MICROBLOG TEXT PARSING: A COMPARISON OF STATE-OF-THE-ART PARSERS

by

Syed Muhammad Faisal Abbas

Submitted in partial fulfillment of the requirements
for the degree of Master of Computer Science

at

Dalhousie University
Halifax, Nova Scotia
July 2015

Table of Contents

List of Tables

List of Figures

Abstract

Parsing is a natural language processing task in which relationships between words are deduced. It is essential for higher levels of semantic analysis, especially when predicates are required to be extracted from the text.

Parsing is a widely established task and much effort has been put into devising good methods for it, which has resulted in reasonably accurate processing of this task. However, most of the work has been limited to formally written text such as news articles or discussion groups. Microblog text is a significant body of text that is written by laypeople in quite an informal language which is significantly different from formal written language so as to require special considerations. There are many applications in the area of analysis of microblog text that require high-quality and fast parsing, such as identification of user intentions.

Dealing with large amount microblog text, we need to consider the running-time performance of the methods for many reasons: the amount of microblog text is huge and the pace new text is being generated is insurmountable, as well as the life span of its significance is very short.

In this thesis we evaluated various parsers and their parsing performance as it relates to microblog text: we evaluated eight (8) state of the art parsers, five (5) of these parsers are inherently constituency (Phrase-Structure) parsers, while three (3) of them are dependency parsers. We compared all of the parsers after converting the output of constituency parsers to dependency trees and evaluated the performances using Unlabelled Attachment Score (UAS). In addition we compared the constituency parsers using PARSEVAL and FREVAL measures. Finally, we evaluated the selected parsers for their running-time performance as well.

Acknowledgements

Alhamdolillah, with the grace of God, I have been able to get through the phase of getting this work done.

I am thankful to my supervisor Vlado Keselj without whose continual support throughout this research I could not have completed this study.

I would like to thank my family: my wife Raazia and our boys Ahmad and Ahsan. Their understanding of my engagement was more than what can be expected. Their support was a vital part of my getting to the completion of this work.

I would also like to thank Marie-Michelle Echanique, whose keeping me on focus, specially at the end of the work, prevented distractions and greatly helped me.

I am grateful to my late parents who made me what I am; may their souls be blessed.

And lastly, I thank God for giving me the guidance and strength to overcome the difficulties that come across my way, and making the world in my path just right to my abilities.

Chapter 1

Introduction

1.1 Purpose

The purpose of this thesis is to do a comprehensive study on comparison of available state of the art parsing methods and tools and how they perform on the English-language microblog text. We discuss and evaluate various state-of-the-art methods and tools employed in parsing of microblog text. We present an analysis on the performance of the methods and tools and a discussion on what are the differences between those performances on regular text and that on microblog text.

1.2 Motivation

Words in a sentence do not stand alone; words define, elaborate, explain, and qualify other words within a sentence (and sometimes beyond the sentence boundaries).

Even though classification of text using bag of words (word features) has been a successful approach for many natural language processes, many higher level problems require deeper understanding of the relationships between the words. In applications where predicates need to be extracted from text or user intentions are needed to be inferred, the structure of the sentences become quite important.

For example, as a heuristic rule, the template

"<noun> such as <noun>"

can be used to extract IS-A relationship between two nouns. "...animals such as cats ..." can be used to extract relationship that "cats are animals" or cat IS-A animal. However if we encounter a little more complex phrases, for example "...animals other than cats such as dogs . . . ”, this rule fails. To infer correctly “dogs are animal” from this phrase, the structure of the phrase must be deduced.

Similarly, when classification of a text to a topic or sentiment is not sufficient for an application, for example where intent of the user is also needed, structure of the text becomes more important. When someone writes something about ‘car’, it makes more sense to an automobile seller to know whether that person is interested in buying a car more than if she is just interested in cars in general.

To solve problems like these, parsing comes to rescue. Parsing is an important stage towards processing of meaning of any language by computational mechanisms. It is the step where the syntactic structure of the sentences is captured and it is determined how the words within a sentences are related to each other. Many times such as in our example above, there are distant relationships between words.

Microblogging is a recent phenomenon that has emerged with the technological advances the 21st century has brought us. The prime example of microblogging is Twitter. Microblogging provides easy access to vast amount of textual information, specially, about thought processes and behaviour patterns of lay-people (specialists have been being heard through other means of communications). The importance of utilizing microblogging information to gain insights about people for various applications is getting more and more evident as new applications are being envisioned.

Microblogging text is being used to analyse markets for businesses. It is employed as a better communication channel for collecting buyer/user feedback to businesses. Also in political domain, understanding of political thought of the masses is getting accessible through the use of microblogging communication channel.

Also, aggregating of knowledge from numerous sources is also being utilized, to gain insights, than just asking experts about certain subject matter. It can be argued that aggregates may provide better accuracy than expert knowledge. Moreover, acquiring knowledge from numerous inexpensive sources as opposed to getting it from few experts is making more and more economic sense. Access to Microblog allows us to aggregate knowledge from vastly many and diverse sources.

The parsing technology has been sufficiently developed in the recent past that the problem of parsing was assumed to be reasonably solved. However, almost all of the work done has been focused on formally written work such as news articles. We can see that the language features of the text used by lay-people in microblogging differs considerably from the formally written text of news articles. Thus there is a standing problem in parsing microblog text; the functional performance of the parsers, which have been quite good in parsing formal text, is significantly degraded when it comes to parsing microblog text.

Moreover, the amount of parsing required to parse microblog text, its total size being huge, the running-time performance, in addition to functional performance, is becoming more and more important. In order to process huge amount of microblog text, it is imperative to employ tools and methods that can process the text in minimal amount of time.

In this thesis, we have tried to evaluate various extant methods of parsing for their functional and running-time performance of parsing microblog text. We have come up with recommendations as to which methods and tools to use when parsing microblog text is required. We have analysed the performance differences, and have tried to understand the ways the performance could be improved.

Most of the evaluations of parsers are done by authors who either developed a parser or provided some contributions to improve the performance of a parser. We are not aware of a detailed independent study on evaluation and comparison of parsers performance on microblog data.

1.3 Research Objectives

The research objectives for this thesis are to evaluate various state-of-the-art parsers and compare their functional and running-time performance in parsing microblog text. Impartial and comprehensive evaluation is the objective.

An important part of the objective is an impartial execution of the experiments and analysis of the results so to perform an unbiased inquiry for evaluation of parsers for microblog text. The importance of running-time performance of the parsing technique cannot be over-emphasized for microblog data, and evaluation of the same is also a target.

1.4 Contribution

This study has covered considerably expansive comparison of state of the art parsers on microblog text. Various parsers have been evaluated for their functional performance while keeping a keen focus on running-time performance of the same. We have suggested which parsing techniques to follow to keep the performance under practical feasibilities.

We have evaluated the parsing tools and methods specifically on the microblog text language, which is significantly different from the formally and semi-formally written language of news articles or newsgroup discussions on which most of the existing research is based.

We have collected data points to facilitate decisions for the choice of parsers for microblog parsing as well as we have provided some directions for future research.

1.5 Thesis Overview

In the following chapter, we provide a background for this thesis highlighting some of the language features specific to microblog text. We also present an overview of the existing research done on parsers on microblog text as well as parsers in general.

Before going into our methodology and evaluations, we present a short overview of various natural language parsers. In this chapter we list the parsers we have selected and provide a discussion of their working principles.

Following that, we describe our methodology, detailing the datasets employed in our research as well as the parsers selected for evaluation. We present a brief account of the methods employed by the parsers. We also describe our evaluation methods in detail.

Next, we present evaluation and detailed discussion of our experiments. We provide dependency tree performance comparison of all of the parsers, and constituency tree performance comparison of constituency parsers. We also provide running-time performance of the parsers on microblog text.

Finally, we conclude our thesis with our analysis and recommendations for the parsing methods to be employed for microblog text parsing as well as some future research directions that we envision to be beneficial.

Chapter 2

Background and Related Work

2.1 Twitter Language

The concept of microblogging stared with Simple Message Service (SMS) that was conceived in 1984 when the technical limitation kept the size of the text to 160 characters [30]. Twitter added further restrictions to limit the text size to 140 characters and reserved 20 characters for user-name in order to keep the compatibility with SMS and to avoid tweets to span multiple SMS messages.

A large scale study [31] has been done on language issues of microblog on Twitter. The main idea of this study was to find the various language distributions and cross-language differences in Twitter. Additionally, it attempts to identify the behaviour of Twitter users of different languages. It highlights some features of the form of language used in Twitter.

It was found that 21% of all and 25% of English language tweets contains URLs or links in their tweets. This single feature is significant differentiator of microblog text from the regular text. In addition, there are Twitter specific features of the language used; e.g. there are hash-tags, at mentions, replies and re-tweets. In English language tweets 14% of the tweets contains hash-tags, 47% at-mentions, 29% were replies and 13% identified as re-tweets.

Another study [34] was done to evaluate if language features of English language tweets can be classified with various meta-data that is available about the tweets.

For example, the form of the language has a strong relation to the location of the user. Also the language of people with more than 1000 followers differs significantly from that of those having less than 1000 followers. This study also shows the language variability between tweets of users with different meta-data features.

Various features of Micro Text Twitter is the prime example of microblogging service. The text in twitter has many specific features that not present in other written text. Some of the features of twitter are listed below:

- Hash-tags: Hash-tags are free-form tags or keywords included in tweets, in the form of #keyword [31].
- At-mentions: At-mentions are when a Twitter user refers to a specific user by including a mention anywhere in their tweets, done in the form of @username [31].
- Replies: A reply, a specific form of mention with @username appearing at the beginning of the tweet is a tweet responding to a previous messages [31].
- Re-tweet: Re-tweeting is typically used to spread a tweet received from followees to followers [8]. There is no specific syntax for re-tweet A common form of re-tweeting is “RT @username”. People have also used various de facto conventions to signify re-tweet such as “RT:@”, “re-tweeting @”, “re-tweet @”, “(via @)”, “RT (via @)”, “thx @”, “HT @”, and “r @” [8].

2.2 Related Work on Comparison of Parsers

Since research on parsing methods was very actively looked upon where the parser performance improvement was peaked and plateaued before the advent of microblogging, there has not been extensive work on parsing of microblog text. The problem of parsing microblog text is significantly different from parsing formally written text due to language feature differences. It is worthwhile to look into evaluating parser performance for microblog text.

We discuss some related work in this section. The following discussion employs the use of two measures commonly used for evaluation of parser performance: Labelled Attachment Score (LAS) and Unlabelled Attachment Score (UAS). Labelled Attachment Score (LAS) is a the percentage of correctly identified dependencies between words along with the labels. Whereas, Unlabelled Attachment Score (UAS) is the percentage of correctly identified dependencies between words ignoring the labels. A more precise definition of these is provided in section 4.4.1.

2.2.1 Independent Comparison of Parsers on Microblog Text

Since microblog text processing has relatively recently gained interest in the NLP research community, we don't find expansive work of independent comparisons of parsing of microblog text in the literature.

A study by Derczynski et al. [20] did some effort in evaluating performance on high level tasks specific to microblog genre, pointing out its noise. It covered a comparison of part of speech (POS) tagging and named entity recognition (NER) tasks on microblog.

There has been a study on sentiment analysis on microblog text [24], but it doest not look into the lower level task of parsing.

Another study [22] showed the specific difficulty in parsing microblog text. It was found that POS tagging is a difficult task when it comes to microblog text. The study only uses a single parser, MALT parser, and shows the functional performance of the parser on microblog text. The analysis did not evaluate any other available parsers. It was also reported that a big difference of performance was due to the difference in POS tagging. The LAS on MALT WSJ vanilla was reported to be between 67% and 71% depending upon the number of training data. MALT up training was performed by training it on trees produced by Berkeley parser. They used training resources on Twitter as well as a sports discussion forum. They have reported that a statistically significant improvement in Labelled attachment score (LAS) of 4.67% and in unlabelled attachment score (UAS) of 3.74% was obtained by up training MALT.

The major shortcoming of this study is that it only evaluated one parser. To give a broad understanding of the domain, we need most major parsers to be evaluated to obtain a clearer picture. Moreover, the running-time performance aspect was not touched at all.

2.2.2 Comparison of Parsers on Microblog by Parser Writers

We see that that there is a parser (Tweebo parser) designed specifically for English microblog text [36]. Their work does provide a performance metric of this parser, but it does not compare other parsers on the microblog text. It was claimed that Tweebo parser is the first syntactic dependency parser designed explicitly for English tweets.

They have claimed to achieve 80.7% unlabelled attachment score (UAS) on their test set and 76.1% on Web2.0 TwitterTest dataset (which they called TEST-FOSTER).

Tweebo parser starts with TurboParser, an open-source parser, which uses MIRA [19] to train weights. We included Tweebo parser in our comparison for an independent evaluation.

2.2.3 Comparison of Parsers in General

There have been independent studies in comparing various parsers but the research in this area has experienced significantly reduced activity, seeing the interest of the researchers dwindle when the performance of the parsers plateaued after improving considerably. Much research took the direction of parsing languages other than English and on multilingual parsing [9].

Another study [25] was done on Collins parser [15, 14] with varying training corpora, but again it was not done for comparing various parsers.

2.2.4 Comparison on Running-time Performance of Parsers

Interestingly, we did not see any major work on comparison of parsers in terms of their running-time performances.

2.2.5 Summary

Our limited search for related literature did not satisfy a need for an expansive independent parser evaluation and comparison specifically for microblog text. More importantly, hardly any evaluation did take running-time performance of the parsers into consideration.

Chapter 3

Natural Language Parsers

Natural Language Parsing is a task in which sentence structures and the relationships between words are deduced.

There are two main types of natural language parsers: constituency parsers and dependency parsers. Each of these targeting two different types of parse trees: constituency trees (also known as phrase structure trees) and dependency trees.

3.1 Dependency Parsers

Dependency parsers is a class of parsers that create word dependency trees: identifying head (also known as root) word and dependency relations between words in a sentence. An example of this is illustrated in Fig. 3.1.

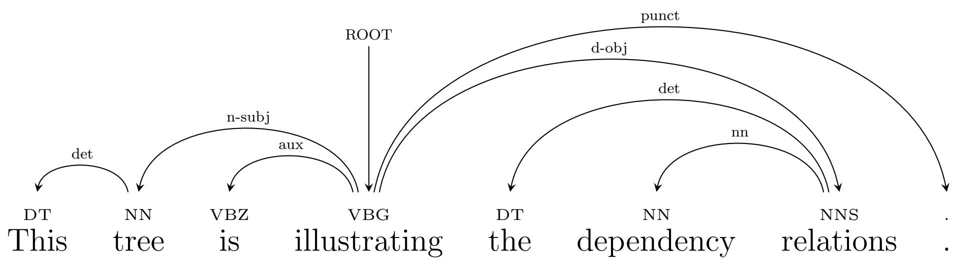

Figure 3.1: Labelled dependency tree of a sentence

Following is a brief introduction of the dependency parsers we selected for our evaluations.

3.1.1 MST Parser

MST parser [39] is a two state multilingual dependency parser. This parser is designed with two components: an unlabelled parser and an edge labeller. The edge labeller is applied after the unlabelled parsers is employed to parse the text to label the edges with dependency labels.

It can define a wide range of functions about parsing decisions. A model was added to integrate morphological features derived from each token [39, p. 217].

Once the tree is established, in the second stage, labels are assigned to the edges. The labelling is done with a pair of edges by using first order Markov factorization [39, p. 217]

$$l = \arg\max_{l} \sum_{m=2}^{M} s(l_{(i,j_m)}, l_{(i,j_{m-1})}, i, \mathbf{y}, \mathbf{x})$$

in which each factor is the score of labelling the adjacent edges (i, j_m) and (i, j_{m-1}), in the tree y for the sentence x. The score function is a dot product of feature representation and weight vector [39, p. 217]

$$s(l_{(i,j_m)}, l_{(i,j_{m-1})}, i, \mathbf{y}, \mathbf{x}) = \mathbf{w} \cdot \mathbf{f}(l_{(i,j_m)}, l_{(i,j_{m-1})}, i, \mathbf{y}, \mathbf{x})$$

Given a feature representation the highest scoring label sequence is obtained using Viterbi's algorithm [39, p. 217]. MIRA [19] online learner is used to set the weights [39, p. 217].

3.1.2 MALT Parser

MALT parser [41] is also a data-driven parser-generator. It generates a parser using a treebank. It is essentially an implementation of inductive dependency parsing in which the derivation of a dependency tree is obtained by the syntactic analysis of the sentence to be parsed by using inductive machine learning.

MALT parser is based a dependency parser that uses deterministic approach. The idea is based on shift-reduce parsing for context-free grammar. Deterministic parsing is better suited for disambiguation. It also uses history based models for predicting the parser actions at non-deterministic choice points. For training, a classifier is trained to predict a parsing action at a particular parsing configuration using the parse history and the input string. Parsing is done by the classifier making a dependency tree deterministically [41, para. 3].

MALT parser can also be turned into a constituency parser that finds the phrase structure trees for a sentence [28].

3.1.3 Tweebo Parser

Tweebo parser [36] is a parser designed specifically to parse Twitter Text. It is trained on Twitter text corpus. It uses Ark Tweet POS Tagger to tag words that uses specialized tag set 'Ark Tweet Tag set', which is provided as an appendix A.3.

Tweebo parser does not ignores annotation tokens, rather it interprets Twitter annotation tokens with syntactic functions [36].

Tweebo parser also tokenizes multi-word expressions. One aspect of Tweebo parser is especially notable; that it considers the possibly of tweets to have multiple sentences and thus marks multiple roots in a single tweet [36]. Tweebo also ignores punctuation tokens. For example the parse under Tweebo for the tweet "OMG I love the Biebs and want to have this babies ! —> LA Times: Teen Pop star Heartthrob is All the Rage on Social Media ... #belieber" is shows in figure 3.2.

