Mosaic n-grams: Avoiding combinatorial explosion in corpus pattern mining for agglutinative languages

Balázs Indig

Pázmány Péter Catholic University, Faculty of Information Technology and Bionics
MTA-PPKE Hungarian Language Technology Research Group
H-1083 Budapest, Práter street 50/A
indig.balazs@itk.ppke.hu

Abstract

With the growing quantity of texts, the number and size of available corpora are increasing even in case of languages other than English. However, the sophisticated n-gram modelling methods that yield corpus patterns are not feasible for agglutinative languages like Hungarian. We invented the notion of mosaic-grams to store information required for mining frequent corpus patterns, which may contain parts that have only a certain constraint on the token like the lemma or the POS tag in itself. We describe an algorithm which aims to eliminate the numerous unnecessary combinations of features from the corpus to avoid combinatorial explosion and make mining of this kind of corpus patterns feasible.

1. Introduction

In the era of growing internet communication, the number of large corpora crawled from the web is increasing. Therefore, the classic notion of *corpus patterns* (Hanks, 2004) comes into fashion again. These pieces of information mostly lie in n-grams, but not in the traditional sense. There were experiments with *Syntactic n-grams* (Sidorov et al., 2013) and *Skip-grams* (Guthrie et al., 2006) as well for English. However, these techniques are not adaptable to agglutinative languages like Hungarian. In case of Hungarian, there are other similar concepts based on the bag-of-words model. As the word order in Hungarian is relatively free, phrases have many times only a little difference (e.g. regarding the conjugation of the verb/declination of the noun phrase), which we may not want to distinguish in our task and we want to treat these phrases as equal, still, maintaining the order of the tokens. The currently available methods (Sass, 2009; Vincze et al., 2011) use full syntactic parsing to handle similar phrases that differ only in the word order uniformly and are specialized in verb constructions and handle basic features like word form, lemma and POS tag (see example 1).

(1)

Form1:	*esett*	*szó*	*róla*
POS:	`V.Past.SG3`	`N.NOM`	`N.DEL`
	fall+P.SG3	word+NOM	it+DEL
Form2:	*róla*	*esik*	*szó*
POS:	`N.DEL`	`V.PRES.SG3`	`N.NOM`
	it+DEL	fall+Past.SG3	word+NOM
Pattern:	{`N.DEL`,	esik@lemma,	szó}

Meaning: (somebody)_said something_about it

In our approach we want to gather any pattern that might be stored in the "human parser" to speed up the parsing process by pattern matching and loading the pre-analysed subtree, no matter how it is called from the point of view of the theory[1]. In order to achieve this, we combine methods of *corpus patterns*, the *factored language models* and the *multi-word expression/terminology extraction* for Hungarian. We initially consider three features that can be extended by other features in the future. These three features are the *word form*, the *lemma*, and the *POS tag*. Each token has these features inherently. The main problem is that their numerous combinations – especially with longer n-grams (n>3) – make the whole process infeasible because combinatorial explosion occurs. Especially in agglutinative languages like Hungarian, where the number of different word forms lemmas and POS-tags which is at least an order of magnitude higher compared to English. In this paper, we present an algorithm that can prevent the combinatorial explosion by pruning duplicate subtrees of the possible patterns. This is a required step before we could apply the existing ranking algorithms that we introduce briefly in the next section.

[1]This phenomenon is called "Gestalt" in linguistics, but in computer science one would rather call it "caching".

2. Background

2.1. Methods for extracting coherent n-grams

In the literature, there are many concepts to search and identify multi-word expressions. Most of them use sophisticated statistical metrics (e.g. the *Pointwise Mutual Information (PMI)* and the *Pearsons Chi-Square Test*) to separate them from their contexts. There are even more complex metrics (e.g. the *LocalMax* algorithm (da Silva and Lopes, 1999) and the C-value/NC-value Method (Frantzi and Ananiadou, 1999)), but their common feature is that they operate solely on the frequency of the tokens in the used corpus. These metrics claimed to be better than the classic language modelling toolkits that can count perplexity and efficiently can compute the frequencies of specific n-grams. However, all the aforementioned tools have the common presumption that the elements, which are the source of n-grams, are in subsequent order in the text. So, all their optimisation are based on this fact. As we want to apply more features per token, these optimised methods are useless in their current form.

