Analysis of the Impact of Linguistic Processing on Large Scale Semantic Similarity

Meryem M'hamdi, Dr. Martin Rajman
School of Computer And Communication Sciences
Swiss Federal Institute of Technology (EPFL)
Lausane, Switzerland
{meryem.mhamdi, martin.rajman} @epfl.ch

***Abstract*—Computing semantic similarity became a popular task nowadays as many approaches have been suggested ranging from lexical matching, hand coded patterns and distributional semantics. While the performance of a pure linguistic rule-based approach is restricted to the availability of linguistically well-formed ground truth corpus and suffers from lack of tools efficient for large scale and language dependability, distributional semantics approaches such as word2vec and fast text were specifically devised to work on a large scale, however, implying minimal to non-existing proper linguistic processing which can lead to poor accuracy. In this project, we bridge the gap between the two approaches by putting in place the building blocks of a clean, debug-gable and efficient NLP pipeline for linguistic pre-processing which could be directly embedded into the mechanism used to compute semantic similarity based on word embedding. The purpose of this project is to show the relevance of a hybrid rule-based iterative linguistic methodology for distributional semantics based on elaborate linguistic processing. For that purpose, we implement several lexical analysis algorithms which recognize tokens based on traversal of Finite State Automata (FSA). Then, we adapt and extend a morphological analyzer based on Finite State Transducers (FST) and well engineering morphological rules. We feed the processed output into a bottom-up dependency parsing algorithm in order to generate the dependency pairs crucial for calculating word co-occurences. We explain how expectation minimization is followed for training those co-occurences in a way that resolves the ambiguities that we take care to propagate throughout the NLP pipeline to deal with them when relevant. We evaluate our algorithms and the interactions between the modules by computing different metrics to assess the accuracy of lexical and syntactic analysis. Out of 12,543, 76.4% were tokenized and analyzed correctly from a morphological point of view, while the remaining were either lacking or not following our EOS conventions. CYK reaches a high performance for training data, and only needs to be made efficient by training it on big corpus to generalize well enough on unknwon text and using semantic similarities to disambiguate the dependencies.**

Keywords*—NLP Pipeline, Semantic Similarity, FSA for lexical recognition, Tokenization FST for morphology, Lemmatization, CYK, bottom-up chart parsing, expectation-minimization

I. Introduction

Semantic Similarity has many exciting applications to text understanding nowadays as it helps summarize main points out of a text, categorize and compare different meanings. Many approaches have been suggested in this regard some of them rely solely on rule-based linguistic analysis and lexical matching such as edit distance and largest common substring relying on the principle that words that have close morphological appearance such as in the word US and UK may exhibit semantic similarity like in the work of Mengqui et al in [1], which employs probabilistic weighted distance. This approach clearly can only capture short-term word alignment and edit distance is not always a good predictor of semantic similarity. For example, words such as America and US will have a lower distance thus low similarity compared to the pair of words consisting of US and UK. Other approaches employ syntactic parsing to suggest which pairs of words can exhibit semantic similarity following their syntactic roles relative to each other. However, it is discouraging to make maximal usage of syntactic tools which can be computationally expensive and its quality may largely depend on the kind of language trained on and the availability of annotated corpus of well-formed text.

On the other hand, other approaches jump directly to exploiting word embeddings such as word2vec, GLOVE and fast text for enhancing their computational efficiency for ready usage in large scale applications. While they helped speed up the development process of many solutions to NLP tasks with a competitive quality, they rely on questionable assumptions that orient the objectives of current NLP tasks more towards efficiency and short response time which is crucial in current time synchronous speech recognition for example. This comes at the expense of lower accuracy and questionable results at later stages of the NLP pipeline by skipping or not giving much importance to the earlier processing stages. In sum, how word embeddings work in current applications only require a large amount of unlabelled text data used to fill a semantic vector word space without any analysis of syntactic structures and by relying on basic techniques for deciding on the definition of a word in vector space. For example, a commonly used approach for tokenization in such tools is the separation based on punctuation and some standard regexes for non-whitespaces and alphabetic sequences which is an assumption that doesn't work in all cases. For example, in the sentence *"Although they denied it during their 5pm press conference today, the $5,000 a day head lawyers of Grace O'Reilly's opponents will probably not let her power their electric bikes with Li-ion battery packs again"* the existence of O'Reilly as a possible token and 's as another token will be ignored.

In the same way, simplistic shortcuts for linguistic pre-processing are employed for removing noisy words by defining and compiling a fixed list of common stopwords and for lemmatization if applied at all (not always used) by following a lookup table. As a consequence, the similarities will be computed on words leading to word relationships that don't exist initially or will miss words that are important (for example, a word might not be always a stopword as this may depend on the context). In the same manner, the definition of a proper context for analyzing the similarities between words seems to be more motivated by reasons to maximize efficiency and simplification of the training process. The context window defines in a measured and rigid way a context that is supposed to be dynamic in its nature. Trying to fit a problem in NLP with its ambiguous nature into a solution with fixed dimensions can only hide the symptoms of the problem and is far from solving it.

Our approach is a proposal to balance between the two extreme approaches by defining a hybrid principled approach that focuses on the design of a well-engineered fully debuggable incremental architecture to which semantic embedding can be embedded and adapted. The main purpose is to make easy to know which parts need improvement and how to make them efficient enough to comply with the type and size of problem or dataset at hand. It also tries to limit the assumptions and unnecessary lazy heuristics that can make the results totally inverifiable and irrelevant afterwards. Our aim is to study the problem from a large scale point of view which would require both efficiency and incrementality.

In Section II, we review previous work and provide a summary of state-of-the-practice methodologies for NLP processing for semantic similarity in general and main concepts, algorithms and inspirations in each step in the pipeline. In Section III, we describe the dataset used, its relevance to the task at hand along with some general statistics. Next, in section IV, we give the general overview of architecture of the NLP pipeline and explain in details each step in it separately and how they connect and interact with each other. In Section V, we explain the performance evaluation methodology used and discuss the results of lexical and syntactic analysis both in terms of accuracy and degree of ambiguity. We conclude with section VI and VII which summarize the main conclusions and findings and gives useful pointers for continuing this work in the future.

II. Related Work

As this work, studies different aspects of the NLP pipeline, we provide hereafter a brief summary of previous work not only at the intersection of all stages of the pipeline but also those works that tackle each aspect separately. In section A, we start by describing the rational and limitations of some tools used in lexical and morphological analysis, then, in section B, we compare common algorithms in syntactic analysis, specifically dependency parsing. In section C, we explain more approaches to word embeddings and main considerations, theories and progress. Then, in the last section, we cover two works that tries incorporating linguistic processing to help with semantic analysis.

A. *Lexical and Morphological Analysis*

PTBTokenizer is a tokenizer developed by Stanford Natural Language Processing Group [2] which relies on a fast and efficient implementation based on deterministic finite state automaton. It can tokenize up to a rate of 1,000,000 tokens per second using some heuristics that decide about sentence boundaries. Its FSA is written using JFlex, a standalone general purpose lexical analyzer generator for Java, written in Java. Given as an input the the set of regular expressions and corresponding regex operations, it can read the input and run the corresponding operations defined for the regex that matches the input giving the longest match possible if it exists, thus not allowing for multiple outputs in case of ambiguity. Its fast backtracking and traversal is largerly due to its simplified assumption that only one possible output is sufficient which is not always the case.

Many different tools have been implemented for lemmatization such as NLTK lemmatizer which lemmatizes based on a WordNet's built-in morphy function. The problem with this current approach is its inability to generalize to new word stems based on their similarity with the existing words. It doesn't focus much on defining any rules that be applied in general to make the process automatic and debuggable.

B. *Syntactic Analysis*

Approaches for data-driven dependency parsing can be classified into three families each with its pros and cons as the diagram in figure 1 shows. While some of them such as Turbo dependency parser run by splitting into two disjoint steps: POS tagging before dependency parsing, others recover dependencies trees in a single step either by using heuristics to select a single parsing possibility along the way which processes input linearly or by keeping all possible trees in the chart which has the worst efficiency but also the best accuracy. The first variant of single step approaches is motivated by a stack-based approach called shift-reduce parsing [3] which was initially developed for analyzing programming languages which rely on formal, unambiguous grammar. It a transition based approach that employs an Oracle which decides on actions to take (shift if it can't do anything with the input or reduce if there is a dependency rule that can apply) regarding the list of tokens to be parsed and managed using a stack. Unlike this approach which uses approximate parsing to make the parsing efficient, the last approach is an exhaustive brute force approach which explores and keeps track of all possible trees and which rely on dynamic chart parsing techniques. Both approaches fall into the category of arc factored models can be either projective or non-projective. Projective

C. *Semantic Analysis*

The distributional approach to semantic similarity relies on the principle that words that appear closer to each other in similar contexts are more likely to exhibit similar semantic meanings. There are many different approaches to distributional semantics which vary in their way of training co-occurence matrix. Since it can be quite computationally

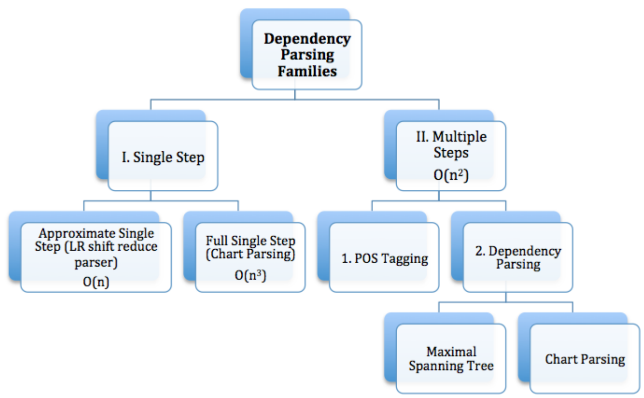

Fig. 1. Dependency Parsing Families

demanding to keep high dimensional word vectors, many approaches were devised to train in such a way that captures the variance while reducing the dimensionality. Pointwise Mutual Information (PMI) and Latent Dirichlet Allocation (LDA) are among those approaches. Word2Vec with its two architectures: continuous bag of words (CBOW) and Skip-gram two simplied versions of neural network models consisting only of two-levels that can learn word similarities from large scale text as was shown by Mikolov et al. [4] as their performance tested on SemEval-2012 Task outpeformed other state-of-the-art methodologies. They show how the skip gram model can learn high-quality distributed vector representations from large amounts of unstructured text data that capture a large number of precise syntactic and semantic word relationships that can be represented as linear translations. In some given conditioning context, the vector representations are initialized arbitrary, then concatenated and fed into different layers of neural network where the output is the softmax layer which will define the probability of observing each possible word in our vocabulary following our context.

