L 715 Seminar: Semantic Natural Language Processing, NoAI, and Big Knowledge
Question and Answer model

Arpitha Kashyap
Shridivya Sharma
Siddharth Thiruvengadam
Sriram Sitharaman

December 14, 2017

Contents

1 Abstract

Our paper talks about a question and answer system that allows natural language question to be asked of a knowledge base of information. The questions are asked to a digital assistant like Alexa or Google Home. We figure out various patterns in the question. Using these patterns, we write cypher queries that can fetch answers for these questions. We use Neo4j to store data in the form of a graph. We return the answer in the form of statements on the web interface. If the answer for the question does not exist in the database, we return the first two sentences of what is found regarding the question from Wikipedia.

2 Introduction

As soon as there were computers, an as soon as there was Natural Language Processing, we tried to use computers to answer textual questions. There are two important modern paradigms of question answering which were implemented by systems in the early 1960's. They were:

1. IR-based question answering and
2. Knowledge based question answering

Every computer out there got into the act. They were trying to implement this on all kinds of platforms and generate the perfect result. There was a computer that Douglas Adams invented in The Hitchhiker's Guide to the Galaxy, called Deep Thought that managed to answer, "the Great Question of Life the Universe and Everything". The answer was 42, but unfortunately the details of the question were never revealed. Recently, IBMs famous question-answering system, "Watson" won the TV game show Jeopardy! It even beat humans at the task of answering questions like William Wilison's "An Account of the Principalities of Wallachia and Moldovia" Inspired this author's most famous novel. Although the goal of quiz shows is entertainment, the technology used to answer these questions both draws on and extends the state of the art in practical question answering.

The most current question answering systems focus on factoid questions that can help humans in their daily lives. It can be a revolution if they could answer like humans. Factoid questions are questions that can be answered with simple facts expressed in short text answers. Some examples for factoid questions can be the following:

1. Who founded Virgin Airlines?
2. Who is the president of the United States?
3. What is the average age of the onset of autism?
4. Where is Apple Computer based?
5. Where is Los Angeles?

These factoid questions can be answered with a short string expressing a personal name, temporal expression, or location.

In this project, we explore Knowledge based question answering and focus on their application to factoid questions. When the answer or data isnt present in the database, we resort to IR based question answering where we rely on the enormous amounts of information available as text on the Web. However, when the answer is available in the database, we resort to knowledge based question answering. In this paradigm we build a semantic representation of the query.

The meaning of a query can be a full predicate calculus statement. Suppose we have the question "What states border Texas?"(This is taken from the GeoQuery database of questions on U.S), Geography (Zelle and

Mooney, 1996) might have the representation: $\lambda x.state(x) \wedge$ borders(x,texas). Alternatively, the meaning of a question could be a single relation between a known and an unknown entity or vice versa. Thus, the representation of the question "When was Ada Lovelace born?" could be birth-year (Ada Lovelace, ?x). No matter what kind of representation we choose, we'll be using it to query databases of facts.

These might be complex databases, perhaps of scientific facts or geospatial information, that need powerful logical or SQL queries. Or these might be databases triple stores of simple relations, triple stores like Freebase or DBpedia. There are large practical systems like the DeepQA system in IBM's Watson which are generally hybrid systems. This means that it uses both text datasets and structured knowledge bases to answer questions. DeepQA extracts a wide variety of meanings from the question like parses, relations, named entities, ontological information, and then finds large numbers of candidate answers in both knowledge bases and in textual sources like Wikipedia or newspapers.

Each candidate answer is scored using a wide variety of knowledge sources, such as geospatial databases, temporal reasoning, taxonomical classification, and various textual sources. We explore using cypher queries and the Neo4j database system to store our data and retrieve answers. We also build a hybrid system where answers not found in the database are further looked for in Wikipedia. We extract patterns from the questions and use these patterns to write cypher queries to retrieve answers. The Neo4j database is a graph database so the answer to a query would be a graph. It is more visually pleasing and helps us understand relationships in a better way.

