University of Kaiserslautern
Department of Computer Science

MASTER THESIS

Automatic Knowledge Base Construction With LLMs For Amplifiers Datasheets

---

Presented: 00.00.2024

Author: Vaibhav Rajendra Khandekar (422425)

First examiner: Prof. Dr. Christoph Grimm

Second examiner: Dr.Ing. Frank Wawrzik

Declaration

I hereby declare and confirm that this thesis is entirely the result of my own original work. Where other sources of information have been used, they have been indicated as such and properly acknowledged. I further declare that this or similar work has not been submitted for credit elsewhere.

Kaiserslautern, January 00, 2024

Vaibhav Rajendra Khandekar (422425)

Acknowledgement

I would like to express my deepest gratitude to my supervisor, Dr. Frank Wawrzik, for his invaluable guidance, support, and encouragement throughout my thesis. His weekly meetings, constructive feedback, and insightful suggestions have greatly contributed to the completion of this work. His insights and recommendations have been instrumental in improving the quality and accuracy of my research.

Furthermore, I would like to extend my sincere appreciation to Prof. Dr. Christoph Grimm for allowing me to work with his research group. It has been an incredible opportunity to be a part of such a visionary project, and I am grateful for the knowledge and experience gained from this collaboration.

I am also grateful to my family and friends for their unwavering support and encouragement throughout this journey. Their love and encouragement have been my source of strength, and I am forever indebted to them.

Kurzfassung

In dieser Studie wird die Anwendung von natürlicher Sprachverarbeitung und maschinellem Lernen für die Erstellung von Wissensdatenbanken untersucht, ein wichtiger Anwendungsfall für diese Technologien seit ihrer Entwicklung. Herkömmliche Methoden der Wissensextraktion haben oft mit Problemen wie erhöhtem Zeit-, Arbeits- und Kostenaufwand bei der Erstellung einer Wissensdatenbank zu kämpfen. Als Antwort darauf stellen wir in unserer Forschung einen neuartigen Ansatz zur Erstellung einer Wissensbasis vor, der speziell für den Bereich der Mikroelektronik entwickelt wurde und auf die Basisontologie GENIAL! abgestimmt ist. Mit Hilfe des Gemini Large Language Model (LLM) haben wir eine Wissensdatenbank über Verstärker und ihre elektrischen Eigenschaften erstellt. Für die Datenerfassung nutzten wir Datasheets.com, das strukturierte Informationen über verschiedene elektronische Komponenten bietet. Wir haben diese Daten extrahiert und vorverarbeitet, um ihre Relevanz und Genauigkeit für unsere Wissensdatenbank sicherzustellen. Darüber hinaus setzten wir Prompt-Engineering ein, um effektive Prompts für die Wissensextraktion zu entwerfen, die die genaue Generierung von Triplets aus den Daten ermöglichen. Die Sicherstellung der Qualität unserer Wissensdatenbank war ein zentraler Punkt, mit besonderem Augenmerk auf die Argumentation und die Implementierung von Beschränkungen, um ihre Integrität zu erhalten. Ein wichtiger Teil unserer Forschung war die Entwicklung der Ontologie, die wir mit Owlready2 automatisiert haben. Dieser Prozess umfasste die programmatische Erstellung von Klassen, Individuen, Dateneigenschaften und Objekteigenschaften. Darüber hinaus haben wir eine vergleichende Analyse von SPARQL-Abfragen, die auf unserer Wissensdatenbank ausgeführt wurden, mit solchen, die auf dem Gemini Large Language Model (LLM) ausgeführt wurden, durchgeführt und dabei die Stärken und Schwächen der einzelnen Methoden herausgestellt.

Abstract

This study investigates the application of natural language processing and machine learning for knowledge base creation, a vital use case for these technologies since their development. Traditional knowledge extraction methods have often struggled with issues like increased time, effort and costs involved in producing a knowledge base. In response, our research presents a novel approach to constructing a knowledge base, specifically designed for the microelectronics domain and aligned with the GENIAL! basic ontology. By harnessing Gemini large language model(LLM), we created a knowledge base centered on amplifiers and their electrical characteristics. To collect data, we utilized Datasheets.com, which offers structured information on various electronic components. We extracted and preprocessed this data to ensure its relevance and accuracy for our knowledge base. Additionally, we employed prompt engineering to design effective prompts for knowledge extraction, enabling the accurate generation of triplets from the data. Ensuring the quality of our knowledge base was a key focus, with particular attention to reasoning and the implementation of restrictions to maintain its integrity. A significant part of our research was the development of ontology, which we automated using Owlready2. This process included the programmatic creation of classes, individuals, data properties, and object properties. Moreover, we performed a comparative analysis of SPARQL queries run on our knowledge base against those executed on the Gemini large language model(LLM), highlighting the strengths and weaknesses of each method.

Contents

Chapter 1

Introduction

A knowledge base serves as a centralized repository that systematically organizes information within a specific domain. It encompasses a structured collection of facts, rules, definitions, procedures, and other forms of explicit knowledge. Unlike traditional databases, a knowledge base is designed to facilitate efficient retrieval and management of information, supporting tasks such as decision support, problem-solving, and information dissemination. knowledge bases are utilized across diverse fields including customer service, technical support, education, and research, providing a foundation for storing and accessing essential knowledge assets. They are instrumental in enhancing organizational efficiency, enabling quick access to relevant information, and promoting continuous learning and innovation within an organization or community.

Despite their advantages, knowledge base face challenges due to the complex, costly, and time-consuming process of constructing them. Creating a knowledge base typically involves the intricate tasks of extracting entities and identifying relevant relationships between them. This has led to increased research interest in automating knowledge base construction. Specifically, there is a growing focus on integrating large language models (LLMs) like Gemini, GPT-4 into this process because of their exceptional language processing capabilities.

In this thesis, we propose a novel approach that utilizes Google's Gemini(LLM)[1] and Python library Owlready2 for the automatic construction of a knowledge base. This process involves web scraping to generate data, employing LLM to extract entities and relationships, and subsequently populating and producing a enriched knowledge base using Owlready2[2].

At the same time, we investigate the potential of the GENIAL![3] Graph Generation Framework, an innovative tool aimed at simplifying knowledge base creation. We employ the predefined structures, relationships, reasoning, and inference rules provided by GENIAL!. The framework's efficiency and effectiveness are assessed alongside the use of LLMs for generating

knowledge base.

The objective of this research is to develop more flexible, scalable, and efficient methods for constructing knowledge base, thereby reducing the difficulty, cost, and time involved. Our aim is to enhance the application of artificial intelligence in everyday life by improving the generation and enrichment of knowledge base. By conducting a comparative analysis between the performance of SPARQL queries on knowledge base and the LLM-based queries, we seek to understand the strengths and limitations of our approach.

1.1 Motivation

Previous methods of constructing knowledge base rely on crowdsourcing or text mining. Crowdsourcing-based approaches, exemplified by knowledge base like WordNet[4] and ConceptNet[5], involve substantial human effort but are constrained by predefined sets of relations. In contrast, text mining methods extract knowledge directly from documents, focusing on relations explicitly stated within the text. This approach entails multiple stages such as knowledge acquisition, coreference resolution, entity linking, named entity recognition and more.

The challenges are exacerbated by the need to tailor knowledge base to specific fields or applications. Each domain uses distinct concepts and terminologies, making it challenging to develop a universal approach for constructing knowledge base. Domains such as service computing benefit greatly from knowledge base for tasks like resource management, personalized recommendations, and customer understanding. However, creating a knowledge base in this context involves integrating knowledge and concepts from diverse fields, often dealing with widely scattered and largely unannotated data. These factors significantly increase the time, effort, and costs required to build a knowledge base.

Our study seeks to share our findings and outcomes from building a knowledge base in the field of microelectronics using a large language model (LLM). This research highlights the distinct capabilities of LLMs and Python for extracting entities and relationships from diverse types of amplifier data tables and integrating them into a knowledge base. We suggest that this approach could enhance the accuracy and automation of generating factual knowledge and information.

1.2 Objectives

LLMs excel in language understanding due to their massive training datasets and ability to fill in missing information using prompts. However, they face challenges such as occasional misinformation, limited reasoning capabilities for complex relationships and numerical computation, and gaps in domain-specific knowledge. Conversely, knowledge base offer

structured and reliable knowledge grounded in factual data sources, with better reasoning capabilities and domain-specific adaptability. They are updated more easily but lack the flexibility and comprehensive understanding of language that LLMs possess. Each has distinct advantages and limitations, prompting ongoing research into their complementary roles in knowledge retrieval and application. knowledge base can enhance LLMs accuracy by providing structured guidance, while LLMs contribute to knowledge base construction by extracting and integrating knowledge from diverse data sources.

One prominent approach involves leveraging LLMs to automate the construction of knowledge bases, where these models analyze data to identify entities, relationships, and semantic connections. This automated process accelerates knowledge base development, making it more scalable and less reliant on manual input. LLMs excel in processing vast amounts of unstructured data, enabling them to uncover implicit relationships and enrich the knowledge base with nuanced insights from text and context. This synergy aims to improve the comprehensiveness and utility of knowledge bases across various domains and applications.

1.3 Overview and Structure

Figure 1.1: Knowledge Base Construction and Population Workflow

1. Data Collection

The execution of this investigation adheres to a five-step process, as depicted in Figure 1.1 of the report. The initial step involved extensive data collection, which includes extracting electrical charaterstics from amplifier datasheets.Extracting electrical characteristics information from amplifier datasheets is challenging due to varying formats, unstructured PDF data, and technical jargon. These datasheets often contain tabular data and graphical representations, making automated extraction complex. Additionally, specific conditions detailed within the datasheets can affect the characteristics, requiring nuanced understanding. That's why we utilized publicly available data of electrical characteristics for various types of amplifiers sourced from the website datasheets.com. This website provides data extracted in a structured format from amplifier PDF documents. The emphasis was specifically on information related to the electrical characteristics of amplifiers, as it is highly relevant to the study's topic.

2. Data Cleaning

A set of various amplifier types, including audio amplifiers, GPS amplifiers, operational amplifiers, and video amplifiers, is extracted from the data source along with their electrical characteristics. The extracted data is then cleaned by identifying and addressing quality issues such as missing values, duplicates, inconsistencies, and outliers. This process involves removing duplicate records and converting the data into a structured, common format. By following these steps, we ensure that the data is accurate, consistent, and ready for subsequent analysis or integration into systems like knowledge bases.

3. Prompt Engineering

Prompt engineering involves designing and refining prompts to effectively guide large language models (LLMs) in generating desired outputs. This process entails creating clear, precise, and contextually relevant instructions, questions, or statements to ensure accurate and meaningful responses from the model.

In our scenario, prompt engineering is used to extract triplets in the format of head, relation, and tail. These triplets define relationships between an amplifier, its electrical characteristics, associated numerical value, and electrical unit. Two distinct prompts are developed: an input prompt and an output prompt. The input prompt specifies the types of entities and relations that the LLM should use when generating the triplets. Meanwhile, the output prompt outlines the expected format or content of the triplets that the model should produce. In our case, the triplets are extracted in JSON format.

4. LLM(Gemini) for Triplet Extraction

Google AI Studio provides access to the Gemini API (LLM). An API key is generated for programmatically accessing the Gemini API. A function is implemented to extract triplets, requiring the API key, input prompt, output prompt, and amplifier data as inputs. Within the function, the Gemini chat model is utilized to extract triplets based on the instructions provided in the prompts. Function iterates through each response from the chat model, retrieves the top candidate text, appends it to results, and returns the combined results containing all anticipated triplets for an amplifier.

5. Populate OWL File with Extracted Triplets

The OWL file, which includes different types of amplifiers and their electrical characteristics as classes, object properties, and data properties, is generated using the Protege application. Protege offers an intuitive interface for creating, modifying, and visualizing OWL[6] files. After

extracting all triplets, they are populated as individuals within the classes defined in the OWL file. To accomplish this, we utilize the Owlready2[2] Python library to manipulate our OWL file directly using various Python-scripted functions. Owlready2[2] enhances efficiency particularly in situations necessitating automated ontology management, reasoning tasks, or integration with other Python-based workflows.

Chapter 2

Literature Review

In this chapter, we will examine the terms proposed by various researchers that relate to our study. Additionally, we will also review previous researchers' methods that are similar to our strategy.

2.1 Knowledge Graph

Knowledge graphs are structured semantic knowledge bases designed to quickly represent concepts and their relationships in the physical world. By consolidating information, data, and connections from the web, knowledge graphs enhance the computability, comprehensibility, and evaluability of information resources, enabling rapid responses and knowledge inferences. They facilitate the analysis and reasoning over vast amounts of data by capturing complex relationships and dependencies between entities within a specific domain or across general knowledge domains. Knowledge graphs allow us to convert text data into a format usable by machines and easily interpretable by humans. Essentially, a knowledge graph is a semantic network with additional constraints.[7].

Figure 2.1 represents the basic building blocks of knowledge graph, two nodes are connected by an edge, where nodes represent entities and edges represent relationships within the knowledge graph. This figure illustrates the fundamental building block of a knowledge graph. In practical applications, however, knowledge graphs can be vast and intricate, featuring numerous nodes interconnected by various relationships. These extensive networks enable the representation of complex data structures, supporting sophisticated data analysis and decision-making processes.

In the figure 2.2, This network management example employs the a graph database to model data center network endpoints, such as servers, routers, firewalls, racks, and other devices, as nodes. Their interconnections are depicted as relationships, allowing for the analysis of dependencies among network devices and the identification of root causes, thereby enhancing

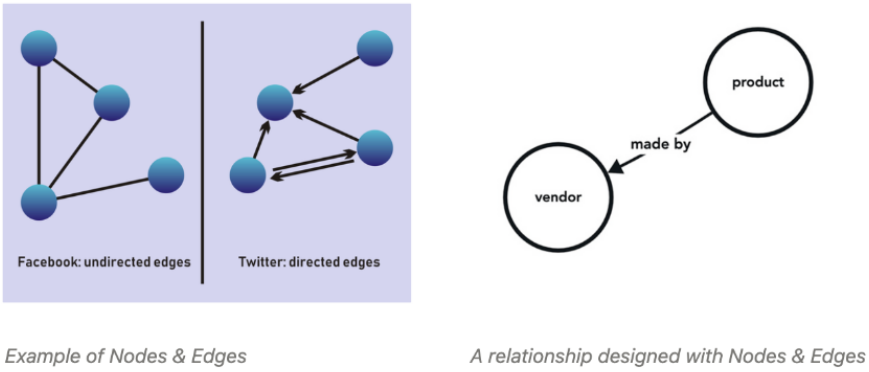


Figure 2.1: Components of knowledge Graph

network and IT management effectiveness.

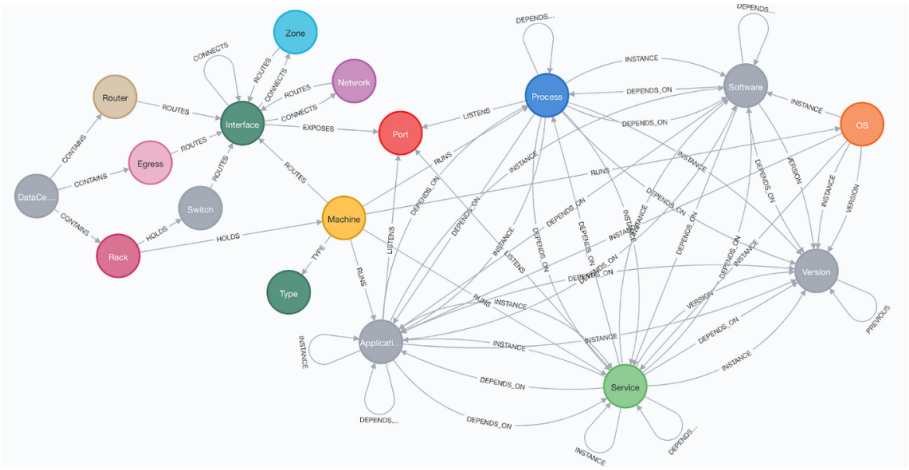


Figure 2.2: Example Graph of Network Management

This Extraction of information by sifting through hundreds of pages of data to identify relationships and entities manually is a daunting task. Machines are well-suited for this job because they can efficiently process vast amounts of text. However, machines inherently struggle to comprehend natural languages. The solution to this issue is natural language processing (NLP). NLP enables machines to understand and interpret text from various sources, facilitating the generation of knowledge graphs from the extracted information.

2.2 Ontology

Ontology involves the examination of existence, delving into diverse types of entities, their classification into distinct categories, and their interconnectedness at a fundamental level[8]. In the realm of information science and artificial intelligence, however, ontology takes on a slightly different meaning. It represents a structured framework for organizing and interpreting information, providing a shared and common understanding of a domain that can be communicated between people and computational systems. An ontology is a formal representation of knowledge that defines the concepts and relationships within a domain. It provides a structured framework for organizing information and capturing knowledge about entities and their interrelationships. Ontologies are typically expressed using a specific language or formalism that allows for clear and unambiguous definitions of terms and the constraints governing their usage.

In practical terms, an ontology specifies:

- Concepts: Representations of entities, ideas, or phenomena within a domain.
- Relationships: Connections and associations between concepts, describing how they are related.
- Properties: Attributes or characteristics associated with concepts.
- Constraints: Rules or axioms that govern the domain, ensuring consistency and correctness.

Ontologies are used extensively in fields such as artificial intelligence, information science, and knowledge management to facilitate data integration, knowledge sharing, and reasoning tasks. They serve as foundational structures for building knowledge graph, semantic networks, and other forms of structured knowledge representation systems.

We are employing the GENIAL! Basic Ontology, comprising 83 atomic classes and 14 object characteristics, encoded using the Ontology Web Language (OWL). Unlike conventional coding practices, ontologies prioritize the use of natural language terms to enhance vocabulary reusability. Key modeling components of ontologies include classes, individuals, data properties, and object properties, which share similarities with constructs found in traditional programming languages, thus bridging ontology design with programming paradigms.[9]

2.3 Natural Language Processing(NLP)

Natural Language Processing (NLP) is a branch of Artificial Intelligence and Linguistics that enables computers to understand human language, whether in text or speech form. The primary aim of NLP is to allow humans to interact with computers using everyday language instead of specialized computer languages. This is particularly beneficial for users who lack the time or

desire to learn new languages or achieve proficiency in them. NLP encompasses various tasks, such as part-of-speech tagging, named entity recognition, and optical character recognition, to help computers comprehend human language. There are numerous real-world applications of NLP, including information extraction, question-answering systems, machine translation, dialogue systems, information retrieval, text categorization, and spam filtering[10].

Natural Language Processing (NLP) plays a crucial role in the training and functioning of Large Language Models (LLMs). It involves the interaction between computers and human language, focusing on programming computers to process and analyze vast amounts of natural language data. The primary objective of NLP is to read, interpret, understand, and derive meaningful insights from human language. In LLMs, NLP is employed to comprehend, generate, and respond to human language in a contextually appropriate and grammatically accurate manner. LLMs are trained on extensive text datasets to learn the statistical structure of language, including grammar, common phrases, and semantic meanings. This understanding enables LLMs to produce human-like text that is contextually relevant and grammatically precise[11].

NLP is the foundation of LLMs, supplying the necessary language structure and knowledge for these models to generate human-like text. Advances in NLP methods and algorithms have greatly enhanced LLM performance, making them increasingly effective across a wide array of applications.

