Imperial College London

Individual Project

Department of Computing

Imperial College of Science, Technology and Medicine

Using Answer Set Grammars For Text Summarization

Author:
Julien Amblard

Supervisor:
Alessandra Russo

Helper:
David Tuckey

Second Marker:
Krysia Broda

ASG Author:
Mark Law

Friday 22nd May, 2020

Submitted in partial fulfillment of the requirements for the Computing MEng of Imperial College London

Contents

Chapter 1 Introduction

1 General Problem

In general, the task of summarization in NLP (natural language processing) is to produce a shortened text which covers the main points expressed in a longer text (given as input). In order to do this, a system performing such a task must analyze/process the input to be able to extract from it the most important information.

2 Specific Problem

There are three different overlapping problems we may wish to pursue with the use of answer set programming (ASG):

1. Given a short text of about a page long, for example a short story aimed at young children, to provide a summary in multiple sentences. The goal here would be to identify which sentences in the passage are important, extract only these, and then possibly apply some post-processing to link them a bit better.

2. Given a very short text less than a page long comprised of very brief sentences, for example reading comprehension exercises for Key Stage 1 students, to provide a summary of just one or two sentences in length. What is important here is to establish which *chunks* (i.e. subparts of sentences) are important, thereby semantically learning the main descriptions and actions occurring in the text. From this, we can construct a meaningful *abstract* (see Chapter 2).

3. Given a short text of about a page long (as in the first problem) as well as a summary of a few sentences, to establish whether the latter is a summary of the passage. The goal here would be to understand the semantics of both inputs, use a metric to determine how well they match, and then apply a decision criterion to give a boolean answer.

3 Objectives

TODO

Chapter 2 **Background**

1 Summarization

1.1 Definition

As described by the author in [**?**], a summary is a way of providing a large part of the information contained in one or more original passages, using at most half of the text. Summaries can be grouped into one of the two following categories:

- An *extract* is made up of sentences which are copied word-for-word.
- An *abstract* is a rewriting of the original text's content in a more concise form.

A different way to group summaries is the following:

- *Generic* summaries do not try and focus on anything in particular, they simply aim to recount the most important features.
- *Focused* (or *query-driven*) summaries, on the other hand, require a user-input, which specifies the focus of the summary. For this project, we may want to introduce a bias to our learning program, so that we end up with a summary which meets a certain number of criteria. For instance, we might want it to ensure it captures at least one action verb from the original passage.

Yet another way [**?**] is shown below, with examples given in Figure 2.1:

- *Indicative* summaries give the overall impression of a text, but without conveying any details.
- *Informative* summaries, on the other hand, take the content from an original document and give a shortened version of it.

> It's going to be windy across the western half of the UK, with gusts reaching 60 to 70mph along Irish Sea coastlines, the west of Scotland and perhaps some English Channel coasts. Those in affected areas are advised to take extra care when driving on bridges or high open roads. Flood warnings were issued on Sunday for two areas – Keswick campsite in Cumbria and a stretch along the River Nene east of Peterborough.

(a) *Indicative* summary

> Yellow warnings of strong winds were put in place for parts of the UK. These very strong winds are likely to cause travel disruption, so those in affected areas are advised to take extra care when driving on exposed routes. In addition, heavy rain is expected in parts of the country, which could cause local flooding.

(b) *Informative* summary

Figure 2.1: Example of *indicative* and *informative* summaries for a news article

2 Syntactic Parsing

Given a text, *syntactic parsing* [?] describes the process of tokenization, whereby the grammatical link between parts of speech is established. Two of the most prominent syntactic parsers are **CoreNLP** (developed by Stanford) and **spaCy**.

This tokenization can then be visualized in the form of a tree, which makes it possible to identify different parts of speech, such as nouns, verbs and adjectives. An example is shown in Table 2.1 for both of the mentioned parsers. Appendix ?? shows how to interpret position of speech (POS) tags.

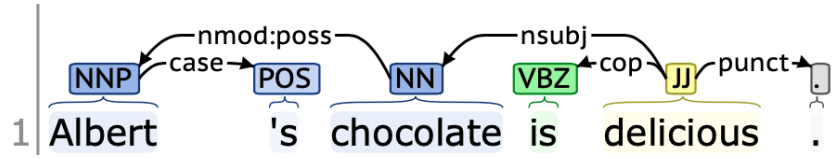
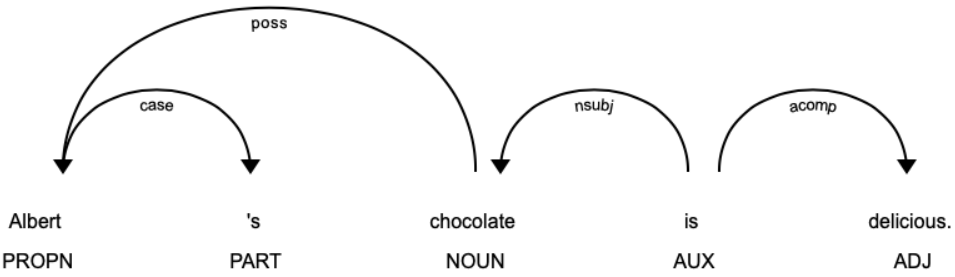
CoreNLP	
spaCy	

Table 2.1: Semantic parsing example for a basic sentence

However, the English language can often be ambiguous and highly context-dependent, meaning that multiple *parse trees* for the same sentence could emerge. Consider the following two sentences [?]:

> He fed her cat food.
> I saw a man on a hill with a telescope.

As shown in Table 2.2, both parsers interpret the meaning of the first example sentence as a person who feeds their "cat food", which, in addition of not being very logical, is also grammatically incorrect as the genders do not match. Unfortunately, computers are generally not very good at context-based inference, something to take into account for this research project.

Therefore accuracy is very important for *syntactic parsing* [?].