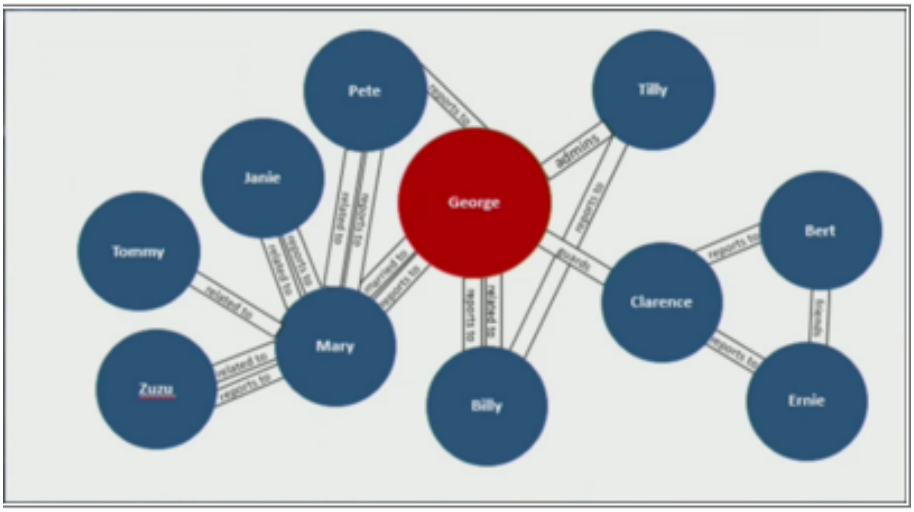
For relations that are very frequent, it may be worthwhile to write hand-written rules to extract relations from the question. For example, to extract the birth-year relation, we could write patterns that search for the question word When, a main verb like born, and that extract the named entity argument of the verb.

Given a user question, like "Who is Donald Trump?", the pattern here is "Who" and Donald Trump would be converted to ".*". This represents what needs to be found in the database(which is a node or a relationship). We search for this and return the nodes and relationships, or the sub graph required to answer the question. If there is nothing related to Donald Trump in the database, information retrieval techniques extract passages directly from Wikipedia, guided by the text of the question. The method processes the question to determine the likely answer type (often a named label in the database like a person, location, or time). This returns the extracted statements from Wikipedia which we have set to be the first two statements from the first paragraph.

3 Motivation behind using graph databases for QA Systems

One of the primary goals of information retrieval is the process of analyzing text and identifying mentions of semantically defined entities and relationships within it. These relationships can then be recorded in a database to search for a particular relationship or to infer additional information from the explicitly stated facts. Given the amount of information that is present today, and the growing number of interconnections between entities, using graphs to store data is both intuitive and convenient.

For instance, social networking sites use graph databases to provide information about how people are connected to each other. Graph databases make it easy to store relations as shown in the figure below.



4 Background and Related Work

Several question answering systems have been built for various databases. Question answering is not a new research area as Simmons (1965) reviews no less than fifteen English Language QA systems.

Question answering systems can be found in many areas of NLP research, including:

1. Natural language database systems
2. Dialog systems
3. Reading comprehension systems
4. Open domain question answering

Some of the early QA systems were BASEBALL and LUNAR. BASEBALL answered questions about US baseball league. LUNAR answered questions about the geological analysis of rocks returned by Apollo moon missions. Both these systems very extremely effective in their respective domains. LUNAR was also demonstrated at a convention in 1971 and was able to answer 90% of the questions that people asked with respect to the domain.

More restricted domain question answering systems were built in the following years. The common feature of all these systems is that they had a core database or knowledge system that was hand-written by experts of the chosen domain. The language abilities of BASEBALL and LUNAR used techniques similar to ELIZA and DOCTOR, the first chatterbot programs.

Knowledge-bases were developed to target narrower domains of knowledge. The QA systems developed to interface with these expert systems produced more repeatable and valid responses to questions within an area of knowledge. These expert systems closely resembled modern QA systems except in their internal architecture. Expert systems rely heavily on expert-constructed and organized knowledge bases, whereas many modern QA systems rely on statistical processing of a large, unstructured, natural language text corpus.

