Pimlico Documentation

Release 0.5

Mark Granroth-Wilding

October 07, 2016

Contents

The **Pimlico Processing Toolkit** (PIpelined Modular LInguistic COrpus processing) is a toolkit for building pipelines made up of linguistic processing tasks to run on large datasets (corpora). It provides a wrappers around many existing, widely used NLP (Natural Language Processing) tools.

It makes it easy to write large, potentially complex pipelines with the following key goals:

- to provide **clear documentation** of what has been done;
- to make it easy to **incorporate standard NLP tasks**,
- and to extend the code with **non-standard tasks, specific to a pipeline**;
- to support simple **distribution of code** for reproduction, for example, on other datasets.

The toolkit takes care of managing data between the steps of a pipeline and checking that everything's executed in the right order.

The core toolkit is written in Python. Pimlico is open source, released under the GPLv3 license. It is available from its Github repository. To get started with a Pimlico project, follow the getting-started guide.

More NLP tools will gradually be added. See my wishlist for current plans.

CHAPTER 1

Contents

1.1 Pimlico guides

Step-by-step guides through common tasks while using Pimlico.

1.1.1 Setting up a new project using Pimlico

You've decided to use Pimlico to implement a data processing pipeline. So, where do you start?

This guide steps through the basic setup of your project. You don't have to do everything exactly as suggested here, but it's a good starting point and follows Pimlico's recommended procedures. It steps through the setup for a very basic pipeline.

System-wide configuration

Pimlico needs you to specify certain parameters regarding your local system.

It needs to know where to put output files as it executes. Settings are given in a config file in your home directory and apply to all Pimlico pipelines you run. Note that Pimlico will make sure that different pipelines don't interfere with each other's output (provided you give them different names).

There are two locations you need to specify: **short-term** and **long-term storage**.

The **short-term store** should be on a disk that's as fast as possible to write to. For example, avoid using an NFS disk. It needs to be large enough to store output between pipeline stages, though you can easily move output from earlier stages into the long-term store as you go along.

The **long-term store** is where things are typically put at the end of a pipeline. It therefore doesn't need to be super-fast to access, but you may want it to be in a location that gets backed up, so you don't lose your valuable output.

For a simple setup, these could be just two subdirectories of the same directory. However, it can be useful to distinguish them.

Create a file *~/.pimlico* that looks like this:

```
long_term_store=/path/to/long-term/store
short_term_store=/path/to/short-term/store
```

Remember, these paths are not specific to a pipeline: all pipelines will use different subdirectories of these ones.

Getting started with Pimlico

The procedure for starting a new Pimlico project, using the latest release, is very simple.

Create a new, empty directory to put your project in. Download newproject.py into the project directory.

Choose a name for your project (e.g. *myproject*) and run:

```
python newproject.py myproject
```

This fetches the latest version of Pimlico (now in the *pimlico/* directory) and creates a basic config file template, which will define your pipeline.

It also retrieves some libraries that Pimlico needs to run. Other libraries required by specific pipeline modules will be installed as necessary when you use the modules.

Building the pipeline

You've now got a config file in *myproject.conf*. This already includes a *pipeline* section, which gives the basic pipeline setup. It will look something like this:

```
[pipeline]
name=myproject
release=<release number>
python_path=%(project_root)s/src/python
```

The *name* needs to be distinct from any other pipelines that you run – it's what distinguishes the storage locations.

release is the release of Pimlico that you're using: it's automatically set to the latest one, which has been downloaded.

If you later try running the same pipeline with an updated version of Pimlico, it will work fine as long as it's the same major version (the first digit). Otherwise, there may be backwards incompatible changes, so you'd need to update your config file, ensuring it plays nicely with the later Pimlico version.

Getting input

Now we add our first module to the pipeline. This reads input from XML files and iterates of <*doc*> tags to get documents. This is how the Gigaword corpus is stored, so if you have Gigaword, just set the path to point to it.

Todo

Use a dataset that everyone can get to in the example

```
[input-text]
type=pimlico.datatypes.XmlDocumentIterator
path=/path/to/data/dir
```

Perhaps your corpus is very large and you'd rather try out your pipeline on a small subset. In that case, add the following option:

```
truncate=1000
```

Note: For a neat way to define a small test version of your pipeline and keep its output separate from the main pipeline, see Pipeline variants.

Grouping files

The standard approach to storing data between modules in Pimlico is to group them together into batches of documents, storing each batch in a tar archive, containing a file for every document. This works nicely with large corpora, where having every document as a separate file would cause filesystem difficulties and having all documents in the same file would result in a frustratingly large file.

We can do the grouping on the fly as we read data from the input corpus. The *tar_filter* module groups documents together and subsequent modules will all use the same grouping to store their output, making it easy to align the datasets they produce.

```
[tar-grouper]
type=pimlico.modules.corpora.tar_filter
input=input-text
```

Doing something: tokenization

Now, some actual linguistic processing, albeit somewhat uninteresting. Many NLP tools assume that their input has been divided into sentences and tokenized. The OpenNLP-based tokenization module does both of these things at once, calling OpenNLP tools.

Notice that the output from the previous module feeds into the input for this one, which we specify simply by naming the module.

```
[tokenize]
type=pimlico.modules.opennlp.tokenize
input=tar-grouper
```

Doing something more interesting: POS tagging

Many NLP tools rely on part-of-speech (POS) tagging. Again, we use OpenNLP, and a standard Pimlico module wraps the OpenNLP tool.

```
[pos-tag]
type=pimlico.modules.opennlp.pos
input=tokenize
```

Running Pimlico

Now we've got our basic config file ready to go. It's a simple linear pipeline that goes like this:

> read input docs -> group into batches -> tokenize -> POS tag

Before we can run it, there's one thing missing: three of these modules have their own dependencies, so we need to get hold of the libraries they use. The input reader uses the Beautiful Soup python library and the tokenization and POS tagging modules use OpenNLP.

Fetching dependencies

All the standard modules provide easy ways to get hold of their dependencies automatically, or as close as possible. Most of the time, all you need to do is tell Pimlico to install them.

You can use the *check* command, with a module name, to check whether a module is ready to run.

```
./pimlico.sh myproject.conf check tokenize
```

In this case, it will tell you that some libraries are missing, but they can be installed automatically. Simply issue the *install* command for the module.

```
./pimlico.sh myproject.conf install tokenize
```

Simple as that.

