components of the implementation include the design of the NLP pipeline, the synthesis of a searchable dataset, and the integration of approximate nearest neighbor (ANN) search techniques. The goal of this implementation is to develop a system that bridges the gap between human language and technical domain-specific syntax, such as Schemdraw, for generating accurate and contextually relevant circuit diagrams.

3.1 Methodology

This section begins with the design of a natural language processing pipeline capable of interpreting text inputs into vectorized forms. The system matches user descriptions with entries from a synthesized dataset of electrical circuit components and configurations. To address the challenge of limited pre-existing labeled datasets, a synthetic dataset is generated, enabling the interpreter to dynamically search for the best possible match. This dataset synthesis process employs Google Apps Script, which automatically constructs natural language and schematic pairs that are used as the search space.

Furthermore, the interpreter employs ANN search techniques to retrieve the most relevant schematics by comparing the vectorized representations of user inputs with the synthetic dataset. This hybrid approach leverages both semantic and structural similarities, combining cosine similarity, schematic similarity, and Jaccard-based chunking to identify accurate matches. In this chapter, we provide a detailed overview of the system architecture, the tools and technologies employed, and the integration of these components into both a graphical user interface (GUI) prototype and a Markdown editor prototype. These prototypes demonstrate the real-time generation of circuit diagrams from user inputs and illustrate the system's functionality in technical environments.

3.1.1 NL to DSL Pipeline Design

Natural Language Processing (NLP) plays a key role in interpreting textual data across various domains, including technical fields such as electrical circuit design. The NLP similarity matcher pipeline introduced here combines textual and schematic analysis to match user-input descriptions against a dataset of pre-defined entries. By cleaning and preprocessing the text, correcting spelling errors, and utilising SpaCy's language model, the system computes cosine similarities to gauge how semantically aligned the input is with the dataset descriptions. Simultaneously, it extracts technical elements like components, ratings, and connections, generating a schematic representation that forms the basis for schematic similarity computations. The pipeline's strength lies in its hybrid approach, combining both cosine similarity for text and schematic similarity for structure. These metrics are merged to deliver accurate matches that account for both linguistic content and technical structure. This dual-layered system enhances the ability to identify relevant descriptions within technical datasets, offering a powerful solution for text-to-schematic matching in engineering and related fields. Let's probe the inner workings of each part of the pipeline in Figure 3.1.

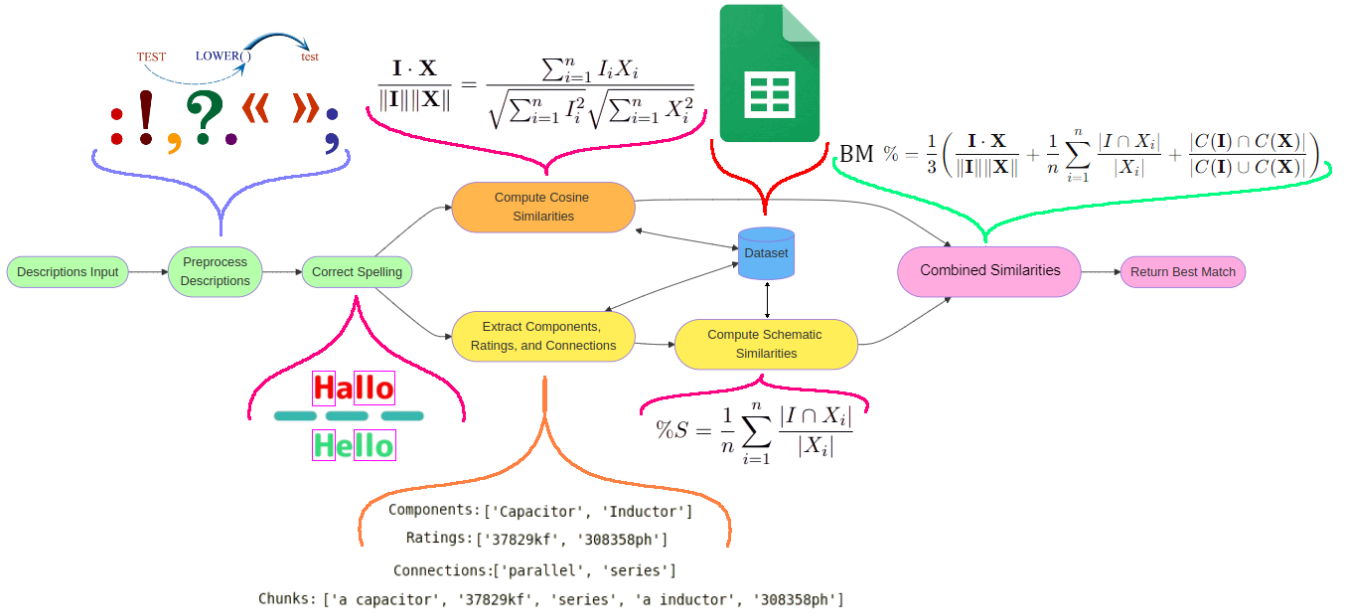


Figure 3.1: NLP Similarity Matcher Pipeline

Preprocess Input Descriptions

The processing of user-input descriptions is crucial to ensure uniformity and improve accuracy during analysis. As highlighted in (Etaiwi and Naymat 2017) removing punctuation marks plays an important role in text preprocessing by eliminating characters that do not contribute directly to the meaning of the text but can interfere with tokenization and word frequency calculations. By stripping out punctuation, we prevent discrepancies caused by symbols like periods and commas, which could alter the word frequency or comparison between text features. In conjunction with this, converting the text to lower-case is essential not only for preventing case sensitivity issues, where words like "Resistor" and "resistor" would be treated as different tokens but also for improving the effectiveness of spell-checking and auto-correction. Upper-cased words can hinder the spell-checking process, as they may be misinterpreted as proper nouns, such as a person's name. For instance, if a resistor is misspelled as "Resester", the auto-correct feature might not identify it as a technical term requiring correction and could leave it unaltered, assuming it to be a proper noun.

Spell Check & Auto-correction

The pipeline's Spell Check & Auto-correction function plays a crucial role in ensuring that user-input descriptions are free of spelling errors, which can hinder accurate similarity checking. The spell checker identifies and corrects misspelled words based on a preloaded dictionary of components, connections, and units. Misspelled words could cause issues during similarity matching, for instance, if a word like "resester" is misspelled and not corrected, the similarity check may fail to recognise it as a "resistor," leading to inaccurate results. The paper (Dashti et al. 2024) supports this approach by highlighting the importance of context-aware spell-checking in minimising errors during text analysis. Similarly, the spell checker in the pipeline ensures that common errors are corrected before the similarity-checking step.

Cosine Similarity Operation

In the pipeline, the Cosine Similarity Operation is used to measure how similar two text descriptions are by comparing them as vectors in a multi-dimensional space. This involves a process called vectorization, where words or phrases are converted into numerical representations that capture their meaning and context based on their relationships to other words. SpaCy's a Python based pre-trained language model is

employed to transform text descriptions into vectors, where each vector represents the semantic meaning of the entire description.

Example 1 Consider the following user-description:

- I = A resistor is in series with a capacitor, are rated 220ohm and 16uF respectfully

Let's assume The dataset corpus contains the following electrical circuit descriptions:

- A = A circuit with a 100 ohm resistor and a 10nF capacitor connected in parallel.
- B = Connected in series, are a 2mH inductor and a 22nF capacitor.
- C = The circuit includes a 50mH inductor and a 10pF capacitor in series, the combined unit is parallel to a 2.2Mohm resistor.

Next, the two sentences are processed through the SpaCy language model 'en_core_web_lg' where they are transformed into some numerical (1×300) vector space representing the sentences. Each vector encodes information based on how the sentence relates to other words in the language model's training data.

- $I = [-3.60128117, 4.29950505, -8.8440996, 1.26357329, \dots, 8.47496271]$
- $A = [-4.24620342, 1.17173064, -2.22489977, 1.43447065, \dots, 3.38199474]$
- $B = [-3.482512, 0.13526, -2.0611498, 1.4352629, \dots, 0.5889826]$
- $C = [-3.4131851, 1.8159893, -1.7825712, 0.889119, \dots, 1.2461514]$

Thereafter a cosine similarity computation is done to compare the two vectors by calculating the cosine of the angle between them given by;

$$\cos(\theta) = \frac{\mathbf{I} \cdot \mathbf{X}}{\|\mathbf{I}\| \|\mathbf{X}\|} = \frac{\sum_{i=1}^n I_i X_i}{\sqrt{\sum_{i=1}^n I_i^2} \sqrt{\sum_{i=1}^n X_i^2}}$$

the closer the angle is to zero, the more similar the vectors are, and hence the more similar the descriptions are. Cosine similarity ranges between -1 and 1 , where a positive 1 means the vectors are identical, while 0 means the vectors are completely different, and -1 indicates they are opposite in meaning (Akbas et al. 2014).

1. Cosine Similarity between **Input I & Descriptions A:**

$$\frac{\mathbf{I} \cdot \mathbf{A}}{\|\mathbf{I}\| \|\mathbf{A}\|} = 0.8805348$$

2. Cosine Similarity between **Input I & Descriptions B:**

$$\frac{\mathbf{I} \cdot \mathbf{B}}{\|\mathbf{I}\| \|\mathbf{B}\|} = 0.93922335$$

3. Cosine Similarity between **Input I & Descriptions C:**

$$\frac{\mathbf{I} \cdot \mathbf{C}}{\|\mathbf{I}\| \|\mathbf{C}\|} = 0.9132876$$

By transforming each sentence into a vector and calculating the cosine of the angle between these vectors, the NL to DSL pipeline measures how semantically aligned the user's input description is with the dataset descriptions. This approach ensures that even if words differ, the underlying semantic context is captured

and compared, enhancing the system's ability to identify relevant descriptions.

Component, Rating, & Connection Extraction for Schematic Similarity Check

In addition to capturing semantic similarity, it's essential to analyse the technical structure of the descriptions to ensure accurate matches. The Component, Rating, & Connection Extraction step focuses on identifying and comparing key technical elements between the user's input and the dataset descriptions. This involves isolating components (e.g., resistors, capacitors, etc.), ratings (e.g., ohms, microfarads, etc.), and the connections between them (e.g., series, parallel). Lets take a look at an example on how this similarity check is performed:

Example 2 Again let's consider the following user description from Example 1:

- I = "A resistor is in series with a capacitor, are rated 220ohm and 16uF respectfully"

Again we assume the dataset corpus contains the following electrical circuit descriptions:

- A = "A circuit with a 100 ohm resistor and a 10nF capacitor connected in parallel.
- B = Connected in series, are a 2mH inductor and a 22nF capacitor.
- C = The circuit includes a 50mH inductor and a 10pF capacitor in series, the combined unit is parallel to a 2.2Mohm resistor.

The first step is the extraction of identified components, ratings, and connections from the user's description and the dataset corpus descriptions:

Input I:

- **Components** = [resistor, capacitor]
- **Ratings** = [220ohm, 16uF]
- **Connections** = [series]

Description A:

- **Components** = [resistor, capacitor]
- **Ratings** = [100 ohm, 10nF]
- **Connections** = [parallel]

Description B:

- **Components** = [inductor, capacitor]
- **Ratings** = [2mH, 22nF]
- **Connections** = [series]

Description C:

- **Components** = [inductor, capacitor, resistor]
- **Ratings** = [50mH, 10pF, 2.2Mohm]
- **Connections** = [series, parallel]

The next step is the computation of Schematic Similarity (%S) between the **Input I** and each of the **Descriptions A, B, & C** in the dataset corpus using the formula:

$$\%S = \frac{1}{n} \sum_{i=1}^n \frac{|I \cap X_i|}{|X_i|}$$

1. Schematic Similarity between **Input I** & **Descriptions A**:

$$\%S = \frac{1}{3} \left(\frac{2}{2} + \frac{2}{2} + \frac{0}{1} \right) = 0.67$$

2. Schematic Similarity between **Input I** & **Descriptions B**:

$$\%S = \frac{1}{3} \left(\frac{1}{2} + \frac{1}{2} + \frac{1}{1} \right) = 0.67$$

3. Schematic Similarity between **Input I** & **Descriptions C**:

$$\%S = \frac{1}{3} \left(\frac{2}{3} + \frac{2}{3} + \frac{1}{2} \right) = 0.61$$

Thus extracting all technical elements allows for a Schematic similarity check, which calculates how well the components and their configurations and ratings align between the input and the dataset. This comparison ensures that the system doesn't just match descriptions on a semantic level but also on a technical one, thus the schematic similarity calculation ultimately refines the system's ability to match input descriptions with structurally similar entries in the dataset.

Combined Similarities to Generate Schemdraw Syntax

Now we discuss how, in addition to cosine similarity and schematic similarity, Jaccard similarity serves as the final piece of the puzzle to find the best match between user descriptions and the dataset, ultimately used to generate the Schemdraw syntax. Before we explore how Jaccard similarity functioned in the pipeline, it's important to review the foundational research that informed our approach.

