Large Scale Dependency-Based Word Embeddings Implementation

Marc Adams, Dr. Martin Rajman
Department of Computer Science, EPFL Lausanne, Switzerland
{marc.adams, martin.rajman} @ epfl.ch

***Abstract*—Word embeddings algorithms such as word2vec and fastText gained massive popularity, as they can be applied to huge unannotated corpuses in an efficient way. A possible amelioration of those algorithms is exploiting more linguistic features. Work has already been done in this domain, with very encouraging results. In this work, we build a full NLP-pipeline that implements dependency based word embeddings. The main goal is to reach enterprise level code quality of an NLP-pipeline that can easily be used and extended to experiment new techniques.**

I. Introduction

Representing word semantics is a real challenge. The most efficient way today are word similarities. Word similarities cannot formalize the definition of a word as we would read it in the dictionary, they simply tell us which words are semantically close. What does semantically close mean? This is not so well defined. For example in [2], the algorithm put countries and their capitals as semantically close. In [3] *Einstein* and *scientist* are semantically close, as well as *big* and *bigger*. The point is, all these pairs come from different kind of relations: knowledge of capitals, knowledge of personalities and a comparative relation. In that sens, semantically close means "some semantic relation". Word similarities is a powerful tool though, it has applications in: request expension for search engines, automatic translation, summarization algorithms, etc.

Word2vec and fastText are the most popular algorithms to learn vectorial representation of words. They are based on the idea that similar words appear close together in sentences. They are capable of handling billion word long corpuses, in a matter of days. A general definition of word embeddings could be: given $(word, context)$ pairs, find a semantic representation of the words. These techniques are also referred to as distributional semantics. In word2vec and fastText, for a word w_i appearing at position i in the input text, the k closest words $w_{i-k}, w_{i-(k-1)}, ..., w_{i-1}, w_{i+1}, ...w_{i+k}$ are used as the context of word w_i. Such a context is called linear context, or CBOW-k (Continuous Bag Of Words of size k).

In [4], the word2vec model is extended to use costum contexts instead of the linear contexts. They then use the dependency trees of the sentences to derive contexts of words, we refer to this as dependency based word embeddings. Their results are very motivating, they are shown in figure 1. The task being tested is: given a word, rank the most similar words. The WordSim353 dataset and Chiarello et al. dataset are used as the reference for word similarities. Their dependency based word embeddings (the *Deps* curve) is compared to CBOW-2 and CBOW-5 (the $K = 2$ and $K = 5$ curves). For this task, the dependency based contexts performed better. As they explain, that does not mean dependency based word embeddings are better than CBOW-5, the dependency based technique managed to extract more topic based representation of words, while CBOW-5 extracted larger definitions of words. What this shows us though, is that dependency based word embeddings have the potential of extracting interesting new similarities. There is more or less only CBOW that can be applied to extract contexts from a tokenized text, but there are many ways of extracting contexts from dependency trees. Using dependency trees, the input of word embeddings can be filtered, concentrated on some aspect to extract specific relations. New algorithms, different than word2vec and fastText can be tested.

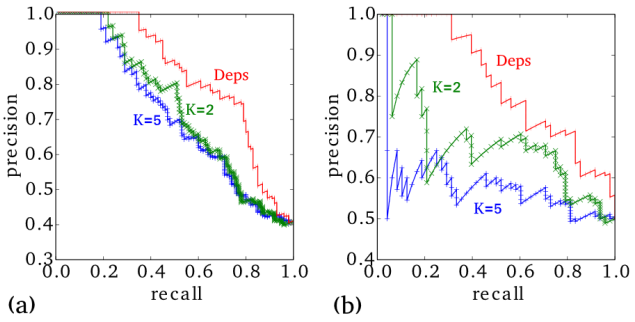


Fig. 1. Dependency based word embedding results compared to linear contexts obtain in [4]. (a) is based on the WordSim353 dataset, and (b) is based on the Chiarello et al. dataset

These results are part of the motivations for building a full NLP-pipline, with the focus on being: easily extendable, easily debuggable and deployable on clusters. Very good full NLP-pipelines already exist, like the Stanford CoreNLP and NLTK frameworks. Our framework distinguishes itself from the others in many aspects:

- Ambiguities are kept as long as possible, the idea being

solving ambiguities in the semantic layer is closer to how humans approach text
- The algorithms are configurable outside of the code as much as possible, meaning a fair amount of experimentation is possible without touching the code
- Modules can be easily replaced with new ones, as the communication between them are fixed
- The input text to the system is annotated, which allows communication to the system at runtime. With this system, exceptions in the input text can be handled easily.

Carrying on the work of Meryem M'hamdi [1], who developed the ideas of the project, this paper is meant to describe in details the concepts and the structure of the NLP-pipeline we developed. First in II we will describe the high level structure of the framework, then in III, IV, V, and VI we will describe the main modules of the nlp stack, and finally in VII we will describe how the full system behaves and uses the modules. To execute the code, please refer to the README.md on the github. Details about the code can be found as javadoc in the code.

II. High level Structure

The structure of the project is oriented towards big data. Word embeddings is a part of NLP that needs huge corpuses to converge to satisfactory results. The project had to be thought from the start in that aspect. The main communication interfaces between modules are therefore streams. Another aspect that was thought from the start is the possibility to easily extend the code for other uses. For this reason, the project is divided in two main packages: *nlpstack* that defines all the high level interfaces, and *implementations* which is the word embeddings implementation of the interfaces.

The framework is written in java. Why java? Simply because most popular big data frameworks, like spark, flink, kafka and cassandra have java/scala as there api.

In the next two sections we will describe the contents of each the *nlpstack* and *implementations* packages.

A. nlpstack package

The nlp stack package describes all the high level interfaces: What kind of modules are expected, and what is communicated between the modules. The stack is represented in figure 2. *LexicalAnalyzer*, *SyntacticAnalyzer* and *SemanticAnalyzer* are, as their name indicate, the algorithms responsible for lexical, syntactic and semantic analysis. There interfaces are described in *nlpstack.analyzers*. The arrows between the modules represent the objects being communicated. All the objects being communicated are enclosed in the java *Stream<>* interface. The streams objects of flink and spark can be wrapped in a *java.util.Stream* object and directly use in the system.

