Algorithm for Extraction of Subtrees of a Sentence Dependency Parse Tree

Juan-Pablo Posadas-Durán[1], Grigori Sidorov[2], Helena Gómez-Adorno[2], Ildar Batyrshin[2], Elibeth Mirasol-Mélendez[3], Gabriela Posadas-Durán[1], Liliana Chanona-Hernández[1]

[1]Instituto Politécnico Nacional (IPN), Escuela Superior de Ingeniería Mecánica y Eléctrica Unidad Zacatenco (ESIME-Zacatenco),
Av. Luis Enrique Erro S/N 07738, Mexico City, Mexico.

[2]Instituto Politécnico Nacional, Centro de Investigación en Computación,
Av. Juan de Dios Bátiz 07738, Mexico City, Mexico.

[3]Instituto Politécnico Nacional, Escuela Nacional de Medicina y Homeopatía ,
Guillermo Massieu Helguera 239, 07320, Mexico City, Mexico.

E-mail: jdposadas@esimez.mx, sidorov@cic.ipn.mx, batyr1@cic.ipn.mx, hgomeza1400@alumno.ipn.mx, emirasolm0800@alumno.ipn.mx, gposadasd@ipn.mx, lchanonah@ipn.mx

Abstract: In this paper, we introduce an algorithm for obtaining the subtrees (continuous and non-continuous syntactic n-grams) from a dependency parse tree of a sentence. Our algorithm traverses the dependency tree of the sentences within a text document and extracts all its subtrees (syntactic n-grams). Syntactic n-grams are being successfully used in the literature (by ourselves and other authors) as features to characterize text documents using machine learning approach in the field of Natural Language Processing.

Keywords: syntactic n-grams; subtrees extraction; tree traversal; linguistic features

1 Introduction

Stylometry is an active research field that studies how to model the style of an author from a linguistic point of view by proposing reliable features based on the use of the language. These features, known as style markers, characterize the writing style of an author and are used to solve various tasks in the field of Natural Language Processing like authorship attribution [1], author profiling [2, 3, 4], author verification [5], author clustering [6], and plagiarism detection [7, 8, 9], among others.

The problem of authorship characterization can be tackled using different approaches. The majority of the proposed methods use n-gram based, morphological and lexical characteristics that only exploit the superficial information of a text. Let us remind that n-grmas are sequences of elements as they appear in the texts, they can be formed by words/lemmas/stems, characters, or POS tags.

Traditional approaches ignore syntactic features despite the fact that the syntactic information is topic independent and therefore is robust to characterize style of an author. Note that the surface representation is prone to noise, for example, insertion of a subordinate clause or an adjective changes the n-grams, while the syntactically based features handle this problem correctly.

In this paper, we present an algorithm, which uses syntactic information contained in dependency trees to extract complete syntactic n-grams. The main idea is that we form n-grams by following the paths in the dependency tree, instead of taking them in the order of their appearance at the surface level. Dependency trees represent the syntactic relations between words, which form the sentence. The proposed algorithm builds complete syntactic n-grams in a general manner. It considers both types of syntactic n-grams: continuous [10] and non-continuous [11], which is called complete syntactic n-grams. The difference between them is that non-continuous n-grams have bifurcations, i.e., the corresponding subtrees have several branches, while continuous n-grams represent exactly one branch. The reason for this distinction is the supposition that there is different linguistic reality in each case. In addition, syntactic n-grams can be formed by various types of elements, like words/lemmas/stems, POS tags, dependency tags or a combination of them. Note that dependency tags are not used in traditional n-grams.

The algorithm extracts the syntactic n-grams from a dependency tree by performing a two-stage procedure. The first stage the algorithm conducts a breadth-first search of the tree and finds all the subtrees of height equal to 1. In the second stage, the algorithm traverses the tree in postorder replacing the node occurrence in a subtree with the subtrees from higher levels where the node is the root. The extracted subtrees correspond to syntactic n-grams of the tree.

We implemented our algorithm in Python and made it freely available at our website[1]. Note that though we presented the idea of syntactic n-grams in our previous works, until now we did not present the description of the algorithm used for their extraction. It is worth mentioning that the implementation of the algorithm was freely available for three years and it was used by other researches who used syntactic n-grams.

The rest of the paper is organized as follows. The concept of complete syntactic n-grams is discussed in Section 2. Section 3 describes the use of the syntactic n-grams in various problems related to Natural Language Processing. The algorithm for extraction of syntactic n-grams (all subtrees of a dependency parse tree) is presented in Section 4. Finally, in the last section of the paper, we draw conclusions and discuss directions of future work.

[1] `http://www.cic.ipn.mx/~sidorov/MultiSNgrams_3.py`

2 The Idea of Syntactic N-grams

As we mentioned in the previous section, the concept of syntactic n-grams (sn-grams) was first introduced in [11, 10] as an alternative idea to the well-known representation based on character n-grams or word n-grams. Standard character or word n-grams are sequences of elements extracted from a given text by using an imaginary window of size n, which slides over the text with certain offset, typically equal to 1. On the other hand, syntactic n-grams are the paths of size n generated by following the branches of a dependency tree, i.e., they do not depend on the surface order of elements. Syntactic n-grams are extracted by traversing the sentence dependency trees of a text and correspond to all subtrees of the dependency tree. Syntactic n-grams capture the syntactic relations between words in a sentence.

Two important differences can be observed between syntactic and traditional n-grams: (1) syntactic n-grams are able to capture information (internal information), which traditional n-grams cannot access (they use only the surface information), (2) each syntactic n-gram always has a meaning from a linguistic point of view (i.e., there is always underlying grammar that was used for parsing), unlike the traditional n-grams, which do not have it in many cases (i.e., there are many n-grams that just represent noise because there position of one near the other is a pure coincidence).

