Imperial College London

Individual Project

Department of Computing

Imperial College of Science, Technology and Medicine

Using Answer Set Grammars For Text Summarization

Author:
Julien Amblard

Supervisor:
Alessandra Russo

Helper:
David Tuckey

Second Marker:
Krysia Broda

ASG Author:
Mark Law

Monday 25th May, 2020

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

Contents

Chapter 1 **Introduction**

1 General Problem

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

2 Specific Problem

Given a brief paragraph of text, for example a short story aimed at young children, we should be able to provide a summary in multiple sentences. The goal here is to extract the most important information from all the relevant sentences, and from this generate one or more grammatically-correct summaries.

This should be accomplished with the use of Answer Set Grammars (ASG), although pre- and/or post-processing may be required. Apart from the ASG scripts, all code should be written in Python.

3 Objectives

In all of the problem cited above, should keep the below objectives in mind.

Objective 1 (Proper Nouns)**.** When our program encounters a proper noun, it must be able to relate it to other words in the text, by knowing what type of word it is and also having a bit of background knowledge about it. For example, in the text "He went to Paris yesterday. He arrived in France, then he travelled across the city to his hotel.", our system would need to relate the words "Paris", "France" and "city", but keep "hotel" as a distinct entity.

Objective 2 (Periphrasis)**.** At the very least, our solution must be able to link synonyms, such as "house" and "home". On a more advanced note, it might hopefully also have enough semantic analysis capabilities to know that "To the left of the goat is a pig." and "To the right of the pig is a goat." mean the same thing.

Objective 3 (Negation)**.** We should try to make our program able to interpret sentences which involve negation, which can be tricky for a semantic analysis tool. On a basic level we should be able to relate a negated word with its antonym, for example "not happy" with "sad" or "unhappy".

Objective 4 (Event Chronology)**.** Understanding the order of events in a story extremely important, as getting this wrong could make an entire summary incorrect. In order to achieve a good understanding of this, our program must be able to interpret connectives as well as time markers. A good example for the former would

be “He washed his hands and then ate”. As a stretch objective, if a text contains the string “Today is Monday. In two days she has an exam.”, our solution should be context-aware and know that the exam is held on a Wednesday.

Objective 5 (Sentence Subject)**.** Aside from our program being able to identify the subject of a basic sentence, it should also be able to understand which sentences have the same subject, so that it can link actions and descriptions with subjects. In passage “John likes music. He has a guitar”, our solution should be able to pick up on the fact that John has a guitar.

Objective 6 (Adjectives And Adverbs)**.** When a sentence contains adjectives and/or adverbs, the program should be able to know what subjects or actions they apply to. The following is an extremely difficult but all-encompassing example: “The white cat quickly jumped over the brown squirrel who was slowly eating a fresh cobnut”.

Objective 7 (Context)**.** A solution must be highly context-aware, and use the correct definition of a word depending on the context. For example, the word “address” can either be a verb meaning “to speak to”, or the location of a physical place.

Chapter 2 **Background**

1 Summarization

1.1 Definition

As described by the author in [1], a summary is a way of providing a large part of the information contained in one or more original passages, using at most half of the text. Summaries can be grouped into one of the two following categories:

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

A different way to group summaries is the following:

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

Yet another way [2] is shown below, with examples given in Figure 2.1:

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

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

(a) *Indicative* summary

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

(b) *Informative* summary

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

2 Syntactic Parsing

Given a text, *syntactic parsing* [3] describes the process of tokenization, whereby the grammatical link between parts of speech is established. Two of the most prominent syntactic parsers are **CoreNLP** (developed by Stanford) and **spaCy**.

This tokenization can then be visualized in the form of a tree, which makes it possible to identify different parts of speech, such as nouns, verbs and adjectives. An example is shown in Table 2.1 for both of the mentioned parsers. Appendix 5 shows how to interpret position of speech (POS) tags.

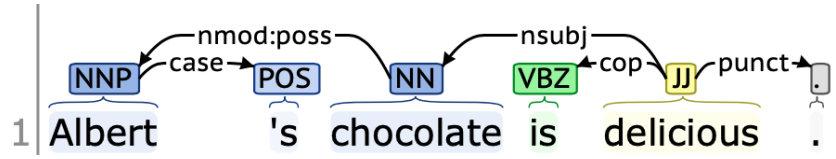
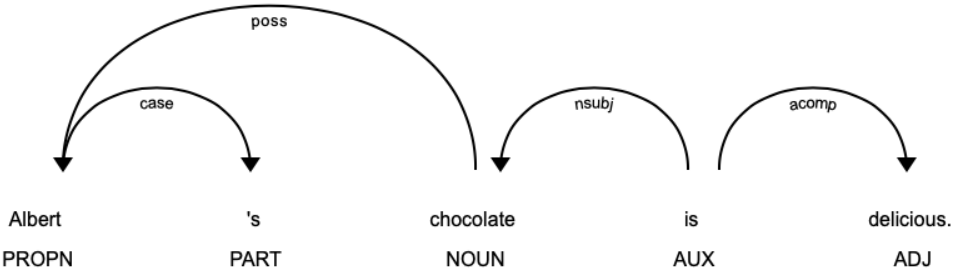
CoreNLP	
spaCy	

Table 2.1: Semantic parsing example for a basic sentence

However, the English language can often be ambiguous and highly context-dependent, meaning that multiple *parse trees* for the same sentence could emerge. Consider the following two sentences [4]:

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

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

Therefore accuracy is very important for *syntactic parsing* [5].

