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

Co-Supervisor:
David Tuckey

Second Marker:
Krysia Broda

Monday 1st June, 2020

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

Abstract

To this day, text summarization remains a largely open-ended problem in Natural Language Processing, and is most often resolved using some form of Machine Learning.

In this project, we aim to resolve the problem for short texts about a paragraph in length, via a novel approach that makes use of Answer Set Grammars [1]. By combining ideas from the fields of logic-based learning, knowledge representation and linguistics, we have created a system that is capable of producing partially *abstractive*, *generic* and *informative* summaries, as well as scoring them by order of pertinence and information density.

The approach chosen for this project relies on a central context-free grammar as its internal representation for a simplified version of the English language. Using Answer Set Programming the system is able to perform logic-based learning after preprocessing the input text, the result of which then goes through a number of summarization logic rules, producing a set of possible summaries.

Apart from being developed using a suite of targeted examples, we tried to train an *encoder-decoder* with summaries that were generated by our system, helping to ensure that our approach to summarization made sense.

Overall, we believe that this logic-based approach is promising and has the potential to play a part in the future of automated text summarization.

Acknowledgements

Firstly, I would to thank my supervisor Prof. Alessandra Russo, as well as my co-supervisor David Tuckey, for the time you have put into this project. Week after week, we have had very interesting discussions in which you have provided your respective expertise in logic programming and NLP, helping to move the project forward. I would also like to thank Mark Law for creating ASG, as well as helping we with some of the syntax and numerous optimizations. In addition, I would like to thank my Personal Tutor, Antonio Filieri, for providing advice and guidance throughout the four years at Imperial. Finally, I am grateful for my family, who supported me during my studies.

Contents

Chapter 1 Introduction

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. To this end, a system performing such a task must analyse and process the input in order to extract from it the most important information.

1 Motivations

In recent years, state-of-the-art systems that accomplish text summarization have relied largely on Machine Learning. These include Bayesian classifiers, hidden Markov models, neural networks and fuzzy logic, among others [2]. Given a training corpus, along with some careful preprocessing as well as fine-tuning of hyper-parameters and feature extraction functions, such systems are able to produce effective summaries.

However to learn what is a summary, these systems require a very large amount of data, and take a long time to train. In contrast, using logic means that we can hard-code this definition directly into our program, avoiding the problem of randomness and uncertainty. By carefully constructing its structure, we can get results with just a short list of rules, and know that it will always produce a complete and valid output with respect to the background knowledge we encode into it.

2 Objectives

The main goal of this project is to explore the task of text summarization via logic-based learning with Answer Set Grammars (ASG). Below you will find the principal objectives which were established as being vital to achieving this goal.

Objective 1 (Research Existing Summarization Methods)**.** Before diving into the task of logic-based summary generation, we should investigate the approaches used by existing state-of-the-art systems. Even if these are not logic-based, they might rely on techniques that could be beneficial to use for this project.

Objective 2 (Translate English Into ASG)**.** Our system should be capable of taking a text written in English and converting it into some logic-based form that can be interpreted by ASG. Moreover, this internal representation should capture as much of the semantics from the original text as possible, and not be limited to a particular domain.

Objective 3 (Generate Summaries Automatically)**.** Given a brief paragraph of text, for example a short story aimed at young children, we should be able to provide a grammatically correct summary in multiple sentences. This should be fully-automated and not require any human intervention during the process.

Objective 4 (Evaluate The Approach)**.** Once we have implemented the basic approach, we should run our system on a suite of examples to verify that it can

produce summaries that closely resemble the corresponding human-generated ones. On a larger scale, it is important to also run it against a popular summarization approach to ensure the sanity of our logic-based mechanism.

3 Approach Overview

In what follows, we shall give a brief overview of the various tasks that were undertaken and completed as part of the project. Where relevant, references to the corresponding sections are provided.

The approach described in this paper, known as SumASG*, can be diagrammatically represented as a three step pipeline, as seen in Figure 1.1. Although the focus of this project is a mechanism written in ASG, it relies on a number of Python scripts to process information and coordinate its flow.

Figure 1.1: Main Pipeline

We begin by describing the essential role of the Preprocessor in Chapter 3. Given an input story, its goal is to simplify the story's sentences into a simpler and more consistent structure that will then be easier to parse by ASG (Section 2). On top of this, the Preprocessor also removes irrelevant sentences from the story and reduces the lexical diversity (Section 3), which helps increase the chances of generating a high-quality summary.

In Chapter 4 we discuss the use of ASG in the context of this project, forming a procedure we call SumASG. It all revolves around a purpose-built internal representation of English sentences, represented as a tree (Section 2). The first of two steps, SumASG_1, involves translating sentences from the input story into our internal representation using ASG's learning abilities (Section 3). From this, we then use a number of logic-based rules to generate sentences which may be used to form a summary (Section 4).

The third part of the pipeline, as outlined in Chapter 5, serves to turn the output of SumASG into usable summaries. To begin, we post-process the summary sentences given to us as output, and combine them in different ways so as to form potential summaries (Section 2). Afterwards, we discuss a mechanism used to score these summaries (Section 3), and explore ways in which to select those that are optimal (Section 4).

3.1 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 1.2.

Additional examples of stories can be found in Appendix A, along with summaries generated by SumASG* and in some cases the corresponding *reference summary*.

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 mathematics and quantum physics. He studied hard for his exams and became an astrophysicist. Now he is famous.

(a) Story of Peter Little

A. Peter Little was interested in space so he studied hard and became a famous astrophysicist.
B. Peter Little was curious about astronomy. He was always serious in school, and now he is famous.

(b) *Reference summaries*

Figure 1.2: Example of the task of summarization for the story of Peter Little

4 Contributions

The main contribution of this project to the field of NLP is the creation of an end-to-end logic-based system capable of text summarization, without the need of any training whatsoever, as would be the case with a typical Machine Learning-based approach these days. More specifically, the fundamental achievements are as follows:

Contribution 1. TODO

TODO more in-depth

Chapter 2 **Background**

1 Summarization

1.1 Definition

As described by the author in [3], 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 [4] 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* [5] 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 *parse tree*, which makes it possible to identify different parts of speech, such as nouns, verbs and adjectives. Additionally, each of these *tokens* is assigned a *lemma*, which is essentially the "neutral" form for a word, be it the singular of a noun or the base form of a verb.

Example *parse trees* are shown in Table 2.1 for both of the mentioned parsers. Appendix 2.3 shows how to interpret position of speech (POS) tags.

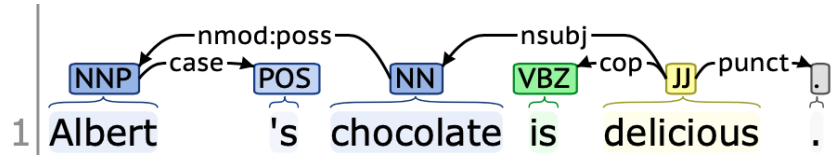
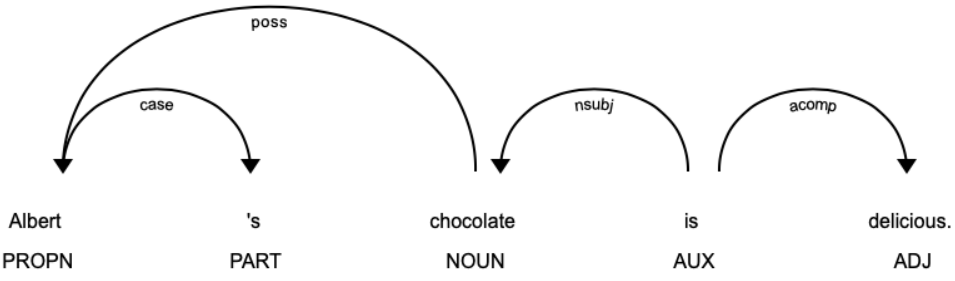
CoreNLP	
spaCy	

Table 2.1: Semantic parsing example for a basic sentence

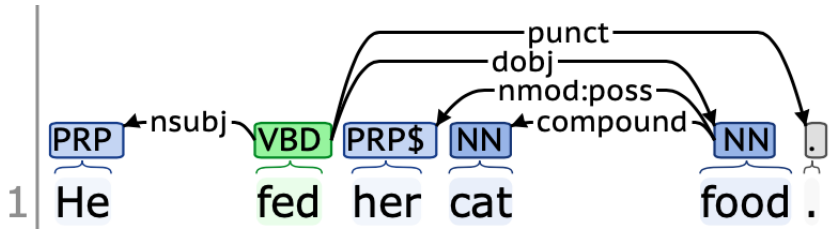
However, the English language can often be ambiguous and highly context-dependent, meaning that multiple *parse trees* for the same sentence could emerge. Consider the following two sentences [6]:

> 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* [7].

CoreNLP



spaCy

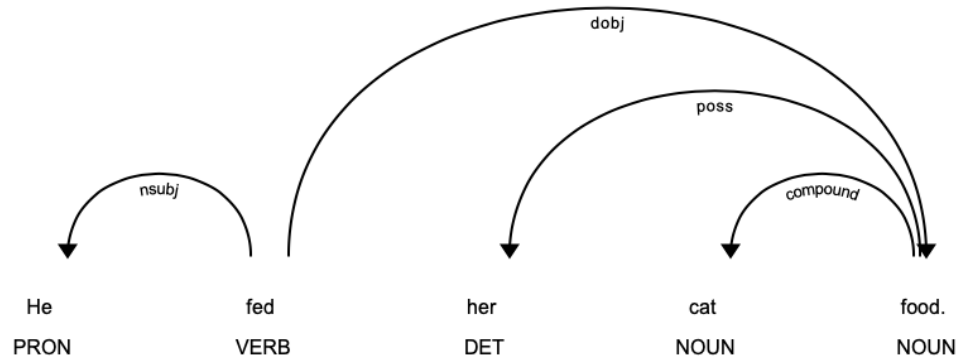


Table 2.2: Semantic parsing example for an ambiguous sentence

3 Logic Programming

Definition 1 (Term [8])**.** 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 [8])**.** 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 [9]. 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* [1].

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

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 [1])**.** 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 [1])**.** 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 [1])**.** 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 [1])**.** 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 [1])**.** 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 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 [10])**.** 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 [1])**.** 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 $|children(PT)| = k$.

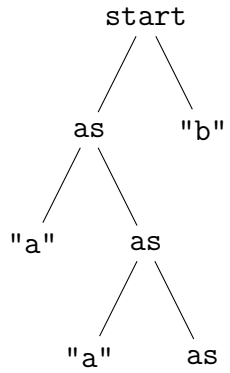
Definition 11 (Trace [1])**.** We can represent each node n in a *parse tree* by its *trace*, $trace(n)$, through the tree. The *trace* of the root is the empty list `[]`; the i$^{\text{th}}$ child of the root is `[i]`; the j$^{\text{th}}$ child of the i$^{\text{th}}$ child of the root is `[i, j]`, and 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 4.3 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 [1])**.** 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 [1])**.** 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 [1])**.** 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. [1]

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 RNNs

A recurrent neural network (RNN) is a chain-like neural network that is applied once for each *token* in the input sequence. At each timestep t in a "vanilla" RNN (i.e., for each item in the sequence), the current *hidden state* h_t is computed as a function of the previous *hidden state* h_{t-1} and the current input *token* x_t, using a non-linear *activation function* (usually sigmoid). [11]

4.2 LSTMs

A long short-term memory (LSTM) is a particular kind of RNN, motivated by the problems of vanishing and exploding gradients which sometimes occur due to the chain-like nature of RNNs. The solution here is to at each timestep use what is called a *memory block*, which holds at it center a linear unit that is connected to itself. In addition, it has three *gates*: input (i), forget (f) and output (o). Respectively, these three *gates* are concerned with which information to store, how long to store it, and when it should be passed on. [12]

Finally, each *memory block* also stores a *cell state* c_t, which combines information from the previous *cell state* c_{t-1} and from the *candidate state* g_t (defined as per the *hidden state* in a vanilla RNN), using the *forget* and *input gates* to regulate information flow. The *hidden state* h_t is then defined as a function of this *cell state*, controlled by the *output gate*. Figure 2.6 shows this diagrammatically. [13]

$$
\begin{aligned}
i_t &= \sigma(W_{xi} \cdot x_t + W_{hi} \cdot h_{t-1} + W_{ci} \cdot c_{t-1} + b_i) \\
f_t &= \sigma(W_{xf} \cdot x_t + W_{hf} \cdot h_{t-1} + W_{cf} \cdot c_{t-1} + b_f) \\
o_t &= \sigma(W_{xo} \cdot x_t + W_{ho} \cdot h_{t-1} + W_{co} \cdot c_t + b_o) \\
g_t &= tanh(W_{xg} \cdot x_t + W_{hg} \cdot h_{t-1} + b_g) \\
c_t &= f_t \cdot c_{t-1} + i_t \cdot g_t \\
h_t &= tanh(c_t)
\end{aligned}
\tag{2.2}
$$

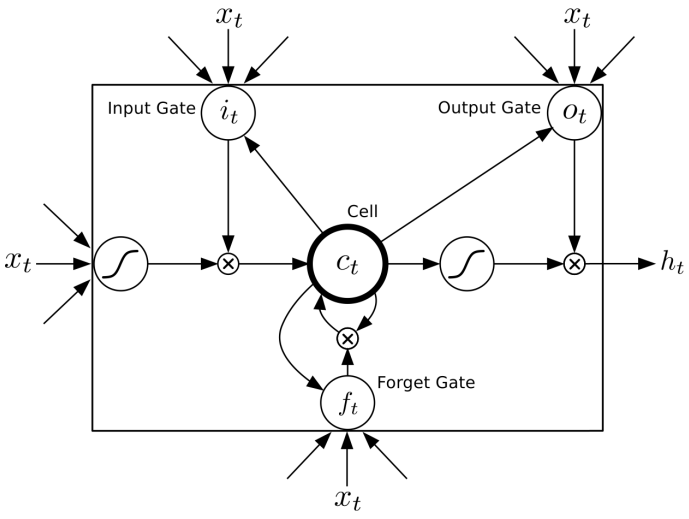


Figure 2.6: [13] Diagram showing the flow of information in an LSTM *memory block*