Tweebo parser [36] is a parser designed specifically to parse Twitter Text. It is trained on Twitter text corpus. It uses Ark Tweet POS Tagger to tag words, which uses specialized tag set 'Ark Tweet Tag set' as listed in appendix A.3.

3.2 Constituency Parsers

Constituency parsers is a class of parsers that parse sentences with constituency grammars, also known as phrase structure grammars.

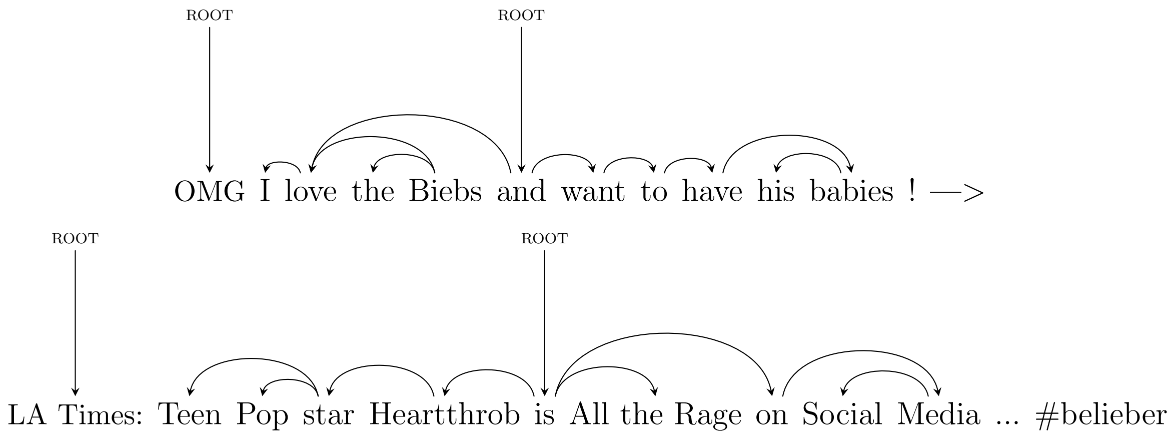
ROOT
ROOT
OMG I love the Biebs and want to have his babies ! —>
ROOT
ROOT
LA Times: Teen Pop star Heartthrob is All the Rage on Social Media ... #belieber

Figure 3.2: Dependency tree of an example tweet under Tweebo parser

The term 'Phrase Structure Grammar' was originally introduced by Noam Chomsky in 1957 [42] as defined by phrase structure rules. In these grammars, a sentence is composed of structure of phrases and terms. Terms are individual words or part of speech tags of those words, while phrases are a sequence of part of speech that have a specific syntactic and semantic value.

For example, a very common rule defined in English is that a sentence is composed of a noun phrase followed by a verb phrase. Noun phrase is a sequence of words, that define the subject, while the verb phrase contains the verb and any object.

A phrase structure grammar is defined by a set of terminals, a set of non-terminals, production rules, and a special non-terminal denoting the sentence.

A sentence "This tree is illustrating the constituency relation" can be parsed as Figure 3.3 where the grammar (for this sentence) is defined as

S ::= NP VP

NP ::= ADJ N

NP ::= DT NP

VP ::= V VP

VP ::= V NP

DT ::= *the*

N ::= *tree* | *relation*

V ::= *is* | *illustrating*

ADJ ::= *constituency*

Constituency parsers work with constituency grammars to find the structure of the parse tree according to the grammar.

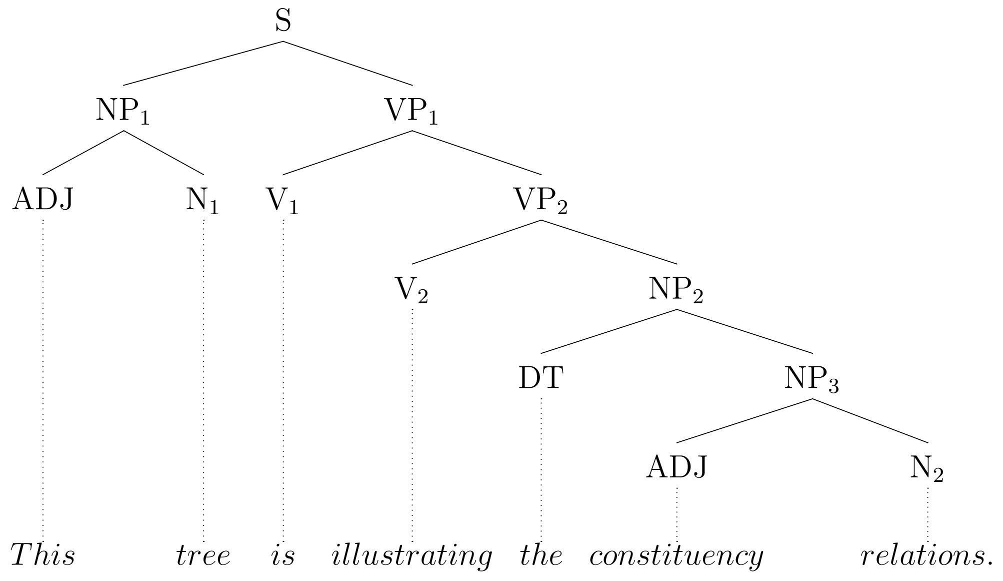

Figure 3.3: Parse tree of a sentence under a constituency grammar

3.2.1 Senna Parser

Senna is a software that outputs various Natural Language Processing (NLP) predictions, such as part of speech (POS) tags, chunking (CHK), named entity recognition (NER), semantic role labelling (SRL) and syntactic parsing (PSG) [17].

Senna parser is a discriminative parser, as opposed to a generative parser; instead of guessing what is the language model behind a target sentence, it tries to predict the structure which the sentence most probably corresponds to [16].

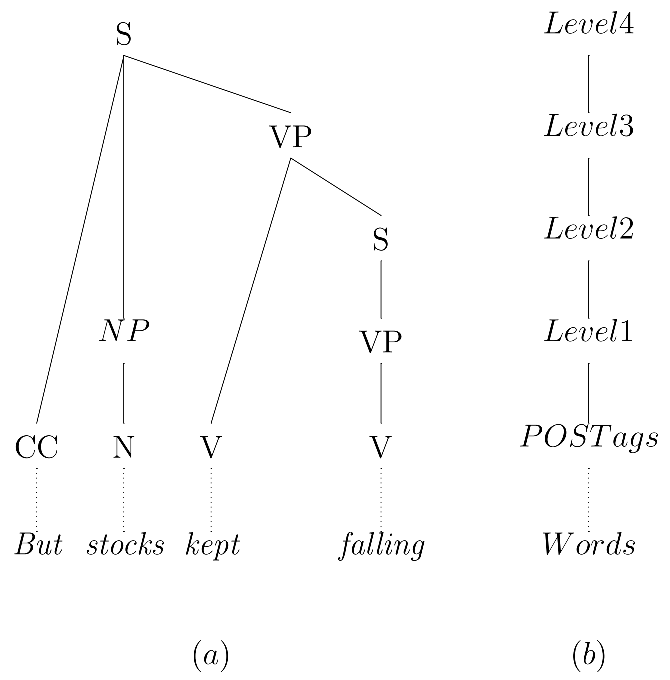

Figure 3.4: (a) Parse tree representation (b) Senna definition of levels

Senna parser is a fast discriminative parser, which does not rely on information extracted from PCFGs [29], and it does not rely on most of the classical parsing features [16].

Senna uses specialized word representations [18] and then tackles the problem of parsing as a recursive tagging task.

Senna parser is a neural network parser, and is based on a Convolutional Neural Network (CNN) adapted for text. In Senna CNN is combined with a structured tag inference in a graph, where the resulting model is named Graph Transformer Network (GTN).

Senna generates a parse tree using chunking as its base task. Chunking is an NLP task, in which a sub-sequence of words in a sentence are selected to represent a phrase.

It uses a bottom up approach: in each level a task of chunking is performed to get the higher level, recursively performing this process until a root label is achieved.

3.2.2 Stanford Parser

Stanford parser is an unlexicalized PCFG parser [35], i.e. it uses POS tags as leaf nodes of the parser tree.

Stanford parser works on probabilistic context free grammar (PCFG) and uses maximum likelihood estimation over subcategorized grammar. Subcategorization means that a word category is divided into several categories, for example verb phrases may be divided into finite and non-finite verb phrases [35].

Stanford parser uses context for phrase capture [35]. It uses parent annotation to denote the context of the phrase, i.e. phrases are marked with their parents; a phrase P whose parent is S is marked as Pˆ S.

The main distinction of Stanford parser, is that it de-emphasizes importance of words in parsing [35], and puts more importance on the categories and subcategories of words in finding most likely parse trees.

3.2.3 Bikel-Collins Parser

Bikel-Collins parser is an implementation of Collins parser [14] by Bikel. This statistical parser is based on probabilities of dependencies between head-words in parse-trees [6].

Collins parser decomposes the process of tree generation into many smaller problems and thus makes the estimation of the parameters tractable [6, p. 481].

This parser uses lexical information by modelling head-modifier relations between pairs of words [15, p. 184]. It is also a maximum likelihood parser, which finds a tree that has the maximum likelihood for a given sentence based on the training corpus.

Formally, given a sentences S and a tree T, the model [15, p. 184] estimates the conditional probability $P(T|S)$. The most likely parse under the model is then:

$$T_{best} = argmax_T P(T|S)$$

Collins parser assumes a different representation of Parse tree as a set of baseNP and a set of dependencies [15, p. 184]. Lets call the set of baseNPs B, and the set of dependencies D. The model then becomes

$$P(T|S) = P(B, D|S) = P(B|S) \times P(D|S, B)$$

Sentence S is a list of words tagged with part-of-speech tags: $S =< (w_1, t_1), (w_2, t_2), \ldots, (w_n, t_n) >$. For POS-tagging Collins parser employs maximum entropy tagger.

Parsing algorithm is a simple bottom-up chart parser [6, p. 481].

3.2.4 Berkeley Parser

Berkeley parser is an implementation of most likely parse tree parsing [43]. However, the major difference with other parsers is that Berkeley parser employs a technique which alternately merges and splits basic non-terminal symbols to maximize the likelihood of training treebank while keeping the size of the learned grammar under control.

Berkeley parser learns by obtaining an $\overline{X}$ grammar from the training set by binarizing the grammar X of the treebank [44, p. 405]. This binarization is chosen to be simple left branching.

Learning is done by Expectation-Maximization algorithm, using inside-outside probabilities to find out latent annotation. Formally, given a sentence S and its unannotated tree T, consider a non-terminal A spanning (r, t) and its children B and C spanning (r, s) and (s, t) respectively. Let A_x be a sub-symbol of A, B_y of B and C_z of C. Then the inside and outside probabilities can be computed recursively as follows [43, p. 434]:

$$\begin{aligned}
P_{IN}(r, t, A_x) &= \sum_{y,z} \beta(A_x \to B_y C_z) \times P_{IN}(r, s, B_y) P_{IN}(s, t, C_z) \\
P_{OUT}(r, s, B_y) &= \sum_{x,z} \beta(A_x \to B_y C_z) \times P_{OUT}(r, t, A_x) P_{IN}(s, t, C_z) \\
P_{OUT}(r, t, C_z) &= \sum_{x,y} \beta(A_x \to B_y C_z) \times P_{OUT}(r, t, A_x) P_{IN}(r, s, B_y)
\end{aligned}$$

In Expectation step, posterior probabilities (P_{IN}, P_{OUT}) of each annotated rule is computed, while in the Maximization step, the rule probabilities (β) are updated by the weighted observations of above probabilities [43, p. 434].

To break the local maxima of the EM algorithm, the learning algorithm repeatedly performs splitting of the symbols. The symbols are split into two with 1% randomness added to break the symmetry. To reduce the size of the resulting grammar, merging step is performed after each splitting [43, p. 435].

3.2.5 BLLIP (Charniak Reranking) Parser

BLLIP is an implementation of Charniak reranking parser [12, 11, 10] that uses Maximum-Entropy inspired parser learning. Formally, if the generative model for the parse tree π is defined as:

$$p(\pi) = \prod_{c \in \pi} p(t|l, H) \times p(h|t, l, H) \times p(e|l, t, h, H)$$

Where, c is a constituent of the tree, t is the pre-terminal of c, h is the head of c, l is the label of c, H is the information outside c that is relevant and e is the expansion of c into other constituents [11, p. 132].

Chapter 4

Methodology

4.1 Overview

In this chapter, we will describe the methodology of evaluation we have employed for this thesis.

We evaluated nine (9) parsers: 6 of these parsers are constituency (phrase-structure) parsers, while three (3) are dependency parsers.

We performed evaluation in two classes: we compared constituency parsers using PARSEVAL and FREVAL measures, whereas, we evaluated all nine (9) of the parsers, using unlabelled attachment score (UAS) on dependency trees. For constituency parsers, we converted the constituency (phrase-structure) trees to dependency trees for evaluation purposes. LTH constituent-to-dependency conversion tool [33] for Penn-style treebanks was used for this conversion.

We selected Twitter data to be the target for our parsing evaluations. The reasons for this choice is the pervasiveness of Twitter as a common user sharing microblogging service available as well as availability of access to Twitter text generated by users.

We evaluated the contemporary methods and tools of parsing text, including one that specifically targets microblog text. We evaluated various combinations of methods and tools in order to understand and to evaluate the efficacy of the state of the art methods.

4.2 Datasets

We have used mainly two data sets in our study. For our main dataset we have selected one that is prepared by Foster et al [23] named Web2.0 dataset. This is the dataset we used as a gold standard for evaluation purposes. The other dataset we used is Penn Treebank (Wall Street Journal) corpus. Though this is a standard dataset used for parser training and evaluations, we used this dataset for training purposes only. We trained the parsers using PTB WSJ chapters 2 through 22 or used pre-trained models (trained on the same sections of PTB-WSJ) that are provided with the parsers distributions.

4.2.1 Web2.0 Twitter Dataset

This dataset consists of 1000 manually annotated sentences taken from tweets and a discussion forum posts. The data was prepared by extensive process whereby it was first parsed automatically then parses were annotated using Penn Treebank bracketing and then hand annotated using Penn Treebank Trees and subjective decision were taken in order to decide on elements specific to tweets. The process was done twice by the annotator and then the parsing was validated by a second annotator who performed this process on 10% of the data. A 95.8% agreement was found between the two annotators. The Twitter dataset consists of 519 manually annotated sentences taken from tweets from a corpus of 60 million tweets on 50 themes. 269 sentences are grouped for development of parser (called TwitterDev) and the remaining 250 as a test set (called TwitterTest). Usernames have been replaced by a generic text 'Username' and URLs have been replaced by a generic text 'Urlname' . The discussion forum dataset consists of 481 sentences taken from BBC Sports 606 threads. The forum posts were split into sentences by hand. The development set consists of 258 sentences (called FootballDev) and the test set consists of 223 sentences (called FootballTest)

Corpus Name	**Number of Sentences**	**Mean Sentence Length**	**Median Sentence-Length**	**Standard Deviation Sentence Length**
TwitterDev	269	11.1	10	6.4
TwitterTest	250	11.4	10	6.8
FootballDev	258	17.7	14	13.9
FootballTest	223	16.1	14	9.7

Table 4.1: Basic statistics on the Web2.0 datasets

The dataset is provided in three files. The files ending in _gold contain the gold phrase structure trees. The files ending in _tokens contain the tokenized sentences. The files ending in _goldpos contain the POS-tagged sentences (in reduced CoNLL format). The files ending in _sd165 are the result of applying the Stanford constituency-to-dependency converter (v.1.6.5) to the gold trees.

4.2.2 Penn Treebank

We have used Wall Street Journal (WSJ) archive of Penn Tree bank 3 (PTB3). PTB-WSJ contains 2,499 English language news stories obtained from Wall Street Journal.

We used Wall street journal section 02 through 21 for training purposes and 00 and 01 for evaluation purposes. The training set consists of 38,541 parsed sentences with mean sentence length of 23.84 words with standard deviation of 11.18 words and median sentence length of 23.

In preparation of this dataset, PTB POS tagging is done in a two stage process where an automated tagging process is employed in the first stage and human annotators are employed in the second stage.

PTB Bracketing (Syntactic Parsing) is also done in a two stage process; 1) Automated Bracketing which is corrected by 2) Human annotators.

Figure 4.1 shows the sentence length distribution differences of the two dataset: Web2.0 TwitterTest dataset and PTB-WSJ dataset. Web2.0 Twitter dataset has an average length of 11.4 words with standard deviation of 6.8 while the PTB-WSJ dataset has an average sentence length of 23.84 words with a standard deviation of 11.18.

