



M2 DATA**SCALE**

Rapport
PROJET KG-Enabled DevOps RAG chatbot

Etudiants:
MAZIZ Yassine
Yaici Walid

Ufr des sciences
Université de Paris saclay

Contents

1	Introduction	2
2	Etat de l'art	2
2.1	Chatbot integration in few patterns	2
2.2	Knowledge Graph Prompting for Multi-Document Question Answering	3
2.3	Exploring Large Language Models for Knowledge Graph Completion	4
2.4	LayoutLMv3: Pre-training for Document AI with Unified Text and Image Masking	5
3	Documents utilisés	5
4	Création du graphe	5
4.1	Création du graphe sur le contenu	5
4.1.1	Extraction des entités	6
4.1.2	Construction des requetes cypher	7
4.1.3	Visualisation	8
4.2	Création du graphe de structure	9
4.2.1	Extraction des entités/reliations	9
4.2.2	Construction du graphe	10
4.2.3	Visualisation	11
5	Requêtage sur graphe	12
5.1	Requêtage sur le graphe contenu	12
5.1.1	Creations des embeddings de noeuds	12
5.1.2	Exploration du graph pour la récolte d'informations	12
5.1.3	Utilisation du LLM pour générer la réponse	13
5.1.4	Expérimentations	14
5.2	Requêtage sur le graphe structure	14
5.2.1	Extraction de schéma	14
5.2.2	Génération de la requete Cypher	15
5.2.3	Envoie de la requete cypher	15
5.2.4	Formulation de la réponse	15
5.2.5	Expérimentation	16
5.3	Requêtage final	17
6	Création du chatbot	17
6.1	Streamlit	17
6.2	Interface du chatbot	17

1 Introduction

Dans un monde où l'information est omniprésente mais souvent éparse, l'accès rapide et efficace aux connaissances pertinentes est devenu un défi majeur. Pour répondre à cette exigence croissante, l'intelligence artificielle (IA) offre des solutions prometteuses, notamment dans le domaine de la recherche et de l'assistance informationnelle. Dans ce contexte, ce rapport présente un projet innovant visant à exploiter les capacités avancées de GPT-3.5, un modèle de langage développé par OpenAI, pour créer un chatbot capable de répondre à des questions en se basant sur des documents.

2 Etat de l'art

2.1 Chatbot integration in few patterns

Ce document traite de l'intégration des chatbots dans divers systèmes logiciels. Les chatbots sont des agents logiciels qui peuvent interagir avec les humains en utilisant le langage naturel.

1. Chatbot main dimensions

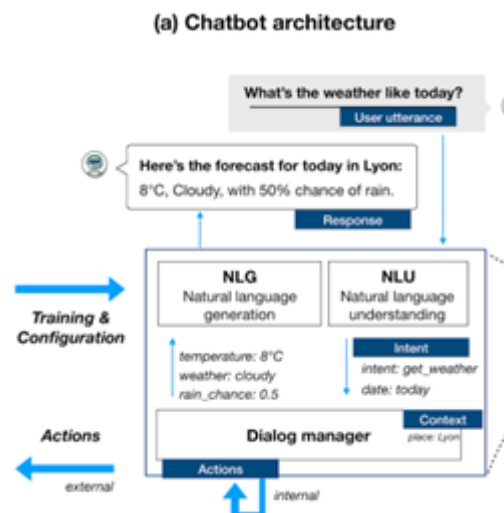


Figure 1: Chatbot integration in few patterns

- Intents : il s'agit des demandes conceptuelles formulées par les utilisateurs, c'est-à-dire des tâches que le chatbot doit effectuer.
- Contrôle du dialogue : il s'agit de gérer le flux de la conversation entre le chatbot et l'utilisateur, en tenant compte du contexte et des interactions précédentes.
- Actions : il s'agit des opérations ou des tâches spécifiques que le chatbot effectue en réponse aux intentions de l'utilisateur (appel API).

2. Chatbot integration

Nous définissons l'intégration des chatbots comme le problème de l'intégration des capacités conversationnelles dans les systèmes logiciels existants.

- IN-APP Assistant: un assistant in-app est un chatbot qui réside dans une application existante, telle qu'un site web ou une application de bureau, et qui étend les fonctionnalités de l'application avec des capacités de conversation.
- Agent GUI : Ce modèle implique l'intégration des capacités du chatbot dans les interfaces utilisateur graphiques (GUI) en rendant des widgets indépendants ou des composants UI qui ont leur propre logique d'application et leurs propres données.
- API caller : Dans ce schéma, le chatbot interagit avec d'autres systèmes logiciels par l'intermédiaire de leurs API ou de leurs services dorsaux. Il peut faire appel à ces API pour effectuer des tâches spécifiques ou récupérer des informations.

