Data Science Mission

Graph Databases and Ontology for Recruitment Platform

Entreprise tutors: Marie Joubert, Julien Capitaine

Academic tutors: Franck JAOTOMBO, Imène BRIGUI-CHTIOUI

Group members:

Zebing LYU

Yaohui Wang

Armando Ponce Sesma

Meri Martirosyan

Venkata Saketh Teki

early makers

em lyon business school

datacraft

Table of Contents

Introduction

The data science mission aims to bring together a variety of skills in the ambition to work with a real business case in a professional setting. This report documents the results of a data science project that aimed to create a network of newly graduated masters and PhD candidates, and identify their research interests through an analysis of their publications. The project was a collaborative effort of a team of students with diverse backgrounds and skills, working together to address a real business case in a professional setting.

The report outlines the methodology used to extract data from a website and populate a graph database, creating nodes for various aspects of the candidates' profiles. One such node is the keyword node, which was populated using NLP techniques with cosine similarity calculations on batches of publication summaries. This enabled the team to identify new keywords related to AI, Data Science, and adjacent topics.

The project demonstrated the power of collaboration among students with different areas of expertise, and highlights the importance of utilising cutting-edge techniques and technologies to solve complex problems. The report concludes with an evaluation of the success of the project, and the potential for future applications in the field of data science.

This report shows our solutions to achieve this goal and contains three major parts in total including the literature review, business case review and technical solutions suggested by the team.

The literature review discusses some crucial methods and subjects that are conducted in the mission, which include data extraction methods, graph databases, Neo4j, NLP and cosine similarity. It gives a brief and comprehensive overview of tools and builds a theoretical framework.

In the business case part, we outline the business objectives of the mission, introduce Datacraft and our mentors, and describe how our methods contribute to business goal achievement. We highlight the work we have done so far, including building a scanning module, designing a graph database, and developing an ontology framework.

Then, the report elaborates the technical solutions. It provides an overview of the code and methodology we have developed, highlights the benefits of our approach and the key features of our solution.

However, while our solution is effective, we also acknowledge its limits and further recommendations.

I. Literature Review

The aim of this project was to develop a network of newly graduated Masters and PhD candidates based on their publications. To achieve this, we utilized a combination of web scraping tools, graph databases, natural language processing (NLP) techniques, and cosine similarity calculations.

The first stage of the project involved data extraction, where we explored various tools and techniques for web scraping, including their advantages and limitations. This allowed us to extract data about the newly graduated candidates from various websites, ensuring that we had a comprehensive dataset to work with.

Next, we examined the different graph databases available and evaluated their suitability for our project. We then chose to use Neo4j, a popular graph database that offers a range of features, including data modeling capabilities, a powerful query language, and scalability.

We then explored how NLP techniques could be used to extract data and build the database, with a focus on using cosine similarity calculations to identify and categorize keywords related to AI and data science. By utilizing NLP, we were able to extract more information about the candidates' publications and create a more comprehensive and accurate graph database.

Overall, this report will provide an in-depth analysis of our approach to developing a network of newly graduated Masters and PhD candidates using web scraping, graph databases, NLP techniques, and cosine similarity calculations. We will also discuss the advantages and limitations of each approach and evaluate the overall effectiveness of our methodology.

1. Data Extraction Stage

Data extraction is the process of retrieving data from various sources and transforming it into a useful format. It is a critical step in data management and analytics, as it lays the foundation for data analysis, data modelling, and machine learning. There are several ways to extract data from various sources, including through APIs, web scraping, and database queries.

1.1.1. API based extraction

API-based extraction is one of the most common methods used to extract data from various sources. An API (Application Programming Interface) is a set of protocols and tools that enable software applications to communicate with each other. APIs can be used to extract data from various web-based services such as social media platforms, e-commerce sites, weather services, and more. The data can be extracted in various formats such as JSON, XML, or CSV, depending on the API's specifications.

Extracting data through APIs is a straightforward and reliable method, as it provides structured and consistent data. It involves sending requests to a web-based service using a specific API. These use various HTTP methods such as GET, POST, PUT, DELETE, and more to perform specific tasks. In API-based data extraction, developers use the GET and POST methods to retrieve and manipulate data from the API.

The **GET** method is used to retrieve data from the API. A GET request sends a request to the API and expects a response containing the requested data. This method is useful when you want to retrieve data from a particular resource, such as a user profile, a product listing, or a weather forecast.

The **POST** method is used to send data to the API for manipulation or storage. A POST request sends data to the API in a structured format, such as JSON or XML, and expects a response indicating whether the operation was successful or not. This method is useful when you want to create new resources, update existing ones, or delete them.

1.1.2. Scrapping

Web scraping is another popular method of extracting data from websites. It involves using software tools to extract data from HTML pages by parsing the page's DOM (Document Object Model). Web scraping can be used to extract structured data such as product details, user reviews, and news articles. It can also be used to extract unstructured data such as text from blog posts, tweets, and forum discussions. Web scraping can be performed using tools such as *Beautiful Soup, Selenium, and Scrapy*. *However, it is not always a legal or ethical method of data extraction, as it may violate website terms of use and copyright laws.*

1.1.3. DB queries

Database queries are another method of data extraction, used primarily to extract data from relational databases. A relational database is a collection of data organised in tables, where each table represents a specific entity or data category. Database queries are written in Structured Query Language (SQL) and can be used to extract specific data based on various criteria, such as date user ID, name, gender, and more. Database queries can be used to extract large amounts of structured data quickly and efficiently, making it a popular method for business intelligence and data analytics.

These different methods allow us to obtain the data and create a first step of the flow towards the database. However, data storage is ensured by databases that have structural differences allowing us to satisfy a variety of business needs. According to the business case suggested, it is crucial to first determine whether the database used should be relational or non relational.

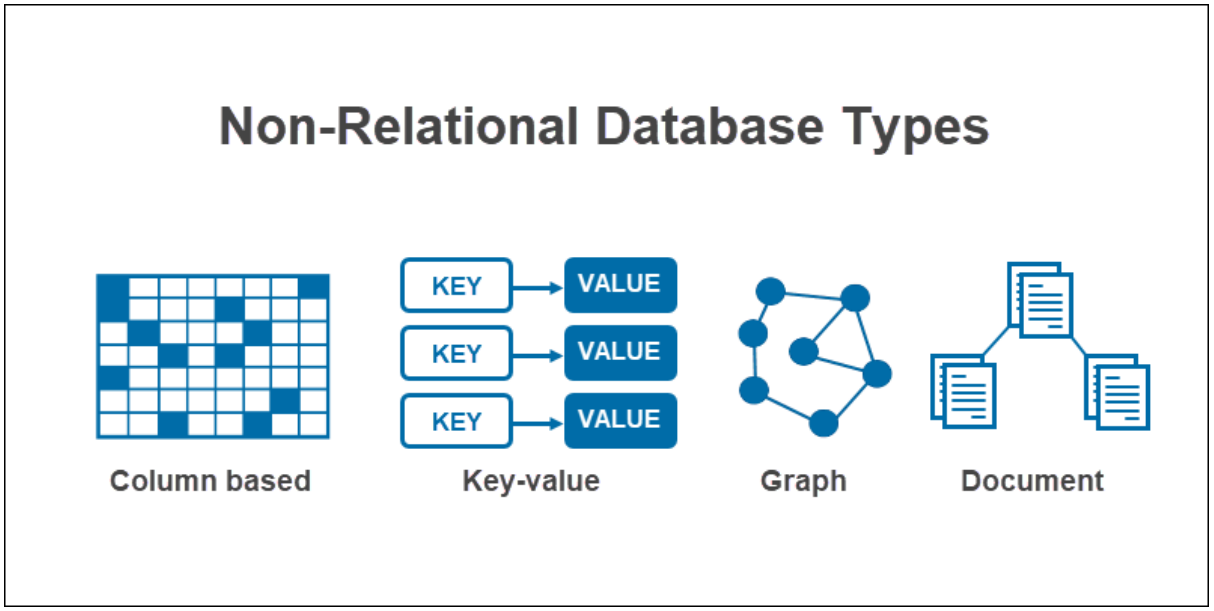
Relational and non-relational database management systems are used for storing and managing data. Relational databases have been the dominant type for many years, but non-relational databases have gained popularity in recent times.

Relational databases store data in a tabular format with rows and columns, and the relationship between data is defined by the primary and foreign keys. They are generally used for transactional systems that require a high degree of data consistency and data integrity. Relational databases have been around for decades and have been proven to be reliable, scalable, and secure. They are widely used in enterprise applications such as customer relationship management (CRM), enterprise resource planning (ERP), and human resource management (HRM) systems.

