ESCOLA TÉCNICA SUPERIOR
DE ENXEÑARÍA

P2: DEPENDENCY PARSING

[Master's degree in Artificial Intelligence 2023-24]

20/12/2023

USC Practice Group Members:

Pérez Ferreiro, Pablo Miguel
Romero Romero, Martín

Introduction

In the following work, we have implemented a Dependency Parsing model capable of simulating the workings of the arc-eager parsing algorithm to output the dependency syntactic structure of a given input sentence. For this purpose, we will use the Universal Dependency (UD) treebanks as our datasets and use an arc-eager oracle over them to obtain suitable inputs for our neural networks.

Preprocessing

Before developing the model itself, the first step was to treat and transform the training, validation and test data available. In this case, there were very few design decisions, as the process was quite streamlined by the assignment itself. It was required that we removed both multiword tokens and empty nodes from the treebanks. Everything was done using auxiliary files.

Next, we move from the treebank files themselves to in-memory data structures, in this case Data Frames from the Pandas library. The "text" column would contain the treebank sentences themselves, the "items" column contains the information about individual words, and the "arcs" column contains the arcs between the different words in the sentence. The special id "ROOT" is also added as id 0 and the non-projective sentences (with crossed arcs) are removed.

Once the data frames have been created (for training, validation, and test), we need to create an oracle implementing the arc-eager parsing algorithm, which will allow us to process the sentences using the ground truth information of their arcs to generate an execution trace of actions and dependencies that will later be used to train and test our models.

With the oracle created, we now need to encode the words so that the model can then predict the relationships between them. For that we create a TextVectorization model, to which we pass all the words of the sentences contained in the training file. With this TextVectorization model, we modify the output of the previous oracle so that the numbers assigned to each word by the oracle are replaced by the one assigned by the TextVectorization model.

The last step before being able to train the model is to finish creating the training inputs. For that we create a data frame, in which the columns will be "stack_tokens" (the words in the stack encoded with the TextVectorization), "buffer_tokens" (the words in the buffer encoded with the TextVectorization), "stack_pos" (part of speech tag of the words of the stack), "buffer_pos" (same for the buffer), "action" (action to execute) and "dependency" (type of dependency in the arc to be created, if there is one). From this dataframe, the first four columns will correspond to the model inputs, and the last two to the model outputs. Once this is obtained, we are ready to train our neural network.

Training

For model training, two models have been defined. The first model has four input layers, four embedding layers and two hidden layers (one for each output, action and dependency). Each embedding layer corresponds to each input, and then they are concatenated (Concatenate layer) and finally flattened (Flatten) before being passed to the dense output layers.

Using the test data frame, we see that the accuracy percentages vary according to the number of features passed (i.e. the amount of stack and buffer passed at a time). The results obtained for different features are:

1. For 1 feature (1 stack and 1 buffer): 78% for action prediction and 80% for dependency prediction.
2. For 2 features (2 stack and 2 buffer): 83% for action prediction and 85% for dependency prediction.
3. For 3 features (3 stack and 3 buffer): 83% for the prediction of actions and 85% for the prediction of dependencies.
4. For 4 features (4 stack and 4 buffer): 83% for the prediction of actions and 85% for the prediction of dependencies.

For the second model, we have followed a simpler architecture, equivalent to the one proposed in class, in which an input layer, a single Embedding layer (which is later flattened) and two Dense layers (one for actions and one for dependencies) have been created. This architecture has also been tested with several different features. The results obtained were:

1. For 2 features (2 stack and 2 buffer): 81% for action prediction and 82% for dependency prediction.
2. For 4 features (4 stack and 4 buffer): 81% for action prediction and 83% for dependency prediction.

Prediction and post-processing

Once the model is trained, we need to use it to predict the arcs of our test set (which involves an iterative process of predicting for a given state, updating it, and predicting again until the sentence is fully processed), and then post-process the output. It is possible that some of the trees predicted by the model are not valid, and we need to correct them. For example, nodes without assigned parent, sentences without root, etc. For that we create a function that repairs the trees that are incorrect, and then these new, valid trees are saved as a file in CoNLL-U format so that we can later evaluate the behavior of our model. This evaluation will be carried out using the evaluation program conll18_ud_eval.py

Evaluation

Finally, we are going to evaluate our models with the program conll18_ud_eval.py. This program will provide us with a series of metrics such as accuracy, recall or f1 score that will allow us to know how our model behaves from the CoNLL-U file that we have explained in the previous section. We have applied this evaluation program to all the models described above and the results have been the following:

Table 1: Model metrics

Model	Metric	Precision	Recall	F1-Score	AligndAcc
Model 1, 1 feature	UAS	63.47	63.47	63.47	63.47
	LAS	57.36	57.36	57.36	57.36
Model 1, 2 feature	UAS	68.10	68.10	68.10	68.10
	LAS	61.98	61.98	61.98	61.98
Model 1, 3 feature	UAS	66.58	66.58	66.58	66.58
	LAS	60.22	60.22	60.22	60.22
Model 1, 4 feature	UAS	67.83	67.83	67.83	67.83
	LAS	61.35	61.35	61.35	61.35
Model 2, 2 feature	UAS	60.22	60.22	60.22	60.22
	LAS	53.80	53.80	53.80	53.80
Model 2, 4 feature	UAS	65.86	65.86	65.86	65.86
	LAS	58.34	58.34	58.34	58.34

After analyzing the results, we have observed that the best model is the model with the first architecture, and with 2 features, and thus we've decided to keep it as our final model. It's interesting to see that adding more features did not always result in a better model, as was our initial intuition, at least for the first, more complex model. It seems that the additional context provided to the network was not always that useful, or maybe we weren't working with a data volume big enough to make it count. It also seems that the first model is slightly superior to the second one. The difference between them is slim, and our only explanation for the performance discrepancy is that the use of individual embeddings, each with its own input dimensionality, helps the model understand the different types of input features used a bit better. However, the difference in final performance is not as big as to mean that selecting the simpler model is a critical mistake, and if the structure of said model is easier to understand, then it actually makes sense to go with it.