Jaccard similarity, first introduced by Paul Jaccard in 1908, was designed to measure the overlap between two sets by dividing the size of the intersection by the size of the union (Alarcon 2019).

$$\mathbf{J}(A, B) = \frac{|A \cap B|}{|A \cup B|}$$

This formula has been widely used to compare sets of discrete items, such as words or phrases. Ivchenko and Honov 1998 extended this method into a generalized form, which allowed for more complex comparisons beyond simple set theory. Their work applied this generalization in the field of statistical analysis, particularly in testing for homogeneity of polynomial samples in datasets, which expanded the Jaccard index's potential applications.

This generalized approach, incorporating separable statistics, enabled us to adapt Jaccard similarity for use in chunking similarity. Chunking is a method used in NLP to break down a sentence into smaller syntactic units, such as noun phrases, without constructing a full parse tree. This approach provides a partial but computationally efficient analysis of the sentence structure, as discussed by Attardi and Dell'Orletta 2008, where they used chunking for shallow parsing in translation tasks. For the LN to DSL interpreter chunking similarity was calculated using the Jaccard index, comparing the sets of noun chunks extracted from both the input and the dataset descriptions.

$$\text{Chunk Similarity} = \frac{|C(\mathbf{I}) \cap C(\mathbf{X})|}{|C(\mathbf{I}) \cup C(\mathbf{X})|}$$

Here $C(\mathbf{I})$ and $C(\mathbf{X})$ represent the sets of chunks (noun phrases) from the user's input \mathbf{I} and the dataset description \mathbf{X} , respectively. By leveraging Jaccard similarity for chunking, we were able to measure how much of the sentence structure was shared between the input and the dataset. Thus the final similarity score by the NL to DSL interpreter combined three critical elements:

1. **Cosine Similarity:** Measured how similar the overall meaning of the text was between the user input and the dataset descriptions.
2. **Schematic Similarity:** Checked how well the components, ratings, and connections matched between the input and the dataset.
3. **Chunking Similarity:** Used Jaccard similarity to measure how much of the sentence structure was shared between the input and the dataset.

Where the best match percentage is calculated using the following formula:

$$\text{BM \%} = \frac{1}{3} \left(\frac{\mathbf{I} \cdot \mathbf{X}}{\|\mathbf{I}\| \|\mathbf{X}\|} + \frac{1}{n} \sum_{i=1}^n \frac{|I \cap X_i|}{|X_i|} + \frac{|C(\mathbf{I}) \cap C(\mathbf{X})|}{|C(\mathbf{I}) \cup C(\mathbf{X})|} \right)$$

The combination of Cosine Similarity, Schematic Similarity, and Jaccard-based Chunking Similarity ensures a comprehensive comparison between the user's input and dataset descriptions. These similarities are merged into a single "Best Match Percentage" that determines the closest dataset match. This integrated approach accounts for both the semantic meaning of the input and its technical details, making the system capable of producing moderately accurate Schemdraw syntax from natural language inputs.

3.1.2 Dataset Synthesis and Approximate Nearest Neighbour Searching

In the development of the NL to DSL Interpreter, the corpus dataset functions as a search space rather than a conventional training set. This dataset serves as a repository of preprocessed descriptions, components, ratings, and connections, allowing the system to dynamically retrieve and match relevant information from user inputs. By leveraging the vector-based similarity measures discussed earlier, such as cosine similarity and schematic matching algorithms, the dataset enables real-time component identification and schematic generation. This search-based approach aims to produce accurate, context-aware circuit diagrams from natural language descriptions without requiring extensive model training, as no existing datasets were available for such a machine learning model.

The dataset used within the NL to DSL interpreter framework focuses primarily on RLC (Resistor, Inductor, Capacitor) circuits, representing basic electrical configurations such as series and parallel, with varying power sources, including AC and DC supplies. Sheet 1 in the synthesized dataset contains foundational circuit elements with key attributes like resistance in ohms, capacitance in farads, and inductance in henries, along with corresponding units and prefixes (e.g., kilo, milli, micro).

Rather than functioning as a traditional training set, this dataset serves as a search space. The synthesis process uses Google Apps Script to generate structured pairs of natural language descriptions and corresponding schematic syntax in Sheet 3. These pairs provide the search space for the interpreter to operate in real-time.

How the Dataset Synthesis Process Works

1. **Template Generation:** Predefined templates stored in Sheet 2 contain placeholders for circuit components, their ratings, and configurations. These placeholders, such as `{COMPONENT_0}`, `{RATING_0}`, and `{CONFIGURATION_0}`, correspond to both the textual descriptions of circuits and their schematic syntax. Figure 3.2 demonstrates how these placeholders appear in the dataset.
2. **Random Component Selection:** The Google Apps Script dynamically selects random components, ratings, and configurations from Sheet 1. The placeholders in the templates are then replaced with actual values. For example, `{COMPONENT_0}` could represent a resistor in one iteration and a capacitor in the next. The power ratings (e.g., voltage or current values) are also generated randomly within set ranges, giving each description a unique specification.
3. **Population of Data:** Each template is used to generate multiple descriptions by iterating through various combinations of components and configurations. The final output a pair of natural language descriptions and their schematic syntax is stored in Sheet 3 as shown in Figure 3.3, thus building the synthesized dataset. This approach eliminates the need for manual data entry and enables the rapid creation of large datasets.

This method allows the system to create a synthetic search space instead of relying on a static, manually curated dataset. Since manually labeled data is not readily available for such tasks, synthetic data provides an efficient solution to approximate matches between user inputs and existing descriptions.

	A	B	C
37	Connected to a {POW_RATING_0} {POW_COMPONENT_NL_0} is a {RATING_0} {COMPONENT_0} arranged in {CONFIGURATION_0} with a {RATING_1} {COMPONENT_1}, and this {CONFIGURATION_0} combination is linked in {CONFIGURATION_1} with a {RATING_2} {COMPONENT_2}.	#(CONFIGURATION_0 === parallel && CONFIGURATION_1 ===series) e.Line().right() d.push() e.Line().up(d.unit*.5) e.(COMPONENT_0)().right().label("{RATING_0}") e.Line().down(d.unit*.5) d.pop() e.Line().down(d.unit*.5) e.(COMPONENT_1)().right().label("{RATING_1}") e.Line().up(d.unit*.5) e.(COMPONENT_2)().right().label("{RATING_2}") e.Line().down() e.Line().left() e.(POW_COMPONENT_DSL_0)().left().label("{POW_RATING_0}") e.Line().left() e.Line().up()	#(CONFIGURATION_0 === series && CONFIGURATION_1 ===parallel) e.Line().right() d.push() e.Line().up(d.unit*.5) e.(COMPONENT_0)().right().label("{RATING_0}") e.(COMPONENT_1)().right().label("{RATING_1}") e.Line().down(d.unit*.5) d.pop() e.Line().down(d.unit*.5) e.(COMPONENT_2)().right().label("{RATING_2}") e.Line().right() e.Line().up(d.unit*.5) e.Line().right() e.Line().down() e.Line().left() e.Line().left(d.unit*.5) e.(POW_COMPONENT_DSL_0)().left().label("{POW_RATING_0}") e.Line().left(d.unit*.5) e.Line().left() e.Line().up()
38	A {COMPONENT_0}, connected in {CONFIGURATION_0} with a {COMPONENT_1}, is connected in {CONFIGURATION_1} with a {COMPONENT_2} and are powered with a {POW_COMPONENT_NL_0}. Thier ratings are {RATING_0} , {RATING_1} , {RATING_2}, and {POW_RATING_0} respectively.	#(CONFIGURATION_0 === parallel && CONFIGURATION_1 ===series) e.Line().right() d.push() e.Line().up(d.unit*.5) e.(COMPONENT_0)().right().label("{RATING_0}") e.Line().down(d.unit*.5) d.pop() e.Line().down(d.unit*.5) e.(COMPONENT_1)().right().label("{RATING_1}") e.Line().up(d.unit*.5) e.(COMPONENT_2)().right().label("{RATING_2}") e.Line().down() e.Line().left() e.(POW_COMPONENT_DSL_0)().left().label("{POW_RATING_0}") e.Line().left() e.Line().up()	#(CONFIGURATION_0 === series && CONFIGURATION_1 ===parallel) e.Line().right() d.push() e.Line().up(d.unit*.5) e.(COMPONENT_0)().right().label("{RATING_0}") e.(COMPONENT_1)().right().label("{RATING_1}") e.Line().down(d.unit*.5) d.pop() e.Line().down(d.unit*.5) e.(COMPONENT_2)().right().label("{RATING_2}") e.Line().right() e.Line().up(d.unit*.5) e.Line().right() e.Line().down() e.Line().left() e.Line().left(d.unit*.5) e.(POW_COMPONENT_DSL_0)().left().label("{POW_RATING_0}") e.Line().left(d.unit*.5) e.Line().left() e.Line().up()

Figure 3.2: Dataset Template

	A	B
680	Connected to a 191.566Gvolts DC power supply is a 69.371Tfarad Capacitor arranged in series with a 15.319nfarad Capacitor, and this series combination is linked in parallel with a 346.404nohm Resistor.	e.Line().right() d.push() e.Line().up(d.unit*.5) e.Resistor().right().label("69.371Tfarad") e.Resistor().right().label("15.319nfarad") e.Line().down(d.unit*.5) d.pop() e.Line().down(d.unit*.5) e.Resistor().right().label("346.404nohm") e.Line().right() e.Line().up(d.unit*.5) e.Line().right() e.Line().down() e.Line().left() e.Line().left(d.unit*.5) e.SourceV().left().label("191.566Gvolts") e.Line().left(d.unit*.5) e.Line().left() e.Line().up()
681	A Capacitor, connected in parallel with a Capacitor, is connected in series with a Inductor and are powered with a DC supply. Thier ratings are 792.385MF , 87.457Gfarad , 289.639GH, and 965.453uvolts respectfully.	e.Line().right() d.push() e.Line().up(d.unit*.5) e.Inductor().right().label("792.385MF") e.Line().down(d.unit*.5) d.pop() e.Line().down(d.unit*.5) e.Inductor().right().label("87.457Gfarad") e.Line().up(d.unit*.5) e.Inductor().right().label("289.639GH") e.Line().down() e.Line().left() e.SourceV().left().label("965.453uvolts") e.Line().left() e.Line().up()

Figure 3.3: Synthesised Dataset

Once the synthetic dataset is complete, the NL to DSL interpreter employs Approximate Nearest Neighbor (ANN) search methods to match user inputs with existing descriptions. Tschopp and Diggavi 2009 research on ANN methods provides the theoretical foundation for this approach. Their work, especially in high-dimensional search spaces, highlights how ANN methods efficiently locate approximate matches between input data and large datasets. By utilizing this approach, the interpreter does not rely on conventional machine learning models, but instead uses vector-based similarity metrics discussed in section 3.1.1 cosine similarity, schematic matching and Jaccard similarity to find the closest matches between user descriptions and the generated synthetic data.

As discussed by Tschopp and Diggavi, ANN search algorithms operate by efficiently exploring large datasets and returning results that are close to the desired input, even when exact matches are unavailable. This method provides flexibility and scalability in handling real-time circuit generation, where user inputs might vary in format and complexity. Their contributions underscore the robustness of ANN techniques in multimedia and textual data searches, which is adapted here for matching circuit descriptions.

It is also crucial to acknowledge the limitations of using synthetic data and how it affects the generalizability to real-world inputs. While synthetic data offers significant advantages, such as privacy preservation and the ability to generate large datasets, it also presents several challenges that must be considered:

- **Bias and Inaccuracies:** Synthetic data is often generated based on existing real-world datasets. If the original data contains biases or inaccuracies, these issues can be propagated or even amplified in the synthetic dataset, leading to potentially misleading outcomes. This concern is discussed in studies highlighting the ethical considerations of synthetic data, emphasizing that biases in original datasets can reflect in synthetic versions (UK Statistics Authority 2022).
- **Loss of Outliers and Rare Events:** Synthetic data generation processes may not adequately capture outliers or rare events present in real-world data. These elements can be crucial for specific analyses, and their absence may result in a pipeline that fails to account for uncommon but significant scenarios. A review of synthetic data generation techniques notes that synthetic data often does not cover outliers present in the original dataset, which can be relevant for specialized research purposes (Dilmegani 2024).
- **Over-fitting to Synthetic Patterns:** A Pipeline that significantly relies on synthetic data might over-fit to patterns inherent in the synthetic dataset, which may not accurately represent the complexity and variability of real-world data. This over-fitting can lead to reduced performance when applied to actual data. Research indicates that synthetic data generation may fall short in maintaining the utility and flexibility required for real-world applications Goyal and Mahmoud 2024.
- **Validation Challenges:** Validating the quality and representativeness of synthetic data is inherently challenging. Without rigorous validation, there is a risk that the synthetic data may not reflect the statistical properties of real-world data accurately, leading to unreliable system performance. A systematic review of synthetic data generation techniques underscores the importance of addressing computational requirements, training stability, and privacy-preserving measures to ensure real-world usability Goyal and Mahmoud 2024.

