Topik Documentation

Release 0.2.2+115.gde72225.dirty

Topik development team

November 24, 2015

Contents

Topik is a Topic Modeling toolkit.

CHAPTER 1

What's a topic model?

The following three definitions are a good introduction to topic modeling:

- A topic model is a type of statistical model for discovering the abstract "topics" that occur in a collection of documents [1].
- Topic models are a suite of algorithms that uncover the hidden thematic structure in document collections. These algorithms help us develop new ways to search, browse and summarize large archives of texts [2].
- Topic models provide a simple way to analyze large volumes of unlabeled text. A "topic" consists of a cluster of words that frequently occur together [3].

[1] http://en.wikipedia.org/wiki/Topic_model.
[2] http://www.cs.princeton.edu/~blei/topicmodeling.html
[3] http://mallet.cs.umass.edu/topics.php

CHAPTER 2

Yet Another Topic Modeling Library

Some of you may be wondering why the world needs yet another topic modeling library. There are already great topic modeling libraries out there, see *Useful Topic Modeling Resources*. In fact *topik* is built on top of some of them.

The aim of *topik* is to provide a full suite and high-level interface for anyone interested in applying topic modeling. For that purpose, *topik* includes many utilities beyond statistical modeling algorithms and wraps all of its features into an easy callable function and a command line interface.

Topik's desired goals are the following:

- Provide a simple and full-featured pipeline, from text extraction to final results analysis and interactive visualizations.
- Integrate available topic modeling resources and features into one common interface, making it accessible to the beginner and/or non-technical user.
- Include pre-processing data wrappers into the pipeline.
- Provide useful analysis and visualizations on topic modeling results.
- Be an easy and beginner-friendly module to contribute to.

2.1 Contents

2.1.1 Installation

Topik is meant to be a high-level interface for many topic modeling utilities (tokenizers, algorithms, visualizations...), which can be written in different languages (Python, R, Java...). Therefore, the recommended and easiest way to install *Topik* with all its features is using the package manager *conda*. Conda is a cross-platform, language agnostic tool for managing packages and environments.

```
$ conda install -c memex topik
```

There is also the option of just installing the Python features with pip.

```
$ pip install topik
```

Warning: The pip installation option will not provide all the available features, e.g. the LDAvis R package will not be available.

2.1.2 Introduction Tutorial

In this tutorial we will examine *topik* with a practical example: Topic Modeling for Movie Reviews.

Preparing The Movie Review Dataset

In this tutorial we are going to use the Sentiment Polarity Dataset Version 2.0 from Bo Pang and Lillian Lee.

```
$ mkdir doc_example
$ cd doc_example
$ curl -o review_polarity.tar.gz http://www.cs.cornell.edu/people/pabo/movie-review-data/review_polar
$ tar -zxf review_polarity.tar.gz
```

Instead of using the dataset for sentiment analysis, its initial purpose, we'll perform topic modeling on the movie reviews. For that reason, we'll merge both folders *pos* and *neg*, to one named *reviews*:

```
$ mkdir reviews
$ mv txt_sentoken/pos/* txt_sentoken/neg/* reviews/
```

High-level interface

For quick, one-off studies, the command line interface allows you to specify minimal information and obtain topic model plot output. For all available options, please run `topik --help`

```
$ topik --help

Usage: topik [OPTIONS]

Run topic modeling

Options:
    -d, --data TEXT         Path to input data for topic modeling  [required]
    -c, --field TEXT        the content field to extract text from, or for
                             folders, the field to store text as  [required]
    -f, --format TEXT       Data format provided: json_stream, folder_files,
                             large_json, elastic
    -m, --model TEXT        Statistical topic model: lda, plsa
    -o, --output TEXT       Topic modeling output path
    -t, --tokenizer TEXT    Tokenize method to use: simple, collocations,
                             entities, mix
    -n, --ntopics INTEGER   Number of topics to find
    --termite TEXT          Whether to output a termite plot as a result
    --ldavis TEXT           Whether to output an LDAvis-type plot as a result
    --help                  Show this message and exit.
```

To run this on our movie reviews data set:

```
$ topik -d reviews -c text
```

The shell command is a front end to *run_model()*, which is also accessible in python:

```
>>> from topik.simple_run.run import run_model
>>> run_model("./reviews/", content_field="text")
```

Custom topic modeling flow

For interactive exploration and more efficient, involved workflows, there also exists a Python API for using each part of the topic modeling workflow. There are four phases to topic modeling with topik: data import, tokenization/vectorization, modeling and visualization. Each phase is modular, with several options available to you for each step.

An example complete workflow would be the following:

```
>>> from topik import read_input, tokenize, vectorize, run_model, visualize
>>> raw_data = read_input("./reviews/")
>>> content_field = "text"
>>> raw_data = ((hash(item[content_field]), item[content_field]) for item in raw_data)
>>> tokenized_corpus = tokenize(raw_data)
>>> vectorized_corpus = vectorize(tokenized_corpus)
>>> n_topics = 10
>>> model = run_model(vectorized_corpus, n_topics)
>>> plot = visualize(model)
```

2.1.3 Usage in Python

API Levels

There are 3 API levels in the Topik design. At the highest level, very few details need to be provided by the user, but less room for customization exists.

Top-level API (basic, quick experiments)

Basic functions are provided as immediate children of the topik module:

```
>>> from topik import TopikProject, tokenize, vectorize, run_model
```

These provide defaults, and allow pass-through of any other arguments as keyword arguments.

Package structure

This is likely the most useful API for Python power users. Import functions directly from their packages. This will show you their complete list of available arguments, and give you autocompletion of those arguments.

```
>>> from topik.vectorizers.tfidf import tfidf
>>> from topik.models.plsa import plsa
```

Registered functions

This API layer exists primarily for creating extensible GUIs. All functions for each processing step are registered with a Borg-pattern dictionary upon import. This dictionary should be useful in modularly exposing available functionality, simply by querying keys. What is not (yet) implemented is any sort of type hinting that would let you also create appropriate input widgets for all inputs.

