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

Thursday 11th 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. 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 to understand the input text after having pre-processed it, the result of which then goes through a number of summarization logic rules, producing a set of possible summaries.

Throughout the development phase, we ran our system on a suite of targeted examples. When the implementation was complete, we successfully trained a neural network to learn the summarization rules used by our framework, thereby validating our approach. We then performed two experiments to evaluate our system, and to establish some of the key differences compared to a neural network.

From this we conclude that although our approach is less general than a neural network, it is able to produce more consistent results and can detect grammatically-incorrect text. Moreover, it does not rely on noisy annotated data, and can be expanded on without the need to be retrained.

Acknowledgements

Firstly, I would to thank my supervisor Prof. Alessandra Russo, as well as my co-supervisor David Tuckey, for the time they have put into this project. Week after week, we have had very interesting discussions in which they have provided their 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 me 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 my four years at Imperial College London. Finally, I am grateful for my family who supported me during my studies.

Contents

Chapter 1 Introduction

In general, the task of summarization in Natural Language Processing (NLP) 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.

1.1 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 [1]. Given a training corpus, along with some careful pre-processing as well as fine-tuning of hyper-parameters and feature extraction functions, such systems are able to produce effective summaries [1].

Among these approaches, one of the most prominent types of neural networks is the *encoder-decoder*. These are commonly used for sequence-to-sequence translation due to their promising performance in NLP tasks [2]. *encoder-decoders* use a fixed-dimension internal representation which can be trained to act as an intermediate between variable-length inputs and outputs, making this approach highly suitable for text summarization. However to learn what is a summary, these systems require tremendous amounts of data and take a long time to train.

In our case, using logic means that we can hard-code the definition of a summary directly into the program, avoiding the problem of randomness and uncertainty that often comes with neural networks. By carefully constructing the structure of our system, 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.

1.2 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 (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 representation should capture as much of the semantics from the original text as possible, and not be limited to a particular domain.

Objective 2 (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 3 (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 that our logic-based mechanism is sound.

1.3 Approach Overview

The approach described in this paper, known as SumASG*, can be diagrammatically represented as a three step pipeline, as seen in Figure 1.1. While the core part of the the project is written in ASG, Python scripts are used to respectively pre-process and post-process the input and output.

Figure 1.1: Main pipeline for SumASG*

As the first step in the pipeline, the Preprocessor plays an essential role. Given an input story, its goal is to simplify the story's sentences into a simpler and more consistent structure, one that will then be easier to parse by ASG. Additionally, the Preprocessor removes irrelevant sentences from the story and reduces it lexical diversity, which helps increase the chances of generating a high-quality summary.

Once the story has been pre-processed, it then goes through a procedure we call SumASG. This revolves around a purpose-built internal representation of English sentences, represented as a tree. The first of two steps, SumASG_1, involves translating sentences from the input story into our internal representation using ASG's learning abilities. From this, SumASG_2 then exploits a number of logic-based rules to generate sentences which may be used to form a summary.

The third and final step in the pipeline 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. Afterwards we assign to each one a score, and only those with the highest scores are returned.

Example

Throughout this paper, we use 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 B, along with the summaries generated by SumASG*.

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) Original story

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

1.4 Contributions

The main contribution of this project to the field of NLP is the creation of an end-to-end, fully-automated 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. Going more into depth, we discuss some more specific contributions in what follows.

Contribution 1 (Identification Of Existing Techniques Used For Summarization)**.** After researching existing state-of-the-art text summarization systems, identified techniques which were beneficial to use for this project (e.g., *text relationship maps*).

Contribution 2 (Complexity Reduction In Some English Sentences)**.** Implemented an algorithm that dramatically reduces the complexity in the structure of certain English sentences, without losing too much information (e.g., co-referencing).

Contribution 3 (Removal Of Irrelevant Sentences And Homogenization)**.** Implemented an algorithm which uses *similarity* to remove irrelevant sentences from a short story, and replaces synonyms with a single representative in each set of synonyms.

Contribution 4 (Representation Of English In ASG)**.** 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.

Contribution 5 (Translation Of English Into ASG)**.** Wrote an ASG learning task capable of taking English text and turning it into a set of chronologically-ordered *actions* which convey in ASP what occurs in the text.

Contribution 6 (Automatic Generation Of Summaries)**.** Developed a set of rules which, given *actions* from a story, allow ASG to generate both *extractive* and *abstractive* summary sentences.

Contribution 7 (Scoring Mechanism)**.** Implemented a scoring mechanism prioritizing information density, while taking into account words which may appear frequently in English and can be considered the *topic* of the original text.

1.5 Paper Structure

In this paper, we begin by discussing some essential background knowledge in Chapter 2. After giving an overview of text summarization, we introduce the notion of *syntactic parsing*. Then, we go over some concepts that will be necessary in order to understand the ASG part of the pipeline. Finally, we briefly mention some Machine Learning structures which we make use of for evaluating our implementation.

In Chapters 3, 4 and 5, we dive into the various steps involved in the pipeline of SumASG*, as outlined in Section 1.3. In addition, we discuss possible technical improvements for each step of the pipeline in Chapter 6.

After this, we validate our approach by training an *encoder-decoder* for the task of summarization in Chapter 7, then show that our system is capable of producing a more consistent output and can detect invalid input out-of-the-box.

In Chapter 8 we explore some of the existing state-of-the-art implementations and differentiate between statistical, frame and symbolic approaches. Where relevant, we discuss which ideas from these approaches we have borrowed for SumASG*, as well as how our approach differs from these implementations.

Finally, we conclude the paper in Chapter 9 by listing the main achievements of this project, as well as giving a high-level overview of future work which may be done to build on top of SumASG*.

Chapter 2 Background

In this chapter we discuss the background on which our work is based. We first present text summarization and outline the different levels on which this can be done (Section 2.1). We then introduce the notion of *syntactic parsing*, which respectively allows us to understand the grammatical structure of a sentence as well as the relations between its constituents (Section 2.2). Then, we explain a number of logic programming concepts which will later be necessary for the description of our implementation (Section 2.3). Finally, we give a brief overview of the Machine Learning concepts we make use of in order to evaluate our system (Section 2.4).

2.1 Summarization

Summarization is a general task which consists of taking the most important information from a passage and rewriting it into a more concise form, using at most half the amount of text [3].

2.1.1 Types Of Summaries

In what follows we discuss different ways in which we can characterise summaries. Firstly, summaries can be grouped into one of the two following categories, depending on how they are built:

- An *extractive* summary is made up of chunks of sentences which are copied word-for-word from the original text.
- An *abstractive* summary is a rewriting of the text's content in a more concise form.

Summaries can focus on different sections or topics of the text, allowing us to differentiate between the two following types:

- *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.

We can also differentiate the purpose of summaries [4] (see example in Figure 2.1):

- *Indicative* summaries aim to outline what a text is about, without going into much detail.
- *Informative* summaries, on the other hand, take the content from an original document and give a shortened version of it.

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.

(a) *Indicative* summary

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.

(b) *Informative* summary

Figure 2.1: Example of *indicative* and *informative* summaries for a news article[1]

2.1.2 Summarization Levels

Depending on the level of detail at which text analysis is done, we identify three different levels of summarization [3]: *surface*, *entity* and *discourse*. Many current systems employ what is called a *hybrid* approach, combining techniques from different summarization levels.

Surface Level

On a *surface level*, little text 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 most frequently. Usually, the sentences in a passage containing the most important information 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 use 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] Article from The Guardian about storm Brendan: https://www.theguardian.com/uk-news/2020/jan/12/storm-brendan-gales-forecast-uk

Entity Level

A more analytic approach can be done at an *entity level*, where we build a model of a document's individual entities (i.e., words or phrases) and see how they relate. Common techniques are:

- *Similarity* between different words or groups of words, 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.

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 analyse 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.

2.2 Syntactic Parsing

Tokenization is the processes of splitting a text into its individual *tokens*, or words. To each of these *tokens* we can assign a position of speech (POS) tag, which tells us what type of word this is (see Appendix A). Additionally, each of these *tokens* is assigned a *lemma*, which is essentially the "neutral" form of a word, be it the singular of a noun or the base form of a verb.

Furthermore, *syntactic parsing* is a technique that reveals the grammatical links between POS tagged *tokens* [5], the result of which can then be visualised in what is called a *parse tree*. This type of *syntactic parsing* is known as *constituency parsing*. An example of such a *parse tree* is shown in Figure 2.2 for one of the first of the two most prominent *syntactic parsers*: **CoreNLP**[2] and **spaCy**[3].

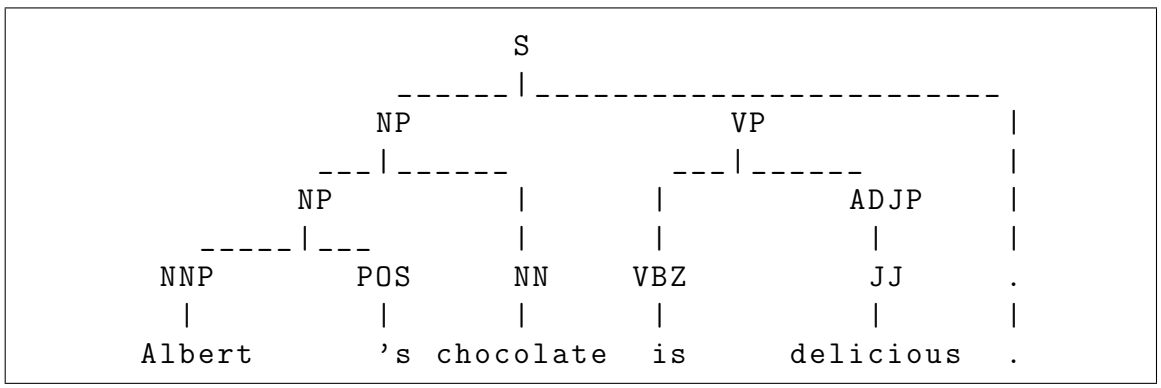


Figure 2.2: *Constituency parse tree* for a basic sentence, as generated by **CoreNLP**

[2]CoreNLP is an NLP toolkit developed by Stanford: https://corenlp.run
[3]spaCy is an NLP toolkit that integrates with deep learning libraries: https://spacy.io

Moreover, many *syntactic parsers* are also capable of *dependency parsing*, which involves linking the *tokens* in a sentence using binary asymmetric relations called *dependencies* [6]. These *dependencies* can help us understand the connections between different *tokens* when visualised in the form of a *parse tree*, as illustrated in Figure 2.3.