2.2 Knowledge Graph Prompting for Multi-Document Question Answering

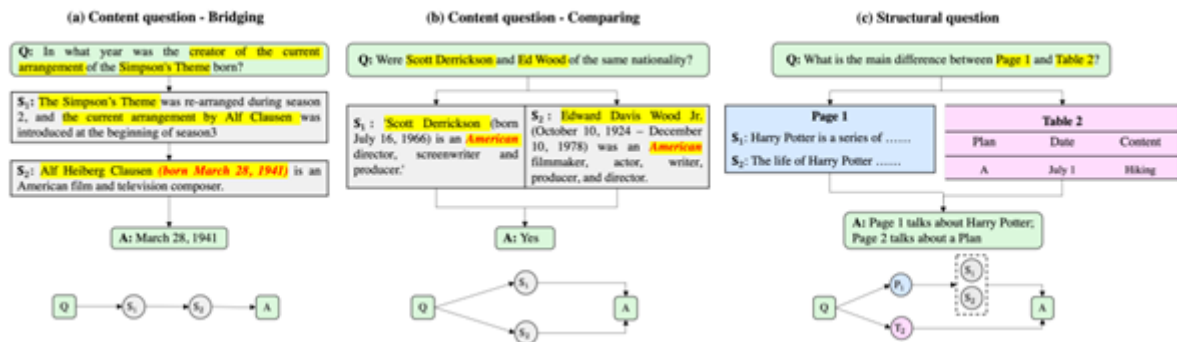


Figure 2: Multi-Document Question Answering 1

Ce document traite d'une méthode appelée Knowledge Graph Prompting (KGP) pour la réponse aux questions multi-documents (MD-QA). Cette méthode vise à combler les lacunes des approches existantes qui se concentrent sur la réponse à des questions portant sur un seul document ou sur un domaine ouvert. KGP se compose de deux modules : la construction de graphes et la traversée de graphes.

Dans le module de construction de graphe, un graphe de connaissances (KG) est créé à partir de plusieurs documents. Les nœuds du graphe représentent des passages ou des structures de documents, et les arêtes représentent la similarité sémantique/lexicale ou les relations structurelles entre les passages. Le graphe sert de règle globale qui régule l'espace de transition entre les passages et réduit la latence de la recherche.

Les auteurs explorent différentes méthodes de construction du KG, notamment le TF-IDF, le K-voisin le plus proche (KNN) basé sur des transformateurs de phrases ou un codeur formé sur une tâche de recherche multi-sauts.

Dans le module de parcours de graphe, un parcours de graphe guidé par LM navigue dans le graphe pour rassembler les passages qui permettent de répondre à la question. Le navigateur guidé par LM est entraîné à générer la preuve suivante pour aborder la question et à sélectionner le passage voisin le plus prometteur à visiter ensuite sur la base de la preuve générée. Cette navigation locale garantit la qualité de la recherche et permet d'approcher progressivement la question.

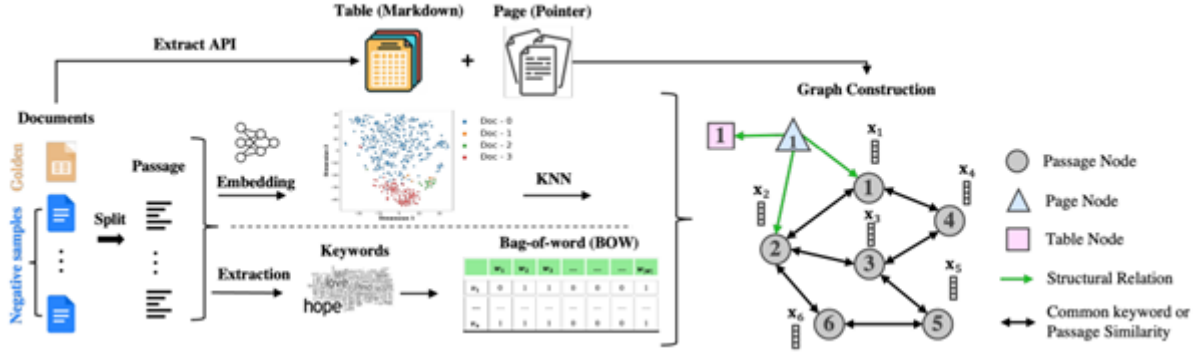


Figure 3: Multi-Document Question Answering 2

2.3 Exploring Large Language Models for Knowledge Graph Completion

Le document explore l'utilisation de grands modèles de langage (LLM) pour compléter les graphes de connaissances, ce qui vise à résoudre le problème de l'incomplétude des graphes de connaissances. L'étude présente un cadre appelé Knowledge Graph LLM (KG-LLM) pour modéliser les triplets dans les graphes de connaissances comme des séquences de texte. Les LLM sont affinés pour traiter les données des graphes de connaissances à l'aide d'invites et de réponses pour prédire la plausibilité d'un triple ou d'une entité/relation candidate.

Le document traite de l'importance des graphes de connaissances dans les tâches d'intelligence artificielle et du défi que représente l'achèvement des graphes de connaissances. Il souligne les limites des méthodes existantes, telles que l'intégration des graphes de connaissances, et met en évidence le potentiel de l'intégration d'informations textuelles pour améliorer la représentation des connaissances.

La méthode KG-LLM proposée est évaluée sur divers graphes de connaissances de référence et il s'avère qu'elle atteint des performances de pointe dans des tâches telles que la classification triple et la prédiction de relations. L'étude compare les performances de KG-LLM avec d'autres méthodes, y compris des modèles linguistiques pré-entraînés plus petits (LLaMA-7B, ChatGLM-6B), et démontre la supériorité de l'ajustement fin de modèles relativement plus petits sur les récents ChatGPT et GPT-4.

Le document comprend des expériences et des résultats détaillés sur quatre ensembles de données de référence de graphes de connaissances, fournissant des statistiques sur le nombre d'entités, de relations, d'instances de formation, de développement et de test. Il présente également les scores de précision de la classification triple, les scores de succès 1 de la prédiction des liens et les scores de succès 1 de la prédiction des relations pour différentes méthodes, notamment KG-LLM, LLaMA, ChatGLM, GPT-4 et d'autres. Les résultats montrent des résultats prometteurs pour KG-LLM dans diverses tâches, indiquant des améliorations significatives dues à la transformation de l'instruction.