Addressing these limitations requires careful consideration during the dataset synthesis process. However, for this research, these aspects were not fully addressed due to certain constraints. Specifically, limitations in resources and time meant that the focus remained on creating a functional, synthetic dataset capable of supporting the initial testing and proof-of-concept phase. The primary aim was to demonstrate the feasibility of the NL to DSL interpreter rather than optimizing for comprehensive generalizability to real-world inputs. Future work should incorporate strategies for mitigating these limitations by integrating real-world data and developing more robust validation techniques.

3.1.3 Prototype Designs and Integration

NL to DSL Interpreter Testing Prototype

The Graphical User Interface (GUI) serves as the interactive front-end for the NL to DSL interpreter testing prototype. It facilitates user interactions, system feedback, and the visualization of generated schematics. The design of the GUI aligns with the underlying NLP similarity matcher pipeline, ensuring an intuitive and efficient workflow for users. Each component of the GUI corresponds to specific stages of the processing pipeline, enhancing the system's functionality and supporting the testing objectives of the prototype. Figure 3.4 illustrates the layout of the GUI, highlighting the primary sections; Generated Schematic Display, System Feedback Display, Generated Code Editor, and Description Text Input. Additionally, interactive elements such as buttons and a progress bar facilitate user actions and provide real-time feedback on the processing status.

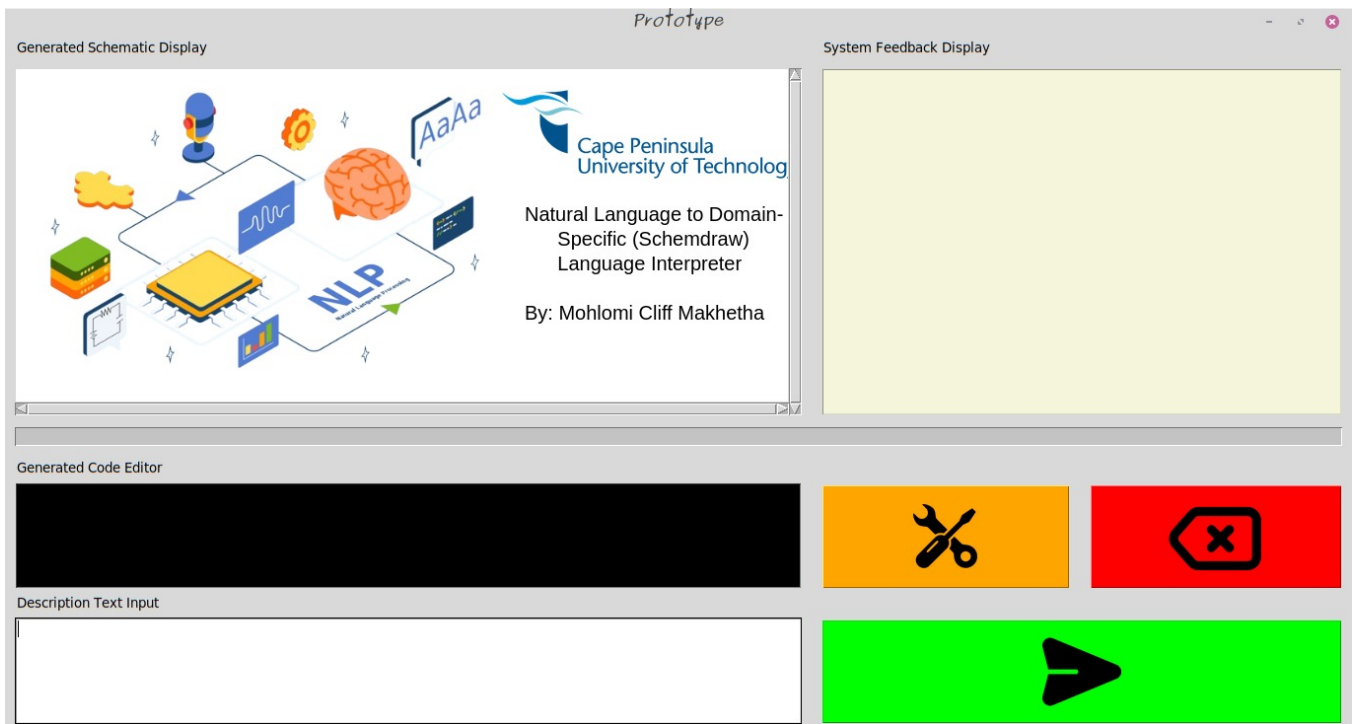


Figure 3.4: GUI Layout for NL to DSL Interpreter Testing Prototype

The Generated Schematic Display visualizes the interpreted DSL code as schematic diagrams. Utilizing the Schemdraw library, the system translates DSL syntax into electrical circuit diagrams, which are then displayed in this section of the GUI. This component is linked to the Schematic Generation phase of the pipeline. After the pipeline identifies the best match from the dataset and formulates the corresponding DSL code, the Inject Schematic function processes this code to produce the schematic. The resulting image is rendered and updated in the Generated Schematic Display.

The System Feedback Display provides real-time updates and comprehensive information about the system's processing stages. This text-based section communicates essential details, including input processing steps, similarity scores, schematic generation status, and any errors encountered during execution. Aligned with the pipeline stages, the display enhances transparency into the system's operations by presenting the raw and processed user inputs during input description processing, outputting cosine similarity, schematic similarity, and Jaccard similarity scores during similarity computations, notifying users about the success or failure of schematic generation in the schematic injection status, and providing timing information for search and rendering operations through performance metrics.

The Generated Code Editor is a text area where the DSL code corresponding to the best-matched schematic is displayed. This editor allows users to view and review the generated code. By presenting the DSL syntax, users gain insight into the instructions that define the schematic, facilitating a better understanding of the system's interpretation process. This component is connected to the DSL Generation phase. After the pipeline determines the best match based on combined similarities, it extracts the associated DSL code from the dataset. The Transform Code and Inject Schematic functions process this code, ensuring it is correctly formatted and executed to generate the schematic. The Generated Code Editor bridges the textual input and its schematic counterpart, encapsulating the translation process. This feature is particularly useful for testing the correctness and consistency of the generated DSL code.

The Description Text Input area is where users input their natural language descriptions of electrical circuits. This text field initiates the entire processing pipeline upon submission. This input engages with the Preprocess Input Descriptions and Spell Check & Auto-correction stages. When a user submits a de-

scription, the system cleans and preprocesses the text to ensure consistency and accuracy by removing punctuation, converting text to lowercase, and correcting spelling errors. This preparation is essential for effective similarity matching and schematic generation.

The GUI includes several interactive elements to facilitate user actions and monitor processing progress:

- **Send Button:** Initiates the processing of the user-input description. When clicked, the system begins similarity matching and schematic generation processes, providing visual feedback through the progress bar and System Feedback Display.
- **Clear Button:** Resets the input fields and feedback displays, allowing users to start a new session without residual data from previous interactions.
- **Correction Button:** Processes the current input from the Generated Code Editor without initiating a search for a new match, enabling users to refine or correct the existing schematic.
- **Progress Bar:** Indicates that the pipeline is still processing, serving as a visual indicator that the system is actively engaged in operations.

Thus, the Graphical User Interface (GUI) for the NL to DSL interpreter testing prototype provides a cohesive and user-friendly interface that seamlessly integrates with the underlying NLP similarity matcher pipeline.

NL to DSL Interpreter Markdown Prototype

A Markdown Editor serves as an innovative interface for integrating the NL to DSL interpreter pipeline within a familiar text-editing environment. This prototype demonstrates the feasibility of embedding schematic diagram generation directly into markdown documents using plain text natural language descriptions, thus enhancing the utility and accessibility of the NL to DSL interpreter. The design leverages the underlying pipeline to process user inputs seamlessly, providing real-time visualization of electrical circuit schematics alongside descriptive text.

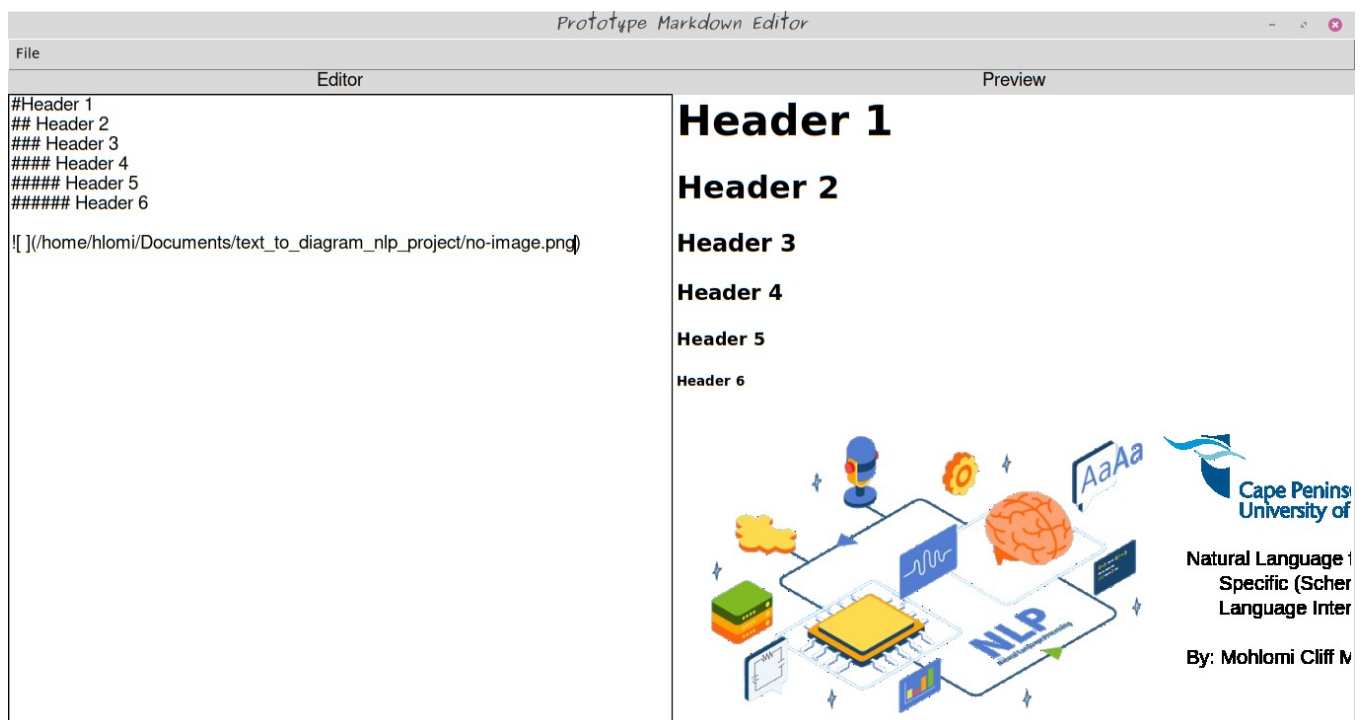


Figure 3.5: NL to DSL Interpreter Markdown Editor Integration Prototype

Figure 3.5 illustrates the layout of the Markdown Editor, which comprises two primary sections: the Editor Pane and the Preview Pane. The Editor Pane is dedicated to markdown text input, where users can write and edit using standard markdown syntax. Where upon detecting a specific markdown patterns `nlp-schem{<natural language description>}[render]` which signals the pipeline to process the enclosed natural language description. This processing involves generating the corresponding DSL code and subsequently creating the schematic diagram based on the interpreted instructions.

The Preview Pane, on the other hand, renders the formatted markdown content, including the dynamically generated schematic diagrams. The system utilizes the Markdown library to convert markdown text into HTML, enabling a live preview of the formatted document within the Preview Pane. Additionally, the integration with tkhtmlview allows for the display of rendered schematic diagrams. When the pipeline generates a schematic, it is first converted from SVG to PNG format using cairosvg to ensure compatibility, and then it is embedded within the preview.

Lastly the editor includes standard file operations such as opening and saving markdown files, facilitated by the `openfile` and `savefile` functions, respectively. The `openfile` function allows users to load existing markdown documents into the editor, while the `savefile` function enables users to preserve their current markdown content. These functionalities ensure that users can manage their documents effortlessly within the integrated environment, maintaining continuity and organization in their work.

This prototype successfully demonstrates the integration of the NL to DSL interpreter pipeline within a Markdown Editor, providing a proof of concept for generating circuit diagrams from natural language descriptions. By seamlessly embedding schematic generation into a widely-used text-editing environment, the system enhances the accessibility and utility of the interpreter. This integration paves the way for more sophisticated and user-centric applications in technical documentation and design, offering users a powerful tool for creating and visualizing electrical circuits through intuitive natural language inputs.

3.2 Tools and Technologies

Python was selected as the primary programming language for developing both the GUI Testing Prototype and the Markdown Editor prototype due to its exceptional versatility and robust ecosystem of libraries tailored for diverse application needs. Python’s readability and straightforward syntax facilitate rapid development and ease of maintenance, which are crucial for prototyping and iterative testing (Yerygin 2024). Additionally, Python’s extensive standard library and third-party packages provide the necessary tools to implement complex functionalities required by the NL to DSL interpreter pipeline effectively.

