Proceedings of the 12th Conference on Language Resources and Evaluation (LREC 2020), pages 7044–7052
Marseille, 11–16 May 2020

The xtsv Framework and the Twelve Virtues of Pipelines

Balázs Indig, Bálint Sass, Iván Mittelholcz
MTA Research Institute for Linguistics
Benczúr u. 33., H-1068, Budapest, Hungary
{LASTNAME.FIRSTNAME}@nytud.mta.hu

Abstract

We present xtsv, an abstract framework for building NLP pipelines. It covers several kinds of functionalities which can be implemented at an abstract level. We survey these features and argue that all are desired in a modern pipeline. The framework has a simple yet powerful internal communication format which is essentially tsv (tab separated values) with header plus some additional features. We put emphasis on the capabilities of the presented framework, for example its ability to allow new modules to be easily integrated or replaced, or the variety of its usage options. When a module is put into xtsv, all functionalities of the system are immediately available for that module, and the module can be be a part of an xtsv pipeline. The design also allows convenient investigation and manual correction of the data flow from one module to another. We demonstrate the power of our framework with a successful application: a concrete NLP pipeline for Hungarian called e-magyar text processing system (emtsv) which integrates Hungarian NLP tools in xtsv. All the advantages of the pipeline come from the inherent properties of the xtsv framework.

Keywords: xtsv, general framework, NLP pipeline, NLP tools, tsv

1. Introduction

Concerning NLP processing tools, there are big players and small ones as well. Numerous small independent tools exist, their common value is that many of them implement brilliant ideas in an elegant manner, however they may not be acknowledged properly. The main reason for this is one of the followings: they handle some corner cases in their own way, they are language-dependent, they are implemented in a less-known programming language or they do not fit in a larger framework. When one starts to develop a new tool, the creative part of the work is to devise and realise one's solution to the well-defined problem in order to showcase the idea by comparing it to the state-of-the-art alternative. However, in order to help this idea to spread widely, one must augment the implementation with a standard I/O interface to let others test it. In some cases, a demo service is also needed to draw attention to the solution, else the program – no matter how good it is – will become obsolete and finally not used by anybody. What is worse, it will probably be reinvented again, possibly in the same way.

Usually, the developer concentrates on the inner-workings that results in neglecting "the interface boilerplate'. Nowadays, the time left for the ideas to reach the market is shorter than ever before. The responsibility of the used framework is to reduce the number of steps needed for shipping a new idea. This is one of the obvious reasons why only big companies/universities/communities can keep up the pace in inventing new ideas that spread easily, while those who do not recognise this are doomed to just follow them. In this paper, we present a framework that helps to unleash the hidden potential of creative ideas by doing all the hard work other than creating the main logic of an NLP module. Hopefully, the framework will prove to be helpful not just for us but for others as well. We believe it will aid the good ideas to find their way more easily to the users.

2. Background

After the deep learning revolution a few product (like *Keras/TensorFlow*, *PyTorch* or *fastText*) tend to rule the market (Sejnowski, 2018). They are backed with stable funding, therefore the user interface is readily customised and maintained while their back end is developed independently at a great pace. When a small group of people stand out with their new solution – that does not fit into the former ecosystems –, they often must choose between developing the back end or the front end. If they choose the back end to be competitive, they will be doomed to small audiences/communities. If they opt for front ends, they can get lost in the jungle of the developing standards and interfaces. Pursuing greater user base by improving the user experience at the level of front end will dwarf the significance of the back end.

As the major "language-independent" tools are targeted to big and resourceful languages and industrial usage scenarios, they may not fit for the numerous less- and mid-resourced languages and the (linguistic) research-centric scenarios. This gap is currently filled with many – sometimes old-fashioned – independent but viable solutions[1] until today. However, such solutions are often developed only to showcase the core idea lying behind them for research purposes not necessary with the goal of widespread usage. This fact makes it difficult for a researcher without the proper knowledge (and time) to restructure them so that they can be examined, compared to other tools and used properly. Therefore, their viability can be rather limited in comparison with the tools and pipelines developed by big companies, universities or communities.

This is a recurring problem which does not only affect the application and further development of such programs and ideas, but also the continuation of research. The citation count can be lower compared to tools that can be integrated to greater frameworks. To sum up, innovation and good

[1]This gap is even more felt in the absence of direct comparisons between the old tools and the new popular ones.

ideas are frequently lost and reinvented because of the lack of interoperability, while the problems these tools solve or handle better than the widespread alternatives are bound to recur. As we will see, our solution targets these tools.

2.1. Standards

In the era of the expansive adoption of the CoNLL-U format and the UD standard, many of the newly developed tools tend to respect these *de facto* standards which mark the way for new tools. Still, we are left with one rather important question: what happens to the old tools that are good but lack the support of modern interfaces and standards? Their users do not possess the knowledge and resources to reimplement or substantially modify them to follow the new standards. Sometimes these tools address tasks which do not fit into the aforementioned standards, i.e. they are not handled by the majority of the other compliant tools. Therefore, these tools have drawbacks, even though they could present a viable solution.

