AUTOMATIC QUESTION GENERATION

A PROJECT REPORT
SUBMITTED IN FULFILLMENT OF THE REQUIREMENTS
FOR THE AWARD OF DEGREE
OF
BACHELOR OF TECHNOLOGY
IN
SOFTWARE ENGINEERING

Submitted By:

NISHANT TANWAR (2K16/SE/055)
SANSKAR JAIN (2K16/SE/073)
SARVAGYA BHARGAVA (2K16/SE/074)

Under the supervision of
Ms. Minni Jain

Department of Computer Science & Engineering
Delhi Technological University
(Formerly Delhi College of Engineering)
Bawana Road, New Delhi - 110042
APR, 2019

Department of Computer Science & Engineering
Delhi Technological University
(Formerly Delhi College of Engineering)
Bawana Road, New Delhi-110042

CANDIDATE'S DECLARATION

We , Nishant Tanwar (2K16/SE/055), Sanskar Jain (2K16/SE/073) and Sarvagya Bhargava (2K16/SE/074) of B.Tech., hereby declare that the project report titled "Automatic Question Generation" which is submitted by us to the Department of Computer Science & Engineering (Software Engineering), Delhi Technological University, Delhi in partial fulfillment of the of the requirement for the award for the award of degree of Bachelor of Technology is original and not copied from any source without proper citation. This work has not previously formed the basis for the award of any Degree, Diploma Associateship, Fellowship or other similar title or recognition.

Place: Delhi Ms. Minni Jain

Date:

ABSTRACT

Texts with potential educational value are becoming available through the Internet (e.g., Wikipedia, news services). However, using these new texts in classrooms introduces many challenges, one of which is that they usually lack practice exercises and assessments. Here, we address part of this challenge by automating the creation of a specific type of assessment item.

Specifically, we focus on automatically generating factual WH questions. Our goal is to create an automated system that can take as input a text and produce as output questions for assessing a reader's knowledge of the information in the text. The questions could then be presented to a teacher, who could select and revise the ones that he or she judges to be useful.

After introducing the problem, we describe some of the computational and linguistic challenges presented by factual question generation. We then present an implemented system that leverages existing natural language processing techniques to address some of these challenges. The system uses a combination of manually encoded transformation rules on each sentence in a paragraph. Offering automatic suggestions reduced the time and effort spent by participants, though it also affected the types of questions that were created. This research supports the idea that natural language processing can help teachers efficiently create instructional content. It provides solutions to some of the major challenges in question generation and an analysis and better understanding of those that remain.

ACKNOWLEDGEMENT

With this project towards its completion, we would like to extend our sincere gratitude towards **Ms. Minni Jain** who acted as a direct guide during the entire course. We are sincerely grateful for their invaluable insight, the time and effort they dedicated to this work and, especially, for their contagious enthusiasm and optimism.

We are also grateful to the Department of Computer Science & Engineering (Software Engineering) at Delhi Technological University for presenting us with this research opportunity which was essential in enhancing learning and promote research culture among ourselves.

We would like to thank our families for their unconditional support and faith in our endeavours. Their prayers and wishes have played a major role in the successful completion of this project.

Nishant Tanwar Sanskar Jain Sarvagya Bhargava

CONTENTS

1. INTRODUCTION

1.1. General

How would someone tell whether you have read this text? They might ask you to summarize it or describe how it relates to your own research. They might ask you to discuss its strengths and weaknesses or to compare it to another paper.

Or, as a first step, just to see if you bothered to get through it, they might ask you what features we used in our ranking model, or what corpora we tested on. That is, at first, they might just ask you questions to check that you remember the basic facts. Then, if it is clear that you read more than the title and abstract, they might move on to more challenging questions. Of course, you are probably a highly skilled and motivated reader, and there would be no need to assess whether you read and retained basic factual information. However, that is not the case with all readers. For example, an elementary school teacher might ask his or her students basic questions since they are still learning to read.

Generating such questions, and authoring reading assessments more generally, can be a time consuming and effortful process. In this research, we work toward automating that process. In particular, we focus on the problem of automatically generating factual questions from individual texts. We aim to create a system for question generation (QG) that can take as input an article of text (e.g., a web page or encyclopedia article that a teacher might select to supplement the materials in a textbook), and create as output a list of factual questions. A user could then select and revise these questions in order to create practice exercises or part of a quiz to assess whether students read the text and retained knowledge about its topic. While QG about narratives and subjective essays would also be interesting and educationally relevant, we leave these problems to future work. To make our factual QG system generally useful, we avoid the use of domain-specific knowledge (e.g., about historical events or geographical locations) and instead focus on modeling fairly general lexical and syntactic phenomena related to questions and the presentation of factual information.

1.2. Applications

Asking relevant and intelligent questions has always been an integral part of human learning, as it can help assess the learner's understanding of a piece of text. However, compiling questions manually is arduous. Automated question generation (AQG) systems can help, as they have the ability to generate questions quicker and on a larger scale.

A typical scenario is evaluating students on reading comprehension, where it becomes tedious for a teacher to manually create questions, find answers to these questions, and then evaluate answer papers after the test has been administered. All these complex tasks can now be automated using an automatic question and answer generation system.

An automatic question generation system has applications in areas as diverse as FAQ generation, intelligent tutoring systems, and virtual assistants. Question generation can be naturally applied in the educational setting such as online courses, automated help systems, and search engines. It can also be applied in a wide variety of other domains — including chatbot systems (e.g. for customer interaction) and health care for analysing mental health.

One use case is the implementation of assisted / guided learning. That is, a question generation system will determine a learners capabilities. Based on the learner's capabilities, the QG system will generate types of questions, such as questions requiring factoid answers or questions involving more complex reasoning processes on the learners behalf. Another use case is online assistants, such as travel assistants. Here a customer inquire some company about a travel request. A QG system will gather travel specifics by questioning the customer. A variation of this use case is simple dialog agents or chat bots, much like Eliza, a dialog agent will take user input, rephrase it as a question conducting an endless dialogue.

