EDAN20 - Lab 6

André Frisk - an8218fr-s

October 2021

Dependency Parsing

In this assignment there are a lot of information to get a grip on. In this assignment the information used is processed in a queue, stack and a graph. For each sentence we have a dependency graph which there is an action sequence that enables the transition parser to generate this graph. The Gold standard parsing corresponds to the sequence of parsing actions which is *left-arc (la)*, *right-arc (re)*, *shift (sh)* and *reduce (re)* which will be used later when using the oracle. The actions does the following (in their respective function):

- **shift** – shifts the first word in the queue onto the stack
- **reduce** – removes the first item in the stack
- **right-arc** – creates an arc from the top of the stack to the first in the queue and shifts
- **left-arc** – creates an arc from the first in the queue to the top of the stack and reduces it.

There are also some helper functions to manage to get the correct score in this assignment which our teacher have provided for us. The following functions are used mainly to solve the assignment:

- **oracle** – Gold standard parsing, produces a sequence of transitions from a manually-annotated corpus. Returns the modified data-structures but also the deprel of the top word
- **can_reduce** – checks that the top of the stack has a head
- **empty_stack** – pops the items in the stack, if they have no head they are assigned a root head
- **exists_root** – checks if there exists a root in the graph
- **init_config** – a configuration initializer which starts the stack, queue and graph
- **extract** – returns the features in a dictionary format of that sentence (compatible with scikit-learn)
- **queue_stack** – fetches the features used in the extract function
- **right_context** – extend the feature vector with words to the left of the top of the stack
- **can_leftarc** – checks that the top of the stack has no head
- **apply_transition** – modifies the data-structures which is based on the action of in a vector. Returns the modified data-structures and a correct deprel. A bit similar to oracle with some differences.

The following code prepares our X-matrix and y-vector which is then used to train our model for the prediction later. The **depth** variable is how deep we go into the queue/stack. What this code does is that we go through our training corpus where in every sentence we initialize the configuration. While the queue exists (not empty) we extract the features in a dictionary format and add the features to our X and get the action from our **oracle** function which is then inserted in the y-symbols vector. This is done for every sentence in the corpus.

```
X_dict = [] # Matrix
y_symbols = [] # Vector
for sentence in formatted_corpus_train_clean:
    stack, queue, graph = init_config(sentence)
    while queue:
        X_dict.append(extract(depth, stack, queue, graph, sentence))
        stack, queue, graph, action = oracle(stack, queue, graph)
        y_symbols.append(action)
```

From the scikit-learn package we use a DictVectorizer to vectorize our X to a X-matrix. This matrix is used with a Logistic Regression classifier along with the y-vector to fit the model to train it for the final prediction. In the prediction we go through every sentence in the test corpus which we extract the features of the sentence, transform it using DictVectorizer and uses the result of it to predict the right action using the Logistic Regression model. This action is sent in to the `apply_transition` function which modifies the data-structures based on the action that is checked in the function. We o through every sentence until the queue is empty.

Now using this modified test corpus with the CoNLL evaluation script we measure the accuracy of the parser developed and the labelled attachment score should reach 0.67 or higher.

Questions from the Lab

- **Parsing an annotated corpus with an oracle**: When a sentence is nonprojective in this assignment is when the gold-standard annotation of the sentence does not corresponds to the graph. The shortest sentence that is nonprojective is *Vad beror ökningen på?* with the corresponding actions sh, sh, la, ra, ra, sh, sh. We lose points in the annotated graph in comparison to the graph obtained from a sequence of transitions.

Trans.	Stack	Queue	Graph
start	ø	[ROOT, Vad, beror, ökningen, på, ?]	{ }
sh	ROOT	[Vad, beror, ökningen, på, ?]	{ }
sh	Vad / ROOT	[beror, ökningen, på, ?]	{ }
la	ROOT	[beror, ökningen, på, ?]	{Vad <- beror}
ra	beror / ROOT	[ökningen, på, ?]	{Vad <- beror, ROOT -> beror}
ra	ökningen / beror / ROOT	[på, ?]	{Vad <- beror, ROOT -> beror, beror -> ökningen}
sh	på / ökningen / beror / ROOT	[?]	{Vad <- beror, ROOT -> beror, beror -> ökningen}
sh	? / på / ökningen / beror / ROOT	[ ]	{Vad <- beror, ROOT -> beror, beror -> ökningen}

- **Checking gold-standard parsing**: With the information from the transition corpus and the train corpus we can apply seven steps of the first sentence with Nivre's Parser (only the queue and stack as it is stated in the question). Please note that in this figure the stack was difficult to write in Excel and the first element in that section is the top of the stack. We acquire the following:

Trans.	Stack	Queue
start	ø	[ROOT, Individuell, beskattning, av, arbetsinkomster]
sh	ROOT	[Individuell, beskattning av arbetsinkomster]
sh	Individuell / ROOT	[beskattning, av, arbetsinkomster]
la	ROOT	[beskattning, av, arbetsinkomster]
ra	beskattning / ROOT	[av, arbetsinkomster]
sh	av / beskattning / ROOT	[arbetsinkomster]
la	beskattning / ROOT	[arbetsinkomster]
ra	arbetsinkomster / beskattning / ROOT	[ ]

- **Evaluation**: The best score we got was 0.7553614369141679 or 75,53%.

Globally Normalized Transition-Based Neural Networks

In the article *Globally Normalized Transition-Based Neural Networks* by Andor and al. (2016) they use similar techniques as us, dependency parsing. However they also use part-of-speech tagging and sentence compression whilst we only use dependency parsing.