One format to rule them all – The `emtsv` pipeline for Hungarian

Balázs Indig[1,2], Bálint Sass[1], Eszter Simon[1], Iván Mittelholcz[1], Noémi Vadász[1], and Márton Makrai[1]

[1]Research Institute for Linguistics, Hungarian Academy of Sciences
[1]`lastname.firstname@nytud.mta.hu`
[2]Centre for Digital Humanities, Eötvös Loránd University
[2]`lastname.firstname@btk.elte.hu`

Abstract

We present a more efficient version of the `e-magyar` NLP pipeline for Hungarian called `emtsv`. It integrates Hungarian NLP tools in a framework whose individual modules can be developed or replaced independently and allows new ones to be added. The design also allows convenient investigation and manual correction of the data flow from one module to another. The improvements we publish include effective communication between the modules and support of the use of individual modules both in the chain and standing alone. Our goals are accomplished using extended `tsv` (tab separated values) files, a simple, uniform, generic and self-documenting input/output format. Our vision is maintaining the system for a long time and making it easier for external developers to fit their own modules into the system, thus sharing existing competencies in the field of processing Hungarian, a mid-resourced language. The source code is available under LGPL 3.0 license[1].

1 Introduction

The `e-magyar` processing system (Váradi et al., 2018) integrates the state-of-the-art Hungarian NLP tools into a single, easy-to-use, maintained, and updated system. It has been designed to facilitate both research and application-oriented processing with the important goal of the system being fully open for research purposes, thus encouraging future expansion, but also being easy for the non-NLP audience to use, and to become a good experimental tool, delivering the best performance available, regarding both processing speed and correctness.

Since its publication, the system has become popular and widely used in the Hungarian NLP community. Attempts have also been made to analyze large corpora with it, such as the Hungarian Webcorpus (Halácsy et al., 2004) and the Hungarian Gigaword Corpus (Oravecz et al., 2014). This work led to the discovery of previously unknown errors and weaknesses, which were taken into account in our developments. In this article, we present our work with two aspects emphasized: the unified communication format and the architecture design.

In the first version of `e-magyar`, the intermodular communication format was the internal `xml` format of GATE (Cunningham et al., 2011), into which the Hungarian system was integrated. However, user experience showed that most users do not know or want to use the GATE system for their work: users with linguistic interest found it inconvenient, while for those with a technical background, it was unnecessarily cumbersome. In many cases, GATE introduces unnecessary complexity regarding installation, debugging, the format, and resource demand, due to the `xml`-based *standoff* annotation (see Section 3), which in many cases undermines stability. Therefore, we voted for the development of a new, standard and GATE-independent inter-modular communication format opening the way to use existing devices as separate modules or with transparent inter-modular messages. The format also simplifies the manual modification of inter-modular content. Available tools, even those independent of their programming language, become easier to integrate into the system.

Another focus of the development was on rethinking the architecture design. Modules which were available before the creation of `e-magyar` were written in various programming languages, following different linguistic annotations, and lacking a modularized and transparent structure. In contrast, our principles are unifor-

[1]`https://github.com/dlt-rilmta/emtsv`

mity, interoperability, comparability, and the interchangeability of individual modules (e.g. when a new candidate performs better).

In this article, we show how we converted the tools of the previous e-magyar version following the UNIX toolbox philosophy: "each does one thing and each does it very well". Restructured modules are supposed to be able to both operate independently of each other or interacting, as needed. Also sections of the pipeline can be run, i.e. users can enter or exit at any point and can modify the data manually, as long as they adhere to format requirements, which are natural and programming language agnostic by design from the beginning.

Design properties of other processing chains were kept in mind during the development of emtsv for the sake of comparability. Other systems mostly take strictly a single natural language as their starting point, but then they are extended to be multilingual or even intended to be universal afterwards. Some of them go with changed needs, which now favor scalable cloud-based technologies – dubbed *microservices* – that do not require end user installation: the chain is provided as a service, sometimes without source code.

In parallel with the development, the potential use of emtsv was also contemplated. For instance, emtsv could be profitable for pre-annotating tasks in corpus building. Thanks to the high performance of the modules, pre-analyzing the text could shorten and ease the otherwise expensive and protracted human annotation. Furthermore, the modular architecture of the toolchain allows us to exit at a certain point of the analysis, carry out some manual correction in the data, and then enter the chain again putting the data back to emtsv. Let us take an imaginable workflow as an illustrative example. Firstly, the output of tokenization and tagging is corrected manually for a revised, finer input for the dependency parser, the second step of the workflow. Therefore, the effect of occurrent errors from the tokenizer or the part of speech (POS) tagger could be eliminated. It is worth to mention, that since the dependency parser allocates ID numbers to each token, modifications in tokenization (inserting, deleting, splitting, or joining tokens) do not cause complications in token numbering. At the end of the workflow, the output of the dependency parser is converted into the CoNLL-U format[2], which is edible for widely-used annotation and visualization tools, e.g. allowing to carry out further corrections in the dependency graph in a drag-and-drop manner.

