Pimlico: A toolkit for corpus-processing pipelines and reproducible experiments

Mark Granroth-Wilding
University of Helsinki
mark.granroth-wilding@helsinki.fi

Abstract

We present Pimlico, an open source toolkit for building pipelines for processing large corpora. It is especially focused on processing linguistic corpora and provides wrappers around existing, widely used NLP tools. A particular goal is to ease distribution of reproducible and extensible experiments by making it easy to document and re-run all steps involved, including data loading, pre-processing, model training and evaluation. Once a pipeline is released, it is easy to adapt, for example, to run on a new dataset, or to re-run an experiment with different parameters. The toolkit takes care of many common challenges in writing and distributing corpus-processing code, such as managing data between the steps of a pipeline, installing required software and combining existing toolkits with new, task-specific code.

1 Introduction

It is becoming more and more common for conferences and journals in NLP and other computational areas to encourage, or even require, authors to make publicly available the code and data required to reproduce their reported results. It is now widely acknowledged that such practices lie at the center of open science and are essential to ensuring that research contributions are verifiable, extensible and useable in applications. However, this requires extensive additional work. And, even when researchers do this, it is all too common for others to have to spend large amounts of time and effort preparing data, downloading and installing tools, configuring execution environments and picking through instructions and scripts before they can reproduce the original results, never mind apply the code to new datasets or build upon it in novel research. Whilst sometimes it may be sufficient to release a script that performs all of the data processing, model training and experimental evaluation steps, often this is not a practical approach to multi-stage processing of large corpora.

We present a toolkit, *Pimlico* (**Pi**pelined **M**odular **Li**nguistic **Co**rpus processing), that addresses these problems. It allows users to write and run potentially complex processing pipelines, with the key goals of making it easy to:

- clearly document what was done;
- incorporate standard NLP and data-processing tasks with minimal effort;
- integrate non-standard code, specific to the task at hand, into the same pipeline; and
- distribute code for later reproduction or application to other datasets or experiments.

The toolkit is written in Python and released under the open source LGPLv3 license[1]. It comes with pre-defined modules to wrap a number of existing NLP toolkits (including non-Python code) and carry out many other common pre-processing or data manipulation tasks. Comprehensive documentation is maintained online[2].

In this paper, we describe the core concepts that Pimlico is built around and some of its key features. We also describe a number of the core modules that come built into the toolkit and we present an example pipeline. Finally, we explain how the toolkit addresses the stated goals and outline plans for future development.

2 Building pipelines

Pimlico addresses the task of building of pipelines to process large datasets. It allows you to run one or several steps of processing at a time, with high-level control over how each step is run, manages

[1] https://github.com/markgw/pimlico/
[2] https://pimlico.readthedocs.io/

Proceedings of Second Workshop for NLP Open Source Software (NLP-OSS), pages 101–109

```
[split]
type=pimlico.modules.corpora.split
input=tokenized_corpus
set1_size=0.8
```

Figure 1: Example configuration section specifying a single module in a pipeline. The module has a single input, taken from an earlier module's output, and a single parameter.

the data produced by each step, and lets you observe these intermediate outputs. Pimlico provides simple, powerful tools to give this kind of control, without needing to write any code.

Developing a pipeline with Pimlico involves defining the structure of the pipeline itself in terms of *modules* to be executed and connections between their inputs and outputs describing the flow of data. Modules correspond to some data-processing code, with some parameters. They may be of a standard type, so-called *core* modules, for which code is provided as part of Pimlico. A pipeline may also incorporate custom module types, for which metadata and data-processing code must be provided by the author.

2.1 Pipeline configuration

At the heart of Pimlico is the concept of a pipeline configuration, defined by a configuration (or *conf*) file, which can be loaded and executed. This specifies some general parameters and metadata regarding the pipeline and then a sequence of modules to be executed.

Each pipeline module is defined by a named section in the file, which specifies the module type, inputs to be read from the outputs of other, previous modules, and parameters. For example, the configuration section in Fig. 1 defines a module called `split`. Its type is the core Pimlico module type *corpus split*[3], which splits a corpus by documents into two randomly sampled subsets (as is typically done to produce training and test sets). The option `input` specifies where the module's only input comes from and refers by name to a module defined earlier in the pipeline whose output provides the data. The option `set1_size` tells the module to put 80% of documents into the first set and 20% in the second. Two outputs are produced, which can be referred to later in the pipeline as `split.set1` and `split.set2`.

