Translating Natural Language Queries to SPARQL

A Project Report

Presented to

Professor Chris Pollett

Professor Katerina Potika

Professor Robert Chun

Department of Computer Science

San Jose State University

In Partial Fulfillment

Of the Requirements of the Class

CS 298

By

Shreya Satish Bhajikhaye

May 2021

Abstract

The Semantic Web is an extensive knowledge base that contains facts in the form of RDF triples. These facts are not easily accessible to the average user because to use them requires an understanding of ontologies and a query language like SPARQL. Question answering systems form a layer of abstraction on linked data to overcome these issues. These systems allow the user to input a question in a natural language and receive the equivalent SPARQL query. The user can then execute the query on the database to fetch the desired results. The standard techniques involved in translating natural language questions to SPARQL queries are natural language processing, machine learning, and information retrieval.

In this report, we describe our English language to SPARQL query translation system. The input for the proposed system reads a complete question in the English language, identifies the type of query to be built, and finds the triples from the question to fit in the query. The system contains two components – template classification which uses the Tree-LSTM technique to identify the query template, and the entity recognition module which uses external libraries to recognize the triples in the question. The Lc-QuAD database, with 200 questions across two unique SPARQL templates, was used to train and evaluate the model. The system queries the Wikidata database to answer the questions and gives 60% correct results.

Keywords – RDF, SPARQL, Question Answering System, Wikidata, Tree-LSTM

Table of Contents

CHAPTER 1. INTRODUCTION

The significant rise in the availability of information on the web requires improvement in the way users can handle the data. Knowledge graphs containing large amounts of data as facts have become an important aspect of this advancement. Knowledge graphs can be open-source like DBpedia and Wikidata or proprietary like Google's Knowledge Graph. The semantic web contains collections of interconnected knowledge graphs that create a machine-readable network. The metadata for the semantic web has been constructed using the RDF framework. RDF gives flexibility to the semantic web to store and access different kinds of documents. One of the reasons many question answering systems are based on RDF data is due to its triple format. Since every triple has two entities connected by a property, the data is stored as a graph. This linked data is helpful for searching and querying purposes. SPARQL has now emerged as the standard language to query graph databases represented in the RDF format. This report describes a question answering system that can translate questions in English to SPARQL query format.

Question answering systems have increased the ease of access of knowledge bases for end users. The focus of recent developments has been on getting more concise and accurate results with state-of-the-art methods. Automatic generation of RDF queries is important because query languages like SPARQL can be challenging to learn for most users. Song, Huang, and Sun [1] make use of query expansion techniques and semantic query graph generation to map the subgraphs to a logical form. Dai, Li, and Xu [2] proposed a deep recurrent neural network along with conditional probabilities based on neural embedding to answer single fact questions. Liu et al. [3] described a model that ranks the subject-predicate pair to retrieve relevant facts from the question. It solves the problem of out-of-vocabulary words and disambiguation with word and character level embedding. Generating and evaluating SPARQL queries from natural language questions (NLQ) is an open research problem. The approach proposed in this report starts with a linguistic analysis of the question with part-of-speech tagging and dependency parsing. These are quantified and fed into a recursive neural network model for a classification task. The outputs of the classification task and entity recognition task are combined to build a complete SPARQL query.

The rest of this paper is organized as follows: Section II highlights the concepts of RDF data, SPARQL language, and related technologies used in our system. Section III describes the dataset used for training the neural network, the design of the system, and the techniques used to build the components. Section IV presents the analysis for the design choices and establishes the results achieved by the system. Lastly, section V concludes the paper with future improvements to the system.

CHAPTER 2. BACKGROUND

This chapter aims to provide a background of the concepts and technologies relevant to this system. It covers the idea of linked data, the vector representations used in natural language processing, and the Tree LSTM model used in the implemented system.

A. Linked Data, RDF, and SPARQL

As described by the W3C, “The Semantic Web provides a common framework that allows data to be shared and reused across application, enterprise, and community boundaries." [4] This web of data is a large collection of interrelated datasets called the linked data. The web maintains not only easy access to documents but also the relation between the documents. This relation can be any property applicable to the data within the document or the document itself. For example, the document for the book ‘Moby-Dick’ can be connected to the document for the person 'Herman Melville' as its writer. Other possible relations can be with the book's publisher, publication date, genre, etc.

