# Language Technology - EDAN20
# Assignment 6 - Fall 2020

Dependency Parsing
Labelled Dependency Graph

Hicham Mohamad, hi8826mo-s

November 15, 2020

## 1 Introduction

In this assignment, we are going to explore **dependency graph** and understand the principles of a **transition-based parser**. Using supervised machine learning techniques, we will extract **features** to learn **parsing actions** from an hand-annotated corpus, and the annotated texts are then used to train classifier/parser to annotate an unseen test set. This assignment is inspired by the CoNLL 2018 shared task of the conference on **computational natural language learning** [7], and the implemented parser will be trained on the Swedish Talbanken corpus. In section 4, two different parsing experiments are solved.

## 2 UD Parsed corpus - CoNLL-U annotation

As in Assignment 5, by using the CoNLL-U reader program, we read/download the latest version of Universal Dependencies (UD 2.6) [6], which contains **treebanks** for 60+ languages. However, in this assignment we will train our parser on training set of the **Swedish Talbanken** corpus and we will evaluate it on the **test set**. Now, it is worthy to remember the following points in order to make processing the corpora easier:

**Annotations in CoNLL-U format:** that annotations in CoNLL-U format are encoded in plain text files where the word lines contain the annotation of a word/token in 10 fields separated by single tab characters. The **10 column names** of the CoNLL-U corpora consists of

```
['ID', 'FORM', 'LEMMA', 'UPOS', 'XPOS', 'FEATS', 'HEAD', 'DEPREL', 'DEPS', 'MISC']
```

**Reading the training corpus:** we use the **CoNLL-U reader** to load the training dataset of Swedish Talbanken corpus. The resulting formatted corpus contains 4303 parsed sentences, where each sentence is a list of lines and each line is a dictionary of columns.

**Removing indices that are not integers:** to ease the processing of some corpora, we remove the indices which are not integers using the function `clean_indicies(formatted_corpus)`. We do this because ID is not necessarily a number.

## 3 Transition parser

In their most general form, the **dependency structures** are simply directed graphs $G = (V, A)$ consisting of a set of vertices $V$, and a set of ordered pairs of vertices $A$, which we refer to as arcs. The set of vertices,

V, corresponds exactly to the set of words in a given sentence. The set of arcs, A, captures the head-dependent and **grammatical function** relationships between the elements in V. An arc from a head to a dependent is said to be **projective** if there is a path from the head to every word that lies between the head and the dependent in the sentence.

For each sentence with a **projective** dependency graph, there is an action sequence that enables the transition parser to generate this graph. **Gold standard parsing** corresponds to the sequence of **parsing actions**:

- **left-arc** (`la`), adds an arc from the top of the stack to the second in the stack and removes the second in the stack.
- **right-arc** (`ra`), adds an arc from the second token in the stack to the top of the stack and pops it.
- **shift** (`sh`), pushes the input token onto the stack.
- **reduce** (`re`), pops the token on the top of the stack.

that produces the manually-obtained, gold standard, graph.

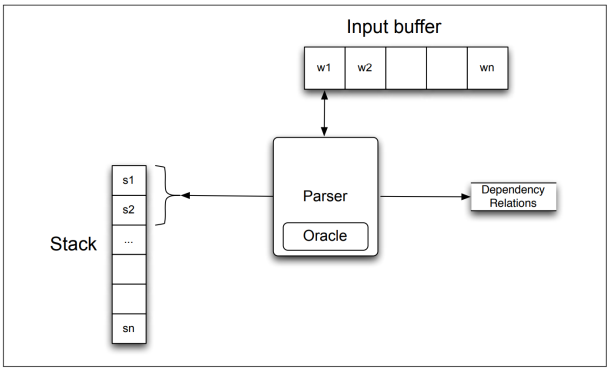
## Transition-based parsing process

A key element in transition-based parsing is the notion of a **configuration** which consists of a stack, an input buffer of words, or tokens, and a set of relations representing a dependency tree.

Input tokens are successively **shifted** onto the stack and the top two elements of the stack are matched against the right-hand side of the rules in the grammar; when a match is found the matched elements are replaced on the stack (**reduced**) by the non-terminal from the left-hand side of the rule being matched.