2. BACKGROUND

2.1. Related Work

Recently, Question Generation (QG) and Question Answering (QA) in the field of computational linguistics have got enormous attention from the researchers (Rus and Graesser, 2009). Twenty years ago, receiving answers to the same questions would take hours or weeks through documents and books. After the computers and internet, wealth of information becomes available and this field holds great promise for making sophisticated question asking and answering facilities mainstream in the future.

There are various types of questions and researchers proposed different taxonomies for organizing them. Bloom (1956) defined taxonomy of education objectives. While ”knowledge” level focuses on facts, ”comprehension” level involves deeper understanding. Similar to Bloom’s objectives, Rus and Graesser (2009) divided questions into two categories, namely deep questions and shallow questions. If a learner wants to acquire difficult scientific and technical material, deep questions (such as why, why not, how, what-if, what-if-not) can be asked. These questions involve more logical thinking than shallow questions. Conversely, shallow questions focus more on facts (such as who, what, when, where, which, how many/much and yes/no questions). While some studies shows strong advantages for asking higher-level questions, these are discussed by Redfield and Rousseau (1981). Others shows no significant difference between asking shallow and deep questions. Deep and shallow questions may supplement each other (Heilman, 2011). While lower-level questions ensure learner to acquire basic knowledge about the subject, higher-level questions let learner to gather various pieces of information to formulate an answer.

Pioneering work in QG is done by Wolfe (1976), who demonstrated the automatically generated questions could be as effective as human generated questions. After that, sporadic researches are done by Kunichika et al. (2004), Mitkov et al. (2006) and Rus et al. (2007). After workshop on the Question

Generation Shared Task and Evaluation Challenge (Rus & Graesser, 2009), QG have got enormous attention from the researchers.

In 2011, Heilman addressed various question generation challenges in his paper:

• **Syntactic challenges**: First syntactic challenge occurs when using NLP tools (parsers) 5 for analyzing syntactic structure of the sentence. Sadly, current parsers are imperfect and sometimes they produce incorrect parse results. Thus, any QG system that relies on an incorrect parse result is most likely to generate ungrammatical questions. Complex syntactic constructions is an another problem. Even if a parser gives an accurate syntactic representation of a sentence, information extraction or transforming that structure becomes nontrivial.

• **Lexical challenges**: For lexical challenges, Heilman addressed mapping answers to question word is an another issue. Especially deciding between who and what question is problematic. Another lexical challenge is non-compositionality. In English, meaning of phrases is not always just a simple accumulation of its component words. Multiword expressions are one of the active research areas in NLP (Sag et al., 2002). It is a problem because most of syntactic parsers represents each word token independently.

• **Discourse challenges**: For discourse challenges, Heilman addressed information that is conveyed from one sentence to other is an another problematic issue. Consider the second sentence in the following example: ”The American professor Robert H. Goddard had worked on developing solid-fuel rockets since 1914, and demonstrated a light battlefield rocket to the US Army Signal Corps only five days before the signing of the armistice that ended World War I. He also started developing liquid-fueled rockets in 1921.” If we generate questions from the second sentence, with using simple transformations, one of the generated questions will be ”When did he start developing liquid-fueled rockets?” If this question specifically asked a reader after he immediately reads this paragraph, this question can be valid. However, most of the time these type of questions is vague and out of context, since there are many possible answers.

2.2. Parsers for Preprocessing

In this section parsers are explained one by one, with their own particular viewpoint examples.

2.2.1. Sentence Tokenization

Tokenizers are used to divide strings into lists of substrings. Sentence tokenizer can be used to find the list of sentences. It is the problem in natural language processing of deciding where sentences begin and end. sent_tokenize uses an instance of PunktSentenceTokenizer from the nltk. Tokenize.punkt module. This instance has already been trained on and works well for many European languages. So it knows what punctuation and characters mark the end of a sentence and the beginning of a new sentence.

sentence_list = nltk.sent_tokenize(text) #this gives a list of sentences

```
import nltk
```

```
text = '''We are students of Delhi Technological University.
        This project is made by Nishant, Sanskar and Sarvagya.'''
sentence_list = nltk.sent_tokenize(text)
sentence_list
```

```
['We are students of Delhi Technological University.',
 'This project is made by Nishant, Sanskar and Sarvagya.']
```

Figure 1: Sentence Tokenization

2.2.2. Part-Of-Speech (POS) Tagging

A Part-Of-Speech Tagger (POS Tagger) is a piece of software that reads text in some language and assigns parts of speech to each word (and other token), such as noun, verb, adjective, etc., although generally computational applications use more fine-grained POS tags like 'noun-plural'. This software is a Java implementation of the log-linear part-of-speech taggers.
There are mainly two type of taggers: rule-based and stochastic.

1. **Rule-based taggers** use hand-written rules to distinguish the tag ambiguity.

2. **Stochastic taggers** are either HMM based, choosing the tag sequence which maximizes the product of word likelihood and tag sequence probability, or cue-based, using decision trees or maximum entropy models to combine probabilistic features.

