Evaluating Dependency Parsers

Andrew Petros Joseph Georgiou
King's College
apjg4@cam.ac.uk

1 Introduction

The task of dependency parsing is a critical task in Natural Language Processing, evaluating these systems requires manually annotated gold data. Not much work goes into actually analysing and evaluating the specific differences between parsing systems and human annotation. The standard techniques solely fall upon quantifiable measures of similarity such as the Labeled Attachment Score (LAS), in this report we will look at key differences between two parsers with the intention of identifying the differences in their functionality and implementation that is often ignored in these quantitative metrics. We will of the most popular parsers publicly available, MaltParser [1] and Stanford's Stanza NLP Pipeline Tools systems [2] which both utilise the updated universal dependencies v2 scheme [3] for their dependency parser outputs.

2 Background

2.1 Dependency Parsers

2.1.1 MaltParser

MaltParser uses data-driven dependency parsing, this is inspired from shift-reduce parsing in that we have a stack that stores processed nodes. One such issue of shift-reduce parsing is it quite often seen as a greedy search for the optimal dependency output. These greedy search systems are restricted to purely projective dependencies, they also suffer from poor performance on long distance dependencies. MaltParser took knowledge from Nivre et al. [4] which utilises graph transformation techniques to aleviate the projective dependency problem by processing both the input before training and processing the output through something they call pseudo-projective parsing. These transition scores are then learn in the case of our parsing through their Support Vector Machine classifier (LIBSVM)[5]. One such issue that still remains though is that it still remains a greedy parsing algorithm which may result in error propogation which is something we see happen quite commonly and discuss in our error evaluation section. Something to note as the MaltParser does not use a

full-pipeline like Stanford's Stanza we utilised the POS tagging MaltParser pipeline built into the NLTK Python library [6].

2.1.2 Stanford's Stanza NLP Pipeline Tools

Stanza, which we will refer to as Stanford's NLP Parser is ran through their Python natural language package "Stanza". This enables pipelining all the important steps of the process of dependency parsing, the main pipeline for dependency parsing uses tokenisation and sentence segmentation, multi-word token expansion, part-of-speech tagging and morphological features and lemmatisation. These we will cover in more details as we discuss the errors produced by the model as they all relate to the final outputted document. The original Stanford Parser produced in 2014 and found through CoreNLP [7] is located on the Stanford NLP Group page, this full pipeline parser utilised word embeddings with a single hidden layer that maps the inputs (word embeddings, POS tag features and arc label features) to the hidden layer using a cube activation function which in their original paper they state captures the interaction of the three input elements. It performs adequately but in testing performance was almost identical to MaltParser and therefore was not used in preference on something that could capture more interesting elements such as non-projectivity and greater long range dependence.
Stanza is their 2020 solution to a full neural network dependency parsing pipeline [8] which no longer uses transition-based parsing and instead uses a graph-based approach, thereby improving on work done by Kiperwasser et al. [9], replacing the traditional multi-layer perceptron attention mechanism and affine label classifier with biaffine ones as well as building the neural network much larger and using regularisation and multi-layer perceptrons to reduce dimensionality and increase efficiency. This made it state-of-the-art for graph-based parsing and very competitive against transition-based parsers such as MaltParser. They noted state-of-the-art performance on CoNLL 09 [10] dataset but it is worth noting that graph-based parsers significantly perform better than transition-based parsers at identification of non-projective dependencies and CoNLL 09's dataset contains a much larger number of non-projective dependencies then other years so the improvement over transition-based parsers is expected.

3 Evaluation

We wished to do an analysis of the efficiency of each model to see if the greedy transition-based parser's speed is worth it over the slower exhaustive approach to inference although since we trained MaltParser using it's java implementation against Stanza's Python wrapper, it proved that Stanza's testing time was significantly slower mainly due to Python being slow as it suffers from a lack of Just In Time optimiser unlike Java. Table 1 shows the Macro and Micro F_1 measure of each parser as a beginning analysis of each model in how it manages to predict

relational labels. The macro-average computes the average independently for each class, this treats all possible classes equally. The micro-average combines all the classes contributions before computing the average. micro-average is more useful for us to look at since we have quite a large class imbalance. For example *punct* appears 55 times in the gold standard labels although modifiers such as *nsubj:pass*, *aux:pass* and *obl:npmod* appear 3,3 and 1 times respectively therefore we do not wish to average precision and recall of these labels equally but handle for class size.