When it comes to support smaller languages trying to adopt standards developed for large languages, it is impossible to do so without losing linguistic information (Vincze et al., 2017). However, there are multiple possibilities for workaround: sometimes it is better to do a fine-grained tagging to get higher accuracy and then merge classes to get the standard compliant representation (for our approach see Section 7.3.). The other way is to hybridise methods with rule-based components to overcome the too small size of the training material. These strategies can hardly fit into the conventionally used CoNLL-U format and therefore into the compliant tools. One can not make a change in these programs supporting the CoNLL-U format as the developer and linguistic resources for these languages are rather limited.

Consequently, we are limited in probing and comparing different alternative tools to select the best one for our needs. Individual solutions will come to a similar fate: they are not usable for greater audiences and therefore may not be acknowledged properly. In greater time frames, the pipeline will become obsolete. New, better alternatives will be discovered, but they can not be replaced due to tight integration.

2.2. Aim

The aforementioned problems are out of sight for the current standards. In order to break the circle, we gathered the requirements that a pipeline has to meet and developed a framework that can satisfy these needs.

In this paper, we present our findings, the format and the framework we have created to solve the aforementioned problems. We describe our implementation of a truly modular pipeline which does not limit users in experimenting with interchangeable modules, swapping them to the best-performing alternatives and making new compatible ones easily[2]. We also present its first successful application by a language processing pipeline for Hungarian. Finally, we compare this pipeline to the existing alternatives.

[2]Allowing a simple manner of creation of modules is very important as it attracts users to create modules in the ecosystem, e.g. by modifying old ones.

3. The Twelve Virtues of Pipelines

We gathered numerous requirements during our practice. Most of them were an actual limiting factor or a future expectation for an ideal pipeline. In the following sections, we present these and argue for their importance. Our point is that instead of having to deal with these aspects for every module separately, they can and should have a general solution at the level of the pipeline providing their functionality automatically for every module included.

3.1. A Pipeline Should Act as a Real Pipeline

First of all, we need to give a clear definition of pipelines, because there is a tendency nowadays that monolith tools handling multiple tasks are also called pipelines in the public speech. The definition of the pipeline is the following: "In software engineering, a pipeline consists of a *chain of processing elements* (processes, threads, coroutines, functions, etc.), arranged so that *the output of each element is the input of the next*; the name is by analogy to a physical pipeline."[3] We must emphasize the differences between monolith tools and pipelines. In the rest of the paper, we follow the above definition for *pipelines*, while "pipelines" implemented as monoliths will be referred to as *tools*.

3.2. A Pipeline Should Truly Work for Any Language

There are many pipelines which are claimed to be language-independent, although they are developed and tested only for a few major languages. These languages are rather different from the numerous smaller languages which these pipelines are also expected to work with. As these pipelines have too many prior assumptions and expectations that could not be met sometimes (e.g. a suitable-sized UD treebank may not exist), they can not be used for these smaller languages or only with a big performance drop. In our view, the pipeline framework should be as abstract as possible. We should not assume anything that has nothing to do with the operation of the framework itself, in other words, that is not at the abstract level. All kinds of linguistic assumptions should be made at module level.

3.3. A Pipeline Should Handle Big Data and Large Number of Requests as Well

One major limiting factor of the existing tools and pipelines is the inability of handling both big and small data. This concerns the handling of training material in those cases when it can be generated automatically in large quantities (e.g. diacritic restoration). One does not want to use different pipelines for numerous small requests (e.g. a web service) and for one large request (e.g. big data queries), one pipeline should be able to handle both cases well.

3.4. A Pipeline Should Allow Reusing Existing Tools

A prominent limiting factor of the existing pipelines is that they do not make it possible to use modules written in various different programming languages together. One may

[3]`https://en.wikipedia.org/wiki/Pipeline_(software)`

have to fully reimplement (or workaround) an existing well-tested implementation of the tool to be included and maintain the modification if the module develops. The problem gets more serious when it comes to tightly coupled pipelines where it is even harder to replace and maintain modules without much effort. A pipeline framework should offer general, easy-to-use, prefabricated wrapper solutions for future modules regardless of their programming language.

3.5. A Pipeline Should Allow Entry and Exit at Any Point

When creating new training material or testing the robustness of the pipeline, it is inevitable to be able to examine the output of each module separately and modify it when required – to reduce error amplification – before letting the pipeline proceed. Beside that, it is not enough to test a module – or any part of the pipeline consisting of one or more modules – in itself, one may also test how robust the module is in the pipeline, i.e. whether the modules can work together or they need to be harmonised (e.g. the rules and the training data). The former method is called *in vitro* and the latter *in vivo* in biology.

3.6. Modules Should be Replaceable in the Pipeline

To keep the pipeline at the state-of-the-art performance, maintainers may need to replace modules in a timely manner. Before this kind of change, one usually wants to perform measurements to verify that the new module is superior to the old one which is currently in the pipeline. Beside that, the pipeline should allow to have multiple alternative modules to be used when a confident decision could not be made e.g. preferring speed or performance. In these cases, the decision should be left for the user.