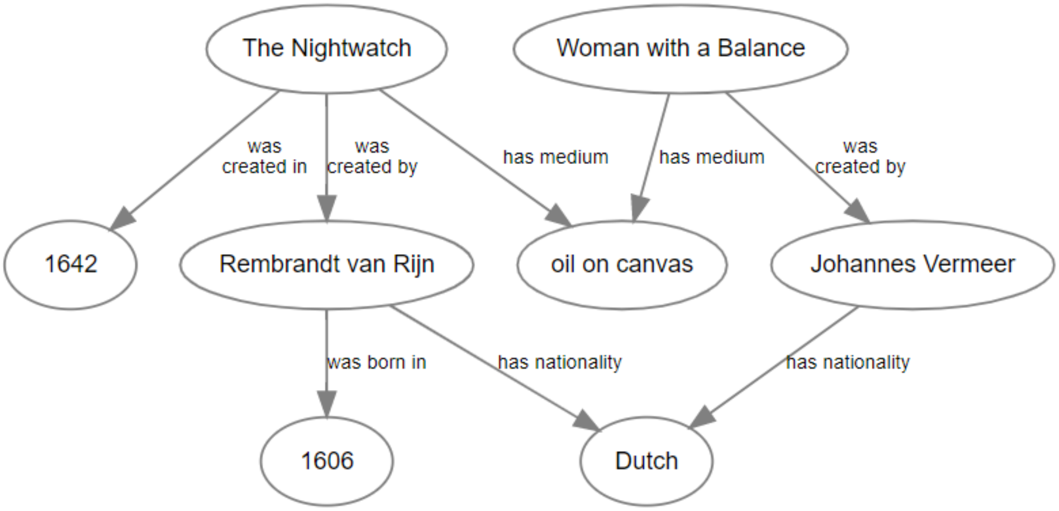
The documents on the web are in various formats like XML, HTML, relational, etc. There is a need for a standardized format to access all kinds of data and convert them from one form to another. The Resource Description Framework (RDF) models different formats of data to a machine-readable format. This makes it easy to describe and publish the document on the web. Its Uniform Resource Identifier (URI) refers to any document on the web. The URI is used for storage, access, and query operations. Resource descriptions in RDF are expressed as triples of subject, predicate, and object. For example, consider the following statement :

```
<The Nightwatch> <was created by> <Rembrandt van Rijn> .
```

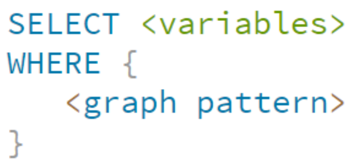
The subject here is the painting 'The Nightwatch', the object is the painter 'Rembrandt van Rijn' and the predicate 'was created by' defines the relationship between the subject and the object. A series of RDF triples are stored together as a database. An example database of records look like :

```
...
<The Nightwatch> <was created by> <Rembrandt van Rijn> .
<The Nightwatch> <was created in> <1642> .
<The Nightwatch> <has medium> <oil on canvas> .
<Rembrandt van Rijn> <was born in> <1606> .
<Rembrandt van Rijn> <has nationality> <Dutch> .
<Johannes Vermeer> <has nationality> <Dutch> .
<Woman with a Balance> <was created by> <Johannes Vermeer> .
<Woman with a Balance> <has medium> <oil on canvas> .
...
```

A graphical representation of the above database can be visualized as:

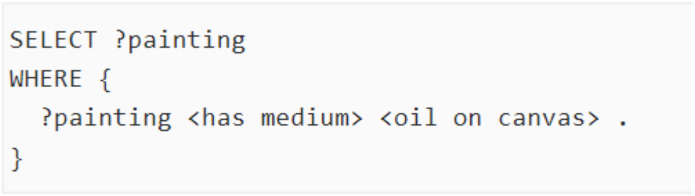


The large amount of data on the web can be stored easily with RDF, but it also needs to be accessible. Among the many query languages created to access and query RDF data, SPARQL is the predominant format recommended by W3C. The query can be structured with two main parts – the SELECT clause which defines the output variables for the query and the WHERE clause which provides the constraints on the query in the triple format.



```
SELECT <variables>
WHERE {
   <graph pattern>
}
```

Consider the search for all paintings that have medium as 'oil on canvas' for the previously shown RDF database. The equivalent SPARQL query for it will be –



```
SELECT ?painting
WHERE {
  ?painting <has medium> <oil on canvas> .
}
```

B. Natural Language Processing

Natural language processing deals with both computer science and linguistics. It involves reading, understanding, and manipulating human languages such that it makes sense to a machine. An important aspect of understanding the natural language is vectorization. This helps the machine to compute the value of each word and character in the language.

1. One-Hot Vector

One hot encoding is used to represent categorical data as a binary vector. A single bit corresponding to the relevant category is set in the vector. For any record in the data, the 1-D vector will always have only one bit set. For example, the one-hot encoding of the dataset `[red, green, green]` will be `[[1,0],[0,1],[0,1]]` where the first bit represents the `red` category and the second bit represents the `green` category.

2. Word Embedding

Another form of word vector representation is word embedding. It conveys the meaning of the word in a numerical format. Word embedding are created from neural networks that are extensively trained over large documents to produce the vectors. Words with similar meanings lie closer to each other in the vector space. They are significant features for text analysis due to their capability to capture the context of a word as well as the semantic and syntactic similarity between words.

C. Recursive Neural Networks

Recursive neural networks are a generalization of recurrent neural networks that use a graph or tree-based input instead of sequential input. Recurrent neural networks are time-dependent. At each step of the learning and evaluation process, the network considers the user input at the current step and the output of the hidden layer at the previous step. But, recursive neural networks are based on the structure of the input instead of the time. It uses a hierarchical structure like the parse tree of a sentence to traverse the user input. At every step, the network recursively takes the output of the operation performed on a smaller chunk of the text.

1. Long Short Term Memory (LSTM)

LSTM takes vectors of arbitrary length as input and processes information by sequentially applying a transition function to the hidden state of the network. The hidden state of the previous time step is used to compute the output of the current time step.

The LSTM transition or activation functions are nonlinearities that allow the network to compute complex problems using a small amount of data. An example of the activation function is the hyperbolic tangent function:

$$h_t = \tanh(W_{xt} + U_{ht-1} + b)$$

Here, W, U, and b are the parameter vectors that are learned by the model in the training phase.