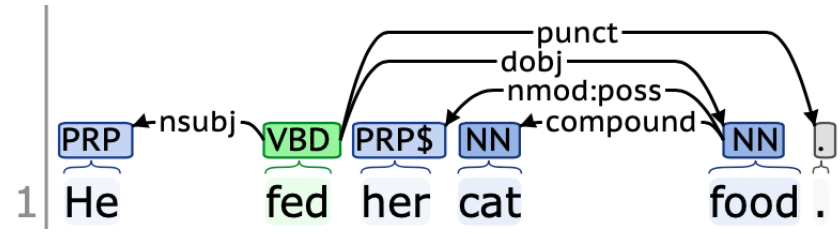
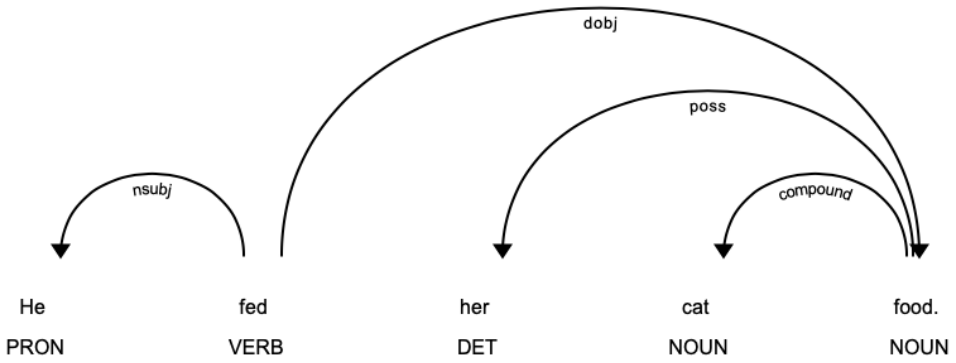
CoreNLP	
spaCy	

Table 2.2: Semantic parsing example for an ambiguous sentence

3 Logic Programming

Definition 1 (Term [6])**.** 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 [6])**.** An *atomic formula* (or *atom*) has the form $P(t_1, t_2, ..., t_k)$, where P is a k-ary predicate (boolean function) symbol and the t_i are terms.

3.1 Answer Set Programming

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

ASP is a declarative first-order (predicate) logic language whose aim is to solve hard search problems [7]. It is built upon the idea of stable model (answer set) semantics, returns answer sets when asked for the solution to a problem.

In ASP, a *literal* is an *atom* a or its negation *not* a (we call this negation as a failure). ASP programs are composed of a set of *normal rules*, whose head is a single *atom* and body is a conjunction of *literals* [8].

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

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

Definition 3 (Safety [8])**.** A variable in a rule is said to be *safe* if it occurs in at least one positive *literal* (i.e. the b_is in the above rule) in the body of the rule.

Definition 4 (Herbrand Base [8])**.** The *Herbrand base* of a program P, denoted HB_P, is the set of variable-free (*ground*) *atoms* that can be formed from predicates and *constants* in P . The subsets of HB_P are called the (Herbrand) *interpretations* of P.

Definition 5 (Satisfiability [8])**.** Given a set A, a *ground normal rule* of P is *satisfied* if the head is in A when all positive *atoms* and none of the negated *atoms* of the body are in A, that is when the body is *satisfied*. A *ground constraint* is *satisfied* when the body is not *satisfied*.

Definition 6 (Reduct [8])**.** Given a program P and an *Herbrand interpretation* $I \subseteq HB_P$, the *reduct* P^I is constructed from the grounding of P in three steps:

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

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

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

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

3.2 Context-Free Grammars

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

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

Definition 10 (Parse Tree [8])**.** Let GCF be a CFG. A *parse tree* PT of GCF for a given string consists of a node $node(PT)$, a list of *parse trees*, called *children* and denoted $children(PT)$, and a rule $rule(PT)$, such that:

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

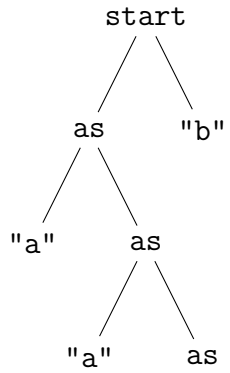
Definition 11 (Trace [8])**.** We can represent each node n in a *parse tree* by its *trace*, $trace(n)$, through the tree. The *trace* of the root is the empty list `[]`; the i$^{\text{th}}$ child of the root is `[i]`; the j$^{\text{th}}$ child of the i$^{\text{th}}$ child of the root is `[i, j]`, and so on.

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

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

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

(b) *Parse tree* table for aab



(c) *Parse tree* graph for aab

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

3.3 Answer Set Grammars

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

Definition 12 (Answer Set Grammar [8])**.** An *annotated* production rule is of the form $n_0 \rightarrow n_1 ... n_k \ P$ where $n_0 \rightarrow n_1 ... n_k$ is an ordinary CFG production rule and P is an *annotated* ASP program, where every *annotation* is an integer from 1 to k.

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

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

Figure 2.3: Example of an ASG

Definition 13 (Parse Tree Program [8])**.** Let G be an ASG and PT be a *parse tree*. $G[PT]$ is the program $\{\ rule(n)@trace(n) \mid n \in PT\ \}$, where for any production rule $n_0 \rightarrow n_1 ... n_k \ P$, and any trace t, $PR@t$ is the program constructed by replacing all annotated atoms $a@i$ with the atom $a@t++[i]$ and all *unannotated atoms* a with the atom $a@t$.

Definition 14 (Conforming Parse Tree [8])**.** Given a string *str* of terminal nodes, we say that $str \in \mathcal{L}(G)$ (*str conforms* to the language of G) if and only if there exists a parse tree PT of G for *str* such that the program $G[PT]$ is *satisfiable*. For such a PT, every single rule in the language must be satisfied (see Definition 5).

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

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

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

3.4 Learning Answer Set Grammars

Given an incomplete ASG, it is possible to learn the complete grammar by induction (which uses ILASP), as long as we provide some *positive examples* (strings which should conform to the language) and/or *negative examples* (strings which must not), as well as a *hypothesis space* and usually some *background* information. Note that the *background* is only used for "global" knowledge, such as defining what is a number, or how to increment one. [8]

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

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

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

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

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

(a) Input incomplete program

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

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

as -> {
 size(0).
}

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

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

(b) Output learned program

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

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

4 Neural Networks

4.1 Encoder-Decoder

TODO

Chapter 3 Contributions

1 Architecture Overview

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

Figure 3.1: Main Pipeline

2 Initial Motivation

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

In contrast, using logic means that we can hard-code this definition directly into our program. By carefully constructing its structure, we can get results with just a short list of rules, and know that it will always produce a complete and valid output.