Topik Projects

Topik can be run by chaining the output of one step directly to the input of another:

```
>>> from topik import read_input, tokenize, vectorize, run_model, visualize
>>> raw_data = read_input("./reviews/")
>>> content_field = "text"
>>> raw_data = ((hash(item[content_field]), item[content_field]) for item in raw_data)
>>> tokens = tokenize(raw_data)
>>> vectors = vectorize(tokens)
>>> model_output = model(vectors, ntopics=4)
>>> plot = visualize(model_output)
```

However, this does not handle any kind of persistence to disk. Your changes will be lost when you exit your Python session. To unify saving all the output from all the steps together, Topik provides projects. Here's a comparable project usage to the example above:

```
>>> with TopikProject("my_project") as project:
>>>     project.read_input("./reviews/", content_field="text")
>>>     project.tokenize()
>>>     project.vectorize()
>>>     project.model(ntopics=4)
>>>     project.plot()
>>> # when project goes out of scope, any files are flushed to disk, and the project can later be loa
```

Projects create files on disk when instantiated if none already exist. When project files already exist, the contents of those files are loaded:

```
>>> with TopikProject("my_project") as project:
>>>     project.plot()
```

Output formats

Output formats are how your data are represented to further processing and modeling. Projects use a particular output format to store your intermediate results. To ensure a uniform interface, output formats implement the interface described by *OutputInterface*. Presently, two such backends are implemented: *InMemoryOutput* and *ElasticSearchOutput*. Available outputs can be examined by checking the keys of the `topik.fileio.registered_outputs` dictionary:

```
>>> from topik.fileio import registered_outputs
>>> list(registered_outputs.keys())
```

The default output is the *InMemoryOutput*. No additional arguments are necessary. *InMemoryOutput* stores everything in a Python dictionary. As such, it is memory intensive. All operations done with a *InMemoryOutput* block until complete. *InMemoryOutput* is the simplest to use, but it will ultimately limit the size of analyses that you can perform.

The *ElasticSearchOutput* can be specified to *TopikProject* using the `output_type` argument. It must be accompanied by another keyword argument, `output_args`, which should be a dictionary containing connection details and any additional arguments.

```
>>> output_args = {"source": "localhost", "index": "destination_index", content_field="text"}
>>> project = TopikProject("my_project", output_type="ElasticSearchOutput",
                           output_args=output_args)
```

ElasticSearchOutput stores everything in an *Elasticsearch* instance that you specify. Operations do not block, and have "eventual consistency": the corpus will eventually have all of the documents you sent available, but not

necessarily immediately after the read_input function returns. This lag time is due to *Elasticsearch* indexing the data on the server side.

Saving and loading projects

Projects are designed to help you go back to some earlier state. There are several dictionary-like objects accessible on the project object:

```
>>> project = TopikProject("my_project")
>>> project.output.tokenized_corpora
>>> project.output.vectorized_corpora
>>> project.output.modeled_corpora
```

These are more quickly accessible as selected properties of the project:

```
>>> project.selected_filtered_corpus
>>> project.selected_tokenized_corpus
>>> project.selected_vectorized_corpus
>>> project.selected_modeled_corpus
```

These selected properties keep track of the last-used technique, and give you the corresponding data.

You can change the selected state using the *select_tokenized_corpus()*, *select_modeled_corpus()*, and *select_modeled_corpus()* methods.

Project objects also persist their state to disk. This is done in two or more files, dependent on the output backend in use. There will always be two files:

- a .topikproject file, describing the project metadata and how to load the project
- a .topikdata file, containing or describing how to obtain the data contained in the project.

Each of the above files are JSON format. Additional files may store data in binary format. If you move your outputs on disk, make sure to move all of them, or Topik will not be able to load your results.

If using the project with a context manager, data is saved and connections are closed when the context ends. Otherwise, call the *save()* to write data to disk, or the *close()* method to write data to disk and close connections.

Loading projects is achieved by providing simply the project name that you provided when creating the project. Additional connection details will be loaded from disk automatically.

```
>>> project = TopikProject("my_project")
```

Data Import

Data import loads your data from some external representation into an iterable, internal representation for Topik. The main front end for importing data is the *read_input()* function:

```
>>> from topik import read_input
>>> corpus = read_input(source="./reviews/")
```

read_input() is a front-end to several potential reader backends. Presently, *read_input()* attempts to recognize which backend to use based on some characteristics of the source string you pass in. These criteria are:

- ends with .js or .json: treat as JSON stream filename first, fall back to "large JSON" (such as file generated by esdump).
- contains 9200: treat as *Elasticsearch* connection address (9200 is the default *Elasticsearch* port).

- result of os.path.splitext(source)[1] is "": treat as folder of files. Each file is considered raw text, and its contents are stored under the key given by content_field. Files may be gzipped.

Any of the backends can also be forced by passing the source_type argument with one of the following string arguments (see the `topik.fileio.registered_inputs` dictionary):

- elastic
- json_stream
- large_json
- folder

JSON additional options

For JSON stream and "large JSON" inputs, an additional keyword argument may be passed, `json_prefix`, which is the period-separated path leading to the single content field. This is for content fields not at the root level of the JSON document. For example, given the JSON content:

```
[ {"nested": {"dictionary": {"text": "I am the text you're looking for."} } } ]
```

You would read using the following `json_prefix` argument:

```
>>> corpus = read_input(source="data_file.json", json_prefix="nested.dictionary")
```

Elasticsearch additional options and notes

The *Elasticsearch* importer expects a full string specifying the *Elasticsearch* server. This string at a minimum must contain both the server address and the index to access (if any). All results returned from the *Elasticsearch* query contain only the contents of the '_source' field returned from the query.

```
>>> corpus = read_input(source="https://localhost:9200", index="test_index")
```

Extra arguments passed by keyword are passed to the *Elasticsearch* instance creation. This can be used to pass additional login parameters, for example, to use SSL:

```
>>> corpus = read_input(source="https://user:secret@localhost:9200",
                        index="test_index", use_ssl=True)
```

The source argument for Elasticsearch also supports multiple servers, though this requires that you manually specify the 'elastic' source_type:

```
>>> corpus = read_input(source=["https://server1", "https://server2"],
                        index="test_index", source_type="elastic")
```

For more information on server options, please refer to Elasticsearch's documentation.

