Natural Language Processing

Executable semantic parsing - compositionality
Reminder

We saw how to write down logical forms for language utterances

- Entities
- Relations
- Superlatives
- Events
- Anaphora
- ...

But how can we build logical forms incrementally?
Outline

- Compositionality
  - CFG-based semantic grammars
  - Example
  - CCG
  - Lexicon
  - Modeling
Outline

• Compositionality
  – CFG-based semantic grammars
  – Example
  – CCG
  – Lexicon
  – Modeling
Grammars and derivations

<table>
<thead>
<tr>
<th>Simple grammar</th>
</tr>
</thead>
<tbody>
<tr>
<td>(lexicon) Set ⇒ Phrase Chicago ⇒ Chicago</td>
</tr>
<tr>
<td>(lexicon) Binary ⇒ Phrase PlacesLived ⇒ lived in</td>
</tr>
<tr>
<td>(join) Set ⇒ Binary Set b.s ⇒ b s</td>
</tr>
<tr>
<td>(intersect) Root ⇒ Set Set s₁ △ s₂ ⇒ s₁ s₂</td>
</tr>
</tbody>
</table>
Grammars and derivations

Simple grammar

(lexicon) Set $\Rightarrow$ Phrase
Chicago $\Rightarrow$ Chicago

(lexicon) Binary $\Rightarrow$ Phrase
PlacesLived $\Rightarrow$ lived in

(join) Set $\Rightarrow$ Binary Set
b.s $\Rightarrow$ b s

(intersect) Root $\Rightarrow$ Set Set
$s_1 \cap s_2 \Rightarrow s_1 s_2$

Type.Person $\cap$ PlaceLived.Chicago

intersect

Type.Person who PlaceLived.Chicago

join

lexicon

people

PlaceLived

Chicago

lexicon

lived in Chicago
A grammar $G$ is a 4-tuple:

$\Sigma$: \textit{chicago, lived}

$\mathcal{N}$: Binary, Set, Root

$\text{Root} \in \mathcal{N}$: start symbol

$\mathcal{R}$: grammar rules
Grammar rules

Simple grammar

(lexicon)  Set ⇒ Phrase  Chicago ⇒ Chicago
(lexicon)  Binary ⇒ Phrase  PlacesLived ⇒ lived in
(join)   Set ⇒ Binary Set  b.s ⇒ b s
(intersect)  Root ⇒ Set Set  s₁ ∩ s₂ ⇒ s₁ s₂

A rule \( r \in \mathcal{R} \) has:

\[
\begin{align*}
A & \in \mathcal{N} : & \text{left-hand-side (LHS) non-terminal} \\
\alpha & \in (\mathcal{N} \cup \Sigma)^+ : & \text{right-hand-side} \\
f & : & \text{semantic function for building derivations}
\end{align*}
\]

Phrase is a compact way to write all \( \alpha \in \Sigma^+ \)

**Semantic functions are the key component!**
A derivation tree $d^A_{i:j}$ over a span $x = (w_i, \ldots, w_j)$:

- Has category $d.c = A \in \mathcal{N}$
- Has logical form $d.z$
- Has children that are derivations or in $\Sigma^+$
Semantic functions

A function $f : D^k \rightarrow 2^D$

Example:

- $r = A \rightarrow B \ C \ [f]$
- $D^{A}_{i:j} = f(d^{B}_{i:k}, d^{C}_{k:j})$

Lexicon function:

$\text{Lex( lincoln )} = \left\{ \begin{array}{l}
\text{Entity:AbeLincoln lincoln, Entity:LincolnFilm lincoln, ...} \\
\text{lincoln, lincoln}
\end{array} \right\}$
Semantic functions

Join function:

$$\text{Join}(\text{Entity: AbeLincoln, Binary: PlaceOfBirthOf}) = \{\text{Set: PlaceOfBirthOf.AbeLincoln}\}$$

Merge function:

$$\text{Intersect}(\text{Set: Type.City, Set: ReleaseDateOf.LincolnFilm}) = \{\}$$

Return sets (unlike CFGs)
Semantic function

IdentityFn Copy logical form from only child
SelectFn(i) Select logical form from child $i$ (skip words)
DateFn Handle date and time language expressions
What city was Abraham Lincoln born?

How about his wife?
Semantic function

IdentityFn  Copy logical form from only child
SelectFn(i)  Select logical form from child \(i\) (skip words)
DateFn       Handle date and time language expressions
ContextFn    Integrate arbitrary context

What city was abraham lincoln born?
How about his wife?