Dans l'ensemble, ce document fournit une vue d'ensemble de la méthode KG-LLM proposée, de ses résultats expérimentaux et de son impact potentiel sur la complétion des graphes de connaissances et d'autres tâches de NLP.

2.4 LayoutLMv3: Pre-training for Document AI with Unified Text and Image Masking

un modèle pré-entraîné d'intelligence artificielle des documents peut analyser la mise en page et extraire des informations clés pour divers documents tels que des formulaires scannés et des documents universitaires.

L'architecture du modèle LayoutLMv3 est un transformateur multimodal texte-image unifié conçu pour apprendre les représentations multimodales. Le transformateur se compose de plusieurs couches, chacune comprenant principalement des réseaux d'auto-attention à plusieurs têtes et des réseaux d'anticipation entièrement connectés par rapport à la position. L'entrée du transformateur est une concaténation de séquences de texte ($Y = y1:L$) et d'images ($X = x1:M$), où L et M sont les longueurs des séquences pour le texte et l'image, respectivement.

Dans l'ensemble, l'architecture du modèle de LayoutLMv3 est conçue pour répondre à la divergence des objectifs de pré-entraînement pour les modalités texte et image, ce qui lui permet d'atteindre des performances de pointe dans les tâches d'IA de documents centrées sur le texte et sur l'image. L'architecture unifiée du modèle et ses objectifs d'entraînement en font un modèle polyvalent et efficace pour un large éventail de tâches d'IA de documents.

3 Documents utilisés

Pour notre projet, nous avons décidé d'utiliser des documents qui parlent de différents films, pour cela nous avons extrait les documents directement sur Wikipedia, à partir de leur API, celle-ci nous permet de récupérer une page Wikipedia en spécifiant le titre de celle-ci.

Nous avons donc défini une liste de films, et grâce à un script Python, nous avons récupéré tout les textes correspondant que nous avons ensuite enregistré dans une liste qui sera utilisé tout au long du projet.

4 Création du graphe

4.1 Création du graphe sur le contenu

Afin de construire un graph sur le contenu des documents extrait de wikipedia nous avons suivie la meme approche que l'article [Knowledge Graph Prompting for Multi-Document Question Answering].

Cette etape ce fait comme suit:

1. Dans un premier temps on commence par diviser le document en chunk. Vu qu'on a utilisé une base de documents de wikipedia, chaque chunk est constitué du contenu d'un titre du document. c'est a dire que pour chaque titre dans le document son contenu est transformé en chunk.
2. Une fois tous les chunk du document construit, on passe à l'étape de construction du graph à partir de ces chunk.
3. La creation du graph se divise en deux étapes

4.1.1 Extraction des entités

- Une première fonction va parcourir l'ensemble des chunk et construire trois dictionnaires:
 - Un dictionnaire avec pour clé une entité extraite et comme valeur tous les chunk desquels elle a été extraite.
 - Un deuxième dictionnaire avec pour clé l'énoncé du chunk et comme valeur toutes les entités extraites de ce chunk.
 - Un troisième dictionnaire avec comme clé le titre du document et comme valeur tous les chunk de ce document.

```
nlp = spacy.load('en_core_web_lg')

def wiki_spacy_extract_chunk(d):
    nlp = spacy.load('en_core_web_lg')
    nlp.add_pipe("entityLinker", last=True)

    kw2chunk = defaultdict(set)
    chunk2kw = defaultdict(set)
    title2chunk = defaultdict(set)
    chunk2title = defaultdict(set)

    for title, chunk in tqdm(d['title_chunks']):
        doc = nlp(chunk)
        entities = doc._.linkedEntities
        for entity in entities[:10]:
            entity = entity.get_span().text

            kw2chunk[entity].add(chunk)
            chunk2kw[chunk].add(entity)

            title2chunk[title].add(chunk)
            chunk2title[chunk].add(title)

    for key in kw2chunk:
        kw2chunk[key] = list(kw2chunk[key])

    for key in chunk2kw:
        chunk2kw[key] = list(chunk2kw[key])

    for key in title2chunk:
        title2chunk[key] = list(title2chunk[key])

    for key in chunk2title:
        chunk2title[key] = list(chunk2title[key])

    d['kw2chunk'] = kw2chunk
    d['chunk2kw'] = chunk2kw
    d['title2chunk'] = title2chunk
```



```

d['chunk2title'] = chunk2title

return d

```

4.1.2 Construction des requetes cypher

La deuxième étape consiste a transformer ces dictionnaires en requetes Cypher pour creer le graph au niveau de Neo4j:

- Dans un premier temps on crée pour chaque chunk un noeud de type Contenu avec comme propriété contenu qui represente le text du chunk.
- Ensuite pour chaque mot clé dans le dictionnaire keyword2chunk, on crée une relation entre tous les noeuds qui ont ce mot clé en commun. la relation crée est de type has_commun_entity avec comme propriété entity qui contiendra le mot clé.
- Enfin pour chaque titre dans le dictionnaire title2chunk qui contient pour chaque titre la liste des chunk ayant ce titre, on crée une relation entre tous les chunk concernés. La relation crée est de type has_commun_title et contient le predicat title avec le titre du document.

```

def kw_graph_construct(file, d):

    with open(file, "w") as f:
        chunk2id = {}
        for i, chunk in enumerate(d['title_chunks']):
            _, chunk = chunk
            chunk_sans_guillemet = chunk
            if '"' in chunk:
                chunk_sans_guillemet = chunk.replace('"', '')
            f.write('MERGE (:Content {contenu:"' + chunk_sans_guillemet +
                    '"', id:'
                        + str(i) + '});\n')
            chunk2id[chunk] = i

        for kw, chunks in d['kw2chunk'].items():
            for i in range(len(chunks)):
                for j in range(i+1, len(chunks)):
                    f.write("MATCH (node1:Content), (node2:Content)\n")
                    f.write("WHERE node1.id = "+str(chunk2id[chunks[i]])+"
                            AND node2.id = "+str(chunk2id[chunks[j]])+"\n")
                    f.write('MERGE (node1)-[:has_commun_entity{entity:"'+kw
                            +'" } ]-(node2);\n')

        for kw, chunks in d['title2chunk'].items():
            for i in range(len(chunks)):
                for j in range(i+1, len(chunks)):
                    f.write("MATCH (node1:Content), (node2:Content)\n")
                    f.write("WHERE node1.id = "+str(chunk2id[chunks[i]])+"
                            AND node2.id = "+str(chunk2id[chunks[j]])+"\n")

```

```
f.write('MERGE (node1)-[:has_commun_title{title:"'+kw+'"}]-(node2);\n')
```

A la fin de la deuxième étape un fichier text est creer avec la liste des requetes à executer pour construire le graph. on execute le code suivant afin de lancer les requetes sur Neo4j.

```
path = './script.txt'
uri = ""
user = ""
password = ""
driver = GraphDatabase.driver(uri, auth=(user, password))
with driver.session() as session:
    with open(path, "r") as file:
        contenu = file.read()
        requetes = contenu.split('\n')
        for i in tqdm(range(len(requetes))):
            session.run(requetes[i])
```

4.1.3 Visualisation

Voici un exemple d'un sous graph obtenu du graph sur le contenu.

On voit bien que chaque noeud contient le text du chunk ainsi que des embeddings pour répondre aux questions par la suite.

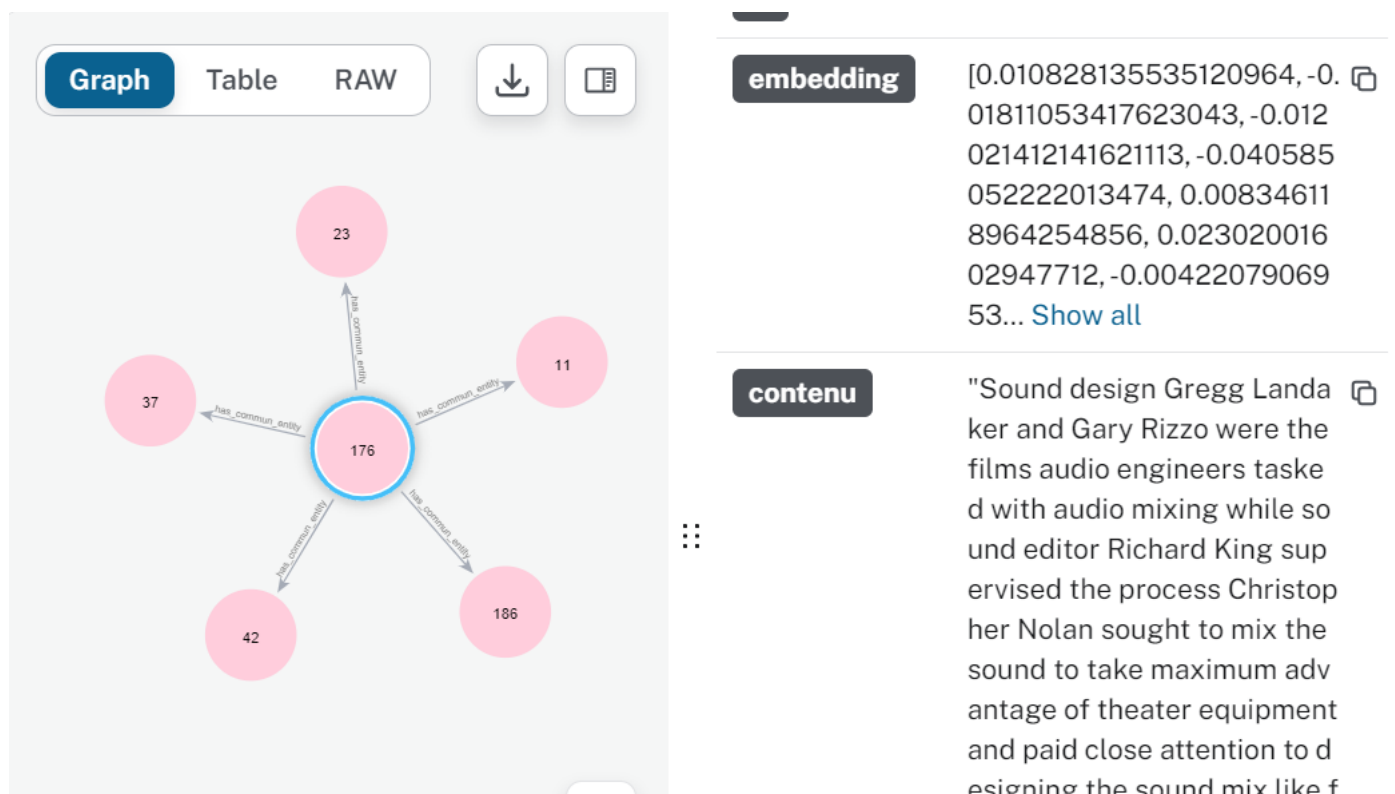


Figure 4: Exemple d'un sous graph du graph contenu

4.2 Création du graphe de structure

Dans cette partie, le défi c'était de créer un graphe pour représenter au mieux les différentes relations que l'on pourrait extraire de nos documents.