Extra keyword arguments are also passed to the scroll helper that returns results. Of special note here, an additional `query` keyword argument can be passed to limit the records imported from the server. This query must follow the Elasticsearch query DSL. For more information on Elasticsearch query DSL, please refer to Elasticsearch's DSL docs.

```
>>> query = "{"filtered": {"query": {"match": { "tweet": "full text search"}}}}"
>>> corpus = read_input(source="https://localhost:9200", index="test_index", query=query)
```

Tracking documents

One important aspect that hasn't come up here is that documents are tracked by hashing their contents. Projects do this for you automatically:

```
>>> project = TopikProject("my_project")
>>> project.read_input("./reviews/", content_field="text")
```

If you do not use Topik's project feature, then you need to create these id's yourself. Tokenization and all subsequent steps expect data that has these id's, with the idea that any future parallelism will use these id's to keep track of data during and after processing. One way to get id's is below:

```
>>> content_field = "text"
>>> raw_data = ((hash(item[content_field]), item[content_field]) for item in corpus)
```

Tokenizing

The next step in topic modeling is to break your documents up into individual terms. This is called tokenization. Tokenization is done using either the *tokenize()* dispatcher function on a Corpus iterator (returned from *read_input()*), or using one of the tokenizer functions directly:

```
>>> tokenized_corpus = tokenize(raw_data)
```

Available methods

The tokenize method accepts a few arguments to specify a tokenization method and control behavior therein. The available tokenization methods are available in the `topik.tokenizers.registered_tokenizers` dictionary. The presently available methods are:

- "simple": (default) lowercases input text and extracts single words. Uses Gensim.
- "ngrams": Collects bigrams and trigrams in addition to single words. Uses NLTK.
- "entities": Extracts noun phrases as entities. Uses TextBlob.
- "mixed": first extracts noun phrases as entities, then follows up with simple tokenization for single words. Uses TextBlob.

All methods accept a keyword argument `stopwords`, which are words that will be ignored in tokenization. These are words that add little content value, such as prepositions. The default, `None`, loads and uses gensim's STOPWORDS collection.

Collocation tokenization

Collocation tokenization collects phrases of words (pairs and triplets, bigrams and trigrams) that occur together often throughout your collection of documents.

To obtain the bigram and trigram patterns, use the *ngrams()* function:

```
>>> from topik.tokenizers import ngrams
>>> tokens = ngrams(corpus, freq_bounds=[(5,10000), (3, 10000)])
```

Tweakable parameters are:

- top_n: limit results to a maximum number
- min_length: the minimum length that any single word can be

- freq_bounds: list of tuples of [(min_freq, max_freq)]. Min_freq is the minimum number of times that a pair occurs before being considered. The first entry in this list is bigrams. Presently, only bigrams and trigrams are supported.

For small bodies of text, you'll need small freq values, but this may be correspondingly "noisy."

Entities tokenization

We refer to entities as noun phrases, as extracted by the TextBlob library. Topik provides the *entities()* function.

You can tweak noun phrase extraction with a minimum and maximum occurrence frequency. This is the frequency across your entire corpus of documents.

```
>>> from topik.tokenizers import entities
>>> tokens = entities(corpus, min_length=1, freq_min=4, freq_max=10000)
```

Mixed tokenization

mixed() tokenization employs both the entities tokenizer and the simple tokenizer, for when the entities tokenizer is overly restrictive, or for when words are interesting both together and apart. Usage is similar to the entities tokenizer:

```
>>> from topik.tokenizers import mixed
>>> tokens = mixed(corpus, min_length=1, freq_min=4, freq_max=10000)
```

Vectorization

Vectorization is the process of transforming words into numerical representation. It involves the accumulation of the aggregate vocabulary, mapping that vocabulary to numeric identifiers, and finally, some arithmetic involving word counts in documents, and potentially in the collection.

Bag of Words vectorization

Bag of words is simply a word count. Words are collected from the tokenized input, assigned numeric identifiers, and counted.

bag_of_words() is the default method applied by the `vectorize()` function:

```
>>> from topik import vectorize
>>> vector_output = vectorize(tokenized_corpus)
```

TF-IDF vectorization

tfidf() (Term-Frequency-Inverse-Document-Frequency) is a scheme that weights words based not only on how much they occur in a single document, but also how much they occur across the entire corpus. Words that occur many times in one document, but not much over the corpus are deemed to be more important under this scheme.

```
>>> from topik import vectorize
>>> vector_output = vectorize(tokenized_corpus, method="tfidf")
```

Vectorizer output

Vectorization is the step at which conversion from text to numerical identifiers happens. For this reason, the output of the vectorizer is an object that contains both the vectorized corpus, and a dictionary mapping the text tokens to their numeric identifiers. This object is defined in *VectorizerOutput*. These output objects are passed directly to modeling functions.

Topic modeling

Topic modeling performs some mathematical modeling of your input data as a (sparse) matrix of which documents contain which words, attempting to identify latent "topics". At the end of modeling, each document will have a mix of topics that it belongs to, each with a weight. Each topic in turn will have weights associated with the collection of words from all documents.

Currently, Topik provides interfaces to or implements two topic modeling algorithms, LDA (latent dirichlet allocation) and PLSA (probablistic latent semantic analysis). LDA and PLSA are closely related, with LDA being a slightly more recent development. The authoritative sources on LDA and PLSA are Blei, Ng, Jordan (2003), and Hoffman (1999), respectively.

Presently, all topic models require you to specify your desired number of topics as input to the modeling process. With too many topics, you will overfit your data, making your topics difficult to make sense of. With too few, you'll merge topics together, which may hide important differences. Make sure you play with the ntopics parameter to come up with the results that are best for your collection of data.