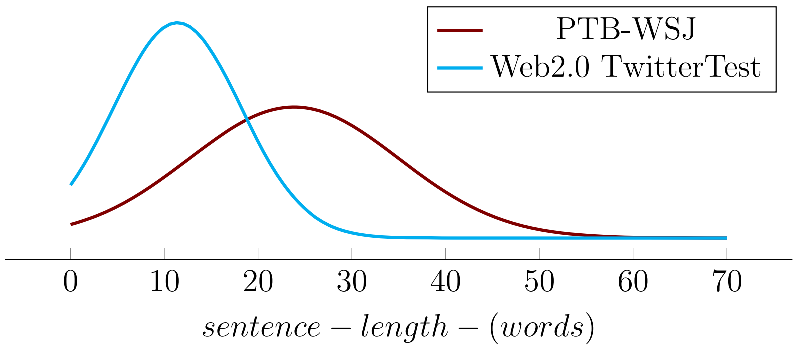

Figure 4.1: Word length distributions of Web2.0 TwitterTest and PTB-WSJ datasets

PTB Tagset

PTB Tag set is based on Tag set of Brown corpus which originally consists of 87 simple tags. PTB has reduced the number of tags based on lexical recoverability. PTB adds to Brown Corpus in encoding words syntactic function into the tags. PTB has indeterminacy where there is an ambiguity between multiple tagging for a work. In such cases, PTB allows multiple tags for a word; however, this is restricted to few common occurrences such as JJ-NN (adjective or noun as pronominal modifier), JJ-VBG (adjective or gerund/present participle), JJ-VBN (adjective or past participle), NN-VBG (noun or gerund/present participle) and RB-RP (adverb or particle). PTB Tag set consists of 36 POS tags and 12 other tags (for punctuation and currency symbols)

4.3 Parsers

We have selected eight (8) parsers for our evaluations. The criteria we used for this selection base basically the popularity of the parsers, the availability of parser executables and/or source code.

Following is a description of the execution details for running the parsers.

MST Parser

MST parser is developed in Java and its source code is available [3] online.

It is bundled with make and build command files, which seamlessly build the executable.

It is bundled with two pre-trained models: *dep.model* and *dep-lab.model*. The *dep-lab.model* is labelled model.

The input data file should contains tokenized text with each sentence separated with a newline character.

The following command:

java -classpath ".:lib/trove.jar" -Xmx1800m mstparser.DependencyParser test model-name:dep.model test-file:test.txt output-file:out.txt format:CONLL

will use the input file *test.txt* and will create an output file *out.txt* containing dependency trees for each of the sentences in test.txt in CONLL format using the pre-trained model *dep.model*.

We employed the unlabelled trained data in our evaluations.

MALT Parser

MALT parser is implemented as a Java program [32]. It is distributed with its source along with compiled java byte code.

We used the latest version (version 1.8.1) at the time of access in our evaluations. A non-optimized version of the MALT parser system was used in our evaluations. We used the parser with the default options, so the performance parameters might not be perfect for our test set.

MALT does not come with a pre-trained model. We trained a model from PTB-WSJ section 02 through 21 by this command:

java -jar maltparser-1.8.1.jar -c wsj-02-21 -i wsj-02-21.conll -m learn

which created a model *wsj-02-21.mco* as specified by the argument *-c*.

We then used the created mode to evaluate our test set by this command:

java -jar maltparser-1.8.1.jar -c wsj-02-21 -i $testFile -o $outFile -m parse

which used the earlier created mode *wsj-02-21.mco* to parse the provided *$testFile* and created an output file *$outFile*.

Tweebo Parser

Tweebo parser is a parser developed in C programming language and is available in source code online [2].

Since this parser is developed specifically for Twitter which is a canonical instance of Microblogging service, this parser is quite relevant in the current discussion. Though determination of the value of specialized tagging in parsing of microblog text for higher level NLP tasks is outside the scope of this thesis, we cover the parsing performance of this parser and compare it with the other parsers available to determine how it fares.

Since, the parser and its precedent POS tagger use specialized POS tag set, the comparison of labelled PARSEVAL measures with the other parsers' output may not be very pertinent, however we will use the unlabelled PARSEVAL measures to compare the parsing performance of various parsers utilizing different tag sets.

We hypothesized that Tweebo parser could outperform the non-Twitter specific parsers since it uses Twitter specific POS tagging and parsing techniques.

Tweebo comes with build command *install.sh* and a run command *run.sh* for Linux environments.

It comes with pre-trained models and thus the *run.sh* command only takes a single argument: the name of the input text file and prints the output on standard out which can be redirected.‘

Preprocessing Tweebo parser has a peculiar default tokenization so that employing the default tokenization created substantially negative performance as compared to the gold standard.

So a pre-tokenized version of the gold standard text was used to evaluate Tweebo parser. However, in the given tokenization scheme, Tweebo does not utilize normal tokenization heuristics and just assumes a white space as a separator in all cases, resulting in an incorrect tokenization in several cases as described below. For example possessive 's' ('s) ending is not tokenized as a separate token, rather, it is assumed to be part of of the single token. This approach creates problem in assigning correct head of the sentence.

Based on our experimentation on the training data, several rules were devised to pre-process the test set. A discussion of these is presented below.

Rules for standardizing Tweebo input Following is the list of the rules that were employed to pre-process test set.

- Short hand notation for multi word tokens such as "I′m" is tokenized as two tokens by default. So a rule is created to add a space delimiter between every non space and instances of the following exact matches.

  (′m, ′M, ′s, ′S, ′re, ′RE, ′ve, ′VE, ′ll, ′LL, ′d, ′D)

  where "′m" is shorthand for "am". "′s" is shorthand for "is". "′re" is shorthand for "are". "′ve" is shorthand for "have". "′ll" is shorthand for "will". "′d" is shorthand for "had' and "would".

  An space was added between "<nonSpace>′m" so that it becomes "<nonSpace><space>′m"

  $< nonSpace >' m \Longrightarrow < nonSpace >< space >' m$

  $< nonSpace >' M \Longrightarrow < nonSpace >< space >' M$

  $< nonSpace >' s \Longrightarrow < nonSpace >< space >' s$

  $< nonSpace >' S \Longrightarrow < nonSpace >< space >' S$

  $< nonSpace >' re \Longrightarrow < nonSpace >< space >' re$

  $< nonSpace >' RE \Longrightarrow < nonSpace >< space >' RE$

  $< nonSpace >' ve \Longrightarrow < nonSpace >< space >' ve$

$< nonSpace >' VE \Longrightarrow < nonSpace >< space >' VE$

$< nonSpace >' ll \Longrightarrow < nonSpace >< space >' ll$

$< nonSpace >' LL \Longrightarrow < nonSpace >< space >' LL$

$< nonSpace >' d \Longrightarrow < nonSpace >< space >' d$

$< nonSpace >' D \Longrightarrow < nonSpace >< space >' D$

- Similarly the Tweebo tokenization is very simple when it comes to currency amount values tokenization. An space was added between the currency symbol ($) and any numeric token.

  A space was added between $<Number> so that it becomes $<space><number>. e.g. $54.0 becomes $ 54.0.

- Finally, the numeric percentage values are tokenized incorrectly, creating problems in parsing. A space was added between a numeric token and the percent sign (%).

  A space was added between <number>% so that it becomes <number><space>% e.g 34% becomes 34 %.

Post Processing: Working on tokenized text, Tweebo separates the apostrophe from the verb, e.g "$I'm$" becomes "I", "$'$", and "m". Whereas in the gold standard, it is "I" and "$'m$". If we process without additional preprocessing as suggested below, Tweebo tokenized this as one token "$I'm$", which is also a wrong head. So a rule is created to join the apostrophe ($'$) with the verb (m).

Also, Tweebo parses single tweets as multiple sentences, whereas in the gold standard, every tweet is parsed as a single sentence with a single head. A rule is applied where the second head is always made a descendent of the first head. A rule was created in which, the first head of the sentence is preserved, and all subsequent heads of the sentence are made dependent on the previous head.

4.3.1 Senna Parser

Senna parser is implemented in C programming language and is available with source code with pre-built binaries for Windows and Linux platforms [45].

Senna takes various parameters to determine what type of output is requested. We executed Senna with *-psg* argument, specifying that we are interested in constituency parse tree output for the provided input.

It takes the input from standard in and outputs the result to the standard out.

please	VB	S-VP	O	please	S-V	(S1(SINV(VP*
no	DT	B-NP	O	-	B-A1	(NP*
adam	NN	I-NP	O	-	I-A1	*
lambert	NN	E-NP	O	-	E-A1	*))
!	.	O	O	-	O	*))

Table 4.2: An example output of Senna parser

The PSG output lists all tokens on a single line with the parse tree in the last column, where the token is represented by an asterisk as in Fig. 4.3.1.

4.3.2 Stanford Parser

Stanford parser is bundled up with a trained PCFG language model for English from PTB-WSJ corpus, as well as a language model that ignores case differences between words: PCFG-case-less model; this model is also used in our evaluations.

Stanford parser is implemented in Java programming language and is available with source code and pre-built Java executable *stanford-parser.jar* [27]. The latest version at the time of our access was 3.2.0.

The pre-trained model is also provided in *stanford-parser-3.2.0-models.jar*. The two models that we used in our evaluations are *PCFG* and *caseless* which are both created by training on PTB-WSJ section 2 through 21.

We used the following command to parse input with PCFG model:

```
java -mx1800m -cp “$scriptdir/*:” edu.stanford.nlp.parser.lexparser.LexicalizedParser
-sentences newline -outputFormat “penn”
edu/stanford/nlp/models/lexparser/englishPCFG.ser.gz $testFile >$outFile
```

which uses englishPCFG model and and the following command to parse input with caseless model:

```
java -mx1800m -cp “$scriptdir/*:” edu.stanford.nlp.parser.lexparser.LexicalizedParser
-sentences newline -outputFormat “penn”
edu/stanford/nlp/models/lexparser/englishPCFG.caseless.ser.gz $testFile >$outFile
```

4.3.3 Bikel-Collins Parser

Bikel-Collins parser is an implementation of Collins parser by Daniel Bikel in Java programming language as is available with source code. It is implemented in Java programming language and pre-built Java executable is bundled with the distribution [5].

We used the settings as defined in the provided *collins.properties* file defined to emulate Mike Collins' 1997 Model 2 without any change.

4.3.4 Berkeley Parser

Berkeley parser is implemented in Java and is available with source code and pre-built Java executable. We used the BerkeleyParser version 1.7 in our evaluations, the latest available at the time of our access [26].

Berkeley parser is distributed with an pre-trained model *eng_sm6.gr* which is trained on Wall Street Journal.

The command which we used to execute the Berkeley parser is follows:

```
java -jar BerkeleyParser-1.7.jar -gr eng_sm6.gr <$testFile >$outFile
```

4.3.5 BLLIP (Charniak Reranking) Parser

BLLIP parser is implemented in C programming language and is distributed with make file.

We executed the BLLIP parser with provided executable file *parse.sh* with command:

./parse.sh $testFile >$outFile

which uses provided *ec50spfinal* model and *cvlm-l1c10P1* estimator and the provided *EN* English data model.

4.4 Evaluation Methods

This section describes the details of the evaluation metrics used for this work. We employed two sets of evaluations: first for the constituency (phrase structure) parsing and the second for dependency parsing. The other orthogonal axis of evaluation was running-time performance of the parsers.

We focused on two criteria of evaluation: one is the functional performance evaluation and the other is running-time performance evaluation.

4.4.1 Functional Performance Evaluation Metrics

Function performance of parsers is evaluated to determine how good parsing they do. The ‘goodness’ of the parsers is determined by metrics which are different for different types of the parsers, i.e. constituency parsers and dependency parsers.

Evaluation Metrics for Constituency Parsers

Parser evaluations have been done using PARSEVAL [1] which is the most commonly used evaluations method for constituency parsers.

Constituency parsing evaluation is done with the help of two measures: recall and precision. Recall is defined as the percentage of phrase boundaries in the candidate parse that are exactly present in the standard parse, while precision is defined as the percentage of phrase boundaries in the standard parse that are exactly present in the candidate parse. If we assume a test set consisting of sentences $\{S_1, S_2, \ldots, S_n\}$ and their corresponding standard parsed trees $T^S = \left\{\tau_1^S, \tau_2^S, \ldots, \tau_n^S\right\}$. Now let the parser output be a set of candidate trees $T^C = \left\{\tau_1^C, \tau_2^C, \ldots, \tau_n^C\right\}$. We can write the definition of trees as a set of all labelled constituents :

$$\tau = \{(i, X, j)\}$$

where (i, X, j) stands for a constituent in τ that cover the span i, to j with label X.

The PARSEVAL (Labelled recall, precision) are as follows:

$$LR(T^C, T^S) = \frac{\sum_i |\tau_i^S \cap \tau_i^C|}{\sum_i |\tau_i^C|}$$

and

$$LP(T^C, T^S) = \frac{\sum_i |\tau_i^S \cap \tau_i^C|}{\sum_i |\tau_i^S|}$$

finally, the PARSEVAL F1-score is the standard

$$F1 = \frac{2 \times (LR \times LP)}{(LR + LP)}$$

We also employed a more elaborate approach to PARSEVAL measure, called FREVAL [4]. This measure is a generalization of PARSEVAL from individual nodes to arbitrary sized fragments, i.e. sub-trees defined as connected non-empty sub-graphs of a tree. To describe formally, let $max = |\tau|$ denote the number of nodes in a tree τ. A tree τ is represented by a sequence of sets of situated fragments

$$Frag_1(\tau), Frag_2(\tau), \ldots, Frag_{max}(\tau)$$

where for ever $1 \leq s \leq max$, we define $Frag_s(\tau)$ as the set of all situated fragments φ in τ of size $|\varphi| = s$. A situated fragment (i, φ, j) is a fragment. More formally,

$$Frag_s(\tau) = \{(i, \varphi, j) | fragment(\varphi, \tau) \wedge |\varphi| = s \wedge span(\varphi) = \langle i, j\rangle\}$$

Given the fragment size s Fragment Labelled Recall of size s (FLR_s) and Fragment Labelled Precision of size s (FLP_s) are defined as:

$$FLR_s(T^C, T^S) = \frac{\sum_i |Frag_s(\tau_i^C) \cap Frag_s(\tau_i^S)|}{\sum |Frag_s(\tau_i^S)|}$$

and

$$FLP_s(T^C, T^S) = \frac{\sum_i |Frag_s(\tau_i^C) \cap Frag_s(\tau_i^S)|}{\sum |Frag_s(\tau_i^C)|}$$

FREVAL shows the performance of parsers on varying fragment sizes, so as to help make finer grain performance comparisons.

Evaluation Metrics for Dependency Parsers

Following the standard route, if we employ recall and precision to evaluate dependency trees, the recall is defined as the percentage of dependency relationships in the key that are also found in the answer while the precision is defined as the percentage of dependency relationships in the answer that are also found in the key. However, for dependency tree evaluation labelled and unlabelled attachment scores are generally used [38] for parser evaluation.

Attachment score is a single accuracy metric as defined in [37]. It is defined as the percentage of words that have the correct head. A single accuracy metric is valuable because of the single head property of dependency trees. This one-to-one correspondence between the words and their heads allows us to use attachment score as the metric to determine the accuracy of the parse.

The evaluation is generally done with two different methods *labelled attachment score* (LAS) and *unlabelled attachment score* (UAS). Assuming we have a given dependency tree parse of a sentence in the form of

$$((word, head, label)_1, (word, head, label)_2, \ldots, (word, head, label)_{max})$$

and that we have a set of candidate parses $D^C = \{D_1^C, D_2^C, \ldots, D_n^C\}$ and a set of standard parse $D^S = \{D_1^S, D_2^S, \ldots, D_n^S\}$ of the same set of n sentences, the labelled attachment score is defined as two metrics:

$$LAS_{micro} = \frac{\sum_i^n (|match(D_i^C, D_i^S)|)}{\sum_i^n |D_i^S|}$$

and

$$LAS_{macro} = \frac{\sum_i^n \frac{|match(D_i^C, D_i^S)|}{|D_i^S|}}{n}$$

where $match(D^C, D^S)$ is defined as

$$match(D^C, D^S) = \left\{ (word, head, label)_i \middle| \begin{array}{c} (word^C, head^C, label^C)_i = D^C[i] \wedge \\ (word^S, head^S, label^S)_i = D^S[i] \wedge \\ word^C = word^S \wedge head^C = head^S \wedge \\ label^C = label^S \end{array} \right\}$$

Here LAS_{micro} is averaged over words and LAS_{macro} is average of sentences.

For the unlabelled attachment score we ignore the label as we use the sets $DU = \{(word, head)|(word, head, label) \in D\}$. Now assuming $\{DU_1^C, DU_2^C, \ldots, DU_n^C\}$ be the set for candidate parses and $\{DU_1^S, DU_2^S, \ldots, DU_n^S\}$ be the set for standard parses, the unlabelled attachment score is again defined as two metrics:

$$UAS_{micro} = \frac{\sum_i^n (|DU_1^C \cap DU_1^S|)}{\sum_i^n |DU_i^S|}$$

and

$$UAS_{macro} = \frac{\sum_i^n \frac{|DU_1^C \cap DU_1^S|}{|DU_i^S|}}{n}$$

Here, UAS_{micro} is averaged over words and UAS_{macro} is average of sentences.

One implementation is given by *MALTEval* [40] for UAS and LAS evaluations.

We have used unlabelled attachment score for the reason that the set of parsers we have evaluated do not share the tag sets. Though most of the parsers use Penn treebank tag set, some parsers use different tag set specific to the language of the Twitter.

One problem we encounter in using attachment score evaluation as implemented in $MALTEval$ is that it requires the candidate and standard parses to be tokenized exactly the same to compare the heads of the tokens. However, we found out that the various parsers employ different tokenization heuristics which cause different tokenizations, which $MALTEval$ cannot handle.

So we implemented our own evaluator to first match the tokenization of the two sentence using maximum spanning the tokens with a dynamic programming solution to the longest common subsequence (LCS) problem, so the match function was changed accordingly:

$$match(D^C, D^S) = \left\{ (word, head)_i \middle| \begin{array}{c} (word^C, head^C)_i = DU^C[i] \wedge \\ (word^S, head^S)_j = DU^S[j] \wedge \\ word^C = word^S \wedge head^C = head^S \wedge \\ LCS(DU^C, DU^S, i) = \\ LCS(DU^S, DU^C, j) \end{array} \right\}$$

Where $LCS(D^C, D^S, i)$ gives the index of the word in the longest common token list corresponding to the ith word in the sentence D^C.