2.2. The properties of the used corpora

For Hungarian, we used the *Humor* morphosyntactical description (Novák, 2014). The number of tags exceeds 200, which is one order of magnitude greater than the 36 Penn POS tags used for English. The number of lemmata and word forms are also higher than in English.

We used three corpora with different sizes to test how the algorithm scales on them. The Szeged Corpus (Csendes et al., 2005) is a manually annotated corpus consisting of 70 000 sentences used for training data in most of the supervised learning tasks. The second corpus is the Hungarian Gigaword Corpus 2.0.3 which consists of 978 tokens including punctuation. It covers various sources (e.g. cross-border newspapers, parliament logs and legal tests) (Oravecz et al., 2014). The Pázmány Corpus contains 1.2 billion tokens crawled from various Hungarian news portals (Endrédy, 2016). These three corpora contain most of the Hungarian texts currently available for language technology purposes.

3. Our approach

3.1. The root cause of combinatorial explosion

Experiments with smaller datasets showed that the root of the problem is that many of the combinations are duplication. Table 1 shows that there are many types of generalisations resulting in multiple forms that are considered as duplicates.

trigram			freq
esett	szó	róla	10
esik@lemma	szó	róla	12
[V][Past][Sg3]	szó	róla	12
...			
esik@lemma	szó@lemma	[N][Del]	12
esik@lemma	[N][Nom]	róla	12
...			
[V][Past][Sg3]	[N][Nom]	[N][Del]	63

Table 1: A sample of the forms generated from the trigram *esett szó róla* used in example 1. The trigrams are sorted by frequency. The first group (in the first three lines of the table) contains the concrete form and its slight generalisation. The second group contains other slight generalisations with the same frequency; these are considered as duplicates. The third group contains overgeneralisation.

3.2. The anatomy of n-grams

The straightforwardness of the trigram models in language modelling lies in the fact that every token has two immediate neighbours, one to the left and one to the right. However, at the beginning and at the end of the sentences there are no natural neighbouring tokens in both ways, therefore the classical n-gram models put two extremal elements – the so-called *sentence start* and *sentence end* token – to every sentence. Using this trick, all the tokens (except the extremal symbols) have two neighbours.

The reason that there is no higher-order model than trigram-models is that any higher-order n-gram can be (a) described with trigrams, (b) interpreted as the further constraint of a trigram from the appropriate side by non-immediate neighbours from viewpoint of the text[2]. It is because all tokens can be constrained from either side by another token as there are many sibling tokens that can precede or succeed a specific token like in a tree on each side. In example 2 the *original* form can be described with *part1* and *part2* occurring together and *part1* can be interpreted as the further constraint of the left context of word *esett*. Similarly, *part2* can be interpreted as the further constraint of the right context of the word *szó*.

(2)

original:	*Sok*	*szó*	*esett*	*róla*
part1:	[Sok	szó	esett]	
part2:		[szó	esett	róla]
	many	word+NOM	fall+Past.SG3	it+DEL

Meaning: there was a lot of talk about it

This fact enables us to use trigrams to filter out unnecessary features because the filtered features would cause the same subtrees for higher-order n-grams as the ones which were not pruned (see Table 2).

sok	szó	esett	róla
			ő@lemma
			N.Del
sok@lemma		esik@lemma	róla
			ő@lemma
			N.Del
Adj		V.Past.Sg3	róla
			ő@lemma
			N.Del

Table 2: The many contexts of the word form *szó* 'word'. The second right context shows the tree-style duplication of longer patterns based on the chosen feature for the immediate right context. This unnecessary duplication – which can be found in other (agglutinative) languages as well – results in a combinatorial explosion. As the duplication here is caused by the multiple equivalents, but different forms (e.g. tense) of *esett* 'fall+Past'. It is essential to select only the necessary forms of it to reduce the number of duplicates in the context of this table. This mechanism needs to be further carried on with every other word and every other context.

