Generating CCG Categories

Yufang Liu, Tao Ji, Yuanbin Wu, Man Lan

School of Computer Science and Technology, East China Normal University
{yfliu.antlpl, taoji.cs}@gmail.com, {ybwu, mlan}@cs.ecnu.edu.cn

Abstract

Previous CCG supertaggers usually predict categories using multi-class classification. Despite their simplicity, internal structures of categories are usually ignored. The rich semantics inside these structures may help us to better handle relations among categories and bring more robustness into existing supertaggers. In this work, we propose to generate categories rather than classify them: each category is decomposed into a sequence of smaller atomic tags, and the tagger aims to generate the correct sequence. We show that with this finer view on categories, annotations of different categories could be shared and interactions with sentence contexts could be enhanced. The proposed category generator is able to achieve state-of-the-art tagging (95.5% accuracy) and parsing (89.8% labeled F1) performances on the standard CCGBank. Furthermore, its performances on infrequent (even unseen) categories, out-of-domain texts and low resource language give promising results on introducing generation models to the general CCG analyses.

Introduction

Supertagging is the first step of parsing natural language with Combinatory Categorial Grammar (Steedman 2000) (Figure 1). The morpho-syntax enriched categories are of great interest not only because they can help to build hierarchical representations of sentences, but also because they provide a compact and concise way to encode syntactic functions behind words. As many other data-driven NLP models, applications of CCG analyses are constrained with the quality of annotations and the pre-defined category set. Here, for helping parsing texts in various domains, we aim to improve the robustness of current taggers and extend their abilities on discovering new unknown categories.

The primary model for CCG supertagging is sequence labeling: a classifier autoregressively predicts categories of sentence words. Internal structures of categories, however, are often ignored in this classification-based methods. As a key feature of CCG, these structures are actually quite informative for handling relations among categories. For example, the two categories in Figure 2 are different, but the functions they represent have the same combining strategy (first takes an argument from left, then from right) and identical argument types (two NPs). For building data-driven taggers, this fine-grained view on categories is able to expose more shared information and thus helps to build a more robust model. For instance, we may rely on internal structures to improve performance of infrequent categories by transferring knowledge from more frequent categories (which are learned more robustly). We can also use them to induce unknown categories by building new structures or filling new arguments, which is impossible for existing supertaggers.

Following Kogkalidis, Moortgat, and Deoskar (2019) work on fine-grained type-logical category generation, in this paper, we propose generation paradigms for CCG supertagging. Instead of viewing categories as simple class labels, we decompose them into smaller atomic tags. Predicting a category is now equal to generate the corresponding atomic tag sequence. For example, one decomposition of \((NP\backslash NP)/NP\) could be \([, NP, [, NP, ], NP]\) which is identical to the same decomposition of category \((S\backslash NP)/NP\) except the first NP is replaced by S. Based on the tag sequences, the classifier can know more about the shared and the private learning sig-

![Figure 1: An example of CCG supertagging and parsing. “United” and “Miami” are noun phrases (NP). The transition verb “serves” has a category (supertag) “(S\backslash NP)/NP” which means it first combines a right NP, then combines a left NP, and finally forms a sentence S.](image1.png)

![Figure 2: Two categories with similar internal structures. The left represents a transitive verb, and the right represents a preposition attached to a noun phrase.](image2.png)
With various decoding oracles and a simple reranker, the word
with BERT). Furthermore, on low frequency and unseen cat-
with vanilla sentence-level sequence to sequence generation
which runs a transition system to get tag sequences in prov-
cably first encodes sentence words into vectors, then per-
supertagger predicts a tag sequence \( t \)
set \( \mathcal{T} \) that can make the fine-grained sharing of
tagging. In the following section, we are going to
structures of categories can make the fine-grained sharing of
annotation possible. In the following section, we are going to
characteristic, which is denoted \( S, NP, N, PP \). In CCG anal-
s, the category generator is significantly better than the
tag-wise Generator
To explore the inner structures of categories, we first decom-
pose them into smaller atomic tags\(^1\). For example, a category
\([NP\setminus NP]/NP\) (prepositions attached to noun phrases) can be
seen as a sequence,
\[
[\{, NP, \backslash, NP, \}, /, NP].
\]
One advantage of such decomposition is that now the tag has a
connection with a different category (SNP)/NP. Specifically,
a model can recognize that both of the categories require a left NP. Atomic tags make these connections
explicit (rather than hidden in model parameters) and provide a
way for including them in taggers.