The part of speech explains how a word is used in a sentence. There are eight main parts of speech - nouns, pronouns, adjectives, verbs, adverbs, prepositions, conjunctions and interjections. POS tagging is a supervised learning solution that uses features like the previous word, next word, is first letter capitalized etc. NLTK has a function to get POS tags. Each sentence is first word tokenized using word_tokenize() function and then passed to the nltk.pos_tag function which returns the list words along with their POS tags.

tag_list = nltk.pos_tag(word_tokenize(sentence))

```
import nltk
```

```
text = 'We have been working on this project for 4 hours.'
tag_list = nltk.pos_tag(nltk.word_tokenize(text))
tag_list
```

```
[('We', 'PRP'),
 ('have', 'VBP'),
 ('been', 'VBN'),
 ('working', 'VBG'),
 ('on', 'IN'),
 ('this', 'DT'),
 ('project', 'NN'),
 ('for', 'IN'),
 ('4', 'CD'),
 ('hours', 'NNS'),
 ('.', '.')]
```

Figure 2: POS Tagging

2.2.3. Chunking

Chunking is a process that first detects constituents in a sentence, then segments them to chunks of syntactically related word groups. Each word is labeled with its own unique tags in the groups. Chunk tags have two parts. The first part indicates the beginning, continuation or end of a chunk. The second part indicates the type of the current word group. For example, beginning of a chunk noun phrase is labeled with B-NP, continuation of a chunk verb phrase is labeled with I-VP. An example

chunking tags of the sample sentence “The Bill of Rights gave the law federal government greater legitimacy.” can be seen as follows:

Token	POS Tag	Chunk Tag
The	DT	B-NP
Bill	NNP	E-NP
of	IN	S-PP
Rights	NNPS	S-NP
gave	VBD	S-VP
the	DT	B-NP
new	JJ	I-NP
federal	JJ	I-NP
government	NN	E-NP
greater	JJR	B-NP
legitimacy	NN	E-NP

2.2.4. Named Entity Recognition

Named Entity Recognition (NER) is an information extraction task that labels words into various semantic categories such as person, date, location, facility. There are various NER approaches based on rule based techniques and statistical models, i.e. machine learning. Rule based approaches rely on hand-crafted rules which is done by experienced linguists, whereas statistical approaches need large set of training data. Statistical models also allow to train custom models and define new categories according to problem domain.

```
import nltk
import spacy
import re
nlp = spacy.load('en')
```

```
def get_named_entities(sent):
    doc = nlp(sent)
    named_entities = [(X.text, X.label_) for X in doc.ents]
    return named_entities
```

```
sent = "Ram was held captive at Castle Black."
get_named_entities(sent)
```

```
[('Ram', 'PERSON'), ('Castle Black', 'ORG')]
```

Figure 3: Named Entity Recognition

3. PROPOSED METHODOLOGY

In this section, templates and their generation are explained in detail. In order to consider a sentence pattern as a template, it should satisfy the following criterias:

- The sentence pattern should be working on different domains.
- It should extract important points in the source sentence and create an unambiguous question.

Firstly, we break the given paragraph into sentences and then classify them into following three categories:

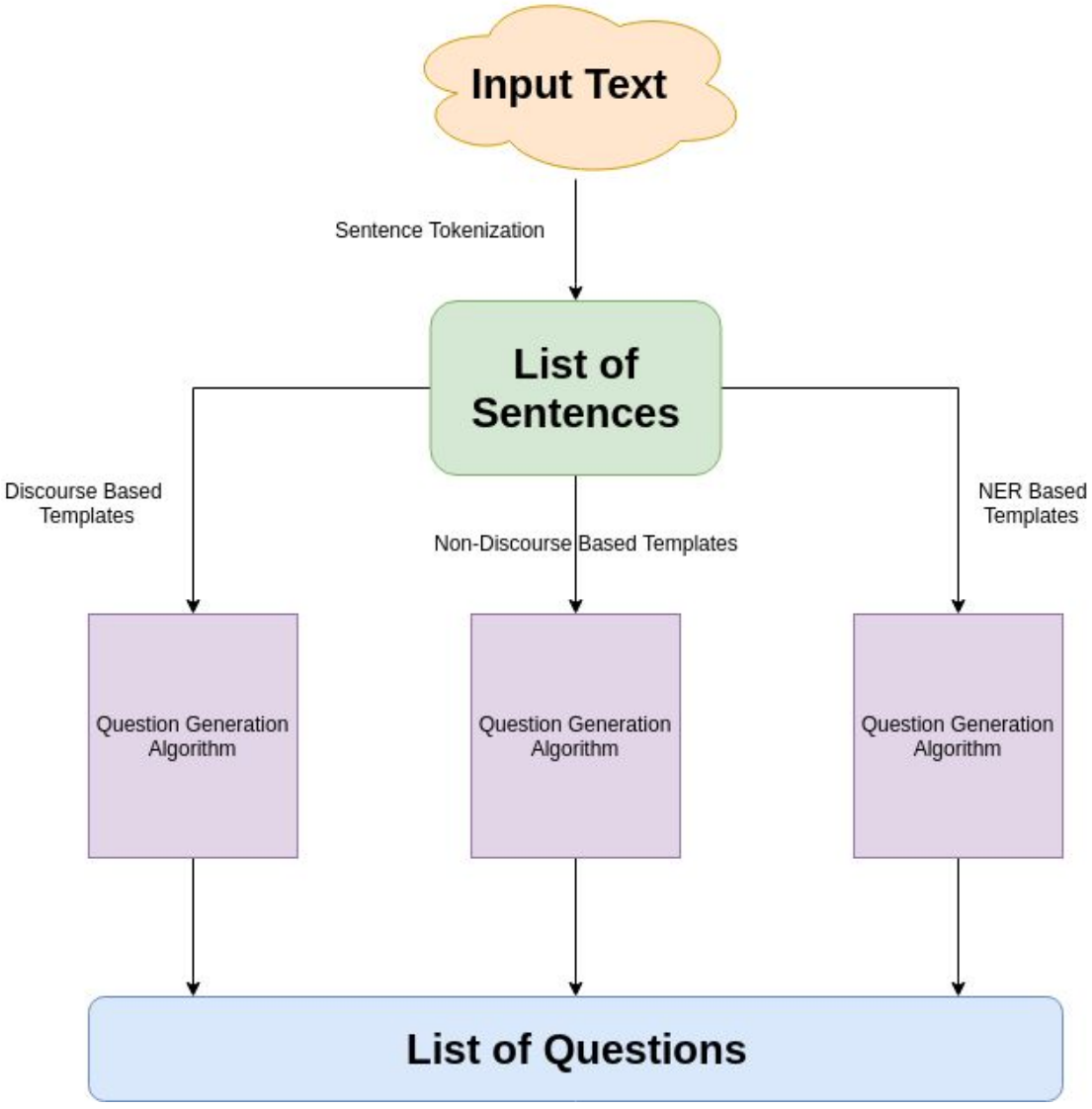


Figure 4: Proposed Methodology

3.1. Discourse Markers Based Templates

A sentence falls into this category if it contains any of the following discourse markers:
discourse_markers = ['because', 'as a result', 'since', 'when', 'although', 'for example', 'for instance']
The following **rule based algorithm** is used for the generation of the questions for the sentences that fall into this template:

