# BigARTM Documentation

***Release 1.0***

**Konstantin Vorontsov**

February 15, 2015

Contents

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.

**Getting help**

Having trouble? We'd like to help!

- Try the *FAQ* – it's got answers to many common questions.
- Looking for specific information? Try the *genindex*, or *search*.
- Search for information in the archives of the bigartm-users mailing list, or post a question.
- Report bugs with BigARTM in our ticket tracker.

**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.

**Distributed**. BigARTM is able to distribute all CPU-intensive processing onto several machines, interconnected by network. We aim to scale up to hundreds of machines, but the real scalability have not been fully tested yet.

**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 Sci-

ence and Technology (project 081-R), Moscow Institute of Physics and Technology.

CHAPTER 2

# Download

- **Windows - latest release**
  - https://github.com/bigartm/bigartm/releases/download/v0.5.7/BigARTM_v0.5.7_win32.7z
  - https://github.com/bigartm/bigartm/releases/download/v0.5.7/BigARTM_v0.5.7_x64.7z
- **Windows - previous releases**
  - https://github.com/bigartm/bigartm/releases/download/v0.5.6/BigARTM_v0.5.6_win32.7z
  - https://github.com/bigartm/bigartm/releases/download/v0.5.6/BigARTM_v0.5.6_x64.7z
  - https://github.com/bigartm/bigartm/releases/download/v0.5.5/BigARTM_v0.5.5_win32.7z
  - https://github.com/bigartm/bigartm/releases/download/v0.5.5/BigARTM_v0.5.5_x64.7z
  - https://github.com/bigartm/bigartm/releases/download/v0.5.4/BigARTM_v0.5.4_win32.7z
  - https://github.com/bigartm/bigartm/releases/download/v0.5.4/BigARTM_v0.5.4_x64.7z
  - https://github.com/bigartm/bigartm/releases/download/v0.5.3/BigARTM_v0.5.3_win32.7z
  - https://github.com/bigartm/bigartm/releases/download/v0.5.3/BigARTM_v0.5.3_x64.7z
  - https://github.com/bigartm/bigartm/releases/download/v0.5.2/BigARTM_v0.5.2_win32.7z
  - https://github.com/bigartm/bigartm/releases/download/v0.5.2/BigARTM_v0.5.2_x64.7z
  - https://github.com/bigartm/bigartm/releases/download/v0.5.1/BigARTM_v0.5.1_win32.7z
  - https://github.com/bigartm/bigartm/releases/download/v0.5.1/BigARTM_v0.5.1_x64.7z

  Please refer to *Tutorial* chapter for installation guide.
- **Linux, Mac OS-X** Currently there is distribution package for Linux or Mac OS-X. To run BigARTM you need to download the source code and built it on your machine. Detailed procesure is available in *Tutorial* and *BigARTM Developer's Guide* chapters.

CHAPTER 3

# Tutorial

This tutorial provides a basic Python programmer's introduction to BigARTM. It demonstrates how to

- install BigARTM library on your computer,
- configure basic BigARTM parameters,
- load the text collection into BigARTM,
- infer topic model and retrieve the results.

## 3.1 Installation on Windows

- Download and install Python 2.7 (https://www.python.org/downloads/).
- Download and unpack the latest BigARTM release (https://github.com/bigartm/bigartm/releases). Choose carefully between win32 and x64 version. The version of BigARTM package must match your version Python installed on your machine.
- Add BigARTM to your `PATH` and `PYTHONPATH` system variables as follows:

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

  Remebmer to change `C:\BigARTM` if you unpacked BigARTM to a different location.
- Setup *Google Protocol Buffers* library, included in the BigARTM release package. To do so, follow the instructions in protobuf/python/README.

The BigARTM package will contain the following files:

| | |
|---|---|
| `bin/` | Precompiled binaries of BigARTM for Windows. This folder must be added to `PATH` system variable. |
| `bin/artm.dll` | Core functionality of the BigARTM library. |
| `bin/node_controller.exe` | Executable that hosts BigARTM nodes in a distributed setting. |
| `bin/cpp_client.exe` | Command line utility allows to perform simple experiments with BigARTM. Remember that not all BigARTM features are available through cpp_client, but it can serve as a good starting point to learn basic functionality. For further details refer to *BigARTM command line utility*. |
| `protobuf/` | A minimalistic version of Google Protocol Buffers (https://code.google.com/p/protobuf/) library, required to run BigARTM from Python. To setup this package follow the instructions in `protobuf/python/README` file. |
| `python/artm/` | Python programming interface to BigARTM library. This folder must be added to `PYTHONPATH` system variable. |
| `` library.py`` | Implements all classes of BigARTM python interface. |
| `` messages_pb2.py`` | Contains all protobuf messages that can be transfered in and out BigARTM core library. Most common features are exposed with their own API methods, so normally you do not use python protobuf messages to operate BigARTM. |
| `python/examples/` | Python examples of how to use BigARTM:<br>• example01_synthetic_collection.py<br>• example02_parse_collection.py<br>• example03_concurrency.py<br>• example04_online_algorithm.py<br>• example05_train_and_test_stream.py<br>• example06_use_dictionaries.py<br>• example07_master_component_proxy.py |

## 3.2 Installation on Linux

Currently there is no distribution package of BigARTM for Linux. BigARTM had been tested on several Linux OS, and it is known to work well, but you have to get the source code and compile it locally on your machine. Please, refer to *Developer's Guide* for further instructions.

To get a live usage example of BigARTM you may check BigARTM's .travis.yml script and the latest continuous integration build.

## 3.3 Intel Math Kernel Library

BigARTM can utilize Intel Math Kernel Library to achieve better performance.

To enable MKL usage on Windows add the path to MKL library to your `PATH` system variable

```
set PATH=%PATH%;"C:\Program Files (x86)\Intel\Composer XE 2013 SP1\redist\intel64\mkl"
```

To enable MKL usage on Linux create a new system variable `MKL_PATH` and set it as follows

```
export MKL_PATH="/opt/intel/mkl/lib/intel64/"
```

## 3.4 First steps

Run example02_parse_collection.py script from BigARTM distributive. It will load a text collection from disk and use iterative scans over the collection to infer some topic models. Then it outputs top works in each topic and topic classification of some random documents. Running the script produces the following output:

```
>python example02_parse_collection.py

 No batches found, parsing them from textual collection...  OK.
 Iter#0 : Perplexity = 6921.336 , Phi sparsity = 0.046  , Theta sparsity = 0.050
 Iter#1 : Perplexity = 2538.800 , Phi sparsity = 0.101  , Theta sparsity = 0.082
 Iter#2 : Perplexity = 2208.745 , Phi sparsity = 0.173  , Theta sparsity = 0.156
 Iter#3 : Perplexity = 1953.304 , Phi sparsity = 0.259  , Theta sparsity = 0.229
 Iter#4 : Perplexity = 1776.102 , Phi sparsity = 0.337  , Theta sparsity = 0.296
 Iter#5 : Perplexity = 1693.438 , Phi sparsity = 0.395  , Theta sparsity = 0.322
 Iter#6 : Perplexity = 1650.383 , Phi sparsity = 0.442  , Theta sparsity = 0.334
 Iter#7 : Perplexity = 1624.210 , Phi sparsity = 0.478  , Theta sparsity = 0.341

 Top tokens per topic:
 Topic#1: democratic campaign dean poll general edwards party voters john republicans
 Topic#2: iraq administration war white bushs officials time people attacks news
 Topic#3: military iraqi abu iraqis fallujah soldiers truth ghraib army forces
 Topic#4: state republican race elections district percent gop election candidate house
 Topic#5: planned soldier cities heart stolen city husband christopher view amp
 Topic#6: cheney debate union politics unions local endorsement space black labor
 Topic#7: president war states united years government jobs tax people health
 Topic#8: delay law court texas committee ballot donors investigation records federal
 Topic#9: november electoral account governor polls republicans senate vote poll contact

 Snippet of theta matrix:
 Item#1: 0.054 0.108  0.017  0.282  0.000  0.000  0.528  0.000  0.011
 Item#2: 0.174 0.060  0.686  0.000  0.000  0.000  0.000  0.081  0.000
 Item#3: 0.000 0.000  0.000  0.000  0.117  0.000  0.000  0.000  0.883
```

```
 Item#4: 0.225 0.128  0.058  0.078  0.012  0.455  0.010  0.027  0.008
 Item#5: 0.455 0.145  0.083  0.124  0.009  0.031  0.136  0.017  0.000
 Item#6: 0.455 0.000  0.000  0.518  0.027  0.000  0.000  0.000  0.000
 Item#7: 0.573 0.023  0.341  0.041  0.000  0.000  0.012  0.000  0.010
 Item#8: 0.759 0.000  0.229  0.013  0.000  0.000  0.000  0.000  0.000
 Item#9: 0.258 0.000  0.070  0.453  0.000  0.000  0.218  0.000  0.000
```

## 3.5 Parse collection

The following python script parses `docword.kos.txt` and `vocab.kos.txt` files and converts them into a set of binary-serialized *batches*, stored on disk. In addition the script creates a *dictionary* with all unique tokens in the collection and stored it on disk. The script also detects if it had been already executed, and in this case it just loads the dictionary and save it in *unique_tokens* variable.