StringWithAnnotations is simply the input string where annotations have been parsed. The implementation of the Annotation parser is in *nlpstack.annotations*. The annotations form an important part of the stack, as they allow the system to be influenced at runtime. They are described in details in III.

Between LexicalAnalyzer, SyntacticAnalyzer and SemanticAnalyzer, we decided to use charts as the communication. A chart represents ambiguity in a compressed way. Using them lets us keep the ambiguities as long as possible in the system. LexicalChart is simply a chart that has been initialized with the tokens of the sentences of the input text, and a SyntacticChart is a filled in chart. As the algorithms are applied on streams, the order of the sentences of the input is lost. LexicalChart and SyntacticChart keep information on the annotations, and can be used in the future to keep metadata on the input. It might be interesting to mark each sentence with its origin to be able to recover the order of the sentences.

The object *nlpstack.communication.ErrorLogger* is used to log errors. It simply uses log4j2 as the underlining logging mechanism, but it defines one method per layer, and each method asks for the object that is at the origin of the problem, which could be useful in the future. The configuration for log4j2 can be found in *src.main.resources*. Error logging is also a very important aspect of the system. If the system is deployed on a cluster and is processing data continuously, it should not stop in case of problem, it should just log the errors and continue processing the data. *log4j2* is also a standard, it's flexible, and is already integrated in frameworks such as spark.

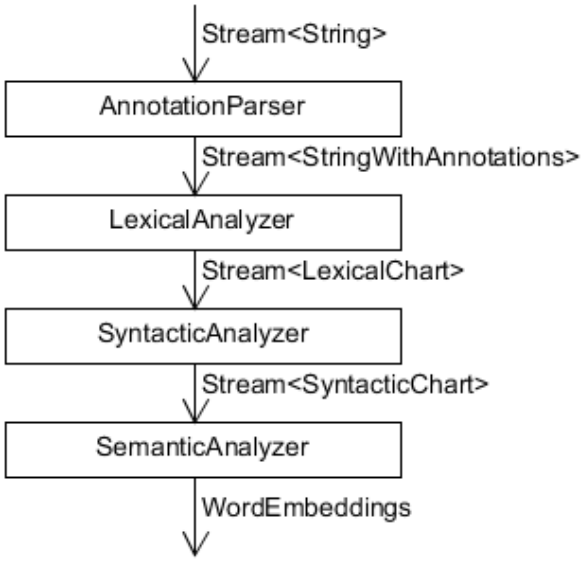


Fig. 2. Main NLP Stack

B. implementations

What fills the gap between the *nlpstack* and the implemented objects is the *Configuration* object. It is responsible for parsing the configuration file, *conf-*.yml* in the current code, and returning the lexical, syntactic and semantic analyzers to the main nlpstack. The interface is defined in *nlpstack.Configuration*, and the current implementation is in *implementations.DefaultConfiguration*.

In the *implementations* package, *DefaultLexicalAnalyzer* and *DefaultSyntacticAnalyzer* implement the lexical and syntactic analyzers. *confile* are the objects used by *jackson* to read the *conf.yml* files. *filereaders* contain the objects responsible of loading FSA's and Grammars. *fomawrapper.FomaWrapper* is an object that can read and write through pipes with foma. *sparkutils* is responsible for instantiating a spark instance, and contains notably a Kryo serializer with Ndj4 support in the view of deploying the system on a cluster. *ConlluUtils* is a small program to get stats on conllu files and that can extract grammar rules and words from these files. *WordEmbeddingsUtils* reads conllu files and trains word embeddings on the extracted dependency trees.

For the system to work, some bootstrapping has to be done. This is where conllu files come in. The format of conllu is defined in more precision on the universal dependencies website http://universaldependencies.org/format.html . This is what *WordEmbeddingsUtils* and *ConlluUtils* are meant for. We will describe this in section VII in more details.

III. Annotations

Annotations are an easy way of influencing the system at runtime. If the system was perfect, they would not be needed, but this is probably not the case. The annotations can indicate: an end of sentence, the grammatical tag of a word, a lexical chart or a syntactic chart of a full sentence. The full implementation is in *nlpstack.annotations*, it was written with *antlr*. The formal grammar of the annotations is written in *nlpstack.annotations.AnnotationParser.g4*

An example of an annotated input is given in figure 3. The annotations all start with a # character, and the contents are inclosed in *[]* brackets. *#["annotated", "V"]* means that the word *annotated* will take the tag "V" when it is initialized in the lexical chart. *#[".", "EOS"]* forces the system to consider this dot as a End Of Sentence. The tags *EOS* and *UNKNOWN* are reserved. *EOS* means the corresponding word should be considered as an End Of Sentence token, and the lexical layer will split the input on that token. *UNKNOWN* is the tag written when the transducer did not recognize the input. The *#[syn: ...]* on line 06 specifies a chart at the syntactic level that will directly go to the semantic level, while *#[lex: ...]* is a lexical chart that will be processed by the syntactic level.

In other words, for the input of figure 3:

```
01 This is an annotated string example
02 with inline #["annotated", "V"] words
03 #[".", "EOS"]
```

will be sent to the lexical layer as a *List<StringSegment>*. The sentence:

```
03 Annotations can also
04 contain full lexical and syntactic
05 charts for sentences:
```

will also be sent to the lexical layer. The chart:

NP	
DET	N
a	chart

is a syntactic chart, so it is considered as a full chart and will be sent to the semantic layer, and the chart:

DET		N
a	green	card

is a lexical chart, it is just an initialized chart that will be sent to the syntactic layer.

The escape character is "\", it removes the meaning of any special character. For example *\#[".", "EOS"]* will not have any special meaning, and will just be considered as the string *#[".", "EOS"]*. This can also be used inside annotations. For example: *#["\"", "EOS"]* will mean that *"* is an EOS character. \\ is a backslash without any special meaning.

```
01 This is an annotated string example
02 with inline #["annotated", "V"] words
03 #[".", "EOS"] Annotations can also
04 contain full lexical and syntactic
05 charts for sentences:
06 #[syn:
07   (a)             (chart) :
08   (1, 1, "DET") (1, 2, "N")
09   (2, 1, "NP")
10 ]
11 #[lex:
12   (a)             (green) (card) :
13   (1, 1, "DET")
14                   (2, 2, "N")
15 ]
```

Fig. 3. Annotated input example

IV. Lexical analysis

This layer is responsible for three things:

- Tokenize the input
- Detect and split the input in sentences
- Associate grammatical tags to tokens

The algorithm used is the one described in [1]. As a running example to illustrate the algorithm, we will apply the algorithm by hand on the input string:

Mr. Smith's green card. It's a 'nice' cat.

