BigARTM Documentation

Release 1.0

Konstantin Vorontsov

February 15, 2017

Contents

Getting help

- Learn more about BigARTM from IPython Notebooks, NLPub.ru, MachineLearning.ru and several publications.
- Search for information in the archives of the bigartm-users mailing list, or post a question.
- Report bugs with BigARTM in our ticket tracker.
- Try the Q&A – it's got answers to many common questions.

CHAPTER 1

Introduction

Warning: Please note that this is a beta version of the BigARTM library which is still undergoing final testing before its official release. Should you encounter any bugs, lack of functionality or other problems with our library, please let us know immediately. Your help in this regard is greatly appreciated.

This is the documentation for the BigARTM library. BigARTM is a tool to infer topic models, based on a novel technique called Additive Regularization of Topic Models. This technique effectively builds multi-objective models by adding the weighted sums of regularizers to the optimization criterion. BigARTM is known to combine well very different objectives, including sparsing, smoothing, topics decorrelation and many others. Such combinations of regularizers significantly improves several quality measures at once almost without any loss of the perplexity.

Online. BigARTM never stores the entire text collection in the main memory. Instead the collection is split into small chunks called 'batches', and BigARTM always loads a limited number of batches into memory at any time.

Parallel. BigARTM can concurrently process several batches, and by doing so it substantially improves the throughput on multi-core machines. The library hosts all computation in several threads withing a single process, which enables efficient usage of shared memory across application threads.

Extensible API. BigARTM comes with an API in Python, but can be easily extended for all other languages that have an implementation of Google Protocol Buffers.

Cross-platform. BigARTM is known to be compatible with gcc, clang and the Microsoft compiler (VS 2012). We have tested our library on Windows, Ubuntu and Fedora.

Open source. BigARTM is released under the New BSD License. If you plan to use our library commercially, please beware that BigARTM depends on ZeroMQ. Please, make sure to review ZeroMQ license.

Acknowledgements. BigARTM project is supported by Russian Foundation for Basic Research (grants 14-07-00847, 14-07-00908, 14-07-31176), Skolkovo Institute of Science and Technology (project 081-R), Moscow Institute of Physics and Technology.

Partners


ANTIPLAGIAT
RESEARCH


bBridge

CHAPTER 2

Downloads

- **Windows**
  - Latest 64 bit release: BigARTM_v0.8.3_win64
  - Latest build from master branch: BigARTM_master_win64.7z (warning, use this with caution)
  - All previous releases are available at https://github.com/bigartm/bigartm/releases
- **Linux, Mac OS-X**

  To run BigARTM on Linux and Mac OS-X you need to clone BigARTM repository (https://github.com/bigartm/bigartm).
- **Docker container**

  On any OS it is possible to run BigARTM as docker container, as described in Installation as Docker container

CHAPTER 3

Installation

3.1 Installation for Windows users

3.1.1 Download

Download latest binary distribution of BigARTM from https://github.com/bigartm/bigartm/releases. Explicit download links can be found at Downloads section (64 bit only).

The distribution will contain pre-build binaries, command-line interface and BigARTM API for Python. The distribution also contains a simple dataset. More datasets in BigARTM-compatible format are available in the Downloads section.

You may also try BigARTM from pre-built docker container, as described in Installation as Docker container.

3.1.2 Use BigARTM from command line

Download the latest package from https://github.com/bigartm/bigartm/releases. Unpack it to `C:\BigARTM`. Open command line or power shell and change working directory to `C:\BigARTM\bin`. Now you may use BigARTM command line utility `bigartm.exe`. Go to BigARTM Command Line Utility for examples and further directions.

3.1.3 Configure BigARTM Python API

1. Install Python, for example from Anaconda distribution (https://www.continuum.io/downloads). You are free to choose `Python 2.7` or `Python 3`, both are supported with BigARTM. You have to choose 64 bit Python installation.
2. Install Python packages that BigARTM depend on.
   - `numpy >= 1.9.2` (skip this if you install via Anaconda)
   - `pandas >= 0.16.2` (skip this if you install via Anaconda)
   - `protobuf >= 3.0.0` (not included in Anaconda; install by running `pip install protobuf==3.0.0`)
   - `tqdm` (not included in Anaconda; install by running `pip install tqdm`)
3. Add `C:\BigARTM\bin` folder to your `PATH` system variable, and add `C:\BigARTM\python` to your `PYTHONPATH` system variable:

```
set PATH=%PATH%;C:\BigARTM\bin
set PYTHONPATH=%PYTHONPATH%;C:\BigARTM\Python
```

Remember to change `C:\BigARTM` if you unpack to another location.

4. Now you can used BigARTM from Python shell or from ipython notebook. Refer to Python Tutorial or Python Guide for examples, or to Python Interface for documentation.

If you are getting errors when configuring or using Python API, please refer to Troubleshooting chapter in Installation for Linux and Mac OS-X users. The list of issues is common between Windows and Linux. Normally you should install protobuf for python with `pip install`. An alternative way is to install it from the package that comes with BigARTM. This is described here.

3.2 Installation for Linux and Mac OS-X users

BigARTM had been tested on several Linux and MAC distributions, and it is known to work well on

- Ubuntu 16.04.1
- Linux Mint 18
- Ubuntu 14.04.5
- Linux Mint 17.3
- Arch Linux
- Manjaro
- Fedora 24
- openSUSE Leap 42.1

To install BigARTM you should build it from source code. Or, you may run BigARTM from pre-built docker container, as described in Installation as Docker container. If you are looking for old instructions, refer to Build C++ code on Linux.

3.2.1 Script to install BigARTM on Ubuntu

The following script is tested with Ubuntu 14.04. Scroll further below for more OS-specific insructinos.

```
# Step 1. Update and install dependencies
apt-get --yes update
apt-get --yes install git
apt-get --yes install make
apt-get --yes install cmake
apt-get --yes install build-essential
apt-get --yes install libboost-all-dev

# Step 2. Insall python packages
apt-get --yes install python-numpy
apt-get --yes install python-pandas
wget https://bootstrap.pypa.io/get-pip.py
python get-pip.py
pip install protobuf
pip install tqdm

# Step 3. Clone repository and build
git clone --branch=stable https://github.com/bigartm/bigartm.git
cd bigartm
mkdir build && cd build
cmake ..
```

```
make

# Step 4. Install BigARTM
make install
export ARTM_SHARED_LIBRARY=/usr/local/lib/libartm.so
```

Now you should be able to use BigARTM command line utility (try `bigartm --help`), or run BigARTM from python, like this: `import artm; print(artm.version());` `print(artm.ARTM(num_topics=10).info)`.

3.2.2 Step 1. Install system dependencies

Ubuntu, Linux Mint:

```
sudo apt-get install git build-essential libboost-all-dev
sudo apt-get install cmake  # For Ubuntu 16.04 and Linux Mint 18
```

For Ubuntu 14.04 and Linux Mint 17.3 ubstakk cmake from PPA:

```
sudo add-apt-repository ppa:george-edison55/cmake-3.x
sudo apt-get update && sudo apt-get install cmake
```

Arch Linux, Manjaro

Ensure that Python and base-devel packages are installed, and current user has root priveledges (e.g. can run sudo):

```
sudo pacman -S git boost cmake
```

Fedora

```
sudo dnf install gcc-c++ glibc-static libstdc++-static
sudo dnf install git boost boost-static cmake
```

openSUSE:

```
sudo zypper install gcc gcc-c++ glibc-devel-static git cmake
```

Currently openSUSE require to install Boost from sources:

```
sudo zypper install libbz2-devel python-devel
wget http://sourceforge.net/projects/boost/files/boost/1.60.0/boost_1_60_0.tar.gz
tar -xf boost_1_60_0.tar.gz
cd boost_1_60_0
./bootstrap.sh
./b2 link=static,shared cxxflags="-std=c++11 -fPIC"
sudo ./b2 install
```

Mac OS distributions

```
brew install git cmake boost
```

3.2.3 Step 2. Python packages

BigARTM require Python packages `numpy`, `pandas`, `protobuf` and `tqdm`. We recommend to install them with recent versino of `pip`.

If `pip` is not available, install system package `python-pip` or `python3-pip` depending on your prefered Python version. For Arch Linux and Manjaro use `python2-pip` and `python-pip`.

Verify that you use latest version of `pip`:

```
sudo pip2 install -U pip  # Python 2
sudo pip3 install -U pip  # Python 3
```

Then install required python packages:

```
sudo pip2 install -U numpy pandas protobuf==3.0.0 tqdm # Python 2
sudo pip3 install -U numpy pandas protobuf==3.0.0 tqdm # Python 3
```

3.2.4 Step 3. Build and install BigARTM library

```
git clone --branch=stable https://github.com/bigartm/bigartm.git
cd bigartm && mkdir build && cd build
```

Next step is to run `cmake`. The following options are available.

- `-DPYTHON=python3` - to use Python 3 instead of Python 2;
- `-DCMAKE_INSTALL_PREFIX=xxx` - for custom install location instead of default `/usr/local`;
- `-DBoost_USE_STATIC_LIBS=ON` — required on openSUSE.

Example:

```
cmake -DPYTHON=python3 -DCMAKE_INSTALL_PREFIX=/opt/bigartm ..
```

Now build and install the library:

```
make
sudo make install
```

3.2.5 Step 4. Register `libartm.so` / `libartm.dylib`

Register shared library `libartm.so` (or `libartm.dylib`):

```
echo /usr/local/lib | sudo tee /etc/ld.so.conf.d/artm.conf
sudo ldconfig
```

If you've installed to another location than `/usr/local` update the first command with new path.

As an alternative you may set up `ARTM_SHARED_LIBRARY` system variable

```
export ARTM_SHARED_LIBRARY=/usr/local/lib/libartm.so     # Linux / Unix
export ARTM_SHARED_LIBRARY=/usr/local/lib/libartm.dylib  # Mac
```

3.2.6 Step 5. Enjoy!!!

If the instructions above did not work for you please let us know, either create an issue or send e-mail to bigartm-users@googlegroups.com.

3.3 Installation as Docker container

On any OS it is possible to run BigARTM as docker container.

1. Get the image from DockerHub

```
docker pull ofrei/bigartm
```

2. Run CLI

```
docker run -t -i ofrei/bigartm bigartm
```

3. Try Python API (use `ipython2` or `ipython3` depending on which python version you prefer)

```
$ docker run -t -i ofrei/bigartm ipython2
Python 2.7.6 (default, Jun 22 2015, 17:58:13)
Type "copyright", "credits" or "license" for more information.

IPython 3.2.1 -- An enhanced Interactive Python.
?         -> Introduction and overview of IPython's features.
%quickref -> Quick reference.
help      -> Python's own help system.
object?   -> Details about 'object', use 'object??' for extra details.

In [1]: import artm

In [2]: model = artm.ARTM()
```

See bigartm-docker repo for more information.

CHAPTER 4

User's Guide

4.1 Input Data Formats and Datasets

- **Formats**

This page describes input data formats compatible with BigARTM. Currently all formats support Bag-of-words representation, meaning that all linguistic processing (lemmatization, tokenization, detection of n-grams, etc) needs to be done outside BigARTM.

1. Vowpal Wabbit is a single-format file, based on the following principles:
   - each document is depresented in a single line
   - all tokens are represented as strings (no need to convert them into an integer identifier)
   - token frequency defaults to `1.0`, and can be optionally specified after a colon (:)
   - namespaces (*`Batch.class_id`*) can be identified by a pipe (|)

   Example 1

```
doc1 Alpha Bravo:10 Charlie:5 |author Ola_Nordmann
doc2 Bravo:5 Delta Echo:3 |author Ivan_Ivanov
```

   Example 2

```
user123 |track-like track2 track5 track7 |track-play track1:10 track2:25 track3:2 track7:8 |trac
user345 |track-like track2 track5 track7 |track-play track1:10 track2:25 track3:2 track7:8 |trac
```

   - putting tokens in each document in their natural order without specifying token frequencies will lead to model with sequential texts (not Bag-of-words)

   Example 3

```
doc1 this text will be processed not as bag of words | Some_Author
```

2. UCI Bag-of-words format consists of two files - `vocab.*.txt` and `docword.*.txt`. The format of the `docword.*.txt` file is 3 header lines, followed by NNZ triples:

```
D
W
NNZ
docID wordID count
docID wordID count
...
docID wordID count
```

The file must be sorted on docID. Values of wordID must be unity-based (not zero-based). The format of the `vocab.*.txt` file is line containing wordID=n. Note that words must not have spaces or tabs. In `vocab.*.txt` file it is also possible to specify the namespace (*`Batch.class_id`*) for tokens, as it is shown in this example:

```
token1 @default_class
token2 custom_class
token3 @default_class
token4
```

Use space or tab to separate token from its class. Token that are not followed by class label automatically get ''@default_class'' as a label (see ''token4" in the example).

Unicode support. For non-ASCII characters save `vocab.*.txt` file in **UTF-8** format.

3. Batches (binary BigARTM-specific format).

   This is compact and efficient format, based on several protobuf messages in public BigARTM interface (*Batch*, *Item* and *Field*).

   - A batch is a collection of several items
   - An item is a collection of several fields
   - A field is a collection of pairs `(token_id, token_weight)`.

   The following example shows a Python code that generates a synthetic batch.

```
import artm.messages, random, uuid

num_tokens = 60
num_items = 100
batch = artm.messages.Batch()
batch.id = str(uuid.uuid4())
for token_id in range(0, num_tokens):
    batch.token.append('token' + str(token_id))

for item_id in range(0, num_items):
    item = batch.item.add()
    item.id = item_id
    field = item.field.add()
    for token_id in range(0, num_tokens):
        field.token_id.append(token_id)
        background_count = random.randint(1, 5) if (token_id >= 40) else 0
        topical_count = 10 if (token_id < 40) and ((token_id % 10) == (item_id % 10)) else 0
        field.token_weight.append(background_count + topical_count)
```

Note that the batch has its local dictionary, `batch.token`. This dictionary which maps `token_id` into the actual token. In order to create a batch from textual files involve one needs to find all distinct words, and map them into sequential indices.

`batch.id` must be set to a unique GUID in a format of `00000000-0000-0000-0000-000000000000`.

- **Datasets**

  Download one of the following datasets to start experimenting with BigARTM. Note that `docword.*` and `vocab.*` files indicate `UCI BOW` format, while `vw.*` file indicate `Vowpal Wabbit` format.

Task	Source	#Words	#Items	Files
kos	UCI	6906	3430	– docword.kos.txt.gz (1 MB) – vocab.kos.txt (54 KB)
nips	UCI	12419	1500	– docword.nips.txt.gz (2.1 MB) – vocab.nips.txt (98 KB)
enron	UCI	28102	39861	– docword.enron.txt.gz (11.7 MB) – vocab.enron.txt (230 KB)
nytimes	UCI	102660	300000	– docword.nytimes.txt.gz (223 MB) – vocab.nytimes.txt (1.2 MB)
pubmed	UCI	141043	8200000	– docword.pubmed.txt.gz (1.7 GB) – vocab.pubmed.txt (1.3 MB)
wiki	Gensim	100000	3665223	– vw.wiki-en.txt.zip (1.8 GB)
wiki_enru	Wiki	196749	216175	– vw.wiki_enru.txt.zip (285 MB)
eurlex	eurlex	19800	21000	– vw.eurlex.txt.zip (13 MB) – vw.eurlex-test.txt.zip (13 MB)
lastfm	lastfm		1k, 360k	– vw.lastfm_1k.txt.zip (100 MB) – vw.lastfm_360k.txt.zip (330 MB)
mmro	mmro	7805	1061	–

4.2 BigARTM Command Line Utility

This document provides an overview of `bigartm` command-line utility shipped with BigARTM.

For a detailed description of `bigartm` command line interface refer to bigartm.exe notebook (in Russian).

In brief, you need to download some input data (a textual collection represented in bag-of-words format). We recommend to download sample colections in **vowpal wabbit** format by links provided in Downloads section of the tutorial. Then you can use `bigartm` as described by `bigartm --help`. You may also get more information about builtin regularizers by typing `bigartm --help --regularizer`.

```
BigARTM v0.8.2 - library for advanced topic modeling (http://bigartm.org):

Input data:
  -c [ --read-vw-corpus ] arg           Raw corpus in Vowpal Wabbit format
  -d [ --read-uci-docword ] arg         docword file in UCI format
  -v [ --read-uci-vocab ] arg           vocab file in UCI format
  --read-cooc arg                       read co-occurrences format
  --batch-size arg (=500)               number of items per batch
  --use-batches arg                     folder with batches to use

Dictionary:
  --dictionary-min-df arg               filter out tokens present in less than
                                        N documents / less than P% of documents
  --dictionary-max-df arg               filter out tokens present in less than
                                        N documents / less than P% of documents
  --dictionary-size arg (=0)            limit dictionary size by filtering out
                                        tokens with high document frequency
  --use-dictionary arg                  filename of binary dictionary file to
                                        use

Model:
  --load-model arg                      load model from file before processing
  -t [ --topics ] arg (=16)             number of topics
  --use-modality arg                    modalities (class_ids) and their
                                        weights
  --predict-class arg                   target modality to predict by theta
                                        matrix

Learning:
  -p [ --num-collection-passes ] arg (=0)
                                        number of outer iterations (passes
                                        through the collection)
  --num-document-passes arg (=10)       number of inner iterations (passes
                                        through the document)
  --update-every arg (=0)               [online algorithm] requests an update
                                        of the model after update_every
                                        document
  --tau0 arg (=1024)                    [online algorithm] weight option from
                                        online update formula
  --kappa arg (=0.699999988)            [online algorithm] exponent option from
                                        online update formula
  --reuse-theta                         reuse theta between iterations
  --regularizer arg                     regularizers (SmoothPhi,SparsePhi,Smoot
                                        hTheta,SparseTheta,Decorrelation)
  --threads arg (=-1)                   number of concurrent processors
                                        (default: auto-detect)
  --async                               invoke asynchronous version of the
```

```
                                        online algorithm

Output:
  --save-model arg                      save the model to binary file after
                                        processing
  --save-batches arg                    batch folder
  --save-dictionary arg                 filename of dictionary file
  --write-model-readable arg            output the model in a human-readable
                                        format
  --write-dictionary-readable arg       output the dictionary in a
                                        human-readable format
  --write-predictions arg               write prediction in a human-readable
                                        format
  --write-class-predictions arg         write class prediction in a
                                        human-readable format
  --write-scores arg                    write scores in a human-readable format
  --write-vw-corpus arg                 convert batches into plain text file in
                                        Vowpal Wabbit format
  --force                               force overwrite existing output files
  --csv-separator arg (=;)              columns separator for
                                        --write-model-readable and
                                        --write-predictions. Use \t or TAB to
                                        indicate tab.
  --score-level arg (=2)                score level (0, 1, 2, or 3
  --score arg                           scores (Perplexity, SparsityTheta,
                                        SparsityPhi, TopTokens, ThetaSnippet,
                                        or TopicKernel)
  --final-score arg                     final scores (same as scores)

Other options:
  -h [ --help ]                         display this help message
  --rand-seed arg                       specify seed for random number
                                        generator, use system timer when not
                                        specified
  --guid-batch-name                     applies to save-batches and indicate
                                        that batch names should be guids (not
                                        sequential codes)
  --response-file arg                   response file
  --paused                              start paused and waits for a keystroke
                                        (allows to attach a debugger)
  --disk-cache-folder arg               disk cache folder
  --disable-avx-opt                     disable AVX optimization (gives similar
                                        behavior of the Processor component to
                                        BigARTM v0.5.4)
  --time-limit arg (=0)                 limit execution time in milliseconds
  --log-dir arg                         target directory for logging
                                        (GLOG_log_dir)
  --log-level arg                       min logging level (GLOG_minloglevel;
                                        INFO=0, WARNING=1, ERROR=2, and
                                        FATAL=3)

Examples:

* Download input data:
  wget https://s3-eu-west-1.amazonaws.com/artm/docword.kos.txt
  wget https://s3-eu-west-1.amazonaws.com/artm/vocab.kos.txt
  wget https://s3-eu-west-1.amazonaws.com/artm/vw.mmro.txt
  wget https://s3-eu-west-1.amazonaws.com/artm/vw.wiki-enru.txt.zip
```

```
* Parse docword and vocab files from UCI bag-of-word format; then fit topic model with 20 topics:
  bigartm -d docword.kos.txt -v vocab.kos.txt -t 20 --num_collection_passes 10

* Parse VW format; then save the resulting batches and dictionary:
  bigartm --read-vw-corpus vw.mmro.txt --save-batches mmro_batches --save-dictionary mmro.dict

* Parse VW format from standard input; note usage of single dash '-' after --read-vw-corpus:
  cat vw.mmro.txt | bigartm --read-vw-corpus - --save-batches mmro2_batches --save-dictionary mmro2.d

* Re-save batches back into VW format:
  bigartm --use-batches mmro_batches --write-vw-corpus vw.mmro.txt

* Parse only specific modalities from VW file, and save them as a new VW file:
  bigartm --read-vw-corpus vw.wiki-enru.txt --use-modality @russian --write-vw-corpus vw.wiki-ru.txt

* Load and filter the dictionary on document frequency; save the result into a new file:
  bigartm --use-dictionary mmro.dict --dictionary-min-df 5 dictionary-max-df 40% --save-dictionary mm

* Load the dictionary and export it in a human-readable format:
  bigartm --use-dictionary mmro.dict --write-dictionary-readable mmro.dict.txt

* Use batches to fit a model with 20 topics; then save the model in a binary format:
  bigartm --use-batches mmro_batches --num_collection_passes 10 -t 20 --save-model mmro.model

* Load the model and export it in a human-readable format:
  bigartm --load-model mmro.model --write-model-readable mmro.model.txt

* Load the model and use it to generate predictions:
  bigartm --read-vw-corpus vw.mmro.txt --load-model mmro.model --write-predictions mmro.predict.txt

* Fit model with two modalities (@default_class and @target), and use it to predict @target label:
  bigartm --use-batches <batches> --use-modality @default_class,@target --topics 50 --num_collection_
  bigartm --use-batches <batches> --use-modality @default_class,@target --topics 50 --load-model mode
          --write-predictions pred.txt --csv-separator=tab
          --predict-class @target --write-class-predictions pred_class.txt --score ClassPrecision

* Fit simple regularized model (increase sparsity up to 60-70%):
  bigartm -d docword.kos.txt -v vocab.kos.txt --dictionary-max-df 50% --dictionary-min-df 2
          --num_collection_passes 10 --batch-size 50 --topics 20 --write-model-readable model.txt
          --regularizer "0.05 SparsePhi" "0.05 SparseTheta"

* Fit more advanced regularize model, with 10 sparse objective topics, and 2 smooth background topics
  bigartm -d docword.kos.txt -v vocab.kos.txt --dictionary-max-df 50% --dictionary-min-df 2
          --num_collection_passes 10 --batch-size 50 --topics obj:10;background:2 --write-model-reada
          --regularizer "0.05 SparsePhi #obj"
          --regularizer "0.05 SparseTheta #obj"
          --regularizer "0.25 SmoothPhi #background"
          --regularizer "0.25 SmoothTheta #background"

* Upgrade batches in the old format (from folder 'old_folder' into 'new_folder'):
  bigartm --use-batches old_folder --save-batches new_folder

* Configure logger to output into stderr:
  tset GLOG_logtostderr=1 & bigartm -d docword.kos.txt -v vocab.kos.txt -t 20 --num_collection_passes
```

Additional information about regularizers:

```
>bigartm.exe --regularizer --help
List of regularizers available in BigARTM CLI:

        --regularizer "tau SmoothTheta #topics"
        --regularizer "tau SparseTheta #topics"
        --regularizer "tau SmoothPhi #topics @class_ids !dictionary"
        --regularizer "tau SparsePhi #topics @class_ids !dictionary"
        --regularizer "tau Decorrelation #topics @class_ids"
        --regularizer "tau TopicSelection #topics"
        --regularizer "tau LabelRegularization #topics @class_ids !dictionary"
        --regularizer "tau ImproveCoherence #topics @class_ids !dictionary"
        --regularizer "tau Biterms #topics @class_ids !dictionary"

List of regularizers available in BigARTM, but not exposed in CLI:

        --regularizer "tau SpecifiedSparsePhi"
        --regularizer "tau SmoothPtdw"
        --regularizer "tau HierarchySparsingTheta"

If you are interested to see any of these regularizers in BigARTM CLI please send a message to
        bigartm-users@googlegroups.com.

By default all regularizers act on the full set of topics and modalities.
To limit action onto specific set of topics use hash sign (#), followed by
list of topics (for example, #topic1;topic2) or topic groups (#obj).
Similarly, to limit action onto specific set of class ids use at sign (@),
by the list of class ids (for example, @default_class).
Some regularizers accept a dictionary. To specify the dictionary use exclamation mark (!),
followed by the path to the dictionary(.dict file in your file system).
Depending on regularizer the dictinoary can be either optional or required.
Some regularizers expect an dictinoary with tokens and their frequencies;
Other regularizers expect an dictinoary with tokens co-occurencies;
For more information about regularizers refer to wiki-page:

        https://github.com/bigartm/bigartm/wiki/Implemented-regularizers

To get full help run `bigartm --help` without --regularizer switch.
```

4.3 Python Tutorial

For more detailes about `artm.LDA`, `artm.ARTM`, `artm.BatchVectorizer` and `artm.Dictionary` see Python Interface and Python Guide.

LDA (most simple)

`artm.LDA` was designed for non-advanced users with minimal knowledge about topic modeling and ARTM. It is a cutted version of artm.ARTM model with pre-defined scores and regularizers. `artm.LDA` has enough abilities for fitting the LDA with regularizers of smoothing/sparsing of Φ and Θ matrices with offline or online algorithms. Also it can compute scores of perplexity, matrices sparsities and most probable tokens in each topic, and return the whole resulting matrices.

Let's make a simple model experiment with 'kos' collection in UCI format (see Input Data Formats and Datasets). You will need two files: `docword.kos.txt vocab.kos.txt`. Both these files should be put into the same directory with this notebook. Let's import the artm module, create BatchVectorizer and run dictionary gathering inside of it (if you are interested in the detailes, you need to read information from given links):

```
import artm

batch_vectorizer = artm.BatchVectorizer(data_path='.', data_format='bow_uci',
                                        collection_name='kos', target_folder='kos_batches')
```

Now let's create the model by defining the number of topics, number of passes through each document, hyperparameters of smoothing of Φ and Θ and dictionary to use. Also let's ask thr model to store Θ matrix to have an ability to look at it in the future. Also you can set here the num_processors parameter, which defines the number of threads to be used on your machine for parallelizing the computing:

```
lda = artm.LDA(num_topics=15, alpha=0.01, beta=0.001, cache_theta=True,
               num_document_passes=5, dictionary=batch_vectorizer.dictionary)
```

Well, let's run the learning process using offline algorithm:

```
lda.fit_offline(batch_vectorizer=batch_vectorizer, num_collection_passes=10)
```

That's all, the fitting is over. You can iterate this process, edit model parameters etc. Now you are able to look at the results of modeling. For instance, let's look at final values of matrices sparsities:

```
lda.sparsity_phi_last_value
lda.sparsity_theta_last_value
```

Or at all values of perplexity (e.g. on each collection pass):

```
lda.perplexity_value
```

You can see the most probable tokens in each topic, at last. The are returned as list of lists of strings (each internal list correspondes to on topic by order). Let's output them with pre-formatting:

```
top_tokens = lda.get_top_tokens(num_tokens=10)
for i, token_list in enumerate(top_tokens):
    print 'Topic #{0}: {1}'.format(i, token_list)
```

To get the matrices you can use th following calls:

```
phi = lda.phi_
theta = lda.get_theta()
```

Here's two more abilities of artm.LDA.

At first, it is the ability to create Θ matrix for new documents after the model was fitted:

```
batch_vectorizer = artm.BatchVectorizer(data_path='kos_batches_test')
theta_test = lda.transform(batch_vectorizer=test_batch_vectorizer)
```

Secondly, in the case, when you need a custom regularization of each topic in Φ matrix, you need to set beta a list instead of scalar value. The list should have th length equal to the number of topics, and then each topic will be regularized with corresponding coefficient:

```
beta = [0.1] * num_topics  # change as you need
lda = artm.LDA(num_topics=15, alpha=0.01, beta=beta, num_document_passes=5,
               dictionary=batch_vectorizer.dictionary, cache_theta=True)
```

ARTM

This is a simple example of usage of `artm.ARTM`, a full-power Python API for BigARTM library. Let's learn two topic models of text collections, ARTM and PLSA, and compare them.

One of the important measures is the perplexity. Nevertheless it's not the only way to chech the quality of the model learning. The list of implemented scores can be found in Scores Description and there interfaces are described in

Scores. We'll use perplexity, sparsities of Φ and Θ, topic kernel scores (the higher values of sparsities and average purity and contrast means the more interpretable model).

The goal of the experiment is to learn the ARTM model in the way to obtain better values of sparsities and kernel characteristics, than in PLSA, without significant decline of the perplexity.

The main tool to control the learning process is the regularization. The list of currently implemented regularizers can be found in Regularizers Description and there interfaces are described in Regularizers. We will use SmoothSparsePhi, SmoothSparseTheta and DecorrelationPhi regularizers. ARTM without the regularization corresponds the PLSA model.

Let's use the same 'kos' collection, that was described above. At first let's import all necessary modules (make sure you have the BigARTM Python API in your PATH variable):

```
%matplotlib inline
import glob
import os
import matplotlib.pyplot as plt

import artm
```

Library Python API similarly to scikit-learn algorithms represents input data in the form of one class called `BatchVectorizer`. This class object get batches or UCI / VW files or n_{dw} matrix as inputs and is used as input parameter in all methods. If the given data is not batches, the object will create them and store to disk.

So let's create the object of `artm.BatchVectorizer`:

```
batch_vectorizer = None
if len(glob.glob(os.path.join('kos', '*.batch'))) < 1:
    batch_vectorizer = artm.BatchVectorizer(data_path='', data_format='bow_uci',
                                            collection_name='kos', target_folder='kos')
else:
    batch_vectorizer = artm.BatchVectorizer(data_path='kos', data_format='batches')
```

ARTM is a class, that represents BigARTM Python API. Allows to use almost all library abilities in scikit-learn style. Let's create two topic models for our experiments. The most important parameter of the model is the number of topics. Optionally the user can define the list of regularizers and quality measures (scores) to be used in this model. This step can be done later. Note, that each model defines its own namespace for names of regularizers and scores.

```
dictionary = batch_vectorizer.dictionary
topic_names = ['topic_{}'.format(i) for i in xrange(15)]

model_plsa = artm.ARTM(topic_names=topic_names, cache_theta=True,
                       scores=[artm.PerplexityScore(name='PerplexityScore',
                                                    dictionary=dictionary)])

model_artm = artm.ARTM(topic_names=topic_names, cache_theta=True,
                       scores=[artm.PerplexityScore(name='PerplexityScore',
                                                    dictionary=dictionary)],
                       regularizers=[artm.SmoothSparseThetaRegularizer(name='SparseTheta',
                                                                       tau=-0.15)])
```

Dictionary is the object of BigARTM, containing the information about the collection (vocabulary, different counters and values, linked to tokens). Provided dictionary will be used for Φ matrix initialization. It means:

- the Φ matrix with the name 'pwt' will be created with numbers of rows and columns corresponding the numbers of tokens in the dictionary and topics in the model;
- this matrix will be filled with random values from (0, 1) and normalized.

Matrix will be initialized during first call of `ARTM.fit_offline()` or `ARTM.fit_online()`.

As it was said earlier, ARTM provides the ability to use all the scores of BigARTM. Once the score was included into model, the model will save all its values, obtained at the time of each Φ matrix update. Let's add the scores we need for our experiment (only ones, missed in the constructors):

```
model_plsa.scores.add(artm.SparsityPhiScore(name='SparsityPhiScore'))
model_plsa.scores.add(artm.SparsityThetaScore(name='SparsityThetaScore'))
model_plsa.scores.add(artm.TopicKernelScore(name='TopicKernelScore',
                                            probability_mass_threshold=0.3))

model_artm.scores.add(artm.SparsityPhiScore(name='SparsityPhiScore'))
model_artm.scores.add(artm.SparsityThetaScore(name='SparsityThetaScore'))
model_artm.scores.add(artm.TopicKernelScore(name='TopicKernelScore',
                                            probability_mass_threshold=0.3))
```

Now we'll do the same thing with the regularizers for artm_model (let's set their start coefficients of the regularization, these values can be changed later):

```
model_artm.regularizers.add(artm.SmoothSparsePhiRegularizer(name='SparsePhi', tau=-0.1))
model_artm.regularizers.add(artm.DecorrelatorPhiRegularizer(name='DecorrelatorPhi', tau=1.5e+5))
```

Now we'll try to learn the model in offline mode (e.g. with one Φ matrix update during one path through the whole collection). Let's start with 15 passes:

```
model_plsa.num_document_passes = 1
model_artm.num_document_passes = 1

model_plsa.fit_offline(batch_vectorizer=batch_vectorizer, num_collection_passes=15)
model_artm.fit_offline(batch_vectorizer=batch_vectorizer, num_collection_passes=15)
```

Let's check the results of first part of learning process by comparing the values of scores of both models:

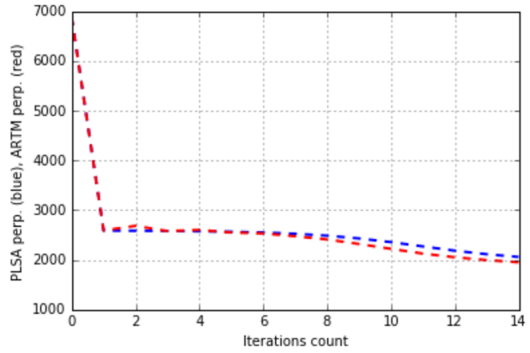
```
def print_measures(model_plsa, model_artm):
    print 'Sparsity Phi: {0:.3f} (PLSA) vs. {1:.3f} (ARTM)'.format(
        model_plsa.score_tracker['SparsityPhiScore'].last_value,
        model_artm.score_tracker['SparsityPhiScore'].last_value)

    print 'Sparsity Theta: {0:.3f} (PLSA) vs. {1:.3f} (ARTM)'.format(
        model_plsa.score_tracker['SparsityThetaScore'].last_value,
        model_artm.score_tracker['SparsityThetaScore'].last_value)

    print 'Kernel contrast: {0:.3f} (PLSA) vs. {1:.3f} (ARTM)'.format(
        model_plsa.score_tracker['TopicKernelScore'].last_average_contrast,
        model_artm.score_tracker['TopicKernelScore'].last_average_contrast)

    print 'Kernel purity: {0:.3f} (PLSA) vs. {1:.3f} (ARTM)'.format(
        model_plsa.score_tracker['TopicKernelScore'].last_average_purity,
        model_artm.score_tracker['TopicKernelScore'].last_average_purity)

    print 'Perplexity: {0:.3f} (PLSA) vs. {1:.3f} (ARTM)'.format(
        model_plsa.score_tracker['PerplexityScore'].last_value,
        model_artm.score_tracker['PerplexityScore'].last_value)

    plt.plot(xrange(model_plsa.num_phi_updates),
             model_plsa.score_tracker['PerplexityScore'].value, 'b--',
             xrange(model_artm.num_phi_updates),
             model_artm.score_tracker['PerplexityScore'].value, 'r--', linewidth=2)
    plt.xlabel('Iterations count')
    plt.ylabel('PLSA perp. (blue), ARTM perp. (red)')
    plt.grid(True)
```

```
    plt.show()

print_measures(model_plsa, model_artm)
```

`artm.ScoreTracker` is an object in model, that allows to retrieve values of your scores. The detailed information can be found in Score Tracker.

