Michael J Iantosca
Senior Director of Content Platforms
Avalara Inc.
Michael.Iantosca@avalara.com

Intelligence in, knowledge out

AUGMENTING LARGE LANGUAGE MODELS WITH KNOWLEDGE GRAPHS FOR EFFECTIVE, RESPONSIBLE AND EXPLAINABLE AI (XAI)

JUNE, 2023

Table of Contents

Preface 3

Generative AI simply cannot do it alone 5

Stochastic versus neuro-symbolic AI 5

The quest for responsible and explainable AI (XAI) 6

Generative AI that works – a hybrid solution 6

So how do we augment generative models, and with what? 7

Public vs Enterprise large language models 7

Fundamental concepts of neuro-symbolic knowledge 7

- Taxonomies 7
- Ontologies 8
- Taxonomies and ontologies: How they differ and how they work together 8
- Knowledge Graphs 9
- How an ontology serves as the basis for creating a knowledge graph 9
- Employing and querying a knowledge graph 10
- SPARQL 11

Fundamental concepts of stochastic generative AI 12

- Vectors 12
- Cosine similarity 13
- Vector selection during text generation 13
- Vectors that represent sentences and entire documents 14
- Embedding versus fine-tuning 15
- Embeddings 16
- Concept embeddings 17
- Creation of concept embeddings 17
- Methods for representing triples during ingestion 17
- LLMs that support custom embeddings 18
- Prompt Prefixes 19
- Improving the accuracy and completeness of generative chatbot answers with an ontology 20
- Prompt Injection 21
- Embedding prompt prefixes to structured text to improve unsupervised training 21
- Fusing neuro-symbolic knowledge with stochastic generative AI 22
- Deployment patterns 22

Patterns for combining knowledge graphs and generative technology 22
User prompt or search argument 23
Knowledge graph prompt refinement 23
RDF tuned LLM 23
Knowledge graph validation of generated content 24
Curated domain ontology (enterprise) 24
Curated domain taxonomies, classification, and autoclassification 24
Terminology base 24
Using a knowledge graph to perform generative pre-retrieval 25
Testing methodology and validating the pattern 26
Developing the required semantic stack 26
Intelligent content objects containers 27
Terminology 28
Taxonomy 29
Ontology 29
Knowledge Graph 29
Source content repositories and the content lake 29
About the Author 31

Preface

I've been told I have the uncanny ability to see around corners throughout my 40-plus-year career. I don't. My unique role as a seasoned enterprise content strategist and knowledge engineer, and the responsibility to deliver practical and industry-leading content solutions inform me of what I'll need to deliver long before they become practical from both a business and technology perspective. That makes it exceedingly challenging to explain advanced concepts to business leaders and worse, obtain the necessary resources to lead and win in the marketplace when the confluence of required elements is ripe and presents itself. Nothing happens in just a few weeks or months; remarkable things are typically years in the making when most others are completely unaware. This is the exact situation we find ourselves in with generative AI; it didn't just suddenly happen – it's been years in the making. Those of us that have been working in AI since the early days of Expert System in the early 90s, machine learning and IBM Watson in the 2000s, localization, computational linguistics, and structured knowledge management knew this was coming. AI and NLP have been part of our jobs for years, and we published many articles and given many webinars over the past few years to prepare folks, but no one could predict exactly when generative AI would reach critical mass and mass awareness, which it recently has.

Moreover, there's a whole community of professional knowledge engineers who understand what's needed to make generative AI work well. These are often professionals that were formally trained in library sciences or OTJ over many years. A skilled and seasoned professional Ontologist is worth his or her weight in gold, and as I often say, once organizations learn that they are essential and scarce, they'll soon be recruited straight out of schools like the NFL draft, even as undergraduates. I've been watching the likes of Microsoft, Amazon, and Google snapping them up in increasing numbers over the past two years. Mark my words.

For the past several years I've been out on the conference and webinar circuit explaining the fundamentals of neuro symbolic models, although I didn't use that term so as not to lose my audience in geek speak. Explaining the necessity of structured knowledge management was geeky enough. As much as I'd try to break down the concepts and practical use cases, I'd watch eyes begin to water and see the confusion written on the faces of the audience when I'd explain the very basics of taxonomies. If I didn't lose them at taxonomies, I more often watched them lose it the moment I mentioned ontologies and knowledge graphs. At the time, it just seemed boring to many that didn't understand their value as the fundamental building blocks for the next generation of AI technology that is now squarely upon us.

Don't believe me? Go read the paper and articles I've posted on my web at www.thinkingdocumenation.com for the past couple of years. It's been difficult being among the small chorus of voices explaining what structured knowledge is, long before generative AI burst onto the scene and into broad public awareness.

Sadly, most organizations are still not listening – nor investing properly.

Michael

Generative AI simply cannot do it alone

Disclaimer: The models and methods discussed herein have yet to be fully tested and validated, but the fundamental concept of fusing curated knowledge and generative technology is quickly growing widespread support. Early practitioners who are currently implementing some of these notions are seeing promising results. What is lacking is defining which of the proposed deployment patterns are most effective and the testing and validation of those models.

Beyond all of the hype and use of generative AI, such as ChatGPT, one basic truth persists: generative AI alone cannot provide the effectiveness, reliability, precision, and trustworthiness that most companies require for customer-facing use. The majority of developers creating generative solutions seem to be in a period of denial regarding this truth, implementing applications such as chatbots that they hope will work well enough to be useful to support their customers. Until they start testing. Their misplaced hope persists thinking that the next version of ChatGPT with exponentially more parameters will fulfill their hopes. I think that they are sadly mistaken as I've seen no evidence that the predictive nature across GPT-3, GPT-3.5, and GPT-4 appears no different. In fact, in some tests, GPT-3 performed better than GPT-4.

Their misplaced hope lies in the incontrovertible fact that large language models cannot infer nor reason. They are, fundamentally, predictive language patterns based on mathematical calculation as opposed to a true understanding of language and relationships. In short, they can't explain their results. Generative AI models such as ChatGPT are not and never can be, explainable AI (XAI) to be trustworthy on their own.

When someone asks a question, we humans are adept at inferring intent. LLMs lack that ability and sadly, will never have those abilities without the additional infusion of external intelligence. When a question is precise and the intent clear, generative AI works better. But a question often does not provide the actual intent of the questioner, leaving the LLM in the lurch and prone to failure, which is a major problem where accuracy and completeness of the generated responses matter.

Stochastic versus neuro-symbolic AI

A stochastic language model is a type of language model that incorporates randomness into its predictions. Unlike deterministic models, which always generate the same output given the same input, a stochastic LLM introduces variability by sampling from a probability distribution during the generation process. This randomness enables the model to produce diverse and creative outputs, making it particularly useful for tasks such as text generation, dialogue systems, and machine translation.

Now let's compare a stochastic model with a neuro-symbolic model.

A neuro-symbolic model refers to an approach that combines neural networks, which excel at learning from substantial amounts of data, with symbolic reasoning, which deals with logical and symbolic representations. By integrating a stochastic language model with a neuro-symbolic model, the integration allows the model to leverage the strengths of both approaches, enabling it to understand and reason about complex relationships in data while also being able to make logical inferences and interpret explicit rules or knowledge. Neuro-symbolic models have shown promise in areas such as natural language understanding, knowledge graph reasoning, and intelligent decision-making systems.

A stochastic model primarily focuses on generating diverse and creative outputs by introducing randomness into its predictions. However, it lacks the explicit symbolic reasoning capabilities found in neuro-symbolic models. Neuro-symbolic models can interpret explicit rules, perform logical inferences, and reason about complex relationships, which stochastic models struggle to achieve. Stochastic models excel at generating novel content, while neuro-symbolic models combine both data-driven learning and explicit reasoning, enabling them to tackle more complex tasks that require symbolic understanding.

A neuro-symbolic model is specifically designed to incorporate symbolic reasoning capabilities. By integrating neural networks with symbolic reasoning techniques, such as knowledge graphs, neuro-symbolic models can perform logical inference, interpret explicit rules, and reason about complex relationships in data. This combination allows the model to leverage the strengths of both approaches, enabling it to handle tasks that involve both data-driven learning and explicit reasoning, making them well-suited for applications requiring advanced reasoning abilities.

The quest for responsible and explainable AI (XAI)

Explainable AI (XAI) is the goal. The trouble is that not enough developers new to the practice of generative AI, let alone advanced knowledge management, understand what XAI and responsible AI, is.

Explainable AI refers to the development of AI systems that can provide understandable and transparent explanations for their decisions and actions. It focuses on making the inner workings of AI models and algorithms more interpretable to humans. XAI aims to bridge the gap between the complex and opaque nature of many AI techniques and the need for human users to understand and trust the decisions made by AI systems. By providing explanations, XAI enhances transparency, and accountability, and enables users to comprehend the reasoning behind AI-based outcomes, fostering trust and facilitating better decision-making.