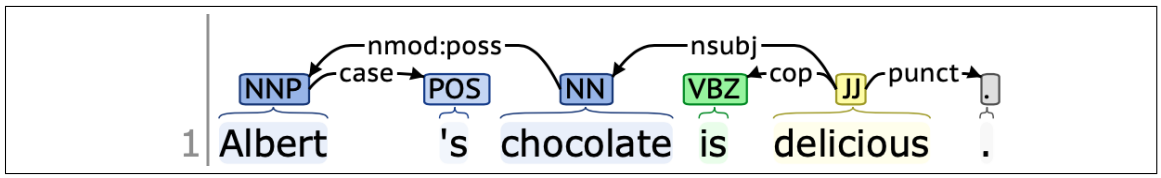


Figure 2.3: *Dependency tree* for a basic sentence, as generated by **CoreNLP**

However, the English language can often be ambiguous and highly context-dependent, meaning that multiple interpretations may be possible, resulting in different *parse trees* for the same sentence. Consider the following two sentences [7]:

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

Depending on the context, we could interpret the first sentence as a person who either fed a woman's cat, fed a woman some cat food, or fed the cat food itself. Although the last meaning does not make much sense, this is one that **CoreNLP** chooses, as shown in Figure 2.4. Therefore, it very important to first accurately perform *syntactic parsing* if we then want to produce a summary which is coherent with the original text [8].

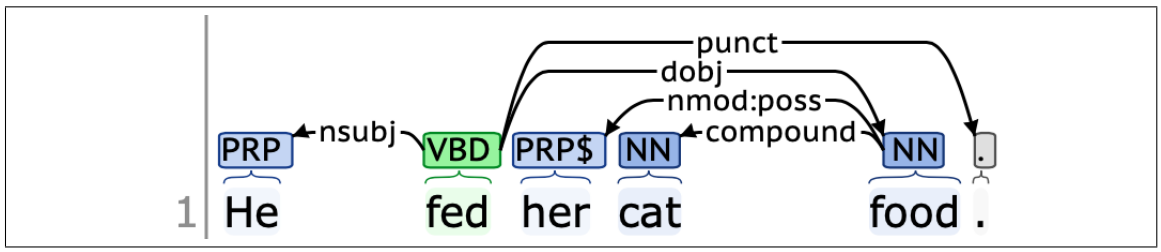


Figure 2.4: *Constituency tree* for an ambiguous sentence, as generated by **CoreNLP**: here, "cat food" is interpreted as a compound entity belonging to "her", erroneously meaning that a man fed a woman's "cat food"

2.3 Answer Set Grammars

Logic programming is a programming paradigm based on formal mathematical logic. It was introduced by R. Kowalski in 1974 and is highly suitable for knowledge representation [9]. Since its introduction, it provided inspiration for a number of languages, such as Answer set programming (ASP).

2.3.1 Answer Set Programming

ASP is a declarative first-order (predicate) logic language whose aim is to solve complex search problems [10]. When asked to solve a problem ASP returns a list of

answer sets (or *stable models*), whose definition (see Definition 5) we will give after having introduced a number of concepts pertinent to logic programming.

Definition 1 (Term [11])**.** 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 [11])**.** 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.

In ASP, a *literal* is an *atom* a or its negation *not* a (we call this "negation as failure"). ASP programs are composed of a set of *normal rules* whose head is a single *atom* and whose body is a conjunction of *literals* [12].

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

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 $b_{k+1}, ..., b_m$.

To compute the *reduct* of a program (see Definition 3), we are first going to need to understand the notion of *Herbrand base*. The *Herbrand base* of a program P, denoted HB_P, is the set of variable-free (*ground*) *atoms* that can be formed from predicates and *constants* in P. The subsets of HB_P are called the (Herbrand) *interpretations* of P [12].

Definition 3 (Reduct [12])**.** Given a program P and a *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 $\{d.\}$.

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 its body is not [12].

Definition 4 (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 5 (Answer Set [12])**.** 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 $AS(P)$.

2.3.2 Answer Set Grammars

A context-free grammar (CFG) is a grammar characterised by a set of *production rules* that describe all possible strings which can be formed by this grammar. Before

discussing ASGs though we must first formally define CFGs (see Definition 6), and introduce the notion of a *parse tree* in the current context (see Definition 7).

Definition 6 (Context-Free Grammar [13])**.** A CFG is a finite set G of *production rules* $\alpha \rightarrow \beta$, 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 $\alpha \rightarrow \alpha$ (which have no effect) are not allowed.

Definition 7 (Parse Tree [12])**.** Let G_{CF} be a CFG. A *parse tree* PT of G_{CF} 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 $node(PT) \rightarrow n_1...n_k$ where each n_i is equal to the node of the ith element in $children(PT)$, and $|children(PT)| = k$.

Definition 8 (Trace [12])**.** 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 ith child of the root is `[i]`; the jth child of the ith child of the root is `[i, j]`, and so on.

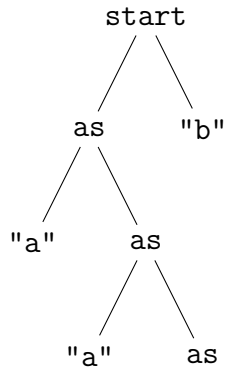
These concepts are illustrated in Figure 2.5, where we have written a set of *production rules* for the grammar $\texttt{a}^\texttt{i}\texttt{b}$ (i.e., strings consisting of any number of "a"s followed by a "b"), along with a *parse tree* for the specific string "aab".

```
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 2.5: Example of a CFG for the grammar $\texttt{a}^\texttt{i}\texttt{b}$

ASGs are an extension of CFGs, whereby each *production rule* is *annotated* (see Definition 10). Using these *annotated production rules*, it is possible to vastly reduce the complexity of the structures that can be produced by a grammar.

Definition 9 (Annotated ASP Program [12])**.** An *annotated* ASP program is an ASP program where some atoms are annotated with a *ground* term. For instance, the *annotated atom* a(1)@2 represents the *atom* a(1) with the annotation 2.

Definition 10 (Annotated Production Rule [12])**.** 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, with every *annotation* being an integer from 1 to k.

In the context of ASGs, the annotations in each *annotated* ASP program refer to index of a node's child. For instance, `a(1)@2` can be read as the truth value of the *term* `a(1)` in the *annotated* ASP program of the second child of the node whose *production rule* we are in.

An example is shown in Figure 2.6, where we have written *annotated production rules* for the language $\texttt{a}^\texttt{n}\texttt{b}$ ($\texttt{n} \geq 2$). By restricting this grammar to strings which contain at least two "a"s, we have effectively captured a subset of the language shown in Figure 2.5.

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

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

Figure 2.6: Example of an ASG for the grammar $\texttt{a}^\texttt{n}\texttt{b}$, where $\texttt{n} \geq 2$

In order to understand the notion of *satisfiability* for a given *parse tree* with respect to an *annotated* grammar, we must first formally define what is a *conforming parse tree* (see Definition 12).

Definition 11 (Parse Tree Program [12])**.** Let G be an ASG and PT be a *parse tree*. $G[PT]$ is the program $\{ \mathit{rule}(n)@\mathit{trace}(n) \mid n \in PT \}$, 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 $a@t$ ($++$ being the concatenation operator).

Definition 12 (Conforming Parse Tree [12])**.** Given a string *str* of terminal nodes, we say that $\mathit{str} \in \mathcal{L}(G)$ (*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*.

The *parse tree program* $G[PT]$ corresponding to the *annotated production rules* from Figure 2.6 and *parse tree* from Figure 2.5 is given in Figure 2.7. This program has a single *answer set* {`size(0)@[1,2,2]`, `size(1)@[1,2]`, `size(2)@[1]`}, confirming that $\texttt{aab} \in \mathcal{L}(G)$. From this example, you can see that the program is *unsatisfiable* for the string `ab`, because from the fact `size(0)@[1,2]` we would end up with `size(1)@[1]` but `1 < 2`.

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

Figure 2.7: $G[PT]$ for the *parse tree* PT of Figure 2.5 and grammar G of Figure 2.6

Learning Answer Set Grammars

Given an incomplete ASG, it is possible to learn the complete grammar (i.e., the missing *production rules*) by induction, which makes use of **ILASP**[4] behind the

[4]ILASP is a logic-based machine learning system: http://www.ilasp.com

scenes. For such a task we must provide some *positive examples* (strings which should conform to the language) and/or *negative examples* (strings which must not), and sometimes *background* information [12]. This *background knowledge* can either serve as context for the program or act as a sort of "helper" (like the predicates `num` and `inc` in Figure 2.8).

In such an *inductive learning program* (ILP) task, we have what is called a *hypothesis space* in the form of *mode declarations*, defining the format of the heads (written `#modeh`) and bodies (written `#modeb`) of *production rules* which can be learned [14]. 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`). This is illustrated in Figure 2.8, where we show a learning task for the grammar that accepts all string of the form $a^n b^n$.

```
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 | shows that multiple *production rules* exist for the same node.

Figure 2.8: Example of an ASG ILP task for the language $a^n b^n$

2.4 Neural Networks

In this section, we present concepts relating to Machine Learning which will later be necessary to understand the evaluation of our project (Chapter 7).

To begin, a neural network is a set of *neurons* organized in *layers*. The first and last *layers* are respectively called the *input layer* and the *output layer*, while all the

intermediate *layers* are known as *hidden layers*.

The role of a *neuron* is to receive input from its immediate predecessors, compute an output signal using a differentiable *activation function* on the weighted sum of its inputs, and then propagate the result to its successors [15]. Some neural connections are stronger then others; these get assigned a higher weight and will thus contribute more to the final output value of the network.

A neural network can be trained to produce a specific output given a certain input. This is done over a certain number of timesteps by feeding some data as input and comparing the predicted output with the expected output. The second phase of each timestep is called *backpropagation*, which involves going backwards from the output and at every *neuron* computing the gradient of what we call a *loss function*, with respect to the weights of that *neuron's* inputs. Using this gradient, we can see which weights need to be increased and which need to be decreased in order to better fit the input data. By repeating this over time, we can eventually find the optimal weights that give us predictions as close as possible to the expected output data [15].

In general, we use a larger dataset for training and a smaller one for what we call validation, which is where we use a different dataset every certain number of timesteps to ensure the network does not overfit the training data. At the end of training, we use an even smaller test dataset to give a final assessment of the neural network. Moreover, training a neural network to perform as well as possible also involves choosing suitable *activation functions*, as well as fine-tuning a number of hyper-parameters (e.g., *learning rate*, number of *hidden layers*, *batch size*...).

2.4.1 Recurrent Neural Networks

A problem with conventional neural networks is that they do not allow variable-length input vectors, making them inadequate for learning sequential data. Recurrent neural networks (RNNs) solve this issue by using an internal state vector that can be updated each time part of the input processed.

To be more precise, an RNN is a chain-like neural network that is applied once for each *token* in the input sequence [16]. 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) [16].

