Middle East Technical University | Department of Computer Engineering

CENG 463

Introduction to Natural Language Processing

Spring 2017-2018

Homework 3

Due date: 10 May 2018, Thursday, 23:55

1 Objectives

In this assignment, you are expected to implement a graph dependency parser. The parser will work in three steps. The first one is to assign (unlabelled) relation scores between the words in a sentence. The second the step is to extract the maximum spanning tree -which will be the dependency tree- based on these scores. The final step is to label the relations.

You will train two models for this assignment. The first one will assign scores to the possible relations (edges in the graph) and the second one will classify those edges with dependency relations. For MST part, you can implement the related algorithms, use a library or use the provided sample codes.

Keywords: *dependency parsing, graph based dependency parsing, maximum spanning trees*

2 Dependency Parsing

Dependency Parsing is the task of analysing the dependency structure of a sentence that is extracting (head, relation, dependent) triplets to cover all the tokens in a sentence. There are two popular approaches to solving this problem: transition based parsing and graph based parsing. In this assignment, you will implement a simple graph based dependency parser.

A dependency tree is directed graph that covers all the tokens in a sentence. For a sentence with n tokens, a dependency graph -ignoring secondary dependencies- will have $n-1$ edges. This is also a tree with a single root which is also called a spanning tree since it covers all the nodes. Each possible spanning tree can be seen as an alternative dependency tree. The graph based methods try to find best spanning trees after assigning scores to all possible edges, which is generating a directed fully connected graph where the nodes are the tokens in the sentence.

2.1 Edge Prediction

The first step is to assign scores to possible edges between every pair of tokens in the sentence. In order to handle the root token, a dummy can be added to the beginning of the sentence with index 0. Then you need to extracted features for all pairs (except $(i, 0)$ for $i \in (1, n)$ since root cannot depend on any other node). In this step, the labels are ignored, you can use 0/1 binary classification in your models. You objective is to find whether an edge exists or not in the final dependency tree.

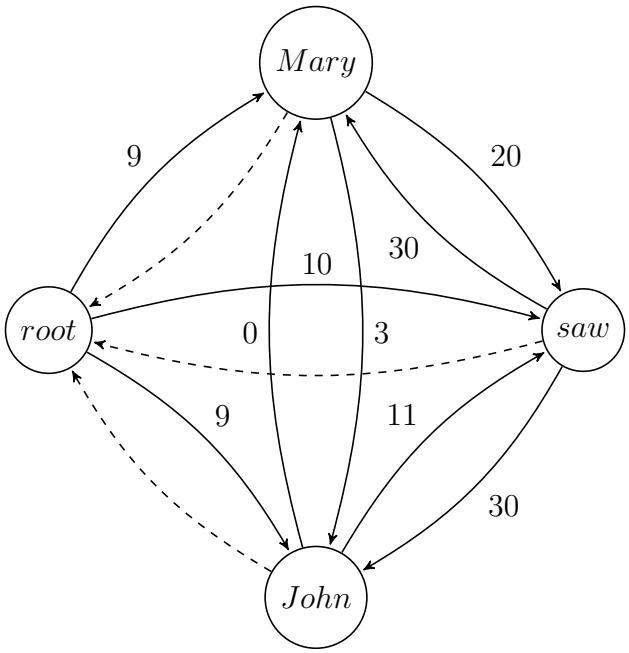

Figure 1: All edges with scores for the sentence *Mary saw John*

If we try to construct a tree by predicting the whether an edge exists or not, we may end up with broken trees and cycles that contain more edges than necessary since we evaluate each edge independently of others. Therefore, we will leave this decision to the second step of the algorithm. In this step we only try to assign scores that are more informative than 0's and 1's. Your models will learn weights for all the features you extracted from the data and calculate a score for for the feature set before outputting a prediction. You can use these scores as the output of the first step. Figure 1 is an example graph with scores assigned to each edge. For reasonable features, check the references.

2.2 Maximum Spanning Trees

Our parsing algorithm considers that the best possible tree is the one with the highest sum of edge scores. Finding that tree is equivalent to finding the maximum spanning tree. Kruskal's and Prim's algorithms for minimum spanning trees work with negative edge weights, therefore they can be used to find maximum spanning trees. However, they do not work on directed graphs. For directed graphs, you can use Chu-Liu/Edmonds' algorithm. There is also Eisner's algorithm that only work on projective dependency trees.