3.3. Solution

The task of identifying the unnecessary features resulting in duplications is not trivial because of the number of mosaic-trigrams. Some of these features are trivial: punctuations should not have separate lemmata because their frequencies equal all the time. This holds for the conjunction words that cannot be suffixed as well. Also, there are cases which hold only in a specific corpus. For instance, if a word only occurs in one sense or form in the whole corpus, there is no need to disambiguate it from the other non-occurring siblings. According to Zipf's law, most of

[2]Still, we need higher-order n-grams for the actually sought patterns.

the words in the text occur only once. Most of their lemmata also occur once, so they can be pruned.

(3)

Form1:	*esett*	*szó*	*róla*
POS:	V.PAST.SG3	N.NOM	N.DEL
	fall+Past.SG3	word+NOM	it+DEL
Pattern:	esik@lemma	szó	N.DEL

In some very interesting cases, the above mentioned "elimination rules" hold only in a specific neighbourhood (trigram). These relationships must be mined statistically from the specific corpus (see example 3).

4. Method

4.1. Overview

First, we generate all the mosaic-trigram combinations from the corpus keeping the origin trigrams for later processing, and then count their frequencies by using the *origin n-gram mapping* described in section 4.4.. We want to reduce those records which are equal in their frequency and their origin n-gram after the mapping, keeping only the *most concrete variant*. The reducing step uses a metric which is defined in section 4.5.. The remaining mosaic-n-grams are the true trigram basepatterns that should be kept and used for the generation of higher-order mosaic-grams. With these basepatterns we created a filtered corpus in order to apply some of the modules again for higher-order mosaic-grams in our processing pipeline.

4.2. Preprocessing

First, we marked the type of feature that the string came from, as there are word forms which are equivalent with their own or some other word's lemma. Technically, we introduced the suffix notation of "[token]@lemma" for lemmata to ease the sorting of the different word forms. The word form is left as it is and the POS tag was already in a fixed extremal form in square brackets (see example 4).

(4)

word:	*esik*	*szó*	*róla*
lemma:	esik@lemma	szó@lemma	ő@lemma
POS:	[V][Pres][s3]	[N][NOM]	[N][DEL]
	fall+Pres.SG3	word+NOM	it+DEL

4.3. Generating the mosaic-trigrams efficiently

We represented the sentence in a sparse 2D matrix where the first row was the tokens following each other, and from the second to the last row came the features for the corresponding token. By this notation, one token with its features was represented in a column. The representation allows to have a different number of features for each column. The first empty element marks the end of the feature sequence for a specific token. This representation enabled us to make a priori filtering too. In this matrix, we used the indices to get the required features and to iterate through the sentence generating all the mosaic-trigrams. We also generated the original trigram for each mosaic-gram in order to use it in the later processing steps. The format is similar to the CoNLL format family, but it is transposed for readability. In example 5, we use the trigram *esik#esik#[V][s3] szó#szó#[N][NOM] róla#ő#[N][DEL]*, but we omit *ő@lemma* from the example to show how it behaves when there are varying number of features for tokens.

(5)

origin:	...	...	...
feat1:	esett	szó	róla
feat2:	esik@lemma	szó@lemma	[N][DEL]
feat3:	[V][s3]	[N][NOM]	∅
feat4:	∅	∅	∅

4.4. Counting frequencies with origin trigram mapping

With the origin trigram, we were able to identify which origin trigrams produced such mosaic-trigrams that are equal in frequency and therefore do not prove to be good abstractions and should be deleted. This usually occurs when a mosaic-trigram is not shared between different origin trigrams (they are generated from the same origin trigram).

