ContextLynx - Context Aware Knowledge Management using Graphs, Embeddings and NLP

Michael A. Helcig
Student ID: 12122104

September 29, 2024

Abstract

ContextLynx is a *Personal Knowledge Management (PKM)* system designed to streamline the process of organizing and connecting personal notes without manual categorization or tagging. Using advanced natural language processing (NLP) techniques and graph Embeddings, ContextLynx automatically builds a dynamic Knowledge Graph that links ideas and topics.

This application supports various inputs such as plain text, web links, and YouTube video transcripts. It automatically identifies semantic connections between notes using local models for Named Entity Recognition (NER), word Embeddings, and API access to large language models (LLMs). Users can visualize their notes as a Knowledge Graph and receive machine learning-based predictions of relationships between them, powered by *Node2Vec*.

ContextLynx was developed as part of the "Knowledge Graph" course at **TU Wien**. The tool can be explored live at contextlynx.com/get-started, and it also supports local deployment via Docker.

Contents

1 Introduction **3**
1.1 Project Introduction . . . 3
1.2 Demo and Testing . . . 3
1.3 Source Code and Statement Regarding the Scope . . . 3

2 Learning Outcomes **4**
2.1 Representation . . . 4
2.1.1 (LO1) Knowledge Graph Embeddings . . . 4
2.1.2 (LO2) Logical Knowledge . . . 5
2.1.3 (LO4) Data Model . . . 5
2.2 Systems . . . 6
2.2.1 (LO5) Architecture . . . 6
2.2.2 (LO6) Scalable Reasoning . . . 7
2.2.3 (LO7/LO8) Creating and Evolving the Knowledge Graph . . . 8
2.3 Application . . . 9
2.3.1 (LO9) Real-World Applications . . . 9
2.3.2 (LO11) Provide Services . . . 10
2.3.3 (LO12) Connections between Knowledge Graphs (KGs), Machine Learning (ML), and Artificial Intelligence (AI) . . . 11

3 Appendix **12**
3.1 Acknowledgements . . . 12
3.2 Code . . . 12
3.2.1 Query for fetching similar Embeddings . . . 12
3.2.2 Definition of NodeVector Model . . . 12
3.2.3 Configuration of Node2Vec model . . . 13
3.2.4 Determination of predicted Edges between Notes . . . 14
3.2.5 Query for getting largest Nodes . . . 14
3.2.6 Query for getting related nodes . . . 14
3.2.7 Using Embeddings to determine similar Topics . . . 17

1 Introduction

1.1 Project Introduction

ContextLynx is a *Personal Knowledge Management (PKM)* system designed to make knowledge organization effortless, eliminating the need for manual categorization or tagging of notes. The core functionality revolves around the automatic identification of topics within notes, categorized into types such as `PERSON`, `CONCEPT`, `LOCATION`, and more. Each note is connected to its corresponding topics, which in turn can relate to one another, creating a dynamic and interconnected Knowledge Graph. Over time, the most relevant topics — those with numerous connected notes and relationships to other topics — grow larger and become more prominent as the system automatically prioritizes them. The assumption behind this design is that, as a user continues to input more notes, a few key topics will naturally dominate, representing the core focus areas of their knowledge.
To further enhance note organization, related notes are identified based on similarity. This is measured by the "distance" between notes, which is the number of topic nodes that separate them. Additionally, *Node2Vec* Embeddings are used to calculate similarities between notes, allowing the system to create "predicted" edges between the most similar notes, displaying them directly as related content.
Moreover, the topic structure is utilized to provide suggestions for topics that are already similar, enabling the Large Language Model (LLM) to generate contextually relevant associations. The overarching goal of **ContextLynx** is to make note-taking—or more precisely, *Knowledge Management*, as frictionless as possible. By removing the need of manual categorization.

1.2 Demo and Testing

For demonstration purposes, I have hosted the entire application, publicly available at contextlynx.com/get-started, of course, it can also be deployed locally with a few extra steps mentioned in the `README.md`. For the online version, a read-only user with demo knowledge was created (Username: **demo**, Password: **demo**) available at contextlynx.com/login. Furthermore, anyone is free to create an account and try around on their own. Finally, some test data is provided with the hand-in, namely a few paragraphs regarding some important figures in graph theory and some plants for home gardening. This was chosen to get two separate sub-graphs and was also chosen for the demo account.