The same logic is implemented in a helper-method `ParseCollectionOrLoadDictionary` method.

```
data_folder = sys.argv[1] if (len(sys.argv) >= 2) else ''
target_folder = 'kos'
collection_name = 'kos'

batches_found = len(glob.glob(target_folder + "/*.batch"))
if batches_found == 0:
  print "No batches found, parsing them from textual collection...",
  parser_config = artm.messages_pb2.CollectionParserConfig();
  parser_config.format = artm.library.CollectionParserConfig_Format_BagOfWordsUci

  parser_config.docword_file_path = data_folder + 'docword.'+ collection_name + '.txt'
  parser_config.vocab_file_path = data_folder + 'vocab.'+ collection_name + '.txt'
  parser_config.target_folder = target_folder
  parser_config.dictionary_file_name = 'dictionary'
  unique_tokens = artm.library.Library().ParseCollection(parser_config);
  print " OK."
else:
  print "Found " + str(batches_found) + " batches, using them."
  unique_tokens  = artm.library.Library().LoadDictionary(target_folder + '/dictionary');
```

You may also download larger collections from the following links. You can get the original collection (docword file and vocab file) or an already precompiled batches and dictionary.

| Task | Source | #Words | #Items | class_id(s) | Files |
|---|---|---|---|---|---|
| kos | UCI | 6906 | 3430 | • @default_class | • docword.kos.txt.gz (1 MB)<br>• vocab.kos.txt (54 KB)<br>• kos_1k (700 KB)<br>• kos_dictionary |
| nips | UCI | 12419 | 1500 | • @default_class | • docword.nips.txt.gz (2.1 MB)<br>• vocab.nips.txt (98 KB)<br>• nips_200 (1.5 MB)<br>• nips_dictionary |
| enron | UCI | 28102 | 39861 | • @default_class | • docword.enron.txt.g (11.7 MB)<br>• vocab.enron.txt (230 KB)<br>• enron_1k (7.1 MB)<br>• enron_dictionary |
| nytimes | UCI | 102660 | 300000 | • @default_class | • docword.nytimes.tx (223 MB)<br>• vocab.nytimes.txt (1.2 MB)<br>• nytimes_1k (131 MB)<br>• nytimes_dictionary |
| pubmed | UCI | 141043 | 8200000 | • @default_class | • docword.pubmed.tx (1.7 GB)<br>• vocab.pubmed.txt (1.3 MB)<br>• pubmed_10k (1 GB)<br>• pubmed_dictionary |
| wiki | Gensim | 100000 | 3665223 | • | • enwiki- |

### 3.5. Parse collection

## 3.6 MasterComponent

Master component is you main entry-point to all BigARTM functionality. The following script creates master component and configures it with several regularizers and score calculators.

```
with artm.library.MasterComponent(disk_path = target_folder) as master:
  perplexity_score     = master.CreatePerplexityScore()
  sparsity_theta_score = master.CreateSparsityThetaScore()
  sparsity_phi_score   = master.CreateSparsityPhiScore()
  top_tokens_score     = master.CreateTopTokensScore()
  theta_snippet_score  = master.CreateThetaSnippetScore()

  dirichlet_theta_reg  = master.CreateDirichletThetaRegularizer()
  dirichlet_phi_reg    = master.CreateDirichletPhiRegularizer()
  decorrelator_reg     = master.CreateDecorrelatorPhiRegularizer()
```

Master component must be configured with a disk path, which should contain a set of batches produced in the previous step of this tutorial.

Score calculators allows you to retrieve important quality measures for your topic model. Perplexity, sparsity of theta and phi matrices, lists of tokens with highest probability within each topic are all examples of such scores. By default BigARTM does not calculate any scores, so you have to create in master component. The same is true for regularizers, that allow you to customize your topic model.

For further details about master component refer to *MasterComponentConfig*.

## 3.7 Configure Topic Model

Topic model configuration defins the number of topics in the model, the list of scores to be calculated, and the list of regularizers to apply to the model. For further details about model configuration refer to *ModelConfig*.

```
model = master.CreateModel(topics_count = 10, inner_iterations_count = 10)
model.EnableScore(perplexity_score)
model.EnableScore(sparsity_phi_score)
model.EnableScore(sparsity_theta_score)
model.EnableScore(top_tokens_score)
model.EnableScore(theta_snippet_score)
model.EnableRegularizer(dirichlet_theta_reg, -0.1)
model.EnableRegularizer(dirichlet_phi_reg, -0.2)
model.EnableRegularizer(decorrelator_reg, 1000000)
model.Initialize(unique_tokens)     # Setup initial approximation for Phi matrix.
```

Note that on the last step we configured the initial approximation of Phi matrix. This step is optional — BigARTM is able to collect all tokens dynamically during first scan of the collection. However, a deterministic initial approximation helps to reproduce the same results from run to run.

## 3.8 Invoke Iterations

The following script performs several scans over the set of batches. Depending on the size of the collection this step might be quite time-consuming. It is good idea to output some information after every step.

```
for iter in range(0, 8):
  master.InvokeIteration(1)        # Invoke one scan of the entire collection...
  master.WaitIdle();               # and wait until it completes.
```

```
  model.Synchronize();              # Synchronize topic model.
  print "Iter#" + str(iter),
  print ": Perplexity = %.3f" % perplexity_score.GetValue(model).value,
  print ", Phi sparsity = %.3f" % sparsity_phi_score.GetValue(model).value,
  print ", Theta sparsity = %.3f" % sparsity_theta_score.GetValue(model).value
```

If your collection is very large you may want to utilize online algorithm that updates topic model several times during each iteration, as it is demonstrated by the following script:

```
master.InvokeIteration(1)         # Invoke one scan of the entire collection...
while True:
  done = master.WaitIdle(100)     # wait 100 ms
  model.Synchronize(0.9)          # decay weights in current topic model by 0.9,
  if (done):                      # append all increments and invoke all regularizers.
        break;
```

## 3.9 Retrieve and visualize scores

Finally, you are interested in retrieving and visualizing all collected scores.

```
artm.library.Visualizers.PrintTopTokensScore(top_tokens_score.GetValue(model))
artm.library.Visualizers.PrintThetaSnippetScore(theta_snippet_score.GetValue(model))
```

CHAPTER 4

# Networking

In *Tutorial* you learned how to use BigARTM locally on your machine, within single process. Now we are going to deploy BigARTM on several machines.

## 4.1 Network modus operandi

First think you do to distribute CPU-intensive processing onto several machines is to launch there a `python_interface.NodeController` component.

One simple way to do so is to use `node_controller` application, included in BigARTM distributive.

```
>node_controller.exe

Usage:
    ./node_controller <endpoint>

Example:
    ./node_controller tcp://*:5555

To connect to node_controller from master replace '*' with fully qualified DNS name of the host.
```

After NodeControler is launched, you may add its endpoint to `MasterComponentConfig.node_connect_endpoint` list and then use your `python_interface.MasterComponent` as usual. No further actions is required on the remote nodes.

**Warning:** Some requirements and limitations apply in network mode.

- `MasterComponentConfig.modus_operandi` must be set to `Network`,
- `MasterComponentConfig.disk_path` must be set to a network file share, accessible from all node controllers.
- `MasterComponentConfig.cache_theta` does not work in Network mode.