The development of these prototypes leverages a wide array of Python libraries, each serving a specific purpose to enhance functionality and performance see Table 3.1.

Table 3.1: Libraries Used for Prototyping

Library	Functionality	Prototype Integration
Tkinter	Standard Python interface to the Tk GUI toolkit, used for creating graphical user interfaces.	Forms the backbone of the GUI prototype, providing widgets and layout management for user interactions.

Continued on next page

Library	Functionality	Prototype Integration
threading	Enables concurrent execution of tasks, allowing the application to remain responsive during long-running operations.	Used in both prototypes to handle background processing such as similarity computations and schematic generation.
markdown2	Converts markdown text to HTML, facilitating the rendering of formatted documents.	Integral to the Markdown Editor prototype for translating user-written markdown into HTML for live preview.
tkhtmlview	Renders HTML content within Tkinter applications, enabling the display of rich text and embedded images.	Utilized in the Markdown Editor prototype to display the HTML-rendered markdown and embedded schematic diagrams.
cairosvg	Converts SVG files to PNG format, ensuring compatibility with various display components.	Used to transform generated SVG schematics into PNG images for embedding within the Preview Pane of the Markdown Editor.
schemdraw	A Python library for creating schematic drawings, particularly electrical circuits.	Employed in both prototypes to generate accurate electrical circuit diagrams based on DSL code.
PIL (Pillow)	Python Imaging Library for image processing tasks such as opening, manipulating, and saving images.	Facilitates image handling and manipulation, particularly for displaying schematics within the GUI and Markdown Editor.
Pandas	Data manipulation and analysis library, providing data structures like DataFrame for handling structured data.	Manages datasets retrieved from Google Sheets, enabling efficient data processing and retrieval for similarity matching.
SpaCy	Advanced NLP library for tasks such as tokenization, part-of-speech tagging, and vectorization.	Processes user inputs and dataset descriptions, performing text preprocessing and generating vector representations.
NumPy	Fundamental package for numerical computations in Python, supporting large, multi-dimensional arrays and matrices.	Performs high-performance numerical operations essential for calculating similarity scores.
SpellChecker	Library for correcting spelling errors in text, enhancing input accuracy.	Corrects spelling errors in user inputs, ensuring that the similarity matching process is accurate and reliable.
re (Regular Expressions)	Provides support for regular expression operations, enabling complex pattern matching and text manipulation.	Detects custom markdown patterns and cleans user input text, facilitating accurate pattern recognition and processing.

Continued on next page

Library	Functionality	Prototype Integration
difflib	Module for comparing sequences, providing tools for computing similarity measures between strings.	Utilized for similarity matching, identifying the closest matches between user inputs and dataset descriptions.
tqdm	Provides progress bars for loops and other iterative processes, enhancing user feedback during long-running tasks.	Displays progress indicators during batch processing of descriptions and similarity computations.
joblib	Library for parallel processing, enabling the distribution of tasks across multiple CPU cores to accelerate computations.	Accelerates similarity computations and schematic generation by parallelizing these tasks across available cores.
functools.lru_cache	Decorator for caching the results of function calls, improving performance by avoiding redundant computations.	Caches spell correction results and other repetitive function calls, speeding up data retrieval and processing tasks.
gsread	Python API for interacting with Google Sheets, allowing for reading and writing data to spreadsheets.	Retrieves and updates datasets stored in Google Sheets, ensuring real-time synchronization and accessibility.
annoy	A library for approximate nearest neighbors, used for efficient similarity searches within large datasets.	Facilitates accelerated similarity matching by enabling fast approximate nearest neighbor searches in the NL to DSL interpreter pipeline.
oauth2client	Provides OAuth 2.0 support for authenticating and authorizing access to Google APIs.	Secures access to Google Sheets data, enabling safe and authorized data retrieval for the prototypes.
Matplotlib	Comprehensive library for creating static, animated, and interactive visualizations in Python.	Generates graphical representations of data and performance metrics, enhancing the visual feedback within the prototypes.
Google App Script	JavaScript-based scripting language for automating tasks across Google products, enabling custom workflows and integrations.	Automates the synthesis and updating of datasets in Google Sheets, facilitating accelerated searching and data management.

Additionally, the prototypes were developed and tested on a Toshiba Tecra M11 laptop running Linux Mint 21.3 Virginia, based on Ubuntu 22.04 Jammy with a 5.15.0-122-generic kernel. The machine is equipped with an Intel Core i5 M560 dual-core processor operating at an average speed of 2.97 GHz, 8 GB of RAM, and integrated Intel HD Graphics, ensuring efficient performance during development and testing. This robust hardware setup facilitated smooth execution of concurrent processing tasks and efficient handling of large datasets, thereby validating the effectiveness and scalability of the NL to DSL interpreter pipeline.

3.3 Conclusion

Chapter 3 detailed the implementation of the Natural Language to Domain-Specific Language (NL to DSL) interpreter, focusing on the system's architecture and key methodologies. The chapter began by outlining the NL to DSL pipeline design, which integrates Natural Language Processing (NLP) techniques with domain-specific knowledge to interpret textual descriptions of electrical circuits. The interpreter utilizes cosine similarity, schematic similarity, and Jaccard-based chunking similarity to match user input descriptions with a predefined dataset of circuit components and configurations.

The pipeline processes input descriptions by cleaning and preprocessing the text, performing spell checks and corrections, and then applying cosine similarity to measure how semantically aligned the input is with existing descriptions. It extracts key technical details, such as components, ratings, and connections, to calculate schematic similarity, ensuring that both textual and structural elements of the input are matched accurately. The combination of cosine similarity, schematic similarity, and chunking similarity enables the generation of Schemdraw syntax, which is rendered into circuit diagrams using a graphical user interface (GUI) and a Markdown editor prototype.

The system successfully handles basic components like resistors, capacitors, and inductors, providing a functional means of generating circuit diagrams from natural language descriptions. The tools and technologies used, such as Python libraries like SpaCy, Pandas, and Schemdraw, were essential in developing and integrating the interpreter into these platforms.

In the next chapter, Chapter 4, the results of testing this interpreter will be presented. This includes an evaluation of the system's accuracy, performance, and limitations through various test cases. The chapter will also provide an in-depth analysis of the data generated during testing, focusing on the interpreter's ability to handle diverse inputs and the success of its similarity matching algorithms in generating accurate schematics.

Chapter 4

Results and Findings

This chapter delves into the results obtained from testing the NL to DSL interpreter and analyzes the findings to evaluate its performance. The section is structured to present the outcomes of various tests, showcasing how effectively the system can convert natural language inputs into structured DSL code for circuit diagram generation. Through detailed data presentations, analysis, and performance comparisons, this chapter provides insights into the strengths and limitations of the model, highlighting key areas of success and potential areas for further enhancement.

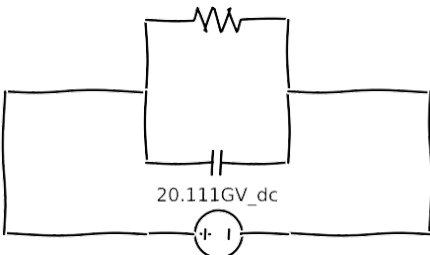
4.1 Result Presentation

In testing the functionality of the NL to DSL interpreter pipeline, which was integrated with a GUI testing prototype, input descriptions that were not part of the original dataset were used. This approach allowed for a comprehensive evaluation of the system’s ability to interpret and generate accurate schematic representations from unseen natural language inputs. As illustrated in Table 4.1, the descriptions involve fundamental RLC circuit elements (Resistors, Inductors, Capacitors) arranged in basic series and parallel configurations, powered by either AC, DC, or battery supplies.

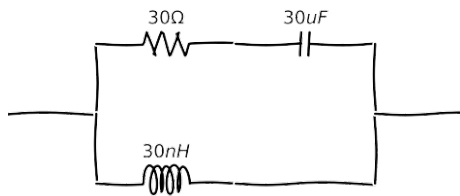
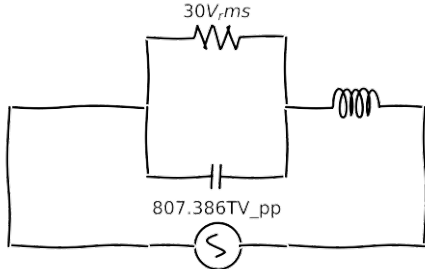
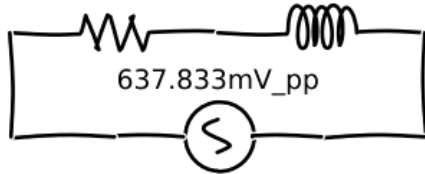
It is important to note that the current pipeline is designed to handle elementary circuits, specifically utilizing RLC components and their combinations. The pipeline successfully parsed these descriptions, generating appropriate schematic diagrams and matching them to the closest possible configurations within the dataset. However, minor errors were noted in relation to the power supplies. For example, in the first row of the table, the description "Place a resistor in parallel with a capacitor and connect a battery to the circuit" was interpreted correctly, except for the power supply, which was represented as a DC supply instead of the intended battery. The system processed this result in 3.21 seconds, demonstrating both the efficiency and precision of the interpreter in dealing with previously unseen inputs.

Each row in Table 4.1 provides insight into the system’s output, illustrating how it interprets input descriptions and returns not only the best-matched schematic but also the time taken to process each query. These results underscore the pipeline’s capacity to generate circuit diagrams from natural language descriptions efficiently, even when the descriptions are not part of the pre-existing dataset.

Table 4.1: Input Description, Best Match, & Generated Image

Input Description	Best Match and Time Taken	Generated Image
Place a resistor in parallel with a capacitor and connect a battery to the circuit	There is a 259.281nohm Resistor connected in parallel with a 282.060Gfarad Capacitor connected to a 20.111GV_dc DC supply (Total Time: 3.21 seconds)	

Continued on next page

Input Description	Best Match and Time Taken	Generated Image
A 30\Omega resistor is in series with a 30uF capacitor, they are then placed in parallel with a 30nH inductor	A 999.936mohm Resistor is arranged in series with a 235.283MF Capacitor, and this series combination is linked in parallel with a 704.247mF Capacitor. (Total Time: 0.18 seconds)	
A resistor in parallel with a capacitor are connected to an inductor in series, they are then powered by a 30V_rms AC supply	The circuit includes a 925.445uhenry Inductor and a 676.653uF Capacitor in parallel, with the combined unit connected in series to a 367.727kH Inductor. The circuit is then powered by 807.386TV_pp AC power supply (Total Time: 0.54 seconds)	
How would a circuit powered by an AC supply look like if it were connected in series with a resistor and an inductor	In series, a 785.742kohm Resistor is combined with a 489.547Ghenry Inductor. The circuit is energized by a 637.833mV_ppACpowersupply (TotalTime : 0.19seconds)	

The following illustrates how the integration of the NL to DSL interpreter pipeline with a simple Markdown editor has resulted in an efficient tool that can parse natural language descriptions and generate schematic representations directly within the editor. This prototype allows users to input descriptions of electrical circuits in plain text, which the pipeline processes to produce corresponding circuit diagrams. As shown in the Figure 4.1, the editor consists of an input section where descriptions are written and a preview section displaying the rendered diagrams. The image illustrates an example use case where the editor is set up to create an Electrical 1 exam paper, showcasing how the tool can assist with generating educational content that includes visual schematics seamlessly embedded within written material.

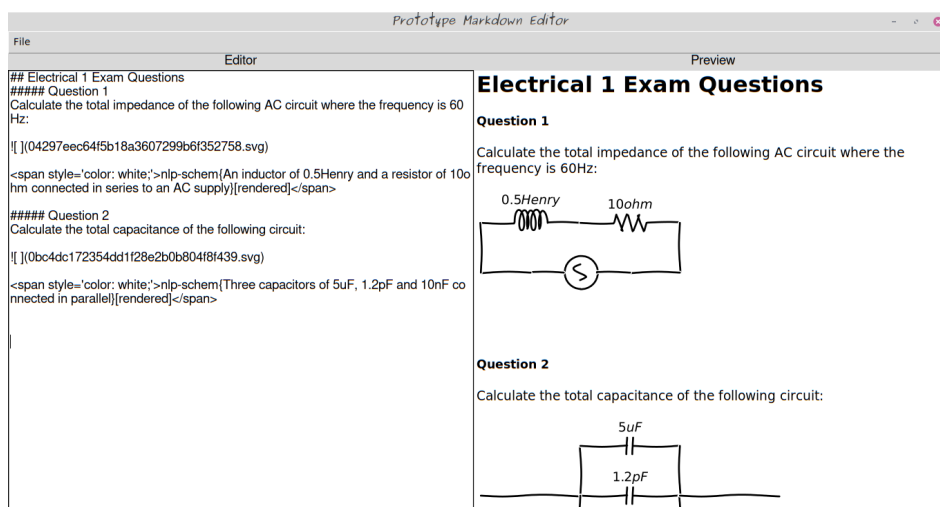


Figure 4.1: Prototype Markdown Editor Demonstrating NL to DSL Interpreter Integration

Having demonstrated the integration of the NL to DSL interpreter pipeline with both the Markdown editor and a GUI testing prototype, it is essential to assess the robustness and precision of the system under varied conditions. To achieve this, comprehensive testing was conducted to evaluate the pipeline's performance when subjected to challenging input scenarios. This evaluation begins with an analysis of how the pipeline performs when noise is introduced into input descriptions.