In Section 2, we present the currently available language processing systems similar to emtsv for the sake of comparison. Section 3 describes our extended tsv format, Section 4 gives an overview of the architecture, while Section 5 presents the individual modules. Section 6 summarizes the paper and Section 7 presents future work.

2 Related Work

As an NLP pipeline primarily for Hungarian, emtsv can be compared to Magyarlánc (Zsibrita et al., 2013), currently in version 3.0, and the hun⋆ toolchain[3]. Though there are overlaps between the modules of the compared chains, here we focus on the structure of the chain as a whole, which is, to some extent, independent of the individual modules.

Magyarlánc provides a Java-based, tightly coupled chain, using the latest international state-of-the-art modules. It is suitable for annotating a large amount of Hungarian text with detailed and proper linguistic analysis, but the modification of the system (e.g. adding possible new modules or replacing existing ones) is cumbersome.

The most relevant modules of the hun⋆ toolchain are the HunToken[4] sentence and word tokenizer (written in Flex and Shell), the HunMorph morphological analyzer (Trón et al., 2005) (w. in OCaml), the HunPOS POS tagger (Halácsy et al., 2006) (w. in OCaml), the HunNER named entity recognizer (Varga and Simon, 2007) (w. in Python), and the HunTag sequential tagger (Recski and Varga, 2009) (w. in Python). The hun⋆ chain and emtsv share several properties: the loosely coupled architecture, the tsv format, the heterogenity of programming languages applied for the development of the modules, and the open source availability. However, hun⋆ only works with Latin-2 character encoding and its direct development has discontinued.

There are several examples of systems similar to Magyarlánc on the international scene, which usually suffer from similar shortcomings

[2]http://universaldependencies.org/format

[3]https://hlt.bme.hu/en/resources/hun-toolchain

[4]https://github.com/zseder/huntoken

mentioned above. At the same time, they have merits like being language independent (or at least supporting many natural languages), fast or able to process large amounts of data. Currently we do not intend to compete with all of these aspects, but focus on producing the best results for Hungarian the most efficiently, and creating a format that is close to standards and easy to convert to other ones. We want to give full control to the user by creating a loosely coupled system. The point here is to involve the community in the development: for transparent operation, systems not only need to be open-source but also need to be accepted and maintained by the NLP community, which is more difficult to achieve.

In the remainder of this section, we highlight a few existing language-independent analyzers to present some of their disadvantageous properties which tend to be common with the tools not included.

`UDPipe` (Straka and Straková, 2017) was written in C++ roughly at the same time as `e-magyar` with the goal of analyzing general texts. Training data follows the Universal Dependencies and Morphology (UD)[5] annotation scheme and format. Although it has bindings for many programming languages, and is truly efficient, it does not allow for easy extension and development of the applied pipeline, despite that its source code is free[6]. This is a shortcoming if developers want to introduce their own modules, such as a custom morphological analyzer.

The Python-based `spaCy`[7] started similarly, originally consisting of closed-end modules, but since version 2.0, it has become more and more open in architecture in order to support more natural languages. Although `spaCy` and `emtsv` are similar in their direction of development, their current status is too far to allow comparison: `emtsv` is more loosely coupled.

Another strategy is followed by `WebSty`[8] and `Weblicht`[9]. These pipelines try to integrate existing tools including even language-dependent ones to better support individual languages. Their only criterion is that the tools have to support the UD format. The principle of this approach is scalability in great computer clusters: running in the cloud asynchronously orchestrated by a task scheduler on demand. The entire system is accessible via a web-based API, where tasks can be specified with data files. The source code of the software is not available for running a local instance, and modules cannot be developed by external developers.

In `emtsv`, we try to eliminate the architectural drawbacks of previous systems described above and, at the same time, reserve their advantageous features.

3 Uniform Data Format

The classic structure of `e-magyar` (Sass et al., 2017) heavily relies on the features inherited from the original tools, depending on their input and output formats. In that system, GATE is the layer of architecture that creates a common, unified data format, thus providing interoperability between the individual modules, that are agnostic of each other. This idea is suitable as long as the user wants to work within the GATE ecosystem.

The common format is GATE `xml`, which is not a standard and easy-to-implement solution, as no DTD or Schema file is available that describes the format. These are dispensable as long as files are produced and processed solely by GATE: the format can be regarded as internal. In a GATE `xml` file, annotation follows the complete text separately. In a typical scenario of processing this format, one must constantly jump between the two parts of the `xml` file, so the entire text and annotation should be kept in memory e.g. by building a tree with DOM strategy. This requirement at best slows down the processing of large `xml` files, whereas it makes impossible to process the data as stream. In addition, the cumbersome deployment of the GATE system in itself greatly increased the complexity of the pipeline for users, developers, and service providers, whether or not they really needed the added functionality provided by GATE.