3 Example

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

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

(a) Story of Peter Little

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

(b) Reference summaries

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

Chapter 4 Preprocessor

1 Overview

In order to prepare the story to be parsed and summarized by SumASG, we have created the Preprocessor. As we will loose information in summary anyway, it is completely acceptable to... which steps are shown below in Figure 4.1.

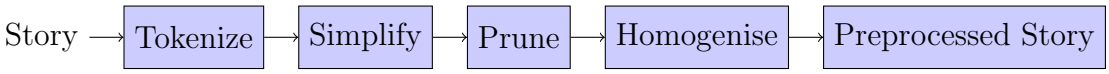


Figure 4.1: Preprocessor Steps

2 Tokenization And Simplification

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

2.1 Punctuation

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

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

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

2.2 Individual Word Transformations

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

2.2.1 Acronyms

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

2.2.2 Contractions

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

2.2.3 Adverbs

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

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

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

2.2.4 Determiners

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

2.2.5 Possessive Pronouns, Interjections And Prepositions

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

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

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

2.3 Clause Transformations

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

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

Examples of the transformations applied to different types of clauses can be seen below in Figure 4.2.

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 4.2: Examples of the splitting of multi-clause sentences

2.3.1 Hypernym Substitution

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

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

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

TODO expand?

2.4 Case And Proper Nouns

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

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

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

2.4.1 Pronoun Substitution

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

If a story contains exactly one distinct singular proper noun and then uses either "he" or "she", then it is safe to assume that this pronoun refers the aforementioned proper noun. The same can be said about plural proper nouns and the pronoun "they". To clarify this, an example is shown in Figure 4.3.

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 4.3: Example of substituting pronouns with proper nouns

2.5 Example

The result of running the simplification transformations is illustrated below in Figure 4.4. Please refer to Chapter 3 for the original story.

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

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

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

(b) Sentences after applying transformations

Figure 4.4: Example of applying simplification to the story of Peter Little

3 Sentence Pruning And Homogenization

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

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

3.1 Word Similarity

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

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

3.2 Sentence Similarity And Pruning

3.2.1 Sentence Similarity

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

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

3.2.2 Pruning

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

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

3.3 Synonyms And Homogenization

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

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

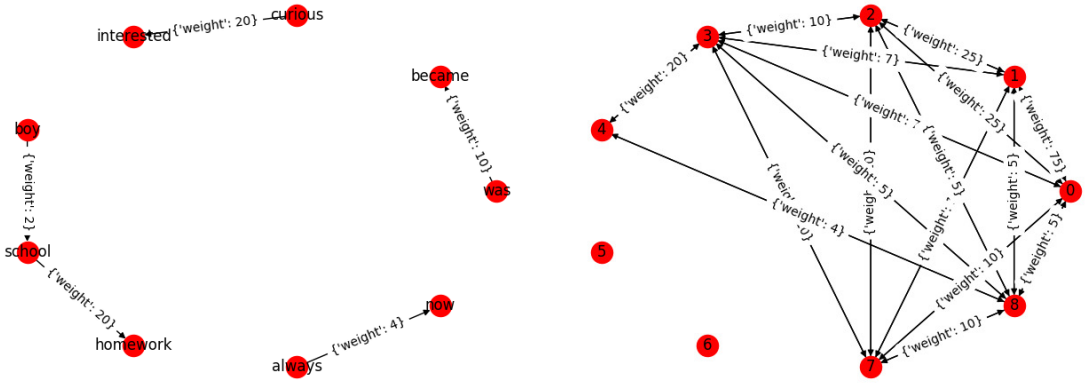
This is what we call *story homogenization*, and it helps SumASG link words that would otherwise be considered completely different tokens in the story.

3.4 Example

An example is shown below in Figure 4.5 for the story of Peter Little, where the weight between two nodes denotes *similarity*. In this example, we consider words to be synonyms whenever their *similarity* is at least 20. The labels of the nodes (starting from 0) for sentence *similarity* correspond to the index of the sentence in the simplified story (see Subsection 2.5).

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

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



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

(a) Text relationship maps

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

(b) Sentences ordered by importance

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

(c) Homogenized story

Figure 4.5: Example of applying sentence pruning and homogenization to the simplified story of Peter Little

4 Expandability

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

4.1 Negation

In order to support negation, we would need to modify the structure of SumASG's internal representation (see Chapter 5). However, to achieve a better semantic understanding in SumASG*, we could add some more simplification logic to the Preprocessor.

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

4.2 Lists

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

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

Chapter 5 ASG

1 Overview

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

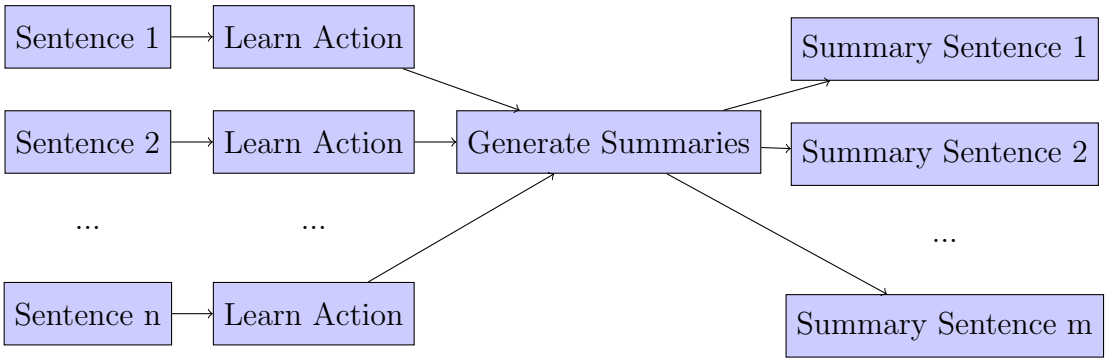


Figure 5.1: ASG Steps

2 Internal Representation

In order to model the structure of sentences the English language, we have created a CFG that has a similar hierarchy to that of an NLP parse tree. The ASG code for this general structure can be seen in Appendix A. 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 5.