1.3 Source Code and Statement Regarding the Scope

I have purposefully decided to keep the hand-in high-level and not delve too deeply into technical details, as it would extend beyond the scope of the project. However, I have already selected key parts to examine in greater detail. In the Learning Objectives (LO) below, I mention interesting sections of the source code. Please refer to the entire project (particularly the `core` application) for further clarification. Where necessary, I have provided paths within the project to directly show where the important code is located. I also recognize that the final outcome of the project may exceed the original scope in certain areas, particularly with the fully deployed web service. While I may have gotten carried away in some aspects, I hope this is acceptable and still aligns with the project's overall objectives. Nonetheless, I have placed the most emphasis on the KG aspects, as described below.

2 Learning Outcomes

2.1 Representation

2.1.1 (LO1) Knowledge Graph Embeddings

I exceeded basic proficiency in Knowledge Graph Embeddings by designing and implementing a system for storing and querying Knowledge Graph Embeddings using PostgreSQL's vector extension. The goal was the automatic discovery of relationships between notes based on their semantic similarity.

In the system, I applied *Node2Vec* to compute node Embeddings that capture semantic relationships between notes. Each note in **ContextLynx** is associated with topics, which form nodes in the Knowledge Graph. By calculating Embeddings for these nodes, I allowed the system to automatically identify related notes based on their vector similarity (code shown in the appendix). Notes with similar topics or context are connected via "predicted" edges that represent their closeness in the Knowledge Graph, making note discovery more intuitive. The entire workings of the node Embeddings can be seen in `core/services/node_embedding_service.py`.

Hyperparameter Selection The parameters used for *Node2Vec* model were chosen to enable efficient discovery of similar notes within clusters of nodes. The return parameter $p = 2$ encourages exploration of nodes that are farther away, helping the model to learn broader relationships between different notes. Meanwhile, the in-out parameter $q = 1$ strikes a balance between depth-first (local) and breadth-first (global) search patterns, ensuring that both local clusters of related notes and broader contexts are considered during the Embedding generation process. A walk length of 10 and 20 walks provide sufficient context without being computationally expensive, and a window size of 3 captures relationships within smaller neighborhoods of the graph. A relatively low vector space (16) was chose, since individual PKM graphs are small by nature and don't require a massive space, to be appropriately represented.

To ensure good performance and scalability of the system, I utilized PostgreSQL's `pgvector` extension, which enables efficient storage and querying of high-dimensional vectors. The Embeddings were stored in a `VectorField` with cosine similarity indexing, allowing for rapid similarity searches across the dataset. This design ensured that related notes could be identified in reasonable time without significant computational overhead. Furthermore, the definition of the model is listed in the appendix.

Problems Whenever new notes are added, the system recalculates node Embeddings and predicts new relationships based on their similarity scores. One problem here was, that I wanted to calculate the Embeddings dynamically, without doing a recalculation for the entire graph - but this in itself is a sophistcated field and was thus not further examined.
Another problem which occurred was, that predictions of related notes was initially done also using Embeddings and the querying of the database. Unfortunatly, this yielded inconsistent results and was thus replaced by the recursive query, mentioned below. Nevertheless, the initial approach shall be mentioned.

The core query allowed the system to find the closest neighbors (in terms of cosine similarity) for any given note or topic (**both for word and vector Embeddings**) is shown in the appendix. Below is an example of the query used to retrieve similar Embeddings. This retrieves the top N closest neighbors for a given Embedding by calculating the cosine distance between vectors. The `embedding_vector` field is indexed using `HnswIndex`, which ensures fast retrieval even for large datasets. By specifying a threshold for similarity (e.g., 0.9), only highly relevant results are returned.
For more details on how the model is trained and how the Embeddings are stored in detail, one can take a look at the code and the definition of the model is mentioned in the appendix
The combination of KG Embeddings, automatic recalculation, fast similarity-based queries (used only for Word Embeddings for now, but working the same way for Node Embeddings) and the recursive query later discussed allows ContextLynx to provide users with highly relevant, contextually connected notes.