To perform topic modeling on your tokenized data, select a model function from the `topik.models.registered_models` dictionary, or simply import a model function directly, and feed it your vectorized text. Note that not all models are compatible with all vectorizer methods. In particular, LDA is not compatible with TF-IDF vectorization.

```
>>> from topik.models import registered_models, lda
>>> model = registered_models["lda"](vectorized_corpora, 4)
>>> model = LDA(vectorized_corpora, 4)
```

The output of modeling is a *ModelOutput*. This class provides simple methods for obtaining the topic-term frequency and document-topic frequency, as well as some supporting metadata. This object is fed directly to visualization methods.

Viewing results

Each model supports a few standard outputs for examination of results:

- Termite plots
- LDAvis-based plots
  - topik's LDAvis-based plots use the pyLDAvis module, which is itself a

  Python port of the R_ldavis library. The visualization consists of two linked, interactive views. On the left is a projection of the topics onto a 2-dimensional space, where the area of each circle represents that topic's relative prevalence in the corpus. The view on the right shows details of the composition of the topic (if any) selected in the left view. The slider at the top of the right view adjusts the relevance metric used when ranking the words in a topic. A value of 1 on the slider will rank terms by their probabilities for that topic (the red bar), whereas a value of 0 will rank them by their probabilities for that topic divided by their probabilities for the overall corpus (red divided by blue).

Example syntax for these:

```
>>> from topik.visualizers.termite_plot import termite_html
>>> termite = termite_html(model_output, "termite.html", "Termite Plot", topn=15)
```

```
>>> from topik.visualizers import lda_vis
>>> lda_vis(model_output)
```

2.1.4 Development Guide

Topik has been designed to be extensible at each of the five steps of topic modeling:

- data import
- tokenization / transformation
- vectorization
- topic modeling
- visualization

Each of these steps are designed using function registries to guide extension development and to make creation of pluggable user interfaces feasible. The general layout of topik is the following:

- a folder (python package) for each step
  - __init__.py : exposes public API methods for the package.
  - _registry.py : implementation and instantiation of the function registry for each package. There is generally only one, but can be more than one registry per package. This file also contains the register decorator, used to register functions with the registry. This decorator runs when the file using it is imported.
  - ???_output.py : class implementation (possibly base class) of output from this package's step. This is not strictly necessary. When a step returns only an iterator of data, there is no additional output class created to wrap it.
  - any function implementations for the given step.
  - a tests folder, containing unit tests for the given step. These should not depend on output from other steps. Hard-code data as necessary.

External code can hook into the dictionary of registered methods using the appropriate register decorator function, which should be made available in each package's __init__.py. This decorator will execute when the foreign code is first run, so make sure that you import your module before requesting the dictionary of registered classes for a given step.

For general command line or IPython notebook usage, it is probably easier to directly import functions from the folder structure, rather than depending on the function registry. The registered dictionary approach makes dynamic UI creation easier, but it hinders autocompletion. An intermediate approach would be to assign the results of dictionary access to a variable before instantiating the class. For example,

```
>>> # one-shot, but autocompletion of function arguments doesn't work
>>> model = registered_models["LDA"](tokenized_corpora, 5)
```

```
>>> model_class = registered_models["LDA"]
>>> # Autocompletion of class arguments should work here
>>> model = model_class(tokenized_corpora, 5)
```

```
>>> # import model implementation directly:
>>> from topik.models import lda
>>> # Autocompletion of class arguments should work here
>>> model = lda(vectorized_corpus, 5)
```

2.1.5 topik package

Subpackages

topik.fileio package

Subpackages

topik.fileio.tests package

Submodules

topik.fileio.tests.test_in_document_folder module

topik.fileio.tests.test_in_document_folder.**test_bad_folder**()
topik.fileio.tests.test_in_document_folder.**test_read_document_folder**()
topik.fileio.tests.test_in_document_folder.**test_read_document_folder_gz**()

topik.fileio.tests.test_in_elastic module

topik.fileio.tests.test_in_elastic.**test_elastic_import**()

topik.fileio.tests.test_in_json module

topik.fileio.tests.test_in_json.**test___is_iterable**()
topik.fileio.tests.test_in_json.**test_read_document_json_stream**()
topik.fileio.tests.test_in_json.**test_read_large_json**()

topik.fileio.tests.test_outputs module

class topik.fileio.tests.test_outputs.**BaseOutputTest**
Bases: object

test_get_date_filtered_data()

test_get_filtered_data()

test_load_file()

test_raw_data = None

test_save_file()

class topik.fileio.tests.test_outputs.**TestElasticSearchOutput**(*methodName='runTest'*)
Bases: unittest.case.TestCase, *topik.fileio.tests.test_outputs.BaseOutputTest*

setUp()

tearDown()

class topik.fileio.tests.test_outputs.**TestInMemoryOutput**(*methodName='runTest'*)
Bases: unittest.case.TestCase, *topik.fileio.tests.test_outputs.BaseOutputTest*

setUp()

topik.fileio.tests.test_project module

class topik.fileio.tests.test_project.**ProjectTest**

Bases: object

test_context_manager()

test_get_date_filtered_corpus_iterator()

test_get_filtered_corpus_iterator()

test_model()

test_read_input()

test_tokenize()

test_vectorize()

test_visualize()

class topik.fileio.tests.test_project.**TestElasticSearchOutput**(*methodName='runTest'*)

Bases: unittest.case.TestCase, *topik.fileio.tests.test_project.ProjectTest*

INDEX = 'test_index'

setUp()

tearDown()

class topik.fileio.tests.test_project.**TestInMemoryOutput**(*methodName='runTest'*)

Bases: unittest.case.TestCase, *topik.fileio.tests.test_project.ProjectTest*

setUp()

topik.fileio.tests.test_reader module

topik.fileio.tests.test_reader.**test_read_input**()

Module contents

Submodules

topik.fileio.base_output module

class topik.fileio.base_output.**OutputInterface**(**args*, ***kwargs*)

Bases: object

close()

get_filtered_data(*field_to_get*, *filter=''*)

save(*filename*, *saved_data=None*)

Persist this object to disk somehow.