RNNs usually deal with a problem due to the activation functions and long input sequences. During the training cycle, the gradient vector can be very small or very large, leading to exponential growth or decay in the weights of the network, respectively. This is called the vanishing gradient problem and may prevent the network from learning further. [5]

The LSTM model addresses this issue of learning long-term dependencies by using a memory cell that is able to preserve state over long time periods. The LSTM unit at each time step can be defined as a collection of input vectors, an input gate i, a forget gate f, an output gate o, a memory cell c and a hidden state h. The values of the gating vectors i, f and o are in [0, 1]. The LSTM transition equations at each time step t are as follows:

$$i_t = \sigma(W^{(i)}x_t + U^{(i)}h_{t-1} + b^{(i)})$$

$$f_t = \sigma(W^{(f)}x_t + U^{(f)}h_{t-1} + b^{(f)})$$

$$u_t = \sigma(W^{(u)}x_t + U^{(u)}h_{t-1} + b^{(u)})$$

$$o_t = \sigma(W^{(o)}x_t + U^{(o)}h_{t-1} + b^{(o)})$$

$$c_t = i_t \odot u_t + f_t \odot c_{t-1}$$

$$h_t = o_t \odot \tanh(c_t)$$

where x_t is the input vector at the current time step, σ denotes the logistic sigmoid function as the activation function and $\odot$ denotes elementwise multiplication. The forget gate controls what information from the previous hidden state is discarded or preserved, the input gate decides which values in the input vector are updated, and the output gate computes the current hidden state to be carried over to the next time step. The hidden state vector of a time step is a partial view of the memory state of the current cell. LSTM functions for varying lengths of inputs because the gated vector values keep changing for every vector.

2. Tree-LSTM

Recursive neural networks (RvNN) are non-linear models that learn information from structured data. RvNNs have been successful in natural language processing tasks such as phrase and sentence predictions. Apart from this, they are also used for classification tasks as well as supervised and unsupervised learning. Compared to other learning approaches based on fixed feature vector length, RvNN provides the flexibility to work with arbitrary length input vector [6]. Tree-LSTM is an extension of the LSTM model. The sequential input of the standard LSTM can be represented as a tree structure where each node has exactly one child. Fig. 1 depicts this difference between the architecture models of the standard LSTM and the Tree-LSTM.

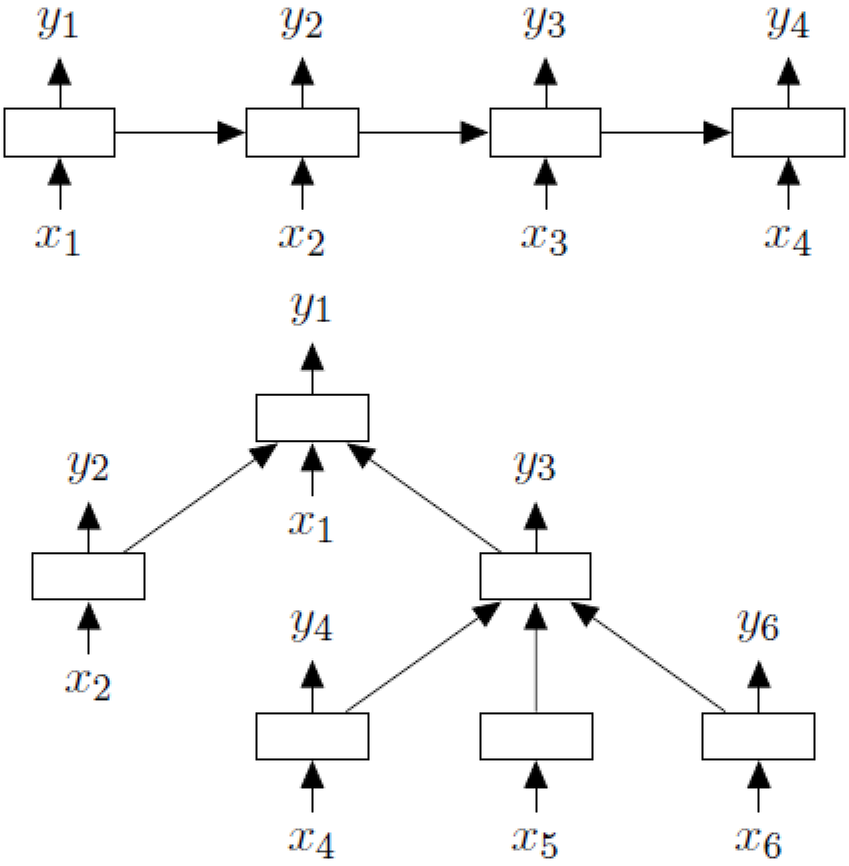


Figure 1. (above) LSTM model architecture; (below) Tree-LSTM model architecture

Our Tree-LSTM implementation is based on the model proposed by Tai et al. [7]. Unlike the standard LSTM, the Tree-LSTM computes its hidden state from the current input vector and the hidden states of its child units.