Atomic tags can also make recognizing unknown category
possible. For categories not shown in the pre-defined label
set, the classification model can never predict them correctly.
However, by decomposing into atomic tags, even if a category
is not presented in the label set, it is highly possible that
subsequences of the category have been seen in the training
set, which enables the model to generate correct unknown
categories.

Different from Kogkalidis, Moortgat, and Deoskar (2019),
we propose to deploy decoders for each individual word in-
stead of decoding a single sequence for all words. Our setting
may have following advantages, first of all, it is less sensi-
tive to error propagation among tags due to the decoupling
of the decoding sequence. Second, tags can be parallelized
in the same sentence. Third, the decoder can explicitly in-
clude knowledge of the current word which fits the idea of
assigning tags to words.

Formally, we define \( \mathcal{T}_a \) to be an atomic tag set of the orig-
inal set \( \mathcal{T} \) if for every category \( t \) in \( \mathcal{T} \), it can be expressed
with a sequence of atomic tags in \( \mathcal{T}_a \), \( t = a^1, a^2, \ldots, a^m \)
where \( a^i \in \mathcal{T}_a \) for a sentence word \( x_i \), our tagger’s object
changes to generate the correct sequence of atomic tags. We
deploy two types of sequence decoders for the category gen-
eration task. The first type follows the tag-by-tag generation
paradigm. It is simple and fast, but the generated sequences are
not guaranteed to be well-formed. The second type runs a
transition system. The validity of its output is guaranteed
without the cost of additional computation steps and hard to
tag.

Tag-wise Generator
The tag-wise generator starts an LSTM at every \( x_i \). Let \( g^j_l \)
be the \( j \)-th hidden state of the generator, \( g^j_l =
LSTM(g^{j-1}_l, d^j_l, \theta) \), where \( d^j_i = [h_i; a^{j-1}_i] \), \( h_i \) is the
hidden state vector of \( x_i \) from the encoder (which keeps the
generator watching \( x_i \) at each step), and \( a^{j-1}_i \) represents
the embedding of the output tag from the previous generation
step. The probability of an atomic tag is defined as,
\[
p(a^j_i = a|x) = \text{Softmax}_{a \in \mathcal{T}_a} w_a^T g^j_l. \quad (1)
\]
\(^1\)To avoid confusion, we always use atomic tag to refer to tokens in a (arbitrary) decomposition of original categories, and following the CCGBank’s user manual (Hockenmaier and Steedman 2005), we use atomic category to refer categories without arguments (e.g., S, NP, N, PP, see the manual for a full definition), which is denoted by \( \mathcal{A} \).
\(^2\)An “EOS” is attached to every sequence as a stop sign.
We can therefore adopt parsing algorithms to obtain a well-formed category. Here, we investigate an in-order transition-based generator. We apply attention layers to help such context-aware category generation step, rather than a new generation model. One key point in category generators is how to define the transition action set. In the following, we are going to show different settings of transition is in the supplementary due to the lack of space.

Table 1: The transition system of generating categories.

<table>
<thead>
<tr>
<th>T</th>
<th>Stack</th>
<th>Buffer</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>$S$</td>
<td>$S$</td>
<td>gen($S$)</td>
</tr>
<tr>
<td>1</td>
<td>$S\backslash$</td>
<td>$S$</td>
<td>op($\backslash$)</td>
</tr>
<tr>
<td>2</td>
<td>$S\backslash$NP</td>
<td>$S$NP</td>
<td>reduce</td>
</tr>
<tr>
<td>3</td>
<td>$S$NP</td>
<td>$S$NP</td>
<td>op($/$)</td>
</tr>
<tr>
<td>4</td>
<td>$S$NP/$</td>
<td>$S$NP</td>
<td>gen($NP$)</td>
</tr>
<tr>
<td>5</td>
<td>$S$NP/$/NP$</td>
<td>$S$NP/$NP$</td>
<td>reduce</td>
</tr>
<tr>
<td>6</td>
<td>$S$NP/$/NP$</td>
<td>$S$NP/$NP$</td>
<td>stop</td>
</tr>
</tbody>
</table>

Figure 3: An example of category $(S\backslash$NP)/NP for transition-based generator.

