Lecture 13.2: Semantic Parsing

Motivation

Goal of logic: represent knowledge and perform inferences

Propositional logic:

AliceIsStudent $\rightarrow$ AliceKnowsArithmetic

BobIsStudent $\rightarrow$ BobKnowsArithmetic

First-order logic:

$\forall x$ Student$(x) \rightarrow$ Knows$(x,$ arithmetic$)$

- In the last two lectures, we have presented two types of logics: propositional logic and first-order logic.
- In propositional logic, we build formulas by putting connectives around propositional symbols. These allow us to make statements about the truth values of specific facts.
- In first-order logic, the key conceptual difference is that we are making statements about objects and their relations. Propositional symbols become atomic formulas which are predicates applied to terms (constants, variables, and functions). Together with quantifiers, this allows us to make compact statements that represent a huge (possibly infinite) set of objects.
- From a modeling point of view, first-order logic is more expressive than propositional logic, but it does come at a increase in computational cost.

Motivation

Why use anything beyond first-order logic?

Expressiveness:

- Temporal logic: express time
- Epistemic logic: express beliefs
- **Lambda calculus: generalized quantifiers**

Notational convenience, computational efficiency:

- Description logic

- There are two ways we can improve on first-order logic.
- The first makes modeling easier by increasing expressiveness, allowing us to "say more things", so we can talk about time, beliefs, not just objects and relations. We can also improve the modeler's life by making notation simpler.
- The second makes life easier for algorithms by decreasing expressiveness.

Lambda calculus

Simple:

Alice has visited some museum.

$\exists x$ Museum$(x) \wedge$ Visited$($alice$, x)$

More complex:

Alice has visited **at least 10** *museums.*

λx Museum$(x) \wedge$ Visited$($alice$, x)$: boolean function representing set of museums Alice has visited

GreaterThan$($Count$(\lambda x$ Museum$(x) \wedge$ Visited$($alice$, x)), 10)$

- Existential and universal quantification only allow a formula to look at one value at a time. But sometimes, we need to look at the set of all values satisfying a formula, for example, if we wanted to count.
- Here, we will use lambda calculus to define **higher-order functions** that allow us to do precisely this.
- In lambda calculus, $\lambda x\, P(x)$ denotes the set of x for which $P(x)$ is true. This set can be passed in as an argument into the function Count, which produces a term.

From language to logic

Alice likes hiking. → Likes(alice, hiking)

Alice likes geometry. → Likes(alice, geometry)

Bob likes hiking. → Likes(bob, hiking)

Bob likes geometry. → Likes(bob, geometry)

Lots of regularities — can we convert language to logic automatically?

- So far, we've used natural language as a pedagogical means of helping you understand the semantics of logical formulas. But after a while, you might note that there is actually quite a bit of similarity between sentences in natural language and logical formulas. This suggests that it might be possible to build a system that can automate this conversion.
- Such a system would be incredibly useful, since it would allow us to talk to a computer using natural language, conveying the rich information contained with the implied logical formulas. Another motivation is that much of human knowledge is actually written down in text. We'd like a computer to internalize this knowledge and support question answering from it. Think next-generation search.

From language to logic

Key idea: principle of compositionality

The semantics of a sentence is combination of meanings of its parts.

Sentence:

Alice likes hiking. → Likes(alice, hiking)

Words:

Alice → alice

hiking → hiking

likes → $\lambda y\, \lambda x$ Likes(x, y)

- To make the mapping from sentences to logical formulas work, we leverage the principle of compositionality, which basically says that this mapping is defined recursively based on subparts.
- We can say that the word *Alice* maps to the logical term alice; *hiking* to hiking. But if *likes* simply Likes, then we don't have enough information about how to combine the parts. Therefore, we let *likes* map onto a function that takes two arguments y and x (in that order), and returns Likes(x, y).

Lambda calculus: example 1

Function:

λx Student$(x) \wedge$ Likes$(x,$ hiking$)$

Argument:

alice

Function application:

$(\lambda x$ Student$(x) \wedge$ Likes$(x,$ hiking$))($alice$)$

Student(alice) $\wedge$ Likes(alice, hiking)

- Before getting into how we build up the full logical form, let's give a few quick lambda calculus examples.
- First, suppose we have a function and argument. Function application substitutes the argument (e.g., alice) in for the variable of the function (x).