At the start of this process we create an initial configuration in which the stack contains the ROOT node, the word list is initialized with the set of the words or lemmatized tokens in the sentence, and an empty set of relations is created to represent the parse. In the final goal state, the stack and the word list should be empty, and the set of relations will represent the final parse.

The process ends when all the words in the sentence have been consumed and the ROOT node is the only element remaining on the stack.




(a)

| Clausal Argument Relations | Description |
|---|---|
| NSUBJ | Nominal subject |
| DOBJ | Direct object |
| IOBJ | Indirect object |
| CCOMP | Clausal complement |
| XCOMP | Open clausal complement |
| **Nominal Modifier Relations** | **Description** |
| NMOD | Nominal modifier |
| AMOD | Adjectival modifier |
| NUMMOD | Numeric modifier |
| APPOS | Appositional modifier |
| DET | Determiner |
| CASE | Prepositions, postpositions and other case markers |
| **Other Notable Relations** | **Description** |
| CONJ | Conjunct |
| CC | Coordinating conjunction |

(b)

*Figure 1: Dependency parsing. a) Basic transition-based parser: The parser examines the top two elements of the stack and selects an action based on consulting an oracle that examines the current configuration [3]. b) Examples of core Universal Dependency relations*

## Program utilities

The implemented parser is designed using the following auxiliary functions, so it is nice to give here a short description:

- Transition operators: here are the functions that implement the parsing transitions:
  - `shift(stack, queue, graph)`, shifts the first word in the queue onto the stack.
  - `reduce(stack, queue, graph)`, removes the first item from the stack.

  - `right_arc(stack, queue, graph, deprel=False)`, creates an arc from the top of the stack to the first in the queue and shifts it
  - `left_arc(stack, queue, graph, deprel=False)`, creates an arc from the first in the queue to the top of the stack and reduces it.

- Constrains on the transitions: given a manually-annotated dependency graph, we need to add preconditions on the stack and the current input list (the queue) to execute left-arc, right-arc, shift, or reduce, to assure that these operators are used properly:
  - `can_reduce(stack, graph)`, checks that the top of the stack has a head.
  - `can_leftarc(stack, graph)`, checks that the top of the stack has no head.
  - `can_rightarc(stack)`, simply checks if there is a stack.
- Finding the transitions from a manually-parsed sentence: using an annotated corpus, we can derive all the **action sequences** producing the manually-parsed sentences (provided that they are **projective**) using an **oracle**, `oracle(stack, queue, graph)`.
- Dealing with nonprojective graphs: Oracle parsing produces a sequence of transitions if the graph is projective and well-formed. If not, we will have **headless words** in the stack. Parsing normally terminates when the queue is empty. Using the function `empty_stack(stack, graph)`, we also empty the stack to be sure that all the words have a head. We attach headless words to the root word of the sentence.
- Checking if two graphs are equal: using `equal_graphs()` utility we can check if the graph obtained from a sequence of transitions is equal to the annotated graph. It is normally the case, except with nonprojective graphs.

# 4 Experiments

## Parsing nonprojective sentence with an oracle

For parsing an annotated corpus with an oracle, we run the provided code and produce a sequence of transitions for each sentence. When the graph is projective, applying the sequence to the sentence will recreate the gold-standard annotation.

For this experiment, we can identify a nonprojective sentence by setting `verbose=True` in the code. Looking at the output of the code we choose a short one, i.e.

*Det skall Bayless hålla reda på.*

In fact by examining the annotation of this sentence, we see that it is nonprojective because the arc from head word 1 *"Det"* to dependent word 6 *"på"* is not projective: there is no path to link from the head to every word that lies between them.

Because the parser can only deal with projective sentences, for this nonprojective one, the parsed graph and the manually-annotated sentence are not equal.

## Parsing the first sentence

By applying manually the obtained transition sequence to the first sentence, as shown in Figure 2, we draw a stack and a queue with seven steps, as shown in Table 1.