The development of comprehensive theories in computational linguistics led to the development of ambitious projects in text comprehension and question answering. An example of such a system was the Unix Consultant (UC). The system answered questions pertaining to the Unix operating system. However, the

systems developed in the UC never went past the stage of simple demonstrations, but they helped the development of theories on computational linguistics and reasoning. Recently, specialized natural language QA systems have been developed, such as EAGLi for health and life scientists.

While an enormous amount of information is encoded in the vast amount of text on the web, information obviously also exists in more structured forms. We use the term knowledge-based question answering for the idea of answering a natural language question by mapping it to a query over a structured database. Like the text based paradigm for question answering, this approach dates to the earliest days of natural language processing, with systems like BASEBALL that we mentioned earlier. This system answered questions from a structured database of baseball.
Nowadays, digital assistants like Alexa by Amazon and Google Home by Google are the most famous examples of question answering systems. They are not build for complete conversations yet, but people are working on it. But follow up questions are answered quite well. This is how it typically works:

1. Message comes in from user
2. An NLP system classifies that message into an "intent," and extracts relevant information into "slots." An example of slot extraction would be extracting "Destination City: New York" from the sentences "I would like a flight to New York", while the intent might be "Find flight".
3. The results from the NLP system get sent to the logic system-in this case a Lambda function that Alexa is invoking-which then routes it

The latest thing that companies are working on is the digital assistant able to handle entire conversations involving multiple questions that are all related to each other. This makes it closer to be an excellent assistant. It considers a huge database and works a lot of queries and code to return the right answer that makes sense and is grammatically correct.

5 Neo4j and how it works

Neo4j is a graph database that is used for our QA system. A graph database is an online database management system with Create, Read, Update and Delete (CRUD) operations working on a graph data model. There are certain advantages of using Neo4j over other databases:

1. By assembling the simple abstractions of nodes and relationships into connected structures, graph databases enable us to build sophisticated models that map closely to our problem domain
2. Graph databases are like the next generation of relational databases, but with first class support for "relationships", or those implicit connections indicated via foreign-keys in the traditional relational databases.

Connected information is everywhere in our world. Neo4j is used to efficiently store, handle and query highly connected elements in your data model. With a powerful and flexible data model you can represent your real-world, variably structured information without a loss of richness. The property graph model is easy to understand and handle, especially for object oriented and relational developers.

Neo4j can be installed from the website or one can start a sandbox online. Cypher, Neo4j's declarative graph query language, is built on the basic concepts and clauses of SQL but has a lot of additional graph-specific functionality to make it simple to work with your rich graph model without being too verbose. It allows you to query and update the graph structures, with concise statements. Cypher is centered around the graph patterns that are core to your use-cases and represents them visually as part of its query syntax.

If Neo4j is installed and started as a server, you can interact with the database with the built-in Neo4j browser application. If it is to be accessed programmatically, we would do so with the integrated HTTP API, which allow us to:

1. POST one or more Cypher statements with parameters per request to the server
2. Keep transactions open over multiple requests
3. Choose different result formats

Running the server:

cd neo4j
./bin/neo4j start

And now we can point the browser to **http://localhost:7474** for a nice web GUI. The first time you open the interface, you'll be asked to set a password for the user "neo4j".

Stopping the server:
./bin/neo4j stop

A sample HTTP request that executes Cypher to create a `Person` would look like this. You can run it directly from the Neo4j browser, here shown with the plain JSON response.

```
:POST http://localhost:7474/db/data/transaction/commit
  {"statements":[
    {"statement":"CREATE (p:Person {name:{name}}) RETURN p",
"parameters":{"name":"Daniel"}}
   ]}
->
{"results":[{"columns":["p"],"data":[{"row":[{"name":"Daniel"}]}]}],"errors":[]}
```

ou can connect to Neo4j with a driver or connector library designed for your stack or programming language. There are drivers for Neo4j for almost all popular programming languages, most of which mimic existing database driver idioms and approaches.