2.1 Leaf Nodes

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

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

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

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

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

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

In Figure 5.2, 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 5.2: Predicates used for the leaf nodes in the internal representation

2.2 Non-Leaf Nodes

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

3 Learning Actions

We first need to convert the preprocessed story’s sentences from English into our internal structure. In other words, we need to learn about the *actions* described by the sentences in our story.

3.1 Formalization

We can formalize the task of learning an action as $\text{SumASG}_1(\underline{\text{CFG}},\underline{\text{BK}},\underline{\text{E}})$. 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.

TODO better formatting of formalization?

3.2 Implementation

3.2.1 Positive Example

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

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

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

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

3.2.2 Mode Bias

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

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

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

3.2.3 Running

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

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

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.

3.3.1 Learning Actions Individually

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

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

3.3.2 Cutting Out Rules

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

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

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

4 Generating Summary Sentences

5 Example

6 Expandability

TODO missing English structures **TODO speed**

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

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

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

Ideas: - use lots of simple/precise rules rather than complicated/general ones to minimize ss - keep rules as restricted as possible, when concept implemented over time add missing rules - to avoid having to add grammar constraints try and rely on grammar of input

- reduce search space using mode bias (simple example: 396-¿16, very complicated example: 9477-¿1044)

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

- for learning actions do one sentence at a time to minimize ss

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

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

Chapter 6 Post-Processing / Scoring

1 Overview

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

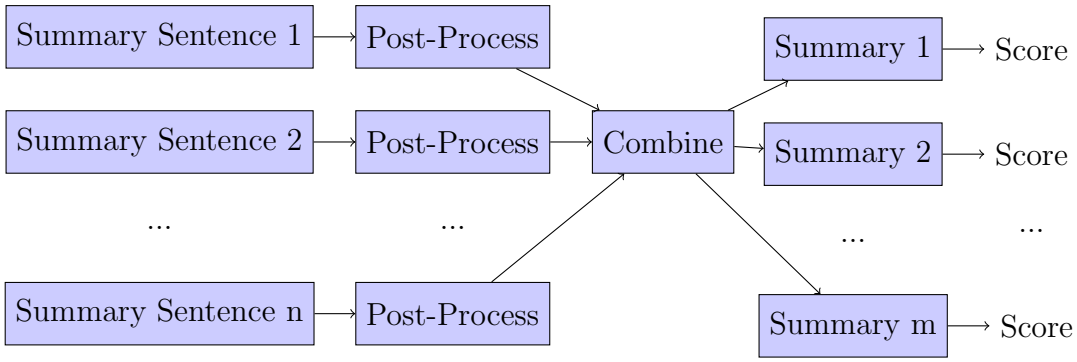


Figure 6.1: Post-Processing / Scoring Steps

2 Summary Creation

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

2.1 Post-Processing

2.1.1 Grammar

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

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

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

2.1.2 Complex Nouns

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

2.2 Combining

Depending on the length of the original story, we can envision and different number of sentences to be in the summary, as shown below in Table 6.1.

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

Table 6.1: Length of a summary depending on the number of sentences in the story

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

3 Scoring

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

3.1 TTR

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

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

3.1.1 Ignored Words

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

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

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

In Figure 6.2 is an example which illustrates this metric. The summary with the highest final score is considered to be the best. As you can see, there is a greater difference between the summaries when using TTR*, which takes into account the commonly-used building blocks of the English language.

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

(a) Story

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

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

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

Figure 6.2: Score computation (column headers ending with * pertain to TTR*)

4 Summary Selection

4.1 Proper Nouns

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

4.2 Top Summaries

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

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

4.3 Reference Summaries

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

In our implementation, we have chosen to use the BLEU score, which measures how closely the output given by a machine matches a text written by a human. If there exists a *top summary* whose BLEU score with one of the references is above a certain threshold, then we consider the summarization to be successful.

5 Example

TODO

6 Expandability

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

6.1 Grammatical Shortcomings

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

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

6.2 Better Summary Selection

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

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

Chapter 7 Evaluation

1 Reflection About Objectives

If we go back to the objectives from Chapter 1, we can see that some were fully achieved, while others were unfortunately only partly realized.

1.1 Successes

1.2 Limitations

TODO

2 Summarization Neural Network

2.1 General Idea

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

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

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

2.2 Datasets

In order to generate the required number of stories, we have used words from word-frequency.info. This database contains 5,000 individual English words, of which 1,001 are verbs, 2,542 nouns and 839 adjectives. We also make use of the 561 first names from National Records of Scotland.

2.3 Libraries

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

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

TODO define meronym/...

2.4 Story Structure

2.5 Training

2.6 Results

TODO

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

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

Chapter 8 Literature Review

1 Summarization Levels

Depending on how much text analysis is done, we identify three different levels of summarization [1]. Many current systems employ what is called a *hybrid approach*, combining techniques from different levels.

1.1 Surface Level

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

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

1.2 Entity Level

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

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

1.3 Discourse Level

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

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

2 Semantic Analysis Methods

2.1 Combinatory Categorial Grammar

In a paper from 2019 [10], 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.