This motivated us to design an inline (i.e. in the sense that annotation should be locally available at the element which is annotated), streamable, simple, customisable, self-documenting and easy-to-use format that can be easily converted into other formats. We support conversion without data loss to standard formats such as CoNLL-X (Buchholz and Marsi, 2006), CoNLL-U, or even GATE `xml`. The newly chosen format specifica-

[5]http://universaldependencies.org
[6]https://github.com/ufal/udpipe
[7]https://spacy.io
[8]http://ws.clarin-pl.eu/websty.shtml
[9]https://weblicht.sfs.uni-tuebingen.de

```
form      lemma  xpostag
#This is a comment.
A         a      [/Det|Art.Def]
kutyák    kutya  [/N][Pl][Nom]
ugatnak   ugat   [/V][Prs.NDef.3Pl]
.         .      [Punct]

A         a      [/Det|Art.Def]
...
```

Table 1: An illustration of the format, a three-column `tsv` file with a header (resembling CoNLL-U column names): word forms, lemmas, and explicit morphological analysis. 'The dog-s bark-[3Pl]. The…'. Comments may occur only at the beginning of sentences.

tion allows adaptation to needs as they emerge (we will see that this is achieved by the flexible definition of `tsv` columns), mainly consisting of recommendations (e.g. free text and JSON are preferred as data), and as few constraints as possible.

We use `tsv` files with a header (Table 1), which can even be loaded into spreadsheet editors. Adhering to the classical vertical format, each row specifies a token, and columns (fields, cells) contain annotations for the token. We introduced two additions to the simple `tsv` following the CoNLL-U format: (i) sentence boundaries are marked with empty lines, and (ii) it is possible to insert comments in the forms of lines starting with a hashmark (#) before each sentence, which will be copied to the output. Although the sentence block comment was possible – switchable, not allowed by default – for optional comparability with the CoNLL-U format, its use is not recommended because of the combination of the free column order and hashmark as control character[10]. We recognize the legitimacy of the line starting hashmark in CoNLL-U due to (i) the fixed order of columns and (ii) the constraint for for the first column to be a positive integer number (more precisely mark the number of token in the sentence). However, we prefer the locality property in our format which allows to process individual tokens, without needing to know their context – where it is useful – compared to sentence leading comments or fixed column order.

The role of the header is particularly important: it determines the operation of the whole system. Modules identify the location of their input data required for processing by strictly defined column names in the header (regardless of the order of the columns), and similarly, they place their output in new columns (with strictly defined names), leaving all other columns unchanged. A consequence is that modules are not allowed to change the number and content of input rows. (If users are about to create a module that will change the number of rows in the future, e.g. by splitting a token to more, they have to be very careful about the contents of the fields in the new rows and the integrity of the complete data, especially in the case of sequential tags.)

Newly created columns are simply placed after the existing columns in the current implementation. This can be taken as our recommendation, but not a mandatory restriction, as columns are identified by name. This way the text remains readable for the human eye, and logically related pieces of annotation are stored close to each other. It is an important property that developers can add any number of extra columns: there is space for expansion with additional information on demand. Column naming and content conventions have to be established by agreement between the producing and processing modules. The recommended field content is free text or the standard `JSON` format[11], which enables passing bound structures without ad-hoc formats or special characters (like they are used to represent lists of key-value pairs in CoNLL-U). In addition, the `JSON` format is suitable to represent alternative analyses or ambiguous annotation, e.g. as a (weighted) list of possible tags.

4 Architecture

The described `tsv` format is simple, easy to manage, supported by several existing tools, and enables users to write additional modules. It was our primary goal to facilitate the easy development and integration of additional modules into the system. Furthermore, besides the traditional command-line interface (CLI) and the format-agnostic Python library interface, we have also created a REST API whose use is independent of programming languages.

[10]Hashmark as every character, however rare, will have its occurrences in a large corpus: using it as special will lead to error on the long run. Collision of occurrences as literals and as special characters in the original corpus often results in unexpected errors that take a long time to debug, limits and slows down the operation and later the extensibility of the system.

[11]Although the spacing between the structuring elements in `JSON` can be selected to be tab, it is prohibited in `emtsv` because of its tab separated layout.

With the help of traditional UNIX pipelines, CLI provides a useful tool for advanced users. The CLI can be used even on large texts without knowing the internal operation of the modules. The Python library can be integrated into larger software systems by IT/NLP users. Finally, the REST API opens up the possibility of using the system according to modern cloud-based trends, even for completely non-NLP users and business circles: with its help, emtsv can be made available as a scalable service in the cloud for a wide range of end-users, without a need for installation on the end-user side.

According to the modern requirements, the emtsv system is also available as a Docker image[12]. This image can be used like a 'standalone executable' with 'batteries included' as it features the CLI interface and the REST API as well. Its advantage over the traditional installation is that the whole system is packed together with all its dependencies pre-configured and can be deployed with a single command. Therefore it is easy to use on any machine running Docker in a form comparable to highly integrated pipelines. The deployed image can instantly be used with HTTP requests from local or remote computers, from the command line or from any software.