To get a comparable result on the abstractions we must use a special counting algorithm which basically checks if the mosaic-trigrams of the two compared records are equal and then chooses the origin trigram, which is less than the other in *lexical comparison*. The monotonicity of the lexical comparison operator ensures that the two mosaic-trigrams that had only been generated from the same source trigrams – which can be numerous distinct trigrams – are mapped to the same origin trigram, which allows later comparison. If one of the records has an origin which is not common with the other record, but the mapping results in the same origin trigram, they can be distinguished by their frequencies which will not be equal.

4.5. Reducing step

In order to delete those mosaic-grams that proved to be a duplicate of each other and keep only one, we introduced a *metric* to compare the mosaic-grams by their *concreteness*. We assign a score for each token based on the type of the remained feature and sum the scores. The mosaic-gram with the higher score is kept as the most concrete abstraction of the equally strong abstractions of the specific origin trigram. We defined the following scores: POS tag - 1, lemma - 2, word form - 4 (see table 3).

score	freq			
7	8	esik	szó@lemma	[N][DEL]
9	8	esik	szó	[N][DEL]
9	12	esik@lemma	szó	róla

Table 3: Scoring mosaic-trigrams: the more concrete one gets the higher score

4.6. Creating the filtered corpus

All the previous steps of the algorithm required an on-disk storage as the number of patterns were too high, but this was not problematic as no random access is needed. The corpus filtering phase heavily requires random access to the filtered mosaic-trigrams. The main problem is that in

the corpus the mosaic features are entangled, therefore we can not do full string search. Creating regular expressions from the many patterns would require an enormous computation power and possibly exceed the amount of memory available. One feasible solution would be the splitting of the patterns into many automata (prefix trie) distributed across the cluster so one could query if any of the automata contains a specific mosaic-trigram from the corpus. This solution can be cumbersome if one would like to parallelise the algorithm because of the size of the corpus.

Hence we came up with a solution that can be easily parallelised, scalable and considerably fast. We split the corpus into parts and for each part, we generated all the mosaic-trigrams in it[3]. We put the resulting mosaic-trigrams into a fast implementation of the prefix trie (Yoshinaga and Kitsuregawa, 2014) and filtered them through the list of the remaining mosaic-trigrams[4] to get the remaining mosaic-trigrams for the specific part of the corpus. With this trie we could filter the specific part corpus. By generating all the mosaic-trigrams for each trigram in the corpus we can clearly distinguish any kept feature by its neighbouring features from the ones to be deleted. We checked for the first marked mosaic-trigram in the prefix trie, which contained the middle feature of that mosaic-trigram. If such marked trigram is found in the tree, we include the feature in the resulting filtered corpus, else we omit the feature for the current trigram. This procedure deletes features that proved to be unnecessary by all the features of the neighbouring tokens. On the filtered corpus we could run the mosaic-gram generator algorithm for higher-order mosaic-grams without generating unnecessarily duplicate entries. In the following sections, we describe two ways for the production of the filtered mosaic-trigram list. The corpus filtering method is common between the two algorithms.

5. Implementation of the filtering algorithm

We expected the described algorithm to work with even the largest corpora available for Hungarian, so the whole implementation was done in a *map-reduce* style. We wanted to be able to create a rapid prototype of our algorithm, so we experimented with *Apache Spark* (Zaharia et al., 2016) and its *Python API (Pyspark)*. We also created our specific pipeline of common Unix tools to get a comparable result. The algorithm described in section 4. is designed from the ground up to be able to work with both the *map-reduce* style and the *merge-sorting* style computation methods to be able to compare them. We used a small number of heterogeneous machines in our department for testing. We used a common fairly-fast storage as the underlying file system to store the data.