- For every discourse marker present in the sentence, we define the question word associated with it.

```
# List of auxiliary verbs
aux_list = ['am', 'are', 'is', 'was', 'were', 'can', 'could', 'does', 'do', 'did', 'has', 'had', 'may', 'might', 'must',
 'ought', 'shall', 'should', 'will', 'would']

# List of all discourse markers
discourse_markers = ['because', 'as a result', 'since', 'when', 'although', 'for example', 'for instance']

# Different question types possible for each discourse marker
qtype = {'because': ['Why'], 'since': ['When', 'Why'], 'when': ['When'], 'although': ['Yes/No'], 'as a result': ['Why'],
'for example': ['Give an example where'], 'for instance': ['Give an instance where'], 'to': ['Why']}
```

Figure 5: Discourse Markers

- Then we identify the part of the sentence which will be used for the generation for the question.
- Now, we convert the sentence into lowercase and remove full stop.
- We then classify the sentence into two types: one having an auxiliary verb and one without an auxiliary verb.
- If the sentence has an auxiliary verb:
    - We first tokenize the sentence and generate tags using POS tagging.
    - Then, iterating over the tags we append each word into the question part until we find the first non auxiliary verb.
    - Split the question part across the auxiliary verb and place that auxiliary verb at the start of the question part.
    - Now, if the question type is of Yes/No, we simply return the formed question part with a question mark appended at the end, otherwise prepend the question word at the start of the question part and return.

```
# If auxiliary verb exists
if(aux_verb):

    # Tokeninze the part of the sentence from which the question has to be made
    text = nltk.word_tokenize(question_part)
    tags = nltk.pos_tag(text)
    question_part = ""

    for word, tag in tags:

        # Break the sentence after the first non-auxiliary verb
        if(re.match('VB*', tag) and word not in aux_list):
            question_part += word
            break
        question_part += word + " "

    # Split across the auxiliary verb and prepend it at the start of question part
    question = question_part.split(" " + aux_list[pos])
    question = [aux_list[pos] + " "] + question

    # If Yes/No, no need to introduce question phrase
    if(type == 'Yes/No'):
        question += ['?']

    elif(type != "non_disc"):
        question = [type + " "] + question + ["?"]

    else:
        question = question + ["?"]

    question = ''.join(question)
```

Figure 6: Discourse Marker Algorithm containing Aux Verb

- If the sentence does not contain any auxiliary verb:
  - We first tokenize the sentence and generate tags using POS tagging.
  - Then, we define some combination rules which help us to generate the required auxiliary verb.

```
# If auxilary verb does ot exist, it can only be some form of verb 'do'
else:
    aux = None
    text = nltk.word_tokenize(question_part)
    tags = nltk.pos_tag(text)
    comb = ""

    '''There can be following combinations of nouns and verbs:
        NN/NNP and VBZ  -> Does
        NNS/NNPS(plural) and VBP -> Do
        NN/NNP and VBN -> Did
        NNS/NNPS(plural) and VBN -> Did
    '''
```

Figure 7: Combination Rules

- Using the combination rules and POS tags, we set the auxiliary verb such as do, does, and did. Also, while appending the verb into the question part we first stem the verb using any stemmer or lemmatizer.

```
for tag in tags:
    if(comb == ""):
        if(tag[1] == 'NN' or tag[1] == 'NNP'):
            comb = 'NN'

        elif(tag[1] == 'NNS' or tag[1] == 'NNPS'):
            comb = 'NNS'

        elif(tag[1] == 'PRP'):
            if tag[0] in ['He','She','It']:
                comb = 'PRPS'
            else:
                comb = 'PRPP'
                tmp = question_part.split(" ")
                tmp = tmp[1: ]
                if(tag[0] in ['I', 'we', 'We']):
                    question_part = 'you ' + ' '.join(tmp)
    if(res == None):
        res = re.match(r"VB*", tag[1])
        if(res):
            # question_part = question_part[:question_part.index(tag[0]) + len(tag[0])]

            # Stem the verb
            question_part = question_part.replace(tag[0], stemmer.stem(tag[0]))
        res = re.match(r"VBN", tag[1])
        res = re.match(r"VBD", tag[1])
```

Figure 8: Discourse Marker Algorithm without Aux Verb

- Append the appropriate question type word at the start and return the question.

```
    if(comb == 'NN'):
        aux = 'does'

    elif(comb == 'NNS'):
        aux = 'do'

    elif(comb == 'PRPS'):
        aux = 'does'

    elif(comb == 'PRPP'):
        aux = 'do'

    if(res and res.group() in ['VBD', 'VBN']):
        aux = 'did'

    if(aux):
        if(type == "non_disc"):
            question = aux + " " + question_part + "?"

        else:
            question = type + " " + aux + " " + question_part + "?"
question = question[0].upper() + question[1:]
print(question)
```

Figure 9: Setting appropriate combination

Example: 'They were angry because their plans had been discovered.'

```
sentensify()
```

```
Why were they angry ?
```