To evaluate the effect of correct tokenization on the parser performance we evaluated the parsers with and without pre-tokenized text.

4.4.2 Running-time Performance Evaluation

The other aspect that we focused on evaluating on the performance of the parsers was to find out how fast they can parse the text. When it comes to parse microblog or Twitterverse, it is imperative that it is done as fast as possible.

For this purpose we timed the parsing of the tweets for each of the parsers in batches running multiple times, and then averaged the time for each tweet.

We timed for the real, user and system time for the processes on a Unix machine, using the standard time command. The Linux 'time' command gives three different measures for time taken by a command: Real time is the clock time the process took to complete the task, User time is the actual CPU time that the process took to run in the user mode, and Sys time is the actual CPU time that the process took to run in the kernel mode. We kept our analysis limited to user time.

The machine specifications were Intel Core i7 Quad Core 2.00 GHz, with 8 GB of RAM, running 64 Bit Windows 8.1 Pro operating system. In order to evaluate all parsers on one environment. Since some of the parsers are only available on Linux, therefore we created a virtual machine on VMware running Ubuntu 14.04 Linux with 1 GB of RAM in the virtual machine.

Since the purpose of this evaluation was a comparison of the performance of the various parsers, so this setup, though far from ideal, worked fine for evaluation various parsers running-time performance.

Chapter 5

Evaluation and Discussion

5.1 Functional Performance Evaluation

We have done functional performance evaluation in two modes: we have converted the constituency tree output of all of the constituency parsers to dependency trees, and then evaluated all dependency trees using unlabelled attachment scores.

In addition, we compared the constituency (phrase-structure) parsers, using standard PARSEVAL (and extended form of it, FREVAL) measures.

5.1.1 Dependency Parse Comparison

We evaluated various parsers for their functional performance in parsing Twitter text to create dependency trees.

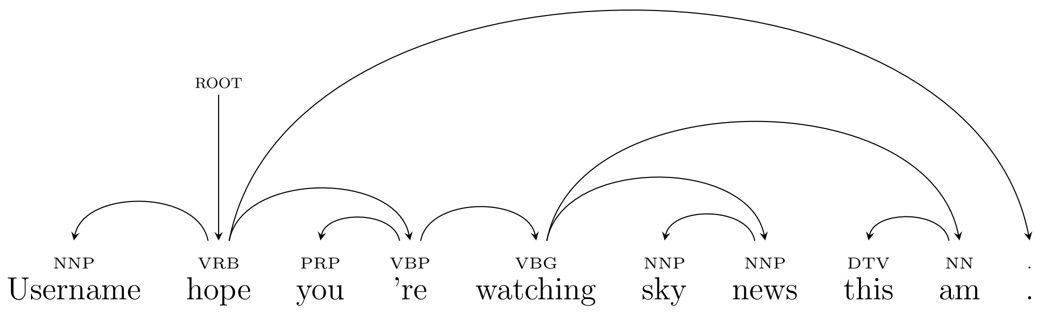

Figure 5.1: Dependency tree of an example sentence in Twittertest gold standard

The comparison were done using two measures: micro-averaged Unlabelled Attachment Score (micro-UAS) and macro-averaged Unlabelled Attachment Score (macro-UAS).

The micro-UAS and macro-UAS are calculated by comparing the parses generated by the parser under evaluation with the gold-parses in our test set.

For example, the tweet “Username hope you’re watching sky news this am” is parsed in TwitterTest as gold standard as in Fig 5.1

Tweebo Parser

Being a dependency parser, Tweebo parser was evaluated using the unlabelled attachment score.

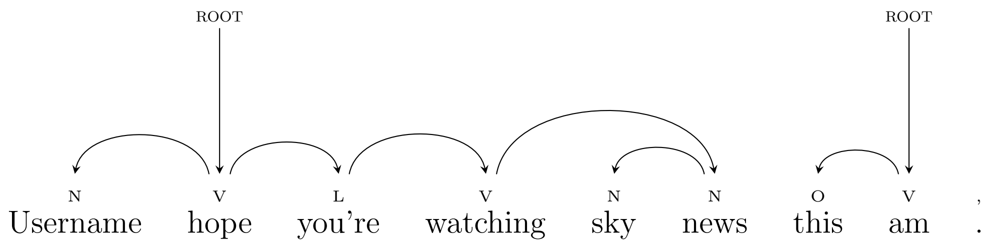

Figure 5.2: Dependency tree of the example sentence as parsed by Tweebo parser

We can see in Fig. 5.2 that the Tweebo parser parses a tweet with multiple roots.

The 250 sentences of Web2.0 TwitterTest dataset was evaluated, the micro average UAS score was 58.67% while the macro Average UAS was almost the same 58.75%.

We also evaluated Micro and Macro UAS ignoring punctuation tokens, and obtained same micro average of 58.67% but increased macro average 69.12%

Evaluating for only the sentences which have complete token matching of the candidate parse with the gold standard parse, which where 162 in number, the score was significantly better at 63.55% and 62.99% respectively.

Only 3 sentences out of 250 had complete match with the gold standard.

Number of Sentences:	250
Micro Average UAS:	58.67
Macro Average UAS:	58.75
Micro Average UAS (Ignore Punctuation):	58.67
Macro Average UAS (Ignore Punctuation):	69.12
Number of Matched Token Sentences:	162
Micro Average UAS (Matched Token):	63.55
Macro Average UAS (Matched Token):	62.99
Number of Complete Matches:	3

Table 5.1: Functional performance of Tweebo parser

Tokenized Input: For tokenized input, the performance of Tweebo parser declined a little bit, the reason being, Tweebo uses some peculiar heuristics in tokenization, for example *I'm* is tokenized as *I*, ', and *m*.

The Micro average Unlabelled Attachment Score is 57.74%, while macro average for UAS was 57.66%. Ignoring punctuation gave a significantly better parse performance in terms of macro-average UAS 68.81% macro UAS, only the same on micro-average UAS with 57.74% micro UAS.

The number of matched tokens for a tokenized input increased from 162 to 191. However the parsing performance decreased, giving a UAS about 57%. Ignoring punctuation tokens gave significant improved parsing score of 67%. Complete match of the parse trees was still limited to 3 sentences.

Number of Sentences:	250
Micro Average UAS:	57.74
Macro Average UAS:	57.66
Micro Average UAS (Ignore Punctuation):	57.74
Macro Average UAS (Ignore Punctuation):	68.81
Number of Matched Token Sentences:	191
Micro Average UAS (Matched Token):	62.11
Macro Average UAS (Matched Token):	61.62
Number of Complete Matches:	3

Table 5.2: Functional performance of Tweebo parser on tokenized input

MST Parser

We performed evaluation on MST parser using the bundled unlabelled trained model as well as trained on Web 2.0 TwitterDev dataset. Since MST parser requires a tokenized input, we used Stanford tokenizer to tokenize the input and provided that to the MST parser.

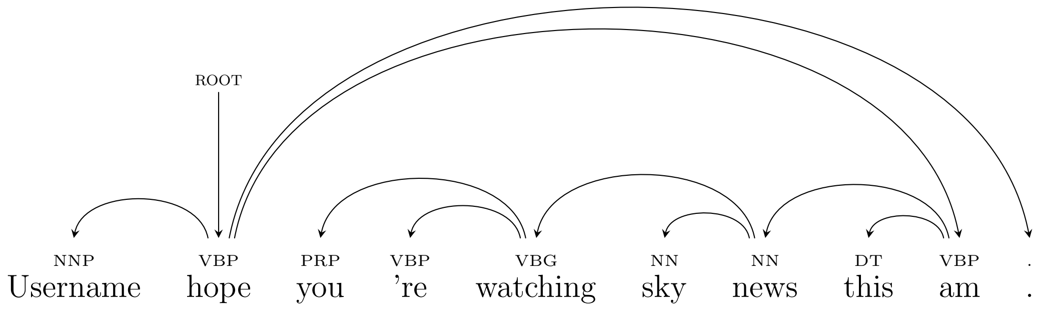

Figure 5.3: Dependency tree of the example sentence as parsed by MST parser

The Stanford tokenizer did a reasonably good job in tokenizing input, 201 sentences out of 250 matched tokenization with the gold standard.

Using the supplied unlabelled trained model, given on tokenization from Stanford tokenizer, the performance of MST parser was quite poor. It gave only 14.41% micro-averaged UAS with 15.31% macro-averaged UAS. Ignoring punctuation tokens decreased micro-averaged UAS but increased macro-average UAS to 13.79% and 17.37% respectively. From the performance of the parser with the provided trained model, it can be deduced that the model is quite naively trained.

A complete match of only 1 sentence out of 250 was observed.

Number of Sentences:	250
Micro Average UAS:	14.67
Macro Average UAS:	15.72
Micro Average UAS (Ignore Punctuation):	13.78
Macro Average UAS (Ignore Punctuation):	17.72
Number of Matched Token Sentences:	231
Micro Average UAS (Matched Token):	14.68
Macro Average UAS (Matched Token):	15.58
Number of Complete Matches:	1

Table 5.3: Functional performance of MST parser on tokenized input

We also observed that for MST parsing the real time was less than the CPU time which means that the MST parser is CPU bound process and it uses multi-core processing/multi-threading to lessen real parsing time.

MALT Parser

MALT parser does not come with a default model, so training was required. To keep the training comparable with other parsers, we used the same training corpus for all the parsers. We used PTB-WSJ 02-21 corpus, which contains 38543 sentences. The training time was about 4 seconds.

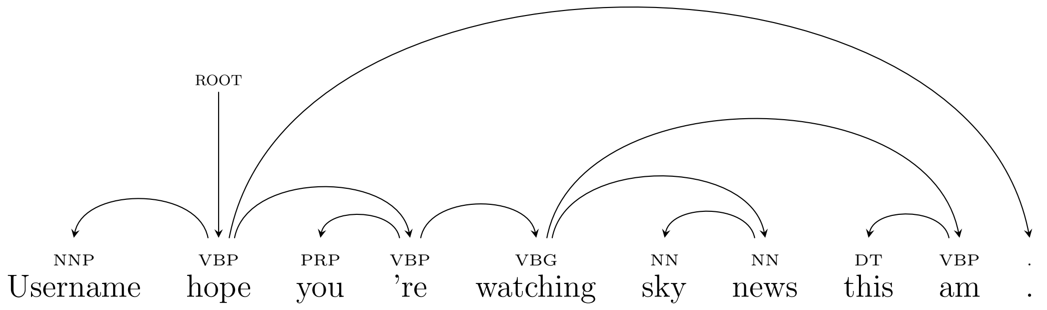

Figure 5.4: Dependency tree of the example sentence as parsed by MALT parser

The performance of MALT parser was not too low but not too high as well. At best it can be classified as a mediocre parser when it comes to parsing microblog text. Though out of 250 sentences, 215 were correctly tokenized, the micro-averaged UAS was 61.72% and macro-averages UAS was found to be 60.01%. Ignoring Punctuation decreased the micro-averaged UAS while increasing the macro-averaged UAS to 57.25% and 62.65% respectively. Since the default correct tokenization rate was high, ignoring the unmatched sentences did not improve the performance level much; only the micro-averaged UAS became 62.88% and macro-averaged UAS became 61.23%.

39 sentences out of 250 had complete matched parses with the parses of the gold-standard.

Number of Sentences:	250
Micro Average UAS:	61.72
Macro Average UAS:	60.01
Micro Average UAS (Ignore Punctuation):	57.25
Macro Average UAS (Ignore Punctuation):	62.65
Number of Matched Token Sentences:	215
Micro Average UAS (Matched Token):	62.88
Macro Average UAS (Matched Token):	61.23
Number of Complete Matches:	39

Table 5.4: Functional performance of MALT parser trained on WSJ

When we trained the MALT parser on Web2.0 TwitterDev data set, the performance declined considerably, due to the limited size of the dataset. Training time was only about 2 seconds (1973 ms).

The parsing performance of MALT parser trained on Web2.0 TwitterDev dataset was only 10.17% micro-averaged UAS and 15.93% macro-averaged UAS. We assume that this performance is a result of small training size and signifies a larger training size requirement of MALT parser. Number of complete matches also reduced to 13 from 37.

Number of Sentences:	250
Micro Average UAS:	10.17
Macro Average UAS:	15.93
Micro Average UAS (Ignore Punctuation):	8.94
Macro Average UAS (Ignore Punctuation):	16.39
Number of Matched Token Sentences:	215
Micro Average UAS (Matched Token):	10.59
Macro Average UAS (Matched Token):	17.15
Number of Complete Matches:	13

Table 5.5: Functional performance of MALT parser trained on Web2.0 TwitterDev

Senna Parser

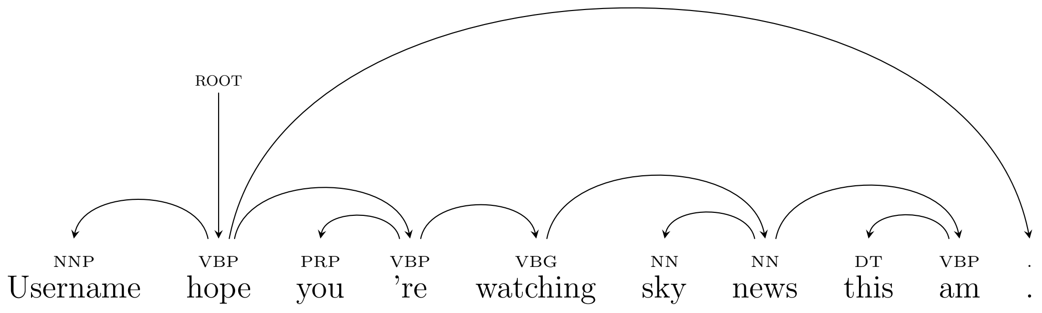

Figure 5.5: Dependency tree of the example sentence as parsed by Senna parser

Senna parser faired quite well on Unlabelled Attachment Score evaluation. For 250 sentences Micro-average UAS was 70.42% and macro-averaged UAS was 71.23. Ignoring punctuation tokens reduced the micro-averaged UAS considerably to 63.52$, signalling that Senna did have a good performance on punctuation tokens and the head assignments, while the macro-averaged UAS improved negligibly to 72.45%.

Also the number of matched token sentences was only 177 out of 250, ignoring the unmatched token sentences improved the parsing score considerably to 75.68% and 77.86% micro-averaged UAS and macro-averaged UAS respectively. This clearly shows importance of tokenizing on Senna's performance.

Number of complete matches was also reasonably high to 75 sentences.

Number of Sentences:	250
Micro Average UAS:	70.42
Macro Average UAS:	71.23
Micro Average UAS (Ignore Punctuation):	63.52
Macro Average UAS (Ignore Punctuation):	72.45
Number of Matched Token Sentences:	177
Micro Average UAS (Matched Token):	75.68
Macro Average UAS (Matched Token):	77.86
Number of Complete Matches:	75

Table 5.6: Functional performance of Senna parser

Running Senna on pre tokenized input did improve performance slightly, though the number of matched token sentences rose to 215 from 177. The micro-averaged UAS was 72.92% while the macro-averaged UAS was 75.91%. This means the tokenization improves better parsing for sentences, but does not improve significantly for words (and phrases).

Number of complete matches for pre-tokenized text also increased from 75 to 87 sentences, about 16%.

Number of Sentences:	250
Micro Average UAS:	72.92
Macro Average UAS:	75.91
Micro Average UAS (Ignore Punctuation):	65.21
Macro Average UAS (Ignore Punctuation):	77.20
Number of Matched Token Sentences:	215
Micro Average UAS (Matched Token):	75.16
Macro Average UAS (Matched Token):	77.96
Number of Complete Matches:	87

Table 5.7: Functional performance of Senna parser on tokenized text

Stanford Parser

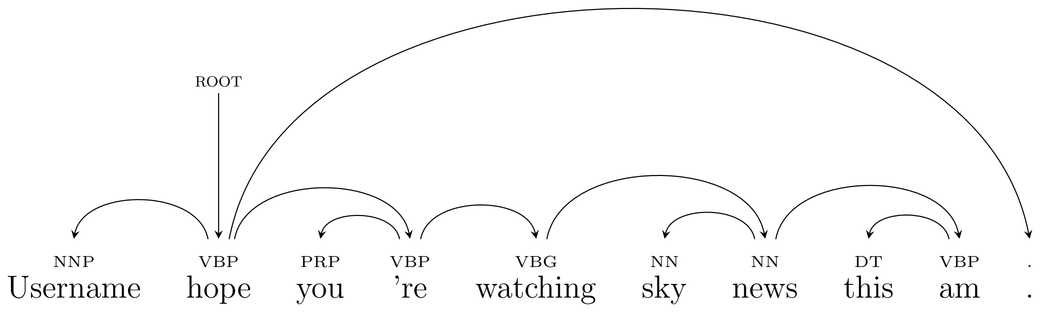

Figure 5.6: Dependency tree of the example sentence as parsed by Stanford parser – PCFG model

We evaluated the Stanford parser using two different models from among those that are provided with the parsers: a standard PCFG model developed from PTB-WSJ and a PCFG-caseless model developed from the same corpus ignoring the letter case.

Evaluating Stanford on PCFG language model which gave quite good results of 71.41% micro-averaged UAS and 71.27% macro-averaged UAS. Ignoring punctuation tokens lowered the micro-average UAS considerably to 64.02% while improved the macro-average negligibly to 72.07%. Similarly since 215 sentences out of 250 had matching tokenization, ignoring mismatched tokens did not improve results significantly.

Number of complete matches were 67 out of 250.