Individual modules are combined together by our newly developed xtsv framework, which handles tsv as a communication format in a general way. This allows both the communication via the format described in Section 3 (i.e. the choice of the input columns, attaching the output columns, and reserving the rest), the creation of REST APIs, and the dynamic format-check (Section 5) regardless of the specific content of the modules. Extra modules can be added to the system with the following parameters specified in a declarative fashion: the unique name of the module, – that distincts it from alternative instances of the same tool with a different model or parameter setting –, the actual tool that performs the function of the module, the names of the input and output columns, and the specification of models and other parameters when needed as the parameters of the tool. If one wants to use a module with other pre-trained models (e.g. the Named Entity tagger trained on financial reports or on encyclopedic text), alternative instances of the same module can also be created within the xtsv framework. xtsv dynamically creates and runs the desired chain as described above. Although the described interfaces (CLI, REST API, Python library) have been implemented in Python to meet the user requirements, the modules can be implemented in other programming languages based on the specification, even in a heterogeneous fashion like in the case of UNIX pipelines.

The description of emtsv so far can be summarized as follows: a loosely coupled architecture, the possibility of adding new modules written in any programming language, the standard tsv format, the three API types (CLI, Python package, REST API, the latter optionally running in the user's cloud), scalability, the openly available source code, and a pipeline adhering to the UNIX philosophy. These enable users being on different levels of programming skills and coming from different backgrounds to combine rule-based and statistical systems, to manually correct the output of any modules then to feed it to any of the next modules, to compare the output of alternative modules as a part of the same pipeline for the same input, to interpret errors, and to retrain the models if needed. This wide spectrum of features exceeds the capabilities of the previously presented tool chains applicable for Hungarian (see Section 2).

The following section describes the role of each of the available modules in the chain, as well as the minimum requirements for new modules, which enable the chain to be expanded with new modules or the modification of existing ones within the framework.

5 Modules

Module management means that the fields required by the given module need to be available by the time of running the module. This can be controlled by the header available already at the time of assembling the pipeline, indicating an error early, even in the case of a dynamically defined pipeline. Recall that each module specifies the needed and produced columns. For example, it is known at the time of chain assembly, on the basis of the specified fields, that the POS tagger needs the form and anas (i.e. analyses) columns, or that dependency parsing must be preceded by POS tagging but not by NP chunking, as shown in Figure 1.

The organization of modules is based on the

[12] https://hub.docker.com/r/mtaril/emtsv

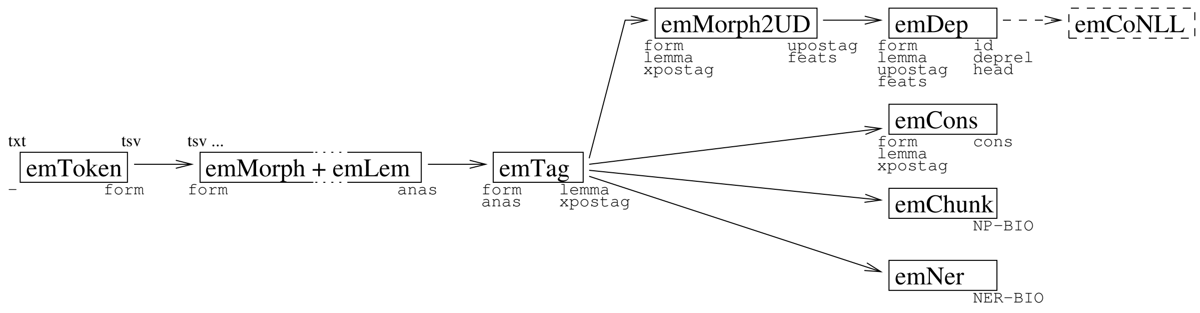

Figure 1: The current processing chain of emtsv, with input and output fields.

previous version of e-magyar, however we split logically independent functions into separate modules, even if they were built into one module previously. Thus, the tasks of each module can be more clearly specified, which makes their testing and development simpler. For a unified handling of modules in xtsv (see Section 4), modules originally written in Java were wrapped into Python modules. The names of these wrappers have been given the uniform ending py. In Python wrappers, Java is uniformly called by the Pyjnius package[13]. The Python wrapper communicates with the original Java module through Java-native types, which cuts down the original input and output handling code, so eliminating the differences between the original input and output formats in favor of emtsv. The additional changes made to the individual modules are described in the subsections below. Module names are prefixed with em: e for electronic and m for *magyar* 'Hungarian'.

5.1 emToken

Although tokenization rules themselves remain unchanged, we revised the tokenizer for the new pipeline significantly. emToken (Mittelholcz, 2017), the tokenizer in e-magyar, consists of several submodules with different functionalities such as checking illegal characters, sentence segmentation, abbreviation processing, and word tokenization itself. So far, these submodules were compiled into a single monolithic binary file. In the new version, each submodule is compiled into a binary file that can be run separately, reading from standard input and writing to standard output. These submodules are linked together by a Python script. For the new structure, the test system for the emToken was also redesigned. These refactoring steps enabled us an organic integration within the emtsv framework.