Given the big size of the softmax layer that grows with big vocabulary, computing all the pre-activations and performing the softmax linearity for every combination of context and word that follow it in the context is not all that efficient and can be further improved. Different techniques such as hierarchical softmax were constantly proposed to extend those models and make them efficiently compute word vectors at the potential loss of some accuracy. The idea behind hierarchical softmax is using a hierarchical layer where we will decompose the probability of observing a word into a sequence of probabilities corresponding to choices as to which group does the next word is more likely to belong to instead of modeling the probability of the next word using a flat layer. In [5], Mikolov et al. presented an even more efficient technique relying on a modified version of sampling. For making its training even faster and enhance its ability to return better vector representations for frequent words, they proposed a simplified variant of Noise Contrastive Estimation which is more stable than Importance Sampling as it uses an auxiliary loss that optimizes the goal of maximizing the probability of correct words.

In [6], Mikolov et al. demonstrate that vector-space representations of word implicitly learned by the input-layer weights can capture syntactic and semantic regularities in language. Those regularities can be observed as constant vector offsets between pairs of words sharing a particular relationship. Words are first converted via a learned lookup-table into real valued vectors, which are used as the inputs to a neural network. Then neural networks are used to build a language model in an unsupervised manner by predicting a probability distribution over the next work given some preceding context words by optimizing the maximum likelihood training criterion. The popularity of neural network for this task comes from its ability to reach outstanding performance in learning word vectors in an efficient manner through the use of implementation variants relying on hierarchical prediction, which bypasses a classical n-gram model. It was demonstrated that vectors learned using recurrent neural networks can capture more significant syntactic regularities than LSA vectors.

D. NLP Processing for Semantic Similarity Analysis

Several research have been conducted to link syntactic analysis to word embedding. Among them, Levy et al. in [7] extend the work done for the Skipgram embedding model by experimenting with dependency-based contexts rather linear bag of-words contexts. They show that the two approaches produce two different kinds of similarities. The main motivation was to capture relations to words that are out-of-reach using small linear windows and filter coincidental contexts which are within the window but whose relationship is not semantically relevant. The higher results in both precision and recall for Dependency-graph based contexts compared to fixed window size encourages more work into this direction. On the other hand, Qiu et al. in [8] experimented with the use of syntactic dependencies for improving analogy detection based on distributed word representations to show that dependency-based that doesn't outperform an n-gram based approach but rather can help with filtering possible candidates.

III. Universal Dependencies Treebank

This corpus consists of 16,662 sentences extracted from various web media including weblogs, newsgroups, emails, reviews and Yahoo! answers which makes up 254,830 words in total. Those sentences were annotated using Universal Dependencies annotation which is a project that is developing cross-linguistically consistent treebank for many languages with the goal of providing a common ground and universal guidelines for facilitating consistent annotation of similar constructions across languages while respecting languages particularities whenever necessary and facilitating multilingual parser development, cross-lingual learning, and parsing research from a language typology perspective [9]. The annotation scheme used is an evolution of the Stanford dependencies and the Interset, Google universal part-of-speech tags (Petrov et al., 2012), and the Interset interlingua for morphosyntactic tagsets (Zeman, 2008).

TABLE I. UNIVERSAL DEPENDENCIES TREEBANK TRAINING AND TESTING STATISTICS

	Train	Test	Overall
# Sentences	12,543	2,003	14,546
# Words	200,688	25,148	225,836
# Unique Words	19672	5495	21335
Max Number of Words/ sentence	159	75	159
Min Number of Words/ sentence	1	1	1
Average Number of Words/ sentence	16	12	15

This corpus comes annotated with the CoNLL-U format which is a revised version of the CoNLL-X format. There are three types of lines:

- Word lines containing the annotation of a word/token in 10 fields separated by single tab characters. Each word line consists of the following fields:
  - **ID:** Word index, integer starting at 1 for each new sentence; may be a range for tokens with multiple words.
  - **FORM:** Word form or punctuation symbol.
  - **LEMMA:** Lemma or stem of word form.
  - **UPOSTAG:** Universal part-of-speech tag drawn from our revised version of the Google universal POS tags.
  - **XPOSTAG:** Language-specific part-of-speech tag; underscore if not available.
  - **FEATS:** List of morphological features from the universal feature inventory or from a defined language-specific extension; underscore if not available.
  - **HEAD:** Head of the current token, which is either a value of ID or zero (0).
  - **DEPREL:** Universal Stanford dependency relation to the HEAD (root iff HEAD = 0) or a defined language-specific subtype of one.
  - **DEPS:** List of secondary dependencies (head-deprel pairs).
  - **MISC:** Any other annotation.
- Blank lines marking sentence boundaries.
- Comment lines starting with hash #.

For assessing the quality of our NLP pipeline with its different functionalities and components in its various stages, this treebank is split into training and testing. Table I shows some statistics of the splitting.

IV. METHODOLOGY

A. *Overview of NLP Pipeline Architecture*

Hereafter, we propose an incremental, debuggable NLP pipeline in which we use the following principles to orient us towards the nature of building blocks and kind of interactions between them. First, before applying any tokenization algorithm, we pay special attention to building a clean process that enables us to extract and classify the list of entries and generalize as much as we can through the use of common regular expressions. Both lexical and morphological analyzers were built to be complementary and contribute to each other mutual development. Entries generated in lexical analysis are further fed into Morphological process which at the same time benefits and can benefit from lexical analysis. The generation module can be applied through the use of already defined rules, and this way morphology can extend those entries by adding more inflections to specific forms of inputs. At the same time, without those entries, the morphological analyzer cannot be customized to cover all possible cases and exceptions specific to the corpus at hand. Before proceeding with any further phase in the pipeline we think it is crucial to evaluate those two steps separately and keep only those judged lexically correct. In the same way, the syntactic and semantic analyzer work hand in hand to fine tune the co-occurrences that were computed using syntactic dependencies as those co-occurrences which will be used to compute semantic similarities will be used to assign weight to the probabilistic part of the syntactic parser in an iterative way.

The diagram in figure 2 gives a general overview of how the building blocks of the pipeline fit together. In sum, the corpus is fed as an input to extract entries in order to build the FSAs. With the help of FSTs, extra lexicon entries are added to the FSAs to make them more clean and general. Those FSAs which are traversed using a tokenization algorithm which comes with an EOS mechanism to decide at which limit to cut sentences with their tokens. Then, the output which is in the form of list of charts where each cell contains a map from surface to cannonical forms are fed to syntactic analysis part which finds dependencies based on the trained dependency grammar rules extracted from the treebank using mechanisms that we explain in the next setcion. The same procedure is tested and debugged on a bigger corpus to get relevant lemma-co-occurrences on a large scale after keeping lemmas that are not stop words based (on their POS tags) and those similarity scores are computed by scaling over the co-occurences using some scaling measure like PPMI. Those lemma similarities will be used to assign scores to their corresponding grammatical dependency rules. Now, probabilistic CYK is used to train again the dependencies and to recompute the co-occurences and based on the similarities the syntactic weights are further adjusted in a iterative manner. Upon convergence of this process, the similarities can used and analyzed in some application such as word clustering.

B. *Lexical Analysis*

The two solutions we propose for separating a bulk text into a sequence of sentences where each is in turn separated into its possible tokens rely on the usage of one or more Finite State Automata depending on the type of the solution (explicit or implicit). Given a raw corpus, our proposed approach produces a list of all possible token sequences for each detected sentence in the form of two dimensional charts. The choice of chart based representation was motivated by its ability to store ambiguous output in a compact and structured way that allows for faster traversal and retrieval of information in later stages of the pipeline.

Finite State Automata in both their deterministic and non-deterministic forms is a well-established approach for surface-form field representation. It provides efficient lookup either by

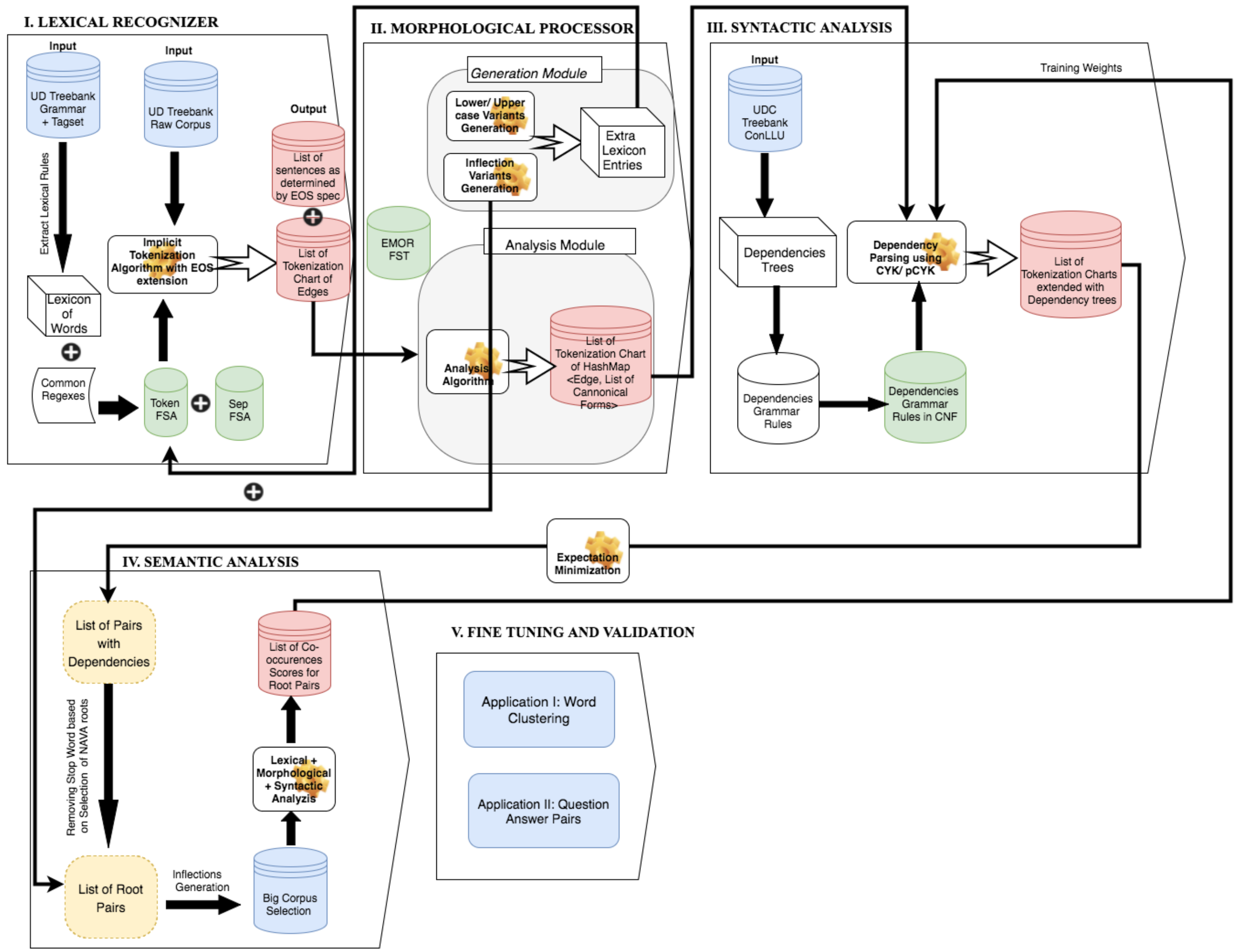

Fig. 2. NLP Pipeline Diagram

value or by reference both in terms of time and space complexity thanks to its ability to represent the entries expressed in terms of regular expressions in a compact way which does better than a look-up list or trie. As a reminder (needed to explain some algorithms in this section), an FSA (figure 3) consists of Q, the finite set of states, Σ, the finite set of alphabets that make up the arcs, δ the set of arcs mapping from Q x Σ, a starting state $q_0 \in Q$ and the set of final states F Q.