The loss function is $\mathcal{L} = -\sum_i \sum_j \log p(a_i^t|x)$. Comparing with the loss of classification model, the loss here is computed on the finer atomic tags, which can assign credit for partially correct category predictions.

Furthermore, the generator is able to handle relations between sentence contexts and categories in a better way. For example, when generating the second NP in NP/NP, it might be helpful to know whether words on the left form a noun phrase. Due to the decomposition of categories, each atomic tag is able to search related information from the sentence. We apply attention layers to help such context-aware category generation. Specifically, at step $j$ of the category generator on word $i$, we use hidden state $g_{i,j}^{t-1}$ to query which sentence words are more important for predicting the next atomic tag, $\alpha_i^{j,t} = \text{Softmax}_{t \in [1,n]}(w^T \tanh(W_1 g_i^{t-1} + W_2 h_i))$, where $w$, $W_1$, $W_2$ are parameters. A soft aggregation of all encoder vectors $h_i$ becomes a part of the generator’s input

$$d_i^j = [h_i; \alpha_i^{j-1}; \sum_{l=1}^n \alpha_i^{l,j} h_l]. \quad (2)$$

In order to reduce computation costs, we could also compute a single attention vector using $h_i$ as query and apply it in every generation step, $\alpha_i^j = \text{Softmax}_{t \in [1,n]}(w^T \tanh(W_1 h_i + W_2 h_i))$

$$d_i^j = [h_i; \alpha_i^{j-1}; \sum_{l=1}^n \alpha_i^l h_l]. \quad (3)$$

**Transition-based Generator**

We can also explicitly explore tree structures of categories during the generation. In fact, by seeing combination operators (“$\&$”, “$\|$”) as non-terminals, atomic categories as terminals, categories resemble (binarized) constituent trees. We can therefore adopt parsing algorithms to obtain a well-formed categories, which is generally not guaranteed in tag-wise generators. Here, we investigate an in-order transition system (Liu and Zhang 2017), which is a variant of the top-down system (Dyer et al. 2016).

Table 1 illustrates the deduction rules of the transition-based generator. Each transition state contains a stack $\sigma$ and the current timestep $t$. gen($a$) generates an atomic category $a \in A^t$ and pushes $a$ to the stack $\sigma$. op($X$) generates a combination operator $X \in \{t, \backslash\}$ and push $X$ to $\sigma$. reduce combines the top three elements of $\sigma$ and concatenates them to the output $t$. stop is the stopping rule. An example of transition is shown in Figure 3.

At each step of the generation, a classifier predicts which action to perform. Following Dyer et al. (2016), we use a stack-LSTM to encode stack states. The detailed configuration is in the supplementary due to the lack of space.

**Discussions**

The transition-based generator produces categories with provably correct form, which is not guaranteed in the tag-wise generator. On the other side, the tag-wise generator is easier to batch and much faster. Empirically, we find that the problem of illegal categories is not severe in the tag-wise generation: all 1-best outputs of the generator are legal and only 0.05% of 4-best outputs are wrong. In fact, like recent practice of sequence-style parsing (Zhang, Cheng, and Lapata 2017; Fernández-González and Gómez-Rodríguez 2019; Shen et al. 2018), it is possible to drop structure constraints with a well-learned sequence decoder. Categories are usually short (average length is 4) and their number is also limited (10^3). All these factors increase the chance of obtaining well-formed categories directly from the tag-wise generator. We thus focus on this simpler implementation.

We also note that it’s straightforward to apply advanced encoder structures (in fact, we apply BERT(Devlin et al. 2019) in our experiments). However, we would like to think the main contribution here is to study CCG Supertagging from a new perspective, rather than a new generation model.