[3] https://pimlico.readthedocs.io/en/latest/modules/pimlico.modules.corpora.split.html

The first module(s) of a pipeline have no inputs, but load datasets, with parameters to specify where the input data can be found on the filesystem. A number of standard **input readers** are among Pimlico's core module types to support reading of simple datasets, such as text files in a directory, and some standard input formats for data such as word embeddings. The toolkit also provides a factory to make it easy to define custom routines for reading other types of input data.

The **type** of a module is given as a fully qualified Python path to a Python package. The package provides separately the module type's metadata, referred to as its 'module info' – input datatypes, options, etc. – and the code that is executed when it is run, the 'module executor'. The example in Fig. 1 uses one of Pimlico's core module types. A pipeline will usually also include non-standard module types, distributed together with the conf file. These are defined and used in exactly the same way as the core module types. Where custom module types are used, the pipeline conf file specifies a directory where the source code can be found.

An example of a complete pipeline conf, using both core and custom module types, is shown in Fig. 2 and is described in more detail in Section 6.

2.2 Datatypes

When a module is run, its output is stored ready for use by subsequent modules. Pimlico takes care of storing each module's output in separate locations and providing the correct data as input.

The module info for a module type defines a **datatype** for each input and each output. Pimlico includes a system of datatypes for the datasets that are passed between modules. When a pipeline is loaded, type-checking is performed on the connections between modules' outputs and subsequent modules' inputs to ensure that appropriate datatypes are provided.

For example, a module may require a vocabulary as an input, for which Pimlico provides a standard datatype. The pipeline will only pass checks if this input is connected to an output that supplies a compatible type. The supplying module does not need to define how to store a vocabulary, since the datatype defines the necessary routines for writing a vocabulary to disk. The subsequent module does not need to define how to read the data, since the datatype takes care of that too, providing the module executor with suitable Python data structures.

```
# Options for the whole pipeline
[pipeline]
name=custom_module_example
# Pimlico version this is designed to work with
release=0.9.23
# Python source dir, relative to config file:
# needed for the custom module type
python_path=src/

# Specify input paths, etc at the top
[vars]
text_path=%(pimlico_root)s/examples/data/input/bbc/data

# Read in the raw text files
[input_text]
type=pimlico.modules.input.text.raw_text_files
files=%(text_path)s/*

# Tokenize the text using the spaCy tokenizer
[tokenize]
type=pimlico.modules.spacy.tokenize
input=input_text

# Rough filter to remove proper nouns: custom module
[filter_prop_nns]
type=pim_example.modules.filter_prop_nns
input=tokenize

# Build vocabulary from words used:
# can be used to map words to IDs
[vocab]
type=pimlico.modules.corpora.vocab_builder
input=filter_prop_nns
# Only include words that occur >=5 times
threshold=5
```

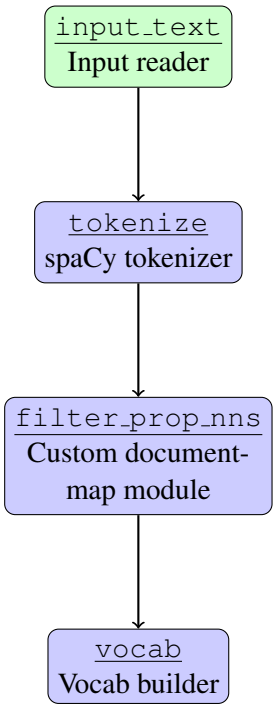

Figure 2: Full example pipeline which loads a dataset from raw text files, tokenizes it and applies some custom processing. The file, together with the source code for the custom module type, are available at https://github.com/markgw/pimlico/tree/master/examples. Alongside is a graphical representation of the pipeline structure.

Often modules read and write *corpora*, consisting of a large number of documents. Pimlico provides a datatype for representing such corpora and a further type system for the **types of the documents** stored within a corpus (rather like Java's *generic* types). For example, a module may specify that it requires as input a corpus whose documents contain tokenized text. All tokenizer modules (of which there are several) provide output corpora with this document type. The corpus datatype takes care of reading and writing large corpora, preserving the order of documents, storing corpus metadata, and much more.