2.1.2 (LO2) Logical Knowledge

I demonstrated basic proficiency in Logical Knowledge, not only but most importantly by two SQL queries that utilize the graph structure of the system. These queries are designed to identify significant nodes and their relationships, which are crucial for providing relevant topics to the Large Language Model (LLM) and also showing the user related topics.

The first query focuses on retrieving the largest nodes within the Knowledge Graph based on their connectivity. It calculates the number of edges associated with each node to identify those that are most influential within the context of the current project. By prioritizing nodes with high edge counts, this query helps highlight the most relevant topics and notes for further processing. The query can be found in the appendix.
The second query employs a recursive approach to find the nearest related nodes starting from specified nodes. It utilizes a breadth-first search method to explore connections within the graph up to a defined depth, ensuring that only nodes within the same connected component are considered. This allows the system to dynamically identify important related notes and topics, thereby enhancing the contextual relevance provided to the LLM and also to the user. Have a look at the appendix for further information.

2.1.3 (LO4) Data Model

The data model for the note-taking use case is designed around a Knowledge Graph, using a relational database (PostgreSQL) to establish connections between various entities. This model consists of nodes and edges, which form the relationships in the knowledge base.

Entities Overview For a closer look of the models, please have a look at the code, provided in the project under `core/models`.

- **Node**: The fundamental unit of the Knowledge Graph, representing discrete pieces of information. Each node has a word Embedding and a node Embedding. Nodes can be of different types, allowing for flexibility in the representation of various data forms.

- **NodeTopic**: A specialized type of node that encapsulates topics of interest within the Knowledge Graph. Each NodeTopic includes a title, data type, and language, along with references to its Embeddings, enabling it to link to its semantic representation in the Embedding space.
- **NodeNote**: Another specialized node type that holds detailed notes. Each NodeNote comprises fields for storing the title, summary, raw input data, and other metadata.
- **Edge**: This entity represents the relationships between nodes in the graph. Each edge connects two nodes (from_node and to_node) and captures their similarity through a similarity score. The `predicted` field indicates whether the relationship was inferred by the Node2Vec model, described above.
- **NodeEmbedding**: This entity stores the Embeddings of nodes, which serve as vector representations of their semantic meanings. It links nodes to their corresponding Embedding models and provides a mechanism for comparing and retrieving nodes based on their semantic similarity.
- **WordEmbedding**: Similar to NodeEmbedding, this entity specifically focuses on word-level Embeddings used to identify semantically similar terms. This allows for the enrichment of the Knowledge Graph with linguistic context, aiding in tasks like text similarity analysis.

As discussed in the LO below, the system doesn't need scalability for hundreds of thousands of nodes, thus a relation DB with a graph-based model was chosen over a more sophisticated approach using e.g. Neo4J.

2.2 Systems

2.2.1 (LO5) Architecture

For this project, a full-stack architecture was selected, with Django serving as the web framework and PostgreSQL as the database, extended with the pgvector extension to handle both the Knowledge Graph and the Embeddings efficiently.

Technology Choices

- **Django (Backend Framework)**: Django was chosen as the backend framework for two main reasons:
  - First, Django offers a robust, high-level Python web framework that simplifies rapid development while having most options with ML tasks (NER, Word Embeddings, and NLP in general) which are quite relevant in this case.
  - Second, I had not worked with Django before, so adopting it for this project offered an opportunity to learn and explore a new technology.
- **PostgreSQL with pgvector (Database)**: PostgreSQL was chosen as the database for the availability of the `pgvector` extension. The key points were:

- **Vector Extension**: The `pgvector` extension allows PostgreSQL to store and retrieve vector Embeddings alongside traditional relational data. This enables both the storage of KG relationships (edges and nodes) and the Embeddings generated from nodes or textual content. By combining these functionalities in one database, PostgreSQL became the preferred choice.
- **Relational Queries**: Since I was already familiar with SQL and relational databases, using PostgreSQL allows for efficient handling of structured data with SQL queries, making it easier to maintain and query the data without needing to switch to a dedicated graph database.