The syntax of a text is the way, in which words relate to each other to express some idea as well as the function that they have within a text. The relations that exist between the words of a sentence can be represented by the two grammatical formalisms: dependency or constituency grammars [12]. In modern research, the dominant approach is dependency analysis, though these formalisms are equivalent, i.e., their representations can be transformed one into another.

The dependency grammar shows the relations between pairs of words, where one of them is the head word and the other word is the dependent. It is represented as a tree structure that starts with a root node (generally the verb of the sentence), which is the head word of more general order. Then, the arc are used between the head and the dependent words according to dependency relations between them. The dependent words, in turn, are considered as head words and their dependent words are added, thus generating a new level in the tree. The structure described above is known as dependency tree [13].

In order to illustrate the concept of syntactic n-grams and the differences between them and the standard n-grams lets consider the sentence: "*Victor sat at the counter on a plush red stool*". We get the following standard word 3-grams using the sliding imaginary window method: *"Victor sat at", "sat at the", "at the counter", "the counter on", "counter on a", "on a plush", "a plush red", "plush red stool"*.

To extract the syntactic n-grams of any sentence it is necessary first to process the sentence by a syntactic parser. We processed the sentence using the Stanford CoreNLP toolkit [14] and as the result we obtain the lemmas, POS tags, dependency relations tags and the relations between the elements of the sentence.

A dependency tree structure $T = (V, E)$ with root v_0 is obtained from the processed

sentence, where the set of nodes $V = \{v_0, v_1, \ldots, v_i\}$ correspond to the words of the sentence and the set of branches $E = \{e_0, e_1, \ldots, e_j\}$ correspond to dependency relations between words.

Table 1 shows the standard syntactic information that can be gathered from the example sentence. Note that the column *Dependent* denotes the set of nodes of dependent words, the column *Head* corresponds to the nodes of the head word and the row *Leaves* shows the set of nodes of words without dependents (leaves). The root node (v_0) is denoted by the tag *root* in the *SR* column. Figure 1 shows graphical representation of the dependency tree T for the example sentence. It depicts the relations between words using a black arrow where the tail of the arrow denotes the head word, and the head of the arrow denotes the dependent word.

Table 1
Syntactic information obtained from the sentence "*Victor sat at the counter on a plush red stool*"

Id	Word	Lemma	POS	Head	SR	Dependent
1	Victor	Victor	NNP	2	nsubj	
2	sat	sit	VBD	0	root	[1,3,6]
3	at	at	IN	2	prep	[5]
4	the	the	DT	5	det	
5	counter	counter	NN	3	pobj	[4]
6	on	on	IN	2	prep	[10]
7	a	a	DT	10	det	
8	plush	plush	JJ	10	amod	
9	red	red	JJ	10	amod	
10	stool	stool	NN	6	pobj	[7,8,9]
Leaves (nodes without children): [1,4,7,8,9]						

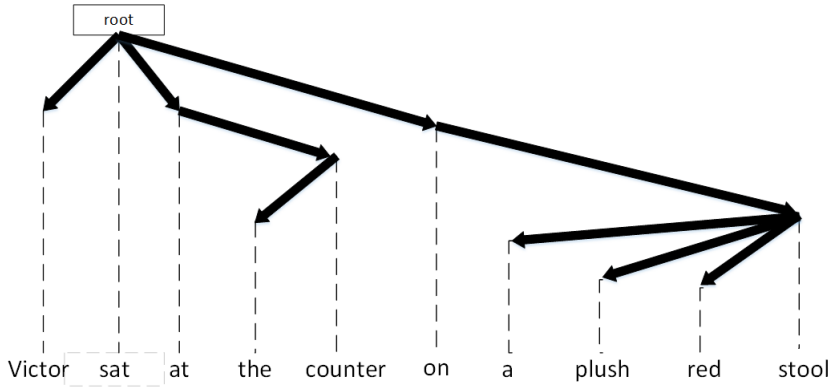

Figure 1
Dependency tree of the sentence "*Victor sat at the counter on a plush red stool*"

The set of labels of dependency relations used by the CoreNLP toolkit is described in [15] and the set of POS tags is described in [16].

The syntactic n-grams can be homogeneous or heterogeneous. We call them homogeneous when they are constructed of the same type of elements, for example, only of words, or only of POS tags. They are heterogeneous, when various types of elements are combined in the same syntactic n-gram.

For the example sentence, we extract syntactic n-grams of size 3 considering the homogeneous case of words. The syntactic n-grams are extracted by traversing the dependency tree and identifying all the subtrees with exactly three nodes. The syntactic n-grams are codified using the metalanguage proposed in [11]. The metalanguage is simple: the head element is on the left of a square parenthesis and inside there are the dependent elements; the elements at the same level are separated by a coma. The syntactic n-grams extracted are: *[Victor,at], sat[Victor,on], sat[at,on], sat[on[stool]], sat[at[counter]], on[stool[plush]], on[stool[a]], on[stool[red]], stool[a,plush], stool[a,red], stool[plush, red], sat[at[counter]], at[counter[the]].*

If we consider heterogeneous syntactic n-grams, then we combining elements of different nature. This kind of sn-grams is obtained by setting one type of information for the head element and use a different type for the rest of elements. The syntactic analysis offers the possibility to work with words, lemmas, POS tags and dependency relation tags (DR tags). So, there are 12 possible pairs for heterogeneous syntactic n-grams that can be extracted [11, 17]: (words, lemmas), (words, POS), (words, DR), (lemmas, words), (lemmas, POS), (lemmas, DR), (POS, words), (POS, lemmas), (POS, DR), (DR, words), (DR, lemmas), and (DR, words).

