esprit

Se former autrement

HONORIS UNITED UNIVERSITIES

2023 / 2024

SPECIALITY : 5DS

Conceptual graph Based Recommendation System for monitoring RM issues

Realized by:

Tasnim Regaieg

Khiari aymen

Syrine AMAMI

Anas BENBRAHIM

Nermine ben amara

Arij zahra soula

Content

Table of figures

The NLP process pipeline

Introduction

This chapter will examine the complex procedure of data preparation, which is a critical step in the creation of ontology learning models. This stage lays the groundwork for building a solid knowledge graph, which is necessary for a recommendation system for project management problems. We will go over the various steps in this data preparation pipeline, putting special emphasis on how crucial careful data extraction, cleaning, and integration are to producing an accurate and thorough ontology.

1. Pipeline presentation

- Data extraction: transform our data (PDF) into structured format which is easier to work with and analyse.
- Data cleaning: reduce noises , standardizing the data and improve data quality.
- Summarization: to quick understand the essential points and reducing the time required to process information.
- Lemmatization: reduce words to their base forms, promoting text normalization and ensuring consistency.
- Similarity: help identify related or similar content, enhancing the relevance of search results.
- Post-Tag: provide detailed information about the grammatical structure of text, which can be used for more advanced text analysis and understanding.
- Knowledge Graph: establish connections between concepts, enabling better comprehension and discovery of relationships between data points.

2. Data extraction

2.1 PDF « PMBOK6 »

We utilize Python with the `re` module for regular expressions and the `pandas` library for creating a Data Frame. We aim to split a text document into chapters based on chapter numbers

and collect this data for future processing. The code also contains commented-out sections related to page numbers.

Additionally, we process and organize content from chapters 1 to 13. We use various data manipulation techniques within Python, primarily relying on Pandas for creating Data Frames and Regular Expressions for pattern matching. We extract content from the structured text, divide it into hierarchical levels (TitleL1, TitleL2, and TitleL3), and combines it into a final Data Frame called `Final_df`. We also ensure consistent lengths for each list by filling missing data with placeholders and consolidates the chapter information in the Data Frame for further processing.

We use Python's functools.reduce function and Pandas to create a new Data Frame df_better with modified columns. The primary objective is to split the text in our Data Frame into two parts: a numeric part and a non-numeric part.

- A column with the numeric part obtained from the original data which represents the number of section and chapter.
- A column with the non-numeric part obtained from the original data which represents the title of the chapter and sections.

Next, we add a new column called “type” containing the extracted type of information from “titlel2” .We extract and return references to sections in a given content.

We use functions to process text containing section references, extract and replace these references with the corresponding section content and ensure that nested references are correctly resolved. The process of filling references is repeated until no more references can be filled or the maximum iteration limit is reached.

fill_section(text): This function takes a text as input and searches for references to sections (e.g., "Section 1.2"). If a reference is found, it tries to replace it with the corresponding content from the Data Frame df_better (based on section numbers) if available. If not available, it removes the reference from the text.

get_all_ref(text, ref): This function extracts all section references from the input text and adds them to a reference set (ref). It does this by recursively searching for section references within the content of sections.

deep_search_set(num, ref): This is a utility function used to deeply search for a specific number (num) within a set of references (ref). It returns True if the number is found within the set or its nested references.

fill_all_ref(text, ref): This function is responsible for filling all references within the input text. It uses a set of references (ref) and iteratively replaces references in the text with their corresponding content from df_better. This process continues until no more references can be filled or a maximum iteration limit (i) is reached.

2.2 PDF « practice standard project risk management »

Nearly the same thing happened to this report for this section.
For one reason, that this report doesn't contain the same information as the preview, we didn't apply the section direction process.

3. Data cleaning

This process has been applied to both PDFs.

3.1 Grammar Correction

We define a function named correct_grammar to perform grammar correction on textual content.
The tool.check(text) method is employed to scan the input text for grammatical errors. This function returns a list of identified issues.
The language_tool_python.utils.correct(text, matches) function is used to correct these errors, thereby generating a refined version of the text with corrected grammar.

3.2 Cleaning Text Data

The primary function, clean_text, is designed to clean and preprocess text content.