Example: 'I think he felt included because he was helping as much as we were.'

```
sentensify()
```

```
Why did you think he felt included ?
```

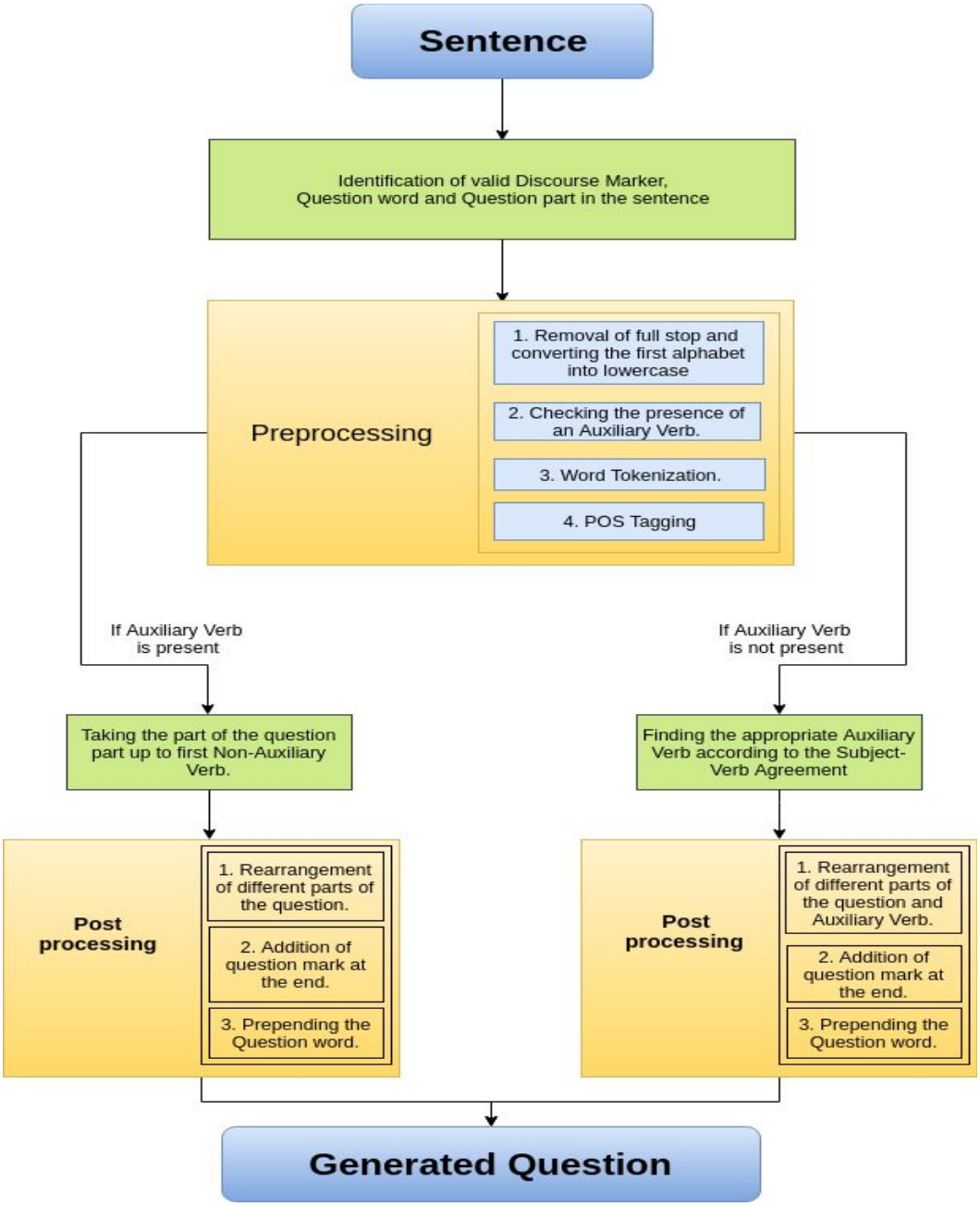


Figure 10: Discourse Markers Based Template Algorithm

3.2. Non-Discourse Markers Based Templates

Sentences that do not contain any discourse marker fall into this category such as simple assertive or declarative sentences or Yes/No sentences. Here, we simply use the appropriate auxiliary verb as the starting word of the question and the other procedure of generating the combination from POS tags is same as in discourse markers based templates which is illustrated in above images.