Number Of Sentences:	250
Micro Average UAS:	71.41
Macro Average UAS:	71.27
Micro Average UAS (Ignore Punctuation):	64.02
Macro Average UAS (Ignore Punctuation):	72.07
Number Of (Matched Token) Sentences:	215
Micro Average UAS (Matched Token):	72.67
Macro Average UAS (Matched Token):	72.50
Number of Complete Matches):	67

Table 5.8: Functional performance of Stanford parser using PCFG language model

Running Stanford parser using PCFG model on pre-tokenized text did not improve the results at all, rather it degraded the results a little, with micro-averaged UAS to 71.00% and macro-averaged UAS to 71.28%, though the number of matched-tokenization sentences increased from 215 to 234.

Also the number of complete matches increased slightly from 67 to 70.

Number Of Sentences:	250
Micro Average UAS:	71.00
Macro Average UAS:	71.28
Micro Average UAS (Ignore Punctuation):	62.96
Macro Average UAS (Ignore Punctuation):	72.27
Number Of (Matched Token) Sentences:	234
Micro Average UAS (Matched Token):	72.17
Macro Average UAS (Matched Token):	72.10
Number of Complete Matches):	70

Table 5.9: Functional performance of Stanford parser using PCFG language model on tokenized text

Running Stanford parser with PCFG-caseless model on Web2.0 TwitterTest dataset

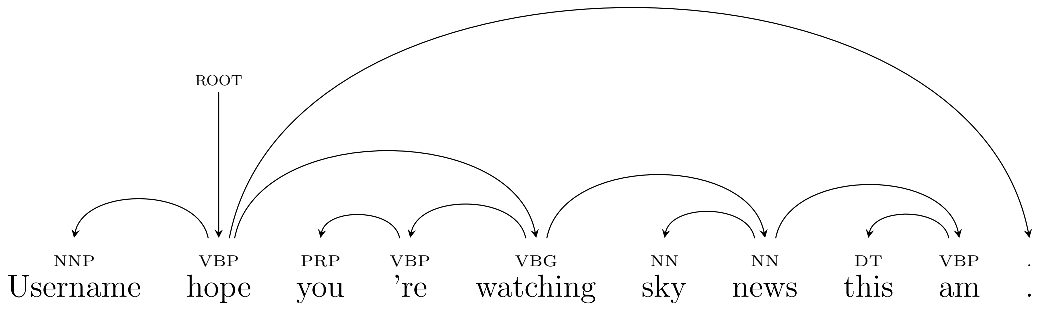

Figure 5.7: Dependency tree of the example sentence as parsed by Stanford parser – PCFG-caseless model

improved the functional performance of the parser a little, increasing the micro-averaged UAS to 72.71% and macro-averaged UAS to 73.07%. Ignoring punctuation increased a little more to 65.14% (compared to 62.96%) and 73.85%(compared to 72.27%) of micro and macro averages.

One additional complete match was obtained using PCFG-caseless model; there were 70 complete matches (100% match between candidate and standard parse trees) when using regular PCFG model, whereas with PCFG-caseless model 71 complete matches were observed.

Number of Sentences:	250
Micro Average UAS:	72.71
Macro Average UAS:	73.07
Micro Average UAS (Ignore Punctuation):	65.14
Macro Average UAS (Ignore Punctuation):	73.85
Number of Matched Token Sentences:	215
Micro Average UAS (Matched Token):	73.86
Macro Average UAS (Matched Token):	74.21
Number of Complete Matches:	71

Table 5.10: Functional performance of Stanford parser using PCFG-caseless language model

Using pre-tokenized text to run Stanford parser with PCFG-caseless language model further improved the functional performance of the parser slightly. It achieved micro-averaged UAS of 72.91% and macro-averaged UAS to 73.59%. Ignoring punctuation decreased the micro-averaged UAS to 64.55% while increasing the macro-averaged UAS to 74.68%. Number of matched-tokenization sentences also rose from 215 to 235.

Only three (3) sentence were additionally completely matched with the parse of the gold-standard.

Number Of Sentences:	250
Micro Average UAS:	72.91
Macro Average UAS:	73.59
Micro Average UAS (Ignore Punctuation):	64.55
Macro Average UAS (Ignore Punctuation):	74.68
Number Of (Matched Token) Sentences:	234
Micro Average UAS (Matched Token):	73.50
Macro Average UAS (Matched Token):	74.04
Number of Complete Matches):	74

Table 5.11: Functional Performance of Stanford parser using PCFG-caseless language model on tokenized text

Berkeley Parser

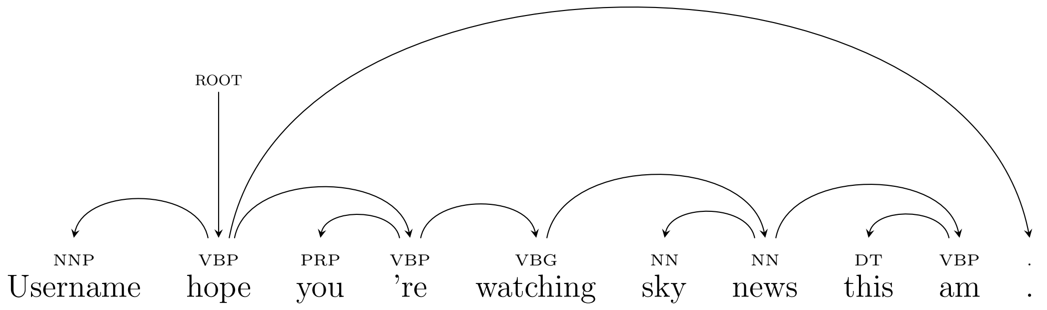

Figure 5.8: Dependency tree of the example sentence as parsed by Berkeley parser

Berkeley parser requires tokenized text. The micro-averaged UAS was to 72.93% and macro-averaged UAS was 72.46% . Ignoring punctuation, degraded micro-average UAS considerably, and improved macro-averaged UAS slightly, to 65.21% and 73.76% respectively. Since almost all of the sentences tokens were matched (249 out of 250), ignoring the unmatched token sentence did almost nothing to the results. Number of complete matches was astounding 80 sentences out of 250 sentences.

Number Of Sentences:	250
Micro Average UAS:	72.94
Macro Average UAS:	72.46
Micro Average UAS (Ignore Punctuation):	65.21
Macro Average UAS (Ignore Punctuation):	73.76
Number Of Matched Token Sentences:	249
Micro Average UAS (Matched Token):	72.94
Macro Average UAS (Matched Token):	72.75
Number of Complete Matches:	80

Table 5.12: Functional performance of Berkeley parser on tokenized text

BLLIP Parser (Charniak Reranking Parser)

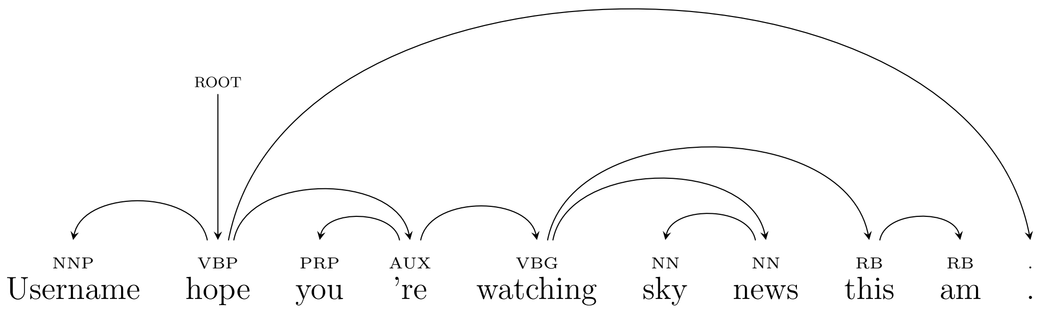

Figure 5.9: Dependency tree of the example sentence as parsed by BLLIP parser

The functional performance of BLLIP (also known as Charniak Reranking) parser was the highest among all the parsers we evaluated.

On Web2.0 Twittertest data, the micro-averaged UAS was 75.04% while the macro-averaged UAS was 76.49%. Ignoring punctuation degraded the word average UAS (micro-averaged) to 67.54% while improved the sentence average UAS (macro-averaged) to 77.37%. Since number of matched token sentences was reasonably high 195 (out of 250), the performance increase ignoring the unmatched token sentences with micro-average increasing to 76.84% and micro-average to 78.27%. A number of complete match was also a great 83 sentences out of 250.

Number of Sentences:	250
Micro Average UAS:	75.04
Macro Average UAS:	76.49
Micro Average UAS (Ignore Punctuation):	67.54
Macro Average UAS (Ignore Punctuation):	77.37
Number of Matched Token Sentences:	195
Micro Average UAS (Matched Token):	76.84
Macro Average UAS (Matched Token):	78.27
Number of Complete Matches:	83

Table 5.13: Functional performance of BLLIP parser

Running BLLIP parser on pre-tokenized text improved the performance results only very slightly, signifying high quality of tokenization as part of the parser.

The micro-averaged UAS was 75.94% while the macro-averaged UAS was 77.10%. Ignoring punctuation decreased the micro-averaged UAS to 68.39% while improved the sentence average to 78.08%.

Number of matched token sentences also increased to 201 (from 195), which is not too much, justifying negligible change in the performance numbers obtained by ignoring non-token-matched sentences. The micro-averaged UAS was 76.38% while macro averaged UAS was 77.59%.

Number of complete matches also increase by one sentence to 84 (out of 250 sentences).

Number Of Sentences:	250
Micro Average UAS:	75.94
Macro Average UAS:	77.10
Micro Average UAS (Ignore Punctuation):	68.39
Macro Average UAS (Ignore Punctuation):	78.08
Number Of Matched Token Sentences:	201
Micro Average UAS (Matched Token):	76.38
Macro Average UAS (Matched Token):	77.59
Number of Complete Matches:	84

Table 5.14: Functional performance of BLLIP parser on tokenized text

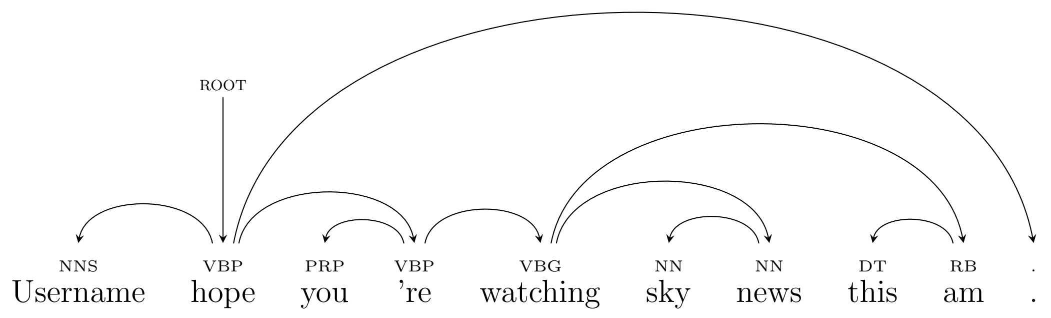

Figure 5.10: Dependency tree of the example sentence as parsed by Bikel-Collins parser

Bikel-Collins Parser

Bikel-Collins parser does not come with included language model, so we trained on PTB WSJ-02-21 to keep consistent baseline. The Bikel-Collins parser requires tokenized text and does not perform any tokenization, other than splitting sentences on space.

Bikel-Collins parser performs at par on Web2.0 TwitterTest dataset as compared to other parsers. In parsing 250 tweets, it performed to micro-averaged UAS of 72.62% and macro-averaged UAS of 73.12%. Ignoring punctuation degraded the micro-averaged UAS to 65.22%, while increased the macro-averaged UAS to 73.54%. Since it does not attempt to tokenize at all, all 250 sentences were matched correct with tokenization.

82 sentences (out of 250) had complete parse match.

Number Of Sentences:	250
Micro Average UAS:	72.62
Macro Average UAS:	73.12
Micro Average UAS (Ignore Punctuation):	65.22
Macro Average UAS (Ignore Punctuation):	74.54
Number Of (Matched Token) Sentences:	250
Micro Average UAS (Matched Token):	72.62
Macro Average UAS (Matched Token):	73.12
Number of Complete Matches):	82

Table 5.15: Functional performance of Bikel-Collins parser on tokenized text

Summary

Comparing all the parsers with or without pre-tokenized text (for those parser that do take non-tokenized text), we see that BLLIP parser (also known as Charniak reranking parser) fares the best in terms of both micro-averaged UAS and macro-averaged UAS. BLLIP parser gave 75.94% micro-averaged UAS over all words in the tweets and gave 77.10 macro-averaged UAS over all sentences in the tweet dataset. This result was obtained by providing pre-tokenized sentences.

We see that Senna parser was the second best parser in terms of its macro-average performance on tokenized text giving 73% micro-average UAS and 76% macro-average UAS.

After Senna parser, Stanford, Berkeley and Bikel-Collins all parsers fared almost the same on around 72% micro and 73% macro-average.

Parser	**micro-avg UAS**	**macro-avg UAS**
Berkeley parser on tokenized text	72.94	72.46
Bikel-Collins parser on tokenized text	72.62	73.12
BLLIP [1] parser	75.04	76.49
BLLIP parser on tokenized text	**75.94**	**77.10**
MALT parser on tokenized text	61.72	60.01
MST parser on tokenized text	14.67	15.72
Senna parser	70.42	71.23
Senna parser on tokenized text	72.92	75.91
Stanford parser-PCFG	71.41	71.27
Stanford parser-PCFG on tokenized text	71.00	71.28
Stanford parser-PCFG-caseless	72.71	73.07
Stanford parser-PCFG-caseless on tokenized text	72.91	73.59
Tweebo parser	58.67	58.75
Tweebo parser on tokenized text	57.74	57.66

Table 5.16: Functional performance of various parsers — micro-averaged and macro-averaged Unlabelled Attachment Score (UAS) measures

5.1.2 Constituency Parse Comparison

Senna Parser

The performance of Senna parser was much degraded, as expected, when it is used to parse Twitter text as compared to the normal news text. Though Senna is not designed to be a parser per se, but an NLP tool for higher level tasks such as SRL, we can get parsed trees output from Senna. It is reported that Parsing is important part of SRL and the SRL evaluation of senna is pretty high going to F1 score of 79.2% which is usually much lower than the F1 score in just the parsing task.

When we evaluated Senna to work with Twitter data, the performance of Senna was degraded. We got the F1 score of PARSEVAL measure of only 72.49% with precision of 73.61% and recall of 71.40%.

In total 250 sentences were evaluated, where 75 sentences were error sentences and 175 valid sentences were present. Complete match of only 28.0% was obtained where tagging accuracy was 84.42%.

We also did FREVAL measure that deals with parser evaluation on fragments and the result was that if we increased the fragment size we get lower performance, however, Senna still has better performance as compared to other parsers.

For fragment size of one (1) (essentially PARSEVAL), out of 1500 fragments 1071 were matched with F1 score of 72.49%.

Increasing the fragment size to two (2), produced an F1 Score of 56.43% with 55.47% recall and 57.42% precision. More than two fragment size the F1 score was considerably lower to an F1 score of 43.83%.

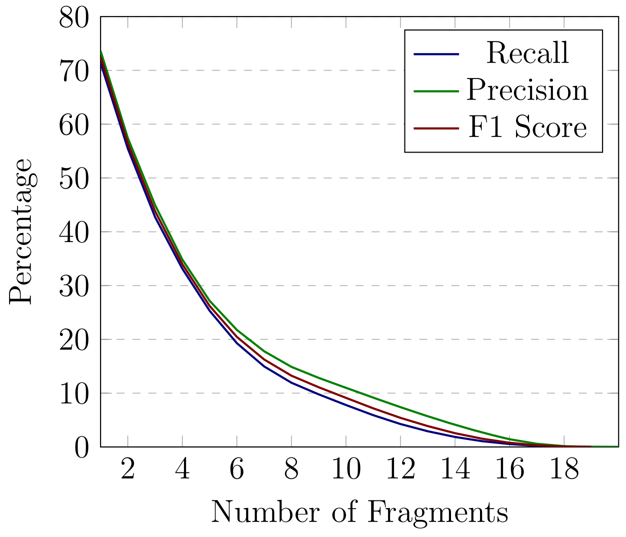

Figure 5.11: FREVAL performance evaluation of Senna on Parsing microblog text

We can see, the performance of parsing with increasing fragment size is a much difficult problem. Also we can note that Senna has higher precision for larger fragments as compared to the recall. For smaller fragments, the recall and precision are closely matched to the F1 score, making Senna a good choice to balance recall and precision.

Stanford Parser

We evaluated Stanford parser's functional performance on Twitter text. The performance is expectedly degraded from that of the regular written work text such as news posts, though, it was is not too bad.

We evaluated two trained models with Stanford parser, one being a standard model trained for Probabilistic Context-Free Grammar (PCFG) on English text and the other being the model trained for case-less English.

The test results were 67.29% F1 Score with 66.82% precision and 67.76% recall for the case-less model, where as for the standard mode the F1 score was 66.93% with 67.04% precision and 66.82% recall.

We can see from the results that both of the models performed almost equally well, but the case-less model slightly preferred recall over precision as compared to the standard model resulting in a very slight improvement on F1 score (0.36%).

For standard PCFG model, in total 250 sentences were evaluated, where 24 sentences were error sentences and 226 valid sentences were present. Complete match of only 20.34% was obtained. Tagging accuracy was 81.80%.

Where as, for case-less model, out of 250 sentences 24 were error sentences and 226 sentences were valid sentences. Complete match of 23.01% was found. Tagging accuracy was found to be 83.46%.

An interesting thing can be noted here, that the case-less model has a little better tagging accuracy of 83.46% as compared to the standard PCFG models tagging accuracy of 81.80%. This can be justified by the language used in Twitter where many people do not worry about using correct cases for words such as pronouns or acronyms.