On the other hand, non-relational databases (also known as NoSQL databases) store data in a variety of formats such as key-value pairs, documents, graphs, and column families. They are used for handling large volumes of unstructured and semi-structured data, such as social media feeds, sensor data, and log files. Non-relational databases are known for their ability to scale horizontally, meaning that they can handle a large volume of data and traffic by adding more servers to the database cluster.

One of the major differences between relational and non-relational databases is the way they handle data consistency. Relational databases have a rigid structure that enforces data consistency, whereas non-relational databases are more flexible and allow for eventual consistency, which means that the data will eventually be consistent across all nodes in the database cluster.

In our case, we will go in depth into studying the non relational database as our use-case requires a DBMS that would easily handle inconsistent data and would be easily scalable. It is also particularly interesting to implement a graph database as the business problem is around creating a network, which means that the db has to handle nodes with multiple interconnected variables.



[1]

1.2. Graph Databases

Graph databases are a type of NoSQL database that store data in a graph-like structure, consisting of nodes, edges, and properties. Unlike traditional relational databases, which store data in tables, graph databases represent data as a network of nodes and edges, making them well-suited for storing and querying highly interconnected data. In a graph database, nodes represent entities, such as people, places, or objects (such as the authors, institutions and publications in our case), and edges represent the relationships between those entities. Each

[1] https://phoenixnap.com/kb/database-types

node and edge can also have properties associated with it, such as a user's name, age, or location.

One of the main advantages of a graph database is their ability to efficiently handle complex and highly connected data. Because data is represented as a graph, queries that involve traversing multiple nodes and edges can be performed quickly and easily. This makes graph databases particularly useful for applications such as social networks, recommendation engines, or fraud detection systems. Another advantage of graph databases is their flexibility. Unlike relational databases, which require a predefined schema, graph databases can be updated and modified on the go, making them well-suited for applications where data structures are subject to change.

Graph databases are also highly scalable, as they can easily handle large and growing data sets. They can be horizontally scaled across multiple servers, and can also be optimised for specific use cases, such as read-heavy or write-heavy workloads.

1.2.1. Coherence of graph databases for interconnected multi-variable data

Graph databases have emerged as a powerful alternative to traditional relational databases when it comes to managing complex data structures with numerous connected variables. Traditional relational databases require complex join operations to retrieve data that is spread across multiple tables, which can become slow and inefficient as the number of connections between entities increases, which would typically be the case when working on a business case requiring creation of a network of variables.Hence the advantage of graph databases is their ability to handle complex queries and analytics.

With the ability to represent complex relationships between entities, graph databases can capture a wide range of data insights that might be difficult or impossible to capture using traditional databases. For example, graph databases can help identify key influencers or hubs within a network, or uncover patterns of behaviour and interaction that might not be apparent using other types of databases. Because they are schema-less, graph databases can easily accommodate changes to the data model without requiring significant modifications to the underlying database schema. Moreover, because nodes and edges can be added or removed as

needed, graph databases can scale to handle large and complex datasets without sacrificing performance or functionality.

Finally, graph databases are well-suited for creating a network-like infrastructure because they provide a natural way to model complex networks of relationships. This can be particularly useful in use cases such as ours, where understanding the connections between people, groups, and interests is critical to creating a successful platform. Similarly, in fields such as logistics and supply chain management, graph databases can be used to model the relationships between suppliers, products, and customers, enabling more efficient and effective management of the supply chain.

1.2.2. Neo4j

Neo4j is a popular graph database system that is widely used for creating databases with numerous connections and variables. It is a highly scalable, high-performance system that is designed to efficiently store and query large amounts of interconnected data.

Another important feature of Neo4j is its support for the Cypher query language. Cypher is a declarative query language that is designed specifically for querying graph databases. It allows developers to easily navigate and query the graph, and supports complex queries that can uncover hidden patterns and insights in the data. Neo4j also provides a number of advanced features that make it particularly well-suited for creating databases with numerous connections and variables. As described above, the database does not need a rigid schema and can be easily modified.

Another important feature of Neo4j is its support for clustering and replication. Clustering allows multiple instances of the database to be run in parallel, providing increased performance and fault tolerance. Replication allows data to be replicated across multiple nodes, providing increased data availability and resilience in the event of a node failure.

In addition, Neo4j provides a number of tools and integrations that make it easy to integrate with other systems and tools. For example, it provides integrations with popular data visualisation tools such as Tableau and Gephi, as well as with popular programming languages such as Java, Python, and .NET.

In order to further advance in the review, It is Important to understand how it is possible to automatise the extraction of data from the API and enhance the flow with accurate data from the data source. For this, we will proceed to studying some of the

tools used in deep learning and how it is possible to include deep learning and Natural Language Processing tools in data extraction in order to automate this process.

1.2.3. NLP

Natural language processing (NLP) techniques, particularly those that use transformer-based models, can be used to extract keywords from unstructured text data. In case of a summary, it is possible to use transformers to vectorise the text and teach the model to understand the importance of the given keywords.

Transformers

Sentence transformers are a type of transformer-based model that is designed to generate sentence embeddings, i.e., fixed-length numerical representations of input sentences that capture their semantic meanings.

To vectorize a corpus of text using sentence transformers, the first step is to tokenize the text, i.e., break it up into individual words and punctuation marks. This is typically done using a tokenizer, such as the tokenizer provided by the Hugging Face library[2].

Once the text has been tokenized, it can be fed into a sentence transformer model, such as BERT or RoBERTa, which will encode each sentence as a vector in a high-dimensional space. These vectors are known as sentence embeddings. The version of BERT or RoBERTa used for the creating the embeddings depends on different parameters such as the size of the corpus, the type of text and further modelling done on the text.

The model has 6 layers, 768 dimensions and 12 heads, totaling 82M parameters (compared to 125M parameters for RoBERTa-base). On average DistilRoBERTa is twice as fast as Roberta-base.

[2] https://www.sbert.net

1.3. Cosine Similarity

To retrieve the meaning of the text, the sentence embeddings can be used to compute cosine similarities between different sentences or documents. Cosine similarity is a measure of the similarity between two vectors, which ranges from -1 to 1, with 1 indicating complete similarity and -1 indicating complete dissimilarity. By comparing the cosine similarities between different sentences or documents, it is possible to identify those that are most similar in meaning.

It is interesting to use these techniques to identify similar topics for feeding and expanding the database. These techniques can be particularly useful in creating a network graph database that represents the relationships between entities in a corpus of text.

The first step in using NLP with transformers to extract keywords is to preprocess the text data to remove noise and irrelevant information. This can include techniques such as tokenization, stop word removal, and stemming. Once the text data has been preprocessed, it can be fed into a transformer-based model, such as BERT or GPT-2, to extract keywords and topics.

After the keywords and topics have been extracted, cosine similarity can be used to identify similar topics and group them together. This can be particularly useful in creating a network graph database where entities are connected by similar topics or keywords. For example, companies can use this technology to identify articles that are related to a particular topic or field of study and represent them in the network graph as nodes that are connected by edges representing the similarity between their topics.

Using NLP with transformers to extract keywords and identify similar topics can also be useful in analysing the structure of a network graph database. Companies can use this technique to identify clusters of nodes that are connected by similar topics, which may represent sub-communities or subtopics within the larger network. This can help companies to better understand the structure and dynamics of the network and identify key nodes or entities that play important roles within it.

In conclusion, this literature review and brief overview of subjects studied in this project allow us to build an idea on the scope of research we conducted in order to work on our solution. The specificity of our use case suggested an important work on the adaptation of

these tools to our business setting. Further in this paper, we will define the business case and how we applied these tools to find a solution to the suggested problem.

II. Business case of our Data Science Mission

The long term goal of this project is to create a platform that connects junior AI experts, such as Master 2 level graduates and young PhDs, for companies looking to hire. The platform will have two sections: one for employers and the other for candidates. The platform will act as an intermediary between these two groups and select candidates based on their qualifications, skills, and personal preferences. The platform will be supported by the datacraft Club, and the junior talents who are recruited will be part of a community of AI and data experts, where they can receive expert mentoring.

To achieve this goal, datacraft assigned a project to EM Lyon students (our team), to work on a basic block of the platform, specifically focusing on the sourcing of PhD candidates. They will use the scanR website, which is an open data resource developed by the French Ministry of Higher Education and Research. This website contains over three million theses, publications, and patents, as well as a search engine that enables users to explore and describe the scientific research and innovation landscape in France. The website also provides information on companies, research laboratories, research authors, funding, and links all these entities together. This resource will provide a wealth of information that can be used to build the platform's database and create a detailed map of the scientific AI ecosystem in France.

