RUPRECHT-KARLS-
UNIVERSITÄT HEIDELBERG
ZUKUNFT SEIT 1386

Institut für Computerlinguistik Heidelberg

Forschungsmodul zu Seminar
Knowledge Acquisition and Inference in NLP

Sentence Transformative Inference Mapping for Python

Betreuerin:
Prof. Dr. Anette Frank

Verfasserin:
Julia Suter
Matrikelnummer 3348630

Masterstudiengang Computerlinguistik

7. April 2019

Table of Contents

1 Introduction

Natural language inference (NLI) describes the task of inferring whether a natural language hypothesis h follows from a given premise p. It has become an essential field of NLP research as it constitutes an important pillar in a wide range of tasks, including information retrieval, question answering, database completion, common-sense reasoning and natural language understanding in general. NLI is a challenging task as it requires knowledge about syntax and semantics as well as logic and reasoning. The variability of linguistic expression adds another layer of difficulty.

(1) *Premise*: Stimpy is a cat.
Hypothesis: Stimpy is not a poodle.

In previous work, NLI has been tackled in different ways. Some approaches incorporate an abstract semantic representation for premise and hypothesis, e.g. Mineshima et al. (2015), whereas others operate directly on the surface structure without the need for symbolic logical notation. A notable example of the latter is the NaturalLI system (Angeli and Manning, 2014) which is based on the natural logic framework NatLog (MacCartney and Manning, 2007, 2008, 2009a).

Here, we present a new natural language inference engine which – similar to NaturalLI – proves whether a set of premises entails a hypothesis by continuous generation of derived facts through alteration of the premises according to a manually engineered rule set. As derived facts are generated, their logical relation with the premise is tracked and adjusted as necessary. Thus, when a derived fact is found that matches the hypothesis, the final relation can be used to determine whether the inference is valid, invalid or unknown.

We named our inference engine *Sentence Transformative Inference Mapping for Python*, short Stimpy, in reference to the explanatory example used in Angeli and Manning (2014), which also served as an important test case during implementation (see Example 1). Although similar to NaturalLI, Stimpy differs in several important aspects. Most notably, we designed a general-purpose formal syntax for writing inference rules and we emphasized the syntactic correctness of derived facts. Furthermore, our engine is not limited to the mutation of an individual lexical item and does not need to follow a specific processing order. Finally, Stimpy is capable of combining several premises and deriving compound facts from them. Together, these features considerably expand the scope of potential applications of our engine.

In this report, we describe the implementation framework of Stimpy, we demonstrate a basic rule set for NLI, evaluate its performance on the FraCaS entailment test suite (35% accuracy), and discuss the strengths and weaknesses of our approach. We also examine the quality and potential use of the derived facts produced by our engine as intermediate results. Finally, we make suggestions for future work based on our engine, including its application to other natural language problems and the possibility of automated rule generation as a form of machine learning.

2 Rule Syntax

To allow transformation rules to be specified in a readable, accessible and scalable fashion, we devised a concise and intuitive rule syntax. Rules are written and maintained in a text file separate from the execution code and are loaded and parsed by STIMPY during execution. In this chapter, we describe the key elements of this syntax and explain the motivation behind it.

A rule consists of three parts: the condition set, the transition set, and the metadata (see Example 2). The condition set is composed from one or more conditions that have to be met by the sentence in order for the rule to be applied. The transition set consists of one or more transitions that are applied to the sentence if all the conditions are met. The condition set and transition set are separated by an arrow (`-->`). The metadata provides information about the rule, most importantly the relation between the original sentence and the transformed one. The beginning of the metadata is marked by a hashtag (`#`).

(2)
```
exists_hypernym(X) --> replace(X, hypernym(X))
# relation = forward entailment
```

2.1 Conditions

A condition describes a specific characteristic of a sentence that it either possesses or not, e.g. it contains a relative pronoun. Upon evaluation a Boolean value is returned, indicating whether the condition is met or not. Conditions consist of either a predicate (see Example 3) or an equation for which the left-hand side is a predicate and the right-hand side is a string (see Example 4).

2.2 Predicates

A predicate is a function that takes a specific number of arguments and returns a value when evaluated. There are two types of predicates: those that return a Boolean value, and those that return a string. In Example 2, the predicate `exists_hypernym` takes 1 argument (i.e. *cat*). If a hypernym does in fact exist for the word `cat`, it returns True, otherwise False. In Example 4, the predicate is `hypernym`, which takes 1 argument (i.e. *cat*) and returns the hypernym for the given word. If this hypernym matches the right-hand side of the equation (i.e. *animal*), the condition is met, otherwise not. Note that both examples represent a condition, while `hypernym("cat")` on its own constitutes only a predicate but not a complete condition as its evaluation does not yield a Boolean value. In summary, both predicates and conditions can be evaluated. Predicates return a Boolean value or string while conditions always yield a Boolean value. A list of all predicates and their functionality can be found in the Appendix.

(3)
```
exists_hypernym("cat")
```

(4)
```
hypernym("cat") = "animal"
```

(5)
```
exists_hypernym(lemma(ADJA))
```

2.3 Arguments

Predicates take a specific number of arguments. These arguments can be strings, variables, or other predicates. Some predicates also take keyword arguments. An example for a string argument is given in Example 3 and Example 4. The use of a variable and predicate argument is shown in Example 5. `ADJA` is a variable, representing any token in the sentence. Refer to Chapter 2.8 for information about variable constraints and short notations.

`lemma(ADJA)` is an example for a predicate argument. It is passed on as an argument to the predicate `exists_hypernym`. When `lemma(ADJA)` is evaluated, the resulting string is used as an argument when processing the predicate `exists_hypernym`.

2.4 Transitions

A transition describes a transformation of the sentence that is applied when all conditions are met. Each transition consist of one of the following four functions: insert, replace, delete, or select. These functions are in form very similar to predicates, however there are two significant differences. Firstly, in addition to strings, variables and predicates, they can also take predefined tokens as arguments, both single ones and lists of them (see Example 6). Predefined tokens are marked by a trailing underscore. Secondly, the transition functions do not return a Boolean value or a string. Instead, they verify that the transformations can be applied and modify the sentence according to the given arguments.

2.5 Insertion

A token can be inserted at an absolute position or relative to the position of another (guiding) token, for instance the main verb. Per default the token is inserted right after the guiding token but the optional keyword argument `correction` allows to modify the position. In Example 6, the token is inserted 1 position before ROOT. The keyword argument `pos` (part-of-speech) gives the option of adjusting the form of the inserted token, for instance in order to to pluralize a noun.

(6)
```
insert(NOUN, 0, pos="NNS")
insert([THAT_, WHO_], ROOT, correction=-1)
```

2.6 Replacement

The replacement function takes the word that is to be replaced as a first argument and the replacement as a second. The optional keyword argument `pos` adjusts the form of the inserted token if necessary. In Example 7, the verb represented by the variable ROOT is replaced by the same verb in plural form.

(7)
```
replace(WORD, SYNONYM)
replace(ROOT, ROOT, pos=pluralize(x_pos(ROOT))
```

2.7 Deletion and Selection

The delete and select functions are counterparts. The delete function removes the specified token from the sentence, while select discards everything but the specified token. If the optional keyword argument `full_phrase` is set to `True` not only the given token but all the tokens in the sentence that are syntactically dependent on it will be affected. This is helpful when deleting or selecting a full phrase since only a single transition rule for the head of the phrase has to designed, not multiple ones for every single token the phrase contains. In future work, this functionality should be added to the insertion and replacement function as well in order to avoid difficulties as described in Example 19.

(8)
```
delete(ADJ)
select(ROOT, full_phrase="True")
```

2.8 Variables

Variables are an essential part of this syntax as they allow writing more general rules. We capitalized variable names so they can easily be spotted within a rule but this is a convention rather than a requirement by the rule parser; variable names can be chosen freely as long as they do not start with an exclamation mark or end with an underscore as such variable names are reserved for predefined tokens (see Chapter 2.4) and for negative variables.

In our system, variables represent a single token from the input sentence. Thus, when evaluating a condition containing a variable, it is tested for each token in the sentence whether the condition is met if it takes the place of the variable. This feature allows far more general rules to be written but also substantially increases the computing time because of the requirement to test all combinations. To ameliorate this issue, we made use of the fact that in practice most variables have specific constraints, meaning most tokens are fundamentally unsuitable to take the variables' place. Whilst such restrictions could be written as additional conditions within the same rule, we instead introduced an optional short notation for variable constraints. For example, if a condition demands that the sentence contains the relative pronoun *that* or *who*, the conventional notation requires three conditions to express this (see 9). During evaluation, one by one each token from the original sentence is put in place of the variable `REL_P` to test whether the conditions are true.