Why Not a Graph Database? Graph databases like Neo4j or Memgraph were considered due to their natural alignment with the KG model. However, PostgreSQL was ultimately selected for the following reasons:

- **Unified Storage**: Storage for Graph as well as Embeddings, as described above.
- **Performance for Personal Knowledge Management (PKM)**: Although graph databases can be more efficient for very large, interconnected datasets, the scale of this project does not need scalability in this dimension. For PKM, which typically deals with hundreds or thousands of nodes at most, PostgreSQL's performance is more than sufficient. If the dataset were to grow excessively large, other issues such as data curation, user interaction or Django performance limitations become an issue.

Hosted solutions for Postgres and Docker containers were tested using Render.com, but the limitations of the free tiers proved to be a significant challenge. The available resources were insufficient, particularly for handling local models, where the demand for memory and processing power exceeds what the free plans can offer. As a result, the entire stack is now hosted using Docker Compose on a VPS.
In summary, the choice of Django and PostgreSQL with `pgvector` provides a flexible, scalable, and efficient architecture for the project's needs, while allowing me to leverage relational data handling alongside state-of-the-art Embedding storage and retrieval.

2.2.2 (LO6) Scalable Reasoning

In KGs, reasoning involves deriving new insights, predictions, or relationships from the stored data. While querying is useful for straightforward questions, reasoning is necessary for more complex inquiries, especially those that require background knowledge or inference.

Reasoning with KG Embeddings A core aspect of this project involves using Knowledge Graph Embeddings to predict and identify new links between notes (or topics maybe in the future). By Embedding nodes in a high-dimensional vector space, we can efficiently calculate similarities between nodes and predict new relationships (or edges) based on those similarities. For example, in the note-taking application, similar notes can be directly linked by predicting edges between them based on their Embeddings. Already discussed above and shown in the appendix.
This goal is to predict new edges between nodes based on their similarity scores, making it an efficient way to extend the graph with "predicted" edges (these are shown in red in

the application, compared to the others in black). By adjusting the similarity threshold, theoretically set per project, we can control the granularity of the predicted links.

Recursive Reasoning with SQL In addition to using Embeddings for link prediction, recursive SQL queries were employed for reasoning about related nodes. A Breadth-First Search (BFS) approach was used to traverse the graph recursively, starting from a specific node(s) and exploring its neighbors in successive layers. This allows ContextLynx to find related notes or topics based on their connections within the graph and also scaling well to larger datasets.

Initially, it was planned to do the related notes query also via a similarity search in the DB of the Node Embeddings, this solution was not used in the end since it yielded mixed results and the traditional recursive query was preferred for that reason. So in summary, this combination of methods—using Embeddings for similarity-based reasoning and recursive SQL for related nodes—provides one robust and scalable solution for reasoning within the Knowledge Graph.

2.2.3 (LO7/LO8) Creating and Evolving the Knowledge Graph

Among all learning objectives, the creation and evolution of the Knowledge Graph demanded the most time and effort. This process uses a integration of Natural Language Processing (NLP) techniques such as Word Embeddings and Named Entity Recognition (NER), along with graph-based methods for selecting the largest and most relevant existing nodes. The approach was designed to efficiently build the KG from raw input and dynamically evolve it as new information is added through notes. A close look on the respective note creation can be taken with the code base (`core/services/note_service.py`). The procedure for both creating the initial KG (for the first note) and evolving it with new notes follows these steps:

1. If the input is a link (e.g., a YouTube video or website), the content is scraped. Otherwise, the plain text provided is used. See `core/services/web_scraper_service.py` for more details.

2. Named Entity Recognition (NER) is applied to extract named entities (such as people, organizations, or locations) from the text.

3. Word Embeddings are employed to retrieve similar topics based on the input, ensuring that semantically related concepts are considered. Note that the general fetching of Embeddings is completely similar for Node and Word Embeddings. These are just two different classes, since the size differs between Word Embedding Vectors (`bert-base-uncased` is used as the Word Embedding model and returns vectors with a dimension of 768, `node2vec` on the other hand is used for Node Embeddings and returns a defined vector of 16 dimensions) and Node Embedding Vectors. See the appendix for further information.