2.4 Large Language Models (LLMs)

A Large Language Model is an AI model that, in a true sense, understands, generates, and responds to human language. In other words, they are trained on giant volumes of text data so that, provided with input, they can generate coherent and relevant text. LLMs rely on an underlying Transformer architecture, which is deep learning–based and serves two significant purposes: the modeling of dependencies between elements in input-output spaces with long-range relations and effective learning from context.

The initial model that gained significant recognition was the GPT (Generative Pre-trained Transformer), created by OpenAI in 2018. The popular ChatGPT is essentially an evolution of GPT-3.5. What set the GPT model apart was its pioneering use of the transformer architecture, a neural network design adept at comprehending extensive textual relationships. This capability enabled the model to produce remarkably coherent and contextually appropriate language. With 117 million parameters, the GPT model marked a pivotal advancement in natural language processing[12].

Since then, we've witnessed the emergence of larger and more advanced language models such as Gemini, BERT, GPT-2 and GPT-3. These models are capable of generating text that

is more refined and human-like compared to the original GPT model. However, despite not being the largest or most advanced anymore, the GPT model remains a significant milestone in the evolution of language models and has profoundly influenced the field of natural language processing.

2.4.1 Types of Large Language Models

There are several different types of large language models, each with its own set of strengths and weaknesses.

- **Autoencoder-Based Model**:

  An example of a large language model is the autoencoder-based model, which operates by compressing input text into a simplified representation and using it to create new text. This approach excels in tasks such as text summarization or content generation[13].

- **Sequence-to-Sequence Model**:

  Another category of extensive language models includes sequence-to-sequence models. These models take an input sequence, such as a sentence, and produce an output sequence, such as a translation into another language. They are commonly employed in tasks like machine translation and text summarization[13].

- **Transformer-Based Models**:

  Transformer-based models represent another widely adopted category of large language models. These models leverage a neural network architecture specifically designed to excel in comprehending extensive textual relationships. This capability makes them highly effective for various language tasks such as text generation, language translation, and question answering[13].

- **Recursive Neural Network Models**:

  Recursive neural network models are crafted to manage structured data such as parse trees, which depict the syntactic arrangement of a sentence. They are valuable for applications such as sentiment analysis and natural language inference[13].

- **Hierarchical Models**:

  Hierarchical models are specifically engineered to process text across various levels of detail, encompassing sentences, paragraphs, and entire documents. They find application in tasks such as document classification and topic modeling[13].

However, the transformer architecture stands out as the most renowned Large Language Model (LLM) design. At the core of models like Gemini and GPT lies the transformer architecture, a groundbreaking neural network structure finely tuned for natural language processing applications. Unlike sequential processing in traditional recurrent neural networks (RNNs), the

Transformer utilizes parallel processing to simultaneously analyze the complete context of a sentence. This parallel approach greatly enhances efficiency and precision, empowering models to effortlessly manage intricate tasks.

2.4.2 Transformer Model

A typical Transformer model consists of four main steps in processing input data. Initially, the model conducts word embedding to transform words into multi-dimensional vector forms. Subsequently, the data progresses through numerous transformer layers where the self-attention mechanism is pivotal for comprehending connections among words within a sequence. Finally, following the processing through these Transformer layers, the model generates text by forecasting the most probable subsequent word or token in the sequence, relying on the acquired contextual understanding.

1. Word Embeddings

In constructing a large language model, word embedding serves as a vital initial phase. It entails converting words into vectors within a high-dimensional space, where words with similar meanings are clustered together. This facilitates the model in grasping word meanings and making predictions grounded in this comprehension[14].

Consider another example using the words "cat" and "dog." These two terms typically appear closer to each other in comparison to a pair like "cat" and "burgers." They share similarities as common pets known for their furry and friendly nature. In word embedding, these words are represented as vectors situated near each other in the vector space. This enables the model to recognize their shared meanings and contextual usage[14].

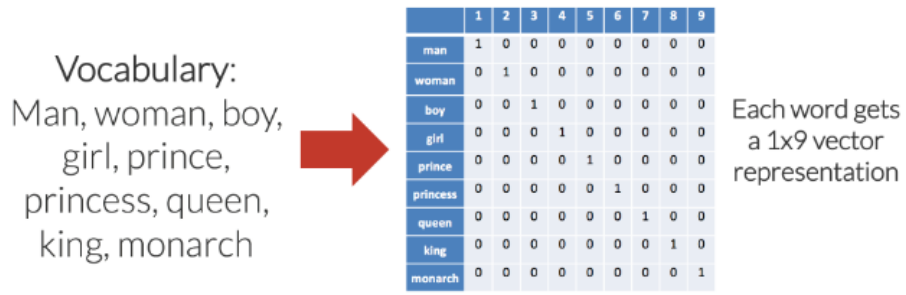


Figure 2.3: Example of a one-hot embedding scheme for a nine-word vocabulary[15]

To perform word embedding, a neural network is trained on a substantial corpus of text data, such as news articles or books. Throughout training, the network learns to predict the probability

of a word appearing within a specific context, based on the surrounding words in a sentence. The vectors derived from this process encapsulate semantic relationships between various words in the corpus. A similar methodology applies to words like "King," "Queen," "Man," and "Woman."

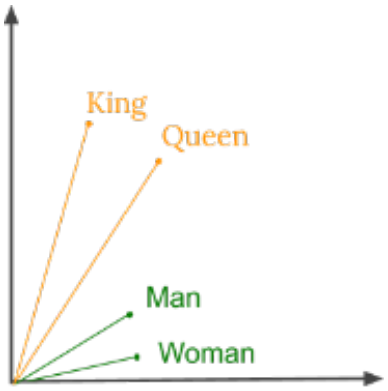


Figure 2.4: Example of Word Vectors[16]

After generating word embeddings, they serve as inputs to a broader neural network tailored for a particular language task, such as text classification or machine translation. Employing these word embeddings enhances the model's ability to grasp word meanings effectively, thereby improving the accuracy of predictions derived from this understanding[14].

2. Positional Encoding

Positional encoding focuses on helping the model understand the positions of words within a sequence, rather than their semantic meanings or relationships, such as the similarity between "cat" and "dog." Its primary role is to maintain the order of words. For instance, when translating a sentence like "The cat is on the mat" into another language, it's critical to preserve the sequence where "cat" precedes "mat." Word order holds significant importance in tasks such as translation, summarization, and question answering[14].

During training, the neural network is exposed to extensive text data and learns to make predictions based on this information. Through iterative adjustments using a backpropagation algorithm, the network's neuron weights are optimized to minimize the disparity between predicted and actual outputs.

3. Transformers

Advanced large language models employ a specific architecture called Transformers. In this framework, the transformer layer is introduced as an additional component following traditional

neural network layers. Its purpose is to enhance the model's capability to handle long-range dependencies in natural language text[14].

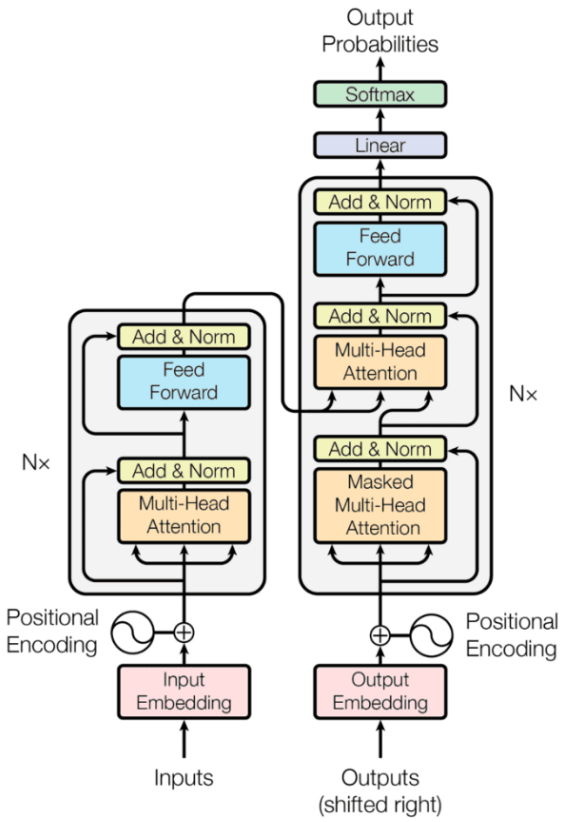


Figure 2.5: Architecture of Transformer [17]

The transformer layer differs by processing the entire input sequence simultaneously rather than sequentially. It consists of two key components: the self-attention mechanism and the feedforward neural network. The self-attention mechanism assigns weights to each word in the sequence based on its relevance for predictions, facilitating the model's understanding of word relationships regardless of their distance apart.

Following the self-attention layer's processing of the sequence, the position-wise feed-forward layer independently processes each position in the input sequence. At each position, a fully connected layer receives a vector representation of the token (word or subword) from the preceding self-attention layer. These fully connected layers transform these input vectors into new representations, optimizing the model's ability to discern intricate patterns and word relationships.

During training, the transformer layer undergoes iterative weight updates to minimize the gap between predicted and actual outputs. This training process employs the backpropagation algorithm, similar to conventional neural network layers.

4. Text Generation

After an LLM model has been trained and optimized, a common final step involves using it to produce sophisticated text in response to prompts or questions. Typically, the model is initialized with a seed input, which could be a few words, a sentence, or even a whole paragraph. Using its learned patterns, the LLM generates a coherent and contextually appropriate response.

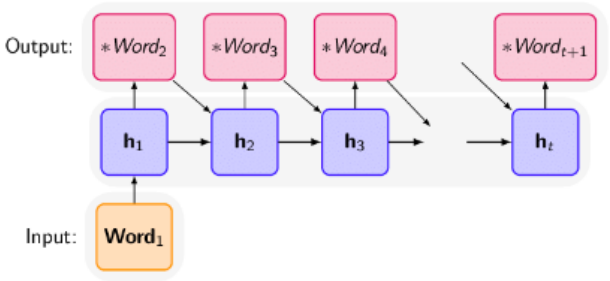


Figure 2.6: Architecture of Transformer[18]

Text generation employs autoregressive techniques, where the model generates each word or token of the output sequence sequentially based on the words it has previously generated. Utilizing its trained parameters, the model calculates the probability distribution for the next word or token and selects the most probable choice as the subsequent output.

2.4.3 LLMs Learning Methods

There are mainly three types of learning methods of LLMs, which are zero shot learning, one shot learning and few shot learning.

1. Zero shot learning

Zero shot learning is the learning method of LLM, where no demonstration is provided. Model generate the output based on the given description of the task. Efficiency of the zero shot learning is lower than the few shot and one shot learning[19]. As the zero shot learning produce the creative output based on the temperature. As the temperature increases creativity increases. Zero shot learning could be useful in the text generation, summarisation or text edition related tasks.

2. One shot learning

One-shot learning is a technique employed by large language models (LLMs) where they are given a single example or demonstration of the desired task. Utilizing this example, the LLM is capable of generating results. This method is akin to few-shot learning, with the primary distinction being the number of examples provided. One-shot learning is inspired by the way tasks are communicated to humans, as it can sometimes be challenging to convey the nature or format of a task without any examples. [19].

3. Few shot learning

In the few shot learning, learning could be performed by providing the various types of examples of required results. Few shot learning is an effective way because LLM can estimate the result which has to be produced by parsing the samples. In few shot learning, LLMs are given few demonstration of the task at interface time as conditioning, weight updates are not allowed. Few shot learning performance increases rapidly, demonstrating that larger models are more proficient at in-context learning[20].

2.4.4 Examples of LLMs

BERT

BERT, short for Bidirectional Encoder Representations from Transformers, is an advanced deep learning model created by Google for comprehending and generating natural language. Employing a bi-directional transformer architecture, BERT processes input text in both forward and backward directions, enhancing its ability to grasp contextual nuances and word relationships[21].

BERT finds application across a diverse array of tasks including question answering, sentiment analysis, named entity recognition, and text classification. It has set new benchmarks in various evaluations such as the Stanford Question Answering Dataset (SQuAD) and the GLUE (General Language Understanding Evaluation) benchmark.

In terms of scale, BERT base incorporates 110 million parameters, whereas the more sophisticated BERT large incorporates 345 million parameters[21].

GPT-4

OpenAI has introduced its latest advancement in the GPT series with much anticipation: GPT-4, or Generative Pre-trained Transformer 4. This groundbreaking large language model sets a new standard with a remarkable 100 trillion parameters, a significant increase from the 175 billion parameters of GPT-3.

Similar to its predecessor GPT-3, GPT-4's primary strength lies in its extensive pre-training on a vast corpus of text data. This allows it to grasp a wide range of language intricacies and relationships. Consequently, GPT-4 can be fine-tuned for specific natural language processing tasks using minimal examples, making it exceptionally efficient and adaptable for diverse applications.

To grasp the magnitude of GPT-4's capabilities, consider that it is 500 times more powerful than GPT-3, the model on which OpenAI based ChatGPT. This remarkable advancement in AI promises even more natural and accurate responses, transforming how we engage with and benefit from artificial intelligence[12].

Gemini

Gemini, created by Google DeepMind, marks a significant advancement in AI model capabilities. It is engineered to be multimodal, capable of comprehending and processing various types of data such as text, code, audio, images, and video. This versatility empowers Gemini to manage intricate tasks that integrate multiple forms of information effortlessly[1].

Gemini distinguishes itself with its multimodal capabilities, representing a notable advancement beyond the primarily text-focused nature of GPT-4. While GPT-4 excelled in comprehending and generating text, Gemini's architecture enables it to handle and interpret a mix of diverse data types such as text, code, audio, images, and video. This versatility equips Gemini to tackle tasks that demand a comprehensive understanding of different forms of information, making it suitable for complex and nuanced applications beyond traditional text processing[1].

Gemini has demonstrated impressive performance in direct comparisons. Gemini Ultra, in particular, has excelled in the MMLU benchmark, surpassing human experts, a level of achievement beyond GPT-4. Moreover, Gemini exhibits superior performance in tasks that involve multimodal inputs. This proficiency extends to coding tasks, where Gemini displays skill across multiple programming languages, outperforming GPT-4 in various coding benchmarks[1].

2.5 Prompt Engineering

Prompt engineering is an AI development technique used to fine-tune large language models (LLMs) by providing specific prompts and desired responses. It involves optimizing input for various generative AI platforms to produce desired text outputs. This practice, especially relevant for powerful models like GPT-3, focuses on crafting effective prompts to guide the model's behavior. The format of a prompt can include text, images, or other data types, although most generative AI tools primarily process natural language queries.

Prompting involves a systematic approach to providing instructions to the LLM, ensuring it

understands the text and generates appropriate results. A prompt is limited by the number of tokens it can include, usually around 2000 tokens, which means all instructions, examples, and output formats must fit within this limit. This constraint can sometimes lead to inaccurate results from the LLM.

Improving the accuracy of outputs from LLMs can be achieved through several methods. One simple and effective technique in prompt engineering is to add the phrase "Think step by step" at the end of a prompt. This strategy, validated by researchers from Google and the University of Tokyo, significantly enhances output accuracy. For example, when this phrase was included in prompts for the text-davinci-002 model, accuracy increased from 17% to 78.7%[22].

2.5.1 Prompting Techniques

1. Tree of Thought

Tree of Thought Prompting is a method for exploring multiple alternatives and ideas by providing several lines of thought, akin to a decision tree. Unlike traditional linear approaches, this technique allows the AI model to simultaneously evaluate and follow numerous pathways. Each idea branches out, creating a tree-like structure of interconnected concepts. The AI model then assesses each path, assigns scalar values based on anticipated outcomes, and eliminates less promising options, ultimately selecting the most promising possibilities.[23].

2. Chain of Thought

Large Language Models (LLMs) employ 'Chain of Thought' prompting to break down complex tasks into several intermediate steps. This technique resembles human problem-solving methods, where intricate issues are divided into smaller, more manageable parts. 'Chain of Thought' prompts guide the model through a complex problem step-by-step. For instance, to enhance an LLM's ability to solve arithmetic word problems, users might provide a detailed, step-by-step example solution. The LLM then applies this step-by-step reasoning to new problems. Generally, 'Chain of Thought' prompting is highly effective for solving complex issues but offers little to no benefit for basic problems. Research shows that outlining each step of the reasoning process on a new line, rather than separating stages with periods, significantly improves results.[24].

3. Few-shot Prompting

Few-shot learning involves providing Large Language Models (LLMs) with specific examples that illustrate the desired output. By presenting these representative samples, LLMs are better equipped to produce the intended results. This approach is effective because a single prompt can generate multiple valid responses, a situation referred to as being under-determined. Providing

clear examples helps narrow the range of possible outputs. However, it's important to ensure these examples are both varied and balanced to avoid bias. For instance, in a sentiment analysis task using GPT, if seven out of eight examples are positive, GPT might be biased towards classifying texts as positive. Additionally, the examples must be relevant to the specific scenarios being addressed. If GPT is trained only on "positive" or "negative" examples, it might have difficulty identifying neutral statements, instead categorizing them as either "positive" or "negative."[19]

4. Role Prompting

It is observed that, improved outcomes can be achieved LLMs are explicitly instructed to assume the role of specialists in a relevant field. For example, when requesting code, the prompt might begin with "You are an expert in coding." This technique is believed to help the LLM focus and identify the most relevant parts of its extensive knowledge base for the given task. Known as 'role prompting,' this straightforward method directs LLMs to generate content in a specific creative style or manner, similar to that of a well-known author [25].

2.5.2 Example Prompts

1. Text Summarization Prompt

Text summarization is a common task in natural language generation, encompassing various styles and domains. One of the most promising applications of language models is their ability to condense articles and concepts into brief, easy-to-read summaries.

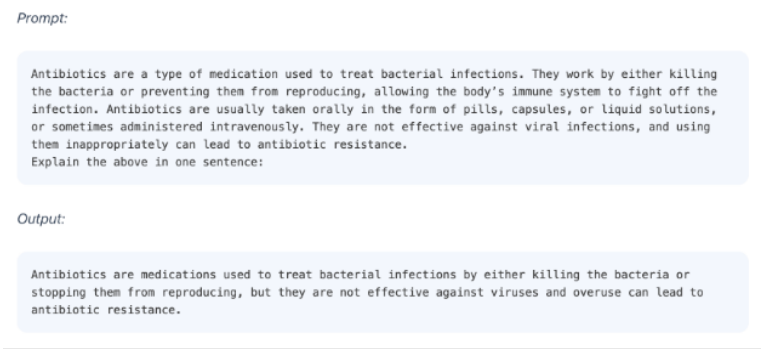


Figure 2.7: Example of Text Summarization Prompt

2. Information Extraction Prompt

Although language models are designed for natural language generation and related tasks, they are also highly capable of performing classification and a variety of other natural language processing (NLP) tasks.

Below is an example of a prompt used to extract information from a given paragraph.

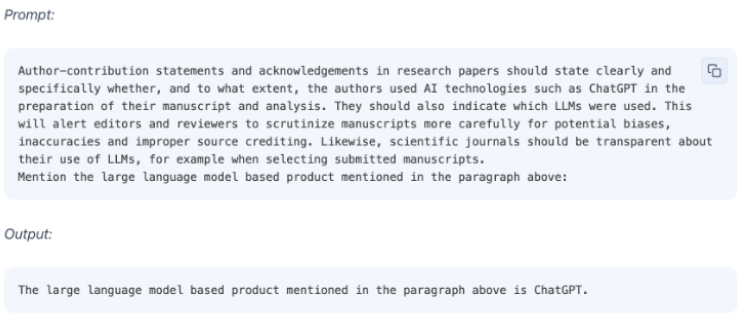


Figure 2.8: Example of Information Extraction Prompt

3. Question Answering Prompt

One effective way to elicit specific answers from the model is to enhance the prompt's format. As previously mentioned, a prompt can include instructions, context, input, and output indicators to achieve better results. Although these components are not mandatory, using them is a good practice. The more precise you are with instructions, the better the outcomes will be. Below is an example of a more structured prompt.