This sentences have no meaning, they were chosen for 3 different interesting problems:

- The algorithm has to differentiate the point of *Mr.* with the one separating the two sentences
- *green card*, *Smith's* and *it's* are composed words, and have to be considered as such
- The ' symbol is used in *it is*, but it is also used as a stylistic effect on *'nice'*.

First we will present the example, then we will describe and apply the algorithm by hand on the example, and finally we will give additional information on the implementation choices.

A. The running example

For an input string x, the output of the lexical analyzer is one or many charts, one chart per sentence found. Each chart is initialized with the correct tokens, they are represented in figure 4. We represent the charts as triangular tables. The algorithm found the two sentences of the running example, so it outputs 2 charts, marked with (a) and (b). The tokens are at the bottom of the charts, every other cell is reserved for grammatical tags. Cells are identified by $(length, position)$ tuples. For example, in chart (a), the token Smith has two *N* grammatical tags above it. *N* stands for noun, the lowest one is identified by $(3, 1)$, because *Smith* is the 3rd token from the left, and the single word *Smith* is a noun. The *N* above is identified by $(3, 3)$ because the sequence of tokens *Smith's* is of length 3, starts at position 3, and can be considered as a noun.

As we can see, all the ambiguities have been left in place. Is *green card* one or two words? Is the point of *Mr.* the end of a sentence or just part of the token? In the chart, both are possible and it will be up to the lexical and semantic layers to decide which is the correct interpretation. A chart can contain one or multiple sentence. For example, if the point of *Mr.* is a end of sentence token, then the chart (a) will contain two sentences. The grammar used in the syntactic analyzer will have to be written in a way that can detect multiple sentences. We could therefor think that splitting the input in sentences in the lexical layer is useless, but there are two good reasons for doing this here:

- the complexity of the syntactic algorithm is in $O(n^3)$ per chart, while the lexical one is $O(w)$, with n the number of tokens in the chart and w the length of the string input.
- the more sentences we detect from the start, the more we can parallelize the execution of the syntactic layer, as each chart can be processed independently of the others.

We will discuss this further in section V.

(a)

		N					
N			V		N		
N	PUNCT	N	AP		N	N	PUNCT
Mr	.	Smith	'	s	green	card	.

(b)

	V							
PRON	AP		DET	AP	ADJ	AP	N	PUNCT
It	'	s	a	'	nice	'	cat	.

Fig. 4. Output of the algorithm for the example sentence

B. Applying the algorithm on the example

There are 6 fields that come in the configuration of the lexical layer:

- wordFSAPath
- separatorFSAPath
- eosSeparatorRegex
- invisibleCharacterRegex
- FomaBinPath
- FomaConfPath

Where *wordFSAPath* and *separatorFSAPath* are paths to finite state automaton files with *.txt* or *.ser* extension (one for the textual form, the other for the binary form). We will refer to the finite state automatons as wordFSA and sepFSA. *eosSeparatorRegex* and *invisibleCharacterRegex* are two regular expressions, that we will refer as eosReg and invReg. *FomaBinPath* is the path to a foma binary, and FomaConfPath is a path to a *.lexec* or *.foma* file.

Before presenting the algorithm we need to defined arcs. An arc is a just a tuple with a start position and an end position. We use them to delimit sub-strings in the main input string. The result of the arc discovery algorithm on the first sentence is shown in figure 5. The arcs are represented as arrows above the input string. The final arc of the first sentence continues on the next line and finished on the I of the second sentence. The extremities of each arrow show the starting and ending position of the arcs discovered in the input. An arc t with start and end

position (i, j) means there is a corresponding token starting at position i included and finishing at position j excluded. For example, at the beginning of the string we have the arcs $(1, 3)$ and $(1, 4)$ which corresponds to the tokens *Mr* and *Mr.* .

Arcs can be of 5 different types:

- *TOKEN*
- *VISIBLE_SEP*
- *INVISIBLE_SEP*
- *VISIBLE_EOS*
- *INVISIBLE_EOS*

WordFSA accepts strings delimited by *TOKEN* arcs, which we will call token arcs, and sepFSA accepts the strings delimited by all the other arc type, which we will call sep arcs. The sub-strings delimited by *VISIBLE_EOS* and *INVISIBLE_EOS* arcs match with the eosReg regex. The sub-strings delimited by *INVISIBLE_SEP* and *INVISIBLE_EOS* arcs match with the invReg regex. A token arc is valid only if it is followed by a sep arc, or the end of the string.

In the step by step execution of the algorithm, we will consider that the configuration of the lexical analyzer is very similar to the *conf-*.yml* conf files in the example directory. The running example can therefor be tested and played around with by executing the code.

There are 4 steps in the algorithm:

1) Arc discovery: To find the arcs, we alternate between all match with wordFSA, and longest match with sepFSA. We use a fifo queue to keep track of the matches we found.

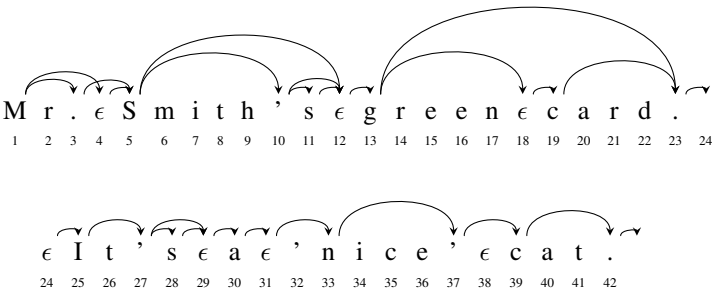
On the example, the algorithm would:

- from position 1, execute all match with wordFSA
- the token arc $(1, 3)$ is found
- try executing longuest match with sepFSA from position 3
- the sep arc $(3, 5)$ is found.
- $(1, 3)$ and $(3, 5)$ are both valid arcs, so they are added to the set of valid arcs. We check if the position 5 is already in the queue, as it is not we add it
- we continue the execution of wordFSA from position 1, and find the token arc $(1, 4)$
- we execute longuest match with sepFSA from position 4 and we find the arc $(4, 5)$
- $(1, 4)$ and $(4, 5)$ are both valid arcs, and added to the set of valid arcs. We check if the position 5 is already in the queue, as it already is we don't add it
- all match from position 1 with wordFSA is finished, so we pop an element from the queue and start again the loop from that position.