## 4.2 Proxy to MasterComponent

BigARTM also supports a special `proxy` mode, which allows you to perform experiments on a remote cluster. As before, you start by running `node_controller` executable on your target machines. Then you deploy MasterComponent in one of you target machines.

```
library = ArtmLibrary('artm.dll')

master_proxy_config = messages_pb2.MasterProxyConfig()
master_proxy_config.node_connect_endpoint = 'tcp://192.168.0.2:5555'

with library.CreateMasterComponent(master_proxy_config) as master_proxy:
  # Use master_proxy in the same way you usually use master component
```

Or, if you launched several nodes, you can utilize all of them by configuring your remote MasterComponent to work in Network modus operandi.

```
library = ArtmLibrary('artm.dll')

master_proxy_config = messages_pb2.MasterProxyConfig()
master_proxy_config.node_connect_endpoint = 'tcp://192.168.0.2:5555'
master_proxy_config.config.modus_operandi = MasterComponentConfig_ModusOperandi_Network
master_proxy_config.disk_path = '/fileshare'
master_proxy_config.node_connect_addpoint.append('tcp://192.168.0.3:5555');
master_proxy_config.node_connect_addpoint.append('tcp://192.168.0.4:5555');

with library.CreateMasterComponent(master_proxy_config) as master_proxy:
  # Use master_proxy in the same way you usually use master component
```

## 4.3 Combining network modus operandi with proxy

This python script assumes that you have started local node_controller process as follows:

```
set GLOG_logtostderr=1 & node_controller.exe tcp://*:5000 tcp://*:5556 tcp://*:5557
```

This python script will use ports as follows:

- 5000 - port of the MasterComponent to communicate between MasterComponent and Proxy (this endpoint must be created by node_controller)
- 5550 - port of the MasterComponent to communicate between MasterComponent and Nodes (this endpoint will be automatically created by the master component)
- 5556, 5557 - ports of the NodeControllerComponent to communicate between MasterComponent and Nodes (this endpoint must be created by node_controller)

```
import artm.messages_pb2, artm.library, sys

# Network path of a shared folder with batches to process.
# The folder must be reachable from all remote node controllers.
target_folder = 'D:\\datasets\\nips'

# Dictionary file (must be located on developer's box that runs python script)
dictionary_file = 'D:\\datasets\\nips\\dictionary'

unique_tokens = artm.library.Library().LoadDictionary(dictionary_file)

# Create master component and infer topic model
proxy_config = artm.messages_pb2.MasterProxyConfig()
proxy_config.node_connect_endpoint = 'tcp://localhost:5000'
proxy_config.communication_timeout = 10000  # timeout (in ms) for communication between proxy and mas
proxy_config.polling_frequency  = 50  # polling frequency (in ms) for long-lasting operations, for ex
proxy_config.config.modus_operandi = artm.library.MasterComponentConfig_ModusOperandi_Network
```

```
proxy_config.config.communication_timeout = 2000  # timeout (in ms) for communication between master
proxy_config.config.disk_path = target_folder
proxy_config.config.create_endpoint = 'tcp://*:5550'
proxy_config.config.connect_endpoint = 'tcp://localhost:5550'
proxy_config.config.node_connect_endpoint.append('tcp://localhost:5556')
proxy_config.config.node_connect_endpoint.append('tcp://localhost:5557')
proxy_config.config.processors_count = 1  # number of processors to create at every node

with artm.library.MasterComponent(config = proxy_config) as master:
  dictionary = master.CreateDictionary(unique_tokens)
  perplexity_score = master.CreatePerplexityScore()
  model = master.CreateModel(topics_count = 10, inner_iterations_count = 10)
  model.EnableScore(perplexity_score)
  model.Initialize(dictionary)

  for iter in range(0, 8):
        master.InvokeIteration(1)        # Invoke one scan of the entire collection...
        master.WaitIdle()                # and wait until it completes.
        model.Synchronize()              # Synchronize topic model.
        print "Iter#" + str(iter),
        print ": Perplexity = %.3f" % perplexity_score.GetValue(model).value
```

CHAPTER 5

# BigARTM Stories

## 5.1 Enabling Basic BigARTM Regularizers

This paper describes the experiment with topic model regularization in BigARTM library using experiment02_artm.py. The script provides the possibility to learn topic model with three regularizers (sparsing Phi, sparsing Theta and pairwise topic decorrelation in Phi). It also allows the monitoring of learning process by using quality measures as hold-out perplexity, Phi and Theta sparsity and average topic kernel characteristics.

**Warning:** Note that perplexity estimation can influence the learning process in the online algorithm, so we evaluate perplexity only once per 20 synchronizations to avoid this influence. You can change the frequency using `test_every` variable.

We suggest you to have BigARTM installed in `$YOUR_HOME_DIRECTORY`. To proceed the experiment you need to execute the following steps:

1. Download the collection, represented as BigARTM batches:

   - https://s3-eu-west-1.amazonaws.com/artm/enwiki-20141208_1k.7z
   - https://s3-eu-west-1.amazonaws.com/artm/enwiki-20141208_10k.7z

   This data represents a complete dump of the English Wikipedia (approximately 3.7 million documents). The size of one batch in first version is 1000 documents and 10000 in the second one. We used 10000. The decompressed folder with batches should be put into `$YOUR_HOME_DIRECTORY`. You also need to move there the dictionary file from the batches folder.

   The batch, you'd like to use for hold-out perplexity estimation, also must be placed into `$YOUR_HOME_DIRECTORY`. In our experiment we used the batch named `243af5b8-beab-4332-bb42-61892df5b044.batch`.

2. The next step is the script preparation. Open it's code and find the declaration(-s) of variable(-s)

   - `home_folder` (line 8) and assign it the path `$YOUR_HOME_DIRECTORY`;
   - `batch_size` (line 28) and assign it the chosen size of batch;
   - `batches_disk_path` (line 36) and replace the string 'wiki_10k' with the name of your directory with batches;
   - `test_batch_name` (line 43) and replace the string with direct batch's name with the name of your test batch;
   - `tau_decor`, `tau_phi` and `tau_theta` (lines 57-59) and substitute the values you'd like to use.

3. If you want to estimate the final perplexity on another, larger test sample, put chosen batches into test folder (in `$YOUR_HOME_DIRECTORY` directory). Then find in the code of the script the declaration of variable `save_and_test_model` (line 30) and assign it `True`.

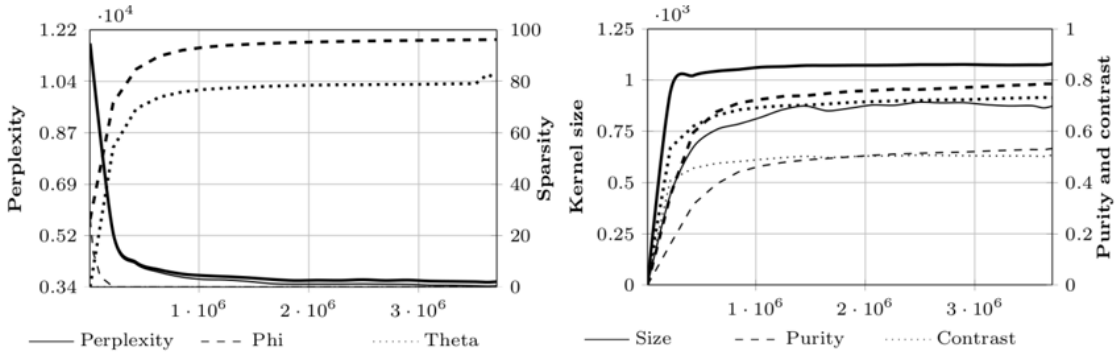
4. After all launch the script. Current measures values will be printed into console. Note, that after synchronizations without perplexity estimation it's value will be replaced with string 'NO'. The results of synchronizations with perplexity estimation in addition will be put in corresponding files in results folder. The file format is general for all measures: the set of strings «(accumulated number of processed documents, measure value)»:

```
(10000, 0.018)
(220000, 0.41)
(430000, 0.456)
(640000, 0.475)
...
```

   These files can be used for plot building.

If desired, you can easy change values of any variable in the code of script since it's sense is clearly commented. If you used all parameters and data identical our experiment you should get the results, close to these ones

| Model/Functional | $\mathcal{P}_{10k}$ | $\mathcal{P}_{100k}$ | $\mathcal{S}_\Phi$ | $\mathcal{S}_\Theta$ | $\mathcal{K}_s$ | $\mathcal{K}_p$ | $\mathcal{K}_c$ |
|---|---|---|---|---|---|---|---|
| LDA | 3436 | 3801 | 0.0 | 0.0 | 873 | 0.533 | 0.507 |
| ARTM | 3577 | 3947 | 96.3 | 80.9 | 1079 | 0.785 | 0.731 |




Here you can see the results of comparison between ARTM and LDA models. To make the experiment with LDA instead of ARTM you only need to change the values of variables tau_decor, tau_phi and tau_theta to 0, 1 / topics_count and 1 / topics_count respectively and run the script again.