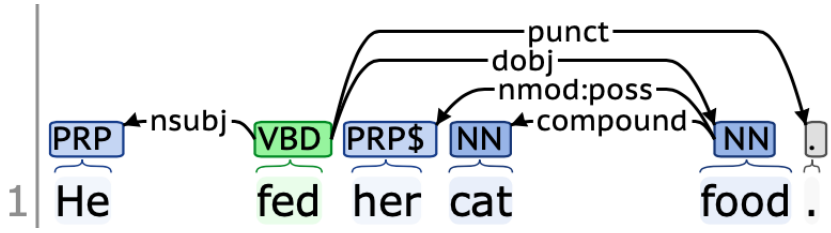
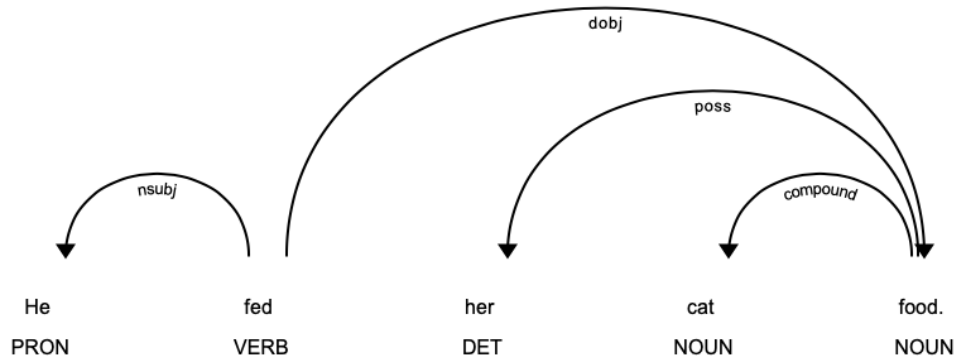
CoreNLP	
spaCy	

Table 2.2: Semantic parsing example for an ambiguous sentence

3 Logic Programming

Definition 1 (Term [?])**.** A *term* is either a *variable* $x, y, z, ...$ or an expression $f(t_1, t_2, ..., t_k)$, where f is a k-ary *function symbol* and the t_i are *terms*. A *constant* is a 0-ary *function symbol*.

Definition 2 (Atom [?])**.** An *atomic formula* (or *atom*) has the form $P(t_1, t_2, ..., t_k)$, where P is a k-ary predicate (boolean function) symbol and the t_i are terms.

3.1 Answer Set Programming

Once we have parsed a text, the next step is to convert this into an answer set program (ASP).

ASP is a declarative first-order (predicate) logic language whose aim is to solve hard search problems [?]. It is built upon the idea of stable model (answer set) semantics, returns answer sets when asked for the solution to a problem.

In ASP, a *literal* is an *atom* a or its negation *not* a (we call this negation as a failure). ASP programs are composed of a set of *normal rules*, whose head is a single *atom* and body is a conjunction of *literals* [?].

$$h \leftarrow b_1, b_2, ..., b_k, \text{not } b_{k+1}, ..., \text{not } b_m. \tag{2.1}$$

If the body is empty ($k = m = 0$) then a rule is called a *fact*. We can also have *constraints*, which are like *normal rules* except that the head is empty. These prevent any answer sets from both including $b_1, b_2, ..., b_k$ and excluding $b_{k+1}, ..., b_m$.

Definition 3 (Safety [?])**.** A variable in a rule is said to be *safe* if it occurs in at least one positive *literal* (i.e. the b_is in the above rule) in the body of the rule.

Definition 4 (Herbrand Base [?]). The *Herbrand base* of a program P, denoted HB_P, is the set of variable-free (*ground*) *atoms* that can be formed from predicates and *constants* in P . The subsets of HB_P are called the (Herbrand) *interpretations* of P.

Definition 5 (Satisfiability [?]). Given a set A, a *ground normal rule* of P is *satisfied* if the head is in A when all positive *atoms* and none of the negated *atoms* of the body are in A, that is when the body is *satisfied*. A *ground constraint* is *satisfied* when the body is not *satisfied*.

Definition 6 (Reduct [?]). Given a program P and an *Herbrand interpretation* $I \subseteq HB_P$, the *reduct* P^I is constructed from the grounding of P in three steps:

1. Remove rules whose bodies contain the negation of an atom in I.
2. Remove all negative *literals* from the remaining rules.
3. Replace the head of any constraint with $\bot$ (where $\bot \notin HB_P$).

For example, the *reduct* of the program $\{a \leftarrow \text{not } b, c. \quad d \leftarrow \text{not } c.\}$ with respect to $I = \{b\}$ is $\{d.\}$.

Definition 7 (Minimal Model). We say that I is a (Herbrand) *model* when I *satisfies* all the rules in the program P. It is a *minimal model* if there exists no smaller *model* than I.

Definition 8 (Answer Set [?]). Any $I \subseteq HB_P$ is an *answer set* of P if it is equal to the *minimal model* of the *reduct* P^I. We will denote the set of *answer sets* of a program P with $AS(P)$.

3.2 Context-Free Grammars

In order to discuss answer set grammars (ASGs), we must first define *context-free grammars* (CFGs) and *parse trees*. An example for these is shown in Figure 2.2.

Definition 9 (Context-Free Grammar [?]). A CFG is a finite set G of "rewriting rules" $\alpha \to \beta$, where α is a single symbol and β is a finite string of symbols from a finite alphabet (vocabulary) V. V contains precisely the symbols appearing in these rules plus the "boundary" symbol ϵ, which does not appear in these rules. Rules of the form $\alpha \to \alpha$ (which have no effect) are not allowed.

Definition 10 (Parse Tree [?]). Let GCF be a CFG. A *parse tree* PT of GCF for a given string consists of a node $node(PT)$, a list of *parse trees*, called *children* and denoted $children(PT)$, and a rule $rule(PT)$, such that:

1. If $node(PT)$ is a terminal node, then $children(PT)$ is empty.
2. If $node(PT)$ is non-terminal, then $rule(PT)$ is of the form $node(PT) \to n_1 ... n_k$ where each n_i is equal to $node(children(PT)[i])$ and $|children(PT)| = k$.