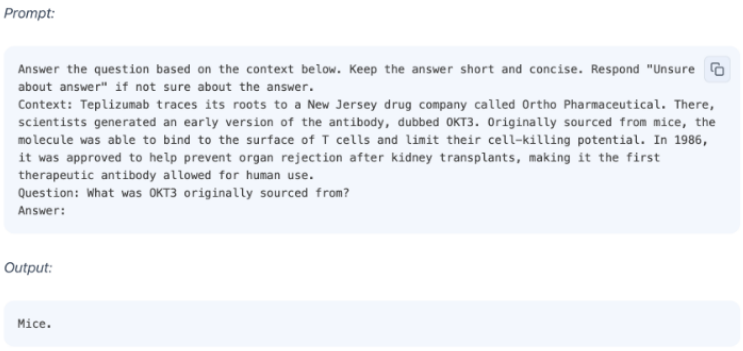


Figure 2.9: Example of Question Answering Prompt

4. Code Generation Prompt

One area where LLMs excel is in code generation, exemplified by tools like Copilot. Clever prompts enable a wide range of code-generation tasks. Here is a example:

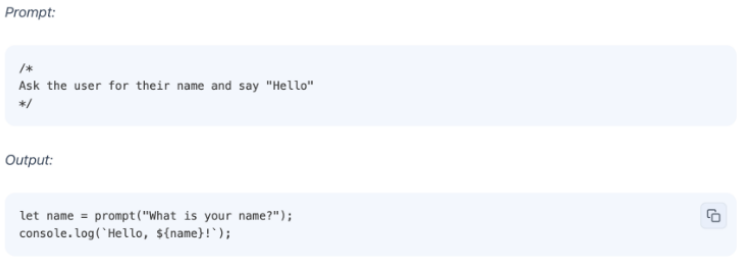


Figure 2.10: Example of Code Generation Prompt

Conversation Prompt

One of the more intriguing capabilities of prompt engineering is instructing the LLM system on its behavior, intent, and identity. This is especially useful when developing conversational systems such as customer service chatbots.

For example, let's design a conversational system capable of generating technical and scientific responses to questions. Notice how the instruction explicitly guides the system's behavior. This technique is often referred to as role prompting.

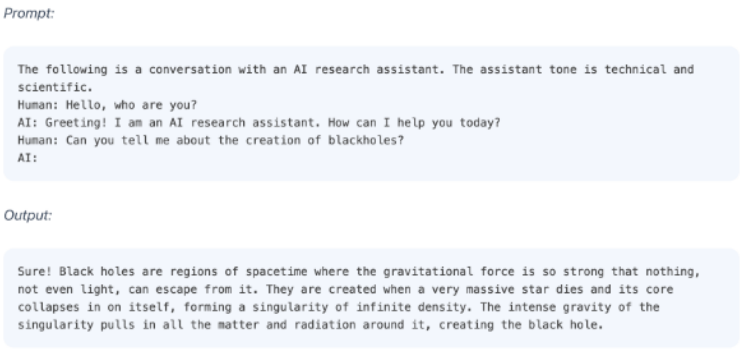


Figure 2.11: Example of Conversation Prompt

Chapter 3

Methodology

In this chapter, we will detail our experiments and findings. We will describe our approach to dataset generation using openly available data from Datasheets.com. Additionally, we will discuss the software and tools used, and and take a deep dive into Google Gemini. Our goal in this section is to build a comprehensive and relevant dataset that forms the backbone of our research, ensuring that the generated knowledge base is both accurate and insightful.

3.1 Overview

Datasheets.com is a web-based repository offering access to datasheets for various electronic components like amplifiers, resistors, capacitors, and transistors. These datasheets, issued by manufacturers, contain detailed information about the specifications, features, and applications of these electronic parts. Such documents are essential for engineers, designers, and technicians requiring precise and thorough information for designing and troubleshooting electronic systems.

Typically, these datasheets are provided in PDF format, which can make it challenging to extract information since each one is structured differently. datasheets.com compiles these datasheets, providing an easily searchable and accessible platform for users seeking specific component information.

3.2 Data Collection and Processing

3.2.1 Source of Amplifier Data Sheets

To facilitate the extraction of knowledge from datasheets, we first collected data on various amplifiers from Datasheets.com. These datasheets were downloaded in excel format and finally

converted into csv files for LLMs to process them. However, these files contained a significant amount of jargon and extraneous information that was not relevant to our study.

To streamline the data, we imported these Excel files into Python as Pandas DataFrames. This allowed us to efficiently manipulate and clean the data using various functions available in Pandas. The initial DataFrames were cluttered with numerous columns and unstructured text, which necessitated a thorough cleaning process.

3.2.2 Data Processing Techniques

Initially, we began our data cleaning process by eliminating extraneous columns that did not align with our research goals. These included columns containing manufacturer-specific details, redundant descriptions, and any metadata that did not directly pertain to the electronic characteristics of the amplifiers.

Subsequently, our focus shifted to refining columns that housed the electronic attributes of the amplifiers. These columns often presented mixed information, such as numerical values intermingled with measurement units. To streamline this, we divided these columns into two separate categories: one for numerical values and another specifically for the corresponding units of measurement. For instance, a column initially labeled as "Gain (dB)" was split into "Gain" for numerical values and "Gain Unit" for the unit of measurement, which in this case was "dB."

Furthermore, electronic property values were frequently provided as a range encompassing minimum, maximum, and average values. To standardize this format, we introduced dedicated columns for each type of value. For example, for a property like "Input Voltage," we created distinct columns titled "Input Voltage Min," "Input Voltage Max," and "Input Voltage Avg." This systematic approach ensured that the data became well-organized and readily interpretable for subsequent analysis tasks.

Once the data had been cleaned, we merged the refined information into a unified DataFrame for each amplifier. Each DataFrame contains a subset of unique amplifiers of a specific amplifier type. This consolidated DataFrame was streamlined, free from unnecessary complexity, and organized for straightforward access and manipulation. To maintain compatibility with our LLM, we saved each cleaned DataFrame in CSV format.

These CSV files were subsequently employed as input for a language model (LLM) to extract triplets, a widely used format for representing knowledge. The LLM can effectively parse the cleaned and standardized CSV files, ensuring precise and efficient extraction of significant information.

The meticulous process of data collection, cleaning, and formatting was essential to ensure the integrity and usability of the data for knowledge graph extraction. By transforming the raw

datasheets into structured CSV files, we laid a solid foundation for generating accurate and insightful knowledge, ultimately advancing our research objectives.

3.3 Google Gemini

Google Gemini, a formidable large language model (LLM) created by Google AI, has made a significant impact globally. Its remarkable proficiency in comprehending and generating language, coupled with its adaptability in addressing diverse tasks, has captured widespread interest and ignited curiosity regarding its fundamental design.

Gemini's foundation is built upon the Transformer architecture[17], an innovative neural network design tailored specifically for tasks in natural language processing. Unlike conventional recurrent neural networks (RNNs), which process information sequentially, the Transformer adopts a parallel approach, enabling it to simultaneously analyze the entire context of a sentence. This parallel processing significantly enhances efficiency and precision, empowering Gemini to adeptly handle complex tasks[1].

The architecture also incorporates a Mixture-of-Experts (MoE) approach, dividing the model into smaller, specialized neural networks (similar to the speculated GPT-4 architecture). This approach enables the model to activate the most pertinent pathways for each input selectively, enhancing both training efficiency and operational performance[1].

It is engineered to be multimodal, capable of processing and comprehending various data types such as text, code, audio, images, and video. This versatility empowers Gemini to seamlessly tackle intricate tasks that integrate diverse forms of information[1].

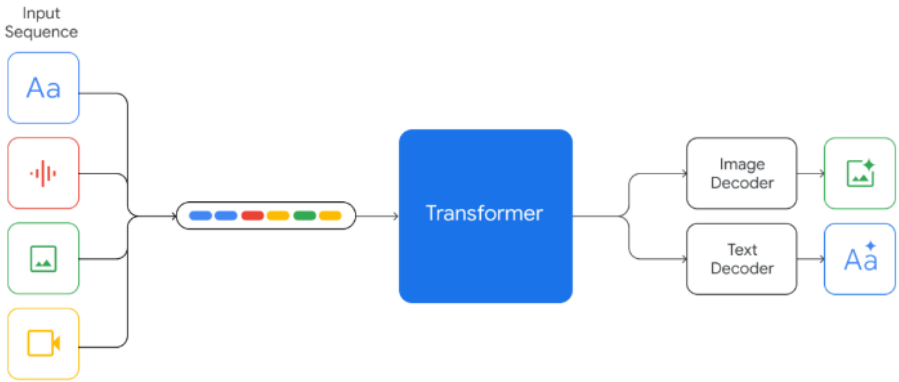


Figure 3.1: Multi-Modal Nature of Gemini[1]

The model is available in three configurations:

Gemini Ultra: Customized for handling exceptionally complex tasks.

Gemini Pro: Ideal for a wide spectrum of tasks across different domains.

Gemini Nano: Optimized for on-device applications, particularly suited for mobile devices.

3.3.1 The Transformer: A Powerful Engine for Language Processing

The Transformer[17] architecture comprises two primary components:

Encoders: These analyze input sequences (e.g., sentences) to extract their meaning and structure using "self-attention," which prioritizes relevant parts of the sequence while filtering out irrelevant details[17].

Decoders: These take encoded representations from the encoders to generate output sequences (e.g., translations, answers to questions, or creatively formatted texts). Decoders utilize "attention mechanisms" to focus on pertinent parts of the encoded input when producing the output[17].

3.3.2 The Role of Attention

Central to the Transformer's capabilities is its attention mechanism, which enables the model to concentrate on crucial elements of the input sequence during output generation, resulting in more precise and coherent outcomes. Analogous to reading a book where one focuses on pivotal details for understanding the narrative, the Transformer uses attention to highlight critical aspects of the input, thereby generating meaningful and contextually relevant outputs[17].

3.3.3 Gemini's Unique Optimizations

While the fundamental Transformer architecture remains unchanged, Google has implemented several optimizations to augment Gemini's performance and capabilities. These enhancements encompass:

Scalability to Large Datasets: Gemini's architecture is adept at processing vast volumes of textual data, enabling it to assimilate a broader spectrum of tasks and contexts.

Enhanced Efficiency: Google employs custom hardware known as Tensor Processing Units (TPUs) tailored specifically for AI operations to power Gemini. This integration results in expedited processing and training times.

Multimodal Capabilities: Gemini's functionality extends beyond text to encompass other modalities such as images and audio. This expansion equips Gemini to tackle more intricate real-world tasks effectively.

	Gemini Ultra	Gemini Pro	GPT-4	GPT-3.5	PaLM 2-L	Claude 2	Inflect-ion-2	Grok 1	LLAMA-2
MMLU Multiple-choice questions in 57 subjects (professional & academic) (Hendrycks et al., 2021a)	**90.04%** CoT@32* 83.7% 5-shot	79.13% CoT@8* 71.8% 5-shot	87.29% CoT@32 (via API**) 86.4% 5-shot (reported)	70% 5-shot	78.4% 5-shot	78.5% 5-shot CoT	79.6% 5-shot	73.0% 5-shot	68.0%***
GSM8K Grade-school math (Cobbe et al., 2021)	**94.4%** Maj1@32	86.5% Maj1@32	92.0% SFT & 5-shot CoT	57.1% 5-shot	80.0% 5-shot	88.0% 0-shot	81.4% 8-shot	62.9% 8-shot	56.8% 5-shot
MATH Math problems across 5 difficulty levels & 7 subdisciplines (Hendrycks et al., 2021b)	**53.2%** 4-shot	32.6% 4-shot	52.9% 4-shot (via API**) 50.3% (Zheng et al., 2023)	34.1% 4-shot (via API**)	34.4% 4-shot	—	34.8%	23.9% 4-shot	13.5% 4-shot
BIG-Bench-Hard Subset of hard BIG-bench tasks written as CoT problems (Srivastava et al., 2022)	**83.6%** 3-shot	75.0% 3-shot	83.1% 3-shot (via API**)	66.6% 3-shot (via API**)	77.7% 3-shot	—	—	—	51.2% 3-shot
HumanEval Python coding tasks (Chen et al., 2021)	**74.4%** 0-shot (PT****)	67.7% 0-shot (PT****)	67.0% 0-shot (reported)	48.1% 0-shot	—	70.0% 0-shot	44.5% 0-shot	63.2% 0-shot	29.9% 0-shot
Natural2Code Python code generation. (New held-out set with no leakage on web)	**74.9%** 0-shot	69.6% 0-shot	73.9% 0-shot (via API**)	62.3% 0-shot (via API**)	—	—	—	—	—
DROP Reading comprehension & arithmetic. (metric: F1-score) (Dua et al., 2019)	**82.4** Variable shots	74.1 Variable shots	80.9 3-shot (reported)	64.1 3-shot	82.0 Variable shots	—	—	—	—
HellaSwag (validation set) Common-sense multiple choice questions (Zellers et al., 2019)	87.8% 10-shot	84.7% 10-shot	**95.3%** 10-shot (reported)	85.5% 10-shot	86.8% 10-shot	—	89.0% 10-shot	—	80.0%***
WMT23 Machine translation (metric: BLEURT) (Tom et al., 2023)	**74.4** 1-shot (PT****)	71.7 1-shot	73.8 1-shot (via API**)	—	72.7 1-shot	—	—	—	—

Figure 3.2: Gemini's Performance on Text Benchmarks[1]

3.3.4 Gemini's Perfomance

Gemini distinguishes itself through its multimodal capabilities, marking a significant departure from the predominantly text-oriented nature of GPT-4. While GPT-4 demonstrated proficiency in generating and understanding text, Gemini's architecture enables it to process and interpret a diverse array of data types, including text, code, audio, images, and video. This flexibility empowers Gemini to undertake tasks that necessitate a comprehensive grasp of multiple forms of information, positioning it effectively for complex and nuanced applications beyond mere text processing[1].

In direct comparisons, Gemini has delivered impressive outcomes. Notably, Gemini Ultra has excelled in the MMLU[26] benchmark, surpassing human experts—a milestone not achieved by GPT-4. Furthermore, in tasks involving multimodal inputs, Gemini has demonstrated superior performance. This heightened capability extends to its proficiency in coding tasks, where Gemini exhibits competence across multiple programming languages, outperforming GPT-4

across various coding benchmarks[1].

Gemini Ultra exhibits robust performance across mathematics, coding tasks, and reading comprehension, achieving notable results in benchmarks such as GSM8K, HumanEval, and HellaSwag. While the paper acknowledges minor challenges related to data contamination, it underscores the critical need for comprehensive and refined evaluation metrics[1].

3.4 Software, Tools and Libraries Used

Throughout this project, a wide array of software tools, libraries, and APIs played crucial roles in different phases of development, data processing, and analysis:

Protege: Protege was utilized for ontology modeling and development, providing an intuitive interface to create and edit ontologies using the Web Ontology Language (OWL), which is essential for structuring domain-specific knowledge[27].

Jupyter Notebook: Jupyter Notebook served as an interactive environment for Python scripting, facilitating the integration of code, visualizations, and explanations. It supported exploratory data analysis and experimentation throughout the project[28].

Python: Python was the primary programming language chosen for its versatility and extensive library ecosystem. It enabled data manipulation, algorithm implementation, and seamless interaction with APIs to support various project tasks.

Pandas: Pandas, a robust Python library for data manipulation and analysis, played a pivotal role in cleaning, transforming, and organizing datasets to prepare them for ontology creation and analysis[29].

Owlready2: Owlready2, another Python library dedicated to ontology-oriented programming, provided essential tools for ontology loading, reasoning, and manipulation directly within Python scripts. This enhanced the project's capabilities in ontology development[2].

Gemini API: The Gemini API was integrated to harness advanced language model capabilities. This API facilitated programmatically interacting with Google's Gemini model, enabling tasks such as natural language understanding, text generation, and managing complex tasks across different modalities within the project[1].

Together, these software tools, libraries, and APIs enabled streamlined ontology development, streamlined data preprocessing, and sophisticated analytics, ensuring thorough exploration and effective application of domain knowledge to achieve the study goals.

Chapter 4

Implementation

In this chapter, we will thoroughly explain our experiments and findings related to creating a knowledge base of amplifiers. We will begin by performing triplet extraction using prompt engineering with the large language model Gemini. Next, we will create an ontology file to represent the overall knowledge base. Finally, we will use Owlready2 to automate the process of populating the ontology file. Additionally, we will discuss the challenges encountered and share insights from our experience. The process of implementing our proposed approach will be detailed step by step.

4.1 Data Extraction

Datasheets.com is an online repository that provides access to datasheets for a wide range of electronic components, including amplifiers, resistors, capacitors, and transistors. These datasheets, published by manufacturers, offer detailed information about the specifications, features, and applications of these electronic parts. Engineers, designers, and technicians rely on these documents for accurate and comprehensive information needed for designing and troubleshooting electronic systems.

Typically provided in PDF format, these datasheets can be difficult to extract information from due to their varying structures. Datasheets.com compiles these documents into a user-friendly platform, making it easy for users to search for and access specific component information.