The datatype system is also extensible in custom code. As well as defining custom module types, a pipeline author may wish to define new datatypes to represent the data required as input to the modules or provided as output.

2.3 Running the pipeline

Pimlico provides a command-line interface for parsing and executing pipelines. The interface provides sub-commands to perform different operations relating to a given pipeline. The conf file defining the pipeline is always given as an argument and the first operation is therefore to parse the pipeline and check it for validity. We describe here a few of the most important sub-commands.

status. Outputs a list of all of the modules in the pipeline, reporting the execution status of each. This indicates whether the module has been run; if so, whether it completed successfully or failed; if not, whether it is ready to be run (i.e. all of its input data is available).

Each of the modules is numbered in the list, and this number can be used instead of the module's full name in arguments to all sub-commands.

Given the name of a module, the command out-

puts a detailed report on the status of that module and its input and output datasets.

run. Executes a module. An option `--dry` runs all pre-execution checks for the module, without running it. These include checking that required software is installed (see Section 3.2) and performing automatic installation if not.

If all requirements are satisfied, the module will be executed, outputting its progress to the terminal and to module-specific log files. Output datasets are written to module-specific directories, ready to be used by subsequent modules later.

Multiple modules can be run in sequence, or even the entire pipeline. A switch `--all-deps` causes any unexecuted modules upon whose output the specified module(s) depend to be run.

browse. Inspects the data output by a module, stored in its pipeline-internal storage. Inspecting output data by loading the files output by the module would require knowledge of both the Pimlico data storage system and the specific storage formats used by the output datatypes. Instead, this command lets the user inspect the data from a given module (and a given output, if there are multiple).

Datatypes, as part of their definition, along with specification of storage format reading and writing, define how the data can be formatted for display. Multiple formatters may be defined, giving alternative ways to inspect the same data.

For some datatypes, browsing is as simple as outputting some statistics about the data, or a string representing its contents. For corpora, a document-by-document browser is provided, using the Urwid[4] library. Furthermore, the definition of corpus document types determines how an individual document should be displayed in the corpus browser. For example, the tokenized text type shows each sentence on a separate line, with spaces between tokens.

2.4 Document map modules

A common type of module is one that takes input from one or more corpora, applies some independent processing to each document in turn and outputs a new corpus containing the processed data for the same set of documents. For example, we might lower-case the text of each document; map words to IDs from a vocabulary; or perform document-level topic inference using a pre-trained topic model.

[4] http://urwid.org/

Pimlico makes it easy to define such modules, referred to as *document map modules*. The module executor can be defined using a factory, simply specifying a function to be applied independently to each document. It may also define pre- and post-processing functions to be run before and after the document mapping process.

Such modules lend themselves naturally to parallelization, since separate documents can be processed independently by worker processes in a pool. When a document map module is defined using the factory, this simple type of parallelization is provided by default, using Python's `multiprocessing` module. The user simply needs to specify when running a module how many processes Pimlico should use and this number of workers will be launched to process documents.

Furthermore, any document map module can be set to run in *filter* mode, using the `filter=T` option. This causes its processing to be performed on the fly as required by subsequent modules, instead of being stored to disk. The module then no longer appears in the list of executable modules, since it will be executed as necessary to provide inputs to subsequent modules when they are run. If an output corpora is used a number of times, this approach is inefficient, but if not, and especially if the per-document processing is fast, this can lead to a more streamlined workflow.

3 Some key features

3.1 Data management

Data output by a module is stored ready for other modules to use. Pimlico manages storage locations specific to the pipeline, module and output, and provides the correct version of the data to modules that use the data as input.

Pimlico can be configured to use any location on the filesystem for pipeline-internal storage. Beyond this, the user does not need to concern themselves with the storage structure, nor data storage formats, which are managed by the datatype system.

The command-line interface provides a `reset` command to remove the output data of a given module and any subsequent modules that depend on it. This is useful, for example, if changing a module parameter and rerunning it.

