IMSurReal Too: IMS at the Surface Realization Shared Task 2020

Xiang Yu, Simon Tannert, Ngoc Thang Vu, Jonas Kuhn
Institut für Maschinelle Sprachverarbeitung
University of Stuttgart, Germany
`firstname.lastname@ims.uni-stuttgart.de`

Abstract

We introduce the IMS contribution to the Surface Realization Shared Task 2020. Our current system achieves substantial improvement over the state-of-the-art system from last year, mainly due to a better token representation and a better linearizer, as well as a simple ensembling approach. We also experiment with data augmentation, which brings some additional performance gain. The system is available at `https://github.com/EggplantElf/IMSurReal`.

1 Introduction

This paper presents our submission to the Surface Realization Shared Task 2020 (Mille et al., 2020). As in the previous year, we participate in both the shallow and the deep track. Additionally, we also participate in the new setting, where additional unlabeled data is permitted to train the models.

Since last year's SR'19 shared task (Mille et al., 2019), we have made some improvements over the winning system, mostly described in Yu et al. (2020). In particular, we designed a new linearization module, where we cast the task of word ordering as a Traveling Salesman Problem (TSP), and use the biaffine attention model (Dozat and Manning, 2016) to calculate the score. The new linearizer greatly reduces the decoding time and improves the performance. We also addressed a problem in the previous system which restricted the output to projective trees by employing a transition-based reordering system, which is inspired by transition-based non-projective parsing (Nivre, 2009).

Inspired by the large improvement from data augmentation in Elder et al. (2020), we utilize extra training data for all languages. We parse unlabeled sentences from news and Wikipedia text, then convert the parse trees into the shared task format, and train our system on the augmented data.

Finally, we apply simple ensembling by majority voting using ten models to improve the stability of the system and further push towards the performance upper bound.

2 Surface Realization System

Our system largely follows our submission from the previous year (Yu et al., 2019), with some major change in the linearization module. It consists of up to five modules working in a pipeline fashion. The first module is linearization, which orders the input dependency tree (shallow or deep alike) into a sequence of lemmata. The second module is reordering, which adjusts the sequence from the previous step such that the new sequence could be non-projective. The third module is function word completion (for the deep track only), which generates function words as the dependents of the content words from the previous step. The fourth module is morphological inflection, which turns the sequence of lemmata into inflected word forms. Finally, the fifth module is contraction, which contracts several words as one token, e.g. from *de el* to *del* in Spanish.

For a more detailed description, we refer the readers to Yu et al. (2020). In this section, we mainly describe the changes we made since last year's shared task.

Figure 1 gives an example of the linearization process (including the reordering), which is our main contribution in this work. We first divide the input dependency tree into subtrees, and use the linearization decoder to order each subtree separately, then combine the ordered subtrees as a full projective sentence respecting the orders in the subtrees. Finally, we use a reordering system to adjust the sentence, and produce potentially non-projective sentences.

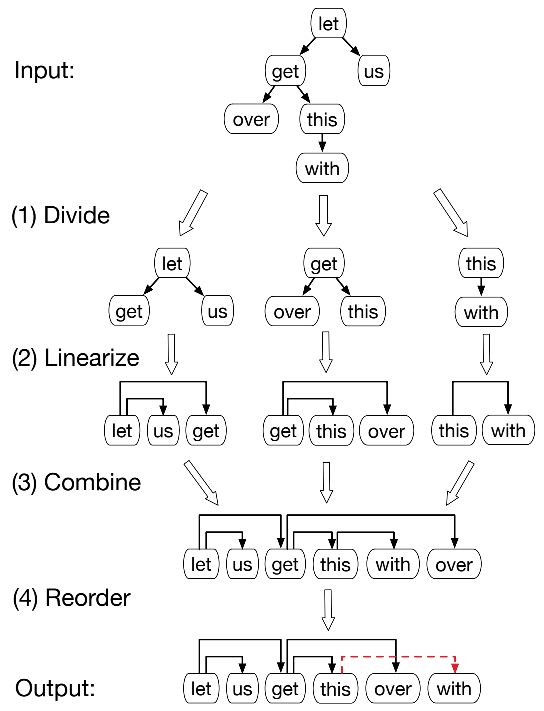


Figure 1: An overview of the linearization module.