Lambda calculus: example 2

Function:

$\lambda y\, \lambda x\, \text{Likes}(x, y)$

Argument:

hiking

Function application:

$(\lambda y\, \lambda x\, \text{Likes}(x, y))(\text{hiking}) =$

$\lambda x\, \text{Likes}(x, \text{hiking})$

Lambda calculus: example 2

Function:

$\lambda f\, \lambda x\, \neg f(x)$

Argument:

$\lambda y\, \text{Likes}(y, \text{hiking})$

Function application:

$(\lambda f\, \lambda x\, \neg f(x))(\lambda y\, \text{Likes}(y, \text{hiking})) =$

$\lambda x\, \neg(\lambda y\, \text{Likes}(y, \text{hiking}))(x) =$

$\lambda x\, \neg\text{Likes}(x, \text{hiking})$

- As a more complex example, arguments can themselves be functions. Working through the substitutions, we get that this function performs negation, turning "people who like hiking" to "people who don't like hiking".
- One of the powerful aspects of lambda calculus is exactly that functions can be passed into other functions. But to be absolutely rigorous, we would use simply-typed lambda calculus, which associates each function with a type (e.g., object $\to$ bool).

Grammar

Grammar

Lexicon:

Alice → alice

hiking → hiking

likes → $\lambda y\, \lambda x\, \text{Likes}(x, y)$

Forward application:

$f \quad x \to f(x)$

Backward application:

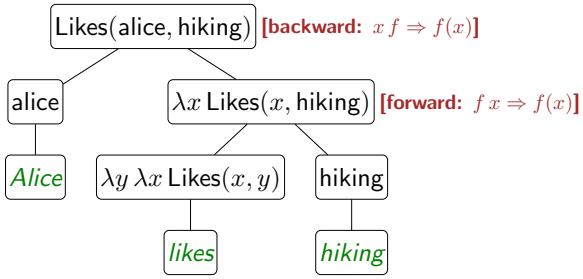
$x \quad f \to f(x)$

- A grammar is a set of rules. The lexicon contains a set of lexical rules, which map natural language to formulas. We also have two rules that perform forward and backward application (depending on whether the function precedes or succeeds the argument).
- Advanced: if you take a semantic class in linguistics or CS224U, you will get a richer linguistic treatment of this material. We have deliberately tried to suppress that in the interest of keeping things simple. Each rule can not only produce a logical formula, but also a syntactic category (e.g., sentence or noun phrase), which can be used to restrict when rules are applied.
- Advanced: also note that a context-free grammar (CFG) is also a different beast than the grammars here because the emphasis is on determining which sentences are licensed under the CFG (yes or no), whereas here, our expressed emphasis is to actually produce the logical formula for the meaning of that sentence.

Basic derivation

Leaves: input words

Internal nodes: produced by applying rule to children

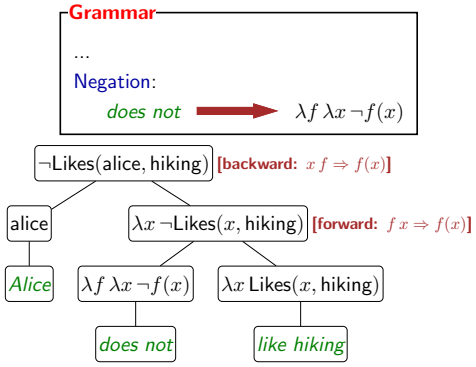


- This is perhaps the most important slide. Given a sequence of input words (as the leaves of the tree), we build a logical form at each intermediate node by applying a rule to the formulas of its children. The formula at the root is the formula that's output and returned.
- Next, we will walk through some important linguistic phenomena.

Negation

Alice does not like hiking.

¬Likes(alice, hiking)

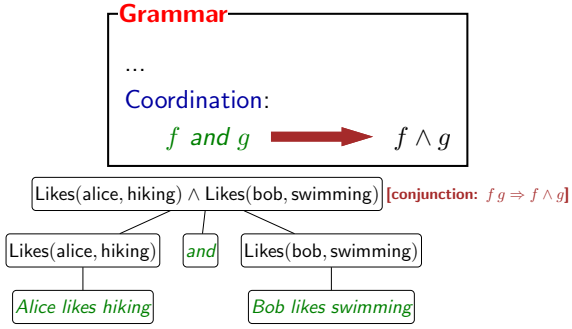


- Negation is an important construct that inverts the meaning of a sentence. As we saw earlier with lambda calculus example 2, we can turn λx Likes$(x,$ hiking$)$ into $\lambda x \neg$Likes$(x,$ hiking$)$.

Coordination 1

Alice likes hiking **and** *Bob likes swimming.*

Likes(alice, hiking) ∧ Likes(bob, swimming)



- Coordination refers to the use of connective words such as *and*, *or*, etc. In the simplest coordination case, we are simply taking the conjunction of two formulas.

Coordination 2

Alice likes hiking **and** *hates swimming.*

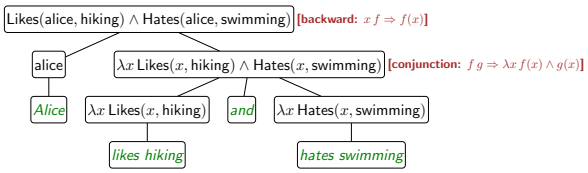
$\text{Likes}(\text{alice}, \text{hiking}) \wedge \text{Hates}(\text{alice}, \text{swimming})$

Grammar

...