We used MaltEval to produce the analysis of our parser's below:
MaltParser (LAS): 60.7%
Stanford Parser (LAS): 70.9%

MaltParser Directional Precision / Recall:

Direction	Precision	Recall	Parser Count	Correct Count
left	92.04	77.61	113	104
right	85.48	95.21	186	159
to root	64.29	75.00	13	9

Stanford Parser Directional Precision / Recall:

Direction	Precision	Recall	Parser Count	Correct Count
left	98.08	76.12	104	102
right	83.76	98.80	197	165
to root	91.67	91.67	12	11

What we can see from the LAS' is that Stanford Parser performs better on our gold standard sentences in both head and dependency identification, this is probably since some of our gold sentences exceed 50 tokens meaning they have some long range dependencies, MaltParser can be more likely to propagate early missed dependencies down the rest of the sequence resulting in an overall LAS reduction.
The 2 tables above show the precision and recall as it relates to directionality, we can see the result of greedy left to right parsing of the MaltParser as it has high precision from arcs in the left direction (identifying a head to the right of the current buffer token) which will be more common as it attempts to empty the buffer as new tokens enter thereby over-predicting left-arc's. This is shown as precision is high but recall is low signifying it is over-predicting left-arc's. It also shows high recall in right-arc's since it is more common for a sentence to flow left-to-right it can assign that less often and achieve higher recall although it prefers left-arc meaning right precision is low. Finally the MaltParser performs very poorly on rooted arc's this is because on sentences with long distance dependencies the parser will often miss them early on and at the end have a large number of unassigned tokens where the largest greedy probability is towards the root since a large number of tokens are

often assigned the root. It then arc's all these remaining tokens in the buffer to the root resulting in poor precision and poor recall as it over assign's these tokens.
The Stanford parser performs worse than the MaltParser in the left-arc recall and right-arc precision, the explanation for these is that the parser actually over-predicts right-arc dependency reducing its ability to be accurate as it predicts 30 more right-arc dependencies then are in the gold standard and under-predicts the left-arc dependencies by 30 mostly likely since the Bi-LSTM still maintains some order of directionality in parsing left-to-right then right-to-left instead of true bi-directionality. It also utilises an affine transformation over a single LSTM output state instead of the recurrent cell promoted by the previous work by Cheng et al. This means that there still exists a sense of order to predicting dependencies that could result in over predicting previous states.

MaltParser				Stanford Parser			
Relation	Precision	Recall	F_1	Relation	Precision	Recall	F_1
nsubj	0.83	0.88	0.86	nsubj	0.78	0.74	0.76
root	0.75	0.64	0.69	root	0.92	0.92	0.92
nmod:poss	1.00	1.00	1.00	nmod:poss	1.00	1.00	1.00
obj	0.64	0.58	0.61	obj	1.00	1.00	1.00
det	0.96	1.00	0.98	det	0.89	0.86	0.88
obl:tmod	1.00	0.50	0.67	obl:tmod	1.00	0.50	0.67
cop	1.00	0.80	0.89	cop	0.75	0.60	0.67
advmod	0.84	0.80	0.82	advmod	0.84	0.80	0.82
acl:relcl	0.33	1.00	0.50	acl:relcl	0.83	0.71	0.77
punct	0.92	0.98	0.95	punct	0.94	0.96	0.95
amod	0.94	0.89	0.92	amod	0.89	0.94	0.91
compound:prt	1.00	1.00	1.00	compound:prt	0.67	1.00	0.80
case	0.89	0.96	0.92	case	0.89	0.96	0.92
compound	0.93	0.72	0.81	compound	0.93	0.81	0.87
obl	0.92	0.73	0.81	obl	1.00	0.92	0.96
nsubj:pass	1.00	0.75	0.86	nsubj:pass	0.67	0.67	0.67
aux:pass	1.00	1.00	1.00	aux:pass	0.67	0.67	0.67
cc	1.00	0.86	0.92	cc	1.00	1.00	1.00
conj	0.45	0.59	0.51	conj	0.64	0.82	0.72
nummod	1.00	1.00	1.00	nummod	1.00	1.00	1.00
nmod	0.59	0.91	0.71	nmod	0.59	0.91	0.71
mark	0.75	0.60	0.67	mark	1.00	0.57	0.73
aux	0.67	1.00	0.80	aux	0.33	0.50	0.40
ccomp	0.00	0.00	0.00	ccomp	1.00	1.00	1.00
appos	0.00	0.00	0.00	appos	0.17	1.00	0.29
expl	1.00	1.00	1.00	expl	1.00	1.00	1.00
parataxis	0.00	0.00	0.00	parataxis	0.33	0.20	0.25
advcl	0.33	0.20	0.25	advcl	1.00	1.00	1.00
obl:npmod	1.00	1.00	1.00	obl:npmod	1.00	1.00	1.00
xcomp	0.00	0.00	0.00	xcomp	1.00	1.00	1.00
macro-average	0.72	0.71	0.72	macro-average	0.82	0.84	0.83
micro-average	0.80	0.80	0.80	micro-average	0.84	0.84	0.84