The call will have the following result:

```
Sparsity Phi: 0.000 (PLSA) vs. 0.469 (ARTM)
Sparsity Theta: 0.000 (PLSA) vs. 0.001 (ARTM)
Kernel contrast: 0.466 (PLSA) vs. 0.525 (ARTM)
Kernel purity: 0.215 (PLSA) vs. 0.359 (ARTM)
Perplexity: 2058.027 (PLSA) vs. 1950.717 (ARTM)
```



We can see, that we have an improvement of sparsities and kernel measures, and the downgrade of the perplexion isn't big. Let's try to increase the absolute values of regularization coefficients:

```
model_artm.regularizers['SparsePhi'].tau = -0.2
model_artm.regularizers['SparseTheta'].tau = -0.2
model_artm.regularizers['DecorrelatorPhi'].tau = 2.5e+5
```

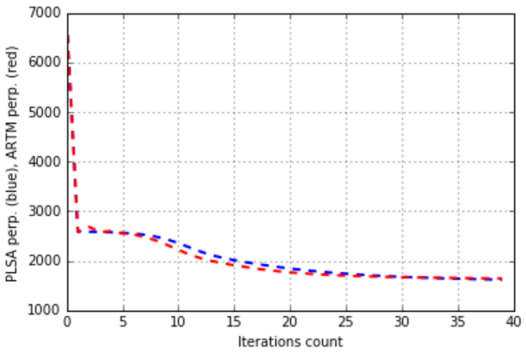
Besides that let's include into each model the `artm.TopTokenScore` measure, which allows to look at the most probable tokens in each topic:

```
model_plsa.scores.add(artm.TopTokensScore(name='TopTokensScore', num_tokens=6))
model_artm.scores.add(artm.TopTokensScore(name='TopTokensScore', num_tokens=6))
```

We'll continue the learning process with 25 passes through the collection, and than will look at the values of the scores:

```
model_plsa.fit_offline(batch_vectorizer=batch_vectorizer, num_collection_passes=25)
model_artm.fit_offline(batch_vectorizer=batch_vectorizer, num_collection_passes=25)

print_measures(model_plsa, model_artm)
```

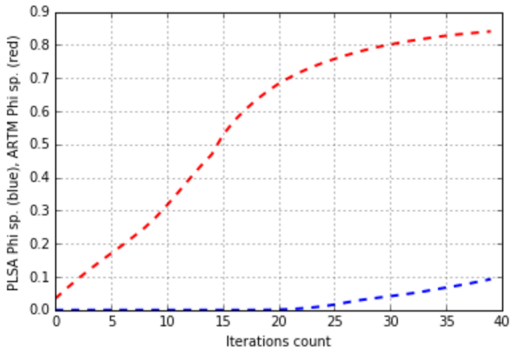
```
Sparsity Phi: 0.093 (PLSA) vs. 0.841 (ARTM)
Sparsity Theta: 0.000 (PLSA) vs. 0.023 (ARTM)
Kernel contrast: 0.640 (PLSA) vs. 0.740 (ARTM)
Kernel purity: 0.674 (PLSA) vs. 0.822 (ARTM)
Perplexity: 1619.031 (PLSA) vs. 1644.220 (ARTM)
```

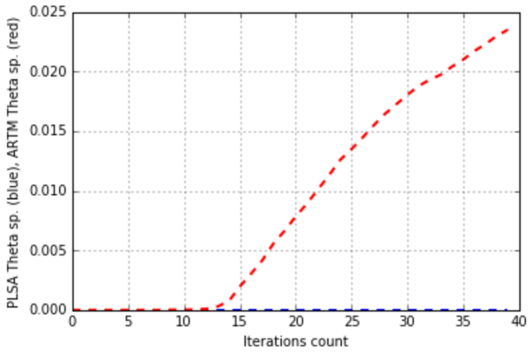


Becides let's plot the changes of matrices sparsities by iterations:

```
plt.plot(xrange(model_plsa.num_phi_updates),
         model_plsa.score_tracker['SparsityPhiScore'].value, 'b--',
         xrange(model_artm.num_phi_updates),
         model_artm.score_tracker['SparsityPhiScore'].value, 'r--', linewidth=2)

plt.xlabel('Iterations count')
plt.ylabel('PLSA Phi sp. (blue), ARTM Phi sp. (red)')
plt.grid(True)
plt.show()

plt.plot(xrange(model_plsa.num_phi_updates),
         model_plsa.score_tracker['SparsityThetaScore'].value, 'b--',
         xrange(model_artm.num_phi_updates),
         model_artm.score_tracker['SparsityThetaScore'].value, 'r--', linewidth=2)

plt.xlabel('Iterations count')
plt.ylabel('PLSA Theta sp. (blue), ARTM Theta sp. (red)')
plt.grid(True)
plt.show()
```

The output:





It seems that achieved result is enough. The regularization helped us to improve all scores with quite small perplexity downgrade. Let's look at top-tokens:

```
for topic_name in model_plsa.topic_names:
    print topic_name + ': ',
    print model_plsa.score_tracker['TopTokensScore'].last_tokens[topic_name]
```

```
topic_0:  [u'year', u'tax', u'jobs', u'america', u'president', u'issues']
topic_1:  [u'people', u'war', u'service', u'military', u'rights', u'vietnam']
topic_2:  [u'november', u'electoral', u'account', u'polls', u'governor', u'contact']
topic_3:  [u'republican', u'gop', u'senate', u'senator', u'south', u'conservative']
topic_4:  [u'people', u'time', u'country', u'speech', u'talking', u'read']
topic_5:  [u'dean', u'democratic', u'edwards', u'primary', u'kerry', u'clark']
topic_6:  [u'state', u'party', u'race', u'candidates', u'candidate', u'elections']
topic_7:  [u'administration', u'president', u'years', u'bill', u'white', u'cheney']
topic_8:  [u'campaign', u'national', u'media', u'local', u'late', u'union']
topic_9:  [u'house', u'million', u'money', u'republican', u'committee', u'delay']
topic_10:  [u'republicans', u'vote', u'senate', u'election', u'democrats', u'house']
topic_11:  [u'iraq', u'war', u'american', u'iraqi', u'military', u'intelligence']
topic_12:  [u'kerry', u'poll', u'percent', u'voters', u'polls', u'numbers']
topic_13:  [u'news', u'time', u'asked', u'political', u'washington', u'long']
topic_14:  [u'bush', u'general', u'bushs', u'kerry', u'oct', u'states']
```

```
for topic_name in model_artm.topic_names:
    print topic_name + ': ',
    print model_artm.score_tracker['TopTokensScore'].last_tokens[topic_name]
```

```
topic_0:  [u'party', u'political', u'issue', u'tax', u'america', u'issues']
topic_1:  [u'people', u'military', u'official', u'officials', u'service', u'public']
topic_2:  [u'electoral', u'governor', u'account', u'contact', u'ticket', u'experience']
topic_3:  [u'gop', u'convention', u'senator', u'debate', u'south', u'sen']
topic_4:  [u'country', u'speech', u'bad', u'read', u'end', u'talking']
topic_5:  [u'democratic', u'dean', u'john', u'edwards', u'primary', u'clark']
```

```
topic_6:  [u'percent', u'race', u'candidates', u'candidate', u'win', u'nader']
topic_7:  [u'administration', u'years', u'white', u'year', u'bill', u'jobs']
topic_8:  [u'campaign', u'national', u'media', u'press', u'local', u'ads']
topic_9:  [u'house', u'republican', u'million', u'money', u'elections', u'district']
topic_10:  [u'november', u'poll', u'senate', u'republicans', u'vote', u'election']
topic_11:  [u'iraq', u'war', u'american', u'iraqi', u'security', u'united']
topic_12:  [u'bush', u'kerry', u'general', u'president', u'voters', u'bushs']
topic_13:  [u'time', u'news', u'long', u'asked', u'washington', u'political']
topic_14:  [u'state', u'states', u'people', u'oct', u'fact', u'ohio']
```

We can see, that topics are approximatelly equal in terms of interpretability, but they are more different in ARTM.

Let's extract the Φ matrix as `pandas.DataFrame` and print it (to do this operation with more options use `ARTM.get_phi()`):

```
print model_artm.phi_
```

```
                  topic_0   topic_1   topic_2   topic_3   topic_4   topic_5  \
parentheses      0.000000  0.000000  0.000000  0.000000  0.000000  0.000277
opinion          0.000000  0.000000  0.000000  0.000000  0.000000  0.000000
attitude         0.000000  0.000000  0.000000  0.000000  0.000000  0.000000
held             0.000000  0.000385  0.000000  0.000000  0.000000  0.000000
impeachment      0.000000  0.000115  0.000000  0.000000  0.000000  0.000000
...                   ...       ...       ...       ...       ...       ...
aft              0.000000  0.000000  0.000000  0.000000  0.000000  0.000419
spindizzy        0.000000  0.000000  0.000647  0.000000  0.000000  0.000000
barnes           0.000000  0.000000  0.000000  0.000792  0.000000  0.000000
barbour          0.000000  0.000000  0.000000  0.000000  0.000000  0.000000
kroll            0.000000  0.000317  0.000000  0.000000  0.000000  0.000000

                  topic_12      topic_13  topic_14
parentheses      0.000000  0.000000e+00  0.000000
opinion          0.002908  0.000000e+00  0.000346
attitude         0.000000  0.000000e+00  0.000000
held             0.000526  0.000000e+00  0.000000
impeachment      0.000000  0.000000e+00  0.000396
...                   ...           ...       ...
aft              0.000000  0.000000e+00  0.000000
spindizzy        0.000000  0.000000e+00  0.000000
barnes           0.000000  0.000000e+00  0.000000
barbour          0.000000  0.000000e+00  0.000451
kroll            0.000000  0.000000e+00  0.000000
```

You can additionally extract Θ matrix and print it:

```
theta_matrix = model_artm.get_theta()
print theta_matrix
```

The model can be used to find θ_d vectors for new documents via `ARTM.transform()` method:

```
test_batch_vectorizer = artm.BatchVectorizer(data_format='batches',
                                             data_path='kos_test',
                                             batches=['test_docs.batch'])
test_theta_matrix = model_artm.transform(batch_vectorizer=test_batch_vectorizer)
```

Topic modeling task has an infinite set of solutions. It gives us a freedom in our choice. Regularizers give an opportunity to get the result, that satisfacts several criteria (such as sparsity, interpretability) at the same time.

Showen example is a demonstrative one, one can choose more flexible strategies of regularization to get better result. The experiments with other, bigger collection can be proceeded in the same way as it was described above.

See Python Guide for further reading, as it was mentioned above.

4.4 Python Guide

4.4.1 1. Loading Data: BatchVectorizer and Dictionary

Detailed description of all parameters and methods of BigARTM Python API classes can be found in Python Interface.

- **BatchVectorizer**:

Before starting modeling we need to convert you data in the library format. At first you need to read about supporting formats for source data in Input Data Formats and Datasets. It's your task to prepare your data in one of these formats. As you had transformed your data into one of source formats, you can convert them in the BigARTM internal format (batches of documents) using `BatchVectorizer` class object.

Really you have one more simple way to process your collection, if it is not too big and you don't need to store it in batches. To use it you need to archive two variables: `numpy.ndarray` `n_wd` with n_{wd} counters and corresponding Python dict with vocabulary (key - index of `numpy.ndarray`, value - corresponding token). The simpliest way to get these data is sklearn `CountVectorizer` usage (or some similar class from sklearn).

If you have archived described variables run following code:

```
batch_vectorizer = artm.BatchVectorizer(data_format='bow_n_wd',
                                        n_wd=n_wd,
                                        vocabulary=vocabulary)
```

Well, if you have data in UCI format (e.g. `vocab.my_collection.txt` and `docword.my_collection.txt` files), that were put into the same directory with your script or notebook, you can create batches using next code:

```
batch_vectorizer = artm.BatchVectorizer(data_path='',
                                        data_format='bow_uci',
                                        collection_name='my_collection',
                                        target_folder='my_collection_batches')
```

The built-in library parser converted your data into batches and covered them with the `BatchVectorizer` class object, that is a general input data type for all methods of Python API. The batches were places in the directory, you specified in the `target_folder` parameter.

If you have the source file in the Vowpal Wabbit data format, you can use the following command:

```
batch_vectorizer = artm.BatchVectorizer(data_path='',
                                        data_format='vowpal_wabbit',
                                        target_folder='my_collection_batches')
```

The result is fully the same, as it was described above.

Note: If you had created batches ones, you shouldn't launch this process any more, because it spends many time while dealing with large collection. You can run the following code instead. It will create the `BatchVectorizer` object using the existing batches (this operation is very quick):

```
batch_vectorizer = artm.BatchVectorizer(data_path='my_collection_batches',
                                        data_format='batches')
```

- **Dictionary**:

The next step is to create `Dictionary`. This is a data structure containing the information about all unique tokens in the collection. The dictionary is generating outside the model, and this operation can be done in different ways (load, create, gather). The most basic case is to gather dictionary using batches directory. You need to do this operation only once when starting working with new collection. Use the following code:

```
dictionary = artm.Dictionary()
dictionary.gather(data_path='my_collection_batches')
```

In this case the token order in the dictionary (and in further Φ matrix) will be random. If you'd like to specify some order, you need to create the vocab file (see UCI format), containing all unique tokens of the collection in necessary order, and run the code below (assuming your file has `vocab.txt` name and located in the same directory with your code):

```
dictionary = artm.Dictionary()
dictionary.gather(data_path='my_collection_batches',
                  vocab_file_path='vocab.txt')
```

Take into consideration the fact that library will ignore any token from batches, that was not presented into vacab file, if you used it. `Dictionary` contains a lot of useful information about the collection. For example, each unique token in it has the corresponding variable - value. When BigARTM gathers the dictionary, it puts the relative frequency of this token in this variable. You can read about the use-cases of this variable in further sections.

Well, now you have a dictionary. It can be saved on the dick to prevent it's re-creation. You can save it in the binary format:

```
dictionary.save(dictionary_path='my_collection_batches/my_dictionary')
```

Or in the textual one (if you'd like to see the gathered data, for example):

```
dictionary.save_text(dictionary_path='my_collection_batches/my_dictionary.txt')
```

Saved dictionary can be loaded back. The code for binary file looks like next one:

```
dictionary.load(dictionary_path='my_collection_batches/my_dictionary.dict')
```

For textual dictionary you can run the next code:

```
dictionary.load_text(dictionary_path='my_collection_batches/my_dictionary.txt')
```

Besides looking the content of the textual dictionary, you also can moderate it (for example, change the value of value field). After you load the dictionary back, these changes will be used.

Note: All described ways of generating batches automatically generate dictionary. You can use it by typing:

```
batch_vectorizer.dictionary
```

If you don't want to create this dictionary, set `gather_dictionary` parameter in the constructor of `BatchVectorizer` to False. But this flag will be ignored if `data_format` == `bow_n_wd`, as it is the only possible way to generate dictionary in this case.

4.4.2 2. Base PLSA Model with Perplexity Score

Detailed description of all parameters and methods of BigARTM Python API classes can be found in Python Interface.

At this moment you need to have next objects:

- directory with `my_collection_batches` name, containing batches and dictionary in binary file `my_dictionary.dict`; the directory should have the same location with your code file;

- `Dictionary` variable `my_dictionary`, containing this dictionary (gathered or loaded);
- `BatchVectorizer` variable `batch_vectorizer` (the same we have created earlier).

If everything is OK, let's start creating the model. Firstly you need to read the specification of the `ARTM` class, which represents the model. Then you can use the following code to create the model:

```
model = artm.ARTM(num_topics=20, dictionary=my_dictionary)
```

Now you have created the model, containing Φ matrix with size "number of words in your dictionary" $\times$ "number of topics" (20). This matrix was randomly initialized. Note, that by default the random seed for initialization is fixed to archive the ability to re-run the experiments and get the same results. If you want to have another random start values, use the seed parameter of the `ARTM` class (it's different non-negative integer values leads to different initializations).

From this moment we can start learning the model. But typically it is useful to enable some scores for monitoring the quality of the model. Let's use the perplexity now.

You can deal with scores using the `scores` field of the `ARTM` class. The score of perplexity can be added in next way:

```
model.scores.add(artm.PerplexityScore(name='my_fisrt_perplexity_score',
                                      dictionary=my_dictionary))
```

Note, that perplexity should be enabled strongly in described way (you can change other parameters we didn't use here). You can read about it in Scores Description.

Note: If you try to create the second score with the same name, the `add()` call will be ignored.

Now let's start the main act, e.g. the learning of the model. We can do that in two ways: using online algorithm or offline one. The corresponding methods are `fit_online()` and `fit_offline()`. It is assumed, that you know the features of these algorithms, but I will briefly remind you:

- **Offline algorithm**: many passes through the collection, one pass through the single document (optional), only one update of the Φ matrix on one collection pass (at the end of the pass). You should use this algorithm while processing a small collection.
- **Online algorithm**: single pass through the collection (optional), many passes through the single document, several updates of the Φ matrix during one pass through the collection. Use this one when you deal with large collections, and with collections with quickly changing topics.

We will use the offline learning here and in all further examples in this page (because the correct usage of the online algorithm is big skill).

Well, let's start training:

```
model.fit_offline(batch_vectorizer=batch_vectorizer, num_collection_passes=10)
```

This code chunk had worked slower, than any previous one. Here we proceeded the first step of the learning, it will be useful to look at the perplexity. We need to use the `score_tracker` field of the `ARTM` class for this. It remember all the values of all scores on each Φ matrix update. These data can be retrieved using the names of scores.

You can extract only the last value:

```
print model.score_tracker['my_fisrt_perplexity_score'].last_value
```

Or you are able to extract the list of all values:

```
print model.score_tracker['my_fisrt_perplexity_score'].value
```

If the perplexity had convergenced, you can finish the learning process. In other way you need to continue. As it was noted above, the rule to have only one pass over the single document in the online algorithm is optional. Both

`fit_offline()` and `fit_online()` methods supports any number of document passes you want to have. To change this number you need to modify the corresponding parameter of the model:

```
model.num_document_passes = 5
```

All following calls of the learning methods will use this change. Let's continue fitting:

```
model.fit_offline(batch_vectorizer=batch_vectorizer, num_collection_passes=15)
```

We continued learning the previous model by making 15 more collection passes with 5 document passes.

You can continue to work with this model in described way. Now one note: if you understand in one moment that your model had degenerated, and you don't want to create the new one, then use the `initialize()` method, that will fill the Φ matrix with random numbers and won't change any other things (nor your tunes of the regularizers/scores, nor the history from `score_tracker`):

```
model.initialize(dictionary=my_dictionary)
```

FYI, this method is calling in the `ARTM` constructor, if you give it the dictionary name parameter. Note, that the change of the seed field will affect the call of `initialize()`.

Also note, that you can pass the name of the dictionary instead of the dictionary object whenever it uses.

```
model.initialize(dictionary=my_dictionary.name)
```

4.4.3 3. Regularizers and Scores Usage

Detailed description of all parameters and methods of BigARTM Python API classes can be found in Python Interface. Description of regularizers can be found in Regularizers Description.

The library has a pre-defined set of the regularizers (you can create new ones, if it's necessary, you can read about it in the corresponding notes in Creating New Regularizer). Now we'll study to use them.

We assume that all the conditions from the head of the section 2. Base PLSA Model with Perplexity Score are executed. Let's create the model and enable the perplexity score in it:

```
model = artm.ARTM(num_topics=20, dictionary=my_dictionary, cache_theta=False)
model.scores.add(artm.PerplexityScore(name='perplexity_score',
                                      dictionary=my_dictionary))
```

I should note the the cache_theta flag: it's allow you to save your Θ matrix in the memory or not. If you have large collection, it can be impossible to store it's Θ in the memory, and in case of short collection it can be useful to look at it. Default value is True. In the cases, when you need to use Θ matrix, but it is too big, you can use `ARTM.transform()` method (it will be discussed later).

Now let's try to add other scores, because the perplexity is not the only one to be used.

Let's add the scores of sparsity of Φ and Θ matrices and the information about the most probable tokens in each topic (top-tokens):

```
model.scores.add(artm.SparsityPhiScore(name='sparsity_phi_score'))
model.scores.add(artm.SparsityThetaScore(name='sparsity_theta_score'))
model.scores.add(artm.TopTokensScore(name='top_tokens_score'))
```

Scores have many useful parameters. For instance, they can be calculated on the subsets of topics. Let's count separately the sparsity of the first ten topics in Φ. But there's a problem: topics are identifying with their names, and we didn't specify them. If we used the `topic_names` parameter in the constructor (instead of `num_topics` one), we should have such a problem. But the solution is very easy: BigARTM had generated names and put them into the `topic_names` field, so you can use it:

```
model.scores.add(artm.SparsityPhiScore(name='sparsity_phi_score_10_topics', topic_names=model.topic_
```

Certainly, we could modify the previous score without creating new one, if the general model sparsity wasn't interesting for us:

```
model.scores['sparsity_phi_score'].topic_names = model.topic_names[0: 9]
```

But let's assume that we are also interested in it and keep everything as is. You should remember that all the parameters of metrics, model and regularizers (we will talk about them soon) can be set and reset by the direct change of the corresponding field, as it was demonstrated in the code above.

For example, let's ask the top-tokens score to show us 12 most probable tokens in each topic:

```
model.num_tokens = 12
```

Well, we achieved the model covered with necessary scores, and can start the fitting process:

```
model.fit_offline(batch_vectorizer=batch_vectorizer, num_collection_passes=10)
```

We saw this code in the first section. But now we can see the values of new added scores:

```
print model.score_tracker['perplexity_score'].value      # .last_value
print model.score_tracker['sparsity_phi_score'].value    # .last_value
print model.score_tracker['sparsity_theta_score'].value  # .last_value
```

As we can see, all the scores didn't change. But we forgot about the top-tokens. Here we need to act more accurately: the score stores the data on each moment of Φ update. Let's assume that we need only the last data. So we need to use the *last_tokens* field. It is a Python dict, where key is a topic name, and value is a list of top-tokens of this topic.

Note: The scores are loading from the kernel on each call, so for such a big scores, as top-tokens (or topic kernel score), it's strongly recommended to store the whole score in the local variable, and then deal with it. So, let's look through all top-tokens in the loop:

```
saved_top_tokens = model.score_tracker['top_tokens_score'].last_tokens

for topic_name in model.topic_names:
    print saved_top_tokens[topic_name]
```

Probably the topics are not very good. For the aim of increasing the quality of the topics you can use the regularizers. The code for dealing with the regularizers is very similar with the one for scores. Let's add three regularizers into our model: sparsing of Φ matrix, sparsing of Θ matrix and topics decorrelation. The last one is need to make topics more different.

```
model.regularizers.add(artm.SmoothSparsePhiRegularizer(name='sparse_phi_regularizer'))
model.regularizers.add(artm.SmoothSparseThetaRegularizer(name='sparse_theta_regularizer'))
model.regularizers.add(artm.DecorrelatorPhiRegularizer(name='decorrelator_phi_regularizer'))
```

Maybe you have a question about the name of the `SmoothSparsePhi\Theta` regularizer. Yes, it can both smooth and sparse topics. It's action depends on the value of corresponding coefficient of the regularization :math:`tau` (we assume, that you know, what is it). `tau` > 0 leads to smoothing, `tau` < 0 to sparsing. By default all the regularizers has `tau` = 1, which is usually not what you want. Choosing good `tau` is a heuristic, sometimes you need to process dozens of the experiments to pick up good values. It is the experimental work, and we won't discuss it here. Let's look at technical details instead:

```
model.regularizers['sparse_phi_regularizer'].tau = -1.0
model.regularizers['sparse_theta_regularizer'].tau = -0.5
model.regularizers['decorrelator_phi_regularizer'].tau = 1e+5
```

We set standard values, but in bad case they can be useless or even harmful for the model.

We draw your attention again to the fact, that setting and changing the values of the regularizer parameters is fully similar to the scores.

Let's start the learning process:

```
model.fit_offline(batch_vectorizer=batch_vectorizer, num_collection_passes=10)
```

Further you can look at metrics, change `tau` coefficients of the regularizers and etc. As for scores, you can ask the regularizer to deal only with given topics, using `topic_names` parameter.

Let's return to the dictionaries. But here's one discussion firstly. Let's look at the principle of work of the `SmoothSparsePhi` regularizer. It simply adds to all counters the same value `tau`. Such a strategy can be unsuitable for us. The probable case: a need for sparsing one part of words, smoothing another one and ignoring the rest tokens. For example, let's sparse the tokens about *magic*, smooth tokens about *cats* and ignore all other ones.

In this situation we need dictionaries.

Let's remember about the value field, that corresponds each unique token. And also the fact, that `SmoothSparsePhi` regularizer has the `dictionary` field. If you set this field, the regularizer will add to counters `tau` * `value` for this token, instead of `tau`. In such way we can set the `tau` to 1, for instance, set the `value` variable in dictionary for tokens about *magic* equal to -1.0, for tokens about *cats* equal to 1.0, and 0.0 for other tokens. And we'll get what we need.

The last problem is how to change these *value* variables. It was discussed in the 1. Loading Data: BatchVectorizer and Dictionary: let's remember about the methods `Dictionary.save_text()` and `Dictionary.load_text()`.

You need to proceed next steps:

- save the dictionary in the textual format;
- open it, each line corresponds to one unique token, the line contains 5 values: `token` - `modality` - `value` - `token_tf` - `token_df`;
- don't pay attention to anything except the token and the value; find all tokens you are interested in and change their values parameters;
- load the dictionary back into the library.

Your file can have such a view after editing (conceptually):

```
cat       smth       1.0       smth       smth
shower    smth       0.0       smth       smth
magic     smth      -1.0       smth       smth
kitten    smth       1.0       smth       smth
merlin    smth      -1.0       smth       smth
moscow    smth       0.0       smth       smth
```

All the code you need to process discussed operation was showed above. Here is an example of creation of the regularizer with dicitonary:

```
model.regularizer.add(artm.SmoothSparsePhiRegularizer(name='smooth_sparse_phi_regularizer',
                                                       dictionary=my_dictionary))
```

4.4.4 4. Multimodal Topic Models

Detailed description of all parameters and methods of BigARTM Python API classes can be found in Python Interface.

Now let's move to more complex cases. In last section mentioned the term *modality*. It's something that corresponds to each token. We prefer to think about it as about the type of token, For instance, some tokens form the main text in the document, some form the title, some from names of the authors, some from tags etc.

In BigARTM each unique token has a modality. It is denoted as `class_id` (don't confuse with the classes in the task of classification). You can specify the `class_id` of the token, or the library will set it to `@default_class`. This class id denotes the type of usual tokens, the type be default.

In the most cases you don't need to use the modalities, but there're some situations, when they are indispensable. For example, in the task of document classification. Strictly speaking, we will talk about it now.

You need to re-create all the data with considering the presence of the modalities. Your task is to create the file in the Vowpal Wabbit format, where each line is a document, and each document contains the tokens of two modalities - the usual tokens, and the tokens-labels of classes, the document belongs to.

Now follow again the instruction from the introduction part, dealing with your new Vowpal Wabbit file to achieve batches and the dictionary.

The next step is to explain your model the information about your modalities and the power in the model. Power of the modality is it's coefficient τ_m (we assume you know about it). The model uses by default only tokens of `@default_class` modality and uses it with τ_m = 1.0. You need to specify other modalities and their weights in the constructor of the model, using following code, if you need to use these modalities:

```
model = artm.ARTM(num_topics=20, class_ids={'@default_class': 1.0, '@labels_class': 5.0})
```

Well, we asked the model to take into consideration these two modalities, and the class labels will be more powerful in this model, than the tokens of the `@default_class` modality. Note, that if you had the tokens of another modality in your file, they wouldn't be taken into consideration. Similarly, if you had specified in the constructor the modality, that doesn't exist in the data, it will be skipped.

Of course, the `class_ids` field, as all other ones, can be reseted. You always can change the weights of the modalities:

```
model.class_ids = {'@default_class': 1.0, '@labels_class': 50.0}
# model.class_ids['@labels_class'] = 50.0 --- NO!!!
```

You need to update the weights directly in such way, don't try to refer to the modality by the key directly: `class_ids` can be updated using Python dict, but it is not the dict.

The next launch of `fit_offline()` or `fit_online()` will take this new information into consideration.

Now we need to enable scores and regularizers in the model. This process was viewed earlier, excluding one case. All the scores of Φ matrix (and perplexity) and Φ regularizers has fields to deal with modalities. Through these fields you can define the modalities to be deal with by score or regularizer, the other ones will be ignored (here's the full similarity with `topic_names` field).

The modality field can be `class_id` or `class_ids`. The first one is the string containing the name of the modality to deal with, the second one is a list of strings.

Note: The missing value of `class_id` means `class_id` = `@default_class`, missing value of `class_ids` means usage of all existing modalities.

Let's add the score of sparsity Φ for the modality of class labels and regularizers of topic decorrelation for each modality, and start fitting:

```
model.scores.add(artm.SparsityPhiScore(name='sparsity_phi_score',
                                       class_id='@labels_class'))

model.regularizers.add(artm.DecorrelatorPhiRegularizer(name='decorrelator_phi_def',
                                                       class_ids=['@default_class']))

model.regularizers.add(artm.DecorrelatorPhiRegularizer(name='decorrelator_phi_lab',
                                                       class_ids=['@labels_class']))
```

```
model.fit_offline(batch_vectorizer=batch_vectorizer, num_collection_passes=10)
```

Well, we will leave you the rest of the work (tuning τ and τ_m coefficients, looking at scores results etc.). In 5. Phi and Theta Extraction. Transform Method we will go to the usage of fitted ready model for the classification of test data.

4.4.5 5. Phi and Theta Extraction. Transform Method

Detailed description of all parameters and methods of BigARTM Python API classes can be found in Python Interface.

- **Phi/Theta exraction**

Let's assume, that you have a data and a model, fitted on this data. You had tuned all necessary regularizers and used scores. But the set of quality measures of the library wasn't enough for you, and you need to compute your own scores using Φ and Θ matrices. In this case you are able to extract these matrices using next code:

```
phi = model.get_phi()
theta = model.get_theta()
```

Note, that you need a `cache_theta` flag to be set True if you are planning to extract Θ in the future without using `transform()`. You also can extract not whole matrices, but part of them, that corresponds different topics (using the same `topic_names` parameter of the methods, as in previous sections). Also you can extract only necessary modalities of the Φ matrix, if you want.

Both methods return `pandas.DataFrame`.

- **Transform new documents**

Now we will go to the usage of fitted ready model for the classification of test data (to do it you need to have a fitted multimodal model with `@labels_class` modality, see 4. Multimodal Topic Models).

In the classification task you have the train data (the collection you used to train your model, where for each document the model knew it's true class labels), and test one. For the test data true labels are known to you, but are unknown to the model. Model need to forecast these labels, using test documents, and your task is to compute the quality of the predictions by counting some metrics, AUC, for instance.

Computation of the AUC or any other quality measure is your task, we won't do it. Instead, we will learn how to get $p(c|d)$ vectors for each document, where each value is the probability of class c in the given document d.

Well, we have a model. We assume you put test documents into separate file in Vowpal Wabbit format, and created batches using it, which are covered by the variable `batch_vectorizer_test`. Also we assume you have saved your test batches into the separate directory (not into the one containing train batches).

Your test documents shouldn't contain information about true labels (e.g. the Vowpal Wabbit file shouldn't contain string '|@labels_class'), also text document shouldn't contain tokens, that doesn't appear in the train set. Such tokens will be ignored.

If all these conditions are met, we can use the `ARTM.transform()` method, that allows you to get $p(t|d)$ (e.g. Θ) or $p(c|d)$ matrix for all documents from your `BatchVectorizer` object.

Run this code to get Θ:

```
theta_test = model.transform(batch_vectorizer=batch_vectorizer_test)
```

And this one to achieve $p(c|d)$:

```
p_cd_test = model.transform(batch_vectorizer=batch_vectorizer_test,
                            predict_class_id='@labels_class')
```

In this way you have got the predictions of the model in `pandas.DataFrame`. Now you can score the quality of the predictions of your model in all ways, you need.

Method allows you to extract dense od sparse matrix. Also you can use for p_{tdw} matrix (see 8. Deal with Ptdw Matrix).

4.4.6 6. Tokens Coocurancy and Coherence Coumputation

4.4.7 7. Attach Model and Custom Phi Initialization

Detailed description of all parameters and methods of BigARTM Python API classes can be found in Python Interface.

Library supports an ability to access all Φ-like matrices directly from Python. This is a low-level functionality, so it wasn't included in the `ARTM` class, and can be used via low-level `master_component` interface. User can attach the matrix, e.g. get reference to it in the Python, and can change it's content between the iterations. The changes will be written in the native C++ memory.

The most evidence case of usage of this feature is a custom initialization of Φ matrix. The library initalizes it with random numbers by default. But there're several more complex and useful methods of initialization, that the library doesn't support yet. And in this case `attach_model` method can help you.

So let's attach to the Φ matrix of our model:

```
(_, phi_ref) = model.master.attach_model(model=model.model_pwt)
```

At this moment you can print Φ matrix to see it's content:

```
model.get_phi(model_name=model.model_pwt)
```

Next code can be used to check whether the attaching was successful:

```
for model_description in model.info.model:
     print model_description
```

The output will be similar to the following

```
name: "nwt"

type: "class artm::core::DensePhiMatrix"

num_topics: 50

num_tokens: 2500



name: "pwt"

type: "class __artm::core::AttachedPhiMatrix__"

num_topics: 50

num_tokens: 2500
```

You can see, that the type of Φ matrix has changed from `DensePhiMatrix` to `AttachedPhiMatrix`.

Now let's assume that you have created `pwt_new` matrix with the same size, filled with custom values. Let's write these values into our Φ matrix.

Note: You need to write the values by accessing `phi_ref` variable, you are not allowed to assing it the whole `pwt_new` matrix, this operation will lead to an error in future work.

```
for tok in xrange(num_tokens):
     for top in xrange(num_topics):
          phi_ref[tok, top] = pwt_new[tok, top]   # CORRECT!

phi_ref = pwt_new   # NO!
```

After that you can print Φ matrix again and check the change of it's values. From this moment you can continue our work.

4.4.8 8. Deal with Ptdw Matrix

4.4.9 Different Useful Techniques

- **Dictionary filtering**:

In this section we'll discuss dictionary's self-filtering ability. Let's remember the structure of the dictionary, saved in textual format (see 4. Multimodal Topic Models). There are many lines, one per each unique token, and each line contains 5 values: token (string), its class_id (string), its value (double) and two more integer parameters, called token_tf and token_df. token_tf is an absolute frequency of the token in the whole collection, and token_df is the number of documents in the collection, where the token had appeared at least once. These values are generating during gathering dictionary by the library. They differ from the value in the fact, that you can't use them in the regularizers and scores, so you shouldn't change them.

They need for filtering of the dictionary. You likely needn't to use very seldom or too frequent tokens in your model. Or you simply want to reduce your dictionary to hold your model in the memory. In both cases the solution is to use the `Dictionary.filter()` method. See its parameters in Python Interface. Now let's filter the modality of usual tokens:

```
dictionary.filter(min_tf=10, max_tf=2000, min_df_rate=0.01)
```

Note: If the parameter has _rate suffix, it denotes relative value (e.g. from 0 to 1), otherwise - absolute value.

This call has one feature, it rewrites the old dictionary with new one. So if you don't want to lose your full dictionary, you need firstly to save it to disk, and then filter the copy located in the memory.

- **Saving/loading model**:

Now let's study saving the model to disk.

It's important to understand that the model contains two matrices: Φ (or p_{wt}) and n_{wt}. To make model be loadable without loses you need to save both these matrices. The current library version can save only one matrix per method call, so you will need two calls:

```
model.save(filename='saved_p_wt', model_name='p_wt')
model.save(filename='saved_n_wt', model_name='n_wt')
```

The model will be saved in binary format. To use it later you need to load it's matrices back:

```
model.load(filename='saved_p_wt', model_name='p_wt')
model.load(filename='saved_n_wt', model_name='n_wt')
```

Note: The model after loading will only contain Φ and n_{wt} matrices and some associated information (like number of topics, their names, the names of the modalities (without weights!) and some other data). So you need to restore all

necessary scores, regularizers, modality weights and all important parameters, like `cache_theta`.

You can use `save/load` methods pair in case of long fitting, when restoring parameters is much more easier than model re-fitting.

- **Creating batches manually**:

There're some cases where you may need to create your own batches without using vowpal wabbit/UCI files. To do it from Python you should create `artm.messages.Batch` object and fill it. The parameters of this meassage can be found in Messages, it looks like this:

```
message Batch {
  repeated string token = 1;
  repeated string class_id = 2;
  repeated Item item = 3;
  optional string description = 4;
  optional string id = 5;
}
```

First two fields are the vocabulary of the batch, e.g. the set of all unique tokens from it's documents (items). In case of no modalities or only one modality you may skip `class_id` field. Last two fileds are not very important, you can skip them. Third field is the set of the documents. The `Item` message has the next structure:

```
message Item {
  optional int32 id = 1;
  repeated Field field = 2;  // obsolete in BigARTM v0.8.0
  optional string title = 3;
  repeated int32 token_id = 4;
  repeated float token_weight = 5;
}
```

First field of it is the identifier, second is obsoleted, third is the title. You need to specify at least first one, or both `id` and `title`. `token_id` is a list of indices of the tokens in this item from `Batch.token` vocabulary. `token_weight` is the list of corresponding counters. In case of Bag-of-Words `token_id` should contain unique indices, in case of sequential text `token_weight` should contain only 1.0. But really you can fill these fields as you like, the only limitation is to keep their lengths equal.

Now let's create a simple batch for collection without modalities (it is quite simple to modify the code to use them). If you have list `vocab` with all uniqie tokens, and also have a list of lists `documents`, where each internal list is a document in it's natural represenatation, you can ran the following code to create batch:

```
import artm
import uuid

vocab = ['aaa', 'bbb', 'ccc', 'ddd']

documents = [
             ['aaa', 'ccc', 'aaa', 'ddd'],
             ['bbb', 'ccc', 'aaa', 'bbb', 'bbb'],
            ]

batch = artm.messages.Batch()
batch.id = str(uuid.uuid4())
dictionary = {}
use_bag_of_words = True

# first step: fill the general batch vocabulary
for i, token in enumerate(vocab):
    batch.token.append(token)
```

```
    dictionary[token] = i

# second step: fill the items
for doc in documents:
    item = batch.item.add()

    if use_bag_of_words:
        local_dict = {}
        for token in doc:
            if not token in local_dict:
                local_dict[token] = 0
            local_dict[token] += 1

        for k, v in local_dict.iteritems():
            item.token_id.append(dictionary[k])
            item.token_weight.append(v)

     else:
         for token in doc:
             item.token_id.append(dictionary[token])
             item.token_weight.append(1.0)

# save batch into the file
with open('my_batch.batch', 'wb') as fout:
    fout.write(batch.SerializeToString())

# you can read it back using the next code
#batch2 = artm.messages.Batch()
#with open('my_batch.batch', 'rb') as fin:
#    batch2.ParseFromString(fin.read())

# to print your batch run
print batch
```

4.5 Regularizers Description

This page describes the features and cases of usage of the regularizers, that have already been implemented in the core of BigARTM library. Detailed description of parameters of the regularizers in the Python API can be seen in Regularizers.

Examples of the usage of the regularizers can be found in Python Tutorial and in Python Guide.

Note: The influence of any regularizer with `tau` parameter in it's M-step formula can be controlled via this parameter.

4.5.1 Smooth/Sparse Phi

- **M-step formula:**

`p_wt` $\propto$ `(n_wt + tau * f(p_wt) * dict[w])`

- **Description:**

This regularizer provides an opportunity to smooth or to sparse subsets of topics using any specified distribution.

To control the type of the distribution over tokens one can use the dictionary (`dict`), and the function `f`. `dict` is an object of Dictionary class, containing list with all unique tokens and corresponding list of values, which can be

specified by the user. These values are `dict[w]`. `f` is a transform function, the derivative of the function under the KL-divergence in the source formula of the regularizer.

If the dictionary is not specified, all values will be 1. If is specified, and there's no value for the token in it, the token will be skipped. The `f` is const 1 by default.

- **Usage:**

There're several strategies of usages of this regularizer:

1. simply smooth or sparse all values in the Φ matrix with value `n`: create one regularizer and assign `tau` to `n`;
2. divide all topics into two groups (subject and background), sparse first group to increase their quality and smooth second one to gather there all background tokens: create two regularizers, specify `topic_names` parameters with corresponding list in both ones and set `tau` in sparsing regularizer to some negative value, and in smoothing to some positive one;
3. smooth or sparse only tokens of specified modalities: create one regularizer and specify parameter `class_ids` with list of names of modalities to be regularized;
4. smooth or sparse only tokens from pre-defined list: edit the Dictionary object of the collection and change `value` field of the dictionary for tokens from this list should be set to some positive constant (remember, that it will be `dict[w]` in formula), and for other tokens `value` field should be set to 0;
5. increase the influence on the small `p_wt` values and reduce on the big: define the `f` function by specifying the `KlFunctionInfo` object to `kl_function_info` parameter (don't forget, that `f` will be the derivative of this function).