4.3 Encoder-Decoders

An *encoder-decoder* is a neural network consisting of two eponymous RNNs, are often LSTMs. The *encoder*'s job is to translate any variable-length sequence given as input into a fixed-length vector representation, while the *decoder*'s role is to transform this into a new variable-length sequence. Together, they are trained to maximize the probability of generating a target sequence given the corresponding input sequence. [11]

4.4 Attention

In the context of neural machine translation (NMT) systems such as *encoder-decoders*, the driver behind *attention* is to improve performance by selectively looking at sub-portions of the input sequence, which becomes especially important for long sequences of text. [14]

4.4.1 Global Attention

On top of being dependant on the last *hidden state* d_{t-1} and output *token* y_{i-1}, every *hidden state* d_t in a *decoder* with *global attention* is computed also in function of its *context vector* c_t. Each *context vector* c_t is defined as a weighted sum of the *encoder*'s *hidden states* h_i. The h_i that are assigned a higher weight are that which are more similar to d_t, which is done using a trainable *alignment model* a. [15]

$$\begin{aligned} c_t &= \sum_i \alpha_{ti} \cdot h_t \\ \text{where } \alpha_{ti} &= \frac{exp(s_{ti})}{\sum_j exp(s_{tj})} \\ s_{ti} &= a(d_{t-1}, h_i) \end{aligned} \tag{2.3}$$

Here the idea is that α_{ti} tells us how important each h_i is with respect to the current timestep t, influencing the value of d_t and hence the decision of the generated output *token* y_t. Essentially, this is a way of telling the *decoder* which parts of the input sequence to pay attention to. Thanks to this mechanism, the *encoder* is no longer forced to compress all the useful information from an input sequence into a fixed-length vector, thus improving performance for longer sequences of *tokens*. [15]

Chapter 3 Preprocessor

1 Overview

In order to prepare the story to be parsed and summarized by SumASG, we have created the Preprocessor. As we will loose information in summary anyway, it is completely acceptable to... which steps are shown below in Figure 3.1.

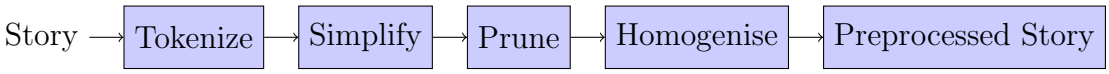


Figure 3.1: Preprocessor Steps

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 throughout.

2.1 Punctuation

To avoid having to build recognition and semantic understanding of different types of punctuation into SumASG, it is preferable to transform the story such that it uses no punctuation apart from full stops. The idea is that each sentence in the resulting text contains exactly one action or description.

Depending on the type of punctuation used at the end of a clause, a different treatment is applied:

- Question marks: We remove the clause, as it is most likely irrelevant for this task. It also helps avoid negation since we are deleting rhetorical questions.
- Dashes: These are used around clauses which add detail, so it is quite safe to delete them for the task of summarization.
- Exclamation marks, commas, semi-colons and colons: We replace any of these with a full stop.

2.2 Individual Word Transformations

One of the main goals of the Preprocessor is to transform the story in a simple and consistent structure, one where a given POS tag may only appear in a limited number of places in a sentence.

2.2.1 Acronyms

Some acronyms are often spelled using full stops after each letter. To prevent these from being recognized as multiple sentences, it is beneficial to remove any punctuation from acronyms. Therefore the word “U.S.A.” would become “USA”.

2.2.2 Contractions

Contractions can be difficult to understand for machines, and they add unwanted complexity to the task of parsing. Therefore it is simplest to expand all of them, for example transforming "it's" into "it is".

2.2.3 Adverbs

In the English language, adverbs can appear almost anywhere in a sentence, and their position has minimal semantic influence. To illustrate this, consider the following sentences, which all have the same meaning:

Slowly he eats toast.
He slowly eats toast.
He eats toast slowly.

In order to provide SumASG with a consistent format for parsing adverbs, we should always move them to the end of the clause in which they appear (in the above example we would keep the last sentence).

2.2.4 Determiners

In the English language, the determiners "a" and "an" are semantically identical, so to makes sense to only use one of the two, with the intention of reducing the number of *tokens* that SumASG has to process. We can always correct the output of SumASG to make it grammatically-correct.

2.2.5 Possessive Pronouns, Interjections And Prepositions

In most cases, possessive pronouns and interjections do not add much to the meaning of a story, especially when the end goal is to create a summary. Therefore, we can remove such words from the text. For instance, the sequence "Ah! She ate her chocolate." would become "She ate chocolate.".

Prepositions which appear at the start of a sentence may be removed, as they are not integral to the meaning. For example, "Besides today is Sunday" gets transformed into "Today is Sunday".

Moreover, prepositions which come after the object in a sentence can sometimes cause it to become syntactically too complex. Rather than encoding such high-level of detail into the internal representation of SumASG, it is preferable to simply omit the final clause. In this case, "They have a picnic under a tree." becomes "They have a picnic.". Although some information is thrown away, this loss will usually have no impact on the quality of the summary.

2.3 Clause Transformations

After going through the Preprocessor, we would like each sentence in the given story to only focus on a single topic.

When possible, we should split sentences containing multiple clauses into individual sentences. Otherwise, we can delete the auxiliary clause and keep the main clause.

Examples of the transformations applied to different types of clauses can be seen below in Figure 3.2.

Conjunctive Clause We looked left and they saw us.
Conjunctive Clause. Cars have wheels and go fast.
Subordinating Clause. She never walks alone because she is afraid.
Dependant Clause. I want to be President when I grow up.
Dependant Clause. When I grow up, I will have a garden.

(a) Before transformation

Conjunctive Clause. We looked left. they saw us.
Conjunctive Clause. Cars have wheels. Cars go fast.
Subordinating Clause. She never walks alone. she is afraid.
Dependant Clause. I want to be President.
Dependant Clause. I will have a garden.

(b) After transformation

Figure 3.2: Examples of the splitting of multi-clause sentences

2.3.1 Hypernym Substitution

However, in some cases we may be able to perform an optimization that allows us to collapse a conjunction of two words into a common *hypernym* (i.e. superclass).

In practice, this involves using **Pattern** to try and find a lexical field to which both words pertain.

For example, the words "chicken" and "goose" both belong to the lexical field of "poultry". Similarly, "cars" and "trucks" have common hypernym "motor-vehicles".

TODO give more details about how chosen?

2.4 Case And Proper Nouns

We want to ensure that all occurrences of a word are treated as the same *token*. Since SumASG will be generating new sentences from scratch, the simplest solution is to convert the entire story to lower-case, apart from proper nouns.

In the case of complex proper nouns (i.e., those constructed from multiple words), we should remove inner spaces so that we end up with a camel-case string. For instance, the sequence "Peter Little" will become "PeterLittle".

We can also do this with complex common nouns, for example transforming "bird house" into "bird-house".

2.4.1 Pronoun Substitution

Sometimes, an author will introduce a character or group by name, and later refer to them using a pronoun.

If a story contains exactly one distinct singular proper noun and then uses either "he" or "she", then it is safe to assume that this pronoun refers the aforementioned proper noun. The same can be said about plural proper nouns and the pronoun "they". To clarify this, an example is shown in Figure 3.3.

Antonio is a cheesemaker. He makes burrata. **Italians** eat pasta. They make it with egg sometimes.

(a) Before transformation

Antonio is a cheesemaker. **Antonio** makes burrata. **Italians** eat pasta. **Italians** make it with egg sometimes.

(b) After transformation

Figure 3.3: Example of substituting pronouns with proper nouns

2.5 Example

The result of running the simplification transformations is illustrated below in Figure 3.4. Please refer to Chapter **??** for the original story.

ⓐ Adverbs: move "always" and "Now" to the end of the sentence
ⓑ Determiners: "an" → "a"
ⓒ Possessive pronouns: ~~"his"~~
ⓓ Prepositions: ~~"So"~~
ⓔ Conjunctive clauses: "serious in school" ‖ "did homework always"
ⓕ Dependant clauses: ~~"When he was older"~~
ⓖ Complex clauses: "was a curious little boy" ‖ "named Peter Little"
ⓗ Hypernym substitution: "stars and planets" → "astronomy"
ⓘ Case: make all words but proper nouns lower case
ⓙ Complex nouns: "Peter Little" → "PeterLittle"
ⓚ Complex nouns: "quantum physics" → "quantum-physics"
ⓛ Pronoun substitution: "he" → "PeterLittle"

(a) Transformations applied (where ‖ means splitting into multiple sentences)

0. there was a curious little boy.
1. ⓖ the curious little boy was named ⓙ PeterLittle.
2. ⓛ PeterLittle was interested in ⓗ astronomy.
3. ⓓ PeterLittle was serious in school.
4. ⓔ PeterLittle did ⓒ homework ⓐ always.
5. ⓕ PeterLittle studied mathematics and ⓚ quantum-physics.
6. PeterLittle studied for ⓒ exams hard.
7. PeterLittle became ⓑ a astrophysicist.
8. PeterLittle is famous ⓐ now.

(b) Sentences after applying transformations

Figure 3.4: Example of applying *simplification* to the story of Peter Little

3 Sentence Pruning And Homogenization

Once the sentence structure of the story has been *simplified*, one of the main jobs of the PREPROCESSOR is to remove irrelevant semantic complexity from the story.

In order to understand what is relevant in a story and what is not, the PREPROCESSOR looks at the semantic similarity between words with the same POS tag (or a related one, i.e. singular noun and plural noun, or verbs with a different tense).

3.1 Word Similarity

It uses a loop to iterate over each sentence, and compares each word with every word from other sentences which have the same or a related POS tag. For each comparison, word *similarity* is computed using the tool ConceptNet, a semantic knowledge network.

As you might imagine, having such nesting of loops can be quite expensive, which is why we keep a cache of previously requested similarities, and use the fact that this *similarity* relation is symmetric.

3.2 Sentence Similarity And Pruning

3.2.1 Sentence Similarity

Once we have computed the *similarity* between words of different sentences, we can add these up on a per-sentence basis, which gives us a binary relation of *similarity* between sentences.

We now have enough information to generate a *text relationship map* over the sentences. The idea is that the more "linked" a sentence is, the more relevant it is to the story. For each sentence, we therefore take the sum of the values of all its *similarity* relations with other sentences. The higher this number, the more relevant, or *important*, the sentence is to the story.

3.2.2 Pruning

In the interest of removing irrelevant sentences to help SUMASG, we compute the 25th percentile over the *importances* of all the sentences. We then prune sentences whose *importance* is strictly less than this value.

In most cases, one quarter of the story will be pruned. However, if every sentence has the same *importance*, then nothing gets removed. On the other hand, if two thirds of the story are very *important* and the rest is irrelevant, then we remove more than a quarter of the sentences.

3.3 Synonyms And Homogenization

Another use for *word similarity* is to find out if the author of the story has used any synonyms. When the *similarity* between two words is above a certain threshold, then we consider them to be synonyms.

For every set of synonyms we find, we can choose a unique *representative* for the set, and replace occurrences of the other words in that set with our *representative*. For simplicity, we choose the *representative* as the shortest word in its synonym set.

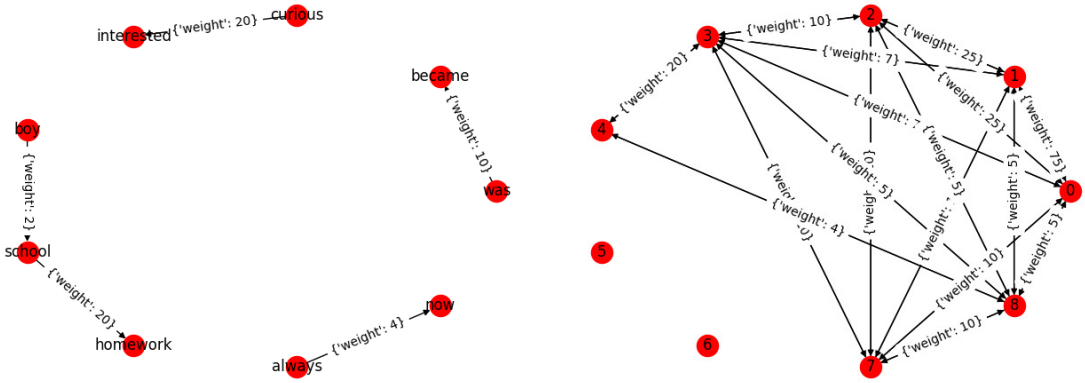
This is what we call *story homogenization*, and it helps SumASG link words that would otherwise be considered completely different *tokens* in the story.

3.4 Example

An example is shown below in Figure 3.5 for the story of Peter Little, where the weight between two nodes denotes *similarity*. In this example, we consider words to be synonyms whenever their *similarity* is at least 20. The labels of the nodes (starting from 0) for sentence *similarity* correspond to the index of the sentence in the *simplified story* (see Subsection 2.5).

As you can see, sentences 5 and 6 are deemed irrelevant and will be pruned, while sentences 0, 1 and 2 are thought as being very important.

Moreover, homework, school and interested, curious are considered synonym sets, and *homogenization* will replace occurrences of these words with their shortest synonym (which may be itself).



(i) Word *similarity* (ii) Sentence *similarity*

(a) *Text relationship maps*

Sentence	0	1	2	3	7	8	4	5	6
Importance	40.7	24.4	18.8	14.8	12.5	11.3	6	0	0

(b) Sentences ordered by importance

there was a curious little boy. the curious little boy was named PeterLittle. PeterLittle was curious in astronomy. PeterLittle was serious in school. PeterLittle did school always. PeterLittle became a astrophysicist. PeterLittle is famous now.

(c) *Homogenized* story

Figure 3.5: Example of applying sentence pruning and *homogenization* to the *simplified* story of Peter Little

4 Expandability

In its current state, SumASG expects positive sentences only, and the only form of punctuation recognized is the full stop.

