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
  
  *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

  \[ \text{Lady Gaga sings} \]

- \text{sings is a \textit{predicate} (with one argument), function f: entity } \rightarrow \text{ true/false} \]

- \text{sings(Lady Gaga) = true or false, have to execute this against some database (world)}

- Quantification: “forall” operator, “there exists” operator

  \[ \forall x \text{sings}(x) \lor \text{dances}(x) \rightarrow \text{performs}(x) \]

  “\text{Everyone who sings or dances performs}”
Montague Semantics

Database containing entities, predicates, etc.

- Richard Montague: operationalized this type of semantics and connected it to syntax
- Denotation: evaluation of some expression against this database

\[
[[\text{Lady Gaga}]] = e470 \\
[[\text{sings}(e470)]] = \text{True}
\]

denotation of this string is an entity
denotation of this expression is T/F
Montague Semantics

sings(e470)

function application: apply this to e470

\[ \lambda y. \text{sings}(y) \]

- 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
- $S \setminus NP$: “if I combine with an NP on my left side, I form a sentence” — verb
- When you apply this, there has to be a parallel instance of function application on the semantics side

\[ \varepsilon 728: \text{Emily}
\]

\[ S \setminus NP : \lambda y. \text{sings(y)} \]

\[ \text{sings(e728)} \]

\[ S \]

\[ E\text{minem} \rightarrow \text{sings} \]
Combinatory Categorial Grammar

- Steedman+Szabolcsi (1980s): formalism bridging syntax and semantics
- 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)/NP: “I need an NP on my right and then on my left” — verb with a direct object

```
S
sings(e728)

NP
e728
Eminem

S\NP
\(\lambda y.\) sings(y)

\(\lambda y\). borders(y,e89)
borders(e101,e89)
```

```
S
sings(e728)

NP
e728
Eminem

S\NP
\(\lambda y.\) sings(y)

\(\lambda y\). borders(y,e89)
borders(e101,e89)
```

```
S\NP
\(\lambda y\). borders(y,e89)
borders(e101,e89)

NP
e101
Oklahoma

(S\NP)/NP
\(\lambda x.\lambda y\). borders(y,x)
borders(e101,e89)

NP
e89
Texas
```
**CCG Parsing**

<table>
<thead>
<tr>
<th>What</th>
<th>states</th>
<th>border</th>
<th>Texas</th>
</tr>
</thead>
<tbody>
<tr>
<td>$(S/(S \backslash NP))/N$</td>
<td>$N$</td>
<td>$(S \backslash NP)/NP$</td>
<td>$NP$</td>
</tr>
<tr>
<td>$\lambda f. \lambda g. \lambda x. f(x) \land g(x)$</td>
<td>$\lambda x. \text{state}(x)$</td>
<td>$\lambda x. \lambda y. \text{borders}(y, x)$</td>
<td>$\text{texas}$</td>
</tr>
<tr>
<td>$\text{border}$</td>
<td>$\lambda y. \text{borders}(y, \text{texas})$</td>
<td>$\text{Texas}$</td>
<td></td>
</tr>
</tbody>
</table>

- “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 (*border Texas*)

Zettlemoyer and Collins (2005)
What

\[(S/(S\backslash NP))/N\]
\[\lambda f.\lambda g.\lambda x.f(x) \land g(x)\]

states

\[N\]
\[\lambda x.\text{state}(x)\]

\[\rightarrow\]

\[S/(S\backslash NP)\]
\[\lambda g.\lambda x.\text{state}(x) \land g(x)\]

border

\[(S\backslash NP)/NP\]
\[\lambda x.\lambda y.\text{borders}(y, x)\]

\[\rightarrow\]

\[(S\backslash NP)\]
\[\lambda y.\text{borders}(y, \text{texas})\]

Texas

\[NP\]
\[\text{texas}\]

\[\rightarrow\]

\[\lambda x.\text{state}(x) \land \text{borders}(x, \text{texas})\]

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

Why are we talking about this in this lecture? Because this lambda calculus expression is basically executable code.
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?
Training CCG Parsers

- Training data looks like pairs of sentences and logical forms

What states border Texas \( \lambda x. \) state\( (x) \) \& borders\( (x, e89) \)

What borders Texas \( \lambda x. \) 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

“what states border Texas”

\[ \lambda x \ ( \text{state} \ ( x ) \ \text{and} \ \text{border} \ ( x \ , \ e89 ) ) \]

- 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)

- What are some benefits of this approach compared to grammar-based?