2.1. Established objectives by Datacraft

1) Build a scanR scraping module in Python (required):
    a) the module should be able to scrape scanR segments specifically for keywords related to all AI domains.
    b) it should be p (daily, weekly, or monthly) to update the platform database,
        I. the idea here is to select only those resources that have been added to scanR since the previous import, regardless of the date of that import,

II. but it is also necessary to provide content checking routines on imported dates, to address all resources that have been made available in a staggered manner.

c) and it will have to store the imported data on a graph database designed during the second objective.

2) Design of a graph database:
   a) it should be optimised and resilient to the growing volume of data (first to absorb new publications from the French ecosystem, then to scale up to a global one),
   b) as far as possible, the database should be secure and durable and should benefit from security back-ups.
3) Work on tools (preferably in Python) for querying and visualising the database built to explore and analyse the French ecosystem:
   a) The tool should notably allow the isolation of the most recent publications to identify PhD students before their graduation and thus their arrival on the market.

2.2. Getting to know Datacraft

Datacraft is a learning & coworking Club for data scientists. It allows data scientists and data engineers to share best practices and train with their peers. Datacraft enables cross-fertilization and helps accelerate the implementation of data and artificial intelligence projects. The first Base opened in February 2020 in Paris, gathering pioneer member companies. Datacraft is also hosting more than 600 high level data science researchers and freelance in residence. Their mission is to contribute to an efficient and responsible use of data.

2.3. Datacraft mentors

We had the privilege of working with two outstanding mentors on this "Data Science Mission" project, Marie Joubert and Julien Capitaine. Marie, the Head of the Datacraft Recruitment Platform Project, was always available to provide guidance and support. Julien, a Data Scientist at Datacraft, was a great mentor who helped us navigate the challenges we

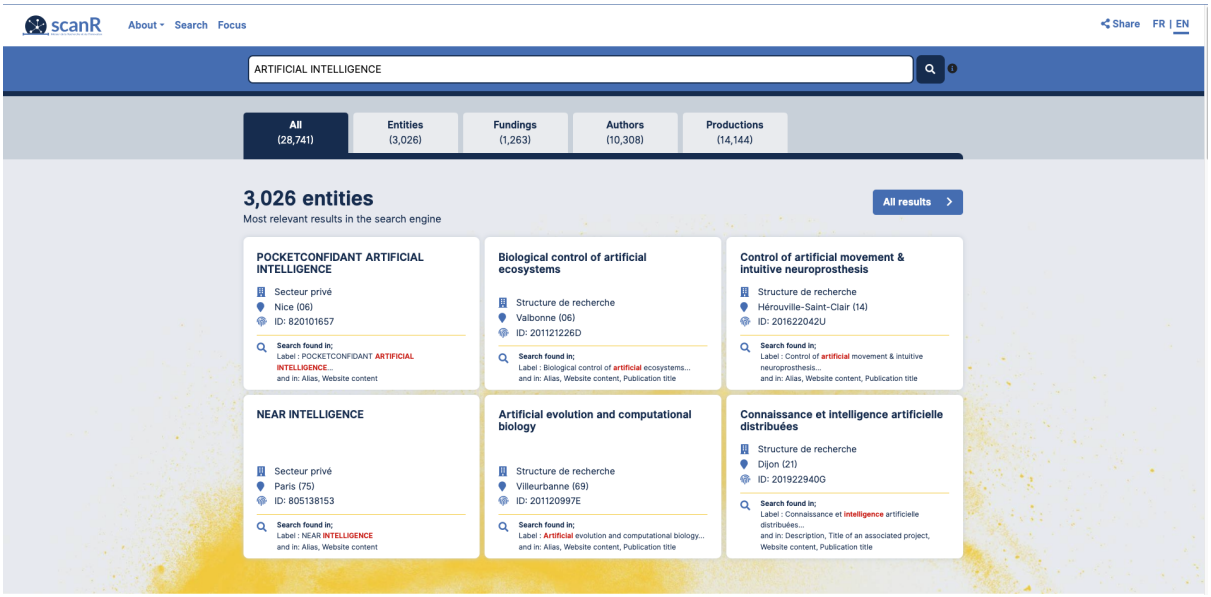
faced during the project and provided us with valuable insights. Together, their support and expertise were instrumental in helping our team achieve success.

2.4. Working on the objectives

2.4.1. Build a scanR scraping module in Python

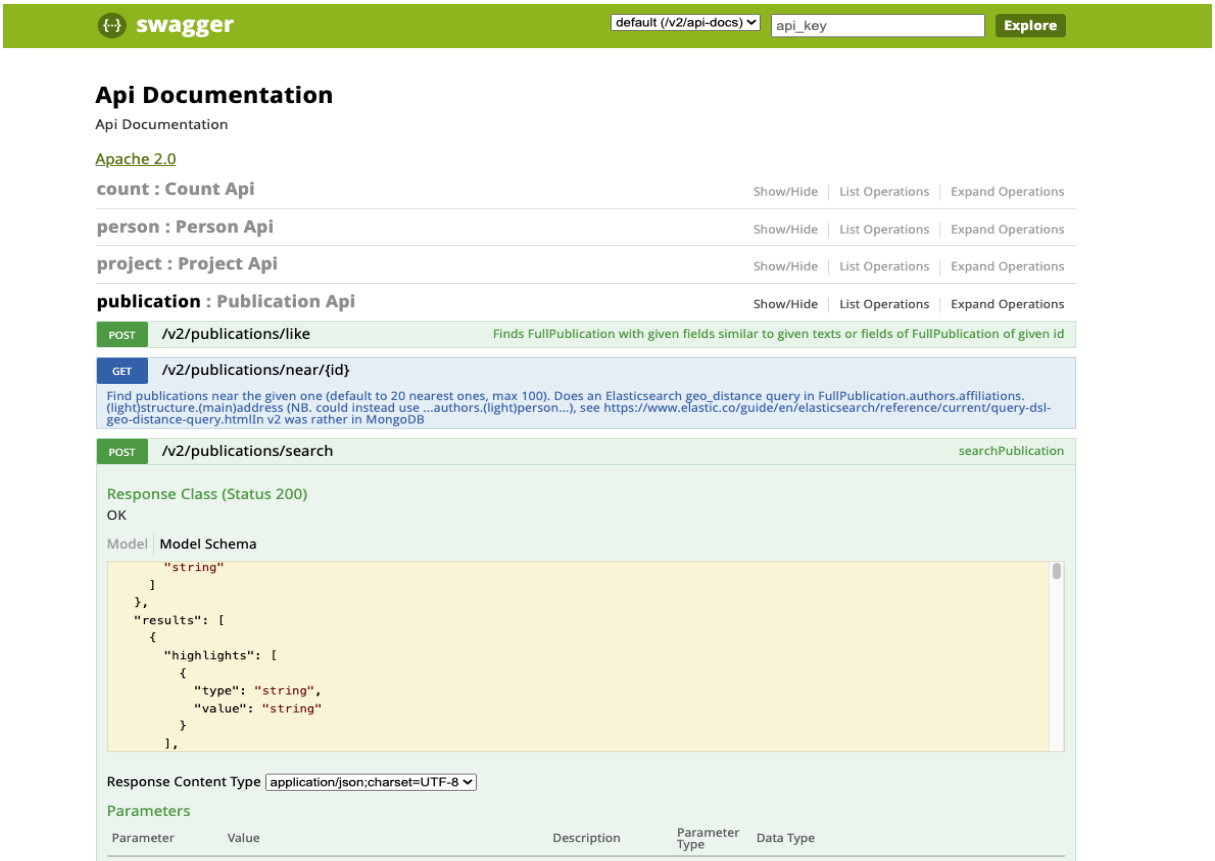
To achieve our first objective, we began by exploring the scanR website (https://scanr.enseignementsup-recherche.gouv.fr/), which serves as a useful tool for investigating the research and innovation scene in France. The primary goal of this website is to promote the work of individuals and organisations involved in research and innovation in France and to help comprehend their roles in the process.

The scanR website focuses on four key "Objects": entities, projects, persons, and productions, which comprise publications, PhD theses, and patents. Among these, scanR covers all research structures mentioned in the National Directory of Research Structures (RNSR) and their corresponding institutions. It also comprehensively covers public or private institutions, whether they are for-profit or not, mentioned in the primary sources utilised. Furthermore, scanR covers a collection of "Projects" that correspond to research works that have received funding from the French government or not, provided that an open and reusable source is available. Finally, scanR covers authors with an Idref identifier, for whom at least one connection to another source such as RNSR, production, or project, could be established.