4. Related nodes are searched to the retrieved topics, helping to identify the most relevant ones within the existing Knowledge Base. This is particularly done on top of using the Word Embeddings in the step before to get topic nodes which are not directly mentioned in the input text. The recursive query for finding related notes

is used again. See the appendix again for further reference. Also note that the query is constructed in such a way, that it can receive any number of nodes, where the closest neighbors are determined.

5. The largest topics, determined by the number of connections to other nodes, are selected for further processing. Again, have a look at the appendix for further reference

6. The largest as well as the most relevant topics are passed to the Large Language Model (OpenAI API), which is tasked with understanding the overall context, summarizing the information, and mapping it to existing topics within the KG. **The prior, Knowledge Graph based scaleable reasoning is necessary, to prompt the LLM, otherwise, with a sufficiently large set of topics (about 100) the LLM does not yield any useful results at all.**

7. Based on the LLM's response, new topics are created or existing topics are linked in the KG. This is again done using the Embeddings stored in the DB to determine Topics which are almost the same, and don't create duplicates in that case. See the appendix.

8. The new note is created and linked to the relevant topics, and the topics are linked to each other to strengthen the graph's structure.

9. The Node2Vec model is retrained to accommodate the newly structured graph and ensure Embeddings remain consistent.

10. Finally, predicted edges between related topics are created based on the retrained model, while outdated edges are deleted.

This workflow ensures that the Knowledge Graph evolves in a structured manner, continuously integrating new information and maintaining its relevance. The combination of NLP techniques and graph-based reasoning allows for a scalable process that grows the KG dynamically with each new note.

2.3 Application

2.3.1 (LO9) Real-World Applications

The project focused on developing a semantic note-taking system was chosen due to the limited number of Personal Knowledge Management (PKM) systems that effectively facilitate "frictionless knowledge saving and retrieval." While existing systems like mem.ai are making strides in this direction, there remains significant potential for alternatives.
From my point of view, the end goal would be to create a PKM system where users no longer have to save information manually, effectively providing them with a "second brain" that retains all essential information and enables context-aware recall when needed. I believe that Knowledge Graphs are the right technology for this task.
Additionally, with the advent of Large Language Models (LLMs) that support near "infinite" context sizes, the idea of frictionless knowledge retrieval becomes increasingly feasible. This advancement enhances the possibility of seamlessly integrating extensive information into a coherent, user-friendly system.

The inspiration for this approach stemmed from my own workflow of saving important messages in my Telegram Saved Messages thread, which often resulted in difficulty retrieving them later. This highlighted the need for a more efficient, intelligent method of managing random pieces of information.

2.3.2 (LO11) Provide Services

The goal of providing a service was exceeded by deploying an early-stage prototype at contextlynx.com.
The prototype features include:

- Adding notes and information in a ChatGPT-like manner or like a "saved messages thread" and automatic context linking (Context Lynx = Context Link)

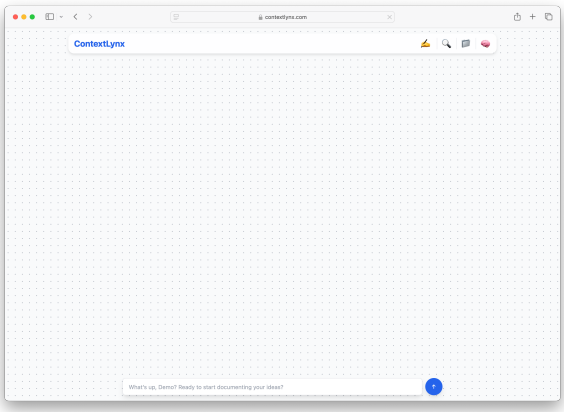

Figure 1: Create Notes in a ChatGPT like manner

- Displaying related notes to each note using the graph structure and KG Embeddings to predict edges (Notes which are predicted highly similar are directly connected through a virtual/predicted edge).