For a formal definition of the algorithm, the implementation is in *implementations.lexicalutils.ArcParsing*

2) Tokenization and End Of Sentence detection: We split the original string everywhere where arcs start. All

(1)



(2)

|Mr |. |ϵ |Smith |' |s |ϵ |green |ϵ |card |. |

|It |' |s |ϵ |a |ϵ |' |nice |' |ϵ |cat |. |

Fig. 5. Step 1 and 2 of the tokenization algorithm. Spaces have been replaced with the ϵ character to explicit them.

the *VISIBLE_EOS* and *INVISIBLE_EOS* arcs that are not overlapped by any other arc type are valid non-ambiguous End Of Sentence tokens. The results is shown in (2) in figure 5. As an illustration of the EOS detection, the arc $(3, 5)$ and the $(23, 25)$ arc separating the two sentences are both *VISIBLE_EOS* arcs. In the case of the arc $(3, 5)$, the token arc $(1, 4)$ is overlapping with it, it is therefor an ambiguous EOS, and it will be left in the chart.

3) Finding invisible characters in tokens: All sub-strings delimited by *INVISIBLE_EOS* and *INVISIBLE_SEP* that do not overlap with a *TOKEN* arc are removed. All matches of invReg in the sub-strings of arcs are removed. The left over tokens are used to initialize the charts, as we can see in the final charts of figure 4. As an illustration of this step, notice the substring of the arc $(12, 13)$ does not show in the final charts. $(12, 13)$ was removed because it is as a *INVISIBLE_SEP* arc. The substring of the arc $(23, 25)$ is present in the first chart, but the space character at position 24 was removed. Only the '.' character at position 23 appears in the last cell. This is due to the scan inside arcs that removes matches of invReg.

4) Initializing the charts: We now have empty charts initialized with the tokens. We need to pass the sub-strings of the arcs found in step 1 to the transducer to find the grammatical tags. We cannot pass the sub-strings of the filtered arcs in step 2, as composed words like *green card* would be lost. The arcs of step 1 are therefor passed as is. If green card was written with a tab space instead of a simple space, this has to be handled by the transducer.

C. Implementation choices

Finding a multiplatform transducer for java was not an easy task. Foma is an excellent transducer that is multiplatform. It is written in C, but does not have bindings for java. This is why a

wrapper was written around the foma executable. The wrapper can be found at *implementations.fomawrapper.FomaWrapper*. The wrapper loads the foma binary only once. Once it is loaded, the only overhead is the pipe communication with the foma process, which is probably negligible. If this code is deployed on clusters, each node of the cluster will need a foma executable. Documentation on how to write foma files can be found on their website: https://fomafst.github.io/

In the implementation, an arc (i, j) designates the substring i to j included. Both systems work, it is just that having j included works more nicely in the code.

V. Syntactic analysis

We now have initialized charts that have to be filled. The best trade off we can have for parsing natural language syntax is using context free grammars. Regular grammars are not powerful enough, and context dependent grammar parsers exist, but run in exponential time. CFGs (Context Free Grammars) on the other hand already have an interesting expressiveness for natural language, and can be run in $O(n^3)$ time. Two algorithms exists for parsing context free ambiguous grammars: CYK and the Earley Parser.

An Earley Parser performs in $O(n^3)$ at worst while CYK always runs in $O(n^3)$ (where n is the number of tokens in the charts). CYK makes it simpler to debug what is going wrong in a grammar. Our choice of implementation was therefor CYK.

Up to today, no CFGs exist that can perfectly parse natural language, and this is the case for all languages. A computer science approach is simply to learn from big corpuses of data. Dependencies, as defined by http://universaldependencies.org/ are consistent and have been annotated on big corpuses in over 60 languages. Dependencies are simply defined by "*which word dominates which word*". An example is given in figure 6. An arrow $w_i \rightarrow w_j$ means w_i dominates w_j. We say w_i is the head of w_j. Dominates can be seen also as w_i is "*more important*" than w_j. In the corpuses, the grammatical tags are also given, which will be useful to extract grammars from the corpuses.

First we will present different techniques for extracting CFGs from the corpuses, then we will talk about our implementation, and finally we will present how CYK integrates with word embeddings.

A. Mapping dependency trees to CFGs

There is not a unique way of extracting CFGs from a corpus of dependency annotated sentences. The extraction technique will decide on how ambiguous the final chart after the CYK algorithm will be. There are three information we can play with: the word roots, the grammatical tags, and the arcs. To extract a CFG, we have the choice of following arcs

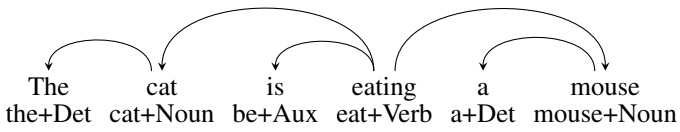


Fig. 6. Dependency example on a simple sentence

on one level, or multiple levels.

To explicit this, in figures 7 and 8 we extracted the rules of figure 6 on one and two levels, and using either grammatical tags or word roots and grammatical tags. There is no reason to use only word roots without grammatical tags, as the lexical layer has a transducer meant to find grammatical tags and words like *process+Noun* and *process+Verb* would become ambiguous. We will note $w \rightarrow w_1, w_2, ..., w_k$ as a shortcut for saying that w is the head of $w_1, w_2, .., w_k$, and we will note $w \Rightarrow w_1, w_2, .., w_k$ to designate a tree with root w and k children $w_1, w_2, .., w_k$. By convention, S is always the starting NonTerminal.