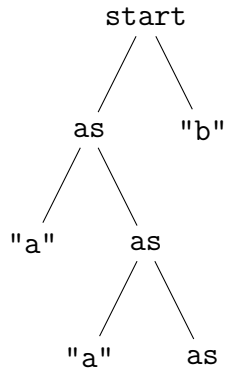
Definition 11 (Trace [?]). We can represent each node n in a *parse tree* by its *trace*, $trace(n)$, through the tree. The *trace* of the root is the empty list `[]`; the i$^{\text{th}}$ child of the root is `[i]`; the j$^{\text{th}}$ child of the i$^{\text{th}}$ child of the root is `[i, j]`, and so on.

```
1:  start -> as "b"
2:  as -> "a" as
3:  as ->
```

(a) CFG for $\texttt{a}^\texttt{i}\texttt{b}$

trace(*n*)	*node*(*n*)
[]	start
[1]	as
[1,1]	a
[1,2]	as
[1,2,1]	a
[1,2,2]	as
[2]	b

(b) *Parse tree* table for aab



(c) *Parse tree* graph for aab

Figure 2.2: Example of a CFG and its *parse tree*

3.3 Answer Set Grammars

ASGs are an extension of CFGs, whereby each production rule is *annotated*. More specifically, P can be a *ground term*, such as in the *annotated atom* `a(1)@2` (referring to the second child of this node). The example shown in Figure 2.3 is a subset of the language $\texttt{a}^\texttt{i}\texttt{b}$ captured by the CFG in Figure 2.2, restricting it to the language $\texttt{a}^\texttt{n}$ where $\texttt{n} \geq 2$ (the string contains at least two as).

Definition 12 (Answer Set Grammar [**?**])**.** An *annotated* production rule is of the form $n_0 \rightarrow n_1 ... n_k \ P$ where $n_0 \rightarrow n_1 ... n_k$ is an ordinary CFG production rule and P is an *annotated* ASP program, where every *annotation* is an integer from 1 to k.

```
1:  start -> as "b" { :- size(X)@1, X < 1.  }
2:  as -> "a" as    { size(X+1) :- size(X)@2.  }
3:  as ->           { size(0).  }
```

* Intuitively, `size` represents the length of the current string.

Figure 2.3: Example of an ASG

Definition 13 (Parse Tree Program [**?**])**.** Let G be an ASG and PT be a *parse tree*. $G[PT]$ is the program $\{ \ rule(n)@trace(n) \mid n \in PT \ \}$, where for any production rule $n_0 \rightarrow n_1 ... n_k \ P$, and any trace t, $PR@t$ is the program constructed by replacing all annotated atoms $a@i$ with the atom $a@t++[i]$ and all *unannotated atoms* a with the atom $a@t$.

Definition 14 (Conforming Parse Tree [**?**])**.** Given a string *str* of terminal nodes, we say that $str \in \mathcal{L}(G)$ (*str conforms* to the language of G) if and only if there exists a parse tree PT of G for *str* such that the program $G[PT]$ is *satisfiable*. For such a PT, every single rule in the language must be satisfied (see Definition 5).

As shown in Figure 2.4, $\texttt{aab} \in \mathcal{L}(G)$, and the corresponding program has a single answer set $\{size(0)@[1,2,2], \ size(1)@[1,2], \ size(2)@[1]\}$. From this example, it is easy to see how the corresponding program would be *unsatisfiable* for the string ab.

```
:- size(X)@[1], X < 1.
size(X+1)@[1] :- size(X)@[1,2].
size(X+1)@[1,2] :- size(X)@[1,2,2].
size(0)@[1,2,2].
```

Figure 2.4: $G[PT]$ for the *parse tree* and ASG from the examples above

3.4 Learning Answer Set Grammars

Given an incomplete ASG, it is possible to learn the complete grammar by induction (which uses ILASP), as long as we provide some *positive examples* (strings which should conform to the language) and/or *negative examples* (strings which must not), as well as a *hypothesis space* and usually some *background* information. Note that the *background* is only used for "global" knowledge, such as defining what is a number, or how to increment one. [**?**]

In such an *inductive learning program* (ILP) task, we have a *hypothesis space* in the form of *mode declarations*, defining the format of the heads (written `#modeh`) and bodies (written `#modeb`) of rules which can be learned. It is also possible to restrict the scope of a particular *mode declaration* by specifying a list of rule numbers at the end. Note that there are two forms of body *mode declarations*: `#modeba` is used for predicates that accept an @ *annotation*, and `#modebb` is intended for those without (which are defined in `#background`). An example is shown in Figure 2.5.

```
start -> as bs {}
as -> "a" as {} | {}
bs -> "b" bs {} | {}

+ []
+ ["a", "b"]
+ ["a", "a", "b", "b"]
- ["a"]
- ["b"]
- ["a", "a"]
- ["b", "b"]
- ["a", "a", "b"]
- ["a", "b", "b"]

#background {
num(0). num(1). num(2). num(3).
inc(X,X+1) :- num(X), num(X+1). }

#modeh(size(var(num))):[2,3,4,5].
#modeh(size(0)):[2,3,4,5].
#modeba(size(var(num))).
```

(a) Input incomplete program

```
start -> as bs {
 :- not size(X)@2, size(X)@1.
}

as -> "a" as {
 size(X+1) :- size(X)@2.
}

as -> {
 size(0).
}

bs -> "b" bs {
 size(X+1) :- size(X)@2.
}

bs -> {
 size(0).
}
```

(b) Output learned program

* Note: the symbol `|` indicates multiplicity of production rules.

Figure 2.5: Example of an ASG ILP task for the language $a^n b^n$

4 Neural Networks

4.1 Encoder-Decoder

TODO

Chapter 3 Contributions

1 Architecture Overview

The central part of the pipeline, which performs story summarization, revolves around the ASG task of SumASG. To build on top of this foundation, we first have the Preprocessor, as well as some final post-processing and scoring mechanisms. We will call this augmented task SumASG*, and describe each step in the following chapters. A diagram of the entire pipeline can be seen in Figure 3.1.

Figure 3.1: Main Pipeline

2 Initial Motivation

Given plenty of time and a large amount of training data in a specific format, a neural network is able to closely approximate the definition of a summary.

In contrast, using logic means that we can hard-code this definition directly into our program. By carefully constructing its structure, we can get results with just a short list of rules. In addition, by strictly adhering to these rules, we know that our logic program will always produce a complete and valid output, as long as the program is correct.

3 Example

Throughout this paper, we shall be using the example of the story of Peter Little to illustrate the different steps of our pipeline, as shown below in Figure 3.2.

> There was a curious little boy named Peter Little. He was interested in stars and planets. So he was serious in school and always did his homework. When he was older he studied physics and maths. He studied hard for his exams and became an astrophysicist. Now he is famous.

(a) Story of Peter Little

> Peter Little was interested in space so he studied hard and became a famous astrophysicist. Peter Little was curious in stars and planets. He was serious in school. Now he is famous. Peter Little was curious in astronomy. He was serious in school. Now he is famous.