*1) **Building FSAs**:* Before explaining the different algorithms used for tokenization, we give in this section a general description of the process used to build and fill FSA. The dk.bricks.automaton [10] interface which consists of Finite-State Automata with its deterministic and non-deterministic implementations and has support for most of the standard regular expression operations such as concatenation, union,

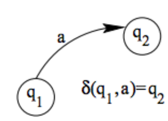

Fig. 3. Finite State Automaton Notations

Kleene star and other non-standard operations like intersection, complement operating on unrestricted representations of regular expressions based on any Unicode character. It also comes with several helper classes such as automaton matching, optimization, shuffling, serialization, exporting and loading. This package is a fast, compact, and implementation of real, unrestricted regular operations that was intended initially for arithmetic expressions recognition but thanks to its polyvance and efficient implementation of the automaton interface and its components, we adapted it to be used it not only for language recognition but analyzis.

For that purpose, we needed to extend this package by adding the following components, functionalities and algorithms described in the following sections:

*2) **Filling in the Lexicon**:* The entries of the lexicon are all extracted from the word forms in the treebank sentences with further cleaning. There are divided into three categories:

- Pure Alpha sequences: those are added directly without further due.
- Separators: those are checked for by observing their usage in the context.
- Entries that can be generalized using regexes: those include urls, emails, numbers, fractions, etc. An example of list of common-regexes in provided in III.
- Entries with special encoding: In order to treat the characters that require either special encoding in java or overlap with regular expression notation such as period, apostrophes, @, slashes and so on.

After treatment, a union of these entries is added to one or more FSAs according to the specification of the algorithm which we explain in next sections.

*3) **Explicit Solution**:* This solution relies on the traversal of the combination(union) of two FSAs as summarized in the table II:

- **tokFSA:** consists of the union of the different possible alphanumeric entries extracted from the lexicon in addition to common regexes for representing alpha sequences, dates, times, digits, fractions in their different forms, percentages, prices, docs, emails, emojis, urls and other special characters as detailed in table III which is made for illustration purposes and by any means exhaustive as it is not possible to cover all possible cases of regexes but debugging and extending them with time is full of reach. This alphanumerical word entries are the ones extracted from the lexicon of the train UD treebank as explained in previous section. Each entry is encapsulated between two special characters "E" and "S" (or a separator entry) which are added to mark its start and end respectively.
- **sepFSA:** consists of all character sequences that are considered as separators for which a sample list is shown in table IV.By definition, a separator is any sequence that cannot stand by itself unless it is occurs between actual word tokens. TokFSA and sepFSA are mutually exclusive in the sense that the only case when we can have a separator considered a token is if it is inclusively included in the token as there are some contexts in which a character/ sequence of characters of separators are part of the word itself. Similar to the mechanism in tokFSA, separator entries are added to sepFSA in such a way that controls which kind of characters can go before and after them. A separator by convention can occur after any kind of character and before the enclosing character marking either the end of the matching string which is "S".

Given any sequence of characters, this algorithm maintains a working copy of it and leaves the original sequence untouched and only for locating the position of the walker pointer. It adds special characters for e.g. "S" and "E" to the start and end of the working copy. It uses two pointer one to walk through the working string which is called the start pointer and the other to keep track of the real position in the original string which we call walker pointer. After initializing the positions of the start and walker pointers to 0, it traverses the FSA looking for the longest prefix match possible while keeping note of the position of the prefixes at which it found a shorter match in a stack which is filled updated the process and reinitialized once the process is done for all possible shortest and longest matches for the longest match are detected. Once it found the longest match, it comes back to the position of the shortest prefix not detected yet by popping it from the stack by updating the start pointer of the working copy with that the position. It carries

TABLE II. EXPLICIT FSA CONTENT

<table>
<tr><th>Explicit FSA</th></tr>
<tr><td>start = S
endTok = sep | S
endSep = . | E
tokFSA = concat (start,tokEntries,endTok)
sepFSA = concat (start,sep,endSep)
FSA = union(tokFSA, sepFSA)</td></tr>
</table>

TABLE III. TOKEN FSA COMMON REGEXES

<table>
<tr><th></th><th>Regexes</th></tr>
<tr><td>alpha</td><td>[a-zA-Z]+</td></tr>
<tr><td>digits</td><td>[0-9]+</td></tr>
<tr><td>fractions</td><td>([0-9]+)[.;]([0-9]+)|
([0-9]+)[.]([0-9]+)[.]([0-9]+)|
([0-9]+)[,]([0-9]+)[.]([0-9]+)|
[0-9]+/[0-9]+|[0-9]+;[0-9]|
([0-9]+)([,]([0-9]+))+|
+[0-9]+|
[0-9]+[-0-9/]+[0-9]+|
[0-9]+[-0-9/]+[0-9]+</td></tr>
<tr><td>percentages</td><td>(([0-9]+)|([0-9]+)[.]([0-9]+))([%])</td></tr>
<tr><td>dates</td><td>(Jan|Feb|Mar|Apr|May|Jun|Jul|Aug|Sep|Oct|Nov|Dec)[.] 1(st)?|
(Jan|Feb|Mar|Apr|May|Jun|Jul|Aug|Sep|Oct|Nov|Dec)[.] 2(nd)?|
(Jan|Feb|Mar|Apr|May|Jun|Jul|Aug|Sep|Oct|Nov|Dec)[.] 3(rd)?|
(Jan|Feb|Mar|Apr|May|Jun|Jul|Aug|Sep|Oct|Nov|Dec)[.] [1-9]+(th)?|
[0-9]+/[0-9]+/[0-9]+</td></tr>
<tr><td>times</td><td>[0-9]+:[0-9]+|[0-9]+:[0-9]+:[0-9]+</td></tr>
<tr><td>prices</td><td>([$])(([0-9]+)| ([0-9]+)[.]([0-9]+)|
([0-9]+)([,]([0-9]+))+)|
(US)([$])(([0-9]+)|
([0-9]+)[.]([0-9]+)|
([0-9]+)([,]([0-9]+))+)</td></tr>
<tr><td>docs</td><td>.+[.](DOC|XLS)</td></tr>
<tr><td>emails</td><td>[a-zA-Z0-9][a-zA-Z0-9._-:]+[
@][a-zA-Z0-9._-]+[a-zA-Z0-9]|
[a-zA-Z0-9][a-zA-Z0-9_
%:]+
@[a-zA-Z0-9_]+[.][a-zA-Z0-9_]+[a-zA-Z0-9]</td></tr>
<tr><td>urls</td><td>(http://|www[.])([a-zA-Z#]+|[.]|/|[0-9]+|-|?|=|,|) +</td></tr>
<tr><td>special characters</td><td>&|[%]|[$]+|[č]|[$∼ $]|[]|[+]|− > |[/]|[@]</td></tr>
</table>

TABLE IV. LIST OF SEPARATORS REGEXES, PERSISTENCE AND EOS SPECIFICATIONS

<table>
<tr><th></th><th>Separators Regexes</th><th>Persistence</th><th>EOS?</th></tr>
<tr><td>Space</td><td></td><td>False</td><td>False</td></tr>
<tr><td>Terminators</td><td>[.]|[.][.]|[.][.][.]|?|??|!|!!|!!!|;</td><td>True</td><td>True</td></tr>
<tr><td>Mixtures</td><td>[.]???|[.]?|?!?|???|!?|?|?!|?!?!?|!!!!!|,?</td><td>True</td><td>True</td></tr>
<tr><td>Comma, colon, slash</td><td>,|,,|:|[(]:|’,|[.]:|[.]/</td><td>True</td><td>False</td></tr>
<tr><td>Quotes</td><td>âĂŸ|âĂŹ|âĂIJ|âĂİ|”|[""]|‘|âĂŸ|["]|’|âĂŸâĂŹ|""</td><td>True</td><td>False</td></tr>
<tr><td>Dashes</td><td>| − −| − |</td><td>True</td><td>False</td></tr>
</table>

on till there are no elements in the stack, in which case it pops the longest match from the working string leaving the start and end special characters untouched. Each time it detects an edge it adds it the two dimensional chart such that the level where it is put depends on the number of the end of shortest prefixes stored in the stack that are contained in the further reaching edge of the element to be added to the chart.