3.2 Software dependencies

Executing a module will often depend on having some software installed. This may be Python pack-

```
[model_train]
type=mycode.modules.train_model
input=tokenized_text
regularization=0.1
layers=5|10

[model_eval]
type=mycode.modules.eval_model
input=model_train
```

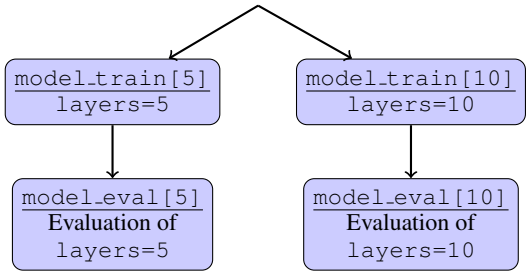

Figure 3: Example pipeline fragment defining a module with alternative values for an option. The diagram shows how the two modules are expanded into branches for the alternatives.

ages, for pure Python modules, or other types of software. For example, Pimlico's core modules include wrappers around the OpenNLP Java toolkit, so running modules of one of these types requires the Java Runtime Environment (JRE) as well as the OpenNLP jar packages.

Pimlico includes a software dependency management system. Software dependencies of many different types can be defined, such as Python packages, Java libraries, compiled C++ binaries and so on. A software dependency definition includes a routine to test whether the software is available and, wherever possible, a routine to automatically install the software in a location that is local to the pipeline's execution environment. For example, Python dependencies can be simply defined by reference to a Pip[5] package, which can be automatically downloaded and installed within a Python virtual environment using the Pip library.

Each module type lists software that it depends on to run as part of its module info. When the user attempts to run a module or checks whether it is ready to run (using the `run` subcommand, Section 2.3), Pimlico checks all the dependencies and installs the necessary software by running the installation routine. A module's executor is strictly separated from its module info and is not loaded until all dependency checks are passed. This allows a module type programmer to freely write code within the executor that loads dependent libraries.

For example, the core module for training topic models using the Gensim toolkit (Řehůřek and Sojka, 2010) can only be run when the Gensim Python library is installed. Its module info declares this dependency. When a user attempts to run a module of this type in a pipeline, Pimlico uses Pip to automatically install the library before executing. In this way, another user subsequently receiving the pipeline does not need to make sure that they have installed this package on their system before running the pipeline.

Requirements of specific versions of dependencies are currently supported for some types of dependencies. In future, this will be extended, including more sophisticated handling of conflicting versions within a pipeline.

3.3 Module alternatives

Examples so far have been of linear pipelines, where each module's output feeds into the input for the next. Pipeline structures are not restricted to this: they may branch arbitrarily by defining multiple modules that take input from the same source, or combine branches with a single module that takes multiple inputs. Several tools are provided to assist concise definition of complex pipeline structures. One we describe here is *module alternatives*.

Consider a hypothetical module type, used in Fig. 3, that takes one input corpus and trains a machine learning model on the data. It has a parameter `layers` which takes a numeric value. We wish to train models with several different values for this parameter and apply the same evaluation to each.

We could do this by defining multiple modules of this type, each training a different model. We would then need to duplicate the subsequent evaluation module to create a version for each model. Pimlico provides a more concise way to do this. We define one module, `model_train`, and specify a list of alternative values for the `layers` parameter: `layers=5|10`. The module is automatically expanded into multiple modules, one for each parameter value. Each is given a distinct name, which may be specified explicitly or automatically generated – `model_train[5]` and `model_train[10]`.

Subsequent modules can also be expanded automatically, propagating the set of alternatives through the pipeline to create separate branches. In our example, we define a single evaluation module

[5] https://pip.pypa.io/en/stable/

`model_eval`, which declares its input to come from `model_train` (the name of the training module prior to expansion). This is expanded into `model_eval[5]` and `model_eval[10]`, each alternative taking input from the respective model training module.

Further details of expansion, combination and naming of module alternatives are given in the documentation.

3.4 Pipeline variants

In a pipeline that processes a large corpus, it can take hours or even days to run a single module. While developing and testing the pipeline, it is not convenient to blindly write the entire configuration and module code without testing, or to have to execute long-running modules simply to get some input data to test custom code later in the pipeline. Pimlico provides a solution: pipeline *variants*.