(b) Possible summaries

Figure 3.2: Example of the task of summarization for the story of Peter Little

Chapter 4 Preprocessor

1 Overview

In order to prepare the story to be parsed and summarized by SumASG, we have created the Preprocessor, which steps are shown below in Figure 4.1.

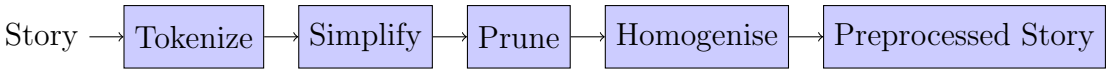


Figure 4.1: Preprocessor Steps

2 Tokenization and Simplification

With the help of **CoreNLP**, we can assign a POS tag to each word, or *token*, from the input story. Using this information, we can now make a number of simplifications which will make the sentence structure more consistent.

3 Sentence Pruning and Homogenisation

4 Example

TODO

5 Expandability

TODO

Choice: 1. Simplify using Python script (faster) 2. Make ASG format more complex (new information probably lost in summary anyway)

- Punctuation - [?]: remove clause (helps avoid negation for rhetorical questions) - [!—,—;—:]: transform into '.' - [———]: delete inner part - Multiple clauses (conjunctive or main+auxiliary): split into multiple sentences - Adverbs: move to end - Possessive pronouns and interjections: remove - Prepositions: remove when at start of sentence - Contractions: expand - Acronyms: remove punctuation - Dependant clauses: split into separate sentence (remove if there is no punctuation) - Verbless sentences: remove - Subordinating conjunctions: split into separate sentence - Preposition clauses: remove if after object - Complex proper nouns: collapse into CamelCase - Proper nouns: replace occurrences of pronouns with relevant proper noun (idea: if they are used in the story then there should be little ambiguity) - Conjunction of common nouns from same lexical field: replace with hypernym (pluralize if items are plural and hypernym plural is used in English)

Chapter 5 **ASG**

1 Overview

Our use ASG is two-fold. Firstly, we pass in each sentence from the story to ASG to obtain its semantic representation in ASP. Secondly, we take these actions and use ASG rules to generate possible summary components. These will later be post-processed and turned into actual valid summaries. A diagram of the two ASG steps is shown below in Figure 5.1.

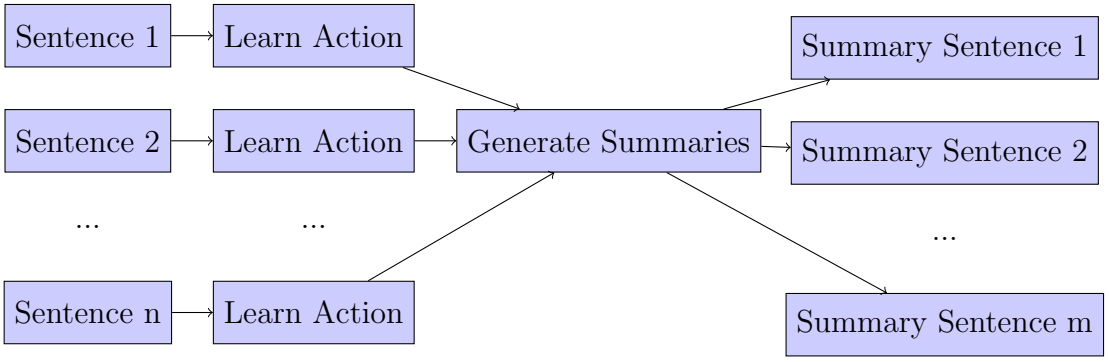


Figure 5.1: ASG Steps

2 Learning Actions

3 Generating Summary Sentences

4 Expandability

TODO

- Learning is not really learning (ASG never learns how to summarize, we build in rules of feature extraction) - Describe action predicates as high level semantic descriptor of all possible actions that can happen in sentences

- Maybe formalize mathematically task of summarization (with CFG, BK, E+, E-) 1. CFG is language, BK is leaf nodes, result is actions 2. CFG is language, BK is leaf nodes, E is actions, result is summaries - Appendix with summary generation rules

http://universalteacher.org.uk/lang/engstruct.htm

Ideas: - final fix-up using language_checker.fix - hard-code determiners into derivations (https://www.ef.com/wwen/english-resources/english-grammar/determiners/) - use lots of simple/precise rules rather than complicated/general ones to minimize ss - keep rules as restricted as possible, when concept implemented over time add missing rules - to avoid having to add grammar constraints try and rely on grammar of input

- reduce search space using mode bias (simple example: 396-¿16, very complicated example: 9477-¿1044) - for learning actions do one sentence at a time to minimize ss - pick best summary according to TTR*

action(INDEX, VERB, SUBJECT, OBJECT) summary(VERB, SUBJECT, OBJECT)

verb(INDICATIVE_FORM, TENSE) subject(NOUN, DET, ADJ_OR_ADV) object(NOUN, DET, ADJ_OR_ADV) noun(NAME) adj_or_adv(NAME) det(...) compound(FIRST, SECOND) for verbs conjunct(FIRST, SECOND) learn both

* Type-Token Ratio (TTR): The basic idea behind that measure is that if the text is more complex, the author uses a more varied vocabulary so there's a larger number of types (unique words). This logic is explicit in the TTR's formula, which calculates the number of types divided by the number of tokens. As a result, the higher the TTR, the higher the lexical complexity.

Chapter 6 Post-Processing / Scoring

1 Overview

Once we have obtained potential sentences from ASG to be used in a summary, we can now post-process these as explained in Section 2. By combining them in different ways, we are able to form summaries. From these, we will retain the highest scoring ones, according to the metric detailed in Section 3. A diagram illustrating these steps is shown below in Figure 6.1.