3.7. A Pipeline Should Exist as Docker Container

Nowadays it is a requirement for most complex program to support building as a Docker image and insertion into an environment as a container. This is especially true for distributed and service workloads. Be a single tool or a full-featured pipeline, the framework should do all the heavy lifting as it is possible to implement this functionality at the level of the framework. The module developer should only be concerned about the issues of creating or running the Docker image.

3.8. A Pipeline Should Support Usage as Service (REST API)

We are living in the age of microservices or in other words *software as a service (SaaS)*. When a dedicated service is to be deployed in the cloud, it is an advantage if it is equipped with a standard, instantly usable REST API. This is also convenient for the non-technical user who "just wants to use" the pipeline with minimal effort. With proper configuration, one can even use it for instant demonstration. This is again a feature which boosts the viability of the pipeline, and should not be written independently for every tool, but can be addressed well at the level of the pipeline by a general solution.

3.9. A Pipeline Should Allow Modification and Extension with New Modules

By allowing modification and extension, not only existing tools can be reused but new ones can be developed as well. The developer only needs to solve the actual task and simply put the implementation into the pipeline. Modules can be split into logically distinct parts. Smaller modules mean speed, generality and more room for experiments, like experimenting with new features or specialising the final output.

3.10. A Pipeline Should be Standard Compliant and Comparable

Not just the individual tools but the pipeline as a whole should be able to take input and yield output that is comparable. This does not necessarily mean that the intermodular data format should be constrained by any standard as it is the "internal matter" of the pipeline itself. This allows freedom, but keeps compatibility and comparability using a simple converter – to transform the data to a standard compliant format – as the last element of the pipeline. This property makes it possible to gain more insight into the inner-workings of the data and the tools – e.g. by subdividing the task as needed – because only the final result is evaluated.

3.11. A Pipeline Should Support Distributed Workloads

When dealing with multiple parallel users and big data, it is an obvious requirement to be able to distribute the load to multiple machines. This is not self-evident with old tools where there is a need of special knowledge that the module developer may not have. Fortunately, it is essentially the same for all modules so it can and should be solved at pipeline level.

3.12. The API and the Communication Format Should Be Built to Last

When one builds a new, general framework that is expected to be trusted and used by others, it is essential to design the API as simple and extensible as possible to accommodate all unexpected future use cases without modification and API breakage, else the future users would not trust and adopt the framework and the API. One well-known example of such an API is the Unix pipeline which has started 50 years ago[4] and still spreading in the world, used by the general public with only slight modifications throughout the years.

4. Related work

The three main pipelines used today are *SpaCy*[5] (Honnibal and Montani, 2017), *StanfordNLP*[6] (Qi et al., 2018), and *UDPipe*[7] (Straka and Straková, 2017). They are all designed around the CoNLL-U and Universal Dependencies

[4]https://www.bell-labs.com/unix50/
[5]https://spacy.io/
[6]https://stanfordnlp.github.io/stanfordnlp/
[7]http://ufal.mff.cuni.cz/udpipe

(UD) ecosystem[8] and they solve the minimal required steps to comply. SpaCy has a rather simple API which is very popular in the scene. Recently, there are wrappers for the two other tools [9] to enable the usage with the same API.

These programs are language-agnostic (but somewhat English-centered), downloadable and each of them has a Python API. Only some of them allow splitting processing into steps to allow corrections on the data. The divergence from the strict series of modules is not possible and some of them do not support creating new modules. This behaviour satisfies the majority of the community, as most of the use cases require these tools for standard preprocessing of text prior to the actual work to be done. In these cases, speed is the first priority.

To sum up, users mostly want to reach the output from the input in the shortest and fastest way – that is, sometimes in one big step. From this point of view, the majority of our requirements (see Section 3.) could appear as unnecessary constraints which just slow down the processing. We think that thanks to our additional features `xtsv` surpasses other approaches in cases when speed is not the only consideration. To characterise the difference in speed, we present our benchmarks in Section 8.

Our framework does more than just providing a trainable, downloadable pipeline with pretrained models: we designed it to enable usage as a service (see 3.8.). Therefore we examined the available solutions that work in the cloud as a service in asynchronous manner. Their source codes are mostly not available, but some of them integrate some open source tools to broaden their repertoire. *WebSty*[10] and *Weblicht*[11] are two good examples of this type of tools. There are three problems with this kind of architecture: a) the user depends on the administrators of the system (e.g. integrating new modules); b) there is no feedback for module developers; c) the user must give (possibly sensitive) data to a third party. We tried to eliminate these drawbacks when designing `xtsv`.

The third group of tools to mention is the language-specific tools which are often tightly coupled. Even though they are open source, it is almost impossible for an outsider to improve their speed or accuracy. For example Hungarian *Magyarlánc* (Zsibrita et al., 2013) is such a tool to name. There is a great need for creating and comparing prototypes, but in a fast moving workflow it is not feasible to rewrite the best candidate for the final product.