(9)
```
x_pos(REL_P) = "WDT"
& (lemma(REL_P) = "that" | lemma(REL_P) = "who")
```

(10)
```
REL_P:[x_pos = WDT; lemma in [that, who]]
```

Example 10 shows the short notation for the same expression. A variable can be given constraints about id, lemma, form, part-of-speech (u_pos and x_pos), head and dependency relation to head (deprel). The constraints are indicated by a colon immediately after the variable name, followed by square brackets that contain constraining

information, separated by semicolons. This notation does not only simplify rule writing and reading, it also reduces processing time as it allows filtering of unsuitable candidates before evaluating the condition. With this notation, only tokens that fit the constraints are considered as candidates to take the position of the variable.

Since this method proved to be very efficient, we also implemented its counterpart, i.e. defining what kind of tokens cannot be contained in the sentence. Example 11 shows how this notation could be employed to express *there cannot be a token with lemma* most*,* all *or* the *at the beginning of the sentence*. If there is a token matching these constraints, the conditions need not be evaluated as it is already certain that they can never be met. Note that `!OP` is not a real variable, although it may look like one. It cannot be filled by a token since a token with these characteristics cannot exist by definition. Thus, such *negated variables* can only appear as an argument for the predicate `is`, which is used to introduce variables and always yields True. Everything that is expressed by negated variables can also be expressed with the conventional rule notation (see Example 12). However, the result is less compact and takes more time to be processed since there is no pre-filtering and thus a larger number of conditions have to be checked.

(11)
```
!OP:[lemma in [most, all, the]; id=0]
```

(12)
```
not_exists(id=0, lemma="most")
& not_exists(id=0, lemma="all")
& not_exists(id=0, lemma="the")
```

Another special notation allows a variable to be defined as any token from a list of predefined tokens. In Example 13, the condition is evaluated for three different cases with the variable representing either *all*, *most* or *many*. This notation prevents the need for repetition of similar rules that vary only in one predefined token.

(13)
```
compare_operators(OP1, OP2:[ALL_, MOST_, MANY_]) = "stronger"
```

2.9 Logical Operators in Condition and Transition Sets

The symbol `&` represents the logical AND and is used to combine several conditions or transitions. The symbol | takes the function of a logical OR and can be used to express an alternative. Both symbols can be used in both condition and transition sets as often as necessary. By convention, AND binds stronger than OR. Brackets may be used to group conditions or transitions. Brackets around groups connected solely by `&` are optional.

The conditions in a condition set are evaluated group by group from left to right, starting from the innermost level. There are AND groups (bound by `&`) and OR groups (bound by |). As soon as there is a condition in an AND group that yields `False`, the entire group is considered `False` without further evaluation. As soon as there is a condition that yields `True` in an OR group, the entire group is considered `True` without further evaluation. This prevents unnecessary processing steps. Thus, we recommend

putting more restrictive conditions in the beginning and more general ones at the end of a group.

In the transition set, the logical OR gives the option of producing several transformed sentences. All OR groups are expanded in order to obtain a non-nested list of AND groups. When applying the transitions, each of these AND groups containing one or several transitions is applied to the original sentence separately, yielding as many transformed sentences as there are AND groups.

3 Implemented Rules

The rule set is a key element of the system. The rules define precisely which transitions are applied under which conditions. Furthermore, they provide information about the relation between the original sentence and its derived fact. The rules implemented here were partially inspired by the NaturalLI relation templates, inference patterns found in the FraCaS test suite (Cooper et al., 1996) and examples from the toy tasks described in Weston et al. (2015). In this chapter, we describe which inference rules were implemented and explain the notation for a selection of them. Whenever possible, we display one condition and transition per line so we can refer to them by line number.

3.1 Single-premise Rules

We implemented 41 rules for single-premise inference.

3.1.1 Replacement with Related Word

Lexical replacements are a straightforward and common pattern of logical inference. Replacing a word with a synonym leaves the derived fact equivalent to the initial. Hypernym replacements cause forward entailment, while hyponym replacements lead to reverse entailment. Antonyms cause alternation relations. These rules are relatively simple since they require only one condition confirming that a related word exists and one transition replacing it (see Example 14).

(14) *Replacing word with its hyponym*

```
exists_hypernym(WORD)
--> replace(WORD, hyponym(WORD))
# relation = reverse entailment
```

3.1.2 Removal of Words and Phrases

Another common inference pattern is the removal of an optional word or phrase from the original sentence, for example adverbs, attributive adjectives, prepositional phrases and non-defining relative clauses. Removing these elements leads to a forward entailment. Example 15 shows the rule for removing attributive adjectives. The first condition defines

the constraints for the adjective variable, the second condition prohibits the adjective from being a non-affirmative adjective (e.g. *former*) since removing those may lead to a different relation than forward entailment. For example *a former student* does not entail *a student*. The third condition dictates that the adjective is dependent on a noun, which makes it attributive rather than predicative.

(15) *Removing attributive adjective*

```
is(ADJA:[u_pos=ADJ; x_pos=JJ; deprel=amod])
& not(in_list(lemma(ADJA), "non_affirm_adj"))
& dependent(ADJA, NOUN:[u_pos=NOUN])

--> delete(ADJA)
# relation = forward entailment
```

The challenge with these rules is to avoid removing phrases and words that are essential for a syntactically correct sentence. The error analysis (see Chapter 5.2.2) revealed that this rule was the reason for incorrectly removed adjectives and prepositional phrases. For example *a really great tenor* was transformed into *a really tenor* or *John is from California* was reduced to *John is*. Future work would have to implement more strict rules for such cases.

3.1.3 Negation

Another essential operation is the negation of sentences, which naturally represents the relation *negation*. We implemented rules for negating auxiliary and modal verbs as well as main verbs. The rule for the latter case is shown in Example 16. The first two conditions confirm that the current sentence does not yet have a negation. The third condition checks that the main verb is not an auxiliary or modal verb, while the final one excludes passive sentences that also contain auxiliary verbs. Three transitions are applied if the conditions are met: the verb *do* is inserted in the form of the current main verb, *not* is included and the main verb is changed to base form.

(16) *Negation of main verb*

```
positive_sent()
& not_exists(lemma="no", deprel="det")
& is(ROOT:[deprel=ROOT; lemma not in [be, do, can, would,
  could, will, must, might, ought, may, should, shall]])
& not(is_passive())

--> insert(DO_, ROOT, correction=-1, pos=x_pos(ROOT))
& insert(NOT_, ROOT, correction=-1)
& replace(ROOT, ROOT, "VB")
# relation = negation
```

We did not implement rules to reverse the negation of auxiliary, modal or main verbs. However, we implemented rules for removing the *no* operator. Future work will have to extend the rule set with other kinds of negation.

3.1.4 Change in Sentence Structure

Some rules change the structure of the sentence. Example 17 shows how predicative noun phrases are turned into predicative adjectives. The first two conditions define the attributive adjective, the third confirms that the verb is *to be* or *to become*. The last two conditions express that the adjective has to be dependent on an attributive noun. If these conditions match, the full noun phrase is removed and the adjective is inserted after the main verb, turning e.g. *John is a fast runner* into *John is fast*.

(17) *Turning predicative noun phrase into predicative adjective*

```
is(ADJA:[u_pos=ADJ; x_pos in [JJ, JJR]; deprel=amod])
& not(in_list(lemma(ADJA), "non_affirm_adj"))
& is(ROOT:[deprel=ROOT; lemma in [be, become]])
& is(NOUN:[u_pos=NOUN; x_pos != WP; deprel=attr])
& dependent(ADJA, NOUN)

--> delete(NOUN, full_phrase="True")
& insert(ADJA, ROOT)
# relation = forward entailment
```

We also included rules for transforming a sentence with the structure *X does Y* into *There are X that do Y* or *There is an X that does Y*. These transformations required several rules in order to take into account both singular and plural sentences as well as possible operators. Example 18 shows how sentences with the operator *all* and *most* are transformed. The first condition describes the subject which has to be in plural form, the second confirms that there is no *there* dependent on the main verb which would suggest that the sentence already contains the desired structure. The final conditions ensure that there is an operator *all* or *most* that is dependent on the subject. If the condition matches, the operator is deleted and *There be* is inserted in the beginning of the sentence, with *be* transformed into the same form as the original main verb. Finally, the relative pronoun is inserted.

This rule transforms for instance *All cats eat salmon* into *There are cats who eat salmon*. Note that the rule in Example 18 implements a special case for the operators *all* and *most*: the operator has to be removed since *There are most cats who eat salmon* is not a syntactically correct sentence. For many other operators, this deletion is not required as sentences such as *There are few cats who eat salmon* are possible.