**Decoding Oracles**

One key point in category generators is how to define the atomic tag set $T_a$ which determines the learning targets (oracles) of the decoder. For the transition-based generator, $T_a$ is simply the transition action set. In the following, we are going to show different settings of $T_a$ for the tag-wise generator. Following the semantics of CCG, we have a natural choice of $T_a$.

$$T_a = A \cup \{\cdot, \backslash, /\}, \quad (AC)$$

where $A$ contains atomic categories $\cdot$ of the grammar.

It’s easy to see that each category $t \in T$ corresponds to a unique atomic tag sequence from $AC$, which forms a deterministic oracle for the category generator.

We can enrich $AC$, for example, with some parentheses expressions (e.g., “NP\NP” in category “(NP\NP)/NP”) in the
original category (which may help to handle some common local syntactic functions),

\[ T_a = \mathcal{AC} \cup \mathcal{P}_k, \]  

(PA)

where \( \mathcal{P} = \{ \tau(\tau) \text{ is a substring of a } t \in \mathcal{T} \} \), \( \mathcal{P}_k \) is the subset of \( \mathcal{P} \) with top-\( k \) frequent items.

Furthermore, we also either completely ignore the semantics of categories by adding their \( n \)-grams or completely accept them by adding all items in \( \mathcal{T} \),

\[ T_a = \mathcal{AC} \cup \mathcal{N}^n_k, \]  

(NG)

\[ T_a = \mathcal{AC} \cup \mathcal{T}, \]  

(OR)

where \( \mathcal{N}^n = \{ \tau(\tau) \text{ is a } n \text{-gram of a } t \in \mathcal{T} \} \), \( \mathcal{N}^n_k \) is the subset of \( \mathcal{N} \) with top-\( k \) frequent \( n \)-grams.

Unlike \( \mathcal{AC} \), when \( T_a \) is set to \( \mathcal{PA} \), \( \mathcal{NG} \) and \( \mathcal{OR} \), a category \( t \) may have more than one correct sequences. For example, with \( \mathcal{PA} \), the tag (NP/NP)\( \backslash \)NP may have two gold standard atomic tag sequences, \( \{ (, NP/NP), , \} \) and \( \{ (, NP, /, NP), , \} \).

We can still pick a deterministic oracle by applying some heuristic rules. Here, the deterministic oracles always perform the longest forward matching (i.e., with a prefix \( a^1, a^2, \ldots, a^m \) is set to a feasible atomic tag with the longest length).\(^3\) On the other hand, we also investigate non-deterministic oracles for training the tag-wise generator. Instead of using a fixed oracle during the entire training process, we select oracles randomly for each category, and all oracles will participate in the learning of the supertagger.

### Re-ranker

To combine a category generator and the category classifier, we further introduce a simple re-ranker. First, using beam search, we can obtain \( k \)-best categories from the category generator. For each category \( t = a^1, a^2, \ldots, a^m \), we assign it a confidence score using probabilities of tags (Equation 1),

\[ u_t = \frac{1}{m^n} \sum_{j=1}^{m^n} \log p(a^j|x) \]  

where \( n \leq 1 \) is a hyperparameter using to penalize long tag sequences.

Next, we use the category classifier to obtain category \( t \)’s probability \( \log p(t|x) \) as its confidence score \( \nu_t \). The final score of \( t \) is defined as the weighted sum of the two scores \( \lambda u_t + (1 - \lambda) \nu_t \). The category with the highest score is taken as the final output. We set \( \nu = 0.15, \lambda = 0.9 \) by selecting them on the development data.

### Experiments

**Datasets and Criteria** We conduct experiments mainly on CCGBank (Hockenmaier and Steedman 2007). We follow the standard splits of CCGBank using section 02-21 for training set, section 00 for development set, and section 23 for test set. There are 1285 different categories in training set, following the previous taggers, we only choose 425 of them which appear no less than 10 times in the training set, and assign UNK to the remaining tags.

For out-of-domain evaluation, we use the Wikipedia corpus (Clark et al. 2009) and the Bioinfen corpus (Rimell and Clark 2009).\(^4\) We also test our models on the news corpus of the Italian CCGBank (Johan, Bosco, and Mazzei 2009). We use the token-POS-category tuples file from the Italian news corpus.\(^5\)

The main criterion for evaluation is tag accuracy. To measure statistical significance, we employ t-test (Dror et al. 2018) with \( p < 0.05 \).\(^6\) The settings of network hyperparameters are in the supplementary. We compare several models,

- CC, the category classifier in Section .
- CG, the tag-wise generator with deterministic oracle AC.
- CGNG2, the tag-wise generator with deterministic NG \( (k = 10, n = 2) \).
- CT, the transition-based system in Section .
- rerank, combining CG and CGNG2 with the ranker (beam size is 4).