There's one more thing to do: the tools we're using require statistical models. We can simply download the pre-trained English models from the OpenNLP website.

At present, Pimlico doesn't yet provide a built-in way for the modules to do this, as it does with software libraries, but it does include a GNU Makefile to make it easy to do:

```
cd ~/myproject/pimlico/models
make opennlp
```

Note that the modules we're using default to these standard, pre-trained models, which you're now in a position to use. However, if you want to use different models, e.g. for other languages or domains, you can specify them using extra options in the module definition in your config file.

Checking everything's dandy

Now you can run the *check* command to check that the modules are ready to run. To check the whole pipeline's dependencies, run:

```
./pimlico.sh myproject.conf check all
```

With any luck, all the checks will be successful. If not, you'll need to address any problems with dependencies before going any further.

Running the pipeline

What modules to run?

Pimlico can now suggest an order in which to run your modules. In our case, this is pretty obvious, seeing as our pipeline is entirely linear – it's clear which ones need to be run before others.

```
./pimlico.sh myproject.conf status
```

The output also tells you the current status of each module. At the moment, all the modules are *UNEXECUTED*.

You'll notice that the *tar-grouper* module doesn't feature in the list. This is because it's a filter – it's run on the fly while reading output from the previous module (i.e. the input), so doesn't have anything to run itself.

You might be surprised to see that *input-text does* feature in the list. This is because, although it just reads the data out of a corpus on disk, there's not quite enough information in the corpus, so we need to run the module to collect a little bit of metadata from an initial pass over the corpus. Some input types need this, others not. In this case, all we're lacking is a count of the total number of documents in the corpus.

Running the modules

The modules can be run using the *run* command and specifying the module by name. We do this manually for each module.

```
./pimlico.sh myproject.conf run input-text
./pimlico.sh myproject.conf run tokenize
./pimlico.sh myproject.conf run pos-tag
```

Adding custom modules

Most likely, for your project you need to do some processing not covered by the built-in Pimlico modules. At this point, you can start implementing your own modules, which you can distribute along with the config file so that people can replicate what you did.

The *newproject.py* script has already created a directory where our custom source code will live: *src/python*, with some subdirectories according to the standard code layout, with module types and datatypes in separate packages.

The template pipeline also already has an option *python_path* pointing to this directory, so that Pimlico knows where to find your code. Note that the code's in a subdirectory of that containing the pipeline config and we specify the custom code path relative to the config file, so it's easy to distribute the two together.

Now you can create Python modules or packages in *src/python*, following the same conventions as the built-in modules and overriding the standard base classes, as they do. The following articles tell you more about how to do this:

- Writing Pimlico modules
- Writing document map modules
- Pimlico module structure

Your custom modules and datatypes can then simply be used in the config file as module types.

1.1.2 Writing Pimlico modules

Pimlico comes with a fairly large number of *module types* that you can use to run many standard NLP, data processing and ML tools over your datasets.

For some projects, this is all you need to do. However, often you'll want to mix standard tools with your own code, for example, using the output from the tools. And, of course, there are many more tools you might want to run that aren't built into Pimlico: you can still benefit from Pimlico's framework for data handling, config files and so on.

For a detailed description of the structure of a Pimlico module, see Pimlico module structure. This guide takes you through building a simple module.

Note: In any case where a module will process a corpus one document at a time, you should write a document map module, which takes care of a lot of things for you, so you only need to say what to do with each document.

Code layout

If you've followed the basic project setup guide, you'll have a project with a directory structure like this:

```
myproject/
    pipeline.conf
    pimlico/
        bin/
        lib/
        src/
        ...
```

```
    src/
        python/
```

If you've not already created the `src/python` directory, do that now.

This is where your custom Python code will live. You can put all of your custom module types and datatypes in there and use them in the same way as you use the Pimlico core modules and datatypes.

Add this option to the `[pipeline]` section of your config file, so Pimlico knows where to find your code:

```
python_path=src/python
```

To follow the conventions used in Pimlico's codebase, we'll create the following package structure in `src/python`:

```
src/python/myproject/
    __init__.py
    modules/
        __init__.py
    datatypes/
        __init__.py
```

Write a module

A Pimlico module consists of a Python package with a special layout. Every module has a file `info.py`. This contains the definition of the module's metadata: its inputs, outputs, options, etc.

Most modules also have a file `execute.py`, which defines the routine that's called when it's run. You should take care when writing `info.py` not to import any non-standard Python libraries or have any time-consuming operations that get run when it gets imported.

`execute.py`, on the other hand, will only get imported when the module is to be run, after dependency checks.

For the example below, let's assume we're writing a module called `nmf` and create the following directory structure for it:

```
src/python/myproject/modules/
    __init__.py
    nmf/
        __init__.py
        info.py
        execute.py
```

Metadata

Module metadata (everything apart from what happens when it's actually run) is defined in `info.py` as a class called `ModuleInfo`.

Here's a sample basic `ModuleInfo`, which we'll step through. (It's based on the Scikit-learn *`matrix_factorization`* module.)

```
import json
from pimlico.core.modules.base import BaseModuleInfo
from pimlico.datatypes.arrays import ScipySparseMatrix, NumpyArray

class ModuleInfo(BaseModuleInfo):
    module_type_name = "nmf"
    module_readable_name = "Sklearn non-negative matrix factorization"
    module_inputs = [("matrix", ScipySparseMatrix)]
```

```
    module_outputs = [("w", NumpyArray), ("h", NumpyArray)]
    module_options = {
        "components": {
            "help": "Number of components to use for hidden representation",
            "type": int,
            "default": 200,
        },
    }

    def get_software_dependencies(self):
        return super(ModuleInfo, self).get_software_dependencies() + \
               [PythonPackageOnPip("sklearn", "Scikit-learn")]
```

The `ModuleInfo` should always be a subclass of `BaseModuleInfo`. There are some subclasses that you might want to use instead (e.g., see Writing document map modules), but here we just use the basic one.