(18) *Transforming* All/most X do Y *into* There are X that do Y

```
is(SUBJ:[x_pos in[NNS, NNPS]; deprel=nsubj])
& has_no_dep_token(ROOT:[deprel=ROOT], lemma="there")
& is(OP:[lemma in [all, most]; deprel in [amod, det]])
& dependent(OP, SUBJ)

--> delete(OP)
& insert(THERE_, position=0)
& insert(BE_, position=1, pos=x_pos(ROOT))
& insert([WHO_, THAT_], ROOT, correction=-1)
# relation = forward entailment
```

Another rule that affects the sentence structure is the passive resolution rule (see Example 19). We only resolved passive structures that include an explicitly mentioned agent. The first condition checks whether there is a preposition *by* with the dependency relation *agent*, which is a clear indicator for passive sentences. The next two conditions assign the prepositional object to the variable `AGENT` and check whether it is dependent on *by*. The remaining conditons identify the action verb (in participle form), the subject and the auxiliary verb. If the sentence is indeed a passive sentence according to the conditions, the following transitions are executed: the preposition *by* and the action verb are deleted and the subject and agent are swapped. Thus, we transform a sentence such as *The car was stolen by the thief* into *The thief stole the car*. Note that this rule only affects the heads of the subject and agent, not the full noun phrases. While sufficient for our test case, this approach would need to be generalized in future work. We recommend extending the *insertion* and *replace* functions with the optional argument *full_phrase* as described in Chapter 2.7.

(19) *Transforming passive voice into active*

```
is(BY:[lemma=by; deprel=agent])
& is(AGENT:[deprel=pobj])
& dependent(AGENT, BY)
& is(ACTION:[deprel=ROOT])
& is(SUBJ:[deprel=nsubjpass])
& is(AUX_VERB:[deprel=auxpass; lemma=be])

--> delete(ACTION)
& delete(BY)
& replace(AUX_VERB, ACTION, pos=x_pos(AUX_VERB))
& replace(SUBJ, AGENT, pos=x_pos(AGENT))
& replace(AGENT, SUBJ, pos=x_pos(SUBJ))
# relation = equivalence
```

3.1.5 Operator Substitution

We implemented rules for operator replacements as it was done for NaturalLI. Depending on whether the substituting operator is stronger, weaker or equal to the original operator, the relation to the original sentence is reverse entailment, forward entailment or equivalence, respectively. For instance, *many* is a stronger operator than *few*, so replacing *few* with *many* results in a reverse entailment. Since there exist both singular operators (e.g. *every*) and plural operators (e.g. *all*), number adjustments for noun and verb have to be considered when implementing rules. Since we had to consider 3 options for operator comparison (stronger, weaker, equal), 3 options for number adjustment (singularization, pluralization, none) and for certain cases the difference between main verb and auxiliary verb, a total of 15 rules was required to implement all forms of operator replacement – and we only considered a small selection of operators. This shows how elaborate rule creation can be.

Example 20 shows how a singular operator is replaced by a weaker plural operator in the context of a main verb. The first two conditions introduce the variable `OP1` – which has to be a singular operator – and the noun it is dependent on. The third condition compares `OP1` to any of the operators listed in `OP2` and evaluates whether `OP2` is weaker. The fourth condition checks whether the required number adjustment is pluralization. The condition group spanning from line 6 to 9 checks whether the sentence could possibly contain a *There are* structure. If yes, the substituting operator cannot be *all* or *most* as already explained in context of Example 18. Finally, the last condition dictates that there cannot be an auxiliary verb since this specific rule does not cover such cases. If successfully evaluated, `OP1` is replaced by `OP2` and the subject and verb are pluralized. This rule transforms for instance *Every child loves to play outside* into *Many children love to play outside.*

(20) *Replacing a singular operator with a weaker plural operator*

```
1  is(OP1:[lemma in [each, every]])
2  & dependent(OP1, NOUN:[u_pos=NOUN])
3  & compare_operators(OP1,OP2:[ALL_, MOST_, MANY_, SOME_,
4    FEW_, EACH_, EVERY_, SEVERAL_]) = "weaker"
5  & number_adjustment(OP1,OP2) = "pluralize"
6  & (has_dep_token(ROOT:[deprel=ROOT],lemma='there')
7  & lemma(OP2) != "all"
8  & lemma(OP2) != "most"
9  | has_no_dep_token(ROOT, lemma="there"))
10 & has_no_dep_token(ROOT, deprel="aux")
11
12 --> replace(OP1,OP2)
13 & replace(NOUN, NOUN, pluralize(x_pos(NOUN))
14 & replace(ROOT, ROOT, pluralize(x_pos(ROOT)))
15 # relation = forward entailment
```

3.1.6 Implicative and Factive Verbs

We implemented rules for the implicative verbs *forget* and *manage* and for a small selection of factive verbs, e.g. *know* and *accept*. Example 21 shows how *He did not do his homework* can be derived from *He forgot to do his homework*. The conditions require the word *forget* and an open clausal complement. After successful evaluation, *to* is deleted and the verb is negated.

(21) *Transforming* forgot to X *into* did not X

```
is(IMPLICATIVE_VERB:[lemma in [forget]; deprel=ROOT])
& is(TO:[lemma = to; deprel=aux])
& is(VERB:[deprel=xcomp])

--> delete(TO)
& replace(IMPLICATIVE_VERB, DO_, pos=x_pos(IMPLICATIVE_VERB))
& insert(NOT_, IMPLICATIVE_VERB)
# relation = forward entailment
```

The rule for factive verbs (see Example 22) concludes for example that *He knows it is raining* entails *It is raining*. The conditions demand a subject, a factive verb and a causal complement starting with *that*. If this is the case, the subject, factive verb and *that* are deleted and the causal complement becomes the derived fact. Since *that* can be omitted, it would be useful to implement a syntax marker for optional conditions to avoid the duplication of the same rule, simply with the variable `THAT` removed.

These 3 implemented rules cover only a small fraction of possible inferences involving implicatives and factives. Future work should extend this set and take into account implication signatures (Nairn et al., 2006) and phrasal implicatives (Karttunen, 2012).

(22) *Removing factive verb and its subject*

```
is(FACTIVE_VERB:[deprel=ROOT;
  lemma in [accept, know, prove, forget]])
& is(THAT:[lemma = that; deprel=mark])
& is(VERB:[deprel=ccomp])
& is(SUBJ:[deprel=nsubj])
& dependent(SUBJ, FACTIVE_VERB)

--> delete(FACTIVE_VERB)
& delete(SUBJ)
& delete(THAT)
# relation = forward entailment
```

3.2 Multi-premise Rules

The multi-premise rules differ slightly in structure from single-premise rules since they have to incorporate conditions for multiple premises. These condition sets for the premises are separated by a dashed line and introduced by a string that names the current premise. The names can be chosen freely, although we decided to consistently use *P1* and *P2*. The final condition set is not associated with a specific premise but allows comparisons across premises. Technically, the multi-premise rules can contain conditions for any number of premises but we only implemented rules for two-premise inferences. The new fact is always derived from the first premise, meaning the transitions are applied to the first premise. Thus, it is reasonable to select the premise most similar to the desired output as first premise. We implemented 12 multi-premise rules.

3.2.1 Transitivity

The first multi-premise rule we implemented captures transitivity. If it is known that *Dogs are bigger than cats* and *Cats are bigger than mice*, we can infer that *Dogs are bigger than mice*. The condition sets *P1* and *P2* in Example 23 evaluate whether the premises contain a specific comparison structure. The last condition set confirms that the adjectives are identical and the subject of the second premise is the same as the comparison object of the first sentence. If so, the comparison object of the first premise is replaced by the one of the second premise.

(23) *Transitivity*

```
P1: is(ADJ1:[x_pos=JJR])
    & is(SUBJ1:[deprel=nsubj])
    & is(COMP_OBJ1:[deprel=pobj])
    ---
P2: ADJ2:[x_pos=JJR])
    & is(SUBJ2:[deprel=nsubj])
    & is(COMP_OBJ2:[deprel=pobj])
    ---
    lemma(ADJ1) = lemma(ADJ2)
    & lemma(SUBJ2) = lemma(COMP_OBJ1)

    --> P1: replace(COMP_OBJ1, COMP_OBJ2)
    # relation = forward entailment
```

3.2.2 So did he, neither did she

Inspired by inference problems in the FraCaS test suite, we implemented two multi-premise rules for structures such as *so did he* or *neither did she*. If it is known that *Billy loves pizza* and *So does Harry*, it can be inferred that *Harry loves pizza*. Example 25 shows the condition sets: if the first sentence is positive and the second contains a *so* and the verb *be* or *do*, the subject of the first sentence is replaced by the subject

of the second sentence. Note that we do not evaluate whether the second premise does indeed directly follow the first premise, which would be a necessary requirement for this rule. Future work should take into account the order of the premises so the rule could be extended by a condition such as `next_premise(P1) = P2`.