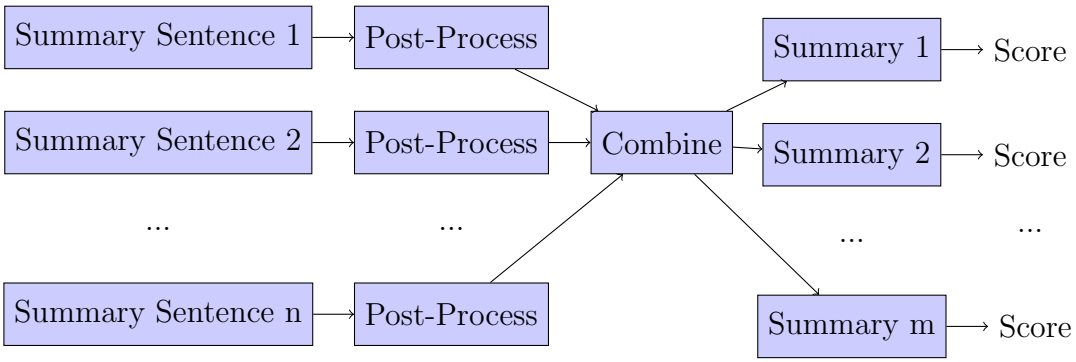


Figure 6.1: Post-Processing / Scoring Steps

2 Summary Creation

3 Scoring

4 Expandability

TODO

- Fix grammar - Final goal: take story-specific ASG and general rules to generate summaries, then use top 5/10

Chapter 7 **Evaluation**

1 General Idea

2 Dataset

3 Results

For for each story: 1. Pick predefined lexical field (topic) 2. Pick a single pronoun (p) 3. Pick a single proper noun (pn) 4. For each sentence: - Subject: p, pn, or synonym/hyponym/hypernym of topic with optional common adjective for it - Verb: verb from same lexical field as topic if possible, otherwise random - Object: p, pn, or holonym/meronym of subject with lexical field of currently used common nouns

- Compare with NN 1. Randomize action(...) to generate summary(...) on trained ASG 2. Train NN to generate same summary(...) 3. Show framework is sane and expandable (computationally tractable) 4. Compute Rouge score (PyRouge, must clone repo into project) on ASG and NN

Chapter 8 **Literature Review**

1 Summarization Levels

Depending on how much text analysis is done, we identify three different levels of summarization [**?**]. Many current systems employ what is called a *hybrid approach*, combining techniques from different levels.

1.1 Surface Level

On a *surface level*, little analysis is performed, and we rely on keywords in the text which are later combined to generate a summary. Techniques which are common include:

- *Thematic features* are identified by looking at the words that appear the most often. Usually, the important sentences in a passage have a higher probability of containing these *thematic features*.
- Often, the *location* of a sentence can help identify its importance; the first and last sentences are generally a good indicator for the respective introduction and conclusion of a document. Moreover, we may want to make use of the title and heading (if any) to find out which topics are most relevant.
- *Cue words* are expressions like "in this article" and "to sum up"; these can give us a clue as to where the relevant information is.

1.2 Entity Level

A more analytic approach can be done at an *entity level*, where we build a model of a document's individual entities and see how they relate. Common techniques include:

- *Similarity* between different words (or phrases), whether it be synonyms or terms relating to the same topic.
- *Logical relations* involve the use of a connector such as "before" or "therefore", and tell us how the information given by such connected phrases relates.

1.3 Discourse Level

Finally at a *discourse level* we go beyond the contents of a text, exploiting its structure instead. Some of the things we can analyze are:

- The *format* can be taken into account to help us extract key information. For example, in a rich-text document we may want to pay close attention to terms that are underlined or italicized.
- The *rhetorical structure* can tell us whether the document is argumentative or narrative in nature. In the latter case a more concise description of the text's contents would suffice, while the former would involve recounting the key points and conclusions made by the author.

2 Semantic Analysis Methods

2.1 Combinatory Categorial Grammar

In a paper from 2019 [?], the author introduces Combinatory Categorial Grammar (CCG), an efficient parsing mechanism to get to the underlying semantics of a text in any natural language. It is combinatory in the sense that it uses functional expressions such as $\lambda p.p$ in order to express the semantics of words.

In CCG, every word, written in English in its *phonological form*, is assigned a *category*. Furthermore, a *category* is comprised of the word's *syntactic type* and *logical form*. As shown in Figure 8.1, the former gives all the conditions necessary for a word to be combined with another, and the latter shows in a simpler form its representation in logic. The *phonological form* comes from the input text, the *syntactic type* is used in the process of conducting semantic analysis, and finally the *logical form* is the result of parsing a passage.

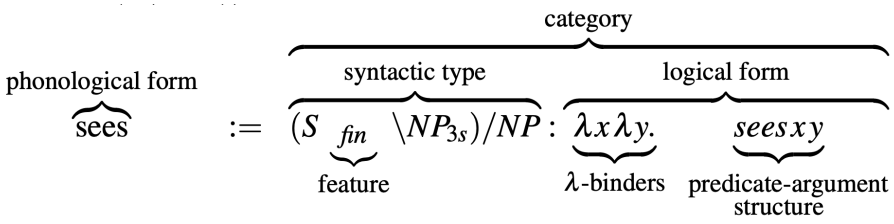
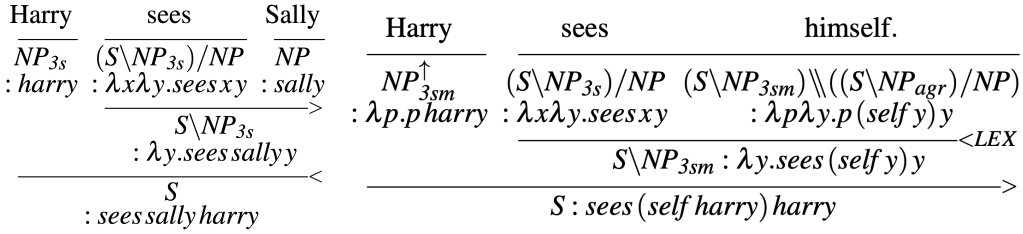


Figure 8.1: [?] Diagram explaining the mains terms used in CCG

In the *syntactic type* of a word, the forward slash / indicates forward application to combine terms, while \ indicates backward combination. If there is no slash, then the expression can be thought of as a clause, and it can combine with any rule.

- $X/Y : f \quad Y : a \implies X : fa \quad (>)$
- $Y : a \quad X\backslash Y : f \implies X : fa \quad (<)$

There also exists a *morphological slash* \\, which restricts application to lexical verbs, ruling out auxiliary verbs (whose role is purely grammatically, hence they do not play any part in providing information). The *morphological slash* can be used when dealing with reflexive pronouns such as "themselves". Furthermore, combining rules directly correlates to obtaining a simpler *logical form* with fewer bound variables, as can be seen in Figure 8.2.