Table 1: showing computed macro/micro average scores of relational labels for MaltParser and Stanford Parser respectively.

4 Error Analysis

For our error analysis we created a set of 10 gold standard conll-u formatted dependency outputs. We then compared each sentence output to a given gold standard dependency graph and identified some of the errors each parser created.

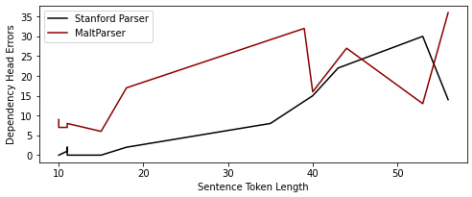

Figure 1: Combined errors of dependency heads as sentence length increases.

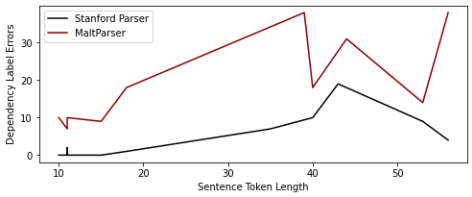

Figure 2: Combined errors of dependency labels as sentence length increases.

The aim of this section is not to just list each error but explain the error in relation to the theory behind the model and why it happened. The sentence outputs and the graph's will be discussed in this section although not all will be included for reducing verbosity.

4.1 Sentence Length

Sentence Length is a difficult problem for dependency parsers as longer sentences often cause them to suffer from lower accuracy, greedy inference algorithms such as MaltParsers transition-based system tend to succeed quite well in shorter sentences as the amount of decisions that are needed to be made are reduced but in longer sentences, inference errors will begin to form earlier in the sequence as close dependencies are greedily selected and therefore missed long dependencies will propogate errors further down the sentence. Figure 1 and Figure 2 show how the number of errors increase relative to the sentence length. Here we can see the difference between the two parsers as the Stanford Parser consistently produces less errors than MaltParser with much better performance in detecting adequate labels of dependencies but less so in detecting the head dependencies themselves. The reason the Stanford Parser does better than MaltParser is as explained before, the MaltParser with greedy arc selection will prefer short sequences over long dependencies where as Stanford's graph-based system evaluates the entire tree and does have the same tunnel vision of MaltParser in selecting an appropriate head for a given node. We of course still expect the number of errors to increase as the sentence length increases due introducing additional complexity and additional token selection probability although we can specifically notice a large difference in the final sequence where the graph-based approach actually reduces in errors from its previous sequence where as the transition-based parser has a significant increase in errors of both head identification and labelling dependencies.
Additionally of note for the last sequence the reason for the significant decrease in performance for the MaltParser is that late in the final sequence it consists of a large amount of punctuation tokens that had long range-dependencies, our analysis is that MaltParser used so many short range dependencies early that at the end of processing the sequence the buffer was left with a large majority of punctuation