4.1 Negation

In order to support negation, we would need to modify the structure of SumASG's internal representation (see Chapter 4). However, to achieve a better semantic understanding in SumASG*, we could add some more simplification logic to the Preprocessor.

After having implemented this, the phrase "not happy" would be transformed into the word "sad".

4.2 Lists

At the moment, SumASG can parse a list of length 2 at the most, i.e. a conjunction of two items. By adding a transformation to the Preprocessor before we modify the punctuation (see Subsection 2.1), we could overcome this limitation. Intuitively, this would mean going from a sentence with a an n-item list, to $\lfloor \frac{n}{2} \rfloor$ sentences with two objects and one sentence with a single object (if n is odd).

For instance, the sentence "Bob had a book, a computer and a chair." would be split into "Bob had a book and a computer. Bob had a chair".

Chapter 4 **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 4.1.

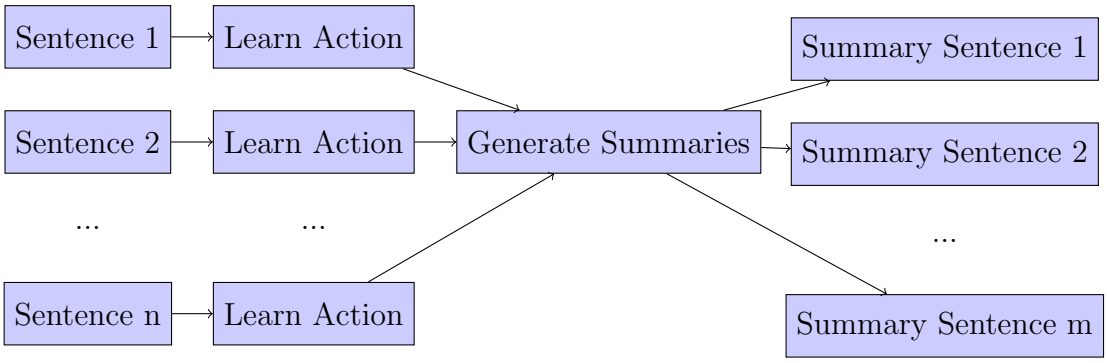


Figure 4.1: ASG Steps

2 Internal Representation

In order to model the structure of sentences the English language, we have created a CFG that has a similar hierarchy to that of an NLP parse tree. The ASG code for this general structure can be seen in Appendix B. Throughout this description of SumASG, please refer to Chapter 2 for information on how to interpret an ASG program. Also, a table listing the possible POS tags is available in Appendix 2.3.

2.1 Leaf Nodes

At the bottom end of the structure, there are leaf nodes that correspond to individual English words. These nodes are added based on the context, that is to say the words appearing in our story.

Each of these nodes has on the LHS (left-hand side) of the derivation its pos tag, and on the RHS (right-hand side) a string containing the word itself. In order to conform to the syntax of ASG, we must write the POS tags in lower-case. Also, we include a space at the end of each word's textual representation so that when we run our program the words appear distinct and not all concatenated together.

In ASG every derivation also has a set of ASP rules, which in the case of leaf nodes is just a single rule telling us the word's *lemma* and sentence *role*. In the case of verbs, the *lemma* is the base form of the verb, so we also need to keep track of its tense.

For example, leaf nodes for the sentence “they drove a race-car fast.” would look like this:

```
1 prp −> ”they ” { noun(they). }
2 vbd −> ”drove ” { verb(drive,past). }
3 dt −> ”a ” { det(a). }
4 nn −> ”race−car ” { noun(race_car). }
5 rb −> ”fast ” { adj_or_adv(fast). }
```

As part of the input to SumASG, we receive some leaf nodes corresponding to words in the story, where the *lemmas* and *roles* have been assigned by the Preprocessor.

In Figure 4.2, you can see which POS tags fall under which *roles*, keeping in mind that this categorization is only an optimization and was not intended to strictly adhere to English grammar.

Role	POS tags
`verb(lemma,tense)`	VBD, VBG, VBN, VBP, VBZ
`noun(lemma)`	EX, NN, NNS, NNP, NNPS, PRP
`det(lemma)`	CD, DT, IN
`adj_or_adv(lemma)`	JJ, JJR, JJS, RB, RP

(a) POS tags by *role*

POS tag	VBD	VBG	VBN	VBP	VBZ
Verb tense	`past`	`gerund`	`past_part`	`present`	`present_third`

(b) Verb tense by POS tag

Figure 4.2: Predicates used for the leaf nodes in the internal representation

2.2 Non-Leaf Nodes

The job of the non-leaf nodes is to join leaf nodes together, matching the way we would join words in English to form a sentence.

In our general grammar, sentences (`s -> np vp`) are made up a *noun part* (`np`) followed by a *verb part* (`vp`). While a *noun part* can be made up of leaf nodes, a *verb part* is always a verb followed by a *noun part*.

2.2.1 Noun Parts

In the derivation for a *noun part*, we use logic rules whose role it is to encapsulate a sentence *subject* and/or an *object*. This is done in a bottom-up manner, by using information from the child node(s) to populate a predicate at the *noun part* level.

The resulting predicates have respective forms `subject(noun,det,adj_or_adv)` and `object(noun,det,adj_or_adv)`. For all these predicates, we use the *ground term* `0` to denote the absence of a *token*.

For instance, we can capture the *noun phrase* "a race-car" using the following derivation:

```
1 np −> dt nn {
2   subject(N,D,0) :− det(D)@1, noun(N)@2.
3   object(N,D,0) :− det(D)@1, noun(N)@2.
4 }
```

To handle the case of more complex *noun phrases*, we have created a special predicate `conjunct(first,second)`, allowing us to join two words with the same *role*. We also need to add constraints that rule out cases where the two *conjuncts* have the same *lemma*.

For example, the *noun phrase* "bread and cheese" would be encompassed by the below derivation:

```
1 np −> nn "and " nn {
2   subject(conjunct(N1,N2),0,0) :− noun(N1)@1, noun(N2)@3.
3   object(conjunct(N1,N2),0,0) :− noun(N1)@1, noun(N2)@3.
4   :− subject(conjunct(N,N),0,0).
5   :− object(conjunct(N,N),0,0).
6 }
```

2.2.2 Verb Parts

The last child node of a *verb part* is always a single *noun part*. Before that comes a verb, whose POS tag may represent any of the forms used in English as seen in Figure 4.2. In each of these cases, the node inherits the `verb(lemma,tense)` and `object(noun,det,adj_or_adv)` from its children.

For instance, the *verb phrase* "drank tea" can be captured with the following derivation:

```
1 vp −> vbd np {
2   verb(N,T) :− verb(N,T)@1.
3   object(N,D,A) :− object(N,D,A)@2.
4 }
```

In order to handle continuous tenses, we introduce the predicate `comp(first,second)`. Without changing the *arity* of our predicate `verb(lemma,tense)`, we can use this to combine two verb *lemmas*, as well as two verb tenses.

For example, the derivation that handles the *verb phrase* "are eating apples" is the following:

```
1 vp −> vbp vbg np {
2   verb(comp(N1,N2),comp(T1,gerund)) :− verb(N1,T1)@1, verb(N2,
      ↪ gerund)@2.
3   object(N,D,A) :− object(N,D,A)@3.
4 }
```

2.2.3 Sentences

In order to join sentences (`s -> np vp`) together we use what is called an `s_group`. Defined recursively, these can either be empty or contain another `s_group` followed by a sentence (`s`) and a full-stop:

```
1 s_group −> { count(0). }
2 s_group −> s_group s ". " { count(X+1) :− count(X)@1. }
```

In the way we currently use this general grammar, only a single sentence is allowed per `parse tree` for efficiency reasons, and because it is not necessary. However, if we we were to increase this limit for another application, it could easily be done by changing the first constraint at the root node:

```
1 start −> s_group {
2     :− count(X)@1, X > 1.
3     :− count(X)@1, X = 0.
4 }
```

3 Learning Actions

We first need to convert the preprocessed story's sentences from English into our internal structure. In other words, we need to learn about the *actions* described by the sentences in our story, which can be thought of as high-level semantic descriptors.

3.1 Formalization

We can formalize the task of learning an action as $\textsc{SumASG}_1$(CFG,BK,E). Given our general grammar (CFG), a set of context-specific leaf nodes (BK), and a grammar-conforming sentence (E), its goal is to return the *action* corresponding to this sentence, which should have the format `action(verb,subject,object)`.

However this is not a learning task in the true sense, as we are only interested in generating ground facts *ground facts*. More to the point, we make use of the built-in learning mechanism offered by ASG to translate the input into our internal representation.

3.2 Implementation

3.2.1 Positive Example

In practice, what we do is append to the program containing our general grammar the context-specific leaf nodes (given to us by the Preprocessor), as well as a positive example containing our sentence to learn from. For instance, we would add the positive example for the sentence "they drove a race-car fast.":

```
+ ["they ", "drove ", "a ", "race−car ", "fast ", ". "]
```

We also need to ensure that the derivation for sentences contains a constraint enforcing that an `action` be learned when an example is given:

```
s −> np vp {
  :− not action(verb(V_N,V_T),subject(S_N,S_D,S_A),object(O_N,O_D
      ↪ ,O_A)), verb(V_N,V_T)@2, subject(S_N,S_D,S_A)@1, object(
      ↪ O_N,O_D,O_A)@2.
  ...
}
```

3.2.2 Mode Bias

In order guide the learning task, we must also specify a *mode bias* as part of the program for SumASG$_1$, which essentially tells ASG the format of the rules which can be learned.

Since we are only interested in learning *facts* (rules with an empty *body)*, it is enough to provide *mode bias* rules of the following form (where `[4]` restricts the learning task to the fourth derivation):

```
#modeh(action(verb(_,_), subject(_,_,_), object(_,_,_)):[4].
```

For the most basic of sentences (ones where there is no need to use any `conjunct` or `comp` predicates), we use this specific rule:

```
#modeh(action(verb(const(main_verb),const(main_form)), subject(
    ↪ const(noun),const(det),const(adj_or_adv)), object(const(noun),
    ↪ const(det),const(adj_or_adv))):[4].
```

As you can see, this would require defining ILASP *constants* corresponding to possible *tokens*. To this end, we do so for each word in the *simplified* text.

For the sentence "they drove a race-car fast", these would look like this:

```
#constant(noun,they).
#constant(main_verb,drive).
#constant(main_form,past).
#constant(det,a).
#constant(noun,race_car).
#constant(adj_or_adv,fast).
```

3.2.3 Running

Once we have augmented our general grammar with all of this information, it is now possible to run the resulting program with the below command, causing an *action* to be output to the command line. Here we use a *depth* of 7, having found that this is the minimum number necessary to ensure that ASG has access to the lowest possible leaf nodes in the tree.

```
asg action.asg −−mode=learn −−depth=7
```

For the input sentence "they drove a race-car fast.", the engine returns a new ASG program without the *mode bias*, where the following *action* has been added to the derivation for sentences (`s -> np vp`):

```
action(verb(drive, past), subject(they, 0, 0), object(race_car, a, fast)).
```

3.3 Search Space Reduction

The set of rules that a task in ILASP is able to learn, as defined by the *mode bias*, is called the *search space*. The more complex the structure of the rules we can learn, the more of these the engine can generate, and so the larger the *search space*. The more leaf nodes we add, the more combinations of *lemmas* we can create, thereby exponentially growing the *search space*. Since ASG tries to run the program with every single rule in the *search space*, we need to keep this as small as possible.

Throughout the development of SumASG$_1$, it was very helpful to view the *search space* via the following command:

```
asg action.asg --mode=ss --depth=7
```

3.3.1 Learning Actions Individually

With this in mind, it is preferable to feed in each sentence separately to SumASG$_1$. Although it might seem easier at first to learn them all in one go, doing so individually limits the number of leaf nodes we need to add to the program.

Using this optimization, learning the *actions* from the *simplified* and *homogenized* story of Peter Little takes just a few minutes, rather than many hours.

3.3.2 Cutting Out Rules

We have also created a number of *mode bias* rules which eliminate impossible or extremely improbable sentences. With this optimization, we have been able to take the search space size for a simple sentence down from 396 to 16, and from 9477 to 1044 for a more complicated one (i.e., one with more leaf nodes).

For example, the following rule says that we cannot have an *action* where the object of sentence is a conjunction of two words which both have the same *lemma*.

```
#bias(":- head(holds_at_node(action(verb(_,_),subject(_,_,_),object(
    ↪ conjunct(V,V),_,_)),var__(1))).").
```

Additionally, a number of extraneous rules can appear in the *search space* when we allow for continuous verbs. Continuous verbs are made up of a *main verb* and an *auxiliary verb*. What can happen is that the *search space* contains rules where the *main verb* is never used as such in English (normally always the verb "to be").

To get around this issue, we enforce that all potential *main verbs* already appear in this form in the input story sentence; the same can be said regarding *auxiliary verbs*. Practically, this means adding *constants* to the program for each *main verb* and *auxiliary verb* appearing in the input.

For instance, the phrase "are eating" would require the following *constants*:

```
1 #constant(main_verb,be).
2 #constant(aux_verb,eat).
3 #constant(main_form,present).
4 #constant(aux_form,gerund).
```

Without the optimization, we would end up with a *search space* size of 176 for the sentence "they are eating apples". We can reduce this number to 20 thanks to a *mode bias* that enforces learned continuous verbs to be exactly as they appear in the *simplified* and *homogenized* story:

```
#modeh(action(verb(comp(const(main_verb),const(aux_verb)),comp(
    ↪ const(main_form),const(aux_form))), subject(const(noun),const(
    ↪ det),const(adj_or_adv)), object(const(noun),const(det),const(
    ↪ adj_or_adv)))):[4].
```

Another way to solve this would have been to add a *mode bias* constraint ruling out cases where both verbs in a continuous form are the same. However, we would usually end up with a *search space* at least as large, since any verb could appear in continuous fo. Also, we would have to handle the edge case where both verbs are "to be", as "is being" is a perfectly acceptable phrase in English.