Detokenization is currently not supported, but emToken returns the spaces besides the tokens properly, so future work can modify the module to insert original (possibly spacial) spaces in a separate column to make detokenization possible. An alternative would be to record the word offset (the index of the starting character) in a column, from which it can be seen whether consecutive tokens were originally separated by a space or not.

5.2 emMorph and emLem

We fixed some bugs that affect the morphological analyzer emMorph (Novák et al., 2016) and its interaction with the lemmatizer emLem. The output of the morphological analysis, i.e. the string representing individual steps of the underlying transducer, is regarded as an internal format, as it is not used directly but is transformed into a more readable form of the morphemic sequence. The post-processing is executed by the emLem module. The original Java implementation of emLem has been replaced by a new Python code[14] to improve simplicity and code transparency. Bugs uncovered during rewriting have been fixed.

The module containing emMorph+emLem has been supplemented with a special REST API, which allows the user to easily access the analyses of individual word forms through the browser, by pasting each word form into a special URL. This demo interface[15] runs in the cloud, where quick access to emMorph is provided.

The output of the extended emMorph module is a specially formatted JSON file with fields for both human and machine use (see Figure 2). Each

[13]https://github.com/kivy/pyjnius

[14]https://github.com/ppke-nlpg/emmorphpy

[15]https://emmorph.herokuapp.com/

```
{
  "bokrot": [
    {
lemma     "bokor"
morphana "bokor[/N]=bokr+ot[Acc]=ot"
readable "bokor[/N]=bokr + ot[Acc]"
tag       "[/N][Acc]"
twolevel "b:b o:o k:k :o r:r :[/N] o:o t:t :[Acc]"
    },
    ...
  ]
}
```

Figure 2: An example of the JSON output of the morphological analyzer and the lemmatizer. The example *bokrot* is the accusative form of the epenthetic stem *bok(o)r* 'bush'.

analysis contains four fields: the `lemma`; the morphemic sequence in two formats: one intended for machine use (`morphana`) and one for human reading (`readable`); the bare tag of the strict morphosyntactic category without phonological-orthographic content (`tag`); and the two-level output of the morphological analyzer (`twolevel`) for debugging purposes. The `readable` field omits redundant surface forms, i.e. those that coincide with the deep form. The REST API is capable to return multiple words at once, when called using the HTTP POST method. The advantage of the standard JSON format is that it protects against errors caused by unexpected characters in large corpora. For the sake of fitting into `tsv`, the use of a tab outside the string is prohibited in the generated JSON.

5.3 emTag

The `emTag` POS tagger is based on PurePOS (Orosz and Novák, 2013). It requires an inconvenient, non-standard input format[16], that is exposed to errors caused by unexpected characters. The new format described in Section 3 makes possible to eliminate errors caused by unexpected characters in large corpora.

Now alternative morphological analyzes can be separately provided for the Java-based PurePOS as native Java data structures with the input text (even from within a Java program). The PurePOS–Python interface contains the add-ons required for `emtsv`. PurePOS can be used in three ways with the Python interface: alone with pre-analyzed input, with its built-in statistical morphological analyzer, or using the `emMorph`+`emLem` rule-based morphological analyzer.

[16]`https://github.com/ppke-nlpg/purepos`

5.4 emChunk and emNER

The configuration of the `HunTag3` (Endrédy and Indig, 2015) sequential tagger, which served as the basis of the `xtsv` framework, has been slightly modified to meet the requirements of the new `emtsv` format: features are now reached by column names not by column numbers. In addition, `HunTag3` has undergone a number of internal transformations, resulting in the standardized management of the input and output formats, completely separate from the rest of the computation.

5.5 emMorph2UD

The original converter (`DepTool`), that converted the output of `emTag` to linearized attribute–value pairs for the `emDep` dependency parser (see Section 5.6) is replaced by `emMorph2UD`, a new converter. There are two main reasons for this improvement. Firstly, looking more closely at `DepTool`, it turned out that it did not handle certain morphological features: the content of the input morphological tags were often lost. Secondly, the tags generated by `DepTool` had a specific format that could be used only within the toolchain between the two modules.

As UD is a cross-linguistically consistent grammatical annotation scheme, it is reasonable to provide the output in that formalism beside the tags of `emMorph`. Therefore, `emMorph2UD` converts the morphological tags emitted by `emTag` to UD[17]. Formerly in `e-magyar`, the model behind `emDep` was trained on POS tags and morphosyntactic features converted by `DepTool`. Consequently, the model had to be replaced with one trained on Szeged Treebank with UD tags (Vincze et al., 2017).

