Reference Paper Title :

Sprucing up trees - Error Detection in Treebanks

Authors :

Ines Rehbien
Liebniz ScienceCampus
Heidelberg/Mannheim

Josef Ruppenhofer
Liebniz ScienceCampus
Heidelberg/Mannheim

Group : G3

Group Members:

Hridaya Annuncio (U101116FCS046)
Analysis of MACE-AL-Tree, POS Tagging, Dependency/Syntactic Parsing

Muskan Gupta (U101116FCS251)
Analysis of MACE (Dirk Hovy et al.), NER

Siddharth Bisht (U101116FCS187)

1. Introduction

"Structural syntactic information is an important ingredient for many NLP applications."

-Sprucing up trees - Error Detection in Treebanks, Ines Rehbein & Josef Ruppenhofer

Universal Dependencies (UD) is a project that is developing cross-linguistically consistent treebank annotation for many languages, with the goal of facilitating multilingual parser development, cross-lingual learning, and parsing research from a language typology perspective. In recent years the UD Project has become increasingly popular because of two reasons: It provides a unified framework for multilingual application, and Its annotation scheme encodes semantic information in a transparent way.

.....

PROBLEM STATEMENT

"The UD project provides treebanks for over 60 languages. A treebank is a parsed corpus. Thus, treebanking is a time consuming task. Many treebanks have been automatically converted from other frameworks and hence include some noise in addition to the noise that gets generated due to errors in human annotations. Due to low budget, manual error correction for entire tree cannot be done. Therefore methods that are able to detect errors in Automatically annotated dependency parse trees are of great value.

In their Paper, 'Sprucing up trees – Error Detection in Treebanks', Ines Rehbein and Josef Ruppenhofer suggest one such model, the MACE-AL-Tree for detecting errors in manually and automatically annotated dependency parse trees."

.....

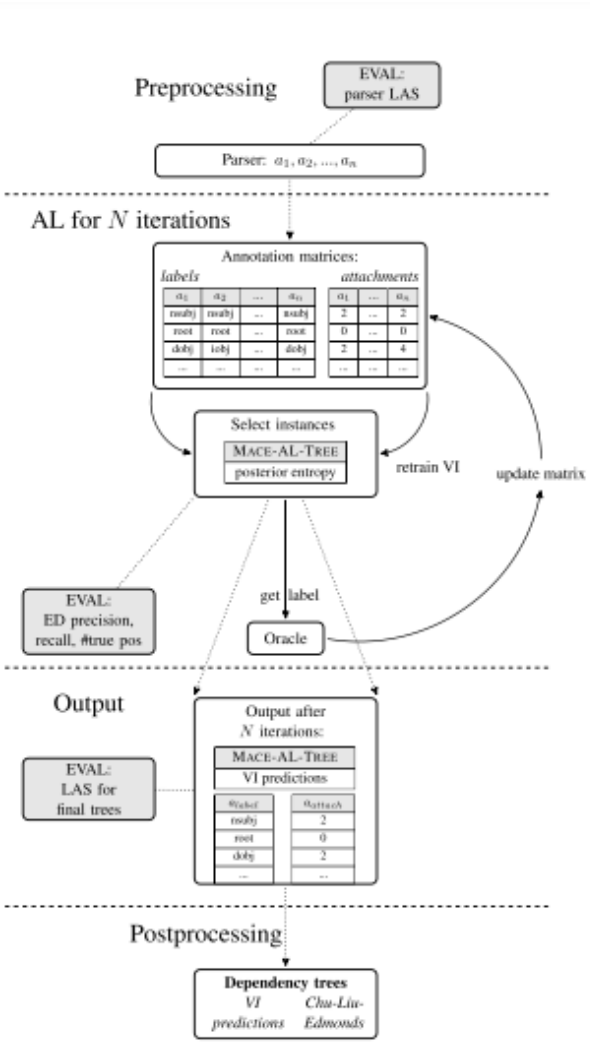
MACE-AL is a model wherein error detection of unstructured data is done. It is an extension of MACE. MACE (Multi-Annotator Competence Estimation) is highly successful when applied to annotations from crowdsourcing. But it cannot outperform the majority baseline when the predictions have been generated by a classifier committee. Thus to overcome this limitation, MACE-AL (AL: Active Learning) is used. This is used to combine variational inference with human feedback from AL. AL only selects instances for labelling that provide useful information to the classifier. This reduces manual effort and can yield the same accuracy as when training on a larger set of randomly selected data.