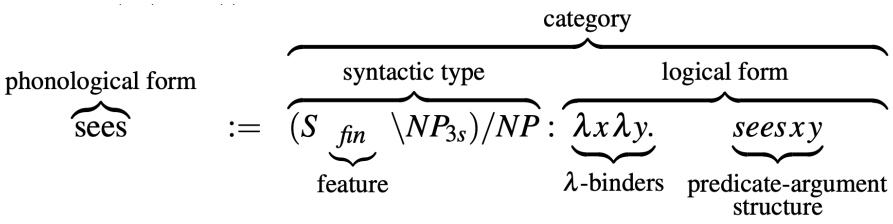
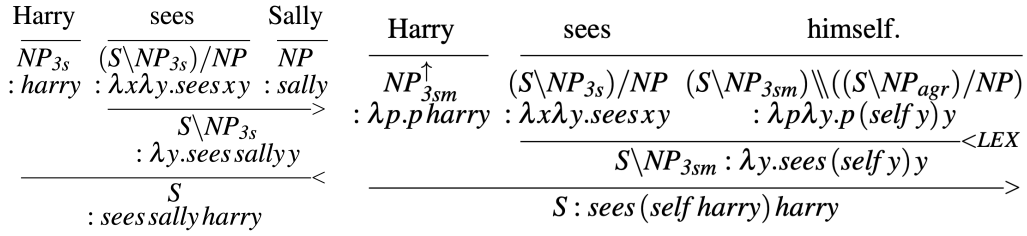


Figure 8.1: [10] Diagram explaining the mains terms used in CCG

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) [10] Basic clause

(b) [10] Reflexive transitive clause

Figure 8.2: Examples of derivations in CCG

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

3 Existing Approaches

3.1 MCBA+GA And LSA+TRM

In a paper by Yeh et al. [11], two different methods are put forward for text summarization. The first is the modified corpus-based approach (MCBA), which uses a score function as well as the *genetic algorithm*, while the second (LSA+TRM) utilizes *latent semantic analysis* (LSA) with the aid of a *text relationship map* (TSA).

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

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

Moreover, the *genetic algorithm* (GA) is used to obtain suitable weights, where a *chromosome* is defined by a set of values for all the features weights. Using the notions of *precision* (proportion of predicted positive cases that are correctly real positives) and *recall* (proportion of real positive cases that are correctly predicted positive) [12], 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.

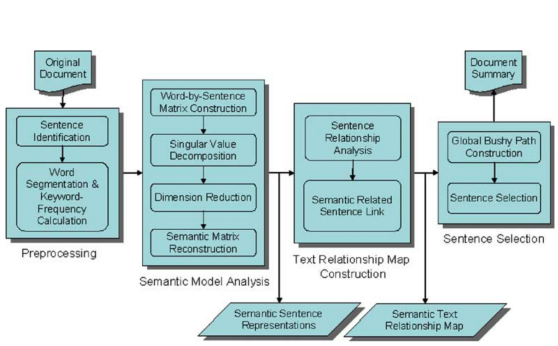
On the other hand, the LSA+TRM approach comprises four major steps: *preprocessing* (1), *semantic model analysis* (2), *text relationship map construction* (3) and *sentence selection* (4).

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

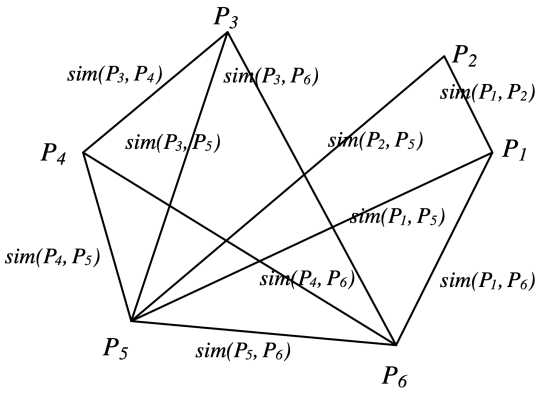
In step (2), a *word-by-sentence matrix* is computed on the scale of the entire document (or corpus). This gets factorized and reduced to leave out words which do not occur often, then turned into a *semantic matrix* linking words to their according relevance with each sentence.

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) [11] Overall process



(b) [11] Example of a text relationship map

Figure 8.3: LSA+TRM approach, diagrammatically

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

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

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

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

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

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

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

3.2 Lexical Chains

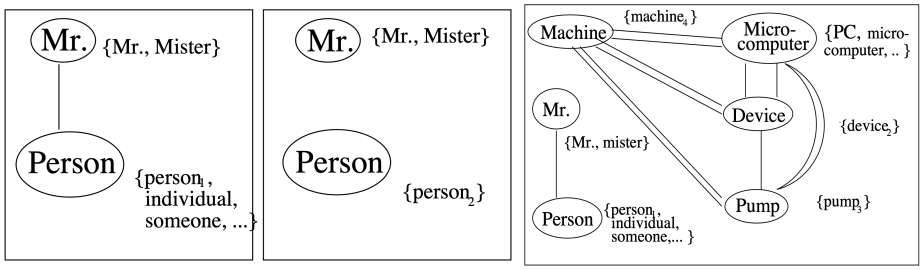
In a paper about *lexical chains* [13], the authors describe a method which relies on semantic links between words. The idea is that we establish chains of related words, in order to learn what a text is about.

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

> **Mr.** Kenny is the **person** that invented an anesthetic **machine** which uses **micro-computers** to control the rate at which an anesthetic is pumped into the blood. Such **machines** are nothing new. But his **device** uses two **micro-computers** to achieve much closer monitoring of the **pump** feeding the anesthetic into the patient. [13]

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: [13] Example of a *lexical chain* and its interpretations

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

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

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

4 Approach Categories

4.1 Statistical

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

The authors of the paper [14] envision what they call a *noisy-channel model*, which at the time of writing was limited to single sentence summarization. For the model, assume that there was at some point a (shorter) summary string s for the (longer) string t to summarize, from which optional words were removed. The idea is that optional details were added to produce t, and we want to know with what probability s contained this information given t. At this stage, there a three problems to solve:

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

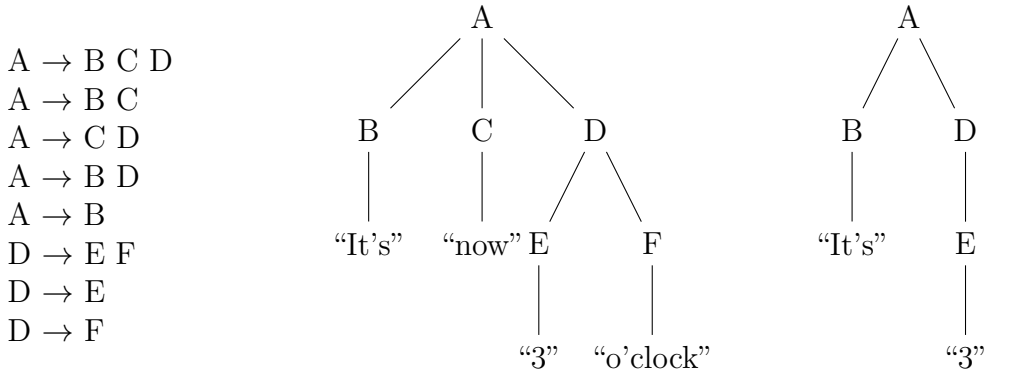
$P(s)$ to less grammatically correct strings, then it helps ensure that our final summary is well-formed.

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

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

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

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



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

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

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

4.2 Frame

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

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

In one of the original papers describing this approach [15], sentences are decomposed into different *affect states* and *affect links*. An *affect state* can either be a *mental state*, or a *positive* or *negative event* which may cause a change to a *mental state*. *Affect links* are then the transitions that explain the sequence of *affect states*.

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

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

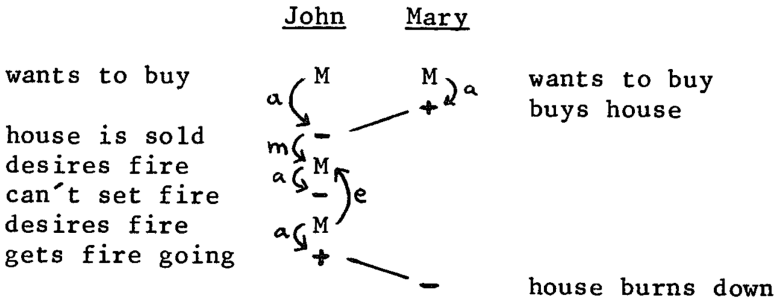


Figure 8.6: Example of a narrative in the frame approach

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

4.3 Symbolic

In the symbolic approach [16], meaning is expressed through logic, and the representation of a sentence is the combination of the meaning of each of its individual components. CCG, as discussed in Subsection 2.1, uses a symbolic approach.

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

Besides CCG, another pertinent case study is that of the Montague Grammar [17]. 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 5), while the latter captures the type of a token (which can either be e for *entity* or t for *truth value*). For instance, the word "John" has *syntactic category* ProperN and *semantic type* e, while for the verb "walks" these are respectively VP and $e \to t$.

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

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

One of the criticisms with this approach [16] is that it is domain-specific and not easily scalable. However more recent work (as with CCG) has shown that the latter issue is not necessarily the case any more, as we now have more powerful parsers. Should we choose to follow a symbolic approach, we shall hope to prove the former issue no longer relevant, once we have moved up from simpler examples.

5 Comparison With Our Approach

TODO

Appendix A. **POS Tags**

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

Table A.1: [18] List of position of speech (POS) tags

Appendix B. **ASG**

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

1 Common Grammar

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

s_group −> {
  count(0).
}

s_group −> s_group s ". " {
  count(X+1) :− count(X)@1.

  % Reject output summaries with duplicate sentences
  sentence(X,V,O,S) :− output(_,V,O,S)@2, count(X).
  sentence(X,V,O,S) :− sentence(X,V,O,S)@1.
  :− sentence(X1,V,O,S), sentence(X2,V,O,S), X1 != X2.
}

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 {
```

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

```
80    verb(comp(N1,N2),comp(T1,base)) :- verb(N1,T1)@1, verb(N2,base)@3.
81    object(N,D,A) :- object(N,D,A)@4.
82  }
83
84  np -> np rb {
85    object(N,D,A) :- object(N,D,0)@1, adj_or_adv(A)@2.
86  }
87
88  np -> np rb {
89    object(N,D,conjunct(A1,A2)) :- object(N,D,A1)@1, adj_or_adv(A2)@2.
90    :- object(N,D,conjunct(A,A)).
91  }
92
93  np -> np rp {
94    object(N,D,A) :- object(N,D,0)@1, adj_or_adv(A)@2.
95  }
96
97  np -> np rp {
98    object(N,D,conjunct(A1,A2)) :- object(N,D,A1)@1, adj_or_adv(A2)@2.
99    :- object(N,D,conjunct(A,A)).
100 }
101
102 np -> nn {
103   subject(N,0,0) :- noun(N)@1.
104   object(N,0,0) :- noun(N)@1.
105 }
106
107 np -> nns {
108   subject(N,0,0) :- noun(N)@1.
109   object(N,0,0) :- noun(N)@1.
110 }
111
112 np -> nnp {
113   subject(N,0,0) :- noun(N)@1.
114   object(N,0,0) :- noun(N)@1.
115 }
116
117 np -> nnps {
118   subject(N,0,0) :- noun(N)@1.
119   object(N,0,0) :- noun(N)@1.
120 }
121
122 np -> prp {
123   subject(N,0,0) :- noun(N)@1.
124   object(N,0,0) :- noun(N)@1.
125 }
126
127 np -> rb {
```

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

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