I

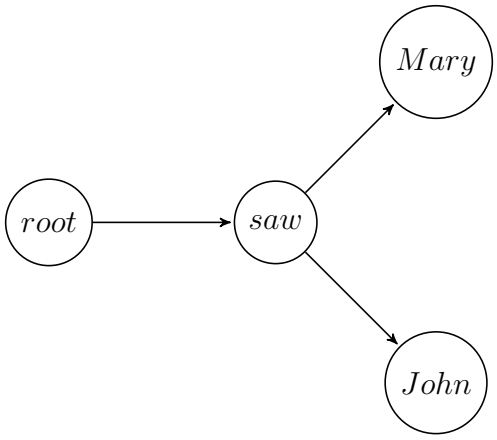

Figure 2: MST extracted from the graph

In this assignment, you are not expected to implement any MST algorithms. An implementation of Chu-Liu/Edmonds' is provided for you. *networkx* library also has Edmonds' implemented. However, you are free to implement your own version or modify the given implementations. Figure 2 shows the MST extracted from the graph in the previous figure.

2.3 Label Prediction

At this stage, you have a dependency tree without any labels. You need another model to predict the dependency relations. This is a separate model from the first step but you can use similar features with different classes as your outputs. This time, you need to use only the edges with dependency relations in the training data. Figure 3 is the final result of our example.

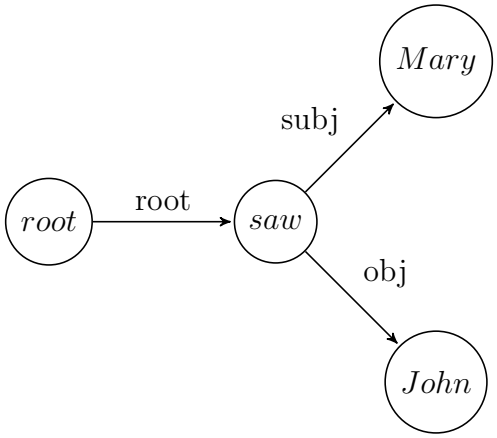

Figure 3: Dependency tree for the sentence *Mary saw John*

3 Data

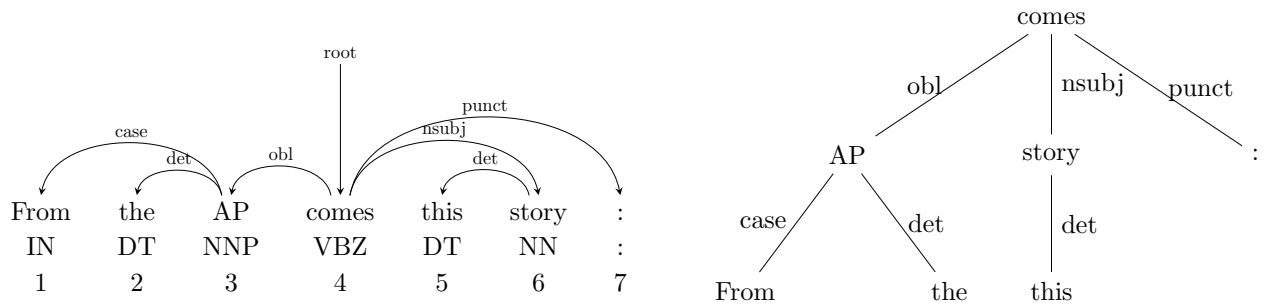

Figure 4: Two ways to show the dependency graph for the sentence *From the AP comes this story :*

The data is a simplified version of English dependency treebank from Universal Dependencies framework [1] that is used in CoNLL 2017 Multilingual Parsing Shared Task. [2] The sentences are separated with empty lines and each line contains a token with its features. There are four tab separated fields in a single line. The first one is the token itself, the second one is the POS tag of the token. The third field will show the head token where the first token of the sentence is indexed with 1. 0 shows the dummy root node. The forth and the final field show the relation between the token and its head.