(24) *Inferring from* so did he

```
P1: is(SUBJ1:[deprel=nsubj])
    & positive_sent()
    ---
P2: is(SO:[deprel=advmod; lemma=so])
    & is(ROOT:[deprel=ROOT; lemma in [do,be]])
    & is(SUBJ2:[deprel=nsubj])
    ---

    --> P1: replace(SUBJ1, SUBJ2)
    # relation = forward entailment
```

3.2.3 Socrates is mortal

With the rules in this section, we aimed at solving inference problems such as the famous Socrates example: if it is known that *Every human is mortal* and that *Socrates is a human*, we can conclude that *Socrates is mortal*. If the first premise contains a subject with the operator *every*, the second premise includes a proper noun and an attribute, and the subject of the first premise and the attribute of the second are identical, the subject of the first premise is replaced with the subject of the second. Note that there is an additional rule covering cases where the subject of the second premise is not a proper noun since this may require the insertion of a determiner.

(25) *Socrates example*

```
P1: is(OP1:[deprel=det;lemma=every])
    & is(SUBJ1:[deprel=nsubj])
    & dependent(OP1, SUBJ1)
    & is(ROOT1:[deprel=ROOT])
    ---
P2: is(SUBJ2:[deprel=nsubj; x_pos=NNP])
    & is(ATTR2:[deprel=attr])
    & is(ROOT2:[deprel=ROOT])
    & dependent(ATTR2, ROOT2)
    ---
    lemma(SUBJ1) = lemma(ATTR2)

    --> P1: delete(SUBJ1, full_phrase="True")
    & insert(SUBJ2, position=0)
    # relation = forward entailment
```

3.2.4 Modus Ponendo Tollens

We implemented rules for covering the inference rule *Modus ponendo tollens*: if its known that ¬(A∧B) and A, we can conclude that ¬B. However, instead of deriving ¬B, we derive B but with the relation *negation*. This prevents the need for negating the sentence. The sentence can still be negated by the negation rule as a later inference step.

Example 26 shows how the rule is implemented. The second premise has to include an *either* and two options. If the subject and verb are identical to the ones in the first premise and the first premise includes one of the options, this option can be replaced by the other. For example, from *Bill either goes to Japan or to China* and *Bill goes to Japan*, we derive the fact *Bill goes to China*. Since the relation is *negation*, the inference is invalid. When negating the derived fact to *Bill does not go to China*, the inference becomes valid.

(26) *Modus ponendo tollens*

```
P1: is(SUBJ1:[deprel=nsubj])
    & is(ROOT1:[deprel=ROOT])
    & is(OPTION)
    ---
P2: is(EITHER:[deprel=preconj;lemma=either])
    & is(OPTION1:[deprel=pobj])
    & is(OPTION2:[deprel=conj])
    & dependent(OPTION2, OPTION1)
    & is(SUBJ2:[deprel=nsubj])
    & is(ROOT2:[deprel=ROOT])
    ---
    lemma(SUBJ1) = lemma(SUBJ2)
    & lemma(ROOT1) = lemma(ROOT2)
    & lemma(OPTION) = lemma(OPTION1)

    -> P1: replace(OPTION, OPTION2)
    # relation = negation
```

3.2.5 Modus Ponens and Modus Tollens

The final two multi-premise rules are simple implementations for *Modus ponens* and *Modus tollens*. Example 27 shows the rule for *Modus ponens*. The first premise requires an *if* and an adverbial clause modifier. If the if-clause is identical to the second premise (evaluated by comparing the full phrases for the verb of both clauses), the if-clause is removed from the first premise. From *If he is sick, he doesn't go to work* and *He is sick* it is inferred that *He doesn't go to work*. For *Modus tollens*, which represents that A → B and ¬B implies ¬A, we implemented a similar rule. As in Section 3.2.4, we derive A instead of ¬A and define the relation as *negation*. This makes it possible to determine that *You work hard* is not a valid inference from *If you work hard, you pass the exam* and *You don't pass the exam*.

(27) *Modus ponens*

```
P1: is(IF:[lemma=if])
    & is(CL_VERB1:[deprel=advcl])
    & is(ROOT1:[deprel=ROOT])
    ---
P2: is(ROOT2:[deprel=ROOT])
    ---
    same_phrase(CL_VERB1, ROOT2)

    --> P1: delete(CL_VERB1, full_phrase="True")
    # relation = forward entailment
```

4 Framework

In this chapter, we give a step-by-step tour through the STIMPY framework. For more details, we refer to the respective chapters and literature.

4.1 Rule Loading and Parsing

As a very first step, the rules are loaded from a text file and parsed into an easily accessible format. Then, the premises and hypothesis are parsed by the syntactic parser SpaCy (Honnibal et al., 2015), which returns the parsing results in CoNLL-format. Any other parser that produces the same format would work as an alternative.

4.2 Inference Tree

To track the intermediate steps taken by the system on the way to the solution we create a graph connecting each resulting sentence to the premise it was derived from. Since each input sentence can yield several output sentences, this results in a tree structure with the initial premise constituting the root (see Figure 1).

4.3 Polarity and Monotonicity

Monotonicity is a property of operators; they can be downward-, upward- or non-monotone on their arguments. Polarity is a property of lexical items determined by operators acting on them. By default, all tokens have *upward* polarity. While upward monotone operators preserve polarity of the items they influence, downward monotone operators reverse polarity. For instance, the operator *all* is downward monotone on the first argument, and upward monotone on the second argument (MacCartney and Manning, 2009b; Angeli and Manning, 2014). We compute the polarity for each token in the premise and hypothesis according to the operators they are connected by. Knowledge of the polarity is relevant for selecting the correct projection function when computing the relation between two sentences.

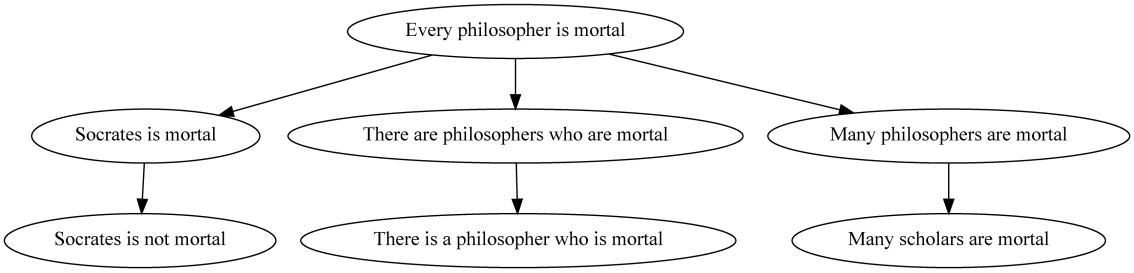

Figure 1: Inference tree example.

4.4 WordNet

Lexical substitutions are essential for many types of common-sense reasoning so we employ WordNet as a source of world knowledge (Miller, 1995). Since WordNet lookups are computationally expensive, we limit ourselves to a one-time retrieval of related words for each premise-hypothesis pair. For each noun, verb and adjective in the premise and hypothesis, we extract a set of synonyms, hyponyms and hypernyms. The search depth defines how many times the cycle of extracting related words for the already collected words is repeated. We do not apply any algorithm to select the most fitting related words for the given context although we recommend tackling this issue in future work.

4.5 Evaluation of Conditions

After the preprocessing steps, we move on to the actual rule application. For each rule, we check whether the conditions are met for the given premise. As a first step, all variables occurring in the rule must be filled with matching tokens from the premise, respecting possible constraints. If there are no fitting candidates for the variables, the rule is skipped. The same procedure is applied to any negated variables, although the outcome is reversed such that the detection of candidate causes the skipping of the rule. Naturally, a given variable may have multiple candidates so the rule is evaluated for every combination of candidates. The conditions or condition groups are evaluated from left to right, from the innermost level to the outermost. The evaluation of groups is carried out only as far as necessary according to the principle described in Chapter 2.9. If the complete condition set is evaluated as True, we proceed to the transition step. Otherwise, the same procedure is repeated with the next rule.

4.6 Transitions

If all conditions are met, we can apply the transition operations described in the rule. If the transition set includes alternatives in form of OR groups, they have to be extended first according to the description in Chapter 2.9. This expansion happens at the rule loading step. The transitions are carried out in order of appearance. Note that the order

can actually influence the output – not just the processing time as is the case for the condition set. Each transition alters the premise according to the transition function and its arguments. The output sentence is used as the input sentence for the next transition.

