NLP - Final Group Project Report

Anton Eklund, Eric Wang, Kevin Xu, Clément Veyssière

June 18 2019

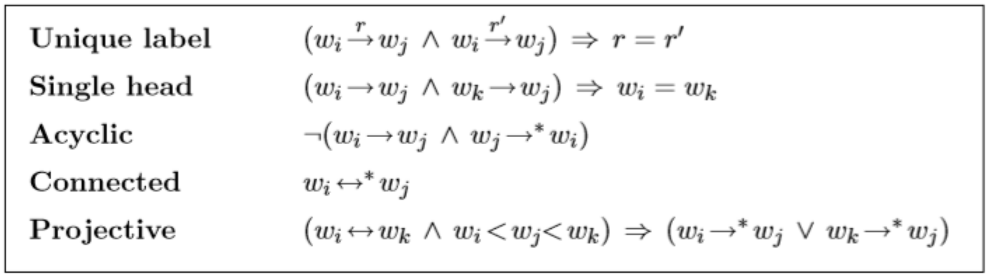
1 Introduction

The goal of this project is to implement the dependency parser of Nivre for the Korean Language version and display the sentence with the dependency parser as a parsing tree. Therefore, a thorough investigation of Nivre's paper Incrementality in Deterministic Dependency Parsing' was done to understand the characteristics and details of the approach. To process natural language, Nivre suggests to take the middle point between *full parsing* and *partial parsing*. The former (non-deterministic) disambiguate entirely the input while the latter (deterministic) partially disambiguate the input and produces a sequence of unconnected phrases. Both are ruling out incrementality which is practical to real-time applications (speech recognition) and useful to theory (connects parsing to cognitive modeling).

Taking out the best of both methods, *deterministic dependency parsing* is syntactically analysing the whole input (full parsing) in a robust, efficient and deterministic way (partial parsing). The path to incrementality seems to be cleared with these features, yet it is not guaranteed. Yamada and Matsumoto (2003) use a multipass bottom-up algorithm, combined with support vector machines, in a way that does not result in incremental processing.

2 Nivre's dependency parser

Dependency parsing is the task of mapping an input sentence satisfying the conditions for being well-formed. Nivre's dependency parser produces a directed graph that satisfies these conditions. We can notice that the 4 first conditions are similar to the characteristics of a tree.



Unique label	$(w_i \xrightarrow{r} w_j \wedge w_i \xrightarrow{r'} w_j) \Rightarrow r = r'$
Single head	$(w_i \rightarrow w_j \wedge w_k \rightarrow w_j) \Rightarrow w_i = w_k$
Acyclic	$\neg(w_i \rightarrow w_j \wedge w_j \rightarrow^* w_i)$
Connected	$w_i \leftrightarrow^* w_j$
Projective	$(w_i \leftrightarrow w_k \wedge w_i < w_j < w_k) \Rightarrow (w_i \rightarrow^* w_j \vee w_k \rightarrow^* w_j)$

Figure 1: Conditions of Well-formedness

We intend to describe briefly the main ideas behind this parser.
The parser configurations at each step are represented by (S, I, A) with :

- S : The processed stack.

- I : The remaining input tokens.
- A : The arcs of the current dependency graph.

2.1 The arc-eager algorithm

The arc-eager algorithm is implemented in order to increase the incrementality of deterministic dependency parsing. Ideally, we would like to require that the graph $(W-I, A)$ is connected at all times for the approach to be incremental. Here are the main steps in this algorithm:

- **Initialization** : We initialize the parser configuration with $([root], I = [w_1, ..., w_n], [])$ where W is the input sentence that we want to parse
- **While** W is not empty
  - Choose the best **action** to apply
  - Apply the chosen action to the parser configuration

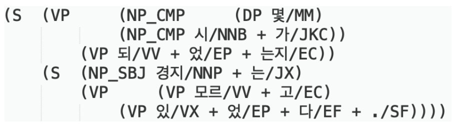
The **actions** possible at each step are :

- **Left-Arc** : Add an arc from the top token of the input stack I to the top token of the processed stack S. Then pop the top token from S.
- **Right-Arc** : Add an arc from the top token of S to the head of the top token of the input stack I. Then push the top token of the input stack on top of S.
- **Reduce** : Pop the top token from S, if there already exists an arc incident to this token.
- **Shift** : Push the top token of the input stack I on top of the processed stack S.