You can save your data in any number of files in any format, but at a minimum, you need one json file that describes enough to bootstrap the loading process. Namely, you must have a key called 'class' so that upon loading the output, the correct class can be instantiated and used to load any other data. You don't have to implement anything for saved_data, but it is stored as a key next to 'class'.

synchronize(*max_wait*, *field*)

By default, operations are synchronous and no additional wait is necessary. Data sources that are asynchronous (ElasticSearch) may use this function to wait for "eventual consistency"

topik.fileio.base_output.**load_output**(*filename*)

topik.fileio.in_document_folder module

topik.fileio.in_document_folder.**read_document_folder**(*folder*, *content_field='text'*)

Iterate over the files in a folder to retrieve the content to process and tokenize.

Parameters **folder** : str

The folder containing the files you want to analyze.

content_field : str

The usage of 'content_field' in this source is different from most other sources. The assumption in this source is that each file contains raw text, NOT dictionaries of categorized data. The content_field argument here specifies what key to store the raw text under in the returned dictionary for each document.

Examples

```
>>> documents = read_document_folder(
...     '{}/test_data_folder_files'.format(test_data_path))
>>> next(documents)['text'] == (
...     u"'Interstellar' was incredible. The visuals, the score, " +
...     u"the acting, were all amazing. The plot is definitely one " +
...     u"of the most original I've seen in a while.")
True
```

topik.fileio.in_elastic module

topik.fileio.in_elastic.**read_elastic**(*hosts*, ***kwargs*)

Iterate over all documents in the specified elasticsearch intance and index that match the specified query.

kwargs are passed to Elasticsearch class instantiation, and can be used to pass any additional options described at https://elasticsearch-py.readthedocs.org/en/master/

Parameters **hosts** : str or list

Address of the elasticsearch instance any index. May include port, username and password. See https://elasticsearch-py.readthedocs.org/en/master/api.html#elasticsearch for all options.

content_field : str

The name fo the field that contains the main text body of the document.

****kwargs: additional keyword arguments to be passed to Elasticsearch client instance and to scan query.**

See https://elasticsearch-py.readthedocs.org/en/master/api.html#elasticsearch for all client options. https://elasticsearch-py.readthedocs.org/en/master/helpers.html#elasticsearch.helpers.scan for all scan options.

topik.fileio.in_json module

topik.fileio.in_json.**read_json_stream**(*filename*, *json_prefix='item'*, ***kwargs*)

Iterate over a json stream of items and get the field that contains the text to process and tokenize.

Parameters **filename** : str

The filename of the json stream.

Examples

```
>>> documents = read_json_stream(
... '{}/test_data_json_stream.json'.format(test_data_path))
>>> next(documents) == {
... u'doi': u'http://dx.doi.org/10.1557/PROC-879-Z3.3',
... u'title': u'Sol Gel Preparation of Ta2O5 Nanorods Using DNA as Structure Directing Agent',
... u'url': u'http://journals.cambridge.org/action/displayAbstract?fromPage=online&aid=8081671&f
... u'abstract': u'Transition metal oxides are being considered as the next generation materials
... u'filepath': u'abstracts/879/http%3A%2F%2Fjournals.cambridge.org%2Faction%2FdisplayAbstract
... u'filename': '{}/test_data_json_stream.json'.format(test_data_path),
... u'vol': u'879',
... u'authors': [u'Humberto A. Monreala', u' Alberto M. Villafañe',
...              u' José G. Chacón', u' Perla E. García',
...              u'Carlos A. Martínez'],
... u'year': u'1917'}
True
```

topik.fileio.in_json.**read_large_json**(*filename*, *json_prefix='item'*, ***kwargs*)

Iterate over all items and sub-items in a json object that match the specified prefix

Parameters **filename** : str

The filename of the large json file

json_prefix : str

The string representation of the hierarchical prefix where the items of interest may be located within the larger json object.

Try the following script if you need help determining the desired prefix: $ import ijson $ with open('test_data_large_json_2.json', 'r') as f: $ parser = ijson.parse(f) $ for prefix, event, value in parser: $ print("prefix = '%r' || event = '%r' || value = '%r'" % $ (prefix, event, value))

Examples

```
>>> documents = read_large_json(
...              '{}/test_data_large_json.json'.format(test_data_path),
...              json_prefix='item._source.isAuthorOf')
>>> next(documents) == {
... u'a': u'ScholarlyArticle',
... u'name': u'Path planning and formation control via potential function for UAV Quadrotor',
... u'author': [
...     u'http://dig.isi.edu/autonomy/data/author/a.a.a.rizqi',
...     u'http://dig.isi.edu/autonomy/data/author/t.b.adji',
...     u'http://dig.isi.edu/autonomy/data/author/a.i.cahyadi'],
... u'text': u"Potential-function-based control strategy for path planning and formation " +
...     u"control of Quadrotors is proposed in this work. The potential function is " +
...     u"used to attract the Quadrotor to the goal location as well as avoiding the " +
...     u"obstacle. The algorithm to solve the so called local minima problem by utilizing " +
...     u"the wall-following behavior is also explained. The resulted path planning via " +
...     u"potential function strategy is then used to design formation control algorithm. " +
...     u"Using the hybrid virtual leader and behavioral approach schema, the formation " +
...     u"control strategy by means of potential function is proposed. The overall strategy " +
...     u"has been successfully applied to the Quadrotor's model of Parrot AR Drone 2.0 in " +
...     u"Gazebo simulator programmed using Robot Operating System.\nAuthor(s) Rizqi, A.A.A. " +
...     u"Dept. of Electr. Eng. & Inf. Technol., Univ. Gadjah Mada, Yogyakarta, Indonesia " +
...     u"Cahyadi, A.I. ; Adji, T.B.\nReferenced Items are not available for this document.\n" +
```

```
    ...     u"No versions found for this document.\nStandards Dictionary Terms are available to " +
    ...     u"subscribers only.",
    ... u'uri': u'http://dig.isi.edu/autonomy/data/article/6871517',
    ... u'datePublished': u'2014',
    ... 'filename': '{}/test_data_large_json.json'.format(test_data_path)}
    True
```

topik.fileio.out_elastic module

class topik.fileio.out_elastic.**BaseElasticCorpora**(*instance*, *index*, *corpus_type*, *query=None*, *batch_size=1000*)