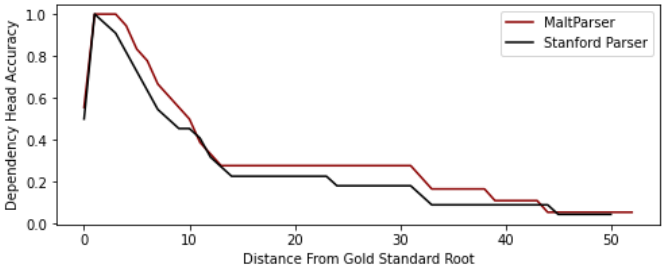

Figure 3: Accuracy of both parsers in relation to distance from gold standard root.

and numbers that had no obvious connection as their long range dependencies had already been connected, therefore the algorithm greedily connected them all to the root verb which is visible in Figure 4.

4.2 Dependency Graphs

When looking at the dependency graphs produced from the CoNLL outputs of each parser against our gold standards we can begin to manually identify common issues each parser produces. In Figure 3 we can see the accuracy of the parser in identifying head tokens as it gets further away from the gold standard root node. What we can identify from this is similar to what we identified through Figure 1 and Figure 2, that the greedy inference algorithm performs significantly better at shorter distances than long range dependencies. MaltParser suffers with longer range dependencies as with its algorithm when it reaches the end and it hasn't found dependency arcs it tends to just assign everything to the far left node in the buffer which is the root, this causes the problem we see in Figure 4 where the Gold Standard has 3 early nodes connected to the root node and MaltParser identifies the root correctly and connects the early nodes but then has many punctuation and left over nodes at the end of its greedy algorithm that are left to all be connected to the root instead of using an exhaustive search to identify a possible representation that can maximise the output score.

4.3 Language Properties

The language present in our test sentences can result in specific errors in our output, these errors often are caused by failured in part-of-speech tagging in our pipeline that can then be propagated by misidentifying a token early in the sentence that will then further propagate those errors to the rest of the sequence. Figure 5 shows the difference in tags that might explain some of the discrepancy in accuracy of the MaltParser vs Stanford Parser. We can see that Stanford Pipeline POS tags predict the NOUN tag less often than the MaltParser, but from the output files we can see that this is because it does a much better job at identifying proper noun's which improves its ability to identify object dependencies.

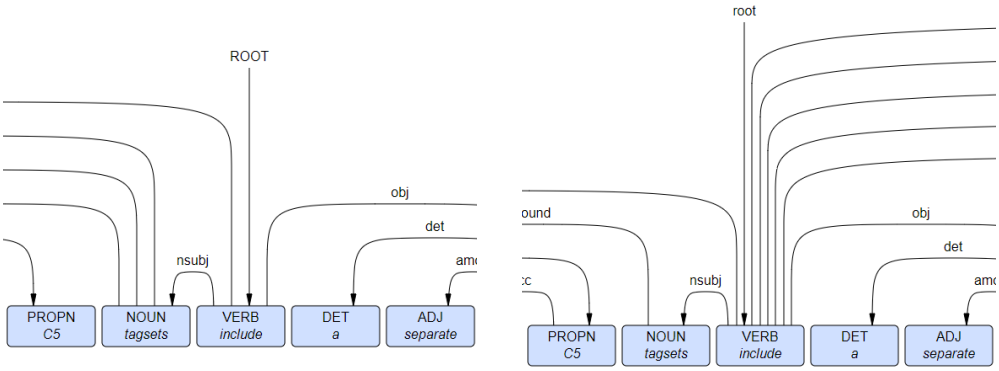

Figure 4: Dependencies Root Token Visualised.
Left: Gold Standard, **Right:** MaltParser.

In terms of identifying the language we can also see this reflected in table 1 as both models perform poorly on conjunctions, this is because we find conjunctions are often involved closer to the root and extend to long distances. The greedy approach of both of these models struggles with identifying not only the heads but the conjunctions themselves as a result. We can also see from table 1

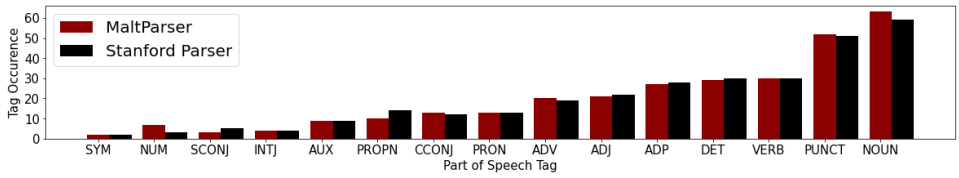

Figure 5: Representation of node occurrence for POS tags.