Depending on the function and the altered tokens, the syntactic information of some tokens has to be updated. For example, some IDs will have to be adjusted if a token is inserted or deleted as this causes a positional shift. Since parsing is computationally expensive, these updates are conducted by simple rules whenever possible. For more complex changes that affect the sentence structure, we re-parse the output sentence. The resulting output sentence from all transitions is called a derived fact. This derived fact may be needed for further processing so it is crucial that the sentence is syntactically correct and the syntactic information is reliable. Thus, if a transition fails, for example because the guiding token in an insertion cannot be found, we discard the resulting output. Note that one rule can derive multiple facts from one premise if the transition set contains an alternative or a list of token arguments (see Example 6).

4.7 Computing Relation and Logical Validity

The projected relation between premise and derived fact is computed by projecting the relation given by the rule according to the appropriate projection function. The projection function can be determined by inspecting the altered token or the head of the altered phrase. If the altered token's polarity is downward, then the projection for downward polarity contexts is applied. It reverses for instance the forward entailment to reverse entailment, and vice versa. In case of upward polarity, no projection needs to be applied, meaning that original relation and the projected relation are identical. If operators are involved in the transformation of the sentence, the corresponding projection for this operator is required for computing the projected relation.

Joining the current relation to the initial premise and the projected relation according to the join table by Icard (2012) yields the final entailment relation from the newly derived fact to the initial premise.

The logical validity for each derived fact to the initial premise can be determined by a finite state automaton described in Angeli and Manning (2014). When traversing the FSA with the final entailment relations from all inference steps, it yields the logical validity, meaning whether the inference is valid, invalid or of unknown validity (see Figure 2). This value ultimately shows whether the hypothesis can be inferred from the premise, while the final relation shows the entailment type.

(28) *Premise:* All carnivores furiously chase mice.
Hypothesis: All cats don't chase mice. [1]

Example 28 demonstrates how relations and validity are computed across several inference steps. The initial premise is *All carnivores furiously chase mice* and the initial relation is axiomatically *equivalence* (≡). The operator *all* is downward-monotone on the first argument, i.e. *carnivores*, and upward-monotone on the second argument, i.e.

[1] This sentence may sound somewhat unnatural but it proved to be suitable for demonstration.

Derived facts	Curr.	Rule	Proj.	Final relation	Validity
All carnivores furiously chase mice					
	$\equiv$	$\sqsubseteq$	$\sqsubseteq$	$\sqsubseteq$	valid
All carnivores chase mice					
	$\sqsubseteq$	$\sqsupseteq$	$\sqsubseteq$	$\sqsubseteq$	valid
All cats chase mice					
	$\sqsubseteq$	$^\wedge$	$^\wedge$	$\#$	invalid
All cats don't chase mice					

the verbal phrase *furiously chase mice*. So *carnivores* has downward polarity while the lexical items in the VP have upward polarity.

As a first inference step, the adverb *furiously* is removed. The given relation for this rule is *forward entailment* as *furiously chasing mice* forward entails *chasing mice*. Since the change is applied to the second argument, the projection function for upward polarity contexts is used, which preserves the relation given by the rule. The join table for the current relation *equivalence* and the projected relation *forward entailment* ($\sqsubseteq$) returns *forward entailment*, which is the final relation to the initial premise.

In the second inference step, *carnivore* is replaced with the hyponym *cat*. A given word always reverse entails its hyponym, thus the rule relation is *reverse entailment* ($\sqsupseteq$). This time, the first argument of the operator *all* is affected so we use the downward polarity projection function. It reverses the relation *reverse entailment* to *forward entailment*. The join table yields *forward entailment* for two *forward entailments*.

For the final inference step, the VP *chase mice* is negated. The projected relation remains a *negation* ($^\wedge$). The join table for the current relation *forward entailment* and the projected relation *negation* yields alternation ($\#$), which represents the final entailment relation to the initial premise.

We conclude that *All carnivores furiously chase mice* is an alternation to *All cats don't eat mice*, or in other words the premise excludes the hypothesis. When traversing the FSA with the final relations for each inference step, the state *invalid* is reached. Thus, it is not a valid inference.

4.8 Expanding the Inference Tree

After successfully producing a newly derived fact (or several) and computing its relation and logical validity to the initial premise, it is determined whether this new fact is worth pursuing further. As the FSA suggests, there is no possibility of leaving the state of unknown validity. Thus, derived facts of unknown validity are discarded and this branch in the inference tree terminates there. Other branches are pursued in order to determine whether the inference validity is indeed unknown. If the newly derived fact has already been seen, we abandon that branch as well since it would result in duplicate nodes otherwise.

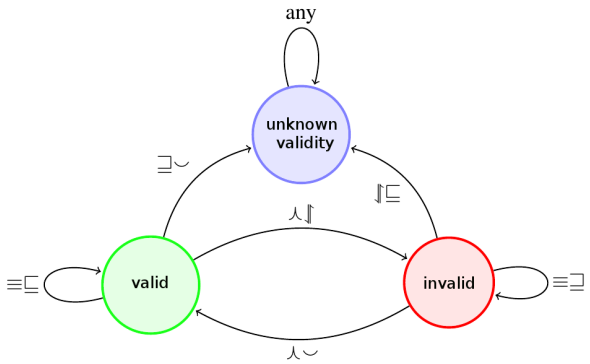

Figure 2: FSA for computing logical validity according to relations to initial premise.

If the derived fact is new and the inference is valid or invalid, we pass the new premise on for further processing and the branch in the inference tree keeps growing. The inference cycle starts over. If the derived fact matches the hypothesis, we consider the tree completed and return the relation and logical validity as the final result.

5 Evaluation

Stimpy is a rule-based system and thus can only solve inference problems for which the required transformations from premise to hypothesis are covered by the rule set. Even for solving only a small fraction of NLI problems, a number of handwritten and carefully refined rules are required. We evaluated how well such a system performs on different NLI problems in order to find strengths and weaknesses. The performance on the inference task, however, should not be regarded as the sole value of this engine.

Stimpy does not only predict the validity of inference for a given premise-hypothesis pair, it also generates syntactically correct intermediate and alternative solutions, i.e. nodes in the inference tree. While a side product of natural language inference, these trees may present valuable data for other applications. Using the derived facts and their relation to the initial premise one could easily and reliably multiply the training data available for machine learning based approaches for textual inference. Furthermore, analysis of the tree structures may reveal ways of optimizing the search for a match, by taking into consideration which paths are more likely to succeed. Since the engine modifies and generates sentences, it may also be of interest for research fields such as paraphrasing, text simplification and normalization, or adversarial stylometry.

Thus, we also evaluated the quality of the derived facts. Since we put great emphasis on producing syntactically correct sentences, we evaluated whether the generated sentences are indeed correct and tracked down the sources of mistakes.

	Problems	NLI Accuracy	Relation Accuracy
Stimpy suite	70	98.5%	97.1%
FraCaS suite	334	35.6%	-

Table 1: Performance of Stimpy engine on natural language inference problems.

5.1 NLI Task

We employed two test suites for the evaluation. The first suite we named Stimpy suite and it functions as the development set as it contains samples that were inspired by typical NLI problems. Some examples imitate patterns found in the FraCaS test suite. It consists of 70 NLI problems, including 18 multi-premise problems. The second test suite is the FraCaS test suite (Cooper et al., 1996). It contains 334 problems with unambiguous solution, of which 187 are multi-premise problems.

For every problem, the engine processes every given premise one-by-one with the goal of finding an inference path to the hypothesis. If none of the derived facts matches the hypothesis, the default solution *unknown validity* is predicted. Since some premises produce very large inference trees that take a long time to compute, we stopped expanding trees when they reached a height greater than 3. This means that there can maximally be 3 inference steps between initial premise and hypothesis. Furthermore, we applied a 15 second limit for processing each premise. If the hypothesis is not found in the inference tree that is created during that time frame, the inference validity again defaults to unknown. The 15 seconds limitation was only necessary for the FraCaS test suite.

5.1.1 Stimpy Suite

Table 1 shows the results for natural language inference on the two test suites. The performance on the development Stimpy suite is inevitably better than for FraCaS since the engine has been optimized for these problems. Nearly all problems can be solved correctly in terms of validity as well as relation.

Stimpy failed on the problem representing that *Every cat eats fish* reverse entails *Every animal eats fish*, which is an invalid inference. The engine first computed the path from *cat* to *dog*, which then led to *animal*, instead of directly moving to the hypernym *animal*. This detour made the system predict *unknown validity* for the inference because the *dog* is an alternation to *cat*. Since the derived fact matched the hypothesis, the system terminated and the direct path was not found. This reveals a weakness of the engine: there are multiple paths to the same derived fact and they do not all yield the same relation. Future work would have to examine how the most direct and most reliable path can be detected.