Performance Testing

Following the initial assessment of the NL to DSL interpreter pipeline's capability to interpret natural language descriptions into schematic diagrams, a test was devised to evaluate the system's resilience and precision. This was achieved through two main tests; Test 1 aimed to evaluate the NL to DSL pipeline's performance when noise was introduced into the input descriptions.

Where in the context of this test, noise refers to unintended deviations or errors introduced into the natural language input descriptions. These deviations could arise due to human error, unclear communication, or other unpredictable factors that may occur in real-world usage. By introducing noise into the inputs, we simulate how the system might handle imperfect or corrupted descriptions, mimicking scenarios where a user might make mistakes while describing circuits.

The types of noise introduced in this test include:

- **Spelling Errors:** Deliberate modifications to individual characters within words to simulate common spelling mistakes.
- **Word Deletions:** Random omission of words in the input, representing incomplete or fragmented descriptions.
- **Synonym Substitution:** Replacement of technical terms or components with their synonyms, testing the system's adaptability to linguistic variation.
- **Numeric Value Alterations:** Modifications to the numeric values describing component ratings (e.g., resistance, capacitance, voltage), simulating erroneous inputs.
- **Component Substitution:** Replacing circuit components (e.g., resistors with capacitors) to assess the system's ability to handle incorrect component names.
- **Connection Substitution:** Altering the configuration of components (e.g., switching series to parallel) to test the system's understanding of circuit layout.
- **Unit and Prefix Substitution:** Random changes to units (e.g., replacing "kilo" with "mega") to simulate miscommunication in unit specifications.

To evaluate the pipelines robustness, resilience, and precision noise was introduced incrementally at levels ranging from 0% to 100% in 5% increments. Each level of noise represents a progressively higher degree of spelling errors in the input descriptions. For instance, at 0% noise, the input descriptions were perfectly accurate, while at 100% noise, the input descriptions contained significant errors, including spelling, component, and configuration mistakes. This method allowed us to observe the interpreter's performance under varying conditions, assessing how well it could still parse and generate accurate schematic diagrams despite the presence of errors in the input.

The test involved taking 20 random descriptions from the synthesised dataset, denoted as *Test Descriptions*. Noise was introduced to these descriptions, generating *Noisy Inputs*. These Noisy Inputs were then fed into the pipeline, and the system generated corresponding *Predicted Descriptions*. The evaluation proceeded by comparing the *Predicted Description* generated by the system against the *Test Description* (the original, noise-free description). This comparison was performed using the pipelines cosine similarity,

schematic similarity, and Jaccard similarity. These scores were then used to compute a confusion matrix as in Figure 4.2, which allowed us to visualize the system’s classification performance and generate additional analysis metrics, such as accuracy at different noise levels.

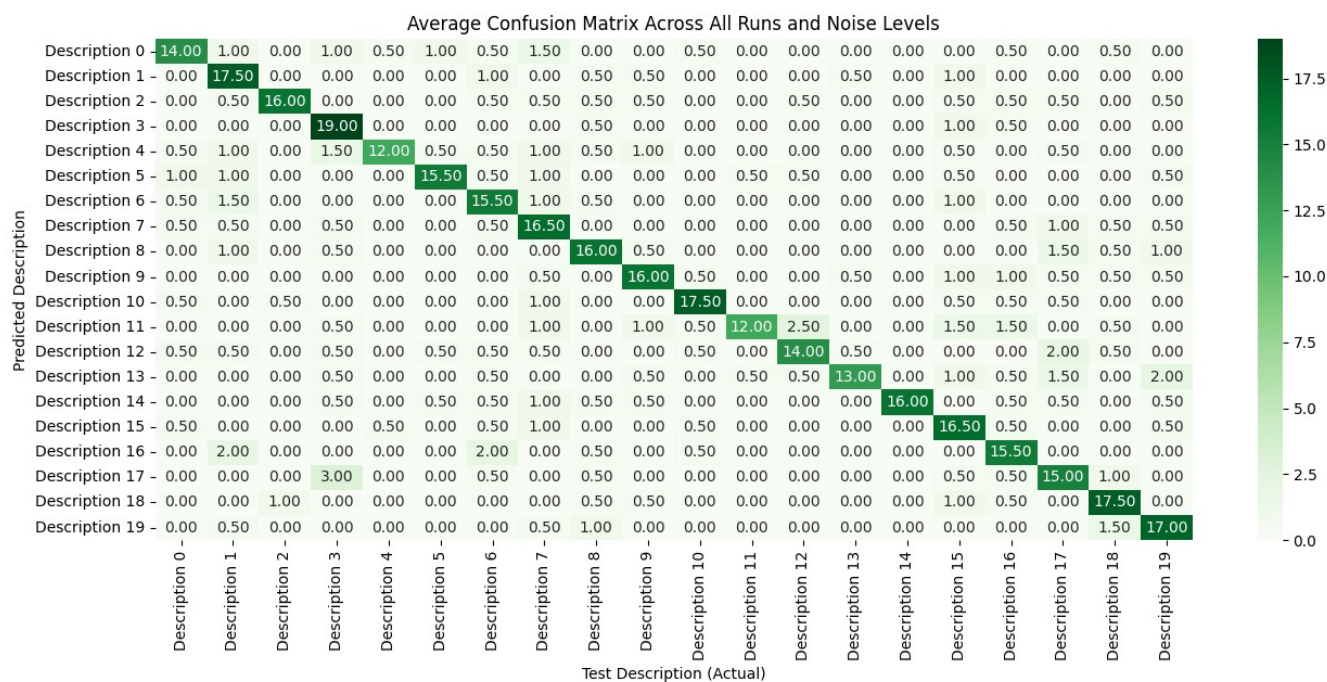


Figure 4.2: Test 1 Computed Confusion Matrix

To further assess the performance of the NL to DSL interpreter pipeline, the average accuracy of the system was measured across different noise levels, ranging from 0% to 100% in 5% increments. The accuracy metric reflects the proportion of test cases where the Predicted Description exactly matched the Test Description. Figure 4.2 (a) presents the average accuracy of the system plotted against noise levels. In addition to evaluating performance across varying noise levels, the average accuracy was also computed for each test case. This provided insight into how individual descriptions fared when subjected to noise. The bar graph in Figure 4.2 (b) represent the average accuracy across all noise levels for each of the 20 test cases. Note that the tests were conducted five times in succession, where each noise level performs 20 test and the results presented in the confusion matrix, the average accuracy vs. noise level, and the average accuracy per test case are the aggregated averages of those five successive tests.

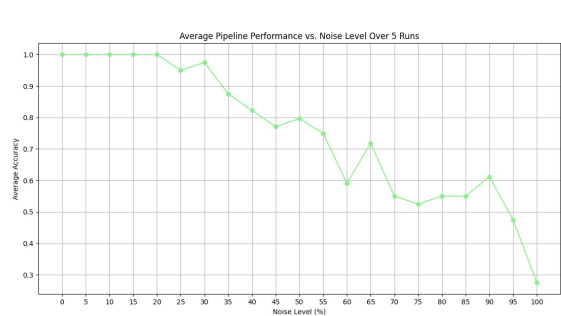


Figure 4.2 (a): Test 1 Average Accuracy vs. Noise Level

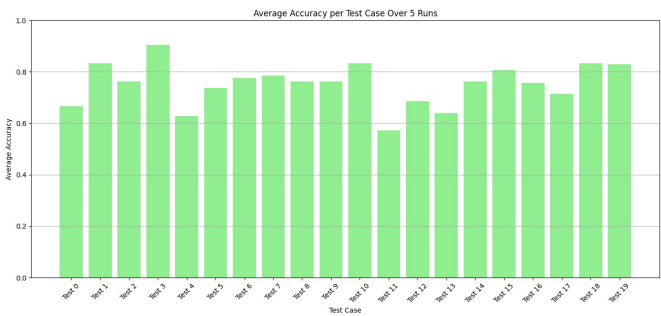


Figure 4.2 (b): Test 1 Average Accuracy per Test Case

In the second stage of evaluating the NL to DSL interpreter, a test was done on the system’s adaptability to linguistic variations. Unlike Test 1, which focused on handling noisy inputs, Test 2 evaluated the interpreter’s performance with paraphrased input descriptions to assess precision and robustness. To accomplish this, utilising the PEGASUS transformer-based language model to generate paraphrased versions of the original input descriptions. These paraphrased inputs preserved the original meaning while using different vocabulary and sentence structures. The choice of PEGASUS was inspired by the study by Muia et al. 2024, which conducted a comparative analysis of several transformer models like GPT, T5, BART, and PEGASUS. The findings demonstrated PEGASUS’s strength in abstractive summarization through its unique gap-sentence generation approach, making it suitable for generating rephrased sentences while preserving contextual integrity. This capability aligns well with our goal to evaluate the system’s ability to interpret diverse linguistic variations while retaining the core information conveyed.

Twenty randomly selected descriptions were paraphrased, and the interpreter’s responses were compared to the original inputs to determine its ability to generate accurate schematic diagrams. Cosine similarity, schematic similarity, and Jaccard similarity were used to measure performance. The results were summarized through an accuracy score and visualized using a bar chart in Figure 4.3 which provides an in-depth view of the average accuracy per which focused on inputs modified with paraphrasing noise.

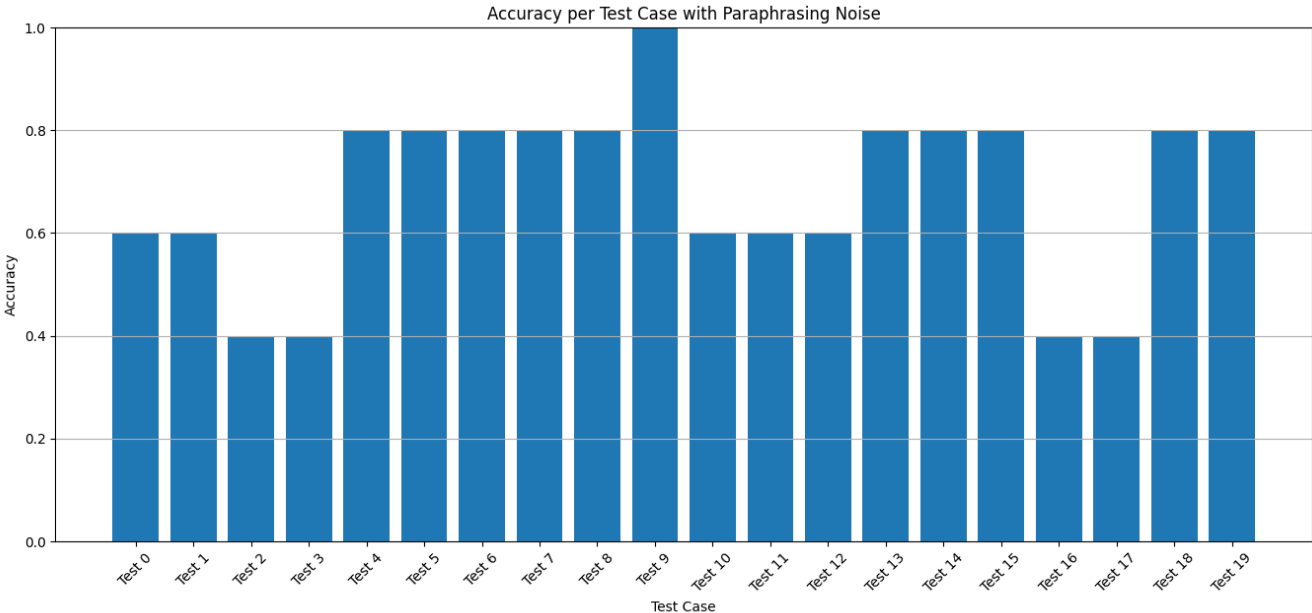


Figure 4.3: Test 2 Accuracy per Test Case Analysis

Finally an evaluation of the combined performance of Test 1 and Test 2 aimed to provide a comprehensive analysis of the NL to DSL interpreter pipeline. This combined test was designed to simulate a realistic environment where the system might encounter input descriptions that include both noise (as seen in Test 1) and linguistic variations (as in Test 2). The goal was to observe how well the interpreter could manage these challenges simultaneously and to assess its robustness and precision in such scenarios.

The comprehensive evaluation of the combined performance from Test 1 and Test 2 provided a more realistic measure of the NL to DSL interpreter pipeline’s robustness. This combined test scenario was designed to replicate practical situations where input descriptions contain both noise from Test 1 and linguistic variations from Test 2. Figure 4.4 displays the confusion matrix summarizing the combined results, showing how the interpreter managed the dual challenges of noise and paraphrasing in input descriptions.