4 Generating Summary Sentences

4.1 Formalization

We can formalize the task of generating a *summary sentence* as $\textsc{SumASG}_2(\underline{\text{CFG}},\underline{\text{BK}},\underline{\text{E}})$. Given our general grammar (CFG), a set of context-specific leaf nodes for the original story (BK), and a set of learned *actions*, (E), its goal is to return a set of user-readable sentences which may be used to summarize the text.

4.2 Implementation

4.2.1 Learned Actions

In order to keep the story's chronological ordering, we assign indices to the learned *actions*, inserting this information directly into the `action` predicates as an additional first argument.

We then put all of these augmented *actions* as rules inside the derivation for sentences (`s -> np vp`) in our general grammar.

4.2.2 Summary Generation Rules And Constraints

A *summary sentence* should have the same structure as a sentence from the story, so we can define the predicate `summary` in the same way as we did for *actions*.

Moreover, we can create rules whose head is a `summary` predicate, and whose body contains one or more `action` predicates. We also assign an identifier to each of these rules, in order to keep track of which one has been used.

In the base case, a *summary sentence* is simply a word-for-word copy of an *action*, in which case we do not care about its position in the story:

```
summary(0, V, S, O) :- action(_, V, S, O).
```

We can create much more complex rules, allowing us to combine information from two *actions* into a single *summary sentence*. In the case where we have two *actions* that share a common *subject* and *verb*, we can define a rule that combines these into a single *summary sentence*, preserving the order in which these *objects* appear originally:

```
summary(7, V, S, object(conjunct(N1,N2),D,0)) :- action(I1, V, S,
    ↪ object(N1,D,_)), action(I2, V, S, object(N2,D,_)), N1 != N2, N1
    ↪  != 0, N2 != 0, I1 < I2.
```

After having defined a suite of such summarization rules, we now need to apply them using our general grammar. To this end, we add to the derivation for sentences (`s -> np vp`) a *choice rule*, enforcing with the predicate `output` that the program must output every derivable *summary sentence* exactly once. Using constraints, an `s` node can require that its children contain the required information.

```
1 0{output(I,V,S,O)}1 :- summary(I,V,S,O).
2 :- not output(_,_,_,_).
3
4 :- output(_,verb(V_N,V_T),subject(S_N,S_D,S_A),object(O_N,O_D,
      ↪ O_A)), not verb(V_N,V_T)@2.
5 :- output(_,verb(V_N,V_T),subject(S_N,S_D,S_A),object(O_N,O_D,
      ↪ O_A)), not subject(S_N,S_D,S_A)@1.
6 :- output(_,verb(V_N,V_T),subject(S_N,S_D,S_A),object(O_N,O_D,
      ↪ O_A)), not object(O_N,O_D,O_A)@2.
```

4.2.3 Running

Once we have augmented the fourth rule of our general grammar (`s -> np vp`) with the learned *actions* and our set of summary generation rules and constraints, we can now generate all the possible *summary sentences* with the below command.

```
asg summary.asg --mode=run --depth=7
```

5 Example

Figure 4.3 shows the different steps of running SumASG on the *simplified* and *homogenized* story of Peter Little, along with a breakdown of the runtime.

After passing in each sentence individually to SumASG_1, we end up with a list of *actions*, which is essentially the original story translated into our internal representation.

From these *actions*, we then apply SumASG_2 to generate all possible *summary sentences* which we can use to summarize Peter Little's story.

```
1 action(0, verb(be, past), subject(there, 0, 0), object(boy, a, conjunct(
      ↪ curious, little))).
2 action(1, verb(comp(be, name), comp(past, past_part)), subject(boy,
      ↪ the, conjunct(curious, little)), object(peterlittle, 0, 0)).
3 action(2, verb(be, past), subject(peterlittle, 0, 0), object(astronomy, in
      ↪ , curious)).
4 action(3, verb(be, past), subject(peterlittle, 0, 0), object(school, in,
      ↪ serious)).
5 action(4, verb(do, past), subject(peterlittle, 0, 0), object(school, 0,
      ↪ always)).
6 action(5, verb(be, present_third), subject(peterlittle, 0, 0), object(0, 0,
      ↪ conjunct(famous, now))).
```

(a) Results from SumASG$_1$

⓪ PeterLittle was serious in school .
⓪ PeterLittle was curious in astronomy .
④ PeterLittle was curious and serious .
⓪ PeterLittle did school always .
⓪ there was a curious little boy .
⓪ the curious little boy was named PeterLittle .
⓪ PeterLittle is famous now .

(b) Results from SumASG$_2$ (where the numbers indicate a summary generation rule)

Action	0	1	2	3	4	5
Running time (s)	18	32	9	9	7	9

(i) SumASG$_1$

Running time (s)	20

(ii) SumASG$_2$

(c) Runtime for each step

Figure 4.3: Example of running SumASG for the story of Peter Little

6 Expandability

Throughout the development of SumASG, it was a constant struggle to try and find the right balance between expressibility, summary pertinence and computational efficiency.

6.1 Missing English Structures

Although our general grammar allows for a wide range in terms of the words that can be used to form a sentence, to no extent does it cover even a tenth of the sentences that are used in formal or informal English. Even if you were to consider only sentences consisting of exactly one clause, SumASG is incapable of understanding most non-general structures commonly used in English.

By greatly simplifying the input story using the PREPROCESSOR, we were able to alleviate a large part of this struggle. However if we were to use our general grammar for a task other than summarization, we would most likely run into issues due to loss of information.

6.2 Missing Summarization Rules

In its current implementation, SUMASG$_2$ only uses 11 summary generation rules, one of which simply repeats the given *action*. While this is largely sufficient to demonstrate the potential of our approach, in no way can it be used as-is in a production text summarization tool.

In order to augment this suite of rules, it would be simple to define a *mode bias* to learn summary generation rules. With just a single example of a story and its corresponding summary, ASG could generate multiple such rules, allowing us to build up a large collection of these. Unfortunately, this is infeasible due to performance reasons.

6.3 Speed

Apart from readability, the main reason for trying to keep our general grammar's structure simple, and the number of summarization rules restricted, has to do with computational cost.

The more complexity we allow in terms of expressible sentences, the more expensive it is to use our general grammar.

Similarly, the more summarization rules we create, the longer it takes to generate summaries. On top of this, having more potential *summary sentences* means that we end up with more summaries to score, many of which could be syntactically different but semantically equivalent.

In order to increase complexity without a detrimental impact on performance, we would need to either optimize ASG itself to run faster with our framework, or use more powerful machines.

Chapter 5 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 5.1.

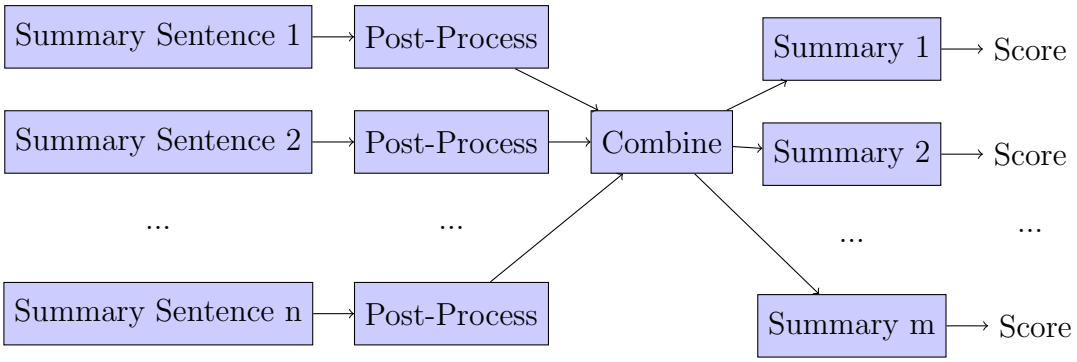


Figure 5.1: Post-Processing / Scoring Steps

2 Summary Creation

The output of SumASG is a list of sentences, each of which could potentially appear in the final summary.

2.1 Post-Processing

2.1.1 Grammar

Because SumASG uses the same capitalization for a given word regardless of its position in the sentence, it means that the first word of each sentence will not be capitalized unless it is a proper noun. We therefore need to fix this, as well as remove the space before each full stop.

Compound nouns, whose hyphen was replaced with an underscore for the internal representation of SumASG, also need to be restored to their grammatically-correct form.

In addition, the task of summarization might have created a sentence where an incorrect verb form is used, or possibly the wrong determiner. To amend this we use a tool called **language-check**, which is able to correct phrases like "they has an dog" to "they have a dog".

2.1.2 Complex Nouns

One of the optimizations done by the PREPROCESSOR was to combine complex nouns such as "Peter Little" into their camel-case form "PeterLittle", so that they would be recognized as a single *token* by SUMASG. We now need to expand them back to their original form, as this is how it should be written in English.

2.2 Combining

Depending on the length of the original story, we can envision and different number of sentences to be in the summary, as shown below in Table 5.1.

Story length	1-2	3-4	5+
Summary length	1	2	3

Table 5.1: Length of a summary depending on the number of sentences in the story

Once we have grammatically-correct summary sentences and know how many should be kept for the summary (say n), we generate all possible order-preserving combinations of length n. For instance, such combinations of length 3 for the list $[0, 1, 2, 3]$ would be the following: $[0, 1, 2]$, $[0, 2, 3]$ and $[1, 2, 3]$.

3 Scoring

As you can imagine, we often end up with a large number of combinations at this phase. We therefore need to determine which of them are best and keep only these.

3.1 TTR

To this end, we utilize an NLP metric called TTR (type-token ratio), a measure of lexical density. To provide the most informative summaries possible, we want to maximize the density of unique words.

To calculate a summary's TTR, we divide the number of unique words in the summary by the total number of words. We then divide this by number of unique words in the story and multiply it by a constant, in order to get a more consistent range for our scores.

3.1.1 Ignored Words

However, we do not want to neglect summaries using the same determiner, proper noun, or the verb "to be" multiple times, as these are extremely common in English.

In addition, a story might revolve around a given *topic*, which could be a person. Regarding the former, it could also be the case that the PREPROCESSOR had replaced different synonyms of this *topic* with a unique word.

To get around this, what we can do is to exclude such words from the summary length and number of unique summary words. This way, we no longer require that these "common" words be unique in a summary. In the following, we will call the enhanced mechanism TTR*.

In Figure 5.2 is an example which illustrates this metric. The summary with the highest final score is considered to be the best. As you can see, there is a greater difference between the summaries when using TTR*, which takes into account the commonly-used building blocks of the English language.

Jonathan was a little boy. He was hungry. Jonathan was eating an apple.

(a) Story

A. Jonathan was a hungry boy. Jonathan was eating an apple.
B. Jonathan was a little boy. Jonathan was a hungry boy.

(b) Possible summaries (underlined words will be ignored by TTR*)

Summary	Words	Unique words	TTR	Words*	Unique words*	TTR*	Score
A	10	8	0.8	4	4	1	38
B	10	6	0.6	4	3	0.75	28

(c) Steps for computing the score

Figure 5.2: Score computation (column headers ending with * pertain to TTR*)

4 Summary Selection

4.1 Proper Nouns

If a story revolves around a given person and the summary mentions their name, it is preferable for this to be in the first sentence. To put this more clearly, we would like the summary of a biography to introduce the protagonist from the very first sentence. To achieve this, we can simply increase the score of every summary starting with a proper noun by a constant.

4.2 Top Summaries

With a more complex story (5 or more sentences), it is highly likely that we will end up with a very long list of possible summaries. As there could be a number of very interesting summaries, we do not want to have to choose exactly one.

Instead, we compute 75th percentile over the scores of all generated summaries, and then discard all those whose score falls below this number. We shall call the remaining summaries *top summaries*.

4.3 Reference Summaries

Finally, we want to be sure that our framework generates good summaries, and that the scoring works as intended. Therefore, if a story has a *reference summary*, then we should make sure that there exists a similar *top summary*.

In our implementation, we have chosen to use the BLEU score, which measures how closely the output given by a machine matches a text written by a human. If

there exists a *top summary* whose BLEU score with one of the *references* is above a certain threshold, then we consider the summarization to be successful.

5 Example

An example is shown in Figure 5.3 for the story of Peter Little.

The first step is to fix the grammar in the *summary sentences* generated by SumASG, which in this case simply involves capitalizing them and removing the space before the full stop. We also need to restore the proper noun "PeterLittle" to "Peter Little". After *combining*, we end up with 35 possible summaries.

The next step is *scoring*, where we augment the standard of ignored words with the case-insensitive *topics* set {`"peter", "little"`} in TTR*. This gives us scores in the range $[10, 17]$, 20 of which fall below the 75th percentile of 15.0 and never become *top summaries*.

Finally, we compare these *top summaries* to our *reference summaries* for Peter Little, as mentioned in Chapter 1. As you can see from the computed BLEU scores, at least one of them achieves a score of at least 0.65, confirming they are close enough to *reference summary* **B**.

Peter Little is famous now.
The curious little boy was named Peter Little.
There was a curious little boy.
Peter Little did school always.
Peter Little was curious and serious.
Peter Little was curious in astronomy.
Peter Little was serious in school.

(a) Post-processed *summary sentences*

1. Peter Little is famous now. Peter Little did school always. Peter Little was curious in astronomy.
2. Peter Little is famous now. There was a curious little boy. Peter Little did school always.
3. Peter Little is famous now. The curious little boy was named Peter Little. Peter Little did school always.
4. Peter Little is famous now. Peter Little did school always. Peter Little was curious and serious.
...

(b) *Top summaries*

Summary	1	2	3	4
Reference A	0.4	0.38	0.32	0.38
Reference B	0.66	0.58	0.49	0.63

(c) BLEU scores for *reference summaries* (summary indices as shown in Figure 5.3b)

Figure 5.3: Example of *post-processing* then *scoring* for the story of Peter Little

6 Expandability

As you can see from the example in Section 5, there is much room for improvement regarding post-processing.

6.1 Grammatical Shortcomings

First of all, we do not revert all the simplification changes made by the Preprocessor. This can lead to a linguistically poor summary, where the same word or name is repeated multiple times, rather than using synonyms or personal pronouns.