Variants are independent pipelines, sharing no internal datasets or state, defined by a single config file. A special syntax can be used in the file to prefix lines that are to appear only in a specific variant. Other lines are included in all variants. This can be used to set different values of module parameters in different variants, or even include whole modules in only one variant. When Pimlico is run, it will by default load the standard variant, always called 'main'. A command-line option can specify another variant to load.

The most common use of this is to define a *small* variant, which only processes a small subset of the input data. It may do this, for example, by setting parameters of the input reader, or including a *subset* module to truncate the corpus. The entire pipeline can then be run to test configuration and custom code and sanity-check the resulting datasets, before setting the pipeline running on the full dataset.

Other uses of this feature include running an identical pipeline on different input corpora.

4 Code distribution

One of the key problems that Pimlico sets out to solve is the difficulty of distributing code in a way that makes it easy for others to reproduce and extend the processing. It achieves this by making the full processing pipeline explicit in the pipeline conf file. It is therefore crucial that (a) it is easy to distribute all the necessary files to re-run a pipeline; and (b) it is easy for someone else, given these files to get the pipeline running.

4.1 Releasing pipelines

Three elements of a pipeline need to be distributed: (1) a full description of the processing pipeline; (2) any code needed to run the pipeline that is not part of a standard library; and (3) input data. (1) is trivial with Pimlico, since a pipeline's conf file is all that is needed. (2) requires simply that all code in the path from which custom code is loaded is distributed. This can simply be packaged into an archive together with the conf file. Pimlico's source code does not need to be distributed, since it can be downloaded as necessary. Other libraries will generally be downloaded and installed automatically by Pimlico when the pipeline is to be run, as described in Section 3.2.

Pimlico does not attempt to address the distribution of datasets used as input data. It is usually appropriate to distribute these separately in a way that respects licenses and handles distribution of large files. Much of the time, input data is not specific to a pipeline, but comes from existing corpora.

4.2 Using and extending pipelines

Upon receiving the files providing (1) and (2) above, you can use Pimlico's *bootstrap* tool to set up a working environment for running the pipeline. A Python script, `bootstrap.py`, is available from the online documentation. This reads the config file to check what version of Pimlico was used when it was originally run and downloads the same release. It then prompts Pimlico to set up a Python virtual environment and install core software dependencies. After this, the pipeline is ready to be loaded and run.

Having loaded a pipeline and set up the environment, it is easy to extend or adjust the pipeline to run further experiments or build on the previous work. New modules can be added and parameters to the existing modules changed. Pimlico's system of standardized internal datatypes for passing data between modules also makes it straightforward to apply the same pipeline to a different dataset. All that is required is a suitable *input reader* for the new data (see Section 2.1). This supplies the dataset in a standard, pipeline-internal format, so the rest of the pipeline can be run without modification.

5 Core module types

Pimlico comes with a large number of *core* module types, for which a pipeline author needs to write no code, but simply define the module configuration

in their config file. This set is being constantly expanded.

The following list gives some examples of core module types provided with Pimlico. The full list is available in the documentation.

- Generic corpus manipulation, including shuffling, concatenation, truncation, subsampling, random splitting
- Vocabulary building, word-to-ID mapping
- Gensim topic model training (Řehůřek and Sojka, 2010)
- Malt dependency parsing (Nivre et al., 2006)
- OpenNLP[6] tokenization, POS tagging, constituency parsing
- Word embedding (Mikolov et al., 2013) loading, manipulation, storing
- Word embedding training using `word2vec` (Mikolov et al., 2013) and *fastText* (Mikolov et al., 2018)
- Text normalization (lower-casing, etc.)
- Scikit-learn classifier training (Pedregosa et al., 2011)

The core module types also serve as a reference for defining custom module types. For example, the current release contains several module types wrapping tools from OpenNLP, but not coreference resolution. If a user wishes to use the OpenNLP coreference resolver, it is a relatively simple matter to define a custom module in their own source directory, using one of the existing wrappers as a model.

6 A worked example

An example of a full pipeline config file is shown in Fig. 2. This simple pipeline loads a corpus from a directory containing text files, each representing a single document. It applies tokenization to each document using the core document-map module that wraps spaCy's tokenizer.