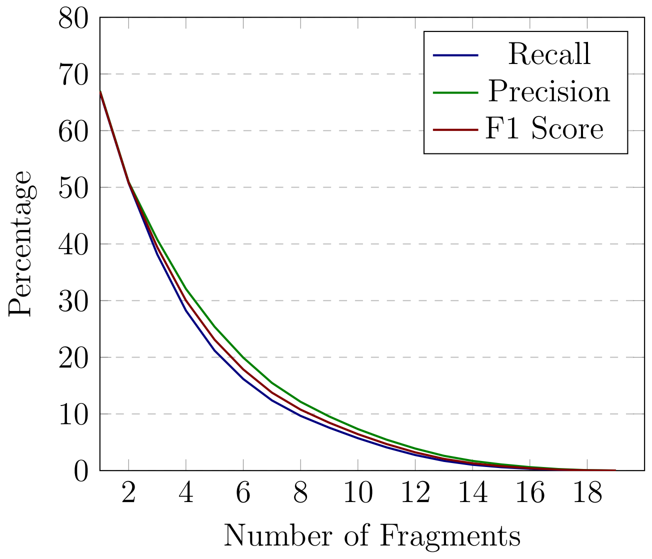

Figure 5.12: FREVAL performance evaluation of Stanford parser with Standard PCFG model on Parsing microblog text

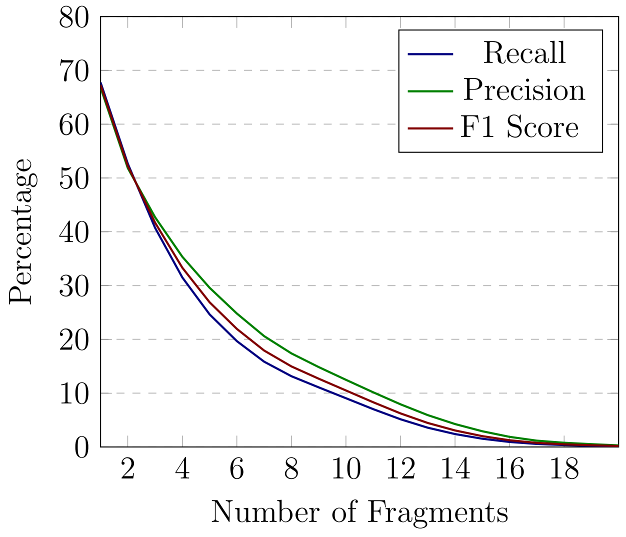

Figure 5.13: FREVAL performance evaluation of Stanford parser with PCFG-caseless model on parsing microblog text

Berkeley Parser

Berkeley parser performed as par with Stanford parser. We evaluated parsing the same Web2.0 test set with standard WSJ grammar provided with the Berkeley parser.

Out of 250 tweets, we got 247 as valid parses with only 3 error sentences. However there was zero complete match, i.e. none of the sentences had a perfect match with corresponding parse in the gold set.

Interesting is the performance of the Berkeley parser that, even though none of the sentences had a perfect match, its performance was comparable to most other parsers. We got 63.23% precision, with 68.65% recall given an F-1 score of 65.83% for PARSEVAL measures.

For FREVAL measures, i.e evaluating fragments with varying lengths, we found slightly slow degradation of performance. For example for fragments of size 5, the precision was 24.33% with recall of 24.22% giving an F-1 score of 24.27%.

POS-tagging accuracy was about 82.59%.

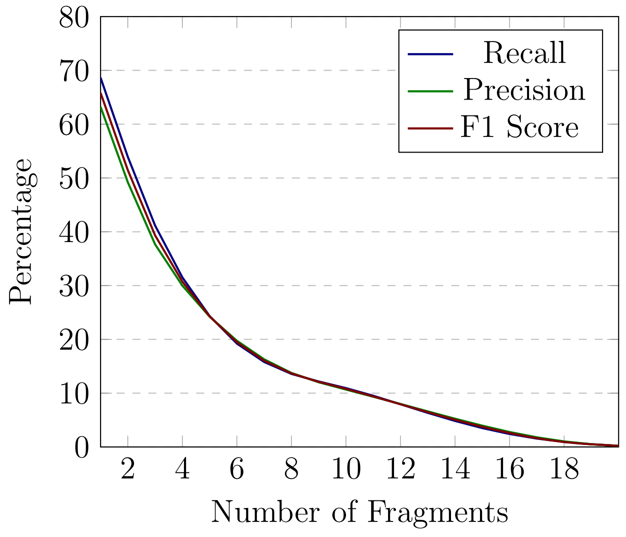

Figure 5.14: FREVAL performance evaluation of Berkeley parser on Parsing Microblog Text

BLLIP Parser (Charniak Reranking Parser)

The parsing Performance of BLLIP parser on microblog text was among the constituency parsers we evaluated.

Out of 250 tweets, 57 were error sentences with 197 valid sentences. A complete match was obtained 25.39%, while the tagging accuracy was only 79%.

The parsing PARSEVAL performance of BLLIP was 72.22% precision with 72.45% recall resulting in a F-1 score of 73.81%.

The FREVAL of larger fragments was also better with a fragment of length 5 giving 39.47% precision, 31.20% recall resulting in a F-1 score of 34.85%.

BLLIP parser performance is considerably better in even higher fragment sizes. The F1 score of 2.9% was found at fragment size of 20 whereas all other parsers had zero F-1 score.

Also we can see that BLLIP parser favours precision over recall for the microblog text parsing. The difference between precision and recall increases as the fragment sizes are increased.

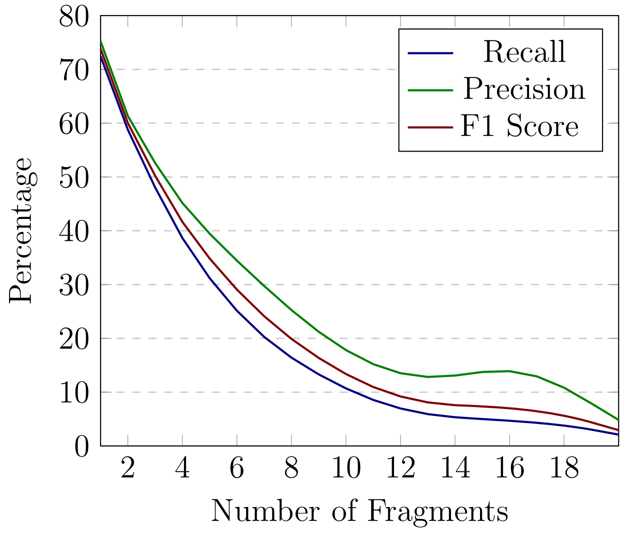

Figure 5.15: FREVAL performance evaluation of BLLIP parser on Parsing Microblog Text

,

Bikel-Collins Parser

The performance of the Bikel parser was quite good. This parser had 23.67% complete match with a tagging accuracy of 79.89%. The PARSEVAL precision was 75.55% and recall was 73.03% giving a PARSEVAL F1-Score of 74.27%.

The larger fragments evaluation was also reasonably high. For fragments of size 5, 40.37% precision with 32.49% recall giving an F1- Score of 36.00%.

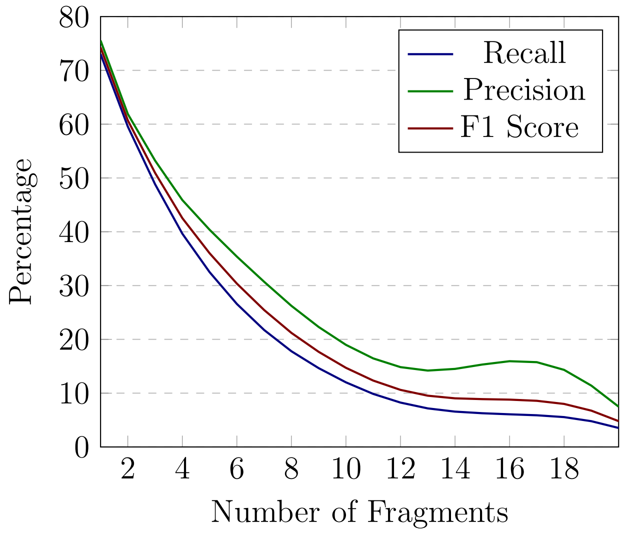

Figure 5.16: FREVAL performance evaluation of Bikel parser on parsing microblog text

Summary

Parser	**Precision**	**Recall**	**F1 score**
Berkeley parser-tokenized	61.84	67.21	64.41
Bikel-Collins-tokenized	**75.55**	**73.03**	**74.27**
BLLIP parser	75.22	72.45	73.81
Senna parser	73.61	71.40	72.49
Stanford parser-PCFG	67.04	66.82	66.93
Stanford parser-PCFG-caseless	66.82	67.76	67.29

Table 5.17: Functional performance of various constituency parsers – PARSEVAL measure

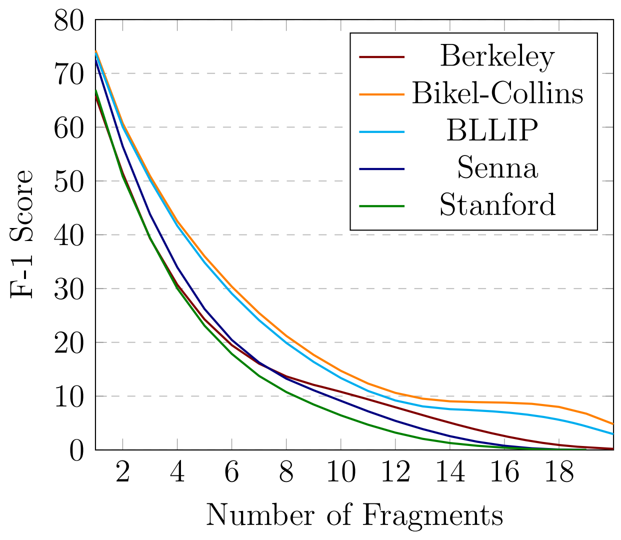

Figure 5.17: FREVAL performance evaluation of various parsers on parsing microblog text

5.2 Running-time Performance Evaluation

5.2.1 Tweebo Parser

Tweebo parser was not found to be fast. It took about 18.24 seconds on average in CPU time to parse 250 tweets taking about 73 milliseconds for each parse on the average. Also there was significant difference between the real time and CPU time, so it can be deduced that the Tweebo is IO bound parsing.

Time	Run 1	Run 2	Run 3	Average Run	Average per sentence
real	0m44.960s	0m42.466s	0m39.975s	42.467s	170ms
user	0m18.961s	0m17.303s	0m18.459s	18.241s	73ms
sys	0m5.875s	0m6.695s	0m5.537s	6.036s	24ms

Table 5.18: Running-time performance of Tweebo parser

Total average time to parse a tweet was 73 ms

Time for tokenized text was slightly greater i.e. 20.86 seconds for 250 tweets, on the average 83 milliseconds per tweet.

Time	Run 1	Run 2	Run 3	Average Run	Average per sentence
real	0m38.600s	0m40.523s	0m43.067s	40.73s	163ms
user	0m20.229s	0m20.965s	0m21.402s	20.865s	83ms
sys	0m5.331s	0m5.396s	0m6.324s	5.684s	23ms

Table 5.19: Running-time performance of Tweebo parser on tokenized text

Total average time to parse a tweet was 83 ms

5.2.2 MST Parser

MST parser time for tokenized text was reasonably fast, perhaps due to simplicity of the model (giving poor performance). It took 8.68 CPU seconds for 250 tweets, on the average 35 milliseconds per tweet.

Time	Run 1	Run 2	Run 3	Average Run	Average per sentence
real	0m9.102s	0m5.839s	0m5.827s	6.923s	28ms
user	0m8.848s	0m8.571s	0m8.608s	8.676s	35ms
sys	0m1.028s	0m0.705s	0m0.638s	0.79s	3ms

Table 5.20: Running-time performance of MST parser on tokenized text

Parsing time for MST parser on tokenized text was 35 ms

5.2.3 MALT Parser

MALT parser requires tokenized text. MALT parser was quite good in terms of running-time performance in parsing. The average time to parse 250 sentences text by MALT parser trained on PTB-WSJ was 4.7s, giving a mere 19ms time per sentence in terms of CPU time. We can also note that the real time is less than CPU time, signifying that MALT utilizes multi-thread and/or multi core processing facilities by default.

Time	Run 1	Run 2	Run 3	Average Run	Average per sentence
real	0m4.783s	0m3.085s	0m3.041s	3.636s	15ms
user	0m4.922s	0m4.716s	0m4.647s	4.762s	19ms
sys	0m0.549s	0m0.454s	0m0.530s	0.511s	2ms

Table 5.21: Running-time performance of MALT parser trained on WSJ

MALT parser, trained on Web2.0 TwitterDev dataset, was quite fast in parsing the Web2.0 TwitterTest dataset. The average CPU time it took to parse 250 sentences text (out of three runs) was only 3.8 seconds, giving 15 ms average per sentence.

Time	Run 1	Run 2	Run 3	Average Run	Average per sentence
real	0m2.088s	0m2.055s	0m2.045s	2.063s	8ms
user	0m3.805s	0m3.812s	0m3.807s	3.808s	15ms
sys	0m0.410s	0m0.430s	0m0.425s	0.422s	2ms

Table 5.22: Running-time performance of MALT parser trained on TwitterDev

Parsing time for MALT parser was 15 ms per tweet on the average.

5.2.4 Senna Parser

As for running-time performance, Senna was among the faster parsers. It was quite fast. Senna only took about 4.3 seconds to parse 250 sentences on the average out of three runs which averages to about 17 milliseconds for each tweet.

We can also note that the real time was more than the CPU time, which shows that Senna does not use multi threading or multi-core facilities available on default.

Time	Run 1	Run 2	Run 3	Average Run	Average per sentence
real	0m8.424s	0m5.586s	0m4.595s	6.202s	25ms
user	0m4.455s	0m4.174s	0m4.428s	4.352s	17ms
sys	0m0.164s	0m0.132s	0m0.123s	0.14s	1ms

Table 5.23: Running-time performance of Senna parser

The conversion time from Senna output to Dependency tree was significant to Senna parsing time. It took about 3.3 seconds of CPU time to convert the output to dependency tree on the average, giving 13 ms of CPU.

Time	Run 1	Run 2	Run 3	Average Run	Average per sentence
real	0m2.246s	0m4.476s	0m1.659s	2.794s	11ms
user	0m3.361s	0m3.520s	0m3.137s	3.339s	13ms
sys	0m1.040s	0m0.897s	0m0.759s	0.899s	4ms

Table 5.24: Running-time of conversions to dependency trees for Senna parser

So in cumulative, Senna takes on the average about 30 ms to parse a single tweet to dependency.

The time taken for tokenized input did not change much for Senna; the average CPU time taken was 4.4 seconds for tokenized input, which averages to about 18 milliseconds per tweet. We observed in these runs that the real time was almost the same as the CPU time, showing that although senna does not use multi-threading, it is quite efficient in terms of IO processes.

Time	Run 1	Run 2	Run 3	Average Run	Average per sentence
real	0m4.747s	0m4.649s	0m4.452s	4.616s	18ms
user	0m4.428s	0m4.502s	0m4.264s	4.398s	18ms
sys	0m0.165s	0m0.106s	0m0.123s	0.131s	1ms

Table 5.25: Running-time performance of Senna parser on tokenized text

The total time for Senna to parse tokenized text to dependency tree structure is about 31 ms per tweet.

5.2.5 Stanford Parser

The running-time performance of Stanford parser was found to be quite low. It took about 26 seconds in average to parse 250 sentences, averaging to 105 milliseconds per tweet, when we employed PCFG language model in parsing using the Stanford parser. We can see that Stanford CPU time was slightly more than the real time, though not much, we can infer that it employs multi-threading.

Time	Run 1	Run 2	Run 3	Average Run	Average per sentence
real	0m25.083s	0m23.334s	0m23.322s	23.913s	96ms
user	0m26.200s	0m25.892s	0m26.345s	26.146s	105ms
sys	0m0.841s	0m0.732s	0m0.672s	0.748s	3ms

Table 5.26: Running-time performance of Stanford parser using PCFG language model

The time to convert the Stanford PCFG output trees to dependency trees is also significant. It took 14 ms of CPU time per tweet to convert to dependency tree.

Time	Run 1	Run 2	Run 3	Average Run	Average per sentence
real	0m1.685s	0m2.189s	0m1.854s	1.909s	8ms
user	0m3.298s	0m3.802s	0m3.759s	3.62s	14ms
sys	0m0.674s	0m0.875s	0m0.742s	0.764s	3ms

Table 5.27: Running-time of conversion to dependency trees for Stanford parser using PCFG language model

The total time taken to parse a tweet to dependency structure is 121 ms.

Time taken to parse pre-tokenized text was the same, with 26 seconds to parse 250 sentences on the average of three runs, averaging 104 milliseconds per sentence.

Time	Run 1	Run 2	Run 3	Average Run	Average per sentence
real	0m23.689s	0m23.675s	0m23.607s	23.657s	95ms
user	0m25.934s	0m25.670s	0m26.267s	25.957s	104ms
sys	0m0.802s	0m0.892s	0m0.839s	0.844s	3ms

Table 5.28: Running-time performance of Stanford parser using PCFG language model on tokenized text

Time taken to convert constituency trees generated by the Stanford parser using PCFG language model on pre-tokenized text was 15 ms per sentences.

Time	Run 1	Run 2	Run 3	Average Run	Average per sentence
real	0m4.168s	0m1.976s	0m1.726s	2.623s	10ms
user	0m3.250s	0m3.661s	0m4.032s	3.648s	15ms
sys	0m1.226s	0m0.745s	0m0.343s	0.771s	3ms

Table 5.29: Running-time of conversion to dependency trees for Stanford parser using PCFG language model on tokenized text

Hence total average time to parse tweets to dependency trees by Stanford parser using PCFG language model on pre-tokenized text was 119 ms

Running the Stanford parser with PCFG-case-less language model was slightly slower than the PCFG language model. It took 27 seconds to parse 250 sentences on the average of three runs, giving an average of 107 milliseconds per sentence.