So, what's the problem? The problem is that new generative AI solution developers, such as those creating generative chatbots, are not asking the right questions about how to make generative AI work with high precision and reliability. They are not asking the fundamental question: how?

Those perplexities stem from the fact that developers are experts in traditional programming and logic. They are not experts in knowledge management. Worse, they might even think that they are when that couldn't be further from the truth. Developing effective generative AI chatbots, for example, is 10% traditional software programming and 90% advanced knowledge management. Eventually, they'll discover this fact, but will continually hit walls until they do, losing critical time and business value.

Generative AI that works – a hybrid solution

What is required to make generative AI work successfully in mission-critical environments? It's not the use of predictive mathematical models alone; they'll never achieve that without augmentation. While a generative chatbot that provides answers fifty or seventy-five percent or more of the time and one that generates false answers (hallucinations) only ten or twenty percent of the time might be acceptable for some applications or users, much of the commercial, government, and scientific world, among others, will find those levels of reliability and accuracy unacceptable, even with disclaimers. Imagine a bank miscalculating account balances or accounting software misleading clients.

There's another significant reason to combine neuro-symbolic and structured knowledge with stochastic generative AI. It will be common that enterprises to develop and deploy multiple generative AI solutions

both to internal staff and to external users. For example, an enterprise might need to deploy several different chatbots that integrate with different platforms due to their differing architectures. By standardizing generative applications on a common knowledge abstraction layer, they can “think” and be consistent with one another and share common knowledge assets such as taxonomies, ontologies, and knowledge graphs.

So how do we augment generative models, and with what?

We must combine neuro-symbolic intelligence with stochastic models that result in a hybrid solution that combines the best of both. After generative AI development teams come to understand that basic truth, they’ll seek out the expertise of knowledge management professionals and come begging for their valuable semantic stacks. Some developers might even attempt an end-run to develop these assets on their own, but they’ll find out eventually that they are amateurs and lack the deep skills required to develop them properly to be effective. It’s highly unlikely you can turn a programmer into a skilled and experienced Ontologist in just a few weeks or months. *Developed improperly, the long-term damage to the enterprise cannot be understated*.

Public vs Enterprise large language models

We cannot address the full breadth of generative AI here so instead; we’re going to focus on content retrieval for a specific domain. We’re also going to assume that we’re dealing with an enterprise (captive) content corpus rather than building an LLM that is equivalent in size and scope to ChatGPT or Google Bard, nor are we going to try to train those huge public LLMs.

Fundamental concepts of neuro-symbolic knowledge

We cannot proceed without defining a few basic concepts. Let’s start with the basics of neuro-symbolic assets.

Taxonomies

In simple terms, taxonomy is the science of classifying and organizing things (not strings!) based on their characteristics and relationships. For example, it helps scientists categorize and name different species so that they can better understand the diversity of life on Earth.

Imagine you have a collection of insects that you want to organize. You start by looking at their features, such as their shape, size, color, and the number of legs they have. You notice that some insects have wings while others don't. Based on these observations, you can start grouping them into distinct categories.

For example, you might create a category called "Beetles" for insects that have a hard shell-like covering on their wings. Another category could be "Butterflies" for insects with colorful wings that are covered in tiny scales. You can further divide the beetles into subcategories based on their size or color.

Each category and subcategory you create forms a level in the taxonomy hierarchy. The highest level is usually the kingdom, such as the animal kingdom, which includes all animals. From there, you move to lower levels like phylum, class, order, family, genus, and finally, species. Each level narrows down the classification and helps identify the unique characteristics of a particular group of organisms.

Ontologies

An ontology is a way to organize knowledge and information about a particular domain or subject. It helps define the relationships between different concepts and provides a structured framework for understanding and representing knowledge.

Think of an ontology as a map or a blueprint for a specific area of knowledge. Just like a map helps you navigate and understand the layout of a city, ontology helps us navigate and understand the concepts, properties, and relationships within a particular field.

For example, let's consider an ontology about animals. It would define various concepts like "mammal," "bird," and "reptile," as well as their properties and relationships. The ontology might state that mammals give birth to live young, while birds lay eggs. It might also define properties like "has fur" for mammals or "has feathers" for birds.

The ontology would establish relationships between these concepts, such as stating that mammals and birds are both types of animals but have distinct characteristics. It might also define hierarchies, like categorizing lions, tigers, and cheetahs as subcategories of the broader concept of "big cats."

By using an ontology, we can organize information in a structured and consistent way, allowing us to reason, search, and retrieve relevant knowledge more effectively. Ontologies are commonly used in various fields, including computer science, artificial intelligence, biology, and many others, to represent and analyze complex domains of knowledge.

Taxonomies and ontologies: How they differ and how they work together

The main difference between taxonomy and ontology lies in their purpose and level of complexity.

Taxonomy: A taxonomy is primarily concerned with classifying and categorizing things based on their observable characteristics. It focuses on organizing and grouping items into hierarchical categories based on their similarities and differences. Taxonomy is often used to classify living organisms, such as plants or animals, but it can also be applied to other domains. For example, a taxonomy of cars might categorize them based on factors like size, make, or fuel type. Taxonomies provide a way to organize information for easier navigation and retrieval.

Ontology: An ontology, on the other hand, goes beyond simple categorization and aims to represent the relationships and properties of concepts within a specific domain. It provides a more detailed and structured representation of knowledge, capturing not only the classifications but also the relationships, attributes, and constraints associated with the concepts. Ontologies are often used in fields like computer science, where they help in knowledge representation and reasoning. For example, an ontology about cars might define the relationships between car parts, the properties of each part, and how they interact with each other.

A taxonomy and an ontology for a specific domain, such as sales tax compliance or a company's product matrix, are best defined together where the ontology defines the classes, super-classes, and sub-classes that categorize and disambiguate the taxonomy labels. By doing so, when we query a knowledge graph for practical applications, the system can distinguish whether we're talking about "windows" as a thing related to a building or "windows" as software.

In summary, a taxonomy is a basic classification system that organizes items into hierarchical categories based on observable characteristics, while an ontology is a more sophisticated representation that not only categorizes but also defines the relationships, properties, and constraints of concepts within a domain. Taxonomies are simpler and focus on categorization, while ontologies provide a more comprehensive understanding of a subject area by capturing complex relationships and knowledge representation.

Knowledge Graphs

A knowledge graph is a way to organize and represent information by connecting different pieces of knowledge. It consists of nodes (representing entities or concepts) and edges (representing relationships between those entities). This structure allows us to see how different pieces of information are related and provides a comprehensive understanding of a particular domain.

Imagine you want to build a knowledge graph about famous scientists. You start by identifying the scientists as nodes and connecting them with edges that represent their relationships. For example, you might connect Isaac Newton and Albert Einstein with an edge labeled "Influenced by" to show that Einstein was influenced by Newton's work.

You can also include other types of information. Let's say you want to include their areas of expertise. You can add nodes representing scientific fields like physics, mathematics, and astronomy, and connect the scientists to their respective fields using edges labeled "Specialized in." For instance, you can connect Newton to the field of physics and Einstein to physics and mathematics.

By adding more nodes and edges, your knowledge graph starts to grow. You can link scientists to their notable discoveries, their academic institutions, or even their personal lives, creating a web of interconnected knowledge.

The power of a knowledge graph lies in its ability to show not just isolated facts but also the connections and relationships between them. It allows you to navigate through information, explore different paths, and uncover insights that might not be apparent from individual data points.

Knowledge graphs are commonly used in various domains, such as search engines, recommendation systems, and data analysis. They help organize and represent complex information in a way that is easier to understand and utilize.

How an ontology serves as the basis for creating a knowledge graph

An ontology can serve as the basis for creating a knowledge graph for a particular domain by providing a structured framework for defining and representing the concepts, relationships, and properties within that domain.

First, an ontology defines the key concepts or entities that exist within a domain, such as sales tax compliance. These concepts become the nodes in the knowledge graph. For example, in the medical domain ontology, concepts could include diseases, symptoms, medications, and medical procedures.

Next, the ontology defines the relationships between these concepts. These relationships become the edges in the knowledge graph. For example, the ontology might define relationships such as "causes," "treats," or "has symptom." These relationships connect the nodes and represent how the concepts are related to each other.

The ontology also captures the properties or attributes of each concept. These properties can be represented as additional information associated with the nodes in the knowledge graph. For example, a disease concept may have properties like "name," "description," and "prevalence."