For instance, the Neo4j JDBC driver would be used like this to query the database for Johns departments:

```
Connection con = DriverManager.getConnection("jdbc:neo4j://localhost:7474/");

String query =
    "MATCH (:Person {name:{1}})-[:EMPLOYEE]-(d:Department) RETURN d.name as dept";
try (PreparedStatement stmt = con.prepareStatement(QUERY)) {
    stmt.setString(1,"John");
    ResultSet rs = stmt.executeQuery();
    while(rs.next()) {
        String department = rs.getString("dept");
        ....
    }
}
```

6 Using Neo4j from Python

After Neo4j is installed on any system, it can then be accessed via its binary and HTTP APIs. You can use the official binary driver for Python (neo4j-python-driver) or connect via HTTP with any of our community drivers.

The Neo4j Python driver is officially supported by Neo4j and connects to the database using the binary protocol. It aims to be minimal, while being idiomatic to Python.

Installation

pip install neo4j-driver

Both Python 2 and Python 3 are supported by most of the libraries and this should help give more flexibility when putting together your Neo4j-based technology stack.

We also have Py2neo. Py2neo is a client library and comprehensive toolkit for working with Neo4j from within Python applications and from the command line. The core library has no external dependencies and has been carefully designed to be easy and intuitive to use.

Installation

pip install py2neo

Now using py2neo we can write code to create nodes and relationships. Also, given an input Natural Language query, it Identifies the matching regex and hits the neo4j graph DB with the corresponding Cypher query.

7 Cypher: Introduction and Basics

Cypher is a declarative graph query language that allows for expressive and efficient querying and updating of the graph store. Cypher is a relatively simple but still very powerful language. Very complicated database queries can easily be expressed through Cypher. This allows you to focus on your domain instead of getting lost in database access.

Cypher is designed to be a humane query language, suitable for both developers and (importantly, we think) operations professionals. Our guiding goal is to make the simple things easy, and the complex things possible. Its constructs are based on English prose and neat iconography which helps to make queries more self-explanatory. We have tried to optimize the language for reading and not for writing.

Being a declarative language, Cypher focuses on the clarity of expressing what to retrieve from a graph, not on how to retrieve it. This is in contrast to imperative languages like Java, scripting languages like Gremlin, and the JRuby Neo4j bindings. This approach makes query optimization an implementation detail instead of burdening the user with it and requiring her to update all traversals just because the physical database structure has changed (new indexes etc.).

Cypher is inspired by a number of different approaches and builds upon established practices for expressive querying. Most of the keywords like WHERE and ORDER BY are inspired by SQL. Pattern matching borrows expression approaches from SPARQL. Some of the list semantics have been borrowed from languages such as Haskell and Python.

7.1 Structure

Cypher borrows its structure from SQLqueries are built up using various clauses.

Clauses are chained together, and they feed intermediate result sets between each other. For example, the matching variables from one MATCH clause will be the context that the next clause exists in.

The query language is comprised of several distinct clauses. You can read more details about them later in the manual.

Here are a few clauses used to read from the graph:

MATCH: The graph pattern to match. This is the most common way to get data from the graph.
WHERE: Not a clause in its own right, but rather part of MATCH, OPTIONAL MATCH and WITH. Adds constraints to a pattern, or filters the intermediate result passing through WITH.
RETURN: What to return.

7.2 How to write cypher queries

Syntax

Read Query Structure

```
[MATCH WHERE]
[OPTIONAL MATCH WHERE]
[WITH [ORDER BY] [SKIP] [LIMIT]]
RETURN [ORDER BY] [SKIP] [LIMIT]
```

Match

```
MATCH (n:Person)-[:KNOWS]->(m:Person)
WHERE n.name = "Alice"

Node patterns can contain labels and properties.
MATCH p = (n)-->(m)

Assign a path to p.
MATCH (n)-->(m)
Any pattern can be used in MATCH.

OPTIONAL MATCH (n)-[r]->(m)
Optional pattern, NULLs will be used for missing parts.
WHERE m.name = "Alice"
Force the planner to use a label scan to solve the query
(for manual performance tuning).
```

WHERE

```
WHERE n.property <> {value}
Use a predicate to filter. Note that WHERE is always part of
```

a MATCH, OPTIONAL MATCH, WITH or START clause. Putting it after a different clause in a query will alter what it does.