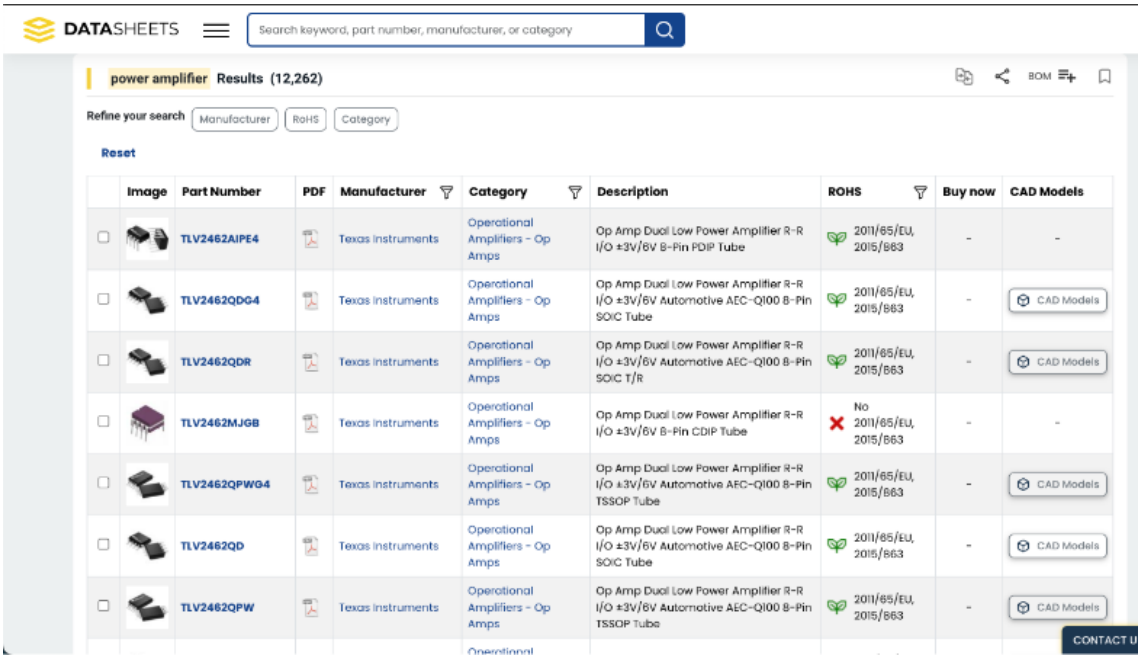


Figure 4.1: Datasheets.com

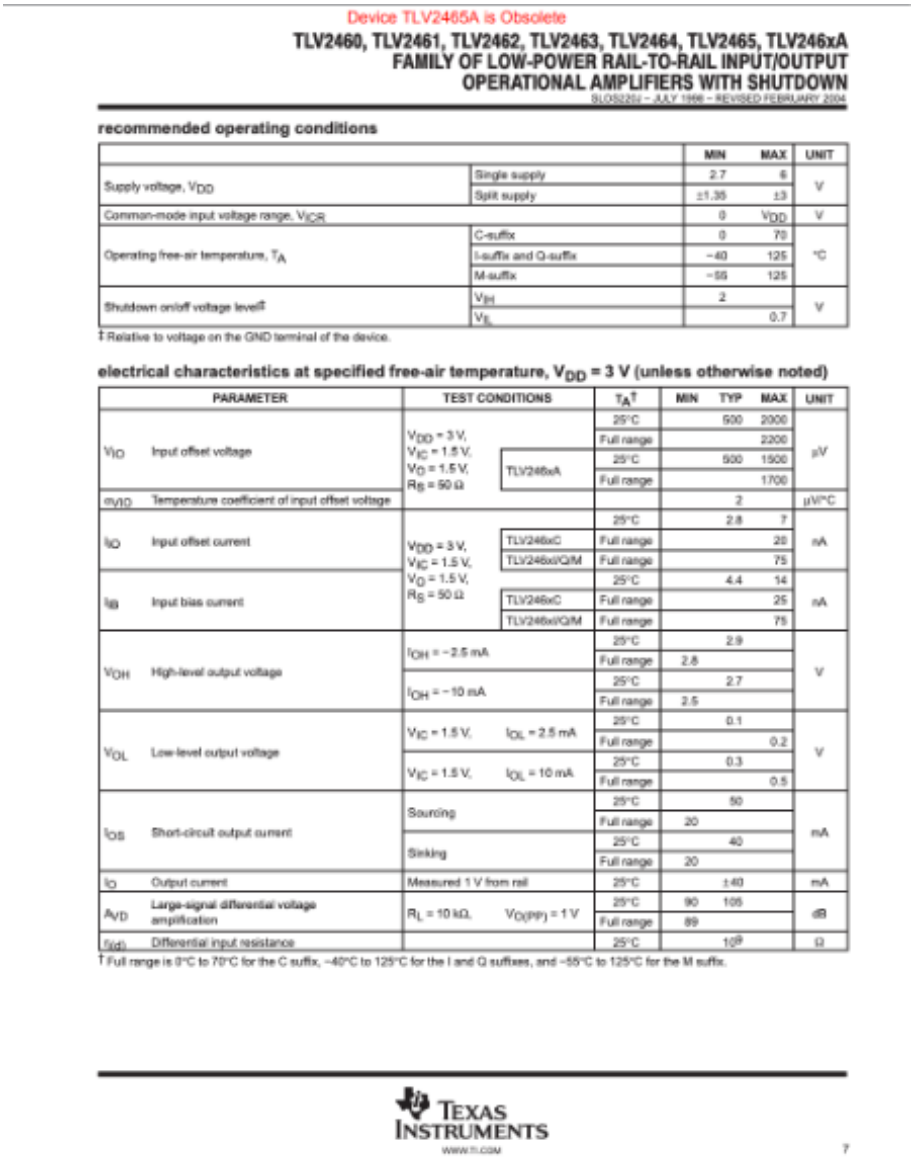
We began by collecting data on various amplifiers from Datasheets.com to facilitate knowledge extraction. These datasheets were initially downloaded in Excel format and subsequently converted into CSV files for processing by large language models (LLMs). However, the files contained a significant amount of jargon and extraneous information that was irrelevant to our study.

To streamline the data, we imported these CSV files into Python as Pandas DataFrames. Utilizing the various functions available in Pandas, we efficiently manipulated and cleaned the data. The initial DataFrames were cluttered with numerous columns and unstructured text, necessitating a thorough cleaning process.

4.1.1 Challenges and Solutions

Extracting data from PDF files is challenging due to the diverse and often inconsistent formatting of these documents. While PDFs are designed to display content uniformly across platforms, they usually lack a structured format that is easy to parse programmatically. Elements like tables, images, and multi-column layouts can complicate extraction, as conventional text extraction tools may misinterpret or miss critical data. Moreover, embedded fonts and graphical elements can cause character encoding issues, making accurate text capture difficult. The variation in design and structure across PDFs from different sources means a universal extraction method is rarely

effective, requiring tailored approaches for each document type. Consequently, extracting useful information from PDFs can be time-consuming and demands sophisticated tools and techniques to ensure both accuracy and completeness.



Device TLV2465A is Obsolete

TLV2460, TLV2461, TLV2462, TLV2463, TLV2464, TLV2465, TLV246xA FAMILY OF LOW-POWER RAIL-TO-RAIL INPUT/OUTPUT OPERATIONAL AMPLIFIERS WITH SHUTDOWN

SLOS222J – JULY 1998 – REVISED FEBRUARY 2004

recommended operating conditions

		MIN	MAX	UNIT
Supply voltage, V_{DD}	Single supply	2.7	6	V
	Split supply	±1.35	±3	
Common-mode input voltage range, V_{ICR}		0	V_{DD}	V
Operating free-air temperature, T_A	C-suffix	0	70	°C
	I-suffix and Q-suffix	−40	125	
	M-suffix	−55	125	
Shutdown on/off voltage level‡	V_{IH}	2		V
	V_{IL}		0.7	

‡ Relative to voltage on the GND terminal of the device.

electrical characteristics at specified free-air temperature, V_{DD} = 3 V (unless otherwise noted)

PARAMETER		TEST CONDITIONS		T_A†	MIN	TYP	MAX	UNIT
V_{IO}	Input offset voltage	V_{DD} = 3 V, V_{IC} = 1.5 V, V_O = 1.5 V, R_S = 50 Ω		25°C		500	2000	µV
				Full range			2200	
			TLV246xA	25°C		500	1500	
				Full range			1700	
α_{VIO}	Temperature coefficient of input offset voltage					2		µV/°C
I_{IO}	Input offset current	V_{DD} = 3 V, V_{IC} = 1.5 V, V_O = 1.5 V, R_S = 50 Ω		25°C		2.8	7	nA
			TLV246xC	Full range			20	
			TLV246xI/Q/M	Full range			75	
I_{IB}	Input bias current			25°C		4.4	14	nA
			TLV246xC	Full range			25	
			TLV246xI/Q/M	Full range			75	
V_{OH}	High-level output voltage	I_{OH} = −2.5 mA		25°C		2.9		V
				Full range	2.8			
		I_{OH} = −10 mA		25°C		2.7		
				Full range	2.5			
V_{OL}	Low-level output voltage	V_{IC} = 1.5 V,	I_{OL} = 2.5 mA	25°C		0.1		V
				Full range			0.2	
		V_{IC} = 1.5 V,	I_{OL} = 10 mA	25°C		0.3		
				Full range			0.5	
I_{OS}	Short-circuit output current	Sourcing		25°C		50		mA
				Full range	20			
		Sinking		25°C		40		
				Full range	20			
I_O	Output current	Measured 1 V from rail		25°C		±40		mA
A_{VD}	Large-signal differential voltage amplification	R_L = 10 kΩ,	$V_{O(PP)}$ = 1 V	25°C	90	105		dB
				Full range	89			
r_{id}	Differential input resistance			25°C		10^{12}		Ω

† Full range is 0°C to 70°C for the C suffix, −40°C to 125°C for the I and Q suffixes, and −55°C to 125°C for the M suffix.

TEXAS INSTRUMENTS
www.ti.com

7

Figure 4.2: Example PDF Data Sheet

Typically provided in PDF format shown in above figure, datasheets can be challenging to extract information from due to their varying structures. That's why we used Datasheets.com,

which compiles these documents into a user-friendly platform. By aggregating datasheets from various manufacturers, it provides a consistent and easily searchable repository. This platform simplifies the process of finding specific component information, saving time and effort. Instead of dealing with the inconsistencies and complexities of individual PDF files, users can quickly access the detailed specifications, features, and applications of electronic components through a streamlined interface. This approach significantly enhances efficiency and accuracy in retrieving the necessary data for design and troubleshooting purposes.

4.2 Data Transformation and Integration

4.2.1 Transformation Rules

Initially, our data cleaning process focused on refining our dataset by removing extraneous columns that did not align with our research goals. This included eliminating manufacturer-specific details, redundant descriptions, and any metadata unrelated to the electronic characteristics of the amplifiers. By streamlining the dataset in this manner, we ensured that our subsequent analysis would be focused and efficient, targeting only the pertinent information needed for our study.

Subsequently, our attention turned to enhancing the clarity and usability of the remaining columns containing electronic attributes. Many of these columns presented mixed information, such as numerical values intermingled with measurement units. To address this, we implemented a systematic approach of separating these mixed columns into distinct categories. For example, columns like "Gain (dB)" were divided into "Gain" for numerical values and "Gain Unit" for the corresponding unit of measurement ("dB"). This separation not only standardized the data format but also facilitated easier interpretation and analysis moving forward.

Data Extraction and Cleaning

In [2]:
```
import pandas as pd

df1 = pd.read_excel('./../AmplifiersData/AudioAmplifiers.xlsx')
df2 = pd.read_excel('./../AmplifiersData/GPSAmplifiers.xlsx')
df4 = pd.read_excel('./../AmplifiersData/OperationalAmplifiers.xlsx')
df5 = pd.read_excel('./../AmplifiersData/VideoAmplifiers.xlsx')
df6 = pd.read_excel('./../AmplifiersData/InstrumentationAmplifiers.xlsx')
```

In [145…
```
AudioAmpData = df1.drop(columns=['Description','Product line'])
AudioAmpData = AudioAmpData.to_csv(index=False)
AudioAmpData
```

Out[145…
```
'Part,Type,Manufacturer,PowerSupplyType,InputSignalType,OutputSignalType,OutputType,PinCount,TypicalSingleSupplyV
oltage,TypicalSingleSupplyVoltageUnit,MinimumSingleSupplyVoltage,MinimumSingleSupplyVoltageUnit,MaximumSingleSupp
lyVoltage,MaximumSingleSupplyVoltageUnit,MinimumOperatingTemperature,MinimumOperatingTemperatureUnit,MaximumOpera
tingTemperature,MaximumOperatingTemperatureUnit,MaximumPowerDissipation,MaximumPowerDissipationUnit\nLM4861M-NOPB
,Class-AB ,Texas Instruments ,Single ,Single ,Differential ,1-Channel Mono ,8,3|5,V,2,V,5.5,V,-40,°C,85,°C,-,mW\n
LM4889MA-NOPB ,Class-AB ,Texas Instruments ,Single ,Differential|Single ,Differential ,1-Channel Mono ,8,3|5,V,2.
2,V,5.5,V,-40,°C,85,°C,-,mW\nTPA3125D2N ,Class-D ,Texas Instruments ,Single ,Single ,Differential|Single ,1-Chann
el Mono|2-Channel Stereo ,20,12|15|18|24,V,10,V,26,V,-40,°C,85,°C,1870,mW\nBD28623MUV-E2 ,Class-D ,ROHM Semicondu
ctor ,Single ,Single ,Differential ,2-Channel Stereo ,24,9|12|15|18,V,8.5,V,24,V,-25,°C,85,°C,3560,mW\nOPA1611AID
,Class-AB ,Texas Instruments ,Single|Dual ,Differential ,Single ,1-Channel Mono ,8,5|9|12|15|18|24|28,V,4.5,V,36,
V,-40,°C,85,°C,-,mW\nPAM8945PJR ,Class-G ,Diodes Incorporated ,Single ,Differential ,Differential ,1-Channel Mono
,12,3|5,V,2.8,V,5.2,V,-40,°C,85,°C,-,mW\nLM4862M-NOPB ,Class-AB ,Texas Instruments ,Single ,Single ,Differential
,1-Channel Mono ,8,3|5,V,2.7,V,5.5,V,-40,°C,85,°C,-,mW\nLM386MMX-1-NOPB ,Class-AB ,Texas Instruments ,Single ,Dif
ferential ,Single ,1-Channel Mono ,8,5|9,V,4,V,12,V,-,°C,70,°C,595,mW\nBM28723AMUV-E2 ,Class-D ,ROHM Semiconducto
r ,Triple ,Single ,Differential ,2-Channel Stereo ,32,-,V,-,V,-,V,-25,°C,85,°C,-,mW\nPAM8610TR ,Class-D ,Diodes I
ncorporated ,Single ,Differential ,Differential ,2-Channel Stereo ,40,12,V,7,V,15,V,-20,°C,85,°C,-,mW\n'
```

Figure 4.3: Data Cleaning

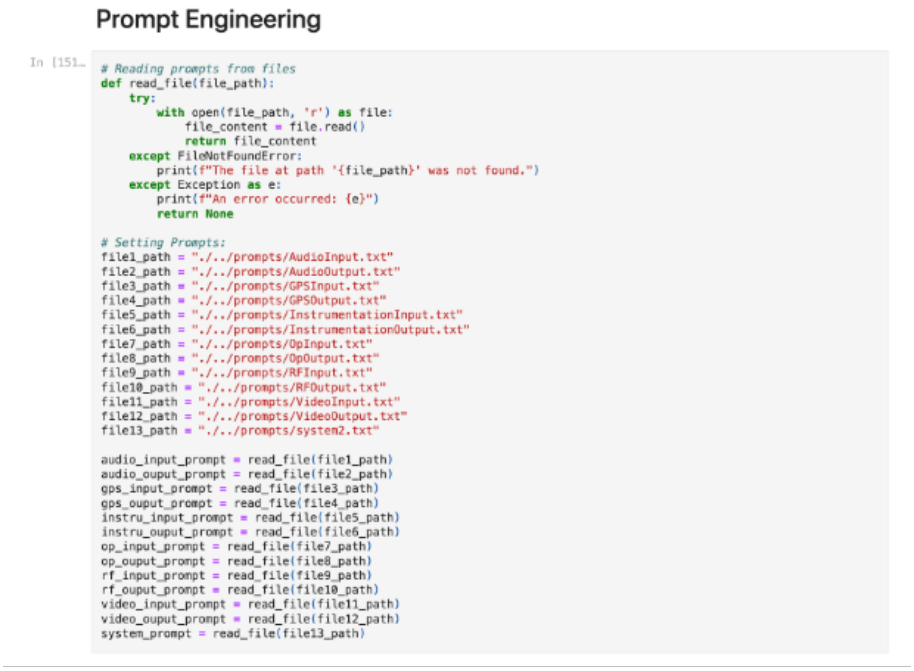
Furthermore, we encountered electronic property values that were provided in ranges, encompassing minimum, maximum, and average values. To ensure consistency and usability, we introduced dedicated columns for each type of value. For instance, properties like "Input Voltage" were divided into "Input Voltage Min," "Input Voltage Max," and "Input Voltage Avg." This standardized format made the dataset more structured and accessible, preparing it effectively for subsequent analytical tasks.

Once the data cleaning and restructuring were complete, we consolidated the refined information into unified data frames for each type of amplifier. Each data frame contained a subset of unique amplifiers with well-organized and standardized attributes. This consolidation not only reduced complexity but also optimized the dataset for straightforward access and manipulation during our analysis.

4.2.2 Prompt Engineering

Prompts play a pivotal role in shaping the behavior of large language models, guiding them towards producing specific responses. This section underscores the critical importance of prompts within the context of large language models, highlighting their role in ensuring accuracy and precision. Effective prompt crafting is emphasized as essential; without these qualities, model outputs can become unpredictable and ambiguous.

In our study, prompt engineering is employed specifically for extracting triplets in the format of head, relation, and tail. These triplets delineate relationships between amplifiers, their electrical characteristics, associated numerical values, and units of measurement.



Prompt Engineering

In [151…

```
# Reading prompts from files
def read_file(file_path):
    try:
        with open(file_path, 'r') as file:
            file_content = file.read()
            return file_content
    except FileNotFoundError:
        print(f"The file at path '{file_path}' was not found.")
    except Exception as e:
        print(f"An error occurred: {e}")
        return None

# Setting Prompts:
file1_path = "./../prompts/AudioInput.txt"
file2_path = "./../prompts/AudioOutput.txt"
file3_path = "./../prompts/GPSInput.txt"
file4_path = "./../prompts/GPSOutput.txt"
file5_path = "./../prompts/InstrumentationInput.txt"
file6_path = "./../prompts/InstrumentationOutput.txt"
file7_path = "./../prompts/OpInput.txt"
file8_path = "./../prompts/OpOutput.txt"
file9_path = "./../prompts/RFInput.txt"
file10_path = "./../prompts/RFOutput.txt"
file11_path = "./../prompts/VideoInput.txt"
file12_path = "./../prompts/VideoOutput.txt"
file13_path = "./../prompts/system2.txt"

audio_input_prompt = read_file(file1_path)
audio_ouput_prompt = read_file(file2_path)
gps_input_prompt = read_file(file3_path)
gps_ouput_prompt = read_file(file4_path)
instru_input_prompt = read_file(file5_path)
instru_ouput_prompt = read_file(file6_path)
op_input_prompt = read_file(file7_path)
op_ouput_prompt = read_file(file8_path)
rf_input_prompt = read_file(file9_path)
rf_ouput_prompt = read_file(file10_path)
video_input_prompt = read_file(file11_path)
video_ouput_prompt = read_file(file12_path)
system_prompt = read_file(file13_path)
```

Figure 4.4: Prompts Retrieval

We developed two distinct prompts: an input prompt specifying the types of entities and relations for the model to use when generating triplets, and an output prompt detailing the expected format or content of the triplets produced by the model. Our approach involves extracting these triplets in JSON format, facilitating structured data representation and analysis.

Input Prompt

```
input_prompt = """Based on the following example, extract entities and relations for each row from the provided specification.

Use the following entity types:
entity_types = {
    "PowerDissipation"
    "SingleSupplyVoltage"
    "SupplyVoltage"
    "OperatingTemperature"
    "OperatingSupplyVoltage"
    "DualSupplyVoltage"
    "SupplyCurrent"
    "InputBiasCurrent"
    "OutputCurrent"
    "PSRR"
    "CMRR"
    "Gain"
    "PowerGain"
    "GainBandwidthProduct"
    "OperatingFrequency"
    "NoiseFigure"
}

Use the following relation types:
relation_types = {
    "hasType",
    "hasProperty",
    "hasUnit",
    "hasNumericalValue",
    "hasLowerBoundNumericalValue",
    "hasUpperBoundNumericalValue",
    "hasManufacturer",
    "hasPowerSupplyType",
    "hasInputSignalType",
    "hasOutputSignalType",
    "hasOutputType",
    "hasPinCount",
    "hasProcessTechnology"
}

# specification
Part,Type,PowerSupplyType,InputSignalType,OutputSignalType,OutputType,PinCount,TypicalSingleSupplyVoltage,TypicalSingleSupplyVoltageUnit,Minimum
TDA7265 ,Class-AB ,Single ,Single ,Differential ,1-Channel Mono ,8,3|5,V,2,V,5.5,V,-40,°C,85,°C,-,mW
TDA7851A ,Class-AB ,Single ,Differential|Single ,Differential ,1-Channel Mono ,8,3|5,V,2.2,V,5.5,V,-40,°C,85,°C,-,mW
"""
```

Figure 4.5: Example Input Prompt

The input prompt provided is designed to guide a large language model (LLM) in extracting entities and relationships from a specification table of amplifiers.

