Topik Documentation

Release 0.2.1+0.g5f7629f.dirty

Topik development team

October 10, 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, solr, 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.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, registered_models
>>> raw_data = read_input("reviews", content_field="text")
>>> tokenized_corpus = raw_data.tokenize()
>>> n_topics = 10
>>> model = registered_models["LDA"](tokenized_corpus, n_topics)
>>> from topik.viz import Termite
>>> termite = Termite(model.termite_data(n_topics), "Termite Plot")
>>> termite.plot('termite.html')
```

2.1.3 Usage in Python

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="data_file.json", content_field="text")
```

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 8983: treat as solr connection address (8983 is the default solr port).
- 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:

- solr
- elastic
- json_stream
- large_json
- folder

The `content_field` is a mandatory argument that in most cases specifies where the actual content to be analyzed will be drawn from. For all hierarchical data sources (everything except folders), this accesses some subfield of the data you feed in.

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", content_field="text",
                        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", content_field="text")
```

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", content_field="text", 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", content_field="text")
```

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",
                        content_field="tweet", query=query)
```

Output formats

Output formats are how your data are represented to further processing and modeling. To ensure a uniform interface, output formats implement the interface described by *`CorpusInterface`*. Presently, two such backends are implemented: *`DictionaryCorpus`* and *`ElasticSearchCorpus`*. Available outputs can be examined by checking the keys of the `registered_outputs` dictionary:

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

The default output is the *`DictionaryCorpus`*. No additional arguments are necessary. *`DictionaryCorpus`* stores everything in a Python dictionary. As such, it is memory intensive. All operations done with a

`DictionaryCorpus` block until complete. `DictionaryCorpus` is the simplest to use, but it will ultimately limit the size of analyses that you can perform.

The `ElasticSearchCorpus` can be specified to `read_input()` 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"}
>>> raw_data = read_input("test_data.json", output_type='elastic',
                          output_args=output_args, content_field="text")
```

`ElasticSearchCorpus` 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.

Synchronous wait

As mentioned above, some output formats are not immediately ready for consumption after loading data. For example, after sending data to Elasticsearch, Elasticsearch will take some time to index that data. Until that indexing is complete, that data will not show up in iterations over the corpus. To force your program to wait for this to finish, use the `synchronous_wait` argument to read_input:

```
>>> output_args = {"source": "localhost", "index": "destination_index"}
>>> raw_data = read_input("test_data.json", output_type='elastic',
                          output_args=output_args, content_field="text",
                          synchronous_wait=30)
```

This example will wait up to 30 seconds for the Elasticsearch indexing to stabilize. This is evaluated as the point at which the number of documents in the output has not changed after 1 second. If the number of documents has not stabilized after the synchronous wait period, you will get a warning message, but execution will proceed.

This is a property only of output formats. Input has no wait associated with it, because the source is assumed to be "complete" when you ask for it. Please make sure that this is true, or your results will be ill-defined and impossible to reproduce.

Saving and loading corpora

The output object of any `read_input()` step is saveable and loadable. This allows you to quickly get back to any filtered state you may have applied to some larger corpus, and also ensures that the corpus you load with a model is consistent with the corpus that was used to create that model. To save a corpus, call its `save()` method:

```
>>> raw_data.save("output_filename")
```

The file format of the saved file is JSON. Depending on the exact class that your corpus is, more or less data may be saved to this JSON file. For example, the `DictionaryCorpus` class saves all of its corpus data to this JSON file, and can be quite large. The `ElasticsearchCorpus` class saves only connection details and filtering metadata to this JSON file, and is much smaller.