After becoming acquainted with the scanR website, we delved deeper into its capabilities by exploring the application programming interface (API) provided by the platform.

This API (https://scanr-api.enseignementsup-recherche.gouv.fr/api/swagger-ui.html#) enabled us to connect to the scanR database and extract data from it. By utilising the API, we were able to access the information contained within scanR and use it for our business case.



Now with the understanding of the API we were able to create our python script that is able to retrieve data from the API and works by using a search request parameter for a given set of keywords. The code then extracts the data for each keyword and inserts it into a database using a connection module. The code utilises the "requests" and "json" modules to make requests to the API and process the data returned.

2.4.2. Design of a graph database

To fulfil our objective of designing a graph database, we selected Neo4j as our preferred choice due to our team's familiarity with the platform and our confidence that it would align well with the database requirements.

To explain why we chose Neo4j, it's important to first understand the fundamental concept of a graph database. In simple terms, a graph database utilises a graph structure to store data, allowing for semantic queries through the use of nodes, relationships, and properties. A graph

database differs from a traditional database because it does not store data in rows and columns.

Neo4j is an open-source, NoSQL native graph database. Unlike other graph databases that may use a "graph abstraction" on top of other technologies, Neo4j stores data in the same way it is visualised on a whiteboard. This means that the storage level is optimised for graph structures, providing unmatched performance and flexibility. As a powerful graph database management system, Neo4j offers several advantages over traditional relational databases. One of its key benefits is its ability to handle highly interconnected data, which is increasingly common in today's applications. By using nodes and edges to represent data relationships, Neo4j makes it easy to model complex networks and query them efficiently. This flexibility is especially useful in applications such as social networks as well as recommendation engines.

Neo4j uses cypher, which is a declarative query language similar to SQL, but optimised for graphs. It allows for complex queries to be expressed concisely, making it easy for developers to work with the database. Additionally, Neo4j can be easily integrated with Python.

Overall, Neo4j provides a powerful tool for managing complex data relationships and making sense of large datasets. Its native graph model and its integration with Python, made it appealing for us to explore the data that we obtained from the API.

2.4.3. Ontology objective

To enhance our search results and to be able to visualise all the AI professionals available on the scanR webpage, we implemented a new module that updates our API requests. This module utilises Natural Language Processing (NLP) techniques to extract relevant keywords from the summaries of previous search results. By analysing the text, the NLP module identifies and extracts keywords that are closely related to AI, which later can be introduced into the search requests that are submitted to the API, thereby improving the accuracy of our subsequent searches. Besides, this approach not only improves the accuracy of our searches but also helps to eliminate irrelevant results and false positives. This is particularly important

as the platform grows and expands to include more candidates and companies, as it ensures that both parties can find the best matches without wasting time sifting through irrelevant data. With this new approach, we are confident that we can obtain more relevant results for our case and provide a more complete view of the AI professionals available on the webpage.

III. Description of our solution and technical overview of the script

In this part of this report, we will provide a detailed description of the script used for data extraction and graph network creation, as well as a discussion of the limits of the project. This section includes an examination of our approach to connecting our database with Neo4j, as well as our use of NLP techniques to extract keywords and enrich the ontology of our dataset.We begin with a description of the script used for data extraction and graph network creation, highlighting the key features and components of our approach. This section also includes a discussion of the challenges we faced in creating an effective and efficient script, including issues related to data drift and inconsistency. Next, we examine our use of NLP techniques for keyword extraction and ontology enrichment, discussing the benefits and limitations of this approach. We explore how our use of cosine similarity calculations helped to identify related keywords and create a more comprehensive and accurate dataset. Finally, we discuss the limits of the project and provide recommendations for further work. We highlight the challenges we faced in addressing issues related to data drift, inconsistency, and lack of accuracy metrics, and suggest strategies for mitigating these challenges in future iterations of the project. Overall, this section provides a comprehensive overview of our approach to data extraction and graph network creation and highlights the key areas for improvement in future iterations of the project.

3.1. Connection Module

A python module that defines a bunch of functions for interacting with a Neo4j graph database using the official neo4j library. The connection code includes functions that are mainly used for creating and retrieving nodes and relations in and from the Neo4j database.

These functions can be classified into two categories including Neo4j connection functions:

- Database connectivity functions
- Graph CRUD operation functions
- Data validation and consistency check functions

3.1.1 Database connectivity functions

est_connection - establishes a connection to a Neo4j database running on the local machine. It firstly uses the get_creds() function to get the database credentials and then creates a GraphDatabase.driver object with the appropriate URI and authentication information. If the connection is successful, it returns a session that can be used to execute Cypher queries on the database.

get_creds(key) is a function that is used to extract authentication credentials from a JSON file. The function extracts the username and password corresponding to the key provided using the dict.get() function. Then it returns both username and password.

These functions are used to establish a connection with the Neo4j database. The 'est_connection' function connects to a Neo4j database using the provided credentials and returns a session object, which can be used to execute Cypher queries.

3.1.2. **Graph CRUD operation functions**

These functions are used to create nodes in the Neo4j database. There are four different types of nodes that can be created in the database: Publication, Author, Affiliation, and Keyword. Each of these nodes is created using a separate function.

- **Update operations**

set_keyword_time(tx,keyword): This function updates the lastupdated property of the primary keyword node for a given keyword.

set_pub_flag is a function that sets the flag property of a Publication node in a Neo4j graph database to the string "False". The Cypher query looks for a Publication node with an id

property equal to the value of the p_id parameter. If such a node exists, the query sets its flag property to the string "False".

- **Create operations**

create_main_nodes is a Python function that creates Publication nodes and keyword nodes in a Neo4j graph database and creates relationships between them using a Cypher query executed by the Neo4j Python driver.

create_affiliation_nodes(tx,p_id,id,kind,label,address,city,country) generates nodes in the Neo4j database for an affiliation and a relationship between the affiliation and a publication. Because we want to establish the relationship between publications and their affiliation so that we can know what the specific area is one affiliation working on. Then we might be able to reach more relevant publications

create_author_nodes(tx,p_id,id,firstname,lastname,gender,role): This function creates nodes in the Neo4j database for authors and relationships between authors and publications. The function generates properties for author nodes, including the publication ID, author ID, author first name, author last name, author gender, and the role.

create_keynodes(tx,p_id,keyword,flag) establishes nodes in the Neo4j database for keywords and relationships between keywords and publications. The function takes several arguments, including the publication ID, the keyword, and a flag that specifies whether the keyword is a primary or secondary keyword.

create_network creates a network of nodes and relationships in a Neo4j graph database using a set of predefined functions.

The function takes two parameters. The first one is con_session which is an instance of the Connection class used to interact with the Neo4j database. The second one is keyword, that is the keyword used to label the nodes in the graph.

The function also accepts an arbitrary number of keyword arguments (**kwargs), which are used to specify the properties of the nodes in the graph.

The function first checks if the summary property exists in the kwargs dictionary and sets a flag accordingly. It then calls the time_items() function to convert the publicationDate and lastUpdated properties to Unix epoch timestamps in seconds.

If the flag is set, the function creates a Publication node and a keyword node using the create_main_nodes() and set_keyword_time() functions, respectively. If the authors or affiliations properties are specified, the function also creates corresponding nodes and relationships using the create_author_nodes() and create_affiliation_nodes() functions.

In summary, this function serves as a high-level interface to a Neo4j graph database and allows for the creation of nodes and relationships based on a set of predefined functions and input parameters.

- **Read operations**

get_keyword_nodes(tx,k_type,flag) retrieves nodes from the Neo4j database for a given keyword type (primary or secondary). Then it returns a list of keyword names and their last update times.

get_pub_summary(tx) extracts summaries for all publications in the Neo4j database that have a "True" flag. It returns a pandas DataFrame with columns for publication ID and publication summary.

get_pub_ids(tx,keyword) retrieves a list of publication IDs for publications in the Neo4j database that are associated with a given keyword. If the keyword argument is "all", it returns a list of all publication IDs in the database.

3.1.3. Data validation and consistency check functions

check_value_dict(x,y) is a function that checks if a dictionary has a certain key and returns the corresponding value if it exists.

check_authors is a function that takes a dictionary x as input and returns a list containing the values of the id, firstName, lastName, and gender keys of the dictionary. If any of these keys are missing, an empty string is returned in their place. Because if one author node lost its author ID and name, it's hard to establish relationships between the author and its other publications, which is easy to suffer from duplication and result in floating nodes.

check_summ_title is another function that takes a dictionary x as input and checks if it contains a key named default, en, or fr. If any of these keys are present, the value associated with the key is returned as a string. If none of the keys are present, an empty string is returned.

check_affiliations takes a dictionary x as input and returns a list containing the values of the id, kind, label, address, city, and country keys of the dictionary. If any of these keys are missing, an empty string is returned in their place.

time_items checks if the key y exists in the dictionary x using the get() method. If the key exists, it returns the corresponding value divided by 1000 and casted to an integer. This operation converts a Unix epoch timestamp in milliseconds to seconds. If the key does not exist, the function returns 0.