Once the ontology is defined, the knowledge graph can be created by instantiating the concepts as nodes and connecting them with edges based on the defined relationships. The resulting knowledge graph represents the interconnectedness and structure of the domain's knowledge.

With the knowledge graph in place, it becomes a powerful tool for organizing, navigating, and retrieving information within the domain. It allows for efficient querying, data analysis, and exploration of the relationships between different concepts. The knowledge graph enables a more holistic understanding of the domain's knowledge by capturing the interconnections and dependencies among various concepts and providing a visual representation of the domain's information.

In summary, an ontology serves as the foundation for creating a knowledge graph by defining the concepts, relationships, and properties within a domain. The resulting knowledge graph represents the structured and interconnected knowledge of that domain, facilitating efficient information organization, exploration, and analysis.

Employing and querying a knowledge graph

Whereas taxonomy is useful for tagging content, an ontology is useful as a blueprint and for mapping real-world content and disambiguation. It does this by creating a mapping to the content by assigning concrete universal resource locators to actual content objects. Many people hear the word “graph”, and they mistakenly think it’s only a conceptual visual rendering of things. Yes, it does provide visual renderings, but more importantly, it is more akin to a minable super-index, on steroids using specialized graph query languages that can be used to provide insights, recommend, and retrieve content that traditional methods cannot. This is not your father’s relational database technology nor master data management that can operate at the performance levels and capability provided by a knowledge graph.

A knowledge graph can be used as a basis for creating powerful and accurate chatbots without the use of generative AI. Its limitation is that it returns specific content components based on concrete facts and intent in the form of declared relationships and ensures referential integrity, and thus, provides explainable AI, whereas generative AI based on stochastic models is wonderful at generating text from multiple sources but lacks the referential integrity and thus, the degree of reliability and trustworthiness required for critical business applications where accuracy and precision matters. Imagine combining the power of both!

So how do we create useful applications using a knowledge graph? We generate the graph based on the subject domain’s ontology in a graph database and query the model using a special kind of query language called a graph query language. A popular open-standard graph query language is SPARQL.

A graph query language is a specialized language used to retrieve and manipulate data stored in a graph database, such as Ontotext’s GraphDB It provides a way to express queries that traverse the relationships between nodes in a graph and retrieve the desired information.

As noted earlier, graph databases organize data in a graph structure, consisting of nodes (also known as vertices) and edges (also known as relationships) that connect the nodes. Each node represents an entity, and the edges represent the relationships or connections between entities.

Graph query languages allow users to perform complex queries on graph databases by specifying patterns of nodes and relationships they are interested in. These languages provide a concise and expressive syntax to navigate the graph and retrieve data based on various conditions and filters.

There are several graph query languages available, each with its unique syntax and features. Some popular graph query languages include:

- **SPARQL** (pronounced "sparkle") is a query language specifically designed for querying data stored in RDF (Resource Description Framework) format. RDF is a widely used graph-based data model for representing structured information on the web and is also an open standard.
- **Cypher**: Cypher is a proprietary query language specifically designed for Neo4j, a popular graph database. It uses ASCII art-like syntax to express patterns and relationships between nodes.
- GraphQL: GraphQL is a query language and runtime for APIs. While it is not exclusively designed for graph databases, it can be used to query graph databases by leveraging its type system and graph-like querying capabilities.
- **Gremlin**: Gremlin is a graph traversal language that supports a wide range of graph databases. It provides a functional and fluent syntax to navigate the graph and perform complex traversals and manipulations.

SPARQL

As we said, SPARQL is an open standard query language specifically designed for querying data stored in RDF (Resource Description Framework) format. RDF is a widely used graph-based data model for representing structured information on the web.

SPARQL stands for "SPARQL Protocol and RDF Query Language." It provides a standardized way to query and manipulate RDF data. SPARQL allows users to express queries that retrieve specific data patterns, perform aggregations, filter results, and navigate the relationships between RDF resources.

Here are some key features and concepts of SPARQL:

- Querying RDF Data: SPARQL enables users to query RDF data by specifying patterns using triple patterns (subject-predicate-object) to match against the data graph.
- Pattern Matching: SPARQL queries can contain triple patterns with variables, allowing for pattern matching against the data graph. Results can be bound to these variables and used in subsequent parts of the query.
- Graph Pattern Matching: SPARQL allows for the combination of triple patterns into graph patterns, enabling the specification of more complex query conditions involving multiple triples and relationships.
- Filtering and Constraints: SPARQL supports filtering and constraints to refine query results based on specific conditions. It provides operators and functions for comparing values, checking for containment, performing arithmetic operations, and more.
- Aggregation and Grouping: SPARQL allows for grouping and aggregating query results using functions like COUNT, SUM, AVG, MIN, MAX, and GROUP BY. This enables statistical analysis and summary of data.

- Modularity and Named Graphs: SPARQL supports modular queries by allowing users to specify named graphs within a dataset. Named graphs provide a way to organize and separate data into distinct graphs that can be queried independently or together.
- Protocol and Results Format: SPARQL includes a protocol specification that defines how clients can interact with SPARQL endpoints to execute queries remotely. The query results can be returned in various formats such as XML, JSON, or CSV.

SPARQL has become a widely adopted standard for querying and working with RDF data. It provides a powerful and expressive language for retrieving information from RDF graphs and has been instrumental in the development of the Semantic Web.

Graph query languages allow developers and users to retrieve specific information from graph databases efficiently. They simplify the process of querying graph data and make it easier to work with the complex relationships present in graph structures.

Fundamental concepts of stochastic generative AI

Let's revisit the basics of stochastic AI models which are the most prevalent in generative AI, including ChatGPT.

Vectors

In large language models, such as ChatGPT, a vector refers to a mathematical representation of data that consists of a sequence of values, often arranged in a one-dimensional array. In the case of language models like ChatGPT, vectors are used to represent various elements, such as words, sentences, or entire documents.

ChatGPT itself is based on a deep learning model called GPT (Generative Pre-trained Transformer), which utilizes a transformer architecture. The model processes input text by breaking it down into tokens, with each token typically corresponding to a word or sub-word unit. These tokens are then embedded into continuous vector representations, often referred to as word embeddings or token embeddings.

The embeddings allow the model to capture semantic and contextual information of the input text. These vectors are then processed through multiple layers of neural networks within the transformer model, enabling the model to understand and generate meaningful responses based on the input it receives.

So, in ChatGPT, vectors are used to represent tokens or units of text within the model, and they play a crucial role in the underlying deep-learning processes that enable the model to generate coherent and contextually relevant responses.

ChatGPT, like other transformer-based models, operates at the token level rather than directly creating vectors for individual words. It takes input text and tokenizes it, breaking it down into smaller units called tokens. Tokens can represent words, sub-words, or even characters, depending on the tokenization scheme used.

The model then generates vector representations, often referred to as token embeddings or word embeddings, for each token in the input. These embeddings capture the semantic and contextual information associated with the corresponding token.

So, rather than directly creating vectors for individual words, ChatGPT creates vectors for tokens that represent words or sub-word units. The tokens act as the basic units of input and are embedded into continuous vector representations, allowing the model to process and understand the input text.

It's important to note that tokenization and vectorization happen simultaneously within the model. The model learns to generate meaningful embeddings for tokens during the training process, capturing the semantic relationships and contextual information associated with the tokens in the input sequence.

Cosine similarity

Cosine similarity is a metric used to measure the similarity between two vectors in a multi-dimensional space. It determines the cosine of the angle between the vectors, providing a value between -1 and 1. A value of 1 indicates that the vectors are identical, while a value of -1 indicates that they are diametrically opposed. A value of 0 suggests no similarity.

In ChatGPT and similar models, cosine similarity is not directly used as part of the text generation or token selection process. It is primarily used as a mathematical tool to measure the similarity between different vector representations, such as word embeddings or contextualized token embeddings.

During the training process of ChatGPT, the model learns to encode words or tokens into vector representations that capture their semantic and contextual information. These vector representations can be compared using cosine similarity to measure their similarity or relatedness.

Cosine similarity can be utilized in downstream tasks or post-processing steps related to text generation. For example:

1. Nearest Neighbor Search: Cosine similarity can be used to find tokens or embeddings that are most similar to a given target vector. This can be useful for finding similar words or phrases in a word embedding space.
2. Contextual Similarity: Cosine similarity can be employed to compare the similarity of different contextualized token embeddings within the same sentence or document. This can help identify the most relevant tokens based on their similarity to a reference context.

It's important to note that cosine similarity is not an inherent component of the core text generation or token selection process in ChatGPT. Rather, it is a mathematical tool that can be utilized as needed in specific use cases or downstream tasks involving vector-based operations.