4.2.1 Extraction des entités/rerelations

La première étape consiste à extraire à partir des documents textes les entités/rerelations que l'on voudrait représenter, pour cela, on a construit une liste manuellement, des triplets que l'on voudrait extraire, et qui représenterait le mieux les informations nécessaires de nos documents:

- (Movie title; Directed_by; Director)
- (Movie title; Written_by; Writer1)
- (Movie title; Written_by; Writer2)
- (Movie title; Written_by; WriterN)
- (Movie title; Produced_by; Producer person)
- (Movie title; Edited_by; Editor)
- (Movie title; Production_company; company)
- (Movie title; Distributed_by ; distributer)
- (Movie title; Release_date; date)
- (Movie title; Cast; Actor)

Pour l'extraction, nous nous sommes servis de la puissance des llm, plus précisément GPT-3.5.

Pour cela nous avons écrit un prompt, avec un assistant comportant le texte pour lesquels les extractions doivent être faites, ainsi que la description des triplets. Pour demander ensuite à GPT, d'extraire à partir du texte les triplets décrits.

```
def extract_entity_relation(page_text):
    relations = """
    Ontology of triplets: (object; relation; object) with this relations
    (Movie title; Directed_by; Director)
    (Movie title; Written_by; Writer1)
    (Movie title; Written_by; Writer2)
    .
    .
    .
    (Movie title; Written_by; WriterN)
    (Movie title; Produced_by; Producer person)
    (Movie title; Edited_by; Editor)
    (Movie title; Production_company; company)
    (Movie title; Distributed_by ; distributer)
    (Movie title; Release_date; date)
    (Movie title; Cast; Actor)
```

```

"""
query_result = ""
assistant = "Here is the text: \n" + page_text + "\n and Here is the
    triplets we want to extract for this texte: " + relations
question = "Extract triplets from the text using the provided ontology.
    Replace the movie title, by the movie name as an entity. \n As response,
    only give the list of the triplets"

messages = [ {"role": "system", "content": "You are an expert in Extracting
    Knowledge Graph relation"},
    {"role": "assistant", "content": assistant},
    {"role": "user", "content": question},
    ]

response = client.chat.completions.create(
    model="gpt-3.5-turbo",
    messages=messages,
    temperature=0
)

```

Le résultat est en format texte, et représente les différents triplets extrait du texte.

4.2.2 Construction du graphe

La partie suivante consiste à utiliser les triplets extrait dans l'étape précédente pour contruire un graphe Neo4J. Pour commencer on a instancier une connexion de notre Code à la base de données Neo4J crée au préalable.

Nous avons ensuite écrit un script Python, qui pour chaque ligne (triplet), crée les noeuds correspondant, et la relation.

Pour savoir quel label donner à nos noeuds et relations, nous suivons certaines règles:

- Le noeud qui se trouve dans la première position du triplet est toujours de type Movie. avec une propriété "title"
- Si la relations du triplet est: 'Directed_by', 'Written_by', 'Produced_by', 'Edited_by', 'Cast', le noeud à la deuxième position du triplet est de type personne. avec une propriété "name"
- Si la relations du triplet est: 'Edited_by', 'Production_company', 'Distributed_by', la relation est de type Company avec une propriété "name"
- Si la relations du triplet est: 'Release_date', le noeud est de type Date, avec une propriété date.

La figure suivante représente la requete cypher qui sera envoyé pour chaque triplet

```

session.run(
    "MERGE (m:Movie {title: $movie})"
    "MERGE (p:" + type_relation+ " {" +property_name+": $value})"

```

```
"MERGE (m)<-[:%s]-(p)" % relation,  
movie=movie,  
value=value)
```

- \$movie: Le nom du film
- type_relation le type du noeud, dépendant de la relation
- property_name: la propriété à rajouter