2.4.2 Long Short-Term Memories

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 [17].

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 2.9 shows this diagrammatically [18].

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

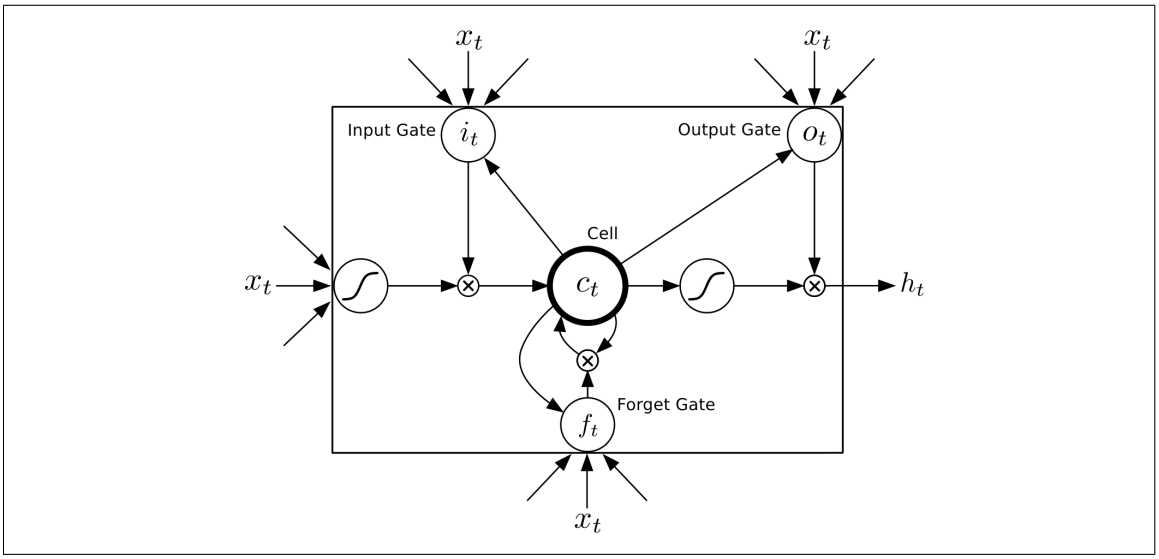


Figure 2.9: [18] Diagram showing the flow of information in an LSTM *memory block*

2.4.3 Encoder-Decoders

An *encoder-decoder* is a neural network consisting of two eponymous RNNs, which 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 [16].

2.4.4 Attention Mechanism

In the context of neural machine translation (NMT) systems such as *encoder-decoders*, the idea 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 [2].

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 those which are more similar to d_t, which is done using a trainable *alignment model* a [19].

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

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* [19].

Chapter 3 Preprocessor

In this chapter, we present the first part of our system's pipeline: the PREPROCESSOR. In order to avoid having to do extremely complicated semantic analysis in SUMASG, we begin by pre-processing the input text so that we can give ASG something which is simpler to understand as well as more computationally tractable.

To ensure a consistent sentence structure for SUMASG, we try and transform every sentence in the story to a basic subject-verb-object form (Section 3.2). To further help with the goal of generating optimal summaries, we take advantage of the PREPROCESSOR's semantic awareness to also remove irrelevant sentences and unnecessary use of synonyms (Section 3.3). We illustrate how the PREPROCESSOR works using our running example of the story of Peter Little (Section 3.4).

3.1 Overview

The different stages of the PREPROCESSOR can be seen in Figure 3.1. The first step is to *tokenize* the story. After this we apply a number of transformations to reduce the complexity of the sentence structures. Once this *simplification* step is complete, we prune away sentences which provide little value, and finally *homogenise* the resulting text so that its lexical diversity is reduced.

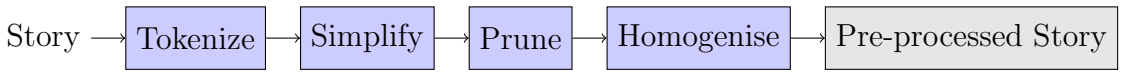


Figure 3.1: Preprocessor Steps

3.2 Tokenization And Simplification

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

3.2.1 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 (i.e., it must consist of only one clause).

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

- Question marks: We remove the clause, as it is very uncommon for questions in a story to contain crucial information. 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.

3.2.2 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.

Acronyms, Contractions And Determiners

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. For instance, the word "U.S.A." becomes "USA".

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

In the English language, the determiners "a" and "an" are semantically identical, so it makes sense to only use one of the two to reduce the number of *tokens* that SUMASG has to process. After generating a summary, we will revert this change so that our framework's output is grammatically correct.

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).

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 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 of the sentence. 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, and there could be a small impact on the quality of the summary, this is a simplification we are willing to make.

3.2.3 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 delete the auxiliary clause to only 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

Hypernym Substitution

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

In practice, this involves using **Pattern**[5] 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".

3.2.4 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

[5]Pattern is a module that is able to perform POS tagging, verb conjugation and noun singularisation, among others: https://www.clips.uantwerpen.be/pages/pattern-en

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

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

3.3 Sentence Pruning And Homogenisation

Once the sentence structure of the story has been *simplified*, one of the main roles 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).

3.3.1 Word Similarity

It iterates 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 **ConceptNet**[6].

Having such nesting of loops is quite expensive, which is why we keep a cache of previously requested similarities, and use the fact that this *similarity* relation is symmetric.

[6]ConceptNet is a semantic knowledge network, providing information about the relations between different words: http://www.conceptnet.io

3.3.2 Sentence Similarity And Pruning

Once we have computed the *similarity* between words of different sentences, we 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* (see Subsection 8.2.1) 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.

Pruning

In the interest of removing irrelevant sentences to help SumASG (see Figure 3.6), we compute the 25th percentile over the *importances* of all the sentences (i.e., the value below which 25% of all *importances* fall). 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.

3.3.3 Synonyms And Homogenisation

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 in the text, we 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 *homogenisation*, and it helps SumASG link words that would otherwise be considered completely different *tokens* in the story.

3.4 Example

To illustrate how all of the steps in the Preprocessor come together to produce a text that will be easy to parse by SumASG, we use our running example of the story of Peter Little, which we have repeated below in Figure 3.4.

> 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.

Figure 3.4: Story of Peter Little

The first half of the PREPROCESSOR's job is to *tokenize* the story of Peter Little and then apply all possible *simplification* transformations, the result of which is outlined in Figure 3.5.

ⓐ 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.

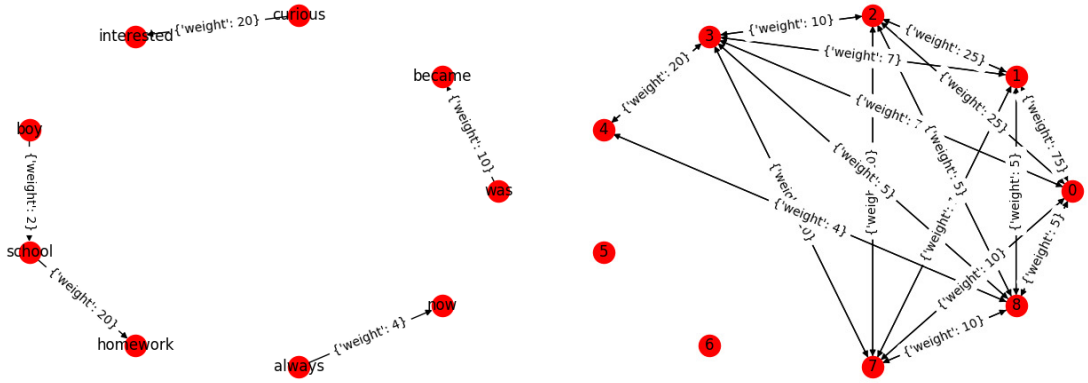
(b) Sentences after applying transformations

Figure 3.5: Example of applying *simplification* to the story of Peter Little

Using the *simplified* form of the story from Figure 3.5, the PREPROCESSOR now applies sentence pruning and finally *homogenisation*, giving a fully-preprocessed story as can be seen in Figure 3.6. In the *text relationship maps* of Figure 3.6a, the weight between two nodes denotes *similarity*, and we consider words to be synonyms whenever their *similarity* is at least 20. Furthermore, the labels of these nodes (starting from 0) for sentence *similarity* correspond to indices of *simplified* sentences from Figure 3.5.

In this particular case, sentences 5 and 6 are deemed irrelevant and will be pruned, while sentences 0, 1 and 2 are identified as being very important.

Moreover, {"homework", "school"} and {"interested", "curious"} are considered synonym sets, and applying *homogenisation* causes occurrences of these words to be replaced 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 Peter-Little. PeterLittle was curious in astronomy. PeterLittle was serious in school. PeterLittle did school always. PeterLittle became a astrophysicist. PeterLittle is famous now.

(c) *Homogenised* story and output of the Preprocessor

Figure 3.6: Example of applying sentence pruning and *homogenisation* to the *simplified* story of Peter Little

Chapter 4 ASG

We present in this chapter the core module of our pipeline: SumASG. After giving an overview of the module (Section 4.1), we introduce a general grammar which we have created specifically for the purpose of representing English sentences in ASG (Section 4.2). We then go further into the details of SumASG's two-part implementation (Sections 4.3 and 4.4), and finish by showing the results of running SumASG on a pre-processed story (Section 4.5).

4.1 Overview

Our use of 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.

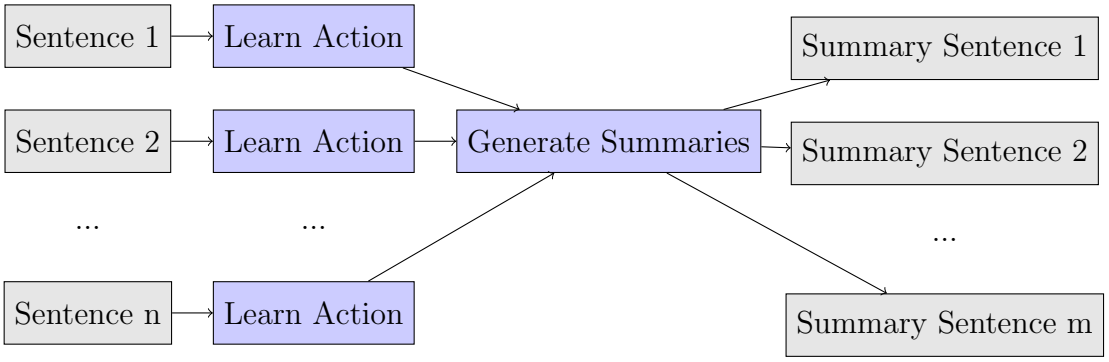


Figure 4.1: ASG Steps

4.2 Internal Representation

In order to model the structure of sentences in English, 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 C. 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 A.