```
224   object(conjunct(N1,N2),0,0) :- noun(N1)@1, noun(N2)@3.
225   :- subject(conjunct(N,N),0,0).
226   :- object(conjunct(N,N),0,0).
227 }
228
229 np -> rb "and " rb {
230   subject(0,0,conjunct(A1,A2)) :- adj_or_adv(A1)@1, adj_or_adv(A2)@3.
231   object(0,0,conjunct(A1,A2)) :- adj_or_adv(A1)@1, adj_or_adv(A2)@3.
232   :- subject(0,0,conjunct(A,A)).
233   :- object(0,0,conjunct(A,A)).
234 }
235
236 np -> jj {
237   object(0,0,A) :- adj_or_adv(A)@1.
238 }
239
240 np -> jj "and " jj {
241   object(0,0,conjunct(A1,A2)) :- adj_or_adv(A1)@1, adj_or_adv(A2)@3.
242   :- object(0,0,conjunct(A,A)).
243 }
244
245 np -> jj rb {
246   subject(0,0,conjunct(A1,A2)) :- adj_or_adv(A1)@1, adj_or_adv(A2)@1.
247   object(0,0,conjunct(A1,A2)) :- adj_or_adv(A1)@1, adj_or_adv(A2)@1.
248   :- subject(0,0,conjunct(A,A)).
249   :- object(0,0,conjunct(A,A)).
250 }
251
252 np -> dt nn {
253   subject(N,D,0) :- det(D)@1, noun(N)@2.
254   object(N,D,0) :- det(D)@1, noun(N)@2.
255 }
256
257 np -> dt nns {
258   subject(N,D,0) :- det(D)@1, noun(N)@2.
259   object(N,D,0) :- det(D)@1, noun(N)@2.
260 }
261
262 np -> jj nns {
263   subject(N,0,A) :- adj_or_adv(A)@1, noun(N)@2.
264   object(N,0,A) :- adj_or_adv(A)@1, noun(N)@2.
265 }
266
267 np -> jj nnp {
268   subject(N,0,A) :- adj_or_adv(A)@1, noun(N)@2.
269   object(N,0,A) :- adj_or_adv(A)@1, noun(N)@2.
270 }
271
```

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

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

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

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

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

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

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

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

```

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

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

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

2 Task: SumASG$_1$

```
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    subject :- subject(S_N,S_D,S_A)@1.
5    :- not subject.
6    object :- object(S_N,S_D,S_A)@2.
7    :- not object.
8  }
9
10 #modeh(action(verb(const(verb_name),const(verb_form)), subject(const(noun),
       ↪ const(det),const(adj_or_adv)), object(const(noun),const(det),const(
       ↪ adj_or_adv))):[4].
11 #modeh(action(verb(const(verb_name),const(verb_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].
12 #modeh(action(verb(const(verb_name),const(verb_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].
13 #modeh(action(verb(const(verb_name),const(verb_form)), subject(conjunct(const(
       ↪ noun),const(noun)),const(det),const(adj_or_adv)), object(const(noun),const
       ↪ (det),const(adj_or_adv))):[4].
14 #modeh(action(verb(const(verb_name),const(verb_form)), subject(const(noun),
       ↪ const(det),const(adj_or_adv)), object(conjunct(const(noun),const(noun)),
       ↪ const(det),const(adj_or_adv))):[4].
15 #modeh(action(verb(const(verb_name),const(verb_form)), subject(const(noun),
       ↪ const(det),const(adj_or_adv)), object(const(noun),conjunct(const(det),
       ↪ const(det)),const(adj_or_adv))):[4].
16 #modeh(action(verb(const(verb_name),const(verb_form)), subject(const(noun),
       ↪ const(det),const(adj_or_adv)), object(const(noun),conjunct(const(prep),
       ↪ const(det)),const(adj_or_adv))):[4].
17
18 #modeh(action(verb(comp(const(verb_name),const(verb_name)),comp(const(
       ↪ main_verb),const(aux_verb))), subject(const(noun),const(det),const(
       ↪ adj_or_adv)), object(const(noun),const(det),const(adj_or_adv)))):[4].
19 #modeh(action(verb(comp(const(verb_name),const(verb_name)),comp(const(
       ↪ main_verb),const(aux_verb))), subject(const(noun),const(det),const(
       ↪ adj_or_adv)), object(const(noun),const(det),conjunct(const(adj_or_adv),
```

```
    ↪ const(adj_or_adv))))):[4].
20 #modeh(action(verb(comp(const(verb_name),const(verb_name)),comp(const(
    ↪ main_verb),const(aux_verb))), subject(const(noun),const(det),conjunct(
    ↪ const(adj_or_adv),const(adj_or_adv))), object(const(noun),const(det),const(
    ↪ adj_or_adv)))):[4].
21 #modeh(action(verb(comp(const(verb_name),const(verb_name)),comp(const(
    ↪ main_verb),const(aux_verb))), subject(conjunct(const(noun),const(noun)),
    ↪ const(det),const(adj_or_adv)), object(const(noun),const(det),const(
    ↪ adj_or_adv)))):[4].
22
23 #bias(":- head(holds_at_node(action(verb(_,_),subject(0,_,_),object(_,_,_)),var__(1))
    ↪ ).").
24 #bias(":- head(holds_at_node(action(verb(_,_),subject(_,_,_),object(0,0,0)),var__(1)
    ↪ )).").
25
26 #bias(":- head(holds_at_node(action(verb(_,_),subject(_,_,_),object(conjunct(V,V),
    ↪ _,_)),var__(1))).").
27 #bias(":- head(holds_at_node(action(verb(_,_),subject(_,_,_),object(conjunct(_,0),_,
    ↪ _)),var__(1))).").
28 #bias(":- head(holds_at_node(action(verb(_,_),subject(_,_,_),object(conjunct(0,_),_,
    ↪ _)),var__(1))).").
29
30 #bias(":- head(holds_at_node(action(verb(_,_),subject(_,_,_),object(_,_,conjunct(V,
    ↪ V))),var__(1))).").
31 #bias(":- head(holds_at_node(action(verb(_,_),subject(_,_,_),object(_,_,conjunct(_
    ↪ ,0))),var__(1))).").
32 #bias(":- head(holds_at_node(action(verb(_,_),subject(_,_,_),object(_,_,conjunct(0,_
    ↪ ))),var__(1))).").
33
34 #bias(":- head(holds_at_node(action(verb(_,_),subject(conjunct(V,V),_,_),object(_,
    ↪ _,_)),var__(1))).").
35 #bias(":- head(holds_at_node(action(verb(_,_),subject(conjunct(_,0),_,_),object(_,_,
    ↪ _)),var__(1))).").
36 #bias(":- head(holds_at_node(action(verb(_,_),subject(conjunct(0,_),_,_),object(_,_,
    ↪ _)),var__(1))).").
37
38 #bias(":- head(holds_at_node(action(verb(_,_),subject(_,_,conjunct(V,V)),object(_,
    ↪ _,_)),var__(1))).").
39 #bias(":- head(holds_at_node(action(verb(_,_),subject(_,_,conjunct(_,0)),object(_,_,
    ↪ _)),var__(1))).").
40 #bias(":- head(holds_at_node(action(verb(_,_),subject(_,_,conjunct(0,_)),object(_,_,
    ↪ _)),var__(1))).").
41
42 #bias(":- head(holds_at_node(action(verb(_,_),subject(_,_,_),object(_,conjunct(V,V
    ↪ ),_)),var__(1))).").
43 #bias(":- head(holds_at_node(action(verb(_,_),subject(_,_,_),object(_,conjunct(_,0),
    ↪ _)),var__(1))).").
44 #bias(":- head(holds_at_node(action(verb(_,_),subject(_,_,_),object(_,conjunct(0,_),
```