Worse than this, we can end up with sentences that would never be written by a human. Because the Preprocessor moves all adverbs to the end of the sentence in which they appear, and is quite eager to homogenize synonyms, summaries generated by SumASG* may end up "sounding wrong".

6.2 Better Summary Selection

Another issue is that we can easily end up with a very large list of summaries. Because the mechanism used to score them is not very advanced, it cannot say for sure that one particular summary is better than all the others. Instead, we usually end up with multiple entries that all have the same maximum score.

We would therefore need to build much more intelligence into this system if we wanted the program to always return a single summary, one that is humans would also consider optimal.

Chapter 6 Evaluation

1 General Idea

As the vast majority of modern text summarization frameworks are based on machine learning, it makes sense to compare the performance SumASG* with that of a neural network.

More specifically, we should generate a set of stories which we can give to our framework in order to obtain corresponding summaries. We can then use this as training data for an encoder-decoder, to see if it is able to learn how SumASG* creates summaries.

If the neural network is able to learn to generate similar summaries, then we can consider our framework to be sane.

2 Story Generation

2.1 Libraries

For this task, we have chosen to use a library called **Pattern**, which allows us to conjugate verbs, as well as toggle nouns between singular and plural.

We also take advantage of the **Datamuse API**, which lets us find words which are semantically related to a given word in a certain way.

2.2 Datasets

In order to generate the required number of stories, we have used words from word-frequency.info. This database contains 5,000 individual English words, of which 1,001 are verbs, 2,542 nouns and 839 adjectives.

For each story we chose a noun from our dataset, which we shall refer to as the *topic*. We then construct four sentences which revolve around this *topic*.

2.3 Sentence Generation

We will begin by detailing how each sentence is generated, starting with a few necessary definitions. Throughout this section, it is important to keep in mind that the goal here is to create a story that is as lexically and semantically coherent as possible, which is tricky to do algorithmically.

2.3.1 Definitions

Definition 15 (Hyponym)**.** A *hyponym* is a word with more specific meaning than another word; "computer" is a *hyponym* of "machine".

Definition 16 (Hypernym)**.** A *hypernym* is a semantic superclass of a word; "vehicle" is a *hypernym* of "bus".

Definition 17 (Holonym)**.** A *holonym* of something is one of its constituents; "lightbulb" is a *holonym* of "lamp".

Definition 18 (Meronym)**.** A *meronym* is an object which something is part of; "house" is a *meronym* of "kitchen".

2.3.2 Lexical Common Nouns

Along with the story's *topic*, we also generate a set of *lexical common nouns*. If the **Datamuse API** is able to find *synonyms* of our *topic* which also belong to our dataset of nouns, then these become the story's *lexical common nouns*.

In addition, we query from the **Datamuse API** for verbs that are related to the chosen *topic*. This becomes our set of *lexical verbs*. If it is empty, then we make it the singleton set containing the verb "to be".

Since we don't know how general or specific this randomly selected *topic* is, we may not find any. In this case, we try the same procedure for *hypernyms* and finally *hyponyms*. If we still are unable to find any (which is very rare), then we pick a new random *topic*.

2.3.3 Subject

For the *subject* of a sentence, we draw a noun from our *lexical common nouns*.

If this word is singular, then we need a determiner, which can be either "the" or "a".

We also ask the **Datamuse API** to find us an adjective which is often modified by the chosen *subject* noun, and is part of our dataset of words. If none is found, then we do not need to use an adjective.

2.3.4 Verb

We chose a verb at random from our set of *lexical verbs*, conjugating it in the past tense so that it agrees with the sentence's *subject*.

2.3.5 Object

For the *object* of our sentence, we look at the *subject* and *verb*. Using the **Datamuse API** we try and find a noun which often appears after the chosen *verb*, and which is related to our *topic* as well as all nouns we have used thus far in the story. With 50% probability we ask it to be a *holonym* of the *subject* noun, otherwise it should be a *meronym*.

In the same way as we did for the *subject*, we try and find an adjective often modified by the chosen noun. Sometimes it will be the case that no noun was found, but it is possible in English to have an adjective as the only word in the *object*.

The determiner is added as for the *subject*; if there is no noun we do not use one.

2.3.6 Example

We take the example of generating a sentence for a story whose *topic* is "soccer". In this case, the *lexical common nouns* are a singleton set containing the word "football". Here also have two *lexical verbs*: "to match" and "to pitch".

For the *subject*, we can choose the *lexical common noun* "football"; an adjective commonly used to modify it is "professional". Since the noun here is singular, we can use the determiner "a".

We can then pick "to pitch" as the *verb*, which becomes "pitched" when conjugated in the past tense.

For the *object*, we take into account our *topic* "soccer", to find a *holonym* of "football" which often appears after the *verb* "pitched". In this case the **Datamuse API** returns the word "reception", resulting in the adjective "warm" being chosen to accompany it.

After repeating this process three more times, we end up with the below story. As you can see, SumASG$_2$ would be able to combine the two last sentences, but the result of this may or may not make it into the summary we choose.

> The professional football matched a place. A professional football pitched the warm reception. A professional football matched a warm reception. A professional football matched a wonder.

2.4 Action Creation

Using a Python script, we generate the corresponding *actions* as would SumASG$_1$, creating the necessary additional leaf nodes for our general grammar in ASG. We do not use SumASG$_1$ to do this mainly for performance reasons, but also as it is not necessary. Because of the way in which we have created our stories, *simplification* would not change the sentence structure whatsoever, and no sentences would be considered irrelevant (or off-topic) by the Preprocessor.

2.5 Summary Generation

For each story, we feed the generated *actions* and leaf nodes directly into SumASG$_2$, skipping the first half of the SumASG* pipeline. After *scoring*, we pick an entry at random from the *top summaries*.

3 Neural Network

3.1 Datasets

Using the mechanism described above, we generate a number of story/summary pairs: 1796 to be used for training, 199 for validation and 5 for testing.

3.2 Tools

To allow for greater flexibility, we have chosen to use a highly versatile open-source framework called **OpenNMT-py** for training our neural network.

In addition, we preprocess the data using Stanford's **GloVe** pre-trained word embeddings, giving our network a head-start when it comes to semantics.

3.3 Encoder-Decoder Architecture

Our *encoder* and *decoder* share embeddings for a vocabulary of size 7295, which is internally represented using a vector of size 250. They both use a two-layer LSTM with *dropout* of 0.25. Additionally, our *decoder* uses *global attention*.

3.4 Training

The neural network was trained using an Adam optimizer with a *learning rate* of 0.0005 and *batch size* of 50. This was done over a period of 10,000 steps (i.e., 200 epochs), validating every 10 epochs. It took 39 epochs for training accuracy to reach 99%, while validation accuracy slowly increased over time to reach about 53% at the end.

3.5 Results

At first glance, it may seem as though this *encoder-decoder* was overfitting the training data. However, it is important to keep in mind that multiple valid summaries may exist for a given input story, and that the target summary was not necessarily the same as the one generated by the network.

Unfortunately, due to framework constraints, it is impossible to provide the training script with multiple target summaries. However, it is simple to run the trained network on our test stories, and then compare the results to the many summaries generated by SumASG*.

1.
2.
3.
4.

(a) *Test stories*

1.
2.
3.
4.

(b) *Summaries generated by the encoder-decoder*

(c) Maximum BLEU scores between *encoder-decoder* summary and SumASG* summaries

Figure 6.1: TODO

TODO

4 Takeaways

After having done this experiment, we can make the following assessments:

- Using a state-of-the-art neural network architecture, it is possible to learn some of the summarization rules that have been programmed into SumASG*.
- A neural network needs vast amounts of data and some time to learn an approximation to what is a summary, whereas in SumASG the definition of summary is built-in.
- A neural network needs to be trained again for each new summarization rule we would like it to learn, while SumASG* can be used directly after expanding it.
- The coherence of summaries produced by a neural network is extremely tied to the nature and diversity of the training corpus, whereas SumASG* performs similarly on all stories whose structure it can parse.
- While SumASG* always uses information from the story to construct its summary, a summary from a neural network can sometimes include irrelevant words or the unknown *token*.
- A neural network will always produce an output sequence regardless of whether the input is valid English, whereas SumASG* will ignore anything that it was not programmed to understand.

TODO make into a table?

Chapter 7 Literature Review

1 Summarization Levels

Depending on how much text analysis is done, we identify three different levels of summarization [3]. 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 [16], 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 7.1, 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.

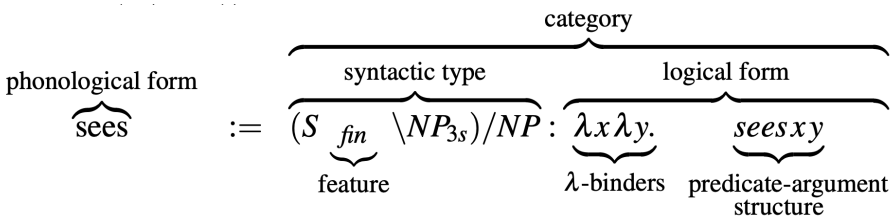
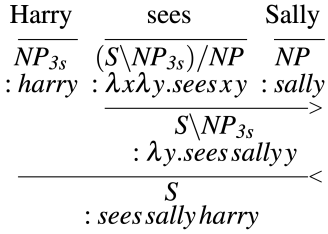


Figure 7.1: [16] Diagram explaining the mains terms used in CCG

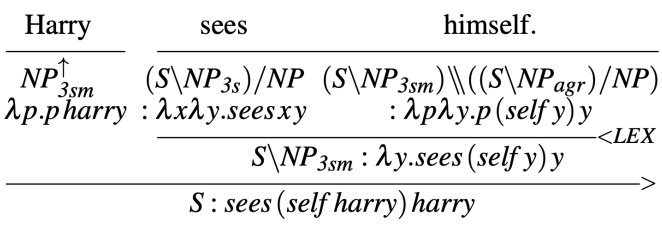
In the *syntactic type* of a word, the forward slash / indicates forward application to combine terms, while \ indicates backward combination. If there is no slash, then the expression can be thought of as a clause, and it can combine with any rule.

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

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 7.2.



(a) [16] Basic clause



(b) [16] Reflexive transitive clause

Figure 7.2: 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. [17], 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) [18], 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.

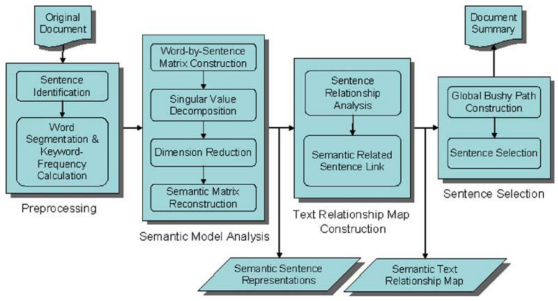
On the other hand, the LSA+TRM approach comprises four major steps: *preprocessing* (1), *semantic model analysis* (2), *text relationship map construction* (3) and *sentence selection* (4).

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

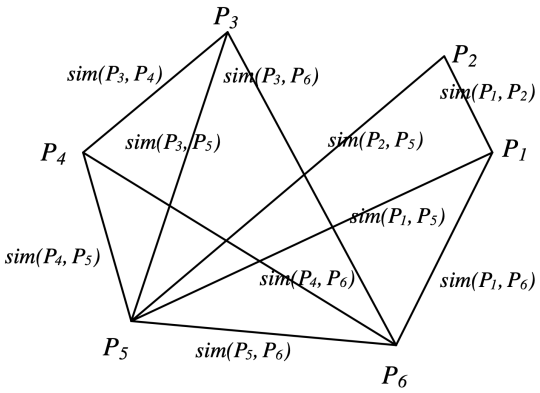
In step (2), a *word-by-sentence matrix* is computed on the scale of the entire document (or corpus). This gets factorized and reduced to leave out words which do not occur often, then turned into a *semantic matrix* linking words to their according relevance with each sentence.

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 7.3 may help you visualize how this works.



(a) [17] Overall process



(b) [17] Example of a text relationship map

Figure 7.3: 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

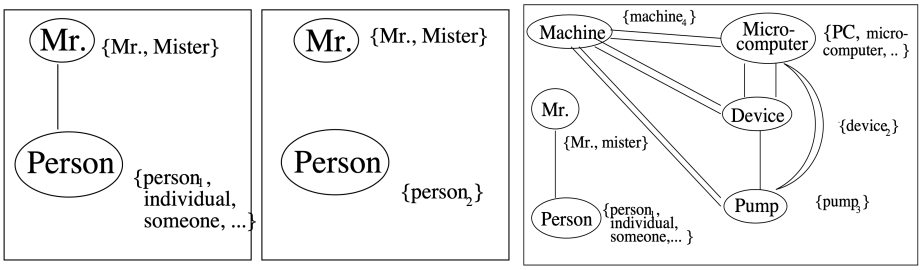
In a paper about *lexical chains* [19], the authors describe a method which relies on semantic links between words. The idea is that we establish chains of related words, in order to learn what a text is about.

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 7.4.

> **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. [19]

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 7.4 (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 7.4: [19] 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 [20] 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

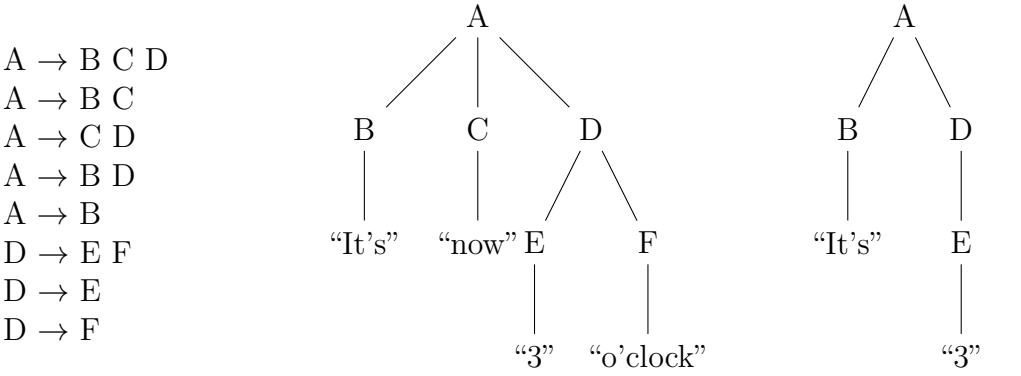
$P(s)$ to less grammatically correct strings, then it helps ensure that our final summary is well-formed.