The `emMorph2UD` module can be used both for inter-modular communication in `emtsv` between `emTag` and `emDep` using the formalism of UD, and as an output format with UD morphological tags. For a detailed description and precise evaluation of `emMorph2UD`[18], see Vadász and Simon (2019).

5.6 emDep and emCons

We also detached the Bohnet dependency parser (Bohnet and Nivre, 2012) and the Berkeley con-

[17]Only UD version 1 has been elaborated for Hungarian, therefore here we mean UDv1 under UD.

[18]For an exhaustive description of annotation schemes for Hungarian morphology with converters, see `https://github.com/dlt-rilmta/panmorph`.

stituent parser (Durrett and Klein, 2015) from Magyarlánc 3.0, so the parsers now work with a smaller resource footprint. The model of emDep has been replaced (see Section 5.5): its input is now the set of POS tags and morphological attribute–value pairs, converted from the output of emMorph to conform the UD annotation scheme. The output of emDep, i.e. the syntactical annotation, did not change.

5.7 emCoNLL

To satify the need of a standard well-proven format, the output can be converted to the CoNLL-U format with the help of the module emCoNLL. By this, the output of emtsv is suitable for tools dealing with CoNLL-U format, such as processing, annotation or visualizaton tools[19].

Since the fields UPOS, HEAD and DEPREL are not allowed to be left unspecified in the CoNLL-U format, emCoNLL depends on the dependency parser, thus only the output of emDep can be used as an input of emCoNLL. In addition, CoNLL-U supports only one extra field (MISC) for a further annotation layer, however, there might be several competing modules for that one field (emMorph, emLem, emChunk and emNer). This problem is solved by leaving this extra column empty, thus only mandatory fields are filled during the conversion. This module serves as a good example for splicing a simple and useful additional module to the end of the toolchain.

6 Summary

In this article, we introduced emtsv, the new version of the e-magyar language processing pipeline that has undergone a major transformation. emtsv is not only competing, but at several points exceeds its competitors. Its main characteristics are the uniform communication format, the easy interoperability of the modules thanks to this format, the free source code, the loosely coupled modules (open for new modules, be they rule-based or statistic), and the scalability. It can run as service through a REST API, as a pipeline in CLI, or can be integrated into larger systems as a Python library API and available as Docker image as well. Developers can plug in their own modules. Modules can be individually upgraded, compared, rewritten, retrained, or customized. Consequently, emtsv is now the Hungarian NLP pipeline with the broadest functionality.

7 Future Work

Bootstrapping a human-annotated corpus Starting with a large free corpus, we plan to pre-process raw text with emtsv, and improve the output module-by-module by semi-manually correcting the output of the nth module and then passing the improved version to module $n + 1$ (this could not be done in the former version of e-magyar). Free availability of the corpus used for this process is important in order to that the research community can experiment with new methods by changing tools and data. The process will provide a good opportunity to test the system in detail, to detect errors, and to turn to computational linguistic research proper, i.e. to justify linguistic theories.

Over-tokenizers In the time of pre-trained deep language models (aka contextualized word representations, such as Peters et al. (2018)), a system with symbolic inter-modular communication may seem anachronous, but we believe emtsv as a pre-processing tool can help state-of-the-art systems, especially in handling less frequent or out-of-vocabulary words. Our approach belongs to subword-level modelling (Botha and Blunsom, 2014), specifically the simple but effective engineering solution of splitting rare or unknown words to their components (going against our own xtsv recommendation of no token splitting introduced in Section 3). Though unsupervised statistical segmentation (e.g. Morfessor (Creutz and Lagus, 2005) or byte-pair encoding (Sennrich et al., 2016)) is widely used, segmentation to meaningful parts (Lazaridou et al., 2013; Avraham and Goldberg, 2017) offers the exploitation of additional linguistic knowledge. Splitting off inflectional suffixes have already hugely reduced word perplexity (Nemeskey, 2017). We plan to extend this line of work to compositional compounds, especially noun+noun compounds like *szín-tan* ‘color-theory’, compositional derivational suffixes e.g. *szeker-estül*, lit. chariot-along.with.one’s ‘along with one’s chariot’, and compositional preverbal prefixes e.g. *agyon-tápol*, ‘over-nurture’. The planned modules can work by assigning probability scores to composition candidates based on gold constructions with similar constituents, where similarity is measured in the

[19]https://universaldependencies.org/tools.html

word embedding space.

Universal guesser We store all possible morphological analyzes to be able to fine-tune them before disambiguating, but these analyzes apply only for the tokens recognized by the fixed lexicon of the rule-based morphological analyzer (`emMorph`). These analyzes also lack weights, which is desired by the latter procesing steps. In order to treat each module equally, we plan to create a *Universal guesser module* (harmonised with `emMorph`) that is able to analyze OOV tokens – with rules or statistical machine-learning – as well and set the appropriate weights for each analyzis (e.g. by using the same training material used by the POS-tagger module (`emTag`) currently). Stripping out this task from the POS-tagger – where it currently resides – creates the possibility to fine-tune analyzes for all tokens prior to POS-tagging if needed. Also it enables us to substitute the guesser module – or the POS-tagger – with others (e.g. Morfessor and Lemmy[20]) and find the one with the best performance by testing it in real-life conditions.