4.2.3 Visualisation

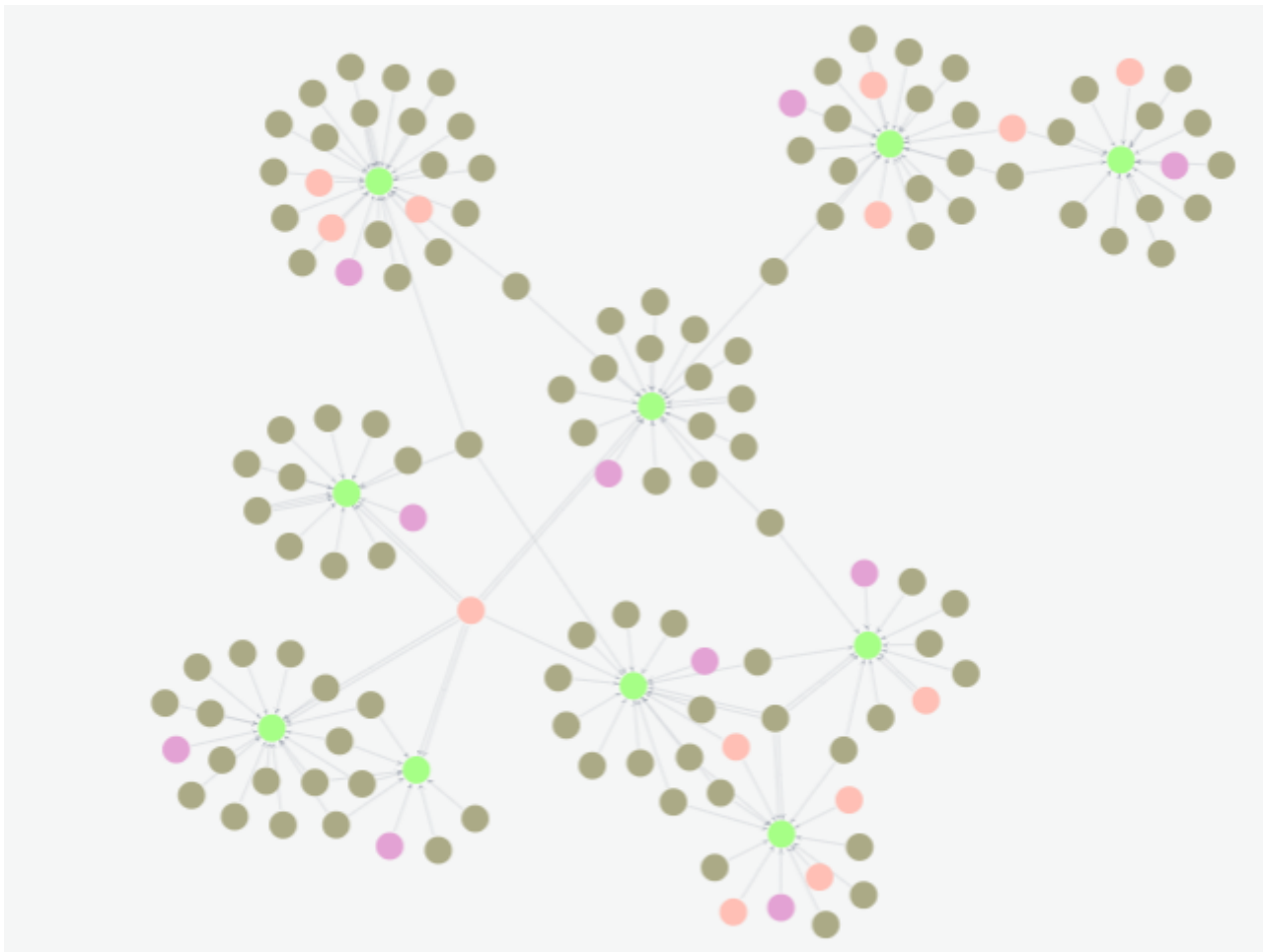


Figure 5: Le graphe Structure

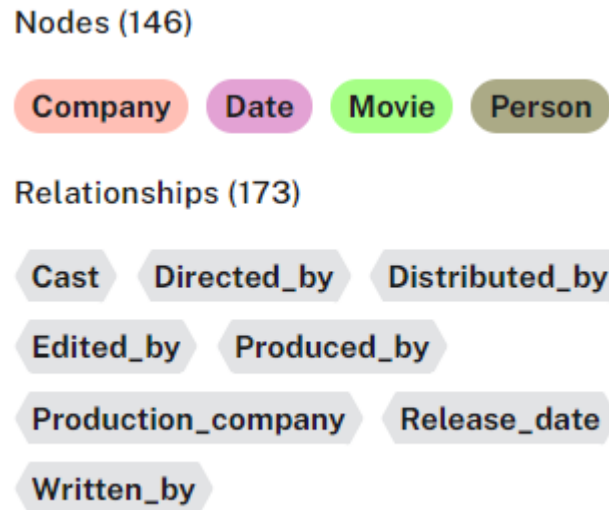


Figure 6: Labels du graphe Structure

5 Requêtage sur graphe

5.1 Requêtage sur le graphe contenu

Pour le requêtage du graph sur le contenu des document, nous avons imaginé une méthode qui se base sur la structure du graph pour recoller les informations necessaire à une question et l'utilisation de LLM pour formuler une réponse à partir de ces informations.

Notre approche ce divise en 3 étapes:

5.1.1 Creations des embeddings de noeuds

La première étape consiste à attribuer a chaque noeud du graph un vecteur d'embeddings qui permettra après de recoller les noeuds les plus pertients pour répondre à une question. les embeddings sont construits en utilisant la bibliothèque langchain et OpenAIEmbeddings.

```
vector_index = Neo4jVector.from_existing_graph(  
    OpenAIEmbeddings(),  
    url=url,  
    username=username,  
    password=password,  
    index_name='tasks',  
    node_label="Content",  
    text_node_properties=['contenu'],  
    embedding_node_property='embedding',  
)
```

5.1.2 Exploration du graph pour la récolte d'informations

Dans le seconde étape, on va commencer par rechercher les noeuds qui ont les embeddings les plus proches de la question.

ensuite pour chaque noeud retourné par l'étape précédente, on utilise le graph pour voir les noeuds voisins qui ont des entités en commun avec ce noeud la.
ceci va permettre d'avoir le maximum d'informations pour répondre à la question et meme de répondre à des questions sur différents documents.

```
def voisin_reponses(question,vector_index):
    response = vector_index.similarity_search(question)
    reponses = []
    for res in response:
        texte_recherche = res.page_content.split("contenu: ")[1]
        reponses.append(texte_recherche)
        cypher_query = f"MATCH (c:Content)-[:has_commun_entity]->(e:Content)
            WHERE c.contenu = '{texte_recherche}' RETURN DISTINCT e.contenu as
            contenu;"
        resultats = run_cypher_query(cypher_query)
        for record in resultats[:10]:
            reponses.append(record["contenu"])
    return list(set(reponses))
```