4.4 Projectivity

Dependencies from a CFG tree must be projective, this means that they will not have any crossing dependencies when we display our words linearly such as in figure 4. Although in theory non-projectivity is quite common and is required to account for displaced constituents. MaltParser states in their original paper that in theory they allow for non-projectivity through one of its two parsing algorithms. The first is Nirve's algorithm which is ran in either arc-eager or arc-standard mode and is therefore limited to projective dependency structures. The second algorithm available is Covington's algorithm which they use for unrestricted dependency structures and has a non-projective mode that utilises an additional context feature that stores unattached tokens occurring between the stack and the remaining tokens. Although through our research into projectivity utilising phrases such as "JetBlue cancelled our flight this morning which was already late.", "John saw a dog yesterday which was a Yorkshire Terrier." and "Who did Bill buy coffee from yesterday?"

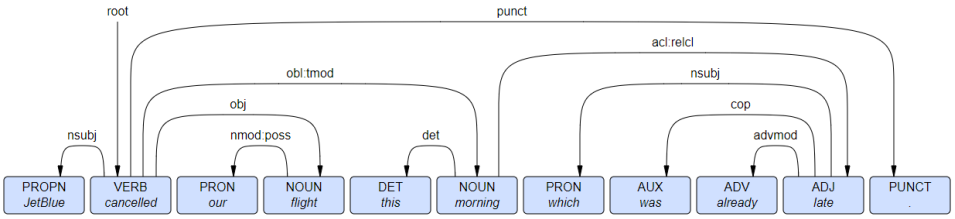

Figure 6: MaltParser incorrect Covington parsing of non-projective text.

we found it still never found the projective arc for any of these cases. In the first statement shown in Figure 6 it failed to find the relative clause *acl:relcl* dependency arc that exists between [late, flight] instead it applied it to [late, morning] which is clearly incorrect. Figure 7 shows that Stanford's parser accurately parses the non-projective sentence's correctly, this would not have been possible in the previous 2014 transition-based parser but in their new graph-based parser this is a much easier task as it looks at the sequence as a whole and therefore attempts to identify the maximising score for a given dependency tree. This allows the parser to explore the longer distance between the nodes [late, flight] which clearly is more coherent than the previous solution thereby increasing the output from its BiLSTM with deep biaffine attention for this output. We can also assume the fact it was trained on the CoNLL 2009 task which contains a large number of non-projective sequences it therefore can express these sequences easier.

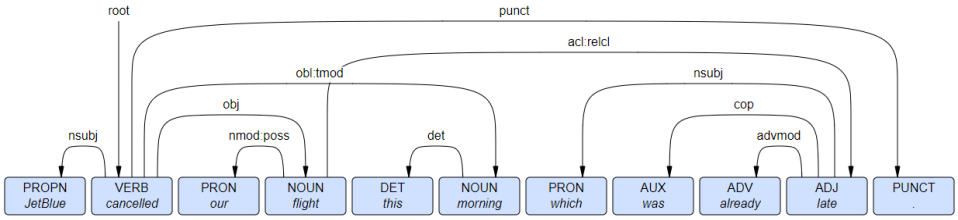

Figure 7: Stanford accurate parsing of non-projective text.

5 Conclusions

In this report we analysed two parser systems, MaltParser and Stanford's Stanza Parser. We outlined the implementation of each system as well as performing an in-depth analysis and evaluation of the parsers performance on our manually tagged gold standard set of sentences. What we discovered is that the problems we expected to face in our introductions of the model in relation to sentence length, visualising graphs, linguistic properties and projectivity were all correct. From this we wished to identify what could be changed in each model to improve its performance in relation to these specific errors. Stanza is a graph-based dependency parser therefore

it is exhaustive in its approach and also as a result of being graph-based has less syntactic features available during predictions, it would be beneficial to address this with some new architecture that can allow a graph-based approach but with the rich syntactic information of a transition-based parser. The MaltParser system suffers quite heavily from error propagation on long sentences with long distance dependencies although it has rich syntactic features available during predictions which means it makes great short-ranged head and dependency predictions. An ideal system would actually be a middle bound between these two systems, one with excellent near range parsing ability but not without calculating in long range dependencies as a factor to maximise the possible score while upgrading the MaltParser SVM to a deeper neural network component similar to Stanza's Deep Affine Attention Mechanism.