2.1 Dependency Tree Encoding

We use the bidirectional Tree-LSTM as in Yu et al. (2019) to encode the input dependency tree. We made two changes in the encoder compared to last year. First, in addition to the default features, i.e., lemma, UPOS, morphological features, and dependency relations, we also use a character-based bidirectional LSTM to encode the lemma, so that out-of-vocabulary words can be better represented. Second, instead of concatenating the feature vectors, we sum them up with the same dimensionality, so that the token representation is more compact and the model has fewer parameters. In addition to the feature-level vectors, we also add the tree-level vectors to the overall representation. Preliminary experiments show that these changes bring substantial improvement.

We run a bidirectional LSTM on the characters in the lemma (denoted as $\mathbf{c}_0$ to $\mathbf{c}_p$) and another bidirectional LSTM for the morphological tags (denoted as $\mathbf{m}_0$ to $\mathbf{m}_q$).

$$\mathbf{v}_i^{(char)} = \overrightarrow{\text{LSTM}}^{char}(\mathbf{c}_0...\mathbf{c}_p) + \overleftarrow{\text{LSTM}}^{char}(\mathbf{c}_0...\mathbf{c}_p) \tag{1}$$

$$\mathbf{v}_i^{(morph)} = \overrightarrow{\text{LSTM}}^{morph}(\mathbf{m}_0...\mathbf{m}_q) + \overleftarrow{\text{LSTM}}^{morph}(\mathbf{m}_0...\mathbf{m}_q) \tag{2}$$

The feature-level representation is then the sum of the feature embeddings from the lemma, UPOS, and dependency relations, together with the LSTM outputs for the lemma characters and morphological tags.

$$\mathbf{v}_i^{(token)} = \mathbf{v}_i^{(lem)} + \mathbf{v}_i^{(pos)} + \mathbf{v}_i^{(dep)} + \mathbf{v}_i^{(morph)} + \mathbf{v}_i^{(char)} \tag{3}$$

We then use the Tree-LSTM as in the previous system to model the tree-level representation, and sum it with the feature-level representation, which is used as the final token representation in the linearization, reordering, and completion.

$$\mathbf{v}_i^{(tree)} = \text{Tree-LSTM}(\mathbf{v}_0^{(token)}...\mathbf{v}_n^{(token)})_i \tag{4}$$

$$\mathbf{x}_i = \mathbf{v}_i^{(token)} + \mathbf{v}_i^{(tree)} \tag{5}$$

For the inflection and contraction modules, since the token sequence is already determined, we additionally use a bidirectional LSTM to model the sequence, and the output vector is also added into the final representation.

$$\mathbf{v}_i^{(seq)} = \text{BiLSTM}(\mathbf{v}_0^{(token)}...\mathbf{v}_n^{(token)})_i \tag{6}$$

$$\mathbf{x}'_i = \mathbf{v}_i^{(token)} + \mathbf{v}_i^{(tree)} + \mathbf{v}_i^{(seq)} \tag{7}$$

2.2 Graph-based Linearization

Similar to our previous system, we take a divide-and-conquer approach by breaking each dependency tree into subtrees, consisting of a head and several dependents, and order the subtrees individually. Finally, we combine the ordered subtrees into a full sentence.

In our previous system, we used beam search to build up the sequence incrementally following Bohnet et al. (2010). It achieves good performance, however, the nature of beam search restricts the training and decoding speed, since the model has to make a calculation at every step. Instead, we use a graph-based decoding method proposed in Yu et al. (2020), in which we model the linearization task as a Traveling Salesman Problem (TSP): each word is treated as a city, the score of a bigram in the output sequence is the traveling cost from a city to another, and the optimal sequence of words is the shortest route that visits each city exactly once. In this formulation, the model only needs to calculate the costs of all the potential bigrams all at once, and an off-the-shelf TSP solver is used to find the optimal sequence. We adapt the biaffine attention model (Dozat and Manning, 2016) to calculate the bigram score, then interpret the score as the traveling cost.