### Main Results

Table 2 lists overall performances on CCGBank. C&C is a non-neural-network-based CCG parser, (Lewis, Lee, and Zettlemoyer 2016) is a LSTM-based supertagger similar to our CC model (with less parameters). It also uses tri-training-based semi-supervised learning (Weiss et al. 2015). Shortcut LSTM (Wu, Zhang, and Zong 2017b) performs best in previous works, which uses the shortcut block as a basic architecture for constructing deep stacked models. Their final

\[^3\]We assume there is only one tag with the longest length.

\[^4\]They include 1000 Wikipedia sentences and 1000 biomedical (GENIA) sentences with noun compounds analysed.

\[^5\]We use period to spilt the dataset and get 740 sentences as train/dev/test(8:1:1). Dataset can be download from [http://www.di.unito.it/~tutreeb/CCG-TUT/](http://www.di.unito.it/~tutreeb/CCG-TUT/).

\[^6\]https://github.com/rtmdrr/testSignificanceNLP

<table>
<thead>
<tr>
<th>Model</th>
<th>Dev</th>
<th>Test</th>
<th>Size</th>
<th>Speed</th>
</tr>
</thead>
<tbody>
<tr>
<td>C&amp;C</td>
<td>91.50</td>
<td>92.02</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Lewis, Lee, and Zettlemoyer (2016)</td>
<td>94.10</td>
<td>94.30</td>
<td>48.88</td>
<td>-</td>
</tr>
<tr>
<td>- tri-training</td>
<td>94.90</td>
<td>94.70</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Vaswani et al. (2016)</td>
<td>94.24</td>
<td>94.50</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Wu, Zhang, and Zong (2017a)</td>
<td>94.50</td>
<td>94.71</td>
<td>99.16</td>
<td>-</td>
</tr>
<tr>
<td>Wu, Zhang, and Zong (2017b)</td>
<td>94.72</td>
<td>95.08</td>
<td>189.37</td>
<td>-</td>
</tr>
<tr>
<td>CC</td>
<td>94.89</td>
<td>95.27</td>
<td>77.11</td>
<td>466</td>
</tr>
<tr>
<td>CG</td>
<td>95.10</td>
<td>95.28</td>
<td>79.94</td>
<td>199</td>
</tr>
<tr>
<td>CGNG2</td>
<td>95.26</td>
<td>95.44</td>
<td>80.02</td>
<td>199</td>
</tr>
<tr>
<td>CT</td>
<td>94.06</td>
<td>94.09</td>
<td>77.97</td>
<td>21</td>
</tr>
<tr>
<td>rerank</td>
<td>95.27</td>
<td>95.48</td>
<td>-</td>
<td>-199</td>
</tr>
</tbody>
</table>

**Table 2:** Comparing with existing supertaggers. Model sizes are the number of parameters (MB). Speeds are in sentence per second. We use BERT-base without fine-tuning. All results of our models are averaged over 3 runs. * indicates significantly better.
model uses 9-layer stacked shortcut block as encoder. And a contemporaneous work (Bhargava and Penn 2020) which use a single decoder for the whole sentence. From the results, we find that,

- Our implementation of the category classifier (CC) outperforms the best previous system (Shortcut LSTM) with much less parameters.
- With the same encoder and a small increase of model size, tag-wise generators could bring further performance gains (CG and CGNG2). However, our current transition-based generator underperforms the classification model. Regarding the implementation of transition systems, we adopt the standard stack-LSTM which doesn't fully explore the features of transition structures. It is possible that further feature engineering and advanced encoders will improve the performances. Finally, the reranker can reach a new state-of-the-art in supertaggers using no external data.
- Regarding tagging speeds, since tag-wise generators need additional decoding steps on sentence words, they speeds are roughly two-fifths of the classification model. The transition-based generator is much slower since it needs to build features from the stack, the current output and history actions using LSTMs at every decoding step.
- All of our models obtain an appreciable increase in performance with the help of BERT (Devlin et al. 2019). The results of NGCG2 (with rerank) are comparable to the results of cross view training (Clark et al. 2018) which uses unsupervised data and annotations from other tasks and the contemporaneous work (Bhargava and Penn 2020) which shares the same idea of generating categories.
- We also test our models on the Italian CCGBank, it shows there is no significant difference between the results of CC and CG models. And our CGNG2 model performs best(64.10%). It proves that our tag-wise generators can still perform well with few data. Detailed results are in the supplementary.