Certain class-level attributes should pretty much always be overridden:

- `module_type_name`: A name used to identify the module internally
- `module_readable_name`: A human-readable short description of the module
- `module_inputs`: Most modules need to take input from another module (though not all)
- `module_outputs`: Describes the outputs that the module will produce, which may then be used as inputs to another module

Inputs are given as pairs `(name, type)`, where `name` is a short name to identify the input and `type` is the datatype that the input is expected to have. Here, and most commonly, this is a subclass of `PimlicoDatatype` and Pimlico will check that a dataset supplied for this input is either of this type, or has a type that is a subclass of this.

Here we take just a single input: a sparse matrix.

Outputs are given in a similar way. It is up to the module's executor (see below) to ensure that these outputs get written, but here we describe the datatypes that will be produced, so that we can use them as input to other modules.

Here we produce two Numpy arrays, the factorization of the input matrix.

Dependencies: Since we require Scikit-learn to execute this module, we override `get_software_dependencies()` to specify this. As Scikit-learn is available through Pip, this is very easy: all we need to do is specify the Pip package name. Pimlico will check that Scikit-learn is installed before executing the module and, if not, allow it to be installed automatically.

Finally, we also define some **options**. The values for these can be specified in the pipeline config file. When the `ModuleInfo` is instantiated, the processed options will be available in its `options` attribute. So, for example, we can get the number of components (specified in the config file, or the default of 200) using `info.options["components"]`.

Executor

Here is a sample executor for the module info given above, placed in the file `execute.py`.

```
from pimlico.core.modules.base import BaseModuleExecutor
from pimlico.datatypes.arrays import NumpyArrayWriter
from sklearn.decomposition import NMF

class ModuleExecutor(BaseModuleExecutor):
    def execute(self):
        input_matrix = self.info.get_input("matrix").array
        self.log.info("Loaded input matrix: %s" % str(input_matrix.shape))
```

```
        # Convert input matrix to CSR
        input_matrix = input_matrix.tocsr()
        # Initialize the transformation
        components = self.info.options["components"]
        self.log.info("Initializing NMF with %d components" % components)
        nmf = NMF(components)

        # Apply transformation to the matrix
        self.log.info("Fitting NMF transformation on input matrix" % transform_type)
        transformed_matrix = transformer.fit_transform(input_matrix)

        self.log.info("Fitting complete: storing H and W matrices")
        # Use built-in Numpy array writers to output results in an appropriate format
        with NumpyArrayWriter(self.info.get_absolute_output_dir("w")) as w_writer:
            w_writer.set_array(transformed_matrix)
        with NumpyArrayWriter(self.info.get_absolute_output_dir("h")) as h_writer:
            h_writer.set_array(transformer.components_)
```

The executor is always defined as a class in `execute.py` called `ModuleExecutor`. It should always be a subclass of `BaseModuleExecutor` (though, again, note that there are more specific subclasses and class factories that we might want to use in other circumstances).

The `execute()` method defines what happens when the module is executed.

The instance of the module's `ModuleInfo`, complete with **options** from the pipeline config, is available as `self.info`. A standard Python **logger** is also available, as `self.log`, and should be used to keep the user updated on what's going on.

Getting hold of the **input data** is done through the module info's `get_input()` method. In the case of a Scipy matrix, here, it just provides us with the matrix as an attribute.

Then we do whatever our module is designed to do. At the end, we write the output data to the appropriate output directory. This should always be obtained using the `get_absolute_output_dir()` method of the module info, since Pimlico takes care of the exact location for you.

Most Pimlico datatypes provide a corresponding **writer**, ensuring that the output is written in the correct format for it to be read by the datatype's reader. When we leave the `with` block, in which we give the writer the data it needs, this output is written to disk.

Pipeline config

Our module is now ready to use and we can refer to it in a pipeline config file. We'll assume we've prepared a suitable Scipy sparse matrix earlier in the pipeline, available as the default output of a module called `matrix`. Then we can add section like this to use our new module:

```
[matrix]
...(Produces sparse matrix output)...

[factorize]
type=myproject.modules.nmf
components=300
input=matrix
```

Note that, since there's only one input, we don't need to give its name. If we had defined multiple inputs, we'd need to specify this one as `input_matrix=matrix`.

You can now run the module as part of your pipeline in the usual ways.

Skeleton new module

To make developing a new module a little quicker, here's a skeleton module info and executor.

```
from pimlico.core.modules.base import BaseModuleInfo

class ModuleInfo(BaseModuleInfo):
    module_type_name = "NAME"
    module_readable_name = "READABLE NAME"
    module_inputs = [("NAME", REQUIRED_TYPE)]
    module_outputs = [("NAME", PRODUCED_TYPE)]
    # Delete module_options if you don't need any
    module_options = {
        "OPTION_NAME": {
            "help": "DESCRIPTION",
            "type": TYPE,
            "default": VALUE,
        },
    }

    def get_software_dependencies(self):
        return super(ModuleInfo, self).get_software_dependencies() + [
            # Add your own dependencies to this list
            # Remove this method if you don't need to add any
        ]
```

```
from pimlico.core.modules.base import BaseModuleExecutor

class ModuleExecutor(BaseModuleExecutor):
    def execute(self):
        input_data = self.info.get_input("NAME")
        self.log.info("MESSAGES")

        # DO STUFF

        with SOME_WRITER(self.info.get_absolute_output_dir("NAME")) as writer:
            # Do what the writer requires
```

1.1.3 Writing document map modules

Todo

Write a guide to building document map modules

Skeleton new module

To make developing a new module a little quicker, here's a skeleton module info and executor for a document map module. It follows the most common method for defining the executor, which is to use the multiprocessing-based executor factory.

```
from pimlico.core.modules.map import DocumentMapModuleInfo
from pimlico.datatypes.tar import TarredCorpusType

class ModuleInfo(DocumentMapModuleInfo):
```

```
    module_type_name = "NAME"
    module_readable_name = "READABLE NAME"
    module_inputs = [("NAME", TarredCorpusType(DOCUMENT_TYPE))]
    module_outputs = [("NAME", PRODUCED_TYPE)]
    module_options = {
        "OPTION_NAME": {
            "help": "DESCRIPTION",
            "type": TYPE,
            "default": VALUE,
        },
    }

    def get_software_dependencies(self):
        return super(ModuleInfo, self).get_software_dependencies() + [
            # Add your own dependencies to this list
        ]

    def get_writer(self, output_name, output_dir, append=False):
        if output_name == "NAME":
            # Instantiate a writer for this output, using the given output dir
            # and passing append in as a kwarg
            return WRITER_CLASS(output_dir, append=append)
```