`xtsv` competes with these types of tools but it puts the emphasis on other aspects: modularity, stable API and fast development over speed. Thanks to this fact it can be an ideal test bed for NLP and also for linguistic experiments that could evolve and be even integrated into the aforementioned pipelines.

Regarding the possible data format, we found that CoNLL-* format family (Stranák and Štepánek, 2010) has been shaped by many experience and has matured a lot over the years. Still, we found it a no-go because it presumes a global knowledge of the possible fields that is hard-wired into each module by the order of the fields. This constraint is somewhat compensable with extra fields used beyond the eleven mandatory fields. Beyond that, although there is a CoNLL-U validator program[12], major programs that claim to support the CoNLL-U format do not produce "valid" output. They produce mostly minor errors, but this shows that the format is somewhat overregulated and some of the tools or treebanks do not bother to adapt to these standards.

There are already a few extensions of the CoNLL-U format. One of them is the CoNLL-U plus format[13]. It passes our requirements on the format (see 3.12.), but we have not found a working implementation for this format. It also tries to keep compatibility and therefore follows the regulated format inherited from CoNLL-U which is not necessary and not general enough in our opinion.

In the following section, we describe the format chosen for our framework. It builds upon the good features found in the CoNLL-U format, but has also differences to leverage the latter one's overregulated nature.

5. The Communication Format of `xtsv`

The inter-module communication format of `xtsv` is essentially `tsv` with headers. This general format follows the principles of *vertical format* where each token is written vertically, one item per line, and the features of the tokens are separated by a tabulator character as columns in the same line. It is important that this simple format is augmented with a header in the first line to label the columns. As the header identifies the columns, they can be in arbitrary order. In fact, the header is what controls the operation of the whole pipeline: it defines input and output columns. Additionally, we support fixed column order without header as well, and modules can choose between the two formats. The pipeline delegates the decision to choose between these (and the possible introduction of further constraints on the content) to the two consecutive modules.

To be compliant to the CoNLL format family, we use empty lines as sentence separators, and we added the possibility of having CoNLL-style comments – starting with a hashmark and a space – directly before sentences. These modifications somewhat break the principles of `tsv` but we decided to use them as CoNLL formats are de facto standards.

We wanted the format to be neutral to all fields to priorise generality and extensibility instead of priorising the (nowadays) frequent use cases. With this design decision we opened the way to divide or extend the classical steps of the pipeline on demand to adhere the underlying logic. The order of the new columns is generally not set, but each module is advised to augment new columns to the right of the existing columns.

[8]`https://universaldependencies.org/`

[9]`https://github.com/explosion/spacy-stanfordnlp` and `https://github.com/TakeLab/spacy-udpipe`

[10]`http://ws.clarin-pl.eu/websty.shtml`

[11]`https://weblicht.sfs.uni-tuebingen.de`

[12]`https://github.com/UniversalDependencies/tools/blob/master/validate.py`

[13]Available at `https://universaldependencies.org/ext-format.html`, but without the proper bibliographical references we can only suspect the creation date (2018-01-17) and status (draft).

With this layout we opened up numerous possibilities which previously were not possible. One of them is the locality of the specifics of the format that can be converted from module to module if needed. The other is the coexistence of alternative modules that do the same task yielding non-linear "pipeline" if needed. Generally, modules may require zero or more fields as input columns and produce zero or more output fields as columns on the output. To synchronise naming conventions between modules from different sources, an abstract *rename* module can be implemented at framework level for renaming columns.

It is not strictly required for input and output data to be in the above defined format. This is an advantage that shows itself mostly at the beginning and at the end of a pipeline. The first step is usually a *tokeniser* which creates `xtsv` format from raw text, while the last step is often a *finaliser* which aggregates data, e.g. printing out result of evaluation or data in appropriate format for plotting directly.

The execution order is determined by the order of the modules. Prior to execution, the framework checks whether the produced and required fields yield a feasible sequence of modules, i.e. whether all required fields are available at all points of the pipeline, taking the input data into account as well. This makes it possible to compute execution paths from the input and the desired output using the available modules with constraint satisfaction.

With the design presented above, we allow new possibilities – that could not be achieved with CoNLL-U format – from processing multilingual texts with the same pipeline[14] to tasks even outside of the domain of natural language processing, essentially using any kind of vertical data. In the next section, we emphasise the importance of the `xtsv` framework. We present how this framework facilitates the insertion of new modules and what can be done with these modules at the pipeline level.

6. The `xtsv` Framework

`xtsv` is a general-purpose, multi-functional `tsv`-based processing framework for NLP. It connects different modules, manages I/O using the format described in the previous section, handles and checks input and output columns and, most importantly, adds several functionalities to the pipeline. The main idea is to implement every feature which works the same way for all modules at framework level, and thereby, to provide them automatically for every module and pipeline built with the framework.