In the case of figure 7, the algorithm for constructing the syntactic tree is: for each $w \rightarrow w_1, w_2, ..., w_k$, we add the subtree $w \Rightarrow w_1, w_2, ..., w, ..., w_k$ in the syntactic tree, where the words $w_1, w_2, ..., w, ..., w_k$ are placed in the same order as the original text. With a one level extraction, only the head of the arc on the first level will have an impact. For example, only the head of the arc $cat \rightarrow The$ is important for rule 5, the The does not need to be known for rule 5 to be applicable.

In the case of figure 8, the algorithm for constructing the syntactic tree is: for each $w \rightarrow w_1, w_2, ..., w_k$, we follow the arcs of $w_1, ..., w_k$ to get:

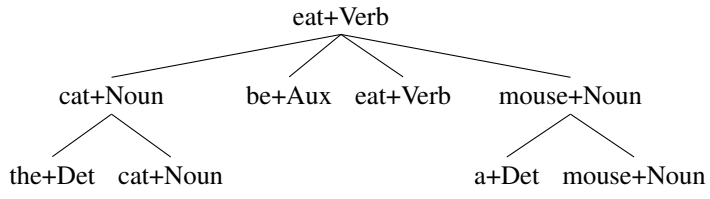
$$w \rightarrow (w_1 \rightarrow w_{11}, ..., w_{1l_1}), ..., (w_k \rightarrow w_{k1}, ..., w_{kl_k})$$

That we then flatten and mark the heads with $\bullet$ to get the subtree:

$$\begin{aligned} w_1 : w_2 : ... : \bullet w : .. : w_k \Rightarrow & w_{11} : ... : \bullet w_1 : ... : w_{1l_1}, \\ & ..., \\ & w_{k1} : ... : \bullet w_k : ... : w_{kl_1} \end{aligned}$$

As we can see, now rule 12 also depends on the The, while rule 5 didn't need it.

Choosing the correct way of extracting rules will be crucial for the performance of the algorithm. Although the rules of figure 7 seem cleaner then 8, executing the rules of 7 make a lot of ambiguity on sentences, and considerably slows down the algorithm. Using only grammatical tags with arcs that have been followed on one level can return millions of possible full parse trees. Rules that are extracted on too many levels could, on the other side, loose in generality. The algorithm would



(a)

$$\text{S} \Rightarrow \text{Verb} \quad (1)$$

$$\text{Verb} \Rightarrow \text{Noun Aux Verb Noun} \quad (2)$$

$$\text{Noun} \Rightarrow \text{Det Noun} \quad (3)$$

(b)

$$\text{S} \Rightarrow \text{eat+Verb} \quad (4)$$

$$\text{eat+Verb} \Rightarrow \text{cat+Noun be+Aux eat+Verb mouse+Noun} \quad (5)$$

$$\text{cat+Noun} \Rightarrow \text{the+Det cat+Noun} \quad (6)$$

$$\text{mouse+Noun} \Rightarrow \text{a+Det mouse+Noun} \quad (7)$$

Fig. 7. Syntax tree extracted from figure 6, where dependency arcs were followed on one level. (a) represents the grammatical rules extracted using only the grammatical tags, and (b) represents the grammatical rules extracted using word roots and grammatical tags.

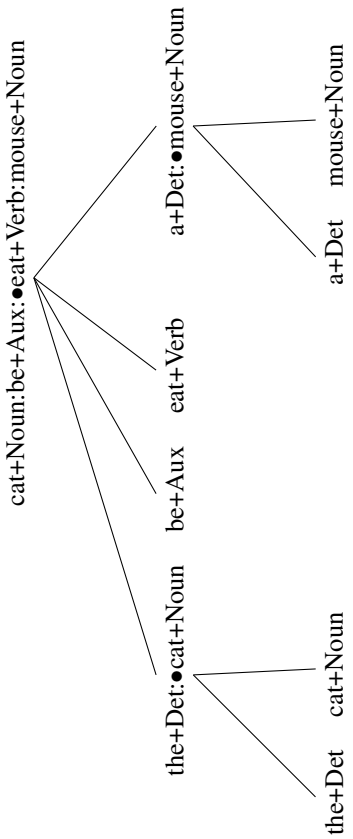
then not be able to adapt to sentences it has never seen before.

On possibility to be as precise and adaptable as possible is to have one grammar with a low precision, and one with a high precision. By precise we mean none ambiguous. A precise rule could be word roots and grammatical tags on two levels and the less precise one would be on one level for example. The high precision grammar would be called first, if it does not find any full parse tree for a sentence then the low precision grammar would be used on that sentence.

B. Implementation details of CYK

CYK can only be applied to rules in the Chomsky normal form. Any rule can easily be transformed to this form: for any rule $X \Rightarrow X_1, X_2, ..., X_k$ we create the rules $X \Rightarrow X_1 X_1'$, $X_1' \Rightarrow X_2, X_2'$, ..., $X_{k-2}' \Rightarrow X_{k-1}, X_k$. This is implemented in the *implementations.syntacticutils.BinaryGrammar* object, and the X_i' non terminals correspond to the *implementations.syntacticutils.AbstractNonTerminal* object. In the final chart, as each non terminal X_i' appear in a unique rule, we can reconstruct the original rules $X \Rightarrow X_1, X_2, ..., X_k$.

BinaryGrammar index's the rules from right to left, so for any pair X_1, X_2 of non terminal, we can find all the rules such that $X \Rightarrow X_1 X_2$ in $O(1)$. If n is the number of tokens in the chart and R is the total number of rules, the worst complexity is $O(n^3.R)$. This complexity is reached with grammars that are too ambiguous, like the grammar (a) of figure 7. In the best case, the grammar is well chosen and



(a)

$$\text{S} \Rightarrow \text{Noun:Aux:•Verb:Noun} \quad (8)$$

$$\text{Noun:Aux:•Verb:Noun} \Rightarrow \text{Det:•Noun Aux Verb Det:•Noun} \quad (9)$$

$$\text{Det:•Noun} \Rightarrow \text{Det Noun} \quad (10)$$

(b)

$$\text{S} \Rightarrow \text{cat+Noun:be+Aux:•eat+Verb:mouse+Noun} \quad (11)$$

$$\text{cat+Noun:be+Aux:•eat+Verb:mouse+Noun} \Rightarrow \text{the+Det:•cat+Noun be+Aux eat+Verb a+Det:•mouse+Noun} \quad (12)$$

$$\text{the+Det:•cat+Noun} \Rightarrow \text{the+Det cat+Noun} \quad (13)$$

$$\text{a+Det:•mouse+Noun} \Rightarrow \text{a+Det mouse+Noun} \quad (14)$$

Fig. 8. Syntax tree extracted from figure 6, where dependency arcs were followed on two levels. (a) represents the grammatical rules extracted using only the grammatical tags, and (b) represents the grammatical rules extracted using word roots and grammatical tags.

only a few rules need to be called per cell in the chart. Then the algorithm will run in $O(n^3)$.