A bare-bones executor:

```
from pimlico.core.modules.map.multiproc import multiprocessing_executor_factory


def process_document(worker, archive_name, doc_name, *data):
    # Do something to process the document...

    # Return an object to send to the writer
    return output


ModuleExecutor = multiprocessing_executor_factory(process_document)
```

Or getting slightly more sophisticated:

```
from pimlico.core.modules.map.multiproc import multiprocessing_executor_factory


def process_document(worker, archive_name, doc_name, *data):
    # Do something to process the document

    # Return a tuple of objects to send to each writer
    # If you only defined a single output, you can just return a single object
    return output1, output2, ...


# You don't have to, but you can also define pre- and postprocessing
# both at the executor level and worker level

def preprocess(executor):
    pass


def postprocess(executor, error=None):
    pass
```

```
def set_up_worker(worker):
    pass


def tear_down_worker(worker, error=None):
    pass


ModuleExecutor = multiprocessing_executor_factory(
    process_document,
    preprocess_fn=preprocess, postprocess_fn=postprocess,
    worker_set_up_fn=set_up_worker, worker_tear_down_fn=tear_down_worker,
)
```

1.2 Core docs

A set of articles on the core aspects and features of Pimlico.

1.2.1 Downloading Pimlico

To start a new project using Pimlico, download the newproject.py script. It will create a template pipeline config file to get you started and download the latest version of Pimlico to accompany it.

See Setting up a new project using Pimlico for more detail.

Pimlico's source code is available on on Github.

Manual setup

If for some reason you don't want to use the *newproject.py* script, you can set up a project yourself. Download Pimlico from Github.

Simply download the whole source code as a *.zip* or *.tar.gz* file and uncompress it. This will produce a directory called *pimlico*, followed by a long incomprehensible string, which you can rename simply *pimlico*.

Pimlico has a few basic dependencies, but these will be automatically downloaded the first time you load it.

1.2.2 Pipeline config

A Pimlico pipeline, as read from a config file (`pimlico.core.config.PipelineConfig`) contains all the information about the pipeline being processed and provides access to specific modules in it.

Todo

Write full documentation for this

1.2.3 Pipeline variants

Todo

Document variants

1.2.4 Pimlico module structure

This document describes the code structure for Pimlico module types in full.

For a basic guide to writing your own modules, see Writing Pimlico modules.

Todo

Write documentation for this

1.2.5 Module dependencies

Todo

Write something about how dependencies are fetched

Note: Pimlico now has a really neat way of checking for dependencies and, in many cases, fetching the automatically. It's rather new, so I've not written this guide yet. Ignore any old Makefiles: they ought to have all been replaced by SoftwareDependency classes now

1.3 Core Pimlico modules

Pimlico comes with a substantial collection of module types that provide wrappers around existing NLP and machine learning tools.

1.3.1 CAEVO event extractor

Path	pimlico.modules.caevo
Executable	yes

CAEVO is Nate Chambers' CAscading EVent Ordering system, a tool for extracting events of many types from text and ordering them.

CAEVO is open source, implemented in Java, so is easily integrated into Pimlico using Py4J.

Inputs

Name	Type(s)
documents	TarredCorpus<RawTextDocumentType>

Outputs

Name	Type(s)
events	`CaevoCorpus`

Options

Name	Description	Type
sieves	Filename of sieve list file, or path to the file. If just a filename, assumed to be in Caevo model dir (models/caevo). Default: default.sieves (supplied with Caevo)	string

1.3.2 C&C parser

Path	pimlico.modules.candc
Executable	yes

Wrapper around the original C&C parser.

Takes tokenized input and parses it with C&C. The output is written exactly as it comes out from C&C. It contains both GRs and supertags, plus POS-tags, etc.

The wrapper uses C&C's SOAP server. It sets the SOAP server running in the background and then calls C&C's SOAP client for each document. If parallelizing, multiple SOAP servers are set going and each one is kept constantly fed with documents.

Inputs

Name	Type(s)
documents	TarredCorpus<TokenizedDocumentType>

Outputs

Name	Type(s)
parsed	`CandcOutputCorpus`

Options

Name	Description	Type
model	Absolute path to models directory or name of model set. If not an absolute path, assumed to be a subdirectory of the candcs models dir (see instructions in models/candc/README on how to fetch pre-trained models)	string

1.3.3 Stanford CoreNLP

Path	pimlico.modules.corenlp
Executable	yes

Process documents one at a time with the Stanford CoreNLP toolkit. CoreNLP provides a large number of NLP tools, including a POS-tagger, various parsers, named-entity recognition and coreference resolution. Most of these tools can be run using this module.

The module uses the CoreNLP server to accept many inputs without the overhead of loading models. If parallelizing, only a single CoreNLP server is run, since this is designed to set multiple Java threads running if it receives multiple queries at the same time. Multiple Python processes send queries to the server and process the output.

The module has no non-optional outputs, since what sort of output is available depends on the options you pass in: that is, on which tools are run. Use the annotations option to choose which word annotations are added. Otherwise, simply select the outputs that you want and the necessary tools will be run in the CoreNLP pipeline to produce those outputs.

Currently, the module only accepts tokenized input. If pre-POS-tagged input is given, for example, the POS tags won't be handed into CoreNLP. In the future, this will be implemented.

We also don't currently provide a way of choosing models other than the standard, pre-trained English models. This is a small addition that will be implemented in the future.

Inputs

Name	Type(s)
documents	TarredCorpus<WordAnnotationsDocumentType|TokenizedDocumentType|RawTextDocumentType>

Outputs

No non-optional outputs

Optional

Name	Type(s)
annotations	`AnnotationFieldsFromOptions`
tokenized	`TokenizedCorpus`
parse	`ConstituencyParseTreeCorpus`
parse-deps	`StanfordDependencyParseCorpus`
dep-parse	`StanfordDependencyParseCorpus`
raw	`JsonDocumentCorpus`
coref	`CorefCorpus`

Options

Name	Description	Type
gzip	If True, each output, except annotations, for each document is gzipped. This can help reduce the storage occupied by e.g. parser or coref output. Default: False	bool
timeout	Timeout for the CoreNLP server, which is applied to every job (document). Number of seconds. By default, we use the server's default timeout (15 secs), but you may want to increase this for more intensive tasks, like coref	float
readable	If True, JSON outputs are formatted in a readable fashion, pretty printed. Otherwise, they're as compact as possible. Default: False	bool
annotators	Comma-separated list of word annotations to add, from CoreNLP's annotators. Choose from: word, pos, lemma, ner	string
dep_type	Type of dependency parse to output, when outputting dependency parses, either from a constituency parse or direct dependency parse. Choose from the three types allowed by CoreNLP: 'basic', 'collapsed' or 'collapsed-ccprocessed'	'basic', 'collapsed' or 'collapsed-ccprocessed'

1.3.4 Corpus-reading

Base modules for reading input from textual corpora.