4.2.1 Leaf Nodes

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

Each of these nodes has on the left-hand side (LHS) of the *production rule* its POS tag, and on the right-hand side (RHS) a string containing the word itself. In order to conform to the syntax of ASG, we must write the POS tags in lowercase. 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 *production rule* 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 simply 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)`	VB, 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	VB	VBD	VBG	VBN	VBP	VBZ
Verb tense	`base`	`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

4.2.2 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*.

Noun Parts

In the *production rule* 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 reason why we differentiate between these two forms is because some *noun parts* in English are only used as subjects (e.g., existential "there"), while others can only be objects (e.g., the adjective "green"), and we want to keep the *search space* as small as possible (see Subsection 4.3.3).

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 *production rule*:

```
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 *production rule*:

```
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 }
```

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 *production rule*:

```
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 *production rule* 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 }
```

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. 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 (line 2 in the code below):

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

4.3 Learning Actions

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

4.3.1 Formalisation

We formalise the task of learning an action as 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*. It is more of an *abduction* task, whereby we only learn *heads* of *production rules*. In our case, we use this as a mechanism to translate from English into our internal representation.

4.3.2 Implementation

In practice, this translation involves taking our general grammar and, on a per-sentence basis (see Subsection 4.3.3), appending to it the sentence's context-specific leaf nodes (given to us by the PREPROCESSOR), a *positive example* (containing the sentence itself), as well as a *mode bias* for learning *actions*.

Positive Example

To give a *positive example* to SUMASG$_1$, we must provide the sentence as a list of *tokens*. For instance, we would use the following *positive example* for the sentence "they drove a race-car fast.":

$$+ [\text{“they ”}, \text{“drove ”}, \text{“a ”}, \text{“race}-\text{car ”}, \text{“fast ”}, \text{“. ”}]$$

In order to ensure that this *positive example* contributes to learning a corresponding *action*, we also need to add a constraint to the *production rule* for sentences (`s -> np vp`). Intuitively, the rule shown in line 2 below says that if we have a sentence (i.e., our *positive example*) which consists of a given *verb*, *subject* and *object*, then we need to learn the matching *action*.

```
1 s -> np vp {
2   :- 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   ...
4 }
```

Mode Bias

In order to 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 *production rule*):

```
#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].
```

Such rules 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).
```

After running the full learning task on this example, the ASG engine returns a new program where the following *action* has been added to the *production rule* for sentences (`s -> np vp`):

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

After doing this for each sentence in the pre-processed story, we end up with at most as many *actions* as there are sentences in this text (those which do not *conform* to our general grammar are ignored).

4.3.3 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.

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.

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*:

```
#constant(main_verb,be).
#constant(aux_verb,eat).
#constant(main_form,present).
#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 are able to 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 form. 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.

4.4 Generating Summary Sentences

The second part of SumASG deals with generating summary sentences using the *actions* that were learned from the story in the previous step.

4.4.1 Formalisation

We formalise the task of generating a *summary sentence* as $\text{SumASG}_2(\underline{\text{CFG}},\underline{\text{BK}},\underline{\text{E}})$. 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 English sentences which may be used to summarize the text.

4.4.2 Implementation

In practice, this involves gathering all of the story-specific leaf nodes and learned *actions* from SumASG_1, adding these to our general grammar, and then using a set of summary generation rules to create summary sentences.

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 *production rule* for sentences (`s -> np vp`) in our general grammar.

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 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 also have more complex rules, allowing us to combine information from multiple *actions* into a single *summary sentence*. In the case where we have two *actions* that share a common *subject* and *verb*, we 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 *production rule* 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, we say that for each `output` (*summary sentence*), the child nodes of a sentence (`s`) must contain the *verb*, *subject* and *object* corresponding to the given `ouput`:

```
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.
```

Once we have augmented the *production rule* for sentences in our general grammar (`s -> np vp`) with the learned *actions* and our set of summary generation rules and constraints, we use to ASG engine to output all strings *conforming* to this augmented grammar. These strings correspond exactly to the possible *summary sentences* as written in English, which is what we use as the output for SumASG$_2$.

4.5 Example

Now that we have discussed how SumASG works, we can show the results of running it on the pre-processed story of Peter Little. Figure 4.3 shows the respective outputs of SumASG_1 and SumASG_2, as well as 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 apply SumASG_2, generating all possible *summary sentences* which are then used 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

Chapter 5 Post-Processing / Scoring

We present in this chapter the final part of our pipeline, which involves taking the output of SumASG to create summaries (Section 5.2), assigning to each one a score (Section 5.3), and then picking out the best ones (Section 5.4). We then illustrate these steps using the example of the story of Peter Little (Section 5.5).

5.1 Overview

Once we have obtained potential sentences from ASG to be used in a summary, we now post-process these as explained in Section 5.2. 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 5.3. A diagram illustrating these steps is shown below in Figure 5.1.

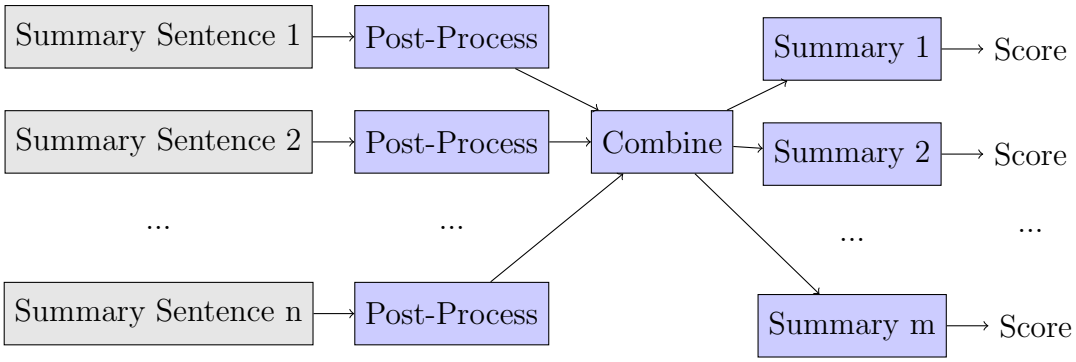


Figure 5.1: Post-Processing / Scoring Steps

5.2 Summary Creation

The output of SumASG is a list of sentences, each of which could potentially appear in the final summary. However before we start concatenating them together to form summaries, we should first post-process them to undo some of the simplifications made by the Preprocessor and ensure that they are grammatically-correct.

5.2.1 Post-Processing

Because SumASG uses the same capitalisation for a given word regardless of its position in the sentence, it means that the first word of each sentence will not be capitalised 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**[7], which is able to correct phrases like "they has an dog" to "they have a dog".

Moreover, 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, which is how they should appear in English.

5.2.2 Combining

Depending on the length of the original story, we envision a 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 $[1, 2, 3]$.

5.3 Scoring

Because we often end up with a large number of combinations at this phase, we need to determine which of these are preferred.

5.3.1 Type-Token Ratio

To this end, we utilize an NLP metric called type-token ratio (TTR), 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.

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.

[7] language-check is Python package that is able to find grammatical errors: https://pypi.org/project/language-check/

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 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. Moreover, 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*)

	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 for each generated summary

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

5.4 Summary Selection

Now that we have assigned a score to each generated summary, we should prioritise those which start with an introduction of the main character, and also get rid of suboptimal summaries. In addition, we should have a way to ensure that the summaries our framework prefers are indeed good-quality.

5.4.1 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 simply increase the score of every summary starting with a proper noun by a constant.

5.4.2 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 the 75th percentile over the scores of all generated summaries (giving us the value below which 75% of all scores fall), and then discard all those whose score falls below. We shall call the remaining summaries *top summaries*.

5.4.3 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*.

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.

5.5 Example

To illustrate how this works for the story of Peter Little, we have outlined the steps of *post-processing* and *scoring* in Figure 5.5. We have also repeated the *reference summaries* for this story in Figure 5.3, and the output of SumASG in Figure 5.4.

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.

Figure 5.3: *Reference summaries* for the story of Peter Little

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

Figure 5.4: *Summary sentences* as generated by SumASG

The first step, *post-processing*, involves fixing the grammar in the *summary sentences* generated by SumASG, which in this case simply means capitalising 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 default set of ignored words with the case-insensitive *topics* set {"peter", "little"} for TTR*. This gives us scores in the range $[10, 17]$, twenty of which fall below the 75th percentile (15.0) and never become *top summaries*.

Finally, we compare these *top summaries* to our *reference summaries* for Peter Little. One of them achieves a BLEU 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 5.5b)

Figure 5.5: Example of *post-processing* then *scoring* for the story of Peter Little

Chapter 6 Possible Technical Improvements

Had there been more time to finalise the project, there are a number of possible improvements we could have implemented. In what follows, we discuss immediate next steps which could be taken to improve each part of the pipeline.

6.1 Preprocessor

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

6.1.1 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".

6.1.2 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 3.2.1), 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.".

6.2 ASG

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

6.2.1 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 we 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.

6.2.2 Learning Summarization Rules

In its current implementation, SumASG$_2$ only uses 12 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.

6.2.3 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.

6.3 Post-Processing / Scoring

Although our approach to post-processing and *scoring* works well for the simple stories we have been using, it remains limited in terms of scope.

6.3.1 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 noun or proper noun 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".

6.3.2 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 determine 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 7 Validation And Evaluation

We present in this chapter the validation and evaluation of our approach, which involve comparing SumASG* to an *encoder-decoder*. After explaining the reasoning behind this comparison (Section 7.1), we present the training data that are used for validation (Section 7.2) and how our neural network performs on them (Section 7.3). We then carry out two evaluation experiments (Section 7.4) and finish by giving an overview of what we have learned (Section 7.5).

7.1 General Idea

As the vast majority of modern text summarization frameworks are based on Machine Learning, it makes sense to evaluate SumASG* against a neural network. There exist a number of text summarization corpora, such as the **CNN / Daily Mail dataset**[8]. However, we cannot use one of these datasets with SumASG* as it is unfortunately not general enough to be able to parse such complicated text.

In order to validate our approach, we have built a generative model which creates stories that our system is capable of summarizing. Using SumASG*, we generate summaries corresponding to these stories, and use this as training data for an *encoder-decoder*. If the neural network is able to learn how to generate these summaries, then we can consider our framework and the summarization rules it uses to be sound.

To evaluate our approach, we then use this generated dataset as a starting point to check both the *encoder-decoder* and SumASG* for robustness to small perturbations in the input, as well as their ability to detect invalid input.

7.2 Story Generation

To give a more generalized representation of what SumASG* can do, we shall create two different types of stories: those with *conjunctive summaries*, and those with *descriptive summaries*. In this section, we go more into depth about these two types of randomly-generated stories, and discuss how they are created.

7.2.1 Datasets

In order to generate the required number of stories, we have used words from **wordfrequency.info**[9]. This dataset contains 5,000 individual English words, of which 1,001 are verbs, 2,542 nouns and 839 adjectives.

[8]The CNN / Daily Mail dataset consists of around 300,000 news articles and their corresponding multi-sentence summaries; these have been used to train state-of-the-art text summarization models, which were evaluated using metrics such as ROUGE and METEOR: http://nlpprogress.com/english/summarization.html

[9]wordfrequency.info hosts a dataset of the most commonly used words in English

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

In addition, we query from the **Datamuse API**[10] for what we call a *lexical verb*, i.e. one that is related to the story's *topic*. If one cannot be found, then we default to the verb "to be".

7.2.2 Main Sentence Generation

We will begin by detailing how what we call *main sentences* are 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. It is important to note that all *main sentences* for the same story share a common *subject* and *verb*.

Definition 13 (Holonym)**.** A *holonym* of something is one of its constituents; "light-bulb" is a *holonym* of "lamp".

Definition 14 (Meronym)**.** A *meronym* is an object which something is part of; "house" is a *meronym* of "kitchen".

For the *subject* of a *main sentence*, we use the story's *topic*. This being a singular noun, we need to add a determiner, which can be "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 are found, then we do not need to use an adjective.

Here we use the *lexical verb*, conjugating it in the past tense using **Pattern**[5] so that it agrees with the sentence's *subject*.

For the *object* of our sentence, we look at the story's *topic* and *lexical verb*. Using the **Datamuse API** we try and find a noun which often appears right after this verb, and which is related to all of the nouns we have used thus far in the story. With 50% probability we ask it to be a *holonym* of the *topic*, 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 fine 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.

Example

We take the example of generating a sentence for a story whose *topic* is "football". In this case, we have two choices for our *lexical verb*: "to match" and "to pitch".

For the *subject*, we use the *topic* "football" with the determiner "a"; an adjective commonly used to modify this word is "professional". We then conjugate the verb "to pitch" in the past tense, which becomes "pitched". For the *object*, we take into account our *topic* "football" to find a *holonym* of this word 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.

The resulting *main sentence* can be seen in Figure 7.1, along with a second *main sentence* that has been generated for the same story.

[10] Datamuse is a lexical knowledge engine which can be used to find words that are semantically related to a given word in a certain way: https://www.datamuse.com/api/

A professional football pitched the warm reception.
A professional football pitched a place.

Figure 7.1: *Main sentences* generated for a story with *topic* "football" and *lexical verb* "to pitch"

As has already been mentioned, generating a coherent and meaningful text is a very difficult problem for computers. This explains why the *main sentences* shown in Figure 7.1 do not make much sense from a human perspective. However, they are both grammatically-correct, as well as highly suitable for the task of summarization, which is what we are looking for in this experiment.

7.2.3 Conjunctive Summaries

Stories with *conjunctive summaries* consist of three *main sentences*. Because of the way in which we have implemented the summarization rules in SumASG, one of the sentences in the corresponding summary should be a combination of two of the input story's sentences. That is to say, its *object* should consist of these two sentences' *objects*, joined using the conjunction "and". An example is shown in Figure 7.2.

The publication printed a lightning. The publication printed a movement. The publication printed the stereotype.

(a) Story

The publication printed a lightning **and** a movement. The publication printed the stereotype.

(b) Summary

Figure 7.2: Example of a story with a *conjunctive summary*

7.2.4 Descriptive Summaries

In contrast, stories with a *descriptive summary* consist of a single *main sentence*, one which *does* contain an adjective in the *object* position.

We use the second of its two sentences to expand on the first. To be more precise, the *object* of this sentence is the same as in the first, apart from the fact that we assign an adjective as is done for typical *main sentences*. However the *subject* here is the preposition "it", while the *verb* is the verb "to be" conjugated in the past tense.

The idea for a *descriptive summary* is that it will be identical to the first sentence of its corresponding story, but augmented with the adjective coming from the second sentence, as illustrated in Figure 7.3.

The heavy traffic transported the birthday. It was the isolated birthday.

(a) Story

The heavy traffic transported the isolated birthday.

(b) Summary

Figure 7.3: Example of a story with a *descriptive summary*

7.2.5 Summary Generation

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 because we can consistently produce the same *actions* as SUMASG$_1$ programmatically with the chosen sentence structure. Also, because of the way in which we have created our stories, *simplification* in the PREPROCESSOR would not change anything whatsoever.

To generate a summary for this experiment, we take a story and feed the corresponding *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*.

7.3 Validation

Using the mechanism described in Section 7.2, we generate for our *encoder-decoder* 4,000 story/summary pairs: 3,582 to be used for training, 398 for validation and 20 for testing.

To allow for greater flexibility, we have chosen to use **OpenNMT-py**[11] to train our neural network. In addition, we preprocess the data using **GloVe**[12], giving our network a head start when it comes to semantics.

Our *encoder* and *decoder* share embeddings for a vocabulary of size 4,239, its contents being internally represented using a vector of size 500. They both use a two-layer LSTM with *dropout* of 0.25 and *hidden size* of 500. Additionally, our *decoder* uses *global attention*.

The neural network was trained using an Adam optimizer with a *learning rate* of 0.001 and *batch size* of 25. In order to preserve the GloVe word embeddings across training, we fix them at the start and use them in both the *encoder* and *decoder*.

Training was done over a period of 10,000 steps (i.e., 200 epochs), validating every 10 epochs. Using **Google Colaboratory**[13] this took a total of 5 minutes and 30 seconds. The final training and validation accuracies are respectively 99.83% and 92.07%, as shown in Table 7.1.

[11] OpenNMT-py is a highly versatile open-source framework for performing Neural Machine Translation: https://github.com/OpenNMT/OpenNMT-py

[12] GloVe is a Machine Learning model consisting of pre-trained word embeddings: https://nlp.stanford.edu/projects/glove/

[13] Google Colaboratory is an online tool for creating Python notebooks, providing free access to GPUs: https://colab.research.google.com/

Epochs	100	500	1000	1500	2000
Training accuracy (%)	60.68	98.73	99.45	99.82	99.83
Validation accuracy (%)	61.54	90.24	91.38	91.38	92.07
Training perplexity	12.45	1.06	1.03	1.01	1.01
Validation perplexity	16.73	2.07	2.15	2.23	2.36
Test perplexity	-	-	-	-	2.05

Table 7.1: Evolution of accuracy and perplexity during training

Discussion

Unsurprisingly, our *encoder-decoder* was able to mostly learn this hard-coded summarization task, as confirmed by the numbers in Table 7.1. This is because neural networks have a much more general scope than SumASG*.

However, it appears that it was not able to fully learn the rules used by SumASG, as the final validation and test perplexities are a little high (> 2).

Part of this discrepancy is due to the use of slightly different words in the predicted summaries. Upon further inspection, the words that differ have very close meanings; this is because we have used pre-trained embeddings and fixed their representation. It is not too problematic though, as it would be much worse to have a semantically-unrelated word in one of our predicted summaries.

Another reason for this discrepancy is the fact that multiple *conjunctive summaries* can be generated from the same story, depending on which words are chosen to be put together.

An example of a predicted summary which highlights both of these discrepancies is shown in Figure 7.4.

A new approach attacked the lesson. A new approach attacked the counseling. A new approach attacked the league.

(a) Story

A new approach attacked the lesson **and** the counseling. A new approach attacked the league.

(b) Expected summary

A new approach attacked the lesson **and** the league. A new approach attacked the patient.

(c) Predicted summary

Figure 7.4: Example of a story whose predicted summary is different from the expected summary, but with very close semantics