Flexible! arbitrary logic can be used
Outline

- Compositionality
  - CFG-based semantic grammars
  - Example
  - CCG
  - Lexicon
  - Modeling
Example

./run @mode=simple-lambdadcs

-Grammar.inPaths esslli_2016/class2_demo.grammar

-SimpleLexicon.inPaths esslli_2016/class2_demo.lexicon
Example

california
the golden state
cities in the golden state
towns located in california
rivers bordering california
Example

california
the golden state
cities in the golden state
towns located in california
rivers bordering california

Is the last utterance meaningful?

How can we avoid such logical forms?
Outline

• Compositionality
  – CFG-based semantic grammars
  – Example
  – **CCG**
  – Lexicon
  – Modeling
CCG

slides by Yoav Artzi
CCG

Main difference:

• CCG pushed complexity from grammar to lexicon
• lambda-DCS tries to push it from lexicon to learning
CCG

Main difference:

- CCG pushed complexity from grammar to lexicon
- lambda-DCS tries to push it from lexicon to learning

Rationale:

- CCG originates in linguistics
- Lambda-DCS originates in NLP
CCG

Main difference:

- CCG pushed complexity from grammar to lexicon
- lambda-DCS tries to push it from lexicon to learning

Rationale:

- CCG originates in linguistics
- Lambda-DCS originates in NLP
- CCG-based semantic parsing addressed many issues with type raising, factored lexicons, etc. [Zettlemoyer and Collins, 2007; Kwiatkowski et al., 2011]
Outline

- Compositionality
  - CFG-based semantic grammars
  - Example
  - CCG
  - Lexicon
  - Modeling
The lexicon problem

california  
the golden state  
cities  
towns  
in  

California  
California  
Type.Citytown  
Type.CityTown  
ContainedBy  

How is the lexicon generated?
The lexicon problem

- California
- the golden state
- cities
- towns
- in

How is the lexicon generated?

- Annotation
- Exhaustive search
- String matching
- Supervised alignment
- Unsupervised alignment
- Learning
where  Type.Location
most  $\lambda x.\lambda y.\text{argmax}(y, x)$
how many  $\lambda x.\text{count}(x)$

Useful for function words
Annotation

where Type.Location

most $\lambda x.\lambda y.\text{argmax}(y, x)$

how many $\lambda x.\text{count}(x)$

Useful for function words

city Type.CityTown

lake Type.Lake

area Area

Useful for small domains
Annotation

where \quad \text{Type.Location}
most \quad \lambda x.\lambda y.\text{argmax}(y, x)
how many \quad \lambda x.\text{count}(x)

Useful for function words


\begin{itemize}
\item city \quad \text{Type.CityTown}
\item lake \quad \text{Type.Lake}
\item area \quad \text{Area}
\end{itemize}

Useful for small domains

Large domains? Different languages?
Exhaustive search

city

state

people

area

how many

...
Exhaustive search

city \quad \text{Type.Lake}
state \quad \text{Population}
people \quad \text{Area}
area \quad \lambda x.\text{count}(x)
how many \quad \text{Type.CityTown}
\ldots \quad \ldots

Size of lexicon: $|\Sigma| \times |\mathcal{L}|$

Intractable for large domains
Exhaustive search

\[
\begin{array}{c}
\text{city} \quad \text{Type.Lake} \\
\text{state} \quad \text{Population} \\
\text{people} \quad \text{Area} \\
\text{area} \quad \lambda x . \text{count}(x) \\
\text{how many} \quad \text{Type.CityTown} \\
\ldots \quad \ldots
\end{array}
\]

Size of lexicon: \(|\Sigma| \times |\mathcal{L}|

Intractable for large domains

Exhaustive?

- actress \quad \text{Profession.Actor} \sqcap \text{Gender.Female}
- relative \quad \lambda x . \text{Parent}.x \sqcup \text{Sibling}.x \sqcup \text{Child}.x
- expat \quad \mu x . \text{BirthPlaceOf}.x \neq \text{argmax}(x, \text{PlacesLived})
String match

Knowledge-bases often have language descriptions

- Language specific
- Limited coverage and noisy
String match

Knowledge-bases often have language descriptions

- Language specific
- Limited coverage and noisy
- Useful as a feature at training time