*4) **Implicit Solution:*** Unlike in explicit solution where we need to explicitly introduce start and end special characters, this solution is more elegant as it doesn't require such enclosing characters where tokFSA and sepFSA are simply the entries that we get from the lexicon and other common regexes for tokens and separators separately. However, it comes with the cost of working out a proper convention to alternate between tokFSA and sepFSA. It first starts by looking for a match in tokFSA, otherwise it looks for it in sepFSA. It keeps moving using the working FSA (the one that gives the match) unless it finds a mispelling or a character for which there is no matching path in FSA from the previous character in which case it pops the string from the starting position and labels as unknown the whole character sequence from the end of last match longest prefix edge up to the current character. When a final state is reached, there are two cases:

- **Non-stuck Match and next character is a proper separator:** there are still further reaching paths that be transitioned to. In this case, the current position is pushed to the stack in which the positions of those shorter prefixes and the walker pointer moves to the next character looking for a longer match without reinitializing the working FSA.
- **Non-stuck Match and next character is not proper separator or Stuck Match and next character is not a proper separator:** It ignores the currently detected edge and moves on with the next character without reinitializing the working FSA.
- **Stuck Match and next character is a proper separator:** there are no further path in the working FSA that ensure a transition from the current final state reached by the current character to the next state that can be reached by the next character in the input string. In this case, the stack of shorter prefixes is fetched and in case it is not empty meaning the current detected edge is the longer match for other shorter match, it comes back to detect those shorter matches by updating the current walker pointer to the first element of the stack (i.e. the end of the first shortest edge in the stack). Once the stack becomes empty, the working FSA gets reinitialized to its starting state and the walker pointer is now at the end of the longest match. Only at this point, it can start investigating the possibility of building a chart. For that, it checks if the end character or sequence of characters of the last edge added to the chart are separators with true EOS specification and only if it is not contained in another further reaching edge. An arc (n1,n2), where n1<n2, is a further reaching arc covering the arc (m1,m2), where m1<m2, iff n1 <= m1 and m2<n2 (notice that the second inequality is strict). For example, in the sentence: "The length is 5.2 meters... Let's go", when it detects "5.2" as a possible token it is not going to build a chart as the EOS separator "." is enclosed in the further reaching edge "5.2"

C. *Morphological Analysis*

Now that the charts are initialized with all possible tokens sequences, the role of the morphological analysis is to extend it with map each matched token in its surface form into the list of all possible canonical forms (root and its syntactic information). The two way mappings (from surface form to cannonical forms and vice versa) was implemented using transducers written using Stuttgart Programming Language for Finite State Transducers (SFST-PL) which is based on extended regular expressions with variables. It is a general-purpose programming language which gives access to all the descriptive means for the development of customized finite state transducers. It provides tools for compiling any defined computational morphology covering derivation, composition and inflection thus giving the required abstraction which separates between the low-level compilation and the mechanisms to transform a written specification of a transducer or multiple transducers into morphological analyzer or generator and the lexicon entries and rules of transformation, mapping, composition which allows the development of more complex transducers easily.

As a reminder, an FST is a deterministic Finite State Automaton that maps one language to another is defined as $\Sigma = ((\Sigma_1 \epsilon) x (\Sigma_2 \epsilon))\ (\epsilon, \epsilon)$ where Σ_1 and Σ_2 are two enumerable alphabets. In morphology Σ_1 and Σ_2 are surface and cannonical forms respectively. Mapping from Σ_1 to Σ_2 is called analysis phase while mapping from Σ_2 to Σ_1 is called the generation module.

I have used EMOR, an already compiled and debuggable morphology tool developed using SFST-PL tools, for which the majority of tokens in UD corpus were detected[11]. For the remaining proportion for which there is no mapping with EMOR, I have followed the following methodology to add them:

- Adding Missing Regular Rules: All regular rules were already added including the following rules for defining inflections corresponding to different paradigms:
  $\$verb - reg - infl\$ = < V > (< 3sg >< PRES > : s| < PAST > : ed| < PPART > : ed| < PROG > : ing| < INF > :)$
  This rule adds "s" to each verb for which we would like to generate its 3 singular present form $< 3sg >< PRES >$, adds "ed" to each verb for which we would like to generate its past $< PAST >$ or past participal $< PPART >$ form and so on.
  Other examples of rules for nouns, adjectives are for analyzing plural for nouns and superlatives and comparatives for adjectives:
  $\$noun - reg - infl\$ = < N > (< 3sg > : | < 3pl > : s)$
  $\$adj - reg - infl\$ = < Adj > (< pos > : | < comp > : er| < sup > : est)$
- Adding Missing POS tags: Those included foreign words tag and adapting the penn POS tag convention to the one used in EMOR which is more informative.

Algorithm 1 Explicit Tokenization Algorithm based on FSA Traversal

Input: s is the string to be tokenized and $automaton$ is the finite state automaton.

```
function ExplicitTraverseFSA(s, automaton)
    states ← set of states of automaton
    pp ← linked list of transitions from currently traversed state
    pp_other ← linked list of other possible transitions from other possible states
    bb ← Bit set of transitions from currently traversed state
    bb_other ← Bit set of other possible transitions from other possible states
    initialState ← Start state of the automaton
    Add initialState to pp
    workingCopy ← "S" + s + "E"
    stuck ← false
    start ← 0                    ▷ To keep track of the current position at the original string
    i ← 0                        ▷ To iterate over the working string
    indices ←                    ▷ To store the edges detected during the traversal
    shortestPrefix ← list of indexes of the shortest prefixes to cut the working copy once the longest prefix is matched
    bottomChart ← list to store the shortest prefixes matches at the bottom of the dynamic chart
    accept ← boolean flag for detecting whether the state at the currently traversed character in the working copy is final
    while workingCopy ≠ "SE" do
        c ← character at i^th position of working string
        for p ∈ pp do
            dest ← list of states that can be reached from p following transition labelled with character c
            for q ∈ dest do
                if q.accept then
                    accept ← true
                end if
                if q.number ∉ bb_other then Add q.number to bb_other Add q to pp_other
                end if
            end for
        end for
        Swap pp with pp_other
        if c is NOT last character of working copy then
            Add edge(start, start + i − 1, flag) to indices
            if shortestPrefix.size() > 0 then
                workingCopy ← "S" + workingCopy.substring(shortestPrefix.get(0) + 1, workingCopy.length())
                start ← start + shortestPrefix.get(0)
                shortestPrefix ← ∅
            else
                workingCopy ← "S" + workingCopy.substring(i, workingCopy.length())
                start ← start + i − 1
            end if
            i ← 0
            pp ← ∅
            Add initialState to pp
            if stuck == false then
                Add indices(last) to bottomChart
            end if
            stuck ← false
```

Algorithm 2 Explicit Tokenization Algorithm based on FSA Traversal (continued)

```
    else
        if accept == true then
            Add edge(start, start + i − 1, true) to indices
            nextSize ← The number of transitions from the current state
            if nextSize > 0 then                      ▷ It doesn't get stuck: Ambiguity Detected
                Add i − 1 to shortestPrefixes
                i ← i + 1
                if stuck == false then
                    Add indices(last) to bottomChart
                end if
                stuck ← true
            else                                                           ▷ It gets stuck
                if shortestPrefix.size() > 0 then
                    workingCopy ← "S" + workingCopy − shortestPrefix
                    start ← start + shortestPrefix.get(0)
                    shortestPrefix ← ∅
                else
                    workingCopy ← "S" + workingCopy.substring(i, workingCopy.length())
                    start ← start + i − 1
                end if
                i ← 0
                pp ← ∅
                Add initialState to pp
                if stuck == false then
                    Add indices(last) to bottomChart
                end if
                stuck ← false
            end if
        else
            i ← i + 1
        end if
    end if
    for k ∈ [1, bottomChart.size()] do
        subChart ← new array list of edges
        for l ∈ [0, bottomChart.size() − k] do
            startIndex ← bottomChart.get(l).getStartIndex()
            endIndex ← bottomChart.get(l).getEndIndex()
            edge ← edge with startIndex=startIndex and endIndex=endIndex and recognized flag=true
            if indices contains edge then
                Add edge to subChart
            end if
        end for
            Add subChart to tokenizationChart
    end for
```

Output: Tokenization Chart = 2 dimensional array of edges where each edge is marked by a start index,end index (not to be considered) and boolean value for whether it is recognized by the lexicon or not.

Algorithm 3 Implicit Tokenization Algorithm based on FSA Traversal

Input: str is the string to be tokenized, $tokFSA$ is the FSA recognizing tokens only and $sepFSA$ is the FSA recognizing separators only

```
function ImplicitTraverseFSA(s, tokFSA, sepFSA)
    Set Start States of tokFSA and sepFSA to their initial states
    shortestPrefixes ← stack holding indices where the FSA finds a final state but can still continue
    bottomChart ← list of short edges to be inserted at the bottom of the chart
    bottomChartFlag ← false
    arcs ← List of all short and long edges
    sizeDestTok ← 0
    sizeDestSep ← 0
    flag_sep ← false
    i ← 0                                        ▷ Walker Pointer to traverse str
    start ← 0                                    ▷ Index at the start of the currently traversed edge
    while i < s.length do
        Traverse tokFSA
        if exists in tokFSA a transition labelled with character at position i in string str and flag_sep == false then
            if FINAL STATE then
                if exists in sepFSA a transition labelled with character at position i + 1 in string str then
                    if bottomChartFlag == false then Add edge from start to i + 1 to bottomChart
                    end ifAdd edge from start to i + 1 to arcs
                    if exists in tokFSA a transition labelled with character at position i + 1 in string str then
                        bottomChartFlag ← true Add i + 1 to shortestPrefixes
                        i ← i + 1
                    else
                        if shortestPrefixes.size() > 0 and i >= shortestPrefixes.first then
                            i ← shortestPrefixes.first
                            start ← i
                            Remove first element from shortestPrefixes
                        else
                            i ← i + 1
                            start ← i
                            Re-Initialize tokFSA to its start state
                        end if
                    end if
                else
                    i ← i + 1
                end if
            else
                if exists in tokFSA a transition labelled with character at position i + 1 in string str then
                    i ← i + 1
                else
                    if shortestPrefixes.size() > 0 and i >= shortestPrefixes.first then
                        i ← shortestPrefixes.first
                        start ← i
                            Remove first element from shortestPrefixes
                    else
                        flag_sep ← true
                        i ← start Re-Initialize tokFSA to its start state
                    end if
                end if =0
```

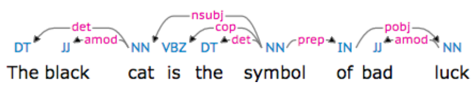

Fig. 4. Dependency Example

- Adding Missing Lemmas / roots (knowledge that applies to the patterns): Out of 19,672 unique words, 3567 lemmas were missing and needed to be added. The approach we followed to add them to the lexicon as canonical forms is by categorizing them using all possible penn POS tags with which they appear in the treebank and then we aggregated all distinct cannonical forms.
- Adding Missing Exceptions: For example, in the case of the past of verb plug which becomes plugged. A rule for the duplication of g was needed to be added by implying that each verb or noun ending with g and to which an inflection of the type "ing" or "ed" follows the g will make the g duplicate. This is the rule written in sfst-PL: $\$g-to-gg\$ = \{g\} : \{gg\} - > (\_\_[< V >< N >][ing|ed])$
- Lower Case / Upper Case Conversion: This involved adding an FST to help add the upper/lower case versions for each lemma in the morphology lexicon to reduce the amount of words missings.