Notably, Figure 4.5 compares the average accuracy across noise levels, revealing a significant decline

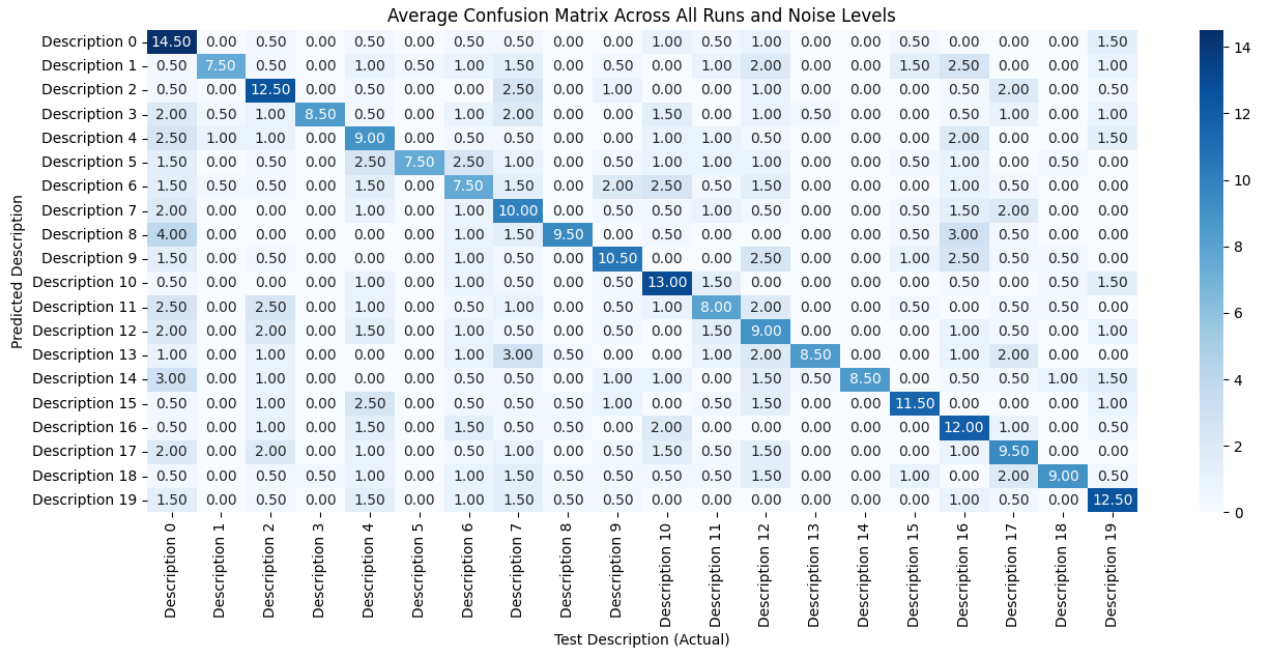


Figure 4.4: Test 1 & 2 combined Confusion Matrix

in performance starting at 30% noise level, particularly when paraphrasing was introduced. This highlighted the system's sensitivity to compounded input variations and underscores the need for enhancements to improve resilience and accuracy in interpreting diverse, imperfect descriptions.

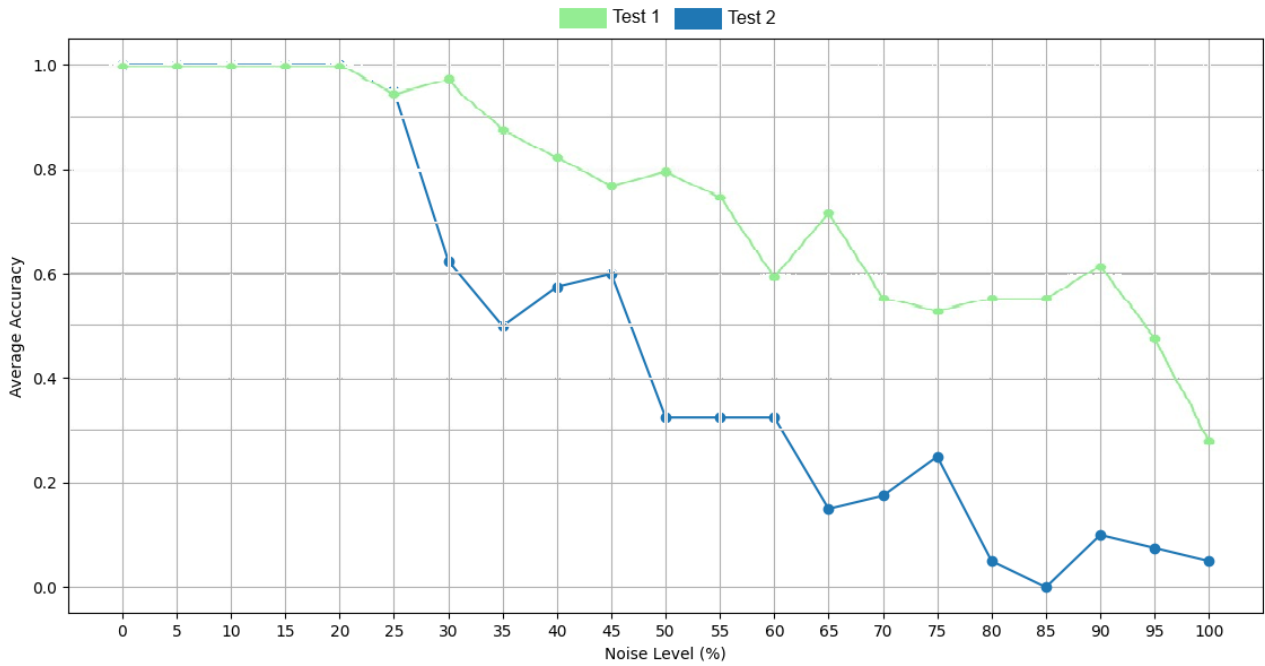


Figure 4.5: Test 1 & 2 combined Average Accuracy vs. Noise Level

Through this test, we gained valuable insight into the NL to DSL pipeline's ability to handle the natural variability in human language, which is crucial for real-world applications where users may describe circuit configurations in different ways. This evaluation contributes to our understanding of the system's capacity to adapt to input phrasing, helping to identify areas for further improvement.

4.2 Analysis of Performance Testing Results

Test 1 Confusion Matrix Analysis

The confusion matrix in Figure 4.2 provides an in-depth analysis of the NL to DSL interpreter’s performance in matching predicted descriptions to actual input descriptions across all test runs and varying noise levels. The strong diagonal dominance in the matrix, illustrated by high values such as 19 matches for "Description 3" and 17.5 matches for "Description 1," indicates the model’s ability to accurately identify and match certain input descriptions even when noise is present. These results highlight the interpreter’s reliability when processing straightforward, less ambiguous circuit descriptions. However, off-diagonal entries, such as the 1.5 and 2.0 in the matrix, reveal instances where the interpreter misclassified the input, which suggests that semantic or syntactic variations introduced by higher noise levels challenge the model’s parsing accuracy. Descriptions with significant off-diagonal values, such as "Description 16" showing 2.0 misclassification, reflect scenarios where the interpreter struggled to maintain clarity amid complex or noisy data. Additionally, mid-level performance in descriptions like "Description 11" and "Description 12," with values of 12.0 and 14.0 on the diagonal, further indicates difficulties in handling intricacies such as component combinations and specific configurations.

Test 1 Average Accuracy vs. Noise Level and Average Accuracy per Test Case

Figure 4.2 (a) provides insight into the performance trends of the NL to DSL interpreter under varying noise levels and across different test cases. The line graph shows a clear pattern where the interpreter maintains high accuracy, exceeding 90%, when noise levels are moderate (up to 30-35%). This indicates that the model is resilient to low to moderate distortions in the input, effectively interpreting and generating correct outputs. However, as the noise level surpasses the 35% threshold, the accuracy begins to decline progressively, with a sharp drop seen at higher noise levels 90-100%, where the accuracy falls to around 30% accuracy. This decline highlights the model’s increasing difficulty in managing highly noisy or distorted inputs, showcasing a key area where improvements are needed to enhance noise tolerance and robustness. This interpretation aligns with the examples presented in Table 4.2, where different noise levels influenced the accuracy of the predicted descriptions.

Table 4.2: Test 1: Different Noise Levels and Different Test cases

Noise Level	Test case
30%	Test Description: Two Resistors in series, rated 941.781uohm, 519.346Mohm
	Noisy Input: deuce resistance components come in in series, by rated 941.781uohm, 519.346Mohm
	Predicted Description: Two Resistors in series, rated 941.781uohm, 519.346Mohm
	Score: 1.0000

Continued on next page

Noise Level	Test case
60%	<p>Test Description: A pair of Inductors is connected in parallel, and then placd in series with a combination of an Capacitor connected in parallel with another Resistor, they are rated 629.021mhenry, 629.021mhenry, 630.575ufarad, and 260.221uohm, respectively.</p> <p>Noisy Input: type volts pair of inductance connected in and combination of an se-ries some other Resistor, they are 715.875.021mhenry, 630.575ufarad, 260.221uohm, respectively.</p> <p>Predicted Description: A pair of Inductors is connected in series, and then placd in parallel with a combination of an Resistor connected in series with another Capacitor, they are rated 430.491Thenry, 430.491Thenry, 656.502pohm, and 988.894Gfarad, respec-tively.</p> <p>Score: 0.6811</p>
90%	<p>Test Description: A 425.015Gfarad Capacitor is placed in series with a 465.509Mohm Resistor.</p> <p>Noisy Input: Inductor, mix hertz 425.015Gfarad 749.798Tohm series, connectd ca-pacitance embody placed indium series with axerophthol 465.509Mohm Resistor.</p> <p>Predicted Description: Two Inductors is connected in parallel, and this parallel pair is in series with a combination of an Resistor in parallel with another Resistor, where the components are rated 435.088Thenry, 676.191p, 257.596Mohm, and 837.141Gohm, respectively.</p> <p>Score: 0.3260</p>

While in Figure 4.2 (b), the bar chart illustrating average accuracy per test case over five runs reveals generally consistent performance, with most test cases achieving accuracy levels between 0.7 and 0.8. This consistency suggests that the interpreter can reliably handle a range of inputs. However, certain test cases, such as Test 12 and Test 4, show accuracy below 0.7, indicating challenges in parsing these specific descriptions. These cases may involve more complex sentence structures or ambiguous language that the model struggles to interpret accurately. Conversely, test cases like Tests 3, 6, and 15 exhibit higher accuracy, nearing or exceeding 0.8, implying that these descriptions align closely with the training data or are more straightforward and unambiguous. When connected to the confusion matrix analysis, these observations emphasize that while the interpreter can perform well under certain conditions, variability in input complexity and noise significantly affects its overall precision.

Test 2 Accuracy per Test Case Analysis

The bar chart in Figure 4.3 provides an in-depth view of the average accuracy per test case for Test 2, which focused on inputs modified with paraphrasing noise. This analysis illustrates the interpreter's varied performance when handling altered inputs. High accuracy in certain cases, such as Test 9 and Test 18, with scores reaching up to 0.8 or above, suggests that the model was effective in recognizing and interpreting descriptions that maintained similar structural and semantic patterns as seen in training. Conversely, lower accuracy observed in cases like Test 3 and Test 16, with scores around 0.4 to 0.5, highlights challenges in processing paraphrased descriptions that introduced greater linguistic complexity or shifted core sentence structures. These differences are reflected in the examples shown in Table 4.3, where

the paraphrased input led to correct predictions in Examples 1 and 2, maintaining a perfect cosine similarity score of 1.0000. However, in Example 3, the altered structure resulted in a significant drop in similarity 0.8679 and an incorrect prediction, demonstrating that the model struggled with identifying component placement when the input deviated too far from its training set. These findings emphasize the need for further dataset expansion and model fine-tuning to better handle diverse linguistic variations in technical descriptions

Table 4.3: Test 2: Different Paraphrasing Test cases

Example	Test case
1	<p>Test Description: A 223.339phenry Inductor and a 967.760mhenry Inductor are connected in parallel, with this parallel circuit connected in series with a 502.073kfarad Capacitor.</p> <p>Noisy Input (Paraphrased): A parallel circuit connecting the 223.339phenry Inductor and the 967.760mhenry Inductor is connected in a series with the 502.073kfarad Capacitor.</p> <p>Predicted Description: A 223.339phenry Inductor and a 967.760mhenry Inductor are connected in parallel, with this parallel circuit connected in series with a 502.073kfarad Capacitor.</p> <p>Score (Cosine Similarity): 1.0000</p> <p>Correct Prediction: Yes</p>
	<p>Test Description: The circuit includes a 802.027pohm Resistor and a 208.971pfarad Capacitor in series, with the combined unit connected in parallel to a 549.710Thenry Inductor. The circuit is then powered by 540.984TV_dc DC power supply</p> <p>Noisy Input (Paraphrased): The circuit includes a Resistor and a Capacitor in a series, parallel to a 549.710Thenry Inductor and DC supply.</p>
	<p>Predicted Description: The circuit includes a 802.027pohm Resistor and a 208.971pfarad Capacitor in series, with the combined unit connected in parallel to a 549.710Thenry Inductor. The circuit is then powered by 540.984TV_dc DC power supply</p> <p>Score (Cosine Similarity): 1.0000</p> <p>Correct Prediction: Yes</p>

Continued on next page

Example	Test case
	<p>Test Description: A Inductor of 307.437GH is connected in parallel with a 161.156Thenry Inductorr, and this parallel combination is connected in series to a 296.200uH Inductor.</p> <p>Noisy Input (Paraphrased): A parallel combination of the 161.156Thenry and the 307.437GH Inductors is connected in a series.</p> <p>3 Predicted Description: Connectd in parallel are three Inductors of values 834.304Mhenry, 450.424Ghenry, and 429.431k. The configuration is powered by 485.900uV_dc DC supply</p> <p>Score (Cosine Similarity): 0.8679</p> <p>Correct Prediction: No</p>

Combined Test 1 and Test 2 Performance Analysis

The comparative performance analysis depicted in Figure 4.5 showcases how the NL to DSL interpreter performs when subjected to combined noise types across Test 1 and Test 2. The line graph reveals that in Test 1 (green line), where direct input distortions are present, the model maintains strong accuracy, particularly up to 30% noise levels, with accuracy consistently close to 1.0. This suggests that the interpreter is well-equipped to handle straightforward input alterations. However, as noise levels increase beyond 35%, a clear decline in performance is observed, indicating the model’s reduced effectiveness in parsing heavily noisy input.