Bases: UserDict.UserDict

class topik.fileio.out_elastic.**ElasticSearchOutput**(*source*, *index*, *hash_field=None*, *doc_type='continuum'*, *query=None*, *iterable=None*, *filter_expression=''*, *vectorized_corpora=None*, *tokenized_corpora=None*, *modeled_corpora=None*, ***kwargs*)

Bases: *topik.fileio.base_output.OutputInterface*

convert_date_field_and_reindex(*field*)

filter_string

get_date_filtered_data(*field_to_get*, *start*, *end*, *filter_field='date'*)

get_filtered_data(*field_to_get*, *filter=''*)

import_from_iterable(*iterable*, *field_to_hash='text'*, *batch_size=500*)

Load data into Elasticsearch from iterable.

iterable: generally a list of dicts, but possibly a list of strings This is your data. Your dictionary structure defines the schema of the elasticsearch index.

field_to_hash: string identifier of field to hash for content ID. For list of dicts, a valid key value in the dictionary is required. For list of strings, a dictionary with one key, "text" is created and used.

save(*filename*, *saved_data=None*)

synchronize(*max_wait*, *field*)

class topik.fileio.out_elastic.**ModeledElasticCorpora**(*instance*, *index*, *corpus_type*, *query=None*, *batch_size=1000*)

Bases: *topik.fileio.out_elastic.BaseElasticCorpora*

class topik.fileio.out_elastic.**VectorizedElasticCorpora**(*instance*, *index*, *corpus_type*, *query=None*, *batch_size=1000*)

Bases: *topik.fileio.out_elastic.BaseElasticCorpora*

topik.fileio.out_elastic.**es_getitem**(*key*, *doc_type*, *instance*, *index*, *query=None*)

topik.fileio.out_elastic.**es_setitem**(*key*, *value*, *doc_type*, *instance*, *index*, *batch_size=1000*)

load an iterable of (id, value) pairs to the specified new or new or existing field within existing documents.

topik.fileio.out_memory module

class topik.fileio.out_memory.**GreedyDict**(*dict=None*, ***kwargs*)

Bases: UserDict.UserDict, object

class topik.fileio.out_memory.**InMemoryOutput**(*iterable=None*, *hash_field=None*, *tokenized_corpora=None*, *vectorized_corpora=None*, *modeled_corpora=None*)

Bases: *topik.fileio.base_output.OutputInterface*

get_date_filtered_data(*field_to_get*, *start*, *end*, *filter_field='year'*)

get_filtered_data(*field_to_get*, *filter=''*)

import_from_iterable(*iterable*, *field_to_hash*)

iterable: generally a list of dicts, but possibly a list of strings This is your data. Your dictionary structure defines the schema of the elasticsearch index.

save(*filename*)

topik.fileio.project module

class topik.fileio.project.**TopikProject**(*project_name*, *output_type=None*, *output_args=None*, ***kwargs*)

Bases: object

close()

get_date_filtered_corpus_iterator(*start*, *end*, *filter_field*, *field_to_get=None*)

get_filtered_corpus_iterator(*field=None*, *filter_expression=None*)

read_input(*source*, *content_field*, *source_type='auto'*, ***kwargs*)
Import data from external source into Topik's internal format

run_model(*model_name='plsa'*, ***kwargs*)
Analyze vectorized text; determine topics and assign document probabilities

save()
Save project as .topikproject metafile and some number of sidecar data files.

select_modeled_corpus(*_id*)
When more than one model output available (ran modeling more than once with different methods), this allows you to switch to a different data set.

select_tokenized_corpus(*_id*)
Assign active tokenized corpus.

When more than one tokenized corpus available (ran tokenization more than once with different methods), this allows you to switch to a different data set.

select_vectorized_corpus(*_id*)
Assign active vectorized corpus.

When more than one vectorized corpus available (ran tokenization more than once with different methods), this allows you to switch to a different data set.

selected_filtered_corpus
Corpus documents, potentially a subset.

Output from read_input step. Input to tokenization step.

selected_modeled_corpus
matrices representing the model derived.

Output from modeling step. Input to visualization step.

selected_tokenized_corpus
Documents broken into component words. May also be transformed.

Output from tokenization and/or transformation steps. Input to vectorization step.

selected_vectorized_corpus
Data that has been vectorized into term frequencies, TF/IDF, or other vector representation.

Output from vectorization step. Input to modeling step.

tokenize(*method='simple'*, ***kwargs*)
Break raw text into substituent terms (or collections of terms)

transform(*method*, ***kwargs*)
Stem or lemmatize input text that has already been tokenized

vectorize(*method='bag_of_words'*, ***kwargs*)
Convert tokenized text to vector form - mathematical representation used for modeling.

visualize(*vis_name='termite'*, *model_id=None*, ***kwargs*)
Plot model output

topik.fileio.reader module

topik.fileio.reader.**read_input**(*source*, *source_type='auto'*, *folder_content_field='text'*, ***kwargs*)

Read data from given source into Topik's internal data structures.

Parameters **source** : str

input data. Can be file path, directory, or server address.

source_type : str

"auto" tries to figure out data type of source. Can be manually specified instead. options for manual specification are ['solr', 'elastic', 'json_stream', 'large_json', 'folder']

folder_content_field : str

Only used for document_folder source. This argument is used as the key (field name), where each document represents the value of that field.

kwargs : any other arguments to pass to input parsers

Returns iterable output object

>> ids, texts = zip(*list(iter(raw_data)))

Examples

```
>>> loaded_corpus = read_input(
```

... '{}/test_data_json_stream.json'.format(test_data_path))

```
>>> solution_text = (
```