Time	Run 1	Run 2	Run 3	Average Run	Average per sentence
real	0m23.925s	0m25.280s	0m24.280s	24.495s	98ms
user	0m26.744s	0m27.908s	0m26.527s	27.06s	108ms
sys	0m0.768s	0m0.764s	0m0.827s	0.786s	3ms

Table 5.30: Running-time performance of Stanford parser using PCFG-caseless language model

CPU time to convert constituency trees generated by Stanford parser using PCFG-case-less language mode was 14 ms per sentences on the average.

Time	Run 1	Run 2	Run 3	Average Run	Average per sentence
real	0m1.584s	0m1.970s	0m1.721s	1.758s	7ms
user	0m3.272s	0m3.146s	0m4.000s	3.473s	14ms
sys	0m0.704s	0m0.829s	0m0.408s	0.647s	3ms

Table 5.31: Running-time of conversion to dependency trees for Stanford parser using PCFG-caseless language model

Total average time to parse sentences by Stanford parser PCFG-case-less was 122 ms

Running time of Stanford parser with PCFG-case-less language on pre-tokenized input was about the same, 27.3 seconds to parse 250 sentences on the average of three runs, averaging to 109 milliseconds per sentence.

Time	Run 1	Run 2	Run 3	Average Run	Average per sentence
real	0m24.703s	0m24.290s	0m24.680s	24.558s	98ms
user	0m27.016s	0m27.094s	0m27.713s	27.274s	109ms
sys	0m0.873s	0m0.806s	0m0.685s	0.788s	3ms

Table 5.32: Running-time performance of Stanford parser using PCFG-caseless language model on tokenized text

Conversion time in terms of CPU from constituency trees generated by Stanford PCFG-caseless model on pre-tokenized text was 14 ms.

Time	Run 1	Run 2	Run 3	Average Run	Average per sentence
real	0m1.829s	0m1.555s	0m1.699s	1.694s	7ms
user	0m3.743s	0m3.256s	0m3.780s	3.593s	14ms
sys	0m0.769s	0m0.619s	0m0.649s	0.679s	3ms

Table 5.33: Running-time of conversion to dependency trees for Stanford parser using PCFG-caseless language model on tokenized text

Total average CPU time per sentence by Stanford parse using PCFG caseless model on pre-tokenized text was 123 ms

5.2.6 Berkeley Parser

The running time for Berkeley parser to parse Web2.0 TwitterTest dataset of 250 tweets was quite high. It took on average (from three runs) a considerable 36 seconds in terms of CPU time, while the real time as even larger to 65 seconds. On the average Berkeley parsers took 145 ms to parse a tweet. We can also see that CPU time is much lesser than real time, signifying IO boundedness of the parser. Though this increase in real time can be seen in only one run (Run 2), suggesting this might have been some system factor at run time.

Time	Run 1	Run 2	Run 3	Average Run	Average per sentence
real	0m56.367s	1m24.681s	0m54.209s	65.086s	260ms
user	0m36.305s	0m35.718s	0m36.449s	36.157s	145ms
sys	0m1.013s	0m1.211s	0m1.178s	1.134s	5ms

Table 5.34: Running-time performance of Berkeley parser

The conversion time for Berkeley parser from constituency trees to dependency trees was 14 ms on the average in terms of CPU time.

Time	Run 1	Run 2	Run 3	Average Run	Average per sentence
real	0m1.698s	0m3.096s	0m1.829s	2.208s	9ms
user	0m2.739s	0m4.379s	0m3.189s	3.436s	14ms
sys	0m0.983s	0m0.768s	0m0.522s	0.758s	3ms

Table 5.35: Running-time of conversion to dependency trees for Berkeley parser

The total average time to parse a sentence by Berkeley parser was found to be 159 ms

Running Berkeley parser to parse pre-tokenized text takes about the same time as non-tokenized text. It took on the average 37.5 seconds to parse the 250 sentence dataset (in three runs), averaging to 150 ms per sentence.

Time	Run 1	Run 2	Run 3	Average Run	Average per sentence
real	0m35.007s	0m37.266s	0m40.850s	37.708s	151ms
user	0m36.586s	0m38.513s	0m37.395s	37.498s	150ms
sys	0m1.069s	0m1.012s	0m1.304s	1.128s	5ms

Table 5.36: Running-time performance of Berkeley parser on tokenized text

It took 15 ms on the average per sentence to convert the output of Berkeley parser on pre-tokenized text to Dependency trees.

Time	Run 1	Run 2	Run 3	Average Run	Average per sentence
real	0m2.990s	0m2.761s	0m2.710s	2.82s	11ms
user	0m3.114s	0m3.952s	0m4.344s	3.803s	15ms
sys	0m1.209s	0m1.000s	0m0.582s	0.93s	4ms

Table 5.37: Running-time of conversion to dependency trees for Berkeley parser on tokenized text

The total time to parse a sentence to Dependency tree was 165 ms on the average.

5.2.7 BLLIP Parser

BLLIP-Charniak parser is among the slower parsers. To parse 250 sentence Web2.0 TwitterTest it took about 90 seconds in terms of CPU time to parse the complete dataset on the average out of three runs. The average time to parse a sentence was quite high, about 364 ms per sentence in terms of CPU time. We also note that the CPU time was slightly greater than the real time, suggesting that it uses multi-threading at least for some part of parsing.

Time	Run 1	Run 2	Run 3	Average Run	Average per sentence
real	1m30.789s	1m22.255s	1m21.951s	84.998s	340ms
user	1m31.178s	1m31.087s	1m30.625s	90.963s	364ms
sys	0m2.716s	0m3.402s	0m2.810s	2.976s	12ms

Table 5.38: Running-time performance of BLLIP parser

The time to convert constituency trees generated by BLLIP parser was 14 ms on the average in terms of CPU time.

Time	Run 1	Run 2	Run 3	Average Run	Average per sentence
real	0m2.035s	0m2.678s	0m1.669s	2.127s	9ms
user	0m3.045s	0m3.938s	0m3.590s	3.524s	14ms
sys	0m1.001s	0m1.110s	0m0.434s	0.848s	3ms

Table 5.39: Running-time of conversion to dependency trees for BLLIP parser

Thus the total time to parse a sentence using BLLIP parser was found to be 380 ms on the average.

The parse time to parse pre-tokenized text was about the same, only slightly better in two runs out of three, so it might be just system factor. The time it took to parse the dataset on the average was 87.6 seconds averaging to 350 ms.

Time	Run 1	Run 2	Run 3	Average Run	Average per sentence
real	1m16.720s	1m16.529s	1m24.177s	79.142s	317ms
user	1m24.464s	1m24.537s	1m33.847s	87.616s	350ms
sys	0m3.108s	0m2.957s	0m3.469s	3.178s	13ms

Table 5.40: Running-time performance of BLLIP parser on tokenized text

It took 15 ms on the average per tweet to convert the constituency trees generated by BLLIP parser when running on pre-tokenized text.

Time	Run 1	Run 2	Run 3	Average Run	Average per sentence
real	0m2.755s	0m1.750s	0m2.081s	2.195s	9ms
user	0m4.192s	0m3.307s	0m3.393s	3.631s	15ms
sys	0m0.717s	0m0.978s	0m0.877s	0.857s	3ms

Table 5.41: Running-time of conversion to dependency trees for BLLIP parser on tokenized text

The Total time to parse a tweet to dependency trees by BLLIP parser when run on pre-tokenized text was found to be 365 ms

5.2.8 Bikel-Collins Parser

Bikel-Collins parser was the slowest among the parsers we evaluate, slower by a multiple of second slowest parser and two order of magnitude slower of the fastest parser. It took a whopping 231 seconds to parse the dataset of 250 tweets in terms of CPU time, averaging to 925 ms to parse a single tweet. We also see the Bikel-Collins parser does not employ threading by default.

Time	Run 1	Run 2	Run 3	Average Run	Average per sentence
real	3m54.312s	3m51.216s	3m51.799s	232.442s	930ms
user	3m54.445s	3m49.614s	3m50.055s	231.371s	925ms
sys	0m1.388s	0m1.262s	0m1.100s	1.25s	5ms

Table 5.42: Running-time performance of Bikel-Collins parser on tokenized text

To convert the phrase-structure (constituency trees) generated by Bikel-Collins parser to dependency trees it took 13 ms per tweet on the average.

Time	Run 1	Run 2	Run 3	Average Run	Average per sentence
real	0m1.674s	0m1.486s	0m2.609s	1.923s	8ms
user	0m3.129s	0m2.858s	0m3.912s	3.3s	13ms
sys	0m0.570s	0m0.579s	0m0.602s	0.584s	2ms

Table 5.43: Running-time of conversion to dependency trees for Bikel-Collins parser on tokenized text

Thus the time taken by Bikel-Collins parser to parse a tweet to dependency tree was found to be 938 ms on the average.

5.2.9 Summary

We see that there is significant variation in parsing time of various parsers. The fastest was MALT parser, with Senna and MST parser coming close second. And the slowest was Bikel-Collins parser.

We can see that this time difference is significant enough to consider parsing time for time sensitive application in choice of a parser, about two orders of magnitude.

Parser	**CPU Time (ms)**
MALT parser – tokenized text	**19**
Senna parser	30
Senna parser – tokenized text	31
MST parser – tokenized text	35
Tweebo parser	73
Tweebo parser – tokenized text	83
Stanford parser-PCFG – tokenized text	119
Stanford parser-PCFG	121
Stanford parser-PCFG-caseless	122
Stanford parser-PCFG-caseless – tokenized text	123
Berkeley parser	159
Berkeley parser – tokenized text	165
BLLIP parser – tokenized text	365
BLLIP parser[2]	380
Bikel-Collins parser – tokenized text	938

Table 5.44: Running-time performance of various parsers - Average parse time for dependency trees per tweet

5.3 Summary

We found out that there are two contenders for parsing microblog text: one is Senna parser and the other is BLLIP parser. BLLIP was the best at functional performance while its running-time performance was not too degraded. And Senna was the best at running-time performance while still keeping up with other parsers on functional performance.

One thing we can also note about Senna and BLLIP parsers, is that the functional performance of BLLIP is quite good in both cases where we provide a pre-tokenized text or not. So the tokenizer of the BLLIP parser works better than the Senna tokenizer. While if we use Senna for non-tokenized text, its performance degrades.

MALT parser was fastest but its functional performance was quite poor. Bikel-Collins

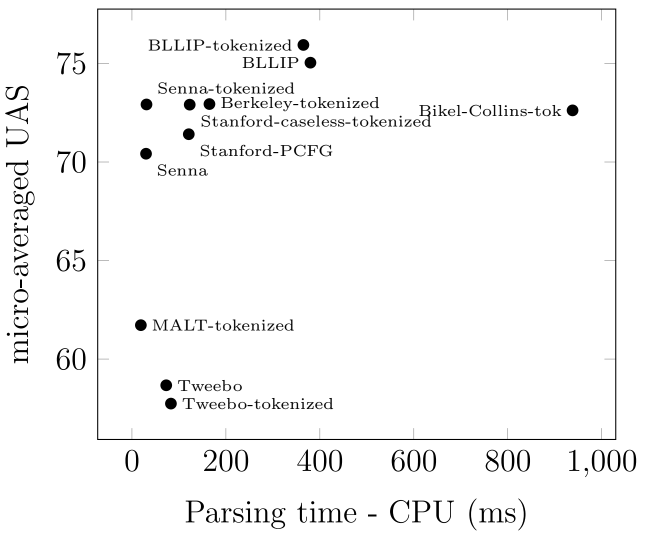

Figure 5.18: Functional performance (dependency parse) vs. Running-time performance

parser was high in performance but its running-time performance was extremely poor.

Parser	**micro-avg UAS**	**macro-avg UAS**	**CPU Time (ms)**
Berkeley parser – tokenized text	72.94	72.46	165
Bikel-Collins parser	72.62	73.12	938
BLLIP [3] parser	75.04	76.49	380
BLLIP parser – tokenized text	**75.94**	**77.10**	365
MALT parser – tokenized text	61.72	60.01	**19**
MST parser – tokenized text	14.67	15.72	35
Senna parser	70.42	71.23	30
Senna parser – tokenized text	72.92	75.91	31
Stanford parser-PCFG	71.41	71.27	121
Stanford parser-PCFG – tokenized text	71.00	71.28	119
Stanford parser-PCFG-caseless	72.71	73.07	122
Stanford parser-PCFG-caseless – tokenized text	72.91	73.59	123
Tweebo parser	58.67	58.75	73
Tweebo parser – tokenized text	57.74	57.66	83

Table 5.45: Functional and running-time performance of various parsers - micro-averaged and macro-averaged UAS measures and average parse time for dependency trees per tweet

Ignoring MST parser, which was extremely poor, Tweebo parser was the worst performer, for both pre-tokenized and non-tokenized text. Stanford parser is also a good contender as its performance, both functional and temporal, is close to the best performing parsers. We also note that correct tokenization has significant effect on functional performance of most of the parsers.

Chapter 6

Conclusion and Future Work

6.1 Conclusion

We found out that there are two contenders for parsing microblog text, one is Senna parser and the other is BLLIP parser. BLLIP was best at functional performance while its running-time performance was not too degraded. And Senna was best at running-time performance while still keeping up with other parsers on functional performance.

Comparing all the parsers with or without pre-tokenized text (for those parser that do take non-tokenized text), we see that BLLIP parser (also known as Charniak reranking parser) fares the best in terms of both micro-averaged UAS and macro-averaged UAS. BLLIP parser gave 75.94% micro-averaged UAS over all words in the tweets and gave 77.10 macro-averaged UAS over all sentences in the tweet dataset. This result was obtained by providing pre-tokenized sentences.

We see that Senna parser was the second best parser in terms of its macro-average performance on tokenized text giving 73% micro-average UAS and 76% macro-average UAS.

After Senna parser, Stanford, Berkeley and Bikel-Collins all parsers fared almost the same on around 72% micro and 73% macro-average.

MALT parser was fastest but its functional performance was quite poor. Bikel-Collins parser was high in performance but its running-time performance was extremely poor.

One important thing we found that dependency parsers as a category did not fare well as a group against constituency parsers, even though we used correctness of dependency parse trees as evaluation criteria even for constituency parsers, by converting the constituency trees into dependency trees. The functional performance of constituency parsers was greater. Even the running-time performance of constituency parsers, even including the additional time taken to convert constituency trees to dependency trees, was much less than that of the dependency parsers.

We have found that microblog text differs considerably from the formally written text. Parsing of microblog text is significantly different problem from parsing of regular formally written text. The performance of all of the parsers is significantly reduced when it comes to parsing microblog text.

Also the running-time performance of the parsers matters substantially more for microblog text as the amount of the microblog text is growing exponentially and the significance of its value is very short lived. The processing methods that need to parse the microblog text must be very high performing to be valued for real world applications to parse microblog text.

6.1.1 Findings

Based on our evaluations following is a list of findings.

- Senna (a neural network parser) worked best considering both the functional and running-time performance.
- Correct tokenization is important for functional performance.
- Constituency parsers fare better than dependency parsers in general.

6.2 Future Work

There are great possibilities of research work on parsing of microblog text.

Following is a synopsis of some of the suggested possible future work.

6.2.1 Treating Microblog as Verbal Communication

There are multiple things that can be done to continue this work. One of them is that Microblog text may be closer to verbal communication as compared to written communication, so it would be worthwhile to look into a different training data than Wall Street Journals written news articles, such as ATIS or switchboard corpus in PTB. It would be interesting to find out how various parsers perform on microblog text when they are trained on switchboard corpus or ATIS corpus in PTB.

6.2.2 Up training

One approach, worthwhile taking up, is up-training of the parsers with additional tweets that are collected with one of the best performing parsers. For example, we saw that Bikel-Collins parser had best FREVAL F-1 score, but its running-time performance was extremely poor. What can be done is to generate a significant amount of parsed microblog text, parsed with Bikel-Collins parser, and then use it to train other parser and to evaluate if this increases their functional performance.

6.2.3 Unsupervised Parsing

The other research direction that we felt valuable is to look into unsupervised parsing, such as suggested by Bod [7].

Since the microblog text is huge and is being generated at a tremendous rate, it can be assumed that creating a significant sized corpus of hand parsed text is quite a daunting task. Hence, approaches in unsupervised parsing might be beneficial to look at.

6.2.4 Evaluating Other Neural Network Based Dependency Parsers

Stanford parser has released a neural network dependency parser [13]. Since Senna, the one of the best performing parser is a neural network parser, it would be worthwhile to evaluate Stanford Neural network parser for microblog text.

Another parser that uses CRF Parsing using Neural net approaches [21] is worth the effort to evaluate.

Bibliography

[1] Steven Abney, S Flickenger, Claudia Gdaniec, C Grishman, Philip Harrison, Donald Hindle, Robert Ingria, Frederick Jelinek, Judith Klavans, Mark Liberman, et al. Procedure for quantitatively comparing the syntactic coverage of English grammars. In *Proceedings of the workshop on Speech and Natural Language*, pages 306–311. Association for Computational Linguistics, 1991.

[2] Chris Dyer Jacob Eisenstein Jeffrey Flanigan Kevin Gimpel Michael Heilman Lingpeng Kong Daniel Mills Brendan O'Connor Olutobi Owoputi Nathan Schneider Noah Smith Swabha Swayamdipta Archna Bhatia, Dipanjan Das and Dani Yogatama. Tweeboparser. [Online; accessed 22-January-2015].

[3] Jason Baldrige and Ryan McDonald. MSTParser-source. [Online; access 2-February-2015].