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

3 Task: SumASG$_2$

```
1   s -> np vp {
2     subject :- subject(S_N,S_D,S_A)@1.
3     :- not subject.
4     object :- object(S_N,S_D,S_A)@2.
5     :- not object.
6
7     summary(0, V, S, O) :- action(_, V, S, O).
8
9     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)).
10    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))).
11
12    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)).
13    summary(4, V, S, object(0,0,conjunct(A1,A2))) :- action(I1, V, S, object(_,_,A1
      ↪ )), action(I2, V, S, object(_,_,A2)), A1 != A2, A1 != 0, A2 != 0, I1 < I2.
14
15    summary(5, V, S, object(conjunct(N1,N2),0,0)) :- action(I1, V, S, object(N1,0,_
      ↪ )), action(I2, V, S, object(N2,0,_)), N1 != N2, N1 != 0, N2 != 0, I1 < I2.
16    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.
17    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.
```

```
18
19  summary(8, V1, subject(N, D1, conjunct(A1, A2)), object(0, 0, A3)) :- action(_
        ↪ , V1, subject(N, D1, A2), object(0, 0, A3)), action(_, V2, subject(_, 0, 0),
        ↪  object(N, D2, A1)), A1 != A3.
20  summary(9, V1, subject(N, D1, conjunct(A1, A2)), object(0, 0, A3)) :- action(_
        ↪ , V1, subject(N, D1, A2), object(0, 0, A3)), action(_, V2, subject(_, 0, 0),
        ↪  object(N, D2, conjunct(A1, _))), A1 != A3.
21  summary(10, V1, subject(N, D1, conjunct(A1, A2)), object(0, 0, A3)) :- action
        ↪ (_, V1, subject(N, D1, A2), object(0, 0, conjunct(A3,_))), action(_, V2,
        ↪ subject(_, 0, 0), object(N, D2, A1)), A1 != A3.
22
23  summary(I, V, S, object(conjunct(N1,N2),D,A)) :- summary(I, V, S, object(
        ↪ conjunct(conjunct(N1,N2),N3),D,A)).
24  summary(I, V, S, object(conjunct(N1,N2),D,A)) :- summary(I, V, S, object(
        ↪ conjunct(N1,conjunct(N2,N3)),D,A)).
25  summary(I, V, S, object(conjunct(N2,N3),D,A)) :- summary(I, V, S, object(
        ↪ conjunct(conjunct(N1,N2),N3),D,A)).
26  summary(I, V, S, object(conjunct(N2,N3),D,A)) :- summary(I, V, S, object(
        ↪ conjunct(N1,conjunct(N2,N3)),D,A)).
27  summary(I, V, S, object(conjunct(N1,N3),D,A)) :- summary(I, V, S, object(
        ↪ conjunct(conjunct(N1,N2),N3),D,A)).
28  summary(I, V, S, object(conjunct(N1,N3),D,A)) :- summary(I, V, S, object(
        ↪ conjunct(N1,conjunct(N2,N3)),D,A)).
29
30  summary(I, V, S, object(N,D,conjunct(A1,A2))) :- summary(I, V, S, object(N,
        ↪ D,conjunct(conjunct(A1,A2),A3))).
31  summary(I, V, S, object(N,D,conjunct(A1,A2))) :- summary(I, V, S, object(N,
        ↪ D,conjunct(A1,conjunct(A2,A3)))).
32  summary(I, V, S, object(N,D,conjunct(A2,A3))) :- summary(I, V, S, object(N,
        ↪ D,conjunct(conjunct(A1,A2),A3))).
33  summary(I, V, S, object(N,D,conjunct(A2,A3))) :- summary(I, V, S, object(N,
        ↪ D,conjunct(A1,conjunct(A2,A3)))).
34  summary(I, V, S, object(N,D,conjunct(A1,A3))) :- summary(I, V, S, object(N,
        ↪ D,conjunct(conjunct(A1,A2),A3))).
35  summary(I, V, S, object(N,D,conjunct(A1,A3))) :- summary(I, V, S, object(N,
        ↪ D,conjunct(A1,conjunct(A2,A3)))).
36
37  % Pick exactly one summary derivation for each sentence
38  0{output(I,V,S,O)}1 :- summary(I,V,S,O).
39  :- not output(_,_,_,_).
40
41  :- 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.
42  :- 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.
43  :- 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.
44  }
```

Bibliography

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

[2] 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.

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

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

[5] 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.

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

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

[8] 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.

[9] 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.

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

[11] 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.

[12] 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.

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

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

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

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

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

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