Vector selection during text generation

When generating text, ChatGPT does not select the "next best vector" directly. Instead, it utilizes a decoding process to generate the next token based on the context provided. The decoding process involves probabilistically sampling or determining the token to append to the current sequence.

During text generation, ChatGPT employs a technique called autoregression, where it predicts the next token based on the preceding context. The model takes the previously generated tokens as input and

uses its internal representations and learned parameters to compute the probability distribution over the vocabulary for the next token.

The probability distribution represents the model's confidence or likelihood for each token to be the next one in the sequence. The selection of the next token is typically performed using one of the following approaches:

1. Greedy Sampling: The model selects the token with the highest probability as the next one. This approach favors tokens that are most likely according to the model's distribution, but it can lead to repetitive or overly predictable output.
2. Top-k Sampling: The model considers only the top-k most probable tokens, where k is a predetermined number. It samples from this reduced set of tokens based on their probabilities. This approach introduces some randomness and allows for more diversity in the generated output.
3. Temperature Sampling: A temperature parameter is introduced to control the randomness of the token selection. Higher temperatures increase randomness, resulting in more diverse output, while lower temperatures make the selection more focused on high-probability tokens.

By using one of these methods, ChatGPT selects the next token during text generation based on the probability distribution over the vocabulary. This process is repeated iteratively to generate subsequent tokens and produce coherent, contextually relevant responses or text.

Vectors that represent sentences and entire documents

ChatGPT represents an entire sentence or document as a vector by combining the individual token embeddings into a single representation. In the case of GPT and similar transformer models, this is typically done using an operation called positional encoding.

Positional encoding is a technique that adds position-specific information to the token embeddings to capture the sequential order of the tokens in the input text. It allows the model to differentiate between different positions in the sentence or document.

Once the tokens are embedded and positional encoding is applied, the model's transformer architecture processes these representations through multiple layers of self-attention and feed-forward neural networks. This processing allows the model to capture the contextual relationships between the tokens, considering both the local context and the global context of the input.

As the input text is processed through the layers of the model, the final output of the model is obtained, typically referred to as the context vector or representation. This context vector represents the entire sentence or document as a single vector in a high-dimensional space, capturing the learned semantic and contextual information from the input.

It's important to note that the representation of an entire sentence or document as a vector is a result of the training process of ChatGPT, where the model learns to encode the input text into meaningful representations through exposure to substantial amounts of training data.

Embedding versus fine-tuning

Embeddings are not the same as fine-tuning. Let's clarify each concept.

Embeddings: In the context of natural language processing (NLP), embeddings refer to vector representations of words, sentences, or documents in a continuous vector space. These embeddings capture semantic and syntactic relationships between different elements of language. Popular embedding models like Word2Vec, GloVe, and fastText are pre-trained on large corpora and provide distributed representations of words.

End users cannot directly create embeddings using ChatGPT. The embeddings used by ChatGPT are pre-trained during the model's training phase, which involves a large-scale language modeling task on a vast corpus of text data. The training process requires significant computational resources and expertise to train a language model like ChatGPT from scratch.

However, end users can make use of the pre-existing embeddings in ChatGPT to enhance natural language processing tasks. ChatGPT can provide contextual embeddings for the text you provide as prompts. By extracting and utilizing these embeddings, you can perform downstream tasks such as sentiment analysis, clustering, or similarity calculations.

To extract embeddings from ChatGPT, you can use the hidden states of the model. During inference, each word in the input sequence has a corresponding hidden state in the model, which contains contextual information about the word's meaning within the given context. You can extract these hidden states from the model's intermediate layers and use them as embeddings for downstream tasks.

It's worth noting that extracting embeddings from ChatGPT in this manner requires additional programming and integration with the model. You would need to use the model's API and process the response to access the hidden states. Alternatively, you can explore other pre-trained models or libraries specifically designed for embedding generation, such as Word2Vec, GloVe, or Universal Sentence Encoder, which provide more straightforward ways to create embeddings.

Fine-tuning:

Note: *As of May 2023, fine-tuning is supported for versions of ChatGPT up to and including the OpenAI Davinci model. Fine-tuning OpenAPI is not yet supported for ChatGPT-4. However, it is not clear how the use of such prefixes might or might not help create more effective embeddings if they are injected into the content source before initial ingestion (that is, during unsupervised training) and should be explored.*

Fine-tuning is a technique used in machine learning, particularly in transfer learning scenarios. It involves taking a pre-trained model (such as a deep neural network) and further training it on a new dataset or a specific task. The pre-trained model serves as a starting point, and the parameters of the model are updated or fine-tuned using the new data to adapt it to the specific task or domain.

When it comes to NLP tasks, fine-tuning is commonly applied to pre-trained language models like BERT, GPT, or ELMO. These models are trained on large-scale language modeling objectives and capture rich contextual information. Fine-tuning involves taking a pre-trained language model and training it on a specific downstream task, such as sentiment analysis or text classification, by updating its parameters with task-specific data.

Embeddings can be used as features for downstream tasks, including fine-tuning. In fine-tuning, the embeddings learned during pre-training are typically used as the input representation, and the model's parameters are updated based on the specific task's labeled data. The embedding layer itself is not modified during fine-tuning, but rather the deeper layers of the model are adapted to the new task.

Embeddings are a general representation of words or texts, while fine-tuning is a specific technique used to adapt pre-trained models to new tasks. Embeddings can be used as part of the input to a fine-tuned model, but they are not the same as the fine-tuning process itself.

Embeddings

Embeddings are special types of vectors used in machine learning and natural language processing (NLP). An embedding is a dense vector representation of a word, phrase, or document, where each dimension of the vector captures a particular feature or meaning of the input. Embeddings are typically learned through an unsupervised learning process, such as word2vec or GloVe, which aim to capture semantic and syntactic relationships between words based on their co-occurrence patterns in a large corpus of text.

Word2Vec and GloVe are methods for generating word embeddings, which are dense vector representations of words. While these embeddings can be used in various natural language processing tasks, including tasks like text classification or word similarity, they are not directly compatible with ChatGPT.

The model underlying ChatGPT 3.5 and ChatGPT 4 already has its own pre-trained embeddings. These embeddings are learned during the model's pre-training phase, where it is trained on a large corpus of text to learn contextual representations of words and phrases. These contextual embeddings capture the meaning and relationships between words in a given context.

When using ChatGPT, you interact with the model using text prompts, and it generates responses based on its understanding of the provided context. The model internally utilizes its pre-trained embeddings to process the input and generate coherent and relevant outputs.

While you cannot directly use Word2Vec or GloVe embeddings with ChatGPT, you can leverage their embeddings in preprocessing steps or as additional features in tasks surrounding ChatGPT. For example, you could use Word2Vec or GloVe embeddings to compute similarities between words, and then incorporate that information into the prompts you provide to ChatGPT.

Embeddings are useful because they can capture the meaning and context of words in a continuous vector space, enabling algorithms to perform computations and comparisons on words or documents. They have become a fundamental technique in NLP tasks, such as language modeling, sentiment analysis, machine translation, and information retrieval.

It's worth noting that while word embeddings are a common type of embeddings, there are also other types of embeddings used for different purposes. For example, image embeddings can be learned to represent images as dense vectors, allowing similarity comparisons or input into machine learning models for tasks like image classification or object detection.

Concept embeddings

Concept embeddings can be considered a special type of vector representation. Concept embeddings capture the semantic meaning of concepts or entities in a continuous vector space. They are often learned through techniques like word2vec, GloVe, or other embedding methods.

Concept embeddings differ from word embeddings in that they represent higher-level concepts or entities rather than individual words. For example, instead of representing each word in a sentence with word embedding, concept embeddings aim to capture the meaning of the entire sentence or a whole document.

Concept embeddings are particularly useful in tasks such as document classification, sentiment analysis, information retrieval, or recommendation systems. By representing concepts or entities as vectors in a continuous space, it becomes possible to compute similarity, perform clustering, or apply machine learning algorithms to these representations.

Overall, concept embeddings provide a way to encode and manipulate the semantic relationships between concepts, enabling more advanced analysis and processing of textual or conceptual data.

Creation of concept embeddings

The direct ingestion of triples can be used to create concept embeddings in LLMs.

A triple typically consists of three elements: a subject, a predicate, and an object. These elements represent entities and their relationships in a structured format. For example, a triple could be "Albert Einstein - was born in - Ulm, Germany".

You can preprocess the input triples and convert them into suitable input formats for an LLM. This may involve representing the triples as input sentences or finding appropriate ways to encode the subject, predicate, and object information.

By training an LLM with the triple-based input data, the model can learn to associate the entities and relationships within the triples. The learned representations, or concept embeddings, can then capture the contextual information and associations related to the concepts present in the triples.