The problem with MACE-AL is is that it is not applicable to tasks with structured outputs such as trees and graphs i.e. It could not solve errors in the directed labelled relations between words of a sentence.

Hence, in this research paper an extension of MACE-AL was used, i,e, **MACE-AL-Tree**..
This paper discusses the detailed mechanism of this model and describes experiments performed using it.

Two main steps are involved when we using MACE-AL Tree :

Step 1: Two errors i.e. Labelling and attachment errors, have to be tracked so that the Variational Inference Models(VIM) are able to learn from the data. 2 models are used, one for processing the labels and one for processing the attachments. Labelling errors happen when wrong labels are assigned to an individual token.



Attachment errors happen when the relation between 2 nodes has been predicted incorrectly. VIM1 aims at learning the reliability of each annotator with regard to edge label prediction and VIM2 learns the reliability of attachment decisions of the annotators.

The input to this model is a set of two matrices. One with the predicted labels and the other of the predicted attachments. These predictions are done by the n annotators. These matrices are developed using the output trees of the n annotators.The extraction of these outputs is called Preprocessing.

These matrices are used for training. Consequently, each VM outputs the previous entropies used for its respective decisions about preprocessed input. Based on the posterior entropy, the VIMs will select the next instances to be annotated during active learning.

This means that using the posterior entropies, the predicted errored head will be indicated in blue, and the potential heads are highlighted in red. The annotator (human or extraction from the gold tree) enters the correct head and label for this token.This information is used to update noth the matrices by randomly selecting one of the annotators and replacing the predictors of that annotator for

the instance in question with new predictors. This does not necessarily result in an increase of accuracy because the prediction must already have been correct from the start. After updation, we start over and retrain the VIMs on the updated annotations. To save time we alternate the training of the two VIMs. But the updation of the matrices is done in each iteration.

Step 2: The output/predictions of the VM have to be taken and translated back into a tree structure. After error correction we need to output the trees. But VIMs make local decisions for individual labels and edges. Thus, they are very hard. Therefore, two methods can be used:

1) *Selection of the most trustworthy predictions by the VIM for the labels and edges, and combine those in the final tree.*

2)*Select the highest scoring well formed tree(the annotation trees) from a weight matrix where the weight is voted for by the parser/annotator committee.*

2. Objectives

The MACE-AL Tree uses the concept of Reinforcement Learning which is a difficult concept to learn. Hence, our group decided to implement other smaller parts of the model which were doable with the resources and time available to us.

- *POS tagging*
- *Dependency Parsing*
- *Understanding the Implementation of MACE*
- *CLE Algorithm*

3. Work Done

3.1 POS Tagging :

3.1.1 *About*

_____This concept takes care of the labelling part of the model.
For eg:

I ate the spaghetti with meatballs.
Pro V Det N Prep N

The coding was done in python using Spyder.

3.1.2 *Code*

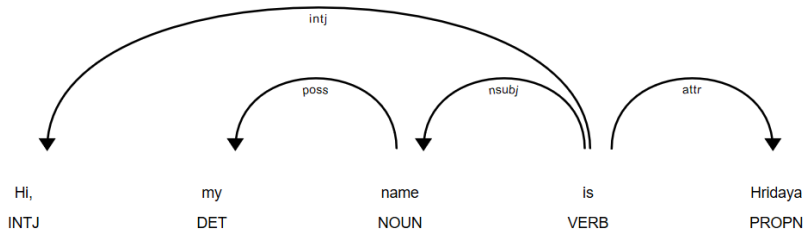
```
import nltk

nltk.download('punkt')

nltk.download('averaged_perceptron_tagger')

nltk.download('stopwords')

import nltk

from nltk.corpus import stopwords

from nltk.tokenize import word_tokenize, sent_tokenize

stop_words = set(stopwords.words('english'))


txt= "My name is Harry Potter. My best friends are Ron and Hermoine. We face all problems together. Hermione is extremely intelligent. Ron is really good at Quidditch."

sentence = sent_tokenize(txt)
for i in sentence:

        # Word tokenizers is used to find the words and punctuation in a string

        wordsList = nltk.word_tokenize(i)

        # removing stop words from wordList

        wordsList = [w for w in wordsList if not w in stop_words]

        # Using a Tagger. Which is part-of-speech tagger or POS-tagger.

        tagged = nltk.pos_tag(wordsList)

print(tagged)
```