- Initially, the input text is converted to lowercase, ensuring uniformity in text casing.
- The function identifies and removes specific phrases known as "stop phrases" from the text. These phrases, such as "See section," "Described in Section," and others, are systematically eliminated from the text to improve clarity and readability.
- Subsequently, special characters and punctuation are removed from the text to focus on the core alphanumeric content.
- Stop words, common words such as "the," "is," "in," etc., that often do not contribute significantly to the meaning of the text, are eliminated. This step enhances the informativeness of the text data.

The function, named preprocess_text, is designed to preprocess text data.

- Splits the text into individual lines based on newline characters ("\n").
- Removes leading and trailing whitespaces from each line and filters out empty lines, ensuring that only lines with actual content are retained.
- Filters out lines that contain specific patterns: Lines containing the copyright notice "©2009 Proect Management Institute. Practice Standard for Project Risk Management" are removed.

4. Summarization

The summarize_content function is defined to perform text summarization. Inside the function, a pretrained summarization model ("facebook/bart-large-cnn") and its associated tokenizer are loaded. These models are capable of generating high-quality summaries. The input text is tokenized using the tokenizer.

The model is then employed to generate a summary of the input text. It does so by taking the tokenized input and using "num_beams" to search for the best summary. Additionally, "min_length" is set to 0, which means there's no minimum length for the summary.

The generated summary is decoded using the tokenizer, removing any special tokens and ensuring that the resulting summary is free from any tokenization artifacts.

5. Lemmatization

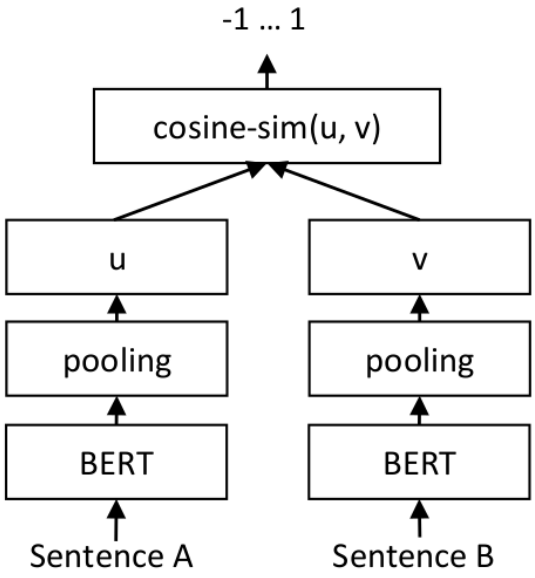
The enhanced_lemmatize function is defined to perform lemmatization on the content.

For each word in the content, lemmatization is applied. The lemmatization process is enhanced with the following considerations:

- The word's part-of-speech tag is used to determine its context and ensure accurate lemmatization.
- Words that are punctuation or part of a predefined set of English stop words are excluded from lemmatization. This helps retain the meaningful content of the text while excluding non-content words.
- The NLTK's WordNetLemmatizer is used to perform lemmatization, which ensures that words are reduced to their base form based on their context and part of speech.

6. Similarity

SBERT (Sentence-BERT) is an adaptation of the BERT (Bidirectional Encoder Representations from Transformers) model specifically optimized for sentence embeddings. When combined with cosine similarity, it allows for the efficient computation of semantically meaningful similarity scores between sentences. Cosine similarity measures the cosine of the angle between two non-zero vectors, and in the context of SBERT, these vectors represent sentence embeddings. A cosine similarity score close to 1 indicates high similarity, while a score close to -1 indicates high dissimilarity. By using SBERT embeddings with cosine similarity, one can effectively capture and compare the semantic content of sentences.



7. Post-Tag

Part-of-speech tagging, often abbreviated as POS tagging, is a fundamental task in natural language processing (NLP) that involves determining the grammatical category of each word in a sentence. These categories include nouns, verbs, adjectives, adverbs, pronouns, prepositions, conjunctions, and interjections. POS tagging is crucial for various NLP applications such as machine translation, text-to-speech synthesis, and information retrieval, as it provides valuable information about the syntactic structure of a sentence. By assigning specific tags to words, computational models can better understand the relationships between words and extract meaningful insights from textual data, enabling the development of more accurate and efficient language processing systems.