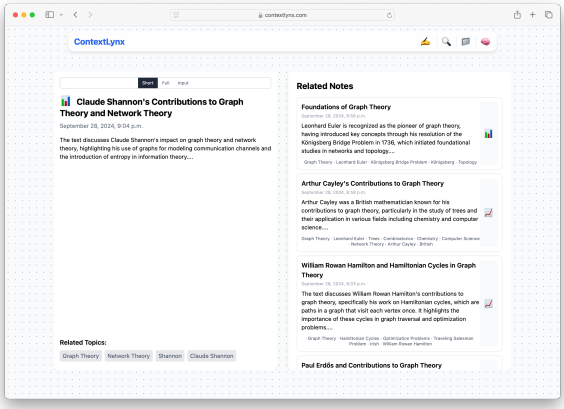

Figure 2: Related Notes Overiew using the Related Nodes Query

- Showing all notes ordered by creation date.

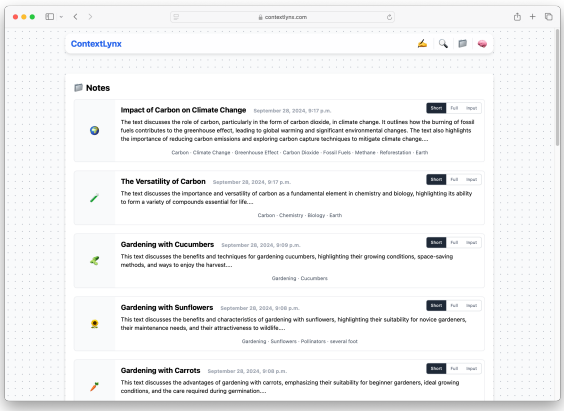

Figure 3: List of all created Notes, Note can be selected to show related notes

- Visualizing the graph, with "real edges" represented in black and predicted edges in red. It is possible to click on any node to see related notes.

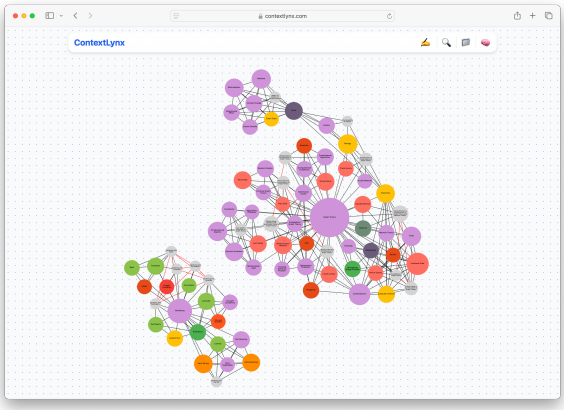

Figure 4: Graph Overview with all Topics, Notes and (predicted) Edges

The service shows how a KG can support user interactions and provide intelligent linking and visualization of personal knowledge.

2.3.3 (LO12) Connections between Knowledge Graphs (KGs), Machine Learning (ML), and Artificial Intelligence (AI)

KGs represent an interesting area within Artificial Intelligence, integrating various techniques to form a cohesive framework for knowledge representation and inference. One key advancement is the emergence of Machine Learning (ML) methods, particularly KG Embeddings.
Additionally, the interplay between traditional logic-based reasoning and ML-based approaches is an interesting aspect, especially in Neural Network-based methods like Graph Neural Networks (GNNs), where KGs play a central role.
This learning objective is achieved through the application of several AI methodologies in combination with traditional KGs.

- **Natural Language Processing (NLP)**: Used to extract insights from text, enhancing the KG's language understanding - this has been particularly interesting with the emergence of LLMs and thus the understanding of Context. It will be interesting to see what can be achieved with the combination of KGs and LLMs in that field.
- **Named Entity Recognition (NER)**: Identifies key entities, enriching the KG with relevant relationships.
- **Word Embeddings**: Represent words in a continuous vector space to capture semantic relationships and enrich the KG with related topics.
- **KG Embeddings**: Similar to Word Embeddings, KG Embeddings map entities and relations in a continuous vector space, preserving the graph's structure and enabling tasks like link prediction. Both techniques aim to encode objects into dense, lower-dimensional representations while capturing semantic or relational information. The similarity lies in how they both enable machine learning algorithms to perform efficiently by representing complex relationships and connections in a form that machines can readily process. This part, already discussed in LO1 was specifically examined with storing and quering such information.

Together, these techniques illustrate how AI manifests within KG and how the interplay between KGs and ML can be achieved, with Word Embeddings and KG Embeddings being particularly important due to their similarity in capturing relationships and structures.