2.2 Dataset

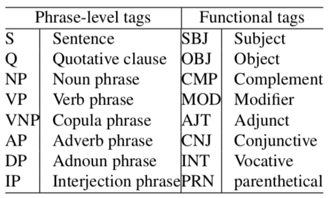
The dataset that was given was the Sejong Institute Language Information Sharing corpus set. It is in the form of a treebank where one sentence in Korean is given and then the corresponding dependency tree.

For example, the following Korean sentence : 몇 시가 되었는지 경지는 모르고 있었다. Its dependency tree is given by :



```
(S  (VP     (NP_CMP     (DP 몇/MM)
            (NP_CMP 시/NNB + 가/JKC))
        (VP 되/VV + 었/EP + 는지/EC))
    (S  (NP_SBJ 경지/NNP + 는/JX)
        (VP     (VP 모르/VV + 고/EC)
            (VP 있/VX + 었/EP + 다/EF + ./SF))))
```

In order to understand the meaning of the different tags used in the dataset, we found the following tables:



Phrase-level tags		Functional tags	
S	Sentence	SBJ	Subject
Q	Quotative clause	OBJ	Object
NP	Noun phrase	CMP	Complement
VP	Verb phrase	MOD	Modifier
VNP	Copula phrase	AJT	Adjunct
AP	Adverb phrase	CNJ	Conjunctive
DP	Adnoun phrase	INT	Vocative
IP	Interjection phrase	PRN	parenthetical

Table 2: Phrase tags used in Sejong treebank.

NNG	General noun	IC	Interjection	JKQ	Quotational CP	XSV	Verb DS
NNP	Proper noun	MM	Adnoun	JX	Auxiliary PR	XSA	Adjective DS
NNB	Bound noun	MAG	General adverb	JC	Conjunctive PR	XR	Base morpheme
NP	Pronoun	MAJ	Conjunctive adverb	EP	Prefinal EM	SN	Number
NR	Numeral	JKS	Subjective CP	EF	Final EM	SL	Foreign word
VV	Verb	JKC	Complemental CP	EC	Conjunctive EM	SH	Chinese word
VA	Adjective	JKG	Adnomial CP	ETN	Nominalizing EM	NF	Noun-like word
VX	Auxiliary predicate	JKO	Objective CP	ETM	Adnominalizing EM	NV	Verb-like word
VCP	Copula	JKB	Adverbial CP	XPN	Noun prefix	NA	Unknown word
VCN	Negation adjective	JKV	Vocative CP	XSN	Noun DS	SF,SP,SS,SE,SO,SW	

Table 1: POS tags used in Sejong treebank (CP: case particle, EM: ending marker, DS: derivational suffix, PR: particle, SF SP SS SE SO: different types of punctuations, SW: currency symbols and mathematical symbols. Table borrowed from (Jinho D. Choi and Palmer, 2011))

Data is imported through python and then ran through both our own functions and konlpy Kkma to clean up and extract important information from the text. We ran into a lot of problems trying to use the text files. We had issues with encoding/decoding because of the Korean and Hanja characters which we are not familiar with manipulating in code. There were also embedded konlpy dependency errors which were hindering our progress.

2.3 Static Grammar rules/Oracle

In the beginning of the project we discussed how we should implement grammar rules for the Korean language. Obviously, this would be a problem for us because no one of the group members know Korean good enough to interpret the meaning behind the grammatical rules. We tried to figure out a way to implement this for a long time because both the project instructions and Nivre 2003/2008 indicated that this was the way to do it. Basically, to try and write a long list of cases comparing labels which in the end would result in one transition choice(left-arc, right-arc, reduce, shift). We imagined that there were many rules that needed to be implemented and therefore also try to generalize much to no success. This task was overwhelming and we could not wrap our heads around how this could be implemented. This made us look for a different approach.

We found out the concept of a neural network oracle way too late. The oracle's role is to make the choice of transition based on learnt data from the treebank. This would solve our problems of implementing all the different static grammar rules because we don't need any understanding of Korean grammar. If we were to implement it, the oracle would learn from taking the current state c as input where c is {step, S, I}. Words in S and I would be represented as unique (w, label) pairs transformed to a one hot vector (it might be sufficient to only have the labels here but we obviously did not have time to try). The output would be one of the four transitions {left-arc, right-arc, reduce, shift}. When training the oracle we would simulate the Nivre arc-eager algorithm. The model chooses one transition based on data and then the loss is calculated from the real transition used in the treebank. The reason we could not train the oracle was because we did not figure out a way to turn the data from the dependency tree into Nivre algorithm transition steps. Basically back-engineering our way from the final tree to the steps that made up the tree. Then after that we could have used the oracle as a predictor in the algorithm. It's used as an addition to the original algorithm by checking if the transition would benefit the algorithm reaching the optimal dependency tree (the golden tree G_{gold}). The oracle.py which is the second appendix contains some ideas of how the deep learning would proceed with TensorFlow.