Then it applies some custom processing to the tokens of each document, using a module type defined specifically for this pipeline and found in the accompanying source directory[7]. The resulting corpus is finally passed through the core vocabulary builder, which builds a vocabulary from all the words used in the corpus.

7 Software licenses

Pimlico itself is released under the GNU LGPLv3 license. However, it provides access to a large number of software packages, with a wide range of different licenses. Software dependencies are installed only when required, so use of Pimlico does not fall under the terms of all of these – only those required by the modules of the user's pipeline.

It can be important to know what licenses apply to all the code used by a pipeline. The Pimlico codebase keeps track of the licenses that apply to software that may be installed to support the use of the core module types. The command `licenses` produces a list of the licenses of all of the software used by a given pipeline, or alternatively just particular modules.

8 Related toolkits

Some proprietary tools exist for similar purposes to Pimlico[8]. However, the use of a proprietary tool to build a pipeline in itself precludes easy replication and extension by other authors, so we focus here on open source tools.

Two recently released examples of toolkits for building NLP pipelines are *Forte* and *PSI*. Forte (Liu et al., 2020) is constructed around similar concepts to Pimlico and it too provides wrappers around other NLP toolkits. PSI (Gralinski et al., 2012, Platform for Situated Intelligence) is similar in its goals and design to Forte. Pimlico's focus is on control of the execution of static pipelines to process large datasets and the management of the data as it passes through the pipeline. For these purposes, it provides a powerful set of tools not built into other toolkits. It does not provide facilities to run pipelines in a way that can be dynamically integrated into other systems. We see this as a distinct use case with different design requirements, one that is well catered for by toolkits like Forte and PSI.

Many other toolkits focus specifically on NLP tools, allowing models to be trained and applied

[6] https://opennlp.apache.org/

[7] The source code is not shown here, but the full example, including code, can be found in the documentation.

[8] For example, I2E, https://www.linguamatics.com/products/i2e.

for standard NLP tasks. Some provide their own structures for defining pipelines that chain multiple tasks (e.g., Qi et al., 2020; Manning et al., 2014; Honnibal and Montani, 2017). Pimlico provides a general framework for processing of large datasets, incorporating NLP tasks by providing wrappers around toolkits such as these. Unlike with these toolkits, data loading, pre- and post-processing can be handled in a single pipeline definition, requiring minimal (or no) code to be written.

Other general toolkits exist for building and running data-processing pipelines, such as Bonobo[9]. An alternative approach to developing Pimlico would have been to define a library of modules for NLP-specific tasks that could be used from such a toolkit. We chose instead to develop an infrastructure tuned to the type of corpus processing and data management that is typical in NLP experiments and tasks.

9 Conclusions

We have introduced the Pimlico toolkit for building pipelines for processing large corpora. We set out to address four key goals in improving the process of writing, running and distributing pipelines.

1. Pimlico provides **clear documentation** of pipelines in the form of a simple definition in a text file, containing pipeline structure and parameters for every step.

2. It is easy to incorporate **standard NLP tasks** using the core modules provided with the toolkit, for which only a definition of inputs and parameters is required. Among these are wrappers for commonly used NLP toolkits.

3. Integrating **custom code** into the pipeline is straightforward, by defining custom module types. An extensive array of factories, tools and templates means that typically only a small amount of code is required beyond the code to be executed.

4. The resulting pipeline definition and code can easily be **packaged and distributed**. Tools are provided to make the process of setting up the execution environment and installing software quick and simple. It is then possible to **extend or adjust** the pipeline by editing the conf file, or **apply to other datasets** by replacing input modules.

[9] https://www.bonobo-project.org/

The toolkit effectively addresses common problems encountered in using NLP tools to process large datasets, releasing code for experiments or other corpus processing for others to use, and running someone else's released code in a new environment or on new data. As such, we present it as a key contribution to free distribution of code to accompany NLP research and replicability of experiments.

9.1 Future work

Pimlico is under active development and new features are constantly being added. Several planned enhancements are worth noting in particular.