We wanted to support the vast majority of the available tools in the research community. So, we not only designed our API in Python (which can be easily integrated with C and C++ libraries), but also added the possibility to insert Java applications into the pipeline using PyJNIus [15] doing all the heavy lifting. PyJNIus is not a direct dependency of `xtsv`, but detected and used if some module needs it. The pipeline delegates the appropriate tasks to the format (see Section 5.) and the modules (Section 6.2.).

[14]This can be achieved by adding a language identifier column by the first module to set the path for appropriate follow-up modules for parts of the input written in different languages.

[15]`https://github.com/kivy/pyjnius`

Another design decision was to reuse as many available general-purpose software components as possible if they are adopted by the community. With this trick, `xtsv` itself consists of only about 400 lines of Python code. Despite of its numerous features that we describe in the following paragraphs, it is a lightweight system, easy to maintain thanks to its small size.

Before reading the data, the pipeline checks the feasibility of the list of the tools required to be run and the supplied input data. As any module can require and produce zero or more input fields, this is a directed acyclic hypergraph. All the data is handled as a stream of tokens – except of the raw text before tokenisation and sentence splitting which is handled as a stream of characters. We currently process data sentence-by-sentence[16] which enables the parallelisation of the processing and distributing it among multiple computer nodes with the same API.

We defined the following API for *using* a pipeline put together in `xtsv`– this is similar to the existing alternatives:

- *build_pipeline(...)* to initialise the defined pipeline and process the input stream in one step;
- *pipeline_rest_api(...)* to initialise the REST API object that will initialise the tools on the first query when they are needed;
- *process(...)* low level function to process a stream with a preinitialised module (e.g. for training);
- *download(...)* download the models if needed
- *tools* (a list) and *presets* (a dictionary) for configuration along with miscellaneous properties and functions.

Suitable tools can run as in multiple instances with different parameters – e.g. loading different models. We defined an object to store the initialised tools. Having this object, other streams can reuse the tools immediately. One frequent use case of this is the REST API.

6.1. Usage options: Docker, REST API, etc.

Beyond the programmatic usage described above, there are two orthogonal dimensions concerning other usage possibilities of `xtsv`. On the one hand, the framework can be used as a CLI (command line) tool, as a REST API or by a generic web front end built on the top of the REST API. On the other hand, it can be used by cloning the repository – to be able to modify the source code directly –, by installing via `pip` or by running its Docker container as a black box (see Table 1).

We encourage our users to use Docker container technology to create prepacked, standalone and easy-to-use software packages that have batteries included and can be put into larger environments by a few commands. Using our framework, instead of complicated installing procedures, one can simplify the setup to one

[16]The API can be extended when lower or higher processing units are needed. For details, see the module specification in Section 6.2.

	clone	pip	Docker
CLI	✓	✓	✓
REST API	✓	✓	✓
web front end	✓	✓	✓

Table 1: Usage options of `xtsv`.

```
docker pull IMAGE_NAME
```

command. The system can be run from docker – just like a common CLI application running in JVM, called runnable docker – or like a traditional container, as a service with the provided general REST API. We also provide a simple generic web front end from `xtsv` which helps human interaction with the REST API and enhances user experience e.g. at a demo session, if some ideas need to be quickly showcased.

We created a front end for the REST API to enable non-technical users to select the required modules and use the service provided by `xtsv` without knowing CLI or docker at all. As some modules can run in multiple copies (e.g. with different parameters or modes) the same front end code also provides a way to choose between these parameter configurations or modes and return a JSON structure with the processed output. This scenario is good for running small microservices in the cloud which require human readable, still machine parsable output e.g. to look-up – and possibly post-process – some input text by typing an URL into a browser. The very same API can be used to handle batch queries if needed. The practical use cases are described in Section 7.1.

The framework uses lazy-initialisation to only load modules that are really needed. This is useful for running services as the user does not have to to modify anything if he or she wants to run (and load) only one tool in the lifetime of the service, or – as another case – wants to use different worker computers for different services systematically. The list of modules has are to be supplied in a form of a simple config that lists the appropriate values[17]. In the following section, we describe the specific API that is required of any module to implement, to show how easy it is to insert a module into an `xtsv` pipeline.

6.2. Module Specification

In order to *create* a new `xtsv` module to put into a pipeline, one must create a Python module with a class implementing the module API which consists of six properties and three functions[18]. As a matter of fact, this includes handling modules with Java classes.

The modules have the following six properties which can be omitted if default values are appropriate for a particular use:

1. *source_fields*: the set of expected input field names if there are any;
2. *target_fields*: the list of the names for the output fields created by the module in order of appearance;
3. *fixed_order_tsv_input*: tells the pipeline if it should expect headers for the module input or not;
4. *pass_header*: tells the pipeline if it should pass the header before the module output or not (e.g. header should not be passed for finalisers – see Section 5.);
5. *class_path*: needed for the JVM to include special classes;
6. *vm_opts*: additional JVM options when needed.