3 Conclusion

Our project failed so there is no new discoveries or information to add. We can by our extensive research conclude that the Nivre-type parser seems to be a well-working instrument to create a dependency tree for sentences in any language. Many examples on how the algorithm works and is used are easy to find. However, many papers assume that the reader understands how to use a treebank, which in our case is false. There is very little information on how the oracle works and how it can be implemented. This goes both for a neural oracle and a rule based one. That could be something for a successful group to post on a forum.

Further exploration of the oracle might complete our algorithm, for which time is not kind to us. Ideally, if the oracle is functional, the next step would construct a classifier model through machine learning based on the treebank dataset. From this, a dependency tree can be built with the help of python libraries.

What was the most time consuming was finding ways to use the dataset. It took a long time just to import it and print it in a console due to mentioned Hangeul and Hanja. Then we were not sure what to do with the dataset because we can't make the connections between sentence meaning and the tags that was in the dependency tree. When we had kind of figured out what to do with the different components we worked on trying to write grammar rules but that became too complex which resulted in trying to find a new way to do the project. In the end we found out about the neural oracle which needed totally new data from the treebank together with an engine that could run the training simulation. We simply did not have time to complete this in the end.

```
#-*- coding:utf-8 -*-

# NLP Final Group Project : Dependency parser
# of Nivre for the Korean Language version
# Yonsei University
# Groupe 8

import matplotlib.pyplot as pyplot
import random as r
import numpy as np
from math import *

import os
import codecs as c
from io import StringIO, BytesIO
from konlpy.tag import Kkma
from konlpy.utils import pprint

# Library for trees
from anytree import Node, RenderTree
from anytree.exporter import DotExporter


def extract_body(data):
    body = ""
    save = False
    for i, char in enumerate(data):
        if char == '<':
            check = data[i:i+6]
            if check == "<body>":
                save = True
            if check == "</body":
                save = False
        if save and char != '\t':
            body = body  + char
    return body[6:]

def parse(body):
    # print(body)
    sentences = []
    sentence_add = False
    sentence = ""
    dictionary = []
    dict_add = False

    for i, char in enumerate(body):
        if char == '\r':
            if sentence_add:
                sentences.append(sentence)
                dictionary.append(dict)
                sentence = ""
                sentence_add = False
        if sentence_add:
            sentence = sentence + char
        if char == ';' and body[i+1] == ' ':
            sentence = ""
            sentence_add = True
    #clear all with 'Q'
    new = []
    for i, s in enumerate(sentences):
        # print(s)
        if 'Q' not in s:
            new.append(s)
    return new

def add_tags(sentences):
    sentence_dict = []
    kkma = Kkma()
    for i, sent in enumerate(sentences):
        sentence_dict.append((sent, kkma.pos(sent)))
    return sentence_dict



# Functions for arc-earger parsing algorithm : leftArc, rightArc, reduce, shift
def leftArc(S, I, A):
    A.append([I[len(I)-1], S.pop()]) # Add the arc (j,i) to A
    # S.pop() # Pop the stack
    return 0

def rightArc(S, I, A):
    A.append([S[len(S)-1], I[len(I)-1]]) # Add an arc from wi to wj from the token wi
    S.append(I[len(I)-1]) # Push wj onto S.
    return 0

def reduce(S):
    S.pop() # Pop the stack S
    return S

def shift(S, I):
    S.append(I.pop()) # Push the top token of the stack I on the stack S and pop the
    return S

'''
Input: A sentence
Output: The dependency parser of the sentence as a parsing tree
```

```
'''
def NivreParser(sentence):
    # Initialisation of the parser configuration
    n = len(sentence)
    S = [0] # Start with root on the stack