7.3 Write-Only Query Structure

```
(CREATE [UNIQUE] | MERGE)*
[SET|DELETE|REMOVE|FOREACH]*
[RETURN [ORDER BY] [SKIP] [LIMIT]]
```

7.4 Read-Write Query Structure

```
[MATCH WHERE]
[OPTIONAL MATCH WHERE]
[WITH [ORDER BY] [SKIP] [LIMIT]]
(CREATE [UNIQUE] | MERGE)*
[SET|DELETE|REMOVE|FOREACH]*
[RETURN [ORDER BY] [SKIP] [LIMIT]]
```

CREATE

```
CREATE (n {name: {value}})
Create a node with the given properties.
```

```
CREATE (n {map})
Create a node with the given properties.
```

```
CREATE (n {collectionOfMaps})
Create nodes with the given properties.
```

```
CREATE (n)-[r:KNOWS]->(m)
Create a relationship with the given type and direction;
bind an identifier to it.
```

```
CREATE (n)-[:LOVES {since: {value}}]->(m)
Create a relationship with the given type, direction, and
properties.
```

RETURN,DELETE and REMOVE

```
RETURN *
Return the value of all identifiers.
```

```
RETURN n AS columnName
Use alias for result column name.
```

```
RETURN DISTINCT n
Return unique rows.
```

```
ORDER BY n.property
Sort the result.
```

```
ORDER BY n.property DESC
Sort the result in descending order.
```

```
SKIP {skipNumber}
Skip a number of results.

LIMIT {limitNumber}
Limit the number of results.

SKIP {skipNumber} LIMIT {limitNumber}
Skip results at the top and limit the number of results.

DELETE n, r
Delete a node and a relationship.

DETACH DELETE n
Delete a node and all relationships connected to it.

MATCH (n) DETACH DELETE n
Delete all nodes and relationships from the database.

REMOVE n:Person
Remove a label from n.

REMOVE n.property
Remove a property
```

7.5 Connecting to Neo4j

One can calculate the memory and CPU recommendations for a Neo4J instance using the [Hardware Sizing Calculator](https://neo4j.com/hardware-sizing-calculator/).

One has to follow the below steps:

1. Download [neo4j](https://neo4j.com/artifact.php?name=neo4j-community-3.2.3-unix.tar.gz) using the following command on AWS EC2 instance *"wget https://neo4j.com/artifact.php?name=neo4j-community-3.2.3-unix.tar.gz"*
2. Extract the binaries using *tar -xf neo4j-community-3.2.3-unix.tar.gz* at Neo4j is installed at /home/ubuntu/graphdb/n community-3.2.3/ on course AWS server
3. Ports 7474,7473 and 7687 were added to the security group
4. Add neo4j.sh to /etc/profile.d for the services to be available to any user who logs into the server.
5. Starting and Stopping neo4j start and neo4j stop
6. Once the server is up the browser is available at [URL](http://34.204.124.142:7474/browser)
7. Login details for Neo4j browser
   - User Name: Neo4j
   - Passowrd: DtwAMjrk6zt1bHifYOJ6

8 Typical components of a Question Answering system

There are three basic steps that are part of most question answering systems - question processing, answer type detection, answer retrieval. The way these steps are performed can vary quite a lot, but the motivating ideas behind them remain the same.

8.1 Question processing,Answer type detection and Answer retrieval

The goal of the question-processing phase is to extract a number of pieces of information from the question. At the very least, this information consists of the question words and structural information about the search phrase (subject, object, leading verbs etc.). One might also perform entity recognition on the words or n-grams extracted from the question. Some systems also extract a focus, which is the string of words in the question that are likely to be replaced by the answer in any answer string found.
With this information, the query used to search the collection of documents or graph can be formulated. This typically involves removing stopwords, including synonyms and other related words to keywords extracted from the search phrase, and including all the lemmatized forms of all verbs.