These properties determine the pipeline-level properties of the actual module. The main idea on the behaviour of each module – be it rule-based, statistic-based or hybrid – is that it is enough for them to have:

1. an *initialisation phase:* `__init__(...)`, where the module can read the trained model or data files and make modifications prior to processing;
2. a function to *process the input token-wise:* `process_token(...)`;
3. a function to *process the input sentence-wise:* `process_sentence(...)`.

In the future, we may add a function to process the input paragraph-wise and another one to process the input document-wise.

These functions have strict signature as the part of the API, except the initialisation function which is allowed to have the custom elements needed for the actual module. Each module is allowed to define any further extension in its API as long as it does not interfere with the expected API of `xtsv`.

To sum up, we emphasise that `xtsv` will fulfil all requirements described in Section 3. in the near future[19]. If the API described in the current section is implemented by a module, it can be simply inserted into an `xtsv` pipeline and, by that, all features which we mentioned will be available for this module, and of course for the whole pipeline which contains the module. This is possible because all pipeline functionalities are implemented in `xtsv` at framework level and in a generic way.

7. An Application: `e-magyar` (emtsv)

In this section, we present the advantageous properties of `xtsv` in practice through an application. This is the new version of `e-magyar` codenamed `emtsv` (Indig et al., 2019), a processing pipeline for Hungarian. We will reference it as `emtsv` in the rest of the paper.

Hungarian is an agglutinative language. There are several specific processing tools for this language, many of them producing high quality output. For example, *Hunspell* is an open-source spell checker, lemmatiser and morphological analyser, developed primarily to handle Hungarian and

[17]For details see the documentation at `https://github.com/dlt-rilmta/xtsv#creating-a-module-that-can-be-used-with-xtsv`

[18]See `emDummy` example module for detailed documentation at `https://github.com/dlt-rilmta/emdummy`

[19]The only exception is parallelisation that has not been implemented yet.

other languages with rich morphology, and also, there are some other NLP tools (Halácsy et al., 2004). Since then some of these tools were substituted for better alternatives, some has been reimplemented, fixed and extended yielding different pipelines competing with each other.

In order to accelerate the development of the tools and resources, we came up with the original idea of `xtsv` earlier (Indig et al., 2019). Back then, it was only a part of the system, but as we realised its importance we separated it as an independent tool. Then, we formalised the desirable properties of this kind of framework in general (see Section 3.).

In this section, we justify the legitimacy of `xtsv` by presenting how it helped `emtsv` to become a full-fledged prototyping ecosystem for Hungarian NLP. We also present some loosely connected applications built around `emtsv` to show how simple ideas can substantially extend the existing tools in practice by using `xtsv`. In Section 9., we present our future plans regarding further applications of our framework.

7.1. Morphological Analyser as a Service (MAaaS)

`emtsv` uses the rule-based morphological analyser `emMorph` (Novák et al., 2016) and the lemmatiser `emLem` together as a module[20] to overcome the sparse data problem introduced by the nature of the language and to assist the ML algorithm in the POS tagger[21]. To be able to verify the output (e.g. if there is a putative problem in the morphological analyser) one needs a preinstalled environment that is not always available or convenient for standalone usage.

For this purpose, we created a microservice[22] from the the morphological analyser with just a few extra lines of code using `xtsv` (Indig et al., 2019). This enables the users to check possible analyses of a specific word form by just typing a special URL into their browser[23]. To make the interface more user-friendly, we introduced a web form to allow the user to choose from available modes with one click. The output is in JSON format which is readable by humans and also by machines.

Since then, we have integrated *Hunspell* into `emtsv` as a module and set up a similar service[24] for fast spellchecking and to provide another lemmatiser and morphological analyser for the users to choose from.

With these at hand, even non-technical users can check the spelling and possible analyses of a word, even from their smartphones. We recommend these tools for accelerating the building of a new training corpus for Hungarian, which we will introduce in the following section.

[20]`https://github.com/dlt-rilmta/emmorphpy`

[21]Actually, this module does morphological disambiguation and lemmatisation as well, in our case.

[22]A demo service running in the cloud is available at `https://emmorph.herokuapp.com/`.

[23]`https://hunspellpy.herokuapp.com/analyze/terem` where "terem" (room, yield, produce) is the word to analyse.

[24]`https://hunspellpy.herokuapp.com/`

7.2. Creating a New Training Corpus

The Szeged treebank is the largest, manually annotated treebank (Vincze et al., 2010) in Hungarian at the time of writing which contains 82,000 sentences, 1.2 million words and 250,000 punctuation marks, while *KorKorpusz* (31,492 tokens) is a recent pilot to create a new one (Vadász, 2020) and therefore it is not included in the examined tools yet. Due to copyright reasons, the Szeged treebank is not publicly available. Only a small part of it can be downloaded[25] in UD format with 42,000 tokens and 1,800 sentences. This corpus is the base of the Hungarian models in Section 4. As these tools do not use rule-based components to improve their decisions, this explains their inferior performance on Hungarian and the issue could only be solved with more training material.