8. Knowledge Graph

Knowledge graphs are essential in various fields such as artificial intelligence, natural language processing, and data mining. They enable the organization and retrieval of information in a way that reflects real-world relationships, allowing algorithms to understand complex concepts, infer new knowledge, and make intelligent decisions. By leveraging semantic connections and contextual information, knowledge graphs enhance the efficiency of search engines, recommendation systems, and question-answering applications, making them invaluable tools in the realm of information retrieval and knowledge discovery.

SpaCy is used in combination with machine learning techniques to extract entities and relationships from text data. By training a custom named entity recognition (NER) model, and teach SpaCy to recognize specific entities (like person names, organizations, or locations) in a given text. Once the entities are extracted, additional a SpaCy's Matcher class to define a pattern based on linguistic dependencies and part-of-speech tags and then searches for this pattern in the input sentence employed to identify relationships between these entities, forming the basis of a Knowledge Graph.

Conclusion

This chapter clarified the complex procedure of data preparation, highlighting its role as the cornerstone for building a solid knowledge graph, essential for establishing an efficient recommendation system for project management issues that will eventually evolve into a web user interface.

GNN architecture modelling

Introduction

In this section, our focus is the process of constructing the Graph Neural Network (GNN) architecture. The development of the GNN encompasses a sequence of precisely defined actions tailored to implement it for the creation of a recommendation system.

1. Pipeline presentation

- Data Preparation:

Data Cleaning: Enhance text quality by converting to lowercase, removing non-English words, and extracting nouns and verbs.

Extracting Triplets: Use a semantic model to find Subject-Object-Verb (SOV) triplets in sentences.

Extracting Taxonomic Relations: Identify taxonomic relations using the Textacy library.

- Node and Edge Embeddings:

Use GloVe embeddings for words and relationships to create numerical representations.

- Construct the Graph:

Build a graph using NetworkX from extracted triplets, assign embeddings, and convert it to RDF format.

- GNN Architecture:

Implement a Graph Neural Network (GNN) with two layers for knowledge graph embedding.

- Message Passing:

Use message passing in GNN to update node representations based on neighbor information.

- Generating the Knowledge Graph:

Apply the GNN model to generate a knowledge graph with refined node embeddings.

- Fine-Tuning:

Further train the GNN model with early stopping and learning rate decay.

- Evaluation:

Evaluate the GNN model on a test set using Mean Squared Error (MSE) and Mean Absolute Error (MAE).

- Metadata:

Serialize metadata into RDF format and save to 'model_metadata_with_weights.rdf.'

- Recommendation System Class:

Introduce a recommendation system class (GNNBasedRecommendationSystem) for making recommendations based on the GNN model.

2. Data Preparation

2.1 Data cleaning

In the data cleaning phase, several functions have been implemented to enhance the quality of textual data. The `clean_text` function initially converts the text to lowercase and removes stop words, numbers, and symbols. It utilizes the NLTK library for tokenization and filtering, creating a cleaner version of the text.

The `remove_non_english_words` function focuses on extracting only English words from the text by utilizing the NLTK words corpus. This ensures that the text consists of meaningful English vocabulary.

The `extract_nouns_and_verbs` function tokenizes the text, performs part-of-speech tagging, and retains only nouns and verbs, providing a more focused representation of the content., the The `clean_text_3` function eliminates specific phrases and removes special characters, punctuation, and stop words.

2.2 Extracting SOV triplets

In this part, we are aiming to extract Subject-Object-Verb (SOV) triplets using a semantic model. The process involves the following steps:

- Load SpaCy Model : The *en_core_web_sm* model from spaCy is loaded. This model is designed for English language processing and provides various linguistic annotations.
- Load Huggingface Model: we utilize a pre-trained model from Hugging Face for sequence-to-sequence tasks. It loads a tokenizer (AutoTokenizer) and a model (AutoModelForSeq2SeqLM) specifically trained for semantic tasks.
- extract_sov_semantic Function: This function takes a sentence as input. It tokenizes the sentence using the loaded tokenizer, and then generates a sequence using the pre-trained model. The generated sequence is decoded using the tokenizer to obtain the SOV triplet text.