(a) [?] Basic clause

(b) [?] Reflexive transitive clause

Figure 8.2: Examples of derivations in CCG

There exist more advanced syntactic rules in CCG, which we shall not go into detail about for the purposes of brevity. However with the basic rules that we explained, you can easily see how this parsing mechanism could be an efficient way to get to the underlying semantics of a sentence. Although the *syntactic type* may seem complicated, it would allows us to get a very precise understanding of English grammar, as well as obtain a simple and consistent *logical form* at the end.

3 Existing Approaches

3.1 MCBA+GA And LSA+TRM

In a paper by Yeh et al. [**?**], two different methods are put forward for text summarization. The first is the modified corpus-based approach (MCBA), which uses a score function as well as the *genetic algorithm*, while the second (LSA+TRM) utilizes *latent semantic analysis* (LSA) with the aid of a *text relationship map* (TSA).

In order to understand MCBA, we must first mention corpus-based approaches, which rely on machine learning applied to a corpus of texts and their (known) summaries. In the *training phase* important features (such as sentence length, position of a sentence in a paragraph, uppercase word...) are extracted from the *training corpus* and used to generate rules. In the *test phase* the learned rules are applied on the *training corpus* to generate corresponding summaries. Most approaches rely on computing a weight for each unit of text, this is based on a combination of a unit's features.

The MCBA builds on the basic corpus-based approach (CBA) by ranking sentence positions and using the genetic algorithm (GA) to train the score function. In the first case, the idea is that the important sentences of a paragraph are likely to have the same position in different texts, such as the first sentence (introduction) and the last one (summary). Depending on a sentence's position, a *rank* (from 1 to some R) is assigned, and used to compute a score for this feature. The paper also discusses other features, whose corresponding scores, along with the aforementioned *rank*, are used to compute a weighted sum of all scores. Only the highest scoring sentences are retained in order to form the summary.

Moreover, the *genetic algorithm* (GA) is used to obtain suitable weights, where a *chromosome* is defined by a set of values for all the features weights. Using the notions of *precision* (proportion of predicted positive cases that are correctly real positives) and *recall* (proportion of real positive cases that are correctly predicted positive) [**?**], a so-called *F-score* is computed to define the fitness for each chromosome. By combing two *chromosomes* to generate children, where the fittest parents are most likely to mate, we end up (after some number of generations) with a set of feature weights suitable for the corpus in question.

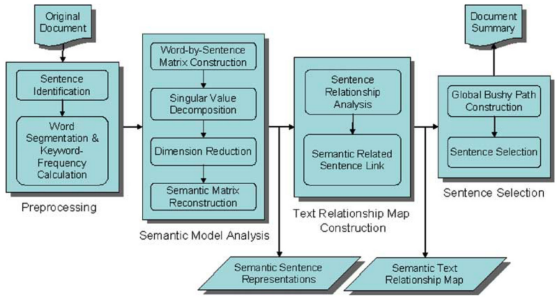
On the other hand, the LSA+TRM approach comprises four major steps: *preprocessing* (1), *semantic model analysis* (2), *text relationship map construction* (3) and *sentence selection* (4).

In step (1), sentences are decomposed according to their punctuation, as well as divided into keywords.

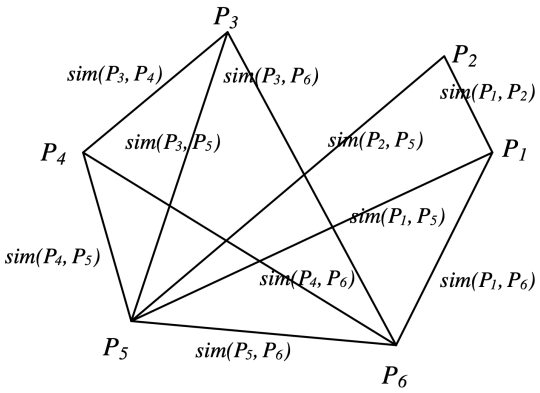
In step (2), a *word-by-sentence matrix* is computed on the scale of the entire document (or corpus). This gets factorized and reduced to leave out words which do not occur often, then turned into a *semantic matrix* linking words to their according relevance with each sentence.

In step (3), the *semantic matrix* is converted to a *text relationship map*. A *text relationship map* is a graph comprised of nodes, each one represents a sentence or paragraph. A link exists between any two which have high semantic similarity, and the idea is that nodes with many links are likely to cover the main topics of the text.

Finally, step (4) uses the *text relationship map* to pick out the most important sentences for the summary. Figure 8.3 may help you visualize how this works.



(a) [?] Overall process



(b) [?] Example of a text relationship map

Figure 8.3: LSA+TRM approach, diagrammatically

Compression rate (CR) is a proportion describing the size of the summary with respect to the size of the original text.

After evaluating both approaches on a news article corpus, it was found that MCBA outperforms the basic CBA by around 3%, confirming the hypothesis that the position of a sentence plays a role in its importance. Furthermore, MCBA+GA performs around 10% better than MCBA.

Concerning LSA+TRM, it was found that on a per-document level this approach outperformed simply using TRM with the sentence keywords rather than LSA by almost 30%. It was thus concluded that LSA helps get a better semantic understanding of a text.

Comparing the two approaches highlighted in the paper, it is mentioned that performance is similar, although LSA+TRM is easier to implement than MCBA in single-document level as it requires no preprocessing, and in some optimal cases performs up to 20% better. Although the former approach is more computationally expensive, it is more adept at understanding the semantics of a text because it does not rely on the genre of the corpus that was used for training. In both cases though, performance improves as CR increases.

As our solution will rely on ASG, no machine learning will be needed. However, the first approach is still interesting in the sense that it uses a certain number of important features to identify the important sentences of a passage. In our approach, we may want to use some of these metrics to construct the summary.

From the second approach, the main takeaways are the storage mechanisms in use such as the *semantic matrix* and *text relationship map*. In our system we may also want to use the idea that sentences or *chunks* which are semantically similar to many others in the *text relationship map* are likely to cover the main topics of a passage.

Finally, we notice that the approach based on machine learning (MCBA) gives summaries of inferior quality in general, confirming that the use of ASG is a good choice. In addition, it was found that the longer the summary (higher CR), the more accurate it is, so we must be particularly careful when generating one to two sentence summaries.