Table 2 shows the heterogeneous syntactic n-grams (words, POS) of size 3 extracted from the previous example sentence along with the homogeneous syntactic n-grams of words. The heterogeneous syntactic n-grams are able to identify new patterns due to the fact that they combine information from different contexts, for example the sn-grams *sat[NNP, IN], sat[IN [NN]], on[NN[JJ]]* are patterns that occur more frequently compared to homogeneous sn-grams.

Table 2
Comparing sn-grams of words vs. sn-grams of (word, POS)

Words	**(words, POS)**
sat[Victor,at]	sat[NNP,IN]
sat[Victor,on]	sat[NNP,IN]
sat[at,on]	sat[IN,IN]
sat[on[stool]]	sat[IN[NN]]
sat[at[counter]]	sat[IN[NN]]
on[stool[plush]]	on[NN[JJ]]
on[stool[a]]	on[NN[DT]]
on[stool[red]]	on[NN[JJ]]
stool[a,plush]	stool[DT,JJ]
stool[a,red]	stool[DT,JJ]
stool[plush,red]	stool[JJ,JJ]
sat[at[counter]]	sat[IN[NN]]
at[counter[the]]	at[NN[DT]]

The heterogeneous sn-grams are not restricted neither to the way of combining the syntactic information (one type for the head and another type for the dependents) nor to the type of information mentioned before (words, lemmas, POS tags and DR tags). In general, it is possible to use different kind of information at each level of the subtree, for example, we can use words in the head, POS tags for the elements at level 1, lemmas for the ones at level 2 and so on.

3 Syntactic N-grams as Features for Natural Language Processing Tasks

The written language provides various levels of language description: semantic, syntactic, morphological, lexical, etc. In different works [18, 19], textual feature representation are classified into one of the following categories: character level, lexical level, syntactic level, semantic level, and format level. The syntactic n-grams fall into the syntactic level category, which is one of the least used levels of text representation for automatic analysis.

Although written language offers a wide range of possibilities for characterization, most of the works in the literature focus on morphological and lexical information because of their direct availability. Some examples of text features that have been proposed in the literature are: size of the sentences, size of tokens, token frequency, frequency characters, richness of used vocabulary, character n-grams, word n-grams, among others [20].

As we already mentioned, the main idea of syntactic n-grams is to follow the path in a syntactic tree for obtaining the sequence of elements, instead of using the surface order. It allows to bridge syntactic knowledge with the machine learning methods in modern Natural Language Processing.

In the first paper on syntactic n-grams, continuous syntactic n-grams was introduced in [21, 10] for text classification tasks. The authors evaluated the sn-grams in the task of authorship attribution and compared their performance against traditional n-grams of characters, words, and POS tags. The corpus used in their experiments includes English texts from the Project Gutenberg. For the classification purposes, they used three algorithms: Support Vector Machines (SVM), Naive Bayes (NB) and Decision Trees (J48). The sn-grams features gave better results with SVM classifier over the other traditional features.

There were two research works on grammatical error correction using syntactic n-grams. The first work, developed by Sidorov [22], presents a methodology that applies a set of simple rules for correction of grammatical errors using sn-grams. The methodology achieved acceptable results on the CONLL Shared Task 2013. The second work, by Hernandez et al. [23], used a language model based on syntactic 3-grams and 2-grams extracted from dependency trees generated from 90% of the English Wikipedia. Their system ranked 11th on the CONLL Shared Task 2014.

Author profiling (AP) is another task related to the authorship attribution. The aim of AP is to determine author's demographics based on a sample of his writing. In [5]

the authors present an approach to tackle the AP task at PAN 2015 competition [24]. The approach relies on syntactic based n-grams of various types in order to predict the age, gender and personality traits of the author of a given tweet. The obtained results indicate that the use of syntactic n-grams along with other specific tweet features (such as number of retweets, frequency of hashtags, frequency of emoticons, and usage of referencing URLs) are suitable for predicting personal traits. However, their usage is not that successful when predicting the age and gender.

In [25], the authors explore the use of syntactic n-grams for the entrance exams question answering task at CLEF. They used syntactic n-grams as features extracted from Syntactic Integrated graphs and Levenshtein distance as the similarity measure between n-grams, measured either in characters or in elements of n-grams. Their experiments show that the soft cosine measure along with syntactic n-grams provides better performance in this case study.

In the paper by Calvo et al. [26] the authors compared the constituency based syntactic n-grams against the dependency based syntactic n-grams for paraphrase recognition. They presented a methodology that combines sn-grams with different NLP techniques (synonyms, stemming, negation handling and stopwords removal). Both types of sn-grams were evaluated independently and compared against state-of-the-art-approaches. Syntactic n-grams outperformed several works in the literature and achieved an overall better performance compared with traditional n-grams in paraphrase recognition. In most cases, syntactic constituent n-grams yielded better scores than syntactic dependency n-grams.

The research work of Laippala et al. [27], studies the usefulness of syntactic n-grams for corpus description in Finnish, including literature texts, Internet forum discussion for social media and newspapers' websites. Their results suggests that in comparison with traditional feature representation, syntactic n-grams offer both complementary information generalizing beyond individual words to concepts and information depending on syntax not reached by lexemes.

A recent work on statistical machine translation (SMT) [28] proposes a relational language model for dependency structures that is suited for languages with a relatively free word order. The authors empirically demonstrate the effectiveness of the approach in terms of perplexity and as a feature function in string-to-tree SMT from English to German and Russian. In order to tune the log-linear parameters of the SMT they use a syntactic evaluation metric based on syntactic n-grams, which increases the translation quality when coupled with a syntactic language model.