7.4 Evaluation Experiments

In order to evaluate our approach and highlight some of the major differences between SumASG* and a neural network-based text summarization system, we carry out two experiments in what follows.

7.4.1 Experiment 1: Robustness To Perturbations

For the first experiment, we take the test data from Section 7.3, which we know works well with our *encoder-decoder* (see Table 7.1), and apply a few small perturbations to the stories, while keeping them grammatically-correct. With these changes, we now run both text summarization systems to see how they perform.

To avoid giving SumASG* an advantage, let us choose a story whose predicted summary is exactly the expected summary. Also, to prevent sentences in the modified stories from being pruned, we bypass the Preprocessor.

For perturbation **A** we move the *subject*'s adjective to each *object* in the story, while perturbation **B** involves exchanging the *subject* and *object* in the first sentence of the story.

The results of applying these two perturbations are shown below in Figure 7.5.

The nervous system processed a fraud. The nervous system processed a whale. The nervous system processed the American.

(a) Original story

A. The system processed a nervous fraud. The system processed a nervous whale. The system processed the nervous American.
B. A fraud processed the nervous system. The nervous system processed a whale. The nervous system processed the American.

(b) Story with perturbations **A** and **B** respectively applied (changes are underlined)

A. The system processed a banking and a killer. The system processed the nervous.
B. A fraud processed the nervous and the American. A fraud processed a whale.

(c) Summaries predicted by the trained *encoder-decoder*

A. The system processed a fraud and a whale. The system processed the nervous American.
B. The nervous system processed the American. The nervous system processed a whale.

(d) Summaries generated by SumASG* (without using the Preprocessor)

Figure 7.5: Results of experiment 1: summaries generated by SumASG* and the *encoder-decoder* after separately applying two different perturbations

Unsurprisingly, SumASG* is able to produce a grammatically-correct summary with little loss of information and no change in meaning for both perturbations.

However, our neural network has created a *conjunctive summary* for perturbation **A** that changes the first two *objects* to words with a slight semantic link, and makes too much of a generalisation of the third sentence of the modified story by omitting the *object* noun. For perturbation **B**, the summary is even more different from the modified story: *subjects* and *objects* from different sentences have been put together, conveying a completely different meaning. Of course, this is extremely tied to the data we have used to train our *encoder-decoder*, so such results were expected.

From this experiment we conclude that SumASG* is more robust to small changes in the input than a neural network (with respect to training corpus), as well as much more consistent when it comes to producing a summary that is coherent with the original text.

7.4.2 Experiment 2: Input Validity Awareness

The goal of the second experiment is to how both systems deal with invalid input (i.e., stories that are not grammatically correct). To avoid giving SumASG* too much of an advantage, we use only words from the vocabulary known by our *encoder-decoder*, and create sequences of similar length compared to the training data used in Section 7.3.

For invalid input **A** we take the original story from Figure 7.5 and make a few small changes to render it grammatically-incorrect, while for invalid input **B** we create a story of length 3 from scratch using randomly selected words from the vocabulary.

While SumASG* is able the recognise that these stories are not grammatically correct, our *encoder-decoder* produces an output, as is shown in Figure 7.6.

A. The system processed processed a nervous fraud. processed The system a nervous whale. a The system processed nervous American.
B. question hunger whole ruled cleared. needle front pound spun rented programming. bought journalism disclosed broad check died delight.

(a) Invalid stories (changes in story **A** from the original story in Figure 7.5 are underlined; story **B** is generated from arbitrary words in the *encoder-decoder*'s vocabulary)

A. The system processed a nervous and a killer. The system processed a pregnancy.
B. The

(b) Summaries predicted by the trained *encoder-decoder*

Figure 7.6: Results of experiment 2: summaries generated by the *encoder-decoder* on invalid input stories (SumASG* recognises these as invalid)

Since SumASG* is built around a grammar that models the structure of English sentences, the fact that it recognises these stories as invalid is no surprise.

By looking at the neural network's prediction for invalid story **A**, we see that only one of the three nouns in the output summary appear in the story. Moreover,

even if we were to correct story **A**, its meaning would have nothing in common with the predicted summary.

In contrast, the prediction for invalid story **B** is not grammatically correct, and its only word does not even appear in the story. Of course, such a result is expected as this story is incredibly different from any of the training pairs, and does make any sense whatsoever.

From this experiment, we can conclude that a neural network trained for text summarization cannot detect when the input invalid, unless it has been specifically trained to do so. However, SumASG* is by construction unable to summarize grammatically-incorrect stories.

7.5 Takeaways

To sum up what we have learned from validating and then evaluating our approach, we use a table to outline the main differences between SumASG* and using a neural network, as shown in Table 7.2.

	Neural network	SumASG*
Rules	Learnable using state-of-the-art *encoder-decoders*	Written directly into program
Training required	Yes; can take a long time	No
Examples required	Vast amounts for training	None
Expansion	Need to retrain	Can be used directly
Coherence of result (Subsection 7.4.1)	Extremely tied to nature and diversity of training corpus	Similar on all parsable texts
Output *tokens*	Can be irrelevant or `<unk>` *token*	Taken from input text
Termination (Subsection 7.4.2)	Always produces output, regardless of whether input is valid English	Sometimes returns no summaries

Table 7.2: Main differences between SumASG* and neural networks used for the task of text summarization

Chapter 8 **Related Work**

8.1 Semantic Analysis Methods

8.1.1 Combinatory Categorial Grammar

In a paper from 2019 [20], the author introduces Combinatory Categorial Grammar (CCG), an efficient parsing mechanism to get to the underlying semantics of a text in any natural language. It is combinatory in the sense that it uses functional expressions such as $\lambda p.p$ in order to express the semantics of words.

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

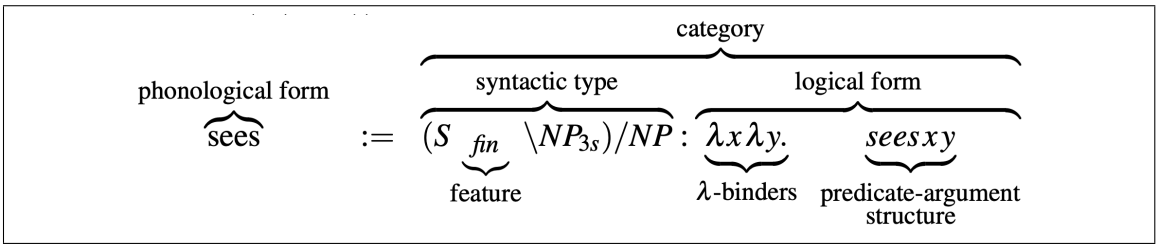
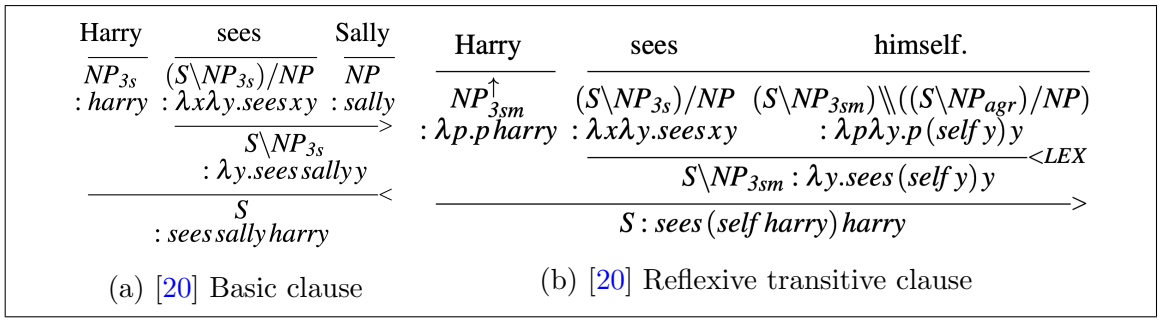


Figure 8.1: [20] Diagram explaining the mains terms used in CCG

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

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

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



(a) [20] Basic clause

(b) [20] Reflexive transitive clause

Figure 8.2: Examples of derivations in CCG

There are 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, it is easy to 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 allows for a very precise understanding of English grammar, as well as giving a simple and consistent *logical form* at the end.

8.2 Existing Text Summarization Approaches

8.2.1 Corpus-Based Approach And Latent Semantic Analysis

In a paper by Yeh et al. [21], 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 actually real

positives) and *recall* (proportion of real positive cases that are correctly predicted positive) [22], a so-called *F-score* is computed to define the fitness for each chromosome. By combing two *chromosomes* to generate children, where the fittest parents are most likely to mate, we end up (after some number of generations) with a set of feature weights suitable for the corpus in question.

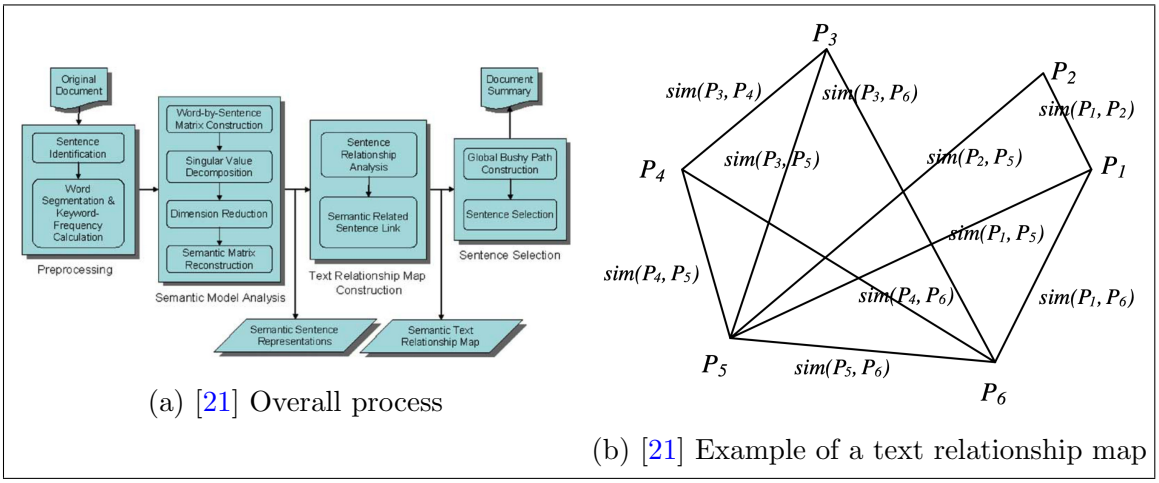
On the other hand, the LSA+TRM approach comprises four major steps: *preprocessing* (1), *semantic model analysis* (2), *text relationship map construction* (3) and *sentence selection* (4).

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

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

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

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



(a) [21] Overall process

(b) [21] Example of a text relationship map

Figure 8.3: LSA+TRM approach, diagrammatically

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

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

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

Comparing the two approaches highlighted in the paper, it is mentioned that performance is similar, although LSA+TRM is easier to implement than MCBA

in single-document level as it requires no preprocessing, and in some optimal cases performs up to 20% better. Although the former approach is more computationally expensive, it is more adept at understanding the semantics of a text because it does not rely on the genre of the corpus that was used for training. In both cases though, performance improves as CR increases.

From the first approach, what is interesting is that it uses a certain number of important features to identify the most pertinent sentences of a passage.

From the second approach, the main takeaways are the storage mechanisms in use such as the *semantic matrix* and *text relationship map*.

8.2.2 Lexical Chains

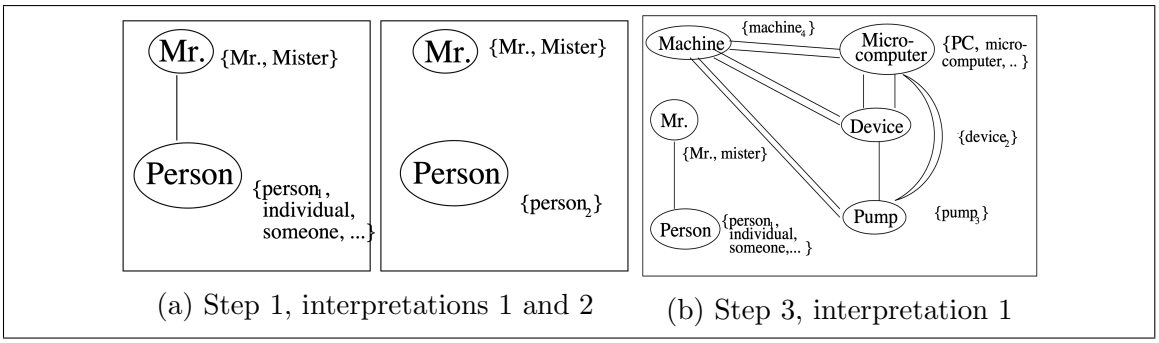
In a paper about *lexical chains* [23], the authors describe a method which relies on semantic links between words. The idea is that we establish chains of related words, in order to learn what a text is about.

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 meanings, then the chain gets split into multiple interpretations; this process continues until we have analysed all *candidate words*. For instance, the word "person" can be interpreted as meaning a human being (interpretation 1), or as a grammatical term used for pronouns (interpretation 2). An example for the below text is shown in Figure 8.4.

> **Mr.** Kenny is the **person** that invented an anesthetic **machine** which uses **micro-computers** to control the rate at which an anesthetic is pumped into the blood. Such **machines** are nothing new. But his **device** uses two **micro-computers** to achieve much closer monitoring of the **pump** feeding the anesthetic into the patient. [23]

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

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



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

Figure 8.4: [23] 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.

8.3 Approach Categories

8.3.1 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 [24] 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

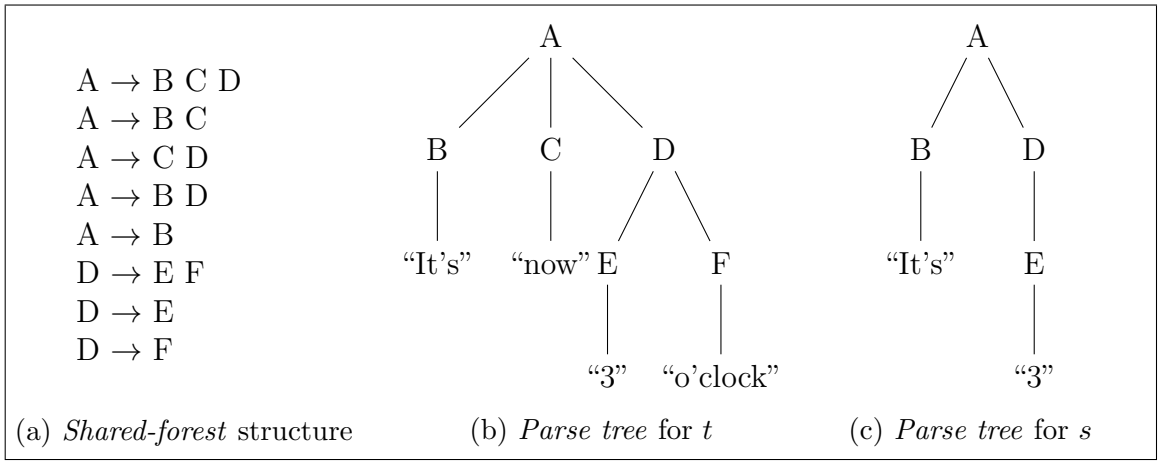
$P(s)$ to less grammatically-correct strings, then it helps ensure that our final summary is well-formed.

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[14], is equivalent to maximizing $P(s) \cdot P(t|s)$.

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

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

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



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

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

In their testing it was found that their algorithm has a conservative nature, promoting correctness over brevity, which sometimes has the consequence of not trimming any words away. We learn from this approach the notion of a *shared-forest* structure, as well as the use of Bayesian probability theory.

8.3.2 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

[14]For the definition of Bayes' theorem, see https://www.investopedia.com/terms/b/bayes-theorem.asp

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

In one of the original papers describing this approach [25], sentences are decomposed into different *affect states* and *affect links*. An *affect state* can either be a *mental state*, or a *positive* or *negative event* which may cause a change to a *mental state*. *Affect links* are then the transitions that explain the sequence of *affect states*.

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

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

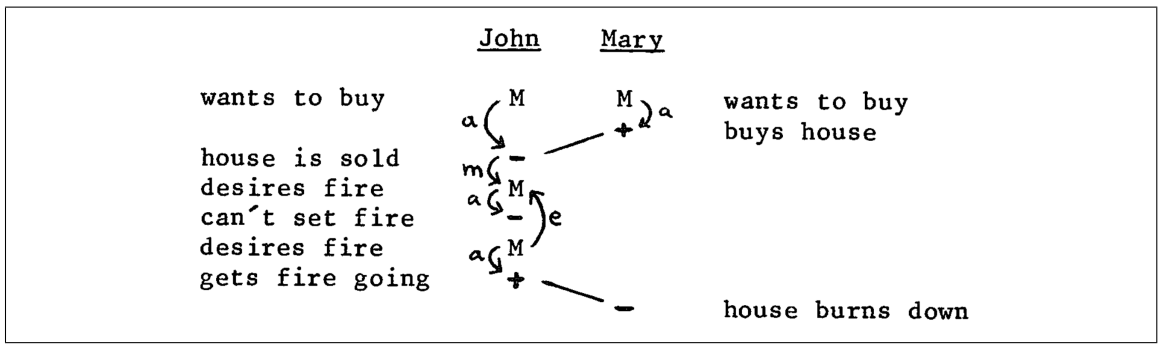


Figure 8.6: [25] Example of a narrative in the frame approach

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.").

8.3.3 Symbolic

In the symbolic approach [26], 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 8.1.1, 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 [27]. 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 A), 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 type of approach [26] 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.

Chapter 9 **Conclusion**

We have engineered a logic-based text summarization system, which solves the task in different way than the commonly-used Machine Learning approaches these days.

9.1 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 some English sentences, without losing too much information (e.g. co-referencing).
- Implemented an algorithm which uses *similarity* to remove irrelevant sentences from a short story, as well as reduces lexical diversity.
- Wrote an ASG learning task capable of taking English text and turning it into a set of chronologically-ordered *actions*.
- Developed a set of rules which, given *actions* from a story, allow ASG to generate both *extractive* and *abstractive* summary sentences.
- Implemented a scoring mechanism prioritizing information density, while taking into account words which may appear frequently in English and are considered the *topic* of the original text.
- Created a framework to automatically generate topical short stories for training a *neural network* in the aim of evaluate the soundness of SumASG*, based on a dataset of words and particular sentence structures.
- Used the trained *encoder-decoder* to show that our approach is able to produce more consistent results than a neural network, and can detect invalid input out-of-the-box.

9.2 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.

9.2.1 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 9.1.

> 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 9.1: 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.

9.2.2 Longer Stories

What we would like to do is 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.

9.2.3 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: [28] List of position of speech (POS) tags

Appendix B **Example Stories**

In what follows are examples designed to test various aspects of SumASG*. For each story we list a few of the generated *top summaries* and underline the best one.

Powerful Car

The green car was moving fast.
It was a very powerful vehicle.

(a) Story

- The green car was moving fast.
- The powerful green car was moving fast.

(b) *Top summaries*

Electric Car

Barack Obama was president.
You and Ian have an electric car.
You own a red vehicle.

(a) Story

- You own a red car.
- You own an electric red car.
- You and Ian own an electric car.

(b) *Top summaries*

Jonathan

Jonathan likes ASG.
Jonathan likes ILASP.
Jonathan eats culatello.

(a) Story

Jonathan likes ASG and ILASP.

(b) *Top summaries*

Mr. Predicate

Mr. Predicate was a logic. The logical system was sound and complete.

(a) Story

Mr Predicate was a sound complete logic.

(b) *Top summaries*

Mary

Mary went out.
It was raining hard.
Mary went back.
She entered quickly.

(a) Story

- Mary was out quickly. Mary was back.
- Mary was out quickly. It was raining hard.
- Mary was out. Mary was back.
- Mary was back quickly. It was raining hard.

(b) *Top summaries*

Jack's Birdhouse

The example shown in Figure B.6 was taken from **K5Learning**[15].

Jack wants to build a birdhouse. He gets some wood.
He gets some nails and paint.
His mom helps too.
She gets a saw and a hammer.
She gets a pencil and ruler.
Jack draws his birdhouse. They build it together. Then they hang it up in a tree.
A bird goes into the bird house.
A second bird goes in. A third bird goes in.
A fourth bird goes in!
Jack and his mom look at each other.
They need a bigger birdhouse!

(a) Story

- Jack wants to build a birdhouse. He gets wood. A bird goes in the birdhouse.
- Jack wants to build a birdhouse. He gets wood and nails. They hang it up.
- Jack wants to build a birdhouse. He gets wood and paint. They hang it up.
- Jack gets birdhouse. A fourth bird goes in. They need a bigger birdhouse.
- Jack wants to build a birdhouse. A second bird goes in. A fourth bird goes in.
- Jack wants to build a birdhouse. He gets wood and nails. A bird goes in the birdhouse.
- Jack gets birdhouse. A bird goes in the birdhouse. They need a bigger birdhouse.
- Jack wants to build a birdhouse. A fourth bird goes in. They need a bigger birdhouse.

(b) *Top summaries*

Figure B.6: Example of the task of summarization for "Jack's Birdhouse"

[15]K5Learning is a site that provides worksheets for Key Stage 1 students: https://www.k5learning.com/reading-comprehension-worksheets/first-grade-1

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 C.2 and C.3 gets appended to the general grammar, giving the complete ASG program.

C.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.
}