5.1.2 FraCaS Suite

Unsurprisingly, the FraCaS test suite proved a greater challenge since many of the required transformation rules are not yet implemented. Nevertheless, the engine was able to predict the correct inference validity for 35.6% of the problems, which proves that even a relatively small rule set can solve a variety of NLI problems. The relation accuracy could not be computed since FraCaS does not provide this information.

For 38 problems, STIMPY found a derived fact that matched the hypothesis, and 30 of these problems were solved correctly. Inspection of the results shows that the logical validity *unknown* was predicted with a precision of 32.1%. and a recall of 92.8%. The large difference between recall and precision is not surprising when considering that the engine predicts this solution by default if no match is found or the engine times out. In fact, the engine correctly guessed *unknown* in 56 cases by default and 33 cases because of a time out. This insight does not necessarily diminish the strength of the engine as default predictions and timeouts are valid and necessary elements. Even so, one should keep this functionality in mind when discussing the 35% accuracy.

FraCaS contains 207 *valid inference* examples. STIMPY predicted these problems with higher precision (77.1%) than recall (13.3%). This suggests that the rules detecting valid inferences are relatively reliable, although many inference steps are not yet implemented. As for the *invalid inference* examples, the engine predicted *invalid* only for one problem, yet correctly. This yields a recall of 3% and a precision of 100% and indicates that the rules are currently geared towards finding valid inferences.

In fact, many rules generate a forward entailment by deleting part of a sentence. We only implemented few insertion rules, which often produce a reverse entailment and thus an invalid inference. This is because the set of possible deletions in a given sentence is relatively small whereas that of possible insertions is effectively unbounded. In future work, one could implement the bidirectional search algorithm introduced by Angeli et al. (2016). Instead of inserting elements in the premise, one would delete them from the hypothesis.

FraCaS does not explicitly label which problems require multi-premise inference to be solved so we made an assumption based on the number of premises given and on comments indicating that only one premise is needed to solve the problem. This method identified 187 samples as multi-premise problems. STIMPY achieved an accuracy of 41.2% for single-premise problems and 28.5% for multi-premise problems. This shows that STIMPY is capable of correctly deriving facts from multiple premises.

5.1.3 Comparison to NaturalLI Performance

Angeli and Manning (2014) evaluated their work on the FraCaS test suite as well, although they concede that it is not a blind test set. They selected three sections of the corpus which are specifically within the scope of their system: quantifiers, adjectives and comparatives. They also discard all multi-premise problems, resulting in a set of 75 problems. NaturalLI achieves an accuracy of 89% on this selected problem set. The results for the subsections are shown in Table 2. STIMPY reaches an accuracy of 57.7%

	Quantifiers	Adjectives	Comparatives	Total
NaturalLI (2014)	95%	73%	87%	89%
Stimpy	68.8%	42.9%	38.5%	57.7%

Table 2: Performance of NaturalLI and Stimpy on FraCaS sections.

on the same set, which which is an increase of more than 20% compared Stimpy's performance on the entire FraCaS suite. Both systems perform best on quantifier problems, which is not surprising since both focus on the implementation of quantifier and operator modifications. For NaturalLI the comparative problems are solved more reliably than the adjective problems. Stimpy solves both types with similar accuracy.

In general, NaturalLI performed better on the NLI task than our engine. However, one should consider that NaturalLI implements a smaller set of rules that was specifically optimized for the patterns in the selected FraCaS sections. Our engine aims to be more general and thus includes rules not relevant to FraCaS. Furthermore, NaturalLI applies transitions in a less strict fashion, which naturally increases recall. It is also less stringent when determining whether a derived fact matches the hypothesis, e.g. comparing lemmas instead of word forms or considering nearest neighbors as synonyms, which makes the system more robust at the cost of precision.

Finally, it should be noted that Stimpy's programmable rule syntax would allow the current pilot implementation to be improved through the addition of further rules, thus a direct comparison of performance between the two systems is therefore somewhat flawed at this point in time. Furthermore, there are important conceptional differences between NaturalLI and Stimpy that go beyond performance evaluation. These are discussed in Chapter 6.3.

5.1.4 Frequently Applied Rules

In this section, we examine which rules were called most often. In the Stimpy test suite, the most frequently applied rules are the following: replacement by stronger operator, replacement by hyponym, replacement by hypernym, negation of a non-modal verb, and the insertion of *There are ... who/that*. There is some overlap with the Top 5 for the FraCaS test suite: replacement by weaker operator, negation of non-modal verbs, replacement by stronger operator, deletion of prepositional phrase, and negation of modal verbs. All of these rules are relatively general and work in many contexts, e.g. most positive sentences can be negated. As for lexical replacement rules (hypernyms, hyponyms, operators), they can even be applied multiple times to the same premise (with different arguments) so their occurrences are naturally high.

In this context a side note on lexical replacements: since the Stimpy test suite served as development set and was tested frequently, we provided the necessary related words for these examples in a hand-written dictionary as to circumvent WordNet look-ups and inappropriate candidates. Thus, the WordNet integration is not evaluated here.

Unfortunately, we also had to deactivate the WordNet function for the FraCaS test suite since the extensive look-up time caused too many time outs. Upon inspection, however, we found that FraCaS problems rarely rely on WordNet knowledge and that hypernym or hyponym relations are usually provided by the premises themselves. Even when activating WordNet and considerably raising the time out limit, the number of correctly solved problems does not increase. However, this does not imply that WordNet relations are irrelevant for textual inference in general. Furthermore, we found that the collected related words were frequently inappropriate for the given context. Future work will have to investigate methods for finding suitable candidates.

5.2 Quality of Derived Facts

We evaluated the derived facts according to their syntactic correctness. Unlike for the NLI evaluation where the expansion of the inference tree is stopped whenever a match is found, we constructed the entire inference trees with maximal height 3 and increased the time limit to 30 seconds.

5.2.1 Stimpy Suite

We inspected all 3417 derived facts resulting from the premises of the 70 problems in the Stimpy test suite, which returns on average 44 derived facts per premise. Evaluation shows that 92.6% of all derived facts in the Stimpy suite are syntactically correct, which makes the engine a reliable sentence generator. However, even on the development set it is challenging to achieve perfection. Table 3 lists which kind of errors occurred most frequently and shows that most mistakes are caused by errors from the syntactic parser. Apparently, the verbs *devour* and *chase* confounded the parser and it often mislabeled them as part of a compound noun. Naturally, transformations on such incorrectly parsed sentences resulted in ungrammatical outputs. The next most frequent form of error is connected to the determiner *a*, more specifically its modification to *an*. Although we implemented a rule to alter the determiner whenever the next word starts with a vowel, we did not consider cases in which a word of the noun phrase is deleted, e.g. *a blue elephant* to *an elephant*. This is only one of many examples that demonstrate how thorough a sentence generating system has to be.

5.2.2 FraCaS Suite

From the problems in the FraCaS test suite, Stimpy could derive 6393 facts in total. Although FraCaS contains almost 5 times as many problems as the Stimpy suite, it produced only twice as many derived facts. This can be explained by the lack of WordNet relations, which would drastically expand the inference tree. We selected 50 problems across the test suite and examined all 1410 facts derived from their premises. 46.3% of these facts were syntactically correct, which means that roughly every other sentence generated by the engine is correct. This is a reassuring result considering that FraCaS

Error type	STIMPY suite	FraCaS suite
Parser error	234	-
Failed negation	2	19
Article *a/an*	14	2
Missing words	1	60
Superfluous words	-	10
Multi-word operator	-	294
Operator in unexpected context	-	131
Wrong plural form	-	77
Named entity exception	-	37
Failed *There are ... who*	-	18
Failed passive resolution	-	7

Table 3: Error analysis for derived facts from STIMPY and FraCaS test suites.

consists of unseen problems. Most often the incorrect sentences require only minor corrections and usually still convey the meaning of the sentence.

Even though some mistakes may be caused by parser errors, we did not consider this category for the FraCaS test suite as it would have been too costly to take the parser output into account for every incorrect sentence. The most frequent source of error we detected were operator rules. We implemented rules for single word operators only but the rules were also triggered for single words in multi-word operators, for instance *most* in *at most* or *few* in *a few*. The current rules do not test for the context of the operator and attempt replacement with any other known operator, resulting in combinations such as *at several* or *a each*. A related issue arises when operators are used in contexts not covered by the rules. The operator *many* for instance can be used in expressions such as *how many* or *as many as*. In these contexts, *many* cannot simply be replaced by another operator.

We found many derived facts that were not complete sentences, generated by removing a non-optional phrase, for instance prepositional phrase: *John is* instead of *John is from California*. The conditions for rules containing a deletion will have to be made more strict in future work.