Word Count: 2,542
CoNLL Formatted Output Files: `https://github.com/A-Georgiou/CoNLL_Format_Outputs_L95.git`

References

[1] Rasoul Samad Zadeh Kaljahi, Jennifer Foster, Raphael Rubino, Johann Roturier, and Fred Hollowood. Parser accuracy in quality estimation of machine translation: A tree kernel approach. In *Proceedings of the Sixth International Joint Conference on Natural Language Processing*, pages 1092–1096, Nagoya, Japan, October 2013. Asian Federation of Natural Language Processing.

[2] Peng Qi, Yuhao Zhang, Yuhui Zhang, Jason Bolton, and Christopher D. Manning. Stanza: A Python natural language processing toolkit for many human languages. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics: System Demonstrations*, 2020.

[3] Marie-Catherine de Marneffe, Christopher D. Manning, Joakim Nivre, and Daniel Zeman. Universal Dependencies. *Computational Linguistics*, 47(2):255–308, 07 2021.

[4] Joakim Nivre and Jens Nilsson. Pseudo-projective dependency parsing. In *Proceedings of the 43rd Annual Meeting of the Association for Computational Linguistics (ACL'05)*, pages 99–106, Ann Arbor, Michigan, June 2005. Association for Computational Linguistics.

[5] Chih-Chung Chang and Chih-Jen Lin. Libsvm: A library for support vector machines. *ACM Trans. Intell. Syst. Technol.*, 2(3), may 2011.

[6] Steven Bird, Ewan Klein, and Edward Loper. *Natural language processing with Python: analyzing text with the natural language toolkit*. " O'Reilly Media, Inc.", 2009.

[7] Christopher Manning, Mihai Surdeanu, John Bauer, Jenny Finkel, Steven Bethard, and David McClosky. The Stanford CoreNLP natural language processing toolkit. In *Proceedings of 52nd Annual Meeting of the Association for Computational Linguistics: System Demonstrations*, pages 55–60, Baltimore, Maryland, June 2014. Association for Computational Linguistics.

[8] Timothy Dozat and Christopher D. Manning. Deep biaffine attention for neural dependency parsing. *CoRR*, abs/1611.01734, 2016.

[9] Eliyahu Kiperwasser and Yoav Goldberg. Simple and accurate dependency parsing using bidirectional LSTM feature representations. *Transactions of the Association for Computational Linguistics*, 4:313–327, 2016.

[10] Jan Hajič, Massimiliano Ciaramita, Richard Johansson, Daisuke Kawahara, Maria Antònia Martí, Lluís Màrquez, Adam Meyers, Joakim Nivre, Sebastian Padó, Jan Štěpánek, Pavel Straňák, Mihai Surdeanu, Nianwen Xue, and Yi Zhang. The CoNLL-2009 shared task: Syntactic and semantic dependencies in multiple languages. In *Proceedings of the Thirteenth Conference on Computational Natural Language Learning (CoNLL 2009): Shared Task*, pages 1–18, Boulder, Colorado, June 2009. Association for Computational Linguistics.