Each Tree-LSTM node contains input gate i_j, output gate o_j , a memory cell c_j , hidden state h_j, and the input vector x_j, where x_j is the vector representation of a word in a sentence. The gating vectors and memory cell updates for a node are dependent on the states of its child nodes. Each node contains one forget gate f_{jk} for each of its child k. This allows the node to include partial information from each child. Tree-LSTM has been found to be useful in semantic relatedness between sentence pairs and sentiment classification tasks. [7]

For a tree, the children of a node j are denoted by $C(j)$. The Tree-LSTM transition equations are the following:

$$\widetilde{h_J} = \sum_{k \epsilon C(j)} h_k$$

$$i_j = \sigma\left(W^{(i)}x_j + U^{(i)}\widetilde{h_J} + b^{(i)}\right)$$

$$f_{jk} = \sigma(W^{(f)}x_j + U^{(f)}h_k + b^{(f)})$$

$$o_j = \sigma(W^{(o)}x_j + U^{(o)}\widetilde{h_J} + b^{(o)})$$

$$u_j = \sigma(W^{(u)}x_j + U^{(u)}\widetilde{h_J} + b^{(u)})$$

$$c_j = i_j \odot u_j + \sum_{k \epsilon C(j)} f_{jk} \odot c_k$$

$$h_j = o_j \odot \tanh(c_j)$$

The Tree-LSTM learns a sentence using the sequence of words and its dependency parsed tree structure. The model begins traversing the tree from its leaf nodes. It learns the hidden state of the leaves and passes it on to their corresponding parent nodes to derive their hidden state and so on, until the traversal reaches the root node. At each level, the learning occurs in the breadth-first approach. Finally, the output from the root node is converted into a N_t dimensional vector using a softmax classifier, where N_t is the number of classes available for classification. To predict template $\hat{t}$ from the set of templates T, the softmax value at the root node is calculated, followed by the argmax to classify the template for the given input with the following equations:

$$\widehat{p_\theta}(t \mid x_{root}) = softmax(W^{(s)} h_{root} + \ b^{(s)})$$

$$\hat{t} = argmax\, \widehat{p_\theta}(t \mid x_{root})$$

The cost function used is the negative log-likelihood of the true class label y and λ is the L2-Regularization hyperparameter as given below:

$$J(\theta) = \ -\log \widehat{p_\theta}\ \ (y \mid x_{root}\ ) + \ \frac{\lambda}{2}\ ||\theta||$$

CHAPTER 3. DESIGN AND IMPLEMENTATION

This chapter focuses on the tools, technologies, environments and dataset used in the system. It also discusses the preprocessing activities performed on the dataset and the implementation of each module in the system.

D. Tools and Environments

This project uses Python 3.7 as the programming language for the implementation of the Tree-LSTM and the query construction. The reason for using Python is its extensive support for machine learning and data transformation libraries. The classification model was built using the PyTorch library. The basic building blocks provided by PyTorch were leveraged for constructing a custom machine-learning model. The code was executed on Google Collaboratory. This service connects to the Google cloud storage with authentication and also facilitates GPU processing for larger datasets.

For natural language processing, external packages were imported. Stanza [8] was used for syntactic analysis and FastText [9] provided the word embedding for tokens. Falcon 2.0 [10] and OpenTapioca [11] were explored for named entity recognition and linking to the Wikidata [13] database.

E. Dataset

The system is trained with the Lc-QuAD dataset [13] that has over 30,000 questions from 38 unique SPARQL templates. The Lc-QuAD dataset consists of three types of queries – list, boolean, and count. The Boolean queries answer the question as either true or false, whereas the count queries answer the question as a cardinal number. This report focuses only on simple questions from the list queries that answer the question as multiple entities from the knowledge graph. Each record in the dataset is structured as a natural language question with the template ID and the equivalent SPARQL query expected. Another property of each record is the canonical form of the natural language question that is generally incorrect.

During preprocessing, 6549 list queries are filtered out with 8 unique templates. The templates are listed below:

Template ID	SPARQL Question Template	Total Count
1	SELECT DISTINCT ?uri WHERE { <S> <P> ?uri }	3304
2	SELECT DISTINCT ?uri WHERE { ?uri <P> <O> }	740
3	SELECT DISTINCT ?uri WHERE { <S> <P1> ?uri . ?uri <P2> <O> }	2505
4	SELECT DISTINCT ?uri WHERE { <S> <P1> <O> . <O> <P2> ?uri }	3713
5	SELECT DISTINCT ?uri WHERE { <S> <P1> ?obj . ?obj <P2> ?uri }	2969
6	SELECT DISTINCT ?uri WHERE { <S> <P1> <O> . <O> <P2> ?uri }	2943
7	SELECT DISTINCT ?uri WHERE { ?uri <P> <O> . ?uri rdf:instance <O> }	2042
8	SELECT DISTINCT ?uri WHERE { <S> <P> ?uri. ?uri rdf:instance <O> }	1872

We created a dataset from template IDs 1, 2, and 3 consisting of 600 questions. The grammar of the questions, as well as the classification of the template, is corrected in the dataset. The final step splits the dataset in an 8:2 ratio for the training and testing files.

F. System Design

In line with the proposed techniques of Diefenbach et al.[14] the main components of the proposed system are as follows:

1. **Question Analysis** This step applies different techniques to identify the distinguishing features of the tokens in the sentence.
2. **Template Classification** This phase trains the recursive neural network with the features identified from the previous analysis step to identify the SPARQL template.
3. **Phrase Matching** This phase matches the phrases recognized in the question to the item or property labels/aliases in Wikidata.
4. **Query Construction** This phase constructs the final SPARQL query by filling the matched items and properties in the resulted template.