3.2. Extraction Script

This python script extracts data from ScanR's API and validates the data presence in the DB and creates the graph network, if necessary. The script imports the necessary libraries firstly, including requests, json, and connection (which contains a bunch of functions that we have already defined), and uses multiprocessing to speed up the data retrieval process.

```
def getdata(**kwargs):
    session = connection.est_connection()
```

The script defines a function called getdata() to get the data and invoke the graph network creation. Inside the function, the script sets up a connection with the Neo4j database for CRUD(create, read, update and delete) operations.

```
for keyword in kwargs.get('keywords_list'):

    url = 'https://scanr-api.enseignementsup-recherche.gouv.fr/api/v2/publications/search'

        page_no = 0
        page_size = 1
        fields_list = ["*"]
        search_fields = ["summary","title"]
        searchreq =
{"lang":"en","searchFields":search_fields,"query":keyword,"page":page_no,"pageSize":pa
ge_size,"sourceFields":fields_list,"filters":{"productionType":
{"type":"MultiValueSearchFilter","op":"any","values":["thesis","publication"]}}}
```

It loops through a list of keywords, which are passed to the function as a list of strings through the keywords_list keyword argument. it initialises several variables to default values and creates a dictionary called "*searchreq*" to check for total items of the search. For each keyword, the function sends a request to the ScanR API using the requests.post() function.

```
payload = requests.post(url, json=searchreq)
json_pl = json.loads(payload.text)
searchreq['pageSize'] = kwargs.get('payload_size')
max_iter = math.ceil(json_pl.get('total')/kwargs.get('payload_size'))
print("keyword: ",keyword)
```

It sends a POST request to the API. The request payload is the "*searchreq*" dictionary that was initialised earlier. After sending the request, the response is returned as JSON and loaded into a new variable called json_pl using the json.loads() method. The next line sets the value of the 'pageSize' key in the "*searchreq*" dictionary to the value of the payload_size argument passed to the function as a keyword argument. The max_iter variable is to calculate the number of iterations required to fetch all the data.

```
    #file1 = open("output_pub.txt","w+",encoding='utf-8')
    existing_pubs = session.execute_read(connection.get_pub_ids,keyword)

    for page_no in range(0,1): #set 1 to max_iter to get all nodes

        searchreq['page']=page_no
        payload = requests.post(url, json=searchreq)
        json_pl = json.loads(payload.text)

        if(json_pl.get('results')):
            loop = len(json_pl.get('results'))
            for item in range(0,loop):
                try:
                    #file1.write(str(pub))
                    #file1.write("\n\n")
                    pub = json_pl.get('results')[item].get('value')
                    if(pub.get('id') not in existing_pubs):
                        #print("creating network")
                        connection.create_network(session,keyword,**pub)
                except Exception as e:
                    print(e)
        else:
            print("No results found")

session.close()
```

The first line of code here is to fetch existing publication IDs from the database based on a keyword search.

Then, it loops over the page number to get the "page sized" amount of data at a time and process it. The loop iterates over a range of page numbers, where each page corresponds to a subset of data fetched from the API. The loop is set to execute for only the first page, since the range is from 0 to 1. However, it can be set to execute for max_iter pages to fetch all data.

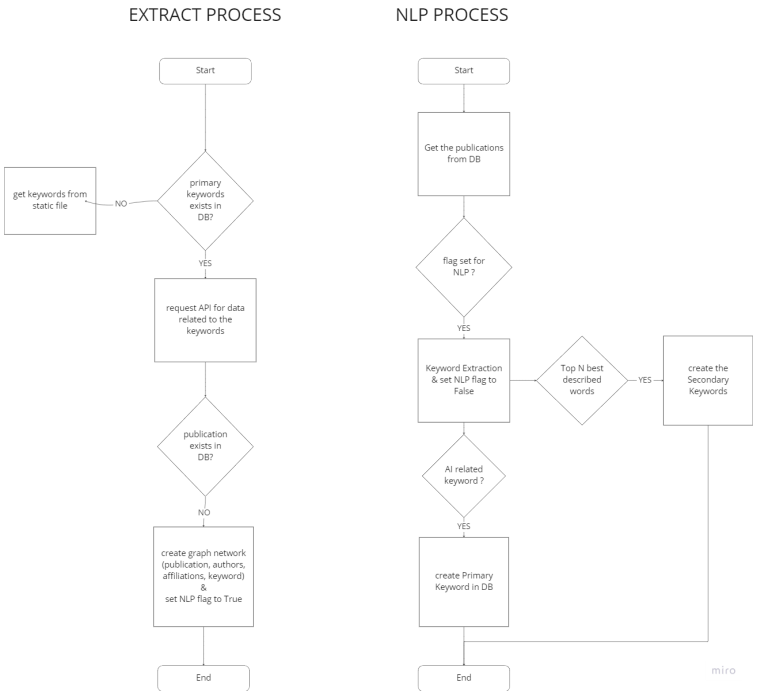
Next it checks if the JSON response from the API contains any results. If it does, then check for the number of items in the page to loop to avoid index overflow constraints. And it iterates over each item in the result set, extracts each result and its main content to write. After that, it checks if the publication ID already exists in the database. If the ID is not in the database, it calls the network creation function to create a network of all the relevant nodes and relations.

If there are any exceptions while creating the network, the code prints the exception message. If the JSON response does not contain any 'results', the code prints the message "No results found".

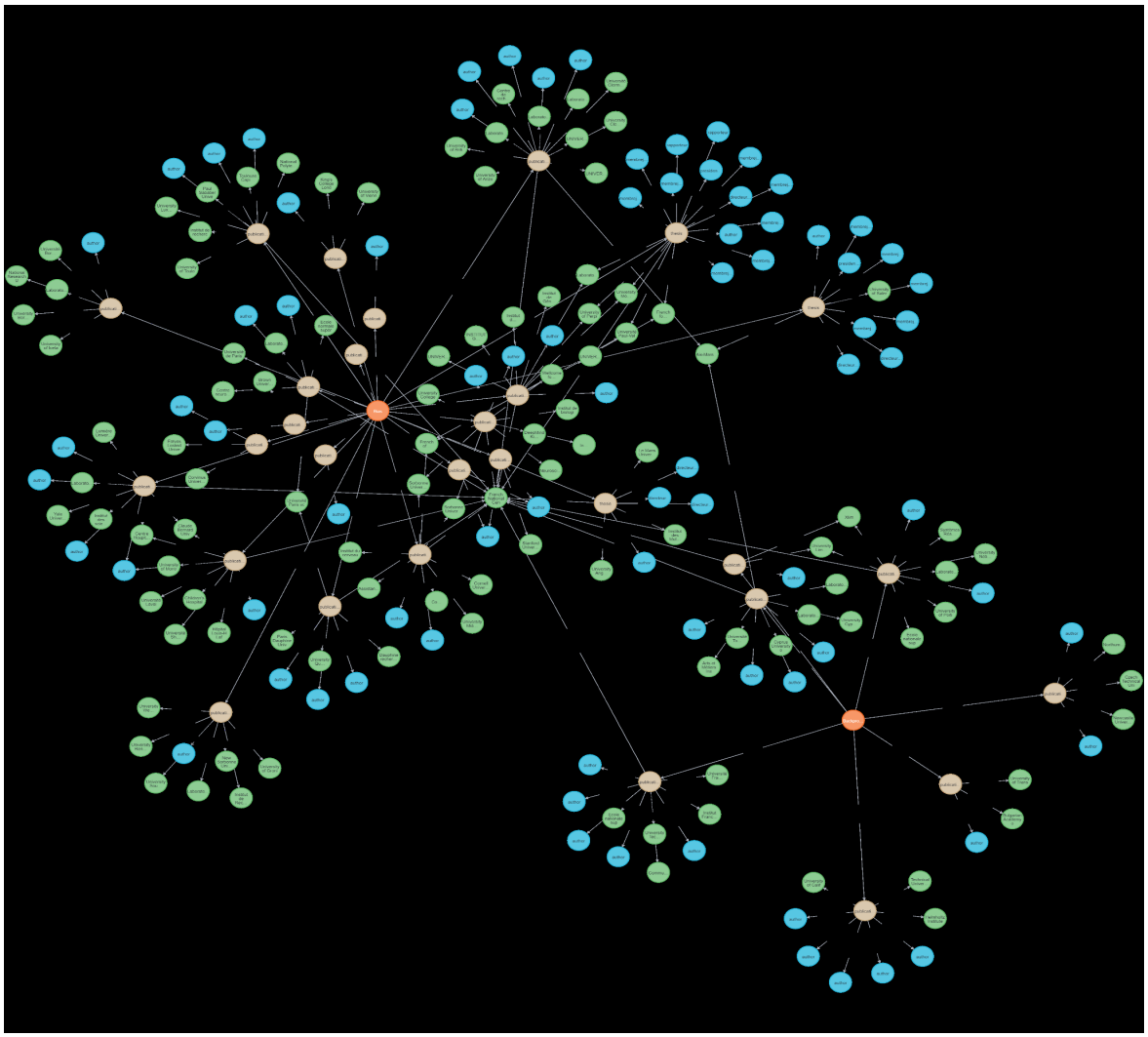
```
if __name__ == '__main__':

    session = connection.est_connection()
    data = session.execute_read(connection.get_keyword_nodes,'primary',0)
    if(data!=[]):
        dict1 = {'payload_size':100,'keywords_list':data}
    else:
        keywords = open("keywords.txt",'r',encoding='utf-8').read().lower().split('\n')
        dict1 = {'payload_size':100,'keywords_list':keywords}

    tasks = Process(target=getdata,kwargs=dict1)
    tasks.start()
    tasks.join()
```