However, creating gold standard quality training materials for Hungarian is a cumbersome task. The point is that we can make the work easier for annotators with `xtsv`. They can bootstrap the annotation with high-quality components and only a few errors remain and have to be be corrected (Indig et al., 2019). This accelerates the annotation process because annotators can work step-by-step. They only have to stop after running the desired module, correct the output and continue running the pipeline on the corrected data. Using this strategy, they can avoid the error amplification in the pipeline and prevent larger modifications e.g. the need for renumbering every dependency relation because of a tokenisation mistake.

7.3. The Particular Module Hierarchy in `emtsv` to Improve POS Tagging

In this section, we describe how we broke down the module hierarchy in order to further enhance annotation experience and to give more fine-grained control to the annotators and other users. With `xtsv`, we had the opportunity to redefine classical NLP modules in `emtsv` to enhance future usage scenarios and to lay out the foundations for future modules (see Section 9.). We divided the "POS tagger" into four components (Indig et al., 2019): a) the morphological analyser to provide morpheme level analyses for words, b) the lemmatiser to simplify the morpheme level analysis to lemmas and (`emMorph` specific) tags suitable for the OOV handling in the next step, c) disambiguate the lemmas and tags using the simplified morphological information, d) convert the disambiguated tags into UD POS tags and features to be able to continue with any standard UD dependency parser, while the computed fine-grained morphological information can be used to do other tasks independently. This layout allows the fine-grained classification and disambiguation of words before losing morphological information at the conversion step to the coarse UD representation. Only the third step – see c) in the previous paragraph – is done by machine learning, the rest of the steps are rule-based. The cooperation of these modules gives the user freedom of choice and high quality annotation at the same time.

[25]`https://universaldependencies.org/treebanks/hu_szeged/index.html`

7.4. Foreign Pipeline Elements as Modules in `emtsv`

To keep up with the state of the art, we have to try new methods and substitute the old ones with them from time to time. In this section, we show how we integrated a full pipeline into `emtsv` in order to provide alternatives for certain tasks.
As the `xtsv` implementation made it possible, we started to search for new opportunities for interoperability with other modules. Simon et al. (2020) have successfully integrated `UDPipe` which shows how our modules correspond to the standard pipeline elements and allows combining alternative tools in the pipeline, either it is speed or quality that is in the user's mind[26]. We also offer additional speed-up as multiple modules of `UPDipe` can be run as a single `xtsv` module if the user does not want to interact with the data between the modules.

7.5. Loosely Connected and Experimental Modules in `emtsv`

It is not rare that other frameworks and software ecosystems (like `tensorflow-keras` and `sci-kit learn`) integrate – and usually reimplement – modules which are only loosely connected to the original core task. Designing `emtsv` Simon et al. (2020) created three such modules to conveniently conduct everyday tasks like performance evaluation and to support interoperability with the CoNLL-U ecosystem. For a further example, the module of `emconll` (Indig et al., 2019) converts the output to CoNLL-U format which has specific columns in a fixed order[27]. It allows e.g. connecting `emtsv` with visual dependency graph editors to help correcting annotation.
Taking advantage of the new possibilities of `xtsv`, some experimental modules were implemented, for example `emTerm`. It is not a traditional, but a useful pipeline element. `emTerm` can be used to find and mark multi-word phrases (e.g. legal terms, proper names, etc.) in POS tagged text, from a simple predefined list.

8. Comparing `xtsv` with Other NLP Pipelines

Simon et al. (2020) compared the speed of vanilla `xtsv`, `emtsv` and other mentioned NLP pipelines to get an insight on their throughput. We used *UDPipe*'s own Hungarian model and an unofficial model for *HuSpaCy*[28]. We compared two scenarios, all starting from raw text (see Table 2). The first is ended at POS tagging and the second is ended at dependency Parsing. All measurements were run on RAM disk to eliminate the artefacts due to I/O and the numbers are averaged from three measurements on 1 million tokens. For dependency parsing, `emtsv` and *Magyarlánc* has only a slight difference in speed, while the other competitors are significantly faster. We note that *HuSpaCy* simplifies the task of POS-tagging (creating the main CoNLL-U POS tags only), so its task is much simpler. As `emtsv` and *Magyarlánc* have the same dependency parser which appears to be a bottleneck.

pipeline	POS	DEP
`emtsv` (CLI)	2.320	300
`emtsv` (REST)	2.600	310
Magyarlánc	5.550	450
UDPipe	9.280	3.300
HuSpaCy	33.980	15.000

Table 2: The speed of the different compared systems in token per sec.

	1	**2**	**3**	**4**
1	825	719	677	618
2	483	412	395	373
3	342	294	283	269
4	265	227	214	207
5	211	186	176	166

Table 3: The vanilla throughput of `xtsv` in thousand tokens per sec with `emDummy` as the only module. The columns shows the number of columns in the tsv, while the rows show the number of consecutive modules.