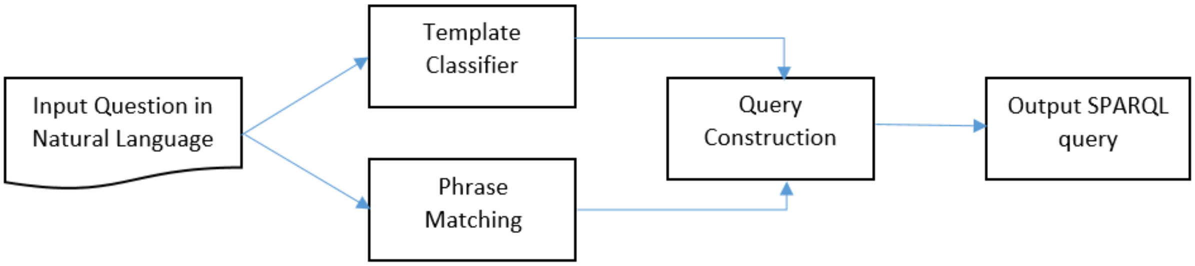


Figure 2.Design of the proposed system

1. Question Analysis

a. Part of Speech Tagging

Part of Speech (POS) tagging is the first step in the neural pipeline after sentence tokenization. This module annotates the tokens with their POS tags like noun, adjective, verb, etc.

The system uses the Stanford Stanza [8] package to generate the morphological features of the natural language string. The library is built with neural network components implemented on top of PyTorch. It includes features related to text analytics like multi-word token expansion, dependency parsing, and named entity recognition. Stanza labels tokens with universal POS tags [15] as well as Penn Treebank-specific POS tags [16]. Here, the treebank-specific POS tags are used for English language semantics.

Example usage of the POS tagger is shown in Fig. 2 for the question: What is the capital city of Demark?

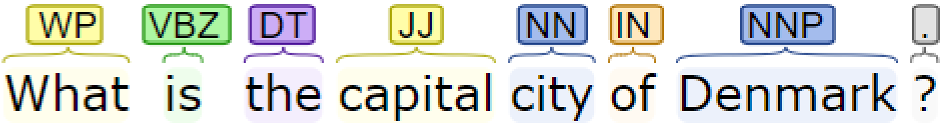


Figure 3. Example of the POS tagger in Stanza

b. Dependency Tree

Sentence parsing means recognizing the syntactic structure of the sentence with a formal grammar. The parser produces a set of rules that were used to generate the sentence from the language grammar. POS tags do not provide enough information on the relationship between different tokens in the sentence.

The Stanza parser builds a tree structure from the input sentence to represent the dependency relations between tokens. The resultant tree is represented with the Universal Dependencies formalism [17]. The parser uses the POS tags and the results from the lemmatization step to produce the tree.

The output for an example dependency parsing is shown in Fig 3. for the same question: What is the capital city of Denmark?

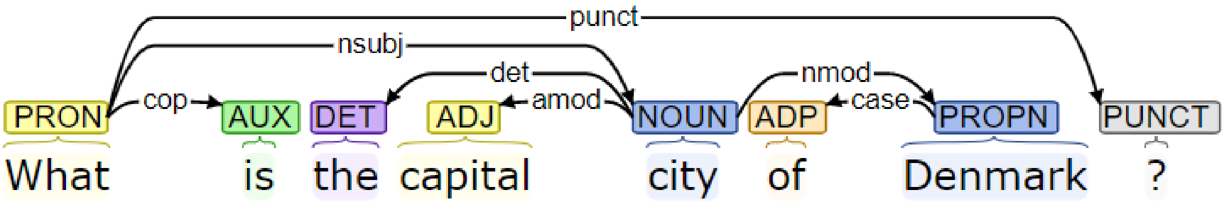


Figure 4. Example of the dependency parsing in Stanza

The output of the parser contains two important types of information. The first is the typed dependency, which is the simple relationship between all words in the question. These dependencies are represented as triples of the relation between two words. These are referred to as relation tags. The second type of information is the dependency parse tree which has words in the parent pointer format. The parse tree is an N-ary tree where each node has a pointer to its parent and not the child. The traversal from any node can only move upwards towards its ancestors and nowhere else. This ensures sequential processing that is useful when training the template classifier model.

c. Word embedding

For word embedding, our system uses Facebook's FastText embedding model [9]. FastText is based on the skip-gram model. Each word is represented as a vector of n-gram characters. FastText. It predicts the surrounding words in the text for each input word. This makes it a better choice to handle out-of-vocabulary words. The entire dataset has over 1500 unique tokens that are transformed into 300x1 vectors using the FastText embedding model.

d. One hot vector

One-Hot encoding is a common word vectorization technique for categorical data. The transformed vector is binary where the position for the current word is set to 1 and other values are 0. For example, the number of POS tags that can be identified by the Stanza library is

43. Each tag is converted to a vector of dimension 43 x 1, where the index of the current tag is set to value 1. This enables the model to learn and predict based on grouping labels.