These concept embeddings can subsequently be used to enhance the model's understanding and generation of responses related to the concepts represented in the input triples.

It's important to note that preparing the triples for fine-tuning and designing an appropriate encoding scheme may require careful consideration and experimentation to ensure effective utilization of the triple-based information within an LLM's training process.

Methods for representing triples during ingestion

When representing triples during ingestion for fine-tuning ChatGPT 3.5 and earlier, there are several methods you can consider. The choice of method depends on the specific requirements of your task and the nature of the triples you are working with. Here are a few commonly used approaches:

1. Concatenation: You can concatenate the subject, predicate, and object of a triple into a single string, separated by special tokens or delimiters. For example, "Albert Einstein | was born in |

Ulm, Germany". This approach allows the model to process the entire triple as a single input sequence.

2. Special Tokens: You can assign special tokens to represent the subject, predicate, and object in the triple. For instance, you could use "[SUBJ]" to represent the subject, "[PRED]" for the predicate, and "[OBJ]" for the object. This allows the model to identify and differentiate the different components of the triple.
3. Positional Encoding: You can incorporate positional encoding to indicate the position or role of each element in the triple. For example, you could add a prefix or suffix to the subject, predicate, and object to indicate their positions. For instance, "subj_Albert Einstein | pred_was born in | obj_Ulm, Germany".
4. Entity Markers: If your triples involve named entities, you can use entity markers to highlight the entities in the triple. For example, you could use "@entity1@" and "@entity2@" to represent the subject and object entities, respectively. This helps the model recognize and focus on the entities in the triple.
5. Separate Input Streams: Instead of representing the entire triple as a single sequence, you can use separate input streams for the subject, predicate, and object. This involves providing each component as a separate input to the model, enabling it to process and attend to the different elements independently.

The choice of representation method depends on the complexity of the triples, the desired level of granularity, and the downstream tasks you are targeting. It is recommended to experiment with different approaches and evaluate their effectiveness in capturing the desired relationships and generating appropriate responses.

LLMs that support custom embeddings

As was previously mentioned, language models like GPT-3.5 and its variants, including GPT-4, do not inherently support the direct usage of user-created Word2Vec or GloVe embeddings. These models have their own pre-trained embeddings learned during the training process, and they rely on those embeddings for generating responses based on the provided context. However, there are other language models and frameworks that allow for the integration of Word2Vec or GloVe embeddings. Some examples include:

FastText: FastText is a library developed by Facebook AI Research that supports training and utilizing word embeddings, including Word2Vec and FastText embeddings. You can use FastText to train word embeddings and incorporate them into your language modeling or text processing pipeline.

AllenNLP: AllenNLP is an open-source NLP library that provides various tools and models for natural language processing tasks. It supports the usage of pre-trained Word2Vec and GloVe embeddings within its models, allowing you to use these embeddings for tasks like text classification, named entity recognition, and more.

TensorFlow and **PyTorch**: TensorFlow and PyTorch are popular deep-learning frameworks that offer flexibility in designing and training custom language models. You can incorporate Word2Vec or GloVe embeddings into your models built with these frameworks, either by using pre-trained embeddings or by training your own embeddings from scratch.

You should check whether the LLM you want to use supports custom embeddings. By leveraging these frameworks and libraries, you can create your own language models that support the usage of Word2Vec

or GloVe embeddings. Keep in mind that training and utilizing custom embeddings may require additional resources and expertise in NLP and deep learning.

Prompt Prefixes

***Note**: Prompt prefixes are typically used for fine-tuning ChatGPT and other LLMs. As of May 2023, fine-tuning is supported for versions of ChatGPT up to and including the OpenAI Davinci model. Fine-tuning OpenAPl is not yet supported for ChatGPT-4. However, it is not clear how the use of such prefixes might or might not help create more effective embeddings if they are injected into the content source before initial ingestion (that is, during unsupervised training) and should be explored.*

A prompt prefix is a specific sequence of characters or tokens that is added to the beginning of an input text to guide the behavior or context of a language model. It serves as an instruction or cue to the model about the desired format or type of response to generate.

Using a prefix when fine-tuning ChatGPT can be beneficial as it allows you to provide specific context or instructions to guide the model's responses. The prefix serves as an initial input to the model, helping to set the desired tone or topic for the conversation.

By including a prefix, you can influence the model's behavior and steer it toward generating responses that align with your requirements. For example, if you want to fine-tune the model to provide support for a particular product or service, you could include a prefix that specifies the context and sets the expectation for the model to provide helpful information related to that product or service.

Prompt prefixes are commonly used in text generation tasks, such as question-answering, dialogue systems, or language translation, to provide context or constraints to the model. They help guide the model to generate responses that align with the given prompt.

For example, consider a question-answering task where the goal is to provide answers to questions. A prompt prefix could be "Question:" followed by the actual question. By using this prompt prefix consistently, the model learns to understand that the input text following "Question:" expects an answer to be generated.

Similarly, in a dialogue system, a prompt prefix could be "User:" followed by the user's query or statement. This helps the model recognize the user's input and generate appropriate responses.

The choice and format of the prompt prefix depend on the specific task and requirements of the application. It can be a simple keyword, a token, or a more elaborate string that provides explicit guidance or context to the model. Prompt prefixes are introduced during fine-tuning to establish the desired relationship between the prompt and the generated responses.

ChatGPT does not have an inherent understanding of prompt prefixes. The model is typically trained on enormous amounts of text data without specific information about prompt prefixes.

Prompt prefixes are commonly used to provide guidance or context to the model by specifying the format or type of input expected. For example, a prompt prefix could indicate that a question is being asked or a command is being given.

You can incorporate prompt prefixes in the training data to help the model learn to respond appropriately to specific types of inputs. By consistently providing prompt prefixes in the training examples, you can guide the model to understand and generate responses that align with the desired prompts.

However, it's important to note that ChatGPT itself does not inherently understand or interpret the prompt prefixes. The model treats the entire input sequence, including the prefix, as a continuous piece of text. It learns to generate responses based on the patterns and associations it discovers during training.

Therefore, while prompt prefixes can be a useful technique, it is the responsibility of the training process and data preparation to establish the desired relationship between prompt prefixes and the expected model responses.

To identify RDF triples, you can use a specific prompt prefix that indicates the presence of triples and helps the model understand the desired behavior. Here are a few examples of prompt prefixes that can be used:

1. "Triple:" Example: "Triple: Albert Einstein was born in Ulm, Germany."
2. "Subject-Predicate-Object:" Example: "Subject-Predicate-Object: Albert Einstein - was born in - Ulm, Germany."
3. "RDF:" Example: "RDF: Albert Einstein was born in Ulm, Germany."

These prompt prefixes explicitly indicate that the following text contains RDF triples. By consistently using the chosen prompt prefix in the training data, you can guide the model to understand and generate responses related to RDF triples.

It's important to note that the choice of a prompt prefix is flexible, and you can adapt it based on your specific use case or preference. The key is to be consistent and ensure that the model can recognize and associate the prompt prefix with RDF triples.

Improving the accuracy and completeness of generative chatbot answers with an ontology

One might logically ask; can we improve the accuracy of responses generated by ChatGPT ad other large language models by fine-tuning the LLM with a domain-specific ontology and also tagging the content before ingestion?

Integrating an ontology and fine-tuning ChatGPT using RDF data from the ontology can improve the accuracy of answers generated by the model. The ontology provides structured information about concepts, relationships, and domain-specific knowledge. RDF data helps the model learn from specific instances and relationships encoded in the ontology.

By pre-tagging the content with ontology information, you provide explicit cues to the model about the underlying concepts and relationships. This helps the model to better understand the context and generate more accurate answers that align with the ontology.

Ingesting RDF data from the ontology further enhances the model's understanding of the specific relationships and instances represented in the ontology. This can enable the model to generate more accurate and contextually relevant responses related to the ontology concepts.

However, it's important to note that the accuracy of the generated answers also depends on several factors, including the quality and completeness of the ontology, the size and representativeness of the RDF data used, and the complexity of the questions or queries posed to the model.

Integrating an ontology and fine-tuning ChatGPT with RDF data can be beneficial, but it's still essential to evaluate and validate the model's performance in your specific use case to ensure the generated answers meet the desired accuracy and quality standards.

Prompt Injection

Prompt injection in ChatGPT refers to the technique of embedding a predefined prompt or instruction into the input provided to the model during conversation generation. It involves incorporating a prompt at the beginning of the conversation to guide the model's responses.

By injecting a prompt, you can influence the behavior of ChatGPT and steer it toward generating responses that align with the desired context or instructions. The prompt can provide specific guidance on the format, content, or tone of the response you expect from the model.