All these strategies can be transformed, merged to each other etc.

Notes:

- smoothing in second strategy should be started from the start of model learning and be constant;
- sparsing should start after some iterations and it's influence should increase gradually;
- setting different values to `value` field for tokens from pre-defined list in pre-last strategy gives an opportunity to deal with each token as it is necessary.

4.5.2 Smooth/Sparse Theta

- **M-step formula:**

`p_td` $\propto$ `(n_td + tau * alpha_iter[iter] * f(p_td) * mult[d][t])`

- **Description:**

This regularizer provides an opportunity to smooth or to sparse subsets of topics in Θ matrix.

To control the influence of the regularizer on each document pass one can use `alpha_iter` parameter, which is a list with length equal to number of passes through the document, and the function `f`. `f` is a transform function, the derivative of the function under the KL-divergence in the source formula of the regularizer. If `alpha_iter` is not specified, it is 1 for all passes. The `f` is const 1 by default.

Also you can control the regularization of any document for any topic, or set a general mask for all documents, that specifies mult coef for each topic.

- **Usage:**

There're several strategies of usages of this regularizer:

1. simply smooth or sparse all values in the Θ matrix with value `n`: create one regularizer and assign `tau` to `n`;

2. divide all topics into two groups (subject and background), sparse first group to increase their quality and smooth second one to gather there all background tokens: create two regularizers, specify `topic_names` parameters with corresponding list in both ones and set `tau` in sparsing regularizer to some negative value, and in smoothing to some positive one;
3. increase the influence of the regularizer on later passes through the document: specify `alpha_iter` list with `k` values from smaller to greater, where `k` is a number of passes though the document;
4. increase the influence on the small `p_td` values and reduce on the big: define the `f` function by specifying the `KlFunctionInfo` object to `kl_function_info` parameter (don't forget, that `f` will be the derivative of this function).
5. influence only specified documents in specified way using `mult` variable. This can be used as the custom initialization of Θ matrix.

All these strategies can be transformed, merged to each other etc.

Notes:

- smoothing in second strategy should be started from the start of model learning and be constant;
- sparsing should start after some iterations and it's influence should increase gradually;
- fitting (with regularization or not) will be different in cases of `ARTM.reuse_theta` flag set to True or False.

4.5.3 Decorrelator Phi

- **M-step formula:**

`p_wt` $\propto$ (`n_wt` - `tau` * `p_wt` * $\sum_{s \in T}$ (`p_ws`))

- **Description:**

This regularizer provides an opportunity to decorrelate columns in the Φ matrix (e.g. make topics more different), that allows to increase the interpretability of topics.

- **Usage:**

There're several strategies of usages of this regularizer:

1. decorrelate all topics: one regularizer with `tau`, that should be tuned experimentally;
2. in the strategy with two background and subject topics it is recommended to deal with each group separately: create two decorrelators and specify `topic_names` parameter in them, as it was done for Smooth/sparse regularizers;
3. deal only with the tokens of given modality: set `class_ids` parameter the list with names of modalities to use in this regularizer.

All these strategies can be transformed, merged to each other etc.

Notes:

- this is a sparsing regularizer, it works well with general sparsing Φ regularizer;
- the recommendation is to run this regularizer from the beginning of the fitting process.

4.5.4 Label Regularization Phi

- **M-step formula:**

`p_wt` $\propto$ `n_wt` + `tau` * `dict[w]` * (`p_wt` * `n_t`) / ($\sum_{s \in T}$ (`p_ws` * `n_s`))

4.5.5 Specified sparse Phi

- **Description:**

It is not a usual regularizer, it's a tool to sparse as many elements in Φ, as you need. You can sparse by columns or by rows.

4.5.6 Improve Coherence Phi

- **M-step formula:**

$\texttt{p\_wt} \propto \texttt{n\_wt} + \texttt{tau} * \sum_{v \in W} \texttt{cooc\_dict[w][v]} * \texttt{n\_vt}$

4.5.7 Smooth Ptdw

- **M-step formula:**

ToDo(anyap)

4.5.8 Topic Selection Theta

- **M-step formula:**

$\texttt{p\_td} \propto \texttt{n\_td - tau} * \texttt{n\_td} * \texttt{topic\_value[t]} * \texttt{alpha\_iter[iter]}$,

where $\texttt{topic\_value[t]} = \texttt{n} / (\texttt{n\_t} * \texttt{|T|})$

4.5.9 Biterms Phi

- **M-step formula:**

$\texttt{p\_wt} \propto (\texttt{n\_wt} + \texttt{tau} * \sum_{u \in W} (\texttt{p\_tuw}))$, where $\texttt{p\_tuw} = \text{norm}_{t \in T} (\texttt{n\_t} * \texttt{p\_wt} * \texttt{p\_ut})$

4.5.10 Hierarchy Sparsing Theta

ToDo(nadiinchi)

4.5.11 Topic Segmentation Ptdw

ToDo(anastasiabayandina)

4.6 Scores Description

This page describes the scores, that have already been implemented in the core of BigARTM library. Detailed description of parameters of the scores in the Python API can be seen in Scores, and their return values in Score Tracker.

Examples of the usage of the scores can be found in Python Tutorial and in Python Guide.

4.6.1 Perplexity

- **Description:**

The formula of perplexity: $\mathcal{P}(D;\Phi,\Theta) = \exp\left(-\frac{1}{n}\sum_{d\in D}\sum_{w\in d} n_{dw}\ln\sum_{t\in T}\phi_{wt}\theta_{td}\right) = \exp\left(-\frac{1}{n}\mathcal{L}(D;\Phi,\Theta)\right)$,

where $\mathcal{L}(D)$ is a log-likelyhood of model parameters on documents set D.

- **Usage:**

This is one of the main scores as it indicates the level and speed of convergence of the model. Some notes:

- smaller perplexity value means better convergenced model and can be used for comparison in case of dense models with the same set of topics, dictionaries and train documents.
- sparse model provides a lot of zero $p(w|d) = \sum_{t\in T}\phi_{wt}\theta_{td}$. There're two replacement strategies to avoid it: document unigram model ($p(w|d) = \frac{n_{dw}}{n_d}$) and collection unigram model ($p(w|d) = \frac{n_w}{n}$). First onr is simplier but it is not correct. Collection unigram model is a better approximation, that allows perplexity comparison of sparse models (in case of equal sets of topics, dictionaries and train documents). This mode is default for score in the library, though it requires BigARTM collection dictionary.
- perplexity can be computeted on both train and test sets of documents, though experiments showed a large correlation of the results, so you may only use train perplexity as a convergence measure.

4.6.2 Sparsity Phi

- **Description:**

Computes the ratio of elements of Φ matrix (or it's part) that are less than given `eps` threshold.

- **Usage:**

One of the goals of regularization is to achive a sparse structure of Φ matrix using different sparsing regularizers. This scores allows to control this process. While using different regularization stratgies in different parts of model you can create a score per each part and one for whole model to have detailed and whole values.

4.6.3 Sparsity Theta

- **Description:**

Computes the ratio of elements of Θ matrix (or it's part) that are less than given `eps` threshold.

- **Usage:**

One of the goals of regularization is to achive a sparse structure of Θ matrix using different sparsing regularizers. This scores allows to control this process. While using different regularization stratgies in different parts of Θ you can create a score per each part and one for whole matrix to have detailed and whole values.

4.6.4 Top Tokens

- **Description:**

Return `k` (= requested number of top tokens) most probable tokens in each requested topic. If the number of tokens with $p(w|t) > 0$ in topic is less than `k` then only these tokens will be returned. Also the score can compute the coherence of top tokens in the topic using coocurancy dictionary (see 6. Tokens Coocurancy and Coherence Coumputation for detailes of usage from Python).

The coherence formula for topic is defined as

$$\mathcal{C}_t = \frac{2}{k(k-1)} \sum_{i=1}^{k-1} \sum_{j=1}^{k} \text{value}(w_i, w_j),$$

where value is some pairwise information about tokens in collection dictionary, which is provided by user according to his goals.

- **Usage:**

Checking top tokens is one of the main ways of topic quality verification. But in practice the interpretation of top tokens not always correlated with the real sence of topic and it is useful to check the documents assigned to it.

Coherence is one of the best measures for automatic inpretability checking. Note, that different types of `value` will lead to different types of coherence.

4.6.5 Topic Kernel Scores

- **Description:**

This score was created as one more way to control the inpretability of the topics. Let's define the *topic kernel* as $W_t = \{w \in W | p(t|w) > \text{threashold}\}$, and determine several measures, based on it:

- $\sum_{w \in W_t} p(w|t)$ - the *purity* of the topic (higher values corresponds better topics);
- $\frac{1}{|W_t|} \sum_{w \in W_t} p(t|w)$ - *contrast* of the topic (higher values corresponds better topics);
- $|W_t|$ - *size* of the topic kernel (approximately optimal value is $\frac{|W|}{|T|}$).

The score computes all measures for requested topics and their average values.

Also it can compute coherence topic kernels. The coherence formula is the same as for Top Tokens score. `k` parameter will be equal to kernel size for given topic.

- **Usage:**

Average purity and contrast can be used as a measure of topics inpretability and difference. `threahold` parameter is recommended to be set to 0.5 and higher, it should be set once and fixed.

4.6.6 Topic Mass

- **Description:**

Computes the n_t values for each requested topic in Φ matrix.

- **Usage:**

Can be useful in external (self written in Python) scores and regularizers, that requires n_t. Much more faster than Φ extraction and normalization.

4.6.7 Class Presicion

ToDo(sashafrey)

4.6.8 Background Tokens Ratio

- **Description:**

Computes KL-divergence between $p(t)$ and $p(t|w)$ distributions $\mathrm{KL}(p(t)||p(t|w))$ (or via versa) for each token and counts the part of tokens that have this value greater than given `delta`. Such tokens are considered to be background ones. Also returns all these tokens, if it was requested.

4.6.9 Items Processed (technical)

- **Description:**

Computes the number of documents, that was used for model training since the score was included into it. During iterations one real document can be counted more than once.

4.6.10 Theta Snippet (technical)

- **Description:**

Returns a requested small part of Θ matrix.

CHAPTER 5

API References

5.1 Python Interface

This document describes all classes and functions in python interface of BigARTM library.

5.1.1 ARTM model

This page describes ARTM class.

class artm.**ARTM**(*num_topics=None*, *topic_names=None*, *num_processors=None*, *class_ids=None*, *scores=None*, *regularizers=None*, *num_document_passes=10*, *reuse_theta=False*, *dictionary=None*, *cache_theta=False*, *theta_columns_naming='id'*, *seed=-1*, *show_progress_bars=False*, *ptd_name=None*)

__init__(*num_topics=None*, *topic_names=None*, *num_processors=None*, *class_ids=None*, *scores=None*, *regularizers=None*, *num_document_passes=10*, *reuse_theta=False*, *dictionary=None*, *cache_theta=False*, *theta_columns_naming='id'*, *seed=-1*, *show_progress_bars=False*, *ptd_name=None*)

Parameters

- **num_topics** (*int*) – the number of topics in model, will be overwrited if topic_names is set
- **num_processors** (*int*) – how many threads will be used for model training, if not specified then number of threads will be detected by the lib
- **topic_names** (*list of str*) – names of topics in model
- **class_ids** (*dict*) – list of class_ids and their weights to be used in model, key — class_id, value — weight, if not specified then all class_ids will be used
- **cache_theta** (*bool*) – save or not the Theta matrix in model. Necessary if ARTM.get_theta() usage expects
- **scores** (*list*) – list of scores (objects of artm.*Score classes)
- **regularizers** (*list*) – list with regularizers (objects of artm.*Regularizer classes)
- **num_document_passes** (*int*) – number of inner iterations over each document
- **dictionary** (*str or reference to Dictionary object*) – dictionary to be used for initialization, if None nothing will be done
- **reuse_theta** (*bool*) – reuse Theta from previous iteration or not

- **theta_columns_naming** (*str*) – either 'id' or 'title', determines how to name columns (documents) in theta dataframe
- **seed** (*unsigned int or -1*) – seed for random initialization, -1 means no seed
- **show_progress_bars** – a boolean flag indicating whether to show progress bar in fit_offline, fit_online and transform operations.
- **ptd_name** – string, name of ptd (theta) matrix

Important public fields

- regularizers: contains dict of regularizers, included into model
- scores: contains dict of scores, included into model
- score_tracker: contains dict of scoring results: key — score name, value — ScoreTracker object, which contains info about values of score on each synchronization (e.g. collection pass) in list

Note

- Here and anywhere in BigARTM empty topic_names or class_ids means that model (or regularizer, or score) should use all topics or class_ids.
- If some fields of regularizers or scores are not defined by user — internal lib defaults would be used.
- If field 'topic_names' is None, it will be generated by BigARTM and will be available using ARTM.topic_names().
- Most arguments of ARTM constructor have corresponding setter and getter of the same name that allows to change them at later time, after ARTM object has been created.
- Setting ptd_name to a non-empty string activates an experimental mode where cached theta matrix is internally stored as a phi matrix with tokens corresponding to item title, and token class ids corresponding to batch id. With ptd_name argument you specify the name of this matrix (for example 'ptd' or 'theta', or whatever name you like). Later you can retrieve this matix with ARTM.get_phi(model_name='ptd'), change its values with ARTM.master.attach_model(model='ptd'), export/import this matrix with ARTM.master.export_model('ptd', filename) and ARTM.master.import_model('ptd', file_name).

clone()

Description returns a deep copy of the artm.ARTM object

Note

- This method is equivalent to copy.deepcopy() of your artm.ARTM object. Both methods perform deep copy of the object, including a complete copy of its internal C++ state (e.g. a copy of all phi and theta matrices, scores and regularizers, as well as ScoreTracker information with history of the scores).
- Attached phi matrices are copied as dense phi matrices.

dispose()

Description free all native memory, allocated for this model

Note

- This method does not free memory occupied by dictionaries, because dictionaries are shared across all models

- ARTM class implements __exit__ and __del___ methods, which automatically call dispose.

fit_offline (*batch_vectorizer=None*, *num_collection_passes=1*)

Description proceeds the learning of topic model in offline mode

Parameters

- **batch_vectorizer** (*object_referenece*) – an instance of BatchVectorizer class
- **num_collection_passes** (*int*) – number of iterations over whole given collection

fit_online (*batch_vectorizer=None*, *tau0=1024.0*, *kappa=0.7*, *update_every=1*, *apply_weight=None*, *decay_weight=None*, *update_after=None*, *async=False*)

Description proceeds the learning of topic model in online mode

Parameters

- **batch_vectorizer** (*object_reference*) – an instance of BatchVectorizer class
- **update_every** (*int*) – the number of batches; model will be updated once per it
- **tau0** (*float*) – coefficient (see 'Update formulas' paragraph)
- **kappa (float)** (*float*) – power for tau0, (see 'Update formulas' paragraph)
- **update_after** (*list of int*) – number of batches to be passed for Phi synchronizations
- **apply_weight** (*list of float*) – weight of applying new counters
- **decay_weight** (*list of float*) – weight of applying old counters
- **async** (*bool*) – use or not the async implementation of the EM-algorithm

Note async=True leads to impossibility of score extraction via score_tracker. Use get_score() instead.

Update formulas

- The formulas for decay_weight and apply_weight:
- update_count = current_processed_docs / (batch_size * update_every);
- rho = pow(tau0 + update_count, -kappa);
- decay_weight = 1-rho;
- apply_weight = rho;
- if apply_weight, decay_weight and update_after are set, they will be used, otherwise the code below will be used (with update_every, tau0 and kappa)

get_phi (*topic_names=None*, *class_ids=None*, *model_name=None*)

Description get custom Phi matrix of model. The extraction of the whole Phi matrix expects **ARTM.phi_** call.

Parameters

- **topic_names** (*list of str or str or None*) – list with topics or single topic to extract, None value means all topics
- **class_ids** (*list of str or str or None*) – list with class_ids or single class_id to extract, None means all class ids

- **model_name** (*str*) – self.model_pwt by default, self.model_nwt is also reasonable to extract unnormalized counters

Returns

- pandas.DataFrame: (data, columns, rows), where:
- columns — the names of topics in topic model;
- rows — the tokens of topic model;
- data — content of Phi matrix.

get_phi_sparse(*topic_names=None*, *class_ids=None*, *model_name=None*, *eps=None*)

Description get phi matrix in sparse format

Parameters

- **topic_names** (*list of str or str or None*) – list with topics or single topic to extract, None value means all topics
- **class_ids** (*list of str or str or None*) – list with class_ids or single class_id to extract, None means all class ids
- **model_name** (*str*) – self.model_pwt by default, self.model_nwt is also reasonable to extract unnormalized counters
- **eps** (*float*) – threshold to consider values as zero

Returns

- a 3-tuple of (data, rows, columns), where
- data — scipy.sparse.csr_matrix with values
- columns — the names of topics in topic model;
- rows — the tokens of topic model;

get_score(*score_name*)

Description get score after fit_offline, fit_online or transform

Parameters **score_name** (*str*) – the name of the score to return

get_theta(*topic_names=None*)

Description get Theta matrix for training set of documents (or cached after transform)

Parameters **topic_names** (*list of str or str or None*) – list with topics or single topic to extract, None means all topics

Returns

- pandas.DataFrame: (data, columns, rows), where:
- columns — the ids of documents, for which the Theta matrix was requested;
- rows — the names of topics in topic model, that was used to create Theta;
- data — content of Theta matrix.

get_theta_sparse(*topic_names=None*, *eps=None*)

Description get Theta matrix in sparse format

Parameters

- **topic_names** (*list of str or str or None*) – list with topics or single topic to extract, None means all topics
- **eps** (*float*) – threshold to consider values as zero

Returns

- a 3-tuple of (data, rows, columns), where
- data — scipy.sparse.csr_matrix with values
- columns — the ids of documents;
- rows — the names of topics in topic model;

info

Description returns internal diagnostics information about the model

initialize(*dictionary=None*)

Description initialize topic model before learning

Parameters **dictionary** (*str or reference to Dictionary object*) – loaded BigARTM collection dictionary

library_version

Description the version of BigARTM library in a MAJOR.MINOR.PATCH format

load(*filename*, *model_name='p_wt'*)

Description loads from disk the topic model saved by ARTM.save()

Parameters

- **filename** (*str*) – the name of file containing model
- **model_name** (*str*) – the name of matrix to be saved, 'p_wt' or 'n_wt'

Note

- Loaded model will overwrite ARTM.topic_names and class_ids fields.
- All class_ids weights will be set to 1.0, you need to specify them by hand if it's necessary.
- The method call will empty ARTM.score_tracker.
- All regularizers and scores will be forgotten.
- etc.
- We strongly recommend you to reset all important parameters of the ARTM model, used earlier.

remove_theta()

Description removes cached theta matrix

reshape_topics(*topic_names*)

Description update topic names of the model.

Adds, removes, and reorders columns of phi matrices according to the new set of topic names. New topics are initialized with zeros.

save(*filename*, *model_name='p_wt'*)

Description saves one Phi-like matrix to disk

Parameters

- **filename** (*str*) – the name of file to store model
- **model_name** (*str*) – the name of matrix to be saved, 'p_wt' or 'n_wt'

topic_names

Description Gets or sets the list of topic names of the model.

Note

- Setting topic name allows you to put new labels on the existing topics. To add, remove or reorder topics use ARTM.reshape_topics() method.
- In ARTM topic names are used just as string identifiers, which give a unique name to each column of the phi matrix. Typically you want to set topic names as something like "topic0", "topic1", etc. Later operations like get_phi() allow you to specify which topics you need to retrieve. Most regularizers allow you to limit the set of topics they act upon. If you configure a rich set of regularizers it is important design your topic names according to how they are regularizerd. For example, you may use names obj0, obj1, ..., objN for *objective* topics (those where you enable sparsity regularizers), and back0, back1, ..., backM for *background* topics (those where you enable smoothing regularizers).

transform(*batch_vectorizer=None*, *theta_matrix_type='dense_theta'*, *predict_class_id=None*)

Description find Theta matrix for new documents

Parameters

- **batch_vectorizer** (*object_reference*) – an instance of BatchVectorizer class
- **theta_matrix_type** (*str*) – type of matrix to be returned, possible values: 'dense_theta', 'dense_ptdw', 'cache', None, default='dense_theta'
- **predict_class_id** (*str*) – class_id of a target modality to predict. When this option is enabled the resulting columns of theta matrix will correspond to unique labels of a target modality. The values will represent p(c|d), which give the probability of class label c for document d.

Returns

- pandas.DataFrame: (data, columns, rows), where:
- columns — the ids of documents, for which the Theta matrix was requested;
- rows — the names of topics in topic model, that was used to create Theta;
- data — content of Theta matrix.

Note

- 'dense_ptdw' mode provides simple access to values of p(t|w,d). The resulting pandas.DataFrame object will contain a flat theta matrix (no 3D) where each item has multiple columns - as many as the number of tokens in that document. These columns will have the same item_id. The order of columns with equal item_id is the same as the order of tokens in the input data (batch.item.token_id).

transform_sparse(*batch_vectorizer*, *eps=None*)

Description find Theta matrix for new documents as sparse scipy matrix

Parameters

- **batch_vectorizer** (*object_reference*) – an instance of BatchVectorizer class

- **eps** (*float*) – threshold to consider values as zero

Returns

- a 3-tuple of (data, rows, columns), where
- data — scipy.sparse.csr_matrix with values
- columns — the ids of documents;
- rows — the names of topics in topic model;

5.1.2 LDA model

This page describes LDA class.

class artm.**LDA**(*num_topics=None*, *num_processors=None*, *cache_theta=False*, *dictionary=None*, *num_document_passes=10*, *seed=-1*, *alpha=0.01*, *beta=0.01*, *theta_columns_naming='id'*)

__init__(*num_topics=None*, *num_processors=None*, *cache_theta=False*, *dictionary=None*, *num_document_passes=10*, *seed=-1*, *alpha=0.01*, *beta=0.01*, *theta_columns_naming='id'*)

Parameters

- **num_topics** (*int*) – the number of topics in model, will be overwrited if topic_names is set
- **num_processors** (*int*) – how many threads will be used for model training, if not specified then number of threads will be detected by the lib
- **cache_theta** (*bool*) – save or not the Theta matrix in model. Necessary if ARTM.get_theta() usage expects
- **num_document_passes** (*int*) – number of inner iterations over each document
- **dictionary** (*str or reference to Dictionary object*) – dictionary to be used for initialization, if None nothing will be done
- **reuse_theta** (*bool*) – reuse Theta from previous iteration or not
- **seed** (*unsigned int or -1*) – seed for random initialization, -1 means no seed
- **alpha** (*float*) – hyperparameter of Theta smoothing regularizer
- **beta** (*float or list of floats with len == num_topics*) – hyperparameter of Phi smoothing regularizer
- **theta_columns_naming** (*str*) – either 'id' or 'title', determines how to name columns (documents) in theta dataframe

Note

- the type (not value!) of beta should not change after initialization: if it was scalar - it should stay scalar, if it was list - it should stay list.

clone()

Description returns a deep copy of the artm.LDA object

Note

- This method is equivalent to copy.deepcopy() of your artm.LDA object. For more information refer to artm.ARTM.clone() method.

fit_offline(*batch_vectorizer*, *num_collection_passes=1*)

Description proceeds the learning of topic model in offline mode

Parameters

- **batch_vectorizer** (*object_referenece*) – an instance of BatchVectorizer class
- **num_collection_passes** (*int*) – number of iterations over whole given collection

fit_online(*batch_vectorizer*, *tau0=1024.0*, *kappa=0.7*, *update_every=1*)

Description proceeds the learning of topic model in online mode

Parameters

- **batch_vectorizer** (*object_reference*) – an instance of BatchVectorizer class
- **update_every** (*int*) – the number of batches; model will be updated once per it
- **tau0** (*float*) – coefficient (see 'Update formulas' paragraph)
- **kappa (float)** (*float*) – power for tau0, (see 'Update formulas' paragraph)
- **update_after** (*list of int*) – number of batches to be passed for Phi synchronizations

Update formulas

- The formulas for decay_weight and apply_weight:
- update_count = current_processed_docs / (batch_size * update_every);
- rho = pow(tau0 + update_count, -kappa);
- decay_weight = 1-rho;
- apply_weight = rho;

get_theta()

Description get Theta matrix for training set of documents

Returns

- pandas.DataFrame: (data, columns, rows), where:
- columns — the ids of documents, for which the Theta matrix was requested;
- rows — the names of topics in topic model, that was used to create Theta;
- data — content of Theta matrix.

get_top_tokens(*num_tokens=10*, *with_weights=False*)

Description returns most probable tokens for each topic

Parameters

- **num_tokens** (*int*) – number of top tokens to be returned
- **with_weights** (*bool*) – return only tokens, or tuples (token, its p_wt)

Returns

- list of lists of str, each internal list corresponds one topic in natural order, if with_weights == False, or list, or list of lists of tules, each tuple is (str, float)

initialize(*dictionary*)

Description initialize topic model before learning

Parameters **dictionary** (*str or reference to Dictionary object*) – loaded BigARTM collection dictionary

load(*filename*, *model_name='p_wt'*)

Description loads from disk the topic model saved by LDA.save()

Parameters

- **filename** (*str*) – the name of file containing model
- **model_name** (*str*) – the name of matrix to be saved, 'p_wt' or 'n_wt'

Note

- We strongly recommend you to reset all important parameters of the LDA model, used earlier.

remove_theta()

Description removes cached theta matrix

save(*filename*, *model_name='p_wt'*)

Description saves one Phi-like matrix to disk

Parameters

- **filename** (*str*) – the name of file to store model
- **model_name** (*str*) – the name of matrix to be saved, 'p_wt' or 'n_wt'

transform(*batch_vectorizer*, *theta_matrix_type='dense_theta'*)

Description find Theta matrix for new documents

Parameters

- **batch_vectorizer** (*object_reference*) – an instance of BatchVectorizer class
- **theta_matrix_type** (*str*) – type of matrix to be returned, possible values: 'dense_theta', None, default='dense_theta'

Returns

- pandas.DataFrame: (data, columns, rows), where:
- columns — the ids of documents, for which the Theta matrix was requested;
- rows — the names of topics in topic model, that was used to create Theta;
- data — content of Theta matrix.

5.1.3 hARTM

This page describes hARTM class.

class artm.**hARTM**(*num_processors=None*, *class_ids=None*, *scores=None*, *regularizers=None*, *num_document_passes=10*, *reuse_theta=False*, *dictionary=None*, *cache_theta=False*, *theta_columns_naming='id'*, *seed=-1*, *tmp_files_path=''*)

__init__(*num_processors=None*, *class_ids=None*, *scores=None*, *regularizers=None*, *num_document_passes=10*, *reuse_theta=False*, *dictionary=None*, *cache_theta=False*, *theta_columns_naming='id'*, *seed=-1*, *tmp_files_path=''*)

Description a class for constructing topic hierarchy that is a sequence of tied artm.ARTM() models (levels)

Parameters

- **num_processors** (*int*) – how many threads will be used for model training, if not specified then number of threads will be detected by the lib
- **class_ids** (*dict*) – list of class_ids and their weights to be used in model, key — class_id, value — weight, if not specified then all class_ids will be used
- **cache_theta** (*bool*) – save or not the Theta matrix in model. Necessary if ARTM.get_theta() usage expects
- **scores** (*list*) – list of scores (objects of artm.*Score classes)
- **regularizers** (*list*) – list with regularizers (objects of artm.*Regularizer classes)
- **num_document_passes** (*int*) – number of inner iterations over each document
- **dictionary** (*str or reference to Dictionary object*) – dictionary to be used for initialization, if None nothing will be done
- **reuse_theta** (*bool*) – reuse Theta from previous iteration or not
- **theta_columns_naming** (*str*) – either 'id' or 'title', determines how to name columns (documents) in theta dataframe
- **seed** (*unsigned int or -1*) – seed for random initialization, -1 means no seed
- **tmp_files_path** (*str*) – a path where to save temporary files (temporary solution), default value: current directory

Usage

- **to construct hierarchy you have to learn several ARTM models:** hier = artm.hARTM() level0 = hier.add_level(num_topics=5) # returns artm.ARTM() instance # work with level0 as with usual model level1 = hier.add_level(num_topics=25, parent_level_weight=1) # work with level1 as with usual model # ...
- **to get the i-th level's model, use**

  level = hier[i]

  or level = hier.get_level(i)

- **to iterate through levels use**

  for level in hier: # some work with level

- method hier.del_level(...) removes i-th level and all levels after it
- other hARTM methods correspond to those in ARTM class and call them sequantially for all levels of hierarchy from 0 to the last one. For example, to fit levels offline you may call fit_offline method of hARTM instance or of each level individually.

add_level(*num_topics=None*, *topic_names=None*, *parent_level_weight=1*)

Description adds new level to the hierarchy

Parameters

- **num_topics** (*int*) – the number of topics in level model, will be overwriten if parameter topic_names is set
- **topic_names** (*list of str*) – names of topics in model

- **parent_level_weight** (*float*) – the coefficient of smoothing n_wt by n_wa, a enumerates parent topics

Returns ARTM or derived ARTM_Level instance

Notes

- hierarchy structure assumes the number of topics on each following level is greater than on previous one
- work with returned value as with usual ARTM model
- to access any level, use [] or get_level method
- Important! You cannot add next level before previous one is initialized and fit.

clone()

Description returns a deep copy of the artm.hARTM object

Note

- This method is equivalent to copy.deepcopy() of your artm.hARTM object. For more information refer to artm.ARTM.clone() method.

del_level(*level_idx*)

Description removes i-th level and all following levels.

Parameters **level_idx** (*int*) – the number of level from what to start removing if -1, the last level is removed

dispose()

Description free all native memory, allocated for this hierarchy

Note

- This method does not free memory occupied by models' dictionaries, because dictionaries are shared across all models
- hARTM class implements __exit__ and __del__ methods, which automatically call dispose.

fit_offline(*batch_vectorizer*, *num_collection_passes=1*)

Description proceeds the learning of all hirarchy levels from 0 to the last one

Parameters

- **batch_vectorizer** (*object_referenece*) – an instance of BatchVectorizer class
- **num_collection_passes** (*int*) – number of iterations over whole given collection for each level

Note

- You cannot add add next level before previous one is fit. So use this method only when all levels are added, initialized and fit, for example, when you added one more regularizer or loaded hierarchy from disk.

get_level(*level_idx*)

Description access level

Parameters **level_idx** (*int*) – the number of level to return

Returns specified level that is ARTM or derived ARTM_Level instance

get_phi (*class_ids=None*, *model_name=None*)

Description get level-wise horizontally stacked Phi matrices

Parameters

- **class_ids** (*list of str or str*) – list with class_ids or single class_id to extract, None means all class ids
- **model_name** (*str*) – self.model_pwt by default, self.model_nwt is also reasonable to extract unnormalized counters

Returns

- pandas.DataFrame: (data, columns, rows), where:
- **columns — the names of topics in format level_X_Y** where X is level index and Y is topic name;
- rows — the tokens of topic model;
- data — content of Phi matrix.

Note

- if you need to extract specified topics, use get_phi() method of individual level model

get_theta (*topic_names=None*)

Description get level-wise vertically stacked Theta matrices for training set of documents

Parameters **topic_names** (*list of str*) – list with topics to extract, None means all topics

Returns

- pandas.DataFrame: (data, columns, rows), where:
- columns — the ids of documents, for which the Theta matrix was requested;
- **rows — the names of topics in format level_X_Y** where X is level index and Y is topic name;
- data — content of Theta matrix.

library_version

Description the version of BigARTM library in a MAJOR.MINOR.PATCH format

load (*path*)

Description loads models of already constructed hierarchy

Parameters **path** (*str*) – a path where hierarchy was saved by hARTM.save method

Notes

- Loaded models will overwrite ARTM.topic_names and class_ids fields of each level.
- All class_ids weights will be set to 1.0, you need to specify them by hand if it's necessary.
- The method call will empty ARTM.score_tracker of each level.
- All regularizers and scores will be forgotten.
- etc.

- We strongly recommend you to reset all important parameters of the ARTM models and hARTM, used earlier.

save(*path*)

Description saves all levels

Parameters **path** (*str*) – a path where to save hierarchy files This must be existing empty path, otherwise exception is raised

transform(*batch_vectorizer*)

Description get level-wise vertically stacked Theta matrices for new documents

Parameters **batch_vectorizer** (*object_reference*) – an instance of BatchVectorizer class

Returns

- pandas.DataFrame: (data, columns, rows), where:
- columns — the ids of documents, for which the Theta matrix was requested;
- **rows — the names of topics in format level_X_Y** where X is level index and Y is topic name;
- data — content of Theta matrix.

Note

- to access p(tld, w) matrix or to predict class use transform method of hierarchy level individually

5.1.4 Batches Utils

This page describes BatchVectorizer class.

class artm.**BatchVectorizer**(*batches=None*, *collection_name=None*, *data_path=''*, *data_format='batches'*, *target_folder=None*, *batch_size=1000*, *batch_name_type='code'*, *data_weight=1.0*, *n_wd=None*, *vocabulary=None*, *gather_dictionary=True*, *class_ids=None*)

__init__(*batches=None*, *collection_name=None*, *data_path=''*, *data_format='batches'*, *target_folder=None*, *batch_size=1000*, *batch_name_type='code'*, *data_weight=1.0*, *n_wd=None*, *vocabulary=None*, *gather_dictionary=True*, *class_ids=None*)

Parameters

- **collection_name** (*str*) – the name of text collection (required if data_format == 'bow_uci')
- **data_path** (*str*) –

  1. if data_format == 'bow_uci' => folder containing 'docword.collection_name.txt' and vocab.collection_name.txt files; 2) if data_format == 'vowpal_wabbit' => file in Vowpal Wabbit format; 3) if data_format == 'bow_n_wd' => useless parameter 4) if data_format == 'batches' => folder containing batches

- **data_format** (*str*) – the type of input data: 1) 'bow_uci' — Bag-Of-Words in UCI format; 2) 'vowpal_wabbit' — Vowpal Wabbit format; 3 'bow_n_wd' — result of CountVectorizer or similar tool; 4) 'batches' — the BigARTM data format
- **batch_size** (*int*) – number of documents to be stored in each batch

- **target_folder** (*str*) – full path to folder for future batches storing; if not set, no batches will be produced for further work
- **batches** (*list of str*) – list with non-full file names of batches (necessary parameters are batches + data_path + data_fromat=='batches' in this case)
- **batch_name_type** (*str*) – name batches in natural order ('code') or using random guids (guid)
- **data_weight** (*float*) – weight for a group of batches from data_path; it can be a list of floats, then data_path (and target_folder if not data_format == 'batches') should also be lists; one weight corresponds to one path from the data_path list;
- **n_wd** (*array*) – matrix with n_wd counters
- **vocabulary** (*dict*) – dict with vocabulary, key - index of n_wd, value - token
- **gather_dictionary** (*bool*) – create or not the default dictionary in vectorizer; if data_format == 'bow_n_wd' - automatically set to True; and if data_weight is list - automatically set to False
- **class_ids** (*list of str or str*) – list of class_ids or single class_id to parse and include in batches

batch_size

Returns the user-defined size of the batches

batches_list

Returns list of batches names

data_path

Returns the disk path of batches

dictionary

Returns Dictionary object, if parameter gather_dictionary was True, else None

num_batches

Returns the number of batches

weights

Returns list of batches weights

5.1.5 Dictionary

This page describes Dictionary class.

class artm.**Dictionary**(*name=None*, *dictionary_path=None*, *data_path=None*)

__init__(*name=None*, *dictionary_path=None*, *data_path=None*)

Parameters

- **name** (*str*) – name of the dictionary
- **dictionary_path** (*str*) – can be used for default call of load() method in constructor
- **data_path** (*str*) – can be used for default call of gather() method in constructor

Note: all parameters are optional

create(*dictionary_data*)

Description creates dictionary using DictionaryData object

Parameters **dictionary_data** (*DictionaryData instance*) – configuration of dictionary

filter(*class_id=None*, *min_df=None*, *max_df=None*, *min_df_rate=None*, *max_df_rate=None*, *min_tf=None*, *max_tf=None*, *max_dictionary_size=None*)

Description filters the BigARTM dictionary of the collection, which was already loaded into the lib

Parameters

- **dictionary_name** (*str*) – name of the dictionary in the lib to filter
- **dictionary_target_name** (*str*) – name for the new filtered dictionary in the lib
- **class_id** (*str*) – class_id to filter
- **min_df** (*float*) – min df value to pass the filter
- **max_df** (*float*) – max df value to pass the filter
- **min_df_rate** (*float*) – min df rate to pass the filter
- **max_df_rate** (*float*) – max df rate to pass the filter
- **min_tf** (*float*) – min tf value to pass the filter
- **max_tf** (*float*) – max tf value to pass the filter
- **max_dictionary_size** (*float*) – give an easy option to limit dictionary size; rare tokens will be excluded until dictionary reaches given size.

Note the current dictionary will be replaced with filtered

gather(*data_path*, *cooc_file_path=None*, *vocab_file_path=None*, *symmetric_cooc_values=False*)

Description creates the BigARTM dictionary of the collection, represented as batches and load it in the lib

Parameters

- **data_path** (*str*) – full path to batches folder
- **cooc_file_path** (*str*) – full path to the file with cooc info. Cooc info is a file with three columns, first two a the zero-based indices of tokens in vocab file, and third one is a value of their coocurance in collection (or another) pairwise statistic.
- **vocab_file_path** (*str*) – full path to the file with vocabulary. If given, the dictionary token will have the same order, as in this file, otherwise the order will be random. If given, the tokens from batches, that are not presented in vocab, will be skipped.
- **symmetric_cooc_values** (*bool*) – if the cooc matrix should considered to be symmetric or not

load(*dictionary_path*)

Description loads the BigARTM dictionary of the collection into the lib

Parameters **dictionary_path** (*str*) – full filename of the dictionary

load_text (*dictionary_path*, *encoding='utf-8'*)

Description loads the BigARTM dictionary of the collection from the disk in the human readable text format

Parameters

- **dictionary_path** (*str*) – full file name of the text dictionary file
- **encoding** (*str*) – an encoding of text in diciotnary

save (*dictionary_path*)

Description saves the BigARTM dictionary of the collection on the disk

Parameters **dictionary_path** (*str*) – full file name for the dictionary

save_text (*dictionary_path*, *encoding='utf-8'*)

Description saves the BigARTM dictionary of the collection on the disk in the human readable text format

Parameters

- **dictionary_path** (*str*) – full file name for the text dictionary file
- **encoding** (*str*) – an encoding of text in diciotnary

5.1.6 Regularizers

This page describes *KlFunctionInfo* and *Regularizer* classes.