First, we use two multi-layer perceptrons (MLPs) to obtain different views of the token representation as the first and second word in the bigram:

$$\mathbf{h}_i^{(fr)} = \text{MLP}^{(fr)}(\mathbf{x}_i) \tag{8}$$

$$\mathbf{h}_i^{(to)} = \text{MLP}^{(to)}(\mathbf{x}_i) \tag{9}$$

We then apply the biaffine transformation on the vectors of the first word $\mathbf{h}_i^{(fr)}$ and the second word $\mathbf{h}_j^{(to)}$ to compute the score $s_{i,j}$ of each bigram (i, j), where $\mathbf{W}$ is the weight matrix of size $(k+1)\times(k+1)$, and k is the size of $\mathbf{h}_i^{(fr)}$ and $\mathbf{h}_j^{(to)}$:

$$s_{i,j} = (\mathbf{h}_i^{(fr)} \oplus 1) \times \mathbf{W} \times (\mathbf{h}_j^{(to)} \oplus 1)^\top \tag{10}$$

In the actual computation, we parallelize Equation 10, where $\mathbf{S}$ is the output score matrix of size $n \times n$, and n is the number of tokens:

$$\mathbf{S} = (\mathbf{H}^{(fr)} \oplus \mathbf{1}) \times \mathbf{W} \times (\mathbf{H}^{(to)} \oplus \mathbf{1})^\top \tag{11}$$

Finally, we turn the score matrix into a non-negative cost matrix for the TSP solver:

$$\mathbf{C} = \max(\mathbf{S}) - \mathbf{S} \tag{12}$$

We use an off-the-shelf TSP solver, OR-Tools[1], to decode the TSP. In particular, we use a greedy decoding algorithm, which does not guarantee the optimal solution, but typically gives good results, since the TSP size is generally quite small (less than 10 tokens in each subtree).

[1] `https://developers.google.com/optimization`

2.3 Transition-based Reordering

The divide-and-conquer strategy greatly reduces the problem size, which facilitates the TSP decoding, however, it comes with the price that the output sentence is bound to be projective.

To remedy this problem, we introduce a transition system to reorder a projective sentence into a potentially non-projective one. It is essentially a simplified version of the non-projective parsing system by Nivre (2009), with only two transitions, *shift* and *swap*, as shown in Table 1.

Transition	Before		After
shift	$(\sigma,\ [i\|\beta])$	$\rightarrow$	$([\sigma\|i],\ \beta)$
swap	$([\sigma\|i],\ [j\|\beta])$	$\rightarrow$	$(\sigma,\ [j\|i\|\beta])$

Table 1: The *shift-swap* transition system.

We use two LSTMs to model the stack and buffer, then concatenate the output from the top of the stack and the front of the buffer and predict the next transition with an MLP. After each transition, the representation of the stack and the buffer will be updated.

2.4 Subsequent Modules

The subsequent modules are exactly the same as in previous year, we only very briefly describe them, and refer the reader to Yu et al. (2019) for the details.

The function word completion module takes the ordered deep dependency tree as input, and generates function words to the left and right for each content word. Only when a special symbol signifying stopping generation is predicted, we move to generate function words to the next content word.

The inflection module is similar to the hard-monotonic attention model by Aharoni and Goldberg (2017), which uses a pointer to indicate the current input character in the lemma and predicts edit operations to generate the inflected word. To increase the robustness on irregular inflection, we combine the model with a rule-based system, which takes precedence if the lemma and morphological tag combination is frequent and unique in the training data.

The contraction module has two steps. We first assign BIO tags to each word, which group the words that need to be contracted together. The grouped words are then concatenated as a sequence, and passed to a standard Seq2Seq model to generate the contracted word.

2.5 Data Augmentation

On top of the aforementioned improvements, we train our model with additional data to allow it to generalize better. To ensure maximum compatibility with the original training data, we filter large publicly available datasets according to certain criteria.

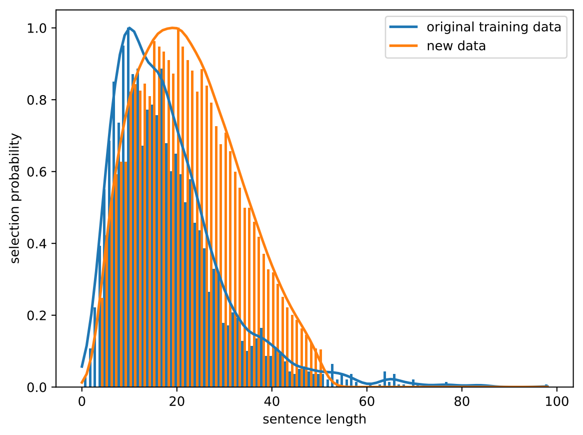
For T1 we use the automatically annotated treebanks from the CoNLL 2017 shared task[2], for T2 we use a subset of the English data from Elder et al. (2020) and for French and Spanish the freely available news datasets from webhose.io[3]. We parse each dataset with UDPipe using the official pretrained models.