In the book "Prominent Feature Extraction for Sentiment Analysis" [29], the authors explore semantic, syntactic and common-sense knowledge features to improve the performance of sentiment analysis systems. This work applies sn-grams for sentiment analysis for the first time. The authors show that syntactic n-grams are more informative and less arbitrary as compared to traditional n-grams. In their experiments, syntactic n-gram feature set produced an F-measure of 86.4% with BMNB classifier for movie review dataset. This feature set performed well as compared to other simple feature extraction techniques like unigrams, bigrams, bi-tagged, and dependency features.

As it can be seen, syntactic n-grams are being used successfully in a wide range of NLP tasks. In order to increment the research on the use of syntactic n-grams as feature representation we introduce the detailed description of the algorithm for extracting complete syntactic n-grams from a sentence parse tree.

4 Algorithm for Extraction of Syntactic N-grams

We mentioned the algorithm for the extraction of sn-grams in our previous works, for example, in [17], but it is the first time that we give it's detailed description. The algorithm handles homogeneous and heterogeneous variants of continuous and non-continuous syntactic n-grams. The algorithm is implemented in Python, we provide the full implementation of this algorithm at our website (see Section 1).

We established in Section 2 that syntactic n-grams are extracted from a dependency tree structure and they correspond to the subtrees of a tree, i.e., given a dependency tree $T = (V, E)$ with the root v_0, all syntactic n-grams are the set of subtrees $ST = \{st_0, st_1, \ldots, st_k\}$ of the tree with the restriction that each st_i must be of size n. Basically, the algorithm traverses the dependency tree in order to find all the subtrees.

The algorithm consists of two stages: first stage performs a breadth-first search over the tree and extracts all the subtrees of height equal to 1, second stage traverses the tree in postorder replacing the node occurrence in a subtree of lower level with the subtrees from higher levels where the node is the root so that subtrees with height greater than 1 are extracted. The algorithm discriminates those subtrees that do not satisfy the restriction of size n.

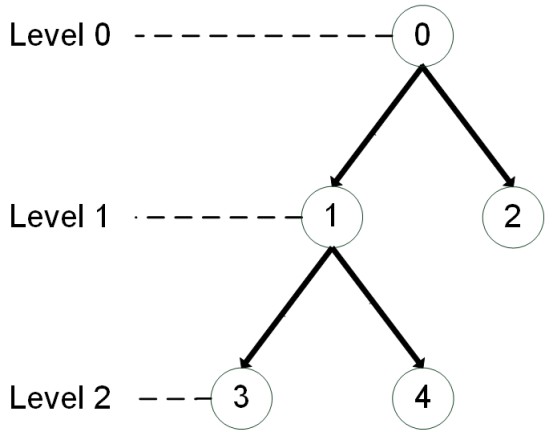

Figure 2
Sample tree

To illustrate how the algorithm extracts the syntactic n-grams, let consider the tree T shown in Figure 2. We express the subtrees according to the proposed metalanguage in [11]. If we perform the first stage of the algorithm to the tree T, we obtain at level

0 the subtrees *0[1], 0[2], 0[1,2]* and *1[3], 1[4], 1[3,4]* at level 1. Note that all the subtrees extracted in the first stage of the algorithm have a height equal to 1.

Then we continue performing the second stage of the algorithm and traverse the tree T in preorder replacing the nodes of the subtrees in level 0 with the subtrees in level 1 that has the node as the root element. Observe that only the node 1 satisfies the condition and can be modified in the subtrees of the lower level. After accomplished the second stage, we obtain the following subtrees: *0[1[3]], 0[1[4]], 0[1[3,4]], 0[1[3],2], 0[1[4],2], 0[1[3,4],2]*.

All the possible subtrees of size greater than 1 are generated by the algorithm. These subtrees may become into syntactic n-grams after adding linguistic information. The case of syntactic n-grams of size equal to 1 represent the words of the sentence and are not meaningful for the scope algorithm.

The algorithm is organized into seven functions. The primary function named *EXTRACT_SNGRAMS* is described in the Algorithm 8, and it coordinates the call of the other functions so the two stages of the described algorithm are performed. It is the interface that user utilize to extract the sn-grams.

The function *EXTRACT_SNGRAMS* calls to *GET_SUBTREES* function described in Algorithm 1 that performs the first stage of the proposed algorithm. The *GET_ SUBTREES* function in turn calls the functions *NEXTC* (Algorithm 2) and *NUM_COMBI NATION* (Algorithm 3), both auxiliary functions.

Then the second stage of the algorithm is realized by the function *COMPOUND_SN GRAMS* (Algorithm 5). Finally functions *LEN_SNGRAM* (Algorithm 4) and *PREPARE_SN GRAMS* (Algorithm 6 and 7) rewrite the subtrees extracted into one of the types of sn-grams.

The input of the algorithm is a plain text that contains syntactic information of a sentence and returns the syntactic n-grams together with their frequency of occurrence. The algorithm requires a syntactic parser, which is an external tool often used in many Natural Language Processing problems.

Various syntactic parsers are available, like Standford CoreNLP, Freeling, Connexor, to name some, and although all of them retrieve the same syntactic information, there is no standard for the output. It is merele a technical detail, but we would like to mention that the our code handles the outputs of the Stanford parser. For Freeling, we generated another Python script, which converts its output to the output of the Stanford parser.

The output generated by the Stanford CoreNLP has the following format: in one section it contains the words, lemmas and POS tags of a sentence and in another section it contains the dependency relation tags and dependency tree structure codified as a list of pairs: head node and dependent node.