Loading corpora is achieved using the `load_persisted_corpus()` function. This function returns the appropriate Corpus object, based on metadata in the JSON file.

```
>>> from topik.intermediaries.raw_data import load_persisted_corpus >>>
raw_data = load_persisted_corpus("output_filename")
```

Tokenizing and Vectorizing

The next step in topic modeling is to break your documents up into individual terms. This is called tokenization. Tokenization is done using the *tokenize()* method on a Corpus object (returned from *read_input()*):

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

Note on tokenize output

The *tokenize()* method returns a new object, presently of the `DigestedDocumentCollection` type. Behind the scenes, the *tokenize()* method is storing the tokenized text alongside your corpus, using whatever storage backend you have. This is an in-place modification of that object. The new object serves two purposes:

- It iterates over the particular tokenized representation of your corpus. You may have multiple tokenizations associated with a single corpus. The object returned from the tokenize function tracks the correct one.
- It also performs vectorization on the fly, counting the number of words in each document, and returning a representation of each document as a bag of words (list of tuples, with each tuple being (word_id, word_count). This is generally the desired input to any topic model.

Make sure you assign this new object to a new variable. It is what you want to feed into the topic modeling step.

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 `tokenizer_methods` dictionary. The presently available methods are:

- "simple": (default) lowercases input text and extracts single words. Uses Gensim.
- "collocation": 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. There are two steps to tokenization with collocation: establishing the patterns of bigrams and trigrams, and subsequently tokenizing each document individually.

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

```
>>> from topik.tokenizers import collect_bigrams_and_trigrams
>>> patterns = collect_bigrams_and_trigrams(corpus)
```

Parameterization is done at this step, prior to tokenization of the corpus. Tweakable parameters are:

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

- min_bigram_freq: the minimum number of times a pair of words must occur together to be included
- min_trigram_freq: the minimum number of times a triplet of words must occur together to be included

```
>>> patterns = collect_bigrams_and_trigrams(corpus, min_length=3, min_bigram_freq=3, min_trigram_freq
```

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

Next, feed the patterns into the `tokenize()` method of your corpus object:

```
>>> tokenized_corpus = raw_data.tokenize(method="collocation", patterns=patterns)
```

Entities tokenization

We refer to entities as noun phrases, as extracted by the TextBlob library. Like collocation tokenization, entities tokenization is a two-step process. First, you establish noun phrases using the `collect_entities()` function:

```
>>> from topik.tokenizers import collect_entities
>>> entities = collect_entities(corpus)
```

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

```
>>> entities = collect_entities(corpus, freq_min=4, freq_max=10000)
```

Next, tokenize the document collection:

```
>>> tokenized_corpus = raw_data.tokenize(method="entities", entities=entities)
```

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 collect_entities
>>> entities = collect_entities(corpus)
>>> tokenized_corpus = raw_data.tokenize(method="mixed", entities=entities)
```

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 class from the `registered_models` dictionary, or simply import a model class directly, and instantiate this object with your corpus and the number of topics to model:

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

Presently, training the model is implicit in its instantiation. In other words, when you create an object using the code above, the data are loaded into the model, and the analysis to identify topics is performed immediately. That means that instantiating an object may take some time. Progress indicators are on our road map, but for now, please be patient and wait for your results.

Saving and loading results

The model object has a *`save()`* method. This method saves a JSON file that describes how to load the rest of the data for your model and for your corpus. The *`load_model()`* function will read that JSON file, and recreate the necessary corpus and model objects to leave you where you saved. Each model has its own binary representation, and each corpus type has its own storage backend. The JSON file saved here generally does not include corpus data nor model data, but rather is simply instructions on where to find those data. If you move files around on your hard disk, make sure to pick up everything with the JSON file.

```
>>> model.save("test_data.json")
>>> from topik.models import load_model
>>> model = load_model("test_data.json")
>>> model.get_top_words(10)
```

Viewing results

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

- List of top N words for each topic
- 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:

```
>>> model.get_top_words(topn=10)
```

```
>>> from topik.viz import Termite
>>> termite = Termite(lda.termite_data(n_topics), "Termite Plot")
>>> termite.plot(os.path.join(dir_path, 'termite.html'))
```