**Warning:** Note, that we used machine with 8 cores and 15 Gb RAM for our experiment.

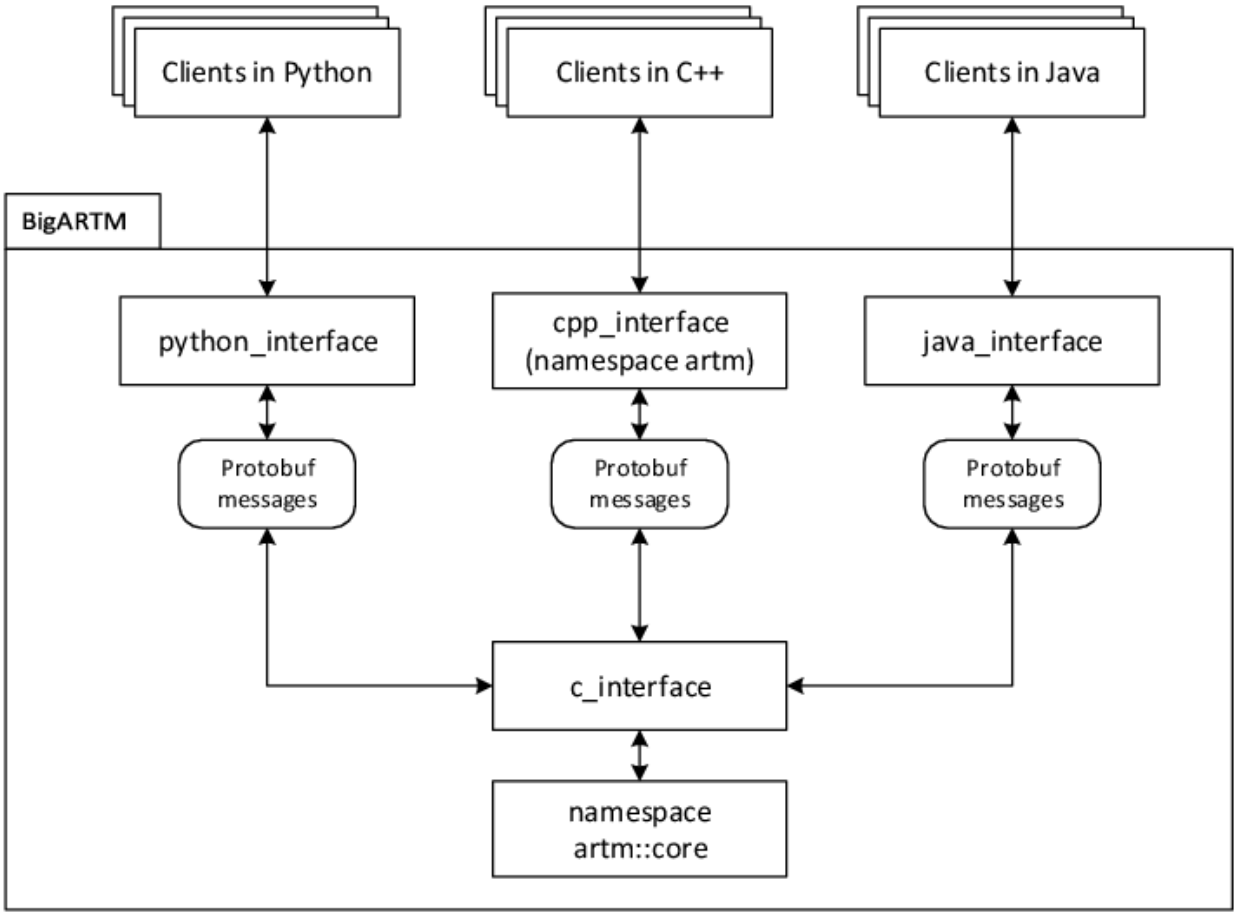
CHAPTER 6

# BigARTM FAQ

## 6.1 Can I use BigARTM from other programming languages (not Python)?

Yes, as long as your language has an implementation of Google Protocol Buffers (the list can be found here). Note that Google officially supports C++, Python and Java.

The following figure shows how to call BigARTM methods directly on `artm.dll` (Windows) or `artm.so` (Linux).




To write your API please refer to *Plain C interface of BigARTM*.

## 6.2 How to retrieve Theta matrix from BigARTM

Theta matrix is a matrix that contains the distribution of several items (columns of the matrix) into topics (rows of the matrix). There are three ways to retrieve such information from BigARTM, and the correct way depends on your scenario.

1. **You want to get Theta matrix for the same collection as you have used to infer the topic model.**

   Set `MasterComponentConfig.cache_theta` to *true* prior to the last iteration, and after the iteration use `MasterComponent::GetThetaMatrix()` (in C++) or `MasterComponent.GetThetaMatrix` (in Python) to retrieve Theta matrix. Note that *cach_theta* flag does not work together with network modus operandi. Current workaround for the network modus operandi is to use the option no. 3 (described below).

2. **You want to repeatedly monitor a small portion of the Theta matrix during ongoing iterations.**

   In this case you should create Theta Snippet score, defined via *ThetaSnippetScoreConfig*, and then use `MasterComponent::GetScoreAs<T>()` to retrieve the resulting *ThetaSnippetScore* message.

   This configuration of Theta Snippet score require you to provide `ThetaSnippetScoreConfig.item_id` listing all IDs of the items that should have Theta's collected. If you created the batches manually you should have specified such IDs in `Item.id` field. If you used other methods to parse the collection from disk then you shouldt try using sequential IDs, starting with 1.

   Remember that Theta snippet score is designed to handle only a small number of items. Attemp to retrieve 100+ items will have a negative effect on performance.

3. **You want to classify a new set of items with an existing model.**

   In this case you need to create a *Batch*, containing your new items. Then copy this batch to `GetThetaMatrixArgs.batch` message, specify `GetThetaMatrixArgs.model_name`, and use `MasterComponent::GetThetaMatrix()` (in C++) or `MasterComponent.GetThetaMatrix` (in Python) to retrieve Theta matrix. In this case there is no need set `MasterComponentConfig.cache_theta` to *true*.

CHAPTER 7

# BigARTM Developer's Guide

This document describes the development process of BigARTM library.

## 7.1 Source code

BigARTM is hosted in public GitHub repository:

https://github.com/bigartm/bigartm

To contribute a fix you should fork the repository, code your fix and submit a pull request. All pull requests are regularly monitored by BigARTM maintainers and will be soon merged into BigARTM's master branch. Please, keep monitoring the status of your pull request on travis, which is a continuous integration system used by BigARTM project.

## 7.2 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_LIBRARY_DIR`.

  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 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`.

## 7.3 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 here: `$(BIGARTM_ROOT)/src/python_tests/python_tests.py`.

- [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`.

## 7.4 Build C++ code on Linux

Simply run CMake on from the root of the project.

The following script had been tested in Ubuntu.

```
sudo apt-get install git make cmake build-essential libboost-all-dev -q -y
git clone https://github.com/bigartm/bigartm
cd ~/bigartm
mkdir build
cd build
cmake ..
make -j8

~/bigartm/build/src/artm_tests/artm_tests
```

It is also possible to use BigARTM from Python on Linux. Just make sure to setup protobuf library as described `$(BIGARTM_ROOT)/3rdparty/protobuf/python/README`, and then you can simply run python scripts under `$(BIGARTM_ROOT)/python_tests/` or `$(BIGARTM_ROOT)/python_client/`.

## 7.5 Compiling .proto files on Windows

1. Open a new command prompt
2. Copy the following two files into `$(BIGARTM_ROOT)/src/`
   - `$(BIGARTM_ROOT)/build/bin/CONFIG/protoc.exe`
   - `$(BIGARTM_ROOT)/build/bin/CONFIG/protoc-gen-cpp_rpcz.exe`

   Here CONFIG can be either Debug or Release (both options will work equally well).
3. Rename `protoc-gen-cpp_rpcz.exe` to `protoc-gen-rpcz_plugin.exe`.
4. Change working directory to `$(BIGARTM_ROOT)/src/`
5. Run the following commands

```
.\protoc.exe --cpp_out=. --python_out=. .\artm\messages.proto
.\protoc.exe --cpp_out=. --rpcz_plugin_out=. .\artm\core\internals.proto
```

## 7.6 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 100 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 100 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=100 <filename>
```

On Windows you may run this master-script to check all required files:

```
$(BIGARTM_ROOT/utils/cpplint_all.bat.
```

CHAPTER 8

# BigARTM Reference

## 8.1 BigARTM: The Algorithm Under The Hood

ToDo: link BigARTM to online batch PLSA algorithm.

ToDo: explain the notation in the algorithm.

ToDo: update the algortihm with regularization.

**Algorithm 1** BigARTM's algorithm

1: Initialize $\phi^0_{wt}$ for all $w \in W$ and $t \in T$;
2: **for all** $i = 1, \dots, I$ **do**
3: $\quad n^i_{wt} := 0, n^i_t := 0$ for all $w \in W$ and $t \in T$;
4: $\quad$ **for all** batches $D_j$, j = 1,...,J **do**
5: $\quad\quad \tilde{n}_{wt} := 0, \tilde{n}_t := 0$ for all $w \in W$ and $t \in T$;
6: $\quad\quad$ **for all** $d \in D_j$ **do**
7: $\quad\quad\quad$ initialize $\theta_{td}$ for all $t \in T$;
8: $\quad\quad\quad$ **repeat**
9: $\quad\quad\quad\quad Z_w := \sum_{t \in T} \phi^{i-1}_{wt} \theta_{td}$ for all $w \in d$;
10: $\quad\quad\quad\quad \theta_{td} := \frac{1}{n_d} \sum_{w \in d} n_{dw} \phi^{i-1}_{wt} \theta_{td} / Z_w$ for all $t \in T$;
11: $\quad\quad\quad$ **until** $\theta_d$ converges;
12: $\quad\quad\quad$ increment $\tilde{n}_{wt}, \tilde{n}_t$ by $n_{dw} \phi^{i-1}_{wt} \theta_{td} / Z_w$ for all $w \in W$ and $t \in T$;
13: $\quad\quad n^i_{wt} := n^i_{wt} + \tilde{n}^i_{wt}$ for all $w \in W$ and $t \in T$;
14: $\quad\quad n^i_t := n^i_t + \tilde{n}_t$ for all $t \in T$;
15: $\quad \phi^i_{wt} := \frac{n^i_{wt}}{n^i_t}$ for all $w \in W$ and $t \in T$;

In this algorithm most CPU resources are consumed on steps 8-11 to infer topic distribution for each document. This operation can be executed concurrently across documents or batches. In BigARTM this parallelization is done across batches to avoid splitting the work into too small junks.

Processing each batch produces counters $tilde n_{wt}$ and $tilde n_{t}$, which should be then merged with the corresponding counters coming from other batches. Since this information is produced by multiple concurrent threads the merging process should be thread-safe and properly synchronised. Our solution is to store all counters $tilde

n_{wt}$ and $tilde n_{t}$ into a single queue, from where they can be picked up by a single *merger thread*. This thread will then accumulate the counters without any locking.

Further in this text the term *outer iteration loop* stands for the loop at the step 2, and the term emph{inner iteration loop} stands for the loop at step 8. Instead of "repeat until it converges" criteria current implementation uses a fixed number of iterations, which is configured manually by the user.

Step 15 is incorporated into all steps that require $phi_{wt}$ (e.g. into steps 9, 10 and 11). These steps utilize counters from the previous iteration ($n^{i-1}_wt$ and $n^{i-1}_t$), which are no longer updated by the merger thread, hence they represent read-only data and can be accessed from multiple threads without any synchronization. At the same time the merger thread will accumulate counters for $n^i_{wt}$ and $n^i_t$ for the current iteration, again in a lock-free manner.

## 8.2 Messages

This document explains all protobuf messages that can be transfered between the user code and BigARTM library.

### 8.2.1 DoubleArray

**class** messages_pb2.**DoubleArray**

Represents an array of double-precision floating point values.

```
message DoubleArray {
  repeated double value = 1 [packed = true];
}
```

### 8.2.2 BoolArray

**class** messages_pb2.**BoolArray**

Represents an array of boolean values.

```
message BoolArray {
  repeated bool value = 1 [packed = true];
}
```

### 8.2.3 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.

### 8.2.4 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;
}
```

### 8.2.5 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;
}
```

`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.

### 8.2.6 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.

### 8.2.7 MasterComponentConfig

**class** messages_pb2.**MasterComponentConfig**

Represents a configuration of a master component.

```
message MasterComponentConfig {
  enum ModusOperandi {
    Local = 0;
    Network = 1;
  }

  optional ModusOperandi modus_operandi = 1 [default = Local];
  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 string create_endpoint = 10;
  optional string connect_endpoint = 11;
  repeated string node_connect_endpoint = 12;
  optional bool online_batch_processing = 13 [default = false];
  optional int32 communication_timeout = 14 [default = 1000];
  optional string disk_cache_path = 15;
}
```

MasterComponentConfig.**modus_operandi**
: A value that defines the modus operandi of the master component.

| | |
|---|---|
| Local | Defines a master component that operates in the local mode. In this mode master component is self-contained, and does not require any external nodes to tune topic model. |
| Network | Defines a master component that operates in the network mode. In this mode master component delegates all heavy processing to externals nodes. The master component is then responsible only for merging the results from external nodes into a single topic model. |

MasterComponentConfig.**disk_path**
: A value that defines the disk location to store or load the collection. In network modus operandi this field is required, and it must point to a network file share, accessible by all nodes connected to the master component.

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'. This feature is currently not supported in network modus operandi.

`MasterComponentConfig.`**`processors_count`**
A value that defines the number of concurrent processor components. In network modus operandi this value defines the processors count at every remote node controller, connected to the master component. 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. In network modus operandi this value defines the maximal queue size at every remote node controller, connected to the master component. Recommendations regarding the maximal queue size are as follows:

- the queue size should be at least as large as the number of concurrent processors;
- the total size of the queues across all node controllers should not exceed the number of batches in the collection.

`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.`**`create_endpoint`**
A value that defines ZeroMQ endpoint to expose the master component service (example: `tcp://*:5555`). For further details about the format of endpoint string refer to ZeroMQ documentation (http://api.zeromq.org). The value is used only in the network modus operandi.

`MasterComponentConfig.`**`connect_endpoint`**
A value that defines ZeroMQ endpoint to expose the master component service (example: `tcp://192.168.0.1:5555`). // For further details about the format of endpoint string refer to ZeroMQ documentation (http://api.zeromq.org). The value is used only in the network modus operandi.

`MasterComponentConfig.`**`node_connect_endpoint`**
A set containing all ZeroMQ endpoints of the external node controllers (example: `tcp://192.168.0.2:5556`). For further details about the format of endpoint string refer to ZeroMQ documentation (http://api.zeromq.org). The value is used only in the network modus operandi. A node controller component at the remote machine must be created prior to configuring master component with its endpoint.

`MasterComponentConfig.`**`online_batch_processing`**
A flag indicating whether to enable online batch processing. This mode imply that all batches added with `ArtmAddBatch()` will be automatically processed, without explicit call to `ArtmInvokeIteration()`. The `ArtmInvokeIteration()` must not be used together with online batch processing mode. Note that online batch processing is currently not allowed together with `cache_theta`.

`MasterComponentConfig.`**`communication_timeout`**
An communication timeout in milliseconds. Any remote network call that exceeds communication timeout will result in `ARTM_NETWORK_ERROR` error.

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.

### 8.2.8 NodeControllerConfig

**class** messages_pb2.**NodeControllerConfig**

Represents a configuration of a NodeController

```
message NodeControllerConfig {
  optional string create_endpoint = 1;
}
```

NodeControllerConfig.**create_endpoint**
A value that defines ZeroMQ endpoint to expose the node component service (example: tcp://*:5556). For further details about the format of endpoint string refer to ZeroMQ documentation (http://api.zeromq.org).

### 8.2.9 MasterProxyConfig

**class** messages_pb2.**MasterProxyConfig**

Represents a configuration of a proxy to MasterComponent. The purpose of the proxy is to operate MasterComponent from a remote machine. There is no requirement to run MasterComponent and its proxy in the same operating system (MasterComponent can run on linux while the proxy can be on Windows). Any type of MasterComponent (either in local or in network modus operandi) can be operated by a proxy.

```
message MasterProxyConfig {
  optional string node_connect_endpoint = 1;
  optional MasterComponentConfig config = 2;
  optional int32 communication_timeout = 3 [default = 1000];
  optional int32 polling_frequency = 4 [default = 50];
}
```

MasterProxyConfig.**node_connect_endpoint**
A value that defines ZeroMQ endpoint of an external node controller. (example: tcp://192.168.0.2:5556). For further details about the format of endpoint string refer to ZeroMQ documentation (http://api.zeromq.org). A node controller component at the remote machine must be created prior to configuring master component with its endpoint.

MasterProxyConfig.**config**
A message that defines MasterComponent configuration of a remote node.

MasterProxyConfig.**communication_timeout**
An communication timeout in milliseconds. Any remote network call that exceeds communication timeout will result in ARTM_NETWORK_ERROR error.

MasterProxyConfig.**polling_frequency**
Defines the frequency that the proxy object uses to repeatedly pool remote master component for a status.

When ArtmWaitIdle() on the remote master component reports ARTM_STILL_WORKING, the proxy object will retry the request within specified pooling frequency.

### 8.2.10 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"];
  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];
}
```

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**
: A value that defines which field of an item the model should use.

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.