3 Appendix

3.1 Acknowledgements

I declare that I have used generative AI tools as an aid and that my own intellectual and creative efforts predominate in this work. All changes made by AI tools were carefully reviewed to ensure that none of the original semantics were altered. For this, OpenAI's ChatGPT 3.5/4, Anthropic's Claude, and GitHub's CoPilot were used.

3.2 Code

3.2.1 Query for fetching similar Embeddings

```
SELECT id, 1 - (Embedding_vector <-> %s::vector) AS similarity
    FROM {table_name}
    WHERE project_id = %s
      AND 1 - (Embedding_vector <-> %s::vector) >= %s
```

3.2.2 Definition of NodeVector Model

```
class NodeEmbedding(models.Model):
    project = models.ForeignKey(Project, on_delete=models.CASCADE)
    embedding_model = models.CharField(max_length=50)
```

```
    embedding_vector = VectorField(dimensions=16)

    content_type = models.ForeignKey(
        ContentType,
        on_delete=models.SET_NULL,
        null=True,
        blank=True)
    object_id = models.PositiveIntegerField(null=True, blank=True)
    content_object = GenericForeignKey('content_type', 'object_id')

    class Meta:
        indexes = [
            HnswIndex(
                name="node_embedding_vector",
                fields=["embedding_vector"],
                m=32,
                ef_construction=128,
                opclasses=["vector_cosine_ops"],
            )
        ]
```

3.2.3 Configuration of Node2Vec model

See full context in `core/services/node_embedding_service.py`.

```
if not (project.latest_node_embedding_calculated
    and project.id in self.models):
        edges = Edge.objects.filter(project=project).all()

        if len(edges) != 0:
            g = self._generate_graph(edges)

            node2vec = Node2Vec(
                g,
                dimensions=16,
                walk_length=10,
                num_walks=20,
                p=2,  # Return parameter
                q=1,  # In-out parameter
                workers=1
            )

            model = node2vec.fit(
                window=3,
                min_count=1,
                batch_words=4
            )

            self.models[project.id] = model
```

3.2.4 Determination of predicted Edges between Notes

See full context in `core/services/node_embedding_service.py`.

```
def predict_edges(self, project, node, threshold=0.9):
    model = self._get_model(project)
    node_Embedding = model.wv[str(node.id)]

    count = NodeTopic.count(project) + NodeNote.count(project)

    similar_nodes = model.wv.most_similar([node_Embedding], topn=count)
    similar_nodes = [node for node, similarity
        in similar_nodes if similarity > threshold]
    similar_nodes = NodeNote.objects.filter(id__in=similar_nodes).all()

    # remove the node itself
    similar_nodes = [n for n in similar_nodes if n.id != node.id]
    return similar_nodes
```

3.2.5 Query for getting largest Nodes

See full context in `core/models/edge.py`.

```
WITH ranked_nodes AS (
        SELECT DISTINCT ON (node_id)
            COALESCE(from_object_id, to_object_id) as node_id,
            CASE
                WHEN from_object_id IS NOT NULL
                    THEN from_content_type_id
                ELSE to_content_type_id
            END as content_type_id,
            COUNT(*) OVER (PARTITION BY
                COALESCE(from_object_id, to_object_id)) as edge_count
        FROM
            %(edge_table)s
        WHERE
            project_id = %%s AND
            (
                (from_content_type_id = %%s AND
                    from_object_id IS NOT NULL) OR
                (to_content_type_id = %%s AND
                    to_object_id IS NOT NULL)
            )
    )
    SELECT node_id, content_type_id, edge_count
    FROM ranked_nodes
    ORDER BY edge_count DESC
    FETCH FIRST %%s ROWS ONLY
```

