CS388: Natural Language Processing

Lecture 20: Language and Code

Greg Durrett

credit: Deepmind

Announcements

‣ Sebastian Gerhmann talk on Tuesday

This Lecture

‣ Semantic parsing
  ‣ Logical forms
  ‣ Parsing to lambda calculus
  ‣ Seq2seq semantic parsing
  ‣ Language-to-code
  ‣ Applications in software engineering

Semantic Parsing
**Model Theoretic Semantics**
- Key idea: can ground out natural language expressions in set-theoretic expressions called *models* of those sentences
- Natural language statement $S \Rightarrow$ interpretation of $S$ that models it
  
  \[
  \text{She likes going to that restaurant}
  \]
- Interpretation: defines who she and that restaurant are, make it able to be concretely evaluated with respect to a world
- This is a type of truth-conditional semantics: reduce a sentence to its truth conditions (configuration of the world under which it is true)
- Our modeling language is *first-order logic*
- Entailment (statement A implies statement B) reduces to: in all worlds where A is true, B is true

**First-order Logic**
- Powerful logic formalism including things like entities, relations, and quantifications
  
  \[
  \text{Lady Gaga sings}
  \]
- sings is a *predicate* (with one argument), function $f$: entity $\rightarrow$ true/false
- $\text{sings(Lady Gaga)} = \text{true or false}$, have to execute this against some database (*world*)
- Quantification: “forall” operator, “there exists” operator
  
  \[
  \forall x \text{sings}(x) \vee \text{dances}(x) \rightarrow \text{performs}(x) \\
  \text{“Everyone who sings or dances performs”}
  \]

**Montague Semantics**
- Richard Montague: operationalized this type of semantics and connected it to syntax
- Denotation: evaluation of some expression against this database
  
  \[
  [[\text{Lady Gaga}]] = \text{e470} \\
  [[\text{sings(e470)}]] = \text{True}
  \]
  
  Denotation of this string is an entity  
  Denotation of this expression is T/F

**Montague Semantics**
- Database containing entities, predicates, etc.
  
  \[
  \text{e470 Stefani Germanotta Lady Gaga 3/28/1986 T} \\
  \text{e728 Marshall Mathers Eminem 10/17/1972 T}
  \]
  
  Database containing entities, predicates, etc.

**Montague Semantics**
- We can use the syntactic parse as a bridge to the lambda-calculus representation, build up a logical form (our model) *compositionally*
**Combinatory Categorial Grammar**

- Steedman+Szabolcsi (1980s): formalism bridging syntax and semantics
- Parallel derivations of syntactic parse and lambda calculus expression
- Syntactic categories (for this lecture): S, NP, “slash” categories
- S\NP: “if I combine with an NP on my left side, I form a sentence” — verb
- S\NP: “I need an NP on my right and then on my left” — verb with a direct object
- When you apply this, there has to be a parallel instance of function application on the semantics side

**CCG Parsing**

\[
\frac{\text{What}}{(S/(S\backslash NP))/N} \quad \frac{\text{states}}{N \quad \lambda x.\lambda y.\text{borders}(y,x)} \quad \frac{\text{border}}{(S\backslash NP)/NP} \quad \frac{\text{Texas}}{\lambda y.\text{borders}(y,\text{Texas})} \quad S
\]

- “What” is a very complex type: needs a noun and needs a S\NP to form a sentence. S\NP is basically a verb phrase (\textit{border Texas})

Zettlemoyer and Collins (2005)
**CCG Parsing**

- These questions are *compositional*: we can build bigger ones out of smaller pieces
  - *What states border Texas?*
  - *What states border states bordering Texas?*
  - *What states border states bordering states bordering Texas?*

  Zettlemoyer and Collins (2005)

**Training CCG Parsers**

- Training data looks like pairs of sentences and logical forms
  - *What states border Texas* \( \lambda x. \text{state}(x) \land \text{borders}(x, e89) \)
  - *What borders Texas* \( \lambda x. \text{borders}(x, e89) \)
  - ...  

  - Unlike PCFGs, we don’t know which words yielded which fragments of CCG
  - Very hard to build a conventional parser for this problem

  Zettlemoyer and Collins (2005)

**Semantic Parsing as Translation**

- Write down a linearized form of the semantic parse, train seq2seq models to directly translate into this representation (similar to code generation like GitHub Copilot)
  
  **Geo**
  
  \[
  \begin{align*}
  \text{x: "what is the population of iowa ?"} \\
  \text{y: \_answer ( NV , ( \_population ( NV , V1 ) , \_const ( \_g , \_stateid ( iowa ) ) ) )}
  \end{align*}
  \]

  **ATIS**
  
  \[
  \begin{align*}
  \text{x: "can you list all flights from chicago to milwaukee"} \\
  \text{y: \_lambda s0 \_and ( \_flight s0 )}
  \end{align*}
  \]

  **Overnight**
  
  \[
  \begin{align*}
  \text{x: "when is the weekly standup"} \\
  \text{y: \{ call listValue ( call getProperty meeting.weekly_standup ( string start_time ) ) \}}
  \end{align*}
  \]

  Jia and Liang (2016)

**Semantic Parsing as Translation**

- Prolog
  - Handle all of these with uniform machinery!

- Lambda calculus
  - Other DSLs

Jia and Liang (2016)
Applications

- GeoQuery (Zelle and Mooney, 1996): answering questions about states (~80% accuracy)
- Jobs: answering questions about job postings (~80% accuracy)
- ATIS: flight search
- Can do well on all of these tasks if you handcraft systems and use plenty of training data: these domains aren’t that rich

Code Generation

- Suppose we are going to generate source code like in Codex/GitHub Copilot. What differs from generating natural language?
- In spite of these differences, the “obvious” thing is to do some pre-training and see how far we get!