... u'Transition metal oxides are being considered as the next generation '+

... u'materials in field such as electronics and advanced catalysts; '+

... u'between them is Tantalum (V) Oxide; however, there are few reports '+

... u'for the synthesis of this material at the nanometer size which could '+

... u'have unusual properties. Hence, in this work we present the '+

... u'synthesis of Ta2O5 nanorods by sol gel method using DNA as structure '+

... u'directing agent, the size of the nanorods was of the order of 40 to '+

... u'100 nm in diameter and several microns in length; this easy method '+

... u'can be useful in the preparation of nanomaterials for electronics, '+

... u'biomedical applications as well as catalysts.')

```
>>> solution_text == next(loaded_corpus)['abstract']
```

True

Module contents

topik.models package

Subpackages

topik.models.tests package

Submodules

topik.models.tests.test_data module

topik.models.tests.test_lda module

topik.models.tests.test_lda.**test_train**()

topik.models.tests.test_plsa module

topik.models.tests.test_plsa.**test_cal_likelihood**()
topik.models.tests.test_plsa.**test_em**()

topik.models.tests.test_plsa.**test_rand_mat**()

topik.models.tests.test_plsa.**test_registration**()

topik.models.tests.test_plsa.**test_train**()

Module contents

Submodules

topik.models.base_model_output module

class topik.models.base_model_output.**ModelOutput**(*vectorized_corpus=None*, *model_func=None*, *vocab=None*, *term_frequency=None*, *topic_term_matrix=None*, *doc_lengths=None*, *doc_topic_matrix=None*, ***kwargs*)

Bases: object

Abstract base class for topic models.

Ensures consistent interface across models, for base result display capabilities.

Attributes

_doc_topic_matrix	(mapping of document ids to weights for topic indices) matrix storing the relative topic weights for each document
_topic_term_matrix	(mapping of terms to each topic)

doc_lengths

doc_topic_matrix

term_frequency

topic_term_matrix

vocab

topik.models.lda module

topik.models.lda.**lda**(*vectorized_output*, *ntopics*, ***kwargs*)

topik.models.plsa module

topik.models.plsa.**plsa**(*vectorized_corpus*, *ntopics*, *max_iter=100*, ***kwargs*)

Module contents

topik.simple_run package

Subpackages

topik.simple_run.tests package

Submodules

topik.simple_run.tests.test_run_api module

topik.simple_run.tests.test_run_api.**test_run**()

Module contents

Submodules

topik.simple_run.cli module

topik.simple_run.run module

topik.simple_run.run.**run_model**(*data_source*, *source_type='auto'*, *year_field=None*, *start_year=None*, *stop_year=None*, *content_field=None*, *tokenizer='simple'*, *vectorizer='bag_of_words'*, *ntopics=10*, *dir_path='./topic_model'*, *model='lda'*, *termite_plot=True*, *output_file=False*, *ldavis=False*, *seed=42*, ***kwargs*)

Run your data through all topik functionality and save all results to a specified directory.

Parameters **data_source** : str

Input data (e.g. file or folder or solr/elasticsearch instance).

source_type : {'json_stream', 'folder_files', 'json_large', 'solr', 'elastic'}.

The format of your data input. Currently available a json stream or a folder containing text files. Default is 'json_stream'

year_field : str

The field name (if any) that contains the year associated with each document (for filtering).

start_year : int

For beginning of range filter on year_field values

stop_year : int

For beginning of range filter on year_field values

content_field : string

The primary text field to parse.

tokenizer : {'simple', 'collocations', 'entities', 'mixed'}

The type of tokenizer to use. Default is 'simple'.

vectorizer : {'bag_of_words', 'tfidf'}

The type of vectorizer to use. Default is 'bag_of_words'.

ntopics : int

Number of topics to find in your data

dir_path : str

Directory path to store all topic modeling results files. Default is *./topic_model*.

model : {'LDA', 'PLSA'}.

Statistical modeling algorithm to use. Default 'LDA'.

termite_plot : bool

Generate termite plot of your model if True. Default is True.

ldavis : bool

Generate an interactive data visualization of your topics. Default is False.

seed : int

Set random number generator to seed, to be able to reproduce results. Default 42.

****kwargs** : additional keyword arguments, passed through to each individual step

Module contents

topik.tokenizers package

Subpackages

topik.tokenizers.tests package

Submodules

topik.tokenizers.tests.test_entities module

topik.tokenizers.tests.test_ngrams module

topik.tokenizers.tests.test_ngrams.**test__collect_bigrams_and_trigrams**()

topik.tokenizers.tests.test_ngrams.**test__collocation_document**()

topik.tokenizers.tests.test_ngrams.**test_ngrams**()

topik.tokenizers.tests.test_simple module

topik.tokenizers.tests.test_simple.**test__simple_document**()

topik.tokenizers.tests.test_simple.**test_simple**()

Module contents

Submodules

topik.tokenizers.entities module

topik.tokenizers.entities.**entities**(*corpus*, *min_length=1*, *freq_min=2*, *freq_max=10000*, *stopwords=None*)

A tokenizer that extracts noun phrases from a corpus, then tokenizes all documents using those extracted phrases.

Parameters **corpus** : iterable of str

A collection of text to be tokenized

min_length : int

Minimum length of any single word

freq_min : int

Minimum occurrence of phrase in order to be considered

freq_max : int

Maximum occurrence of phrase, beyond which it is ignored

stopwords : None or iterable of str

Collection of words to ignore as tokens

Examples

```
>>> tokenized_corpora = entities(sample_corpus)
>>> next(tokenized_corpora) == ('doc1',
...      [u'frank', u'swank_tank', u'prancercise', u'sassy_unicorns'])
True
```

topik.tokenizers.entities.**mixed**(*corpus*, *min_length=1*, *freq_min=2*, *freq_max=10000*, *stopwords=None*)

A text tokenizer that retrieves entities ('noun phrases') first and simple words for the rest of the text.

Parameters **corpus** : iterable of str

A collection of text to be tokenized

min_length : int

Minimum length of any single word

freq_min : int

Minimum occurrence of phrase in order to be considered

freq_max : int

Maximum occurrence of phrase, beyond which it is ignored

stopwords : None or iterable of str

Collection of words to ignore as tokens

Examples

```
>>> tokenized_corpora = entities(sample_corpus)
>>> next(tokenized_corpora) == ('doc1',
...      [u'frank', u'swank_tank', u'prancercise', u'sassy_unicorns'])
True
```

topik.tokenizers.ngrams module

topik.tokenizers.ngrams.**ngrams**(*raw_corpus*, *min_length=1*, *freq_bounds=None*, *top_n=10000*, *stopwords=None*)

A tokenizer that extracts collocations (bigrams and trigrams) from a corpus according to the frequency bounds, then tokenizes all documents using those extracted phrases.