D. Syntactic Analysis

In this project, we focus more on studying syntactic structures from the point of view of analyzing functional relationships and dependencies between words rather than decomposing phrases into their constituents in terms of structural and syntactic categories. In the context of this project, it is of outermost importance to analyze the dependencies between words which can give deeper knowledge about the reasons why some words could be more similar to others which can give good approximation of candidates for which it is relevant to study their semantic relationships. We rely on the principle that some strong syntactic relationships can hint into the existence of some kind of relationships at the semantic level which may not be directly suggestive of their natures. For example, in the sentence: "The black cat is the symbol of bad luck" in figure 4, black depends on cat but doesn't depend on luck so it doesn't make sense to study the semantic relationship between black and luck in this context. So syntactic dependencies helped us in this case filter possible candidates.

We restrict our attention to projective labelled dependency graphs which are well-formed. A dependency graph for a string of words $W = w_1...w_n$ is a labeled directed graph $D = (W, A)$, where 1) W is the set of nodes, i.e. word tokens in the input string, 2) A is a set of arcs (w_i, w_j) such that $w_i, w_j W)$. We write $w_i < w_j$ to express that w_i precedes w_j in the string $W(i.e., i < j)$; we write w_i âĘŠ w_j to say that there is a dependency arc from w_i to w_j ; we use to denote the reflexive and transitive closure of the arc relation and we use and for the corresponding undirected relations, i.e. $w_i w_j$ iff $w_i w_j$ or $w_j w_i$. A dependency graph is well-formed if it satisfies the conditions in figure 5. We rely on a dependency parsing algorithm which falls into the category of dynamic bottom up

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

Fig. 5. Rules of Well-Formedness of Dependency Graph

chart parsing namely CYK. This approach accommodates the format of the output from previous stages as well as it allows to learn how to extract all possible ambiguous dependency structures. In the following sections, we detail the algorithms at different sub-stages starting from the way the grammar was built, treatment needed to make it conform to CNF the format need to train CYK algorithm and how resulting trees were recovered.

*1) **Converting Dependencies Structures into Unique Trees:*** Since the approach we chose to rely on is bottom dynamic chart parsing, training a context-free grammar which encodes dependencies rules in a format compatible with the generic algorithm chosen in this project to generate dependency trees is the first step in this phase.Given a list of dependencies in ConLL-U format, we extracted for each treebank sentence its unique corresponding dependency tree in its basic format not necessarily binary by relying on the following convention which is possible since each token has one and only one head and all nodes can have 0 or more children except the root which has one and only one child. Each node representing the word that plays the role of a head has as its children all of the words that depend on it including the head itself all of which are ordered from right to left according to their order of occurrence in the treebank sentence. This convention enables to recover the sentence from the tree for which it is produced since the leaves are the tokens of the sentences. The order of the leaves preserves the same order of the tokens in the sentence if the sentence is projective. Each node has the following information: id, lemma, postag and deprel. The first set of information (id, lemma and postag) for each node are all extracted from the treebank whereas the dependendy relationship (deprel) is assigned in such a way that the child node always holds the deprel that relates it to its head except for the case of root relationship in which deprel for the head (root) is root and its child has * as its deprel or in the case where it is a duplicated head (to unify the format of representing the information relative to each node, we add * for special cases where a head is the child of the root or when it is repeated along with its children). The algorithm below summarizes the steps followed to generate a unique set of grammatical rules extracted from the tree which was generated and exploited in an efficient manner.

The following shows an example of the dependency tree corresponding to a particular projective sentence along with its grammatical rules where the order of the leaves from 1 to 16 is preserved in the same order as in the corresponding sentence: "The cells were operating in the Ghazaliyah and al-Jihad districts of the capital."

As an illustration of the result of the process of extracting the grammar in their non Chomsky Normal Form from the dependency tree, we provide hereafter the grammatical rules of the above dependency tree:

ROOT:root -> VBG:*
VBG:* -> NNS:nsubj VBD:aux VBG:* NNS:obl .:punct
NNS:nsubj -> DT:det NNS:*
NNS:obl -> IN:case DT:det NNP:compound NNS:* NN:nmod
NNP:compound -> NNP:* NNP:conj
NNP:conj -> CC:cc NNP:compound HYPH:punct NN:conj
NN:nmod -> IN:case DT:det NN:*

*2) **Checking for Projectivity**:* Figure 7 shows an example of the dependency tree corresponding to a particular non projective sentence where the order of the leaves from 1 to 16 (1,2,...,9,12,13,14,15,10..) is not preserved in the same order as in the corresponding sentence: "Guerrillas near Hawijah launched an attack that left 6 dead, including 4 Iraqi soldiers." while figure 12 shows the proportion of projective versus non-projective sentences in the training set.

*3) **Converting to Chomsky Normal Form**:* Since CYK is a bottom-up parsing that can handle at most binary rules, we needed to binarise the grammatical rules to make them comply to Chomsky Normal Form. According to Goddard in his lecture notes [12], a grammar is in Chomsky Normal Form if every one of its productions is either of the form A -> BC or A -> c (where A, B and C are arbitrary variables denoting non-terminals and c is a terminal. In the parsing grammar produced in the previous step, there are two types of rules that need our attention: $A : d_A$ -> $B_1 : d_{B_1}$ $B_2 : d_{B_2}$ ... $A : d_{A'}$ ... $B_n : d_{B_n}$ and rule of the form $A : d_A$ -> $B : d_B$ where the second one occurs solely in the root case, where dep_A,dep_B... denote the dependencies relationships. For the first case, we convert it by breaking it into several rules in the following way:

$A : d_A$ -> $B_1 : d_{B_1}$ $A : d_{A'}$
$A : d_{A'}$ -> $B_2 : d_{B_2}$ $A : d_{A'}$
....
$A : d_{A'}$ -> $A : d_{A'}$ $B_n : d_{B_n}$

Whereas in the second case, we join the right hand side of each rule conforming to that format to the left hand sides of other rules and we delete the orphaned rules (where the LHS symbol never gets used on the right hand side. In other words, we duplicate the rules where the left hand side matches the right hand side of the root rule by replacing the left hand side with root node and we delete the unary rule because it becomes orphaned. However, we know that it is not that necessary to use the strict CNF to train CYK, so we use Extended CNF that can also allow for unary rules of the form A->B where A and B are nonterminals. This is to avoid duplication of rules which will make the grammar unnecessarily bigger and to make it easy to recover the trees once they are trained using CYK with a very simple trick that we explain in the next section. So, returning back to our grammatical rules in the example above, the rules after converting them to CNF become:

*ROOT:root -> VBG:**
VBG: -> NNS:nsubj VBG:**
VBG: -> VBD:aux VBG:**
VBG: -> VBG:* NNS:obl*
VBG: -> VBG:* .:punct*
*NNS:nsubj -> DT:det NNS:**
*NNS:obl -> IN:case NNS:**
NNS: -> DT:det NNS:**
NNS: -> NNP:compound NNS:**
NNS: -> NNS:* NN:nmod*
NNP:compound -> NNP: NNP:conj*
*NNP:conj -> CC:cc NNP:**
NNP: -> NNP:compound NNP:**
NNP: -> HYPH:punct NNP:**
*NN:nmod -> IN:case NN:**
NN: ->DT:det NN:**

Alongside those rules which are comprised of only non-terminals (variables), we add the unary rules necessary to prepare the terminals into non-terminals existing in the rules. For that purpose, we exhaustively assign for each pos-tag all possible dependencies relationships as they figure in the rules. So, for example, we notice that a word with pos tag NNS exists in the rules either as NNS:nsubj, NNS:obl, NNS:* thus we define the terminal rules as follows:

ROOT -> ROOT:root
*VBG -> VBG:**
VBD -> VBD:aux
NNS -> NNS:nsubj
NNS -> NNS:obl
*NNS -> NNS:**
NNP -> NNP:compound
NNP -> NNP:conj
*NNP -> NNP:**
NN -> NN:nmod
*NN -> NN:**
DT -> DT:det
IN -> IN:case
CC -> CC:cc
HYPH -> HYPH:punct
. -> .:punct

Figure 9 shows the tree constructed by following the above binary rules.

*4) **Training CYK Algorithm**:* Given as an input, the trained grammar and unary rules extracted from the unique binary trees, we perform the same usual CYK algorithm used for constituency parsing extended with minimal information to recover the trees. This non-probabilistic version keeps track of all possible trees as it goes up to extend the morphological charts with syntax information. Each cell contains a list of NonTerminals where each NonTerminal keeps tokenization (id in the sentence), morphology aspects (lemma, word form), syntactic information in the form of POS tags and pointers to recover NonTerminals in the lower level that built the head at the current cell. The nonterminals carrying a null pointer