5.1.3 Utilisation du LLM pour générer la réponse

La dernière étape consiste à donner les informations récoltés lors de l'étape précédente à un LLM. Dans le prompt de ce dernier on lui précise que ces informations sont des facts et qu'il servent à répondre à la question qu'on lui transmet aussi.

```
def reponse_question(question,vector_index, client):
    facts = voisin_reponses(question,vector_index)
    assistant = "here's a set of facts: " + str(facts)
    prompt_template = """
        give me an answer to this question based only on the set of facts: {}.
        Some fo these facts may not contain information necessary to the answer,
        you can ignore them.
        Dont add information from your knowledge, if you dont have the
        information in the set of facts dont add answer from your own.
    """.format(question)

    messages = [
        {"role": "user", "content": prompt_template},
        {"role": "assistant", "content": assistant},
    ]

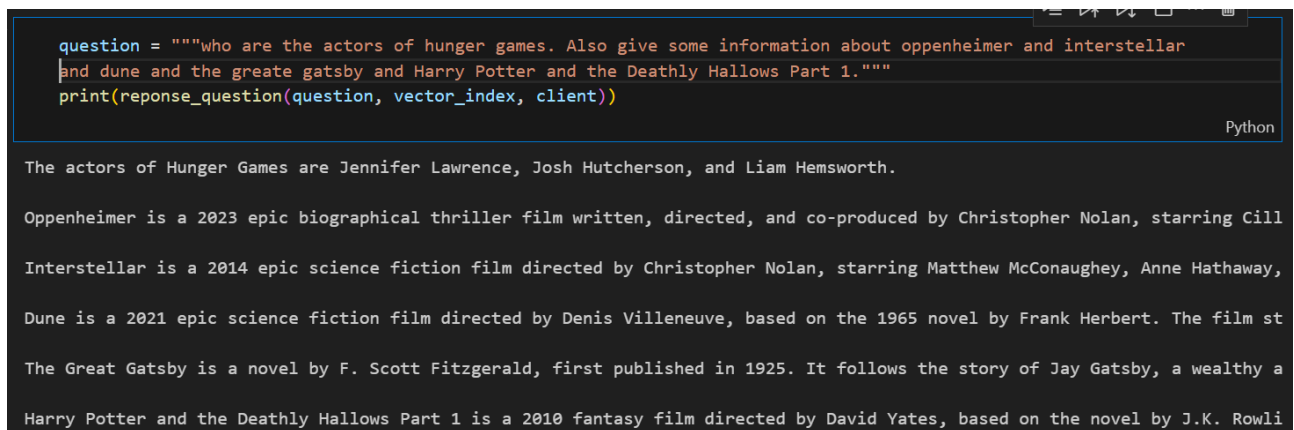
    response = client.chat.completions.create(
        model="gpt-3.5-turbo",
        messages=messages,
        temperature=0
    )
    return response.choices[0].message.content
```

5.1.4 Expérimentations

Pour tester la capacité de notre modèle à générer des réponses à une question qui nécessite plusieurs documents on posé la question suivante:

```
question = """who are the actors of hunger games. Also give some information  
about oppenheimer and interstellar  
and dune and the greates gatsby and Harry Potter and the Deathly Hallows Part  
1."""
```

Voici la réponse générée par notre méthode:



```
question = """who are the actors of hunger games. Also give some information about oppenheimer and interstellar  
and dune and the greates gatsby and Harry Potter and the Deathly Hallows Part 1."""  
print(reponse_question(question, vector_index, client))
```

The actors of Hunger Games are Jennifer Lawrence, Josh Hutcherson, and Liam Hemsworth.

Oppenheimer is a 2023 epic biographical thriller film written, directed, and co-produced by Christopher Nolan, starring Cill

Interstellar is a 2014 epic science fiction film directed by Christopher Nolan, starring Matthew McConaughey, Anne Hathaway,

Dune is a 2021 epic science fiction film directed by Denis Villeneuve, based on the 1965 novel by Frank Herbert. The film st

The Great Gatsby is a novel by F. Scott Fitzgerald, first published in 1925. It follows the story of Jay Gatsby, a wealthy a

Harry Potter and the Deathly Hallows Part 1 is a 2010 fantasy film directed by David Yates, based on the novel by J.K. Rowli

Figure 7: Question sur le contenu

On remarque que la méthode arrive à récupérer les informations nécessaires contenu dans plusieurs documents et générer une réponse cohérente.

5.2 Requêtage sur le graphe structure

L'objectif de cette partie est de pouvoir répondre à la question d'un utilisateur, en prenant compte le graphe sur la structure contenu au préalable, le but étant d'extraire des informations qui requièrent des données structurées pour répondre à l'utilisateur, un exemple serait, des questions faisant appels à des calculs d'agrégats comme le count, l'average etc... Pour cela, après que l'utilisateur ait posé sa question, nous utilisons GPT-3.5, en passant par ces étapes:

5.2.1 Extraction de schéma

Pour que le llm puisse nous générer une requête Cypher répondant à la question d'utilisateur, il doit connaître le schéma du graphe que nous utilisons, pour cela, nous effectuons trois requêtes cypher pour extraire les informations nécessaires sur le schéma du graphe.

```
schema_visualisation = graph.query("call db.schema.visualization")  
rel_properties = graph.query("call db.schema.relTypeProperties")  
node_properties = graph.query("call db.schema.nodeTypeProperties")
```

Le résultat sera ensuite utilisé comme assistant pour le llm.