### 8.2.11 RegularizerConfig

**class** messages_pb2.**RegularizerConfig**

Represents a configuration of a general regularizer.

```
message RegularizerConfig {
  enum Type {
    SmoothSparseTheta = 0;
    SmoothSparsePhi = 1;
    DecorrelatorPhi = 2;
  }

  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 |

RegularizerConfig.**config**
: A serialized protobuf message that describes regularizer config for the specific regularizer type.

### 8.2.12 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.

### 8.2.13 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.

### 8.2.14 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.

### 8.2.15 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;
}
```

### 8.2.16 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.

### 8.2.17 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.

### 8.2.18 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.

### 8.2.19 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.

### 8.2.20 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"];
  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**
: A value that defines which field of an item should be used in perplexity calculation.

PerplexityScoreConfig.**stream_name**
: A value that defines which stream should be used in perplexity calculation.

### 8.2.21 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(w|t,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.

### 8.2.22 SparsityThetaScoreConfig

**class** messages_pb2.**SparsityThetaScoreConfig**

Represents a configuration of a theta sparsity score.

```
message SparsityThetaScoreConfig {
  optional string field_name = 1 [default = "@body"];
  optional string stream_name = 2 [default = "@global"];
  optional float eps = 3 [default = 1e-37];
  repeated string topic_name = 4;
}
```

SparsityThetaScoreConfig.**field_name**
: A value that defines which field of an item should be used in theta sparsity calculation.

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.

### 8.2.23 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.

### 8.2.24 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.

### 8.2.25 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.

### 8.2.26 ItemsProcessedScoreConfig

**class** `messages_pb2.`**`ItemsProcessedScoreConfig`**

Represents a configuration of an items processed score.

```
message ItemsProcessedScoreConfig {
  optional string field_name = 1 [default = "@body"];
  optional string stream_name = 2 [default = "@global"];
}
```

`ItemsProcessedScoreConfig.`**`field_name`**
: A value that defines which field of an item should be used in calculation of processed items.

`ItemsProcessedScoreConfig.`**`stream_name`**
: A value that defines which stream should be used in calculation of processed items.

### 8.2.27 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 have the field ItemsProcessedScoreConfig.field_name and belong to the stream ItemsProcessedScoreConfig.stream_name and have been processed during iterations. Currently this number is aggregated throughout all iterations.

### 8.2.28 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.

### 8.2.29 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.

### 8.2.30 ThetaSnippetScoreConfig

**class** messages_pb2.**ThetaSnippetScoreConfig**

Represents a configuration of a theta snippet score.

```
message ThetaSnippetScoreConfig {
  optional string field_name = 1 [default = "@body"];
  optional string stream_name = 2 [default = "@global"];
  repeated int32 item_id = 3 [packed = true];
}
```

ThetaSnippetScoreConfig.**field_name**
: A value that defines which field of an item should be used in calculation of a theta snippet.

ThetaSnippetScoreConfig.**stream_name**
: A value that defines which stream should be used in calculation of a theta snippet.

ThetaSnippetScoreConfig.**item_id**
: A set of values that define items for which theta snippet should be calculated. Items are identified by the item id.

### 8.2.31 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.

### 8.2.32 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).

### 8.2.33 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.

### 8.2.34 TopicModel

**class** `messages_pb2.`**`TopicModel`**

Represents a topic model.

```
message TopicModel {
  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;
}
```

`TopicModel.`**`name`**
: A value that describes the name of the topic model. This name will match the name of the corresponding model config.

`TopicModel.`**`topics_count`**
: A value that describes the number of topics in the topic model. This value will match `ModelConfig.topics_count` value, defined in the model config.

`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 'token'. The length of each FloatArray will match the topics_count field.

`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 'token'.

`TopicModel.`**`internals`**
: A serialized instance of TopicModelInternals message.

### 8.2.35 ThetaMatrix

**class** `messages_pb2.`**`ThetaMatrix`**

Represents a theta matrix.

```
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;
}
```

`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 'item_id'. The length of each FloatArray will match the number of topics in the model.

`ThetaMatrix.`**`topic_name`**
: A set of values that represent the names of the topics, included in this theta matrix. The names correspond to `ModelConfig.topic_name`.

`TopicModel.`**`topics_count`**
: A value that describes the number of topics in the topic model. This value will match `ModelConfig.topics_count` value, defined in the model config.

`ThetaMatrix.`**`item_id`**
: A set of item titles, corresponding to `Item.title` values.

### 8.2.36 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;
}
```

`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:<br><br>D<br>W<br>NNZ<br>docID wordID count<br>docID wordID count<br>...<br>docID wordID count<br><br>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:<br><br>token1 @default_class<br>token2 custom_class<br>token3 @default_class<br>token4<br><br>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`.

### 8.2.37 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 = 1.0];
  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.

### 8.2.38 InitializeModelArgs

**class** messages_pb2.**InitializeModelArgs**

Represents an argument of initialize model operation.

```
message InitializeModelArgs {
  optional string model_name = 1;
  optional string dictionary_name = 2;
}
```

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.

### 8.2.39 GetTopicModelArgs

Represents an argument of get topic model operation.

```
message GetTopicModelArgs {
  optional string model_name = 1;
  repeated string topic_name = 2;
  repeated string token = 3;
  repeated string class_id = 4;
}
```

`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.

### 8.2.40 GetThetaMatrixArgs

Represents an argument of get theta matrix operation.

```
message GetThetaMatrixArgs {
  optional string model_name = 1;
  optional Batch batch = 2;
  repeated string topic_name = 3;
  repeated int32 topic_index = 4;
}
```

`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*.

### 8.2.41 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()`.

### 8.2.42 AddBatchArgs

Represents an argument of `ArtmAddBatch()` operation.

```
message AddBatchArgs {
  optional Batch batch = 1;
  optional int32 timeout_milliseconds = 2;
}
```

`AddBatchArgs.`**`batch`**
: The *Batch* to add.

`AddBatchArgs.`**`timeout_milliseconds`**
: Timeout in milliseconds for this operation.

### 8.2.43 InvokeIterationArgs

Represents an argument of `ArtmInvokeIteration()` operation.