In order to maximize compatibility, we filter the new datasets to be more similar to the original training data. We control for sentence length and branching factor by estimating their distributions in both the original training data and the new datasets. To be able to efficiently filter using rejection sampling, we normalize the distributions in the original training data by the maximum value to obtain probabilities between 0 and 1.

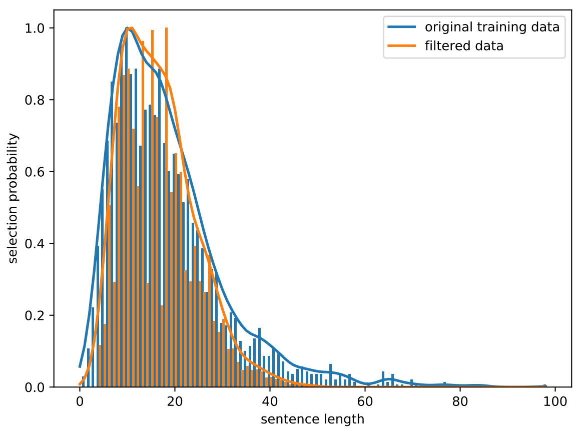
During sampling we decide if we accept or reject a sentence using the product of both probabilities as well as the ratio of old vocabulary. An example of the result of filtering can be seen in Figure 2.

$$\mathbf{p}(s) = \mathbf{D_1}(s) * \mathbf{D_2}(s) * \frac{\|v_{original} \cap v_s\|}{\|v_s\|} \tag{13}$$

[2]`https://lindat.mff.cuni.cz/repository/xmlui/handle/11234/1-1989`
[3]`https://webhose.io/free-datasets/language/`



(a) Distributions of the original data in blue and the new data before filtering in red.



(b) Distribution of the original data in blue and the new data after filtering in red.

Figure 2: Sentence length distribution in *en_lines* before and after filtering

We sample 200,000 sentences for each treebank and convert them into the shared task format as extra training data, which is only used to train the linearization and function word completion modules, since it did not show significant improvement for other modules. The amount of data is rather small compared the 4.5 million sentences in Elder et al. (2020). We also experimented with 1 million sentences, but the improvement was slightly smaller than with 200,000 sentences, the exact reason is yet to be determined.

2.6 Ensembling

We apply a simple ensembling method to further push the performance. We train 10 instances of each module with the same hyperparameters but different random seed, and use majority voting to select the final output sentence. On the development sets, the ensemble prediction performs about 1 BLEU point higher than the single model on average.

For the open track, we train multiple models with different ratios between the original and the augmented data. Concretely, after training one step with the original data, we train N steps with the augmented data, where N ranges from 1 to 10. For different treebanks, the optimal value of N varies drastically, we simply take the majority prediction from these ten models.

2.7 Hyperparameters

For simplicity, we use the same dimensionality of 128 for all the components in the model, including the embeddings, LSTMs, MLPs. All LSTMs have only one layer, and all MLPs have two layers. The total number of parameters range from 1 to 8 millions, mainly depending on the vocabulary size, which is capped at 50,000. We use the Adam optimizer (Kingma and Ba, 2014) with the default parameters for training, and when the model stops improving, we switch to simple stochastic gradient descent to continue training until it again stops improving. No mini-batch or dropout are used during training, as they did not show improvements in the preliminary experiments. The model is implemented with the CPU version of DyNet[4], which takes several hours to train on a single CPU core.

3 Evaluation

This year, we are the only team to submit predictions on all the tracks and languages. The average BLEU scores in the shallow and deep tracks for last year and this year's closed and open subtracks are listed in Table 2, while the detailed evaluations are reported in Mille et al. (2020).