5.1. The map-reduce solution

The *mosaic-gram generation step* was written as a *mapping* task, and the *reducing step* was implemented as a *reduce* task. Our implementation is based on the classic example of *map-reduce style word counting* as a pattern. But instead of *the word extraction step*, we generated the mosaic-trigrams and the tuples of the origin-trigram and the frequency (initially 1), in order to help the hash function to put the same mosaic-trigram to the same bucket for counting. After this step, we reduced the same mosaic-trigrams by counting their frequency and keeping the lexically larger origin-trigram. To continue with the next step, we needed to reverse the dictionary. This involved rehashing by the tuple of the counted frequency and the origin trigram. On the reversed dictionary, we could remove the mosaic-trigrams that compared less by our metric. Finally, we printed the resulting mosaic-grams.

The main problem was that due to the nature of the map-reduce framework a key and a value field had to be defined for each record, which was very artificial in the concrete case, but the whole program could be written in about a hundred lines of code.

5.2. The merge-sort style solution

We split the corpus with the *GNU split* utility to specific-sized parts splitting strictly on the record boundary. We used *GNU parallel* (Tange, 2011) to run the mosaic-trigram generation, counting, and sorting task on multiple machines and cores for all parts of the corpus. The mosaic-trigram generation and the counting was implemented in *MAWK*[5], which is a faster *AWK* implementation than *GNU AWK*. We could not use the *GNU uniq* utility because of our special counting need, but the algorithm implemented in MAWK resembles the same concept. The sorting was done with *GNU sort* in the binary locale (C) with the *parallel* option and all previously sorted chunks were merged with the *-m* option. All intermediate data were compressed with *pbzip2*.

er the merging, another turn of counting was necessary on the resulting set. Then the data were split again to be sorted in a distributed manner by the frequency and the origin-trigram as keys. Along with the sorting, the preliminary filtering of the equivalent keys took place, which was followed by merging and another run of filtering. This resulted in the final data. Note that instead of the two hashings, this solution contained two distributed merge-style sortings.

6. Results

The results of the two described algorithms are in Table 4. One can see that the pipeline of the Unix tools is almost 2 times faster than the Spark variant of the algorithm. This is due to the heterogeneous cluster that we could use for our experiment. The Spark scaled itself down to the largest common amount of RAM for all the machines in the cluster and did not utilise the rest of the resources. During the mosaic-trigram generation, the memory often overflowed and Spark had to write the temporary data to the disk, which resulted in a major slowdown. The Unix tools allowed more configurability, but required the expertise because of the number of command line switches. The

[3]This step is already made when generating the mosaic-trigrams in parallel.

[4]We checked each pattern from the remaining mosaic-trigrams for containment in the prefix trie and marked the positive result in the trie.

[5]http://invisible-island.net/mawk/

final pipeline with the optimised split size of the data along with the perfect combination of command-line switches resulted in a fast and elegant way of computing the results in a scalable manner, even on our heterogeneous cluster. The implementation of the two algorithms required almost the same time.

Corpus	Szeged	HGC	Pázmány
No. of mosaic-trigrams	16.5*10^6	5.1*10^9	21*10^9
No. of filtered mosaic-trigrams	1.75*10^6	538*10^6	2*10^9
Time Spark	16 min	6 days	20 days
Time Unix tools	5 min	4 days	12 days

Table 4: The running times and the number of unfiltered and filtered mosaic-trigrams

7. Conclusion and future work

The task of computing higher-order mosaic-grams proved to be harder than we imagined. Even for medium-sized corpora, special, carefully designed, distributed algorithms were necessary. In this paper, we presented the first part of our ongoing research, which could not be finalised as the required multi-word expression extractor algorithms were not designed in a form that suits distributed computing. We presented our novel notion of the *mosaic-grams* and our corpus filtering approach that helps to eliminate unnecessary features from the corpus in order to reduce the number of possible states in the later processing[6]. From the raw data, we see that there is potential in this approach, but it requires the careful design of the ranking algorithm.

The map-reduce style implementation was easier to implement but left almost no space to fine-tune the algorithm. The generation of mosaic-trigrams mostly resulted in disk writes, because the data was suddenly grown large. The heterogeneity of the cluster resulted in another problem: Spark used only the amount of RAM available in the smallest machine on every node, wasting valuable resources. This slowed down the processing very much. The fast, well-tested and generic utilities, which allowed more control in the pipeline, turned out to run significantly faster. They were able to handle the heterogeneity of the cluster as well as the size of the batches could be tuned for speed. Spark is still an immature platform, so we expect some improvement in the configurability and also in the available tools, for example, distributed prefix tries.