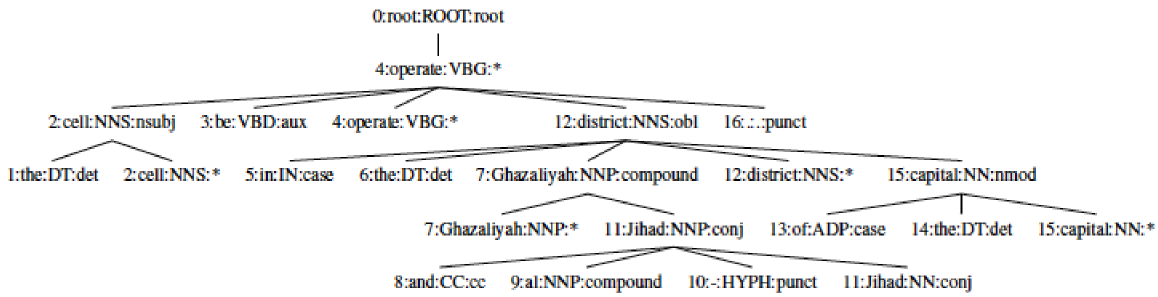

Fig. 6. An example of a dependency tree of a projective sentence

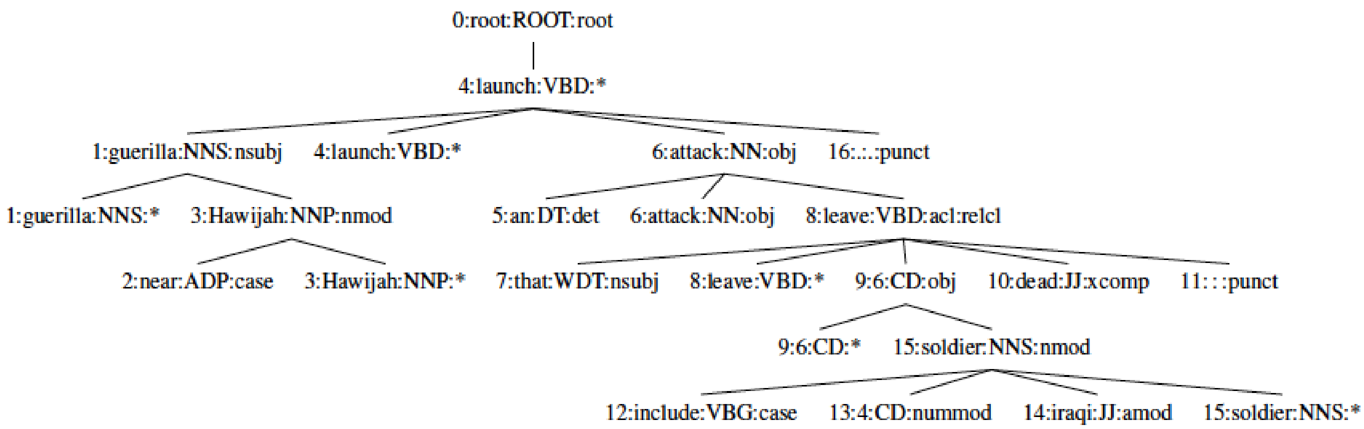

Fig. 7. An example of a dependency tree of a non projective sentence

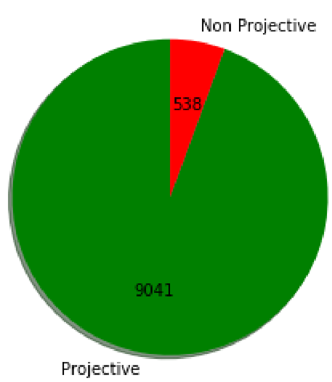

Fig. 8. The proportion of Projective Vs Non-Projective out of the total number of correcly tokenized sentences

are the original possible tokens that make up the sentence which come already filled as a result of morphological phase and to which the one sided rules are applied to make them nonTerminals. Unlike the conventional way in which the CYK is trained those cells are not necessarily all in the bottom of the chart some of them may occur at upper levels, this is why the use of a pointer is very helpful in this case as it distinguishes between nonterminals which are heads and those which are children of each dependency subtree in the chart. The algorithm belows describes the process followed which consists of three steps:

- Traversing the morphology chart and apply one-sided when necessary
- Traversing possible subtrees for each nonterminal in each cell in the chart by applying the binary rules and storing the pointers for backtracking
- Applying root rule at the upper leftmost cell and storing the root nonterminals in the same cell.

*5) **Probabilistic Extension**:* The probabilistic version of CYK for dependency parsing follows the same principles and structures except that instead of keeping all possible routes for every possible head that can constructed from many possible subtrees, we compute for only one distinct root for each distinct head by selecting the route that maximizes the score based on the product of the score of the grammatical rule applied and the accumulated product of the respective heads of the subtrees that lead to the combined tree constructed in the new cell. This probabilistic selection followed in all steps of the pCYK training (application of all one sided, binary and

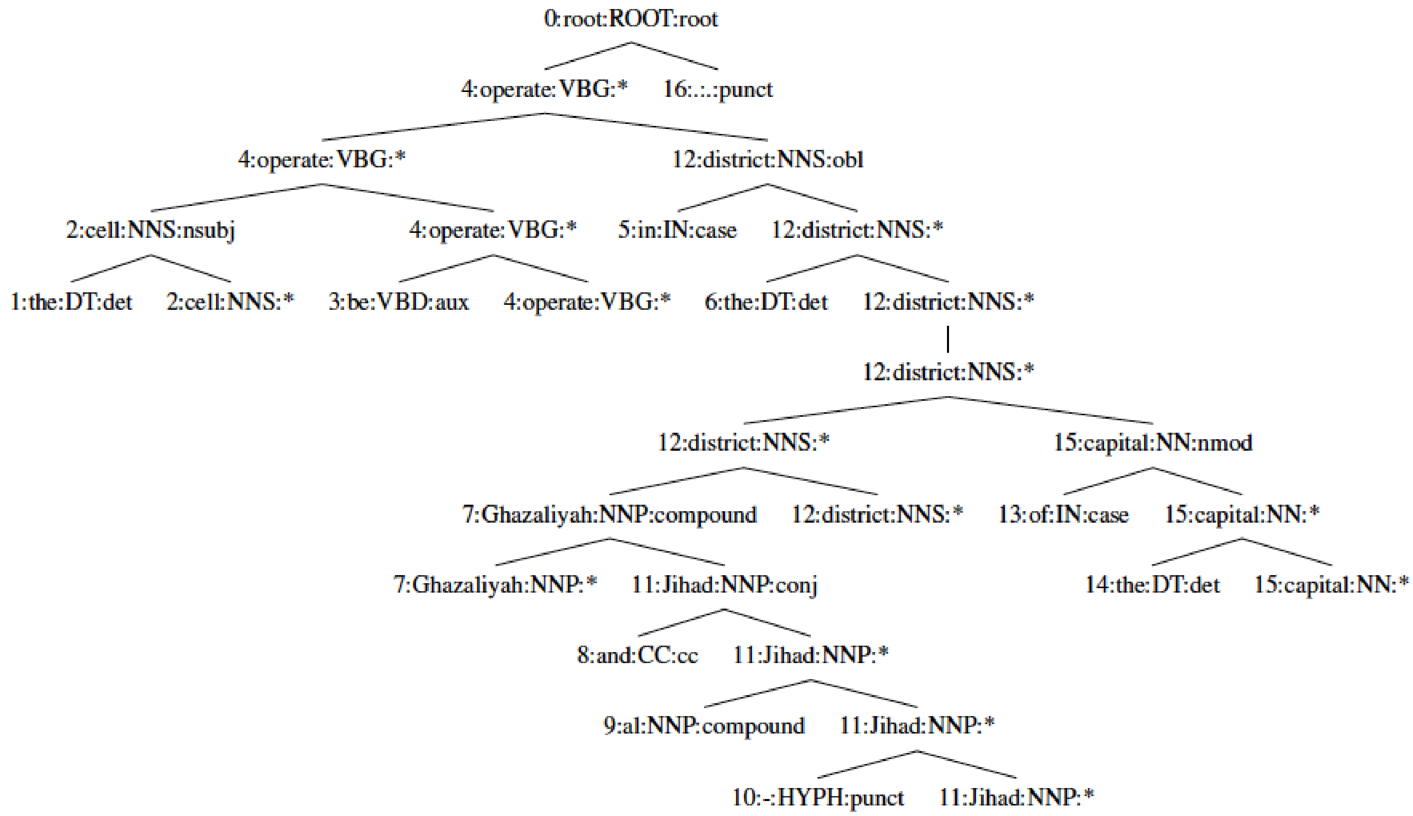

Fig. 9. Binary Dependency tree

root rules) as long as we have generated through a separate process the weighted probabilities related to the use of each rule. The figure 10 shows a partial pseudo code algorithm for binary rules followed to implement pCYK [13]

*6) **Recovering Syntactic Dependencies Structures**:* Once we get the output of CYK training, recovering the syntactic information can be performed depending on the type of information we would like to get. For checking for existence of a syntactic dependency tree, it suffices to check if the top and leftmost cell in the chart is not empty. This same cell can tell us about the number of possible trees in the chart by simply counting the number of root nonterminals. More tricky to check would be the existence of a correct tree. Even trickier is how many different correct trees exist. Since it is computationally infeasible to traverse and unpack all possible trees as the complexity grows exponentially from the top cell to the bottom, thinking about a simpler version to check for the existence of a correct tree in chart inspired the following methodology. We map each parsing chart to the golden truth CYK chart which we construct directly from the rules trained from only the corresponding sentence in the treebank. This will give us only correct trees in the golden truth CYK which we use to check against in the ones generated by our training algorithm. Starting from the top leftmost cell in the golden truth chart, if there exists many nonterminals we pick only one

```
for each max from 2 to n
  for each min from max - 2 down to 0
    for each syntactic category C
      double best = undefined
      for each binary rule C -> C1 C2
        for each mid from min + 1 to max - 1
          double t1 = chart[min][mid][C1]
          double t2 = chart[mid][max][C2]
          double candidate = t1 * t2 * p(C -> C1 C2)
          if candidate > best then
            best = candidate
      chart[min][max][C] = best
```

Fig. 10. Probabilistic CYK on binary rules

since all trees are correct. Then, we check in the parsing chart in the cell having the same position if the non-terminal that we picked in the golden truth chart matches one of those (the match should be complete including pointers match). If that is the case, we move on using the pointers of that the chosen non-terminal in the current cell and move on to check if there is a match between the non-terminal pointed to in golden truth CYK and any of the ones in the cell with the same location in the parsing chart. If there is no possible match at some cell, we stop the checking and return false. Otherwise, if it continue till it reaches null pointers, it returns true.

E. Semantic Analysis

Based on the dependencies structures extracted from the most probable tree that we got as a result of training the probabilistic chart parsing in the syntactic analysis, we can compute the word pairs similarities. The way to assign those probabilities to rules in our trained grammar is what needs our attention in this part. It is an iterative process that takes into account the impact of semantic co-occurrences to fine tune the syntactic probabilities.

As a result of the first pass over syntactic parsing, we get the pairs of lemmas for which at least one type of dependency exists. However, those dependencies are ambiguous and not all of them are correct for the specific context in which they appears in the sentences they were detected in based on the built parsing chart. At this stage, syntactic parsing only helped us filter out syntactically incorrect dependencies in the corpus (dependencies that cannot exist). However, there are still many ambiguities and the way to resolve them is by filtering them using their semantic relationships. For example, in the sentence "I ate the fish with bones", there is a dependency between eat and bone and bone and fish. We know that we cannot eat with bones as bones are not a utensil of eating, therefore the only way to filter out this incorrect dependency is by looking at the semantic relationship between eat and bone versus the relationship between bone and fish. The latter will weight more than the former as it will appear in more contexts. The idea in this stage is to use semantic knowledge as a golden truth to train syntactic tools by feeding the similarities as a way to compute syntactic probabilities till it becomes robust enough to generalize on new instances with unknown similarities.