Algorithm 1 presents the function *GET_SUBTREES*, which receives an adjacency table with the syntactic information of a sentence (as shown in Table 1) and returns a list of codified subtrees bounded in size by two parameters (minimum and maximum size). The function also requires the indexes of the nodes that can be roots of the

Algorithm 1 Function *GET_SUBTREES*

Parameters: *sentence* (syntactic information matrix), *subroots* (possible roots of subtrees), *min_size* (minimum size), *max_size* (maximum size), *max_num_children* (maximum number of children to be consider for a node).
Output: the list of the subtrees' indexes for a sentence dependency tree.

```
 1: function GET_SUBTREES(sentence, min_size, max_size)
 2:     Vars: unigrams[ ], combinations[ ], counter = 0, aux[ ]
 3:     for all node in subroots do
 4:         if max_num_children! = 0 then
 5:             aux ← [ ]
 6:             counter ← 0
 7:             for all child in sentence.children[node] do
 8:                 if min_size < 2 or max_size == 0 then
 9:                     unigrams.add([child])
10:                 end if
11:                 aux.add(child)
12:                 counter ← counter + 1
13:                 if counter > max_num_children then
14:                     aux.pop()
15:                     combinations.add(
16:                         NEXTC(node, sentence.children[node], sentence.leaves))
17:                     counter ← 0, aux ← [ ]
18:                     aux.add(child)
19:                 end if
20:             end for
21:             if length(aux) > 0 then
22:                 combinations.add(
23:                     NEXTC(node, sentence.children[node], sentence.leaves))
24:             end if
25:         else
26:             combinations.add(
27:                     NEXTC(node, sentence.children[node], sentence.leaves))
28:             for all child in sentence.children[node] do
29:                 if min_size < 2 or max_size == 0 then
30:                     unigrams.add([child])
31:                 end if
32:             end for
33:         end if
34:     end for
35:     return unigrams, combinations
36: end function
```

subtrees (difference between set of nodes and set of leaves). Note that the extracted subtrees are codified keeping the natural order of occurrence of words (from left to

Algorithm 2 Function *NEXTC*

Parameters: *idx* (index of the node), *children* (children nodes), *leaves* (leaves nodes).

Output: the list with the indexes of all subtrees in the tree.

```
1: function NEXTC(idx, children, leaves)
2:     Vars: ngram[ ], options[ ], combination[ ], list[ ], val_max, m
3:     for all r in [1, length(children)] do
4:         for all j in [1, r+1] do
5:             combination[j−1] ← j−1
6:         end for
7:         options ← [ ], ngram ← [ ], ngram.add(idx, −delizq−)
8:         for all z en [0, r] do
9:             ngram.add(children[combination[z]])
10:            if children[combination[z]] ∉ leaves then
11:                options.add(children[combination[z]])
12:            end if
13:            ngram.add(−delsep−)
14:        end for
15:        ngram.add(−delder−), list.add(ngram, options)
16:        top ← NUM_COMBINATION(length(children), r)
17:        for all j in [2, top+1] do
18:            m ← r, val_max ← length(children)
19:            while combination[m−1]+1 == val_max do
20:                m ← m−1, val_max ← val_max−1
21:            end while
22:            combination[m−1] ← combination[m−1]+1
23:            for all k in [m+1, r+1] do
24:                combination[k−1] ← combination[k−2]+1
25:                options ← [ ], ngram ← [ ]
26:                ngram.add(value, −delizq−)
27:                for all z in [0, r] do
28:                    ngram.add(children[combinations[z]])
29:                    if children[combinations[z]] ∉ leaves then
30:                        options.add(children[combinations[z]])
31:                    end if
32:                    ngram.add(−delsep−)
33:                end for
34:                ngram.add(−delder−), list.add(ngram, options)
35:            end for
36:        end for
37:    end for
38:    return list
39: end function
```

right) and the node indexes are assigned also following the same order.

The function *GET_SUBTREES* requires another function described in Algorithm 2, named *NEXTC*, which receives as input an index (that will be considered as the root node of the subtrees), a list containing indexes of its children nodes, the list of leaves nodes, and the minimum and maximum size of the subtrees. It returns the list of all extracted subtrees that contain the given root and satisfy the specified size.

Note that the extracted sn-grams are codified using the word indexes and the metalanguage proposed in [11]. In Algorithm 2 the tags $-delizq-$ and $-delder-$ denote the parenthesis [ ] in the metalanguage, which means the new level of the subtree, while the tag $-delsep-$ refers to the element *coma* which separate nodes at the same level.

An important aspect is the one related with the complexity of the algorithm. The problem has high complexity (higher than polynominal), so some sentences may be difficult to process by the algorithm because of their nature, especially those cases where nodes with a high degree (many children) are found. For example, in the sentences in which facts or things are listed, a node can have many children. In this case, the algorithm execution time will be unacceptable for practical purposes. Although in practice it is rare to find nodes with more than three children, the algorithm uses the parameter *max_num_children* to limit the number of children of a node and proceeds as follows: if the number of children is greater than the value of the parameter then only the first *max_num_children* are taken from left to right, discarding the rest of children. We set the default value of this parameter to 5.

Algorithm 3 shows the function *NUM_COMBINATION* that calculates the number of combinations of size r that can be obtained from a given list of elements sz. The algorithm 3 calculates the combinations $C(sz,r)$ implementing the equation 1:

$$C(sz,r) = \frac{sz!}{(sz-r)!r!}. \tag{1}$$

Algorithm 4 presents a function that receives a codified sn-gram using the proposed metalanguage and returns the size of the sn-gram calculated by adding the number of times the square parenthesis [and coma , appears in the sn-gram. The parameter *sngram* is the variable of string type and the function *count* () is the standard method that returns the number of times the argument occurs in the string.