See detailed description of regularizers Regularizers Description for understanding their sense.

class artm.**KlFunctionInfo** (*function_type='log'*, *power_value=2.0*)

__init__ (*function_type='log'*, *power_value=2.0*)

Parameters

- **function_type** (*str*) – the type of function, 'log' (logarithm) or 'pol' (polynomial)
- **power_value** (*float*) – the double power of polynomial, ignored if type = 'log'

class artm.**SmoothSparsePhiRegularizer** (*name=None*, *tau=1.0*, *gamma=None*, *class_ids=None*, *topic_names=None*, *dictionary=None*, *kl_function_info=None*, *config=None*)

__init__ (*name=None*, *tau=1.0*, *gamma=None*, *class_ids=None*, *topic_names=None*, *dictionary=None*, *kl_function_info=None*, *config=None*)

Parameters

- **name** (*str*) – the identifier of regularizer, will be auto-generated if not specified
- **tau** (*float*) – the coefficient of regularization for this regularizer
- **gamma** (*float*) – the coefficient of relative regularization for this regularizer
- **class_ids** (*list of str or str or None*) – list of class_ids or single class_id to regularize, will regularize all classes if empty or None
- **topic_names** (*list of str or single str or None*) – list of names or single name of topic to regularize, will regularize all topics if empty or None

- **dictionary** (*str or reference to Dictionary object*) – BigARTM collection dictionary, won't use dictionary if not specified
- **kl_function_info** (*KlFunctionInfo object*) – class with additional info about function under KL-div in regularizer
- **config** (*protobuf object*) – the low-level config of this regularizer

class artm.**SmoothSparseThetaRegularizer**(*name=None*, *tau=1.0*, *topic_names=None*, *alpha_iter=None*, *kl_function_info=None*, *doc_titles=None*, *doc_topic_coef=None*, *config=None*)

__init__(*name=None*, *tau=1.0*, *topic_names=None*, *alpha_iter=None*, *kl_function_info=None*, *doc_titles=None*, *doc_topic_coef=None*, *config=None*)

Parameters

- **name** (*str*) – the identifier of regularizer, will be auto-generated if not specified
- **tau** (*float*) – the coefficient of regularization for this regularizer
- **alpha_iter** (*list of str*) – list of additional coefficients of regularization on each iteration over document. Should have length equal to model.num_document_passes
- **topic_names** (*list of str or single str or None*) – list of names or single name of topic to regularize, will regularize all topics if empty or None
- **kl_function_info** (*KlFunctionInfo object*) – class with additional info about function under KL-div in regularizer
- **doc_titles** (*list of strings*) – list of titles of documents to be processed by this regularizer. Default empty value means processing of all documents. User should guarantee the existence and correctness of document titles in batches (e.g. in src files with data, like WV).
- **doc_topic_coef** (*list of doubles or list of lists of doubles*) – Two cases: 1) list of doubles with length equal to num of topics. Means additional multiplier in M-step formula besides alpha and tau, unique for each topic, but general for all processing documents. 2) list of lists of doubles with outer list length equal to length of doc_titles, and each inner list length equal to num of topics. Means case 1 with unique list of additional multipliers for each document from doc_titles. Other documents will not be regularized according to description of doc_titles parameter. Note, that doc_topic_coef and topic_names are both using.
- **config** (*protobuf object*) – the low-level config of this regularizer

class artm.**DecorrelatorPhiRegularizer**(*name=None*, *tau=1.0*, *gamma=None*, *class_ids=None*, *topic_names=None*, *config=None*)

__init__(*name=None*, *tau=1.0*, *gamma=None*, *class_ids=None*, *topic_names=None*, *config=None*)

Parameters

- **name** (*str*) – the identifier of regularizer, will be auto-generated if not specified
- **tau** (*float*) – the coefficient of regularization for this regularizer
- **gamma** (*float*) – the coefficient of relative regularization for this regularizer
- **class_ids** (*list of str or str or None*) – list of class_ids or single class_id to regularize, will regularize all classes if empty or None

- **topic_names** (*list of str or single str or None*) – list of names or single name of topic to regularize, will regularize all topics if empty or None
- **config** (*protobuf object*) – the low-level config of this regularizer

class artm.**LabelRegularizationPhiRegularizer**(*name=None*, *tau=1.0*, *gamma=None*, *class_ids=None*, *topic_names=None*, *dictionary=None*, *config=None*)

__init__(*name=None*, *tau=1.0*, *gamma=None*, *class_ids=None*, *topic_names=None*, *dictionary=None*, *config=None*)

Parameters

- **name** (*str*) – the identifier of regularizer, will be auto-generated if not specified
- **tau** (*float*) – the coefficient of regularization for this regularizer
- **gamma** (*float*) – the coefficient of relative regularization for this regularizer
- **class_ids** (*list of str or str or None*) – list of class_ids or single class_id to regularize, will regularize all classes if empty or None
- **topic_names** (*list of str or single str or None*) – list of names or single name of topic to regularize, will regularize all topics if empty or None
- **dictionary** (*str or reference to Dictionary object*) – BigARTM collection dictionary, won't use dictionary if not specified
- **config** (*protobuf object*) – the low-level config of this regularizer

class artm.**SpecifiedSparsePhiRegularizer**(*name=None*, *tau=1.0*, *gamma=None*, *topic_names=None*, *class_id=None*, *num_max_elements=None*, *probability_threshold=None*, *sparse_by_columns=True*, *config=None*)

__init__(*name=None*, *tau=1.0*, *gamma=None*, *topic_names=None*, *class_id=None*, *num_max_elements=None*, *probability_threshold=None*, *sparse_by_columns=True*, *config=None*)

Parameters

- **name** (*str*) – the identifier of regularizer, will be auto-generated if not specified
- **tau** (*float*) – the coefficient of regularization for this regularizer
- **gamma** (*float*) – the coefficient of relative regularization for this regularizer
- **class_id** – class_id to regularize
- **topic_names** (*list of str or single str or None*) – list of names or single name of topic to regularize, will regularize all topics if empty or None
- **num_max_elements** (*int*) – number of elements to save in row/column
- **probability_threshold** (*float*) – if m elements in row/column sum into value >= probability_threshold, m < n => only these elements would be saved. Value should be in (0, 1), default=None
- **sparse_by_columns** (*bool*) – find max elements in column or in row
- **config** (*protobuf object*) – the low-level config of this regularizer

class artm.**ImproveCoherencePhiRegularizer**(*name=None*, *tau=1.0*, *gamma=None*, *class_ids=None*, *topic_names=None*, *dictionary=None*, *config=None*)

__init__(*name=None*, *tau=1.0*, *gamma=None*, *class_ids=None*, *topic_names=None*, *dictionary=None*, *config=None*)

Parameters

- **name** (*str*) – the identifier of regularizer, will be auto-generated if not specified
- **tau** (*float*) – the coefficient of regularization for this regularizer
- **gamma** (*float*) – the coefficient of relative regularization for this regularizer
- **class_ids** (*list of str or str or None*) – list of class_ids or single class_id to regularize, will regularize all classes if empty or None dictionaty should contain pairwise tokens coocurancy info
- **topic_names** (*list of str or single str or None*) – list of names or single name of topic to regularize, will regularize all topics if empty or None
- **dictionary** (*str or reference to Dictionary object*) – BigARTM collection dictionary, won't use dictionary if not specified, in this case regularizer is useless
- **config** (*protobuf object*) – the low-level config of this regularizer

class artm.**SmoothPtdwRegularizer**(*name=None*, *tau=1.0*, *config=None*)

__init__(*name=None*, *tau=1.0*, *config=None*)

Parameters

- **name** (*str*) – the identifier of regularizer, will be auto-generated if not specified
- **tau** (*float*) – the coefficient of regularization for this regularizer
- **config** (*protobuf object*) – the low-level config of this regularizer

class artm.**TopicSelectionThetaRegularizer**(*name=None*, *tau=1.0*, *topic_names=None*, *alpha_iter=None*, *config=None*)

__init__(*name=None*, *tau=1.0*, *topic_names=None*, *alpha_iter=None*, *config=None*)

Parameters

- **name** (*str*) – the identifier of regularizer, will be auto-generated if not specified
- **tau** (*float*) – the coefficient of regularization for this regularizer
- **alpha_iter** (*list of str*) – list of additional coefficients of regularization on each iteration over document. Should have length equal to model.num_document_passes
- **topic_names** (*list of str or single str or None*) – list of names or single name of topic to regularize, will regularize all topics if empty or None
- **config** (*protobuf object*) – the low-level config of this regularizer

class artm.**TopicSegmentationPtdwRegularizer**(*name=None*, *window=None*, *threshold=None*, *background_topic_names=None*, *config=None*)

__init__(*name=None*, *window=None*, *threshold=None*, *background_topic_names=None*, *config=None*)

Parameters

- **name** (*str*) – the identifier of regularizer, will be auto-generated if not specified
- **window** (*int*) – a number of words to the one side over which smoothing will be performed
- **threshold** (*float*) – probability threshold for a word to be a topic-changing word
- **background_topic_names** (*list of str or str or None*) – list of names or single name of topic to be considered background, will not consider background topics if empty or None
- **config** (*protobuf object*) – the low-level config of this regularizer

5.1.7 Scores

This page describes *Scores classes.

See detailed descrition of scores Scores Description for understanding their sense.

class artm.**SparsityPhiScore**(*name=None*, *class_id=None*, *topic_names=None*, *model_name=None*, *eps=None*)

__init__(*name=None*, *class_id=None*, *topic_names=None*, *model_name=None*, *eps=None*)

Parameters

- **name** (*str*) – the identifier of score, will be auto-generated if not specified
- **class_id** (*str*) – class_id to score
- **topic_names** (*list of str or str or None*) – list of names or single name of topic to regularize, will score all topics if empty or None
- **model_name** – phi-like matrix to be scored (typically 'pwt' or 'nwt'), 'pwt' if not specified
- **eps** (*float*) – the tolerance const, everything < eps considered to be zero

class artm.**ItemsProcessedScore**(*name=None*)

__init__(*name=None*)

Parameters **name** (*str*) – the identifier of score, will be auto-generated if not specified

class artm.**PerplexityScore**(*name=None*, *class_ids=None*, *topic_names=None*, *dictionary=None*)

__init__(*name=None*, *class_ids=None*, *topic_names=None*, *dictionary=None*)

Parameters

- **name** (*str*) – the identifier of score, will be auto-generated if not specified
- **class_ids** (*list of str*) – class_id to score, means that tokens of all class_ids will be used
- **dictionary** (*str or reference to Dictionary object*) – BigARTM collection dictionary, is strongly recommended to be used for correct replacing of zero counters.

class artm.**SparsityThetaScore**(*name=None*, *topic_names=None*, *eps=None*)

__init__(*name=None*, *topic_names=None*, *eps=None*)

Parameters

- **name** (*str*) – the identifier of score, will be auto-generated if not specified
- **topic_names** (*list of str or str or None*) – list of names or single name of topic to regularize, will score all topics if empty or None
- **eps** (*float*) – the tolerance const, everything < eps considered to be zero

class artm.**ThetaSnippetScore**(*name=None*, *item_ids=None*, *num_items=None*)

__init__(*name=None*, *item_ids=None*, *num_items=None*)

Parameters

- **name** (*str*) – the identifier of score, will be auto-generated if not specified
- **item_ids** (*list of int*) – list of names of items to show, default=None
- **num_items** (*int*) – number of theta vectors to show from the beginning (no sense if item_ids was given)

class artm.**TopicKernelScore**(*name=None*, *class_id=None*, *topic_names=None*, *eps=None*, *dictionary=None*, *probability_mass_threshold=None*)

__init__(*name=None*, *class_id=None*, *topic_names=None*, *eps=None*, *dictionary=None*, *probability_mass_threshold=None*)

Parameters

- **name** (*str*) – the identifier of score, will be auto-generated if not specified
- **class_id** (*str*) – class_id to score
- **topic_names** (*list of str or str or None*) – list of names or single name of topic to regularize, will score all topics if empty or None
- **probability_mass_threshold** (*float*) – the threshold for p(tlw) values to get token into topic kernel. Should be in (0, 1)
- **dictionary** (*str or reference to Dictionary object*) – BigARTM collection dictionary, won't use dictionary if not specified
- **eps** (*float*) – the tolerance const, everything < eps considered to be zero

class artm.**TopTokensScore**(*name=None*, *class_id=None*, *topic_names=None*, *num_tokens=None*, *dictionary=None*)

__init__(*name=None*, *class_id=None*, *topic_names=None*, *num_tokens=None*, *dictionary=None*)

Parameters

- **name** (*str*) – the identifier of score, will be auto-generated if not specified
- **class_id** (*str*) – class_id to score
- **topic_names** (*list of str or str or None*) – list of names or single name of topic to regularize, will score all topics if empty or None
- **num_tokens** (*int*) – number of tokens with max probability in each topic
- **dictionary** (*str or reference to Dictionary object*) – BigARTM collection dictionary, won't use dictionary if not specified

class artm.**TopicMassPhiScore**(*name=None*, *class_id=None*, *topic_names=None*, *model_name=None*, *eps=None*)

__init__(*name=None*, *class_id=None*, *topic_names=None*, *model_name=None*, *eps=None*)

Parameters

- **name** (*str*) – the identifier of score, will be auto-generated if not specified
- **class_id** (*str*) – class_id to score
- **topic_names** (*list of str or str or None*) – list of names or single name of topic to regularize, will score all topics if empty or None
- **model_name** – phi-like matrix to be scored (typically 'pwt' or 'nwt'), 'pwt' if not specified
- **eps** (*float*) – the tolerance const, everything < eps considered to be zero

class artm.**ClassPrecisionScore**(*name=None*)

__init__(*name=None*)

Parameters **name** (*str*) – the identifier of score, will be auto-generated if not specified

class artm.**BackgroundTokensRatioScore**(*name=None*, *class_id=None*, *delta_threshold=None*, *save_tokens=None*, *direct_kl=None*)

__init__(*name=None*, *class_id=None*, *delta_threshold=None*, *save_tokens=None*, *direct_kl=None*)

Parameters

- **name** (*str*) – the identifier of score, will be auto-generated if not specified
- **class_id** (*str*) – class_id to score
- **delta_threshold** (*float*) – the threshold for KL-div between p(t|w) and p(t) to get token into background. Should be non-negative
- **save_tokens** (*bool*) – save background tokens or not, save if field not specified
- **direct_kl** (*bool*) – use KL(p(t) || p(t|w)) or via versa, true if field not specified

5.1.8 Score Tracker

This page describes *ScoreTracker classes.

class artm.score_tracker.**SparsityPhiScoreTracker**(*score*)

__init__(*score*)

Properties

•Note: every field is a list of info about score on all synchronizations.

•value - values of Phi sparsity.

•zero_tokens - number of zero rows in Phi.

•total_tokens - number of all rows in Phi.

•Note: every field has a version with prefix '**last_**', means retrieving only info about the last synchronization.

class artm.score_tracker.**SparsityThetaScoreTracker**(*score*)

__init__(*score*)

Properties

•Note: every field is a list of info about score on all synchronizations.

•value - values of Theta sparsity.

•zero_topics - number of zero rows in Theta.

•total_topics - number of all rows in Theta.

•Note: every field has a version with prefix '**last_**', means retrieving only info about the last synchronization.

class artm.score_tracker.**PerplexityScoreTracker**(*score*)

__init__(*score*)

Properties

•Note: every field is a list of info about score on all synchronizations.

•value - values of perplexity.

•raw - raw values in formula for perplexity.

•normalizer - normalizer values in formula for perplexity.

•zero_tokens - number of zero p(w|d) = sum_t p(w|t) p(t|d).

•Note: every field has a version with prefix '**last_**', means retrieving only info about the last synchronization.

class artm.score_tracker.**TopTokensScoreTracker**(*score*)

__init__(*score*)

Properties

•Note: every field is a list of info about score on all synchronizations.

•num_tokens - number of requested top tokens.

•coherence - each element of list is a dict, key - topic name, value - topic coherence counted using top-tokens

•average_coherence - average coherencies of all scored topics.

•tokens - each element of list is a dict, key - topic name, value - list of top-tokens

•weights - each element of list is a dict, key - topic name, value - list of weights of corresponding top-tokens (weight of token == p(w|t))

•Note: every field has a version with prefix '**last_**', means retrieving only info about the last synchronization.

class artm.score_tracker.**TopicKernelScoreTracker**(*score*)

__init__(*score*)

Properties

•Note: every field is a list of info about score on all synchronizations.

•tokens - each element of list is a dict, key - topic name, value - list of kernel tokens

•size - each element of list is a dict, key - topic name, value - kernel size

•contrast - each element of list is a dict, key - topic name, value - kernel contrast

•purity - each element of list is a dict, key - topic name, value - kernel purity

•coherence - each element of list is a dict, key - topic name, value - topic coherence counted using kernel tokens

•average_size - average kernel size of all scored topics.

•average_contrast - average kernel contrast of all scored topics.

•average_purity - average kernel purity of all scored topics.

•average_coherence - average coherencies of all scored topics.

•Note: every field has a version with prefix '**last_**', means retrieving only info about the last synchronization.

class artm.score_tracker.**ItemsProcessedScoreTracker**(*score*)

__init__(*score*)

Properties

•Note: every field is a list of info about score on all synchronizations.

•value - numbers of processed documents.

•Note: every field has a version with prefix '**last_**', means retrieving only info about the last synchronization.

class artm.score_tracker.**ThetaSnippetScoreTracker**(*score*)

__init__(*score*)

Properties

•Note: every field is a list of info about score on all synchronizations.

•document_ids - each element of list is a list of ids of returned documents.

•snippet - each element of list is a dict, key - doc id, value - list with corresponding p(t|d) values.

•Note: every field has a version with prefix '**last_**', means retrieving only info about the last synchronization.

class artm.score_tracker.**TopicMassPhiScoreTracker**(*score*)

__init__(*score*)

Properties

•Note: every field is a list of info about score on all synchronizations.

•value - values of ratio of sum_t n_t of scored topics.and all topics

•topic_mass - each value is a dict, key - topic name, value - topic mass n_t

•topic_ratio - each value is a dict, key - topic name, value - topic ratio

•Note: every field has a version with prefix '**last_**', means retrieving only info about the last synchronization.

class artm.score_tracker.**ClassPrecisionScoreTracker**(*score*)

__init__(*score*)

Properties

•Note: every field is a list of info about score on all synchronizations.

•value - values of ratio of correct predictions.

•error - numbers of error predictiona.

•total - numbers of all predictions.

•Note: every field has a version with prefix '**last_**', means retrieving only info about the last synchronization.

class artm.score_tracker.**BackgroundTokensRatioScoreTracker**(*score*)

__init__(*score*)

Properties

•Note: every field is a list of info about score on all synchronizations.

•value - values of part of background tokens.

•tokens - each element of list is a lists of background tokens (can be acceced if 'save_tokens' was True)

•Note: every field has a version with prefix '**last_**', means retrieving only info about the last synchronization.

5.1.9 Master Component

This page describes MasterComponent class.

class artm.**MasterComponent**(*library=None*, *topic_names=None*, *class_ids=None*, *scores=None*, *regularizers=None*, *num_processors=None*, *pwt_name=None*, *nwt_name=None*, *num_document_passes=None*, *reuse_theta=None*, *cache_theta=False*, *config=None*, *master_id=None*)

__init__(*library=None*, *topic_names=None*, *class_ids=None*, *scores=None*, *regularizers=None*, *num_processors=None*, *pwt_name=None*, *nwt_name=None*, *num_document_passes=None*, *reuse_theta=None*, *cache_theta=False*, *config=None*, *master_id=None*)

Parameters

- **library** – an instance of LibArtm

- **topic_names** (*list of str*) – list of topic names to use in model
- **class_ids** (*dict*) – key - class_id, value - class_weight
- **scores** (*dict*) – key - score name, value - config
- **regularizers** (*dict*) – key - regularizer name, value - tuple (config, tau) or triple (config, tau, gamma)
- **num_processors** (*int*) – number of worker threads to use for processing the collection
- **pwt_name** (*str*) – name of pwt matrix
- **nwt_name** (*str*) – name of nwt matrix
- **num_document_passes** (*in*) – num passes through each document
- **reuse_theta** (*bool*) – reuse Theta from previous iteration or not
- **cache_theta** (*bool*) – save or not the Theta matrix

attach_model(*model*)

Parameters **model** (*str*) – name of matrix in BigARTM

Returns

- messahes.TopicModel() object with info about Phi matrix
- numpy.ndarray with Phi data (i.e., p(w|t) values)

clear_score_array_cache()
Clears all entries from score array cache

clear_score_cache()
Clears all entries from score cache

clear_theta_cache()
Clears all entries from theta matrix cache

create_dictionary(*dictionary_data*, *dictionary_name=None*)

Parameters

- **dictionary_data** – an instance of DictionaryData with info about dictionary
- **dictionary_name** (*str*) – name of exported dictionary

create_regularizer(*name*, *config*, *tau*, *gamma=None*)

Parameters

- **name** (*str*) – the name of the future regularizer
- **config** – the config of the future regularizer
- **tau** (*float*) – the coefficient of the regularization

create_score(*name*, *config*, *model_name=None*)

Parameters

- **name** (*str*) – the name of the future score
- **config** – an instance of ***ScoreConfig

export_dictionary(*filename*, *dictionary_name*)

Parameters

- **filename** (*str*) – full name of dictionary file
- **dictionary_name** (*str*) – name of exported dictionary

export_model(*model*, *filename*)

filter_dictionary(*dictionary_name=None*, *dictionary_target_name=None*, *class_id=None*, *min_df=None*, *max_df=None*, *min_df_rate=None*, *max_df_rate=None*, *min_tf=None*, *max_tf=None*, *max_dictionary_size=None*, *args=None*)

Parameters

- **dictionary_name** (*str*) – name of the dictionary in the core to filter
- **dictionary_target_name** (*str*) – name for the new filtered dictionary in the core
- **class_id** (*str*) – class_id to filter
- **min_df** (*float*) – min df value to pass the filter
- **max_df** (*float*) – max df value to pass the filter
- **min_df_rate** (*float*) – min df rate to pass the filter
- **max_df_rate** (*float*) – max df rate to pass the filter
- **min_tf** (*float*) – min tf value to pass the filter
- **max_tf** (*float*) – max tf value to pass the filter
- **max_dictionary_size** (*float*) – give an easy option to limit dictionary size; rare tokens will be excluded until dictionary reaches given size.
- **args** – an instance of FilterDictionaryArgs

fit_offline(*batch_filenames=None*, *batch_weights=None*, *num_collection_passes=None*, *batches_folder=None*)

Parameters

- **batch_filenames** (*list of str*) – name of batches to process
- **batch_weights** (*list of float*) – weights of batches to process
- **num_collection_passes** (*int*) – number of outer iterations
- **batches_folder** (*str*) – folder containing batches to process

fit_online(*batch_filenames=None*, *batch_weights=None*, *update_after=None*, *apply_weight=None*, *decay_weight=None*, *async=None*)

Parameters

- **batch_filenames** (*list of str*) – name of batches to process
- **batch_weights** (*list of float*) – weights of batches to process
- **update_after** (*list of int*) – number of batches to be passed for Phi synchronizations
- **apply_weight** (*list of float*) – weight of applying new counters (len == len of update_after)
- **decay_weight** (*list of float*) – weight of applying old counters (len == len of update_after)
- **async** (*bool*) – whether to use the async implementation of the EM-algorithm or not

gather_dictionary(*dictionary_target_name=None*, *data_path=None*, *cooc_file_path=None*, *vocab_file_path=None*, *symmetric_cooc_values=None*, *args=None*)

Parameters

- **dictionary_target_name** (*str*) – name of the dictionary in the core
- **data_path** (*str*) – full path to batches folder
- **cooc_file_path** (*str*) – full path to the file with cooc info
- **vocab_file_path** (*str*) – full path to the file with vocabulary
- **symmetric_cooc_values** (*bool*) – whether the cooc matrix should considered to be symmetric or not
- **args** – an instance of GatherDictionaryArgs

get_dictionary(*dictionary_name*)

Parameters **dictionary_name** (*str*) – name of dictionary to get

get_info()

get_phi_info(*model*)

Parameters **model** (*str*) – name of matrix in BigARTM

Returns messages.TopicModel object

get_phi_matrix(*model*, *topic_names=None*, *class_ids=None*, *use_sparse_format=None*)

Parameters

- **model** (*str*) – name of matrix in BigARTM
- **topic_names** (*list of str or None*) – list of topics to retrieve (None means all topics)
- **class_ids** (*list of str or None*) – list of class ids to retrieve (None means all class ids)
- **use_sparse_format** (*bool*) – use sparsedense layout

Returns numpy.ndarray with Phi data (i.e., p(w|t) values)

get_score(*score_name*)

Parameters

- **score_name** (*str*) – the user defined name of score to retrieve
- **score_config** – reference to score data object

get_score_array(*score_name*)

Parameters

- **score_name** (*str*) – the user defined name of score to retrieve
- **score_config** – reference to score data object

get_theta_info()

Returns messages.ThetaMatrix object

get_theta_matrix(*topic_names=None*)

Parameters **topic_names** (*list of str or None*) – list of topics to retrieve (None means all topics)

Returns numpy.ndarray with Theta data (i.e., p(t|d) values)

import_dictionary(*filename*, *dictionary_name*)

Parameters

- **filename** (*str*) – full name of dictionary file
- **dictionary_name** (*str*) – name of imported dictionary

import_model(*model*, *filename*)

Parameters

- **model** (*str*) – name of matrix in BigARTM
- **filename** (*str*) – the name of file to load model from binary format

initialize_model(*model_name=None*, *topic_names=None*, *dictionary_name=None*, *seed=None*, *args=None*)

Parameters

- **model_name** (*str*) – name of pwt matrix in BigARTM
- **topic_names** (*list of str*) – the list of names of topics to be used in model
- **dictionary_name** (*str*) – name of imported dictionary
- **seed** (*unsigned int or -1, default None*) – seed for random initialization, None means no seed
- **args** – an instance of InitilaizeModelArgs

merge_model(*models*, *nwt*, *topic_names=None*, *dictionary_name=None*)

Merge multiple nwt-increments together.

Parameters

- **models** (*dict*) – list of models with nwt-increments and their weights, key - nwt_source_name, value - source_weight.
- **nwt** (*str*) – the name of target matrix to store combined nwt. The matrix will be created by this operation.
- **topic_names** (*list of str*) – names of topics in the resulting model. By default model names are taken from the first model in the list.
- **dictionary_name** – name of dictionary that defines which tokens to include in merged model

normalize_model(*pwt*, *nwt*, *rwt=None*)

Parameters

- **pwt** (*str*) – name of pwt matrix in BigARTM
- **nwt** (*str*) – name of nwt matrix in BigARTM
- **rwt** (*str*) – name of rwt matrix in BigARTM

process_batches(*pwt*, *nwt=None*, *num_document_passes=None*, *batches_folder=None*, *batches=None*, *regularizer_name=None*, *regularizer_tau=None*, *class_ids=None*, *class_weights=None*, *find_theta=False*, *reuse_theta=False*, *find_ptdw=False*, *predict_class_id=None*)

Parameters

- **pwt** (*str*) – name of pwt matrix in BigARTM

- **nwt** (*str*) – name of nwt matrix in BigARTM
- **num_document_passes** (*int*) – number of inner iterations during processing
- **batches_folder** (*str*) – full path to data folder (alternative 1)
- **batches** (*list of str*) – full file names of batches to process (alternative 2)
- **regularizer_name** (*list of str*) – list of names of Theta regularizers to use
- **regularizer_tau** (*list of float*) – list of tau coefficients for Theta regularizers
- **class_ids** (*list of str*) – list of class ids to use during processing
- **class_weights** (*list of float*) – list of corresponding weights of class ids
- **find_theta** (*bool*) – find theta matrix for 'batches' (if alternative 2)
- **reuse_theta** (*bool*) – initialize by theta from previous collection pass
- **find_ptdw** (*bool*) – calculate and return Ptdw matrix or not (works if find_theta == False)
- **predict_class_id** (*str, default None*) – class_id of a target modality to predict

Returns

- tuple (messages.ThetaMatrix, numpy.ndarray) — the info about Theta (if find_theta == True)
- messages.ThetaMatrix — the info about Theta (if find_theta == False)

reconfigure(*topic_names=None*, *class_ids=None*, *scores=None*, *regularizers=None*, *num_processors=None*, *pwt_name=None*, *nwt_name=None*, *num_document_passes=None*, *reuse_theta=None*, *cache_theta=None*)

reconfigure_regularizer(*name*, *config=None*, *tau=None*, *gamma=None*)

reconfigure_score(*name*, *config*, *model_name=None*)

reconfigure_topic_name(*topic_names=None*)

regularize_model(*pwt*, *nwt*, *rwt*, *regularizer_name*, *regularizer_tau*, *regularizer_gamma=None*)

Parameters

- **pwt** (*str*) – name of pwt matrix in BigARTM
- **nwt** (*str*) – name of nwt matrix in BigARTM
- **rwt** (*str*) – name of rwt matrix in BigARTM
- **regularizer_name** (*list of str*) – list of names of Phi regularizers to use
- **regularizer_tau** (*list of double*) – list of tau coefficients for Phi regularizers

transform(*batches=None*, *batch_filenames=None*, *theta_matrix_type=None*, *predict_class_id=None*)

Parameters

- **batches** – list of Batch instances
- **batch_weights** (*list of float*) – weights of batches to transform
- **theta_matrix_type** (*int*) – type of matrix to be returned

- **predict_class_id** (*int*) – type of matrix to be returned

Returns messages.ThetaMatrix object

5.2 C++ interface

BigARTM C++ interface is currently not documented. The main entry point is `MasterModel` class from `src/artm/cpp_interface.cc`. Please referto `src/bigartm//srcmain.cc` for usage examples, and ask questions at bigartm-users or open a new issue.

```
class MasterModel {
 public:
  explicit MasterModel(const MasterModelConfig& config);
  ~MasterModel();

  int id() const { return id_; }
  MasterComponentInfo info() const;  // misc. diagnostics information

  const MasterModelConfig& config() const { return config_; }
  MasterModelConfig* mutable_config() { return &config_; }
  void Reconfigure();  // apply MasterModel::config()

  // Operations to work with dictionary through disk
  void GatherDictionary(const GatherDictionaryArgs& args);
  void FilterDictionary(const FilterDictionaryArgs& args);
  void ImportDictionary(const ImportDictionaryArgs& args);
  void ExportDictionary(const ExportDictionaryArgs& args);
  void DisposeDictionary(const std::string& dictionary_name);

  // Operations to work with dictinoary through memory
  void CreateDictionary(const DictionaryData& args);
  DictionaryData GetDictionary(const GetDictionaryArgs& args);

  // Operatinos to work with batches through memory
  void ImportBatches(const ImportBatchesArgs& args);
  void DisposeBatch(const std::string& batch_name);

  // Operations to work with model
  void InitializeModel(const InitializeModelArgs& args);
  void ImportModel(const ImportModelArgs& args);
  void ExportModel(const ExportModelArgs& args);
  void FitOnlineModel(const FitOnlineMasterModelArgs& args);
  void FitOfflineModel(const FitOfflineMasterModelArgs& args);

  // Apply model to batches
  ThetaMatrix Transform(const TransformMasterModelArgs& args);
  ThetaMatrix Transform(const TransformMasterModelArgs& args, Matrix* matrix);

  // Retrieve operations
  TopicModel GetTopicModel(const GetTopicModelArgs& args);
  TopicModel GetTopicModel(const GetTopicModelArgs& args, Matrix* matrix);
  ThetaMatrix GetThetaMatrix(const GetThetaMatrixArgs& args);
  ThetaMatrix GetThetaMatrix(const GetThetaMatrixArgs& args, Matrix* matrix);

  // Retrieve scores
  ScoreData GetScore(const GetScoreValueArgs& args);
```

```
template <typename T>
T GetScoreAs(const GetScoreValueArgs& args);
```

Warning: What follows below in this page is really outdated.

5.2.1 MasterComponent

class `MasterComponent`

`MasterComponent`(**const** MasterComponentConfig &*config*)
Creates a master component with configuration defined by *MasterComponentConfig* message.

void **`Reconfigure`**(**const** MasterComponentConfig &*config*)
Updates the configuration of the master component.

const MasterComponentConfig &**`config`**() **const**
Returns current configuration of the master component.

MasterComponentConfig ***`mutable_config`**()
Returns mutable configuration of the master component. Remember to call *`Reconfigure()`* to propagate your changes to master component.

void **`InvokeIteration`**(int *iterations_count* = 1)
Invokes certain number of iterations.

bool **`AddBatch`**(**const** Batch &*batch*, bool *reset_scores*)
Adds batch to the processing queue.

bool **`WaitIdle`**(int *timeout* = -1)
Waits for iterations to be completed. Returns true if BigARTM completed before the specific timeout, otherwise false.

std::shared_ptr<TopicModel> **`GetTopicModel`**(**const** std::string &*model_name*)
Retrieves Phi matrix of a specific topic model. The resulting message *TopicModel* will contain information about token weights distribution across topics.

std::shared_ptr<TopicModel> **`GetTopicModel`**(**const** GetTopicModelArgs &*args*)
Retrieves Phi matrix based on extended parameters, specified in *GetTopicModelArgs* message. The resulting message *TopicModel* will contain information about token weights distribution across topics.

std::shared_ptr<ThetaMatrix> **`GetThetaMatrix`**(**const** std::string &*model_name*)
Retrieves Theta matrix of a specific topic model. The resulting message *ThetaMatrix* will contain information about items distribution across topics. Remember to set *`MasterComponentConfig.cache_theta`* prior to the last iteration in order to gather Theta matrix.

std::shared_ptr<ThetaMatrix> **`GetThetaMatrix`**(**const** GetThetaMatrixArgs &*args*)
Retrieves Theta matrix based on extended parameters, specified in *GetThetaMatrixArgs* message. The resulting message *ThetaMatrix* will contain information about items distribution across topics.

std::shared_ptr<T> **`GetScoreAs`**<T>(**const** *Model* &*model*, **const** std::string &*score_name*)
Retrieves given score for a specific model. Template argument must match the specific *ScoreData* type of the score (for example, *PerplexityScore*).

5.2.2 Model

class Model

Model(**const** *MasterComponent* &*master_component*, **const** ModelConfig &*config*)
Creates a topic model defined by *ModelConfig* inside given `MasterComponent`.

void **Reconfigure**(**const** ModelConfig &*config*)
Updates the configuration of the model.

const std::string &**name**() **const**
Returns the name of the model.

const ModelConfig &**config**() **const**
Returns current configuration of the model.

ModelConfig ***mutable_config**()
Returns mutable configuration of the model. Remember to call `Reconfigure()` to propagate your changes to the model.

void **Overwrite**(**const** TopicModel &*topic_model*, bool *commit* = true)
Updates the model with new Phi matrix, defined by *topic_model*. This operation can be used to provide an explicit initial approximation of the topic model, or to adjust the model in between iterations.

Depending on the *commit* flag the change can be applied immediately (*commit* = *true*) or queued (*commit* = *false*). The default setting is to use *commit* = *true*. You may want to use *commit* = *false* if your model is too big to be updated in a single protobuf message. In this case you should split your model into parts, each part containing subset of all tokens, and then submit each part in separate Overwrite operation with *commit* = *false*. After that remember to call `MasterComponent::WaitIdle()` and `Synchronize()` to propagate your change.

void **Initialize**(**const** *Dictionary* &*dictionary*)
Initialize topic model based on the `Dictionary`. Each token from the dictionary will be included in the model with randomly generated weight.

void **Export**(**const** string &*file_name*)
Exports topic model into a file.

void **Import**(**const** string &*file_name*)
Imports topic model from a file.

void **Synchronize**(double *decay_weight*, double *apply_weight*, bool *invoke_regularizers*)
Synchronize the model.

This operation updates the Phi matrix of the topic model with all model increments, collected since the last call to `Synchronize()` method. The weights in the Phi matrix are set according to *decay_weight* and *apply_weight* values (refer to `SynchronizeModelArgs.decay_weight` for more details). Depending on *invoke_regularizers* parameter this operation may also invoke all regularizers.

Remember to call `Model::Synchronize()` operation every time after calling `MasterComponent::WaitIdle()`.

void **Synchronize**(**const** SynchronizeModelArgs &*args*)
Synchronize the model based on extended arguments *SynchronizeModelArgs*.

5.2.3 Regularizer

class Regularizer

Regularizer(**const** *MasterComponent* &*master_component*, **const** RegularizerConfig &*config*)
Creates a regularizer defined by *RegularizerConfig* inside given `MasterComponent`.

void **Reconfigure**(**const** RegularizerConfig &*config*)
Updates the configuration of the regularizer.

const RegularizerConfig &**config**() **const**
Returns current configuration of the regularizer.

RegularizerConfig ***mutable_config**()
Returns mutable configuration of the regularizer. Remember to call `Reconfigure()` to propagate your changes to the regularizer.

5.2.4 Dictionary

class Dictionary

Dictionary(**const** *MasterComponent* &*master_component*, **const** DictionaryConfig &*config*)
Creates a dictionary defined by *DictionaryConfig* inside given `MasterComponent`.

void **Reconfigure**(**const** DictionaryConfig &*config*)
Updates the configuration of the dictionary.

const std::string **name**() **const**
Returns the name of the dictionary.

const DictionaryConfig &**config**() **const**
Returns current configuration of the dictionary.