Human-readable formatting

Path	pimlico.modules.corpora.format
Executable	yes

Corpus formatter

Pimlico provides a data browser to make it easy to view documents in a tarred document corpus. Some datatypes provide a way to format the data for display in the browser, whilst others provide multiple formatters that display the data in different ways.

This module allows you to use this formatting functionality to output the formatted data as a corpus. Since the formatting operations are designed for display, this is generally only useful to output the data for human consumption.

Inputs

Name	Type(s)
corpus	`TarredCorpus`

Outputs

Name	Type(s)
formatted	`TarredCorpus`

Options

Name	Description	Type
formatter	Fully qualified class name of a formatter to use to format the data. If not specified, the default formatter is used, which uses the datatype's browser_display attribute if available, or falls back to just converting documents to unicode	string

Corpus document list filter

Path	pimlico.modules.corpora.list_filter
Executable	yes

Similar to :mod:pimlico.modules.corpora.split, but instead of taking a random split of the dataset, splits it according to a given list of documents, putting those in the list in one set and the rest in another.

Inputs

Name	Type(s)
corpus	`TarredCorpus`
list	`StringList`

Outputs

Name	Type(s)
set1	`same as input corpus`
set2	`same as input corpus`

Corpus subset

Path	pimlico.modules.corpora.split
Executable	yes

Split a tarred corpus into two subsets. Useful for dividing a dataset into training and test subsets. The output datasets have the same type as the input. The documents to put in each set are selected randomly. Running the module multiple times will give different splits.

Note that you can use this multiple times successively to split more than two ways. For example, say you wanted a training set with 80% of your data, a dev set with 10% and a test set with 10%, split it first into training and non-training 80-20, then split the non-training 50-50 into dev and test.

The module also outputs a list of the document names that were included in the first set. Optionally, it outputs the same thing for the second input too. Note that you might prefer to only store this list for the smaller set: e.g. in a training-test split, store only the test document list, as the training list will be much larger. In such a case, just put the smaller set first and don't request the optional output *doc_list2*.

Inputs

Name	Type(s)
corpus	`TarredCorpus`

Outputs

Name	Type(s)
set1	`same as input corpus`
set2	`same as input corpus`
doc_list1	`StringList`

Optional

Name	Type(s)
doc_list2	`StringList`

Options

Name	Description	Type
set1_size	Proportion of the corpus to put in the first set, float between 0.0 and 1.0. Default: 0.2	float

Corpus subset

Path	pimlico.modules.corpora.subset
Executable	no

Simple filter to truncate a dataset after a given number of documents, potentially offsetting by a number of documents. Mainly useful for creating small subsets of a corpus for testing a pipeline before running on the full corpus.

This is a filter module. It is not executable, so won't appear in a pipeline's list of modules that can be run. It produces its output for the next module on the fly when the next module needs it.

Inputs

Name	Type(s)
documents	`IterableCorpus`

Outputs

Name	Type(s)
documents	`same as input corpus`

Options

Name	Description	Type
offset	Number of documents to skip at the beginning of the corpus (default: 0, start at beginning)	int
size	(required)	int

Tar archive grouper

Path	pimlico.modules.corpora.tar
Executable	yes

Group the files of a multi-file iterable corpus into tar archives. This is a standard thing to do at the start of the pipeline, since it's a handy way to store many (potentially small) files without running into filesystem problems.

The files are simply grouped linearly into a series of tar archives such that each (apart from the last) contains the given number.

After grouping documents in this way, document map modules can be called on the corpus and the grouping will be preserved as the corpus passes through the pipeline.

Inputs

Name	Type(s)
documents	`IterableCorpus`

Outputs

Name	Type(s)
documents	`TarredCorpus`

Options

Name	Description	Type
archive_size	Number of documents to include in each archive (default: 1k)	string
archive_basename	Base name to use for archive tar files. The archive number is appended to this. (Default: 'archive')	string

Tar archive grouper (filter)

Path	pimlico.modules.corpora.tar_filter
Executable	no

Like *tar*, but doesn't write the archives to disk. Instead simulates the behaviour of tar but as a filter, grouping files on the fly and passing them through with an archive name

This is a filter module. It is not executable, so won't appear in a pipeline's list of modules that can be run. It produces its output for the next module on the fly when the next module needs it.

Inputs

Name	Type(s)
documents	`IterableCorpus`

Outputs

Name	Type(s)
documents	`tarred corpus with input doc type`

Options

Name	Description	Type
archive_size	Number of documents to include in each archive (default: 1k)	string
archive_basename	Base name to use for archive tar files. The archive number is appended to this. (Default: 'archive')	string

Corpus vocab builder

Path	pimlico.modules.corpora.vocab_builder
Executable	yes

Builds a dictionary (or vocabulary) for a tokenized corpus. This is a data structure that assigns an integer ID to every distinct word seen in the corpus, optionally applying thresholds so that some words are left out.

Similar to *pimlico.modules.features.vocab_builder*, which builds two vocabs, one for terms and one for features.

Inputs

Name	Type(s)
text	TarredCorpus<TokenizedDocumentType>

Outputs

Name	Type(s)
vocab	`Dictionary`

Options

Name	Description	Type
threshold	Minimum number of occurrences required of a term to be included	int
max_prop	Include terms that occur in max this proportion of documents	float
include	Ensure that certain words are always included in the vocabulary, even if they don't make it past the various filters, or are never seen in the corpus. Give as a comma-separated list	comma-separated list of string
limit	Limit vocab size to this number of most common entries (after other filters)	int

Tokenized corpus to ID mapper

Path	pimlico.modules.corpora.vocab_mapper
Executable	yes

Inputs

Name	Type(s)
text	TarredCorpus<TokenizedDocumentType>
vocab	`Dictionary`

Outputs

Name	Type(s)
ids	`IntegerListsDocumentCorpus`

1.3.5 Embedding feature extractors and trainers

Modules for extracting features from which to learn word embeddings from corpora, and for training embeddings.