5.2.2 Génération de la requete Cypher

Pour cette partie, nous avons formulé un prompt, en utilisant le schéma comme assistant, ainsi qu'on procurant la question de l'utilisateur à GPT-3.5, à fin qu'il puisse générer une requete Cypher répondant aux besoins de la question.

```
assistant = "Here is the graph schema: " + str(schema_visualisation) + "\n
Here is the properties of the relations of the graph: " + str(
rel_properties) + "\n Here is the properties of the nodes of the graph:" +
str(node_properties)

prompt_template = """
give me a cypher Query to response to this:
Question: {}
""".format(question)
messages = [ {"role": "system", "content": "You are an expert in generating
Cypher Query"},
{"role": "user", "content": prompt_template},
{"role": "assistant", "content": assistant}
]

response = client.chat.completions.create(
model="gpt-3.5-turbo",
messages=messages,
temperature=0
)
print(response.choices[0].message.content)
```

5.2.3 Envoie de la requete cypher

Le résultat étant en format texte, le but de cette partie est d'extraire la requete, ou les requetes en question de l'output de GTP-3.5. Pour cela nous avons utilisé une expression régulière, car, dans la sortie de GPT-3.5, les requetes sont généralement données entre triple backticks ("`"), donc nous extrayons le texte, ou les textes qui se trouvent entre triple backticks, pour les enregistrer dans une listes de requetes.

Nous envoyons ensuite les requetes résultantes, et nous enregistrant le résultat en chaines de caractère pour la suite du processus.

5.2.4 Formulation de la réponse

Pour cette partie, après qu'on ai extrait tout les information du graphe de structure, nous les utiliseront pour formuler une réponse en langage humain pour l'utilisateur, pour cela nous formulons un prompt à GPT-3.5, ou nous lui donnons comme assistant un set of facts, qui rassemblent les informations extraite du graphe, et nous lui demandons de formuler une réponse à partir de cela.

```
assistant = "This is the set of facts for the result of the question: \n" +
query_result
print(assistant)
```

```

question2 = "Generate a response to this question, using the given set of
    facts " + question
print(question2)
messages = [{"role" : "system", "content" : "You are an expert in Answering
    generation using ONLY the given set of facts."},
    {"role": "user", "content": question2 },
    {"role": "assistant", "content": assistant}

    ]
response = client.chat.completions.create(
    model="gpt-3.5-turbo",
    messages=messages,
    temperature=0
)
return ("La reponse la question est: " + response.choices[0].message.
    content)

```

5.2.5 Expérimentation

Voici quelques exemple de question/réponses:

Question = **Tell me about Spiderman, and the cast of the film Far From Home**

La requete généré par GPT:

```

```cypher
MATCH (m:Movie)-[:Cast]->(p:Person)
WHERE m.title CONTAINS 'Spiderman' OR m.title CONTAINS 'Far From Home'
RETURN m.title AS Movie, COLLECT(p.name) AS Cast
```

```

Figure 8: Cypher request

Le résultat de la requete est:

['Movie': 'Spider-Man: Far From Home', 'Cast': ['Zendaya', 'Tom Holland', 'Samuel L. Jackson', 'Cobie Smulders', 'Jon Favreau', 'J. B. Smoove', 'Jacob Batalon', 'Martin Starr', 'Tony Revolori', 'Marisa Tomei', 'Jake Gyllenhaal']]

La réponse à la question est:

In the film "Spider-Man: Far From Home," the cast includes Zendaya, Tom Holland, Samuel L. Jackson, Cobie Smulders, Jon Favreau, J. B. Smoove, Jacob Batalon, Martin Starr, Tony Revolori, Marisa Tomei, and Jake Gyllenhaal.

On voit que pour cette partie, le llm ne réussit qu'à répondre aux question qui concerne la structure du graphe, mais pas au contenu des documents.

5.3 Requête final

Le but de cette partie, est de rassembler, les deux approches de requête décrites au dessus, c'est à dire extraire le maximum d'information sur le contenu des documents, et sur la structure des documents, à fin de formuler une réponse final.

Pour cela, après avoir lancé, les deux appels à GPT-3.5, pour avoir 2 réponses différentes pour la question, une sur le contenu, l'autre sur la structure, nous rassemblons les deux réponses comme entrée pour un nouveau prompt à GPT, en lui demandant, à partir de ces 2 réponses, formule une réponse final qui rassemble les 2 réponses.

```
answer1 = questionAnswering(question, graph, client)
answer2 = reponse_question(question,vector_index, client, graph2)

question2 = "Generate a response to this question, using the given set of
            facts: \n" + answer1 + "\n" + answer2
print(question2)

messages = [{"role" : "system", "content" : "You are an expert in Answering
            generation using ONLY the given set of facts."},
            {"role": "user", "content": question2 },
            ]
response = client.chat.completions.create(
    model="gpt-3.5-turbo",
    messages=messages,
    temperature=0
)
```

6 Création du chatbot

6.1 Streamlit

Nous avons conçu notre chatbot à partir de la bibliothèque streamlit. Streamlit est une bibliothèque open source en Python qui permet de créer rapidement des applications web interactives pour l'analyse de données et la visualisation.

6.2 Interface du chatbot

GraphBot



what are the movies in which Zendaya has appeared ?



Based on the given set of facts, Zendaya has appeared in the following movies:

- Spider-Man: Homecoming
- Spider-Man: Far From Home
- Spider-Man: No Way Home

Ask a question:



Figure 9: Interface du chatbot