The SOV triplet represents the semantic structure of the sentence in terms of its subject, object, and verb.

2.3 Extracting taxonomic relations

In this part, we aim to extract taxonomic relations from sentences using the Textacy library. The steps are as follows:

- Load SpaCy Model:The en_core_web_sm model from spaCy is loaded.
- extract_taxonomic Function: This function takes a sentence as input. The sentence is processed using the spaCy model to extract subject-verb-object (SVO) triples, which represent syntactic relationships. The results are collected in a list.
- Processing Paragraphs and Sentences: Similar to the SOV triplet extraction, the code iterates over paragraphs and sentences, processing each cleaned sentence. For each sentence, taxonomic relations are extracted using the extract_taxonomic function.
- DataFrame Creation: The results, including the original sentence, the semantic SOV triplet, and taxonomic relations, are stored in the same DataFrame (sov_df).

2.4 Data processing

In this data processing step we take information from the original DataFrame (sov_df), extract specific components using regular expressions, and structure the data into a new DataFrame (new_df).

- Regular Expressions for:
  - subject_pattern.
  - verb_pattern
  - object_pattern
  - pipe_verb_pattern
  - before_pipe_pattern
  - after_pipe_pattern

3. Node and Edge Embeddings

This section employs pre-trained GloVe embeddings from the Stanford NLP website, the 50-dimensional version of GloVe embeddings is used in this case, to create embeddings for nodes (individual words) and edges (relationships between words) in a graph, facilitating the

representation of semantic triplets and taxonomic relations in a more continuous and numerical form for subsequent processing or analysis.

4. Construct the Graph

We leverage the provided data to create a graph representation using the NetworkX library. Nodes are constructed from the Subject and Object of Subject-Verb-Object (SVO) triples and the parts before and after the pipe. Embeddings are generated for both nodes and relations based on pre-trained GloVe vectors. Additionally, attribute 'action' is assigned to relations based on the corresponding verb. Duplicate nodes are removed, resulting in a directed graph `G`. Subsequently, the graph is converted into an RDF representation using the rdflib library. Nodes are represented as instances of the 'Class' type, and their embeddings are captured using RDF triples with 'hasEmbedding' predicates. Similarly, relations are modeled with 'action' attributes, and their embeddings are linked through RDF blank nodes. The RDF graph is then serialized into RDF/XML format and saved to a file named 'long_KG.rdf.'

5. GNN Architecture

Graph Neural Network (GNN) architecture is employed for knowledge graph embedding. The architecture consists of two Graph Convolutional Layers (GraphConv), transforming input node features with a 50-dimensional embedding into an intermediate representation with a specified hidden dimension (99 in this case) and then to the final output representation. The chosen dimensions are optimized through hyperparameter tuning with Optuna. The model is trained for 1000 epochs using Mean Squared Error (MSE) loss and the Adam optimizer. The resulting GNN model is then saved to a file named 'node_generator_model.pth.'

6. Message Passing

The Message Passing mechanism is an essential component of the GNN architecture. It involves propagating information through the graph, updating node representations based on their neighbors. In this case, the GraphConv layers in the GNN employ message passing to iteratively refine node embeddings using information from neighboring nodes.

7. Generating the Knowledge Graph

The trained GNN model is utilized to generate node embeddings for the knowledge graph. The RDF graph, previously constructed, is loaded. Nodes are assigned embeddings using the GNN model, and these embeddings are used to update the RDF graph. Information about the layers and their parameters in the GNN model is added to the RDF graph for metadata purposes. The RDF graph is serialized into RDF/XML format, and both the graph and the model's weights are saved separately to 'model_metadata.rdf' and 'model_weights.pth,' respectively.

8. Fine-Tuning

The GNN model undergoes further training with early stopping and exponential decay in the learning rate. Nodes are split into training, validation, and test sets, with 70%, 15%, and 15% proportions, respectively. The model's training is monitored for stale validation loss, and training stops if the loss does not improve within a specified patience window (10 epochs). The best hyperparameters are determined through Optuna hyperparameter optimization.