- What might be some concerns about this approach? How do we mitigate them?

Jia and Liang (2016)
Semantic Parsing as Translation

- Prolog
- Lambda calculus
- Other DSLs

- Handle all of these with uniform machinery!

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
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)

<table>
<thead>
<tr>
<th>PLs</th>
<th>W/ NL</th>
<th>W/o NL</th>
<th>Identifier</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ruby</td>
<td>49,009</td>
<td>110,551</td>
<td>32.08%</td>
</tr>
<tr>
<td>JavaScript</td>
<td>125,166</td>
<td>1,717,933</td>
<td>19.82%</td>
</tr>
<tr>
<td>Go</td>
<td>319,132</td>
<td>379,103</td>
<td>19.32%</td>
</tr>
<tr>
<td>Python</td>
<td>453,772</td>
<td>657,030</td>
<td>30.02%</td>
</tr>
<tr>
<td>Java</td>
<td>457,381</td>
<td>1,070,271</td>
<td>25.76%</td>
</tr>
<tr>
<td>PHP</td>
<td>525,357</td>
<td>398,058</td>
<td>23.44%</td>
</tr>
<tr>
<td>C</td>
<td>1M</td>
<td>-</td>
<td>24.94%</td>
</tr>
<tr>
<td>CSharp</td>
<td>228,496</td>
<td>856,375</td>
<td>27.85%</td>
</tr>
<tr>
<td>Total</td>
<td>3,158,313</td>
<td>5,189,321</td>
<td>8,347,634</td>
</tr>
</tbody>
</table>

Wang et al. (2021)
CodeT5

- Generation task from CONCODE (Iyer et al., 2018):

```java
public class SimpleVector implements Serializable {
    double[] vecElements;
    double[] weights;

    NL Query: Adds a scalar to this vector in place.
    Code to be generated automatically:
    public void add(final double arg0) {
        for (int i = 0; i < vecElements.length; i++){
            vecElements[i] += arg0;
        }
    }
}
```

- What do you think about this evaluation?

<table>
<thead>
<tr>
<th>Methods</th>
<th>EM</th>
<th>BLEU</th>
<th>CodeBLEU</th>
</tr>
</thead>
<tbody>
<tr>
<td>GPT-2</td>
<td>17.35</td>
<td>25.37</td>
<td>29.69</td>
</tr>
<tr>
<td>CodeGPT-2</td>
<td>18.25</td>
<td>28.69</td>
<td>32.71</td>
</tr>
<tr>
<td>CodeGPT-adapted</td>
<td>20.10</td>
<td>32.79</td>
<td>35.98</td>
</tr>
<tr>
<td>PLBART</td>
<td>18.75</td>
<td>36.69</td>
<td>38.52</td>
</tr>
<tr>
<td>CodeT5-small</td>
<td>21.55</td>
<td>38.13</td>
<td>41.39</td>
</tr>
<tr>
<td>+dual-gen</td>
<td>19.95</td>
<td>39.02</td>
<td>42.21</td>
</tr>
<tr>
<td>+multi-task</td>
<td>20.15</td>
<td>35.89</td>
<td>38.83</td>
</tr>
<tr>
<td>CodeT5-base</td>
<td>22.30</td>
<td>40.73</td>
<td>43.20</td>
</tr>
<tr>
<td>+dual-gen</td>
<td>22.70</td>
<td>41.48</td>
<td>44.10</td>
</tr>
<tr>
<td>+multi-task</td>
<td>21.15</td>
<td>37.54</td>
<td>40.01</td>
</tr>
</tbody>
</table>

Table 3: Results on the code generation task. EM denotes the exact match.  
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, 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)
HumanEval

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)
- 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!

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

OLD CODE:

"""
Write a python function to find the sum of the three lowest positive numbers from a given list of numbers.