```
>>> from topik.viz import LDAvis
>>> raw_data = read_input("reviews", content_field=None)
>>> tokenized_corpus = raw_data.tokenize()
>>> n_topics = 10
>>> model = registered_models["LDA"](tokenized_corpus, n_topics)
>>> from topik.viz import plot_lda_vis
>>> plot_lda_vis(model.to_py_lda_vis())
```

Each model is free to implement its own additional outputs - check the class members for what might be available.

2.1.4 Development Guide

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

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

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

- __init__.py imports registered class dictionaries and functions from each folder
- a folder (python package) for each step
  - base???.py
    - abstract interface to be implemented by any concrete classes
    - register function that is used as a decorator to register classes
    - dictionary of registered classes for this step (global variable)
  - any concrete implementations of the abstract classes, each in their own .py file
  - __init__.py imports each of the concrete implementations, so that they are registered

External code can hook into the dictionary of registered methods using the appropriate register decorator function. 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 usage, it is probably easier to directly import classes from the folder structure. 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 class arguments doesn't work
>>> model = registered_models["LDA"](tokenized_data, 5)
```

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

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

2.1.5 topik package

Subpackages

topik.intermediaries package

Submodules

topik.intermediaries.digested_document_collection module

topik.intermediaries.persistence module This file handles the storage of data from loading and analysis.

More accurately, the files written and read from this file describe how to read/write actual data, such that the actual format of any data need not be tightly defined.

class `topik.intermediaries.persistence.`**`Persistor`**(*filename=None*)

Bases: `object`

`get_corpus_dict`()

`get_model_details`(*model_id*)

`list_available_models`()

`load_data`(*filename*)

`persist_data`(*filename*)

`store_corpus`(*data_dict*)

`store_model`(*model_id*, *model_dict*)

topik.intermediaries.raw_data module This file is concerned with providing a simple interface for data stored in Elasticsearch. The class(es) defined here are fed into the preprocessing step.

class `topik.intermediaries.raw_data.`**`CorpusInterface`**

Bases: `object`

`append_to_record`(*record_id*, *field_name*, *field_value*)

Used to store preprocessed output alongside input data.

Field name is destination. Value is processed value.

classmethod **`class_key`**()

Implement this method to return the string ID with which to store your class.

`filter_string`

`get_date_filtered_data`(*start*, *end*, *field*)

`get_generator_without_id`(*field=None*)

Returns a generator that yields field content without doc_id associate

`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 prcess. 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"

tokenize (*method='simple'*, *synchronous_wait=30*, ***kwargs*)
Convert data to lowercase; tokenize; create bag of words collection.

Output from this function is used as input to modeling steps.

raw_data: iterable corpus object containing the text to be processed. Each iteration call should return a new document's content.

tokenizer_method: string id of tokenizer to use. For keys, see topik.tokenizers.tokenizer_methods (which is a dictionary of classes)

kwargs: arbitrary dicionary of extra parameters. These are passed both to the tokenizer and to the vectorizer steps.

class `topik.intermediaries.raw_data.`**DictionaryCorpus** (*content_field*, *iterable=None*, *generate_id=True*, *reference_field=None*, *content_filter=None*)
Bases: `topik.intermediaries.raw_data.CorpusInterface`

append_to_record (*record_id*, *field_name*, *field_value*)

classmethod class_key ()

filter_string

get_date_filtered_data (*start*, *end*, *field='year'*)

get_field (*field=None*)
Get a different field to iterate over, keeping all other details.

get_generator_without_id (*field=None*)

import_from_iterable (*iterable*, *content_field*, *generate_id=True*)

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*, *saved_data=None*)

class `topik.intermediaries.raw_data.`**ElasticSearchCorpus** (*source*, *index*, *content_field*, *doc_type=None*, *query=None*, *iterable=None*, *filter_expression=''*, ***kwargs*)
Bases: `topik.intermediaries.raw_data.CorpusInterface`

append_to_record (*record_id*, *field_name*, *field_value*)

classmethod class_key ()

convert_date_field_and_reindex (*field*)

filter_string

get_date_filtered_data (*start*, *end*, *field='date'*)

get_field (*field=None*)
Get a different field to iterate over, keeping all other connection details.

get_generator_without_id (*field=None*)

import_from_iterable(*iterable*, *id_field='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.

id_field: 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*)

topik.intermediaries.raw_data.**load_persisted_corpus**(*filename*)

topik.intermediaries.raw_data.**register_output**(*cls*)

Module contents

topik.models package

Submodules

topik.models.lda module

class topik.models.lda.**LDA**(*corpus_input=None*, *ntopics=10*, *load_filename=None*, *binary_filename=None*, ***kwargs*)
Bases: *topik.models.model_base.TopicModelBase*

A high-level interface for an LDA (Latent Dirichlet Allocation) model.