We plan to continue to expand the set of core modules to include wrappers around other NLP and machine learning toolkits. Many excellent new NLP toolkits have been released in recent years and have yet to be wrapped by core Pimlico modules, or have only partial wrappers. In many case, the addition of a wrapper is quick and requires only a small amount of code. Further commonly used preprocessing methods not currently covered by core modules, like Byte-Pair Encoding, would make pipeline development for modern NLP methods faster.

Pimlico includes a number of input readers for standard formats in which corpora are stored. However, many different formats are used for NLP corpora, often specific to one corpus. We plan to expand the set of core input reader modules, to allow more corpora to be read into a pipeline without requiring custom module code.

Modules currently assume that a corpus is a fixed unit, with a known size. Whilst this is often the case, there are exceptions. For example, if data is generated on the fly, a corpus could in effect have an infinite length. In future, it may be desirable to extend Pimlico's conception of a corpus to cover such cases.

Pipeline development and use could be helped by a visual tool to inspect pipeline structure and execution status. This could take the form of a tool to output images like those in the figures of this paper, or an interactive graphical interface as an alternative to the command-line interface.

We plan to add a system similar to the management of software dependencies for fetching pretrained models. For example, OpenNLP provides models for a number of languages for some of its components. Currently, the user must download

these models themselves in order to be able to run a module that uses them. The specification of which model to use, however, is part of the pipeline config. The new model management system would be able to download the models prior to running the module in question, just as software dependencies are downloaded and installed automatically.

We have chosen not to build into the toolkit any system for storing, fetching or managing input data. However, corpora are increasingly available online in standard repositories and formats, thanks to projects like Hugging Face[10]. Pipelines using such corpora could include a specification of where their input data can be retrieved from, such that it could be automatically downloaded as part of the execution process.

10 Acknowledgements

Pimlico has been developed to support work in a number of different projects. It has been supported by: European Commission FP7 framework grant 611560 (WHIM); the Academy of Finland grant 12933481 (Digital Language Typology). European Union Horizon 2020 research and innovation programme grants 770299 (NewsEye) and 825153 (EMBEDDIA).

References

Filip Gralinski, Krzysztof Jassem, and Marcin Junczys-Dowmunt. 2012. Psi-toolkit: A natural language processing pipeline. *Computational Linguistics*, 458:27–39.

Matthew Honnibal and Ines Montani. 2017. spaCy 2: Natural language understanding with Bloom embeddings, convolutional neural networks and incremental parsing. *To appear*.

Zhengzhong Liu, Avinash Bukkittu, Mansi Gupta, Pengzhi Gao, Swapnil Singhavi, Atif Ahmed, Wei Wei, Zecong Hu, Haoran Shi, Eric P. Xing, and Zhiting Hu. 2020. Forte: Composing Diverse NLP tools For Text Retrieval, Analysis and Generation.

Christopher D. Manning, Mihai Surdeanu, John Bauer, Jenny Finkel, Steven J. Bethard, and David McClosky. 2014. The Stanford CoreNLP natural language processing toolkit. In *Association for Computational Linguistics (ACL) System Demonstrations*, pages 55–60.

Tomas Mikolov, Edouard Grave, Piotr Bojanowski, Christian Puhrsch, and Armand Joulin. 2018. Advances in pre-training distributed word representations. In *Proceedings of the International Conference on Language Resources and Evaluation (LREC 2018)*.

Tomas Mikolov, Wen-tau Yih, and Geoffrey Zweig. 2013. Linguistic regularities in continuous space word representations. In *Proceedings of the 2013 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 746–751, Atlanta, Georgia. Association for Computational Linguistics.

Joakim Nivre, Johan Hall, and Jens Nilsson. 2006. Maltparser: A data-driven parser-generator for dependency parsing.

F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. 2011. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830.

Peng Qi, Yuhao Zhang, Yuhui Zhang, Jason Bolton, and Christopher D. Manning. 2020. Stanza: A Python natural language processing toolkit for many human languages. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics: System Demonstrations*.

Radim Řehůřek and Petr Sojka. 2010. Software Framework for Topic Modelling with Large Corpora. In *Proceedings of the LREC 2010 Workshop on New Challenges for NLP Frameworks*, pages 45–50, Valletta, Malta. ELRA. http://is.muni.cz/publication/884893/en.

[10] https://huggingface.co/datasets