```
message InvokeIterationArgs {
  optional int32 iterations_count = 1 [default = 1];
}
```

`InvokeIterationArgs.`**`iterations_count`**
: An integer value describing how many iterations to invoke.

### 8.2.44 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.

## 8.3 Python Interface

This document explains all classes in python interface of BigARTM library.

### 8.3.1 Library

**class** `artm.library.`**Library**(*artm_shared_library* = *""*)
Creates an ArtmLibrary object, wrapping the BigARTM shared library.

The *artm_shared_library* is an optional argument, which provides full file name of artm shared library (a disk path plus `artm.dll` on Windows or `artm.so` on Linux). When *artm_shared_library* is not specified the shared library will be searched in folders listed in `PATH` system variable. You may also configure `ARTM_SHARED_LIBRARY` system variable to provide full file name of artm shared library.

**CreateMasterComponent**(*config* = *messages_pb2.MasterComponentConfig()*)
Creates and returns an instance of `MasterComponent` class. *config* defines an optional *MasterComponentConfig* parameter that may carry the configuration of the master component.

**CreateNodeController**(*endpoint*)
Creates a node controller at a specific endpoint.

Returns an instance of `NodeController` class.

**SaveBatch**(*batch*, *disk_path*)
Saves a given *Batch* into a disk location specified by *disk_path*.

**ParseCollection**(*collection_parser_config*)
Parses a text collection as defined by *collection_parser_config* (*CollectionParserConfig*). Returns an instance of *DictionaryConfig* which carry all unique words in the collection and their frequencies.

For more information refer to `ArtmRequestParseCollection()` and `ArtmRequestLoadDictionary()`.

**LoadDictionary**(*full_filename*)
Loads a *DictionaryConfig* from the file, defined by *full_filename* argument.

For more information refer to `ArtmRequestLoadDictionary()`.

**LoadBatch**(*full_filename*)
Loads a *Batch* from the file, defined by *full_filename* argument.

For more information refer to `ArtmRequestLoadBatch()`.

**ParseCollectionOrLoadDictionary**(*docword_file_path*, *vocab_file_path*, *target_folder*)
A simple helper method that runs `ParseCollection()` when *target_folder* is empty, otherwise tried to use `LoadDictionary()` to load the dictionary from *target_folder*.

The *docword_file_path* and *vocab_file_path* arguments should provide the disk location of docword and vocab files of the collection to be parsed.

### 8.3.2 MasterComponent

**class** `artm.library.`**MasterComponent**(*config* = *messages_pb2.MasterComponentConfig()*, *lib* = *None*, *disk_path* = *None*, *proxy_endpoint* = *None*)
Creates a master component.

*config* is an optional instance of *MasterComponentConfig*, providing an initial configuration of the master component.

*lib* is an optional argument pointing to `Library`. When not specified, a default library will be used. Check the constructor of `Library` for more details.

*disk_path* is an optional value providing the disk folder with batches to process by this master component. If *disk_path* is not specified, MasterComponent will go for in-memory mode. In this mode you may add data with `AddBatch()` method, and process as usual. Note that changing *disk_path* is not supported (you must recreate a new instance MasterComponent to do so).

*proxy_endpoint* is an optional string value that provides connect endpoint of a remote node controller. When specified, the master component will operate in a proxy mode (that is, it will redirect all commands to the remote master component instantiated in the remote node controller).

**Dispose**()
Disposes the master component and releases all unmanaged resources (like memory, network connections, etc).

**config**()
Returns current *MasterComponentConfig* of the master component.

**CreateModel**(*config = messages_pb2.ModelConfig()*, *topics_count = None*, *inner_iterations_count = None*)
Creates and returns an instance of `Model` class based on a given *ModelConfig*. Note that the model has to be further tuned by several iterative scans over the text collection. Use `InvokeIteration()` to perform such scans.

Parameters *topics_count* and *inner_iterations_count* will override values, specified in *config*.

**RemoveModel**(*model*)
Removes an instance of `Model` from the master component. After this operation the *model* object became invalid and must not be used.

**CreateRegularizer**(*name*, *type*, *config*)
Creates and returns an instance of `Regularizer` component. *name* can be any unique identifier, that you can further use to identify regularizer (for example, in `ModelConfig.regularizer_name`). *type* can be any regularizer type (for example, the `RegularizerConfig_Type_DirichletTheta`). *config* can be any regularizer config (for example, a *SmoothSparseThetaConfig*).

**CreateDirichletThetaRegularizer**(*name = None*, *config = messages_pb2.DirichletThetaConfig()*)
Creates an instance of DirichletThetaRegularizer.

**CreateDirichletPhiRegularizer**(*name = None*, *config = messages_pb2.DirichletPhiConfig()*)
Creates an instance of DirichletPhiRegularizer.

**CreateSmoothSparseThetaRegularizer**(*name = None*, *config = messages_pb2.SmoothSparseThetaConfig()*)
Creates an instance of SmoothSparseThetaRegularizer.

**CreateSmoothSparsePhiRegularizer**(*name = None*, *config = messages_pb2.SmoothSparsePhiConfig()*)
Creates an instance of SmoothSparsePhiRegularizer.

**CreateDecorrelatorPhiRegularizer**(*name = None*, *config = messages_pb2.DecorrelatorPhiConfig()*)
Creates an instance of DecorrelatorPhiRegularizer.

**RemoveRegularizer**(*regularizer*)
Removes an instance of `Regularizer` from the master component. After this operation the *regularizer* object became invalid and must not be used.

**CreateScore**(*name*, *type*, *config*)
Creates a score calculator inside the master component. *name* can be any unique identifier, that you

can further use to identify the score (for example, in `ModelConfig.score_name`). *type* can be any score type (for example, the `ScoreConfig_Type_Perplexity`). *config* can be any score config (for example, a *PerplexityScoreConfig*).

**CreatePerplexityScore**(*self*, *name = None*, *config = messages_pb2.PerplexityScoreConfig()*, *stream_name = None*)
Creates an instance of PerplexityScore.

**CreateSparsityThetaScore**(*self*, *name = None*, *config = messages_pb2.SparsityThetaScoreConfig()*)
Creates an instance of SparsityThetaScore.

**CreateSparsityPhiScore**(*self*, *name = None*, *config = messages_pb2.SparsityPhiScoreConfig()*)
Creates an instance of SparsityPhiScore.

**CreateItemsProcessedScore**(*self*, *name = None*, *config = messages_pb2.ItemsProcessedScoreConfig()*)
Creates an instance of ItemsProcessedScore.

**CreateTopTokensScore**(*self*, *name = None*, *config = messages_pb2.TopTokensScoreConfig()*, *num_tokens = None*, *class_id = None*)
Creates an instance of TopTokensScore.

**CreateThetaSnippetScore**(*self*, *name = None*, *config = messages_pb2.ThetaSnippetScoreConfig()*)
Creates an instance of ThetaSnippetScore.

**CreateTopicKernelScore**(*self*, *name = None*, *config = messages_pb2.TopicKernelScoreConfig()*)
Creates an instance of TopicKernelScore.

**RemoveScore**(*name*)
Removes a score calculator with the specific name from the master component.

**CreateDictionary**(*config*)
Creates and returns an instance of `Dictionary` class component with a specific *DictionaryConfig*.

**RemoveDictionary**(*dictionary*)
Removes an instance of `Dictionary` from the master component. After this operation the *dictionary* object became invalid and must not be used.

**Reconfigure**(*config = None*)
Updates the configuration of the master component with new *MasterComponentConfig* value, provided by *config* parameter. Remember that some changes of the configuration are not allowed (for example, the `MasterComponentConfig.disk_path` must not change). Such configuration parameters must be provided in the constructor of `MasterComponent`.

**AddBatch**(*batch*)
Adds an instance of *Batch* class to the master component. This method is only used for in-memory processing of the collection, and require an empty value of `MasterComponentConfig.disk_path` in current configuration. Master component creates a copy of the *batch*, so any further changes of the *batch* object will not be picked up.

The behavior of this method also depends on `MasterComponentConfig.online_batch_processing` flag.

**InvokeIteration**(*iterations_count = 1*)
Invokes several iterations over the collection. The recommended value for *iterations_count* is 1. For more iterations use for loop around `InvokeIteration()` method. This operation is asynchronous. Use `WaitIdle()` to await until all iterations succeeded.

**WaitIdle**(*timeout = -1*)
Awaits for ongoing iterations. Returns *True* if iterations had been finished within the timeout, other-

wise returns *False*. The provided timeout is in milliseconds. Use *timeout = -1* to allow infinite time for `WaitIdle()` operation. Remember to call `Model.Synchronize()` operation to synchronize each model that you are currently processing.

**CreateStream**(*stream*)
Creates a data stream base on the *stream* (*Stream*).

**RemoveStream**(*stream_name*)
Removes a stream with the specific name from the master component.

**GetTopicModel**(*model = None*, *args = messages_pb2.GetTopicModelArgs()*)
Retrieves and returns an instance of *TopicModel* class, carrying all the data of the topic model (including the Phi matrix). Parameter *model* should be an instance of `Model` class. For more settings use *args* parameter (see *GetTopicModelArgs* for all available options).

**GetRegularizerState**(*regularizer_name*)
Retrieves and returns the internal state of a regularizer with the specific name.

**GetThetaMatrix**(*model = None*, *batch = None*, *args = messages_pb2.GetThetaMatrixArgs()*)
Retrieves an instance of *ThetaMatrix* class. The content depends on *batch* parameter. When *batch* is provided, the resulting *ThetaMatrix* will contain theta values estimated for all documents in the batch. When *batch* is not provided, the resulting *ThetaMatrix* will contain theta values gathered during the last iteration.

Parameter *model* should be an instance of `Model` class. For more settings use *args* parameter (see *GetThetaMatrixArgs* for all available options).

When used without *batch*, this operation require `MasterComponentConfig.cache_theta` to be set to *True* before starting the last iteration. In this case the entire ThetaMatrix must fit into CPU memory, and for this reason `MasterComponentConfig.cache_theta` is turned off by default.

This operation is not supported when `MasterComponentConfig.modus_operandi` is set to *Network*.