Test 2, represented by the blue line, introduces an added layer of complexity by combining the existing noise from Test 1 with paraphrased noise. This additional paraphrasing significantly impacts the model’s interpretative accuracy, leading to a more pronounced drop in performance. Beyond the 25% noise threshold, the decline becomes steeper, highlighting how syntactic and semantic variations disrupt the interpreter’s ability to correctly parse and match input descriptions. At higher noise levels above 50%, accuracy falls below 0.4 and bottoms out at 100% noise, illustrating the interpreter’s substantial struggles with handling input that deviates from its learned patterns due to paraphrasing.

These findings are further supported by the confusion matrix in Figure 4.4, where Test 2 demonstrates an increased frequency of off-diagonal entries, signifying higher misclassification rates. For example, descriptions such as "Description 14" and "Description 17" show notable drops in accuracy, reinforcing the model’s difficulties when faced with paraphrased input. This variability indicates that while the interpreter can handle direct distortions relatively well, paraphrased modifications introduce a level of unpredictability that compromises its accuracy.

4.3 Summary Analysis of Test Performance

The performance analysis of the NL to DSL interpreter pipeline provided a comprehensive understanding of its capabilities and limitations when faced with various input challenges. This evaluation, conducted through Test 1 (with noise levels) and Test 2 (paraphrased inputs), offered key insights into the model’s robustness and areas for improvement.

Key Insights from Test 1 (Noise Analysis): The confusion matrix and accuracy trends for Test 1 show-

cased the interpreter's strong performance when handling moderate noise levels, with a high rate of correct predictions maintained up to 30-35% noise. This resilience was demonstrated by high diagonal values in the confusion matrix, indicating accurate matches for straightforward input descriptions. However, as noise levels increased beyond 35%, a notable decline in accuracy was observed. At extreme noise levels (90-100%), performance fell sharply to around 30%, reflecting the model's struggles to parse highly distorted inputs. This trend highlighted the need for noise resilience improvements, particularly for cases where descriptions involve intricate combinations or higher input complexity.

Insights from Test 2 (Paraphrased Inputs): The analysis of Test 2, focused on paraphrased inputs, revealed the model's challenges in adapting to linguistic variations. While certain test cases, such as Test 9 and Test 18, demonstrated high accuracy (0.8 or above), suggesting that the model could effectively parse familiar structures, other cases like Test 3 and Test 16 showed scores around 0.4 to 0.5. These results indicated difficulties when faced with significant paraphrasing that altered sentence structures or introduced linguistic complexity. The lower performance in these cases pointed to the model's limited adaptability to unexpected linguistic variations, emphasizing the need for expanding the dataset that includes a wider range of paraphrased examples to enhance interpretative flexibility.

Combined Performance Analysis (Test 1 and Test 2): When comparing the combined results of Test 1 and Test 2, it was evident that while the interpreter could maintain relatively strong performance with direct input noise (as observed in Test 1), the addition of paraphrased noise in Test 2 presented more significant challenges. The combined analysis indicated that paraphrasing, which introduces both syntactic and semantic shifts, led to a more rapid decline in accuracy beyond the 35% noise threshold. This decline was steeper than in Test 1, with accuracy dropping below 0.4 at higher noise levels (>50%). The confusion matrix for the combined tests reinforced these findings, showing increased off-diagonal entries and more frequent misclassifications. This trend suggested that while the model can parse direct input distortions with reasonable accuracy, its ability to manage combined noise and paraphrasing was compromised.

Overall Findings and Recommendations: The overall findings from Test 1 and Test 2 underscore that the NL to DSL interpreter pipeline is robust when handling moderate noise and direct input distortions but struggles with input variability that includes paraphrasing. The need for improvement is most pronounced when inputs involve both high noise levels and rephrased descriptions, where accuracy declines significantly. To enhance the interpreter's performance, further development should focus on diversifying the dataset with more examples featuring paraphrased and complex inputs. Additionally, integrating advanced NLP techniques could improve the model's resilience and accuracy in real-world scenarios, where linguistic variation is more common.

Chapter 5

Discussions and Conclusions

This chapter provides a comprehensive discussion of the project results and findings, emphasizing the key outcomes and their significance in the context of the objectives. Section 5.1, "Discussion of Project Results and Findings," elaborates on the performance and limitations of the Natural Language to Domain-Specific Language (NL to DSL) interpreter, examining its effectiveness in translating natural language descriptions into Schemdraw syntax. Additionally, this chapter highlights the broader implications of these findings and identifies areas for potential improvement and future research directions.

5.1 Discussion of Project Results and Findings

The project results demonstrate the effectiveness of the NL to DSL interpreter in translating natural language descriptions into structured Schemdraw code for generating circuit diagrams. The accuracy of the interpreter was evaluated through performance tests using synthesized datasets of electrical components and their configurations. The findings indicate that the hybrid approach, which combines cosine similarity, schematic similarity, and Jaccard-based chunking similarity, successfully captured both the semantic and technical aspects of user inputs. This comprehensive similarity check enabled the system to identify the most relevant matches from the dataset, leading to moderately accurate schematic diagrams.

The interpreter's integration into the Markdown editor further highlights its practicality for technical documentation. The real-time generation of circuit diagrams from plain text descriptions enhances the efficiency of the documentation process, particularly for users without in-depth knowledge of circuit design. The GUI and Markdown prototypes both demonstrated the ease of use, allowing non-experts to generate visual content with minimal effort. Despite the success, the results also pointed out limitations in handling complex circuits beyond basic RLC configurations and a reliance on the quality of synthesized datasets, which impacted the accuracy of the generated schematics in less common scenarios.

Overall, the findings suggest that the NL to DSL interpreter effectively addresses the gap between natural language input and circuit diagram generation, making technical content creation more accessible. However, further improvements could focus on expanding the component variety in the dataset, refining similarity measures for higher accuracy, and enhancing the interpreter's ability to handle more intricate circuit configurations to broaden its applicability in professional engineering contexts.

5.2 Identified Project Outcome Limitations

Throughout the system performance testing phase of the current NL to DSL interpreter, several limitations related to the project outcomes were identified. These limitations highlight the practical boundaries observed during performance evaluation, which could impact the interpreter's efficiency, reliability, accuracy, and overall scalability. Understanding these limitations is essential for assessing the current system's constraints and guiding potential enhancements in future iterations. This subsection outlines the limitations of the NL to DSL interpreter, the limitations are categorized as follows:

Component Variety

The system can handle basic electrical components like resistors, capacitors, inductors, and standard power sources, which are well-defined in its component list. However, it lacks the ability to interpret complex or non-standard components like integrated circuits or specialized sensors. This limitation confines

the interpreter's usefulness to simple circuit designs, reducing its applicability in advanced engineering contexts where a broader set of components is required.

Schematic Complexity

The interpreter is limited to simple circuit configurations, such as series and parallel circuits. It does not support more intricate features like nested loops or interconnected sub-circuits found in complex electronic systems. As a result, it cannot generate schematics for advanced engineering projects, which hinders its usefulness for designing detailed and multifaceted circuits.

Dataset Dependency

The system's accuracy heavily depends on the quality and diversity of the training dataset, which is synthesized and accessed via Google Sheets. The limited scope of this dataset means the interpreter may struggle with circuit descriptions that deviate from familiar patterns. This dependency can lead to reduced accuracy and overfitting, making it challenging for the system to generalize to new and diverse input descriptions effectively.

Language Variability

The interpreter uses NLP techniques, including SpaCy, to handle common variations and misspellings, but it struggles with ambiguous or highly variable inputs. It relies on recognizing specific keywords and patterns, which means unconventional language or complex sentence structures can hinder accurate parsing. This limitation affects the interpreter's reliability, particularly when faced with creative or unclear user inputs.

Scalability and Adaptability

The interpreter's specialization in electrical circuit diagrams using a Domain-Specific Language (DSL) limits its scalability and adaptability to broader technical domains or more advanced designs. The issue is not merely the reliance on libraries, which is common across software, but the tight coupling of these libraries to specific parts of the system, as well as the use of hardcoded paths. This lack of modularity reduces the flexibility of the system, making it cumbersome to adapt to new environments or different datasets. Increasing dataset size or complexity can introduce performance bottlenecks, particularly due to the use of the Annoy library for approximate nearest neighbor searches. Adapting the system to new technical areas would require significant changes, reducing its ease of extension and long-term viability in evolving engineering fields.

5.3 Implications

The development of the NL to DSL interpreter pipeline for generating circuit diagrams from natural language descriptions carries significant practical and theoretical implications. This foundational study not only addresses current challenges in technical documentation and diagram generation but also opens avenues for future advancements across various domains. The following sections outline five key implications of this research, highlighting the pipeline's potential to enhance accessibility, efficiency, integration, collaboration, and scalability in technical fields.

Enhanced Accessibility for Technical Design

The pipeline allows non-experts to create technical circuit diagrams using natural language descriptions, significantly lowering the barrier for individuals without domain-specific knowledge. This accessibility

fosters inclusion by allowing educators, students, and technical professionals to generate accurate diagrams with minimal effort. This is particularly valuable for educators who need to create visual aids for students, simplifying technical content delivery.

Increased Efficiency in Documentation Processes

By automating the translation of natural language to structured schematic code, the pipeline streamlines technical documentation and reduces the time required to produce accurate diagrams. This efficiency helps technical writers, engineers, and educators save time, allowing them to focus more on analysis and design rather than the manual process of creating diagrams. Moreover, the automation reduces manual errors commonly associated with traditional methods, thereby improving overall documentation quality.

Integration with Technical Tools and Collaborative Platforms

The pipeline's ability to convert descriptions directly into machine-readable formats enables seamless integration with technical simulation tools like SPICE, MATLAB, or ANSYS, allowing for quick validation and testing of designs. Additionally, the pipeline integrates well with collaborative platforms such as Slack, Overleaf, and Blackboard, enabling multiple users to contribute to and modify diagrams in real time. This integration is valuable in remote and distributed learning environments, making it easier for educators and students to collaborate effectively on technical documentation.

Scalability Across Engineering Domains

The modular architecture of the pipeline allows expansion beyond electrical engineering, making it applicable in fields like mechanical engineering, software design, and hydraulic systems. For example:

- **Mechanical Engineering:** The pipeline can translate natural language descriptions of mechanical components into detailed mechanical diagrams, aiding in the design and analysis of machinery.
- **Hydraulic Systems:** It can generate schematics of fluid control mechanisms from textual descriptions, facilitating the design and maintenance of hydraulic systems.
- **Software Design:** The pipeline can assist in creating software architecture diagrams from natural language specifications, simplifying software documentation.

This versatility encourages interdisciplinary collaboration and enables technical teams to work across various domains using the same intuitive, natural language-based tool.

5.4 Recommendations for Future Research

The current research successfully demonstrates the potential of using NLP and DSLs to generate circuit diagrams from natural language descriptions. However, there are several areas that future work could explore to expand and enhance the capabilities of this system. The following recommendations outline key areas for potential improvement and exploration:

1. **Expansion of Circuit Complexity and Variety:** Future research could incorporate more complex and diverse circuit types, including multi-stage circuits, nested loops, and non-linear components such as transistors and integrated circuits. Extending the system's capabilities may increase its applicability to more advanced engineering contexts.
2. **Integration with Advanced NLP Models:** Leveraging state-of-the-art NLP models such as transformer-based architectures (e.g., GPT or BERT) could improve the semantic understanding and contextual interpretation of technical descriptions, enhancing the accuracy of circuit diagram generation.