3.1.3 *Output*

```
[('My', 'PRP$'), ('name', 'NN'), ('Harry', 'NNP'), ('Potter', 'NNP'), ('.', '.')]
[('My', 'PRP$'), ('best', 'JJS'), ('friends', 'NNS'), ('Ron', 'NNP'), ('Hermoine', 'NNP'), ('.', '.')]
[('We', 'PRP'), ('face', 'VBP'), ('problems', 'NNS'), ('together', 'RB'), ('.', '.')]
[('Hermione', 'NNP'), ('extremely', 'RB'), ('intelligent', 'JJ'), ('.', '.')]
[('Ron', 'NNP'), ('really', 'RB'), ('good', 'JJ'), ('Quidditch', 'NN'), ('.', '.')]
```

3.2 Dependency Parsing

3.2.1 *About*



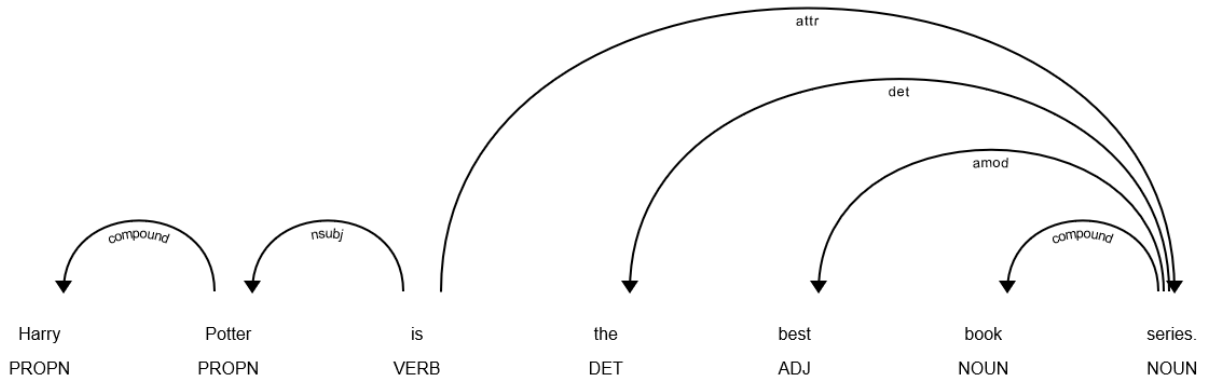
This concept takes care of the attachment part of the model. For eg:

The coding was done in Python using Jupyter Notebook..

3.2.2 *Code*

```
import spacy

from spacy import displacy

nlp = spacy.load("en_core_web_sm")

doc = nlp(u"Harry Potter is the best book series.")

displacy.render(doc, style="dep")
```

3.2.3 *Output*



3.3 Named Entity Recognition

3.3.1 *About: It is used to extract information out of a sentence and helps locating and* classify named entity mentions in unstructured text into pre-defined categories such as the person names, locations, time, expressions, quantities, monetary values, percentages, organizations etc.

3.3.2 *Code :*

```
import spacy
nlp = spacy.load("en_core_web_sm")
doc = nlp(u"Apple is looking at buying U.K. startup for $1 billion")
for ent in doc.ents:
print(ent.text, ent.start_char, ent.end_char, ent.label_)
```

3.3.3 *Output*

TEXT	START	END	LABEL	DESCRIPTION
Apple	0	5	ORG	Companies, agencies, institutions.
U.K.	27	31	GPE	Geopolitical entity, i.e. countries, cities, states.
$1 billion	44	54	MONEY	Monetary values, including unit.

3.4 MACE

3.4.1 *About*

MACE (Multi-Annotator Competence Estimation) is an implementation of an item-response model that lets you evaluate redundant annotations of categorical data. It provides competence estimates of the individual annotators and the most likely answer to each item.

If we have 10 annotators answer a question, and five answer with 'yes' and five with 'no', we would normally have to rely on a different method to decide what the right answer is. If we knew, however, that one of the people who answered 'yes' is an expert on the question, while one of the others just always selects 'no', we would take this information into account to weight their answers. MACE does exactly that. It tries to find out which annotators are more trustworthy and upweighs their answers.

In tests, MACE's trust estimates correlated highly wth the annotators' true competence, and it achieved accuracies of over 0.9 on several test sets. MACE can take annotated items into account, if they are available. This helps to guide the training and improves accuracy.