>>> Example:
sum_three_smallest_nums([10, 20, 30, 40, 50, 60, 7]) = 37
"""
def sum_three_smallest_nums(lst):
    lst.sort()
    return sum(lst[:3])

FEEDBACK:
This code finds the sum of the smallest 3 numbers, not the smallest 3 positive numbers. It needs to disregard negatives and 0.

REFINEMENT:

Expected completion

"""
Write a python function to find the sum of the three lowest positive numbers from a given list of numbers.

>>> Example:
sum_three_smallest_nums([10, 20, 30, 40, 50, 60, 7]) = 37
"""
def sum_three_smallest_nums(lst):
    lst = [x for x in lst if x > 0]
    lst.sort()
    return sum(lst[:3])

Angelica Chen 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
Type Inference

Rule-based Type Inference
- works
- doesn’t work
- doesn’t care

ML-based Type Inference

• Callers
• Callees
• User-defined Types

slide credit: Jiayi Wei

```
def predict(self, data: ChunkedDataset, n_segs: Optional[int] = None) -> dict[int, list[PythonType]]:
    pred_types = dict()
    for batch in data.data:
        batch["input_ids"] = batch["input_ids"].to(device)
        preds, _ = self.predict_on_batch(batch, n_segs)
        for i, c_id in enumerate(batch["chunk_id"]):
            if n_segs is None:
                pred_types[c_id] = preds[i]
            else:
                span = i * n_segs : (i + 1) * n_segs
                pred_types[c_id] = preds[span]
    return pred_types
```

```
def predict_on_batch(self, batch: dict, n_segs: Optional[int] = None) -> tuple[list[PythonType], dict]:
    ...
```

```
chunks = chunk_srcs(data, window)
return model.predict(chunks, n_segs=None)
```
Type Inference

- Typing this code snippet:

```python
chunks = chunk_srcs(data, window)
return model.predict(chunks, n_seqs=None)
```

...requires looking at this function:

```python
def predict(
    self,
    data: ChunkedDataset,
    n_seqs: Optional[int] = None,
) -> dict[int, list[PythonType]]:
    pred_types = dict()
    for batch in data.data:
        batch["input_ids"] = batch["input_ids"].to(device)
        preds, _ = self.predict_on_batch(batch, n_seqs)
        for i, c_id in enumerate(batch["chunk_id"]):
            if n_seqs is None:
                pred_types[c_id] = preds[i]
            else:
                span = i * n_seqs : (i + 1) * n_seqs
                pred_types[c_id] = preds[span]
    return pred_types
```

- 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

---

**Output types**
- `<extra_id_0>` ModelWrapper
- `<extra_id_1>` TokenizedSrcSet
- `<extra_id_2>` Optional[int]
  ...

**Decoding**

**CodeT5 Decoder**

**CodeT5 Encoder**

**Input code element**
- `def eval_on_dataset(
    model: <extra_id_0>,
    data: <extra_id_1>,
    window_size: <extra_id_2> = None,
) ...`

---

Slide credit: Jiayi Wei
Type Inference

(4000 tokens)

Acc. on common types

Acc. on rare types

<table>
<thead>
<tr>
<th></th>
<th>2020</th>
<th>2022</th>
<th>2022</th>
<th>2023</th>
</tr>
</thead>
<tbody>
<tr>
<td>Typilus</td>
<td>54.05</td>
<td>50.34</td>
<td>54.28</td>
<td></td>
</tr>
<tr>
<td>Type4Py</td>
<td></td>
<td></td>
<td></td>
<td>78.04</td>
</tr>
<tr>
<td>HiTyper</td>
<td></td>
<td></td>
<td></td>
<td>81.43</td>
</tr>
<tr>
<td>CodeT5</td>
<td></td>
<td></td>
<td></td>
<td>61.47</td>
</tr>
<tr>
<td>TypeT5</td>
<td></td>
<td></td>
<td></td>
<td>61.47</td>
</tr>
</tbody>
</table>

Typilus (Allamanis et al.)
Type4Py (Mir et al.)
HiTyper (Peng et al.)
TypeT5 (Wei et al.)

TypeInference

slide credit: Jiayi Wei

Jiayi Wei, Durrett, Dillig (ICLR 2023)
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