C. Integrating CYK in the pipeline

CYK can work hand in hand with the semantic layer to resolve all ambiguities. In the semantic layer, word embeddings gives us the probability that a word w appears in a context c, which we will note $p(D = 1|w, c; \Theta)$. The $p(D = 1|w, c; \Theta)$ notation is described in the semantic analyzer section VI. In the way we extracted CFGs from the dependencies, any rule $X \Rightarrow X_1, X_2, .., X_k$ has a word w as head and forms some context c. For example the rule 12 in figure 8 has *eat+Verb* as head, and a context *the+Det:•cat+Noun be+Aux eat+Verb a+Det:•mouse+Noun*.

The probability of a parse tree that uses the rules $R1, R2, R2, ..., R_m$ is therefor just the multiplication of the probabilities:

$$p(D = 1|w_1, c_1; \Theta).p(D = 1|w_2, c_2; \Theta). ... .p(D = 1|w_m, c_m; \Theta)$$

Where each w_i, c_i correspond to the word and context of the rule R_i. With this, each cell of the CYK algorithm does not need to keep information about all the rules, but just the rule that has the maximum probability for each non terminal.

VI. Semantic analysis

As we saw in the previous section, word embeddings can be used to guide CYK and resolve ambiguities. The problem now is:

- what is the model
- what contexts do we use
- how to optimize the model

First we will present the model, then we will present some contexts, and finally we will talk about our implementation.

A. Word Embeddings model (word2vec)

In the original model, explained in [5] and [3], each word w is associated with a vector $v_w \in R^n$, and each context c is associated with a vector $v_c \in R^n$, where n is a hyper parameter. There are two models, but the skipgram model having better results is the one used in word2vec. Let D be the set of training tuples (w, c), D_w be the set of all words, D_c be the set of all contexts, w_i be the word at position i in the tokenized training text. Note that a word probably appears multiple time in the training text, so there most probably exists a j such that $w_i = w_j$, in this case we also have $v_{w_i} = v_{w_j}$ (as each word is associated with a unique vector). What the model maximizes is:

$$arg\,max_\Theta \prod_{(w,c)\in D} p(D = 1|c, w; \Theta) \tag{15}$$

Where $p(D = 1|c, w; \Theta)$ is the probably of the event: the tuple (w, c) comes from the train data. $p(D = 1|c, w; \Theta)$ is defined as:

$$p(D = 1|c, w; \Theta) = \frac{1}{1 + e^{-v_c \cdot v_w}} = \sigma(v_c \cdot v_w)$$

Equation 15 can be maximized by setting $v_c = v_w, \forall (c, w) \in D$, with vectors that are big enough, meaning $||v|| > A$ with A a large number. The solution in word2vec is to add negative sampling D'. D' will contain tuples (w, c) that should not appear together and that will have to be minimized. Equation 15 to maximize becomes:

$$\left(\prod_{(w,c)\in D} p(D = 1|c, w; \Theta) \right) \left(\prod_{(w,c)\in D'} p(D = 0|c, w; \Theta) \right) \tag{16}$$

With $p(D = 0|c, w; \Theta) = \sigma(-v_c \cdot v_w)$.

B. Defining the contexts

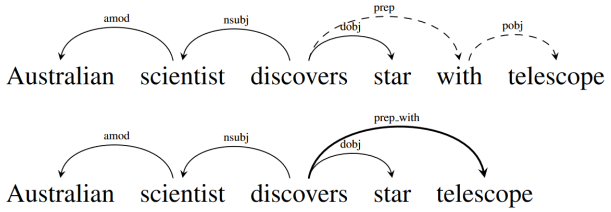
Now that we have the model, that last thing to choose is contexts. There are different ways of choosing contexts. The vectors for the words can be used for the contexts or not. Formally that would mean $D_c = D_w$ or $D_c \cap D_w = \emptyset$. Also dependency trees can be used, or not. We will present three different ways of defining contexts.

1) Linear contexts (word2vec): In word2vec, the contexts of a word w_i at position i in the original training text are all the words at distance k of w_i: $w_{i-k}, w_{i-(k-1)}, ..., w_{i-1}, w_{i+1}, ...w_{i+k}$, where k is a hyper parameter. In other words, w_i will have the tuples $(w_i, w_{i-k}), (w_i, w_{i-(k-1)})...(w_i, w_{i+k})$ in D. In this case, $D_c = D_w$ and no dependency trees are needed.

2) Dependency based linear contexts (defined in [4]): For a word w_i, all the words at distance 1 of w_i in the dependency tree are the contexts. The example given is shown in figure 9. They filtered out words like prepositions, and followed the dependency arcs to extract the neighboring words. In this case $D_c = D_w$ and dependency trees are needed.

Using dependencies has advantages over linear contexts. As was shown in figure 9, words like prepositions can be filtered out, and the arcs can be collapsed to obtain words that might of been missed by CBOW-k. One problem of linear contexts and dependency based linear contexts is that they loose the order of the words. As we will see, arbitrary dependency based contexts can keep all available information.

3) Arbitrary dependency based contexts: Arbitrary dependency based contexts is any context defined from the dependency trees. It can respect $D_c = D_w$ or not. As an example we will create contexts that keep information about words and order, and has $D_c \cap D_w = \emptyset$. Given a input text with dependency trees properly parsed, let $child(w_i, i)$ be the ordered list $[w_{i_1}, w_{i_2}, ..., w_{i_l}]$ of all the children of word w_i in the dependency tree at position i in the input text. For example in figure 9, $child(discovers, 3)$ would be $scientist, star, telescope$.