Answer type detection and Answer retrieval
The task of answer type recognition is to determine the named-entity or similar class categorizing the answer. The search phrase "Who is the CEO of Apple" requires an answer of type PERSON. A question such as "What is the Capital of India" requires an answer of type PLACE or LOCATION or CITY. If the answer type for a question is known, it helps avoid looking at every sentence or noun phrase in the documents for the answer, instead focusing on, for example, just people or cities.

Answer type recognizers are built by user-defined rules, machine learning, or some combination of the two. For instance, Webclopedia QA Typology has 276 hand-written rules that can identify 180 answer types. A regular expression for detecting the answer type BIOGRAPHY could be who is or was or are or were PERSON. Most answer type recognizers, however, are built using machine learning. They are trained on databases of questions that have been hand-labeled with an answer type. Features that are commonly used for classification include the words in the questions, the part-of-speech of each word, and named entities in the questions.

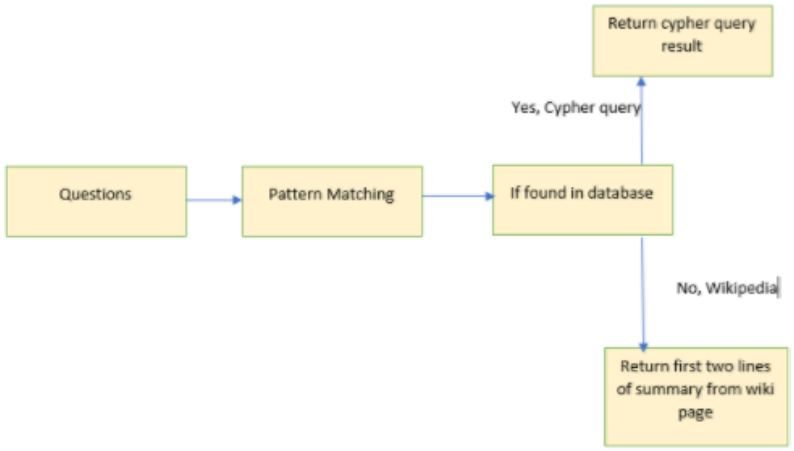
Answer retrieval
The query that is created using information extracted in the above two steps is used to query the database or set of documents. The result of this document retrieval stage is the set of candidate documents or subgraphs. One of these will be the best answer to the question posed by the end-user, and further filtering must now be done to find this answer. This task is somewhat easier in graph databases, where each unique entity only has one graph. In document search on the other hand, the set of words in the search query can appear in many documents, and it becomes hard to select between them.

Candidate answers are typically ranked using metrics such as:

1. The number of named entities of the right type in the passage
2. The number of question keywords in the passage
3. The proximity of the keywords from the original query to words
4. The N-gram overlap between the answer and the question

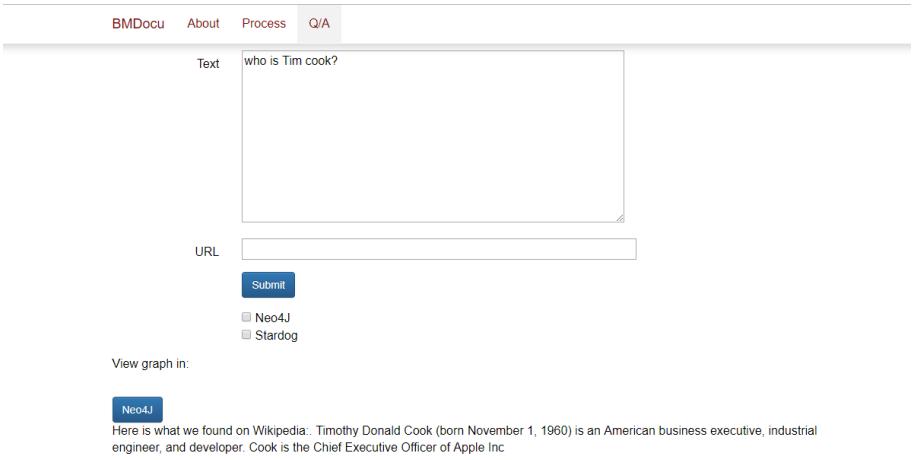
9 Architecture

Concept extraction team converts the LingData object i.e. raw linguistic information into graphable knowledge. The output is given to graph mapper team which converts concept table into abstract graph. Neo4j team converts this abstract graph into cypher queries and database is created in Neo4j.
An XML file with question patterns and their corresponding cypher queries has been created. As soon as the user ask a question, if it's corresponding pattern is found then it's corresponding query is fetched from XML file and is executed. If the query returns null value that means the answer cannot be found in database. In this case, program will return first two lines of summary from Wikipedia Page.