Entity Types: The prompt specifies various types of entities that the LLM should recognize and extract from the specification table. It specifies a range of entity types, such as PowerDissipation, SupplyVoltage, and Gain, each representing specific characteristics of amplifiers.

Relation Types: It also defines several types of relationships (or properties) that should be identified between entities. It outlines various relation types like hasType, hasNumericalValue, and hasUnit, which dictate how these entities should be linked and categorized within the extracted data.

Specification Table: The specification table provided contains rows describing different amplifiers (e.g., TDA7265, TDA7851A) and their associated characteristics such as type, power supply type, input/output signal types, pin count, voltage ranges, temperature ranges, and power dissipation limits along with their values and respective units. It serves as an example data for LLM. The LLM's task is to interpret each row of this table, identifying entities and their associated properties as defined by the prompt.

Output Prompt

```
output_prompt = """
# Example output in JSON triplets for first and second row of specifications
[
    {
        "head":"TDA7265",
        "relation":"hasType",
        "tail":"Class-AB"
    },
    {
        "head": "TDA7265",
        "relation":"hasManufacturer",
        "tail":"Texas Instruments"
    },
    {
        "head":"TDA7265",
        "relation":"hasPowerSupplyType",
        "tail":"Single"
    },
    {
        "head":"TDA7265",
        "relation":"hasInputSignalType",
        "tail":"Single"
    },
    {
        "head":"TDA7265",
        "relation":"hasOutputSignalType",
        "tail":"Differential"
    },
    {
        "head":"TDA7265",
        "relation":"hasOutputType",
        "tail":"1-Channel Mono"
    },
    {
        "head":"TDA7265",
        "relation":"hasPinCount",
        "tail":"8"
    },
```

Figure 4.6: Example Output Prompt

The output prompt provided is a structured example of JSON triplets that represent extracted entities and their relationships from the first and second rows of the amplifier specifications. Each JSON object in the list corresponds to a specific triplet, structured as follows:

- Head: Represents the entity or concept being described. In this case, it typically refers to the amplifier part number (e.g., "TDA7265", "TDA7851A").
- Relation: Specifies the type of relationship or property between the head and the tail.
- Tail: Represents the value or attribute associated with the head entity, depending on the specified relation.

For instance, for the amplifier "TDA7265" is represented as follows:

- It has a type of "Class-AB"

```
{
    "head":"TDA7265",
    " relation ":"hasType",
    " tail ":"Class-AB"
}
```

- Manufactured by "Texas Instruments"

```
{
    "head": "TDA7265",
    " relation ": "hasManufacturer",
    " tail ": "Texas Instruments"
}
```

- Operates with a single power supply type

```
{
    "head": "TDA7265",
    " relation ": "hasPowerSupplyType",
    " tail ": "Single"
}
```

- Has a single input signal type of "Single"

```
{
    "head": "TDA7265",
    " relation ": "hasInputSignalType",
    " tail ": "Single"
}
```

- Output signal type is "Differential"

```
{
    "head": "TDA7265",
    " relation ": "hasOutputSignalType",
    " tail ": " Differential "
}
```

- Output type is "1-Channel Mono"

```
{
    "head": "TDA7265",
    " relation ": "hasOutputType",
    " tail ": "1-Channel Mono"
}
```

- It has 8 pins

```
{
    "head": "TDA7265",
    " relation ": "hasPinCount",
    " tail ": "8"
}
```

- Specific properties like "SingleSupplyVoltage" have associated numerical values and units

```
{
    "head": "SingleSupplyVoltage",
    " relation ": "hasNumericalValue",
    " tail ": "3|5"
},
{
    "head": "SingleSupplyVoltage",
    " relation ": "hasLowerBoundNumericalValue",
    " tail ": "2"
},
{
    "head": "SingleSupplyVoltage",
    " relation ": "hasUpperBoundNumericalValue",
    " tail ": "5.50"
},
{
    "head": "SingleSupplyVoltage",
    " relation ": "hasUnit",
    " tail ": "V"
}
```

- Operating temperature ranges from -40°C to 85°C

```
{
    "head": "OperatingTemperature",
    " relation ": "hasLowerBoundNumericalValue",
    " tail ": "−40"
},
{
    "head": "OperatingTemperature",
    " relation ": "hasUpperBoundNumericalValue",
    " tail ": "85"
},
{
    "head": "OperatingTemperature",
    " relation ": "hasUnit",
    " tail ": "Celsius"
}
```

- Power dissipation information is denoted as "-" with units in milliwatts

```
{
    "head": "PowerDissipation",
    " relation ": "hasUpperBoundNumericalValue",
    " tail ": "−"
},
{
    "head": "PowerDissipation",
    " relation ": "hasUnit",
    " tail ": "mW"
}
```

Each triplet encapsulates a specific aspect of the amplifier's specifications, formatted in JSON for structured representation. This output prompt serves as a guide for the LLM to generate structured data based on the input specifications, facilitating the creation of a knowledge base or dataset for further analysis and processing

4.2.3 Triplet Extraction using Gemini API

LLM for Triplets Extraction

```python
import vertexai
from vertexai import language_models

project_id = "modular-glider-408308"
location = "us-central1"

def extract_triplets(project_id: str, location: str, data: str, input_prompt:str, output_prompt:str) -> str:
    """Extract Triplets with a Large Language Model."""

    vertexai.init(project=project_id, location=location)

    chat_model = language_models.ChatModel.from_pretrained("gemini-1.0-pro-001")

    parameters = {
        "temperature": 0.8,
        "max_output_tokens": 2048,
        "top_p": 0.95,
        "top_k": 40,
    }

    chat = chat_model.start_chat(
        context=system_prompt,   #add our system_prompt
        examples=[
            language_models.InputOutputTextPair(
                input_text=input_prompt,   #add input data example
                output_text=output_prompt, #add output we expect
            ),
        ],
    )

    responses = chat.send_message_streaming(
        message=data,   #provide our own data
        **parameters,
    )

    results = []
    for response in responses:
        results.append(str(response.candidates[0].text))
    results = "".join(results)
    return results
```

Figure 4.7: Triplet Extraction Using Gemini API

Above code from figure 4.7 provides a Python function `extract_triplets` that uses Google's language model Gemini to extract structured information (triplets) from amplifier specification data sheets.The code starts by importing necessary libraries from the vertexai module, which allows interaction with Vertex AI, Google's machine learning platform. It sets `project_id` and `location` variables, which are essential for initializing and making API calls to Gemini.

The function `extract_triplets` is defined with parameters `project_id`, `location`, `data`, `input_prompt`, and `output_prompt`. The purpose of this function is to extract structured data (triplets) from a given text input (amplifier specifications). The pre-trained language model "gemini-1.0-pro-001" is loaded using Vertex AI's ChatModel class. This model will be used for extracting the triplets.

The parameters dictionary specifies various settings for the model:

1. `temperature:` Controls the randomness of the model's responses.
2. `max_output_tokens:` Maximum number of tokens (words or characters) the output can have
3. `top_p and top_k:` Control the diversity of the responses.

The `start_chat` method initiates a chat session with the model, providing a `system_prompt` (general context for the model) and examples of input-output pairs. These examples help the model understand what kind of output is expected based on the given input. The `send_message_streaming` method sends the data (amplifier specifications) to the model and streams the response using the specified parameters. The responses are collected from the model's streaming output. Each response is appended to the results list, which is then joined into a single string and returned.

This function is designed to take CSV data containing amplifier specifications and use a pre-trained large language model to extract structured triplets representing the electrical characteristics of the amplifiers. The triplets are in the format of (head, relation, tail), capturing relationships such as types, properties, and numerical values. This structured extraction facilitates the creation of a knowledge base or dataset, enabling more efficient analysis, understanding, and utilization of the amplifier specifications in various applications.

In [153…

```python
import csv
import json
from io import StringIO

def process_csv_to_triplets(csv_data, project_id, location, input_prompt, output_prompt):

    # Use StringIO to simulate reading from a file
    csv_file = StringIO(csv_data.strip())

    # Use csv.reader to read the CSV data
    reader = csv.reader(csv_file)

    # Read the header row
    header = next(reader)

    for row in reader:

        # Combine the header and row into a single list
        combined_row = header + row
        row_data = ','.join(combined_row)
        json_data = extract_triplets(project_id, location, row_data, input_prompt, output_prompt)

        print(json_data)
```

Figure 4.8: Call To Triplet Extraction Function

The `process_csv_to_triplets` function is designed to convert CSV data into structured JSON triplets using a large language model (LLM). This function is particularly useful for extracting detailed relationships and attributes from amplifier specifications and organizing them into a machine-readable format.

`csv_data`: A string containing CSV data. This data is expected to be structured, with the first row being the header and subsequent rows containing the data entries.

`project_id`: The project identifier used for initializing the Vertex AI environment.

`location`: The location parameter specifies the geographic location for the Vertex AI services.

`input_prompt`: A string defining the input prompt to guide the LLM on how to interpret the CSV data.

`output_prompt`: A string that specifies the expected format of the JSON triplets to be generated by the LLM.

For instance, if the CSV data contains specifications of amplifiers with columns such as `Part`, `Type`, `PowerSupplyType`, etc., the `process_csv_to_triplets` function will read each row, combine it with the header to maintain context, and use the `extract_triplets` function to generate JSON triplets. These triplets will represent relationships such as `Part`, `Type`, `PowerSupplyType`, and other properties, formatted in a structured JSON format given in figure 4.9 suitable for integration into knowledge bases.

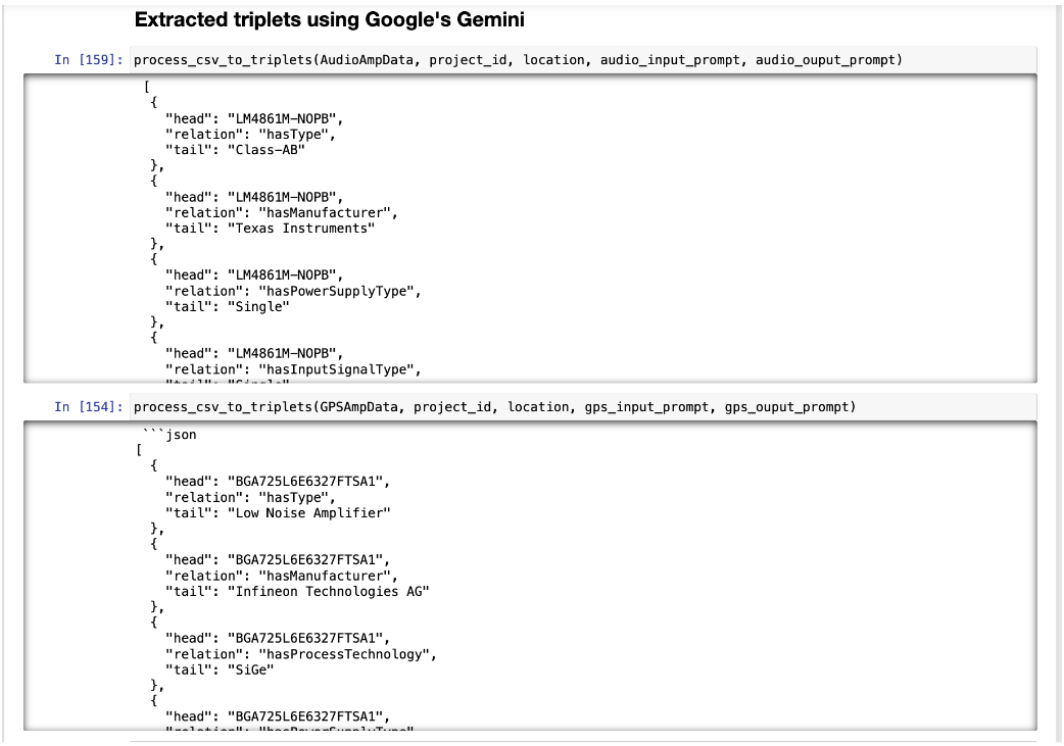


Figure 4.9: Extracted Triplets

4.3 Automating the Data Population Process Using Owlready2

Owlready2 is a comprehensive package designed for manipulating OWL 2.0 ontologies in Python. It provides capabilities to load, modify, and save ontologies, with reasoning support through the included HermiT reasoner. This library allows for seamless access to OWL ontologies and offers several powerful features. It can import ontologies in RDF/XML, OWL/XML, or NTriples formats and enables users to handle ontology classes, instances, and annotations as if they were standard Python objects. Additionally, Owlready2 allows the addition of Python methods to ontology classes and can automatically re-classify instances using the HermiT reasoner[2].

In our project, we extensively utilize Owlready2 to programmatically create individuals for various amplifier classes and to assign data and object properties to these individuals within their respective classes. This approach significantly streamlines our workflow by automating the assignment process, allowing us to save time and enhance efficiency. Rather than manually assigning properties to each individual one by one, we leverage Owlready2's capabilities to handle these tasks automatically, ensuring consistency and accuracy across the ontology.

4.3.1 Creating Amplifier Individuals

The provided code snippet in figure 4.10 illustrates a method for integrating JSON data into an OWL ontology using the Owlready2 library. The ontology is first loaded from an RDF file `AmpKB.rdf` in a specified directory. Subsequently, a JSON file containing triplets `InstruAmpTriplets.json` is read and converted into a Python dictionary. The code iterates over this dictionary to create individuals in the ontology according to the data. It checks the `tail` value of each triplet: if the value is `"Single"`, a new individual of the `SingleInstrumentalAmplifier` class is created; otherwise, a new individual of the `DualInstrumentalAmplifier` class is created. This automated approach streamlines the addition of individuals to the ontology, significantly reducing the time and effort needed compared to manual entry, thereby improving efficiency and consistency in ontology management.

Importing OWL File Using owlready2

In [3]:
```
from owlready2 import *
filepath = "./../OntoFiles/AmpKB.rdf"
onto = get_ontology(filepath)
onto.load()
```

Out[3]: get_ontology("http://w3id.org/amplifiers#")

1. Creating Amplifier's Individuals In OWL File Using JSON Triplets

Instrumental Amplifiers

In [45]:
```
import json

# Specify the path to your JSON file
json_file_path = './../ExtractedTriplets/InstruAmpTriplets.json'

# Read and parse the JSON file
with open(json_file_path, 'r') as file:
    data = json.load(file)

# Access the data as a Python dictionary
for array in data:
    if array[0]["tail"] == "Single":
            SingleInstrumentalAmplifier = onto.SingleInstrumentalAmplifier
            array[0]["head"] = SingleInstrumentalAmplifier(array[0]["head"])
    else:
        DualInstrumentalAmplifier = onto.DualInstrumentalAmplifier
        array[0]["head"] = DualInstrumentalAmplifier(array[0]["head"])
```

Figure 4.10: Creating Amplifier individuals using Owlready2

4.3.2 Creating Individuals For Electrical Charaterstics

The code snippet provided in figure 4.11 showcases an advanced technique for automating the creation of ontology individuals with specific attributes using the Owlready2 library. The procedure starts with loading an ontology from an RDF file named `AmpKB.rdf` located in a designated directory. This is achieved through the `get_ontology(filepath)` function to access the ontology and the `onto.load()` method to completely load it into memory, thus preparing it for further modifications.

Once the ontology is loaded, the code then processes a JSON file `GPSAmpTriplets.json` that contains triplet data. This JSON file is read and converted into a Python dictionary using the `json.load()` function. The dictionary is structured with arrays of objects, each representing a data triplet with attributes such as `"relation"` and `"tail"`.

The main functionality of the code involves iterating through these triplets to create ontology individuals. It examines the `"relation"` attribute to check if it corresponds to the value `"hasProperty"`. Based on the `"tail"` attribute, which indicates the type of property e.g. `"Gain"`, `"SingleSupplyVoltage"`, `"OperatingFrequency"` the script generates instances of the respective ontology classes.

In particular, the code employs a loop to manage multiple instances of each property type.

For each type, it dynamically creates a unique name for the new individual by appending a numerical index (from 1 to 10) to a property-specific prefix. This naming convention ensures each individual is uniquely identifiable within the ontology. For instance, if the property is `"Gain"`, the script produces instances named `"GPSGain1"`, `"GPSGain2"`, up to `"GPSGain10"`, with similar naming for other properties.

This automated process greatly improves efficiency by minimizing the manual labor involved in adding and configuring each individual within the ontology. By programmatically generating and naming individuals based on structured JSON data, the script accelerates the data integration process while ensuring consistency and accuracy. This approach is especially beneficial for managing large-scale ontologies where manual entry would be impractical and error-prone.

GPS Amplifiers

In [48]:

```
import json

# Specify the path to your JSON file
json_file_path = './../ExtractedTriplets/GPSAmpTriplets.json'

# Read and parse the JSON file
with open(json_file_path, 'r') as file:
    data = json.load(file)

# Access the data as a Python dictionary
for array in data:
    for obj in array:
        for i in range(1,11):
            if obj["relation"] == "hasProperty" and obj["tail"] == "Gain":
                Gain = onto.Gain
                gain_instance = "GPSGain"+str(i)
                gain_instance = Gain(gain_instance)
                gain_instance.
            if obj["relation"] == "hasProperty" and obj["tail"] == "SingleSupplyVoltage":
                SingleSupplyVoltage = onto.SingleSupplyVoltage
                SingleSupplyVoltage_instance = "GPSSV"+str(i)
                SingleSupplyVoltage_instance = SingleSupplyVoltage(SingleSupplyVoltage_instance)
            if obj["relation"] == "hasProperty" and obj["tail"] == "OperatingFrequency":
                OperatingFrequency = onto.OperatingFrequency
                OperatingFrequency_instance = "GPSOF"+str(i)
                OperatingFrequency_instance = OperatingFrequency(OperatingFrequency_instance)
            if obj["relation"] == "hasProperty" and obj["tail"] == "NoiseFigure":
                NoiseFigure = onto.NoiseFigure
                NoiseFigure_instance = "GPSNF"+str(i)
                NoiseFigure_instance = NoiseFigure(NoiseFigure_instance)
            if obj["relation"] == "hasProperty" and obj["tail"] == "OperatingTemperature":
                OperatingTemperature = onto.OperatingTemperature
                OperatingTemperature_instance = "GPSOT"+str(i)
                OperatingTemperature_instance = OperatingTemperature(OperatingTemperature_instance)
            if obj["relation"] == "hasProperty" and obj["tail"] == "SupplyCurrent":
                SupplyCurrent = onto.SupplyCurrent
                SupplyCurrent_instance = "GPSSC"+str(i)
                SupplyCurrent_instance = SupplyCurrent(SupplyCurrent_instance)
            if obj["relation"] == "hasProperty" and obj["tail"] == "PowerDissipation":
                PowerDissipation = onto.PowerDissipation
                PowerDissipation_instance = "GPSPD"+str(i)
                PowerDissipation_instance = PowerDissipation(PowerDissipation_instance)
```

Audio Amplifiers

Figure 4.11: Assigning Object Properties using Owlready2

4.3.3 Assigning Data Properties To Amplifier Individuals

Video Amplifiers

In [93]:

```
import json

# Specify the path to your JSON file
json_file_path = './../ExtractedTriplets/VideoAmpTriplets.json'

# Read and parse the JSON file
with open(json_file_path, 'r') as file:
    data = json.load(file)

instances = list(onto.VideoAmplifier.instances())

count = 0
# Access the data as a Python dictionary
for array in data:
    for obj in array:
        for instance in instances:
            if instance.name == obj["head"]:
                if obj["relation"] == "hasManufacturer":
                    print("Assigning: "+obj["head"]+" "+obj["relation"]+" "+obj["tail"])
                    instance.hasManufacturer.append(obj["tail"])
                if obj["relation"] == "hasPowerSupplyType":
                    print("Assigning: "+obj["head"]+" "+obj["relation"]+" "+obj["tail"])
                    instance.hasPowerSupplyType.append(obj["tail"])
                if obj["relation"] == "hasPinCount":
                    print("Assigning: "+obj["head"]+" "+obj["relation"]+" "+obj["tail"])
                    instance.hasPinCount.append(obj["tail"])
```

```
Assigning: LMH6714MF-NOPB hasManufacturer Texas Instruments
Assigning: LMH6714MF-NOPB hasPowerSupplyType Single|Dual
Assigning: LMH6714MF-NOPB hasPinCount 5
Assigning: LM6172IN-NOPB hasManufacturer Texas Instruments
Assigning: LM6172IN-NOPB hasPowerSupplyType Dual
Assigning: LM6172IN-NOPB hasPinCount 8
Assigning: LMH6720MF-NOPB hasManufacturer Texas Instruments
Assigning: LMH6720MF-NOPB hasPowerSupplyType Single|Dual
Assigning: LMH6720MF-NOPB hasPinCount 6
Assigning: LMH6739MQ-NOPB hasManufacturer Texas Instruments
Assigning: LMH6739MQ-NOPB hasPowerSupplyType Single
Assigning: LMH6739MQ-NOPB hasPinCount 16
Assigning: LMH6738MQ-NOPB hasManufacturer Texas Instruments
Assigning: LMH6738MQ-NOPB hasPowerSupplyType Single
Assigning: LMH6738MQ-NOPB hasPinCount 16
Assigning: AD812ARZ hasManufacturer Analog Devices
Assigning: AD812ARZ hasPowerSupplyType Single|Dual
Assigning: AD812ARZ hasPinCount 8
Assigning: LMH6722MA-NOPB hasManufacturer Texas Instruments
```

Figure 4.12: Assigning Data Properties using Owlready2

The provided code snippet in figure 4.12 demonstrates a technique for updating the properties of individuals within an OWL ontology using the Owlready2 library, with data sourced from a JSON file. This method involves several key steps: loading the ontology, processing the JSON data, and applying property updates to the ontology's existing individuals.