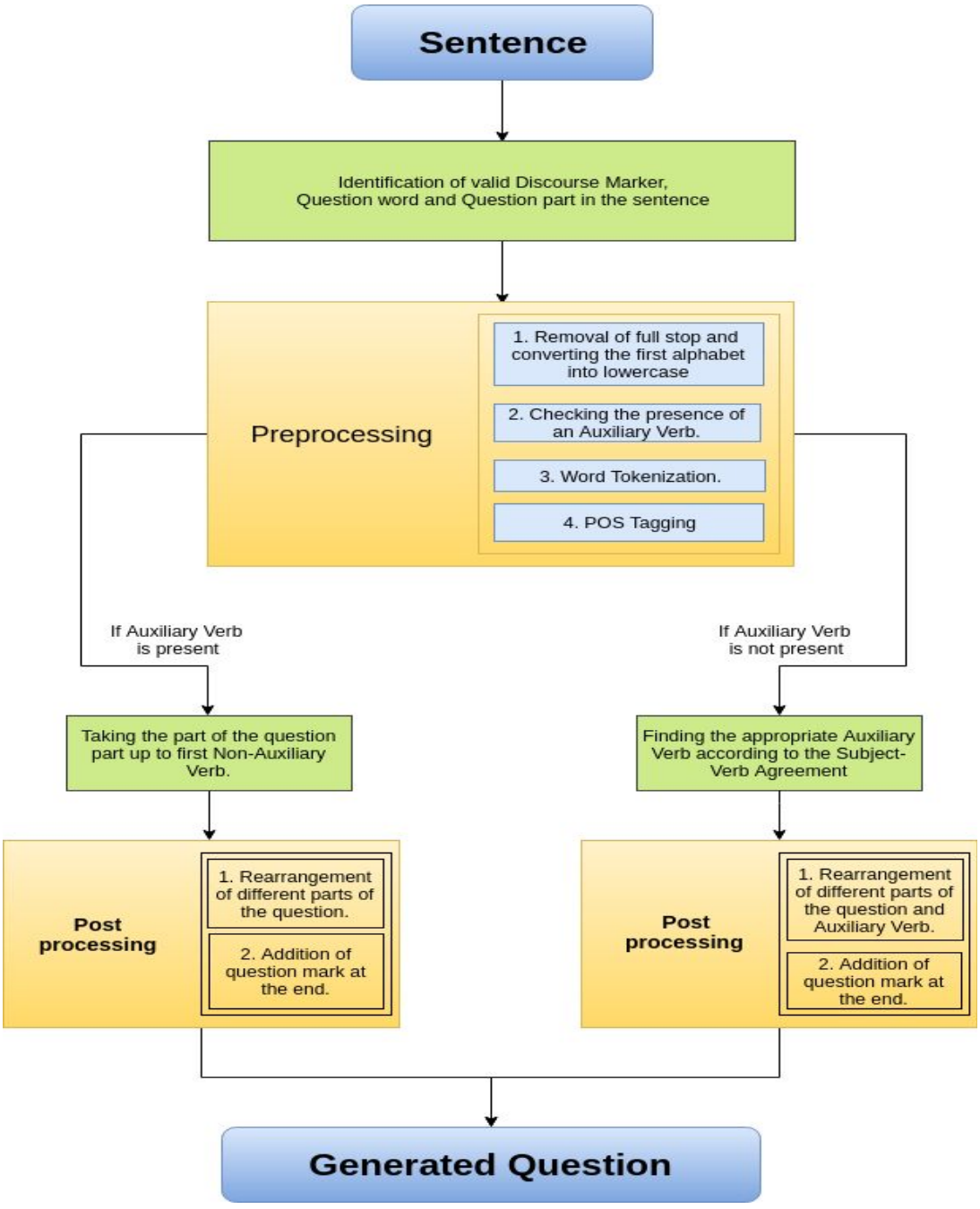


Figure 11: Non Discourse Markers Based Template Algorithm

Example: 'Yes, I learned NLP yesterday.'

```
sentensify()
```

```
Did you learn NLP yesterday?
```

Example: No, I was not playing cricket.

```
sentensify()
```

```
Were you not playing cricket ?
```

3.3. NER (Named Entity Recognition) Based Templates

Sentences that do not contain any discourse marker fall into this category such as simple assertive or declarative sentences or Yes/No sentences. Here we use the POS tagging to find the Nouns and Named Entity Recognition to find the *wh_word* associated with the Named Entity. The following **rule based algorithm** is used for the generation of the questions for the sentences that fall into this template:

- Find all the Named Entities present in the sentence.
- Then for every named entity present in the sentence we identify the *wh_word* associated with that entity.

```
sent = "Ram was held captive at Castle Black."
get_named_entities(sent)
```

```
[('Ram', 'PERSON'), ('Castle Black', 'ORG')]
```

```
def get_wh_word(entity, sent):
    wh_word = ""
    if entity[1] in ['TIME', 'DATE']:
        wh_word = 'When'
    elif entity[1] == ['PRODUCT', 'EVENT', 'WORK_OF_ART', 'LAW', 'LANGUAGE']:
        wh_word = 'What'
    elif entity[1] in ['PERSON']:
            wh_word = 'Who'
    elif entity[1] in ['NORP', 'FAC' ,'ORG', 'GPE', 'LOC']:
        index = sent.find(entity[0])
        if index == 0:
            wh_word = "Who"
        else:
            wh_word = "Where"
    else:
        wh_word = "Where"
    return wh_word
```

```
get_wh_word(get_named_entities(sent)[0],sent)
```

```
'Who'
```

Figure 12: Identification of Wh Word

- Now, we remove full stop from the sentence, if present.
- If Named Entity is present in the beginning of the sentence, we simply replace the Named Entity with its *wh_word* and append the question mark at the end. The procedure is continued for the next Named Entity in the sentence.

```
if(sent[-1]== '.'):
    sent= sent[:-1]

if sent.find(entity[0]) == 0:
    questions.append(sent.replace(entity[0],wh_word) + '?')
    continue
```

Figure 13: Placing Wh word and Question Mark

- If Named Entity is not present in the beginning, we find the Auxiliary Verb present in the sentence.
- Then the Tokenization and POS Tagging of the sentence is done.

```
for i in range(len(aux_list)):
    if(aux_list[i] in sent.split()):
        aux_verb= True
        pos= i
        break

text = nltk.word_tokenize(sent)
tags= nltk.pos_tag(text)
question_part= ""
```

Figure 14: Word Tokenization and POS Tagging

- Then, iterating over the tags we append each word into the question part until we find the first non auxiliary verb.
- We include the next word after the Non-Auxiliary Verb into the question part only if it is not Noun.
- Then the rearrangement of the question part is done to satisfy the Grammatical Syntax of Language.
- Finally, the *wh_word* is prepended and the '?' is appended to the sentence.

```
    for i, grp in enumerate(tags):
        #Break the sentence after the first non-auxiliary verb

        word = grp[0]
        tag = grp[1]

        if(re.match("VB*", tag) and word not in aux_list):
            question_part+= word

            if i<len(tags) and 'NN' not in tags[i+1][1] and wh_word != 'When':
                question_part+= " "+tags[i+1][0]

            break
        question_part+= word+ " "
    print(question_part)
    question= question_part.split(" "+ aux_list[pos])
    question= [aux_list[pos]+ " "]+ question


    question= [wh_word+ " "]+ question + ["?"]

    question= ''.join(question)

    questions.append(question)

return questions
```