A Code Appendices

Appendix A: Processing Graph and Data code for MaltParser and Stanford Parser against Gold Standard.

```
errors_accuracy = [1.0-(x[1]/x[0]) for x in sentence_length_error]
errors_depen_accuracy = [1.0-(x[1]/x[0]) for x in sentence_depen_error]

errors_accuracy_malt = [1.0-(x[1]/x[0]) for x in
    sentence_length_error_malt]
errors_depen_accuracy_malt = [1.0-(x[1]/x[0]) for x in
    sentence_depen_error_malt]

fig = plt.figure()
fig.set_size_inches(8, 3)
plt.plot([x[0] for x in sentence_length_error], [x[1] for x in
    sentence_length_error], color='black', label='Stanford Parser')
plt.plot([x[0] for x in sentence_length_error_malt], [x[1] for x in
    sentence_length_error_malt], color='darkred', label='MaltParser')
plt.xlabel('Sentence Token Length')
plt.ylabel('Dependency Head Errors')
plt.legend()
fig.savefig('SentenceErrorGraphTokens.jpg')


fig = plt.figure()
fig.set_size_inches(8, 3)
plt.plot([x[0] for x in sentence_depen_error], [x[1] for x in
    sentence_depen_error], color='black', label='Stanford Parser')
plt.plot([x[0] for x in sentence_depen_error_malt], [x[1] for x in
    sentence_depen_error_malt], color='darkred', label='MaltParser')
plt.xlabel('Sentence Token Length')
plt.ylabel('Dependency Label Errors')
plt.legend()
fig.savefig('SentenceErrorGraphDepens.jpg')

fig = plt.figure()
fig.set_size_inches(8, 3)
plt.plot([x[0] for x in sentence_depen_error], errors_accuracy, color='
    black')
plt.xlabel('Sentence Token Length')
plt.ylabel('Dependency Head Accuracy')

fig.savefig('SentenceErrorGraphDepens.jpg')

fig = plt.figure()
fig.set_size_inches(8, 3)
plt.plot([x[0] for x in sentence_depen_error], errors_depen_accuracy,
    color='black')
plt.xlabel('Sentence Token Length')
plt.ylabel('Dependency Label Accuracy')
fig.savefig('SentenceErrorGraphDepens.jpg')

fig = plt.figure()
fig.set_size_inches(8, 3)
plt.plot([x[0] for x in sentence_depen_error_malt], errors_accuracy_malt,
    color='darkred')
plt.xlabel('Sentence Token Length')
plt.ylabel('Dependency Head Accuracy')
```

```
fig.savefig('SentenceErrorGraphDepens.jpg')

fig = plt.figure()
fig.set_size_inches(8, 3)
plt.plot([x[0] for x in sentence_depen_error_malt],
    errors_depen_accuracy_malt, color='darkred')
plt.xlabel('Sentence Token Length')
plt.ylabel('Dependency Label Accuracy')
fig.savefig('SentenceErrorGraphDepens.jpg')


X = ['SYM','NUM','SCONJ','INTJ', 'AUX', 'PROPN', 'CCONJ', 'PRON', 'ADV', '
    ADJ', 'ADP', 'DET', 'VERB', 'PUNCT', 'NOUN']

y_malt = [deprel_dict_malt[y.lower()] for y in X]
y_stan = [deprel_dict_stan[y.lower()] for y in X]

X_axis = np.arange(len(X))

plt.figure(figsize=(20, 3))  # width:20, height:3
plt.bar(X_axis - 0.2, y_malt, width=0.4, label = 'MaltParser', align='edge
    ', color='darkred')
plt.bar(X_axis + 0.2, y_stan, width=0.4, label = 'Stanford Parser', align=
    'edge', color='black')
plt.xticks(X_axis, X, fontsize=15)
plt.yticks(fontsize=15)
plt.xlabel("Part of Speech Tag", fontsize=15)
plt.ylabel("Tag Occurence", fontsize=15)
plt.legend(fontsize=20)
plt.show()


precision_malt, recall_malt, f1_malt = [],[],[]
precision_stan, recall_stan, f1_stan = [],[],[]
tags = []

for key, val in dep_gold_dict.items():

    tp_malt = correct_dep_malt[key]
    tp_stan = correct_dep_stan[key]

    actual_results = val
    pred_malt = error_dep_malt[key] + tp_malt
    pred_stan = error_dep_stan[key] + tp_stan

    prec_malt = tp_malt/actual_results
    prec_stan = tp_stan/actual_results

    rec_malt = 0
    rec_stan = 0
    f1_malt_s = 0
    f1_stan_s = 0

    if(pred_malt != 0):