2. Template Classification

The Child Sum variation of the Tree-LSTM was implemented using the PyTorch library according to Section 2.C. The proposed system is built using the original framework provided by Tai et al. in Lua script. [7]

The template classification implementation contains the following modules:

Dependency parser: Post cleaning and splitting of the dataset, the training dataset is used to build the feature parameters. The dependency parser implements the question analysis module described in the previous section. It tokenizes each record in the training datasets and creates files for its respective tokens, parent nodes, relationship dependencies, parts of speech, and the length of the sentence.

Vocabulary builder: This module builds the vocabulary for categorical data like the tokens, relationship dependency tags, characters, part-of-speech tags, and the output templates.

Embedding model builder: Embedding models for input features are built with the word embedding and one-hot vector techniques.

Dependency Tree Builder: The feature files are also used to construct the dependency parse tree over which the Tree-LSTM will traverse recursively.

Tree LSTM model: The dependency tree, embedding models, and vocabulary files are used to train the template classification model.

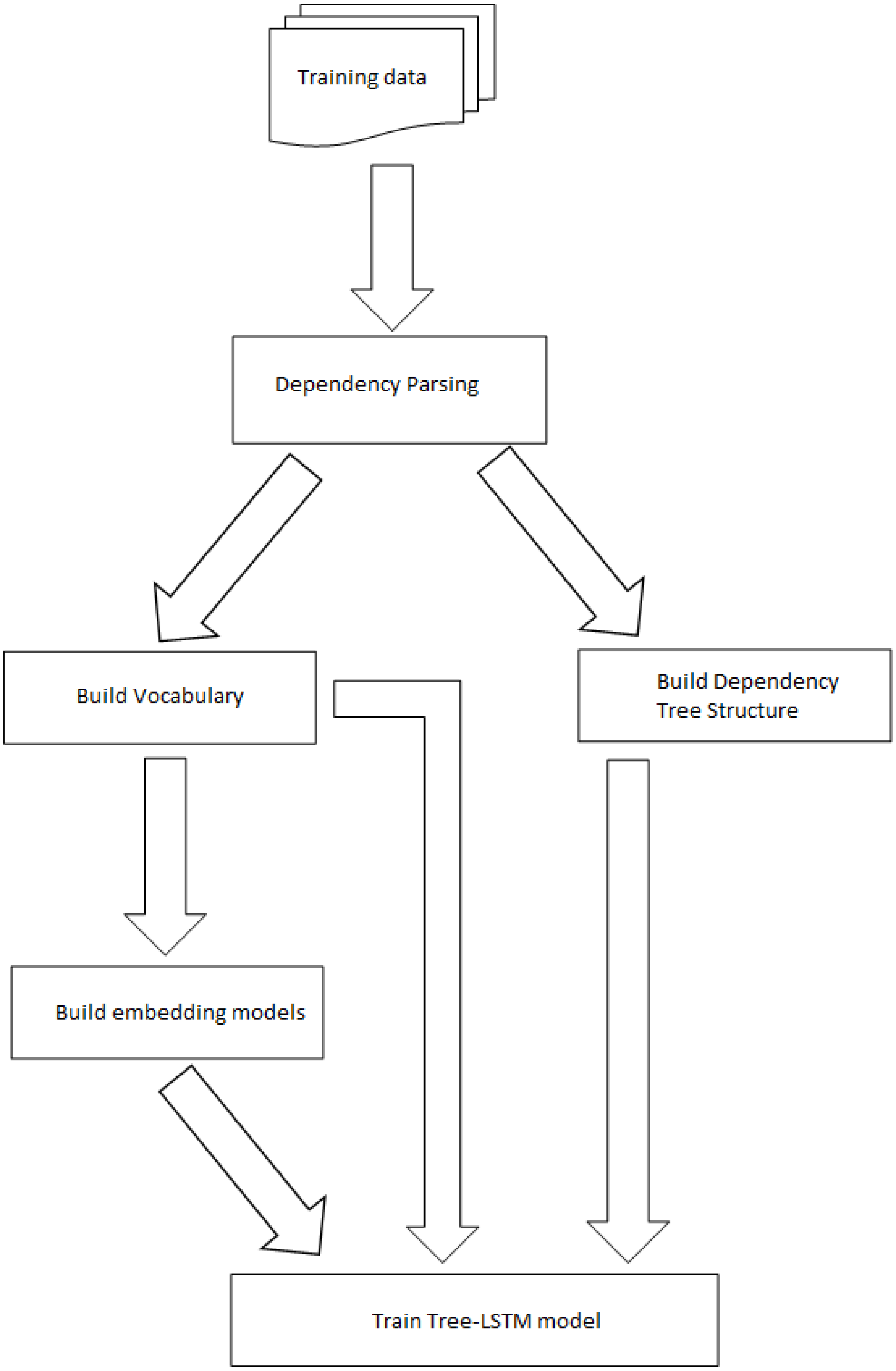


Figure 5. Modules in the Question Analysis and Template Classification phase

3. Phrase Matching

The third step of our system is the phrase matching module which performs named entity recognition and relation extraction. To convert any natural language to a different form, it is essential to recognize the construct and the relevant tokens in the sentence. Named entity recognition is meant to identify these tokens that can be utilized in the query.