\( \textit{area} \quad \textit{Area} \)
Supervised alignment

Assume access to \((x, z)\) language-logical form pairs

*State with the largest area*

\[ \text{argmax}(\text{Type.State}, \text{Area}) \]
Supervised alignment

Assume access to \((x, z)\) language-logical form pairs

**State with the largest area**

\[
\text{argmax}(\text{Type.State, Area})
\]

Define patterns over \((x, z)\) to generate candidate lexicon entries

\[
\begin{align*}
\text{State} & \quad \text{Type.State} \\
\text{with the largest} & \quad \lambda x. \text{Type}.x \\
\text{largest} & \quad \text{Area} \\
\text{area} & \quad \lambda x. \lambda y. \text{Argmax}(x, y)
\end{align*}
\]
Supervised alignment

Assume access to \((x, z)\) language-logical form pairs

*State with the largest area*

\[
\text{argmax}(\text{Type.State}, \text{Area})
\]

Define patterns over \((x, z)\) to generate candidate lexicon entries

\[
\begin{align*}
\text{State} & \quad \text{State} \\
\text{with the largest largest area} & \quad \text{Type.State} \\
\text{largest area} & \quad \lambda x.\text{Type.x} \\
\end{align*}
\]

\[
\begin{align*}
\text{Area} & \quad \text{Area} \\
\lambda x.\lambda y.\text{Argmax}(x, y) & \\
\end{align*}
\]

Keep counts for each lexical entry as a feature
Supervised alignment

Assume access to \((x, z)\) language-logical form pairs

\[
\text{State with the largest area} \\
\text{argmax}(\text{Type.State}, \text{Area})
\]

Define patterns over \((x, z)\) to generate candidate lexicon entries

\[
\begin{align*}
\text{State} & \quad \text{Type.State} \\
\text{with the largest} & \quad \lambda x.\text{Type}.x \\
\text{largest} & \quad \text{Area} \\
\text{area} & \quad \lambda x.\lambda y.\text{argmax}(x, y)
\end{align*}
\]

Keep counts for each lexical entry as a feature

Predefined patterns

Generalization?
Unsupervised alignment

(Barack Obama, was born in, Honolulu)
(Albert Einstein, was born in, Ulm)
(Barack Obama, lived in, Chicago)

... 15M triples ...

(BarackObama, HasBirthplace, Honolulu)
(Albert Einstein, HasBirthplace, Ulm)
(BarackObama, PlacesLived.Location, Chicago)

... 600M triples ...
Unsupervised alignment

\[\text{grew up in}[\text{Person,Location}] \Rightarrow \text{DateOfBirth}\]
\[\text{born in}[\text{Person,Date}] \Rightarrow \text{PlaceOfBirth}\]
\[\text{married in}[\text{Person,Date}] \Rightarrow \text{Marriage.StartDate}\]
\[\text{born in}[\text{Person,Location}] \Rightarrow \text{PlacesLived.Location}\]

(RandomPerson,Seattle) (BarackObama,Honolulu) (BarackObama,Chicago) (MichelleObama,Chicago)
Unsupervised alignment

grew up in [Person, Location]
born in [Person, Date]
mARRIED in [Person, Date]
bORN in [Person, Location]
[Person, Date]
[Person, Location]
[Person, Date]
[Person, Location]

Alignment features
phrase-count: 15,765
intersection-count: 6,048

(RandomPerson, Seattle)
(BarackObama, Honolulu)
(MichelleObama, Chicago)
(BarackObama, Chicago)
Unsupervised alignment

Alignment features

<table>
<thead>
<tr>
<th>phrase-count</th>
<th>intersection-count</th>
</tr>
</thead>
<tbody>
<tr>
<td>15,765</td>
<td>6,048</td>
</tr>
</tbody>
</table>

Generalizes to test set

Restricted domain
Advanced

- Change lexicon at training time
- Batch voting
- Generative models

[Kwiatkowski et al., 2010; Kwiatkowski et al. 2011; Artzi et al., 2014; Krishnamurthy, 2015]
Outline

• Compositionality
  – CFG-based semantic grammars
  – Example
  – CCG
  – Lexicon
  – Modeling
Bridging/trace predicates

Problem: Language defines semantics implicitly

- *Italy’s language*  
  Type.Language ⊓ LanguageOf.Italy
- *Where is Beijing*  
  Type.Location ⊓ Contains.Beijing
- *Boston flight*  
  Type.Flight ⊓ To.Boston
Bridging/trace predicates

Problem: Language defines semantics implicitly

- Italy’s language: Type.Language $\sqcap$ LanguageOf.Italy
- Where is Beijing: Type.Location $\sqcap$ Contains.Beijing
- Boston flight: Type.Flight $\sqcap$ To.Boston

Lexicon?