Figure 15: NER Algorithm

Example: 'Ram was held captive at Castle Black.'

```
sentensify()
```

```
['Who was held captive at Castle Black?', 'Where was Ram held captive?']
```

Example: 'Ram was playing with toy.'

```
sentensify()
```

```
['Who was playing with toy?']
```

Example:

```
que = ""
for sent in nltk.sent_tokenize(text):
    questions = generate_one_word_questions(sent)
    print(questions)
```

```
['Who was awarded Bharat Ratna in 2013?', 'Who was Sachin Tendulkar awarde
d?', 'When was Sachin Tendulkar awarded Bharat Ratna ?']
```

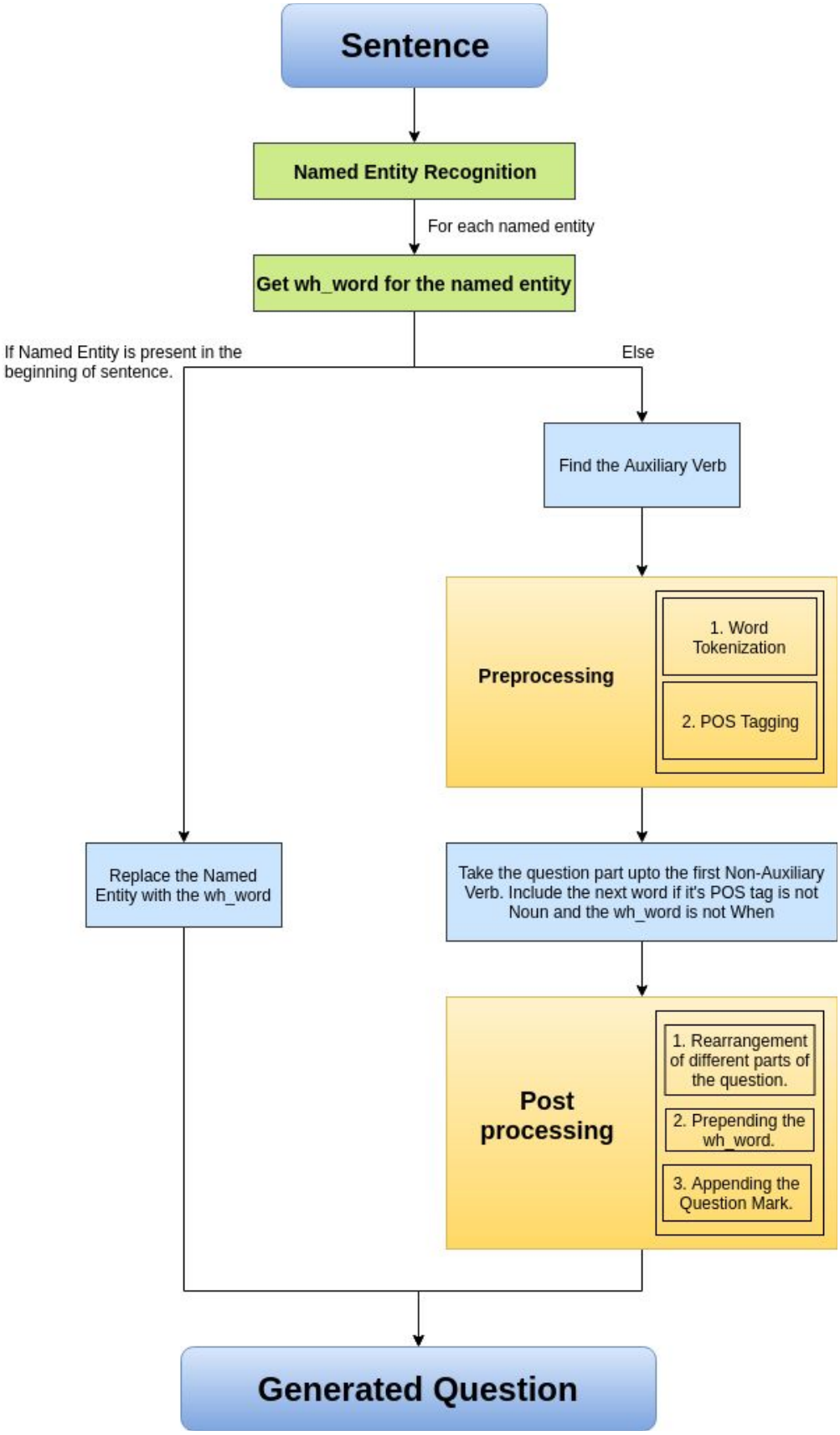


Figure 16: NER Based Template Algorithm

4. EVALUATION

There is no standard way to evaluate the output of a Question Generation system. We go with manual evaluation, where 3 independent human evaluators, all non-native English speakers but proficient in English, give scores to questions generated from the system. To judge syntactic correctness, the evaluators give a score of 1 when the questions are syntactically well-formed and natural and 0 when they are syntactically unacceptable. Similarly, for fluency, the raters give a score of 1 when the questions are fluent and 0 when they aren't. For example, a question like *Who is that girl who works at Google which has its main office in America which is a big country?* is syntactically fine, but isn't as fluent as the question *Who is that girl who works at Google?* which is basically the same question but is more fluent.