For example, if you want to instruct ChatGPT to provide a summary of a given text, you can inject a prompt like "Summarize the following text:" followed by the text you want to be summarized. This informs the model about the intended task and prompts it to generate a summary accordingly.

Prompt injection is often used in conjunction with fine-tuning to further customize the model's behavior and make it more suitable for specific tasks or domains. By including relevant prompts during fine-tuning, you can encourage the model to generate responses that adhere to the desired guidelines or requirements.

However, it's important to note that prompt injection is just one technique among several methods for influencing the model's behavior. The effectiveness of prompt injection depends on the task, the quality of the prompt, and the training data used to fine-tune the model. Experimentation and iteration are crucial to finding the most effective prompts for a given scenario.

Embedding prompt prefixes to structured text to improve unsupervised training

Prompt prefixes can be added to the structured text to improve unsupervised training. By incorporating prompt prefixes, you can provide explicit instructions or context during ingestion to guide the model's understanding and generation of text.

When working with structured text, such as tables, lists, or other formatted data, you can use prompt prefixes to inform the model about the desired output format or task. For example, if you want the model to generate a description for each item in a table, you can prepend a prompt like "Generate a description for each item in the table below:" before presenting the structured data.

By including these prompts, you help the model understand the expected task and encourage it to generate text that aligns with your requirements. It can facilitate the generation of coherent and structured responses, enhancing the quality of the unsupervised training process.

Prompt prefixes in the structured text can be particularly useful when fine-tuning models or training them for specific domains or tasks. They help in conditioning the model to generate text that follows the desired structure or format, improving its ability to handle structured data effectively.

Fusing neuro-symbolic knowledge with stochastic generative AI

To succeed, we must infuse our large language model with neuro-symbolic intelligence to the greatest degree possible. We also must apply the power of neuro-symbolic during the training and use of our LLMs for maximum effectiveness. There are several methods to accomplish these goals at various stages that we'll now break down.

Deployment patterns

There are several model variants (deployment patterns) that should be considered when fusing neuro-symbolic knowledge with stochastic generative AI given a controlled corpus of content (what we've termed "captive" content), such as a domain-specific knowledge base. Note that these models and methods are not mutually exclusive and may produce progressively better results when combined. There are other implementation patterns used beyond LLM-augmented content retrieval and generative chatbots, such as for knowledge graph construction and other AI applications we won't cover here.

Patterns for combining knowledge graphs and generative technology

The proposed model for augmenting structured knowledge with generative AI for content search and retrieval Combines several methods and processes for a repeatable and maintainable solution.

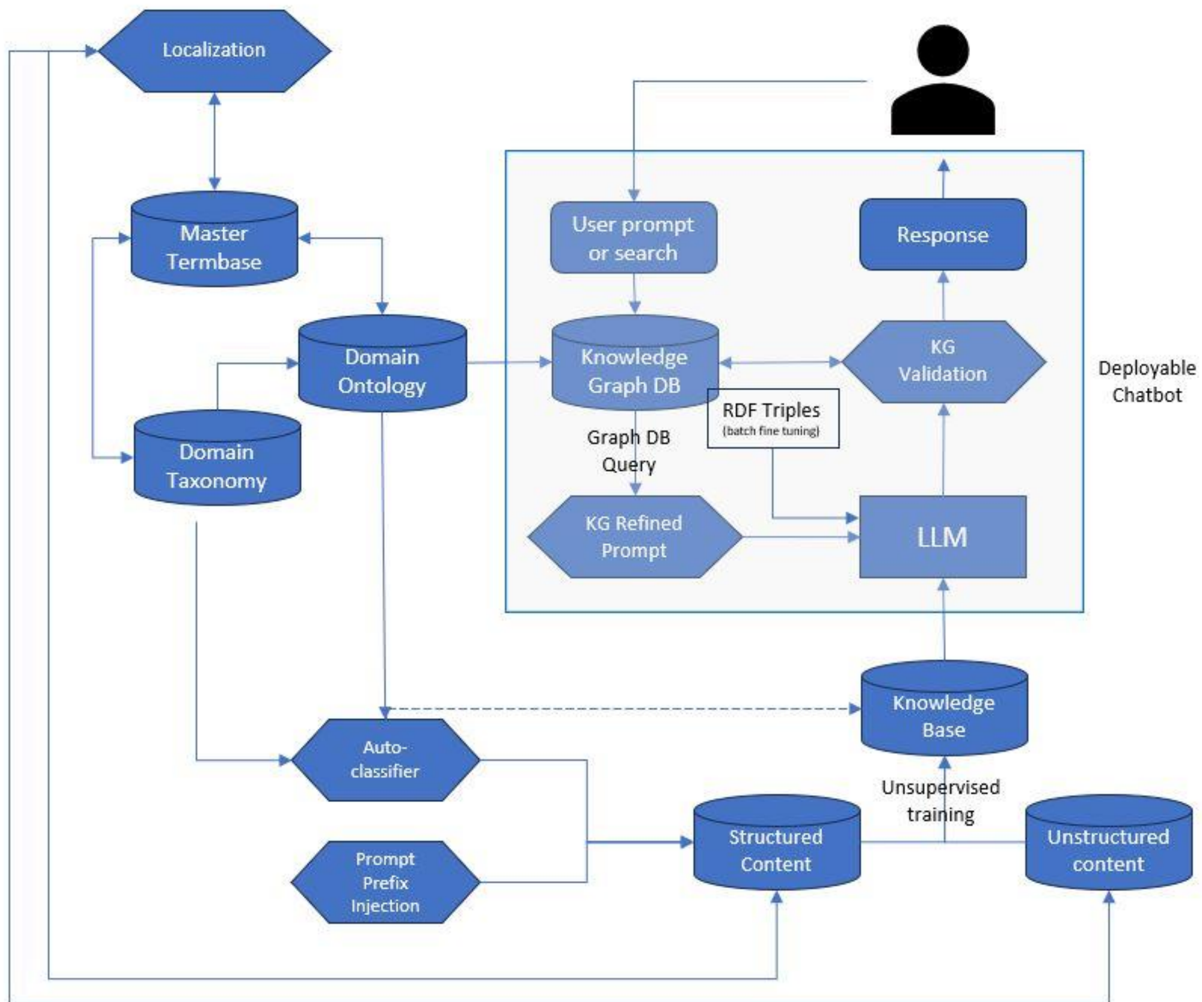

Figure 1: KG-driven generative pattern

User prompt or search argument

The user inputs a prompt or search argument as they normally would with an LLM, but the input is instead routed to the domain-curated knowledge graph.

Knowledge graph prompt refinement

Next, the user prompt is parsed, and the entities and other properties are identified and extracted. The KG query is used to interrogate the graph and identify the matching nodes, properties, and relationships. That intelligence is then used to programmatically reconstruct an improved prompt.

RDF tuned LLM

The LLM has been pre-tuned using the triples from the ontology. The LLM is then better able to map the explicitly declared concepts from the ontology to the implicitly generated concept embeddings that the LLM created upon unsupervised training during source content ingestion. The result is more effective retrieval and assembly of tokenized vectors by the LLM from which the resulting answers are generated.

Knowledge graph validation of generated content

The knowledge graph is queried after the LLM generates its response. A graph query is performed using the extracted entities from the generated content and validated against the curated knowledge. A confidence rating can also be generated and optionally provided to the user.

Curated domain ontology (enterprise)

A professionally curated domain-specific ontology is required and serves as the blueprint for pre-generating the knowledge graph from the knowledge base. The ontology not only serves to generate the RDF triples and map the nodes to addressable URIs but is also used to disambiguate entities based on concept classification. It is critical that the ontologies be consistent across all content in the enterprise, and, ideally, should span all enterprise systems and data to eventually achieve enterprise knowledge graphs (EKGs) if not initially within scope.

Curated domain taxonomies, classification, and autoclassification

Taxonomies are used for classifying (labeling) the content in the knowledge base. Labeling can be done manually by the content creators (assisted or unassisted) or automated using a content classifier (also sometimes referred to as power tagging). This is how ChatGPT was initially trained by employing people to tag subsets content before unsupervised training and ingestion.

The format of the source content matters. When ingesting monolithic content objects, such as PDFs and other binary large objects (BLOBs), or unstructured content, it is difficult, if not impractical or impossible to assign taxonomy labels at a granular level.

For example, componentized and structured content, such as content in XML or JSON format, is better suited for this task, but not to the exclusion of other formats. Content source formats such as DITA are described as being *self-describing*. That is, the content is not only highly componentized, and its type explicitly known (such as an FAQ document) but its content containers (elements) explicitly describe what the content is about, such as QUESTION, and ANSWER. The same holds especially true for DITA task topics and other purpose-specific DITA specialized topics. When the source content is componentized and self-describing, taxonomy labels can be assigned at more explicit levels within a document, such as the main document wrapper, its child topics, and self-describing blocks within topics. Doing so provides a knowledge graph with far more intelligence and granularity of nodal relationships for content interrogation and retrieval.