```

```
vp -> vbd ‘‘to ’’ vb np {
  verb(comp(N1,N2),comp(T1,base)) :- verb(N1,T1)@1, verb(N2,base)@3.
  object(N,D,A) :- object(N,D,A)@4.
}

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

vp -> vbp 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 -> vbp 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.
}

vp -> vbp ‘‘to ’’ vb np {
  verb(comp(N1,N2),comp(T1,base)) :- verb(N1,T1)@1, verb(N2,base)@3.
  object(N,D,A) :- object(N,D,A)@4.
}

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

vp -> vbz 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 -> vbz 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.
}

vp -> vbz ‘‘to ’’ vb np {
  verb(comp(N1,N2),comp(T1,base)) :- verb(N1,T1)@1, verb(N2,base)@3.
  object(N,D,A) :- object(N,D,A)@4.
}

np -> np rb {
```

```
82    object(N,D,A) :- object(N,D,0)@1, adj_or_adv(A)@2.
83  }
84
85  np -> np rb {
86    object(N,D,conjunct(A1,A2)) :- object(N,D,A1)@1, adj_or_adv(A2)@2.
87    :- object(N,D,conjunct(A,A)).
88  }
89
90  np -> np rp {
91    object(N,D,A) :- object(N,D,0)@1, adj_or_adv(A)@2.
92  }
93
94  np -> np rp {
95    object(N,D,conjunct(A1,A2)) :- object(N,D,A1)@1, adj_or_adv(A2)@2.
96    :- object(N,D,conjunct(A,A)).
97  }
98
99  np -> nn {
100   subject(N,0,0) :- noun(N)@1.
101   object(N,0,0) :- noun(N)@1.
102 }
103
104 np -> nns {
105   subject(N,0,0) :- noun(N)@1.
106   object(N,0,0) :- noun(N)@1.
107 }
108
109 np -> nnp {
110   subject(N,0,0) :- noun(N)@1.
111   object(N,0,0) :- noun(N)@1.
112 }
113
114 np -> nnps {
115   subject(N,0,0) :- noun(N)@1.
116   object(N,0,0) :- noun(N)@1.
117 }
118
119 np -> prp {
120   subject(N,0,0) :- noun(N)@1.
121   object(N,0,0) :- noun(N)@1.
122 }
123
124 np -> rb {
125   subject(0,0,A) :- adj_or_adv(A)@1.
126   object(0,0,A) :- adj_or_adv(A)@1.
127 }
128
129 np -> rp {
```

```
130   subject(0,0,A) :- adj_or_adv(A)@1.
131   object(0,0,A) :- adj_or_adv(A)@1.
132 }
133
134 np -> ex {
135   subject(N,0,0) :- noun(N)@1.
136 }
137
138 np -> in {
139   object(0,D,0) :- det(D)@1.
140 }
141
142 np -> prp ‘‘and ’’ nnp {
143   subject(conjunct(N1,N2),0,0) :- noun(N1)@1, noun(N2)@3.
144   object(conjunct(N1,N2),0,0) :- noun(N1)@1, noun(N2)@3.
145   :- subject(conjunct(N,N),0,0).
146   :- object(conjunct(N,N),0,0).
147 }
148
149 np -> nnp ‘‘and ’’ prp {
150   subject(conjunct(N1,N2),0,0) :- noun(N1)@1, noun(N2)@3.
151   object(conjunct(N1,N2),0,0) :- noun(N1)@1, noun(N2)@3.
152   :- subject(conjunct(N,N),0,0).
153   :- object(conjunct(N,N),0,0).
154 }
155
156 np -> dt nn ‘‘and ’’ prp {
157   subject(conjunct(N1,N2),D,0) :- det(D)@1, noun(N1)@2, noun(N2)@4.
158   object(conjunct(N1,N2),D,0) :- det(D)@1, noun(N1)@2, noun(N2)@4.
159   :- subject(conjunct(N,N),_,0).
160   :- object(conjunct(N,N),_,0).
161 }
162
163 np -> prp ‘‘and ’’ dt nn {
164   subject(conjunct(N1,N2),D,0) :- noun(N1)@1, det(D)@3, noun(N2)@4.
165   object(conjunct(N1,N2),D,0) :- noun(N1)@1, det(D)@3, noun(N2)@4.
166   :- subject(conjunct(N,N),_,0).
167   :- object(conjunct(N,N),_,0).
168 }
169
170 np -> dt nn ‘‘and ’’ nnp {
171   subject(conjunct(N1,N2),D,0) :- det(D)@1, noun(N1)@2, noun(N2)@4.
172   object(conjunct(N1,N2),D,0) :- det(D)@1, noun(N1)@2, noun(N2)@4.
173   :- subject(conjunct(N,N),_,0).
174   :- object(conjunct(N,N),_,0).
175 }
176
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).
```

```
}

np -> prp ‘‘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 -> rb ‘‘and ’’ rb {
  subject(0,0,conjunct(A1,A2)) :- adj_or_adv(A1)@1, adj_or_adv(A2)@3.
  object(0,0,conjunct(A1,A2)) :- adj_or_adv(A1)@1, adj_or_adv(A2)@3.
  :- subject(0,0,conjunct(A,A)).
  :- object(0,0,conjunct(A,A)).
}

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

np -> jj ‘‘and ’’ jj {
  object(0,0,conjunct(A1,A2)) :- adj_or_adv(A1)@1, adj_or_adv(A2)@3.
  :- object(0,0,conjunct(A,A)).
}

np -> jj rb {
  subject(0,0,conjunct(A1,A2)) :- adj_or_adv(A1)@1, adj_or_adv(A2)@1.
  object(0,0,conjunct(A1,A2)) :- adj_or_adv(A1)@1, adj_or_adv(A2)@1.
  :- subject(0,0,conjunct(A,A)).
  :- object(0,0,conjunct(A,A)).
}

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

np -> dt nns {
  subject(N,D,0) :- det(D)@1, noun(N)@2.
  object(N,D,0) :- det(D)@1, noun(N)@2.
}

np -> jj nns {
  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 nnp {
```

```
272   subject(N,0,A) :- adj_or_adv(A)@1, noun(N)@2.
273   object(N,0,A) :- adj_or_adv(A)@1, noun(N)@2.
274 }
275
276 np -> jj nnps {
277   subject(N,0,A) :- adj_or_adv(A)@1, noun(N)@2.
278   object(N,0,A) :- adj_or_adv(A)@1, noun(N)@2.
279 }
280
281 np -> dt jj nn {
282   subject(N,D,A) :- det(D)@1, adj_or_adv(A)@2, noun(N)@3.
283   object(N,D,A) :- det(D)@1, adj_or_adv(A)@2, noun(N)@3.
284 }
285
286 np -> dt jj nns {
287   subject(N,D,A) :- det(D)@1, adj_or_adv(A)@2, noun(N)@3.
288   object(N,D,A) :- det(D)@1, adj_or_adv(A)@2, noun(N)@3.
289 }
290
291 np -> dt jj jj nn {
292   subject(N,D,conjunct(A1,A2)) :- det(D)@1, adj_or_adv(A1)@2, adj_or_adv(A2)
          ↪ @3, noun(N)@4.
293   object(N,D,conjunct(A1,A2)) :- det(D)@1, adj_or_adv(A1)@2, adj_or_adv(A2)
          ↪ @3, noun(N)@4.
294   :- subject(N,D,conjunct(A,A)).
295   :- object(N,D,conjunct(A,A)).
296 }
297
298 np -> dt jj jj nns {
299   subject(N,D,conjunct(A1,A2)) :- det(D)@1, adj_or_adv(A1)@2, adj_or_adv(A2)
          ↪ @3, noun(N)@4.
300   object(N,D,conjunct(A1,A2)) :- det(D)@1, adj_or_adv(A1)@2, adj_or_adv(A2)
          ↪ @3, noun(N)@4.
301   :- subject(N,D,conjunct(A,A)).
302   :- object(N,D,conjunct(A,A)).
303 }
304
305 np -> dt jjr nn {
306   subject(N,D,A) :- det(D)@1, adj_or_adv(A)@2, noun(N)@3.
307   object(N,D,A) :- det(D)@1, adj_or_adv(A)@2, noun(N)@3.
308 }
309
310 np -> dt jjr nns {
311   subject(N,D,A) :- det(D)@1, adj_or_adv(A)@2, noun(N)@3.
312   object(N,D,A) :- det(D)@1, adj_or_adv(A)@2, noun(N)@3.
313 }
314
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 }
```