Pluralization of nouns also failed frequently, creating words such as *peoples* or *Europeanss*. We assume that *people* and *Europeans* were not recognized as plural forms and thus an *s* was added. Finally, we discovered that we did not implement sufficient exceptions for named entities. They are treated as normal nouns by the rules, resulting in slightly absurd sentences such as *Each John is from California*.

Many incorrect sentences are caused by the flaws in rule design, so adjusting just one rule could prevent many mistakes and subsequent errors. If STIMPY was optimized according to the error analysis, the performance could likely increase drastically.

6 Discussion

The previous chapter describes the results for the evaluation of the STIMPY engine, which revealed some strengths and weaknesses. In this chapter, we summarize and discuss them in greater detail.

6.1 Strengths

The performance of STIMPY on the two test suites is encouraging as it demonstrates that our engine can solve both single-premise and multi-premise problems. Starting from this proof-of-principle, STIMPY can be adapted to apply virtually any desirable inference pattern. The general-purpose rule syntax makes it comparably easy to reason about, extend and optimize the system, all without any knowledge of the execution code itself.

Importantly, our rule syntax supports the design of very precise rules: the condition set can specify exactly on which sentences a rule can be applied, while the transition set does not only derive the new fact but also ensures correct output. Our rules are designed to derive syntactically correct sentences, which makes the derived facts a valuable resource for other applications, and it ensures reliable input quality for further processing steps and thus reduces subsequent errors.

Unlike machine learning approaches, STIMPY does not require any training data in the classical sense. Although rules are based on known inference patterns, there is no need for a large number of samples to "learn" a new rule. Another advantage over machine learning approaches is the fact that STIMPY is completely transparent in the computation of logical validity and relation. Every processing step can be inspected and could be adapted if necessary. The engine returns all the inference steps as well as the inference tree which facilitates error analysis and optimization. Furthermore, analysis of these outputs may identify frequent transition sequences, which could be used to optimize performance and reduce processing time. The derived facts could also be useful for other applications, most importantly as a reliable extension of training data for machine learning based systems.

We implemented STIMPY using the the Python programming language and plan to make it available as an open-source project. Therefore, the engine is expected to run reliably on modern systems and can be improved and extended by the community.

6.2 Weaknesses

STIMPY is clearly limited by the relatively small number of rules implemented thus far. Consequently, it was only able to solve a fraction of the FraCaS problems. However, we believe that performance could be increased considerably by simply designing additional rules inspired by inference patterns seen in FraCaS problems.

Furthermore, the evaluation on the FraCaS test suite revealed that only half of the derived facts were syntactically correct. Although this may seem a low number at first glance, the error analysis suggests that many of the mistakes originate from the same few

flaws in the inference rules. It stands to reason that fixing these flaws would substantially improve the quality of derived facts.

While rule improvement and extension of the rule set may increase performance of the engine, the most fundamental downside of this approach remains: the necessity for manually engineered rules. Due to variability of lexical expression and the diversity of inference patterns, rule creation is a very time-consuming and challenging task, especially with the goal of generating only syntactically correct derived facts. For each new rule, tests are required to ascertain that they produce correct output, both on their own and in combination with other rules. Despite the relative ease of rule implementation afforded by the engine's syntax and by direct operation on natural language, it thus remains hard to imagine that an entirely rule-based system could scale to real-world applications.

However, we believe it may be possible to automatically extract and optimize rule sets, which would render manual rule engineering largely unnecessary whilst retaining the transparency and interpretability of the automatically generated system.

6.3 Comparison to NaturalLI

There are several conceptional differences between NaturalLI and Stimpy, as mentioned in the introduction and evaluation. Here, we discuss them in further detail. One essential difference is that Stimpy allows easy modification and addition of inference rules as they are maintained outside the execution code. We specifically designed a syntax for writing and loading these rules in an effective way (see Chapter 2).

Angeli and Manning (2014) do not explicitly mention how they implemented and maintained their *transition templates*. Inspection of the NaturalLI code[2] did not reveal a functionality that would facilitate adjusting the relation templates so we assume that they view their template set as complete and immutable. Furthermore, they do not describe how or if they tested whether a template can be applied to a premise.

These missing features can be explained by another difference: NaturalLI does not necessarily attempt to generate syntactically correct sentences. It applies the transitions in a less restricted way without ensuring a correct output. For instance, it allows the deletion of any word in a sentence, disregarding its syntactic functionality. Also, it does not adjust words forms, for instance in case of an operator replacement that requires pluralization of noun phrase and verb (see Example 20). A condition set defining precisely when a rule can be applied and a transition set maintaining grammaticality are thus not needed in NaturalLI. Although Angeli and Manning (2014) do not make mention of it, this simplification must generate a large number of syntactically incorrect derived facts. They probably accept this as a trade-off for higher recall. Although it is never explicitly stated that they do not aim at producing syntactically correct derived facts, no means of preserving correctness are implemented. In fact, some of NaturalLIs limitations suggest that it would not be easy to do so.

First of all, NaturalLI only allows one mutation of an individual lexical item per inference step. Thus, word form adjustments, such as those needed when switching

[2] `https://github.com/gangeli/NaturalLI`

certain operators (e.g. *every* with *all*), cannot be covered by the transition template. STIMPY, on the other hand, can perform any number of transitions to derive a new syntactically correct fact from a premise.

Secondly, NaturalLI does not have access to the parse tree. Syntax information is only incorporated in later follow-up work (Angeli et al., 2016), and even there in a limited fashion. In most cases, the mutations are simply string-based and the syntactic information is disregarded. As for the STIMPY engine, most rules contain syntactic conditions so syntactic parser information is crucial.

Thirdly, the mutation order for NaturalLI is set from left to right, while for STIMPY the order is not fixed and thus rules can be applied more flexibly.

It should be noted that these short comings may not be of interest to Angeli and Manning (2014) since they may not considerably reduce the performance on the natural language inference task. As a matter of fact, they may even increase recall. For the common sense reasoning task, they discarded the original word forms altogether and worked with lemmatized text snippets. However, syntactically correct derived facts are not merely a useful side product of inference. The computation of lexical polarity and the scope of operators relies heavily on information provided by the syntactic parser, which performs less reliably on incorrect sentences. Thus, incorrectly computed polarities and scopes may constitute a source of errors for NaturalLI.

Another difference in system architecture is that NaturalLI allows derivations that are only *likely* valid, which are then accompanied by an associated confidence score. They declare that their mapping from transition template to relation is imprecise so they introduce costs for each transition and learn their appropriate value. Angeli and Manning (2014) use Example 29 to demonstrate the imprecision of their templates: the same transition template with a given relation, i.e. *antonym* with the relation *alternation* (⫲), does not yield the same final entailment relation in every context. As as matter of fact, STIMPY is able to resolve this inference problem correctly by applying the appropriate projectivity functions to compute the projected relation for the operators *all* and *some*.

(29) All cats like milk ⫲ All cats hate milk (invalid)
Some cats like milk # Some cats hate milk (unknown)

NaturalLI employs a smaller and more imprecise set of transition templates, which is mitigated by introducing costs for transition. STIMPY, on the other hand, incorporates a larger number of precise rules, so there was no need to implement a cost function for the rules. However, one may consider introducing this feature in future work as a means to reduce the number of hand-designed rules, to identify the most valuable rules and to save computing time.

Another small difference between the two engines is that NaturalLI is implemented for reverse inference, meaning that facts are derived from the hypothesis and matched to the premises, while STIMPY infers facts from premises and compares them to the hypothesis. For single-premise inference problems directionality is mostly irrelevant. Angeli and Manning (2014) probably decided on reverse inference since they worked on

database completion and thus aimed to prove whether a hypothesis is in some way related to a premise. We decided to build a forward inference system and derive facts from the premises, not only because this seems to correlate better with people's understanding of inference but also because of the multi-premise problems. It is easier and more reasonable to derive facts from multiple premises than to generate premises that support the hypothesis, although doing so would technically be possible.

This leads to the final difference between NaturalLI and Stimpy: NaturalLI can only process single-premise problems, while Stimpy is fit to solve multi-premise problems as well. Taken together, these differences illustrate the strengths of Stimpy as a precise, extendable and highly versatile framework for natural language inference.

6.4 Outlook

The most straightforward way of improving the Stimpy engine would be the extension and addition of rules. Since the currently implemented rules should be considered exemplary rather than complete, there is much work left to do on optimizing and generalizing the existing rules. For instance, we only implemented rules for a selection of single-word operators, while completely ignoring multi-word operators such as *a couple of*.