Some of these don't actually learn the embeddings, they just produce features which can then be fed into an embedding learning module, such as a form of matrix factorization. Note that you can train embeddings not only using the trainers here, but also using generic matrix manipulation techniques, for example the factorization methods provided by sklearn.

Dependency feature extractor for embeddings

Path	pimlico.modules.embeddings.dependencies
Executable	yes

Todo

Document this module

Inputs

Name	Type(s)
dependencies	TarredCorpus<CoNLLDependencyParseDocumentType>

Outputs

Name	Type(s)
term_features	`TermFeatureListCorpus`

Options

Name	Description	Type
lemma	Use lemmas as terms instead of the word form. Note that if you didn't run a lemmatizer before dependency parsing the lemmas are probably actually just copies of the word forms	bool
condense_prep	Where a word is modified ...TODO	string
term_pos	Only extract features for terms whose POSs are in this comma-separated list. Put a * at the end to denote POS prefixes	comma-separated list of string
skip_types	Dependency relations to skip, separated by commas	comma-separated list of string

Word2vec embedding trainer

Path	pimlico.modules.embeddings.word2vec
Executable	yes

Word2vec embedding learning algorithm, using Gensim's implementation.

Find out more about word2vec.

This module is simply a wrapper to call Gensim's Python (+C) implementation of word2vec on a Pimlico corpus.

Inputs

Name	Type(s)
text	TarredCorpus<TokenizedDocumentType>

Outputs

Name	Type(s)
model	`Word2VecModel`

Options

Name	Description	Type
iters	number of iterations over the data to perform. Default: 5	int
min_count	word2vec's min_count option: prunes the dictionary of words that appear fewer than this number of times in the corpus. Default: 5	int
negative_samples	number of negative samples to include per positive. Default: 5	int
size	number of dimensions in learned vectors. Default: 200	int

1.3.6 Feature set processing

Various tools for generic processing of extracted sets of features: building vocabularies, mapping to integer indices, etc.

Key-value to term-feature converter

Path	pimlico.modules.features.term_feature_compiler
Executable	yes

Todo

Document this module

Inputs

Name	Type(s)
key_values	TarredCorpus<KeyValueListDocumentType>

Outputs

Name	Type(s)
term_features	`TermFeatureListCorpus`

Options

Name	Description	Type
term_keys	Name of keys (feature names in the input) which denote terms. The first one found in the keys of a particular data point will be used as the term for that data point. Any other matches will be removed before using the remaining keys as the data point's features. Default: just 'term'	comma-separated list of string
include_feature_keys	If True, include the key together with the value from the input key-value pairs as feature names in the output. Otherwise, just use the value. E.g. for input [prop=wordy, poss=my], if True we get features [prop_wordy, poss_my] (both with count 1); if False we get just [wordy, my]. Default: False	bool

Term-feature matrix builder

Path	pimlico.modules.features.term_feature_matrix_builder
Executable	yes

Todo

Document this module

Inputs

Name	Type(s)
data	`IndexedTermFeatureListCorpus`

Outputs

Name	Type(s)
matrix	`ScipySparseMatrix`

Term-feature corpus vocab builder

Path	pimlico.modules.features.vocab_builder
Executable	yes

Todo

Document this module

Inputs

Name	Type(s)
term_features	TarredCorpus<TermFeatureListDocumentType>

Outputs

Name	Type(s)
term_vocab	`Dictionary`
feature_vocab	`Dictionary`

Options

Name	Description	Type
feature_limit	Limit vocab size to this number of most common entries (after other filters)	int
feature_max_prop	Include features that occur in max this proportion of documents	float
term_max_prop	Include terms that occur in max this proportion of documents	float
term_threshold	Minimum number of occurrences required of a term to be included	int
feature_threshold	Minimum number of occurrences required of a feature to be included	int
term_limit	Limit vocab size to this number of most common entries (after other filters)	int

Term-feature corpus vocab mapper

Path	pimlico.modules.features.vocab_mapper
Executable	yes

Todo

Document this module

Inputs

Name	Type(s)
data	TarredCorpus<TermFeatureListDocumentType>
term_vocab	`Dictionary`
feature_vocab	`Dictionary`

Outputs

Name	Type(s)
data	`IndexedTermFeatureListCorpus`

1.3.7 Malt dependency parser

Wrapper around the Malt dependency parser and data format converters to support connections to other modules.

Annotated text to CoNLL dep parse input converter

Path	pimlico.modules.malt.conll_parser_input
Executable	yes

Converts word-annotations to CoNLL format, ready for input into the Malt parser. Annotations must contain words and POS tags. If they contain lemmas, all the better; otherwise the word will be repeated as the lemma.

Inputs

Name	Type(s)
annotations	`WordAnnotationCorpus` with 'word' and 'pos' fields

Outputs

Name	Type(s)
conll_data	`CoNLLDependencyParseInputCorpus`

Malt dependency parser

Path	pimlico.modules.malt.parse
Executable	yes

Todo

Document this module

Todo

Replace check_runtime_dependencies() with get_software_dependencies()

Inputs

Name	Type(s)
documents	TarredCorpus<CoNLLDependencyParseDocumentType>

Outputs

Name	Type(s)
parsed	`CoNLLDependencyParseCorpus`

Options

Name	Description	Type
model	Filename of parsing model, or path to the file. If just a filename, assumed to be Malt models dir (models/malt). Default: engmalt.linear-1.7.mco, which can be acquired by 'make malt' in the models dir	string
no_gzip	By default, we gzip each document in the output data. If you don't do this, the output can get very large, since it's quite a verbose output format	bool

1.3.8 OpenNLP modules

A collection of module types to wrap individual OpenNLP tools.