Algorithm 5 introduces the function *COMPOUND_SNGRAMS* that generates new subtrees by the composition of subtrees, i.e., given a subtree with the root node v_i, it substitutes the node by the complete subtree into another subtree that contains it. The algorithm receives as parameters an adjacency table with the syntactic information of a sentence, set of initial subtrees with height equal to 1 (root node is at level 0), the minimum and maximum size of the subtrees. The function return as output a new set of subtrees obtained as the composition of subtrees (height greater than 1).

Algorithm 6 presents a function named *PREPARE_SNGRAM* that receives as parameters the sn-gram codified with the nodes indexes, the syntactic information matrix, an integer value that indicates the type of sn-gram (homogeneous or heterogeneous)

Algorithm 3 Function *NUM_COMBINATION*

Parameters: sz (number of elements), r (size of the combinations).
Output: The number of combinations of size r from a set of sz elements.

function NUM_COMBINATION(sz, r)
  Vars: $numerator, divisor, aux$
  if $sz == r$ **then**
    $numerator \leftarrow 1$
  else
    $numerator \leftarrow sz$
  end if
  for all i in $[1, sz]$ **do**
    $numerator \leftarrow numerator \times (sz - i)$
  end for
  $aux \leftarrow r$
  for all i in $[1, r]$ **do**
    $aux \leftarrow aux \times (r - i)$
  end for
  $divisor \leftarrow sz - r$
  for all i in $[1, sz - r]$ **do**
    $divisor \leftarrow divisor \times (sz - r - i)$
  end for
  $numerator \leftarrow numerator / (aux \times divisor)$
  return $numerator$
end function

Algorithm 4 Function *LEN_SNGRAM*

Parameters: *sngram* (representation of syntatic n-grams).
Output: Size of the sn-gram (number of nodes that contains the sn-gram)

```
1: function LEN_SNGRAM(sngram)
2:     Vars: n
3:     n ← 1
4:     n ← n + sngram.count([ )
5:     n ← n + sngram.count(,)
6:     return n
7: end function
```

and the information to be used (words, lemmas, POS or DR tags), as an output the function returns the sn-gram codified with the syntactic information instead of the nodes indexes.

Parameter op of the function *PREPARE_SNGRAM* indicates the type of sn-grams to extract: values from 0 to 3 refer to homogeneous sn-grams (of words, lemmas, POS and DR tags respectively), values from 4 to 6 refer to heterogeneous sn-grams with words as head elements, values from 7 to 9 refer to sn-grams with lemmas as head

Algorithm 5 Function *COMPOUND_SNGRAMS*

Parameters: *container* (first set of subtrees), *sentence* (syntactic information matrix), *min_size* (minimum size), *max_size* (maximum size).
Output: New set of subtrees.

```
 1: function COMPOUND_SNGRAMS(container,sentence, min_size, max_size)
 2:     Vars: newsngrams[ ], combinations[ ], candidates[ ], value.
 3:     for all item ∈ container do
 4:         if length(item) > 0 then
 5:             combinations.add(item)
 6:         end if
 7:         if item does not contain sentence.root_idx then
 8:             candidates.add(item)
 9:         end if
10:     end for
11:     while length(candidates) > 0 do
12:         candidate ← candidates.pop[0], value ← candidate.pop[0]
13:         value ← candidate.pop[0]
14:         for all combination ∈ combinations do
15:             if value ∈ combination[1] then
16:                 position ← first occurrence of value in combination[0]
17:                 sngram ← combination
18:                 sngram.pop(position)
19:                 sngram.add(position, candidate)
20:                 if LEN_SNGRAM(sngram) ∈ [min_size, self.max_size + 1] then
21:                     newsgrams.add(sngram)
22:                 end if
23:                 if LEN_SNGRAM(sngram) < max_size then
24:                     if sngram contains sentence.root_idx then
25:                         combinations.add(sngram)
26:                     else
27:                         combinations.add(sngram)
28:                         candidates.add(sngram)
29:                     end if
30:                 end if
31:             end if
32:         end for
33:     end while
34:     return newsngrams
35: end function
```

elements, values from 10 to 12 refer to sn-grams with POS tags as head elements and values from 13 to 15 refer to sn-grams with DR tags as head elements.

PREPARE_SNGRAM is a recursive function that in each invocation translates an

Algorithm 6 Function *PREPARE_SNGRAM*

Parameters: *line* (sn-gram index codification), *sentence* (syntactic information matrix), *op* (type of n-gram).

Output: an sn-gram codified with the corresponding information (words, lemmas, POS and DR tags), either heterogeneous or homogeneous.

```
1: function PREPARE_SNGRAM(line, sentence, op)
2:     Vars: ngram
3:     ngram ← ""
4:     for all item ∈ line do
5:         if data_type(item) is str then
6:             ngram ← ngram + item
7:         else if data_type(item) is int then
8:             if op == 0 then
9:                 ngram ← ngram + sentence.word[item]
10:            else if op == 1 then
11:                ngram ← ngram + sentence.lemma[item]
12:            else if op == 2 then
13:                ngram ← ngram + sentence.pos[item]
14:            else if op == 3 then
15:                ngram ← ngram + sentence.rel[item]
16:            else if op == 4 then
17:                ngram ← ngram + sentence.word[item]
18:                op ← 1
19:            else if op == 5 then
20:                ngram ← ngram + sentence.word[item]
21:                op ← 2
22:            else if op == 6 then
23:                ngram ← ngram + sentence.word[item]
24:                op ← 3
25:            else if op == 7 then
26:                ngram ← ngram + sentence.lemma[item]
27:                op ← 0
28:            else if op == 8 then
29:                ngram ← ngram + sentence.lemma[item]
30:                op ← 2
31:            else if op == 9 then
32:                ngram ← ngram + sentence.lemma[item]
33:                op ← 3
34:            else if op == 10 then
35:                ngram ← ngram + sentence.pos[item]
36:                op ← 0
37:            else if op == 11 then
38:                ngram ← ngram + sentence.pos[item]
39:                op ← 1
```