9. Evaluation

The GNN model is evaluated on the test set using Mean Squared Error (MSE) and Mean Absolute Error (MAE) metrics. The model is applied to generate predictions, and the resulting errors are calculated and displayed.

10. Meta data

The metadata is serialized into RDF/XML format and saved to 'model_metadata_with_weights.rdf.'

11. Recommendation system class

A reccomendation system class, `GNNBasedRecommendationSystem`, is introduced with RDF metadata and model weights, enabling recommendations based on the GNN model’s predections.

Conclusion

To conclude, this process blends language processing, semantic modeling, and graph techniques to simplify complex information into a clear knowledge graph. The refined Graph

Neural Network, with pre-learned embeddings, demonstrates its ability to understand and represent data relationships. Evaluation metrics offer insights for improvement. The resulting recommendation system illustrates the method's effectiveness, providing a practical way to implement knowledge graphs into a web interface.

The implementation of the Knowlege Graph for Recommendation system

Introduction

This section will cover the development tools, architecture, and final implementation of the knowledge graph for recommendation system in our web interface, along with some figures that illustrate our interface.

1. Development tools

1.1 Deep Graph Library

This is a Python library designed to make it easy to implement and work with deep learning models that operate on graphs. It's useful for models that require graph structures, such as social networks or molecular structures.

1.2 Natural Language Toolkit

NLTK is a leading platform for building Python programs to work with human language data. It provides easy-to-use interfaces for over 50 corpora and lexical resources, such as WordNet, along with a suite of text processing libraries for classification, tokenization, stemming, tagging, parsing, and semantic reasoning.

1.3 Torch-PyTorch

PyTorch is an open-source machine learning library based on the Torch library, used for applications such as computer vision and natural language processing, primarily developed by Facebook's AI Research lab. It's known for its flexibility and is used extensively for deep learning models.

1.4 NumPy

This is the fundamental package for scientific computing in Python. It provides a high-performance multidimensional array object and tools for working with these arrays. It's often used for numerical data processing and transformation.

1.5 Scikit-learn

While not explicitly mentioned, the cosine_similarity function suggests the use of scikit-learn, a library for machine learning that provides simple and efficient tools for data mining and data analysis.

1.6 Sentence Transformers

This is a Python framework for state-of-the-art sentence, text embedding, and representation models. It simplifies the process of generating embeddings for sentences or paragraphs so that they can be compared semantically.

1.7 Django

Django is a high-level Python web framework that encourages rapid development and clean, pragmatic design. It's used to build web applications quickly and with less code.

1.8 Docker

Docker is a set of platform-as-a-service products that use OS-level virtualization to deliver software in packages called containers. It's widely used for deploying applications in a consistent environment with all its dependencies.

1.9 Pandas

This is a fast, powerful, flexible, and easy-to-use open-source data analysis and manipulation tool, built on top of the Python programming language.

2. Architecture of the Web Application

In the context of Django, the MVC architecture is slightly modified and is often referred to as the Model-View-Template (MVT) pattern. However, the principles remain the same with a focus on separating concerns to organize code better.

2.1 Model

In Django, a model is the definitive source of information about your data. It contains the essential fields and behaviors of the data you’re storing. Django follows the DRY Principle

(Don't Repeat Yourself). The model defines the structure of the database, and Django automatically generates the database schema (tables, fields, and indices) from the model.

2.2 View

In Django, the view responds to the user's request by performing the appropriate operation, which may involve reading or writing to the model. A view can be a Python function or a class that takes a web request and returns a web response. This response can be the HTML contents of a web page, a redirect, a 404 error, an XML document, an image, etc.

2.3 Template

The template is Django's version of a view in the classic MVC pattern. It's a presentation layer that handles the part of the application that the user sees and interacts with. Django's templating system allows mixing HTML with Django template tags and filters for rendering dynamic content.

2.4 Controller

The "controller" part in Django is handled by the framework itself. It's the underlying mechanism that sends a user request to the appropriate view, based on the Django URL configuration. You don't really write controllers in Django; instead, you write views which are a combination of the MVC's view and controller. The mapping between the URLs and the views is defined in the urls.py file.

3. Integration of Machine Learning Models

To enhance clarity in our report, the integration process of the deep neural network can be delineated as a sequence of steps.