Hence MACE solves the following problems:

1. *Aggregate annotations to recover the most likely answer*
2. *Find out which annotators are trustworthy*
3. *Evaluate item and task difficulty*

3.4.2 *Algorithm*

N= *instances* , M = *annotator*, T_i = *True label*

For i =1....N:
T_i ~ *Uniform*
For j=1.....M:
S_{ij} ~ *Bernoulli*(1-θ_j)
If S_{ij}=0: (*No spamming*)
A_{ij}=T_i (*True label is copied to produce annotation* A_{ij})
else: (*Spamming*)
A_{ij} ~ *Multinomial*(ξ_j) (A_{ij} *is sampled from a multinomial with parameter vector* ξ_j)

3.5 Chu-Liu-Edmonds' Algorithm

3.5.1 *About*

Chu–Liu-Edmonds' algorithm is an algorithm for finding a spanning arborescence of maximum weight (sometimes called an optimum branching). It is the directed analog of the maximum spanning tree problem.

"A *branching in a directed graph is defined as a set of directed edges that contain no cycles and such that no two edges are directed towards the same vertex.*"

Given weights on the edges of a directed graph, we define a maximum branching as a branching such that the sum of the weights of its edges is maximal.

CLE Algorithm is extensively used in Graph-based dependency parsing for non-projective languages. Graph based methods score entire trees and hence are independent of length of dependencies. Here the vertices are the words of any sentence and the edges are th possible dependencies. We also define an extra vertex known as the root*.

*It should be noted that the root does not have any incoming edges.

3.5.2 *Algorithm*

Input: graph G = (V, E)

Output: a maximum spanning tree in G

Greedily select the incoming edge with highest weight

- *Tree*
- *Cycle in G*

Contract cycle into a single vertex and recalculate edge weights going into and out the cycle

Each recursive call takes $O(n^2)$ *to find highest incoming edge for each word.*

At most $O(n)$ *recursive calls(contracting n times)*

Total: $O(n^3)$

3.5.3 *Pseudocode*

function MAXSPANNINGTREE(G=(V,E), root, score) **returns** *spanning tree*

```
F←[ ]
T'←[ ]
score'←[ ]
```

for each $v \in V$ **do**

$bestInEdge \leftarrow argmax_{e=(u,v) \in E} score[e]$

$F \leftarrow F \cup bestInEdge$

for each $e = (u,v) \in E$ **do**

$score'[e] \leftarrow score[e] - score[bestInEdge]$

if T= (V,F) *is a spanning tree then return it*

else

$c \leftarrow$ *a cycle in* F

$G' \leftarrow CONTRACT(G,C)$

$T' \leftarrow MAXSPANNINGTREE(G', root, score')$

$T \leftarrow EXPAND(T',C)$

return T

function CONTRACT(G,C) **returns** *contracted graph*

function EXPAND(T,C) **returns** *expanded graph*

4. Experiments and Results

a. Here we have tested the Chu-Liu-Edmonds' Algorithm on an example sentence

python cle.py

Enter the sentence:

Book that flight.

Following are the indices for the words

root/head-->0

Book --> 1

that --> 2

flight --> 3

Enter the weighted matrix ***(Adjacency Matrix)***

0 12 4 4

0 0 5 7

0 6 0 8

0 5 7 0

Graph: (incoming edges)

0-->0 0.0 1-->0 12.0 2-->0 4.0 3-->0 4.0

0-->1 0.0 1-->1 0.0 2-->1 5.0 3-->1 7.0

0-->2 0.0 1-->2 6.0 2-->2 0.0 3-->2 8.0

0-->3 0.0 1-->3 5.0 2-->3 7.0 3-->3 0.0

Reverse Graph: (outgoing edges)

0-->1 12.0 2-->1 6.0 3-->1 5.0 0-->2 4.0

1-->2 5.0 3-->2 7.0 0-->3 4.0 1-->3 7.0

2-->3 8.0

Maximum Incoming edges for the three vertices:

0-->1 12.0 3-->2 7.0 2-->3 8.0

Identify the cycles

CONTRACT and EXPAND

Max Span Tree:

0-->1 12.0

1-->3 7.0

3-->1 7.0

5. Conclusion and Future work

MACE-AL Tree seems to be an efficient model given all that we have read in the paper about it. We have yet to understand how to use reinforcement learning and in the future would like to implement the complete model.