OpenNLP coreference resolution

Path	pimlico.modules.opennlp.coreference
Executable	yes

Todo

Document this module

Todo

Replace check_runtime_dependencies() with get_software_dependencies()

Use local config setting opennlp_memory to set the limit on Java heap memory for the OpenNLP processes. If parallelizing, this limit is shared between the processes. That is, each OpenNLP worker will have a memory limit of *opennlp_memory / processes*. That setting can use *g*, *G*, *m*, *M*, *k* and *K*, as in the Java setting.

Inputs

Name	Type(s)
parses	TarredCorpus<TreeStringsDocumentType>

Outputs

Name	Type(s)
coref	`CorefCorpus`

Options

Name	Description	Type
gzip	If True, each output, except annotations, for each document is gzipped. This can help reduce the storage occupied by e.g. parser or coref output. Default: False	bool
model	Coreference resolution model, full path or directory name. If a filename is given, it is expected to be in the OpenNLP model directory (models/opennlp/). Default: '' (standard English opennlp model in models/opennlp/)	string
readable	If True, pretty-print the JSON output, so it's human-readable. Default: False	bool
timeout	Timeout in seconds for each individual coref resolution task. If this is exceeded, an InvalidDocument is returned for that document	int

OpenNLP coreference resolution

Path	pimlico.modules.opennlp.coreference_pipeline
Executable	yes

Runs the full coreference resolution pipeline using OpenNLP. This includes sentence splitting, tokenization, pos tagging, parsing and coreference resolution. The results of all the stages are available in the output.

Use local config setting opennlp_memory to set the limit on Java heap memory for the OpenNLP processes. If parallelizing, this limit is shared between the processes. That is, each OpenNLP worker will have a memory limit of *opennlp_memory / processes*. That setting can use *g*, *G*, *m*, *M*, *k* and *K*, as in the Java setting.

Inputs

Name	Type(s)
text	TarredCorpus<RawTextDocumentType>

Outputs

Name	Type(s)
coref	`CorefCorpus`

Optional

Name	Type(s)
tokenized	`TokenizedCorpus`
pos	`WordAnnotationCorpusWithPos`
parse	`ConstituencyParseTreeCorpus`

Options

Name	Description	Type
gzip	If True, each output, except annotations, for each document is gzipped. This can help reduce the storage occupied by e.g. parser or coref output. Default: False	bool
token_model	Tokenization model. Specify a full path, or just a filename. If a filename is given it is expected to be in the opennlp model directory (models/opennlp/)	string
parse_model	Parser model, full path or directory name. If a filename is given, it is expected to be in the OpenNLP model directory (models/opennlp/)	string
timeout	Timeout in seconds for each individual coref resolution task. If this is exceeded, an InvalidDocument is returned for that document	int
coref_model	Coreference resolution model, full path or directory name. If a filename is given, it is expected to be in the OpenNLP model directory (models/opennlp/). Default: '' (standard English opennlp model in models/opennlp/)	string
readable	If True, pretty-print the JSON output, so it's human-readable. Default: False	bool
pos_model	POS tagger model, full path or filename. If a filename is given, it is expected to be in the opennlp model directory (models/opennlp/)	string
sentence_model	Sentence segmentation model. Specify a full path, or just a filename. If a filename is given it is expected to be in the opennlp model directory (models/opennlp/)	string

OpenNLP NER

Path	pimlico.modules.opennlp.ner
Executable	yes

Named-entity recognition using OpenNLP's tools.

By default, uses the pre-trained English model distributed with OpenNLP. If you want to use other models (e.g. for other languages), download them from the OpenNLP website to the models dir (*models/opennlp*) and specify the model name as an option.

Note that the default model is for identifying person names only. You can identify other name types by loading other pre-trained OpenNLP NER models. Identification of multiple name types at the same time is not (yet) implemented.

Inputs

Name	Type(s)
text	TarredCorpus<TokenizedDocumentType|WordAnnotationsDocumentType>

Outputs

Name	Type(s)
documents	`SentenceSpansCorpus`

Options

Name	Description	Type
model	NER model, full path or filename. If a filename is given, it is expected to be in the opennlp model directory (models/opennlp/)	string

OpenNLP constituency parser

Path	pimlico.modules.opennlp.parse
Executable	yes

Todo

Document this module

Inputs

Name	Type(s)
documents	TarredCorpus<TokenizedDocumentType> or `WordAnnotationCorpus` with 'word' field

Outputs

Name	Type(s)
parser	`ConstituencyParseTreeCorpus`

Options

Name	Description	Type
model	Parser model, full path or directory name. If a filename is given, it is expected to be in the OpenNLP model directory (models/opennlp/)	string

OpenNLP POS-tagger

Path	pimlico.modules.opennlp.pos
Executable	yes

Part-of-speech tagging using OpenNLP's tools.

By default, uses the pre-trained English model distributed with OpenNLP. If you want to use other models (e.g. for other languages), download them from the OpenNLP website to the models dir (*models/opennlp*) and specify the model name as an option.

Inputs

Name	Type(s)
text	TarredCorpus<TokenizedDocumentType|WordAnnotationsDocumentType>

Outputs

Name	Type(s)
documents	`AddAnnotationField`

Options

Name	Description	Type
model	POS tagger model, full path or filename. If a filename is given, it is expected to be in the opennlp model directory (models/opennlp/)	string

OpenNLP tokenizer

Path	pimlico.modules.opennlp.tokenize
Executable	yes

Sentence splitting and tokenization using OpenNLP's tools.

Inputs

Name	Type(s)
text	TarredCorpus<RawTextDocumentType>

Outputs

Name	Type(s)
documents	`TokenizedCorpus`

Options

Name	Description	Type
token_model	Tokenization model. Specify a full path, or just a filename. If a filename is given it is expected to be in the opennlp model directory (models/opennlp/)	string
tokenize_only	By default, sentence splitting is performed prior to tokenization. If tokenize_only is set, only the tokenization step is executed	bool
sentence_model	Sentence segmentation model. Specify a full path, or just a filename. If a filename is given it is expected to be in the opennlp model directory (models/opennlp/)	string

1.3.9 Regular expressions

Regex annotated text matcher

Path	pimlico.modules.regex.annotated_text
Executable	yes

Todo

Document this module

Inputs

Name	Type(s)
documents	TarredCorpus<WordAnnotationsDocumentType>

Outputs

Name	Type(s)
documents	`KeyValueListCorpus`

Options

Name	Description	Type
expr	(required)	string

1.3.10 Scikit-learn tools

Scikit-learn ('sklearn') provides easy-to-use implementations of a large number of machine-learning methods, based on Numpy/Scipy.