Next, we show performances of the tag-wise generator with different oracles (Table 3). In general, comparing with

<table>
<thead>
<tr>
<th>oracles</th>
<th>Dev</th>
<th>Test</th>
</tr>
</thead>
<tbody>
<tr>
<td>CC</td>
<td>-</td>
<td>94.89</td>
</tr>
<tr>
<td></td>
<td>AC</td>
<td>95.10</td>
</tr>
<tr>
<td></td>
<td>PA</td>
<td>95.12</td>
</tr>
<tr>
<td></td>
<td>NG(n = 2)</td>
<td>95.26</td>
</tr>
<tr>
<td></td>
<td>NG(n = 3)</td>
<td>95.13</td>
</tr>
<tr>
<td></td>
<td>NG(n = 4)</td>
<td>95.25</td>
</tr>
<tr>
<td></td>
<td>PA</td>
<td>95.08</td>
</tr>
<tr>
<td></td>
<td>OR</td>
<td>95.00</td>
</tr>
<tr>
<td></td>
<td>NG(n = 2)</td>
<td>95.07</td>
</tr>
<tr>
<td></td>
<td>NG(n = 3)</td>
<td>95.25</td>
</tr>
<tr>
<td></td>
<td>NG(n = 4)</td>
<td>95.20</td>
</tr>
</tbody>
</table>

Table 3: Results of tag-wise generators combined with various oracles. D and ND represent deterministic and non-deterministic oracles. For PA and NG, we set k = 10. Except AC and OR, the improvements are significant (with p < 0.05).

It is interesting to see that n-gram oracles NG perform better (on Dev) than other oracles both on deterministic and non-deterministic settings. We guess that, besides existing atomic categories in AC (and their simple combinations in PA), which have clear definitions from linguistic prior, there still exist some other latent linguistic structures which might help CCG analyses. How to uncover them is our important future work.

- Except NG (n = 3), the non-deterministic oracle is not able to get better accuracies than deterministic oracles. One reason might be that simply using random learning targets may make the generator harder to learn, thus more advanced fusion strategies are desired.
- We have tested oracle AC and NG with larger k (i.e., including more items). The results are similar to those in Table 3, which may suggest that the oracles are not quite sensitive to items’ frequencies when choosing properly.

Third, we show the effectiveness of attention layers. Constrained by our hardware platform, instead of using the default setting, we evaluate a smaller model (the batch size becomes 128, the dimensions of the encoder and the decoder LSTM are decreased to 300 and 200). The results (Table 4) show that, though attention layers require more computation resources, they can help to achieve significantly better tagging accuracies than the vanilla category generator. The two different attention settings (Equation 2, 3) performs nearly the same (thus we may prefer the faster one (Equation 3)).

Finally, we test the percentages of illegal categories generated from category generators, the results show that all 1-best outputs of CC and NGCG2 are legal and only 0.05%, 0.04% of 4-best outputs are wrong. It suggests that it is not hard for tag-wise generators to build well-formed categories given our moderate capacity decoding structures.

### Robustness

By inspecting the CCGBank training set, we see that there are about two-thirds of categories which appear less than 10 times (1−425/1285), and more than half of the remaining

Table 4: Ablation studies. The last row means the model without atomic tag embedding in the last decoding step. △ denotes a smaller model for running attention.
Table 5: Accuracy of infrequent categories on the test set. We group categories with their frequency in the training set. The last row shows the proportion of categories in the test set. * indicates significantly better than model CC.