However, manually designing and optimizing these results is a time-consuming task with diminishing returns, since at a certain point there will simply be too many patterns, cases and exceptions to consider. Thus, we suggest that in future work these rules should be extracted automatically from large corpora of inference examples, by essentially reversing the current engine. For a premise and hypothesis with a given relation and/or validity, the necessary transition steps to get from premise to hypothesis are computed. Based on the structure of the original premise, the affected tokens and the transition steps, the conditions and transitions are derived and the rule is automatically created. The Stimpy engine could assist this process by deriving facts from the premise. These facts are potentially closer to the hypothesis than the initial premise and require fewer transition steps, which makes it easier to infer the rules required. The inference tree may also allow rule inferences since we can (reversely) compute the relation of every node to another.

Rules extracted in this way would then have to be merged and generalized, which could potentially be achieved automatically, for instance using evolutionary programming. In this iterative process, Stimpy would be used as a "forward pass" to evaluate the performance of a given rule set. Using this approach, one could potentially generate comprehensive rule sets beyond the scope of manual engineering.

As the number of rules grows, the processing time increases as well. This issue could be addressed in several ways. First, the available premises could be ranked according to a similarity measure indicating similarity to the hypothesis, so premises closer to the hypothesis are processed first. Second, rules or sets of rules could be assigned probabilities or costs which ensure that rules that are more likely to succeed are applied first. These costs could be made dependent on the current structure of the sentence and the relation to the initial premise. For instance, it is usually not meaningful to apply a rule for which it is certain that it will create an unknown inference validity. Finally,

there are technological possibilities for reducing the computing time. Most importantly, several aspects of the engine's execution can readily be parallelized and could therefore be processed considerably faster by clustering computing.

Finally, we believe that our syntax and sentence transformation engine could be applied to other NLP tasks since it is not specific to inference. Many rule-based approaches would benefit from a clean syntax for writing, maintaining and applying rules. An example would be the text simplification system for German designed by Suter et al. (2016), for which the simplification rules are hard-coded in the system. Rule extensions or adjustments, for example for other languages, are difficult for such systems since the rules and the execution code are intermixed. Our rule syntax with conditions and transitions allow these simplification rules to be written and maintained in a clean, readable and accessible way. The metadata for each rule can be used to track transition types as it is done for inference. If necessary, the syntax can also be extended in order to enable more suitable ways of writing the rules, for example by introducing templates for frequent transitions. The predicates and transition functions can naturally be extended as well.

7 Conclusion

We built a precise, extendable and versatile natural language inference system for Python. Stimpy predicts whether the hypothesis can be inferred from a given set of premises based on manually engineered inference rules. In order to facilitate rule writing, reading and loading, we devised a syntax for these inference rules. We extensively described the elements of this syntax and demonstrated their functionality by explaining some of the implemented rules. We designed 41 rules for single-premise and 12 for multi-premise inference patterns. We elaborated on how Stimpy processes a premise-hypothesis pair step-by-step and explained how the final relation and logical validity is computed.

We evaluated the performance of the system with regards to accuracy in natural language inference and quality of the computed inference trees on two test suites. We reached an accuracy of 35% on the FraCaS test suite, which we consider an encouraging result for an exemplary set of rules. As for the quality of the derived facts, we identified 46% of them as syntactically correct and conducted an error analysis.

We discussed the general strengths and weaknesses of the engine. In comparison to NaturalLI, Stimpy puts more emphasis on clean application of rules and correct output. The rule syntax allows designing very precise rules and helps in producing syntactically correct output, although the necessity for manually engineered rules is a clear downside.

Finally, we gave an outlook for possible and interesting extensions of the system, including a reverse system that allows automatic extraction of inference rules.

This work demonstrates that textual inference is a very challenging research field as it requires combined knowledge from syntax, semantics and logic. We believe that a clean and easily extendable framework such as Stimpy can serve as a foundation for meeting this challenge in a fully automated, highly precise, and readily interpretable fashion.

Appendix

Predicates that return Boolean value

Predicate	
is(*variable*)	introduces variable, always returns True
not(*predicate*)	inverts Boolean for evaluated predicate
	Evaluates whether...
exists(*kwargs*)	there exists a token with constraints given by the keyword arguments
not_exists(*kwargs*)	there exists no token with constraints given by the keyword arguments
dependent(token, head)	token is dependent on head
has_dep_token(head, *kwargs*)	there is a dependent token for head that matches constraints given by keyword arguments
has_no_dep_token(head, *kwargs*)	there is no dependent token for head that matches constraints given by keyword arguments
exists_hypernym(word)	there exists a hypernym for word
exists_hyponym(word)	there exists a hyponym for word
exists_antonym(word)	there exists a antonym for word
exists_synonym(word)	there exists a synonym for word
exists_sibling(word)	there exists a sibling for word
positive_sent()	the sentence is positive (no negation)
is_passive()	the sentence contains a passive construction
same_phrase(phrase1, phrase2)	phrase1 is equal to phrase2
same_phrase(phrase1, phrase2, "neg")	phrase1 is negation of phrase2
id_difference(token1, token2, diff)	token1 and token2 are *diff* positions apart
context(ref_token, left=*string*)	*string* is to the left of ref_token
context(ref_token, right=*string*)	*string* is to the right of ref_token
in_list(*string*, *list_name*)	*string* is in pre-defined list called *list_name*

Table 4: Predicates that return Boolean value.

Predicates that return string

Predicate	*Returns...*
id(token)	token id
form(token)	token form
lemma(token)	token lemma
u_pos(token)	universal part-of-speech for token
x_pos(token)	language-specific part-of-speech for token
head(token)	token head
deprel(token)	dependency relation from token to head
hypernym(word)	hypernym for word
hyponym(word)	hyponym for word
antonym(word)	antonym for word
synonym(word)	synonym for word
sibling(word)	sibling for word
compare_operators(OP1, OP2)	'weaker', 'stronger' or 'equal'
number_adjustment(OP1, OP2)	'singularize', 'pluralize' or 'none'
singularize(pos)	singular form for pos
pluralize(pos)	plural form for pos

Table 5: Predicates that return string.

References

Angeli, G. and Manning, C. D. (2014). NaturalLI: Natural Logic Inference for Common Sense Reasoning. In *EMNLP*, pages 534–545.

Angeli, G., Nayak, N., and Manning, C. D. (2016). Combining Natural Logic and Shallow Reasoning for Question Answering.

Cooper, R., Crouch, D., Van Eijck, J., Fox, C., Van Genabith, J., Jaspars, J., Kamp, H., Milward, D., Pinkal, M., Poesio, M., et al. (1996). Using the framework. Technical report, Technical Report LRE 62-051 D-16, The FraCaS Consortium.

Honnibal, M., Johnson, M., et al. (2015). An Improved Non-monotonic Transition System for Dependency Parsing. In *EMNLP*, pages 1373–1378.

Icard, T. F. (2012). Inclusion and exclusion in natural language. *Studia Logica*, 100(4):705–725.

Karttunen, L. (2012). Simple and Phrasal Implicatives. In *Proceedings of the First Joint Conference on Lexical and Computational Semantics-Volume 1: Proceedings of the main conference and the shared task, and Volume 2: Proceedings of the Sixth International Workshop on Semantic Evaluation*, pages 124–131. Association for Computational Linguistics.

MacCartney, B. and Manning, C. D. (2007). Natural logic for textual inference. In *Proceedings of the ACL-PASCAL Workshop on Textual Entailment and Paraphrasing*, pages 193–200. Association for Computational Linguistics.

MacCartney, B. and Manning, C. D. (2008). Modeling semantic containment and exclusion in natural language inference. In *Proceedings of the 22nd International Conference on Computational Linguistics-Volume 1*, pages 521–528. Association for Computational Linguistics.

MacCartney, B. and Manning, C. D. (2009a). An extended model of natural logic. In *Proceedings of the eighth international conference on computational semantics*, pages 140–156. Association for Computational Linguistics.

MacCartney, B. and Manning, C. D. (2009b). *Natural language inference*. Ph.D. thesis, Stanford University.

Miller, G. A. (1995). WordNet: A Lexical Database for English. *Communications of the ACM*, 38(11):39–41.

Mineshima, K., Martınez-Gómez, P., Miyao, Y., and Bekki, D. (2015). Higher-order logical inference with compositional semantics. In *Proceedings of EMNLP*, pages 2055–2061.

Nairn, R., Condoravdi, C., and Karttunen, L. (2006). Computing relative polarity for textual inference. *Inference in Computational Semantics (ICoS-5)*, pages 20–21.

Suter, J., Ebling, S., and Volk, M. (2016). Rule-based Automatic Text Simplification for German. *Proceedings of the 13th Conference on Natural Language Processing (KONVENS 2016)*, pages 279–287.

Weston, J., Bordes, A., Chopra, S., Rush, A. M., van Merriënboer, B., Joulin, A., and Mikolov, T. (2015). Towards AI-Complete Question Answering: A Set of Prerequisite Toy Tasks. *arXiv preprint arXiv:1502.05698*.