In this part, it firstly initialises the input variables for the getdata function. Beginning with establishing the connection to Neo4j database, then getting all the primary keywords present in the DB, if they exist. It sets up the input variable for getdata function, if data exists in DB, it just updates the network with any new publications. But if no new or old primary keywords exist in DB, it sets up the primary keywords from a static file.

The code then creates a parallel processing task using the Process class from the multiprocessing module. The process runs the getdata function for each keyword in parallel, and performs API data extraction and network creation concurrently.

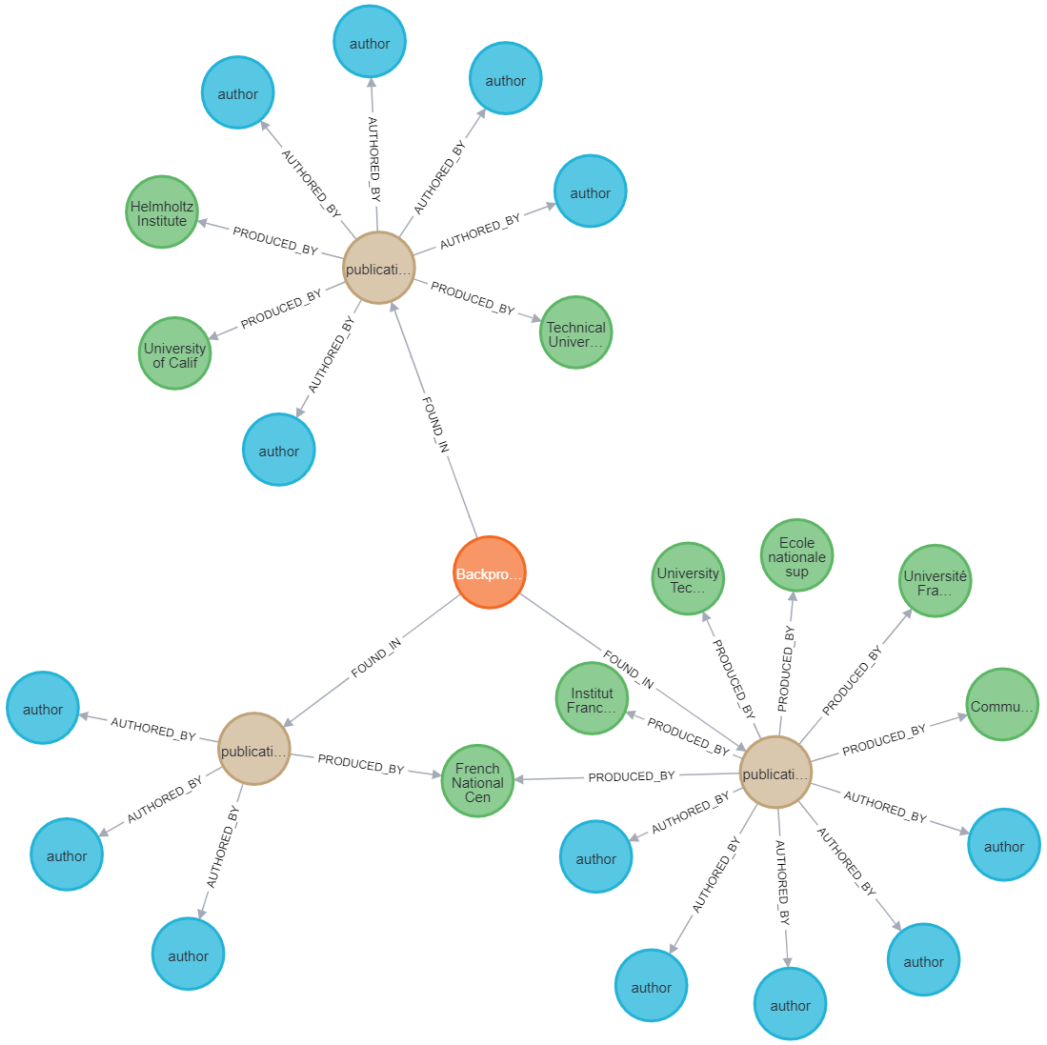


3.3. Overview of graph network



5.3.1. Nodes and Relationships

Above is the sample of 250 nodes containing Publications, Keywords, authors, affiliations and the corresponding relationships between them. Keyword nodes will get attached to multiple publications in which the keyword will be found with a relation [FOUND_IN] irrespective of the type i.e. primary or secondary. The publication is connected to the affiliations via [PRODUCED_BY] relation and to the Authors by a [AUTHORED_BY] relationship. These are created by the graph network creation function which is the combination of multiple CRUD and data validation functions from the connections module.



5.3.2. Properties and Values of Entities

Keyword Node

```
{
  "identity": <id>,
  "labels": [
    "keyword"
  ],
  "properties": {
    "name": <keyword>,
    "type": "secondary" / "primary"
  },
  "elementId": <id>
}
```

Author Node

```
{
 "identity": <id>,
 "labels": [
  "Author"
 ],
 "properties": {
  "lastName": "<lastname>",
  "firstName": "<firstname>",
  "gender": "<gender>",
  "name": "author",
  "id": "<>"
 },
 "elementId": "<id>"
}
```


Affiliation Node


```
{
 "identity": <id>,
 "labels": [
  "Affiliation"
 ],
 "properties": {
  "country": "<country>",
  "address": "<address>",
  "city": "<city>",
  "kind": "<type of institute>",
  "label": "<institution name>",
  "id": "<institution ID>"
 },
 "elementId": "<id>"
}
```


Publication Node


```
{
 "identity": <id>,
 "labels": [
  "Publication"
 ],
 "properties": {
  "LastUpdatedatScanR": "<last updated time>",
  "summary": <string>,
  "flag": <boolean>,
  "name": "publication",
  "id": "<publication unique ID>",
  "type": "<type>",
  "title": string,
  "publicationDate": "<date time>"
 },
 "elementId": "<id>"}
```

3.4. Keyword extraction and ontology enrichment (NLP)

Overview:

The last part of our solution suggests a Language model that uses transformer and word embeddings in order to identify new meaningful keywords. The purpose of this model is to extract keywords & enhance the database with keywords that would be retrieved from the dataset itself.

The procedure starts by retrieving a batch of summaries and proceeds to conduct processes to retrieve the raw meaningful words (using NLTK for tokenization and lemmatization). Once the text is tokenized and lemmatised, we drop the stopwords and create the pool of words to fetch required keywords.

Next, we split the text into meaningful chunks of sentences with length not exceeding 512 and create its embedding.Once this process is completed, we then proceed to calculate the cosine similarity of the whole sentence embedding with the embeddings of all the grammed keywords extracted from custom extract function.

The words with greater distance from the list are dropped and only those that have a small distance are appended to the new list called best_words. These will be later inserted into DB as secondary keywords for ontology.

Now from the pool of keywords(all possible combinations of summary published) are embedded and are compared to the embeddings of the primary keywords present in the DB and the word which was matched the most with the set of primary keywords will be considered as AI related word and pushed back into DB with type parameter as primary.