You can build Numpy arrays from your corpus using the *feature processing tools* and then use them as input to Scikit-learn's tools using the modules in this package.

Sklearn matrix factorization

Path	pimlico.modules.sklearn.matrix_factorization
Executable	yes

Todo

Document this module

Todo

Replace check_runtime_dependencies() with get_software_dependencies()

Inputs

Name	Type(s)
matrix	`ScipySparseMatrix`

Outputs

Name	Type(s)
w	`NumpyArray`
h	`NumpyArray`

Options

Name	Description	Type
class	(required)	'NMF', 'SparsePCA', 'ProjectedGradientNMF', 'FastICA', 'FactorAnalysis', 'PCA', 'RandomizedPCA', 'LatentDirichletAllocation' or 'TruncatedSVD'
options	Options to pass into the constructor of the sklearn class, formatted as a JSON dictionary (potentially without the {}s). E.g.: 'n_components=200, solver="cd", tol=0.0001, max_iter=200'	string

1.3.11 General utilities

General utilities for things like filesystem manipulation.

Copy file

Path	pimlico.modules.utility.copy_file
Executable	yes

Copy a file

Simple utility for copying a file (which presumably comes from the output of another module) into a particular location. Useful for collecting together final output at the end of a pipeline.

Inputs

Name	Type(s)
source	list of `File`

Outputs

Name	Type(s)
documents	`TarredCorpus`

Options

Name	Description	Type
target_name	Name to rename the target file to. If not given, it will have the same name as the source file. Ignored if there's more than one input file	string
target_dir	(required)	string

1.3.12 Visualization tools

Modules for plotting and suchlike

Bar chart plotter

Path	pimlico.modules.visualization.bar_chart
Executable	yes

Inputs

Name	Type(s)
values	list of `NumericResult`

Outputs

Name	Type(s)
plot	`PlotOutput`

1.4 Future plans

Various things I plan to add to Pimlico in the futures. For a summary, see Pimlico Wishlist.

1.4.1 Pimlico Wishlist

Things I plan to add to Pimlico.

- Further modules:
  - CherryPicker for coreference resolution
  - Berkeley Parser for fast constituency parsing
  - Reconcile coref. Seems to incorporate upstream NLP tasks. Would want to interface such that we can reuse output from other modules and just do coref.
- Output pipeline graph visualizations: Outputting pipeline diagrams
- Bug in counting of corpus size (off by one, sometimes) when a map process restarts

Todos

Todo

Write full documentation for this

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/pimlico/checkouts/v0.5/docs/core/config.rst, line 10.)

Todo

Write something about how dependencies are fetched

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/pimlico/checkouts/v0.5/docs/core/dependencies.rst, line 5.)

Todo

Write documentation for this

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/pimlico/checkouts/v0.5/docs/core/module_structure.
line 9.)

Todo

Document variants

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/pimlico/checkouts/v0.5/docs/core/variants.rst, line 5.)

Todo

Write a guide to building document map modules

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/pimlico/checkouts/v0.5/docs/guides/map_module.rst
line 5.)

Todo

Use a dataset that everyone can get to in the example

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/pimlico/checkouts/v0.5/docs/guides/setup.rst, line 90.)

Todo

Document this module

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/pimlico/checkouts/v0.5/docs/modules/pimlico.modu
line 12.)

Todo

Document this module

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/pimlico/checkouts/v0.5/docs/modules/pimlico.modu
line 12.)

Todo

Document this module

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/pimlico/checkouts/v0.5/docs/modules/pimlico.modu
line 12.)

Todo

Document this module

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/pimlico/checkouts/v0.5/docs/modules/pimlico.modu
line 12.)

Todo

Document this module

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/pimlico/checkouts/v0.5/docs/modules/pimlico.modu
line 12.)

Todo

Document this module

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/pimlico/checkouts/v0.5/docs/modules/pimlico.modu
line 12.)

Todo

Replace check_runtime_dependencies() with get_software_dependencies()

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/pimlico/checkouts/v0.5/docs/modules/pimlico.modu
line 17.)

Todo

Document this module

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/pimlico/checkouts/v0.5/docs/modules/pimlico.modu
line 12.)

Todo

Replace check_runtime_dependencies() with get_software_dependencies()

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/pimlico/checkouts/v0.5/docs/modules/pimlico.modu
line 17.)

Todo

Document this module

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/pimlico/checkouts/v0.5/docs/modules/pimlico.modu
line 12.)

Todo

Document this module

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/pimlico/checkouts/v0.5/docs/modules/pimlico.modu
line 12.)

Todo

Document this module

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/pimlico/checkouts/v0.5/docs/modules/pimlico.modu
line 12.)

Todo

Replace check_runtime_dependencies() with get_software_dependencies()

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/pimlico/checkouts/v0.5/docs/modules/pimlico.modu
line 17.)

1.4.2 Berkeley Parser

https://github.com/slavpetrov/berkeleyparser

Java constituency parser. Pre-trained models are also provided in the Github repo.

Probably no need for a Java wrapper here. The parser itself accepts input on stdin and outputs to stdout, so just use a subprocess with pipes.

1.4.3 Cherry Picker

Coreference resolver

http://www.hlt.utdallas.edu/~altaf/cherrypicker/

Requires NER, POS tagging and constituency parsing to be done first. Tools for all of these are included in the Cherry Picker codebase, but we just need a wrapper around the Cherry Picker tool itself to be able to feed these annotations in from other modules and perform coref.

Write a Java wrapper and interface with it using Py4J, as with OpenNLP.

1.4.4 Outputting pipeline diagrams

Once pipeline config files get big, it can be difficult to follow what's going on in them, especially if the structure is more complex than just a linear pipeline. A useful feature would be the ability to display/output a visualization of the pipeline as a flow graph.

It looks like the easiest way to do this will be to construct a DOT graph using Graphviz/Pydot and then output the diagram using Graphviz.

http://www.graphviz.org

https://pypi.python.org/pypi/pydot

Building the graph should be pretty straightforward, since the mapping from modules to nodes is fairly direct.

We could also add extra information to the nodes, like current execution status.

- genindex
- API docs
- search

m

P