<table>
<thead>
<tr>
<th>Category</th>
<th>10 ~ 100</th>
<th>100 ~ 400</th>
<th>400 ~ 2000</th>
<th>p@1</th>
<th>p@2</th>
<th>p@4</th>
<th>p@8</th>
</tr>
</thead>
<tbody>
<tr>
<td>CC</td>
<td>60.41</td>
<td>77.06</td>
<td>86.77</td>
<td>11.54</td>
<td>17.31</td>
<td>20.19</td>
<td>25.00</td>
</tr>
<tr>
<td>CT</td>
<td>41.86</td>
<td>63.81</td>
<td>79.73</td>
<td>20.19</td>
<td>32.69</td>
<td>35.58</td>
<td>43.27</td>
</tr>
<tr>
<td>CG</td>
<td>62.44</td>
<td>77.51</td>
<td>87.58*</td>
<td>8.65</td>
<td>14.42</td>
<td>15.38</td>
<td>22.12</td>
</tr>
<tr>
<td>CGNG2</td>
<td>65.83*</td>
<td>78.95*</td>
<td>87.93*</td>
<td>21.15</td>
<td>29.81</td>
<td>31.73</td>
<td>39.42</td>
</tr>
<tr>
<td>rerank</td>
<td>64.25*</td>
<td>79.84*</td>
<td>88.16*</td>
<td>0.96</td>
<td>4.81</td>
<td>5.77</td>
<td>7.69</td>
</tr>
<tr>
<td>% in test</td>
<td>40.46%</td>
<td>17.70%</td>
<td>11.49%</td>
<td>6.73</td>
<td>14.42</td>
<td>25.96</td>
<td>31.73</td>
</tr>
</tbody>
</table>

Table 6: The results on unknown categories. “p@k” measures whether the correct category appears in the top-k outputs of category generators. “w/o feature” means when comparing categories, we ignore their features (e.g., S[cl] equals S).

Table 7: Some examples of prediction on categories not in the training label set.

<table>
<thead>
<tr>
<th>Category</th>
<th>Prediction</th>
</tr>
</thead>
<tbody>
<tr>
<td>S[wq]/N</td>
<td>S[wq]/(S[q]/NP)</td>
</tr>
<tr>
<td>(NP/NP)/N</td>
<td>N/PN/N</td>
</tr>
<tr>
<td>conj/PP</td>
<td>(S[p]/NP)/PP/N</td>
</tr>
<tr>
<td>((S[p]/NP)/PP/PP)/NP</td>
<td>conj</td>
</tr>
<tr>
<td>N/S[sem]</td>
<td>S[wq]/S[dcl]</td>
</tr>
<tr>
<td>S[wq]/S[dcl]/(N</td>
<td>N)/i(N</td>
</tr>
</tbody>
</table>

categories appear less than 100 times (253/425). We now show how the category generator performs on these infrequent (or even unknown) categories, which can be a sign for model robustness.

First, from Table 5, category generators exhibit significantly better tagging accuracies on infrequent categories, and as the annotation becomes less, the gap between generators and the classifier becomes larger. Similarly, we compare systems with respect to the length of a category (empirically, a longer length implies a small frequency). Figure 4 shows that for categories with length less than 16, the models perform almost identical, but for longer categories, the category generator gives more robust tagging results (especially for CGNG2 which has fewer generation steps than CG).

Next, we show performances of category generators on unknown categories. Recall that we only use 425 of 1285 categories in the training set (the remaining categories are tagged with UNK, and we still test all categories on test set). In order to avoid models considering UNK tag as a true tag, for UNK tags in the training set, we exclude their loss during the training (they are still fed into encoders in order to not break the input sentence). On the test set, there are 104 words with categories not included in the 425 training tags, and we show the results on these tags in Table 6. We can observe that, given the top-k candidates, the unseen tags can have a chance to be included, thus the generator might be a reasonable method to deal with unseen categories. We also find that CGNG2 now has lower performances comparing with CG. One reason might be that when generating unseen categories, due to the lack of prior knowledge, the semantic of original atomic categories (established by linguists) are more important than the implicit (raw) information hidden in n-gram tags.