You have *en-ud-train.conllu* and *en-ud-dev.conllu* files as training and development sections. A sample for the sentence in Figure 4 from the data can be seen below:

```
From    IN    3   case
the     DT    3   det
AP      NNP   4   obl
comes   VBZ   0   root
this    DT    6   det
story   NN    4   nsubj
:       :     4   punct
```

[1] http://universaldependencies.org/

[2] http://universaldependencies.org/conll17/data.html

4 Specifications

4.1 Implementation

1. You will implement a single *Parser* class in *dep_parser.py* module
2. Parser class will have the following methods: *train, test, parse_sentence, parse, save, load*
   See Appendix A for details.
3. You are free to add other classes, modules and methods. Given source files and classes are for guidance only and designed to minimize your work.
4. Extracted features for all data will use a lot of memory, you may not pass all training features to the model (*fit* function for scikit). In that case, you need to pass data in batches (and use *partial_fit* in scikit) You can use SGDClassifier or Perceptron models.
5. For edge prediction you need to assign scores to the edge candidates. The *predict* function will try to assign class labels (in this case 0 or 1). *decision_function* is a good alternative, but you may need to adjust negative scores.
6. You can use NLTK or another library to calculate attachment scores.
7. Submit the best performing parser with parameters and feature set.
8. Submit additional tests, experiments and supplementary materials in other files.
9. See provided test sections in source files for usage.

4.2 Report

1. Explain projective and non-projective dependency parsing, give example sentences.
2. Compare projective and non-projective dependency grammars with context-free grammars.
3. Explain transition-based and graph-based dependency parsing.
4. Explain labelled and unlabelled attachment scores.
5. Explain your feature selection process
   - Rationale behind your features, why do you think they help your parsers?
   - Plot the effects of adding different features, how does the accuracy change?
6. Analyze your parsers' performance with the scores in the metrics above.
   - Take a look at common errors in your parsers and try to comment on them.
7. Did you try to find better parameters for your classifiers? Report your parameter search results with plots.

5 Regulations

1. **Programming Language:** You will use Python 3.
2. In addition to the libraries mentioned in the previous assignment, you can use the library below:
   - networkx : a comprehensive graph library
3. **Late Submission:** You have 3 days of late submission that can be shared between all assignments.
4. **Cheating:** We have zero tolerance policy for cheating. In case of cheating, all parts involved (source(s) and receiver(s)) get zero. People involved in cheating will be punished according to the university regulations.
5. **R**emember that students of this course are bounded to code of honour and its violation is subject to severe punishment.
6. **Newsgroup:** You must follow the newsgroup (news.ceng.metu.edu.tr) for discussions and possible updates on a daily basis.

6 Submission

- Submission will be done via COW.
  Create a tar.gz file named `hw3.tar.gz` that contains all necessary files (including modules and log files, except saved models and data) and your report as a PDF file.

7 References

- Dependency Parsing, Sandra Kübler, Ryan McDonald, and Joakim Nivre, 2009
- Online Large-Margin Training of Dependency Parsers, R. McDonald, K. Crammer, and F. Pereira, 2005
- Non-projective Dependency Parsing using Spanning Tree Algorithms, R. McDonald, F. Pereira, K. Ribarov, and J. Hajič, 2005
- Implementation of the Dependency Parser using Spanning Trees Algorithms, Chen Huang, 2008

A Class Interfaces

A.1 Base Tagger Class

```
class Parser():
    '''
    a dependency parser
    '''

    def __init__(self, params=None):
        '''
        initialize the parser models (and the vectorizers if necessary)
        params is a dict of arguments that can be passed to models
        '''
        raise NotImplementedError("__init__ not implemented")

    def train(self, file_name):
        '''
        extract the features and train the models
        file_name shows the path to the training file with task specific format
        '''
        # maybe separate two tasks
        # self._learn_edges(...args...)
        # self._learn_labels(...args...)
        raise NotImplementedError("train not implemented")

    def test(self, file_name):
        '''
        test the model, extract features and parse the sentences
        file_name shows the path to the test file with task specific format
        return uas and las
        '''
        raise NotImplementedError("test not implemented")


    def parse_sentence(self, sentence):
        '''
        tag a single tokenized and pos tagged sentence
        sentence: [[tkn1, tag1], [tkn2, tag2], ...]
        return the parsed with the same data format
            4 tab separated fields per line
        '''
        raise NotImplementedError("tag not implemented")

    def parse(self, sentences):
        '''
        tag a list of sentences
        return a list of parsed sentences with same data format
        '''
        raise NotImplementedError("tag_sents not implemented")
```

```
    def save(self, file_name):
        '''
        save the trained models to file_name
        '''
        raise NotImplementedError("save not implemented")

    def load(self, file_name):
        '''
        load the trained models from file_name
        '''
        raise NotImplementedError("load not implemented")
```