Parameters **corpus_input** : CorpusBase-derived object

object fulfilling basic Corpus interface (preprocessed, tokenized text). see topik.intermediaries.tokenized_corpus for more info.

ntopics : int

Number of topics to model

load_filename : None or str

If not None, this (JSON) file is read to determine parameters of the model persisted to disk.

binary_filename : None or str

If not None, this file is loaded by Gensim to bring a disk-persisted model back into memory.

Examples

```
>>> raw_data = read_input('{}/test_data_json_stream.json'.format(test_data_path), "abstract")
>>> processed_data = raw_data.tokenize()  # preprocess returns a DigestedDocumentCollection
>>> model = LDA(processed_data, ntopics=3)
```

Attributes

corpus	(CorpusBase-derived object, tokenized)
model	(Gensim LdaModel instance)

get_model_name_with_parameters()

get_top_words(*topn*)

save(*filename*)

topik.models.model_base module

class `topik.models.model_base.`**TopicModelBase**

Bases: `object`

Abstract base class for topic models.

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

Attributes

_corpus	(topik.intermediaries.digested_document_collection.DigestedDocumentCollection-derived object) The input data for this model
_persistor	(topik.intermediaries.persistor.Persistor object) The object responsible for persisting the state of this model to disk. Persistor saves metadata that instructs load_model how to load the actual data.

get_model_name_with_parameters()

Abstract method. Primarily internal function, used to name configurations in persisted metadata for later retrieval.

get_top_words(*topn*)

Abstract method. Implementations should collect top n words per topic, translate indices/ids to words.

Returns list of lists of tuples:

- outer list: topics
- inner lists: length topn collection of (weight, word) tuples

save(*filename*, *saved_data*)

Abstract method. Persist the model metadata and data to disk.

Implementations should both save their important data do disk with some known keyword (perhaps as filename or server address details), and pass a dictionary to saved_data. The contents of this dictionary will be passed to the class' constructor as **kwargs.

Be sure to either call super(YourClass, self).save(filename, saved_data) or otherwise duplicate the base level of functionality here.

Parameters **filename** : str

The filename of the JSON file to be saved, containing model and corpus metadata that allow for reconstruction

saved_data : dict

Dictionary of metadata that will be fed to class __init__ method at load time. This should include such things as number of topics modeled, binary filenames, and any other relevant model parameters to recreate your current model.

termite_data(*topn_words=15*)

Generate the pandas dataframe input for the termite plot.

Parameters **topn_words** : int

number of words to include from each topic

Examples

```
>>> raw_data = read_input('{}/test_data_json_stream.json'.format(test_data_path), "abstract"
>>> processed_data = raw_data.tokenize()  # tokenize returns a DigestedDocumentCollection
>>> # must set seed so that we get same topics each run
>>> import random
>>> import numpy
>>> random.seed(42)
>>> numpy.random.seed(42)
>>> model = registered_models["LDA"](processed_data, ntopics=3)
>>> model.termite_data(5)
    topic    weight         word
0       0  0.005337           nm
1       0  0.005193         high
2       0  0.004622        films
3       0  0.004457       matrix
4       0  0.004194     electron
5       1  0.005109   properties
6       1  0.004654         size
7       1  0.004539  temperature
8       1  0.004499           nm
9       1  0.004248   mechanical
10      2  0.007994         high
11      2  0.006458           nm
12      2  0.005717         size
13      2  0.005399    materials
14      2  0.004734        phase
```

to_py_lda_vis()

topik.models.model_base.**load_model**(*filename, model_name*)

Loads a JSON file containing instructions on how to load model data.