The process starts by loading the ontology from an RDF file named AmpKB.rdf, located in a specified directory. The `get_ontology(filepath)` function is used to access the ontology, and the `onto.load()` method is employed to fully load it into memory, preparing it for subsequent modifications.

After the ontology is loaded, the script specifies the path to the JSON file `VideoAmpTriplets.json` that contains the relevant data. This file is then read and parsed into a Python dictionary using the `json.load()` function. The data is organized as arrays

of objects, each representing a data triplet with attributes such as "head", "relation", and "tail".

The script retrieves a list of VideoAmplifier class instances from the ontology using list(onto.VideoAmplifier.instances()). It then iterates through the parsed JSON data, matching each "head" value with the names of these ontology instances. For each matched instance, the script examines the "relation" attribute to determine the type of property to update. Based on the relation type e.g. "hasManufacturer", "hasPowerSupplyType", "hasPinCount" the script appends the appropriate "tail" value to the instance's corresponding property.

The code also prints messages to indicate which property values are being assigned to which individuals, providing transparency and facilitating verification of the updates.

This approach of programmatically updating ontology individuals with structured JSON data significantly enhances efficiency by minimizing manual effort. Automating these updates ensures consistency and accuracy within the ontology, which is particularly beneficial for large-scale ontology management, where manual updates would be labor-intensive and prone to errors.

4.3.4 Assigning Data Properties To Amplifier Properties Classes

The provided code snippet demonstrates a methodical process for updating ontology properties using data from a JSON file. This process involves several critical steps to ensure that the appropriate attributes are assigned to ontology instances correctly.

Initially, the script reads and parses the JSON file AudioAmpTriplets.json into a Python dictionary. This data is organized into arrays of objects, where each object represents a data triplet with attributes such as "head", "relation", and "tail". This structured data is used to update the properties of ontology instances across different classes of audio amplifiers.

Next, the script identifies and creates lists of instances for three specific classes PowerDissipation, SingleSupplyVoltage, and OperatingTemperature that include the term "AUDIO" in their names. It achieves this by iterating through all instances of these classes and appending those with "AUDIO" in their names to corresponding lists pd_audio_list, ssv_audio_list, ot_audio_list.

For each list, the code processes the instances and the parsed JSON data concurrently using zip(). It checks the "head" value of each JSON object to match the class of the current instance and assigns the appropriate values to the instance's properties based on the "relation" attribute.

For PowerDissipation instances, if the "head" value is "PowerDissipation", the script updates properties such as hasUpperBoundNumericalValue and hasUnit with the "tail" values from the JSON data.

Audio Amplifiers

In [79]:

```
import json

# Specify the path to your JSON file
json_file_path = './../ExtractedTriplets/AudioAmpTriplets.json'

# Read and parse the JSON file
with open(json_file_path, 'r') as file:
    data = json.load(file)

pd_audio_list = []
for item in onto.PowerDissipation.instances():
    if "AUDIO" in item.name:
        pd_audio_list.append(item)

ssv_audio_list = []
for item in onto.SingleSupplyVoltage.instances():
    if "AUDIO" in item.name:
        ssv_audio_list.append(item)

ot_audio_list = []
for item in onto.OperatingTemperature.instances():
    if "AUDIO" in item.name:
        ot_audio_list.append(item)


# Access the data as a Python dictionary
for item, json_data in zip(pd_audio_list, data):
    for obj in json_data:
        if obj["head"] == "PowerDissipation":
            if obj["relation"] == "hasUpperBoundNumericalValue":
                print(item)
                print("Assigned: "+obj["head"]+" "+obj["relation"]+" "+obj["tail"])
                item.hasUpperBoundNumericalValue.append(obj["tail"])
            if obj["relation"] == "hasUnit":
                print("Assigned: "+obj["head"]+" "+obj["relation"]+" "+obj["tail"])
                item.hasUnit.append(obj["tail"])

# Access the data as a Python dictionary
for item, json_data in zip(ssv_audio_list, data):
    for obj in json_data:
        if obj["head"] == "SingleSupplyVoltage":
            if obj["relation"] == "hasNumericalValue":
                print(item)
                print("Assigned: "+obj["head"]+" "+obj["relation"]+" "+obj["tail"])
                item.hasNumericalValue.append(obj["tail"])
            if obj["relation"] == "hasLowerBoundNumericalValue":
                print("Assigned: "+obj["head"]+" "+obj["relation"]+" "+obj["tail"])
                item.hasLowerBoundNumericalValue.append(obj["tail"])
            if obj["relation"] == "hasUpperBoundNumericalValue":
                print("Assigned: "+obj["head"]+" "+obj["relation"]+" "+obj["tail"])
                item.hasUpperBoundNumericalValue.append(obj["tail"])
            if obj["relation"] == "hasUnit":
                print("Assigned: "+obj["head"]+" "+obj["relation"]+" "+obj["tail"])
                item.hasUnit.append(obj["tail"])

# Access the data as a Python dictionary
for item, json_data in zip(ot_audio_list, data):
    for obj in json_data:
        if obj["head"] == "OperatingTemperature":
            if obj["relation"] == "hasLowerBoundNumericalValue":
                print(item)
                print("Assigned: "+obj["head"]+" "+obj["relation"]+" "+obj["tail"])
                item.hasLowerBoundNumericalValue.append(obj["tail"])
            if obj["relation"] == "hasUpperBoundNumericalValue":
                print("Assigned: "+obj["head"]+" "+obj["relation"]+" "+obj["tail"])
                item.hasUpperBoundNumericalValue.append(obj["tail"])
            if obj["relation"] == "hasUnit":
                print("Assigned: "+obj["head"]+" "+obj["relation"]+" "+obj["tail"])
                item.hasUnit.append(obj["tail"])
```

```
AmpKB.AUDIOPD1
Assigned: PowerDissipation hasUpperBoundNumericalValue -
Assigned: PowerDissipation hasUnit mW
AmpKB.AUDIOPD10
Assigned: PowerDissipation hasUpperBoundNumericalValue -
Assigned: PowerDissipation hasUnit mW
AmpKB.AUDIOPD2
Assigned: PowerDissipation hasUpperBoundNumericalValue 1870
Assigned: PowerDissipation hasUnit mW
AmpKB.AUDIOPD3
Assigned: PowerDissipation hasUpperBoundNumericalValue 3560
Assigned: PowerDissipation hasUnit mW
```

x

Figure 4.13: Assigning Data Properties To Amplifier Properties Classes

For SingleSupplyVoltage instances, if the "head" value is "SingleSupplyVoltage", it updates properties like hasNumericalValue, hasLowerBoundNumericalValue, hasUpperBoundNumericalValue, and hasUnit.

For OperatingTemperature instances, if the "head" value is "OperatingTemperature", it updates hasLowerBoundNumericalValue, hasUpperBoundNumericalValue, and hasUnit.

Each update operation is logged with a print statement that shows the instance and the property being updated. This provides transparency and allows verification of the data integration process.

This method ensures precise updates of ontology instances with data from the JSON file, minimizing manual entry errors and improving the efficiency of data management. By automating these updates, the script promotes consistent and accurate ontology management, which is essential for handling extensive and complex ontologies.

4.3.5 Assigning Object Properties To Amplifiers

The code snippet outlines a process for incorporating properties from a JSON file into an OWL ontology using the Owlready2 library, specifically targeting video amplifier components. The process begins by loading and parsing the JSON file VideoAmpTriplets.json, which is converted into a Python dictionary containing arrays of objects, each representing a data triplet with attributes like "head", "relation", and "tail". The script then identifies and organizes instances of three specific ontology classes DualSupplyVoltage, SingleSupplyVoltage, and OperatingTemperature that include the term "VIDEO" in their names, storing these instances in distinct lists dsv_video_list, ssv_video_list, ot_video_list.

The main task of the script is to update VideoAmplifier instances with the parsed data. It achieves this by iterating over the JSON data and the relevant instance lists in parallel using zip(). For each VideoAmplifier instance, the script checks if the "head" value from the JSON data matches the instance's name. Depending on the instance's class DualSupplyVoltage, SingleSupplyVoltage, or OperatingTemperature the script appends the appropriate data triplets to the hasProperty attribute of the matched VideoAmplifier instances. Throughout this process, the script outputs messages to log which values are being assigned to which instances, offering transparency and tracking of the operations.

This automated method streamlines the process of updating ontology instances with data from structured JSON files, reducing the need for manual data entry and improving the accuracy and consistency of data integration. By automating these updates, the script facilitates efficient ontology management, which is crucial for handling complex ontologies and ensuring data integrity in extensive applications.

Video Amplifiers

In [127…

```
import json

# Specify the path to your JSON file
json_file_path = './../ExtractedTriplets/VideoAmpTriplets.json'

# Read and parse the JSON file
with open(json_file_path, 'r') as file:
    data = json.load(file)

dsv_video_list = []
for item in onto.DualSupplyVoltage.instances():
    if "VIDEO" in item.name:
        dsv_video_list.append(item)

ssv_video_list = []
for item in onto.SingleSupplyVoltage.instances():
    if "VIDEO" in item.name:
        ssv_video_list.append(item)

ot_video_list = []
for item in onto.OperatingTemperature.instances():
    if "VIDEO" in item.name:
        ot_video_list.append(item)

# Access the data as a Python dictionary
individuals_list = []
for array in data:
        individuals_list.append(array[0]["head"])

video_amp = list(onto.VideoAmplifier.instances())

# DualSupplyVoltage
for array,indi in zip(data,dsv_video_list):
    for instance in video_amp:
        if array[0]["head"] == instance.name:
            instance.hasProperty.append(indi)
            print("Assiging:",instance,"value",indi)
print("----")

# SingleSupplyVoltage
for array,indi in zip(data,ssv_video_list):
    for instance in video_amp:
        if array[0]["head"] == instance.name:
            instance.hasProperty.append(indi)
            print("Assiging:",instance,"value",indi)
print("----")

# OperatingTemperature
for array,indi in zip(data,ot_video_list):
    for instance in video_amp:
        if array[0]["head"] == instance.name:
            instance.hasProperty.append(indi)
            print("Assiging:",instance,"value",indi)
print("----")
```

```
Assiging: AmpKB.LMH6714MF-NOPB value AmpKB.VIDEODSV1
Assiging: AmpKB.LM6172IN-NOPB value AmpKB.VIDEODSV10
Assiging: AmpKB.LMH6720MF-NOPB value AmpKB.VIDEODSV2
Assiging: AmpKB.LMH6739MQ-NOPB value AmpKB.VIDEODSV3
Assiging: AmpKB.LMH6738MQ-NOPB value AmpKB.VIDEODSV4
Assiging: AmpKB.AD812ARZ value AmpKB.VIDEODSV5
Assiging: AmpKB.LMH6722MA-NOPB value AmpKB.VIDEODSV6
Assiging: AmpKB.THS7360IPW value AmpKB.VIDEODSV7
Assiging: AmpKB.THS7374IPW value AmpKB.VIDEODSV8
Assiging: AmpKB.THS7360IPWR value AmpKB.VIDEODSV9
----
Assiging: AmpKB.LMH6714MF-NOPB value AmpKB.VIDEOSSV1
Assiging: AmpKB.LM6172IN-NOPB value AmpKB.VIDEOSSV10
Assiging: AmpKB.LMH6720MF-NOPB value AmpKB.VIDEOSSV2
Assiging: AmpKB.LMH6739MQ-NOPB value AmpKB.VIDEOSSV3
Assiging: AmpKB.LMH6738MQ-NOPB value AmpKB.VIDEOSSV4
Assiging: AmpKB.AD812ARZ value AmpKB.VIDEOSSV5
Assiging: AmpKB.LMH6722MA-NOPB value AmpKB.VIDEOSSV6
Assiging: AmpKB.THS7360IPW value AmpKB.VIDEOSSV7
Assiging: AmpKB.THS7374IPW value AmpKB.VIDEOSSV8
Assiging: AmpKB.THS7360IPWR value AmpKB.VIDEOSSV9
----
```

Figure 4.14: Assigning Object Properties To Amplifiers

4.4 Knowledge Base

4.4.1 Overview

The `AmpKB.rdf` file represents a structured ontology specifically designed to model knowledge about amplifiers. Ontologies like this are crucial for organizing domain-specific knowledge in a way that is both machine-readable and human-interpretable. The use of the OWL (Web Ontology Language) within the RDF (Resource Description Framework) structure enables this knowledge base to encapsulate detailed information about amplifier characteristics, relationships, and hierarchies.

4.4.2 Classes and Hierarchies

The ontology is built around several key classes, each representing a fundamental aspect of amplifiers. The ontology structure details various amplifier types and their properties. `VideoAmplifiers` come in different configurations such as `SingleVideoAmplifier`, `DualVideoAmplifier`, `TripleVideoAmplifier`, `QuadVideoAmplifier`, and `HexVideoAmplifier` each designed to handle multiple video signal outputs with preserved quality. `AudioAmplifiers` are categorized into `ClassAB`, `ClassD`, and `ClassG`, each offering different trade-offs between efficiency and sound quality. `OperationalAmplifiers` include `AutoZeroOperationalAmplifier`, `ChopperOperationalAmplifier`, `HighSpeedOperationalAmplifier` and others, catering to a range of applications from precision to low noise. `GPSAmplifiers` like the `LowNoiseGPSAmplifier` are specialized to enhance weak GPS signals. `InstrumentalAmplifiers` are available as `DualInstrumentalAmplifier` and `SingleInstrumentalAmplifier` for precise signal amplification. Key properties across these amplifiers include `Gain`, `PowerDissipation`, `OperatingFrequency` and many others which are essential for evaluating their performance in various applications.

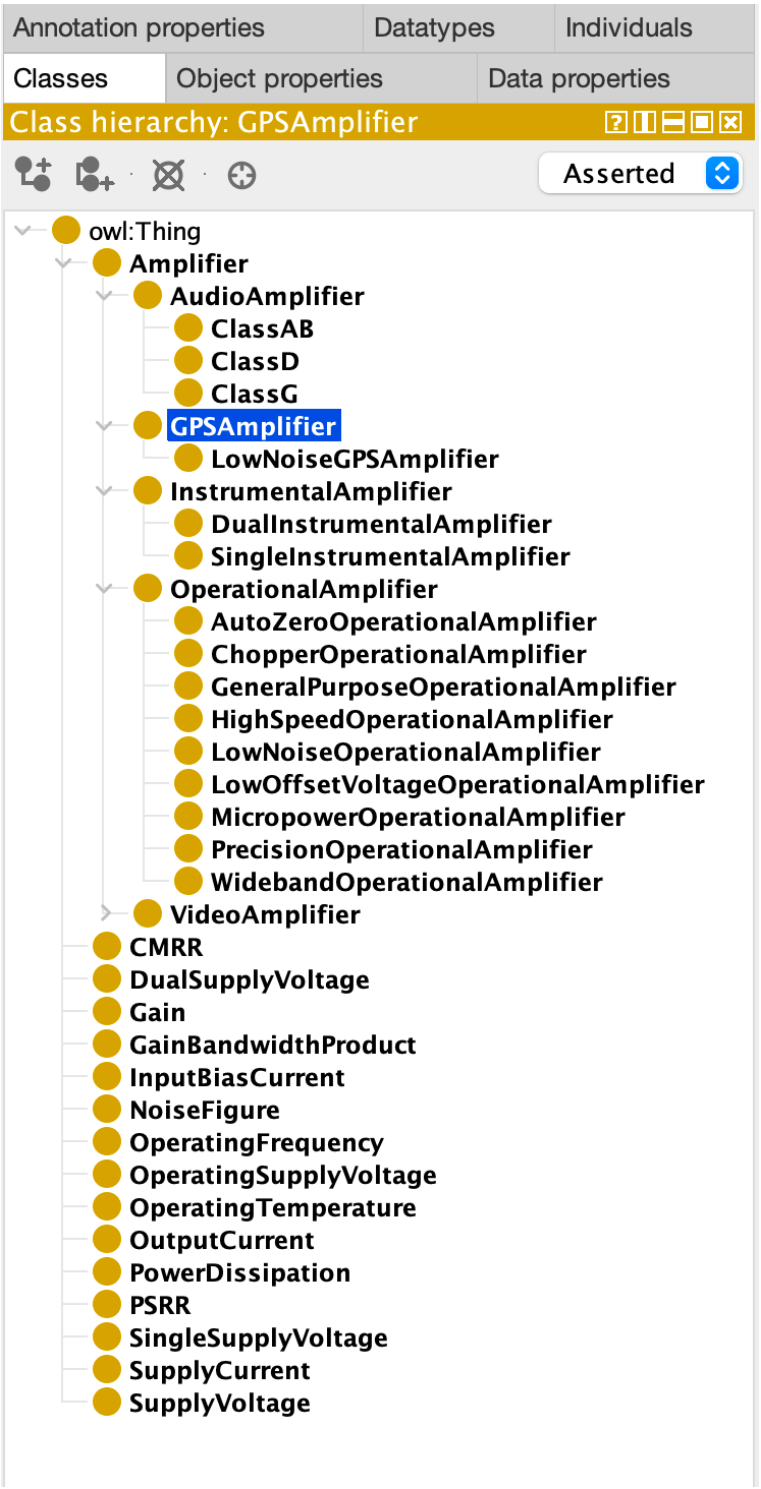


Figure 4.15: Classes and Hierarchies

These classes include:

- **VideoAmplifier**: Amplifies video signals while preserving quality, used in systems requiring signal transmission over distances or to multiple devices.
  - DualVideoAmplifier
  - HexVideoAmplifier
  - QuadVideoAmplifier
  - SingleVideoAmplifier
  - TripleVideoAmplifier
- **AudioAmplifier**: Boosts low-power audio signals to drive loudspeakers, essential in audio equipment like stereo systems and public address systems.
  - ClassAB
  - ClassD
  - ClassG
- **OperationalAmplifier**: Versatile amplifier used in signal processing, control systems, and analog computations, amplifying the difference between two input signals.
  - AutoZeroOperationalAmplifier
  - ChopperOperationalAmplifier
  - GeneralPurposeOperationalAmplifier
  - HighSpeedOperationalAmplifier
  - LowNoiseOperationalAmplifier
  - LowOffsetVoltageOperationalAmplifier
  - MicropowerOperationalAmplifier
  - PrecisionOperationalAmplifier
  - WidebandOperationalAmplifier
- **GPSAmplifier**: Enhances weak GPS signals to improve reliability and accuracy in navigation, timing, and geolocation systems.
  - LowNoiseGPSAmplifier
- **InstrumentalAmplifier**: Precision amplifier used for low-noise amplification of small signals, crucial in medical devices and applications requiring high accuracy.
  - DualInstrumentalAmplifier
  - SingleInstrumentalAmplifier
- **DualSupplyVoltage**

- **Gain**
- **GainBandwidthProduct**
- **InputBiasCurrent**
- **NoiseFigure**
- **OperatingFrequency**
- **OperatingSupplyVoltage**
- **OperatingTemperature**
- **OutputCurrent**
- **PowerDissipation**
- **CMRR**
- **PSRR**
- **SingleSupplyVoltage**
- **SupplyCurrent**
- **SupplyVoltage**

Each of these classes contains several instances that represent specific real-world amplifiers within the amplifier domain. These instances are connected through properties that define their relationships and attributes.