3.2.6 Query for getting related nodes

Knowing that this query is extremely long, I decided to include it anyway, since it constitutes one of the most important parts in the project, namely, generically finding nodes

that are related **to a given set of nodes**. See full context in `core/models/edge.py`.

```
@classmethod
def get_n_nearest_nodes(cls,
    nodes,
    target_content_type,
    n=10,
    max_depth=5):
# Convert a single node to a list if needed
if not isinstance(nodes, list):
    nodes = [nodes]

# Extract ids and content types from the list of nodes
start_node_ids = []
start_content_type_ids = []
for node in nodes:
    start_node_ids.append(node.id)
    start_content_type_ids.append(node.get_content_type().id)

target_content_type_id = target_content_type.id

with connection.cursor() as cursor:
    cursor.execute("""
        WITH RECURSIVE connected_nodes AS (
            -- Base case: start with the given list of nodes
            SELECT
                start_node_id,
                start_content_type_id,
                CASE
                    WHEN from_object_id = start_node_id
                        AND from_content_type_id = start_content_type_id
                        THEN to_object_id
                    ELSE from_object_id
                END AS node_id,
                CASE
                    WHEN from_object_id = start_node_id
                        AND from_content_type_id = start_content_type_id
                        THEN to_content_type_id
                    ELSE from_content_type_id
                END AS content_type_id,
                similarity,
                1 AS depth,
                ARRAY[CASE
                    WHEN from_object_id = start_node_id
                        AND from_content_type_id = start_content_type_id
                        THEN to_object_id
                    ELSE from_object_id
                END] AS visited_nodes
            FROM
                """ + cls._meta.db_table + """
            CROSS JOIN UNNEST(%s::integer[], %s::integer[])
                AS t(start_node_id, start_content_type_id)
```

```
    WHERE
        (from_object_id = start_node_id
            AND from_content_type_id = start_content_type_id) OR
        (to_object_id = start_node_id
            AND to_content_type_id = start_content_type_id)

    UNION ALL

    -- Recursive case: find connected nodes
    SELECT
        cn.start_node_id,
        cn.start_content_type_id,
        CASE
            WHEN e.from_object_id = cn.node_id
            AND e.from_content_type_id = cn.content_type_id
            THEN e.to_object_id
            ELSE e.from_object_id
        END AS node_id,
        CASE
            WHEN e.from_object_id = cn.node_id
            AND e.from_content_type_id = cn.content_type_id
            THEN e.to_content_type_id
            ELSE e.from_content_type_id
        END AS content_type_id,
        e.similarity,
        cn.depth + 1 AS depth,
        cn.visited_nodes || CASE
            WHEN e.from_object_id = cn.node_id
            AND e.from_content_type_id = cn.content_type_id
            THEN e.to_object_id
            ELSE e.from_object_id
        END AS visited_nodes
    FROM
        """ + cls._meta.db_table + """ e
    INNER JOIN
        connected_nodes cn ON
        ((e.from_object_id = cn.node_id
            AND e.from_content_type_id = cn.content_type_id) OR
        (e.to_object_id = cn.node_id
            AND e.to_content_type_id = cn.content_type_id))
    WHERE
        cn.depth < %s
        AND NOT (CASE
            WHEN e.from_object_id = cn.node_id
            AND e.from_content_type_id = cn.content_type_id
            THEN e.to_object_id
            ELSE e.from_object_id
        END) = ANY(cn.visited_nodes)
)
SELECT DISTINCT ON (node_id, content_type_id)
    node_id,
```

```
            content_type_id,
            similarity,
            depth
        FROM
            connected_nodes
        WHERE
            content_type_id = %s
        ORDER BY
            node_id ASC,
            content_type_id ASC,
            depth ASC,
            similarity DESC
        LIMIT %s
    """, [
        start_node_ids,
        start_content_type_ids,
        max_depth,
        target_content_type_id,
        n
    ])

    tuples = cursor.fetchall()
    return sorted(tuples, key=lambda x: x[3], reverse=False)
```

3.2.7 Using Embeddings to determine similar Topics

See full context in `core/services/topic_service.py`.

```
def ensure_topic(self,
    project,
    title,
    language,
    data_type=NodeTopicType.OTHER):
    created = False
    topic = NodeTopic.objects.filter(project=project,
        title=title,
        data_type=data_type).first()

    if not topic:
        topics = self.search_topics_word_embedding(project,
            title,
            threshold=0.99)
        if topics:
            topic = topics[0]

    if not topic:
        topic = self.create_topic(project, title, language, data_type)
        created = True

    return topic, created
```