    I = [k for k in range(n)]
    A = [] # Arcs

    while (len(I) != 0):
        topStack = S[len(S)-1]
        topBuffer = I[len(I)-1]
        noAction = True

        hasParent = False
        for i in range(len(A)):
            if A[i][1] == topStack:
                hasParent = True
                break
        if hasParent == False:
            leftArc(S, I, A)
            noAction = False

        if noAction == True:
            hasParent = False
            for i in range(len(A)):
                if A[i][1] == topBuffer:
                    hasParent = True
                    break
            if hasParent == False:
                rightArc(S, I, A)
                noAction = False

        if hasParent == True:
            reduce(S)

        shift(S, I)

    return A

'''
Input: A the arcs in the tree and the sentence
Output: Generate a PNG file of the tree
'''
def printTree(A, sentence):
    print(sentence)
    tree = [Node("root")]
    for i in range(len(sentence)):
        tree.append(Node(sentence[i]))
    tree[1] = Node(sentence[A[0][0]], parent=tree[0])

    for i in range(len(A)):
        tree[A[i][1]+1] = Node(sentence[A[i][1]], parent = tree[A[i][0]+1])

    # print(tree)
    DotExporter(tree[0]).to_picture("tree.png")
    return 0

def main():
    '''
    Functions for importing corpus and extracting sentences+tags_dict
    dir = os.getcwd()
    raw = c.open(dir + "/dataset/BGEO0292.txt", "rb", "utf-16")
    data = raw.read() # to be able to manipulate input
    body = extract_body(data)    #extract sentences
    sentences = parse(body)      #extract tags
    sentence_dict = add_tags(sentences) #sentence_dict = [sentence, [(word, tag)]
    '''

    data = "Hi my name is Joakim Nivre."

    s1 = data.split()
    print(s1)

    # Nivre Parser
    parser = NivreParser(s1)
    parser = [[0, 1], [1, 2], [2, 3], [3, 4], [4, 5], [5, 6]]
    print(parser)

    printTree(parser, s1)

if __name__ == '__main__':
    main()
```

```
import tensorflow as tf
import numpy as np

EMBEDDING_SIZE = 10
LEARNING_RATE = 3
EPOCHS = 100

def create_dictionary(list_of_sentences):
    words = []
    for sentence in list_of_sentences:
        for word in sentence:
            if word not in words:
                words.append(word)

    int2word = {}
    word2int = {}
    for i, word in enumerate(words):
        word2int[word] = i
        int2word[i] = word

    return int2word, word2int

def to_one_hot(data_point_index, vocab_size):
    temp = np.zeros((vocab_size, 1))
    temp[data_point_index] = 1
    return temp



def main():
    #import data
    '''
    Could not import good data in time
    input_data = [current_state]
    current_state = [step, stack, buffer]
    Words in stack & buffer are represented as one_hot vectors where one position is
    unique [word, label] from the dataset.
    Output_data = the 4 the transition actions [left, right, reduce, shift]
    '''

    int2word, word2int = create_dictionary(sentences)
    batch = generate_batch(sentences)

    #placeholder
    x = tf.placeholder(tf.float32, shape=(None, dict_size))
    y_label = tf.placeholder(tf.float32, shape=(None, dict_size))
    #create model
    W1 = tf.Variable(tf.random_normal([dict_size, EMBEDDING_SIZE]))
    b1 = tf.Variable(tf.random_normal([EMBEDDING_SIZE])) #bias
    hidden_representation = tf.add(tf.matmul(x,W1), b1)

    W2 = tf.Variable(tf.random_normal([EMBEDDING_SIZE, dict_size]))
    b2 = tf.Variable(tf.random_normal([dict_size]))
    prediction = tf.nn.softmax(tf.add( tf.matmul(hidden_representation, W2), b2))

    sess = tf.Session()
    init = tf.global_variables_initializer()
    session.run(init)

    #Loss function
    loss = tf.reduce_mean(-tf.reduce_sum(y_label * tf.log(prediction), reduction_indi

    #Step function
    step = tf.train.GradientDescentOptimizer(LEARNING_RATE).minimize(loss)

    #train
    for _ in range(EPOCHS):
        session.run(step, feed_dict={x: x_train, y_label: y_train})
        print('loss␣is␣:␣', session.run(loss, feed_dict={x: x_train, y_label: y_train


    #save
    saver = tf.train.Saver()
    save_path = saver.save(sess, SAVE_PATH + "/tmp/oracle.ckpt")
    print("Model␣saved␣in␣path:␣%s" % save_path)

if __name__ == '__main__':
    main()
```