Also, consider that the structured and self-describing nature of DITA lends itself to the pre-injection of prompt prefixes that are embedded in the content components for initial unsupervised training and retraining (reingestion) as new content is added and existing content is updated. This is a fertile area for experimentation and testing. Again, this isn't exclusive to DITA or other self-describing and componentized content formats, but when combined with the inherent semantic intelligence of DITA results in a sort of super-intelligent source on which algorithms and generative engines can operate.

Terminology base

While novices might consider that a thesauri system that is used to create and manage taxonomies and ontologies is sufficient to use as a termbase, they'd be mistaken. Dedicated terminology platforms have necessary extended facilities to manage enterprise terminology for many downstream needs, such as language localization, and automated content governance for consistent use of terminology (the pre-

requisite for which cannot be over-emphasized). Creation and maintenance of an enterprise termbase are required. It matters little whether an organization starts by creating an initial termbase or starts with the creation of taxonomy and ontology. The latter two must be developed in tandem and interleaved with each other, then their vocabularies, definitions, and or lexical detail synchronized with the termbase.

Using a knowledge graph to perform generative pre-retrieval

Next is an alternate pattern employing content pre-fetch. The notion here is that the explicit URIs from which LLM-generated content are explicitly known, and thus, we're able to provide accessible source references to the user. While that might also be possible with the non-pre-fetch model, there is more certainty for providing referential integrity, proof and proving the user to self-validate the resulting generated responses. The downside is that a much smaller subset of the knowledge base is used for generation.

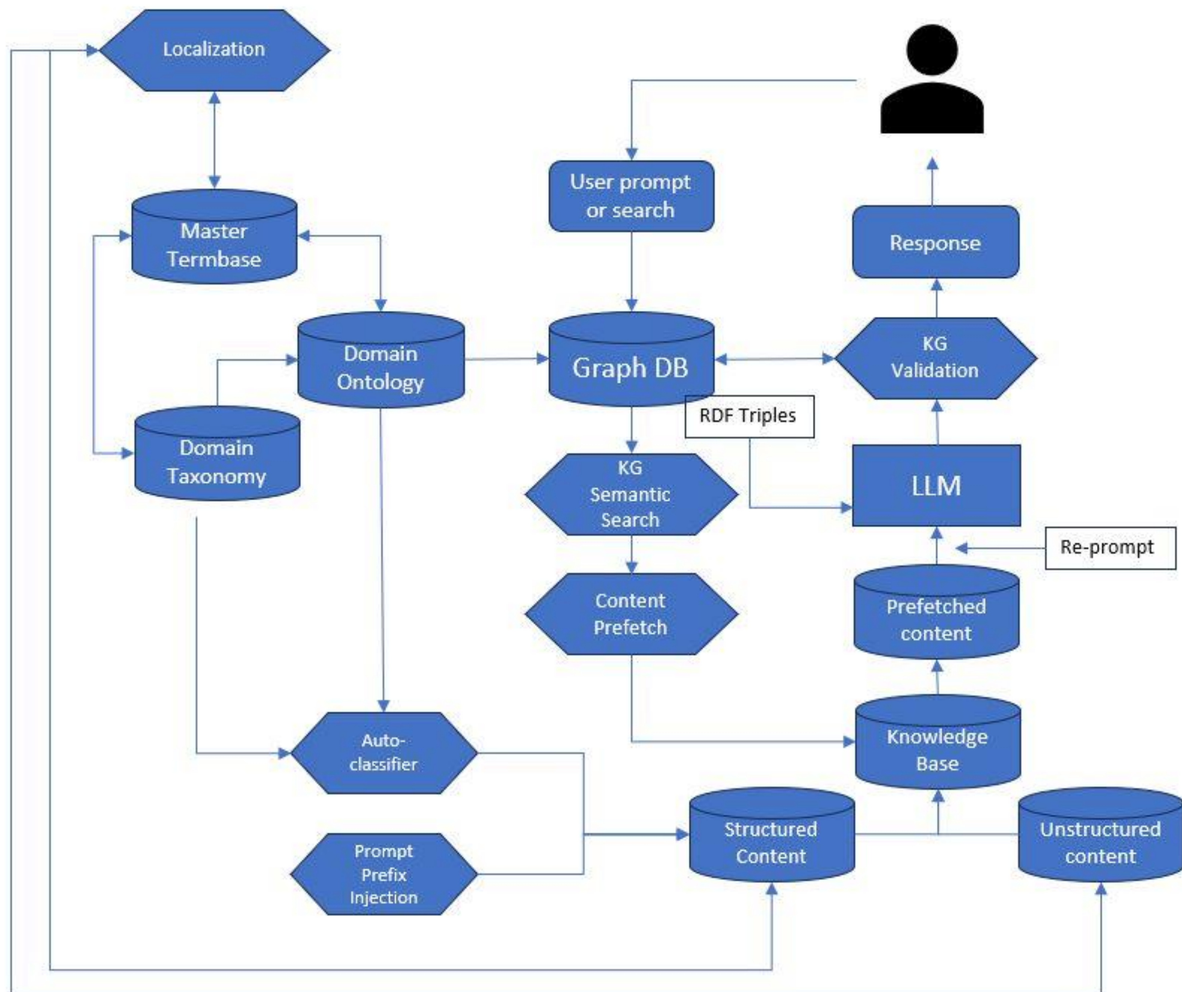

Figure 2: KG-driven generative pattern with KG pre-fetched content

Testing methodology and validating the pattern

Testing a generative model using a consistent methodology and consistent test cases is required to accurately measure the relative effectiveness and accuracy of each of the proposed models and various combinations.

1. Create a baseline assessment after initial unsupervised training without any post-ingestion tuning.
2. Create a corpus of representative questions for which valid answers are known and present in the ingested content corpus and for each answer, note the document(s) and the location(s) in which the correct answer resides.
3. Implement a pattern, then test the model with standard questions. For each question, note if the model generated an answer or could not generate an answer. If an answer was generated, note whether the answer was correct, adequate, or incorrect (a hallucination). If the answer was incorrect, record the answer. If the pattern generates answers from a pre-search, record the location/URL from which the answer was generated.
4. Rinse and repeat for each pattern.

This basic testing can be laid out in a spreadsheet format. When testing it is useful to test using subject matter experts who know the content and the subject matter well, such as technical writers for product documentation and assistance, technical support engineers or staff that write knowledge-centered support (KCS) solution articles, and so on.

Developing the required semantic stack

As discussed earlier, creating a generative chatbot is far less about traditional software development and programming logic, and far more about knowledge management. A few years ago, I proposed a model in the knowledge management community called the Semantic Content Maturity Model (SCMM). The SCMM provides a building block approach to growing an effective semantic stack.

A word of warning: Bypass the lower levels of the stack at your own peril and staggering costs to enterprise in the future. If possible, develop the stack as enterprise-wide assets. If you cannot do so, develop them for an enterprise-wide subdomain at a minimum and tie them to the larger enterprise semantic stack later. For example, if you need a semantic stack to manage content retrieval for search or a chatbot for user assistance, make every effort to define them for all user assistance content across organizational/functional business boundaries. Failure to do so is a guaranteed path to permanent content silos.

Allow me to introduce the semantic content maturity model (SCMM).