3. **Adaptive Dataset Synthesis:** Future work could focus on adaptive dataset generation that evolves based on user input patterns and new circuit configurations. Machine learning techniques could be utilized to identify dataset gaps and dynamically generate relevant training examples.
4. **Enhanced Schematic Similarity Metrics:** Research could investigate the integration of more robust similarity metrics that account for the hierarchical structure and dependencies within circuit descriptions. Hierarchical similarity or graph-based measures may be beneficial in capturing intricate relationships between components.
5. **Real-Time Feedback and User Interaction Improvements:** Incorporating real-time feedback mechanisms could improve usability by offering sophisticated spell-checking, auto-correction, and context-aware prompts to guide users in refining their inputs.
6. **User Customizability and Scalability:** Developing modular functionalities that allow users to customize circuit components and configurations could enhance flexibility. Exploring scalable solutions for handling large-scale circuit designs and integrations could benefit industries requiring complex circuit documentation.
7. **Cross-Disciplinary Applications:** Adapting the NL to DSL framework for other technical fields, such as mechanical system schematics, process flow diagrams in chemical engineering, or network topology diagrams in IT, could expand the scope and utility of the system.
8. **Improved Integration with CAD Software:** Building deeper integration with industry-standard CAD tools such as AutoCAD or Altium Designer could enable seamless transitions between textual circuit descriptions and detailed, editable CAD files.
9. **Simulation Integration for Verification and Testing:** Future research could focus on integrating simulation capabilities within the interpreter to allow users to verify the functionality of generated circuit diagrams. This integration could include linking with existing simulation tools like SPICE or MATLAB Simulink for automatic testing and validation.
10. **Exploration of Multi-Language Support:** Implementing NLP models that understand diverse linguistic structures could support multiple languages, making the tool more globally accessible and valuable across different engineering contexts.
11. **Advanced Error Handling and Diagnostics:** Including comprehensive error-handling mechanisms that suggest corrections and provide detailed diagnostics for user input errors could make the tool more robust and user-friendly.

5.5 Conclusion

The research presented in this thesis aimed to bridge the gap between natural language descriptions and domain-specific language (DSL) outputs, specifically for generating circuit diagrams using Markdown editors integrated with Schemdraw. By developing a natural language to DSL interpreter, the research provides an innovative and accessible solution for users who lack extensive technical knowledge in circuit diagramming. This approach has demonstrated its potential in reducing the technical barriers associated with manually writing complex syntax for circuit representations.

The interpreter, combining natural language processing, dataset synthesis, and schematic generation, successfully translates human-readable descriptions into accurate circuit diagrams. Throughout the development of this system, significant emphasis was placed on ensuring accessibility, accuracy, and seamless integration within familiar environments like Markdown editors. The integration of various similarity measures including cosine similarity, schematic similarity, and Jaccard-based chunking proved effective in matching user inputs with existing dataset descriptions, ultimately generating Schemdraw diagrams

that capture the intended configurations and components.

Despite these achievements, the research also encountered certain limitations, such as the handling of complex circuit configurations beyond basic series and parallel components, and a reliance on synthetic datasets that may not cover all real-world use cases. However, the contributions of this work lay a robust foundation for future developments. Potential areas for further exploration include expanding component variety, incorporating advanced NLP models for more nuanced text interpretations, and enhancing the dataset with more complex circuit examples. These advancements would further enhance the ability of the interpreter to generate more intricate and context-specific diagrams.

Ultimately, this work contributes to the field of automated technical documentation, aiming to simplify the process of circuit diagram generation for technical and non-technical users alike. By making these tools more accessible, this research opens new possibilities for education, content creation, and engineering documentation, promoting efficiency and ease in generating precise visual representations of technical systems.

Bibliography

1. Akbas, Cem Emre, Alican Bozkurt, Musa Tunc Arslan, Huseyin Aslanoglu, and A. Enis Cetin (2014). "L1 norm based multiplication-free cosine similarity measures for big data analysis". In: *2014 International Workshop on Computational Intelligence for Multimedia Understanding (IWCIM)*, pp. 1–5. DOI: 10.1109/IWCIM.2014.7008798.
2. Alarcon, Nefi (May 2019). *Similarity in Graphs: Jaccard Versus the Overlap Coefficient*. NVIDIA Technical Blog. Accessed: 2024-10-26. URL: <https://developer.nvidia.com/blog/similarity-in-graphs-jaccard-versus-the-overlap-coefficient/>.
3. Attardi, Giuseppe and Felice Dell'Orletta (2008). "Chunking and dependency parsing". In: *LREC Workshop on Partial Parsing*, pp. 27–32.
4. Cambria, Erik and Bebo White (2014). "Jumping NLP Curves: A Review of Natural Language Processing Research [Review Article]". In: *IEEE Computational Intelligence Magazine* 9.2, pp. 48–57. DOI: 10.1109/MCI.2014.2307227.
5. Dahmen, Jessamyn and Diane Cook (2019). "SynSys: A Synthetic Data Generation System for Healthcare Applications". In: *Sensors* 19.5. ISSN: 1424-8220. DOI: 10.3390/s19051181. URL: <https://www.mdpi.com/1424-8220/19/5/1181>.
6. Dashti, Seyed Mohammad Sadegh, Amid Khatibi Bardsiri, and Mehdi Jafari Shahbazzadeh (2024). "Automatic real-word error correction in persian text". In: *Neural Computing and Applications*, pp. 1–25.
7. Dilmegani, Cem (Oct. 2024). *Synthetic Data vs Real Data: Benefits, Challenges*. Accessed: 2024-10-01. URL: <https://research.aimultiple.com/synthetic-data-vs-real-data/>.
8. Eppright, Caroline (2021). *What Is Natural Language Processing (NLP)?* Accessed: 2024-10-06. Oracle South Africa Cloud Artificial Intelligence. URL: <https://www.oracle.com/za/artificial-intelligence/what-is-natural-language-processing>.
9. Etaiwi, Wael and Ghazi Naymat (2017). "The impact of applying different preprocessing steps on review spam detection". In: *Procedia computer science* 113, pp. 273–279.
10. Fantechi, Alessandro, Stefania Gnesi, Samuele Livi, and Laura Semini (2021). "A spaCy-based tool for extracting variability from NL requirements". In: *Proceedings of the 25th ACM International Systems and Software Product Line Conference - Volume B. SPLC '21*. Leicester, United Kindom: Association for Computing Machinery, pp. 32–35. ISBN: 9781450384704. DOI: 10.1145/3461002.3473074.
11. Foote, Keith D. (July 2023). "A Brief History of Natural Language Processing". In: *DATAVERSITY*. Accessed: 2024-11-03. URL: <https://www.dataversity.net/a-brief-history-of-natural-language-processing-nlp/>.
12. Goyal, Mandeep and Qusay H. Mahmoud (2024). "A Systematic Review of Synthetic Data Generation Techniques Using Generative AI". In: *Electronics* 13.17. ISSN: 2079-9292. DOI: 10.3390/electronics13173509. URL: <https://www.mdpi.com/2079-9292/13/17/3509>.
13. Gulia, Sarita and Tanupriya Choudhury (2016). "An efficient automated design to generate UML diagram from Natural Language Specifications". In: *2016 6th International Conference - Cloud System and Big Data Engineering (Confluence)*, pp. 641–648. DOI: 10.1109/CONFLUENCE.2016.7508197.
14. Hudak, Paul (1997). "Domain-specific languages". In: *Handbook of programming languages* 3.39-60, p. 21. URL: <https://ncatlab.org/nlab/files/Hudak-DSLs.pdf>.
15. Ivchenko, GI and SA Honov (1998). "On the jaccard similarity test". In: *Journal of Mathematical Sciences* 88, pp. 789–794.
16. Joshi, SD and Dhanraj Deshpande (2012). "Textual requirement analysis for UML diagram extraction by using NLP". In: *International journal of computer applications* 50.8, pp. 42–46. URL: <https://citeseerx.ist.psu.edu/document?repid=rep1&type=pdf&doi=51f7e6c0992d80dcdf808d7427385015758dba9a>.
17. Katre, Mandeep, Jaya Gaur, Manas Srivastava, Ritik Gupta, and Sourdeep Ghosh Roy (2022). "Digital Content Editor with Markdown Support". In: *International Journal for Research in Applied Science Engineering Technology (IJRASET)* 10.III. SJ Impact Factor 7.538, ISRA Journal Impact Factor 7.894, pp. 1352–1358. ISSN: 2321-9653. DOI: 10.22214/ijraset.2022.40883. URL: <https://www.ijraset.com>.

18. Lally, Adam and Paul Fodor (2011). "Natural language processing with prolog in the ibm watson system". In: *The Association for Logic Programming (ALP) Newsletter* 9, p. 2011. URL: https://www.cs.miami.edu/home/odelia/teaching/csc419_spring19/syllabus/IBM_Watson_Prolog.pdf.
19. Muia, Charles M., Aaron M. Oirere, and Rachel N. Ndung'u (Apr. 2024). "A Comparative Study of Transformer-based Models for Text Summarization of News Articles". In: *International Journal of Advanced Trends in Computer Science and Engineering* 13.2, pp. 37–43. DOI: 10.30534/ijatcse/2024/011322024. URL: <http://www.warse.org/IJATCSE/static/pdf/file/ijatcse011322024.pdf>.
20. Niu, Hongwei, Cees Van Leeuwen, Jia Hao, Guoxin Wang, and Thomas Lachmann (2022). "Multimodal Natural Human–Computer Interfaces for Computer-Aided Design: A Review Paper". In: *Applied Sciences* 12.13. ISSN: 2076-3417. DOI: 10.3390/app12136510. URL: <https://www.mdpi.com/2076-3417/12/13/6510>.
21. Paulin, Goran and Marina Ivasic-Kos (2023). "Review and analysis of synthetic dataset generation methods and techniques for application in computer vision". In: *Artificial intelligence review* 56.9, pp. 9221–9265.
22. Polozov, Oleksandr and Sumit Gulwani (Oct. 2015). "FlashMeta: a framework for inductive program synthesis". In: *SIGPLAN Not.* 50.10, pp. 107–126. ISSN: 0362-1340. DOI: 10.1145/2858965.2814310.
23. Samad, Tariq (1986). *A natural language interface for computer-aided design*. Springer Science & Business Media.
24. Sarthi, Partho, Monojit Choudhury, Arun Iyer, Suresh Parthasarathy, Arjun Radhakrishna, and Sriram Rajamani (2021). *ProLinguist: Program Synthesis for Linguistics and NLP*. Microsoft Research India. URL: <https://nsnli.github.io/assets/ProLinguist.pdf>.
25. Shi, Pinyi, Yongwook Song, Zongming Fei, and James Griffioen (2021). "Checking Network Security Policy Violations via Natural Language Questions". In: *2021 International Conference on Computer Communications and Networks (ICCCN)*, pp. 1–9. DOI: 10.1109/ICCCN52240.2021.9522325.
26. Shliselberg, Michael, Ashkan Kazemi, Scott A. Hale, and Shiri Dori-Hacohen (2024). "SynDy: Synthetic Dynamic Dataset Generation Framework for Misinformation Tasks". In: *SIGIR '24*. Washington DC, USA: Association for Computing Machinery, pp. 2801–2805. ISBN: 9798400704314. DOI: 10.1145/3626772.3657667. URL: <https://doi.org/10.1145/3626772.3657667>.
27. Svistkov, Alexander I., Anton A. Sutchenkov, and Anton I. Tikhonov (2021). "STEM and STEAM Technologies in Problem Solving with Python". In: *3rd International Youth Conference on Radio Electronics, Electrical and Power Engineering (REEPE)*. IEEE, pp. 1–5. ISBN: 978-1-7281-8398-5. DOI: 10.1109/REEPE51337.2021.9388001.
28. Tschopp, Dominique and Suhas N. Diggavi (2009). "Approximate Nearest Neighbor Search through Comparisons". In: *CoRR* abs/0909.2194. arXiv: 0909.2194. URL: <http://arxiv.org/abs/0909.2194>.
29. UK Statistics Authority (Oct. 2022). *Ethical Considerations Relating to the Creation and Use of Synthetic Data*. Accessed: 2024-10-03. URL: <https://uksa.statisticsauthority.gov.uk/publication/ethical-considerations-relating-to-the-creation-and-use-of-synthetic-data/>.
30. Vijaymeena, MK and K Kavitha (2016). "A survey on similarity measures in text mining". In: *Machine Learning and Applications: An International Journal* 3.2, pp. 19–28.
31. Wang, Jiapeng and Yihong Dong (2020). "Measurement of Text Similarity: A Survey". In: *Information* 11.9. ISSN: 2078-2489. DOI: 10.3390/info11090421. URL: <https://www.mdpi.com/2078-2489/11/9/421>.
32. Yerygin, Dmitry (2024). "The Pros and Cons of Python Programming Language". In: *Redwerk Blog*. Accessed: October 26, 2024. URL: <https://redwerk.com/blog/pros-and-cons-of-python/>.