Algorithm 7 Function *PREPARE_SNGRAM* (cont.)

```
40:             else if op == 12 then
41:                 ngram ← ngram + sentence.pos[item]
42:                 op ← 3
43:             else if op == 13 then
44:                 ngram ← ngram + sentence.rel[item]
45:                 op ← 0
46:             else if op == 14 then
47:                 ngram ← ngram + sentence.rel[item]
48:                 op ← 1
49:             else if op == 15 then
50:                 ngram ← ngram + sentence.rel[item]
51:                 op ← 2
52:             end if
53:         else
54:             ngram ← ngram + PREPARE_SNGRAM(item, sentence, op)
55:         end if
56:     end for
57:     return ngram
58: end function
```

index of a sn-gram element into a linguistic type of information. For the cases of heterogeneous sn-grams, the function changes the value of the parameter *op*, so the elements other than the head are codified with a different type of information.

Finally, Algorithm 8 contains the main function that performs the extraction of syntactic n-grams named *EXTRACT_SNGRAMS*. The parameters that the function receives are the adjacency table with the syntactic information of a sentence, the integer variable that indicates the type of sn-grams to be extracted (heterogeneous or homogeneous, and the syntactic information to be used), the minimum and maximum size of the subtrees. As output, the function returns the extracted syntactic n-grams.

Conclusions

In this paper we presented the detailed description of the algorithm for extracting complete syntactic n-grams (heterogeneous and homogeneous) from syntactic trees. Syntactic n-grams allow obtaining full description of the information expressed in the syntactic trees that correspond to the sentences of texts. They are suitable as feature representation for several NLP problems, because they explore directly the syntactic information and allow to introduce it into machine learning methods, for example, identify more accurate patterns of how a writer uses the language.

We also presented a current state-of-the-art on usage of syntactic n-grams as features for natural language processing problems. In future research, we are planning to evaluate the syntactic n-grams as features for other NLP tasks such as question answering and sentiment analysis. For the authorship attribution task, we are considering to complement the sn-grams with other features from the literature such as

Algorithm 8 Function EXTRACT_SNGRAMS

Parameters: *sentence* (matrix with the syntactic information), *min_size* (minimum size), *max_size* (maximum size), *op* (type of sn-gram).
Output: the list containing the extracted sn-grams.

```
 1: function EXTRACT_SNGRAMS(sentence, min_size, max_size, op)
 2:     Vars: unigrams[ ], combinations[ ], aux[ ], sngrams[ ]
 3:     unigrams, combinations ← GET_SUBTREES(sentence, min_size, max_size)
 4:     if size of (unigrams) > 0 then
 5:         sngrams.add(sentence.root_idx)
 6:         sngrams.add(unigrams)
 7:     end if
 8:     for item in combinations do
 9:         if self.min_size <> 0 OR self.max_size <> 0 then
10:             size ← LEN_SNGRAM(PREPARE_SNGRAM(item, op))
11:             if size >= min_size and size <= max_size then
12:                 sngrams.add(item)
13:             end if
14:             if size < max_size then
15:                 aux.add(item)
16:             end if
17:         else:
18:             sngrams.add(item)
19:         end if
20:     end for
21:     if min_size <> 0 OR max_size <> 0 then
22:         COMPOUND_SNGRAMS(aux, sentence, min_size, max_size)
23:     else
24:         COMPOUND_SNGRAMS(combinations, sentence, min_size, max_size)
25:     end if
26:     return sngrams
27: end function
```

character n-grams, word n-grams, typed character n-grams in order to build a more accurate authorship attribution methodology.

With respect to the algorithm for the extraction of syntactic n-grams, we would like to implement different filter functions such as removing or keeping stop words in n-grams, n-grams of nouns, n-grams of verbs, etc. For example, with these functions, we will be able to extract syntactic n-grams only using stop words or nouns. We believe that syntactic n-grams of stop words will be an efficient feature set for the authorship identification problem given that it has been shown before that stop words play a crucial role in this task [30].

Acknowledgments

This work was partially supported by the Mexican Government and Instituto Politécnico Nacional (CONACYT project 240844, SNI, COFAA-IPN, SIP-IPN projects 20151406, 20161947, 20161958, 20151589, 20162204, 20162064).

References

[1] J. Diederich, J. Kindermann, E. Leopold, and G. Paass, "Authorship attribution with support vector machines," *Applied intelligence*, vol. 19, no. 1, pp. 109–123, 2003.

[2] J. Posadas-Durán, H. Gómez-Adorno, I. Markov, G. Sidorov, I. Batyrshin, A. Gelbukh, and O. Pichardo-Lagunas, "Syntactic n-grams as features for the author profiling task," in *Working Notes of CLEF 2015 - Conference and Labs of the Evaluation forum*, 2015.

[3] I. Markov, H. Gómez-Adorno, G. Sidorov, and A. Gelbukh, "Adapting cross-genre author profiling to language and corpus," in *Proceedings of the CLEF*, pp. 947–955, 2016.

[4] I. Markov, H. Gómez-Adorno, J.-P. Posadas-Durán, G. Sidorov, and A. Gelbukh, "Author profiling with doc2vec neural networkbased document embeddings," in *Proceedings of the 15th Mexican International Conference on Artificial Intelligence (MICAI 2016). Lecture Notes in Artificial Intelligence*, In press.

[5] J.-P. Posadas-Durán, G. Sidorov, I. Batyrshin, and E. Mirasol-Meléndez, "Author verification using syntactic n-grams," in *Working Notes of CLEF 2015 - Conference and Labs of the Evaluation forum*, 2015.