WORD	CONTEXTS
australian	scientist/amod^{-1}
scientist	australian/amod, discovers/nsubj^{-1}
discovers	scientist/nsubj, star/dobj, telescope/prep_with
star	discovers/dobj^{-1}
telescope	discovers/prep_with^{-1}

Fig. 9. Contexts extracted from dependencies, example from [4]

Contexts name	Corresponding words
c_0	australian+N, $\bullet$
c_1	scientist+N, star+N, $\bullet$, telescope+N

Fig. 10. Contexts extracted from figure 9 taking in account lemma's, dependency tree and the order of the words

The maximum precision for a context we could image would be: words are represented by lemmas, and for every unique list $child(w_i, i)$ in the training set we create a context c. For example, figure 10 represents such contexts that would be extracted from the sentence of figure 9. The dot represents the position of the head w_i relatively to the children $child(w_i, i)$. This time the order of the words is taken in account, for example the context $scientist{+}N, star{+}N, \bullet, telescope{+}N$ is different from $scientist + N, \bullet, star + N, telescope + N$.

C. Implementation

Word embeddings becomes quickly a very expensive calculation, especially with arbitrary contexts with $D_c \cap D_w = \emptyset$. For this reason, the frameworks used behind become crucial. We decided to use Nd4j and spark. Nd4j is used for linear algebra, it has two backends: a library written in *C* and cuda. In the idea of staying multiplatform, the *C* backend was used. Spark is a very flexible and powerful framework for writing algorithms on clusters. The implementation can therefor be easily adapted to use cuda backend, and to be deployed on clusters. Two challenges remain:

- getting the negative samples
- finding and algorithm for finding the v_c and v_w vectors

First we will present how we implemented negative sampling, then we will present the algorithm for finding the v_c and v_w vectors, and finally we will give some notes about the implemented classes.

1) Getting the negative samples: The *implementations.semanticutils.TrainData* class was used to find negative samples. It follows the recommendations of [5]. For every word w in D, M contexts c_i are randomly chosen as negative samples. The c_i contexts are chosen according to the distribution:

$$p(c) = \frac{p_{contexts}(c)^{3/4}}{Z}$$

Where $p_{contexts}(c) = \frac{\#c}{\sum_{c_i \in D} \#c_i}$, and $\#c$ is the number of times context c appears in the training set D. The $3/4$ power is a way to reduce the impact of the most popular contexts, at the benefit of mediumly frequent contexts. For efficiency, the negative samples are calculated by:

- getting the rdd rdd_c of all unique contexts
- using a flatMap to repeat $|D_w| * M * p(c_i)$ times c_i in rdd_C for each $c_i \in D_c$
- distributing the new rdd randomly on each word

2) Finding the v_c and v_w vectors: Similarly to the matrix factorization algorithm, alternating gradient descent was used, although this time it is 'gradient ascent' as we try to maximize and not minimize. At each iteration, we update each vector v_w with $v_w \leftarrow v_w + \gamma \nabla_{v_w}$, and then we update each vector v_c with $v_c \leftarrow v_c + \gamma \nabla_{v_c}$. As recommended in [5], we take the log of the equation 16. In addition we also added regularization factors, as it is often done in machine learning. The equation to maximize becomes:

$$\left(\sum_{(w_i,c_j) \in D} log(p(D = 1|c_j, w_i; \Theta)) \right)$$
$$+ \left(\sum_{(w_i,c_j) \in D'} log(p(D = 0|c_j, w_i; \Theta)) \right)$$
$$-\lambda_c \sum_{c_j \in D_c} \frac{1}{2} ||v_{c_j}||_2^2$$
$$-\lambda_w \sum_{w_i \in D_w} \frac{1}{2} ||v_{w_i}||_2^2$$

At each iteration, we update v_c and v_w with their gradient. With positive samples, the gradients are:

$$\nabla_{v_c}(D) \left[\sum_{w_i,(w_i,c) \in D} log(\sigma(v_c \cdot v_{w_i})) \right]$$
$$= \sum_{w_i,(w_i,c) \in D} \frac{exp(-v_{w_i} \cdot v_c)}{1 + exp(-v_{w_i} \cdot v_c)} v_{w_i}$$

$$\nabla_{v_w}(D) \left[\sum_{c_j,(w,c_j) \in D} log(\sigma(v_{c_j} \cdot v_w)) \right]$$
$$= \sum_{c_j,(w,c_j) \in D} \frac{exp(-v_w \cdot v_{c_j})}{1 + exp(-v_w \cdot v_{c_j})} v_{c_j}$$

And for negative samples, the gradients are:

$$\nabla_{v_c}(D')\left[\sum_{w_i,(w_i,c)\in D'} log(\sigma(-v_c \cdot v_{w_i}))\right] = \sum_{w_i,(w_i,c)\in D'} -\frac{exp(v_{w_i} \cdot v_c)}{1+exp(v_{w_i} \cdot v_c)} v_{w_i}$$

$$\nabla_{v_w}(D')\left[\sum_{c_j,(w,c_j)\in D'} log(\sigma(-v_{c_j} \cdot v_w))\right] = \sum_{c_j,(w,c_j)\in D'} -\frac{exp(v_w \cdot v_{c_j})}{1+exp(v_w \cdot v_{c_j})} v_{c_j}$$

The final gradients $\gamma\nabla_{v_c}, \gamma\nabla_{v_w}$ used in the algorithm are normalized with the number of samples:

$$\nabla_{v_c} = \frac{1}{|\{w_i,(w_i,c)\in D\}|}\nabla_{v_c}(D) + \frac{1}{|\{w_i,(w_i,c)\in D'\}|}\nabla_{v_c}(D')$$

$$\nabla_{v_w} = \frac{1}{|\{c_j,(w,c_j)\in D\}|}\nabla_{v_w}(D) + \frac{1}{|\{c_j,(w,c_j)\in D'\}|}\nabla_{v_w}(D')$$

If $D_c = D_w$, then the function being maximized is similar to logistic regression and is concave. If $D_c \cap D_w = \emptyset$, the function is neither concave or convex.

3) Notes on the implementation: The algorithm above works for both dependency based linear contexts and arbitrary dependency based contexts. Dependency based linear contexts is only a special case of arbitrary dependency based contexts. When the algorithm is used for dependency based linear contexts, the algorithm will simply update the vectors v_w two times per iteration instead of one.