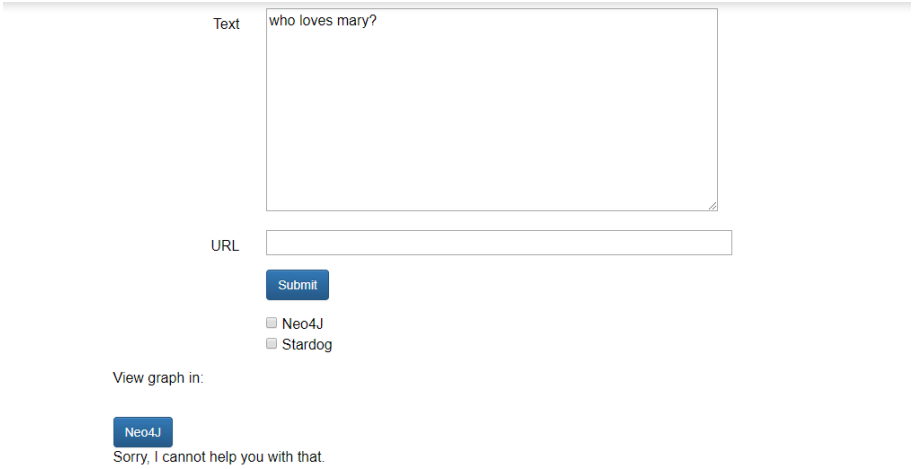


Examples :

1. Let's ask question "Who is Tim cook?"



2. let's ask "who loves mary?"



10 Code Explanation

10.1 Regular expression XML File creation

XML was chosen for creating a pattern matching file as I can be used across a variety of platforms and programming languages. Figure (no) shows the created XML file which contains the regular expression and the corresponding cypher query that would be used to query the neo4j database with the matched "(.*)" part from the regular expression pattern.

```
<collection>
<match>
   <Pattern>who (is) (.*)?</Pattern>
   <Cypher>MATCH (a:PERSON)-[r]-(b) where a.Name contains "(.*)" RETURN r,a,b</Cypher>
</match>
<match>
   <Pattern>what (is|are) (.*)?</Pattern>
   <Cypher>MATCH (a)-[r]-(b) where a.Name contains "(.*)" RETURN a</Cypher>
</match>
<match>
   <Pattern>(what do you know|tell me) about (.*)?</Pattern>
   <Cypher>MATCH (a)-[r]-(b) where a.Name contains "(.*)" RETURN r,a,b</Cypher>
</match>
<match>
   <Pattern>who (like|likes) (.*)?</Pattern>
   <Cypher>MATCH (a:PERSON)-[r:like]-(b) where b.Name contains "(.*)" RETURN a</Cypher>
...
...
...

<match>
   <Pattern>A woman who (like|likes) (.*)?</Pattern>
   <Cypher>MATCH (a)-[r:like]->(b:(.*)) where a.gender=~"(?i)female" RETURN a</Cypher>
</match>
</collection>
```

10.2 Question Matching with Cypher

The XML file is given as an input to the Question Answering python script which is kept running as a server that would get the input from the Dispatcher script which gets the natural language question query from its web interface. Once the QA script receives the input is checks the input question against each of the regular expression obtained from the regular expression file. Once a match is obtained, the corresponding cypher is taken for querying the database. BeautifulSoup html/xml parser from Python was used to parse the XML script to separate the regular expression and corresponding cypher queries into their respective lists.