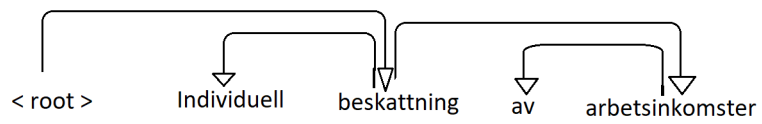



*Figure 2: Drawing the dependency graph to the first sentence: Individuell beskattning av arbetsinkomster*

| Step | Stack | Word List | Action | Relation Added |
|---|---|---|---|---|
| 0 | [root] | [Indiv., beskat., av, arbetsink.] | `shift` | |
| 1 | [root, indiv.] | [beskat., av, arbetsink.] | `shift` | |
| 2 | [root, indiv., beskat] | [av, arbetsink.] | `leftarc` | (indiv. ← beskat.) |
| 3 | [root, beskat.] | [av, arbetsink.] | `shift` | |
| 4 | [root, beskat., av] | [arbetsink.] | `shift` | |
| 5 | [root, beskat., av, arbetsink.] | [ ] | `lefttarc` | (av ← arbetsink. ) |
| 6 | [root, beskat., arbetsink.] | [ ] | `rightarc` | ( beskat. → arbetsink.) |
| 7 | [root, beskat.] | [ ] | `rightarc` | (root → beskat.) |

*Table 1: Trace of a transition-based parse for the first sentence: "Individuell beskattning av arbetsinkomster".*

# 5 Training a classifier

We can now train a classifier to **predict an action** from a current parsing context. To be able to predict the next action from a given parsing state, gold standard parsing must also **extract feature vectors** at each step of the parsing procedure. The simplest parsing context corresponds to words' **part-of-speech** (POS) on the top of the stack and at the head of the input list (the queue).

Once the data is collected, the training procedure will produce a **4-class classifier** that we embed in **Nivre's parser** to choose the next action. During parsing, Nivre's parser will call the classifier to choose the next action in the set {`la, ra, sh, re`} using the current context.

## Parsing the grammatical functions

Using the actions in the set {`la, ra, sh, re`} produces an **unlabelled graph**. It is easy to extend the parser so that it can label the graph with **grammatical functions**. In this case, we must complement the actions `la` and `ra` with their **function** using this notation for example: `la.mod`, `la.case`, `ra.nmod`, etc. where the **prefix** is the action and the **suffix** is the function.

## Extracting features

For the **feature-based classifiers** like logistic regression or random forests the most important step is to identify useful features. The **final goal** is to parse the Swedish corpus and produce a **labelled dependency graph**. To solve this task, we consider **two feature sets** and we train the corresponding **logistic regression** models using scikit-learn:

1. The first set will use the **word** and the **part of speech** extracted from the **top** of the stack and the **first element** in the queue,
2. the second one will use **two first elements** (word forms and POS) from the stack and two from the input list.
3. These two sets will include **two additional Boolean parameters**, which will model **constraints** on the parser's actions:
   - "can do left arc"
   - "can do reduce"

To carry out these features extraction, we take in consideration the following points in the our implementation:

- Feature parameters. In total, the feature sets will then have **six**, respectively **ten** parameters.
- Depth parameter: we use the **depth** parameter for the depth of the stack and the queue, either 1 or 2.
- Features extension: we extend the feature vector with words to the left of the top of the stack with the `right_context()` function. If the top of the stack has index $i$, we can extract the words and their parts-of-speech at index $i + 1$ and $i + 2$. This will noticeably improve the performance.

- We have a function named `extract(depth, stack, queue, graph, sentence)` which returns the features in a **dictionary** format compatible with scikit-learn.
- A loop is implemented to parse the annotated corpus using the **oracle** and collect the features in a **matrix X** and the transitions in a **vector y**. The columns of the matrix correspond to `stack0_POS`, `stack1_POS`, `stack0_word`, `stack1_word`, `queue0_POS`, `queue1_POS`, `queue0_word`, `queue1_word`, `can-re`, `can-la`. Where, we store the matrix in a Python dictionary and the classes in a list.

```
verbose = False
projectivization = False

print('Extracting the features...')
for sent_cnt, sentence in enumerate(formatted_corpus_train_clean):
    #print(sentence)
    if sent_cnt % 1000 == 0:
        print(sent_cnt, 'sentences on', len(formatted_corpus_train_clean), flush=True)

    stack, queue, graph = init_config(sentence)

    while queue:
        # collect features in X and y
        X_dict.append(extract(depth, stack, queue, graph, sentence))

        # parse using oracle
        stack, queue, graph, trans = oracle(stack, queue, graph)
        #transition_sent.append(trans)
        y_symbols.append(trans)


    stack, graph = empty_stack(stack, graph)

    #transition_corpus.append(transition_sent)
    #y_symbols.append(yvec)
    #graph_corpus.append(graph)
    #X_dict.append(Xmat)

    if verbose:
        if not equal_graphs(sentence, graph):
            print('Annotation and gold-standard parsing not equal')
            print('Sentence:', sentence)
            print('Gold-standard graph', graph)

    # Poorman's projectivization to have well-formed graphs.
    # We just assign the same heads as what gold standard parsing did
    # This guarantee a projective sentence
    if projectivization:
        for word in sentence:
            word['HEAD'] = graph['heads'][word['ID']]
print('Extracting the features: Done')
```