Coordination:

f *and* g → $\lambda x\, f(x) \wedge g(x)$



- However, the same word *and* can also be used to conjoin two incomplete sentences (*likes hiking* and *hates swimming*). Notice that *Alice* only shows up once in the sentence but twice in the formula.
- Here, we define a different coordination rule which takes the conjunction not of two formulas, but of two functions. Intuitively, this allows us to combine Likes and Hates "underneath" the λx.

Quantification

Every *student likes hiking.*

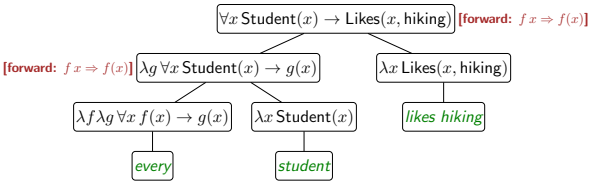
$\forall x\, \text{Student}(x) \rightarrow \text{Likes}(x, \text{hiking})$

Grammar

...

Universal quantification:

every → $\lambda f \lambda g \forall x\, f(x) \rightarrow g(x)$



- The last phenomenon that we will discuss is quantification, which is a very rich topic, and is a major point of first-order logic too.
- In the simplest case where there is just one quantifier, we can add a rule that maps *every* to something that takes two functions, f and g, and returns the appropriate formula.
- In the context of the sentence, f gets bound to $\lambda x\, \text{Student}(x)$ and g gets bound to $\lambda x\, \text{Likes}(x, \text{hiking})$.

Ambiguity

Lexical ambiguity:

Alice went to the bank. → $\text{Travel}(\text{alice}, \text{RiverBank})$

Alice went to the bank. → $\text{Travel}(\text{alice}, \text{MoneyBank})$

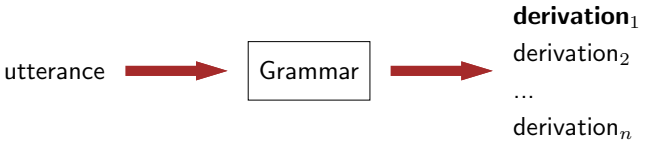
Scope ambiguity:

Everyone likes someone. → $\forall x \exists y\, \text{Likes}(x, y)$

Everyone likes someone. → $\exists y \forall x\, \text{Likes}(x, y)$

- Everything has worked out smoothly because we have deliberately chosen the right set of rules to apply so that we get the correct logical formulas. But in some case, the same piece of text can map to multiple valid logical formula that have different meanings!
- This discrepancy might be due to lexical ambiguity (individual words mean many different meanings) or scope ambiguity where the actual order of quantification is not specified by the text (natural language is weird like that).

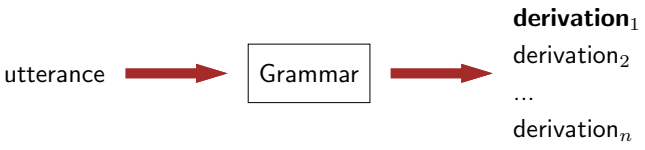
Algorithms

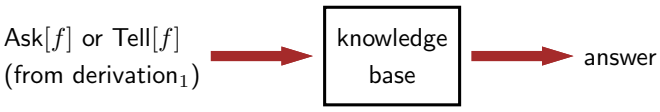


- Inference (parsing): construct derivations recursively (dynamic programming)
- Learning: define ranking loss function, optimize with stochastic gradient descent

- We've seen many different linguistic phenomena, which all can be handled by adding rules. In practice, we construct a parsing algorithm that can output a set of derivations gotten by parsing the utterance with respect to the grammar.
- These algorithms are based on the recursive structure of language and look a lot like dynamic programming algorithms. Specifically, for each subsequence of the sentence, we build a set of sub-derivations.
- In learning semantic parsers, we are given a set of utterances paired with derivations. To learn the parameters, we can define a standard ranking function, which can be optimized with stochastic gradient descent.

Putting it together





[demo: python nli.py]

- The result of the derivation is a logical formula equipped with either an ask or tell request. This can be again used to tell the knowledge base facts or ask questions. This KB could even be probabilistic via Markov logic.

Summary

- Logic is used to **represent** knowledge and perform **inferences** on it
- Considerations: expressiveness, notational convenience, inferential complexity
- First-order logic: objects/relations, quantify over variables
- Semantic parsing: map natural language to logic

- In conclusion, hopefully you have gotten a taste of how powerful logic can be as a means of expressing pretty complex facts using a very small means.
- There is no one true logic, and which one is the best to choose depends on the needs of the application coupled with computational budget.
- Semantic parsing provides an initial step towards the goal of being able to process arbitrary text into a knowledge base and automatically make inferences about it. While we talked mostly about the structural aspects of semantic parsers (e.g., the grammar), there has been a surge of interest lately in making these semantic parsers more robust and scalable. There is still much work to be done though.