        rec_malt = tp_malt/pred_malt

    if(prec_malt != 0 or rec_malt != 0):
        f1_malt_s = (2*(prec_malt*rec_malt))/(prec_malt+rec_malt)

    if(pred_stan != 0):
        rec_stan = tp_stan/pred_stan
```

```
    if(prec_stan != 0 or rec_stan != 0):
        f1_stan_s = (2*(prec_stan*rec_stan))/(prec_stan+rec_stan)

    print(key, "&", "%.2f" % prec_malt, "&", "%.2f" % rec_malt, "&", "%.2f
        " % f1_malt_s, "\\\\")
    #print(key, "&", "%.2f" % prec_stan, "&", "%.2f" % rec_stan, "&", "%.2
        f" % f1_stan_s, "\\\\")

    tags.append(key)
    precision_malt.append(prec_malt)
    precision_stan.append(prec_stan)
    recall_malt.append(rec_malt)
    recall_stan.append(rec_stan)

    f1_malt.append(f1_malt_s)
    f1_stan.append(f1_stan_s)

malt_tp = 0
overall_malt = 0

stan_tp = 0
overall_stan = 0

overall_gold = 0

for key, val in dep_gold_dict.items():
    overall_gold += val

for key, val in correct_dep_malt.items():
    malt_tp += val
    overall_malt += val

for key, val in correct_dep_stan.items():
    stan_tp += val
    overall_stan += val

for key, val in error_dep_malt.items():
    overall_malt += val

for key, val in error_dep_stan.items():
    overall_stan += val

malt_micro_average_prec = malt_tp/overall_malt
stan_micro_average_prec = stan_tp/overall_stan

malt_micro_average_rec = malt_tp/overall_gold
stan_micro_average_rec = stan_tp/overall_gold

malt_macro_average_prec = sum(precision_malt)/len(precision_malt)
stan_macro_average_prec = sum(precision_stan)/len(precision_stan)

malt_macro_average_rec = sum(recall_malt)/len(recall_malt)
stan_macro_average_rec = sum(recall_stan)/len(recall_stan)

malt_micro_f1 = (2*(malt_micro_average_prec * malt_micro_average_rec)/(
    malt_micro_average_prec + malt_micro_average_rec))
stan_micro_f1 = (2*(stan_micro_average_prec * stan_micro_average_rec)/(
    stan_micro_average_prec + stan_micro_average_rec))

malt_macro_f1 = (2*(malt_macro_average_prec * malt_macro_average_rec)/(
    malt_macro_average_prec + malt_macro_average_rec))
stan_macro_f1 = (2*(stan_macro_average_prec * stan_macro_average_rec)/(
```

```
    stan_macro_average_prec + stan_macro_average_rec))


distance_sum_malt = defaultdict(lambda: 0)
distance_sum_stan = defaultdict(lambda: 0)

for key, item in distance_root_correct_malt.items():
    distance_sum_malt[key] += item

for key, item in distance_root_errors_malt.items():
    distance_sum_malt[key] += item

for key, item in distance_root_correct_stan.items():
    distance_sum_stan[key] += item

for key, item in distance_root_errors_stan.items():
    distance_sum_stan[key] += item

distance_malt = sorted(list(distance_sum_malt.items()), key=lambda x: x
    [0])
distance_stan = sorted(list(distance_sum_stan.items()), key=lambda x: x
    [0])

accuracy_malt = []
accuracy_stan = []
for dist, total in distance_malt:
    acc = distance_root_correct_malt[dist]/total
    accuracy_malt.append([dist, acc])

for dist, total in distance_stan:
    acc = distance_root_correct_stan[dist]/total
    accuracy_stan.append([dist,acc])

fig = plt.figure()
fig.set_size_inches(8, 3)

max_y_malt = max(distance_malt, key=lambda x: x[1])[1]
max_y_stan = max(distance_stan, key=lambda x: x[1])[1]

plt.plot([x[0] for x in accuracy_malt], [x[1] for x in accuracy_malt],
    color='darkred', label='MaltParser')
plt.plot([x[0] for x in accuracy_stan], [x[1] for x in accuracy_stan],
    color='black', label='Stanford Parser')
plt.xlabel('Distance From Gold Standard Root')
plt.ylabel('Dependency Head Accuracy')
plt.legend()
fig.savefig('SentenceErrorGraphDepens.jpg')
```