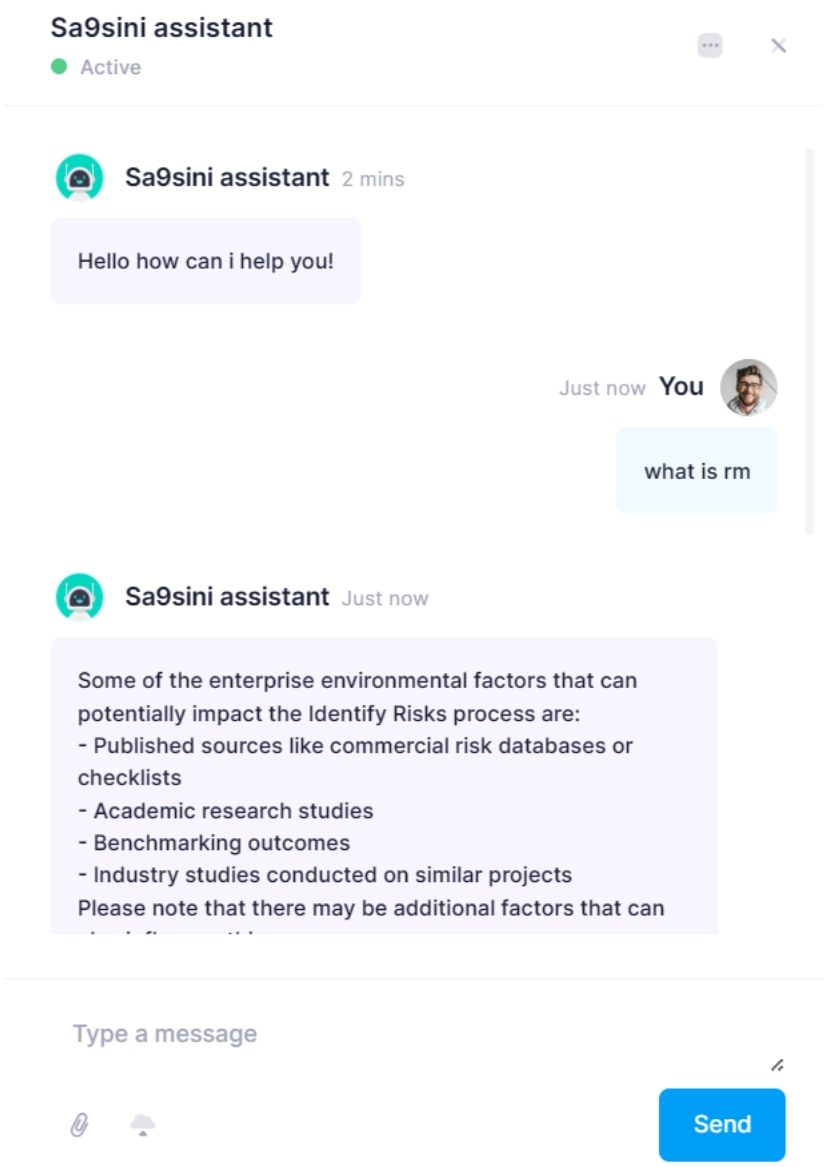


Figure 1: Recommender chat-bot interface

3.1 Input Embedding

Initially, user inputs are transformed into numerical representations through an embedding process. This is essential because our neural network is designed to interpret and process numerical values only.

3.2 Recommendation System Activation

These numerical embeddings are then fed into the recommendation system. The system utilizes these inputs to accurately determine the most pertinent nodes and their immediate network neighbors.

3.3 Subgraph Generation

Upon identification of the relevant nodes, the model proceeds to output a subgraph. This subgraph represents a focused section of the network that is most significant to the user's input.

3.4 Text Decoding

In the final step, the subgraph is decoded back into textual information. This step is crucial as it converts the model's numerical output into human-readable text, thereby providing actionable and intelligible content based on the user's original query.

Conclusion

In conclusion, this section has outlined the development tools, architecture, and final implementation of our web interface's recommendation system knowledge graph. Accompanied by illustrative figures, these insights highlight the robust foundation supporting an enhanced and user-centric experience.