5.2.5 Utility methods

void **SaveBatch**(**const** Batch &*batch*, **const** std::string &*disk_path*)
Saves *Batch* into a specific folder. The name of the resulting file will be autogenerated, and the extention set to *.batch*

std::shared_ptr<DictionaryConfig> **LoadDictionary**(**const** std::string &*filename*)
Loads the *DictionaryConfig* message from a specific file on disk. *filename* must represent full disk path to the dictionary file.

std::shared_ptr<Batch> **LoadBatch**(**const** std::string &*filename*)
Loads the *Batch* message from a specific file on disk. *filename* must represent full disk path to the batch file, including *.batch* extention.

std::shared_ptr<DictionaryConfig> **ParseCollection**(**const** CollectionParserConfig &*config*)
Parses a text collection as defined by *CollectionParserConfig* message. Returns an instance of *DictionaryConfig* which carry all unique words in the collection and their frequencies.

5.3 C Interface

This document explains all public methods of the low level BigARTM interface, written in plain C language.

5.3.1 Introduction

The goal of low level API is to expose all functionality of the library in a set of simple functions written in plain C language. This makes it easier to consume BigARTM from various programming environments. For example, the Python Interface of BigARTM uses ctypes module to call the low level BigARTM interface. Most programming environments also have similar functionality: PInvoke in C#, loadlibrary in Matlab, etc.

Typical methods of low-level API may look as follows:

```
int ArtmCreateMasterModel(int length, const char* master_model_config);
int ArtmFitOfflineMasterModel(int master_id, int length, const char* fit_offline_master_model_args);
int ArtmRequestTopicModel(int master_id, int length, const char* get_model_args);
```

This methods, similarly to most other methods in low level API, accept a serialized binary representation of some Google Protocol Buffer message. From BigARTM v0.8.2 it is also possible to pass JSON-serialized protobuf message. This might be useful if you are planing to use low-level C interface from environment where configuring protobuf libraries would be challenging. Please, refer to Messages for more details about each particular message, and Protobuf documentation regarding JSON mapping.

Note that this documentation is incomplete. For the actual list the methods of low-level C API please refer to c_interface.h. Same is true about messages documentation. It is always recommended to review the messages.proto definition.

If you plan to implement a high-wrapper around low-level API we recoomend to review the source code of existing wrappers cpp_interface.h, cpp_interface.cc (for C++ wrapper) and spec.py, api.py (for python wrapper).

5.3.2 List of all methods with corresponding protobuf types

```
                            ArtmConfigureLogging           (           artm.ConfigureLoggingArgs);
const char*                 = ArtmGetVersion();
const char*                 = ArtmGetLastErrorMessage();

artm.CollectionParserInfo = ArtmParseCollection            (           artm.CollectionParserConfig);

master_id                   = ArtmCreateMasterModel        (           artm.MasterModelConfig);
                              ArtmReconfigureMasterModel   (master_id, artm.MasterModelConfig);
                              ArtmReconfigureTopicName     (master_id, artm.MasterModelConfig);
                              ArtmDisposeMasterComponent   (master_id);

                              ArtmImportBatches            (master_id, artm.ImportBatchesArgs);

                              ArtmGatherDictionary         (master_id, artm.GatherDictionaryArgs);
                              ArtmFilterDictionary         (master_id, artm.FilterDictionaryArgs);
                              ArtmCreateDictionary         (master_id, artm.DictionaryData);
                              ArtmImportDictionary         (master_id, artm.ImportDictionaryArgs);
                              ArtmExportDictionary         (master_id, artm.ExportDictionaryArgs);
artm.DictionaryData         = ArtmRequestDictionary        (master_id, artm.GetDictionaryArgs);

                              ArtmInitializeModel          (master_id, artm.InitializeModelArgs);
                              ArtmExportModel              (master_id, artm.ExportModel);
                              ArtmImportModel              (master_id, artm.ImportModel);
                              ArtmOverwriteTopicModel      (master_id, artm.TopicModel);

                              ArtmFitOfflineMasterModel    (master_id, artm.FitOfflineMasterModelArgs
                              ArtmFitOnlineMasterModel     (master_id, attm.FitOnlineMasterModelArgs)
artm.ThetaMatrix            = ArtmRequestTransformMasterModel(master_id, artm.TransformMasterModelArgs)
```

```
artm.ThetaMatrix          = ArtmRequestTransformMasterModelExternal(master_id, artm.TransformMasterMo

artm.MasterModelConfig    = ArtmRequestMasterModelConfig   (master_id);
artm.ThetaMatrix          = ArtmRequestThetaMatrix         (master_id, artm.GetThetaMatrix);
artm.ThetaMatrix          = ArtmRequestThetaMatrixExternal (master_id, artm.GetThetaMatrix);
artm.TopicModel           = ArtmRequestTopicModel          (master_id, artm.GetTopicModel);
artm.TopicModel           = ArtmRequestTopicModelExternal  (master_id, artm.GetTopicModel);
artm.ScoreData            = ArtmRequestScore               (master_id, artm.GetScoreValueArgs);
artm.ScoreArray           = ArtmRequestScoreArray          (master_id, artm.GetScoreArrayArgs);
artm.MasterComponentInfo  = ArtmRequestMasterComponentInfo (master_id, artm.GetMasterComponentInfoArg

                            ArtmDisposeModel               (master_id, const char* model_name);
                            ArtmDisposeDictionary          (master_id, const char* dictionary_name);
                            ArtmDisposeBatch               (master_id, const char* batch_name);
                            ArtmClearThetaCache            (master_id, artm.ClearThetaCacheArgs);
                            ArtmClearScoreCache            (master_id, artm.ClearScoreCacheArgs);
                            ArtmClearScoreArrayCache       (master_id, artm.ClearScoreArrayCacheArgs)

                            ArtmCopyRequestedMessage        (int length, char* address);
                            ArtmCopyRequestedObject         (int length, char* address);

                            ArtmSetProtobufMessageFormatToJson();
                            ArtmSetProtobufMessageFormatToBinary();
int                       = ArtmProtobufMessageFormatIsJson();
```

Below we give a short description of these methods.

- `ArtmConfigureLogging` allows to configure logging parameters; this method is optional, you may not use it
- `ArtmGetVersion` returns the version of BigARTM library
- `ArtmParseCollection` parse collection in VW or UCI-BOW formats, creates batches and stores them to disk
- `ArtmCreateMasterModel` / `ArtmReconfigureMasterModel` / `ArtmDisposeMasterComponent` create master model / updates its parameters / dispose given instance of master model.
- `ArtmImportBatches` loads batches from disk into memory for quicker processing. This is optional, most methods that require batches can work directly with files on disk.
- `ArtmGatherDictionary` / `ArtmFilterDictionary` / `ArtmImportDictionary` / `ArtmExportDictionary` Main methods to work with dictionaries. *Gather* initialized the dictionary based on a folder with batches, *Filter* eliminates tokens based on their frequency, *Import*/*Export* save and re-load dictionary to/from disk.
- You may also created the dictionary from `artm.DictionaryData` message, that contains the list of all tokens to be included in the dictionary. To do this use method `ArtmCreateDictionary` (to create a dictionary) and `ArtmRequestDictionary` (to retrieve `artm.DictionaryData` for an existing dictionary).
- `ArtmInitializeModel` / `ArtmExportModel` / `ArtmImportModel` handle *models* (e.g. matrices of size `|T|*|W|` such as *pwt*, *nwt* or *rwt*). *Initialize** fills the matrix with random `0..1` values. *Export* and *Import* saves the matrix to disk and re-loads it back.
- `ArtmOverwriteTopicModel` allows to overwite values in topic model (for example to manually specify initial approximation).
- `ArtmFitOfflineMasterModel` — fit the model with *offline* algorithm
- `ArtmFitOnlineMasterModel` — fit the model with *online* algorithm

- `ArtmRequestTransformMasterModel` — apply the model to new data
- `ArtmRequestMasterModelConfig` — retrieve configuration of master model
- `ArtmRequestThetaMatrix` — retrieve cached theta matrix
- `ArtmRequestTopicModel` — retrieve a model (e.g. pwt, nwt or rwt matrix)
- `ArtmRequestScore` – retrieve score (such as perplexity, sparsity, etc)
- `ArtmRequestScoreArray` – retrieve historical information for a given score
- `ArtmRequestMasterComponentInfo` – retrieve diagnostics information and internal state of the master model
- `ArtmDisposeModel` / `ArtmDisposeDictionary` / `ArtmDisposeBatch` – dispose specific objects
- `ArtmClearThetaCache` / `ArtmClearScoreCache` / `ArtmClearScoreArrayCache` – clear specific caches
- `ArtmSetProtobufMessageFormatToJson` / `ArtmSetProtobufMessageFormatToBinary` / `ArtmProtobufMessageFormatIsJson` – configure the low-level API to work with JSON-serialized protobuf messages instead of binary-serialized protobuf messages

The following operations are less important part of low-level BigARTM CLI. In most cases you won'e need them, unless you have a very specific needs.

```
master_id                 = ArtmDuplicateMasterComponent     (master_id, artm.DuplicateMasterCompone
                            ArtmCreateRegularizer            (master_id, artm.RegularizerConfig);
                            ArtmReconfigureRegularizer       (master_id, artm.RegularizerConfig);
                            ArtmDisposeRegularizer           (master_id, const char* regularizer_name
                            ArtmOverwriteTopicModelNamed     (master_id, artm.TopicModel, const char*
                            ArtmCreateDictionaryNamed        (master_id, artm.DictionaryData, const c
                            ArtmAttachModel                  (master_id, artm.AttachModelArgs, int ac
artm.ProcessBatchesResult = ArtmRequestProcessBatches        (master_id, artm.ProcessBatchesArgs);
artm.ProcessBatchesResult = ArtmRequestProcessBatchesExternal(master_id, artm.ProcessBatchesArgs);
                            ArtmAsyncProcessBatches          (master_id, artm.ProcessBatchesArgs);
                            ArtmMergeModel                   (master_id, artm.MergeModelArgs);
                            ArtmRegularizeModel              (master_id, artm.RegularizeModelArgs);
                            ArtmNormalizeModel               (master_id, artm.NormalizeModelArgs);
artm.Batch                = ArtmRequestLoadBatch             (           const char* filename);
                            ArtmAwaitOperation               (           int operation_id, artm.Await
                            ArtmSaveBatch                    (           const char* disk_path, artm
```

5.3.3 Protocol for retrieving results

The methods in low-level API can be split into two groups — those that *execute* certain action, and those that *request* certain data. For example `ArtmCreateMasterModel` and `ArtmFitOfflineMasterModel` just execute an action, while `ArtmRequestTopicModel` is a request for data. Naming convention is that such requests always start with `ArtmRequest` prefix.

1. To call execute-action method is fairly straighforward — first you create a protobuf message that describe the arguments of the operation. For example, `ArtmCreateMasterModel` expects `artm.MasterModelConfig` message, as defined in the documentation of `ArtmCreateMasterModel`. Then you serialize protobuf message, and pass it to the method along with the length of the serialized message. In some cases you also pass the *master_id*, returned by `ArtmCreateMasterModel`, as described in details futher below on this page. The execute-action method will typically return an error code, with zero value (or `ARTM_SUCCESS`) indicating successfull execution.

2. To call request-data method is more tricky. First you follow the same procedure as when calling an execute-action method, e.g. create and serialize protobuf message and pass it to your `ArtmRequestXxx` operation. For

example, ArtmRequestTopicModel expects artm.GetTopicModelArgs message. Then the method like ArtmRequestTopicModel will return the size (in bytes) of the memory buffer that needs to be allocated by caller. To fill this buffer with actual data you need to call method

```
int ArtmCopyRequestedMessage(int length, char* address)
```

where address give a pointer to the memory buffer, and length must give the length of the buffer (e.g. must match the value returned by ArtmRequestXxx call). After ArtmCopyRequestedMessage the buffer will contain protobuf-serialized message. To deserialize this message you need to know its protobuf type, which will be defined by the documentation of the ArtmRequestXxx method that you are calling. For ArtmRequestTopicModel it will be a artm.TopicModel message.

3. Note that few ArtmRequestXxx methods has a more complex protocol that require two subsequent calls — first, to ArtmCopyRequestedMessage, and then to ArtmCopyRequestedObject. If that's the case the name of the method will be ArtmRequestXxxExternal (for example ArtmRequestThetaMatrixExternal or ArtmRequestTopicModelExternal). Typically this is used to copy out large objects, such as theta or phi matrices, and store them directly as dense matrices, bypassing protobuf serialization. For more information see cpp_interface.cc.

A side-note on thread safety: in between calls to ArtmRequestXxx and ArtmCopyRequestedMessage the result is stored in a thread local storage. This allows you to call multiple ArtmRequestXxx methods from different threads.

5.3.4 Error handling

All methods in this API return an integer value. Negative return values represent an error code. See *error codes* for the list of all error codes. To get corresponding error message as string use *ArtmGetLastErrorMessage()*. Non-negative return values represent a success, and for some API methods might also incorporate some useful information. For example, ArtmCreateMasterModel() returns the ID of newly created master component, and ArtmRequestTopicModel() returns the length of the buffer that should be allocated before calling ArtmCopyRequestedMessage().

5.3.5 MasterId and MasterModel

The concept of *Master Model* is central in low-level API. Almost any interaction with the low-level API starts by calling method ArtmCreateMasterModel, which creates an instance of so-called *Master Model* (or *Master Component*), and returns its master_id – an integer identifier that refers to that instance. You need master_id in the remaining methods of the low-level API, such as ArtmFitOfflineMasterModel. master_id creates a context, or scope, that isolate different models from each other. An operation applied to a specific master_id will not affect other master components. Each master model occupy some memory — potentially a very large amount, depending on the number of topics and tokens in the model. Once you are done with a specific instance of master component you need to dispose its resources by calling ArtmDisposeMasterComponent(master_id). After that master_id is no longer valid, and it must not be used as argument to other methods.

You may use method ArtmRequestMasterComponentInfo to retrieve internal diagnostics information about master component. It will reveal its internal state and tell the config of the master component, the list of scores and regularizers, the list of phi matrices, the list of dictionaries, cache entries, and other informatino that will help to understand how master component is functioning.

Note there might be confusion between terms *MasterComponent* and *MasterModel*, throughout this page as well as in the actual naming of the methods. This is due to historical reasons, and for all practical purposes you may think that this terms refer to the same thing.

5.3.6 ArtmConfigureLogging

You may use `ArtmConfigureLogging` call to set logging parameters, such as verbosity level or directory to output logs. You are not require to call `ArtmConfigureLogging`, in which case logging is automaticaly initialized to INFO level, and logs are placed in the active working folder.

Note that you can set log directory just one time. Once it is set you can not change it afterwards. Method `ArtmConfigureLogging` will return error code `INVALID_OPERATION` if it detects an attempt to change logging folder after logging had been initialized. In order to set log directory the call to `ArtmConfigureLogging` must happen prior to calling any other methods in low-level C API. (with exception to `ArtmSetProtobufMessageFormatToJson`, `ArtmSetProtobufMessageFormatToBinary` and `ArtmProtobufMessageFormatIsJson`). This is because methods in `c_interface` may automatically initialize logging into current working directory, which later can not be changed.

Setting log directory require that target folder already exist on disk.

The following parameters can be customized with `ArtmConfigureLogging`.

```
message ConfigureLoggingArgs {
  // If specified, logfiles are written into this directory
  // instead of the default logging directory.
  optional string log_dir = 1;

  // Messages logged at a lower level than this
  // do not actually get logged anywhere
  optional int32 minloglevel = 2;

  // Log messages at a level >= this flag are automatically
  // sent to stderr in addition to log files.
  optional int32 stderrthreshold = 3;

  // Log messages go to stderr instead of logfiles
  optional bool logtostderr = 4;

  // color messages logged to stderr (if supported by terminal)
  optional bool colorlogtostderr = 5;

  // log messages go to stderr in addition to logfiles
  optional bool alsologtostderr = 6;

  // Buffer log messages for at most this many seconds
  optional int32 logbufsecs = 7;

  // Buffer log messages logged at this level or lower
  // (-1 means do not buffer; 0 means buffer INFO only; ...)
  optional int32 logbuflevel = 8;

  // approx. maximum log file size (in MB). A value of 0 will be silently overridden to 1.
  optional int32 max_log_size = 9;

  // Stop attempting to log to disk if the disk is full.
  optional bool stop_logging_if_full_disk = 10;
}
```

We recommend to set `logbuflevel = -1` to not buffer log messages. However by default BigARTM does not set this parameter, using the same default as provided by glog.

5.3.7 ArtmReconfigureTopicName

To explain `ArtmReconfigureTopicName` we need to first start with `ArtmReconfigureMasterModel`. `ArtmReconfigureMasterModel` allow user to rename topics, but the number of topics must stay the same, and the order and the content of all existing phi matrices remains unchanged. On contrary, `ArtmReconfigureTopicName` may change the number of topics by adding or removing topics, as well as re-order columns of existing phi matrices. In `ArtmReconfigureMasterModel` the list of topic names is treated as new identifiers that should be set for existing columns. In `ArtmReconfigureTopicName` the list of topic names is matched against previous topic names. New topic names are added to phi matrices, topic names removed from the list are excluded from phi matrices, and topic names present in both old and new lists are re-ordered accordingly to match new topic name list.

Examples for `ArtmReconfigureMasterModel`:

- `t1, t2, t3 -> t4, t5, t6` sets new topic names for existing columns in phi matrices.

Examples for `ArtmReconfigureTopicName`:

- `t1, t2, t3 -> t1, t2, t3, t4` adds a new column to phi matrices, initialized with zeros
- `t1, t2, t3 -> t1, t2` removes last column from phi matrices
- `t1, t2, t3 -> t2, t3` removes the first column from phi matrices
- `t1, t2, t3 -> t3, t2` removes the first column from phi matrices and swaps the remaining two columns
- `t1, t2, t3 -> t4, t5, t6` removes all columns from phi matrices and creates three new columns, initialized with zeros

Note that both `ArtmReconfigureTopicName` and `ArtmReconfigureMasterModel` only affect phi matrices where set of topic names match the configuration of the master model. User-created matrices with custom set of topic names, for example created via `ArtmMergeModel`, will stay unchanged.

If you change topic names you should also consider changing your configuration of scores and regularizers. Also take into account that `ArtmReconfigureTopicName` and `ArtmReconfigureMasterModel` do not update theta cache. It is a good idea to call `ArtmClearThetaCache` after changing topic names.

5.3.8 ArtmGetLastErrorMessage

const char* **ArtmGetLastErrorMessage**()
Retrieves the textual error message, occured during the last failing request.

5.3.9 Error codes

```
#define ARTM_SUCCESS 0
#define ARTM_STILL_WORKING -1
#define ARTM_INTERNAL_ERROR -2
#define ARTM_ARGUMENT_OUT_OF_RANGE -3
#define ARTM_INVALID_MASTER_ID -4
#define ARTM_CORRUPTED_MESSAGE -5
#define ARTM_INVALID_OPERATION -6
#define ARTM_DISK_READ_ERROR -7
#define ARTM_DISK_WRITE_ERROR -8
```

ARTM_SUCCESS
The API call succeeded.

ARTM_STILL_WORKING
This error code is applicable only to `ArtmAwaitOperation()`. It indicates that library is still processing the collection. Try to retrieve results later.

ARTM_INTERNAL_ERROR
The API call failed due to internal error in BigARTM library. Please, collect steps to reproduce this issue and report it with BigARTM issue tracker.

ARTM_ARGUMENT_OUT_OF_RANGE
The API call failed because one or more values of an argument are outside the allowable range of values as defined by the invoked method.

ARTM_INVALID_MASTER_ID
An API call that require *master_id* parameter failed because MasterComponent with given ID does not exist.

ARTM_CORRUPTED_MESSAGE
Unable to deserialize protocol buffer message.

ARTM_INVALID_OPERATION
The API call is invalid in current state or due to provided parameters.

ARTM_DISK_READ_ERROR
The required files coult not be read from disk.

ARTM_DISK_WRITE_ERROR
The required files could not be writtent to disk.

CHAPTER 6

VisARTM

6.1 Introduction

VisARTM is a web-based tool for operating with text collection and topics models.

6.2 Installation

You can deploy VisARTM locally on your computer. It supports any OS: Windows, Linux or MacOS. Please follow next steps.

1. Make sure you have installed python 3. We recommend use Anaconda.
2. Make sure you have installed BigARTM.
3. Install PostgreSQL and pgAdmin. Of cource, ou can use any database management system with django, but we recommend PostgreSQL.
4. Open pgAdmin and create new database. Please remember username and password to this database. Default username in PostgreSQL is "postgres".
5. Make sure you have installed git.
6. Clone VisARTM with console command

```
git clone https://github.com/bigartm/visartm.git
```

7. Install all required dependencies, including django:

```
cd visartm
pip install -r requirements.txt
```

8. Now link database created in step 4 with VisARTM. For that, open file **visartm/settings.py** and lines like these:

```
DATABASES = {
             'default': {
                     'ENGINE': 'django.db.backends.postgresql_psycopg2',
                     'NAME': 'artmonlinedb',
                     'USER': 'postgres',
                     'PASSWORD': '******',
                     'HOST': '127.0.0.1',
                     'PORT': '5432',
```

```
            }
    }
```

Modify values as follows. NAME is name of database. USER is name of user of database. PASSWORD is his password. If you use local PostgreSQL, you will not need change ENGINE, otherwise it's up to you. You are running database server locally, so don't modify HOST value.

9. Now you need django to create tables. For that, go to folder named **visartm** and run:

```
python manage.py makemigrations
python manage.py migrate
```

10. Create superuser for the service. Django will ask you for username and passwrod. Please remember them, you will need them to use service.

```
python manage.py createsuperuser
```

11. Run the server.

```
python manage.py runserver
```

12. Open your favorite web browser and navigate to http://127.0.0.1:8000

6.3 Datasets

6.3.1 Object model

Dataset is a collection of **documents**. It contains vocabulary (set of **terms**) and **documents**.

Documents technically are multisets or lists of terms. Also they can contain *raw text*.

Each term belongs to **modality**. If **modalities** are undefined, all terms belongs to **modality** @default_class.

ArtmModel is topic model.

6.3.2 Dataset creating

Go to folder **visartm/data/datasets**. There create a folder, named after your dataset. Put there single file named vw.txt, which describes your dataset in Vowpal Vabbit format. This file is necessary and enough to go on.

Then go to **Datasets**, click on **Create new dataset**, choose tab **Local**, select in combo-box **Folder** folder, that yo have created and click **Create**.

6.3.3 Features

You can not ony provide data in VW format. You have various options.

1. I have only raw text files and I don't want do any preprocessing.

Create folder named **documents** in dataset folder. Put all documents, encoded in UTF-8 there (you can create subdirectories inside). Then, just enable **Parse** option in **Prerocessing** section on the page of dataset creation. Documents will be automatically parsed and lemmatized and Vowpal Wabbit file will be created.

2. I have additional meta data.

Create folder **meta** in dataset folder. Put there file **meta.json**. This file should be JSON dictionary, keys of which are names of documents, stated in VW file. If you upload raw text, names of documents should coinside with file names (if you have folders, then relative pathes instead of file names).

The values of this JSON dictionary contain meta data for document. They are also JSON dictionaries with following keys (no one is obligatory).

- **title**
- **snippet**
- **url**
- **time** (must be UNIX timestamp)

3. I have raw text and some additional modalities (like tags or authors) in VW format.

Name your VW file with additional modalitites **meta.vw.txt** and put it in folder **meta**. If you enable **Parse** option then, automatically extracted words will be merged with this data.

4. I know what's your wordpos files and I want create those myself.

Just create folder **wordpos** next to documents, create there exact file structure as in documents, but instaed of text write positions of words. Of cource, that will mean that you also have **vw.txt** file, so don't enable Parse option and system will use your wordpos.

5. I have collection in UCI format.

Use this script to convert your collection into Vowpal Wabbit format, then upload single **vw.txt** file.

6.3.4 Preprocessing

VisARTM can do some preprocessing with your dataset. It can be enabled on dataset creation page.

- **Parse** - automatically parse and lemmatize raw text. Options:
  - **Store order** - if you enable this, each occurence of each word will be stored in VW file. So, information about order of terms in initial document will be stored. But everything will work slower. If you disable this, documents will be treated like bag of words. Disable it if unsure.
  - **Hashtags** - if you enable thiis, all terms, beginning with # will be treated as hashtags. They will not be lemmatized, and they will be stored as separate modality.
- **Filter** - remove some terms from vocabulary. Options:
  - **Lower bound** - all terms, which ocured in whole dataset less than lower_bound times, will be removed.
  - **Upper bound** - all terms, which ocured in whole dataset more than upper_bound times, will be removed.
  - **Lower bound** (relative) - all terms, which ocured in whole dataset more than upper_bound_relative * number_od_documents_in_collection times, will be removed.

6.4 Model creation

6.4.1 Automatic model creation

For quick start, use built-in model builder.

Go to **Datasets**, select dataset, click **Create new model**. Select tab **Flat** or **Hierarchial**, choose number of topcs and iterations and click **Create**.

6.4.2 Creating model with script

Go to **Datasets**, select dataset, click **Create new model**. Select tab **Empty** and click **Create**.

VisARTM will create a folder and shows you full path to that folder. There you will find sample script named **sample.py**. This script will tell you, how initialize ARTM model with dataset, and where to put result.

For a flat model, result is two matrices **theta** and **phi**. You should save them to folder of model, using **to_pickle()** methd of **pandas.DataFrame**.

When matrices are ready, just press **Reload** on page of model.

6.4.3 Hierarchical models

If you are building N-tier hierarchy using hARTM, it is necessary to save also files **psi1**, **psi2**, ..., **psi(N-1)**. Save them also using **to_pickle()**, next to **theta** and **phi**.

6.4.4 Multiple models

You can create another model. Just go to dataset page and click **Create new model**. You can delete model from page of models, or delete all models, using option **Clear** on dataset page. List of all models is available on the right of dataset page.

6.5 Visualizations

You can explore documents, topic, terms, modalities and model itself.

Besides, you can look at differnet visualizations of model in whole.

If you have several models, choose model to be visualized using radio-buttons at dataset page.

CHAPTER 7

Release Notes

7.1 Changes in Python API

This page describes recent changes in BigARTM's Python API. Note that the API might be affected by changes in the underlying protobuf messages. For this reason we recommend to review Changes in Protobuf Messages.

For further reference about Python API refer to ARTM model, Q & A or tutorials.

7.1.1 v0.8.3

- Enable copying of ARTM, LDA, hARTM and ARTM_Level objects with `clone()` method and `copy.deepcopy(obj)`.
- Experimental support for import/export/editing of theta matrices; for more details see python reference of `ARTM.__init__(ptd_name='ptd')`.

7.1.2 v0.8.2

Warning: BigARTM 3rdparty dependency had been upgraded from `protobuf 2.6.1` to `protobuf 3.0.0`. This may affect you upgrade from previous version of bigartm. Pelase report any issues at bigartm-users@googlegroups.com.

Warning: BigARTM now require you to install tqdm library to visualize progress bars. To install use `pip install tqdm` or `conda install -c conda-forge tqdm`.

- Add support for python 3.0
- Add hARTM class to support hierarchy model
- Add HierarchySparsingTheta for advanced inference of hierarchical models
- Enable replacing regularizers in ARTM-like models:

```
# using operator[]-like style
model.regularizers['somename'] = SomeRegularizer(...)
# using keyword argument overwrite in add function
model.regularizers.add(SomeRegularizer(name='somename', ...), overwrite=True)
```

- Better error reporting: raise exception in fit_offline, fit_online and transform if there is no data to process)

- Better support for changes in topic names, with reconfigure(), initialize() and merge_model()
- Show progress bars in `fit_offline`, `fit_online` and `transform`.
- Add `ARTM.reshape_topics` method to add/remove/reorder topics.
- Add `max_dictionary_size` parameter to `Dictionary.filter()`
- Add `class_ids` parameter to `BatchVectorizer.__init__()`
- Add `dictionary_name` parameter to `MasterComponent.merge_model()`
- Add `ARTM.transform_sparse()` and `ARTM.get_theta_sparse()` for sparse retrieval of theta matrix
- Add `ARTM.get_phi_sparse()` for sparse retrieval of phi matrix

7.1.3 v0.8.1

- New source type 'bow_n_wd' was added into BatchVectorizer class. This type oriented on using the output of CountVectorizer and TfIdfVectorizers classes from sklearn. New parameters of BatchVectorizer are: n_wd (numpy.array) and vocabulary(dict)
- LDA model was added as one of the public interfaces. It is a restricted ARTM model created to simplify BigARTM usage for new users with few experience in topic modeling.
- BatchVectorizer got a flag 'gather_dictionary', which has default value 'True'. This means that BV would create dictionary and save it in the BV.dictionary field. For 'bow_n_wd' format the dictionary will be gathered whenever the flag was set to 'False' or to 'True'.
- Add relative regularization for Phi matrix

7.1.4 v0.8.0

Warning: Note that your script can be affected by our changes in the default values for `num_document_passes` and `reuse_theta` parameters (see below). We recommend to use our new default settings, `num_document_passes = 10` and `reuse_theta = False`. However, if you choose to explicitly set `num_document_passes = 1` then make sure to also set `reuse_theta = True`, otherwise you will experience very slow convergence.

- all operations to work with dictionaries were moved into a separate class `artm.Dictionary`. (details in the documentation). The mapping between old and new methods is very straighforward: `ARTM.gather_dictionary` is replaced with `Dictionary.gather` method, which allows to gather a dictionary from a set of batches; `ARTM.filter_dictionary` is replaced with `Dictionary.filter` method, which allows to filter a dictionary based on term frequency and document frequency; `ARTM.load_dictionary` is replaced with `Dictionary.load` method, which allows to load a dictionary previously exported to disk in `Dictionary.save` method; `ARTM.create_dictionary` is replaced with `Dictionary.create` method, which allows to create a dictionary based on custom protobuf message `DictionaryData`, containing a set of dictionary entries; etc... The following code snippet gives a basic example:

```
my_dictionary = artm.Dictionary()
my_dictionary.gather(data_path='my_collection_batches', vocab_file_path='vocab.txt')
my_dictionary.save(dictionary_path='my_collection_batches/my_dictionary')
my_dictionary.load(dictionary_path='my_collection_batches/my_dictionary.dict')
model = artm.ARTM(num_topics=20, dictionary=my_dictionary)
model.scores.add(artm.PerplexityScore(name='my_fisrt_perplexity_score',
```

```
                                        use_unigram_document_model=False,
                                        dictionary=my_dictionary))
```

- added `library_version` property to `ARTM` class to query for the version of the underlying BigARTM library; returns a string in MAJOR.MINOR.PATCH format;
- `dictionary_name` argument had been renamed to `dictionary` in many places across python interface, including scores and regularizers. This is done because those arguments can now except not just a string, but also the `artm.Dictionary` class itself.
- with `Dictionary` class users no longer have to generate names for their dictionaries (e.g. the unique `dictionary_name` identifier that references the dictionary). You may use `Dictionary.name` field to access to the underlying name of the dictionary.
- added `dictionary` argument to `ARTM.__init__` constructor to let user initialize the model; note that we've change the behavior that model is automatically initialized whenever user calls `fit_offline` or `fit_online`. Now this is no longer the case, and we expect user to either pass a dictionary in `ARTM.__init__` constructor, or manually call `ARTM.initialize` method. If neither is performed then `ARTM.fit_offline` and `ARTM.fit_online` will throw an exception.
- added `seed` argument to `ARTM.__init__` constructor to let user randomly initialize the model;
- added new score and score tracker `BackgroundTokensRatio`
- remove the default value from `num_topics` argument in `ARTM.__init__` constructor, which previously was defaulting to `num_topics = 10`; now user must always specify the desired number of topics;
- moved argument `reuse_theta` from `fit_offline` method into `ARTM.__init__` constructor; the argument is still used to indicate that the previous theta matrix should be re-used on the next pass over the collection; setting `reuse_theta = True` in the constructor will now be applied to `fit_online`, which previously did not have this option.
- moved common argument `num_document_passes` from `ARTM.fit_offline`, `ARTM.fit_online`, `ARTM.transform` methods into `ARTM.__init__` constructor.
- changed the default value of `cache_theta` parameter from `True` to `False` (in `ARTM.__init__` constructor); this is done to avoid excessive memory usage due to caching of the entire Theta matrix; if caching is indeed required user has to manually turn it on by setting `cache_theta = True`.
- changed the default value of `reuse_theta` parameter from `True` to `False` (in `ARTM.__init__` constructor); the reason is the same as for changing the default for `cache_theta` parameter
- changed the default value of `num_document_passes` parameter from `1` to `10` (in `ARTM.__init__` constructor);
- added arguments `apply_weight`, `decay_weight` and `update_after` in `ARTM.fit_online` method; each argument accepts a list of floats; setting all three arguments will override the default behavior of the online algorithm that rely on a specific formula with `tau0`, `kappa` and `update_every`.
- added argument `async` (boolean flag) in `ARTM.fit_online` method for improved performance.
- added argument `theta_matrix_type` in `ARTM.transform` method; potential values are: `"dense_theta"`, `"dense_ptdw"`, `None`; default matrix type is `"dense_theta"`.
- introduced a separate method `ARTM.remove_theta` to clear cached theta matrix; remove corresponding boolean switch `remove_theta` from `ARTM.get_theta` method.
- removed `ARTM.fit_transform` method; note that the name was confusing because this method has never fitted the model; the purpose of `ARTM.fit_transform` was to retrieve Theta matrix after fitting the model (`ARTM.fit_offline` or `ARTM.fit_online`); same functionality is now available via `ARTM.get_theta` method.

- introduced `ARTM.get_score` method, which will exist in parallel to score tracking functionality; the goal for `ARTM.get_score(score_name)` is to always return the latest version of the score; for Phi scores this means to calculate them on fly; for Theta scores this means to return a score aggregated over last call to `ARTM.fit_offline`, `ARTM.fit_online` or `ARTM.transform` methods; opposite to `ARTM.get_score` the score tracking functionality returns the overall history of a score. For further details on score calculation refer to Q&A section in our wiki page.
- added `data_weight` in `BatchVectorizer.__init__` constructor to let user specify an individual weight for each batch
- score tracker classes had been rewritten, so you should make minor changes in the code that retrieves scores; for example:
- added an API to initialize logging with custom logging directory, log level, etc... Search out wiki page Q&A for more details.

```
# in v0.7.x
print model.score_tracker['Top100Tokens'].last_topic_info[topic_name].tokens

# in v0.8.0
last_tokens = model.score_tracker['Top100Tokens'].last_tokens
print last_tokens[topic_name]
```

7.1.5 v0.7.x

See BigARTM v0.7.X Release Notes.

7.2 Changes in Protobuf Messages

7.2.1 v0.8.2

- added `CollectionParserConfig.num_threads` to control the number of threads that perform parsing. At the moment the feature is only implemented for VW-format.
- added `CollectionParserConfig.class_id` (repeated string) to control which modalities should be parsed. If token's class_id is not from this list, it will be excluded from the resulting batches. When the list is empty, all modalities are included (this is the default behavior, as before).
- added `CollectionParserInfo` message to export diagnostics information from `ArtmParseCollection`
- added `FilterDictionaryArgs.max_dictionary_size` to give user an easy option to limit his dictionary size
- added `MergeModelArgs.dictionary_name` to define the set of tokens in the resulting matrix
- added `ThetaMatrix.num_values`, `TopicModel.num_values` to define number of non-zero elements in sparse format

7.2.2 v0.8.0

Warning: New batches, created in *BigARTM v0.8*, **CAN NOT** be used in the previous versions of the library. Old batches, created prior to *BigARTM v0.8*, can still be used. See below for details.

- added `token_id` and `token_weight` field in `Item` message, and obsoleted `Item.field`. Internally the library will merge the content of `Field.token_id` and `Field.token_weight` across all fields, and store the result back into `Item.token_id`, `Item.token_weight`. New `Item` message is as follows:

```
message Item {
  optional int32 id = 1;
  repeated Field field = 2;  // obsolete in BigARTM v0.8.0
  optional string title = 3;
  repeated int32 token_id = 4;
  repeated float token_weight = 5;
}
```

- renamed `topics_count` into `num_topics` across multiple messsages (`TopicModel`, `ThetaMatrix`, etc)
- renamed `inner_iterations_count` into `num_document_passes` in `ProcessBatchesArgs`
- renamed `passes` into `num_collection_passes` in `FitOfflineMasterModelArgs`
- renamed `threads` into `num_processors` in `MasterModelConfig`
- renamed `topic_index` field into `topic_indices` in `TopicModel` and `ThetaMatrix` messages
- added messages `ScoreArray`, `GetScoreArrayArgs` and `ClearScoreArrayCacheArgs` to bring score tracking functionality down into BigARTM core
- added messages `BackgroundTokensRatioConfig` and `BackgroundTokensRatio` (new score)
- moved `model_name` from `GetScoreValueArgs` into `ScoreConfig`; this is done to support score tracking functionality in BigARTM core; each Phi score needs to know which model to use in calculation
- removed `topics_count` from `InitializeModelArgs`; users must specify topic names in `InitializeModelArgs.topic_name` field
- removed `topic_index` from `GetThetaMatrixArgs`; users must specify topic names to retrieve in `GetThetaMatrixArgs.topic_name`
- removed `batch` field in `GetThetaMatrixArgs` and `GetScoreValueArgs.batch` messages; users should use `ArtmRequestTransformMasterModel` or `ArtmRequestProcessBatches` to process new batches and calculate theta scores
- removed `reset_scores` flag in `ProcessBatchesArgs`; users should use new API `ArtmClearScoreCache`
- removed `clean_cache` flag in `GetThetaMatrixArgs`; users should use new API `ArtmClearThetaCache`
- removed `MasterComponentConfig`; users should user `ArtmCreateMasterModel` and pass `MasterModelConfig`
- removed obsolete fields in `CollectionParserConfig`; same arguments can be specified at `GatherDictionaryArgs` and passed to `ArtmGatherDictionary`
- removed `Filter` message in `InitializeModelArgs`; same arguments can be specified at `FilterDictionaryArgs` and passed to `ArtmFilterDictionary`
- removed `batch_name` from `ImportBatchesArgs`; the field is no longer needed; batches will be identified via their `Batch.id` identifier
- removed `use_v06_api` in `MasterModelConfig`
- removed `ModelConfig` message
- removed `SynchronizeModelArgs`, `AddBatchArgs`, `InvokeIterationArgs`, `WaitIdleArgs` messages; users should use new APIs based on MasterModel

- removed `GetRegularizerStateArgs`, `RegularizerInternalState`, `MultiLanguagePhiInternalState` messages
- removed `model_name` and `model_name_cache` in `ThetaMatrix`, `GetThetaMatrixArgs` and `ProcessBatchesArgs`; the code of master component is simplified to only handle one theta matrix, so there is no longer any reason to identify theta matrix with `model_name`
- removed `Stream` message, `MasterComponentConfig.stream` field, and all `stream_name` fields across several messages; train/test streaming functionality is fully removed; users are expected to manage their train and test collections (for example as separate folders with batches)
- removed `use_sparse_bow` field in several messages; the computation mode with dense matrices is no longer supported;
- renamed `item_count` into `num_items` in `ThetaSnippetScoreConfig`
- add global enum `ScoreType` as a replacement for enums `Type` from `ScoreConfig` and `ScoreData` messages
- add global enum `RegularizerType` as a replacement for enum `Type` from `RegularizerConfig` message
- add global enum `MatrixLayout` as a replacement for enum `MatrixLayout` from `GetThetaMatrixArgs` and `GetTopicModelArgs` messages
- add global enum `ThetaMatrixType` as a replacement for enum `ThetaMatrixType` from `ProcessBatchesArgs` and `TransformMasterModelArgs` messages
- renamed enum `Type` into `SmoothType` in `SmoothPtdwConfig` to avoid conflicts in C# messages
- renamed enum `Mode` into `SparseMode` in `SpecifiedSparsePhiConfig` to avoid conflicts in C# messages
- renamed enum `Format` into `CollectionFormat` in `CollectionParserConfig` to avoid conflicts in C# messages
- renamed enum `NameType` into `BatchNameType` in `CollectionParserConfig` to avoid conflicts in C# messages
- renamed field `transform_type` into `type` in `TransformConfig` to avoid conflicts in C# messages
- remove message `CopyRequestResultArgs`; this is a breaking change; please check that
  - all previous calls to `ArtmCopyRequestResult` are changed to to `ArtmCopyRequestedMessage`
  - all previous calls to `ArtmCopyRequestResultEx` with request types `GetThetaSecondPass` and `GetModelSecondPass` are changed to `ArtmCopyRequestedObject`
  - all previous calls to `ArtmCopyRequestResultEx` with `DefaultRequestType` are changed to `ArtmCopyRequestedMessage`
- remove field `request_type` in `GetTopicModelArgs`; to request only topics and/or tokens users should set `GetTopicModelArgs.matrix_layout` to `MatrixLayout_Sparse`, and `GetTopicModelArgs.eps = 1.001` (any number greather that 1.0).
- change `optional FloatArray` into `repeated float` in field `coherence` of `TopTokensScore`
- change `optional DoubleArray` into `repeated double` in fields `kernel_size`, `kernel_purity`, `kernel_contrast` and `coherence` of `TopicKernelScore`
- change `optional StringArray` into `repeated string` in field `topic_name` of `TopicKernelScore`

7.2.3 v0.7.x

See BigARTM v0.7.X Release Notes.