In order to understand this process in further detail and to assess its application to our use case, we will proceed by describing each of these steps and issues and challenges related to each.

1. Preprocessing the text and preparing the keyword extractor

In order to identify new meaningful keywords, the model first retrieves a batch of summaries from the dataset. This allows us to update the database with new equally relevant subjects and expand the already created network.

To do this, we first download necessary packages and libraries such as predefined python libraries such as numpy, pandas and NLP specific library, which Is the NLTK (with relevant packages), as well as the sklearn library with relevant packages allowing us to run models.

Once the batch of summaries has been retrieved, the model proceeds to conduct several preprocessing steps to retrieve the raw meaning of the words. This involves using the NLTK library to perform part-of-speech tagging (POS), tokenization, and lemmatization on the text. These steps help to reduce each word to its base form, so that variations of the same word

```
def text_prepros(doc_str):
    try:
        doc_str = re.sub(r"([0-9]+\.{0,1}[0-9]{0,})",r"",doc_str)
        doc_str = re.sub(r"([\.\[\&\,\/\\\]\[\-\^\+\`\$\%\!\@\#\>\<\?\;\:\{\}\=\_\'\|\]\*\)\(])",r"",doc_str)
        doc_str = doc_str.lower()
    except Exception as e:
        print(e)
    return doc_str
```

(e.g. "walk", "walks", "walking", etc.) are treated as the same word.

We need a text preprocessing function to clean the text to further tokenize it and avoid the special characters and numbers as tokens.

This function is used to identify the different parts-of-speech present in the corpus. By flagging each word as an adjective, adverb, verb or noun. This is done thanks to the function that we defined as “pos_tagger”.

```
def pos_tagger(nltk_tag_lst):
    nltk_tag = nltk_tag_lst[0][1]
    if nltk_tag.startswith('J'):
        return wordnet.ADJ
    elif nltk_tag.startswith('V'):
        return wordnet.VERB
    elif nltk_tag.startswith('N'):
        return wordnet.NOUN
    elif nltk_tag.startswith('R'):
        return wordnet.ADV
    else:
        return wordnet.ADJ_SAT
```

Once this stage is completed, we proceed to define the tokenization and lemmatization processes. Tokenization allows to shorten the corpus into smaller units and lemmatization allows to retrieve the primary root of the word. In order to do these processes we create a class called “LSTokenizer”.

```
class LSTokenizer:
    def __init__(self):
        #ignore_tokens = [',', '.', ';', ':', '"', '``', "''", '`',')']
        self.wnl = WordNetLemmatizer() # to lemmatize the word tokens
        #self.snb = snowball.SnowballStemmer(language='english') # for stemming the word token (currently not in use)
        self.stpw = stopwords.words('english') #+ ignore_tokens
    def __call__(self, doc):
        return [self.wnl.lemmatize(t,pos=pos_tagger(nltk.pos_tag([t]))) for t in word_tokenize(doc) if t not in self.stpw]
```

The class has an "**init**" method that initializes several properties of the class. The "wnl" property is a WordNetLemmatizer object from the NLTK library, which is used to perform lemmatization on the tokens.The "stpw" property is a list of stopwords from the NLTK library, which will be removed from the tokens during tokenization.

The class also has a "**call**" method that takes a single argument called "doc", which is a string containing the text to be tokenized. The method tokenizes the input string by first calling "word_tokenize" from the NLTK library to split the string into individual words. Then, it removes any stopwords contained in the "stpw" and performs lemmatization on each remaining token using the "wnl".

```
def get_keywords_str(doc_str):

    # --------- TOKENIZING AND CREATING THE POOL OF POSSIBLE KEYWORDS----------
    tokenizer = LSTokenizer()
    vectorizer = TfidfVectorizer(ngram_range=(2,2),tokenizer=tokenizer,preprocessor=text_prepros)
    matrix_ = vectorizer.fit_transform([doc_str])
    doc_words = vectorizer.get_feature_names_out()
    return doc_words
```

Once these steps are completed we can proceed to vectorise the cleaned and lemmatised tokens and fir transform to get the feature names out i.e. keywords in string format.

```python
def split_into_chunks(doc_str):

    input_ = ""
    final_list = []
    list_doc_str = doc_str.split('. ')
    list_lengths = [len(item) for item in list_doc_str]

    #Loops over each split sentence and tries to combine it with next with corresponding length checks
    for i in range(0,len(list_lengths)):
        if(i==0):
            if(list_lengths[i]<512):
                input_+= list_doc_str[i]
            else:
                final_list.append(input_)
                if(len(list_lengths)>i+1):
                    input_= list_doc_str[i+1]
        else:
            if(len(input_)<512):
                if(len(input_)+list_lengths[i]<=512):
                    input_ += list_doc_str[i]
                    if(i==len(list_lengths)-1):
                        final_list.append(input_)
                else:
                    final_list.append(input_)
                    input_= list_doc_str[i]
            else:
                final_list.append(input_)
                input_= list_doc_str[i]

    return final_list
```

This function is used to split the text into chunks not exceeding 512 characters in order to be able to run the model. This is because some language models have a maximum input length, and breaking the text into smaller chunks ensures that the model can process the text efficiently. This is done with the help of the “split_into_chunks" function which loops over the doc_str and splits it once the character limit of 512 is achieved. He then stores each splitter chunk into a list of lists called final_list.

2. Encoding the text and using cosine similarity

Once the text has been preprocessed and split into chunks, the model encodes the text using a pre-trained transformer-based language model. This allows the model to represent each word in the text as a numerical vector that captures its semantic meaning. The sentence transformer used in this case is ‘all-distilroberta-v1’.

In this last part of the script, we proceed to finding the best described words through the get_bestdesc_words function. It runs a cosine similarity function allowing to compare the encoded embedding of the input text with the embeddings of all possible keywords extracted using the custom function and we filter out the top N closest words to the document/sentence. We use cosine similarity to calculate closeness between two embeddings and get the distance values.

```
def get_bestdesc_words(doc_str,most_n):

    # --------- CREATING THE EMBEDDINGS FOR EXTRACTED WORDS & PRIMARY WORDS----------
    all_words = get_keywords_str(doc_str)
    candidate_embeddings = model.encode(all_words)
    doc_embedding = model.encode([doc_str])

    # --------- COMPARING EMBEDDINGS BASED ON COSINE SIMILARITY AND GETTING TOP N WORDS----------
    distances = cosine_similarity(doc_embedding, candidate_embeddings)
    sec_keywords_ = [all_words[index] for index in distances.argsort()[0][-most_n:]]

    return sec_keywords_
```

After this, we proceed to an AI related word extraction. It takes two input string variables:

1. All the extracted keywords from the custom function
2. The primary keywords from the graph Database which are created by the network creation functions earlier.

```
def get_ai_words(doc_str,prim_keys,most_n):

    # --------- CREATING THE EMBEDDINGS FOR EXTRACTED WORDS & PRIMARY WORDS ----------
    all_words = get_keywords_str(doc_str)
    candidate_embeddings = model.encode(all_words)
    prime_embeddings = model.encode(prim_keys)

    # --------- COMPARING EMBEDDINGS BASED ON COSINE SIMILARITY ----------
    distances = cosine_similarity(prime_embeddings, candidate_embeddings)
    ai_indx_lst = Counter([i[0] for i in distances.argsort()[:,-1:]]).most_common(most_n)
    ai_keywords_ = [all_words[tp[0]] for tp in ai_indx_lst]

    return ai_keywords_
```

This compares the above embeddings and after cosine similarity it creates the ndarray of dimensions prime_keywords_X, extracted_keywords_Y, now we are sorting the indexes based on values and taking the most common occurred index(i.e. most similar word) of the last column of every row. (this indicates the most common matched word with all the primary AI related words).