4.4.3 Properties and Relationships

The ontology leverages various properties to define the relationships between instances and to assign specific attributes to them. These properties include:

- **hasNumericalValue**: This property links an amplifier to a specific numerical attribute, such as its gain or power dissipation, providing a precise measurement for evaluation.
- **hasLowerBoundNumericalValue**: This property specifies the minimum value for an attribute, such as operating temperature or voltage, establishing the lower limit for the operational range.
- **hasUpperBoundNumericalValue**: This property defines the maximum value for an attribute, like power supply voltage or current, establishing the upper limit for the operational range.

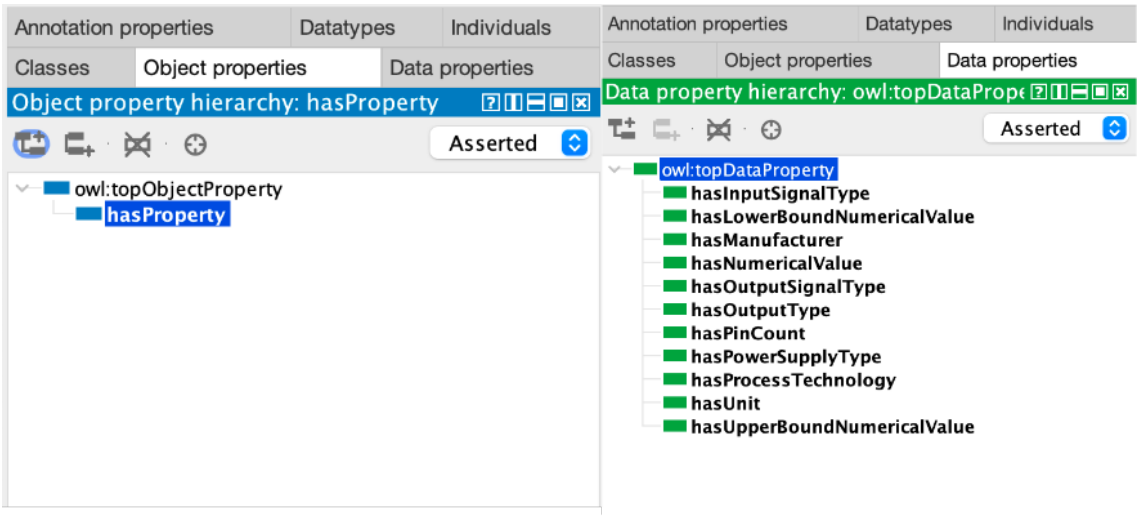


Figure 4.16: Object And Data Properties

- **hasUnit**: This property attaches units of measurement to numerical values, such as volts, amps, or degrees Celsius, ensuring clarity and standardization of data.
- **hasManufacturer**: This property associates an amplifier with its manufacturer, providing information about the origin and potential quality of the component.
- **hasPowerSupplyType**: This property specifies the type of power supply required by the amplifier, such as single or dual supply, which affects how the amplifier is integrated into a system.
- **hasInputSignalType**: This property defines the nature of the signal input that the amplifier can handle, such as analog or digital, determining its compatibility with various signal sources.
- **hasOutputSignalType**: This property describes the type of signal output produced by the amplifier, such as video or audio, indicating the format and use of the amplified signal.
- **hasOutputType**: This property specifies the nature of the output provided by the amplifier, such as buffered or direct, which affects how the signal is delivered to subsequent stages.
- **hasPinCount**: This property indicates the number of pins or terminals on the amplifier, relevant for understanding the physical connections and integration requirements.
- **hasProcessTechnology**: This property refers to the manufacturing process used to create the amplifier, such as CMOS or bipolar, which influences performance characteristics and efficiency.

4.4.4 Example of Individual In The Knowledge Base

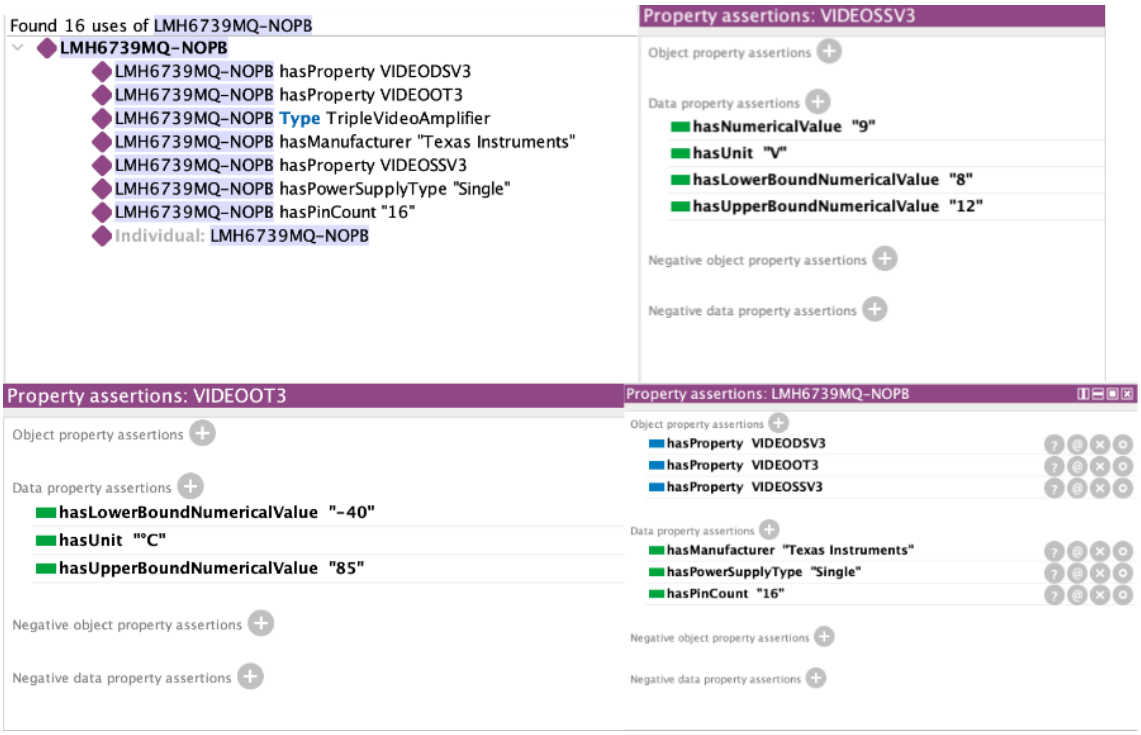


Figure 4.17: Individual in knowledge base

Class Hierarchy

- **Amplifier** (Superclass)
  - **TripleVideoAmplifier** (Subclass)

Property Definitions

- **hasManufacturer**
  - Domain: Amplifier
  - Range: Manufacturer
- **hasPinCount**
  - Domain: Amplifier
  - Range: `xsd:integer`

Instance Example

- **Instance**: LMH6739MQ-NOPB
- **Type**: TripleVideoAmplifier
- **Properties**:
  - **hasManufacturer**: Texas Instruments
  - **hasPinCount**: 14
  - **hasPowerSupplyType**: Single

4.4.5 Data Integration and Management

The ontology is designed to be flexible and extensible, allowing for easy integration of external data sources. For example, JSON files like `VideoAmpTriplets.json` can be parsed and used to dynamically populate the ontology with new instances or update existing ones. This is particularly valuable in scenarios where the ontology needs to be frequently updated with new data or when managing large-scale data that would be impractical to input manually.

The ontology also supports automated data population, which is achieved through scripts that parse external JSON data and programmatically update the ontology using owlready2. This process significantly reduces the time and effort required to maintain the ontology, ensuring that it remains up-to-date with minimal manual intervention. By matching JSON data with the ontology's structure, the system can accurately assign values to relevant properties, such as voltage levels, temperature ranges, and power dissipation limits.

4.4.6 Application and Utility

The `AmpKB.rdf` ontology is a powerful tool for managing knowledge in the field of amplifier technology. It provides a centralized repository of structured information that can be queried, analyzed, and extended as needed. Potential applications include:

- **Product Design and Optimization**: Engineers can use the ontology to quickly access relevant data about different amplifier components, aiding in the design and optimization of new products.
- **Supply Chain Management**: By including properties like **hasManufacturer**, the ontology can assist in tracking and managing supply chains, ensuring that all components meet the necessary specifications.
- **Educational Tools**: The ontology could be used in educational contexts to teach students about the relationships between different amplifier components and their respective properties.

- **AI and Machine Learning Integration**: The structured data within the ontology can serve as input for AI and machine learning algorithms, which can analyze the data to predict performance outcomes or identify potential issues in amplifier designs.

Chapter 5

Results

In previous chapters, we described the implementation process for extracting triples using Gemini model and automating the population of knowledge base using owlready2. In this chapter, we talk about the results achieved by our experiment. We will also discuss the query performance on the created knowledge base and the queries made to Gemini.

5.0.1 Data Acquisition and Cleaning

The data acquisition process began with obtaining amplifier-related datasheets from Datasheets.com, a comprehensive online resource. Using this platform, we efficiently downloaded the required datasheets in Excel format, which we then converted into CSV files to facilitate easier data manipulation and integration into our analysis.

When we imported these CSV files into Python as Pandas DataFrames, we encountered several challenges concerning the data's structure and content. The initial DataFrames contained numerous columns filled with unstructured text and irrelevant information, which did not align with our research objectives. To resolve this, we undertook a detailed data cleaning process, encompassing several key steps.

First, we removed unnecessary columns that included metadata, manufacturer-specific details, and redundant descriptions that were unrelated to the electronic characteristics of the amplifiers. This step effectively narrowed the dataset to only the most pertinent information, streamlining our subsequent analysis.

Next, we addressed the issue of mixed information within the remaining columns, where numerical values were often combined with their corresponding units of measurement. By systematically separating these into distinct categories, we standardized the data format, greatly improving its clarity and usability.

Furthermore, columns containing electronic property values expressed as ranges—such as minimum, maximum, and average values—were restructured into separate columns for each type of value. This adjustment ensured consistency across the dataset and enhanced its suitability for further analysis.

Through these cleaning and restructuring efforts, we created a well-organized and standardized dataset. We consolidated the refined information into unified DataFrames for each type of amplifier, containing only the most relevant and clean attributes. This organization not only simplified the dataset but also optimized it for easy access and manipulation during subsequent analytical tasks. As a result we have,

- **Streamlined Dataset:** We successfully refined the dataset to focus solely on relevant amplifier attributes by removing unnecessary columns and unstructured text, resulting in a more targeted and manageable dataset.
- **Standardized Data Format:** By separating numerical values from their corresponding units and restructuring range-based properties into distinct columns, we enhanced the dataset's consistency and usability.
- **Improved Data Organization:** The final DataFrames, containing only standardized and relevant data, were well-organized and optimized for efficient access and analysis, establishing a solid foundation for accurate and meaningful insights in later stages of the research.

These results significantly enhanced the quality and usability of the data, enabling more efficient and accurate analysis.

5.0.2 Triplet Extraction Performance

The triplet extraction process, utilizing prompt engineering and Google's Gemini API, led to notable enhancements in data structuring and analysis, especially concerning amplifier specification data. The methodology achieved the following outcomes:

- **Effective Triplet Extraction:** By designing precise input and output prompts, we effectively directed the language model to accurately and relevantly extract triplets from complex amplifier specification data. This approach enabled us to clearly define relationships among various amplifier attributes, including their electrical characteristics, numerical values, and measurement units, with high accuracy.
- **Structured Data Representation:** The triplets were extracted in JSON format, ensuring a structured and standardized presentation of the data. This formatting improved the accessibility and usability of the information, making it more straightforward to integrate into subsequent analytical processes or knowledge repositories.

- **Consistency Across Data Samples:** The use of prompt engineering ensured uniformity in the extraction process across different data samples. Regardless of the complexity or layout of the amplifier specifications, the language model consistently produced coherent and logically structured triplets, minimizing the chances of ambiguous or incorrect outputs.
- **Enhanced Analytical Foundation:** The organized and standardized triplets provided a solid foundation for further analysis. By converting the specifications into structured data, we streamlined the process of analyzing amplifier characteristics, which led to more precise insights and more efficient data handling.
- **Optimized Workflow Integration:** The triplet extraction process, particularly through the `process_csv_to_triplets` function, was seamlessly integrated into our workflow. This allowed for the smooth conversion of CSV data into JSON-formatted triplets, optimizing our ability to manage large datasets and ensuring that all relevant amplifier attributes were captured and prepared for subsequent tasks.

These outcomes highlight the effectiveness of prompt engineering and the application of advanced language models in extracting and structuring complex data, significantly improving the accuracy, consistency, and usability of the dataset for future research and analysis.

5.0.3 Knowledge Base Population

The deployment of Owlready2 for populating the knowledge base resulted in significant improvements in managing amplifier ontology data, particularly in terms of efficiency and consistency. The outcomes are summarized as follows:

- **Streamlined Creation of Amplifier Individuals:** Automating the integration of JSON data into the ontology greatly simplified the creation of amplifier individuals. By leveraging the Owlready2 library, individuals were programmatically generated from triplet data. This process reduced manual data entry time and improved accuracy, enabling the swift addition of instances for various amplifier classes, such as `SingleInstrumentalAmplifier` and `DualInstrumentalAmplifier`. Overall, this automation enhanced the efficiency of data entry and ontology management.
- **Enhanced Assignment of Electrical Characteristics:** The automation of assigning electrical characteristics to ontology individuals saw substantial improvement. The script efficiently processed triplet data from JSON files to create ontology individuals with attributes like `Gain`, `SingleSupplyVoltage`, and `OperatingFrequency`. It also automatically generated unique names for each instance and assigned the correct properties, thus minimizing manual effort and reducing errors.
- **High Consistency in Data Property Assignment:** Automated data property assignment ensured uniformity across the ontology. The script updated properties such as

`hasManufacturer`, `hasPowerSupplyType`, and `hasPinCount` by matching JSON data with existing ontology instances. This method enhanced the reliability of the ontology by ensuring that all relevant data properties were consistently and accurately applied.

- **Precise Property Updates for Amplifier Property Classes:** The approach used to update properties for amplifier property classes, including `PowerDissipation`, `SingleSupplyVoltage`, and `OperatingTemperature`, demonstrated high precision. The script processed JSON data and applied it to the relevant ontology instances, ensuring accurate updates and facilitating transparency and verification.
- **Efficient Object Property Integration:** The integration of object properties for video amplifiers was managed effectively through automation. The script assigned properties from JSON data to instances of `DualSupplyVoltage`, `SingleSupplyVoltage`, and `OperatingTemperature` classes. This automation reduced the need for manual data entry and improved the accuracy and consistency of property assignments across the ontology.

Overall, the use of Owlready2 for automating the knowledge base population process led to a more efficient, accurate, and consistent ontology management workflow. Automating the creation of individuals, the assignment of data properties, and the integration of object properties significantly enhanced the ability to handle complex and large-scale ontologies, improving data management and analysis processes.

Our research has successfully encapsulated the complexity of amplifiers electrical characteristics into a structured ontology, with Figure 5.3 providing a detailed view of all the classes. This figure is a snapshot from the ontology editing tool, Protégé, which shows the intricate properties and relationships associated within domain of automotive systems.

The development and integration of a structured ontology into the knowledge base represent a key outcome of this research. As illustrated in Figure 5.1, the ontology diagram details the complex interrelations and hierarchical classification of different amplifiers and their electrical characteristics. The ontology diagram provides a visual representation of the relationships between the amplifiers and their electric characteristics. The ontology has 40 classes, 1 object property, 11 Datatype Properties. It contains total 697 nodes and 2,560 edges between nodes[1].