This huge number of syntactic dependencies that we get as a result of the first pass will be used as a selective process to qualify some candidate lemmas which will substantially reduce the number of possible combinations for pairs for which it is relevant to compute semantic similarities. This training set is not enough for computing word/lemma co-occurrences because for word embedding to work, we need to train it on big enough corpus that covers large and varied instances of usages of the lemmas. So we take another big un-annotated corpus (for which we don't have the dependency structures) in which the same lemmas with some minimal co-occurrence threshold exist (we use the morphological analyzer to generate all possible surface forms of lemmas for which we would like to have a big corpus in which they appear by adding inflections corresponding to the different POS tags with which

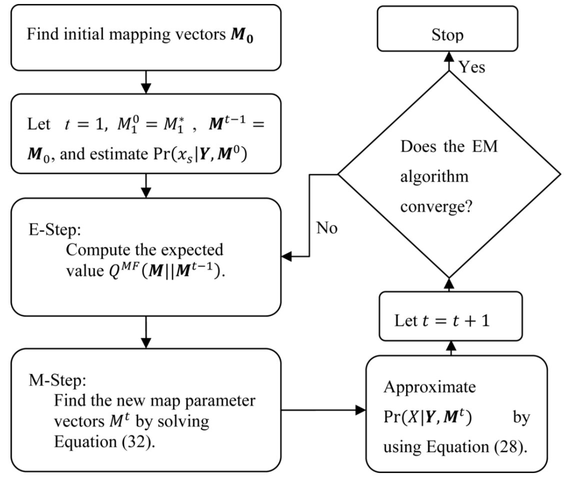

Fig. 11. General Expectation Minimization Algorithm

they appear in the training corpus). We apply the same NLP pipeline by applying the same grammar trained and tested with high accuracy and known to generalize well on an untouched subset of the small annotated corpus. We don't expect the NLP pipeline trained on the small training corpus to work perfectly well on the big corpus, but at least it will work for the subset of lemmas that matches the training corpus. Then, this corpus is used to initialize the counts of co-occurrence matrix without taking into account the grammar. Those counts are mapped into probabilities for grammatical rules which are used to train probabilistic CYK to disambiguate and to generate the dependencies structures again. This process is optimized using a forward backward approach like Expectation Minimization algorithm to fine tune those probabilities until convergence.

V. Results and Analysis

A. Lexical Analysis

*1) **Performance Evaluation Approach**:* To assess the quality of the tokenization, we proceeded by inspection. By taking random sample of sentences, we checked it by hand and looked if any of the following alarms were raised at some point:

- Unknown token false in the edges to assess the quality of tokenization part.
- Compute the number of tokens at the base of each chart and extract the top 1% shortest and top 1% longest and analyze the distribution of the lengths and look at the 1% most frequent and 1% less frequent to assess the quality of EOS.

*2) **Analysis of Results**:* The pie chart in figure 12 gives the proportion of sentences for there are correct, incorrect uniques charts, multiple charts and no charts. The number of sentences for which there exists one unique chart correctly tokenized and morphologically analyzed reaches 76.4% which tells us that our lexical process gave a correct analysis for the majority of sentences.

There are two types of limitations encountered in this part one has relation to the way the FSA was built and the second

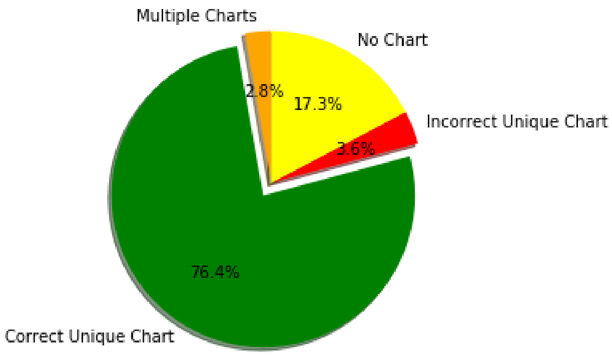

Fig. 12. Pie charts of Lexical Processing Accuracy

TABLE V. TOKENIZATION STATISTICS

	Treebank		Implicit Tokenizer	
	Train	Test	Train	Test
# Sentences	12543	2003	9602	1441
# Words/sent	21	12	21	17
Min Sent Length	1	1	1	1
Max Sent Length	159	75	221	163

one has to do with the tokenization methodology. The first is correctable by simply adding, removing or modifying lexical entries are the most trivial but also cumbersome to control as there are countless regexes and possible word entries. The second is not correctable by adjusting lexical entries. The major sources of limitations with the current tokenization approach has to do with entries in tokFSA that end with period or any other separator which takes an integral part of the lexical entries such as in Mr., Mrs., Corp., U.S., etc, The problem in this case is that the tokenizer doesn't distinguish between the case where the separator is a part of the token or when it is a possible EOS separator that is taken into consideration for building a chart. The same applies to cases that commonly occur in natural language where a proper sentence not only ends with an EOS character but also with another bracket, parenthesis, quotes etc such as in the sentence: She said "I didn't go the to the party yesterday." In this case, a chart is build for the sentence up to . without including the ending quote ".

The main limitations encountered while running the implicit version of the tokenization algorithm have to do with the EOS mechanism. These can be summarized in the following cases:

- **Sentences ending with an end of sentence (EOS) character followed with either parenthesis, double/single quote or any character which is not EOS:** for e.g.:
  - *This willingness is the main difference in the number of bombings in the south as opposed to the center - north of the country .)*
  - *" Can you name the general who is in charge of Pakistan ? "*

  In this case, the algorithm returns the sentence prior to the occurrence of EOS character ignoring the characters after, that is any additional parenthesis, quote, etc.
- **Sentences not containing with any EOS characters not covered with any further reaching arc such as in the sentence:** for e.g.:
  - *"Just remember that death is not the end - Bob Dylan"*
  - *http://www.ibrae.ac.ru/IBRAE/eng/chernobyl/natrep-/natrepe.htm#24*

  In those cases, no chart is returned. For example, the whole sentence http://www.ibrae.ac.ru/IBRAE/eng/chernobyl/natrep-/natrepe.htm#24 is recognized as an URL. So, although it contains some EOS characters, they are not considered as true EOS since they are all covered by the further reaching arc which is that of the URL.
- **Sentences containing an EOS character but not at the end and not covered with any further reaching arc:**
  - *Web posted at : 3:29 p.m. EST (2029 GMT)*
  - *WHERE : The Front Porch 217 Gray St. (713) 571-9571*

  Those sentences are tokenized up to the EOS character that is not included in any further reaching arc as shown in VI and VII leaving up the other remaining words after the EOS considered trusted in this case to allow the building of a chart.
- **Sentences containing EOS character both in the end and the middle not covered with any further reaching arc:** for e.g.:
  " The Prophet 's guidance , " says Michael Scheuer , an al - Qaeda analyst who recently retired from the CIA and once headed its Bin Laden unit , " was always , Before you attack someone , warn them very clearly ... " The anthrax mailings followed the pattern of letters they sent in January 1997 to newspaper branches in Washington , D.C. and New York City , as well as symbolic targets.
  In this case, many different charts are returned as shown in the following charts in VIII, IX, X

TABLE VI. EXAMPLE SENTENCE WITH TOKENIZATION LIMITATION

							p.m.			
				3:29						
Web	posted	at	:	3	:	29	p	.	m	.

TABLE VII. EXAMPLE SENTENCE WITH TOKENIZATION LIMITATION

							St.	
WHERE	:	The	Front	Porch	217	Gray	St	.

TABLE VIII. EXAMPLE OF TOKENIZATION LIMITATION CHART 1

								....		
								....		
"	The	prophet	's	guidance	,	"	says	....	clearly	...

TABLE IX. EXAMPLE OF TOKENIZATION LIMITATION CHART 2

							D.C.			
"	The	anthrax	mailings	followed	the	....	D	.	C	.

TABLE X. EXAMPLE OF TOKENIZATION LIMITATION CHART 3

and	New	York	City	,	as	well	as	symbolic	targets	.

TABLE XI. SYNTACTIC RESULTS

		Train	Test Pass with no Addition	Test Pass with 20% Addition	Test Pass with 90% Addition
No Tree		0%	5%	4%	2%
One Tree	*Correct*	0%	0%	0%	0%
	Incorrect	0%	0%	0%	0%
Many Trees	*Correct Compatible*	100%	5%	30%	95%
	Correct Incompatible	0%	90%	66%	3%
Total Accuracy		100%	5%	30%	95%
Maximum Number of Trees		383	381	401	466
Minimum Number of Trees		4	0	0	0
Average Number of Trees		242	267	284	331

B. Syntactic Analysis

*1) **Performance Evaluation Approach**:* To test the generalization power of the grammar and to accommodate the fact that it is still not computationally efficient to run all the CYK on a grammar trained on the whole training set of 12K, we select a random sample from the training set of 500 sentences correctly tokenized and lemmatized. Then, we shuffle this set and split it randomly into training and testing set consisting of 400 and 100 sentences respectively. We train the grammar on 400 sentences and keep the remaining 100 untouched. To check the quality of parsing, we start by computing how many sentences don't have a correct tree in the training set of 400. This number serves as a debugging method to tell us what went wrong with the grammar training, the CYK implementation and the way we check for the existence of one correct tree. After getting 100% sentences containing all of them a correct tree in the training set of one sample 400, we don't stop here but we continue debugging by running the same procedure on another randomly generated 400 sentences for 10 times. This allows us to be more sure about the correctness of the training process. Then, we apply the same grammar generated from the grammar generated at the last iteration to the testing subset. We report on the resulting accuracy for each iteration in which we keep injecting the minimal number of the sentences to be put in the training set by continuously checking for each testing sentence whether it has a tree, otherwise, we put it back to update the grammar and check again to see how this impacts the generalizability of the grammar by checking if the testing accuracy increases. If the accuracy is still not 100%, we check the next sentence that doesn't have a tree and we inject it back in the grammar and recheck again and so on.

*2) **Analysis of Results**:* In tables XI, we summarize accuracy statistics of syntactic analysis in terms of the percentages of training and testing sentences are syntactically correct. We also detailed the proportion of sentences for which many syntactic trees are generated vs the proportion of sentences for which no or only one dependency tree is recognized, and the accuracy is reported for each category and sub-category, in addition to general statistics about the minimum, maximum and average number of trees generated per sentence. In sum, we notice that we either get no tree or many trees which implies the high level of ambiguity of the grammar and the many morphological interpretations of the tokens. This suggests that probabilistic extension is required to filter out incorrect dependencies. Although the training grammar has been debugged extensively to reach 100% on the training set, its ability to generalize with the same level of accuracy is relatively weak with only a maximum of 5% of trees on small testing samples chosen randomly in 10 iterations (we performed the testing separately on 10 random samples). Adding 20 instances chosen randomly from the testing set itself and feeding it back to train the grammar increased the accuracy to 30% which means that apart from the 20% which is expected to be syntactically correct(since we added 20 instances out of 100 instances from the testing sample) 5% more unknown sentences were recognized but still 4% have no tree at all. Even after adding 90% of testing instances, the accuracy still didn't converge to 100% which implies the variability of the testing sample and is also largely due to the very small training size used.