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 7.5. 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 7.5: 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 [21], 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 7.6, 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).

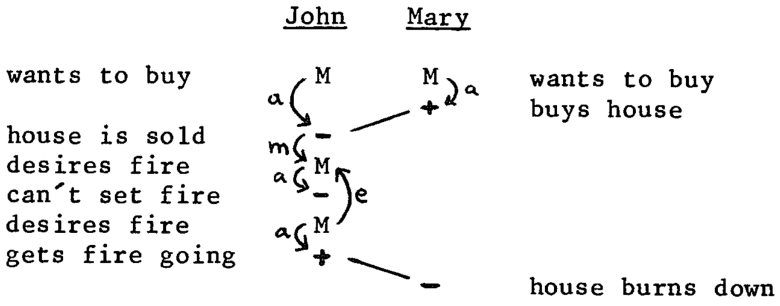


Figure 7.6: Example of a narrative in the frame approach

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 [22], 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 [23]. 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 2.3), 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 [22] 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.

Chapter 8 **Conclusion**

1 Achievements

SumASG* is a symbolic system which is able to generate *generic*, *informative* and partially-*abstractive* summaries given a simple story about a paragraph in length.

Internally, it relies on the ASG engine, which is used for both for understanding text and creating summary sentences. This is an *entity level* approach to the task of text summarization, whereby the Preprocessor makes use of the *similarity* between words and sentences, and creates a *text relationship map* to aid in simplifying the given input story.

The core accomplishments of this project are the following:

- Created a context-free grammar that models the structure of basic English sentences, and can be used both for semantic learning, as well as generating grammatically-correct text.
- Implemented an algorithm that dramatically reduces the complexity in the structure of English sentences, without losing too much information.
- Implemented an algorithm which uses *similarity* to remove irrelevant sentences from a short story.
- Implemented a scoring mechanism prioritizing information density, while taking into account words which may appear frequently in English, can be considered as the *topic* of the original text.
- Created a framework to automatically generate topical short stories to evaluate SumASG*, based on a dataset of words and particular sentence structure.

2 Future Work

Text summarization is a highly involved task in NLP, bringing together many different fields of study. For this reason, there any many ways in which we could take the overall pipeline forward, the most beneficial of which we shall discuss in what follows. Although the ideas which follow may seem rather involved, it is easy to reduce them to a more manageable task.

2.1 Better Semantic Understanding

By way of the Preprocessor, SumASG* is able to transform complex sentence structures into simple ones. Unfortunately, this comes at the cost of removing connectors, as well many auxiliary clauses whose structure SumASG cannot parse.

By doing so we can lose some information, or worse: convey a false meaning due to the intricacy of English semantics. To illustrate this consider the following story, as shown in Figure 8.1.

> John and Mary are siblings. Today is Monday 25 May. Unless it rains in London, which is highly likely, John's sister is going running and then buying a brioche feuilletée aux pralines roses tomorrow. However she doesn't know that her favorite bakery doesn't make pastries on Tuesdays.

(a) Story

> Unless is rains, Mary is going running and then buying a pastry on Tuesday. However she doesn't know that her favorite bakery doesn't make pastries on that day of the week.

(b) "Ideal" partially-*abstractive* and *informative* summary

Figure 8.1: Example of a short story with complex semantics

With the way things are currently set up, the PREPROCESSOR might remove the second sentence, getting rid of all useful temporal information from the story. It could also remove the last sentence, which is crucial in the narrative. Finally, it would definitely delete the first two clauses from the third sentence, missing out on an important *nuance*.

Even if all this information had been preserved and passed through to SUMASG, it would be unable to understand it in a contextually-aware enough manner. More to the point, the following facts are relevant to create a good summary:

- John and Mary are siblings, so "John's sister" refers to Mary.
- The current day of the week is Monday, so Mary is thinking of going running on a Tuesday.
- It is probably going to rain on Tuesday, so there is a slim chance Mary will carry out her plans.
- The bakery where she was planning on getting her pastry not be selling any that day (inference is necessary here).

Even though they are essential to comprehend the story, there is also a set of facts that should not appear in the summary:

- John is Mary's brother.
- The current date is 25 May.
- Mary is in London.
- She is interested specifically in a brioche feuilletée aux pralines roses.

In order to understand such a story, we would therefore need to strengthen SUMASG* at each step in the pipeline. As well as much better parsing, this would require creating a more complex set of predicates to better capture the meaning in a story. Finally, we would also have the change the *scoring* mechanism, so that it looks for summaries which lose as little meaning as possible from the original story.

2.2 Longer Stories

What we would like it to use this mechanism to summarize longer texts, such as newspaper articles or even whole books. Using a supercomputer, we could run a much more advanced and polished version of SumASG* in order to generate a summary of one or more pages in length.

As we have seen, runtime is one of the major bottlenecks of SumASG, which is why SumASG* is limited to very succinct stories. What we would therefore need to do is to carefully reason about the most efficient implementation of our logic program. This could involve separately running SumASG* on each paragraph or page, and then gathering the results together to construct the final summary.

2.3 Domain-Specific Understanding

There are many domains where a certain background knowledge is assumed, such as research papers in Computer Science, or scripts for plays. With an enhanced version of SumASG*, we could help authors by automating (or at least partially) their respective tasks of writing abstracts and loglines. A way to accomplish this would be to create a suite of *extensions*, each providing background knowledge to help understand a particular subject.

In the case of reading a paper, the *extension* would include the relevant encyclopedic knowledge translated into logic. By combining this with the information contained in the paper, a machine would be able to understand what the author is talking about.

When it comes to understanding a play, we would need encode into the parsing mechanism the difference in format between reading narrative and dialog (from various characters). By carefully keeping track of the timeline, and using *action* predicates that know who is speaking, we would be able to programmatically learn about the evolution of the characters in the story.

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: [24] List of position of speech (POS) tags

Appendix B. **Example Stories**

Appendix C. **ASG**

Although SumASG$_1$ and SumASG$_2$ share the same grammar, they need to augment a few of its derivations with some extra rules. The code that you see in Sections 2 and 3 gets appended to the general grammar, giving the complete ASG program.

1 Common Grammar

```
start -> s_group {
    :- count(X)@1, X > 1.
    :- count(X)@1, X = 0.
}

s_group -> { count(0). }
s_group -> s_group s ". " { count(X+1) :- count(X)@1. }

s -> np vp {
  subject :- subject(S_N,S_D,S_A)@1.
  :- not subject.
  object :- object(S_N,S_D,S_A)@2.
  :- not object.
}

vp -> vbn np {
  verb(N,T) :- verb(N,T)@1.
  object(N,D,A) :- object(N,D,A)@2.
}

vp -> vbd np {
  verb(N,T) :- verb(N,T)@1.
  object(N,D,A) :- object(N,D,A)@2.
}

vp -> vbd vbg np {
  verb(comp(N1,N2),comp(T1,gerund)) :- verb(N1,T1)@1, verb(N2,gerund)@2.
  object(N,D,A) :- object(N,D,A)@3.
}

vp -> vbd vbn np {
  verb(comp(N1,N2),comp(T1,past_part)) :- verb(N1,T1)@1, verb(N2,past_part)
      ↪ @2.
  object(N,D,A) :- object(N,D,A)@3.
}
```

```
35
36 vp -> vbd “to ” vb np {
37   verb(comp(N1,N2),comp(T1,base)) :- verb(N1,T1)@1, verb(N2,base)@3.
38   object(N,D,A) :- object(N,D,A)@4.
39 }
40
41 vp -> vbp np {
42   verb(N,T) :- verb(N,T)@1.
43   object(N,D,A) :- object(N,D,A)@2.
44 }
45
46 vp -> vbp vbg np {
47   verb(comp(N1,N2),comp(T1,gerund)) :- verb(N1,T1)@1, verb(N2,gerund)@2.
48   object(N,D,A) :- object(N,D,A)@3.
49 }
50
51 vp -> vbp vbn np {
52   verb(comp(N1,N2),comp(T1,past_part)) :- verb(N1,T1)@1, verb(N2,past_part)
       ↪ @2.
53   object(N,D,A) :- object(N,D,A)@3.
54 }
55
56 vp -> vbp “to ” vb np {
57   verb(comp(N1,N2),comp(T1,base)) :- verb(N1,T1)@1, verb(N2,base)@3.
58   object(N,D,A) :- object(N,D,A)@4.
59 }
60
61 vp -> vbz np {
62   verb(N,T) :- verb(N,T)@1.
63   object(N,D,A) :- object(N,D,A)@2.
64 }
65
66 vp -> vbz vbg np {
67   verb(comp(N1,N2),comp(T1,gerund)) :- verb(N1,T1)@1, verb(N2,gerund)@2.
68   object(N,D,A) :- object(N,D,A)@3.
69 }
70
71 vp -> vbz vbn np {
72   verb(comp(N1,N2),comp(T1,past_part)) :- verb(N1,T1)@1, verb(N2,past_part)
       ↪ @2.
73   object(N,D,A) :- object(N,D,A)@3.
74 }
75
76 vp -> vbz “to ” vb np {
77   verb(comp(N1,N2),comp(T1,base)) :- verb(N1,T1)@1, verb(N2,base)@3.
78   object(N,D,A) :- object(N,D,A)@4.
79 }
80
```

```
np −> np rb {
  object(N,D,A) :− object(N,D,0)@1, adj_or_adv(A)@2.
}

np −> np rb {
  object(N,D,conjunct(A1,A2)) :− object(N,D,A1)@1, adj_or_adv(A2)@2.
  :− object(N,D,conjunct(A,A)).
}

np −> np rp {
  object(N,D,A) :− object(N,D,0)@1, adj_or_adv(A)@2.
}

np −> np rp {
  object(N,D,conjunct(A1,A2)) :− object(N,D,A1)@1, adj_or_adv(A2)@2.
  :− object(N,D,conjunct(A,A)).
}

np −> nn {
  subject(N,0,0) :− noun(N)@1.
  object(N,0,0) :− noun(N)@1.
}

np −> nns {
  subject(N,0,0) :− noun(N)@1.
  object(N,0,0) :− noun(N)@1.
}

np −> nnp {
  subject(N,0,0) :− noun(N)@1.
  object(N,0,0) :− noun(N)@1.
}

np −> nnps {
  subject(N,0,0) :− noun(N)@1.
  object(N,0,0) :− noun(N)@1.
}

np −> prp {
  subject(N,0,0) :− noun(N)@1.
  object(N,0,0) :− noun(N)@1.
}

np −> rb {
  subject(0,0,A) :− adj_or_adv(A)@1.
  object(0,0,A) :− adj_or_adv(A)@1.
}