Phrases and verb constructions Our plans include creating new modules for `emNer` trained on texts in different domains, as well as new models for chunking (i.e. annotating all types of phrases in the sentence), and even enhancing `emDep` based on lessons learned from Mazsola (Sass, 2008).

Load-balancing Currently, every module runs in one instance. A rather technical follow-up development would be to run bottleneck modules in multiple copies: paralleling increases performance. For example, if the disambiguator processes 10 sentences while the syntactic parser finishes with 2 sentences, then it is worth starting the syntactic parser in 5 instances and process sentences in parallel. This technology is called load-balancing and it is popular both within the Python and the Docker world.

A multilingual chain The new `xtsv` framework is actually completely language and module independent. We may create a multilingual analyzer whose pipeline can start with a language identifier. In order to do so, we need modules for other natural languages. It is important for these toolsets not to be monolithic like `Magyarlánc`, but separated into modules – by their logical role in the pipeline – that can be given to a tsv-wrapper and combined freely in `xtsv`.

[20] `https://github.com/sorenlind/lemmy`

Acknowledgements

Work supported by the MARCELL project of EU CEF. Márton Makrai was partially supported by project fund 2018-1.2.1-NKP-00008: Exploring the Mathematical Foundations of Artificial Intelligence and National Research, Development and Innovation Office (NKFIH) grant #120145 Recognizing Word Structure with Deep Learning.

References

Oded Avraham and Yoav Goldberg. 2017. The interplay of semantics and morphology in word embeddings. In *Proceedings of the 15th Conference of the European Chapter of the Association for Computational Linguistics: Volume 2, Short Papers*, pages 422–426, Valencia, Spain. Association for Computational Linguistics.

Bernd Bohnet and Joakim Nivre. 2012. A Transition-based System for Joint Part-of-speech Tagging and Labeled Non-projective Dependency Parsing. In *Proceedings of the 2012 Joint Conference on Empirical Methods in Natural Language Processing and Computational Natural Language Learning*, EMNLP-CoNLL '12, pages 1455–1465, Stroudsburg, PA, USA. Association for Computational Linguistics.

Jan A Botha and Phil Blunsom. 2014. Compositional morphology for word representations and language modelling. In *Proceedings of the31st International Conference on Machine Learning*, pages 1899–1907, Beijing, China.

Sabine Buchholz and Erwin Marsi. 2006. CoNLL-X Shared Task on Multilingual Dependency Parsing. In *Proceedings of the Tenth Conference on Computational Natural Language Learning (CoNLL-X)*, pages 149–164, New York City. Association for Computational Linguistics.

Mathias Creutz and Krista Lagus. 2005. Unsupervised morpheme segmentation and morphology induction from text corpora using Morfessor 1.0. Technical Report A81, Helsinki University of Technology.

Hamish Cunningham, Diana Maynard, Kalina Bontcheva, Valentin Tablan, Niraj Aswani, Ian Roberts, Genevieve Gorrell, Adam Funk, Angus Roberts, Danica Damljanovic, Thomas Heitz, Mark A. Greenwood, Horacio Saggion, Johann Petrak, Yaoyong Li, and Wim Peters. 2011. *Text Processing with GATE (Version 6)*. GATE (April 15, 2011).

Greg Durrett and Dan Klein. 2015. Neural CRF Parsing. In *Proceedings of the 53rd Annual Meeting* *of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, pages 302–312. Association for Computational Linguistics.

István Endrédy and Balázs Indig. 2015. HunTag3: a General-purpose, Modular Sequential Tagger – Chunking Phrases in English and Maximal NPs and NER for Hungarian. In *7th Language & Technology Conference, Human Language Technologies as a Challenge for Computer Science and Linguistics (LTC '15)*, pages 213–218, Poznań, Poland. Poznań: Uniwersytet im. Adama Mickiewicza w Poznaniu.

Péter Halácsy, András Kornai, László Németh, András Rung, István Szakadát, and Viktor Trón. 2004. Creating open language resources for Hungarian. In *Proceedings of the Fourth International Conference on Language Resources and Evaluation (LREC 2004)*, pages 203–210. ELRA.

Péter Halácsy, András Kornai, Csaba Oravecz, Viktor Trón, and Dániel Varga. 2006. Using a morphological analyzer in high precision POS tagging of Hungarian. In *Proceedings of the Fifth International Conference on Language Resources and Evaluation (LREC'06)*, Genoa, Italy. European Language Resources Association (ELRA).

Angeliki Lazaridou, Marco Marelli, Roberto Zamparelli, and Marco Baroni. 2013. Compositionally derived representations of morphologically complex words in distributional semantics. In *Proceedings of the 51st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 1517–1526, Sofia, Bulgaria. Association for Computational Linguistics.