VI. CONCLUSION

In this project, we implemented a full, clean, efficient, fully debuggable and incremental NLP pipeline well adapted to the problem of semantic similarity on a large scale. We have designed and used a tokenization algorithm that based on FSA traversal processes any input and stores all possible tokens in an efficient way. Then, we have used FSTs to map the surface forms to their cannonical forms. We have implemented a well-thought of methodology modified version of CYK trained on the constructed dependency grammar. We have evaluated the quality and accuracy of those three steps by splitting the data into training and testing subsets and using various checks. All in all, the tokenization process reaches a good performance except for EOS mechanism which sometimes returns more or less sentences than the true cuttings in the treebank. Syntactic Analysis returns syntactically compatible parses for sentences using a grammar to which the associated grammatical rules have been added. The role remains for semantic analysis to fine tune weights to be fed to the probabilistic component of CYK to resolve Ambiguity and eventually compute relevant co-occurence scores to be used and analyzed in large scale applications.

VII. Challenges Encountered and Future Work

One of the most difficult challenges to surround in this project is how to define ambiguity in each stage in the NLP pipeline, store it efficiently and at which stages it makes more sense to solve it. Dealing with ambiguity is innate to natural language processing. While keeping all possibilities is the most accurate way to deal with it, it is not that efficient for a large-scale application. A good intuition could be not to rely only on theoretical hypothesis but also to try with different models of disambiguiting the processing output at different stages and to compare their performance experimentally. One such potential hypothesis advocated theoretically in this project is that semantic co-occurrences computed based on syntactic dependencies can be better fine tuned at the semantic level rather than using probabilistic theory at the syntactic level which needs to be verified and validated in future work. Also, more attention needs to be given to embed the NLP pre-processing architecture into the way co-occurrences scores are computed and fine-tuned using supervised approaches relying on bigger corpus. Big data is not only needed in the semantic level but also to train syntactic tools reliable for big scale that can capture the variability of dependency structures and generalize well enough to unknown data. More attention needs to be given to either implement a more efficient version of chart parsing algorithm or adapt an existing version to make it compatible with the form of inputs and outputs.

But before jumping into the semantic level, more robust tools against bad input need to be built especially in the tokenization phase using edit distance. With the lack of free and big annotated corpuses and the submerging availability of information systems and social media and other forms of easy and fast communication, dealing with mispellings and not so well formed sentences is a crucial aspect of any large scale application. The way to building a tool that is robust enough and at the same time efficient and keeps assumptions deployed at their minimum is through setting up the necessary building blocks in such a way that allows incrementality and troubleshooting come with the minimum cost possible.

VIII. Acknowledgment

I would like to express my gratitude to my supervisor Professor Dr. Martin Rajman for suggesting and explaining this project to me and for his patient guidance, enriching discussions, enthusiastic encouragement and immeasurable support.

References

[1] Wang, M., Cer, D. Stanford: Probabilistic Edit Distance Metrics for STS. Computer Science Department. Stanford University. https://nlp.stanford.edu/pubs/wang-cer-sts12.pdf [2] Stanford Tokenizer.The Stanford Natural Language Processing Group. (2013-2017) https://nlp.stanford.edu/software/tokenizer.html

[3] Aho, A., Lam, M., Sethi, R., Ullman, J. (2006) Compilers: Principles, Techniques, and Tools. Prentice Hall. (2nd Edition).

[4] Mikolov, T., Chen, K., Corrado, G. Dean, J. (2013). Efficient Estimation of Word Representations in Vector Space. CoRR, abs/1301.3781.

[5] Mikolov, T., Sutskever, I., Chen, K., Corrado, G.-s., Dean, J. (2013). Distributed representations of words and phrases and their compositionality. Advances in neural information processing systems, 3111-3119

[6] Mikolov, T., Yih, W.-t. Zweig, G. (2013). Linguistic Regularities in Continuous Space Word Representations.. HLT-NAACL (p./pp. 746–751), .

[7] Levy, O., Goldberg Y. (2014) Dependency-Based Word Embeddings. Proceedings of the 52nd Annual Meeting of the Association for Computational Linguistics, pages 302-308. Association for Computational Linguistics. http://www.aclweb.org/anthology/P14-2050

[8] Qiu, L., Zhang, Y., Lu Y. (2015). Syntactic Dependencies and Distributed Word Representations for Chinese Analogy Detection and Mining. Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing, pages 2441-2450. Association for Computational Linguistics. http://www.emnlp2015.org/proceedings/EMNLP/pdf/EMNLP291.pdf

[9] Silveira, N., Dozat, T., de Marneffe, M.-C., Bowman, S., Connor, M., Bauer, J. & Manning, C. D. (2014). A Gold Standard Dependency Corpus for English. Proceedings of the Ninth International Conference on Language Resources and Evaluation (LREC-2014). http://universaldependencies.org/introduction.html

[10] MÃÿller, A., et al. dk.brics.automaton (2011). Aarhus University. http://www.brics.dk/automaton/index.html

[11] Schmid, H. (2005) A programming Language For Finite State Transducers. Institute for Natural Language Processing (IMS). University of Stuttgart. http://www.cis.uni-muenchen.de/ schmid/tools/SFST/

[12] Goddard, W. Chomsky Normal Form. Theory of Computation. https://people.cs.clemson.edu/ goddard/texts/theoryOfComputation/9a.pdf

[13] Stymne,S.. The CYK algorithm part 2: Probabilistic Parsing. http://stp.lingfil.uu.se/ sara/kurser/5LN455-2014/lectures/5LN455-F3.pdf

Other References:

[14] Kaplan, R. M. A Method for Tokenizing Text

[15] Recursive Auto-encoders would benefit from using phrase vectors instead of word vectors.

[16] A Scalable Hierarchical Distributed Language Model Mnih and Hinton, 2008

[17] Hierarchical Probabilistic Neural Network language Model
Morin and Bengio, 2005

Algorithm 4 Implicit Tokenization Algorithm based on FSA Traversal (continued)

Traverse $sepFSA$
if exists in $sepFSA$ a transition labelled with character at position i in string str **then**
if FINAL STATE **then**
if exists in $sepFSA$ a transition labelled with character at position $i+1$ in string str **then**
$i \leftarrow i+1$
$bottomChartFlag \leftarrow true$
else
$flag_sep \leftarrow false$
if separator is persistent **then** Add edge from $start$ to $i+1$ to bottomChart Add edge from $start$ to $i+1$ to arcs
$bottomChartFlag \leftarrow false$ Re-Initialize $sepFSA$ to its start state
if separator is EOS and exists no further reaching edge covering edge of the separator **then** Build Chart
else
$i \leftarrow i+1$
$start \leftarrow i$
end if
end if
end if
else
$i \leftarrow i+1$
end if
elseAdd edge from $start$ to $i+1$ to bottomChart with flag unknown
$i \leftarrow i+1$
$start \leftarrow i$
end if
end if

Comment: Tokenization Chart is built from indices array in the same way as in Explicit Solution

Output: Tokenization Chart = 2 dimensional array of edges where each edge is marked by a start index,end index (not to be considered) and boolean value for whether it is recognized by the lexicon or not.

Algorithm 5 Training Unique Non-Binary Dependency Parsing Grammar

Input: List of dependencies structures of a sentence

$visited \leftarrow$ stack holding the children which are the potential heads of the sub-trees to searched for
$dependents \leftarrow$ nodes that have already been found as dependents to some head node
$rules \leftarrow$ output rules
$child \leftarrow$ child of root
Add $child$ to $visited$
Add root:ROOT -> $child$:* to $rules$
while $visited$ not empty **do**
    $first \leftarrow$ popped element of $visited$
    $children \leftarrow$ list of children of the $first$
    $newRule \leftarrow$ the new grammatical Rule
    $rightHandSides \leftarrow$ the list of right hand side nodes of $newRule$
    if $dependents$ contains $first$ **then**
        $deprel \leftarrow$ deprel of $first$ in $dependents$
    else
        $deprel \leftarrow$ *
    end if
    $head \leftarrow first$ with modified $deprel$
    Add $head$ to $rightHandSides$
    for each $child$ in $children$ **do**
        Add $child$ to $visited$
        Add $child$ to $rightHandSides$
    end for
    Sort $rightHandSides$
end while=0

Output: List of non-CNF Grammatical Rules

Algorithm 6 Checking for Projectivity of a dependency Tree

Input: List of grammatical rules in their non-CNF $grammaticalRules$ and list of ordered IDs of the ground truth dependencies $truePaths$

$paths \leftarrow$ Map of IDs of heads to their children
$flatten \leftarrow$ List of IDs after flattening $paths$ to get the ordered IDs the leaves
$visited \leftarrow$ List of IDs heads that have already been replaced by their corresponding children
Add 0 to $flatten$
while $|visited| < |grammaticalRules|$ **do**
    for $element$ in $flatten$ **do**
        if $element \in paths$ and $\notin visited$ **then**
            Add $element$ to $visited$
            for $child \in values$ of $element$ in $paths$ **do**
                Add $child$ to $flatten$
            end for
        else
            Add $element$ to $flatten$
        end if
    end for
end while
if $flatten == truePaths$ **then**
    return $true$
else
    return $false$
end if

Output: Boolean Flag

Algorithm 7 CYK Dependency Parsing

Input: morphology chart $parsingChart$, Hashmap of nonterminals $binary$, one sided rules $unary$ and root rules $unaryRoot$

```
for 0 <= i < n do
    for 0 <= j < n do
        for X−> x ∈ unary do
            for parsingChart[i][j] = x ∈ parsingChart[i][j]andXnotinparsingChart[i][j] do
                Add X to parsingChart[i][j]
            end for
        end for
    end for
end for
for 1 <= i < n do
    for 0 <= j < ni do
        for 0 <= k <= i − 1 do
            for Z−> XY ∈ binary do
                for X ∈ parsingChart[i − k − 1][j]andY ∈ parsingChart[k][i + j − k] do
                    Add Z to parsingChart[i][j]
                    if j < i + j − k then
                        pointer ← i − k − 1        ▷ This is used as pointer to the number of the row that contains X
                    else
                        pointer ← k
                    end if
                    right1Index ← order of X in parsingChart[i − k − 1][j]
                    right2Index ← order of Y in parsingChart[i][j]
                end for
            end for
        end for
    end for
end for
for Z−> X ∈ unaryRoot do
    for X ∈ parsingChart[length − 1][0] do
        Add Z to parsingChart[length − 1][0]
        pointer ← length − 1 ▷ This is used as pointer to the number of the row that contains X which is itself in this case as the root is stored in the same cell as its dependent
        right1Index ← order of X in parsingChart[length − 1][0]
    end for
end for
```

Output: Tokenization Chart extended with Dependencies