C.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))).").
41  #bias(":- head(holds_at_node(action(verb(_,_),subject(_,_,_),object(_,conjunct(0,_),
    ↪ _)),var__(1))).").
42
43  #bias(":- head(holds_at_node(action(verb(_,_),subject(_,_,conjunct(V,_)),object(_,_
    ↪ ,conjunct(V,_))),var__(1))).").
44  #bias(":- head(holds_at_node(action(verb(_,_),subject(_,_,conjunct(V,_)),object(_,_
    ↪ ,conjunct(_,V))),var__(1))).").
45  #bias(":- head(holds_at_node(action(verb(_,_),subject(_,_,conjunct(_,V)),object(_,_
    ↪ ,conjunct(V,_))),var__(1))).").
46  #bias(":- head(holds_at_node(action(verb(_,_),subject(_,_,conjunct(_,V)),object(_,_
    ↪ ,conjunct(_,V))),var__(1))).").
47
48  #bias(":- head(holds_at_node(action(verb(comp(V,V),comp(_,past_part)),subject(
    ↪ _,_,_),object(0,0,0)),var__(1))).").
49
50  #constant(noun,0).
51  #constant(det,0).
52  #constant(adj_or_adv,0).
```

C.3 Task: SumASG$_2$

```
1   s -> np vp {
2     ...
3
4     summary(0, V, S, O) :- action(_, V, S, O).
5
6     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)).
7     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))).
8     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)).
9     summary(4, V, S, object(N,D1,conjunct(A1,A2))) :- action(I1, V, S, object(N,
        ↪ D1,A1)), action(I2, V, S, object(N,D2,A2)), A1 != A2, A1 != 0, A2 !=
        ↪ 0, I1 < I2.
10    summary(5, V, S, object(N,D2,conjunct(A1,A2))) :- action(I1, V, S, object(N,
        ↪ D1,A1)), action(I2, V, S, object(N,D2,A2)), A1 != A2, A1 != 0, A2 !=
        ↪ 0, I1 < I2.
11    summary(6, 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.
12    summary(7, 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.
13    summary(8, 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.
14  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, A1)), A1 != A3.
15  summary(10, 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.
16  summary(11, 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.
17
18  summary(I, V, S, object(conjunct(N1,N2),D,A)) :- summary(I, V, S, object(
    ↪ conjunct(conjunct(N1,N2),N3),D,A)).
19  summary(I, V, S, object(conjunct(N1,N2),D,A)) :- summary(I, V, S, object(
    ↪ conjunct(N1,conjunct(N2,N3)),D,A)).
20  summary(I, V, S, object(conjunct(N2,N3),D,A)) :- summary(I, V, S, object(
    ↪ conjunct(conjunct(N1,N2),N3),D,A)).
21  summary(I, V, S, object(conjunct(N2,N3),D,A)) :- summary(I, V, S, object(
    ↪ conjunct(N1,conjunct(N2,N3)),D,A)).
22  summary(I, V, S, object(conjunct(N1,N3),D,A)) :- summary(I, V, S, object(
    ↪ conjunct(conjunct(N1,N2),N3),D,A)).
23  summary(I, V, S, object(conjunct(N1,N3),D,A)) :- summary(I, V, S, object(
    ↪ conjunct(N1,conjunct(N2,N3)),D,A)).
24  summary(I, V, S, object(N,D,conjunct(A1,A2))) :- summary(I, V, S, object(N,
    ↪ D,conjunct(conjunct(A1,A2),A3))).
25  summary(I, V, S, object(N,D,conjunct(A1,A2))) :- summary(I, V, S, object(N,
    ↪ D,conjunct(A1,conjunct(A2,A3)))).
26  summary(I, V, S, object(N,D,conjunct(A2,A3))) :- summary(I, V, S, object(N,
    ↪ D,conjunct(conjunct(A1,A2),A3))).
27  summary(I, V, S, object(N,D,conjunct(A2,A3))) :- summary(I, V, S, object(N,
    ↪ D,conjunct(A1,conjunct(A2,A3)))).
28  summary(I, V, S, object(N,D,conjunct(A1,A3))) :- summary(I, V, S, object(N,
    ↪ D,conjunct(conjunct(A1,A2),A3))).
29  summary(I, V, S, object(N,D,conjunct(A1,A3))) :- summary(I, V, S, object(N,
    ↪ D,conjunct(A1,conjunct(A2,A3)))).
30
31  % Pick exactly one summary sentence for each applicable rule
32  0{output(I,V,S,O)}1 :- summary(I,V,S,O).
33  :- not output(_,_,_,_).
34
35  :- 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.
36  :- 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.
37  :- 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.
38  }
```

Bibliography

[1] 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.

[2] 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.

[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] Kübler S. Dependency Parsing;p. 40.

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

[8] 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.

[9] Apt KR. Logic Programming. Handbook of Theoretical Computer Science, Volume B: Formal Models and Sematics (B). 1990;1990:493–574.

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

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

[12] 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.

[13] 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.

[14] Law M, Russo A, Broda K. Inductive Learning of Answer Set Programs - User Manual;p. 25.

[15] Kröse B, Krose B, Smagt Pvd, Smagt P. An introduction to Neural Networks; 1993.

[16] 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.

[17] 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.

[18] 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/.

[19] 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.

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

[21] 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.

[22] 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.

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

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

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

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

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

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