Some failed examples of generating unknown category are shown in Table 7. In the first and second lines, CG gives partially correct results. In the third and fourth line, an argument (PP) is missing. In the fifth and sixth lines, the prediction is mostly right except for wrong features of S (a declarative sentence is predicted as a yes-no question, since we have no special treatment on features of categories, it could be further improved). CG is completely wrong in the last row.

We also show overall performances when we reduce the size of the training set in Figure 5 (which may not increase the number of unknown tags, but provide an approximate setting). The generation model consistently outperforms the classification model with limited training data.

Then, we show the tagging results on out-of-domain data (Table 8) using models trained on the CCGBank. We find that CG performs significantly better than the baseline CC model. Therefore, the robustness of category generator can also extend to texts in different domains.

Parsing Results
To show the CCG parsing performances (Table 9), we feed outputs of our supertaggers into the C&C parser (Clark and Curran 2007). We compare our models with the C&C parser with a RNN supertagger (Xu, Auli, and Clark 2015), the A* parser with a feed-forward neural network supertagger (Lewis and Steedman 2014b), the A* parser with a LSTM supertagger (Lewis, Lee, and Zettlemoyer 2016), the A* parser with a language model enhanced biLSTM supertagger (Vaswani

Figure 4: Accuracy on categories of different lengths.
The work closest to ours is Bhargava and Penn (2020). We investigated in many tasks, such as computer vision (Torralba, Murphy, and Freeman 2007; Bart and Ullman 2005; Lampert, Nickisch, and Harmeling 2014) and transfer learning (Yu and Aloimonos 2010; Rohrbach, Stark, and Schiele 2011). These works perform a multi-class classification on pre-defined category sets and they can’t capture the inside connections between categories because categories are independent of each other. Clark et al. (2018) propose Cross-View Training to learn the representations of sentences, which effectively leverages predictions on unlabeled data and achieves the best result. However, their model needs a large amount of unlabeled data. Vaswani et al. (2016) also want to model the interactions between supertags, but unlike our methods they use a language model to capture these connections. The difference is that we no longer treat every category as a label but a sequence of atomic tags.

The work closest to ours is Bhargava and Penn (2020). We share the same idea of generating categories but there are still some key differences. They decode a single sequence for all words while we deploy decoders for each individual word which may solve some problems (see Section ). Besides, Prange, Schneider, and Sri kumar (2020) also investigate the internal structure of CCG supertag. They treat each category as a single tree (just like our transition system) and use TreeRNNs for tree-structured category prediction.

Seq2Seq model has been used in many NLP tasks, such as machine translation (Sutskever, Vinyals, and Le 2014; Bahdanau, Cho, and Bengio 2015), text summarization (Nal lapati et al. 2016; See, Liu, and Manning 2017), and especially on syntax parsing. More related, Vinyals et al. (2015) and Ma et al. (2017) use Seq2Seq model to generate constituency grammar, and Li et al. (2018), Zhang et al. (2017) use Seq2Seq model to generate dependency grammar. Inspired by their works, we apply Seq2Seq model to generate CCG supertags. But the difference is our generation is token level while theirs are sentence level. By splitting categories into smaller units, we decrease the size of label set. And the results show our category generating model performs well.

Techniques for classifying the unseen label have been investigated in many tasks, such as computer vision (Torralba, Murphy, and Freeman 2007; Bart and Ullman 2005; Lampert, Nickisch, and Harmeling 2014) and transfer learning (Yu and Aloimonos 2010; Rohrbach, Stark, and Schiele 2011). It would be an important future work to introduce advanced algorithms for dealing with these unknown categories.

Conclusion

We proposed a category generator to improve supertagging performance. It provides a new way to capture relations among different categories and recognizing unseen categories. We studied a Seq2Seq-based model, as well as a set of learning targets for the generator. Experiments on CCGBank, out-of-domain datasets and an Italian dataset show the effectiveness of our model. Future work will explore improving the accuracy of non-deterministic oracle and different rerankers. We will also study how to further improve tagging infrequent categories.
Acknowledgments
We would like to thank all reviewers for their helpful comments and suggestions. The corresponding author is Yuanbin Wu. This research is (partially) supported by NSFC (62076097), STCSM (18ZR1411500, 19511120200), and the Foundation of State Key Laboratory of Cognitive Intelligence, iFLYTEK(COGOS-20190003).

References