In case when there is a regular expression match but the corresponding cypher does not yield any graph from the neo4j database, the matched part is taken and checked in the Wikipedia pages using the Pythons Wikipedia API. If there exists a page in Wikipedia, the first 2 lines of the summary from the Wikipedia is returned. This ensures that questions that the system cant cover would get covered. When there is no results from both the neo4j database and the Wikipedia, the system returns a default answer of "Sorry, I cannot help you with that."

11 Limitation

A key phase of answering a question is understanding it - Question Processing. Question Processing is about converting a natural language question into an unambiguous form that a computer is capable of understanding. Properly understanding a question is vital in today's question answering systems, because most of them attempt to let users ask their questions in 'free form'.

Questions don't necessarily have semantic structure as written texts. Here are some typical kinds of questions that are found on search engines:

'where can i find pictures of hairstyles'
'the difference between white eggs and brown eggs'
'who be the richest man in the world'
'the meaning of life is'
'can you drink milk after the expiration date'

Yet another challenge is to do with word sense disambiguation. The question 'why do people dislike apple' could be referring to the fruit or the company Apple Inc. The system needs to be able to infer the correct sense of the word based on the other information that is available such as other words in the questions, the structure of the sentence, other similar searches etc.

Sophisticated question answering systems use a combination of POS tagging, entity recognition, handwritten rules (based on questions words, regex patterns) and supervised machine learning techniques to decipher the meaning and context contained in the user search query.

12 Future work

The most basic representation of a question is simply the individual words of that question (ignoring contextual information such as the ordering of the words). While simple, this is also by far the most important part of our program - the best indicators of certain question types are single words, and in particular question words clearly reveal a lot about the type of question being asked. On the other hand, a large amount of information would be lost by stopping here, since words can often mean many different things depending on their context.
We can regain some of this contextual information by examining part of speech tags. We run a parser over

the question and take the preterminal nodes of the parser as the parts of speech for each word. These parts of speech alone wouldn't help much, so we add as features word and part of speech pairs, thus helping to disambiguate words which have different senses depending on their part of speech.

As of now, we are not checking the right lemmatization in the output. For example, for questions like,"Who is Tim Cook?", the output looks like- "Tim Cook is a Person. Tim Cook buys Apple" We need to work on lemmatization and reverse morphology to make the output grammatically correct like "Tim Cook is a person,who bought Apple".

One other we need to work on is to store answers in memory for a short amount of time, and use them in case the follow up question is linked to this answer Who is the president of the USA?
- Donald Trump
Who is his wife?
- The system should now be able to answer 'Melania Trump'

13 Conclusion

Question Answering is a field of Natural Language Processing that attempts to answer questions stated in a natural language. Open-Domain Question Answering aims to extract concise, complete answers from large document collections or knowledge. Most current question answering systems focus on factoid questions. Factoid questions are questions that can be answered with simple facts expressed in short text answers or graphs.
Described here are two common paradigms in modern question answering systems that tackle factoid questions. The first one is called IR-based question answering. It typically uses open content from the web or specialized collections such as PubMed to search for answers. Given a user question, information retrieval techniques extract passages directly from these documents, guided by the text of the question. The method processes the question to determine the likely answer type (often a named entity like a person, location, or time), and formulates a search. The search engine returns ranked documents which are broken up into suitable passages and reranked. Finally, candidate answer strings are extracted from the passages and ranked.
In the second paradigm, we instead build a representation of the query that can be used to search through data stores. While an enormous amount of information is encoded in the vast amount of text on the web, information obviously also exists in more structured forms. We use the term knowledge-based question answering for the idea of answering a natural language question by mapping it to a query over a structured database.

References

[1] Neo4j Installation https://neo4j.com/docs/operations-manual/current/installation/

[2] Python installation
`https://neo4j.com/developer/python/`

[3] QA talks ppt
`www.dcs.shef.ac.uk/ mark/nlp/talks/qa1.ppt`

[4] Cypher references
`http://neo4j.com/docs/pdf/neo4j-cypher-refcard-stable.pdf`

[5] Cypher Documentation- introduction
`https://neo4j.com/docs/developer-manual/current/cypher/`

[6] Building a Question Classifier for a TREC-Style Question Answering System
https://nlp.stanford.edu/courses/cs224n/2004/may-steinberg-project.pdf