8. References

Csendes, Dóra, János Csirik, Tibor Gyimóthy, and András Kocsor, 2005. The szeged treebank. In Václav Matoušek, Pavel Mautner, and Tomáš Pavelka (eds.), *Text, Speech and Dialogue: 8th International Conference, TSD 2005, Karlovy Vary, Czech Republic, September 12-15, 2005. Proceedings*. Berlin, Heidelberg: Springer.

da Silva, J Ferreira and G Pereira Lopes, 1999. A local maxima method and a fair dispersion normalization for extracting multi-word units from corpora. In *Sixth Meeting on Mathematics of Language*.

Endrédy, István, 2016. *Nyelvtechnológiai algoritmusok korpuszok automatikus építéséhez és pontosabb feldolgozásukhoz [Language technology algorithms for automatic corpus building and more precise data processing]*. Ph.D. thesis, PPKE-ITK, Budapest.

Frantzi, Katerina T and Sophia Ananiadou, 1999. The C-value/NC-value domain-independent method for multi-word term extraction. *Journal of Natural Language Processing*, 6(3):145–179.

Guthrie, David, Ben Allison, Wei Liu, Louise Guthrie, and Yorick Wilks, 2006. A closer look at skip-gram modelling. In *Proceedings of the 5th International Conference on Language Resources and Evaluation*.

Hanks, Patrick, 2004. Corpus pattern analysis. In *Euralex Proceedings*, volume 1.

Novák, Attila, 2014. A New Form of Humor – Mapping Constraint-Based Computational Morphologies to a Finite-State Representation. In *Proceedings of the Ninth International Conference on Language Resources and Evaluation (LREC'14)*. Reykjavik, Iceland: ELRA.

Oravecz, Csaba, Tamás Váradi, and Bálint Sass, 2014. The Hungarian Gigaword Corpus. In Nicoletta Calzolari (Conference Chair), Khalid Choukri, Thierry Declerck, Hrafn Loftsson, Bente Maegaard, Joseph Mariani, Asuncion Moreno, Jan Odijk, and Stelios Piperidis (eds.), *Proceedings of the Ninth International Conference on Language Resources and Evaluation (LREC 2014)*. Reykjavik, Iceland: European Language Resources Association (ELRA).

Sass, Bálint, 2009. A unified method for extracting simple and multiword verbs with valence information and application for hungarian. In *RANLP*.

Sidorov, Grigori, Francisco Velasquez, Efstathios Stamatatos, Alexander Gelbukh, and Liliana Chanona-Hernández, 2013. *Syntactic Dependency-Based N-grams as Classification Features*. Berlin, Heidelberg: Springer Berlin Heidelberg, pages 1–11.

Tange, O., 2011. GNU Parallel - The Command-Line Power Tool. *;login: The USENIX Magazine*, 36(1):42–47.

Vincze, Veronika, T István Nagy, and Gábor Berend, 2011. Detecting noun compounds and light verb constructions: a contrastive study. In *Proceedings of the Workshop on Multiword Expressions: from Parsing and Generation to the Real World*. ACL.

Yoshinaga, Naoki and Masaru Kitsuregawa, 2014. A Self-adaptive Classifier for Efficient Text-stream Processing. In *COLING*.

Zaharia, Matei, Reynold S Xin, Patrick Wendell, Tathagata Das, Michael Armbrust, Ankur Dave, Xiangrui Meng, Josh Rosen, Shivaram Venkataraman, Michael J Franklin, et al., 2016. Apache Spark: A unified engine for big data processing. *Communications of the ACM*, 59(11):56–65.

[6]For the actual implementation, see: `https://github.com/ppke-nlpg/mosaic-n-grams`