7.3 Changes in BigARTM CLI

7.3.1 v0.8.2

- added option `--rand-seed` to initialize random number generator; without this options, RNG will be set using system time
- added option `--write-vw-corpus` to convert batches into plain text file in Vowpal Wabbit format
- change the naming scheme of the batches, saved with `--save-batches` option. Previously file names were guid-based, while new format will look like this: `aabcde.batch`. New format ensures the ordering of the documents in the collection is be preserved, given that user scans batches alphabetically.
- added switch `--guid-batch-name` to enable old naming scheme of batches (guid-based names). This option is useful if you launch multiple instances of BigARTM CLI to concurrently generate batches.
- speedup parsing large files in VowpalWabbit format
- when `--use-modality` is specified, the batches saved with `--save-batches` will only include tokens from these modalities. Other tokens will be ignored during parsing. This option is implemented for both VW and UCI BOW formats.
- implement `TopicSelection`, `LabelRegularization`, `ImproveCoherence`, `Biterms` regularizer in BigARTM CLI
- added option `--dictionary-size` to give user an easy option to limit his dictionary size
- add more diagnostics information about dictionary size (before and after filtering)
- add strict verification of scores and regularizers; for example, BigARTM CLI will raise an exception for this input: `bigartm -t obj:10,back:5 --regularizer "0.5 SparsePhi #obj*"`. There shouldn't be star sign in `#obj*`.

7.3.2 v0.8.0

- renamed `--passes` into `--num-collection-passes`
- renamed `--num-inner-iterations` into `--num-document-passes`
- removed `--model-v06` option
- removed `--use-dense-bow` option

7.3.3 v0.7.x

See BigARTM v0.7.X Release Notes.

7.4 Changes in c_interface

7.4.1 v0.8.2

Warning: BigARTM 3rdparty dependency had been upgraded from `protobuf 2.6.1` to `protobuf 3.0.0`. This may affect you upgrade from previous version of bigartm. Pelase report any issues at bigartm-users@googlegroups.com.

- Change `ArtmParseCollection` to return `CollectionParserInfo` message
- Add APIs to enable JSON serialization for all input and output protobuf messages
  - `ArtmSetProtobufMessageFormatToJson()`
  - `ArtmSetProtobufMessageFormatToBinary()`
  - `ArtmProtobufMessageFormatIsJson()`

  The default setting is, as before, to serialize all message into binary buffer. Note that for with json serialization one should use `RegularizerConfig.config_json`, `ScoreConfig.config_json` and `ScoreData.data_json` instead of `RegularizerConfig.config`, `ScoreConfig.config` and `ScoreData.data`.
- Revisit documentation for c_interface
- Change integer types in c_interface from `int` to `int64_t` (from `stdint.h`). This allows to validate 2 GB limit for protobuf messages, and also to passing larger objects in `ArtmCopyRequestedObject`.
- Add `ArtmReconfigureTopicName` method to add/remove/reorder topic names
- Support sparse format for external retrieval of theta and phi matrices

7.4.2 v0.8.0

- Removed `ArtmCreateMasterComponent` and `ArtmReconfigureMasterComponent`
- Removed `ArtmCreateModel` and `ArtmReconfigureModel`
- Removed `ArtmAddBatch`, `ArtmInvokeIteration`, `ArtmWaitIdle`, `ArtmSynchronizeModel`
- Removed `ArtmRequestRegularizerState`
- Renamed `ArtmCopyRequestResult` into `ArtmCopyRequestedMessage`
- Renamed `ArtmCopyRequestResultEx` into `ArtmCopyRequestedObject`
- Added `ArtmClearThetaCache` and `ArtmClearScoreCache`
- Added `ArtmRequestScoreArray` and `ArtmClearScoreArrayCache`
- Added `GetArtmVersion` to query for the version; returns a string in "<MAJOR>.<MINOR>.<PATCH>" format

7.4.3 v0.7.x

See BigARTM v0.7.X Release Notes.

7.5 BigARTM v0.7.X Release Notes

7.5.1 BigARTM v0.7.0 Release notes

We are happy to introduce BigARTM v0.7.0, which brings you the following changes:

- New-style models
- Network modus operandi is removed
- Coherence regularizer and scores (experimental)

New-style models

BigARTM v0.7.0 exposes new APIs to give you additional control over topic model inference:

- ProcessBatches
- MergeModel
- RegularizeModel
- NormalizeModel

Besides being more flexible, new APIs bring many additional benefits:

- Fully deterministic inference, no dependency on threads scheduling or random numbers generation
- Less bottlenecks for performance (DataLoader and Merger threads are removed)
- Phi-matrix regularizers can be implemented externally
- Capability to output Phi matrices directly into your NumPy matrices (scheduled for BigARTM v0.7.2)
- Capability for store Phi matrices in sparse format (scheduled for BigARTM v0.7.3)
- Capability for async ProcessBatches and non-blocking online algorithm (BigARTM v0.7.4)
- Form solid foundation for high performance networking (BigARTM v0.8.X)

The picture below illustrates scalability of BigARTM v0.7.0 vs v0.6.4. Top chart (in green) corresponds to CPU usage at 28 cores on machine with 32 virtual cores (16 physical cores + hyper threading). As you see, new version is much more stable. In addition, new version consumes less memory.

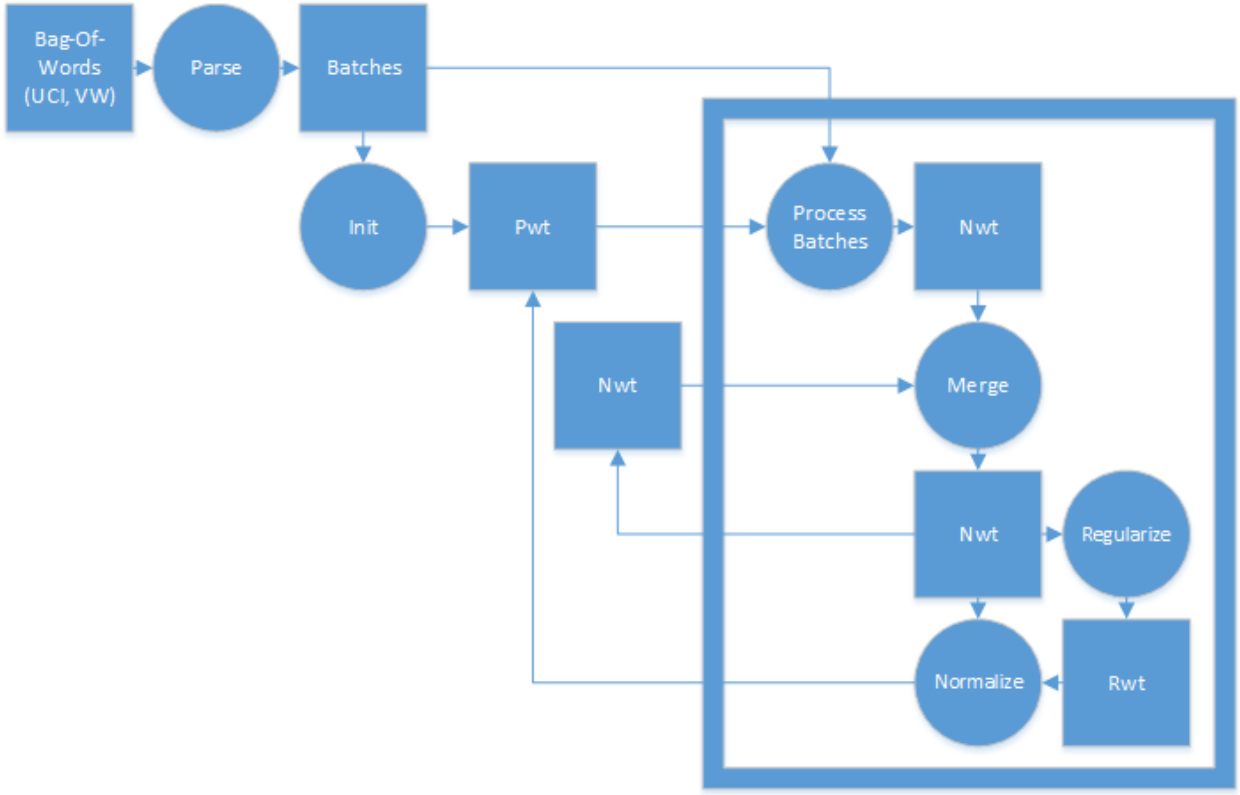
Refer to the following examples that demonstrate usage of new APIs for offline, online and regularized topic modelling:

- example17_process_batches.py
- example18_merge_model.py
- example19_regularize_model.py

Models, tuned with the new API are referred to as *new-style models*, as opposite to *old-style models* inferred with AddBatch, InvokeIteration, WaitIdle and SynchronizeModel APIs.

Warning: For BigARTM v0.7.X we will continue to support old-style models. However, you should consider upgrading to new-style models because old APIs (AddBatch, InvokeIteration, WaitIdle and SynchronizeModel) are likely to be removed in future releases.

The following flow chart gives a typical use-case on new APIs in online regularized algorithm:



Notes on upgrading existing code to new-style models

1. New APIs can only read batches from disk. If your current script passes batches via memory (in AddBatchArgs.batch field) then you need to store batches on disk first, and then process them with ProcessBatches method.
2. Initialize your model as follows:
   - For python_interface: using MasterComponent.InitializeModel method
   - For cpp_interface: using MasterComponent.InitializeModel method
   - For c_interface: using ArtmInitializeModel method

   Remember that you should not create ModelConfig in order to use this methods. Pass your topics_count (or topic_name list) as arguments to InitializeModel method.
3. Learn the difference between Phi and Theta scores, as well as between Phi and Theta regularizes. The following table gives an overview:

Object	Theta	Phi
Scores	• Perplexity • SparsityTheta • ThetaSnippet • ItemsProcessed	• SparsityPhi • TopTokens • TopicKernel
Regularizers	• SmoothSparseTheta	• DecorrelatorPhi • ImproveCoherencePhi • LabelRegularizationPhi • SmoothSparsePhi • SpecifiedSparsePhi

Phi regularizers needs to be calculated explicitly in RegularizeModel, and then applied in NormalizeModel (via optional *rwt* argument). Theta regularizers needs to be enabled in ProcessBatchesArgs. Then they will be automatically calculated and applied during ProcessBatches.

Phi scores can be calculated at any moment based on the new-style model (same as for old-style models). Theta scores can be retrieved in two equivalend ways:

```
pwt_model = "pwt"
master.ProcessBatches(pwt_model, batches, "nwt")
perplexity_score.GetValue(pwt_model).value
```

or

```
pwt_model = "pwt"
process_batches_result = master.ProcessBatches(pwt_model, batches, "nwt")
perplexity_score.GetValue(scores = process_batches_result).value
```

Second way is more explicit. However, the first way allows you to combine aggregate scores accross multiple ProcessBatches calls:

```
pwt_model = "pwt"
master.ProcessBatches(pwt_model, batches1, "nwt")
master.ProcessBatches(pwt_model, batches2, "nwt", reset_scores=False)
perplexity_score.GetValue(pwt_model).value
```

This works because BigARTM caches the result of ProcessBatches together (in association with pwt_model). The *reset_scores* switch disables the default behaviour, which is to reset the cache for pwt_model at the beginning of each ProcessBatch call.

4. Continue using GetThetaMatrix and GetTopicModel to retrieve results from the library. For GetThetaMatrix to work you still need to enable cache_theta in master component. Remember to use the same model in GetThetaMatrix as you used as the input to ProcessBatches. You may also omit "target_nwt" argument in ProcessBatches if you are not interested in getting this output.

```
master.ProcessBatches("pwt", batches)
theta_matrix = master.GetThetaMatrix("pwt")
```

5. Stop using certain APIs:
   - For python_interface: stop using class Model and ModelConfig message
   - For cpp_interface: stop using class Model and ModelConfig message
   - For c_interface: stop using methods ArtmCreateModel, ArtmReconfigureModel, ArtmInvokeIteration, ArtmAddBatch, ArtmWaitIdle, ArtmSynchronizeModel

Notes on models handling (reusing, sharing input and output, etc)

Is allowed to output the result of ProcessBatches, NormalizeModel, RegularizeModel and MergeModel into an existing model. In this case the existing model will be fully overwritten by the result of the operation. For all operations except ProcessBatches it is also allowed to use the same model in inputs and as an output. For example, typical usage of MergeModel involves combining "nwt" and "nwt_hat" back into "nwt". This scenario is fully supported. The output and input of ProcessBatches must refer to two different models. Finally, note that MergeModel will ignore all non-existing models in the input (and log a warning). However, if none of the input models exist then MergeModel will thrown an error.

Known differences

1. Decorrelator regularizer will give slightly different result in the following scenario:

```
master.ProcessBatches("pwt", batches, "nwt")
master.RegularizeModel("pwt", "nwt", "rwt", phi_regularizers)
master.NormalizeModel("nwt", "pwt", "rwt")
```

   To get the same result as from model.Synchronize() adjust your script as follows:

```
master.ProcessBatches("pwt", batches, "nwt")
master.NormalizeModel("nwt", "pwt_temp")
master.RegularizeModel("pwt_temp", "nwt", "rwt", phi_regularizers)
master.NormalizeModel("nwt", "pwt", "rwt")
```

2. You may use GetThetaMatrix(pwt) to retrieve Theta-matrix, previously calculated for new-style models inside ProcessBatches. However, you can not use GetThetaMatrix(pwt, batch) for new models. They do not have corresponding ModelConfig, and as a result you need to go through ProcessBatches to pass all parameters.

Network modus operandi is removed

Network modus operandi had been removed from BigARTM v0.7.0.

This decision had been taken because current implementation struggle from many issues, particularly from poor performance and stability. We expect to re-implement this functionality on top of new-style models.

Please, let us know if this caused issues for you, and we will consider to re-introduce networking in v0.8.0.

Coherence regularizer and scores (experimental)

Refer to example in example16_coherence_score.py.

7.5.2 BigARTM v0.7.1 Release notes

We are happy to introduce BigARTM v0.7.1, which brings you the following changes:

- BigARTM noteboks — new source of information about BigARTM
- ArtmModel — a brand new Python API
- Much faster retrieval of Phi and Theta matrices from Python
- Much faster dictionary imports from Python
- Auto-detect and use all CPU cores by default
- Fixed Import/Export of topic models (was broken in v0.7.0)
- New capability to implement Phi-regularizers in Python code

- Improvements in Coherence score

Before you upgrade to BigARTM v0.7.1 please review the changes that *break backward compatibility*.

BigARTM notebooks

BigARTM notebooks is your go-to links to read more ideas, examples and other information around BigARTM:

- BigARTM notebooks in English
- BigARTM notebooks in Russian

ArtmModel

Best thing about ArtmModel is that this API had been designed by BigARTM users. Not by BigARTM programmers. This means that BigARTM finally has a nice, clean and easy-to-use programming interface for Python. Don't believe it? Just take a look and some examples:

- ArtmModel examples in English
- ArtmModel examples in Russian

That is cool, right? This new API allows you to load input data from several file formats, infer topic model, find topic distribution for new documents, visualize scores, apply regularizers, and perform many other actions. Each action typically takes one line to write, which allows you to work with BigARTM interactively from Python command line.

ArtmModel exposes most of BigARTM functionality, and it should be sufficiently powerful to cover 95% of all BigARTM use-cases. However, for the most advanced scenarios you might still need to go through the previous API (artm.library). When in doubt which API to use, ask bigartm-users@googlegroups.com — we are there to help!

Coding Phi-regularizers in Python code

This is of course one of those very advanced scenarios where you need to go down to the old API :) Take a look at this example:

- example19_regularize_model
- example20_attach_model

First one tells how to use Phi regularizers, built into BigARTM. Second one provides a new capability to manipulate Phi matrix from Python. We call this **Attach** numpy matrix to the model, because this is similar to attaching debugger (like gdb or Visual Studio) to a running application.

To implement your own Phi regularizer in Python you need to to **attach** to `rwt` model from the first example, and update its values.

Other changes

Fast retrieval of Phi and Theta matrices. In BigARTM v0.7.1 dense Phi and Theta matrices will be retrieved to Python as numpy matrices. All copying work will be done in native C++ code. This is much faster comparing to current solution, where all data is transferred in a large Protobuf message which needs to be deserialized in Python. ArtmModel already takes advantage of this performance improvements.

Fast dictionary import. BigARTM core now supports importing dictionary files from disk, so you no longer have to load them to Python. ArtmModel already take advantage of this performance improvement.

Auto-detect number of CPU cores. You no longer need to specify `num_processors` parameter. By default BigARTM will detect the number of cores on your machine and load all of them. `num_processors` still can be used to limit CPU resources used by BigARTM.

Fixed Import/Export of topic models. Export and Import of topic models will now work. As simple as this:

```
master.ExportModel("pwt", "file_on_disk.model")
master.ImportModel("pwt", "file_on_disk.model")
```

This will also take care of very large models above 1 GB that does not fit into single protobuf message.

Coherence scores. Ask bigartm-users@googlegroups.com if you are interested :)

Breaking changes

- **Changes in Python methods `MasterComponent.GetTopicModel` and `MasterComponent.GetThetaMatrix`**

  From BigARTM v0.7.1 and onwards method `MasterComponent.GetTopicModel` of the low-level Python API will return a tuple, where first argument is of type TopicModel (protobuf message), and second argument is a numpy matrix. TopicModel message will keep all fields as usual, except token_weights field which will became empty. Information from token_weights field had been moved to numpy matrix (rows = tokens, columns = topics).

  Similarly, `MasterComponent.GetThetaMatrix` will also return a tuple, where first argument is of type ThetaMatrix (protobuf message), and second argument is a numpy matrix. ThetaMatrix message will keep all fields as usual, except item_weights field which will became empty. Information from item_weights field had been moved to numpy matrix (rows = items, columns = topics).

  Updated examples:

  - example11_get_theta_matrix.py
  - example12_get_topic_model

  Warning: Use the followign syntax to restore the old behaviour:
  - MasterComponent.GetTopicModel(use_matrix = False)
  - MasterComponent.GetThetaMatrix(use_matrix = False)

  This will return a complete protobuf message, without numpy matrix.

- **Python method ParseCollectionOrLoadDictionary is now obsolete**
  - Use ParseCollection method to convert collection into a set of batches
  - Use MasterComponent.ImportDictionary to load dictionary into BigARTM
  - Updated example: example06_use_dictionaries.py

7.5.3 BigARTM v0.7.2 Release notes

We are happy to introduce BigARTM v0.7.2, which brings you the following changes:

- Enhancements in high-level python API (`ArtmModel` -> `ARTM`)
- Enhancements in low-level python API (`library.py` -> `master_component.py`)
- Enhancements in CLI interface (`cpp_client`)
- Status and information retrievals from BigARTM

- Allow float token counts (`token_count` -> `token_weight`)
- Allow custom weights for each batch (`ProcessBatchesArgs.batch_weight`)
- Bug fixes and cleanup in the online documentation

Enhancements in Python APIs

Note that `ArtmModel` had been renamed to `ARTM`. The naming conventions follow the same pattern as in scikit learn (e.g. `fit`, `transform` and `fit_transform` methods).

Also note that all input data is now handled by `BatchVectorizer` class.

Refer to noteboods in English and in Russian for further details about `ARTM` interface.

Also note that previous low-level python API `library.py` is superseeded by a new API `master_component.py`. For now both APIs are available, but the old one will be removed in future releases. Refer to this folder for futher examples of the new low-level python API.

Remember that any use of low-level APIs is discouraged. Our recommendation is to always use the high-level python API `ARTM`, and e-mail us know if some functionality is not exposed there.

Enhancements in CLI interface

BigARTM command line interface `cpp_client` had been enhanced with the following options:

- `--load_model` - to load model from file before processing
- `--save_model` - to save the model to binary file after processing
- `--write_model_readable` - to output the model in a human-readable format (CSV)
- `--write_predictions` - to write prediction in a human-readable format (CSV)
- `--dictionary_min_df` - to filter out tokens present in less than N documents / less than P% of documents
- `--dictionary_max_df` - filter out tokens present in less than N documents / less than P% of documents
- `--tau0` - an option of the online algorith, describing the weight parameter in the online update formula. Optional, defaults to `1024`.
- `--kappa` - an option of the online algorithm, describing the exponent parameter in the online update formula. Optional, defaults to `0.7`.

Note that for `--dictionary_min_df` and `--dictionary_max_df` can be treated as number, fraction, percent.

- Use a percentage `%` sign to specify percentage value
- Use a floating value in `[0, 1)` range to specify a fraction
- Use an integer value (`1` or greater) to indicate a number

7.5.4 BigARTM v0.7.3 Release notes

BigARTM v0.7.3 releases the following changes:

- New command line tool for BigARTM
- Support for classification in bigartm CLI
- Support for asynchronous processing of batches
- Improvements in coherence regularizer and coherence score

- New *TopicMass* score for phi matrix
- Support for documents markup
- New API for importing batches through memory

New command line tool for BigARTM

New CLI is named `bigartm` (or `bigrtm.exe` on Windows), and it supersedes previous CLI named `cpp_client`. New CLI has the following features:

- Parse collection in one of the 1. Loading Data: BatchVectorizer and Dictionary
- Load dictionary
- Initialize a new model, or import previously created model
- Perform EM-iterations to fit the model
- Export predicted probabilities for all documents into CSV file
- Export model into a file

All command-line options are listed here, and you may see several exampels on BigARTM page at github. At the moment full documentation is only available in Russian.

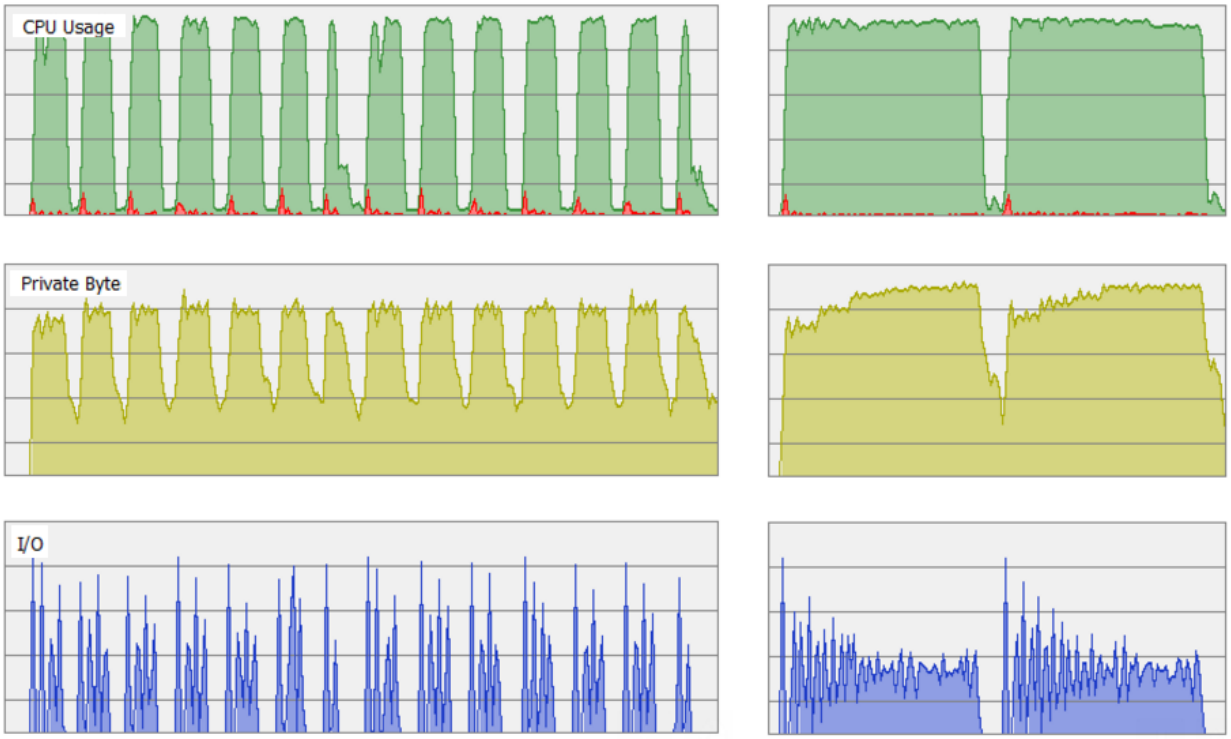
Support for classification in BigARTM CLI

BigARTM CLI is now able to perform classification. The following example assumes that your batches have `target_class` modality in addition to the default modality (`@default_class`).

```
# Fit model
bigartm.exe --use-batches <your batches>
            --use-modality @default_class,target_class
            --topics 50
            --dictionary-min-df 10
            --dictionary-max-df 25%
            --save-model model.bin

# Apply model and output to text files
bigartm.exe --use-batches <your batches>
            --use-modality @default_class,target_class
            --topics 50
            --passes 0
            --load-model model.bin
            --predict-class target_class
            --write-predictions pred.txt
            --write-class-predictions pred_class.txt
            --csv-separator=tab
            --score ClassPrecision
```

Support for asynchronous processing of batches

Asynchronous processing of batches enables applications to overlap EM-iterations better utilize CPU resources. The following chart shows CPU utilization of `bigartm.exe` with (left-hand side) and without async flag (right-hand side).



TopicMass score for phi matrix

Topic mass score calculates cumulated topic mass for each topic. This is a useful metric to monitor balance between topics.

Support for documents markup

Document markup provides topic distribution for each word in a document. Since BigARTM v0.7.3 it is posible to extract this information to use it. A potential application includes color-highlighted maps of the document, where every work is colored according to the most probable topic of the document.

In the code this feature is refered to as `ptdw` matrix. It is possible to extract and regularizer `ptdw` matrices. In future versions it will be also possible to calculate scores based on `ptdw` matrix.

New API for importing batches through memory

New low-level APIs `ArtmImportBatches` and `ArtmDisposeBatches` allow to import batches from memory into BigARTM. Those batches are saved in BigARTM, and can be used for batches processing.

7.5.5 BigARTM v0.7.4 Release notes

BigARTM v0.7.4 is a big release that includes major rework of dictionaries and MasterModel.

bigartm/stable branch

Up until now BigARTM has only one `master` branch, containing the latest code. This branch potentially includes untested code and unfinished features. We are now introducing `bigartm/stable` branch, and encourage all users

to stop using `master` and start fetching from `stable`. `stable` branch will be lagging behind `master`, and moved forward to `master` as soon as mainteiners decide that it is ready. At the same point we will introduce a new tag (something like v0.7.3) and produce a new release for Windows. In addition, `stable` branch also might receive small urgent fixes in between releases, typically to address critical issues reported by our users. Such fixes will be also included in `master` branch.

MasterModel

MasterModel is a new set of low-level APIs that allow users of C-interface to infer models and apply them to new data. The APIs are `ArtmCreateMasterModel`, `ArtmReconfigureMasterModel`, `ArtmFitOfflineMasterModel`, `ArtmFitOnlineMasterModel` and `ArtmRequestTransformMasterModel`, togehter with corresponding protobuf messages. For a usage example see `src/bigartm/srcmain.cc`.

This APIs should be easy to understand for the users who are familiar with Python interface. Basically, we take `ARTM` class in Python, and push it down to the core. Now users can create their model via `MasterModelConfig` (protobuf message), fit via `ArtmFitOfflineMasterModel` or `ArtmFitOnlineMasterModel`, and apply to the new data via `ArtmRequestTransformMasterModel`. This means that the user no longer has to orchestrate low-level building blocks such as `ArtmProcessBatches`, `ArtmMergeModel`, `ArtmRegularizeModel` and `ArtmNormalizeModel`.

`ArtmCreateMasterModel` is similar to `ArtmCreateMasterComponent` in a sence that it returns `master_id`, which can be later passed to all other APIs. This mean that most APIs will continue working as before. This applies to `ArtmRequestThetaMatrix`, `ArtmRequestTopicModel`, `ArtmRequestScore`, and many others.

Rework of dictionaries

Previous implementation of the dictionaries was really messy, and we are trying to clean this up. This effort is not finished yet, however we decided to release current version because it is a major improvement comparing to the previous version. At the low-level (`c_interface`), we now have the following methods to work with dictionaries:

- `ArtmGatherDictionary` collects a dictionary based on a folder with batches,
- `ArtmFilterDictionary` filter tokens from the dictinoary based on their term frequency or document frequency,
- `ArtmCreateDictionary` creates a dictionary from a custom `DictionaryData` object (protobuf message),
- `ArtmRequestDictionary` retrieves a dictionary as `DictionaryData` object (protobuf message),
- `ArtmDisposeDictionary` deletes dictionary object from BigARTM,
- `ArtmImportDictionary` import dictionary from binary file,
- `ArtmExportDictionary` expor tdictionary into binary file.

All dictionaries are identified by a string ID (`dictionary_name`). Dictionaries can be used to initialize the model, in regularizers or in scores.

Note that `ArtmImportDictionary` and `ArtmExportDictionary` now uses a different format. For this reason we require that all imported or exported files end with `.dict` extension. This limitation is only introduced to make users aware of the change in binary format.

Warning: Please note that you have to re-generate all dictionaries, created in previous BigARTM versions. To force this limitation we decided that `ArtmImportDictionary` and `ArtmExportDictionary` will require all imported or exported files end with `.dict` extension. This limitation is only introduced to make users aware of the change in binary format.
Please note that in the next version (*BigARTM v0.8.0*) we are planing to break dictionary format once again. This is because we will introduce `boost.serialize` library for all import and export methods. From that point `boost.serialize` library will allow us to upgrade formats without breaking backwards compatibility.

The following example illustrate how to work with new dictionaries from Python.

```
# Parse collection in UCI format from D:\Datasets\docword.kos.txt and D:\Datasets\vocab.kos.txt
# and store the resulting batches into D:\Datasets\kos_batches
batch_vectorizer = artm.BatchVectorizer(data_format='bow_uci',
                                        data_path=r'D:\Datasets',
                                        collection_name='kos',
                                        target_folder=r'D:\Datasets\kos_batches')

# Initialize the model. For now dictionaries exist within the model,
# but we will address this in the future.
model = artm.ARTM(...)


# Gather dictionary named `dict` from batches.
# The resulting dictionary will contain all distinct tokens that occur
# in those batches, and their term frequencies
model.gather_dictionary("dict", "D:\Datasets\kos_batches")

# Filter dictionary by removing tokens with too high or too low term frequency
# Save the result as `filtered_dict`"
model.filter_dictionary(dictionary_name='dict',
                        dictionary_target_name='filtered_dict',
                        min_df=10, max_df_rate=0.4)

# Initialize model from `diltered_dict`
model.initialize("filtered_dict")

# Import/export functionality
model.save_dictionary("filtered_dict", "D:\Datasets\kos.dict")
model.load_dictionary("filtered_dict2",  "D:\Datasets\kos.dict")
```

Changes in the infrastructure

- Static linkage for bigartm command-line executable on Linux. To disable static linkage use `cmake -DBUILD_STATIC_BIGARTM=OFF ..`
- Install BigARTM python API via `python setup.py install`

Changes in core functionality

- Custom transform function for KL-div regularizers
- Ability to initialize the model with custom seed
- `TopicSelection` regularizers

- `PeakMemory` score (Windows only)
- Different options to name batches when parsing collection (`GUID` as today, and `CODE` for sequential numbering)

Changes in Python API

- `ARTM.dispose()` method for managing native memory
- `ARTM.get_info()` method to retrieve internal state
- Performance fixes
- Expose class prediction functionality

Changes in C++ interface

- Consume `MasterModel` APIs in C++ interface. Going forward this is the only C++ interface that we will support.

Changes in console interface

- Better options to work with dictionaries
- `--write-dictionary-readable` to export dictionary
- `--force` switch to let user overwrite existing files
- `--help` generates much better examples
- `--model-v06` to experiment with old APIs (`ArtmInvokeIteration` / `ArtmWaitIdle` / `ArtmSynchronizeModel`)
- `--write-scores` switch to export scores into file
- `--time-limit` option to time-box model inference(as an alternative to `--passes` switch)

CHAPTER 8

BigARTM Developer's Guide

These pages describe the development process of BigARTM library. If your intent to use BigARTM as a typical user, please proceed to Installation for Windows users or Installation for Linux and Mac OS-X users, depending on your operating system. If you intent is to contribute to the development BigARTM, please proceed to the links below.

8.1 Downloads (Windows)

Download and install the following tools:

- **Git for Windows from http://git-scm.com/download/win**
  - https://github.com/msysgit/msysgit/releases/download/Git-1.9.5-preview20141217/Git-1.9.5-preview20141217.exe
- **Github for Windows from https://windows.github.com/**
  - https://github-windows.s3.amazonaws.com/GitHubSetup.exe
- Visual Studio 2013 Express for Windows Desktop from https://www.visualstudio.com/en-us/products/visual-studio-express-vs.aspx
- **CMake from http://www.cmake.org/download/**
  - http://www.cmake.org/files/v3.0/cmake-3.0.2-win32-x86.exe
- **Prebuilt Boost binaries from http://sourceforge.net/projects/boost/files/boost-binaries/, for example these two:**
  - http://sourceforge.net/projects/boost/files/boost-binaries/1.57.0/boost_1_57_0-msvc-12.0-32.exe/download
  - http://sourceforge.net/projects/boost/files/boost-binaries/1.57.0/boost_1_57_0-msvc-12.0-64.exe/download
- **Python from https://www.python.org/downloads/**
  - https://www.python.org/ftp/python/2.7.9/python-2.7.9.amd64.msi
  - https://www.python.org/ftp/python/2.7.9/python-2.7.9.msi
- (optional) If you plan to build documentation, download and install sphinx-doc as described here: http://sphinx-doc.org/latest/index.html
- (optional) 7-zip – http://www.7-zip.org/a/7z920-x64.msi
- (optional) Putty – http://the.earth.li/~sgtatham/putty/latest/x86/putty.exe

All explicit links are given just for convenience if you are setting up new environment. You are free to choose other versions or tools, and most likely they will work just fine for BigARTM. Remember to match the following: * Visual Studio version must match Boost binaries version, unless you build Boost yourself * Use the same configuration (32 bit or 64 bit) for your Python and BigARTM binaries

8.2 Source code

BigARTM is hosted in public GitHub repository:

https://github.com/bigartm/bigartm

We maintain two branches: master and stable. *master* branch is the latest source code, potentially including some unfinished features. *stable* branch will be lagging behind *master*, and moved forward to *master* as soon as mainteiners decide that it is ready. Typically this should happen at the end of each month. At the same point we will introduce a new tag (something like v0.7.3) and produce a new release for Windows. In addition, *stable* branch also might receive small urgent fixes in between releases, typically to address critical issues reported by our users. Such fixes will be also included in *master* branch.

To contribute a fix you should fork the repository, code your fix and submit a pull request. against *master* branch. All pull requests are regularly monitored by BigARTM maintainers and will be soon merged. Please, keep monitoring the status of your pull request on travis, which is a continuous integration system used by BigARTM project.

8.3 Build C++ code on Windows

The following steps describe the procedure to build BigARTM's C++ code on Windows.

- Download and install GitHub for Windows.
- Clone https://github.com/bigartm/bigartm/ repository to any location on your computer. This location is further refered to as `$(BIGARTM_ROOT)`.
- Download and install Visual Studio 2012 or any newer version. BigARTM will compile just fine with any edition, including any Visual Studio Express edition (available at www.visualstudio.com).
- Install CMake (tested with cmake-3.0.1, Win32 Installer).