Returns TopicModelBase-derived object

topik.models.model_base.**register_model**(*cls*)

Decorator function to register new model with global registry of models

topik.models.plsa module

class topik.models.plsa.**PLSA**(*corpus=None, ntopics=2, load_filename=None, binary_filename=None*)

Bases: *topik.models.model_base.TopicModelBase*

get_model_name_with_parameters()

get_top_words(*topn*)

inference(*doc, max_iter=100*)

post_prob_sim(*docd, q*)

save(*filename*)

train(*max_iter=100*)

Module contents

Submodules

topik.cli module

topik.readers module

topik.readers.**read_input**(*source*, *content_field*, *source_type='auto'*, *output_type='dictionary'*, *output_args=None*, *synchronous_wait=0*, ***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.

content_field : str

Which field contains your data to be analyzed. Hash of this is used as id.

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']

output_type : str

Internal format for handling user data. Current options are in the registered_outputs dictionary. Default is DictionaryCorpus class. Specify alternatives using string key from dictionary.

output_args : dict

Configuration to pass through to output

synchronous_wait : positive, real number

Time in seconds to wait for data to finish uploading to output (this happens asynchronously.) Only relevant for some output types ("elastic", not "dictionary")

kwargs : any other arguments to pass to input parsers

Returns iterable output object

Examples

```
>>> raw_data = read_input(
...         '{}/test_data_json_stream.json'.format(test_data_path),
...          content_field="abstract")
>>> id, text = next(iter(raw_data))
>>> 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.')
True
```

topik.run module

`topik.run.`**`run_model`**(*data_source*, *source_type='auto'*, *year_field=None*, *start_year=None*, *stop_year=None*, *content_field=None*, *tokenizer='simple'*, *n_topics=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'.

n_topics : 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.

output_file : bool

Generate a final summary csv file of your results. For each document: text, tokens, lda_probabilities and topic.

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

topik.tokenizers module

`topik.tokenizers.`**`collect_bigrams_and_trigrams`**(*collection*, *top_n=10000*, *min_length=1*, *min_bigram_freq=50*, *min_trigram_freq=20*, *stopwords=None*)

collects bigrams and trigrams from collection of documents. Input to collocation tokenizer.

bigrams are pairs of words that recur in the collection; trigrams are triplets.

Parameters collection : iterable of str

body of documents to examine

top_n : int

limit results to this many entries

min_length : int

Minimum length of any single word

min_bigram_freq : int

threshold of when to consider a pair of words as a recognized bigram

min_trigram_freq : int

threshold of when to consider a triplet of words as a recognized trigram

stopwords : None or iterable of str

Collection of words to ignore as tokens

Examples

```
>>> from topik.readers import read_input
>>> raw_data = read_input(
...                 '{}/test_data_json_stream.json'.format(test_data_path),
...                 content_field="abstract")
>>> bigrams, trigrams = collect_bigrams_and_trigrams(raw_data, min_bigram_freq=5, min_trigram_fr
>>> bigrams.pattern
u'(free standing|ac electrodeposition|centered cubic|spatial resolution|vapor deposition|wear re
>>> trigrams.pattern
u'(differential scanning calorimetry|face centered cubic|ray microanalysis analytical|physical v
```

`topik.tokenizers.`**`collect_entities`**(*collection*, *freq_min=2*, *freq_max=10000*)

Return noun phrases from collection of documents.

Parameters collection: Corpus-base derived object or iterable collection of raw text

freq_min: int

Minimum frequency of a noun phrase occurrences in order to retrieve it. Default is 2.

freq_max: int

Maximum frequency of a noun phrase occurrences in order to retrieve it. Default is 10000.

`topik.tokenizers.`**`tokenize_collocation`**(*text*, *patterns*, *min_length=1*, *stopwords=None*)

A text tokenizer that includes collocations(bigrams and trigrams).

A collocation is sequence of words or terms that co-occur more often than would be expected by chance. This function breaks a raw document up into tokens based on a pre-established collection of bigrams and trigrams. This collection is derived from a body of many documents, and must be obtained in a prior step using the collect_bigrams_and_trigrams function.

Uses nltk.collocations.TrigramCollocationFinder to find trigrams and bigrams.