Two versions of the algorithm were implemented in the *implementations.semanticutils* package:

- *WordEmbeddingsGradientDescentImpl1*
- *WordEmbeddingsGradientDescentImpl2*

Implementing word embeddings for clusters efficiently is tricky, as at every iteration one node of the cluster will need the data from all the other nodes. This is due to the negative sampling, which spreads randomly contexts across the nodes. This is why we tested two different implementations. When executed on a single machine, the first implementation was nearly 3 times as fast as the second one. Impl1 concentrates all the communications between the nodes with a *collectAsMap* operation. Once this operation is done, the rest of the iteration does not need any communication between the nodes. On the other hand, impl2 does everything with joins, which could possibly be more efficient than impl1 if careful repartition of the data is used.

In the best of cases, an external library could of been used to avoid writing our own, but none seem to exist that can handle arbitrary contexts. In this project we are interested in trying different techniques. The deeplearning4j word2vec implementation can be used for linear contexts, and our custom implementation can be used for all the other contexts.

VII. Putting everything together

As a recap, the final goal of the implementation is to get word embeddings. The lexical, syntactic and semantic layers all need bootstrapping, and we have the conllu files for this. In the implementation, the *implementations.filereaders.ConlluReader* object reads conllu files.

A. Bootstrapping the system

The lexical layer needs a transducer, wordFSA, sepFSA, invReg and sepReg configured. Word to grammatical tag mappings, and the lexicon for wordFSA and sepFSA can be extracted directly from the conllu file. *implementations.ConlluUtils* is a main class that can take as argument -extractLexec, it is meant for bootstrapping the lexical layer. invReg and sepReg have to be configured by hand. The way to test them would be to set them to reasonable values, and then apply the lexical analyzer on the text of the conllu file. The improperly tokenized sentences could then be automatically detected, and the invReg and sepReg updated as necessary.

The syntactic layer needs a grammar as configuration. The *implementations.ConlluUtils* main class, can take the argument -extractGrammar for doing the extraction. For the syntactic layer, there is no sens in applying the analyzer on the conllu file for testing, as the analyzer will just return all possible parse trees. For this, the semantic analyzer also needs to be bootstrapped, so that the syntactic layer can remove all ambiguities and return the most probable parse tree (as explained in section V-C). The syntactic and semantic layer therefor have to be tested together against the original conllu file.

B. Fully running system

The lexical and syntactic layers can operate on continuous streams, while the semantic layer has to process the input in batches. This is represented in figure 11. The DependencyTree object is simply a chart without ambiguities that has been found by the syntactic and semantic analyzers working together. Batch processing can be done in many ways, the biggest question being how to sample the (w, c) pairs for each batch.

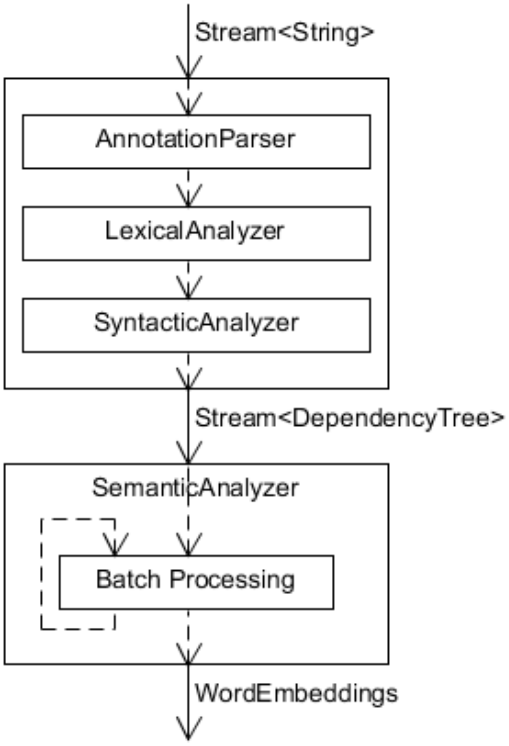


Fig. 11. Full running system

During the batches, the system can either converge or diverge. If the bootstrap to the system was not similar enough to the new texts the system is applied to, than the input dependency trees to the SemanticAnalyzer would be wrong, and the vectors learned would be wrong too. We can check if the system is diverging by applying between each batch the system on the conllu file used for bootstrapping. The number of properly parsed trees over the total number of sentences in this conllu file gives us the precision of the system. If the precision goes down, we are diverging, if it goes up we are converging.

VIII. Conclusions

In this work, we carryed on the work of [1], adapted the code for deploying the project on clusters, and researched into what can be done at the semantic level. A lot of time has been put into software engineering and documentation. This was an important aspect of the project as it should be easily expendable and understandable. The full stack from lexical to semantic is now implemented, with 47 unit and integration tests.

IX. Future work

Fine tuning each algorithm is still needed. Such a project needs to be applied to big corpuses. The code needs to be deployed on clusters along side word2vec and fastText to compare the performances. The fine tuning, testing and experimentation part is a time consuming operation for which, unfortunately, time ran out during this iteration of the project. The hope is: this report and the documentation of the code are precise enough for the project to live on.

X. Acknowledgments

To Dr Martin Rajman for suggesting the project, his amazing support and ideas, enthusiasm and enriching discussions. I would also like to acknowledge Meryem M'hamdi for her wonderful work on the project before me and her support.

References

[1] Meryem M'hamdi, Dr Martin Rajman. *Analysis of the Impact of Linguistic Processing on Large Scale Semantic Similarity* , 2017

[2] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg Corrado, Jeffrey Dean. *Distributed Representations of Words and Phrases and their Compositionality*, arXiv:1310.4546, 2013

[3] Tomas Mikolov, Kai Chen, Greg Corrado, Jeffrey Dean. *Efficient Estimation of Word Representations in Vector Space* arXiv preprint arXiv:1301.3781, 2013

[4] Omer Levy, Yoav Goldberg. *Dependency-Based Word Embeddings*, in ACL (2), 2014 - aclweb.org

[5] Yoav Goldberg, Omer Levy. *word2vec Explained: Deriving Mikolov et al.s Negative-Sampling Word-Embedding Method* arXiv:1402.3722, 2014