[4] Joost Bastings and Khalil Sima'an. All Fragments Count in Parser Evaluation. In *Proceedings of the Ninth International Conference on Language Resources and Evaluation (LREC'14)*, Reykjavik, Iceland, may 2014. European Language Resources Association (ELRA).

[5] Daniel M. Bikel. Software. [Online; accessed 10-January-2015].

[6] Daniel M Bikel. Intricacies of Collins' parsing model. *Computational Linguistics*, 30(4):479–511, 2004.

[7] Rens Bod. Unsupervised parsing with U-DOP. In *Proceedings of the Tenth Conference on Computational Natural Language Learning*, pages 85–92. Association for Computational Linguistics, 2006.

[8] Danah Boyd, Scott Golder, and Gilad Lotan. Tweet, tweet, retweet: Conversational aspects of retweeting on Twitter. In *System Sciences (HICSS), 2010 43rd Hawaii International Conference on*, pages 1–10. IEEE, 2010.

[9] Sabine Buchholz and Erwin Marsi. CoNLL-X shared task on multilingual dependency parsing. In *Proceedings of the Tenth Conference on Computational Natural Language Learning*, pages 149–164. Association for Computational Linguistics, 2006.

[10] Eugene Charniak. Eugene Charniak-Software. [Online; accessed 2-February-2015].

[11] Eugene Charniak. A maximum-entropy-inspired parser. In *Proceedings of the 1st North American chapter of the Association for Computational Linguistics conference*, pages 132–139. Association for Computational Linguistics, 2000.

[12] Eugene Charniak and Mark Johnson. Coarse-to-fine n-best parsing and MaxEnt discriminative reranking. In *Proceedings of the 43rd Annual Meeting on Association for Computational Linguistics*, pages 173–180. Association for Computational Linguistics, 2005.

[13] Danqi Chen and Christopher D. Manning. A fast and accurate dependency parser using neural networks. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, volume 1, pages 740–750, 2014.

[14] Michael Collins. Michael Collins - Software and Data Sets. [Online; accessed 2-February-2015].

[15] Michael John Collins. A new statistical parser based on bigram lexical dependencies. In *Proceedings of the 34th annual meeting on Association for Computational Linguistics*, pages 184–191. Association for Computational Linguistics, 1996.

[16] Ronan Collobert. Deep learning for efficient discriminative parsing. In *International Conference on Artificial Intelligence and Statistics*, number EPFL-CONF-192374, 2011.

[17] Ronan Collobert et al. Natural language processing (almost) from scratch. *The Journal of Machine Learning Research 12*, pages 2493–2537, 2011.

[18] Ronan Collobert and Jason Weston. A unified architecture for natural language processing: Deep neural networks with multitask learning. In *Proceedings of the 25th international conference on Machine learning*, pages 160–167. ACM, 2008.

[19] Koby Crammer and Yoram Singer. Ultraconservative online algorithms for multiclass problems. *The Journal of Machine Learning Research*, 3:951–991, 2003.

[20] Leon Derczynski, Diana Maynard, Niraj Aswani, and Kalina Bontcheva. Microblog-genre noise and impact on semantic annotation accuracy. In *Proceedings of the 24th ACM Conference on Hypertext and Social Media*, pages 21–30. ACM, 2013.

[21] Greg Durrett and Dan Klein. Neural CRF Parsing. In *Proceedings of the 53rd Annual Meeting on Association for Computational Linguistics.* Association for Computational Linguistics, 2015.

[22] Jennifer Foster, Özlem Çetinoglu, Joachim Wagner, Joseph Le Roux, Stephen Hogan, Joakim Nivre, Deirdre Hogan, Josef Van Genabith, et al. # hardtoparse: POS Tagging and Parsing the Twitterverse. In *proceedings of the Workshop On Analyzing Microtext (AAAI 2011)*, pages 20–25, 2011.

[23] Jennifer Foster, Ozlem Cetinoglu, Joachim Wagner, Joseph Le Roux, Joakim Nivre, Deirdre Hogan, and Josef VanGenabith. From news to comment: Resources and benchmarks for parsing the language of web 2.0. 2011.

[24] Guo Fu-liang and Zhou Gang. Research on micro-blog sentiment orientation analysis based on improved dependency parsing. In *Consumer Electronics, Communications and Networks (CECNet), 2013 3rd International Conference on*, pages 546–550. IEEE, 2013.

[25] Daniel Gildea. Corpus variation and parser performance. In *Proceedings of the 2001 Conference on Empirical Methods in Natural Language Processing*, pages 167–202, 2001.

[26] The Berkeley NLP Group. Berkeley parser. [Online; accessed 15-January-2015].

[27] The Stanford Natural Language Processing Group. The Stanford Parser: A statistical parser. Online; accessed 18-January-2015.

[28] Johan Hall and Joakim Nivre. A dependency-driven parser for German dependency and constituency representations. In *Proceedings of the Workshop on Parsing German*, pages 47–54. Association for Computational Linguistics, 2008.

[29] James Henderson. Discriminative training of a neural network statistical parser. In *Proceedings of the 42nd Annual Meeting on Association for Computational Linguistics*, page 95. Association for Computational Linguistics, 2004.

[30] F Hillebrand. The Creation of the SMS Concept from Mid-1984 to Early 1987. *Short Message Service (SMS): The Creation of Personal Global Text Messaging*, pages 23–44.

[31] Lichan Hong, Gregorio Convertino, and Ed H Chi. Language Matters In Twitter: A Large Scale Study. In *ICWSM*, 2011.

[32] Jens Nilsson Johan Hall and Joakim Nivre. Maltparser. [Online; accessed 20-January-2015].

[33] Richard Johansson and Pierre Nugues. Extended constituent-to-dependency conversion for English. In *16th Nordic Conference of Computational Linguistics*, pages 105–112. University of Tartu, 2007.

[34] Tibor Kiss and Jan Strunk. Unsupervised multilingual sentence boundary detection. *Computational Linguistics*, 32(4):485–525, 2006.

[35] Dan Klein and Christopher D Manning. Accurate unlexicalized parsing. In *Proceedings of the 41st Annual Meeting on Association for Computational Linguistics-Volume 1*, pages 423–430. Association for Computational Linguistics, 2003.

[36] Lingpeng Kong, Nathan Schneider, Swabha Swayamdipta, Archna Bhatia, Chris Dyer, and Noah A Smith. A dependency parser for tweets. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing, Doha, Qatar, to appear*, 2014.

[37] Kübler, Sandra and McDonald, Ryan and Nivre, Joakim. Dependency parsing. *Synthesis Lectures on Human Language Technologies*, 1(1):1–127, 2009.

[38] Dekang Lin. A dependency-based method for evaluating broad-coverage parsers. *Natural Language Engineering*, 4(02):97–114, 1998.

[39] Ryan McDonald, Kevin Lerman, and Fernando Pereira. Multilingual dependency analysis with a two-stage discriminative parser. In *Proceedings of the Tenth Conference on Computational Natural Language Learning*, pages 216–220. Association for Computational Linguistics, 2006.

[40] Jens Nilsson and Joakim Nivre. MaltEval: an Evaluation and Visualization Tool for Dependency Parsing. In *LREC*, 2008.

[41] Joakim Nivre. An efficient algorithm for projective dependency parsing. In *Proceedings of the 8th International Workshop on Parsing Technologies (IWPT*. Citeseer, 2003.

[42] Chomsky Noam. Syntactic structures. *The Hague: Mouton*, 1957.

[43] Slav Petrov, Leon Barrett, Romain Thibaux, and Dan Klein. Learning accurate, compact, and interpretable tree annotation. In *Proceedings of the 21st International Conference on Computational Linguistics and the 44th annual meeting of the Association for Computational Linguistics*, pages 433–440. Association for Computational Linguistics, 2006.

[44] Slav Petrov and Dan Klein. Improved Inference for Unlexicalized Parsing. In *HLT-NAACL*, pages 404–411. Citeseer, 2007.

[45] Leon Bottou Michael Karlen Koray Kavukcuoglu Pavel Kuksa Ronan Collobert, Jason Weston. Senna. [Online; accessed 22-January-2015].

Appendix A

Tagsets

A.1 Comparison of Various Tagsets

Category	**Examples**	**Claws c5**	**Brown**	**Penn**
Adjective, ordinal number	sixth, 72nd, last	ORD	OD	JJ
Adjective, comparative	happier, worse	AJC	JJR	JJR
Adjective, superlative	happiest, worst	AJS	JJT	JJS
Adjective, superlative, semantically	chief, top	AJ0	JJS	JJ
Adjective, cardinal number	3, fifteen	CRD	CD	CD
Adjective, cardinal number, one	One	PNI	CD	CD
Adverb	often, particularly	AV0	RB	RB
Adverb, negative	not, nt	XX0	*	RB
Adverb, comparative	Faster	AV0	RBR	RBR
Adverb, superlative	Fastest	AV0	RBT	RBS
Adverb, particle	up, off, out	AVP	RP	RP
Adverb, question	when, how, why	AVQ	WRB	WRB
Adverb, degree &question	how, however	AVQ	WQL	WRB
Adverb, degree	very, so, too	AV0	QL	RB
Adverb, degree, postposed	enough, indeed	AV0	QLP	RB
Adverb, nominal	here, there, now	AV0	RN	RB
Conjunction, coordination	and, or	CJC	CC	CC
Conjunction, subordinating	although, when	CJS	CS	IN
Conjunction, complementizer that	That	CJT	CS	IN
Determiner	this, each, other	DT0	DT	DT
Determiner, pronoun	any, some	DT0	DTI	DT
Determiner, pronoun, plural	these, those	DT0	DTS	DT
Determiner, prequalifier	Quite	DT0	ABL	PDT
Determiner, prequantifer	all, half	DT0	ABN	PDT
Determiner, pronoun or double conj.	Both	DT0	ABX	DT(CC)
Determiner, pronoun or double conj.	either, neither	DT0	DTX	DT(CC)
Determiner, article	the, a, an	AT0	AT	DT

Determiner, post determiner	many, same	DT0	AP	JJ
Determiner, possessive, second	mine, yours	DPS	PP$$	PRP
Determiner, question	which, whatever	DTQ	WDT	WDT
Determiner, possessive &question	Whose	DTQ	WP$	WP$
Noun	aircraft, data	NN0	NN	NN
Noun, singular	woman, book	NN1	NN	NN
Noun, plural	women, books	NN2	NNS	NNS
Noun, proper, singular	London, Michael	NP0	NP	NNP
Noun, proper, plural	Australians, Methodists	NP0	NPS	NNPS
Noun, adverbial	tomorrow, home	NN0	NR	NN
Noun, adverbial, plural	Sundays, weekdays	NN2	NRS	NNS
Pronoun, nominal (indefinite)	none, everything, one	PNI	PN	NN
Pronoun, personal, subject	you, we	PNP	PPSS	PRP
Pronoun, personal, subject, 3SG	she, he, it	PNP	PPS	PRP
Pronoun, personal object	you, them, me	PNP	PPO	PRP
Pronoun, reflexive	herself, myself	PNX	PPL	PRP
Pronoun, reflexive, plural	themselves, ourselves	PNX	PPLS	PRP
Pronoun, question, subject	who, whoever	PNQ	WPS	WP
Pronoun, question, object	who, whoever	PNQ	WPO	WP
Pronoun, existential there	There	EX0	EX	EX
Verb, base present form(not infinitive)	take, live	VVB	VB	VBP
Verb, infinitive	take, live	VVI	VB	VB
Verb, past tense	took, lived	VVD	VBD	VBD
Verb, present participle	taking, living	VVG	VBG	VBG
Verb, past/passiveparticiple	taken, lived	VVN	VBN	VBN
Verb, present 3SG -s form	takes, lives	VVZ	VBZ	VBZ
Verb, auxiliary do, base	Do	VDB	DO	VBP
Verb, auxiliary do, infinitive	Do	VDB	DO	VB
Verb, auxiliary do, past	Did	VDD	DOD	VBD
Verb, auxiliary do, present participle	Doing	VDG	VBG	VBG
Verb, auxiliary do, past participle	Done	VDN	VBN	VBN
Verb, auxiliary do, present 3SG	Does	VDZ	DOZ	VBZ
Verb, auxiliaryhave, base	Have	VHB	HV	VBP
Verb, auxiliary have, infinitive	Have	VHI	HV	VB
Verb, auxiliary have, past	Had	VHD	HVD	VBD
Verb, auxiliary have, present participle	Having	VHG	HVG	VBG
Verb, auxiliaryhave, past participle	Had	VHN	HVN	VBN

Verb, auxiliary have, present3SG	Has	VHZ	HVZ	VBZ
Verb, auxiliary be, infinitive	Be	VBI	BE	VB
Verb, auxiliary be, past	Were	VBD	BED	VBD
Verb, auxiliary be, past 3SG	Was	VBD	BEDZ	VBD
Verb, auxiliary be, present participle	Being	VBG	BEG	VBG
Verb, auxiliary be, past participle	Been	VBN	BEN	VBN
Verb, auxiliary be, present 3SG	is, s	VBZ	BEZ	VBZ
Verb, auxiliary be, present 1SG	am, m	VBB	BEM	VBP
Verb, auxiliary be, present	are, re	VBB	BER	VBP
Verb, modal	can, could, ll	VM0	MD	MD
Infinitive marker	To	TO0	TO	TO
Preposition, to	To	PRP	IN	TO
Preposition	for, above	PRP	IN	IN
Preposition, of	of	PRF	IN	IN
Possessive	s,	POS	$	POS
Interjection (or otherisolate)	oh, yes, mmm	ITJ	UH	UH
Punctuation, sentence ender	. ! ?	PUN	.	.
Punctuation, semicolon	;	PUN	.	:
Punctuation, colon orellipses	:	PUN	:	:
Punctuation, comma	,	PUN	,	,
Punctuation, dash	-	PUN	-	-
Punctuation, dollar sign	$	PUN		$
Punctuation, quotation mark left		PUQ		
Punctuation, quotationmark right		PUQ		
Foreign words (not in English lexicon)		UNC	(FW-)	FW
Symbol, alphabetical	A, B, c, d	ZZ0		
Symbol, List item	A A. First			LS

A.2 PTB Tagset

POSTags	Definition		
CC	Coordinating Conjunction	RP	Particle
CD	Cardinal number	SYM	Symbol (mathematical or scientific)
DT	Determiner	TO	To
EX	Existential there	UH	Interjection
FW	Foreign word	VB	Verb, base form
IN	Preposition / subordinating conjunction	VBD	Verb, past tense
JJ	Adjective	VBG	Verb, gerund/present participle
JJR	Adjective Comparative	VBN	Verb, past participle
JJS	Adjective Superlative	VBP	Verb, non-third person singular present
LS	List Item Marker	VBZ	Verb, third person singular present
MD	Modal	WDT	wh-determiner
NN	Noun, singular or mass	WP	wh-pronoun
NNS	Noun, plural	WP$	Possessive wh-pronoun
NNP	Proper noun, singular	WRB	wh-adverb
NNPS	Proper noun, plural	#	Pound sign
PDT	Predeterminer	$	Dollar sign
POS	Possessive ending	.	Sentence-final punctuation
PRP	Personal pronoun	,	Comma
PP$ [1]	Possessive pronoun	:	Colon, semi-colon
RB	Adverb	(	Left bracket character
RBR	Adverb, comparative	)	Right bracket character
RBS	Adverb, superlative		

[1]There is an inconsistency aboutPossessive Pronoun in Penn treebank POS tagset in various sources. (Mar93), (San90) and (Tay03) list PP$ whereas(Man99) lists PRP$. Most online webresource suggest PRP$ but still there are some web resources suggesting PP$The Treebank data itself tags possessive pronouns with PRP$, although most ofthe documentation with Treebank_2 lists PP$. However, the README.pos (in Treebank_2/Treebank_2/cdrom/tagged/) points out the change in possessivepronoun from PP$ to PRP$, along with other changes which incidentally havebeen incorporated by most of the references.

A.3 ARK Tweet tagset

Tag	Description	Example
Nominal, Nominal+Verbal		
N	common noun(NN, NNS)	books someone
O	pronoun (personal/WH; not possessive; PRP, WP)	it you u meeee
S	nominal+possessive	books someones
ˆ	proper noun (NNP, NNPS)	lebron usa iPad
Z	proper noun + possessive	Americas
L	nominal + verbal	hes bookll iono(= I dont know)
M	proper noun + verbal	Markll
Other open-class words		
V	verb including copula, auxiliaries (V*, MD)	might gonna ought couldn’t is eats
A	adjective (J*)	good fav lil
R	adverb (R*, WRB)	2 (i.e too)
!	interjection (UH)	lol haha FTW yea right
Other closed-class words		
D	determiner (WDT, DT, WP$, PRP$)	the teh its its
P	pre- or post-position, or subordinating conjunction (IN, TO)	while to for 2 (i.e., to) 4(i.e., for)
$	coordinating conjunction (CC)	and n & +BUT
T	verb particle (RP)	out off Ud UP
X	existential there there, predeterminer (EX, PDT)	Both
Y	X+verb	theres all
Twitter online-specific		
#	hashtag (indicates topic/category for tweet)	#acl
@	at-mention (indicates another user as a recipient of a tweet)	@BarakObama
~	discourse marker, indication of continuation of message across multiple tweets	RT and : in re tweets construction RT @user : hello http://bit.ly/xyz
U	URL or email address	
E	emoticon	:-) :b (: <3 o_O
Miscellaneous		
$	numeral (CD)	2010 four 9:30
,	punctuation (#, $, ’ ’, (,), , , . , : , ‘ ‘)	!!! ?!?
G	Other abbreviations, foreign words, possessive endings, symbols, garbage (FW, POS, SYS, LS)	–>awesome ... Im