There are existing libraries built over Wikidata that perform these tasks with sophisticated models. One of these libraries – Falcon 2.0 [10], was used to identify the entities and the relations. Falcon 2.0 is a joint entity and relation linking tool over Wikidata that outputs a list of entities and relations annotated with their candidates in Wikidata. These candidates are represented by their International Resource Identifier (IRI) in Wikidata. It is built specifically for short natural language text in the English language. It uses techniques like N-gram tilling and N-gram splitting for the recognition model and then performs optimization for the linking task.

The system makes use of the Falcon 2.0 Web API to send a POST request with the user question as a field in the JSON payload. The library request returns a JSON object with the list of entities and relations identified. For an example question, 'What is the capital city of Denmark?' the output of the API request is as follows:

```
{
  "entities_wikidata": [
    [
      "<http://www.wikidata.org/entity/Q35>",
      "Denmark"
    ]
  ],
  "relations_wikidata": [
    [
      "<http://www.wikidata.org/entity/P36>",
      "capital city"
    ]
  ]
}
```

4. Query Construction

The Tree-LSTM model identifies the SPARQL query template based on the semantic structure of the user question. The system then maps the user question with the underlying knowledge graph to capture the items that can be filled in the template slots. There are two kinds of slots in the template: Resources and Predicates. The resources and predicates are mapped to the identified items and properties in Wikidata, respectively. For example, consider the question ‘What is the capital city of Denmark?’ The underlying template detected for this question from the template classification model would be :

```
SELECT DISTINCT ?answer WHERE { ?answer wdt:<P> wd:<R> }
```

The template here requires one resource and one predicate to be completed. With the phrase matching results, the completed query can be constructed as:

```
SELECT DISTINCT ?answer WHERE { ?answer wdt:P36 wd:Q35 }
```

In the case of multiple items or properties identified for one slot of the template, there are multiple variations of the final query.

The correctness of the query is subjective to interpretation by the human mind. The method used for this system is by verifying a sample of the query results with pre-determined queries created manually. Having multiple verification points increases the validity of the query.

CHAPTER 4. ANALYSIS AND RESULTS

This chapter describes the experiments conducted on our system. It presents the results for the correctly identified templates by our system with different subsets of the data. Lastly, this report tries to find how many queries were constructed with the correct templates and triples.

A. *Template Classification*

The training dataset contains 600 queries of the list type from 3 templates. The testing dataset contains a random distribution of 120 queries from the corrected dataset.

1. Template count in the training dataset

This experiment is to check the impact of the size and composition of different templates on the accuracy of the classifier model. The model was trained for 20 epochs for each dataset.

	Composition of the training dataset	Correctly identified templates	Total test data
Uncleaned full set records of three templates	Template 1 (3327) + Template 2 (740)+ Template 3 (2505)	56.5%	6572
Cleaned dataset with subset of two templates	Template 1 (200) + Template 2 (200)	70%	400
Cleaned dataset with subset of three templates	Template 1 (200) + Template 2 (200) + Template 3 (200)	72.83%	600

2. Feature Selection

The experiments of feature selection focused on the cleaned subsets of data for template ids 1, 2, and 3. Each experiment was trained for 20 epochs with the previously defined parameters.

Test data Composition	Feature List	Accuracy
Template 1 + Template 2	Dependency Tree + Word Embedding	62.5%
	Dependency Tree + Parts-of-speech + Word embedding	65.5%
	Dependency Tree + Parts-of-speech + Relation tags + Character + Word Embedding	70%
Template 1 + Template 2 + Template 3	Dependency Tree + Word Embedding	71.6%
	Dependency Tree + Parts-of-speech + Word embedding	72.5%
	Dependency Tree + Parts-of-speech + Relation tags + Character + Word Embedding	72.83%

The results depict that parts-of-speech tags add important information to the model. It is clear that the performance of the model varies with the dataset composition of templates and the features that are used for training it.

B. Parameter Tuning

Due to the small size of the training dataset, the model had to be heavily regularized and the learning rate periodically slowed down to prevent overfitting. This was done by the following three strategies:

1. Weight decay – This is a component of the weight update rule that changes the weight for every epoch by multiplying it with a factor less than 1. This prevents the gradient from exploding with large values.

2. Dropout – A large number of parameters in neural networks allow for a more powerful model, but it can also lead to overfitting of training data. Dropout enables random dropping of

units with their connections from the network. This ensures that the network does not develop complex adaptation on the training data.

3. Adaptive learning rate – The learning rate for the model was determined to be 1 x 10^{-1}, but the accuracy of the model seemed to plateau after a few epochs. This could be due to the rapid overfitting of the training data. To prevent this, a step scheduler was added that periodically decreased the learning rate after a few epochs. In this model, the learning rate was decreased by a multiplicative factor of 0.1 x 10^{-3} after every 5 epochs.

The final parameters of the model are defined in the below table:

Parameter	Value
Input dimensions	444 x 1
Tree-LSTM memory dimensions	150 x 1
Epochs	20
Batch size	25
Learning rate	1 x 10^{-1}
Weight decay	0.1 x 10^{-3}
Dropout	0.2
Loss function	Cross entropy loss
Optimizer	Adam optimizer
Scheduler	Stepwise learning rate decay
LR step size	once every 5 epochs
LR step decay	0.1

Figure 6. Table of hyperparameter values for the model trained

C. Phrase Matching

There are two libraries available for entity recognition in Wikidata from an input sentence – Falcon 2.0 and Open Tapioca. Falcon 2.0 is an entity and relation linker where it recognizes the items and properties from Wikidata, whereas OpenTapioca only recognizes items belonging to locations, people, and organizations in the question. OpenTapioca accurately identifies the placements of entities and suggests the best possible Wikidata item match using log-likelihood. The Falcon 2.0 is preferred because it is a combination of the entity and relation linking modules.