Parameters **text** : str

A single document's text to be tokenized

patterns: tuple of compiled regex object to find n-grams

Obtained from collect_bigrams_and_trigrams function

min_length : int

Minimum length of any single word

stopwords : None or iterable of str

Collection of words to ignore as tokens

Examples

```
>>> from topik.readers import read_input
>>> id_documents = read_input('{}/test_data_json_stream.json'.format(test_data_path), content_fi
>>> patterns = collect_bigrams_and_trigrams(id_documents, min_bigram_freq=2, min_trigram_freq=2)
>>> id, doc_text = next(iter(id_documents))
>>> tokenized_text = tokenize_collocation(doc_text, patterns)
>>> tokenized_text
[u'transition_metal', u'oxides', u'considered', u'generation', u'materials', u'field', u'electro
```

`topik.tokenizers.`**`tokenize_entities`**(*text*, *entities*, *min_length=1*, *stopwords=None*)

A tokenizer that extracts noun phrases from text.

Requires that you first establish entities using the collect_entities function

Parameters **text** : str

A single document's text to be tokenized

entities : iterable of str

Collection of noun phrases, obtained from collect_entities function

min_length : int

Minimum length of any single word

stopwords : None or iterable of str

Collection of words to ignore as tokens

Examples

```
>>> from topik.readers import read_input
>>> id_documents = read_input('{}/test_data_json_stream.json'.format(test_data_path), "abstract"
>>> entities = collect_entities(id_documents)
>>> len(entities)
220
>>> i = iter(id_documents)
>>> _, doc_text = next(i)
>>> doc_text
u'Transition metal oxides are being considered as the next generation materials in field such as
>>> tokenized_text = tokenize_entities(doc_text, entities)
>>> tokenized_text
[u'transition']
```

topik.tokenizers.**tokenize_mixed**(*text*, *entities*, *min_length=1*, *stopwords=None*)
A text tokenizer that retrieves entities ('noun phrases') first and simple words for the rest of the text.

Parameters **text** : str

A single document's text to be tokenized

entities : iterable of str

Collection of noun phrases, obtained from collect_entities function

min_length : int

Minimum length of any single word

stopwords: None or iterable of str

Collection of words to ignore as tokens

Examples

```
>>> from topik.readers import read_input
>>> raw_data = read_input('{}/test_data_json_stream.json'.format(test_data_path), content_field=
>>> entities = collect_entities(raw_data)
>>> id, text = next(iter(raw_data))
>>> tokenized_text = tokenize_mixed(text, entities, min_length=3)
>>> tokenized_text
[u'transition', u'metal', u'oxides', u'generation', u'materials', u'tantalum', u'oxide', u'nanom
```

topik.tokenizers.**tokenize_simple**(*text*, *min_length=1*, *stopwords=None*)
A text tokenizer that simply lowercases, matches alphabetic characters and removes stopwords.

Parameters **text** : str

A single document's text to be tokenized

entities : iterable of str

Collection of noun phrases, obtained from collect_entities function

min_length : int

Minimum length of any single word

stopwords: None or iterable of str

Collection of words to ignore as tokens

Examples

```
>>> from topik.readers import read_input
>>> id_documents = read_input(
...                 '{}/test_data_json_stream.json'.format(test_data_path),
...                 content_field="abstract")
>>> id, doc_text = next(iter(id_documents))
>>> doc_text
u'Transition metal oxides are being considered as the next generation materials in field such as
>>> tokens = tokenize_simple(doc_text)
>>> tokens
[u'transition', u'metal', u'oxides', u'considered', u'generation', u'materials', u'field', u'ele
```

topik.utils module

topik.viz module

class `topik.viz.`**`Termite`**(*input_file*, *title*)
Bases: `object`

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

```
>>> termite = Termite("{}/termite.csv".format(test_data_path),
...                     "My lda results")
>>> termite.plot('my_termite.html')
```

`plot`(*output_file='termite.html'*)

`topik.viz.`**`plot_lda_vis`**(*model_data*)
Designed to work with to_py_lda_vis() in the model classes.

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