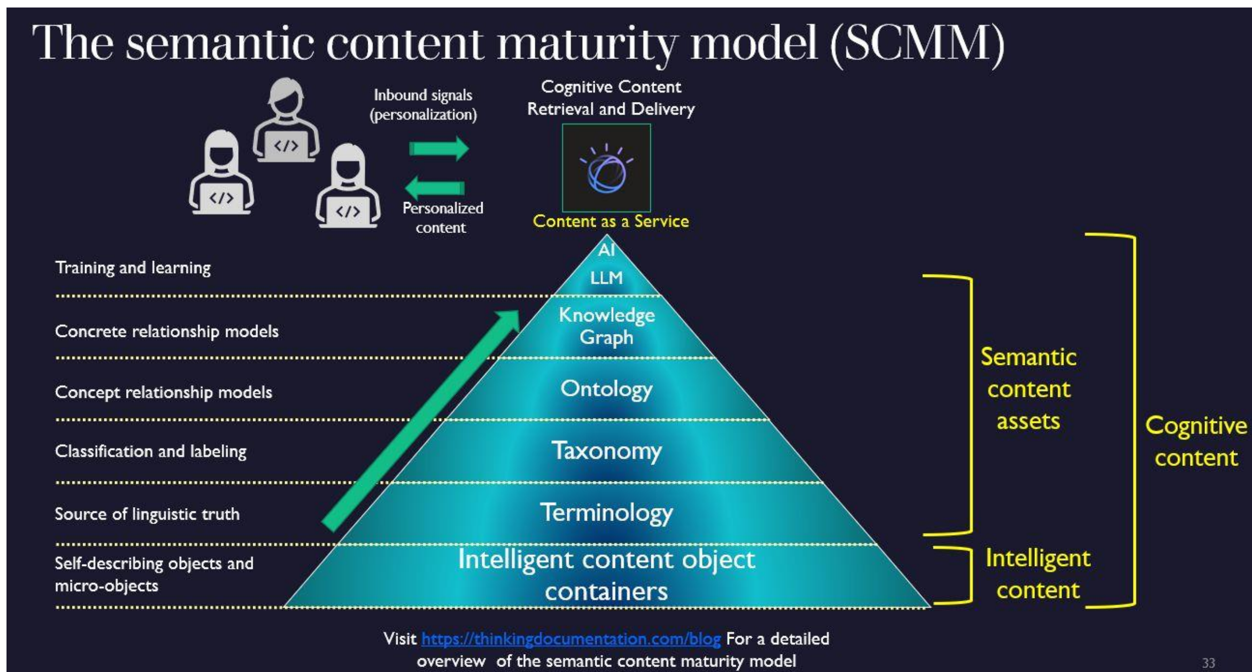

Figure 3. The semantic content maturity model

This model encapsulates the core building clocks. Each layer builds upon the next, and there's no skipping directly to Go and collecting $200.

The main elements of the semantic content maturity model include:

- Intelligent content objects containers
- Terminology
- Taxonomy
- Ontology
- Knowledge Graph

Let's explore each.

Intelligent content objects containers

Any content object can be an intelligent one. Some are natural content objects with their own inherent containers, while others take more time and effort to make them intelligent. By intelligent we mean that the content objects carry with them, or are linked to, the needed predictable semantic data structures that graph-driven applications require.

Ideally, a content object is the smallest container of semantically enriched content that has a useful purpose and one that we can identify, retrieve, reuse, and assemble into some aggregate, channel-agnostic deliverable.

Document object model (DOM) content formats are a natural and best fit for semantic-driven applications. The content is inherently encapsulated in containers and sub-containers. You don't get such encapsulation with unstructured formats such as markdown and RST, which is why such formats are

doomed as an over-arching enterprise content strategy in the era of graph-driven content applications. But can we use them anyway? Sure, just not as easily nor with the flexibility most will demand.

Even without DOM containers, algorithmically challenged formats such as these are not unlike other monolithic content, such as video clips, PowerPoints, PDFs, and many other binary large objects (BLOBs). There are a couple of ways to turn markdown, RST, and other non-DOM document formats into intelligent objects.

For example, one method is to associate the needed semantic intelligence, such as taxonomy labels, by creating another data resource that is then linked to the source content object. This is how some component content management systems do it. Another method is to add the needed semantic intelligence as CMS properties. The downside of both these methods is that semantic intelligence is not easily portable with the content. Yet another method would be to include the semantic intelligence in custom semantic structures in the content files themselves (if it's possible, such as with text-based content), but there are several major drawbacks:

- There's no universal standard for doing so with unstructured text-based content formats.
- Custom development would be needed in the CMS and/or processing routines to extract and use the metadata.
- Semantic intelligence would apply to only an entire content object and not apply to sub-objects (microcontent). Doing so would require adding self-describing structured containers – which would defeat the purpose of such lightweight formats and make them far more complex than a content architecture such as DITA.

Terminology

Semantic technologies are driven by language, but without consistent language, we're building semantic assets that equate to a house of cards.

Managed terminology is a prerequisite that must be developed before or in parallel with other semantic assets. An enterprise termbase is the source of truth for all the words used in our taxonomies, ontologies, knowledge graphs, and in the content itself.

There's no substitute for managed terminology. Thesauri management platforms are insufficient for managing terminology. Terminology management requires a system that can capture and manage the lexical complexities of language, including, but not limited to parts of speech, synonyms, multiple definitions, localized variants, preferred and deprecated variants – dozens of elements of language.

When labels are added to a taxonomy, they had better be harmonized and made consistent with the termbase, and similarly so with the use of language in ontologies and knowledge graphs. It is also of paramount importance that the content itself be consistent with terminology; that is what assistive computational linguistic services such as Acrolinx, Congree, and HyperSTE do with great efficiency.

Although such platforms can also be used as a termbase, it is advisable to use a dedicated terminology management system (TMS) from which all other tools that require managed terminology can interface.

It is essential that the content has consistent and accurate terminology. After all, the success of LLMs hangs on words, yet how many organizations have programs, processes, or tools such as Acrolinx or Congree to assist content creators govern and ensure the correct and consistent use of terminology?

Taxonomy

Machines cannot process that which they cannot identify and classify. Classifying things is a fundamental human activity. It helps us make sense of our world and the vast array of information that we encounter every day. In the field of content science, taxonomy is the practice of classifying content according to predetermined categories.

While some may see taxonomy as a dry or boring topic it is vital to our work as content professionals. By understanding how taxonomies are created and used, we can more effectively search, access, retrieve, organize, and deliver content. We can also use taxonomies to create new ways of looking at data or knowledge. Having a well-defined taxonomy for content enables us to apply semantic models such as a content ontology over our content corpus or corpora and generate knowledge graphs to power an entirely new genre of content applications that were never before feasible or possible.

Ontology

An ontology defines the relationships between concepts and objects. Concepts are defined by their relationships to other concepts, and objects are defined by their relationship to other objects.

For our purposes, an ontology provides a template for a knowledge graph, much like a schema or document-type definition does for structured content. An ontology models concepts, not actual physical content assets – that's the job of a knowledge graph that is built on an ontology as a specific instance, much like a written DITA topic is a document instance modeled after a DITA schema.

Knowledge Graph

A knowledge graph is a data structure that stores information about relationships between physical entities that represent real-world objects. It can be used to understand and extract information from large bodies of text, to find similar or related items, or to build predictive models. For our uses with content, knowledge graphs can power many new content applications.

PoolParty™ from Semantic Web Company is an easy-to-use platform for developing taxonomies, ontologies, and knowledge graphs, especially collaboratively across functional organizations, and integrates with Ontotext's GraphDB™ to form a complete semantic solution based on open standards.

Source content repositories and the content lake

There's no hard-and-fast rule that says that all the content needs that is used to train a large language model needs to reside in the same content repository. The content can be ingested (and re-ingested when adding updates and changes) from multiple repositories and formats. However, keeping the number of source repositories to a minimum makes training and updating of the LLM far more efficient and consistent.

The need for a central content delivery platform, also called a *content lake*, is a content repository that first aggregates publish-ready content from multiple, even diverse content repositories. It separates content creation from content delivery and is only concerned with content delivery where the source systems manage content creation and processes up to, but not including content delivery.

A content lake is ideal when using an augmented model that employs a knowledge graph as all the URIs are in one central repository. Also, a good content lake permits the assignment of uniform semantic labels when the originating source creation repositories cannot. Again, there is no hard-and-fast rule

that employing the use of a content lake is required, but it establishes a unification layer across heterogeneous content management systems that are typically strewn throughout an organization that cannot otherwise be consolidated due to the specialized nature and requirements of each.

About the Author

Michael Iantosca is the Senior Director of Content Platforms for user assistance at Avalara Inc. Michael spent 38 of his 40+ years at IBM as a senior enterprise content strategist and the content management platform lead. A content professional and pioneer – Michael led the design and development of multiple generations of advanced content management systems and technology that began at the very dawn of the structured content revolution in the early 80s. Dual-trained as a content professional and systems engineer, he led the charge of building some of the earliest content platforms based on structured content. Michael was responsible for forming the team at IBM that developed DITA XML – the most successful and widely adopted open standard for developing intelligent user assistance and other structured documentation.

Michael also led significant efforts in AI, starting with some of the earliest AI application development at the turn of the 90s beginning what was then called Expert Systems using Smalltalk, working directly with the scientists at IBM Watson Research Center in Hawthorn, New York developing computational linguistics and NLP technology, some of which would become core elements of the famed IBM Watson platform, co-developing computational linguistic extensions that became part of the Acrolinx content quality governance platform from Acrolinx GmbH, along with multiple published invention disclosures and patents.

The ideas and opinions discussed herein are my own and do not necessarily reflect those of my employer. Also, a subset of general concepts described herein were adapted using generative text and independently validated (why not?). As a result, this document is not copyrighted and may be freely distributed and shared.