# 6 Prediction - Evaluation

After generating and training the **logistic regression model** using Scikit-learn, we use this model to predict the sentences in the **test corpus**. Then, after parsing the test dataset, we evaluate the **accuracies** of the implemented parser using the **CoNLL evaluation script**, where we could reach a **labelled attachment score** (LAS) of 0.7434362271188104.

# 7 Reading

In this task, we need to read the article: *Globally Normalized Transition-Based Neural Networks* by Andor and al. (2016) [pdf] and relate it to our work in this assignment.

In this paper, they demonstrate that simple **feedforward networks** without any recurrence can achieve comparable or better accuracies than long short-term memory netowrks (LSTM), as long as they are **globally normalized**. To empirically demonstrate the effectiveness of global normalization, they evaluate their model on part-of-speech POS tagging, **syntactic dependency parsing** and sentence compression. For dependency parsing on the **Wall Street Journal** they could achieve the best-ever published unlabeled attachment score (LAS) of 94.61%

Without going in details, in the following we can summarize the model and the characteristics related to our assignment:

- To overcome the **label bias problem**, they perform **beam search** for maintaining multiple hypotheses and introduce global normalization with a **conditional random field** (CRF) objective.
- They uses a transition system (Nivre, 2006) and feature embeddings. In this way, the model combines the flexibility of **transition-based** algorithms and the modeling power of neural networks.
- At its core, the model is an **incremental transition based parser** (Nivre, 2006). To apply it to different tasks they only need to adjust the transition system and the input features.
- Transition system: Given an input x, most often a sentence, they define:
  - A set of states $S(x)$
  - A special start state $s \in S(x)$
  - A set of allowed decisions $A(s, x)$
  - A transition function $t(s, d, x)$ returning a new state $s'$ for any decision $d \in A(s, x)$
- They use the **arc-standard transition** system and extract a set of features: words, part of speech tags, and dependency arcs and labels in the surrounding context of the state.
- Data and evaluation: They conducted experiments on a number of different datasets and use the standard parsing splits of the WSJ.

# 8 Appendix

# References


[1] Pierre Nugues, *Language processing with Perl and Prolog*, 2nd edition, 2014, Springer.

[2] P. Nugues, Language Technology - EDAN20, Lectures notes: `https://cs.lth.se/edan20/`

[3] D. Jurafsky, J. Martin, *Speech and Language Processing*, 3rd edition. Online: `https://web.stanford.edu/~jurafsky/slp3/`

[4] VanderPlas, A Whirlwind Tour of Python. O'Reilly 2016. Online reading: `https://jakevdp.github.io/WhirlwindTourOfPython/`

[5] Pierre Nugues, Assignment 6: `https://github.com/pnugues/edan20/blob/master/notebooks/6-dependency_parsing.ipynb`

[6] Universal Dependencies: `https://universaldependencies.org/`

[7] CoNLL 2018 Shared Task, Multilingual Parsing from Raw Text to Universal Dependencies: `http://universaldependencies.org/conll18/`

[8] D Andor et al., Globally Normalized Transition-Based Neural Networks, 2016. Google Inc New York, NY

[9] Joakim Nivre, An Efficient Algorithm for Projective Dependency Parsing, (2003) https://cl.lingfil.uu.se/~nivre/docs/iwpt03.pdf

[10] Geron, Jupyter notebook on linear algebra from Machine Learning and Deep Learning in Python using Scikit-Learn, Keras and TensorFlow: https://github.com/ageron/handson-ml2