  Make sure that CMake executable is added to the `PATH` environmental variable. To achieve this either select the option *"Add CMake to the system PATH for all users"* during installation of CMake, or add it to the `PATH` manually.
- Download and install Boost 1.55 or any newer version.

  We suggest to use the Prebuilt Windows Binaries. Make sure to select version that match your version of Visual Studio. You may choose to work with either x64 or Win32 configuration, both of them are supported.
- Configure system variables `BOOST_ROOT` and `BOOST_LIBRARYDIR`.

  If you have installed boost from the link above, and used the default location, then the setting should look similar to this:

```
setx BOOST_ROOT C:\local\boost_1_56_0
setx BOOST_LIBRARYDIR C:\local\boost_1_56_0\lib32-msvc-12.0
```

  For all future details please refer to the documentation of FindBoost module. We also encourage new CMake users to step through CMake tutorial.

- Install Python 2.7 (tested with Python 2.7.6).

  You may choose to work with either x64 or Win32 version of the Python, but make sure this matches the configuration of BigARTM you have choosed earlier. The x64 installation of python will be incompatible with 32 bit BigARTM, and virse versus.

- Use CMake to generate Visual Studio projects and solution files. To do so, open a command prompt, change working directory to `$(BIGARTM_ROOT)` and execute the following commands:

```
mkdir build
cd build
cmake ..
```

  You might have to explicitly specify the cmake generator, especially if you are working with x64 configuration. To do so, use the following syntax:

```
cmake .. -G"Visual Studio 12 2013 Win64"
```

  CMake will generate Visual Studio under `$(BIGARTM_ROOT)/build/`.

- Open generated solution in Visual Studio and build it as you would usually build any other Visual Studio solution. You may also use MSBuild from Visual Studio command prompt.

  The build will output result into the following folders:

  - `$(BIGARTM_ROOT)/build/bin/[Debug|Release]` — binaries (.dll and .exe)
  - `$(BIGARTM_ROOT)/build/lib/[Debug|Release]` — static libraries

At this point you should be able to run BigARTM tests, located here: `$(BIGARTM_ROOT)/build/bin/*/artm_tests.exe`.

8.4 Python code on Windows

- Install Python 2.7 (this step is already done if you are following the instructions above),
- Add Python to the `PATH` environmental variable

  http://stackoverflow.com/questions/6318156/adding-python-path-on-windows-7

- Follow the instructions in `README` file in directory `$(BIGARTM_ROOT)/3rdparty/protobuf/python/`. In brief, this instructions ask you to run the following commands:

```
python setup.py build
python setup.py test
python setup.py install
```

  On second step you fill see two failing tests:

```
Ran 216 tests in 1.252s
FAILED (failures=2)
```

  This 2 failures are OK to ignore.

At this point you should be able to run BigARTM tests for Python, located under `$(BIGARTM_ROOT)/python/tests/`.

- [Optional] Download and add to MSVS Python Tools 2.0. All necessary instructions can be found at https://pytools.codeplex.com/. This will allow you debug you Python scripts using Visual Studio. You may start with the following solution: `$(BIGARTM_ROOT)/src/artm_vs2012.sln`.

8.5 Compiling .proto files on Windows

1. Open a new command prompt
2. Copy the following file into `$(BIGARTM_ROOT)/src/`
   - `$(BIGARTM_ROOT)/build/bin/CONFIG/protoc.exe`

   Here CONFIG can be either Debug or Release (both options will work equally well).
3. Change working directory to `$(BIGARTM_ROOT)/src/`
4. Run the following commands

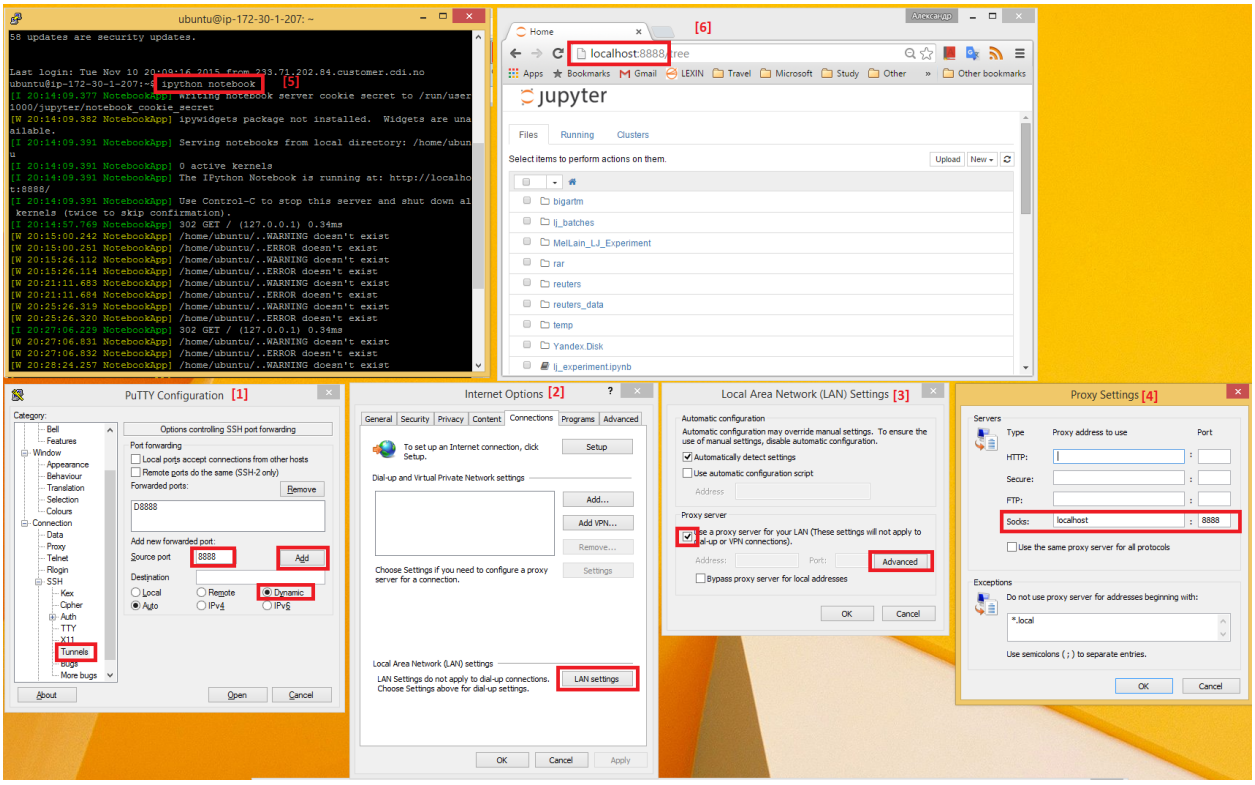
```
.\protoc.exe --cpp_out=. --python_out=. .\artm\messages.proto
.\protoc.exe --cpp_out=. .\artm\core\internals.proto
```

8.6 Working with iPython notebooks remotely

It turned out to be common scenario to run BigARTM on a Linux server (for example on Amazon EC2), while connecting to it from Windows through `putty`. Here is a convenient way to use `ipython notebook` in this scenario:

1. Connect to the Linux machine via putty. Putty needs to be configured with dynamic tunnel for port `8888` as describe here on this page (port `8888` is a default port for `ipython notebook`). The same page describes how to configure internet properties:

   Clicking on Settings in Internet Explorer, or Proxy Settings in Google Chrome, should open this dialogue. Navigate through to the Advanced Proxy section and add localhost:9090 as a SOCKS Proxy.
2. Start `ipython notebook` in your putty terminal.
3. Open your favourite browser on Windows, and go to http://localhost:8888. Enjoy your notebook while the engine runs on remotely :)



8.7 Build C++ code on Linux

Refer to Installation for Linux and Mac OS-X users.

8.7.1 System dependencies

Building BigARTM requires the following components:

- git (any recent version) – for obtaining source code;
- cmake (at least of version 2.8), *make*, *g++* or *clang* compiler with c++11 support, boost (at least of version 1.40) – for building library and binary executable;
- python (version 2.7 or 3.4) – for building Python API for BigARTM.

8.7.2 Building C++ code with CMake

BigARTM is hosted at github repository, with two branches — `stable` and `master`. `stable` contains latest stable release, while `master` is the latest version of the code. Normally `master` is also quite stable, so do not hesitate to try this branch.

Build is built via CMake, as in the following script.

```
cd ~
git clone --branch=stable https://github.com/bigartm/bigartm.git
cd bigartm
mkdir build && cd build
```

```
cmake ..
make
```

Note for Linux users: By default building binary executable `bigartm` requiers static versions of Boost, C and C++ libraries. To alter it, run `cmake` command with option `-DBUILD_BIGARTM_CLI_STATIC=OFF`.

8.7.3 System-wide installation

To install command-line utility, shared library module and Python interface for BigARTM, you can type:

```
sudo make install
```

Normally this will install:

- `bigartm` utility into folder `/usr/local/bin/`;
- shared library `libartm.so` (`artm.dylib` for Max OS-X) into folder `/usr/local/lib/`;
- Python interface for BigARTM into Python-specific system directories, along with necessary dependencies.

If you want to alter target folders for binary and shared library objects, you may specify common prefix while running `cmake` command via option `-DCMAKE_INSTALL_PREFIX=path_to_folder`. By default `CMAKE_INSTALL_PREFIX=/usr/local/`.

8.7.4 Configure BigARTM Python API

Python's interface of BigARTM is normally configured by running `make install`. As an alternative you may configure it manually as described below, however you still need to build native code of BigARTM with `make`.

```
# Step 1 - install Google Protobuf as dependency
# (this can be replaced by "pip install protobuf")
cd ~/bigartm/3rdparty/protobuf/python
sudo python setup.py install

# Step 2 - install Python interface for BigARTM
cd ~/bigartm/python
sudo python setup.py install

# Step 3 - point ARTM_SHARED_LIBRARY variable to libartm.so (libartm.dylib) location
export ARTM_SHARED_LIBRARY=~/bigartm/build/lib/libartm.so        # for linux
export ARTM_SHARED_LIBRARY=~/bigartm/build/lib/libartm.dylib     # for Mac OS X
```

We strongly recommend system-wide installation (e.g. `make install`) as there is no need to keep BigARTM code after it, so you may safely remove folder `~/bigartm/`.

8.7.5 Troubleshooting

If you build BigARTM in existing folder `build` (e.g. you built BigARTM before) and encounter any errors, it may be due to out-of-date file `CMakeCache.txt` in folder `build`. In that case we strongly recommend to delete this file and try to build again.

Using BigARTM Python API you can encounter this error:

```
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
File "build/bdist.linux-x86_64/egg/artm/wrapper/api.py", line 19, in __init__
File "build/bdist.linux-x86_64/egg/artm/wrapper/api.py", line 53, in _load_cdll
OSError: libartm.so: cannot open shared object file: No such file or directory
Failed to load artm shared library. Try to add the location of `libartm.so` file into your LD_LIBRARY
```

This error indicates that BigARTM's python interface can not locate libartm.so (libartm.dylib) files. In such case type `export ARTM_SHARED_LIBRARY=path_to_artm_shared_library`.

8.7.6 BigARTM on Travis-CI

To get a live usage example of BigARTM you may check BigARTM's .travis.yml script and the latest continuous integration build.

8.8 Creating New Regularizer

This manual describes all necessary steps you need to proceed to create your own regularizer in the core of BigARTM library. We assume you are now in the root directory of BigARTM. The Google Protocol Buffers technology will be used, so we also assume you familiar with it. The instructions will be forwarded with corresponding examples of two regularizers, one per matrix (New Regulrizer Phi and New regularizer Theta).

8.8.1 General steps

1. Edit protobuf messages

- Open `src/artm/messages.proto` file and find there the `RegularizerType` message. As you can see, this enum contains all BigARTM regularizers. Add constants for your regularizer (save the natural numeric order, 14 and 15 is an example in case when the last constant is 13):

```
enum RegularizerType {
  RegularizerType_SmoothSparseTheta = 0;
  RegularizerType_SmoothSparsePhi = 1;
  RegularizerType_DecorrelatorPhi = 2;
  ...
  RegularizerType_NewRegularizerPhi = 14;
  RegularizerType_NewRegularizerTheta = 15;
}
```

- In the same file you need to define the configuration of your regularizer. It should contain any meta-data your regularizer will use in it's work. You can see the messages for other regularizers, but in general any regularizer has `topic_name` field, that contains the names of topics, the regularizer will deal with. Regularizers of Phi matrix usually have `class_id` field, that can be an array (and then it denotes all modalities, which tokens will be regularized) or single string (the name of one modality to be regularized). Phi regularizers usually also contains `dictionary_name` parameter, because dictionaries are often contain useful information. Theta regularizers should contain `alpha_iter` parameter, that denotes the additional multipliers for regularization addition r_wt. It is an array with length equal to the number of document passes, and helps to change the influence of the regularizer on each pass through the document in a special way.

Your messages can have the following form:

```
message NewRegularizerPhiConfig {
  repeated string topic_name = 1;
  repeated string class_id = 2;
  optional string dictionary_name = 3;
  ...
}

message NewRegularizerThetaConfig {
  repeated string topic_name = 1;
  repeated float alpha_iter = 2;
  ...
}
```

- You may use the following command to compile messages.proto, see Compiling .proto files on Windows for details):

```
.\protoc.exe --cpp_out=. --python_out=. .\artm\messages.proto
```

Alternatively, we recommend you to build re-project Visual Studio or Linux, and this step will be proceeded automatically. The only recommendation is to remove the old `messages_pb2.py` file from the `python/artm/wrapper` directory.

2. Edit core files and utilities

- The regularizers are the part of C++ core, so you need to create .h and .cc files for you regularizer and store them in the `src/artm/regularizers` directory. We recommend you to use `smooth_sparse_phi.h` and `smooth_sparse_phi.cc` (or `smooth_sparse_theta.h` and `smooth_sparse_theta.cc` respectively) as an example. We will talk about the content of these files in next sections. At first you need to change all names of macroses, classes, methods and types to new ones releated with name of your regularizer (do it in analogy to naming in this file).
- In the head of file `src/artm/core/instance.cc` include file of your new regularizer:

```
#include "artm/regularizer_interface.h"
#include "artm/regularizer/decorrelator_phi.h"
#include "artm/regularizer/multilanguage_phi.h"
#include "artm/regularizer/smooth_sparse_theta.h"
...
#include "artm/regularizer/new_regularizer_phi.h"
#include "artm/regularizer/new_regularizer_theta.h"

#include "artm/score/items_processed.h"
#include "artm/score/sparsity_theta.h"
...
```

- There is a switch/case statement in the same file in a need of expansion:

```
switch (regularizer_type) {
  case artm::RegularizerType_SmoothSparseTheta: {
    CREATE_OR_RECONFIGURE_REGULARIZER(::artm::SmoothSparseThetaConfig,
                                      ::artm::regularizer::SmoothSparseTheta);
    break;
  }

  case artm::RegularizerType_SmoothSparsePhi: {
    CREATE_OR_RECONFIGURE_REGULARIZER(::artm::SmoothSparsePhiConfig,
```

```
                                        ::artm::regularizer::SmoothSparsePhi);
    break;
  }

  ...

  case artm::RegularizerType_NewRegularizerPhi: {
    CREATE_OR_RECONFIGURE_REGULARIZER(::artm::NewRegularizerPhiConfig,
                                        ::artm::regularizer::NewRegularizerPhi);
    break;

  case artm::RegularizerType_NewRegularizerTheta: {
    CREATE_OR_RECONFIGURE_REGULARIZER(::artm::NewRegularizerThetaConfig,
                                        ::artm::regularizer::NewRegularizerTheta);
    break;
  }
```

- Modify file `src/artm/CMakeLists.txt`:

```
regularizer/decorrelator_phi.cc
regularizer/decorrelator_phi.h
regularizer/multilanguage_phi.cc
regularizer/multilanguage_phi.h
regularizer/smooth_sparse_phi.cc
regularizer/smooth_sparse_phi.h
...
regularizer/new_regularizer_phi.cc
regularizer/new_regularizer_phi.h
regularizer/new_regularizer_theta.cc
regularizer/new_regularizer_theta.h
```

- Proceed the same operation with `utils/cpplint_files.txt`

3. Changes in Python API code

- Edit `python/artm/wrapper/constants.py` to reflect the changes made to `enum RegularizerType` in `messages.proto`:

```
RegularizerType_SmoothSparseTheta = 0
RegularizerType_SmoothSparsePhi = 1
 ...
RegularizerType_NewRegularizerPhi = 14
RegularizerType_NewRegularizerTheta = 15
```

- Update `_regularizer_type` in `python/artm/master_component.py` with something like this:

```
def _regularizer_type(config):
    if isinstance(config, messages.SmoothSparseThetaConfig):
        return constants.RegularizerType_SmoothSparseTheta

    ...

    elif isinstance(config, messages.NewRegularizerPhiConfig):
        return constants.RegularizerType_NewRegularizerPhi

    elif isinstance(config, messages.NewRegularizerThetaConfig):
        return constants.RegularizerType_NewRegularizerTheta
```

- You need to add class-wrapper for your regularizer into the `python/artm/regularizers.py`. Note, that the Phi regularizer should be inherited from the `BaseRegularizerPhi`, and Theta one from `BaseRegularizerTheta`. Use any other class as an example. Note, that these two classes and `BaseRegularizer` has pre-defined fields with properties and setters. Don't repeat these fields and add warning methods for ones that doesn't appear in your regularizer:

```
@property
def class_ids(self):
    raise KeyError('No class_ids parameter')

...
@class_ids.setter
def class_ids(self, class_ids):
    raise KeyError('No class_ids parameter')
```

Also take into consideration the notation of parameters naming (for example, class_ids is a list, and class_id is a scalar). Learn attentively other classes and don't forget to write the doc-strings in the same format.

- Add your regularizers into `__all__` list in `regularizers.py`:

```
__all__ = [
    'KlFunctionInfo',
    'SmoothSparsePhiRegularizer',
    ...
    'NewRegularizerPhi'
    'NewRegularizerTheta'
]
```

- You may need to run

```
python setup.py build
python setup.py install
```

for the changes to take effect.

8.8.2 Phi regularizer C++ code

All you need is to implement the method

```
bool NewRegularizerPhi::NewRegularizerPhi(const ::artm::core::PhiMatrix& p_wt,
                                          const ::artm::core::PhiMatrix& n_wt,
                                          ::artm::core::PhiMatrix* result);
```

Here you use p_wt, n_wt and all information you have got as parameters through the config to count r_wt and put it in the `result` variable. The multiplication on `tau` and usage of coefficients of relative regularzation will be processed in further computations automaticaly and shouldn't worry you.

8.8.3 Theta regularizer C++ code

You need to create a class implementing the `RegularizeThetaAgent` interface (e.g., `NewRegularizerThetaAgent`) and a class implementing `RegularizerInterface` interface (e.g., `NewRegularizerTheta`).

In the `NewRegularizerTheta` class you need to define a `CreateRegularizeThetaAgent` method, which checks arguments and does some initialization work. This method will be called every outer iteration, once for every batch.

In the `NewRegularizerThetaAgent` class you need to define an `Apply` method, which takes the (unnormalized) probability distribution `p(t|d)` for a given `d` and transforms it in a some way (e.g. by adding a constant). This method will be called every inner iteration, once for every document in this batch (`inner_iter * batch_size` times in total).

```
void Apply(int item_index, int inner_iter, int topics_size, float* theta);
```

For an example, take a look at `smooth_sparse_theta.cc`.

Note that handling `tau` and `alpha_iter` is your responsibility: your code is assumed to be of form `theta[topic_id] += tau * alpha_iter[inner_iter] * x` instead of just `theta[topic_id] += x`.

8.9 Code style

Configure Visual Studio

Open *Tools / Text Editor / All languages / Tabs* and configure as follows:

- Indenting - smart,
- Tab size - 2,
- Indent size - 2,
- Select "insert spaces".

We also suggest to configure Visual Studio to show space and tab crlf characters (shortcut: Ctrl+R, Ctrl+W), and enable vertical line at 120 characters.

In the code we follow google code style with the following changes:

- Exceptions are allowed
- Indentation must be 2 spaces. Tabs are not allowed.
- No lines should exceed 120 characters.

All .h and .cpp files under `$(BIGARTM_ROOT)/src/artm/` must be verified for code style with cpplint.py script. Files, generated by protobuf compiler, are the only exceptions from this rule.

To run the script you need some version of Python installed on your machine. Then execute the script like this:

```
python cpplint.py --linelength=120 <filename>
```

On Windows you may run this master-script to check all required files:

```
$(BIGARTM_ROOT/utils/cpplint_all.bat.
```

8.10 Release new version

This is a memo for BigARTM maintaineres about releasing new BigARTM version.

- Submit and integrate pull request against `master` branch to
  - bump version in `src/artm/CMakeLists.txt`,

  - bump version in `python/setup.py`,
  - update links to the latest release in `docs/downloads.txt`.
- Run installation tests in bigartm-install repository
  - Bump version in install.sh script
  - Submit pull request and inspect both personal builds - for Appveyor (Windows) and for Travis (Linux)
  - If installation build fails investigate and fix issues before releasing the new version
- Submit and integrate pull request against *stable* branch to integrate all changes from *master* branch.

> **Warning:** Choose REBASE option when merging pull request. DO NOT use squash commits option.

> **Warning:** Make sure to wait for personal build (appveyor and travis), and investigate issues if builds are failing.

- Create new release here:
  - fill in `vX.Y.Z` tag
  - fill in a description, using previous release as an example
  - download `BigARTM.7z` from `python2` job in the latest appveyor build. Rename it to `BigARTM_vX.Y.Z_win64.7z`, and attach as binary to the release.
- Update bigartm-docker repo and image
  - Only once: install docker from https://www.docker.com/ . If you run docker in a VM, configure memory to at least 8 GB (otherwise docker build fails to compile bigartm c++ code). If you are new to docker, read a tutorial on how to publish containers on docker hub.
  - Edit Dockerfile to update BigARTM version in the *git clone* command. Click `Commit changes` to submit this change directly to master branch of `bigartm-docker` repository.
  - Clone `bigartm/bigartm-docker` repository and open command shell. Change working directory to the root of `bigartm-docker` repository.
  - Build and publish the image. Example:

```
docker build .
docker login
docker images   # to find the tag
docker tag e744db1be133 ofrei/bigartm:v0.8.2
docker tag e744db1be133 ofrei/bigartm:latest
docker push ofrei/bigartm:v0.8.2
docker push ofrei/bigartm:latest
```

  - Try to execute commands from bigartm-docker/README.md to test the image.
- Send an announce e-mail to `bigartm-users@googlegroups.com`

8.11 Messages

This document explains all protobuf messages that can be transfered between the user code and BigARTM library.

Warning: Remember that all fields is marked as *optional* to enhance backwards compatibility of the binary protobuf format. Some fields will result in run-time exception when not specified. Please refer to the documentation of each field for more details.

Note that we discourage any usage of fields marked as *obsolete*. Those fields will be removed in future releases.

8.11.1 DoubleArray

class messages_pb2.**DoubleArray**

Represents an array of double-precision floating point values.

```
message DoubleArray {
  repeated double value = 1 [packed = true];
}
```

8.11.2 FloatArray

class messages_pb2.**FloatArray**

Represents an array of single-precision floating point values.

```
message FloatArray {
  repeated float value = 1 [packed = true];
}
```

8.11.3 BoolArray

class messages_pb2.**BoolArray**

Represents an array of boolean values.

```
message BoolArray {
  repeated bool value = 1 [packed = true];
}
```

8.11.4 IntArray

class messages_pb2.**IntArray**

Represents an array of integer values.

```
message IntArray {
  repeated int32 value = 1 [packed = true];
}
```

8.11.5 Item

class messages_pb2.**Item**

Represents a unit of textual information. A typical example of an item is a document that belongs to some text collection.

```
message Item {
  optional int32 id = 1;
  repeated Field field = 2;
  optional string title = 3;
}
```

Item.**id**
An integer identifier of the item.

Item.**field**
A set of all fields withing the item.

Item.**title**
An optional title of the item.

8.11.6 Field

class messages_pb2.**Field**

Represents a field withing an item. The idea behind fields is that each item might have its title, author, body, abstract, actual text, links, year of publication, etc. Each of this entities should be represented as a Field. The topic model defines how those fields should be taken into account when BigARTM infers a topic model. Currently each field is represented as "bag-of-words" — each token is listed together with the number of its occurrences. Note that each Field is always part of an Item, Item is part of a Batch, and a batch always contains a list of tokens. Therefore, each Field just lists the indexes of tokens in the Batch.

```
message Field {
  optional string name = 1 [default = "@body"];
  repeated int32 token_id = 2;
  repeated int32 token_count = 3;
  repeated int32 token_offset = 4;

  optional string string_value = 5;
  optional int64 int_value = 6;
  optional double double_value = 7;
  optional string date_value = 8;

  repeated string string_array = 16;
  repeated int64 int_array = 17;
  repeated double double_array = 18;
  repeated string date_array = 19;
}
```

8.11.7 Batch

class messages_pb2.**Batch**

Represents a set of items. In BigARTM a batch is never split into smaller parts. When it comes to concurrency this means that each batch goes to a single processor. Two batches can be processed concurrently, but items in one batch are always processed sequentially.

```
message Batch {
  repeated string token = 1;
  repeated Item item = 2;
  repeated string class_id = 3;
  optional string description = 4;
  optional string id = 5;
}
```

`Batch.`**`token`**
: A set value that defines all tokens than may appear in the batch.

`Batch.`**`item`**
: A set of items of the batch.

`Batch.`**`class_id`**
: A set of values that define for classes (modalities) of tokens. This repeated field must have the same length as *`token`*. This value is optional, use an empty list indicate that all tokens belong to the default class.

`Batch.`**`description`**
: An optional text description of the batch. You may describe for example the source of the batch, preprocessing technique and the structure of its fields.

`Batch.`**`id`**
: Unique identifier of the batch in a form of a GUID (example: `4fb38197-3f09-4871-9710-392b14f00d2e`). This field is required.

8.11.8 Stream

class `messages_pb2.`**`Stream`**

Represents a configuration of a stream. Streams provide a mechanism to split the entire collection into virtual subsets (for example, the 'train' and 'test' streams).

```
message Stream {
  enum Type {
    Global = 0;
    ItemIdModulus = 1;
  }

  optional Type type = 1 [default = Global];
  optional string name = 2 [default = "@global"];
  optional int32 modulus = 3;
  repeated int32 residuals = 4;
}
```

`Stream.`**`type`**
: A value that defines the type of the stream.

Global	Defines a stream containing all items in the collection.
ItemIdModulus	Defines a stream containing all items with ID that matches modulus and residuals. An item belongs to the stream iff the modulo reminder of item ID is contained in the residuals field.

Stream.**name**
A value that defines the name of the stream. The name must be unique across all streams defined in the master component.

8.11.9 MasterComponentConfig

class messages_pb2.**MasterComponentConfig**

Represents a configuration of a master component.

```
message MasterComponentConfig {
  optional string disk_path = 2;
  repeated Stream stream = 3;
  optional bool compact_batches = 4 [default = true];
  optional bool cache_theta = 5 [default = false];
  optional int32 processors_count = 6 [default = 1];
  optional int32 processor_queue_max_size = 7 [default = 10];
  optional int32 merger_queue_max_size = 8 [default = 10];
  repeated ScoreConfig score_config = 9;
  optional bool online_batch_processing = 13 [default = false];  // obsolete in BigARTM v0.5.8
  optional string disk_cache_path = 15;
}
```

MasterComponentConfig.**disk_path**
A value that defines the disk location to store or load the collection.

MasterComponentConfig.**stream**
A set of all data streams to configure in master component. Streams can overlap if needed.

MasterComponentConfig.**compact_batches**
A flag indicating whether to compact batches in AddBatch() operation. Compaction is a process that shrinks the dictionary of each batch by removing all unused tokens.

MasterComponentConfig.**cache_theta**
A flag indicating whether to cache theta matrix. Theta matrix defines the discrete probability distribution of each document across the topics in topic model. By default BigARTM infers this distribution every time it processes the document. Option 'cache_theta' allows to cache this theta matrix and re-use theha values when the same document is processed on the next iteration. This option must be set to 'true' before calling method ArtmRequestThetaMatrix().

MasterComponentConfig.**processors_count**
A value that defines the number of concurrent processor components. The number of processors should normally

not exceed the number of CPU cores.

`MasterComponentConfig.`**`processor_queue_max_size`**
A value that defines the maximal size of the processor queue. Processor queue contains batches, prefetch from disk into memory. Recommendations regarding the maximal queue size are as follows:

- the queue size should be at least as large as the number of concurrent processors;

`MasterComponentConfig.`**`merger_queue_max_size`**
A value that defines the maximal size of the merger queue. Merger queue size contains an incremental updates of topic model, produced by processor components. Try reducing this parameter if BigARTM consumes too much memory.

`MasterComponentConfig.`**`score_config`**
A set of all scores, available for calculation.

`MasterComponentConfig.`**`online_batch_processing`**
Obsolete in BigARTM v0.5.8.

`MasterComponentConfig.`**`disk_cache_path`**
A value that defines a writtable disk location where this master component can store some temporary files. This can reduce memory usage, particularly when *`cache_theta`* option is enabled. Note that on clean shutdown master component will will be cleaned this folder automatically, but otherwise it is your responsibility to clean this folder to avoid running out of disk.

8.11.10 ModelConfig

class `messages_pb2.`**`ModelConfig`**

Represents a configuration of a topic model.

```
message ModelConfig {
  optional string name = 1 [default = "@model"];
  optional int32 topics_count = 2 [default = 32];
  repeated string topic_name = 3;
  optional bool enabled = 4 [default = true];
  optional int32 inner_iterations_count = 5 [default = 10];
  optional string field_name = 6 [default = "@body"];  // obsolete in BigARTM v0.5.8
  optional string stream_name = 7 [default = "@global"];
  repeated string score_name = 8;
  optional bool reuse_theta = 9 [default = false];
  repeated string regularizer_name = 10;
  repeated double regularizer_tau = 11;
  repeated string class_id = 12;
  repeated float class_weight = 13;
  optional bool use_sparse_bow = 14 [default = true];
  optional bool use_random_theta = 15 [default = false];
  optional bool use_new_tokens = 16 [default = true];
  optional bool opt_for_avx = 17 [default = true];
}
```

`ModelConfig.`**`name`**
A value that defines the name of the topic model. The name must be unique across all models defined in the master component.

`ModelConfig.`**`topics_count`**
A value that defines the number of topics in the topic model.

ModelConfig.**topic_name**
A repeated field that defines the names of the topics. All topic names must be unique within each topic model. This field is optional, but either *topics_count* or *topic_name* must be specified. If both specified, then *topics_count* will be ignored, and the number of topics in the model will be based on the length of *topic_name* field. When *topic_name* is not specified the names for all topics will be autogenerated.

ModelConfig.**enabled**
A flag indicating whether to update the model during iterations.

ModelConfig.**inner_iterations_count**
A value that defines the fixed number of iterations, performed to infer the theta distribution for each document.

ModelConfig.**field_name**
Obsolete in BigARTM v0.5.8

ModelConfig.**stream_name**
A value that defines which stream the model should use.

ModelConfig.**score_name**
A set of names that defines which scores should be calculated for the model.

ModelConfig.**reuse_theta**
A flag indicating whether the model should reuse theta values cached on the previous iterations. This option require cache_theta flag to be set to 'true' in MasterComponentConfig.

ModelConfig.**regularizer_name**
A set of names that define which regularizers should be enabled for the model. This repeated field must have the same length as *regularizer_tau*.

ModelConfig.**regularizer_tau**
A set of values that define the regularization coefficients of the corresponding regularizer. This repeated field must have the same length as *regularizer_name*.

ModelConfig.**class_id**
A set of values that define for which classes (modalities) to build topic model. This repeated field must have the same length as *class_weight*.

ModelConfig.**class_weight**
A set of values that define the weights of the corresponding classes (modalities). This repeated field must have the same length as *class_id*. This value is optional, use an empty list to set equal weights for all classes.

ModelConfig.**use_sparse_bow**
A flag indicating whether to use sparse representation of the Bag-of-words data. The default setting (*use_sparse_bow* = *true*) is best suited for processing textual collections where every token is represented in a small fraction of all documents. Dense representation (*use_sparse_bow* = *false*) better fits for non-textual collections (for example for matrix factorization).

Note that *class_weight* and *class_id* must not be used together with *use_sparse_bow*=*false*.

ModelConfig.**use_random_theta**
A flag indicating whether to initialize `p(t|d)` distribution with random uniform distribution. The default setting (*use_random_theta* = *false*) sets `p(t|d) = 1/T`, where `T` stands for *topics_count*. Note that *reuse_theta* flag takes priority over *use_random_theta* flag, so that if *reuse_theta* = *true* and there is a cache entry from previous iteration the cache entry will be used regardless of *use_random_theta* flag.

ModelConfig.**use_new_tokens**
A flag indicating whether to automatically include new tokens into the topic model. This setting is set to *True* by default. As a result, every new token observed in batches is automatically incorporated into topic model during the next model synchronization (`ArtmSynchronizeModel()`). The `n_wt_` weights for new tokens randomly generated from `[0..1]` range.

ModelConfig.**opt_for_avx**
: An experimental flag that allows to disable AVX optimization in processor. By default this option is enabled as on average it adds ca. 40% speedup on physical hardware. You may want to disable this option if you are running on Windows inside virtual machine, or in situation when BigARTM performance degrades from iteration to interation.

  This option does not affect the results, and is only intended for advanced users experimenting with BigARTM performance.

8.11.11 RegularizerConfig

class messages_pb2.**RegularizerConfig**

Represents a configuration of a general regularizer.

```
message RegularizerConfig {
  enum Type {
    SmoothSparseTheta = 0;
    SmoothSparsePhi = 1;
    DecorrelatorPhi = 2;
    LabelRegularizationPhi = 4;
  }

  optional string name = 1;
  optional Type type = 2;
  optional bytes config = 3;
}
```

RegularizerConfig.**name**
: A value that defines the name of the regularizer. The name must be unique across all names defined in the master component.

RegularizerConfig.**type**
: A value that defines the type of the regularizer.

SmoothSparseTheta	Smooth-sparse regularizer for theta matrix
SmoothSparsePhi	Smooth-sparse regularizer for phi matrix
DecorrelatorPhi	Decorrelator regularizer for phi matrix
LabelRegularizationPhi	Label regularizer for phi matrix

RegularizerConfig.**config**
: A serialized protobuf message that describes regularizer config for the specific regularizer type.

8.11.12 SmoothSparseThetaConfig

class messages_pb2.**SmoothSparseThetaConfig**

Represents a configuration of a SmoothSparse Theta regularizer.

```
message SmoothSparseThetaConfig {
  repeated string topic_name = 1;
  repeated float alpha_iter = 2;
}
```

SmoothSparseThetaConfig.**topic_name**
: A set of topic names that defines which topics in the model should be regularized. This value is optional, use an empty list to regularize all topics.

SmoothSparseThetaConfig.**alpha_iter**
: A field of the same length as *ModelConfig.inner_iterations_count* that defines relative regularization weight for every iteration inner iterations. The actual regularization value is calculated as product of `alpha_iter[i]` and *ModelConfig.regularizer_tau*.

  To specify different regularization weight for different topics create multiple regularizers with different *topic_name* set, and use different values of *ModelConfig.regularizer_tau*.

8.11.13 SmoothSparsePhiConfig

class messages_pb2.**SmoothSparsePhiConfig**

Represents a configuration of a SmoothSparse Phi regularizer.

```
message SmoothSparsePhiConfig {
  repeated string topic_name = 1;
  repeated string class_id = 2;
  optional string dictionary_name = 3;
}
```

SmoothSparsePhiConfig.**topic_name**
: A set of topic names that defines which topics in the model should be regularized. This value is optional, use an empty list to regularize all topics.

SmoothSparsePhiConfig.**class_id**
: This set defines which classes in the model should be regularized. This value is optional, use an empty list to regularize all classes.

SmoothSparsePhiConfig.**dictionary_name**
: An optional value defining the name of the dictionary to use. The entries of the dictionary are expected to have *DictionaryEntry.key_token*, *DictionaryEntry.class_id* and *DictionaryEntry.value* fields. The actual regularization value will be calculated as a product of *DictionaryEntry.value* and *ModelConfig.regularizer_tau*.

  This value is optional, if no dictionary is specified than all tokens will be regularized with the same weight.

8.11.14 DecorrelatorPhiConfig

class messages_pb2.**DecorrelatorPhiConfig**

Represents a configuration of a Decorrelator Phi regularizer.

```
message DecorrelatorPhiConfig {
  repeated string topic_name = 1;
  repeated string class_id = 2;
}
```

DecorrelatorPhiConfig.**topic_name**
: A set of topic names that defines which topics in the model should be regularized. This value is optional, use an empty list to regularize all topics.

DecorrelatorPhiConfig.**class_id**
: This set defines which classes in the model should be regularized. This value is optional, use an empty list to regularize all classes.

8.11.15 LabelRegularizationPhiConfig

class messages_pb2.**LabelRegularizationPhiConfig**

Represents a configuration of a Label Regularizer Phi regularizer.

```
message LabelRegularizationPhiConfig {
  repeated string topic_name = 1;
  repeated string class_id = 2;
  optional string dictionary_name = 3;
}
```

LabelRegularizationPhiConfig.**topic_name**
: A set of topic names that defines which topics in the model should be regularized.

LabelRegularizationPhiConfig.**class_id**
: This set defines which classes in the model should be regularized. This value is optional, use an empty list to regularize all classes.

LabelRegularizationPhiConfig.**dictionary_name**
: An optional value defining the name of the dictionary to use.

8.11.16 RegularizerInternalState

class messages_pb2.**RegularizerInternalState**

Represents an internal state of a general regularizer.

```
message RegularizerInternalState {
  enum Type {
    MultiLanguagePhi = 5;
  }

  optional string name = 1;
  optional Type type = 2;
  optional bytes data = 3;
}
```

8.11.17 DictionaryConfig

class messages_pb2.**DictionaryConfig**

Represents a static dictionary.

```
message DictionaryConfig {
  optional string name = 1;
  repeated DictionaryEntry entry = 2;
  optional int32 total_token_count = 3;
  optional int32 total_items_count = 4;
}
```

DictionaryConfig.**name**
: A value that defines the name of the dictionary. The name must be unique across all dictionaries defined in the master component.