Generating Code

CodeT5

- Key idea: code analogue of T5 that should be able to map language to source code

Wang et al. (2021)
CodeT5

- Predict (a) spans; (c) identifiers; (d) language from code and vice versa
- What’s different from normal T5?

Wang et al. (2021)

CodeT5

- Pre-trained on data from several language and NL
- Applied to several generation tasks: code summarization, generation, and translation (between programming languages)
- Also used for classification like bug detection (can be fine-tuned like BERT-style models)

Wang et al. (2021)

Codex

- GPT-3 additionally fine-tuned on code (although they state that pre-training on NL isn’t really helpful)
  - Modified tokenizer to handle whitespace better. Otherwise, no real modifications!
  - Up to 12B parameter models fine-tuned on Python
  - One challenge is evaluation. How to go beyond BLEU/EM?

Mark Chen et al. (2021)
HumanEval

- Generate standalone Python functions from docstrings and execute them!

```python
def solution(lst):
    """Given a non-empty list of integers, return the sum of all of the odd elements
    that are in even positions.
   "
    Examples
    solution([5, 8, 7, 1]) => 12
    solution([3, 3, 3, 3]) => 9
    solution([30, 13, 24, 321]) => 0
    ""
    return sum(lst[i] for i in range(0, len(lst)) if i % 2 == 0 and lst[i] % 2 == 1)
```

- Handwritten benchmarks evaluated for correctness ("pass@k": generate k, see if one of them works)

Mark Chen et al. (2021)

---

Pass Rate vs Model Size

Mark Chen et al. (2021)

---

HumanEval

- Another setting: can we generate a bunch of samples and then pick the correct one? This would be useful for rejection sampling

- Other experiments: additional fine-tuning on competitive programming problems, docstring generation

---

Reflexion

> go to desk 2
On the desk 2, you see a alarmclock 1, a bowl 1, a mug 3, a pencil 3, and a pencil 2.
> think: Now I find a desklamp (1). Next, I need to use it.
OK.
> use desklamp 1
Nothing happens
> use desklamp 1
Nothing happens
Status: Fail

Reflection: In this environment, my plan was to find a mug then find and use a desklamp. However, the task says to examine the mug with the desklamp. I should have looked for the desklamp first, then looked for the mug. I noticed that the desklamp was found on desk 1. In the next trial, I will go to desk 1, find the lamp, then look for the mug and examine it with the desklamp.

Trial #2
> go to desk 1
On the desk 1, you see a creditcard 3, a desklamp 1, a laptop 2, a mug 1, a pen 1, and a pencil 1.
> think: To solve the task, I need to find and take a mug, then find and use a desklamp.
OK.
> take mug 1 from desk 1
You pick up the mug 1 from the desk 1.

https://twitter.com/johnjnay/status/1639362071807549446

Shinn et al. (2023)
**Reflexion**

- Interact with environment, generate a “reflection” about that interaction, then condition on that interaction for the next round.
- Very little details about this, but very strong results on HumanEval!

![Graph showing performance metrics for different models](https://twitter.com/johnnay/status/1639362071807549446)

Shinn et al. (2023)

**Applications in Software Development**

**Applications**

- Generating complete code is nice, but is very challenging: can’t read the user’s mind, if generated code has errors they may be time-consuming to spot.
- There are a range of applications in software engineering: bug detection, type inference, etc. — solving these subproblems can still help save developers time.
- Here: focus on type inference.

**NL Feedback**

<table>
<thead>
<tr>
<th>Prompt</th>
<th>Expected completion</th>
</tr>
</thead>
</table>
| OLD CODE: ```
>>> Write a python function to find
the sum of the three lowest
positive numbers from a given list
of numbers.
>>> Example:
sun_three_smallest_num([10,20,30,40,50,60,7]) = 37
``` |
| FEEDBACK: This code finds the sum of the smallest 3 numbers, not the smallest 3 positive numbers. It needs to disregard negatives and 0. |
| REFINE: ```
``` |

Improving Code Generation by Training with Natural Language Feedback

Angelica Chen et al. (2023)
Rule-based Type Inference

ML-based Type Inference

Type Inference

- Typing this code snippet:
  ```python
def predict(self, data, chunk_id):
    return model.predict(data, chunk_id)
```  ...
  requires looking at this function:
- Changes are non-local:
  even with GPT-4-length contexts, you usually can’t have a whole project in Transformer context

Type Inference

Can use CodeT5 to predict the types...but what context do we feed it?

- Solution: use static analysis to determine relevant parts of the program
- Use the call graph to assemble a context for CodeT5 consisting of callers, callees, and skeletons of various files
Other Applications

- Bug detection: spot bugs in code
- Comments: code-to-comment translation, updating comments when code has changed, and more (see papers by Sheena Panthaplackel)
- Debugging: ask GPT-4 to fix code given an error message (see Greg Brockman’s GPT-4 demo)
- Program synthesis: have some specification other than language (e.g., input-output examples, formal spec) and produce code to follow that

Takeaways

- Language was being interpreted into logical forms that looked like code for a long time (including in formal semantics)
- Rather than doing this with parsers, now we just use seq2seq models
  - Powerful enough models will almost always generate code that compiles. You don’t need special constraints on the output.
- ...and because of pre-training, rather than using customized DSLs, we just use source code because models have seen more of it