3.2 Lexical Chains

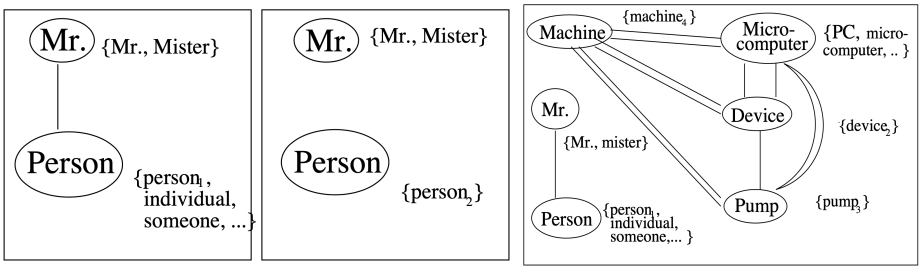
In a paper about *lexical chains* [**?**], the authors describe a method which relies on semantic links between words. The idea is that we establish chains of related words, in order to learn what a text is about.

In order to create such a chain, the algorithm begins by choosing a set of *candidate words* for the chain. These *candidate words* are either nouns, or *noun compounds* (such as "analog clock"). Starting from the first word, the task is to find the next related word which has a similar meaning (a dictionary is used here). If the word has multiple senses, then the chain gets split into multiple interpretations; this process continues until we have analysed all *candidate words*. For instance, the word "person" can be interpreted as meaning a human being (interpretation 1), or as a grammatical term used for pronouns (interpretation 2). An example for the below text is shown in Figure 8.4.

> **Mr.** Kenny is the **person** that invented an anesthetic **machine** which uses **micro-computers** to control the rate at which an anesthetic is pumped into the blood. Such **machines** are nothing new. But his **device** uses two **micro-computers** to achieve much closer monitoring of the **pump** feeding the anesthetic into the patient. [**?**]

Furthermore, *lexical chains* are attributed a *strength*, which is based on three criteria: repetition (of the same word), density (the concentration of chain members in a given portion of the text) and length (of the chain). For instance, the *lexical chain* beginning with the word "machine" shown in Figure 8.4 (interpretation 1) has considerable repetition, moderate density, and is quite long (it spans almost the entire text).

Based on this indicator, interpretations of a *lexical chain* with higher *strength* will be preferred. (In reality this is a bit more complex, but we will omit the details for simplicity.)



(a) Step 1, interpretations 1 and 2 (b) Step 3, interpretation 1

Figure 8.4: [?] Example of a *lexical chain* and its interpretations

In order to construct the summary, a single "important" sentence is extracted from the original text. To this end, they use a heuristic which is based on the fact that an important topic will discussed across the entire passage. Once a *lexical chain* has been chosen according to this metric (i.e., one that is well distributed across the text), the output of the algorithm is the sentence which has the highest number of words from the selected *lexical chain*.

Although the proposed solution is very interesting in that it tries to link important words, it does not do anything whatsoever to learn any of the actions which are described in a passage. This means it has no knowledge of chronology (problematic when we have an action causing a change of state, such as a someone acquiring a good), nor does it try to link subjects with objects or verbs (for instance in the phrase "Mary has a pencil", it does not link Mary to the pencil).

Furthermore, the algorithm outputs a singe unchanged sentence from the original text; this is suboptimal when equally important information is conveyed across multiple sentences. In a solution to the problem of text summarization, we would hope that important facts or actions are given as a summary. For the approach discussed here, this would mean picking out *chunks* of text from the original passage, and combining them in a suitable manner.

4 Approach Categories

4.1 Statistical

In the statistical approach, the methodology is to use probabilities in order to generate a summary that is both grammatically correct and conveys the important details of a text.

The authors of the paper [?] envision what they call a *noisy-channel model*, which at the time of writing was limited to single sentence summarization. For the model, assume that there was at some point a (shorter) summary string s for the (longer) string t to summarize, from which optional words were removed. The idea is that optional details were added to produce t, and we want to know with what probability s contained this information given t. At this stage, there a three problems to solve:

1. To obtain the *source model*, we must assign a probability $P(s)$ to every string s, which tells us how likely it is that this is the summary. If we assign a lower

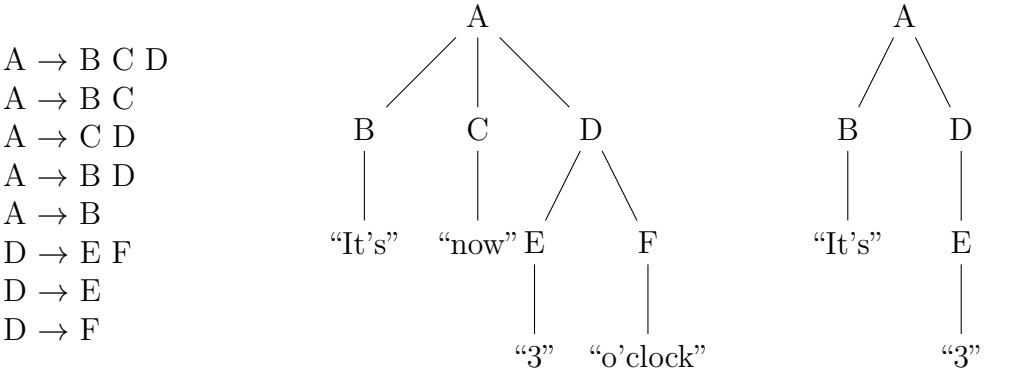
$P(s)$ to less grammatically correct strings, then it helps ensure that our final summary is well-formed.

2. To obtain the *channel model*, we now assign the probabilities $P(t|s)$ to every pair $\langle s, t\rangle$. This contributes to preserving important information, as we take into account the differences between s and t when computing the corresponding probability. In this case, we may want to assign a very low $P(t|s)$ when s omits an important verb or negation (these are not optional to get the correct meaning), while this can sometimes be much higher if the only difference is the word "that".
3. For a given string t the goal is now to maximize $P(s|t)$ which, because of Bayes' theorem, is equivalent to maximizing $P(s) \cdot P(t|s)$.

In practice, the implementation discussed in the paper uses *parse trees* rather than whole strings. Also, they use machine learning techniques in order to train their summarizer.

To this end, they created what was referred to in the paper as a *shared-forest* structure, allowing them to represent all compressions given an original text t; an example is shown in Figure 8.5. Their system picks out high-scoring trees from the forest, and based on this score we can choose the best compression s for t (i.e., the summary s which has the highest $P(s) \cdot P(t|s)$).