In Table 3, one can see the raw throughput on `emDummy` – a module which echoes input without modification. We can see that there is a room for improvement on the raw speed, but the optimisation should start at the modules as they clearly are the real bottleneck.

9. Conclusion and Future Work

We collected the desirable properties of pipelines, then we designed an open source pipeline framework called `xtsv`[29] which meets these requirements. We emphasise that it is completely modular: the pipeline can be extended by new modules easily and modules can be replaced at any time, every module or combination of modules can be run separately. Besides, the framework has a convenient intermodular communication format and adds useful functionality to the pipeline automatically, e.g. Docker container, REST API or a web front end. We proved the usefulness of `xtsv` in practice, investigating the `emtsv` pipeline built with it for Hungarian.
We have many ideas we plan to implement as new modules beyond the aforementioned ones we have already realised, thanks to the possibilities of `xtsv`: diacritic restorer hyphenator (to be used after the morphological analyser), the *jump and stay algorithm* (Sass, 2019) (to be used after the dependency parser) to find proper verb-centered constructions, disambiguate the morpheme level analyses to enable task dependent lemmatisation, or reuse implemented metrics e.g. for stylometry analysis[30]. Integration, use and evaluation of all these is now quite straightforward using `xtsv`.

[26]It must be noted that, there are some incompatibilities between the tools because the different training material (UDv1 and UDv2).
[27]`http://universaldependencies.org/format`
[28]`https://github.com/oroszgy/spacy-hungarian-models/`

[29]Available under LGPL 3.0 license `https://github.com/dlt-rilmta/xtsv` or `pip install xtsv`.
[30]`https://github.com/tsproisl/Linguistic_and_Stylistic_Complexity`

10. Bibliographical References

Honnibal, M. and Montani, I. (2017). spacy 2: Natural language understanding with bloom embeddings. *Convolutional Neural Networks and Incremental Parsing*.

Indig, B., Sass, B., Simon, E., Mittelholcz, I., Vadász, N., and Makrai, M. (2019). One format to rule them all – the emtsv pipeline for Hungarian. In *Proceedings of the 13th Linguistic Annotation Workshop*, pages 155–165, Florence, Italy, August. Association for Computational Linguistics.

Novák, A., Siklósi, B., and Oravecz, C. (2016). A new integrated open-source morphological analyzer for Hungarian. In *Proceedings of the Tenth International Conference on Language Resources and Evaluation (LREC'16)*, pages 1315–1322, Portorož, Slovenia, May. European Language Resources Association (ELRA).

Qi, P., Dozat, T., Zhang, Y., and Manning, C. D. (2018). Universal dependency parsing from scratch. In *Proceedings of the CoNLL 2018 Shared Task: Multilingual Parsing from Raw Text to Universal Dependencies*, pages 160–170, Brussels, Belgium, October. Association for Computational Linguistics.

Sass, B. (2019). The "jump and stay" method to discover proper verb centered constructions in corpus lattices. In *Proceedings of the International Conference Recent Advances in Natural Language Processing, RANLP 2019*, Varna, Bulgaria. INCOMA Ltd.

Sejnowski, T. J. (2018). *The Deep Learning Revolution*. MIT Press, Cambridge, MA.

Simon, E., Indig, B., Kalivoda, A., Mittelholcz, I., Sass, B., and Vadász, N. (2020). Újabb fejlemények az e-magyar háza táján. In Gábor Berend, et al., editors, *XVI. Magyar Számítógépes Nyelvészeti Konferencia (MSZNY 2020)*, pages 29–42, Szeged. Szegedi Tudományegyetem, TTIK, Informatikai Intézet.

Straka, M. and Straková, J. (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.

Stranák, P. and Štepánek, J. (2010). Representing layered and structured data in the conll-st format. In *Proceedings of the Second International Conference on Global Interoperability for Language Resources*, pages 143–152.

Vincze, V., Simkó, K., Szántó, Z., and Farkas, R. (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, April. Association for Computational Linguistics.

Zsibrita, J., Farkas, R., and Vincze, V. (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.

11. Language Resource References

Halácsy, P., Kornai, A., Németh, L., Rung, A., Szakadát, I., and Trón, V. (2004). Creating open language resources for Hungarian. In *Proceedings of the Fourth International Conference on Language Resources and Evaluation (LREC'04)*, Lisbon, Portugal, May. European Language Resources Association (ELRA).

Vadász, N. (2020). KorKorpusz: kézzel annotált, többrétegű pilotkorpusz építése. In Gábor Berend, et al., editors, *XVI. Magyar Számítógépes Nyelvészeti Konferencia (MSZNY 2020)*, pages 141–154, Szeged. Szegedi Tudományegyetem, TTIK, Informatikai Intézet.

Vincze, V., Szauter, D., Almási, A., Móra, Gy., Alexin, Z., and Csirik, J. (2010). Hungarian Dependency Treebank. In *Proceedings of LREC 2010*, Valletta, Malta, May. ELRA.