Syntactic Score = (No. of Syntactically correct Questions) / (Total No. of Questions)

Fluency Score = (No. of Fluent Questions) / (No. of Syntactically Correct Questions)

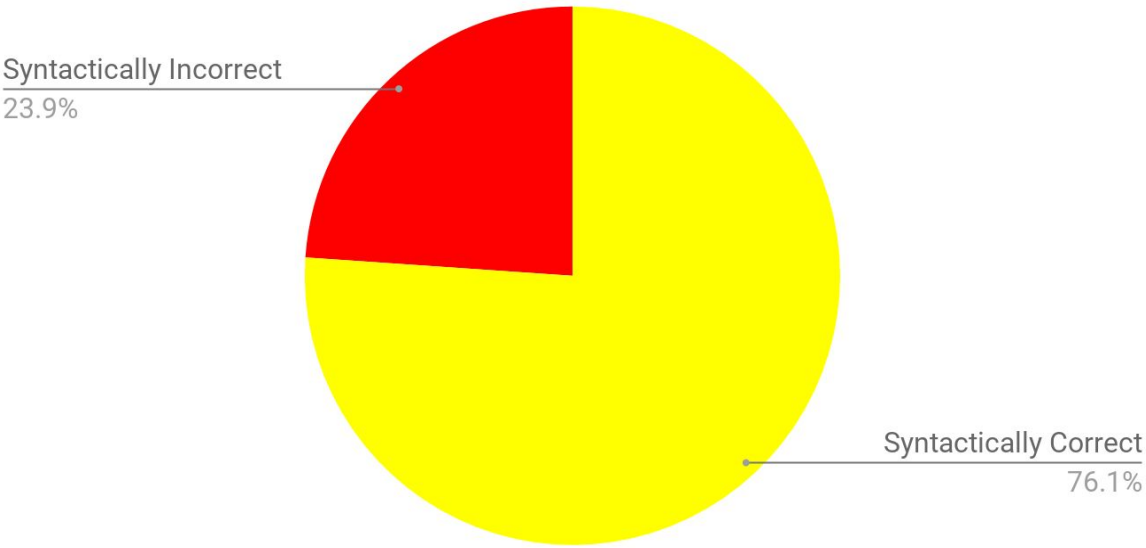
4.1. Results

We take a few number of sentences from different sources such as Wikipedia Corpus, etc. for manual evaluation. We give the questions generated per sentence to human evaluators who rate on the basis of described two criterias. The average ratings of both criterias that our system gets is 0.7608 / 1.0 on syntactic correctness and 0.8428 / 1.0 in fluency.

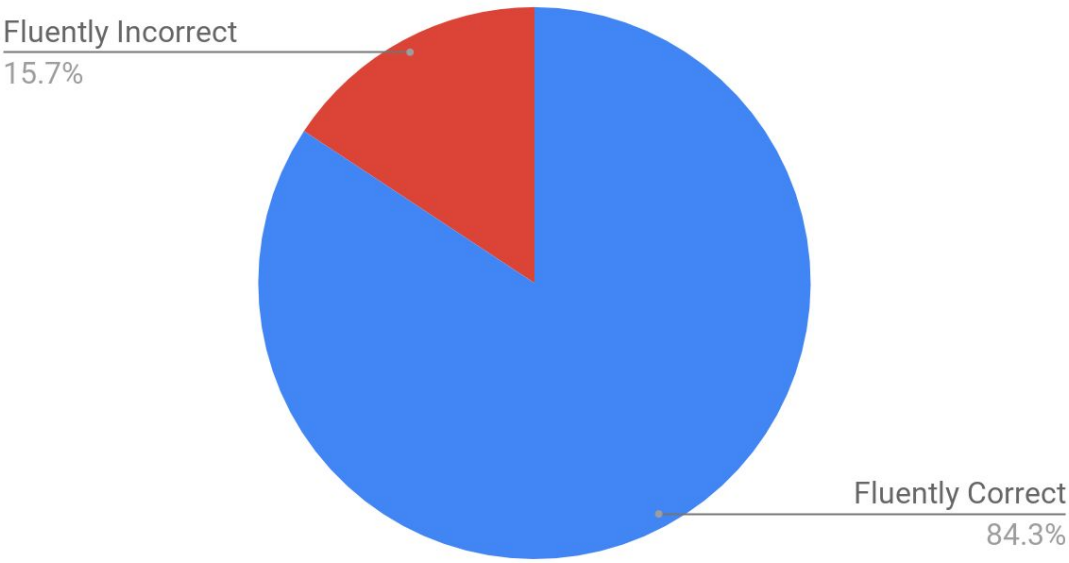
Syntactic Correctness	Fluency
0.7608 / 1.0	0.8428 / 1.0
76.08 %	84.28%

Evaluation Results

Syntactic Correctness



Fluency



4.2. Shortcomings

- Results are not 100% accurate and complete.
- Some of the entities such as things, objects, etc are not recognized by Named Entity Recognition by Spacy library.
- All types of Wh word questions are not being generated by our system.
- Stemmer or Lemmatizer does not always stem the main word correctly. In other words, it sometimes yields a word that is not a real word.
- As the sentence becomes more and more complex, it would be difficult to generate an appropriate question through our system.
- It is a rule based approach so we have to perform manual evaluation for different criterias.

4.3. Future Work

Currently, our templates do not achieve the best performance across all question categories.

- In addition, in order to improve the performance of paragraph based questions in all templates, we need to investigate how to better use the paragraph-level information.
- We can introduce some kind of machine learning algorithms to improve the accuracy and to evaluate the results efficiently.
- Information conveyed from one sentence to other is a problematic issue that needs to be worked upon.
- Implement method for sentences containing possessive and demonstrative pronouns as subject.
- Implement method for questions framed with Which, Whose and How.

REFERENCES

1. https://aclweb.org/anthology/P18-3022

2. http://openaccess.iyte.edu.tr/bitstream/handle/11147/6938/T001801.pdf?sequence=1&isAllowed=y

3. https://lti.cs.cmu.edu/sites/default/files/research/thesis/2011/michael_heilman_automatic_factual_question_generation_for_reading_assessment.pdf

4. https://link.springer.com/chapter/10.1007/978-3-319-06569-4_24