Parameters **raw_corpus** : iterable of tuple of (doc_id(str/int), doc_text(str))

body of documents to examine

min_length : int

Minimum length of any single word

freq_bounds : list of tuples of ints

Currently ngrams supports bigrams and trigrams, so this list should contain two tuples (the first for bigrams, the second for trigrams), where each tuple consists of a (minimum, maximum) corpus-wide frequency.

top_n : int

limit results to this many entries

stopwords: None or iterable of str

Collection of words to ignore as tokens

Examples

```
>>> tokenized_corpora = ngrams(sample_corpus, freq_bounds=[(2,100),(2,100)])
>>> next(tokenized_corpora) == ('doc1',
...      [u'frank_swank', u'tank', u'walked', u'sassy', u'unicorn', u'brony',
...      u'prancercise', u'class', u'daily', u'prancercise', u'tremendously',
...      u'popular', u'pastime', u'sassy_unicorns', u'retirees', u'alike'])
True
```

topik.tokenizers.simple module

`topik.tokenizers.simple.`**`simple`**(*raw_corpus*, *min_length=1*, *stopwords=None*)

A text tokenizer that simply lowercases, matches alphabetic characters and removes stopwords.

Parameters **raw_corpus** : iterable of tuple of (doc_id(str/int), doc_text(str))

body of documents to examine

min_length : int

Minimum length of any single word

stopwords: None or iterable of str

Collection of words to ignore as tokens

Examples

```
>>> sample_corpus = [("doc1", "frank FRANK the frank dog cat"),
...                  ("doc2", "frank a dog of the llama")]
>>> tokenized_corpora = simple(sample_corpus)
>>> next(tokenized_corpora) == ("doc1",
... ["frank", "frank", "frank", "dog", "cat"])
True
```

Module contents

topik.transformers package

Module contents

topik.vectorizers package

Subpackages

topik.vectorizers.tests package

Submodules

topik.vectorizers.tests.test_bag_of_words module
topik.vectorizers.tests.test_bag_of_words.**test_vectorizer**()

topik.vectorizers.tests.test_output module
topik.vectorizers.tests.test_output.**test_document_term_count**()
topik.vectorizers.tests.test_output.**test_global_term_count**()
topik.vectorizers.tests.test_output.**test_term_frequency**()

topik.vectorizers.tests.test_tfidf module
topik.vectorizers.tests.test_tfidf.**test_vectorize**()

Module contents

Submodules

topik.vectorizers.bag_of_words module
topik.vectorizers.bag_of_words.**bag_of_words**(*tokenized_corpora*)

topik.vectorizers.tfidf module
topik.vectorizers.tfidf.**tfidf**(*tokenized_corpus*)

topik.vectorizers.vectorizer_output module
class topik.vectorizers.vectorizer_output.**VectorizerOutput**(*tokenized_corpus=None*, *vectorizer_func=None*, *id_term_map=None*, *document_term_counts=None*, *doc_lengths=None*, *term_frequency=None*, *vectors=None*)

Bases: object

doc_lengths

document_term_counts

get_vectors()

global_term_count

id_term_map

term_frequency

term_id_map

vectors

Module contents

topik.visualizers package

Subpackages

topik.visualizers.tests package

Submodules

topik.visualizers.tests.test_ldavis module

topik.visualizers.tests.test_ldavis.**test__to_py_lda_vis**()
topik.visualizers.tests.test_ldavis.**test_ldavis**()

topik.visualizers.tests.test_termite module

topik.visualizers.tests.test_termite.**test_termite**()
topik.visualizers.tests.test_termite.**test_termite_pandas_output**()

topik.visualizers.tests.test_termite.**test_top_words**()

Module contents

Submodules

topik.visualizers.pyldavis module

topik.visualizers.pyldavis.**lda_vis**(*modeled_corpus*, *mode='show'*, *filename=None*)
Designed to work with to_py_lda_vis() in the model classes.

topik.visualizers.termite_plot module

topik.visualizers.termite_plot.**termite**(*modeled_corpus*, *plot_title='Termite plot'*, *topn=15*)
A Bokeh Termite Visualization for LDA results analysis.

Parameters **input_file** : str or pandas DataFrame

A pandas dataframe from a topik model get_termite_data() containing columns "word", "topic" and "weight". May also be a string, in which case the string is a filename of a csv file with the above columns.

title : str

The title for your termite plot

Examples

```
>>> plot = termite(test_model_output, plot_title="My model results", topn=5)
```

topik.visualizers.termite_plot.**termite_html**(*modeled_corpus*, *filename*, *plot_title='Termite plot'*, *topn=15*)

Module contents

Submodules

topik.singleton_registry module

Topik uses a modular design at each step in the topic modeling process, to support future extension. One of the core concepts here is that we provide registries for each step that can be extended through registration using a decorator.

The general pattern for using this registry is to subclass BaseRegistry, and then create a new decorator that uses that registry:

register = functools.partial(_base_register_decorator, RegistrySubclass())

we keep the name as simply register across all files for simplicity, but the register decorator in each folder is using a different registry specific to that folder. See the _registry.py module in each folder.

Module contents

2.2 Useful Topic Modeling Resources

- Topic modeling, David M. Blei

2.2.1 Python libraries

- Gensim
- Pattern
- TextBlob
- NLTK

2.2.2 R libraries

- lda
- LDAvis

2.2.3 Other

- Ditop

2.2.4 Papers

- Probabilistic Topic Models by David M.Blei

CHAPTER 3

License Agreement

topik is distributed under the BSD 3-Clause license.

3.1 Indices and tables

- genindex
- modindex
- search

3.2 Footnotes

t

N

O

P

R

S

T

V