[1] https://github.com/oovk/masters-thesis/tree/main

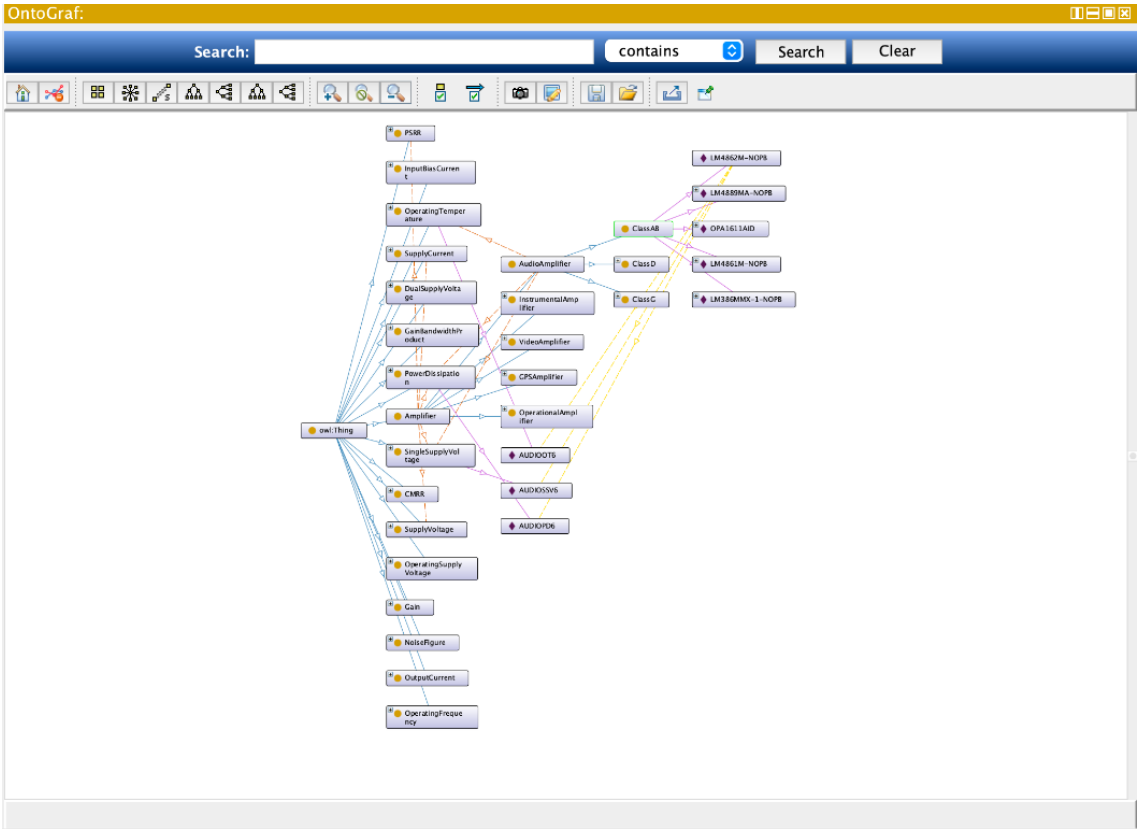


Figure 5.1: Amplifier Knowledge Base

The ontology structure depicted in above figure 5.1 illustrates a hierarchical and interconnected framework of amplifier electrical characteristics, attributes, and specific instances within a particular domain. At the foundation of this ontology is the concept labeled "`owl:Thing`" which serves as the most general category in the Web Ontology Language (OWL). From this fundamental concept, more specialized classes emerge, with "`Amplifier`" being a key subclass. The "`Amplifier`" class acts as an overarching category that includes various types of amplifiers, each designed for specific applications such as audio, video, instrumentation, GPS, and operational functions.

These various amplifier types such as `AudioAmplifier`, `InstrumentalAmplifier`, `VideoAmplifier`, `GPSAmplifier`, and `OperationalAmplifier` are all derived from the general "`Amplifier`" category, meaning they share fundamental amplifier characteristics while being tailored for unique purposes. Furthermore, there are additional classifications within these types, including `ClassAB`, `ClassD`, and `ClassG`, which likely pertain to different operational modes or amplifier design architectures. Each of these classifications is associated with specific

amplifier models like `LM4862M-NOPB`, `LM4889MA-NOPB`, and `OPA1611AID` representing real-world examples of amplifiers categorized under these classes.

The ontology also comprises a set of properties or attributes, such as `PSRR (Power Supply Rejection Ratio)`, `InputBiasCurrent`, `OperatingTemperature`, `SupplyCurrent`, and `GainBandwidthProduct`. These attributes play a vital role in defining the performance characteristics of the amplifiers. They are linked to the `"Amplifier"` class and, in some instances, to specific amplifier types, demonstrating how these properties are relevant across different categories and how they might affect the choice or operation of a particular amplifier type.

Overall, this ontology offers a well-structured and comprehensive representation of knowledge related to amplifiers. It systematically organizes information into categories, subcategories, and specific instances, facilitating a clearer understanding of the relationships between various amplifier types, their properties, and specific models. Such an ontology is particularly valuable in fields like data modeling, knowledge representation, and semantic web technologies, where organizing complex information systematically is crucial for effective data integration and retrieval.

5.0.4 Comparative Analysis of Queries to KG Vs LLM's

I've used multiple SPARQL queries to gather knowledge from a custom knowledge base I've built, and through this process, I've noticed distinct differences in how Knowledge Graphs and Large Language Models (LLMs) retrieve information. Both technologies can be used for querying knowledge, but they function in fundamentally different ways. As illustrated in the figure below, Knowledge Graphs find answers by connecting relevant nodes, while LLMs are designed to fill in gaps in text, often by predicting the [MASK] token to complete sentences[30].

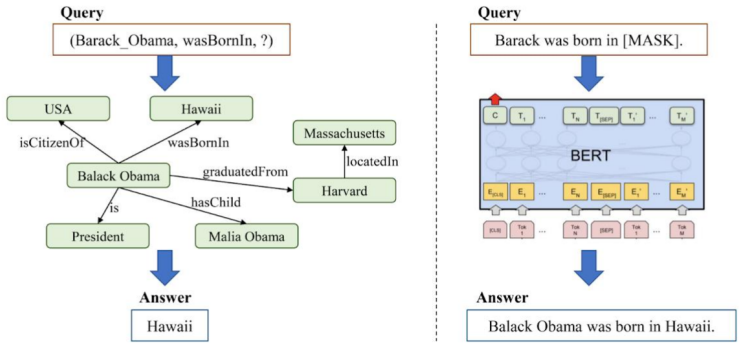


Figure 5.2: KG vs LLM[30]

Recently, LLMs like Gemini and GPT-4 have gained significant attention for their exceptional ability to understand and process language. These models are rapidly growing, both in size and the volume of data they are trained on, enabling them to hold vast amounts of knowledge. This has led to a growing trend where people are increasingly turning to tools like ChatGPT for answers, rather than using traditional search engines like Google. This shift has prompted researchers to investigate whether LLMs, such as GPT, could eventually replace knowledge graphs, as the main sources of information.

2. QUERY: ALL AMPLIFIERS WITH SINGLE SUPPLY VOLTAGE GREATER THAN 5

In [91]:

```
data = list(default_world.sparql("""
        SELECT DISTINCT ?individual ?singlesupplyvoltage ?value
        WHERE {
        ?individual rdf:type owl:NamedIndividual .
        ?individual <http://w3id.org/amplifiers#hasProperty> ?singlesupplyvoltage .
        ?singlesupplyvoltage rdf:type <http://w3id.org/amplifiers#SingleSupplyVoltage> .
        ?singlesupplyvoltage <http://w3id.org/amplifiers#hasNumericalValue> ?value .
        FILTER(?value > "3")
        }
    """))
df = pd.DataFrame(data, columns=["Individual", "SingleSupplyVoltage", "Value"])
print(df)
```

	Individual	SingleSupplyVoltage	Value
0	AmpKB.BD28623MUV-E2	AmpKB.AUDIOSSV3	9
1	AmpKB.INA122PA	AmpKB.INSTRUSSV2	5
2	AmpKB.INA125UA	AmpKB.INSTRUSSV4	5
3	AmpKB.LM386MMX-1-NOPB	AmpKB.AUDIOSSV7	5
4	AmpKB.LMH6639MA-NOPB	AmpKB.OPSSV6	5
5	AmpKB.LMH6714MF-NOPB	AmpKB.VIDEOSSV1	9
6	AmpKB.LMH6720MF-NOPB	AmpKB.VIDEOSSV2	9
7	AmpKB.LMH6722MA-NOPB	AmpKB.VIDEOSSV6	9
8	AmpKB.LMH6738MQ-NOPB	AmpKB.VIDEOSSV4	9
9	AmpKB.LMH6739MQ-NOPB	AmpKB.VIDEOSSV3	9
10	AmpKB.MCP6N11T-100E-MNY	AmpKB.INSTRUSSV10	3.3
11	AmpKB.MCP6N16-010E-MS	AmpKB.INSTRUSSV8	3.3
12	AmpKB.OPA1611AID	AmpKB.AUDIOSSV4	5
13	AmpKB.OPA343UA	AmpKB.OPSSV8	5
14	AmpKB.OPA625IDBVT	AmpKB.OPSSV5	5

3. QUERY: COUNT INDIVIDUALS WHO HAVE TEXAS INSTRUMENTS AS MANUFACTURER

In [96]:

```
data = list(default_world.sparql("""
        SELECT DISTINCT (COUNT(?individual) AS ?individualCount)
        WHERE {
        ?individual rdf:type owl:NamedIndividual .
        ?individual <http://w3id.org/amplifiers#hasManufacturer> ?manufacturer .
        FILTER(?manufacturer = "Texas Instruments")
        }
    """))
df = pd.DataFrame(data, columns=["Count"])
print(df)
```

	Count
0	30

Figure 5.3: SPARQL Queries to KG

In [8]:
```python
import vertexai
from vertexai.preview.generative_models import GenerativeModel, Part
import vertexai.preview.generative_models as generative_models

query_path = "./../prompts/QueriesPrompt.txt"

def llm_queries(query:str):
    config = {
        "max_output_tokens": 2048,
        "temperature": 0.9,
        "top_p": 1
    }
    model = GenerativeModel("gemini-1.0-pro-001")
    chat = model.start_chat()
    #print(chat.send_message(data + "what is this data about?"))
    response = chat.send_message(data + query)
    print(response.text)
```

In [9]:
```python
llm_queries("what is this data about?")
```

```
This data is about the different types of amplifiers, their electrical characteristics, and their manufacturers.
```

In [10]:
```python
llm_queries("Give me all the audio amplifier names from the part column of data")
```

```
- LM4861M-NOPB
- LM4889MA-NOPB
- TPA3125D2N
- BD28623MUV-E2
- OPA1611AID
- PAM8945PJR
- LM4862M-NOPB
- LM386MMX-1-NOPB
- BM28723AMUV-E2
- PAM8610TR
```

In [13]:
```python
llm_queries("Give me all the amplifier names from the part column of data")
```

```
- LM4861M-NOPB
- LM4889MA-NOPB
- TPA3125D2N
- BD28623MUV-E2
- OPA1611AID
- PAM8945PJR
- LM4862M-NOPB
- LM386MMX-1-NOPB
- BM28723AMUV-E2
- PAM8610TR
- BGA725L6E6327FTSA1
```

Figure 5.4: Queries to LLM

However, further research has shown that while LLMs possess broad general knowledge, they face significant challenges when it comes to recalling specific relational facts and understanding the connections between actions and events. Despite their many strengths, LLMs encounter several issues, including:

- **Hallucinations:** LLMs can sometimes produce convincing but inaccurate information, whereas Knowledge Graphs provide structured, fact-based answers grounded in verified data.
- **Limited reasoning abilities:** LLMs often struggle with tasks that require logical reasoning or using supporting evidence to draw conclusions, particularly in areas like numerical computation or symbolic reasoning. Knowledge Graphs, with their explicit mapping of

relationships, offer superior reasoning capabilities.

- **Lack of domain-specific knowledge:** While LLMs are trained on large amounts of general data, they often lack specialized knowledge, such as the technical terminology found in medical or scientific reports. Knowledge Graphs, on the other hand, can be customized for specific domains.
- **Knowledge obsolescence:** The training of LLMs is expensive and time-consuming, which means their knowledge can become outdated over time. In contrast, Knowledge Graphs can be updated more easily without the need for retraining.
- **Bias, privacy, and toxicity:** LLMs may generate biased or offensive responses, while Knowledge Graphs, built from reliable data sources, are generally free from such issues.

While Knowledge Graphs do not face these challenges and usually offer better consistency, reasoning ability, and interpretability, they have their own limitations. Unlike LLMs, Knowledge Graphs lack the flexibility that comes from the unsupervised training processes of LLMs, which allows them to handle a broader range of queries and contexts. Despite these differences, the strengths and weaknesses of both approaches highlight their complementary roles in the evolving landscape of AI-driven knowledge retrieval.

Chapter 6

Conclusion

This thesis has investigated novel methods for the automatic creation and enhancement of knowledge bases, utilizing advanced tools such as Google's Gemini LLM, the Owlready2 Python library, and the GENIAL! Graph Generation Framework. By harnessing these technologies, we aimed to tackle the complexities, high costs, and time-consuming challenges traditionally associated with knowledge base development.

Our findings indicate that integrating large language models (LLMs) like Gemini into the knowledge base creation process offers significant benefits in terms of entity extraction and relationship identification. These strengths, when combined with the structured methodologies of Owlready2 and the predefined reasoning frameworks provided by GENIAL!, contribute to a powerful approach for generating enriched and scalable knowledge bases.

A comparative analysis of SPARQL query performance on conventional knowledge bases versus LLM based queries highlighted important insights into the strengths and weaknesses of each method. While LLMs demonstrate exceptional flexibility and the ability to handle unstructured queries, traditional knowledge bases excel in consistency, precision, and reasoning capabilities, particularly within specific domains.

In conclusion, our proposed approach shows considerable potential for making knowledge base creation more accessible, efficient, and adaptable to the evolving needs of various fields. This research contributes to the broader application of artificial intelligence, facilitating faster access to relevant information, improving organizational efficiency, and encouraging ongoing learning and innovation.

Future Work

Building on the findings of this thesis, future research can explore several promising areas. Our research could explore the integration of knowledge graphs and large language models (LLMs) to leverage the strengths of both technologies. While Knowledge Graphs can guide LLMs toward greater accuracy, LLMs can, in turn, assist in the automatic construction of knowledge graphs by extracting relevant knowledge and improving their overall quality. There are several promising approaches to merging these concepts:

- **Utilizing LLMs for automatic Knowledge Graph construction**: LLMs can be employed to extract knowledge from unstructured data sources to populate a Knowledge Graph, thereby streamlining the construction process.
- **Enhancing LLM reasoning with Knowledge Graphs**: Knowledge Graphs can be integrated into LLMs' reasoning processes, helping them search for and apply knowledge more effectively, ultimately leading to more accurate answers.
- **Developing Knowledge Graph-enhanced pre-trained language models (KGPLMs)**: These approaches aim to incorporate Knowledge Graphs into the training process of LLMs, creating models that benefit from the structured knowledge and reasoning capabilities of Knowledge Graphs.

Future work will focus on these approaches, aiming to develop more sophisticated methods for integrating Knowledge Graphs and LLMs. This integration has the potential to significantly enhance the accuracy, efficiency, and scalability of knowledge retrieval systems, contributing to further advancements in artificial intelligence.

List of Figures

Literature

[1] Gemini Team, Rohan Anil, Sebastian Borgeaud, Yonghui Wu, Jean-Baptiste Alayrac, Jiahui Yu, Radu Soricut, Johan Schalkwyk, Andrew M Dai, Anja Hauth, et al. Gemini: a family of highly capable multimodal models. *arXiv preprint arXiv:2312.11805*, 2023.

[2] Jean-Baptiste Lamy. Owlready: Ontology-oriented programming in python with automatic classification and high level constructs for biomedical ontologies. *Artificial intelligence in medicine*, 80:11–28, 2017.

[3] Frank P Wawrzik and Andreas Lober. A reasoner-challenging ontology from the microelectronics domain. In *SemREC@ ISWC*, pages 1–12, 2021.

[4] George A Miller. Wordnet: a lexical database for english. *Communications of the ACM*, 38(11):39–41, 1995.

[5] Robyn Speer, Joshua Chin, and Catherine Havasi. Conceptnet 5.5: An open multilingual graph of general knowledge. In *Proceedings of the AAAI conference on artificial intelligence*, volume 31, 2017.

[6] Deborah L McGuinness, Frank Van Harmelen, et al. Owl web ontology language overview. *W3C recommendation*, 10(10):2004, 2004.

[7] Arzoo Katiyar and Claire Cardie. Going out on a limb: Joint extraction of entity mentions and relations without dependency trees. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 917–928, Vancouver, Canada, July 2017. Association for Computational Linguistics.

[8] Wikipedia contributors. Ontology. https://en.wikipedia.org/wiki/Ontology, 2022. [Online; accessed 13-October-2023].

[9] Frank Wawrzik. *Knowledge Representation in Engineering 4.0*. PhD thesis, Technische Universität Kaiserslautern, 2022.

[10] Diksha Khurana, Aditya Koli, Kiran Khatter, and Sukhdev Singh. Natural language processing: State of the art, current trends and challenges. *Multimedia Tools and*

Applications, 82, 07 2022.

[11] Unknown Author. A comprehensive overview of large language models. *ar5iv*, 2023. Accessed: 2024-01-21.

[12] Yunpeng Huang, Jingwei Xu, Zixu Jiang, Junyu Lai, Zenan Li, Yuan Yao, Taolue Chen, Lijuan Yang, Zhou Xin, and Xiaoxing Ma. Advancing transformer architecture in long-context large language models: A comprehensive survey, 2023.

[13] Ashutosh Gore. Understanding GPT — ashutoshfvt. https://medium.com/@ashutoshfvt/understanding-gpt-c6e64278f5a5. [Accessed 14-08-2024].

[14] Mostafa Ibrahim. An Overview of Large Language Models (LLMs) — wandb.ai. https://wandb.ai/mostafaibrahim17/ml-articles/reports/An-Overview-of-Large-Language-Models-LLMs---VmlldzozODA3MzQz. [Accessed 14-08-2024].

[15] Intro to Word Embeddings and Vectors for Text Analysis. — shanelynn.ie. https://www.shanelynn.ie/get-busy-with-word-embeddings-introduction/. [Accessed 14-08-2024].

[16] Great Learning Team. What is Word Embedding | Word2Vec | GloVe — mygreatlearning.com. https://www.mygreatlearning.com/blog/word-embedding/. [Accessed 14-08-2024].

[17] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need, 2023.

[18] John D Kelleher and Simon Dobnik. What is not where: the challenge of integrating spatial representations into deep learning architectures. *arXiv preprint arXiv:1807.08133*, 2018.

[19] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners, 2020.

[20] Lilian Weng. Prompt engineering. https://lilianweng.github.io/posts/2023-03-15-prompt-engineering/, 2023. Accessed: 2024-01-21.

[21] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.

[22] Andrew Kean Gao. Prompt engineering for large language models. 2023.

[23] Brad Nikkel. Advanced prompt engineering techniques: Tree-of-thoughts prompting. https://deepgram.com/learn/tree-of-thoughts-prompting, 2023. Accessed: 2024-01-21.

[24] Boshi Wang, Sewon Min, Xiang Deng, Jiaming Shen, You Wu, Luke Zettlemoyer, and Huan Sun. Towards understanding chain-of-thought prompting: An empirical study of what matters, 2023.

[25] Chris. Exploring role-play prompting llms – what does science say? https://blog.finxter.com/exploring-role-play-prompting-llms-what-does-science-say/, 2023. Accessed: 2024-01-21.

[26] Xiang Yue, Yuansheng Ni, Kai Zhang, Tianyu Zheng, Ruoqi Liu, Ge Zhang, Samuel Stevens, Dongfu Jiang, Weiming Ren, Yuxuan Sun, et al. Mmmu: A massive multi-discipline multimodal understanding and reasoning benchmark for expert agi. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 9556–9567, 2024.

[27] Mark A Musen. The protégé project: a look back and a look forward. *AI matters*, 1(4):4–12, 2015.

[28] Thomas Kluyver, Benjamin Ragan-Kelley, Fernando Pérez, Brian Granger, Matthias Bussonnier, Jonathan Frederic, Kyle Kelley, Jessica Hamrick, Jason Grout, Sylvain Corlay, et al. Jupyter notebooks–a publishing format for reproducible computational workflows. In *Positioning and power in academic publishing: Players, agents and agendas*, pages 87–90. IOS press, 2016.

[29] Wes McKinney et al. pandas: a foundational python library for data analysis and statistics. *Python for high performance and scientific computing*, 14(9):1–9, 2011.

[30] Linyao Yang, Hongyang Chen, Zhao Li, Xiao Ding, and Xindong Wu. Give us the facts: Enhancing large language models with knowledge graphs for fact-aware language modeling. *IEEE Transactions on Knowledge and Data Engineering*, 36(7):3091–3110, 2024.