```

```
np -> rp {
  subject(0,0,A) :- adj_or_adv(A)@1.
  object(0,0,A) :- adj_or_adv(A)@1.
}

np -> ex {
  subject(N,0,0) :- noun(N)@1.
}

np -> in {
  object(0,D,0) :- det(D)@1.
}

np -> prp ‘‘and ” nnp {
  subject(conjunct(N1,N2),0,0) :- noun(N1)@1, noun(N2)@3.
  object(conjunct(N1,N2),0,0) :- noun(N1)@1, noun(N2)@3.
  :- subject(conjunct(N,N),0,0).
  :- object(conjunct(N,N),0,0).
}

np -> nnp ‘‘and ” prp {
  subject(conjunct(N1,N2),0,0) :- noun(N1)@1, noun(N2)@3.
  object(conjunct(N1,N2),0,0) :- noun(N1)@1, noun(N2)@3.
  :- subject(conjunct(N,N),0,0).
  :- object(conjunct(N,N),0,0).
}

np -> dt nn ‘‘and ” prp {
  subject(conjunct(N1,N2),D,0) :- det(D)@1, noun(N1)@2, noun(N2)@4.
  object(conjunct(N1,N2),D,0) :- det(D)@1, noun(N1)@2, noun(N2)@4.
  :- subject(conjunct(N,N),_,0).
  :- object(conjunct(N,N),_,0).
}

np -> prp ‘‘and ” dt nn {
  subject(conjunct(N1,N2),D,0) :- noun(N1)@1, det(D)@3, noun(N2)@4.
  object(conjunct(N1,N2),D,0) :- noun(N1)@1, det(D)@3, noun(N2)@4.
  :- subject(conjunct(N,N),_,0).
  :- object(conjunct(N,N),_,0).
}

np -> dt nn ‘‘and ” nnp {
  subject(conjunct(N1,N2),D,0) :- det(D)@1, noun(N1)@2, noun(N2)@4.
  object(conjunct(N1,N2),D,0) :- det(D)@1, noun(N1)@2, noun(N2)@4.
  :- subject(conjunct(N,N),_,0).
  :- object(conjunct(N,N),_,0).
}
```

```
177 np -> nnp ‘‘and ” dt nn {
178   subject(conjunct(N1,N2),D,0) :- noun(N1)@1, det(D)@3, noun(N2)@4.
179   object(conjunct(N1,N2),D,0) :- noun(N1)@1, det(D)@3, noun(N2)@4.
180   :- subject(conjunct(N,N),_,0).
181   :- object(conjunct(N,N),_,0).
182 }
183
184 np -> nnp ‘‘and ” nnp {
185   subject(conjunct(N1,N2),0,0) :- noun(N1)@1, noun(N2)@3.
186   object(conjunct(N1,N2),0,0) :- noun(N1)@1, noun(N2)@3.
187   :- subject(conjunct(N,N),0,0).
188   :- object(conjunct(N,N),0,0).
189 }
190
191 np -> nn ‘‘and ” nn {
192   subject(conjunct(N1,N2),0,0) :- noun(N1)@1, noun(N2)@3.
193   object(conjunct(N1,N2),0,0) :- noun(N1)@1, noun(N2)@3.
194   :- subject(conjunct(N,N),0,0).
195   :- object(conjunct(N,N),0,0).
196 }
197
198 np -> nn ‘‘and ” nns {
199   subject(conjunct(N1,N2),0,0) :- noun(N1)@1, noun(N2)@3.
200   object(conjunct(N1,N2),0,0) :- noun(N1)@1, noun(N2)@3.
201   :- subject(conjunct(N,N),0,0).
202   :- object(conjunct(N,N),0,0).
203 }
204
205 np -> nns ‘‘and ” nn {
206   subject(conjunct(N1,N2),0,0) :- noun(N1)@1, noun(N2)@3.
207   object(conjunct(N1,N2),0,0) :- noun(N1)@1, noun(N2)@3.
208   :- subject(conjunct(N,N),0,0).
209   :- object(conjunct(N,N),0,0).
210 }
211
212 np -> nns ‘‘and ” nns {
213   subject(conjunct(N1,N2),0,0) :- noun(N1)@1, noun(N2)@3.
214   object(conjunct(N1,N2),0,0) :- noun(N1)@1, noun(N2)@3.
215   :- subject(conjunct(N,N),0,0).
216   :- object(conjunct(N,N),0,0).
217 }
218
219 np -> dt nn ‘‘and ” dt nn {
220   subject(conjunct(N1,N2),D,0) :- det(D)@1, det(D)@4, noun(N1)@2, noun(N2)
        ↪ @5.
221   object(conjunct(N1,N2),D,0) :- det(D)@1, det(D)@4, noun(N1)@2, noun(N2)
        ↪ @5.
222   :- subject(conjunct(N,N),_,0).
```

```
223   :- object(conjunct(N,N),_,0).
224 }
225
226 np -> prp ‘‘and ’’ prp {
227   subject(conjunct(N1,N2),0,0) :- noun(N1)@1, noun(N2)@3.
228   object(conjunct(N1,N2),0,0) :- noun(N1)@1, noun(N2)@3.
229   :- subject(conjunct(N,N),0,0).
230   :- object(conjunct(N,N),0,0).
231 }
232
233 np -> rb ‘‘and ’’ rb {
234   subject(0,0,conjunct(A1,A2)) :- adj_or_adv(A1)@1, adj_or_adv(A2)@3.
235   object(0,0,conjunct(A1,A2)) :- adj_or_adv(A1)@1, adj_or_adv(A2)@3.
236   :- subject(0,0,conjunct(A,A)).
237   :- object(0,0,conjunct(A,A)).
238 }
239
240 np -> jj {
241   object(0,0,A) :- adj_or_adv(A)@1.
242 }
243
244 np -> jj ‘‘and ’’ jj {
245   object(0,0,conjunct(A1,A2)) :- adj_or_adv(A1)@1, adj_or_adv(A2)@3.
246   :- object(0,0,conjunct(A,A)).
247 }
248
249 np -> jj rb {
250   subject(0,0,conjunct(A1,A2)) :- adj_or_adv(A1)@1, adj_or_adv(A2)@1.
251   object(0,0,conjunct(A1,A2)) :- adj_or_adv(A1)@1, adj_or_adv(A2)@1.
252   :- subject(0,0,conjunct(A,A)).
253   :- object(0,0,conjunct(A,A)).
254 }
255
256 np -> dt nn {
257   subject(N,D,0) :- det(D)@1, noun(N)@2.
258   object(N,D,0) :- det(D)@1, noun(N)@2.
259 }
260
261 np -> dt nns {
262   subject(N,D,0) :- det(D)@1, noun(N)@2.
263   object(N,D,0) :- det(D)@1, noun(N)@2.
264 }
265
266 np -> jj nns {
267   subject(N,0,A) :- adj_or_adv(A)@1, noun(N)@2.
268   object(N,0,A) :- adj_or_adv(A)@1, noun(N)@2.
269 }
270
```

```
np -> jj nnp {
  subject(N,0,A) :- adj_or_adv(A)@1, noun(N)@2.
  object(N,0,A) :- adj_or_adv(A)@1, noun(N)@2.
}

np -> jj nnps {
  subject(N,0,A) :- adj_or_adv(A)@1, noun(N)@2.
  object(N,0,A) :- adj_or_adv(A)@1, noun(N)@2.
}

np -> dt jj nn {
  subject(N,D,A) :- det(D)@1, adj_or_adv(A)@2, noun(N)@3.
  object(N,D,A) :- det(D)@1, adj_or_adv(A)@2, noun(N)@3.
}

np -> dt jj nns {
  subject(N,D,A) :- det(D)@1, adj_or_adv(A)@2, noun(N)@3.
  object(N,D,A) :- det(D)@1, adj_or_adv(A)@2, noun(N)@3.
}

np -> dt jj jj nn {
  subject(N,D,conjunct(A1,A2)) :- det(D)@1, adj_or_adv(A1)@2, adj_or_adv(A2)
      ↪ @3, noun(N)@4.
  object(N,D,conjunct(A1,A2)) :- det(D)@1, adj_or_adv(A1)@2, adj_or_adv(A2)
      ↪ @3, noun(N)@4.
  :- subject(N,D,conjunct(A,A)).
  :- object(N,D,conjunct(A,A)).
}

np -> dt jj jj nns {
  subject(N,D,conjunct(A1,A2)) :- det(D)@1, adj_or_adv(A1)@2, adj_or_adv(A2)
      ↪ @3, noun(N)@4.
  object(N,D,conjunct(A1,A2)) :- det(D)@1, adj_or_adv(A1)@2, adj_or_adv(A2)
      ↪ @3, noun(N)@4.
  :- subject(N,D,conjunct(A,A)).
  :- object(N,D,conjunct(A,A)).
}

np -> dt jjr nn {
  subject(N,D,A) :- det(D)@1, adj_or_adv(A)@2, noun(N)@3.
  object(N,D,A) :- det(D)@1, adj_or_adv(A)@2, noun(N)@3.
}

np -> dt jjr nns {
  subject(N,D,A) :- det(D)@1, adj_or_adv(A)@2, noun(N)@3.
  object(N,D,A) :- det(D)@1, adj_or_adv(A)@2, noun(N)@3.
}

```

```
315 np −> dt jjs nn {
316   subject(N,D,A) :− det(D)@1, adj_or_adv(A)@2, noun(N)@3.
317   object(N,D,A) :− det(D)@1, adj_or_adv(A)@2, noun(N)@3.
318 }
319
320 np −> dt jjs nns {
321   subject(N,D,A) :− det(D)@1, adj_or_adv(A)@2, noun(N)@3.
322   object(N,D,A) :− det(D)@1, adj_or_adv(A)@2, noun(N)@3.
323 }
324
325 np −> in nn {
326   object(N,D,0) :− det(D)@1, noun(N)@2.
327 }
328
329 np −> in dt nn {
330   object(N,conjunct(D1,D2),0) :− det(D1)@1, det(D2)@2, noun(N)@3.
331 }
332
333 np −> in nns {
334   object(N,D,0) :− det(D)@1, noun(N)@2.
335 }
336
337 np −> in dt nns {
338   object(N,conjunct(D1,D2),0) :− det(D1)@1, det(D2)@2, noun(N)@3.
339 }
340
341 np −> in nnp {
342   object(N,D,0) :− det(D)@1, noun(N)@2.
343 }
344
345 np −> in nnps {
346   object(N,D,0) :− det(D)@1, noun(N)@2.
347 }
348
349 np −> jj in nn {
350   object(N,D,A) :− adj_or_adv(A)@1, det(D)@2, noun(N)@3.
351 }
352
353 np −> jj in nn ‘‘and ’’ nn {
354   object(conjunct(N1,N2),D,A) :− adj_or_adv(A)@1, det(D)@2, noun(N1)@3,
        ↪ noun(N2)@5.
355 }
356
357 np −> jj in nns {
358   object(N,D,A) :− adj_or_adv(A)@1, det(D)@2, noun(N)@3.
359 }
360
361 np −> jj in nns ‘‘and ’’ nns {
```

```
362   object(conjunct(N1,N2),D,A) :- adj_or_adv(A)@1, det(D)@2, noun(N1)@3,
          ↪ noun(N2)@5.
363 }
364
365 np -> jj in nnp {
366   object(N,D,A) :- adj_or_adv(A)@1, det(D)@2, noun(N)@3.
367 }
368
369 np -> jj in nnp ‘‘and ” nnp {
370   object(conjunct(N1,N2),D,A) :- adj_or_adv(A)@1, det(D)@2, noun(N1)@3,
          ↪ noun(N2)@5.
371 }
372
373 np -> jj in prp {
374   object(N,D,A) :- adj_or_adv(A)@1, det(D)@2, noun(N)@3.
375 }
376
377 np -> jj in prp ‘‘and ” prp {
378   object(conjunct(N1,N2),D,A) :- adj_or_adv(A)@1, det(D)@2, noun(N1)@3,
          ↪ noun(N2)@5.
379 }
380
381 np -> jj in nn ‘‘and ” nns {
382   object(conjunct(N1,N2),D,A) :- adj_or_adv(A)@1, det(D)@2, noun(N1)@3,
          ↪ noun(N2)@5.
383 }
384
385 np -> jj in nns ‘‘and ” nn {
386   object(conjunct(N1,N2),D,A) :- adj_or_adv(A)@1, det(D)@2, noun(N1)@3,
          ↪ noun(N2)@5.
387 }
388
389 np -> cd nn {
390   subject(N,D,0) :- det(D)@1, noun(N)@2.
391   object(N,D,0) :- det(D)@1, noun(N)@2.
392 }
393
394 np -> cd nns {
395   subject(N,D,0) :- det(D)@1, noun(N)@2.
396   object(N,D,0) :- det(D)@1, noun(N)@2.
397 }
398
399 np -> cd jj nn {
400   subject(N,D,A) :- det(D)@1, adj_or_adv(A)@2, noun(N)@3.
401   object(N,D,A) :- det(D)@1, adj_or_adv(A)@2, noun(N)@3.
402 }
403
404 np -> cd jj nns {
```

```
405   subject(N,D,A) :- det(D)@1, adj_or_adv(A)@2, noun(N)@3.
406   object(N,D,A) :- det(D)@1, adj_or_adv(A)@2, noun(N)@3.
407 }
408
409 np -> cd nns jj {
410   object(N,D,A) :- det(D)@1, noun(N)@2, adj_or_adv(A)@3.
411 }
412
413 np -> dt jj cd {
414   subject(0,conjunct(D1,D2),A) :- det(D1)@1, adj_or_adv(A)@2, det(D2)@3.
415   object(0,conjunct(D1,D2),A) :- det(D1)@1, adj_or_adv(A)@2, det(D2)@3.
416 }
```