[6] E. Stamatatos, M. Tschuggnall, B. Verhoeven, W. Daelemans, G. Specht, B. Stein, and M. Potthast, "Clustering by authorship within and across documents," in *Working Notes Papers of the CLEF*, 2016.

[7] M. A. Sanchez-Perez, G. Sidorov, and A. Gelbukh, "The winning approach to text alignment for text reuse detection at PAN 2014," in *Working Notes for CLEF 2014 Conference*, pp. 1004–1011, 2014.

[8] M. A. Sanchez-Perez, A. F. Gelbukh, and G. Sidorov, "Adaptive algorithm for plagiarism detection: The best-performing approach at PAN 2014 text alignment competition," in *Experimental IR Meets Multilinguality, Multimodality, and Interaction - 6th International Conference of the CLEF Association*, pp. 402–413, 2015.

[9] M. A. Sánchez-Pérez, A. F. Gelbukh, and G. Sidorov, "Dynamically adjustable approach through obfuscation type recognition," in *Working Notes of CLEF 2015 - Conference and Labs of the Evaluation forum, Toulouse, France, September 8-11, 2015.*, 2015.

[10] G. Sidorov, F. Velasquez, E. Stamatatos, A. Gelbukh, and L. Chanona-Hernández, "Syntactic n-grams as machine learning features for natural language processing," *Expert Systems with Applications*, vol. 41, no. 3, pp. 853–860, 2013.

[11] G. Sidorov, "Non-continuous syntactic n-grams," *Polibits*, vol. 48, no. 1, pp. 67–75, 2013.

[12] S. Galicia-Haro and A. Gelbukh, *Investigaciones en Anlisis Sintctico para el español*. Instituto Politcnico Nacional, 2007.

[13] H. Beristáin and H. Beristáin, *Gramática estructural de la lengua española*. Universidad Nacional de México, 2001.

[14] C. D. Manning, M. Surdeanu, J. Bauer, J. Finkel, S. J. Bethard, and D. McClosky, "The Stanford CoreNLP natural language processing toolkit," in *Proceedings of 52nd Annual Meeting of the Association for Computational Linguistics: System Demonstrations*, pp. 55–60, 2014.

[15] M.-C. De Marneffe and C. D. Manning, "Stanford typed dependencies manual," tech. rep., Technical report, Stanford University, 2008.

[16] B. Santorini, *Part-of-speech tagging guidelines for the Penn Treebank Project (3rd revision)*. 1990.

[17] J.-P. Posadas-Duran, G. Sidorov, and I. Batyrshin, "Complete syntactic n-grams as style markers for authorship attribution," in *LNAI*, vol. 8856, pp. 9–17, Springer, 2014.

[18] E. Stamatatos, "A survey of modern authorship attribution methods," *Journal of the American Society for Information Science and Technology*, vol. 60, no. 3, pp. 538–556, 2009.

[19] P. Juola, "Future trends in authorship attribution," in *Advances in Digital Forensics III* (P. Craiger and S. Shenoi, eds.), vol. 242 of *IFIP International Federation for Information Processing*, pp. 119–132, Springer Boston, 2007.

[20] E. Stamatatos, "A survey of modern authorship attribution methods," *Journal of the American Society for information Science and Technology*, vol. 60, no. 3, pp. 538–556, 2009.

[21] G. Sidorov, F. Velasquez, E. Stamatatos, A. Gelbukh, and L. Chanona-Hernández, "Syntactic dependency-based n-grams as classification features," in *Mexican International Conference on Artificial Intelligence MICAI 2012*, pp. 1–11, Springer, 2012.

[22] G. Sidorov, "Syntactic dependency based n-grams in rule based automatic english as second language grammar correction," *International Journal of Computational Linguistics and Applications*, vol. 4, no. 2, pp. 169–188, 2013.

[23] S. D. Hernandez and H. Calvo, "Conll 2014 shared task: Grammatical error correction with a syntactic n-gram language model from a big corpora.," in *CoNLL Shared Task*, pp. 53–59, 2014.

[24] F. Rangel, F. Celli, P. Rosso, M. Potthast, B. Stein, and W. Daelemans, "Overview of the 3^{rd} author profiling task at PAN 2015," in *CLEF 2015 Labs and Workshops, Notebook Papers* (L. Cappelato, N. Ferro, G. Jones, and E. S. Juan, eds.), vol. 1391, CEUR, 2015.

[25] G. Sidorov, A. Gelbukh, H. Gómez-Adorno, and D. Pinto, "Soft similarity and soft cosine measure: Similarity of features in vector space model," *Computación y Sistemas*, vol. 18, no. 3, pp. 491–504, 2014.

[26] H. Calvo, A. Segura-Olivares, and A. García, "Dependency vs. constituent based syntactic n-grams in text similarity measures for paraphrase recognition," *Computación y Sistemas*, vol. 18, no. 3, pp. 517–554, 2014.

[27] V. Laippala, J. Kanerva, and F. Ginter, "Syntactic ngrams as keystructures reflecting typical syntactic patterns of corpora in finnish," *Procedia-Social and Behavioral Sciences*, vol. 198, pp. 233–241, 2015.

[28] R. Sennrich, "Modelling and optimizing on syntactic n-grams for statistical machine translation," *Transactions of the Association for Computational Linguistics*, vol. 3, pp. 169–182, 2015.

[29] B. Agarwal and N. Mittal, *Prominent Feature Extraction for Sentiment Analysis*. Springer, 2016.

[30] E. Stamatatos, "Plagiarism detection using stopword n-grams," *Journal of the American Society for Information Science and Technology*, vol. 62, no. 12, pp. 2512–2527, 2011.