- ’s: LanguageOf
- is: Contains
- ???: To
Naive solution

language Type.Language
language LanguageOf
where Type.Location
where Contains
flight Type.Filght
flight To
flight From

More pressure on lexicon learning
Naive solution

More pressure on lexicon learning

Infer missing predicates from context

Can be implemented as a semantic function! (BridgeFn)
Bridging 1: two unaries
Bridging 1: two unaries

Type.Flight ⊓ To

Insert any type-matching binary

(bridge first) Set ⇒ Set Set
Bridging 1: two unaries

Type.Flight ⊓ To.Boston

Insert any type-matching binary

(bridge first) Set ⇒ Set Set

Learn to choose implicit predicate
Bridging 2: single unary

Beijing

Where is Beijing
Bridging 2: single unary

\[
\text{bridging}
\]

Contains

\[
\mid
\]

Beijing

\[
\mid
\]

\text{lex}

\text{Where is Beijing}

Insert any type-matching binary

\[
\text{Contains.Beijing}
\]

\[
(\text{bridge}) \quad \text{Set} \quad \Rightarrow \quad \text{Set}
\]

More uncertainty
Bridging 3: n-ary relations

Marriage.Spouse.Madonna

join

Madonna  Marriage.Spouse  2000

lex  lex

Who  did  Madonna  marry  in  2000
Bridging 3: n-ary relations

Marriage.(Spouse.Madonna ⊓ StartDate.2000)
Bridging 3: n-ary relations

Marriage.Spouse.Madonna Marriage.StartDate

Who did Madonna marry in 2000

Later: a more general solution called floating parser

• Wang et al., 2015; Pasupat and Liang, 2015
Type raising

Traverses Oregon

Traverse(Oregon)

Traverse  Oregon

traverses  Oregon
Type raising

Traverses Oregon

Traverse(Oregon)

Traverse (Oregon) or Traverse (Nevada)

traverses Oregon or traverses Nevada

Traverse (Oregon) ⊔ Traverse (Nevada)

Traverse (Oregon) or Traverse (Nevada)

traverses Oregon or traverses Nevada
Type raising

Traverse(Oregon) \sqcup Nevada

Traverse(Oregon) or Nevada

Traverse Oregon Nevada

traverses Oregon

How to apply the binary on multiple arguments?
Solution 1

\((\text{ConstFn } \lambda x.\lambda y.\lambda b. b(x) \sqcup b(y)) \text{ Disj } \Rightarrow \text{ or}\)
Solution 1

\[(\text{ConstFn } \lambda x.\lambda y.\lambda b. b(x) \sqcup b(y)) \quad \text{Disj} \quad \Rightarrow \quad \text{or}\]

*Traverses Oregon or Nevada or Colorado*

Another lexical entry?
Type raising

Type raise the entity to a function that is applied on a binary

\[
\text{(ApplyFn } \lambda x.\lambda f.f(x)) \quad \text{FuncSet } \Rightarrow \text{Set}
\]
\[
\text{(ConstFn } \lambda a.\lambda b.\lambda g.a(g) \sqcup b(g)) \quad \text{Disj } \Rightarrow \text{or}
\]

\[
\text{Traverse(Oregon) } \sqcup \text{ Traverse(Nevada)}
\]

\[
\text{Traverse} \quad \lambda g.g(\text{Nevada}) \sqcup g(\text{Oregon})
\]

\[
\text{traverses} \quad \lambda f.f(\text{Oregon}) \quad \lambda b.\lambda g.g(\text{Nevada}) \sqcup b(g)
\]

\[
\text{Oregon} \quad \lambda a.\lambda b.\lambda g.a(g) \sqcup b(g) \quad \lambda f.f(\text{Nevada})
\]

\[
\text{Oregon} \quad \text{or} \quad \text{Nevada}
\]
Philosophy: simple grammar and lexicon

• Capturing correct derivations only requires complex rules
Philosophy: simple grammar and lexicon

- Capturing correct derivations only requires complex rules
- Simple rules generate overapproximation of good derivations
Philosophy: simple grammar and lexicon

- Capturing correct derivations only requires complex rules
- Simple rules generate overapproximation of good derivations

Disambiguate at learning time!
Summary

- Grammars build derivations which compositionally provide logical forms

- Lexicons map phrases to basic logical forms

- Learning lexicons is hard so we prefer to push the complexity to learning