Finally, with help of a function name “main” we connect to the DB to get the primary keywords introduced in the graph. We then obtain best described(Secondary) and the AI related(primary) from all the extracted keywords from vectorizer. Finally it creates the nodes in the db and relates those with the respective publications.

```python
if __name__ == '__main__':

    model = SentenceTransformer('all-distilroberta-v1')

    # establishing connection to the graphDB with specific port and retrieving the session to get data
    session = connection.est_connection()
    data = session.execute_read(connection.get_pub_summary)                         #To get the publication's summaries which are tagged for keyword
    primary_keys = session.execute_read(connection.get_keyword_nodes,'primary',0)   #to get the primary AI related keywords in the graphDB

    for i in range(0,len(data)):     #Looping over all the publications we got from the DB
        p_id = data.iloc[i][0]       #the publication IDs of data present in DB
        summary = data.iloc[i][1]    #the summary of the publication

        #-------- ENRICHING THE ONTOLOGY ---------------
        best_words = []
        for chunk in split_into_chunks(summary):  #splitting the input into chunks of 512 length and extracting the top N best descibe words
            if(chunk!=''):
                best_words += get_bestdesc_words(chunk,1) # updating the best words list by extracting words per chunk
        if(best_words!=[]):                       #inserting the best descibed words of the summary as a Secondary node in graphDB
            for key in set(best_words):
                session.execute_write(connection.create_keynodes,p_id,key.lower(),'secondary')

        #-------- ENHANCING THE AI RELATED KEYWORDS SET ---------------
        for key in get_ai_words(summary,primary_keys,1):   #Looping onto each extracted AI related keyword
            session.execute_write(connection.create_keynodes,p_id,key.lower(),'primary')  #inserting the keyword as primary in the graphDB

        session.execute_write(connection.set_pub_flag,p_id) # setting the publication as done, to not perform keyword extraction in teh next run

    session.close()
```

This process improves the quality of the database by adding new and relevant keywords that were previously not included in the dataset.

3.5. Solution and its limits

The central subject of the project was to create an architecture allowing to collect and store data from the ScanR website. The purpose of this being the creation of a network of AI and data scholars and researchers, we were suggested to build our project around tools allowing us to work with highly interconnected variables and data. In addition to that, during our discussions with datacraft, we were suggested to base our collections queries on ontology around AI and DS subjects.

The Neo4j graph database is the tool that is most adapted for this use case. The concept of working with interconnected nodes is perfectly adapted to the business need of creating a database that would visually express relationships between variables such as authors, institutions, publications etc. Moreover, using an NLP to extract similarity in summaries in order to update the nodes and feed the DB even further is an innovative solution to this business need. However, this task didn't come without challenges and limitations. By pointing out those limitations, we are also able to give an idea on possible areas of improvement and new possibilities for DataCraft to discover for their mission.

Inconsistent data resulting in loss of data in the DB

One of the biggest challenges we faced while working on the database is inconsistent data. The ScanR website is an open-source government provided website, it collects its data from different institutions and in some cases, it is displayed in an inconsistent manner. Graph databases are built on the assumption that the data is highly interconnected, and the relationships between different data points are crucial to understanding the overall structure of the network.

When data in a graph database is inconsistent, it would lead to incorrect or incomplete relationships being formed between data points. For example, some of the publications were attached to authors and co-authors that would have no IDs. This inconsistency is a major problem as it would result in nodes being empty or not connected. Some connections would hence fail and create errors in the database. This would also lead to missing data and inaccurate analysis of the network (publications attached to wrong authors).

Inconsistent data can also lead to issues with querying the graph database. Queries may return incorrect or incomplete results if the data is inconsistent, making it challenging to retrieve the information needed to understand the network fully. In order to face this issue, we needed to have verification loops in the python code to prevent the script from failing if the returned result would be empty.

Overall, a more standardised and systemic source of data would prevent these risks and make the process smoother. These issues required a considerable amount of time, effort and numerous discussions with tutors as well as professors which would lead to taking the decision of adding script which drops a field and its relationship if it is empty. This results hence in a considerable loss of data.

Data Drift

Data drift is a phenomenon that occurs when the statistical properties of data change over time, leading to changes in the accuracy and effectiveness of machine learning or deep learning models and other data-driven algorithms. In the context of this project, data drift could occur in several ways, potentially affecting the accuracy and relevance of the keyword node in the graph database.

One way data drift could happen is if the underlying structure of the publication summaries changes over time. For example, if the publications shift to focus on different topics or use different language or terminology, the NLP algorithm may produce different keyword

suggestions, potentially leading to a mismatch between the keywords and the actual content of the publications. This could lead to inaccuracies in the graph database and potentially affect the ability to find people through their publications.

Another way data drift could occur is if the NLP algorithm itself changes over time. This could happen if the algorithm is updated or modified, potentially leading to changes in how it processes and analyses the publication summaries. If these changes are significant, they could affect the accuracy and relevance of the keyword node, potentially leading to incorrect or incomplete relationships in the graph database.

To address the risk of data drift, it may be necessary to periodically review and update the NLP algorithm used to generate the keyword node, as well as the data sources and processing methods used to extract and summarise the publication data. Regular maintenance and monitoring of the graph database may also be necessary to identify and address any inaccuracies or inconsistencies that arise due to data drift.

Lack of accuracy metrics for predicted keywords

Building an NLP requires a considerable amount of time and effort. Despite the satisfying results the model has been showing so far, it is important to state that the lack of accuracy metrics is a potential risk and has to be considered when the company implements the algorithm.

Not using metrics to define the accuracy of predicted keywords can be a significant risk, as it may lead to inaccurate or incomplete data being used to populate the keyword node in the graph database. Without metrics to assess the accuracy of the NLP algorithm used to generate keywords, it may be challenging to identify and correct errors or inconsistencies in the DB, potentially leading to incorrect or incomplete relationships being formed between data points.

To address this risk, it may be helpful to implement metrics to assess the accuracy of the NLP algorithm and the quality of the generated keywords. This could involve using techniques such as cross-validation or split testing to evaluate the performance of the NLP algorithm against a set of known or labeled data. By comparing the predicted keywords against known keywords or other sources of information, it may be possible to identify and correct any inaccuracies or inconsistencies in the algorithm and improve the quality of the keyword node in the graph database.

Another solution could be to implement a feedback loop or monitoring system that allows for the continuous assessment of the accuracy of the keyword node. This could involve setting up

a process for users to provide feedback on the relevance and accuracy of the keywords and using this feedback to refine and improve the NLP algorithm and the keyword node over time. By continually monitoring and evaluating the accuracy of the keyword node, it may be possible to identify and correct errors or inconsistencies as they arise and improve the overall quality of the graph database.

Conclusion

In conclusion, this project has demonstrated the power of combining NLP techniques and Graph Databases to create a network of newly graduated masters and PhD candidates, and identify their research interests through an analysis of their publications.

The NLP analysis performed in this project enabled the team to extract new keywords related to AI, Data Science, and adjacent topics, which were then used to populate the keyword node in the Graph Database. The use of Graph Databases allowed the team to represent complex relationships between the candidates, their publications, and their research interests, providing a comprehensive network of newly graduated researchers.

The success of this project showcases the importance of collaboration among individuals with diverse backgrounds and skills, who were able to use cutting-edge techniques and technologies to solve a real business problem in a professional setting. The iterative approach to refining the NLP analysis also demonstrates the value of an agile approach to data science projects.

It is important to note that this project also presents limitations and areas for improvement. For example, the NLP analysis was only performed on a batch of publication summaries and could be expanded to include a wider range of data sources. Additionally, the algorithm used for the cosine similarity calculation could be optimized for increased accuracy.

The report also provides recommendations for the company to consider when integrating the script into their workflow. These include considerations such as data privacy and security, as

well as the need for ongoing maintenance and updates to ensure the continued accuracy and relevance of the results.

Overall, this project highlights the potential of data science to provide valuable insights into complex problems, and the importance of using innovative techniques such as NLP and Graph Databases to unlock the full potential of data. While there are limitations and areas for improvement, the recommendations presented in this report can help guide the company in leveraging these techniques to unlock the full potential of their data.

Bibliography

1. ”A Survey of Web Data Extraction Tools" by Chia-Hui Chang, Wei-Kuan Shih, and Shou-De Lin (2017)
2. "Data Extraction Techniques for Web Pages: A Survey" by Deependra Kumar Jha, Srikantaiah K. C., and Kavi Mahesh (2016)
3. "Web Scraping for Data Acquisition: A Review of Tools and Techniques" by Aqsa Saeed, Muhammad Aftab, and Saif Ur Rehman Malik (2021)
4. "Why Graph Databases are Perfect for AI and Machine Learning Applications" by Will Kelly, published in TechRepublic (2021).
5. "Graph Databases: Everything You Need to Know" by TechTarget, published on SearchDataManagement.com (2021).
6. Rouse, M. (2018). SQL vs. NoSQL: What's the Difference? TechTarget. Retrieved from
7. https://searchsqlserver.techtarget.com/feature/SQL-vs-NoSQL-What-s-the-difference
8. RoBERTa: A Robustly Optimised BERT Pre Training Approach (2019) by Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, Veselin Stoyanov
   https://huggingface.co/sentence-transformers/all-distilroberta-v1
9. Attention Is All You Need(2017) by Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, Illia Polosukhin
   https://arxiv.org/abs/1706.03762