In the closed subtrack, our system improves over the previous year by 3.59 points in T1 and 2.68 points in T2. The improvement mainly comes from a better token representation, a better linearizer, and the model ensembling. In the open subtrack, the augmented data brings an additional gain of 0.49 in T1

[4]`http://dynet.io/`

	2019	2020a	2020b
T1	80.17	83.76	84.25
T2	51.60	54.28	55.76
T1 ∩ T2	84.24	86.53	86.88

Table 2: Comparing the average BLEU scores of our SR'19 results with this year's closed (2020a) and open (2020b) subtrack.

and 1.48 in T2. When we consider only the subset of T1 treebanks that also appear in T2 (T1 ∩ T2), the gain of the additional data is only 0.35. It is clear that the additional training data are more beneficial for the T2 track, since the function word completion module requires more training data. However, the improvement of augmented data in our system is much smaller compared with the ADAPT team which only submitted results for English. This suggests that there is still room for improvement for applying data augmentation in our system.

In the human evaluation, our systems (with or without data augmentation) come in second place for English in terms of meaning similarity and readability, and first for Russian and Spanish.

4 Conclusion

In this paper, we describe our submission to the SR'20 shared task, which is based on our system in SR'19, with some improvements in the feature representation and the linearization module. Additionally, we augment the training data with parsed news and Wikipedia texts. The results show that the additional data can improve the performance to some extent, especially in the deep track, but not as prominent as in Elder et al. (2020). More careful analysis and design are thus required for the augmentation as well as the training process in order to fully capitalize on the vast amount of unlabeled data.

Acknowledgements

Xiang Yu is supported by the Ministry of Science, Research and the Arts of the State of Baden-Württemberg (MWK), within the CLARIN-D research project. Simon Tannert is supported by IBM Research AI through the IBM AI Horizons Network.

References

Roee Aharoni and Yoav Goldberg. 2017. Morphological Inflection Generation with Hard Monotonic Attention. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 2004–2015.

Bernd Bohnet, Leo Wanner, Simon Mille, and Alicia Burga. 2010. Broad Coverage Multilingual Deep Sentence Generation with a Stochastic Multi-Level Realizer. In *Proceedings of the 23rd International Conference on Computational Linguistics*, pages 98–106. Association for Computational Linguistics.

Timothy Dozat and Christopher D. Manning. 2016. Deep Biaffine Attention for Neural Dependency Parsing. *ArXiv*, abs/1611.01734.

Henry Elder, Robert Burke, Alexander O'Connor, and Jennifer Foster. 2020. Shape of synth to come: Why we should use synthetic data for English surface realization. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pages 7465–7471, Online, July. Association for Computational Linguistics.

Diederik P Kingma and Jimmy Ba. 2014. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*.

Simon Mille, Anja Belz, Bernd Bohnet, Yvette Graham, and Leo Wanner. 2019. The Second Multilingual Surface Realisation Shared Task (SR'19): Overview and Evaluation Results. In *Proceedings of the 2nd Workshop on Multilingual Surface Realisation (MSR), 2019 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, Hong Kong, China.

Simon Mille, Anya Belz, Bernd Bohnet, Thiago Castro Ferreira, Yvette Graham, and Leo Wanner. 2020. The third multilingual surface realisation shared task (SR'20): Overview and evaluation results. In *Proceedings of the 3nd Workshop on Multilingual Surface Realisation (MSR 2020)*, Dublin, Ireland, December. Association for Computational Linguistics.

Joakim Nivre. 2009. Non-Projective Dependency Parsing in Expected Linear Time. In *Proceedings of the Joint Conference of the 47th Annual Meeting of the ACL and the 4th International Joint Conference on Natural Language Processing of the AFNLP*, pages 351–359, Suntec, Singapore, August. Association for Computational Linguistics.

Xiang Yu, Agnieszka Falenska, Marina Haid, Ngoc Thang Vu, and Jonas Kuhn. 2019. IMSurReal: IMS at the Surface Realization Shared Task 2019. In *Proceedings of the 2nd Workshop on Multilingual Surface Realisation (MSR 2019)*, pages 50–58, Hong Kong, China, November. Association for Computational Linguistics.

Xiang Yu, Simon Tannert, Ngoc Thang Vu, and Jonas Kuhn. 2020. Fast and accurate non-projective dependency tree linearization. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pages 1451–1462, Online, July. Association for Computational Linguistics.