If the user wants a shorter or longer summary, the system can simply return the highest-scoring tree for a given length. In reality though their solution is a bit more complex, but the important points of the approach are here.



(a) *Shared-forest* structure (b) *Parse tree* for t (c) *Parse tree* for s

Figure 8.5: Example of an original text t and possible summarization s

In their testing it was found that their algorithm has a conservative nature, promoting correctness over brevity, which sometimes has the consequence of not trimming any words away. Therefore, this implementation is highly unsuitable for our purposes, but we shall nonetheless keep in mind the notion of the *shared-forest* structure, as well as the use of Bayesian probability theory.

4.2 Frame

In the frame approach, the idea is that we try and keep track of how the plot in a story progresses, recording each action as well as the links which connect them. From

this understanding, we should then have enough information to build an accurate summary.

In one of the original papers describing this approach [?], sentences are decomposed into different *affect states* and *affect links*. An *affect state* can either be a *mental state*, or a *positive* or *negative event* which may cause a change to a *mental state*. *Affect links* are then the transitions that explain the sequence of *affect states*.

We are given the example of John and Mary who both want to buy the same house, but it ends up being sold to Mary. At the start, both characters have the same *mental state* (desire to buy the house). However the *actualization* (a type of *affect link* which denotes realization of an action) of Mary's desire is recognized as a *positive event* for Mary and a *negative event* for John.

By combining sequences of *affect links* (transitions) between *affect states* for different characters in a story, it is easy to see how one can build the narrative of the entire text. Such an example is shown in Figure 8.6, where m denotes the *motivation affect link* (connecting an action with a *mental state* which it motivates), and e denotes the *equivalence affect link* (i.e., when a character has the same *mental state* before and after the transition).

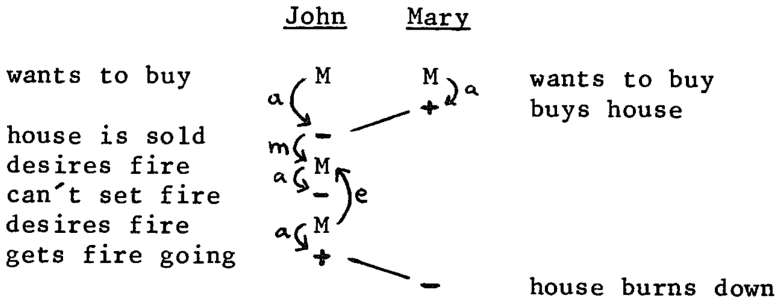


Figure 8.6: Example of a narrative in the frame approach

While this approach is interesting from a semantic point of view, it can easily become very complicated when many sentences are involved. In addition, it would not be suitable for purely descriptive texts, involving only continuous actions without any direct link ("It was October. Leaves were falling from the trees.").

4.3 Symbolic

In the symbolic approach [?], meaning is expressed through logic, and the representation of a sentence is the combination of the meaning of each of its individual components. CCG, as discussed in Subsection 2.1, uses a symbolic approach.

Generally the semantics of a word is captured as a single predicate in logic, and sometimes this is automatically derived from a large dataset such as an online dictionary. In order to obtain the final (sentential) logical form however, parsed sentences (represented for example as a *parse tree*) must first be translated from their original (natural language) syntax. It then becomes possible to combine the meanings of words to obtain sentence fragments, and then combine these to understand the whole sentence. These can be further composed to cover an entire passage. This representation can then be provided to a theorem prover in *first-order logic* (FOL), or directly converted into a logic program (for instance in ASP).

Besides CCG, another pertinent case study is that of the Montague Grammar [**?**]. In such a grammar, we have what is called a *syntactic language* and a *semantic language*. The former is similar to POS tags (see Appendix **??**), while the latter captures the type of a token (which can either be e for *entity* or t for *truth value*). For instance, the word "John" has *syntactic category* ProperN and *semantic type* e, while for the verb "walks" these are respectively VP and $e \to t$.

For both of these languages, there exist rules that dictate how we are allowed to compose tokens. In the *syntactic* case, this is simply a restriction on the format of *parse trees* (i.e., $S \to NP\ VP$ means that a sentence node must have an NP and a VP as its children to be grammatically correct). For the *semantic language*, combining tokens with respective types $A \to B$ and A results in one whose type is B. Taking the example from before, "John walks" would have type t; this makes sense because "John" is an entity, and whether he is walking can either be true or false.

As we have seen above, these rules allow us to compose tokens together, and in the Montague Grammar everything has a logical representation (expressed in the λ-calculus). For instance, we would compose $\lambda P.[P(john)]$ with $walk$ to obtain $\lambda P.[P(john)](walk) \equiv walk(john)$. A more advanced example would be that "every student walks" is represented as $\forall x.(student(x) \to walk(x))$.

One of the criticisms with this approach [**?**] is that it is domain-specific and not easily scalable. However more recent work (as with CCG) has shown that the latter issue is not necessarily the case any more, as we now have more powerful parsers. Should we choose to follow a symbolic approach, we shall hope to prove the former issue no longer relevant, once we have moved up from simpler examples.

5 Comparison With Our Approach

TODO

Appendix A. **POS Tags**

Tag	Description
CC	Coordinating conjunction
CD	Cardinal number
DT	Determiner
EX	Existential there
FW	Foreign word
IN	Preposition or subordinating conjunction
JJ	Adjective
JJR	Adjective, comparative
JJS	Adjective, superlative
LS	List item marker
MD	Modal
NN	Noun, singular or mass
NNS	Noun, plural
NNP	Proper noun, singular
NNPS	Proper noun, plural
PDT	Predeterminer
POS	Possessive ending
PRP	Personal pronoun
PRP$	Possessive pronoun
RB	Adverb
RBR	Adverb, comparative
RBS	Adverb, superlative
RP	Particle
SYM	Symbol
TO	to
UH	Interjection
VB	Verb, base form
VBD	Verb, past tense
VBG	Verb, gerund or present participle
VBN	Verb, past participle
VBP	Verb, non-3rd person singular present
VBZ	Verb, 3rd person singular present
WDT	Wh-determiner
WP	Wh-pronoun
WP$	Possessive wh-pronoun
WRB	Wh-adverb

Table A.1: [**?**] List of position of speech (POS) tags

Appendix B. **ASG**

TODO