Iván Mittelholcz. 2017. `emToken`: Unicode-képes tokenizáló magyar nyelvre [Unicode-able tokenizer for Hungarian]. In *XIII. Magyar Számítógépes Nyelvészeti Konferencia [13th Conference on Hungarian Computational Linguistics]*, pages 61–69, Szeged.

Dávid Márk Nemeskey. 2017. `emLam` – a Hungarian Language Modeling baseline. In *XIII. Magyar Számítógépes Nyelvészeti Konferencia [13th Conference on Hungarian Computational Linguistics]*, pages 91–102, Szeged.

Attila Novák, Borbála Siklósi, and Csaba Oravecz. 2016. A New Integrated Open-source Morphological Analyzer for Hungarian. In *Proceedings of the Tenth International Conference on Language Resources and Evaluation (LREC 2016)*, Paris, France. European Language Resources Association (ELRA).

Csaba Oravecz, Tamás Váradi, and Bálint Sass. 2014. The Hungarian Gigaword Corpus. In *Proceedings of the Ninth International Conference on Language Resources and Evaluation (LREC-2014)*. European Language Resources Association (ELRA).

György Orosz and Attila Novák. 2013. PurePos 2.0: a Hybrid Tool for Morphological Disambiguation. In *Proceedings of the International Conference Recent Advances in Natural Language Processing RANLP 2013*, pages 539–545, Hissar, Bulgaria. INCOMA Ltd. Shoumen, BULGARIA.

Matthew Peters, Mark Neumann, Mohit Iyyer, Matt Gardner, Christopher Clark, Kenton Lee, and Luke Zettlemoyer. 2018. Deep contextualized word representations. In *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long Papers)*, pages 2227–2237. Association for Computational Linguistics.

Gábor Recski and Dániel Varga. 2009. A Hungarian NP Chunker. *The Odd Yearbook. ELTE SEAS Undergraduate Papers in Linguistics*, pages 87–93.

Bálint Sass. 2008. The verb argument browser. In *Text, Speech and Dialogue*, pages 187–192, Berlin, Heidelberg. Springer Berlin Heidelberg.

Bálint Sass, Márton Miháltz, and Péter Kundráth. 2017. Az `e-magyar` rendszer GATE környezetbe integrált magyar szövegfeldolgozó eszközlánca [The e-magyar Hungarian text processing system embedded into the GATE framework]. In *XIII. Magyar Számítógépes Nyelvészeti Konferencia [13th Conference on Hungarian Computational Linguistics]*, pages 79–90.

Rico Sennrich, Barry Haddow, and Alexandra Birch. 2016. Neural machine translation of rare words with subword units. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 1715–1725, Berlin, Germany. Association for Computational Linguistics.

Milan Straka and Jana Straková. 2017. Tokenizing, POS Tagging, Lemmatizing and Parsing UD 2.0 with UDPipe. In *Proceedings of the CoNLL 2017 Shared Task: Multilingual Parsing from Raw Text to Universal Dependencies*, pages 88–99, Vancouver, Canada. Association for Computational Linguistics.

Viktor Trón, György Gyepesi, Péter Halácsky, András Kornai, László Németh, and Dániel Varga. 2005. Hunmorph: Open source word analysis. In *Proceedings of the ACL Workshop on Software*, pages 77–85. Association for Computational Linguistics, Ann Arbor, Michigan.

Noémi Vadász and Eszter Simon. 2019. Konverterek magyar morfológiai címkekészletek között. [Converters between Hungarian Morphological Tagsets]. In *XV. Magyar Számítógépes Nyelvészeti Konferencia [15th Conference on Hungarian Computational Linguistics]*, pages 99–111, Szeged.

Dániel Varga and Eszter Simon. 2007. Hungarian named entity recognition with a maximum entropy approach. *Acta Cybernetica*, 18:293–301.

Veronika Vincze, Katalin Simkó, Zsolt Szántó, and Richárd Farkas. 2017. Universal dependencies and morphology for Hungarian - and on the price of universality. In *Proceedings of the 15th Conference of the European Chapter of the Association for Computational Linguistics: Volume 1, Long Papers*, pages 356–365, Valencia, Spain. Association for Computational Linguistics.

Tamás Váradi, Eszter Simon, Bálint Sass, Iván Mittelholcz, Attila Novák, Balázs Indig, Richárd Farkas, and Veronika Vincze. 2018. E-magyar – A Digital Language Processing System. In *Proceedings of the Eleventh International Conference on Language Resources and Evaluation (LREC 2018)*, Miyazaki, Japan. European Language Resources Association (ELRA).

János Zsibrita, Richárd Farkas, and Veronika Vincze. 2013. A Toolkit for Morphological and Dependency Parsing of Hungarian. In *International Conference on Recent Advances in Natural Language Processing*, pages 763–771, Shoumen, Bulgária. INCOMA Ltd.