DictionaryConfig.**entry**
: A list of all entries of the dictionary.

DictionaryConfig.**total_token_count**
: A sum of *DictionaryEntry.token_count* across all entries in this dictionary. The value is optional and might be missing when all entries in the dictionary does not carry the *DictionaryEntry.token_count* attribute.

DictionaryConfig.**total_items_count**
: A sum of *DictionaryEntry.items_count* across all entries in this dictionary. The value is optional and might be missing when all entries in the dictionary does not carry the *DictionaryEntry.items_count* attribute.

8.11.18 DictionaryEntry

class messages_pb2.**DictionaryEntry**

Represents one entry in a static dictionary.

```
message DictionaryEntry {
  optional string key_token = 1;
  optional string class_id = 2;
  optional float value = 3;
  repeated string value_tokens = 4;
  optional FloatArray values = 5;
  optional int32 token_count = 6;
  optional int32 items_count = 7;
}
```

DictionaryEntry.**key_token**
: A token that defines the key of the entry.

DictionaryEntry.**class_id**
: The class of the *DictionaryEntry.key_token*.

DictionaryEntry.**value**
: An optional generic value, associated with the entry. The meaning of this value depends on the usage of the dictionary.

DictionaryEntry.**token_count**
: An optional value, indicating the overall number of token occurrences in some collection.

DictionaryEntry.**items_count**
: An optional value, indicating the overall number of documents containing the token.

8.11.19 ScoreConfig

class messages_pb2.**ScoreConfig**

Represents a configuration of a general score.

```
message ScoreConfig {
  enum Type {
    Perplexity = 0;
    SparsityTheta = 1;
    SparsityPhi = 2;
    ItemsProcessed = 3;
    TopTokens = 4;
    ThetaSnippet = 5;
    TopicKernel = 6;
  }

  optional string name = 1;
  optional Type type = 2;
  optional bytes config = 3;
}
```

ScoreConfig.**name**
: A value that defines the name of the score. The name must be unique across all names defined in the master component.

ScoreConfig.**type**
: A value that defines the type of the score.

Perplexity	Defines a config of the Perplexity score
SparsityTheta	Defines a config of the SparsityTheta score
SparsityPhi	Defines a config of the SparsityPhi score
ItemsProcessed	Defines a config of the ItemsProcessed score
TopTokens	Defines a config of the TopTokens score
ThetaSnippet	Defines a config of the ThetaSnippet score
TopicKernel	Defines a config of the TopicKernel score

ScoreConfig.**config**
: A serialized protobuf message that describes score config for the specific score type.

8.11.20 ScoreData

class messages_pb2.**ScoreData**

Represents a general result of score calculation.

```
message ScoreData {
  enum Type {
    Perplexity = 0;
    SparsityTheta = 1;
    SparsityPhi = 2;
    ItemsProcessed = 3;
    TopTokens = 4;
    ThetaSnippet = 5;
    TopicKernel = 6;
  }

  optional string name = 1;
  optional Type type = 2;
  optional bytes data = 3;
}
```

ScoreData.**name**
: A value that describes the name of the score. This name will match the name of the corresponding score config.

ScoreData.**type**
: A value that defines the type of the score.

Perplexity	Defines a Perplexity score data
SparsityTheta	Defines a SparsityTheta score data
SparsityPhi	Defines a SparsityPhi score data
ItemsProcessed	Defines a ItemsProcessed score data
TopTokens	Defines a TopTokens score data
ThetaSnippet	Defines a ThetaSnippet score data
TopicKernel	Defines a TopicKernel score data

ScoreData.**data**
: A serialized protobuf message that provides the specific score result.

8.11.21 PerplexityScoreConfig

class messages_pb2.**PerplexityScoreConfig**

Represents a configuration of a perplexity score.

```
message PerplexityScoreConfig {
  enum Type {
    UnigramDocumentModel = 0;
    UnigramCollectionModel = 1;
  }

  optional string field_name = 1 [default = "@body"];  // obsolete in BigARTM v0.5.8
  optional string stream_name = 2 [default = "@global"];
  optional Type model_type = 3 [default = UnigramDocumentModel];
  optional string dictionary_name = 4;
  optional float theta_sparsity_eps = 5 [default = 1e-37];
  repeated string theta_sparsity_topic_name = 6;
}
```

PerplexityScoreConfig.**field_name**
: Obsolete in BigARTM v0.5.8

PerplexityScoreConfig.**stream_name**
: A value that defines which stream should be used in perplexity calculation.

8.11.22 PerplexityScore

class messages_pb2.**PerplexityScore**

Represents a result of calculation of a perplexity score.

```
message PerplexityScore {
  optional double value = 1;
  optional double raw = 2;
  optional double normalizer = 3;
  optional int32 zero_words = 4;
  optional double theta_sparsity_value = 5;
  optional int32 theta_sparsity_zero_topics = 6;
```

```
  optional int32 theta_sparsity_total_topics = 7;
}
```

PerplexityScore.**value**
: A perplexity value which is calculated as exp(-raw/normalizer).

PerplexityScore.**raw**
: A numerator of perplexity calculation. This value is equal to the likelihood of the topic model.

PerplexityScore.**normalizer**
: A denominator of perplexity calculation. This value is equal to the total number of tokens in all processed items.

PerplexityScore.**zero_words**
: A number of tokens that have zero probability p(wlt,d) in a document. Such tokens are evaluated based on to unigram document model or unigram colection model.

PerplexityScore.**theta_sparsity_value**
: A fraction of zero entries in the theta matrix.

8.11.23 SparsityThetaScoreConfig

class messages_pb2.**SparsityThetaScoreConfig**

Represents a configuration of a theta sparsity score.

```
message SparsityThetaScoreConfig {
  optional string field_name = 1 [default = "@body"];  // obsolete in BigARTM v0.5.8
  optional string stream_name = 2 [default = "@global"];
  optional float eps = 3 [default = 1e-37];
  repeated string topic_name = 4;
}
```

SparsityThetaScoreConfig.**field_name**
: Obsolete in BigARTM v0.5.8

SparsityThetaScoreConfig.**stream_name**
: A value that defines which stream should be used in theta sparsity calculation.

SparsityThetaScoreConfig.**eps**
: A small value that defines zero threshold for theta probabilities. Theta values below the threshold will be counted as zeros when calculating theta sparsity score.

SparsityThetaScoreConfig.**topic_name**
: A set of topic names that defines which topics should be used for score calculation. The names correspond to *ModelConfig.topic_name*. This value is optional, use an empty list to calculate the score for all topics.

8.11.24 SparsityThetaScore

class messages_pb2.**SparsityThetaScoreConfig**

Represents a result of calculation of a theta sparsity score.

```
message SparsityThetaScore {
  optional double value = 1;
  optional int32 zero_topics = 2;
  optional int32 total_topics = 3;
}
```

SparsityThetaScore.**value**
A value of theta sparsity that is calculated as zero_topics / total_topics.

SparsityThetaScore.**zero_topics**
A numerator of theta sparsity score. A number of topics that have zero probability in a topic-item distribution.

SparsityThetaScore.**total_topics**
A denominator of theta sparsity score. A total number of topics in a topic-item distributions that are used in theta sparsity calculation.

8.11.25 SparsityPhiScoreConfig

class messages_pb2.**SparsityPhiScoreConfig**

Represents a configuration of a sparsity phi score.

```
message SparsityPhiScoreConfig {
  optional float eps = 1 [default = 1e-37];
  optional string class_id = 2;
  repeated string topic_name = 3;
}
```

SparsityPhiScoreConfig.**eps**
A small value that defines zero threshold for phi probabilities. Phi values below the threshold will be counted as zeros when calculating phi sparsity score.

SparsityPhiScoreConfig.**class_id**
A value that defines the class of tokens to use for score calculation. This value corresponds to *ModelConfig.class_id* field. This value is optional. By default the score will be calculated for the default class (‘@default_class‘).

SparsityPhiScoreConfig.**topic_name**
A set of topic names that defines which topics should be used for score calculation. This value is optional, use an empty list to calculate the score for all topics.

8.11.26 SparsityPhiScore

class messages_pb2.**SparsityPhiScore**

Represents a result of calculation of a phi sparsity score.

```
message SparsityPhiScore {
  optional double value = 1;
  optional int32 zero_tokens = 2;
  optional int32 total_tokens = 3;
}
```

SparsityPhiScore.**value**
A value of phi sparsity that is calculated as zero_tokens / total_tokens.

SparsityPhiScore.**zero_tokens**
A numerator of phi sparsity score. A number of tokens that have zero probability in a token-topic distribution.

SparsityPhiScore.**total_tokens**
A denominator of phi sparsity score. A total number of tokens in a token-topic distributions that are used in phi sparsity calculation.

8.11.27 ItemsProcessedScoreConfig

class messages_pb2.**ItemsProcessedScoreConfig**

Represents a configuration of an items processed score.

```
message ItemsProcessedScoreConfig {
  optional string field_name = 1 [default = "@body"];  // obsolete in BigARTM v0.5.8
  optional string stream_name = 2 [default = "@global"];
}
```

ItemsProcessedScoreConfig.**field_name**
Obsolete in BigARTM v0.5.8

ItemsProcessedScoreConfig.**stream_name**
A value that defines which stream should be used in calculation of processed items.

8.11.28 ItemsProcessedScore

class messages_pb2.**ItemsProcessedScore**

Represents a result of calculation of an items processed score.

```
message ItemsProcessedScore {
  optional int32 value = 1;
}
```

ItemsProcessedScore.**value**
A number of items that belong to the stream *ItemsProcessedScoreConfig.stream_name* and have been processed during iterations. Currently this number is aggregated throughout all iterations.

8.11.29 TopTokensScoreConfig

class messages_pb2.**TopTokensScoreConfig**

Represents a configuration of a top tokens score.

```
message TopTokensScoreConfig {
  optional int32 num_tokens = 1 [default = 10];
  optional string class_id = 2;
  repeated string topic_name = 3;
}
```

TopTokensScoreConfig.**num_tokens**
A value that defines how many top tokens should be retrieved for each topic.

TopTokensScoreConfig.**class_id**
A value that defines for which class of the model to collect top tokens. This value corresponds to *ModelConfig.class_id* field.

This parameter is optional. By default tokens will be retrieved for the default class (‘@default_class‘).

TopTokensScoreConfig.**topic_name**
: A set of values that represent the names of the topics to include in the result. The names correspond to *ModelConfig.topic_name*.

  This parameter is optional. By default top tokens will be calculated for all topics in the model.

8.11.30 TopTokensScore

class messages_pb2.**TopTokensScore**

Represents a result of calculation of a top tokens score.

```
message TopTokensScore {
  optional int32 num_entries = 1;
  repeated string topic_name = 2;
  repeated int32 topic_index = 3;
  repeated string token = 4;
  repeated float weight = 5;
}
```

The data in this score is represented in a table-like format. sorted on topic_index. The following code block gives a typical usage example. The loop below is guarantied to process all top-N tokens for the first topic, then for the second topic, etc.

```
for (int i = 0; i < top_tokens_score.num_entries(); i++) {
  // Gives a index from 0 to (model_config.topics_size() - 1)
  int topic_index = top_tokens_score.topic_index(i);

  // Gives one of the topN tokens for topic 'topic_index'
  std::string token = top_tokens_score.token(i);

  // Gives the weight of the token
  float weight = top_tokens_score.weight(i);
}
```

TopTokensScore.**num_entries**
: A value indicating the overall number of entries in the score. All the remaining repeated fiels in this score will have this length.

TopTokensScore.**token**
: A repeated field of *num_entries* elements, containing tokens with high probability.

TopTokensScore.**weight**
: A repeated field of *num_entries* elements, containing the *p(t|w)* probabilities.

TopTokensScore.**topic_index**
: A repeated field of *num_entries* elements, containing integers between 0 and (*ModelConfig.topics_count* - 1).

TopTokensScore.**topic_name**
: A repeated field of *num_entries* elements, corresponding to the values of *ModelConfig.topic_name* field.

8.11.31 ThetaSnippetScoreConfig

class messages_pb2.**ThetaSnippetScoreConfig**

Represents a configuration of a theta snippet score.

```
message ThetaSnippetScoreConfig {
  optional string field_name = 1 [default = "@body"];  // obsolete in BigARTM v0.5.8
  optional string stream_name = 2 [default = "@global"];
  repeated int32 item_id = 3 [packed = true];  // obsolete in BigARTM v0.5.8
  optional int32 item_count = 4 [default = 10];
}
```

ThetaSnippetScoreConfig.**field_name**
: Obsolete in BigARTM v0.5.8

ThetaSnippetScoreConfig.**stream_name**
: A value that defines which stream should be used in calculation of a theta snippet.

ThetaSnippetScoreConfig.**item_id**
: Obsolete in BigARTM v0.5.8.

ThetaSnippetScoreConfig.**item_count**
: The number of items to retrieve. ThetaSnippetScore will select last *item_count* processed items and return their theta vectors.

8.11.32 ThetaSnippetScore

class messages_pb2.**ThetaSnippetScore**

Represents a result of calculation of a theta snippet score.

```
message ThetaSnippetScore {
  repeated int32 item_id = 1;
  repeated FloatArray values = 2;
}
```

ThetaSnippetScore.**item_id**
: A set of item ids for which theta snippet have been calculated. Items are identified by the item id.

ThetaSnippetScore.**values**
: A set of values that define topic probabilities for each item. The length of these repeated values will match the number of item ids specified in *ThetaSnippetScore.item_id*. Each repeated field contains float array of topic probabilities in the natural order of topic ids.

8.11.33 TopicKernelScoreConfig

class messages_pb2.**TopicKernelScoreConfig**

Represents a configuration of a topic kernel score.

```
message TopicKernelScoreConfig {
  optional float eps = 1 [default = 1e-37];
  optional string class_id = 2;
  repeated string topic_name = 3;
  optional double probability_mass_threshold = 4 [default = 0.1];
}
```

- *Kernel* of a topic model is defined as the list of all tokens such that the probability p(t | w) exceeds probability mass threshold.

- *Kernel size* of a topic `t` is defined as the number of tokens in its kernel.
- *Topic purity* of a topic `t` is defined as the sum of `p(w | t)` across all tokens `w` in the kernel.
- *Topic contrast* of a topic `t` is defined as the sum of `p(t | w)` across all tokens `w` in the kernel defided by the size of the kernel.

`TopicKernelScoreConfig.`**`eps`**
Defines the minimum threshold on kernel size. In most cases this parameter should be kept at the default value.

`TopicKernelScoreConfig.`**`class_id`**
A value that defines the class of tokens to use for score calculation. This value corresponds to *`ModelConfig.class_id`* field. This value is optional. By default the score will be calculated for the default class (‘@default_class‘).

`TopicKernelScoreConfig.`**`topic_name`**
A set of topic names that defines which topics should be used for score calculation. This value is optional, use an empty list to calculate the score for all topics.

`TopicKernelScoreConfig.`**`probability_mass_threshold`**
Defines the probability mass threshold (see the definition of *kernel* above).

8.11.34 TopicKernelScore

class `messages_pb2.`**`TopicKernelScore`**

Represents a result of calculation of a topic kernel score.

```
message TopicKernelScore {
  optional DoubleArray kernel_size = 1;
  optional DoubleArray kernel_purity = 2;
  optional DoubleArray kernel_contrast = 3;
  optional double average_kernel_size = 4;
  optional double average_kernel_purity = 5;
  optional double average_kernel_contrast = 6;
}
```

`TopicKernelScore.`**`kernel_size`**
Provides the kernel size for all requested topics. The length of this *DoubleArray* is always equal to the overall number of topics. The values of `-1` correspond to non-calculated topics. The remaining values carry the kernel size of the requested topics.

`TopicKernelScore.`**`kernel_purity`**
Provides the kernel purity for all requested topics. The length of this *DoubleArray* is always equal to the overall number of topics. The values of `-1` correspond to non-calculated topics. The remaining values carry the kernel size of the requested topics.

`TopicKernelScore.`**`kernel_contrast`**
Provides the kernel contrast for all requested topics. The length of this *DoubleArray* is always equal to the overall number of topics. The values of `-1` correspond to non-calculated topics. The remaining values carry the kernel contrast of the requested topics.

`TopicKernelScore.`**`average_kernel_size`**
Provides the average kernel size across all the requested topics.

`TopicKernelScore.`**`average_kernel_purity`**
Provides the average kernel purity across all the requested topics.

TopicKernelScore.**average_kernel_contrast**
Provides the average kernel contrast across all the requested topics.

8.11.35 TopicModel

class messages_pb2.**TopicModel**

Represents a topic model. This message can contain data in either dense or sparse format. The key idea behind sparse format is to avoid storing zero p(w|t) elements of the Phi matrix. Please refer to the description of *TopicModel.topic_index* field for more details.

To distinguish between these two formats check whether repeated field *TopicModel.topic_index* is empty. An empty field indicate a dense format, otherwise the message contains data in a sparse format. To request topic model in a sparse format set *GetTopicModelArgs.use_sparse_format* field to True when calling ArtmRequestTopicModel().

```
message TopicModel {
  enum OperationType {
    Initialize = 0;
    Increment = 1;
    Overwrite = 2;
    Remove = 3;
    Ignore = 4;
  }

  optional string name = 1 [default = "@model"];
  optional int32 topics_count = 2;
  repeated string topic_name = 3;
  repeated string token = 4;
  repeated FloatArray token_weights = 5;
  repeated string class_id = 6;

  message TopicModelInternals {
    repeated FloatArray n_wt = 1;
    repeated FloatArray r_wt = 2;
  }

  optional bytes internals = 7;  // obsolete in BigARTM v0.6.3
  repeated IntArray topic_index = 8;
  repeated OperationType operation_type = 9;
}
```

TopicModel.**name**
A value that describes the name of the topic model (*TopicModel.name*).

TopicModel.**topics_count**
A value that describes the number of topics in this message.

TopicModel.**topic_name**
A value that describes the names of the topics included in given *TopicModel* message. This values will represent a subset of topics, defined by *GetTopicModelArgs.topic_name* message. In case of empty *GetTopicModelArgs.topic_name* this values will correspond to the entire set of topics, defined in *ModelConfig.topic_name* field.

TopicModel.**token**
The set of all tokens, included in the topic model.

TopicModel.**token_weights**
: A set of token weights. The length of this repeated field will match the length of the repeated field *TopicModel.token*. The length of each *FloatArray* will match the *TopicModel.topics_count* field (in dense representation), or the length of the corresponding *IntArray* from *TopicModel.topic_index* field (in sparse representation).

TopicModel.**class_id**
: A set values that specify the class (modality) of the tokens. The length of this repeated field will match the length of the repeated field *TopicModel.token*.

TopicModel.**internals**
: Obsolete in BigARTM v0.6.3.

TopicModel.**topic_index**
: A repeated field used for sparse topic model representation. This field has the same length as *TopicModel.token*, *TopicModel.class_id* and *TopicModel.token_weights*. Each element in *topic_index* is an instance of *IntArray* message, containing a list of values between 0 and the length of *TopicModel.topic_name* field. This values correspond to the indices in *TopicModel.topic_name* array, and tell which topics has non-zero `p(w|t)` probabilities for a given token. The actual `p(w|t)` values can be found in *TopicModel.token_weights* field. The length of each *IntArray* message in *TopicModel.topic_index* field equals to the length of the corresponding *FloatArray* message in *TopicModel.token_weights* field.

> **Warning:** Be careful with *TopicModel.topic_index* when this message represents a subset of topics, defined by *GetTopicModelArgs.topic_name*. In this case indices correspond to the selected subset of topics, which might not correspond to topic indices in the original *ModelConfig* message.

TopicModel.**operation_type**
: A set of values that define operation to perform on each token when topic model is used as an argument of `ArtmOverwriteTopicModel()`.

`Initialize`	Indicates that a new token should be added to the topic model. Initial `n_wt` counter will be initialized with random value from `[0, 1]` range. *TopicModel.token_weights* is ignored. This operation is ignored if token already exists.
`Increment`	Indicates that `n_wt` counter of the token should be increased by values, specified in *TopicModel.token_weights* field. A new token will be created if it does not exist yet.
`Overwrite`	Indicates that `n_wt` counter of the token should be set to the value, specified in *TopicModel.token_weights* field. A new token will be created if it does not exist yet.
`Remove`	Indicates that the token should be removed from the topic model. *TopicModel.token_weights* is ignored.
`Ignore`	Indicates no operation for the token. The effect is the same as if the token is not present in this message.

8.11.36 ThetaMatrix

class `messages_pb2.`**ThetaMatrix**

Represents a theta matrix. This message can contain data in either dense or sparse format. The key idea behind sparse format is to avoid storing zero `p(t|d)` elements of the Theta matrix. Sparse representation of Theta matrix is equivalent to sparse representation of Phi matrix. Please, refer to *TopicModel* for detailed description of the sparse format.

```
message ThetaMatrix {
  optional string model_name = 1 [default = "@model"];
  repeated int32 item_id = 2;
```

```
  repeated FloatArray item_weights = 3;
  repeated string topic_name = 4;
  optional int32 topics_count = 5;
  repeated string item_title = 6;
  repeated IntArray topic_index = 7;
}
```

ThetaMatrix.**model_name**
: A value that describes the name of the topic model. This name will match the name of the corresponding model config.

ThetaMatrix.**item_id**
: A set of item IDs corresponding to *Item.id* values.

ThetaMatrix.**item_weights**
: A set of item ID weights. The length of this repeated field will match the length of the repeated field *ThetaMatrix.item_id*. The length of each *FloatArray* will match the *ThetaMatrix.topics_count* field (in dense representation), or the length of the corresponding *IntArray* from *ThetaMatrix.topic_index* field (in sparse representation).

ThetaMatrix.**topic_name**
: A value that describes the names of the topics included in given *ThetaMatrix* message. This values will represent a subset of topics, defined by *GetThetaMatrixArgs.topic_name* message. In case of empty *GetTopicModelArgs.topic_name* this values will correspond to the entire set of topics, defined in *ModelConfig.topic_name* field.

ThetaMatrix.**topics_count**
: A value that describes the number of topics in this message.

ThetaMatrix.**item_title**
: A set of item titles, corresponding to *Item.title* values. Beware that this field might be empty (e.g. of zero length) if all items did not have title specified in *Item.title*.

ThetaMatrix.**topic_index**
: A repeated field used for sparse theta matrix representation. This field has the same length as *ThetaMatrix.item_id*, *ThetaMatrix.item_weights* and *ThetaMatrix.item_title*. Each element in *topic_index* is an instance of *IntArray* message, containing a list of values between 0 and the length of *TopicModel.topic_name* field. This values correspond to the indices in *ThetaMatrix.topic_name* array, and tell which topics has non-zero `p(t|d)` probabilities for a given item. The actual `p(t|d)` values can be found in *ThetaMatrix.item_weights* field. The length of each *IntArray* message in *ThetaMatrix.topic_index* field equals to the length of the corresponding *FloatArray* message in *ThetaMatrix.item_weights* field.

  Warning: Be careful with *ThetaMatrix.topic_index* when this message represents a subset of topics, defined by *GetThetaMatrixArgs.topic_name*. In this case indices correspond to the selected subset of topics, which might not correspond to topic indices in the original *ModelConfig* message.

8.11.37 CollectionParserConfig

class messages_pb2.**CollectionParserConfig**

Represents a configuration of a collection parser.

```
message CollectionParserConfig {
  enum Format {
    BagOfWordsUci = 0;
```

```
    MatrixMarket = 1;
  }

  optional Format format = 1 [default = BagOfWordsUci];
  optional string docword_file_path = 2;
  optional string vocab_file_path = 3;
  optional string target_folder = 4;
  optional string dictionary_file_name = 5;
  optional int32 num_items_per_batch = 6 [default = 1000];
  optional string cooccurrence_file_name = 7;
  repeated string cooccurrence_token = 8;
  optional bool use_unity_based_indices = 9 [default = true];
}
```

`CollectionParserConfig.`**`format`**

A value that defines the format of a collection to be parsed.

BagOfWordsUci	A bag-of-words collection, stored in UCI format. UCI format must have two files - *vocab.*.txt* and *docword.*.txt*, defined by *docword_file_path* and *vocab_file_path*. The format of the docword.*.txt file is 3 header lines, followed by NNZ triples: D W NNZ docID wordID count docID wordID count ... docID wordID count The file must be sorted on docID. Values of wordID must be unity-based (not zero-based). The format of the vocab.*.txt file is line containing wordID=n. Note that words must not have spaces or tabs. In vocab.*.txt file it is also possible to specify *Batch.class_id* for tokens, as it is shown in this example: token1 @default_class token2 custom_class token3 @default_class token4 Use space or tab to separate token from its class. Token that are not followed by class label automatically get ''@default_class'' as a lable (see ''token4" in the example).
MatrixMarket	See the description at http://math.nist.gov/MatrixMarket/formats.html In this mode parameter *docword_file_path* must refer to a file in Matrix Market format. Parameter *vocab_file_path* is also required and must refer to a dictionary file exported in gensim format (dictionary.save_as_text()).

CollectionParserConfig.**docword_file_path**

A value that defines the disk location of a `docword.*.txt` file (the bag of words file in sparse format).

`CollectionParserConfig.`**`vocab_file_path`**
A value that defines the disk location of a `vocab.*.txt` file (the file with the vocabulary of the collection).

`CollectionParserConfig.`**`target_folder`**
A value that defines the disk location where to stores all the results after parsing the colleciton. Usually the resulting location will contain a set of *batches*, and a *DictionaryConfig* that contains all unique tokens occured in the collection. Such location can be further passed MasterComponent via *MasterComponentConfig.disk_path*.

`CollectionParserConfig.`**`dictionary_file_name`**
A file name where to save the *DictionaryConfig* message that contains all unique tokens occured in the collection. The file will be created in *target_folder*.

This parameter is optional. The dictionary will be still collected even when this parameter is not provided, but the resulting dictionary will be only returned as the result of ArtmRequestParseCollection, but it will not be stored to disk.

In the resulting dictionary each entry will have the following fields:

- *DictionaryEntry.key_token* - the textual representation of the token,
- *DictionaryEntry.class_id* - the label of the default class ("@DefaultClass"),
- *DictionaryEntry.token_count* - the overall number of occurrences of the token in the collection,
- *DictionaryEntry.items_count* - the number of documents in the collection, containing the token.
- *DictionaryEntry.value* - the ratio between *token_count* and *total_token_count*.

Use ArtmRequestLoadDictionary method to load the resulting dictionary.

`CollectionParserConfig.`**`num_items_per_batch`**
A value indicating the desired number of items per batch.

`CollectionParserConfig.`**`cooccurrence_file_name`**
A file name where to save the *DictionaryConfig* message that contains information about co-occurrence of all pairs of tokens in the collection. The file will be created in *target_folder*.

This parameter is optional. No cooccurrence information will be collected if the filename is not provided.

In the resulting dictionary each entry will correspond to two tokens ('<first>' and '<second>'), and carry the information about co-occurrence of this tokens in the collection.

- *DictionaryEntry.key_token* - a string of the form '<first>~<second>', produced by concatenation of two tokens together via the tilde symbol ('~'). <*first*> tokens is guarantied lexicographic less than the <*second*> token.
- *DictionaryEntry.class_id* - the label of the default class ("@DefaultClass").
- *DictionaryEntry.items_count* - the number of documents in the collection, containing both tokens ('<first>' and '<second>')

Use ArtmRequestLoadDictionary method to load the resulting dictionary.

`CollectionParserConfig.`**`cooccurrence_token`**
A list of tokens to collect cooccurrence information. A cooccurrence of the pair <*first*>~<*second*> will be collected only when both tokens are present in *CollectionParserConfig.cooccurrence_token*.

`CollectionParserConfig.`**`use_unity_based_indices`**
A flag indicating whether to interpret indices in docword file as unity-based or as zero-based. By default *'use_unity_based_indices = True'*, as required by UCI bag-of-words format.

8.11.38 SynchronizeModelArgs

class messages_pb2.**SynchronizeModelArgs**

Represents an argument of synchronize model operation.

```
message SynchronizeModelArgs {
  optional string model_name = 1;
  optional float decay_weight = 2 [default = 0.0];
  optional bool invoke_regularizers = 3 [default = true];
  optional float apply_weight = 4 [default = 1.0];
}
```

SynchronizeModelArgs.**model_name**
The name of the model to be synchronized. This value is optional. When not set, all models will be synchronized with the same decay weight.

SynchronizeModelArgs.**decay_weight**
The decay weight and *apply_weight* define how to combine existing topic model with all increments, calculated since the last ArtmSynchronizeModel(). This is best described by the following formula:

n_wt_new = n_wt_old * decay_weight + n_wt_inc * apply_weight,

where n_wt_old describe current topic model, n_wt_inc describe increment calculated since last ArtmSynchronizeModel(), n_wt_new define the resulting topic model.

Expected values of both parameters are between 0.0 and 1.0. Here are some examples:

- Combination of *decay_weight=0.0* and *apply_weight=1.0* states that the previous Phi matrix of the topic model will be disregarded completely, and the new Phi matrix will be formed based on new increments gathered since last model synchronize.
- Combination of *decay_weight=1.0* and *apply_weight=1.0* states that new increments will be appended to the current Phi matrix without any decay.
- Combination of *decay_weight=1.0* and *apply_weight=0.0* states that new increments will be disregarded, and current Phi matrix will stay unchanged.
- To reproduce Online variational Bayes for LDA algorighm by Matthew D. Hoffman set *decay_weight = 1 - rho* and *apply_weight = rho*, where parameter rho is defined as *rho = exp(tau + t, -kappa)*. See Online Learning for Latent Dirichlet Allocation for further details.

SynchronizeModelArgs.**apply_weight**
See *decay_weight* for the description.

SynchronizeModelArgs.**invoke_regularizers**
A flag indicating whether to invoke all phi-regularizers.

8.11.39 InitializeModelArgs

class messages_pb2.**InitializeModelArgs**

Represents an argument of ArtmInitializeModel() operation. Please refer to example14_initialize_topic_model.py for further information.

```
message InitializeModelArgs {
  enum SourceType {
    Dictionary = 0;
    Batches = 1;
```

```
  }

  message Filter {
    optional string class_id = 1;
    optional float min_percentage = 2;
    optional float max_percentage = 3;
    optional int32 min_items = 4;
    optional int32 max_items = 5;
    optional int32 min_total_count = 6;
    optional int32 min_one_item_count = 7;
  }

  optional string model_name = 1;
  optional string dictionary_name = 2;
  optional SourceType source_type = 3 [default = Dictionary];

  optional string disk_path = 4;
  repeated Filter filter = 5;
}
```

`InitializeModelArgs.`**`model_name`**
: The name of the model to be initialized.

`InitializeModelArgs.`**`dictionary_name`**
: The name of the dictionary containing all tokens that should be initialized.

8.11.40 GetTopicModelArgs

Represents an argument of `ArtmRequestTopicModel()` operation.

```
message GetTopicModelArgs {
  enum RequestType {
    Pwt = 0;
    Nwt = 1;
  }

  optional string model_name = 1;
  repeated string topic_name = 2;
  repeated string token = 3;
  repeated string class_id = 4;
  optional bool use_sparse_format = 5;
  optional float eps = 6 [default = 1e-37];
  optional RequestType request_type = 7 [default = Pwt];
}
```

`GetTopicModelArgs.`**`model_name`**
: The name of the model to be retrieved.

`GetTopicModelArgs.`**`topic_name`**
: The list of topic names to be retrieved. This value is optional. When not provided, all topics will be retrieved.

`GetTopicModelArgs.`**`token`**
: The list of tokens to be retrieved. The length of this field must match the length of *`class_id`* field. This field is optional. When not provided, all tokens will be retrieved.

GetTopicModelArgs.**class_id**
: The list of classes corresponding to all tokens. The length of this field must match the length of *token* field. This field is only required together with *token*, otherwise it is ignored.

GetTopicModelArgs.**use_sparse_format**
: An optional flag that defines whether to use sparse format for the resulting TopicModel message. See TopicModel message for additional information about the sparse format. Note that setting *use_sparse_format* = *true* results in empty *TopicModel.internals* field.

GetTopicModelArgs.**eps**
: A small value that defines zero threshold for p(w|t) probabilities. This field is only used in sparse format. p(w|t) below the threshold will be excluded from the resulting Phi matrix.

GetTopicModelArgs.**request_type**
: An optional value that defines what kind of data to retrieve in this operation.

Pwt	Indicates that the resulting *TopicModel* message should contain p(w|t) probabilities. This values are normalized to form a probability distribution (sum_w p(w|t) = 1 for all topics t).
Nwt	Indicates that the resulting *TopicModel* message should contain internal n_wt counters of the topic model. This values represent an internal state of the topic model.

Default setting is to retrieve p(w|t) probabilities. This probabilities are sufficient to infer p(t|d) distributions using this topic model.

n_wt counters allow you to restore the precise state of the topic model. By passing this values in ArtmOverwriteTopicModel() operation you are guarantied to get the model in the same state as you retrieved it. As the result you may continue topic model inference from the point you have stopped it last time.

p(w|t) values can be also restored via c:func:*ArtmOverwriteTopicModel* operation. The resulting model will give the same p(t|d) distributions, however you should consider this model as *read-only*, and do not call ArtmSynchronizeModel() on it.

8.11.41 GetThetaMatrixArgs

Represents an argument of ArtmRequestThetaMatrix() operation.

```
message GetThetaMatrixArgs {
  optional string model_name = 1;
  optional Batch batch = 2;
  repeated string topic_name = 3;
  repeated int32 topic_index = 4;
  optional bool clean_cache = 5 [default = false];
  optional bool use_sparse_format = 6 [default = false];
  optional float eps = 7 [default = 1e-37];
}
```

GetThetaMatrixArgs.**model_name**
: The name of the model to retrieved theta matrix for.

GetThetaMatrixArgs.**batch**
: The *Batch* to classify with the model.

GetThetaMatrixArgs.**topic_name**
: The list of topic names, describing which topics to include in the Theta matrix. The values of this field should correspond to values in *ModelConfig.topic_name*. This field is optional, by default all topics will be included.

`GetThetaMatrixArgs.`**`topic_index`**
: The list of topic indices, describing which topics to include in the Theta matrix. The values of this field should be an integers between 0 and (*ModelConfig.topics_count* - 1). This field is optional, by default all topics will be included.

  Note that this field acts similar to *GetThetaMatrixArgs.topic_name*. It is not allowed to specify both *topic_index* and *topic_name* at the same time. The recommendation is to use *topic_name*.

`GetThetaMatrixArgs.`**`clean_cache`**
: An optional flag that defines whether to clear the theta matrix cache after this operation. Setting this value to *True* will clear the cache for a topic model, defined by *GetThetaMatrixArgs.model_name*. This value is only applicable when *MasterComponentConfig.cache_theta* is set to *True*.

`GetThetaMatrixArgs.`**`use_sparse_format`**
: An optional flag that defines whether to use sparse format for the resulting `ThetaMatrix` message. See `ThetaMatrix` message for additional information about the sparse format.

`GetThetaMatrixArgs.`**`eps`**
: A small value that defines zero threshold for `p(t|d)` probabilities. This field is only used in sparse format. `p(t|d)` below the threshold will be excluded from the resulting Theta matrix.

8.11.42 GetScoreValueArgs

Represents an argument of get score operation.

```
message GetScoreValueArgs {
  optional string model_name = 1;
  optional string score_name = 2;
  optional Batch batch = 3;
}
```

`GetScoreValueArgs.`**`model_name`**
: The name of the model to retrieved score for.

`GetScoreValueArgs.`**`score_name`**
: The name of the score to retrieved.

`GetScoreValueArgs.`**`batch`**
: The *Batch* to calculate the score. This option is only applicable to cumulative scores. When not provided the score will be reported for all batches processed since last `ArtmInvokeIteration()`.

8.11.43 AddBatchArgs

Represents an argument of `ArtmAddBatch()` operation.

```
message AddBatchArgs {
  optional Batch batch = 1;
  optional int32 timeout_milliseconds = 2 [default = -1];
  optional bool reset_scores = 3 [default = false];
  optional string batch_file_name = 4;
}
```

`AddBatchArgs.`**`batch`**
: The *Batch* to add.

`AddBatchArgs.`**`timeout_milliseconds`**
Timeout in milliseconds for this operation.

`AddBatchArgs.`**`reset_scores`**
An optional flag that defines whether to reset all scores before this operation.

`AddBatchArgs.`**`batch_file_name`**
An optional value that defines disk location of the batch to add. You must choose between parameters *batch_file_name* or *batch* (either of them has to be specified, but not both at the same time).

8.11.44 InvokeIterationArgs

Represents an argument of `ArtmInvokeIteration()` operation.

```
message InvokeIterationArgs {
  optional int32 iterations_count = 1 [default = 1];
  optional bool reset_scores = 2 [default = true];
  optional string disk_path = 3;
}
```

`InvokeIterationArgs.`**`iterations_count`**
An integer value describing how many iterations to invoke.

`InvokeIterationArgs.`**`reset_scores`**
An optional flag that defines whether to reset all scores before this operation.

`InvokeIterationArgs.`**`disk_path`**
A value that defines the disk location with batches to process on this iteration.

8.11.45 WaitIdleArgs

Represents an argument of `ArtmWaitIdle()` operation.

```
message WaitIdleArgs {
  optional int32 timeout_milliseconds = 1 [default = -1];
}
```

`WaitIdleArgs.`**`timeout_milliseconds`**
Timeout in milliseconds for this operation.

8.11.46 ExportModelArgs

Represents an argument of `ArtmExportModel()` operation.

```
message ExportModelArgs {
  optional string file_name = 1;
  optional string model_name = 2;
}
```

`ExportModelArgs.`**`file_name`**
A target file name where to store topic model.

`ExportModelArgs.`**`model_name`**
: A value that describes the name of the topic model. This name will match the name of the corresponding model config.

8.11.47 ImportModelArgs

Represents an argument of `ArtmImportModel()` operation.

```
message ImportModelArgs {
  optional string file_name = 1;
  optional string model_name = 2;
}
```

`ImportModelArgs.`**`file_name`**
: A target file name from where to load topic model.

`ImportModelArgs.`**`model_name`**
: A value that describes the name of the topic model. This name will match the name of the corresponding model config.

a

Symbols

A

B

C

D

E

F

G

H

I

K

L

M

N

O

P

R

S

T

U

V

W

Z