D. Query Construction

From the testing dataset of 120 questions, we create a sample of 20 random queries constructed by the system. These SPARQL queries are compared with manually written queries for the same question. If the template and the resource identifiers for the entity and relations are identified correctly in the system-generated SPARQL query, then it is marked as a correct query. The system-generated queries were verified with two sets of manually generated SPARQL queries. On a sample of 20 questions, the average correctness is 60%. Since the system is designed for simple questions , the two sets of manually generated queries are 90% similar.

A few example outputs are as below :

```
What is Sanskrit’s writing system?
['SELECT DISTINCT ?answer WHERE { wd:Q11059 wdt:P282 ?answer}',
'SELECT DISTINCT ?answer WHERE { wd:Q58778 wdt:P282 ?answer}']

What is the total equity of Micron Technology?
['SELECT DISTINCT ?answer WHERE { wd:Q1197548 wdt:P2137
?answer}']

What has the decay mode as alpha decay?
['SELECT DISTINCT ?answer WHERE { ?answer wdt:P817 wd:Q179856}']

What has its headquarters at Zibo?
['SELECT DISTINCT ?answer WHERE { ?answer wdt:P159 wd:Q198370}']

What is the magnetic moment of an electron?
['SELECT DISTINCT ?answer WHERE { wd:Q27342851 wdt:P2069
?answer}']
```

Looking at the results, we can classify the errors of the system into the following types:

1. Misclassification of template – This kind of error is due to the Tree-LSTM model incorrectly identifying the SPARQL query template for the user input question.
2. Incorrect entity and relation linking – The phrase matching phase uses an external library for the identification of items and properties in Wikidata. The limitations in Falcon 2.0 are not in our control. Such errors can be mitigated by creating a custom linking module.
3. Multiple triple candidates – Falcon 2.0 can recognize multiple entities and relations from the user question. These items may or may not be a part of the correct query. The system constructs all possible combinations of the SPARQL that can be created from these entities and relations. A wrong selection of the final query from the candidates is possible.

CHAPTER 5. CONCLUSION

The proposed system implements an end-to-end question answering system for RDF data with various techniques in machine learning and natural language processing to generate a correct SPARQL query. The resultant query can be executed using the Wikidata query service to get the desired results.

One of the limitations of the system is the lack of ontology recognition. Currently, the system can only match phrases in the default Wikidata ontology because of the use of third-party entity linking libraries. Customizing the phrase-matching phase using the dataset and the features could result in better matching accuracy.

As part of further developments, the training dataset can be extended to a more extensive template coverage as well as an increase in the number of questions under each template. This increases the possibility of the system recognizing more patterns correctly and prove to be a more complete question answering system.

REFERENCES

[1] Song S., Huang W. and Sun Y. “Semantic query graph based SPARQL generation from natural language questions” in *Cluster Computing* 22**,** pp. 847–858, 2019. https://doi.org/10.1007/s10586-017-1332-3

[2] Dai Z., Li L. and Xu W., “CFO: Conditional Focused Neural Question Answering with Large scale Knowledge Bases” in *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics*, pages 800–810, Berlin, Germany, August 7-12, 2016

[3] Liu Y., Zhang T., Liang Z., Ji H., and McGuinness D. “Seq2RDF: An End-to-end Application for Deriving Triples from Natural Language Text”*,* 2018.

[4]https://www.w3.org/2001/sw/#:~:text=The%20Semantic%20Web%20provides%20a,Resource%20Description%20Framework%20(%20RDF).

[5] Hochreiter S., “The vanishing gradient problem during learning recurrent neural nets and problem solutions” in *International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems,* pp. 107–116, 1998.

[6] Chinea A., “Understanding the principles of recursive neural networks: A generative approach to tackle model complexity”, in *International Conference on Artificial Neural Networks*, pp. 952–963, 2009.

[7] Tai K. S., Socher R. and Manning C. D., “Improved semantic representations from tree-structured long short-term memory networks”, 2015.

[8] https://stanfordnlp.github.io/stanza/

[9] https://fasttext.cc/

[10] Sakor, A., Singh K., Patel A., and Vidal M., “Falcon 2.0: An Entity and Relation Linking Tool over Wikidata”, 2019.

[11] Delpeuch A., “OpenTapioca: Lightweight Entity Linking for Wikidata”, 2019.

[12] https://www.wikidata.org/wiki/Wikidata:Main_Page

[13] Trivedi, Priyansh and Maheshwari, Gaurav and Dubey, Mohnish and Lehmann, Jens, “Lc-quad: A corpus for complex question answering over knowledge graphs" in *International Semantic Web Conference*, pp.210-218, 2017

[14] Diefenbach D., Lopez V., Singh K. and Maret P., “Core techniques of question answering systems over knowledge bases: a survey” in *Knowledge and Information systems*, pp. 1–41, 2017.

[15] https://universaldependencies.org/u/pos/

[16] https://www.ling.upenn.edu/courses/ling001/penn_treebank_pos.html

[17] https://universaldependencies.org/