2 Task: SumASG$_1$

```
1  s -> np vp {
2    ...
3
4    :- not action(verb(V_N,V_T),subject(S_N,S_D,S_A),object(O_N,O_D,O_A)),
         ↪ verb(V_N,V_T)@2, subject(S_N,S_D,S_A)@1, object(O_N,O_D,O_A)@2.
5  }
6
7  #modeh(action(verb(const(main_verb),const(main_form)), subject(const(noun),
       ↪ const(det),const(adj_or_adv)), object(const(noun),const(det),const(
       ↪ adj_or_adv))):[4].
8  #modeh(action(verb(const(main_verb),const(main_form)), subject(const(noun),
       ↪ const(det),const(adj_or_adv)), object(const(noun),const(det),conjunct(
       ↪ const(adj_or_adv),const(adj_or_adv)))):[4].
9  #modeh(action(verb(const(main_verb),const(main_form)), subject(const(noun),
       ↪ const(det),conjunct(const(adj_or_adv),const(adj_or_adv))), object(const(
       ↪ noun),const(det),const(adj_or_adv))):[4].
10 #modeh(action(verb(const(main_verb),const(main_form)), subject(conjunct(const(
       ↪ noun),const(noun)),const(det),const(adj_or_adv)), object(const(noun),const
       ↪ (det),const(adj_or_adv))):[4].
11 #modeh(action(verb(const(main_verb),const(main_form)), subject(const(noun),
       ↪ const(det),const(adj_or_adv)), object(conjunct(const(noun),const(noun)),
       ↪ const(det),const(adj_or_adv))):[4].
12 #modeh(action(verb(const(main_verb),const(main_form)), subject(const(noun),
       ↪ const(det),const(adj_or_adv)), object(const(noun),conjunct(const(det),
       ↪ const(det)),const(adj_or_adv))):[4].
13 #modeh(action(verb(const(main_verb),const(main_form)), subject(const(noun),
       ↪ const(det),const(adj_or_adv)), object(const(noun),conjunct(const(prep),
       ↪ const(det)),const(adj_or_adv))):[4].
14
15 #modeh(action(verb(comp(const(main_verb),const(aux_verb)),comp(const(
       ↪ main_form),const(aux_form))), subject(const(noun),const(det),const(
       ↪ adj_or_adv)), object(const(noun),const(det),const(adj_or_adv))):[4].
16 #modeh(action(verb(comp(const(main_verb),const(aux_verb)),comp(const(
```

```
    ↪ main_form),const(aux_form))), subject(const(noun),const(det),const(
    ↪ adj_or_adv)), object(const(noun),const(det),conjunct(const(adj_or_adv),
    ↪ const(adj_or_adv))))):[4].
17  #modeh(action(verb(comp(const(main_verb),const(aux_verb)),comp(const(
    ↪ main_form),const(aux_form))), subject(const(noun),const(det),conjunct(
    ↪ const(adj_or_adv),const(adj_or_adv))), object(const(noun),const(det),const(
    ↪ adj_or_adv)))):[4].
18  #modeh(action(verb(comp(const(main_verb),const(aux_verb)),comp(const(
    ↪ main_form),const(aux_form))), subject(conjunct(const(noun),const(noun)),
    ↪ const(det),const(adj_or_adv)), object(const(noun),const(det),const(
    ↪ adj_or_adv)))):[4].
19
20  #bias(":- head(holds_at_node(action(verb(_,_),subject(0,_,_),object(_,_,_)),var__(1))
    ↪ ).").
21  #bias(":- head(holds_at_node(action(verb(_,_),subject(_,_,_),object(0,0,0)),var__(1)
    ↪ )).").
22
23  #bias(":- head(holds_at_node(action(verb(_,_),subject(_,_,_),object(conjunct(V,V),
    ↪ _,_)),var__(1))).").
24  #bias(":- head(holds_at_node(action(verb(_,_),subject(_,_,_),object(conjunct(_,0),_,
    ↪ _)),var__(1))).").
25  #bias(":- head(holds_at_node(action(verb(_,_),subject(_,_,_),object(conjunct(0,_),_,
    ↪ _)),var__(1))).").
26
27  #bias(":- head(holds_at_node(action(verb(_,_),subject(_,_,_),object(_,_,conjunct(V,
    ↪ V))),var__(1))).").
28  #bias(":- head(holds_at_node(action(verb(_,_),subject(_,_,_),object(_,_,conjunct(_
    ↪ ,0))),var__(1))).").
29  #bias(":- head(holds_at_node(action(verb(_,_),subject(_,_,_),object(_,_,conjunct(0,_
    ↪ ))),var__(1))).").
30
31  #bias(":- head(holds_at_node(action(verb(_,_),subject(conjunct(V,V),_,_),object(_,
    ↪ _,_)),var__(1))).").
32  #bias(":- head(holds_at_node(action(verb(_,_),subject(conjunct(_,0),_,_),object(_,_,
    ↪ _)),var__(1))).").
33  #bias(":- head(holds_at_node(action(verb(_,_),subject(conjunct(0,_),_,_),object(_,_,
    ↪ _)),var__(1))).").
34
35  #bias(":- head(holds_at_node(action(verb(_,_),subject(_,_,conjunct(V,V)),object(_,
    ↪ _,_)),var__(1))).").
36  #bias(":- head(holds_at_node(action(verb(_,_),subject(_,_,conjunct(_,0)),object(_,_,
    ↪ _)),var__(1))).").
37  #bias(":- head(holds_at_node(action(verb(_,_),subject(_,_,conjunct(0,_)),object(_,_,
    ↪ _)),var__(1))).").
38
39  #bias(":- head(holds_at_node(action(verb(_,_),subject(_,_,_),object(_,conjunct(V,V
    ↪ ),_)),var__(1))).").
40  #bias(":- head(holds_at_node(action(verb(_,_),subject(_,_,_),object(_,conjunct(_,0),
```

```
    ↪ _)),var__(1))).").
#bias(":- head(holds_at_node(action(verb(_,_),subject(_,_,_),object(_,conjunct(0,_),
    ↪ _)),var__(1))).").

#bias(":- head(holds_at_node(action(verb(_,_),subject(_,_,conjunct(V,_)),object(_,_
    ↪ ,conjunct(V,_))),var__(1))).").
#bias(":- head(holds_at_node(action(verb(_,_),subject(_,_,conjunct(V,_)),object(_,_
    ↪ ,conjunct(_,V))),var__(1))).").
#bias(":- head(holds_at_node(action(verb(_,_),subject(_,_,conjunct(_,V)),object(_,_
    ↪ ,conjunct(V,_))),var__(1))).").
#bias(":- head(holds_at_node(action(verb(_,_),subject(_,_,conjunct(_,V)),object(_,_
    ↪ ,conjunct(_,V))),var__(1))).").

#bias(":- head(holds_at_node(action(verb(comp(V,V),comp(_,past_part)),subject(
    ↪ _,_,_),object(0,0,0)),var__(1))).").

#constant(noun,0).
#constant(det,0).
#constant(adj_or_adv,0).
```

3 Task: SumASG$_2$

```
s -> np vp {
  ...

  summary(0, V, S, O) :- action(_, V, S, O).

  summary(1, verb(V,T), S, object(N2,D,A)) :- action(_, verb(V,T), S, object(N2
      ↪ ,D,_)), action(_, verb(be,T), subject(it,_,_), object(_,_,A)).
  summary(2, verb(be,T), S, object(N,D1,conjunct(A2,A3))) :- action(_, verb(be,
      ↪ T), S, object(N,D1,_)), action(_, verb(be,T), subject(N,D2,A1), object(_,_
      ↪ ,conjunct(A2,A3))).

  summary(3, V, subject(N1,0,0), object(N3,D,conjunct(A1,A2))) :- action(_, V,
      ↪ subject(conjunct(N1,N2),_,_), object(N3,D,A1)), action(_, V, subject(N1,
      ↪ _,_), object(N3,D,A2)).
  summary(4, V, S, object(0,0,conjunct(A1,A2))) :- action(I1, V, S, object(_,_,A1
      ↪ )), action(I2, V, S, object(_,_,A2)), A1 != A2, A1 != 0, A2 != 0, I1 < I2.

  summary(5, V, S, object(conjunct(N1,N2),D,0)) :- action(I1, V, S, object(N1,0,
      ↪ _)), action(I2, V, S, object(N2,D,_)), N1 != N2, N1 != 0, N2 != 0, I1 <
      ↪ I2.
  summary(6, V, S, object(conjunct(N1,N2),D,0)) :- action(I1, V, S, object(N1,D
      ↪ ,_)), action(I2, V, S, object(N2,0,_)), N1 != N2, N1 != 0, N2 != 0, I1 <
      ↪ I2.
  summary(7, V, S, object(conjunct(N1,N2),D,0)) :- action(I1, V, S, object(N1,D
      ↪ ,_)), action(I2, V, S, object(N2,D,_)), N1 != N2, N1 != 0, N2 != 0, I1 <
      ↪ I2.
```

```
15
16  summary(8, V1, subject(N, D1, conjunct(A1, A2)), object(0, 0, A3)) :- action(_
        ↪ , V1, subject(N, D1, A2), object(0, 0, A3)), action(_, V2, subject(_, 0, 0),
        ↪  object(N, D2, A1)), A1 != A3.
17  summary(9, V1, subject(N, D1, conjunct(A1, A2)), object(0, 0, A3)) :- action(_
        ↪ , V1, subject(N, D1, A2), object(0, 0, A3)), action(_, V2, subject(_, 0, 0),
        ↪  object(N, D2, conjunct(A1, _))), A1 != A3.
18  summary(10, V1, subject(N, D1, conjunct(A1, A2)), object(0, 0, A3)) :- action
        ↪ (_, V1, subject(N, D1, A2), object(0, 0, conjunct(A3,_))), action(_, V2,
        ↪ subject(_, 0, 0), object(N, D2, A1)), A1 != A3.
19
20  summary(I, V, S, object(conjunct(N1,N2),D,A)) :- summary(I, V, S, object(
        ↪ conjunct(conjunct(N1,N2),N3),D,A)).
21  summary(I, V, S, object(conjunct(N1,N2),D,A)) :- summary(I, V, S, object(
        ↪ conjunct(N1,conjunct(N2,N3)),D,A)).
22  summary(I, V, S, object(conjunct(N2,N3),D,A)) :- summary(I, V, S, object(
        ↪ conjunct(conjunct(N1,N2),N3),D,A)).
23  summary(I, V, S, object(conjunct(N2,N3),D,A)) :- summary(I, V, S, object(
        ↪ conjunct(N1,conjunct(N2,N3)),D,A)).
24  summary(I, V, S, object(conjunct(N1,N3),D,A)) :- summary(I, V, S, object(
        ↪ conjunct(conjunct(N1,N2),N3),D,A)).
25  summary(I, V, S, object(conjunct(N1,N3),D,A)) :- summary(I, V, S, object(
        ↪ conjunct(N1,conjunct(N2,N3)),D,A)).
26
27  summary(I, V, S, object(N,D,conjunct(A1,A2))) :- summary(I, V, S, object(N,
        ↪ D,conjunct(conjunct(A1,A2),A3))).
28  summary(I, V, S, object(N,D,conjunct(A1,A2))) :- summary(I, V, S, object(N,
        ↪ D,conjunct(A1,conjunct(A2,A3)))).
29  summary(I, V, S, object(N,D,conjunct(A2,A3))) :- summary(I, V, S, object(N,
        ↪ D,conjunct(conjunct(A1,A2),A3))).
30  summary(I, V, S, object(N,D,conjunct(A2,A3))) :- summary(I, V, S, object(N,
        ↪ D,conjunct(A1,conjunct(A2,A3)))).
31  summary(I, V, S, object(N,D,conjunct(A1,A3))) :- summary(I, V, S, object(N,
        ↪ D,conjunct(conjunct(A1,A2),A3))).
32  summary(I, V, S, object(N,D,conjunct(A1,A3))) :- summary(I, V, S, object(N,
        ↪ D,conjunct(A1,conjunct(A2,A3)))).
33
34  % Pick exactly one summary sentence for each applicable rule
35  0{output(I,V,S,O)}1 :- summary(I,V,S,O).
36  :- not output(_,_,_,_).
37
38  :- output(_,verb(V_N,V_T),subject(S_N,S_D,S_A),object(O_N,O_D,O_A)), not
        ↪ verb(V_N,V_T)@2.
39  :- output(_,verb(V_N,V_T),subject(S_N,S_D,S_A),object(O_N,O_D,O_A)), not
        ↪ subject(S_N,S_D,S_A)@1.
40  :- output(_,verb(V_N,V_T),subject(S_N,S_D,S_A),object(O_N,O_D,O_A)), not
        ↪ object(O_N,O_D,O_A)@2.
41  }
```

Bibliography

[1] Law M, Russo A, Bertino E, Broda K, Lobo J. Representing and Learning Grammars in Answer Set Programming. Proceedings of the AAAI Conference on Artificial Intelligence. 2019 Jul;33:2919–2928. Available from: https://aaai.org/ojs/index.php/AAAI/article/view/4147.

[2] Kiyani F, Tas O. A survey automatic text summarization. Pressacademia. 2017 Jun;5(1):205–213. Available from: http://www.pressacademia.org/images/documents/procedia/archives/vol_5/029.pdf.

[3] Lloret E. Text summarization: an overview. Paper supported by the Spanish Government under the project TEXT-MESS (TIN2006-15265-C06-01). 2008;.

[4] Radev DR, Hovy E, McKeown K. Introduction to the Special Issue on Summarization. Computational Linguistics. 2002 Dec;28(4):399–408. Available from: http://www.mitpressjournals.org/doi/10.1162/089120102762671927.

[5] Syntactic Parsing with CoreNLP and NLTK | District Data Labs;. Available from: https://www.districtdatalabs.com/syntax-parsing-with-corenlp-and-nltk.

[6] Studying Ambiguous Sentences;. Available from: https://www.byrdseed.com/ambiguous-sentences/.

[7] Gomez-Rodriguez C, Alonso-Alonso I, Vilares D. How important is syntactic parsing accuracy? An empirical evaluation on rule-based sentiment analysis. Artificial Intelligence Review. 2019 Oct;52(3):2081–2097. WOS:000486256400018.

[8] Kowalski R. Predicate logic as programming language. In: IFIP congress. vol. 74; 1974. p. 569–544.

[9] Lifschitz V. What Is Answer Set Programming?;p. 4.

[10] Scheinberg S. Note on the boolean properties of context free languages. Information and Control. 1960 Dec;3(4):372–375. Available from: https://linkinghub.elsevier.com/retrieve/pii/S0019995860909657.

[11] Cho K, van Merrienboer B, Gulcehre C, Bahdanau D, Bougares F, Schwenk H, et al. Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation. arXiv:14061078 [cs, stat]. 2014 Sep;ArXiv: 1406.1078. Available from: http://arxiv.org/abs/1406.1078.

[12] Gers FA, Schmidhuber J, Cummins F. Learning to Forget: Continual Prediction with LSTM. Neural Computation. 2000 Oct;12(10):2451–2471. Available from: http://www.mitpressjournals.org/doi/10.1162/089976600300015015.

[13] Graves A, Jaitly N, Mohamed Ar. Hybrid speech recognition with Deep Bidirectional LSTM. In: 2013 IEEE Workshop on Automatic Speech Recognition and Understanding. Olomouc, Czech Republic: IEEE; 2013. p. 273–278. Available from: http://ieeexplore.ieee.org/document/6707742/.

[14] Yao K, Zhang L, Du D, Luo T, Tao L, Wu Y. Dual Encoding for Abstractive Text Summarization. IEEE Transactions on Cybernetics. 2018;p. 1–12.

[15] Bahdanau D, Cho K, Bengio Y. Neural Machine Translation by Jointly Learning to Align and Translate. arXiv:14090473 [cs, stat]. 2016 May;ArXiv: 1409.0473. Available from: http://arxiv.org/abs/1409.0473.

[16] Steedman M. Combinatory Categorial Grammar;p. 31.

[17] Yeh JY, Ke HR, Yang WP, Meng IH. Text summarization using a trainable summarizer and latent semantic analysis. Information Processing & Management. 2005 Jan;41(1):75–95. Available from: https://linkinghub.elsevier.com/retrieve/pii/S0306457304000329.

[18] Powers DM. Evaluation: from Precision, Recall and F-measure to ROC, Informedness, Markedness and Correlation. 2011 Dec;Available from: https://dspace.flinders.edu.au/xmlui/handle/2328/27165.

[19] Barzilay R, Elhadad M. Using lexical chains for text summarization. 1997;Available from: https://doi.org/10.7916/D85B09VZ.

[20] Knight K, Marcu D. Statistics-based summarization-step one: Sentence compression. AAAI/IAAI. 2000;2000:703–710.

[21] Lehnert WG. 1980 - Narrative Text Summarization;p. 3.

[22] Clark S. Combining Symbolic and Distributional Models of Meaning;p. 4.

[23] Partee B. Lecture 2. Lambda abstraction, NP semantics, and a Fragment of English. Formal Semantics;p. 11.

[24] Penn Treebank P.O.S. Tags;. Available from: https://www.ling.upenn.edu/courses/Fall_2003/ling001/penn_treebank_pos.html.