### 8.3.3 NodeController

**class** `artm.library.`**NodeController**(*endpoint*, *lib = None*)
Creates a node controller on a specific endpoint. See *NodeControllerConfig* for more details.

*lib* is an optional argument pointing to `Library`. When not specified, a default library will be used. Check the constructor of `Library` for more details.

**Dispose**()
Disposes the node controller and releases all unmanaged resources (like memory, network connections, etc).

### 8.3.4 Model

**class** `artm.library.`**Model**
This constructor must not be used explicitly. The only correct way of creating a Model is through `MasterComponent.CreateModel()` method.

**name**()
Returns the string name of the model.

**Reconfigure**(*config = None*)
Updates the configuration of the topic model with new *ModelConfig* value, provided by *config* parameter. When *config* is not specified the configuration is updated with `config()` value. Remember that

some changes of the configuration are applied immediately after this call. For example, changes to `ModelConfig.topics_count` or `ModelConfig.topic_name` will be applied only during the next `Synchronize` call.

Note that changes `ModelConfig.topics_count` or `ModelConfig.topic_name` are only supported on an idle master component (e.g. in between iterations). Changing these values during an ongoing iteration may cause unexpected results.

**topics_count**()
Returns the number of topics in the model.

**config**()
Returns current *ModelConfig* of the topic model.

**Synchronize**(*decay_weight = 0.0*, *apply_weight = 1.0*, *invoke_regularizers = True*)
This operation updates the Phi matrix of the topic model with all model increments, collected since the last call to `Synchronize()` method. The Phi matrix is calculated according to *decay_weight* and *apply_weight* (refer to `SynchronizeModelArgs.decay_weight` for more details). Depending on *invoke_regularizers* parameter this operation may also invoke all regularizers.

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

**Initialize**(*tokens*, *dictionary*)
Generates a random initial approximation for the Phi matrix of the topic model.

*dictionary* must be an instance of `Dictionary` class.

**Overwrite**(*topic_model*, *commit = True*)
Updates the model with new Phi matrix, defined by *topic_model* (*TopicModel*). 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 `Model.Synchronize()` to propagate your change.

**Enable**()
Sets `ModelConfig.enabled` to *True* for the current topic model. This means that the model will be updated on `MasterComponent.InvokeIteration()`.

**EnableScore**(*score*)
By default model does calculate any scores even if they are created with `MasterComponent.CreateScore()`. Method *EnableScore* tells to the model that *score* should be applied to the model. Parameter *tau* defines the regularization coefficient of the regularizer. *score* must be an instance of `Score` class.

**EnableRegularizer**(*regularizer*, *tau*)
By default model does not use any regularizers even if they are created with `MasterComponent.CreateRegularizer()`. Method *EnableRegularizer* tells to the model that *regularizer* should be applied to the model. Parameter *tau* defines the regularization coefficient of the regularizer. *regularizer* must be an instance of `Regularizer` class.

**Disable**()
Sets `ModelConfig.enabled` to *False'* for the current topic model. This means that the model will not be updated on `MasterComponent.InvokeIteration()`, but the the scores for the model still will be collected.

### 8.3.5 Regularizer

**class** artm.library.**Regularizer**

This constructor must not be used explicitly. The only correct way of creating a Regularizer is through MasterComponent.CreateRegularizer() method (or similar methods in MasterComponent class, dedicated to a particular type of the regularizer).

**name**()

Returns the string name of the regularizer.

**Reconfigure**(*type*, *config*)

Updates the configuration of the regularizer with new regularizer configuration, provided by *config* parameter. The *config* object can be, for example, of *SmoothSparseThetaConfig* type (or similar). The type must match the current type of the regularizer.

### 8.3.6 Score

**class** artm.library.**Score**

This constructor must not be used explicitly. The only correct way of creating a Score is through MasterComponent.CreateScore() method (or similar methods in MasterComponent class, dedicated to a particular type of the score).

**name**()

Returns the string name of the score.

**GetValue**(*model = None*, *batch = None*)

Retrieves the score for a specific model. For cumulative scores such as Perplexity of ThetaSparsity score it is possible to use *batch* argument.

### 8.3.7 Dictionary

**class** artm.library.**Dictionary**(*master_component*, *config*)

This constructor must not be used explicitly. The only correct way of creating a Dictionary is through MasterComponent.CreateDictionary() method.

**name**()

Returns the string name of the dictionary.

**Reconfigure**(*config*)

Updates the configuration of the dictionary with new *DictionaryConfig* value, provided by *config* parameter.

### 8.3.8 Visualizers

**class** artm.library.**Visualizers**

This class provides a set of static method to visualize some scores.

**PrintTopTokensScore**(*top_tokens_score*)

Prints the TopTokensScore.

**PrintThetaSnippetScore**(*theta_snippet_score*)

Prints the ThetaSnippetScore.

### 8.3.9 Exceptions

**exception** artm.library.**InternalError**
An exception class corresponding to ARTM_INTERNAL_ERROR error code.

**exception** artm.library.**ArgumentOutOfRangeException**
An exception class corresponding to ARTM_ARGUMENT_OUT_OF_RANGE error code.

**exception** artm.library.**InvalidMasterIdException**
An exception class corresponding to ARTM_INVALID_MASTER_ID error code.

**exception** artm.library.**CorruptedMessageException**
An exception class corresponding to ARTM_CORRUPTED_MESSAGE error code.

**exception** artm.library.**InvalidOperationException**
An exception class corresponding to ARTM_INVALID_OPERATION error code.

**exception** artm.library.**DiskReadException**
An exception class corresponding to ARTM_DISK_READ_ERROR error code.

**exception** artm.library.**DiskWriteException**
An exception class corresponding to ARTM_DISK_WRITE_ERROR error code.

**exception** artm.library.**NetworkException**
An exception class corresponding to ARTM_NETWORK_ERROR error code.

### 8.3.10 Constants

artm.library.**Stream_Type_Global**

artm.library.**Stream_Type_ItemIdModulus**

artm.library.**RegularizerConfig_Type_DirichletTheta**

artm.library.**RegularizerConfig_Type_DirichletPhi**

artm.library.**RegularizerConfig_Type_SmoothSparseTheta**

artm.library.**RegularizerConfig_Type_SmoothSparsePhi**

artm.library.**RegularizerConfig_Type_DecorrelatorPhi**

artm.library.**ScoreConfig_Type_Perplexity**

artm.library.**ScoreData_Type_Perplexity**

artm.library.**ScoreConfig_Type_SparsityTheta**

artm.library.**ScoreData_Type_SparsityTheta**

artm.library.**ScoreConfig_Type_SparsityPhi**

artm.library.**ScoreData_Type_SparsityPhi**

artm.library.**ScoreConfig_Type_ItemsProcessed**

artm.library.**ScoreData_Type_ItemsProcessed**

artm.library.**ScoreConfig_Type_TopTokens**

artm.library.**ScoreData_Type_TopTokens**

artm.library.**ScoreConfig_Type_ThetaSnippet**

artm.library.**ScoreData_Type_ThetaSnippet**

artm.library.**ScoreConfig_Type_TopicKernel**

artm.library.**ScoreData_Type_TopicKernel**

artm.library.**PerplexityScoreConfig_Type_UnigramDocumentModel**

artm.library.**PerplexityScoreConfig_Type_UnigramCollectionModel**

artm.library.**CollectionParserConfig_Format_BagOfWordsUci**

artm.library.**MasterComponentConfig_ModusOperandi_Local**

artm.library.**MasterComponentConfig_ModusOperandi_Network**

## 8.4 Plain C interface of BigARTM

This document explains all public methods of the low level BigARTM interface.

### 8.4.1 Introduction

The goal of low level BigARTM interface is to expose all functionality of the library in a set of simple functions written in good old 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.

Note that most methods in this API accept a serialized binary representation of some Google Protocol Buffer message. Please, refer to *Messages* for more details about each particular message.

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, `ArtmCreateMasterComponent()` returns the ID of newly created master component, and `ArtmRequestTopicModel()` returns the length of the buffer that should be allocated before calling `ArtmCopyRequestResult()`.

### 8.4.2 ArtmCreateMasterComponent

int **ArtmCreateMasterComponent**(int *length*, const char* *master_component_config*)

Creates a master component.

**Parameters**

- **master_component_config** (*const_char**) – Serialized *MasterComponentConfig* message, describing the configuration of the master component.
- **length** (*int*) – The length in bytes of the *master_component_config* message.

**Returns** In case of success, a non-negative ID of the master component, otherwise one of the *error codes*.

The ID, returned by this operation, is required by most methods in this API. Several master components may coexist in the same process. In such case any two master components with different IDs can not share any common data, and thus they are completely independent from each other.

### 8.4.3 ArtmCreateMasterProxy

int **ArtmCreateMasterProxy**(int *length*, const char* *master_proxy_config*)

Creates a proxy-object, connected to a remote master component.

**Parameters**

- **master_proxy_config** (*const_char**) – Serialized *MasterProxyConfig* message, describing the configuration of the master proxy.
- **length** (*int*) – The length in bytes of the *master_proxy_config* message.

**Returns** In case of success, a non-negative ID of the proxy object, otherwise one of the *error codes*.

Prior to calling this method you must ensure that the remote host is running an instance of Node Controller component. The endpoint of the remote Node Controller should be specified in the *master_proxy_config* message.

The effect of this method is as follows:

•a new Master Component is started on the remote Node Controller,

•a proxy object to the remote Master Component is created withing current process.

The ID, returned by this operation, can be used in all methods that require *master_id* parameter. All API calls will be redirected to the remote Master Component.

### 8.4.4 ArtmReconfigureMasterComponent

int **ArtmReconfigureMasterComponent**(int *master_id*, int *length*, const char* *master_component_config*)

Changes the configuration of the master component.

**Parameters**

- **master_id** (*int*) – The ID of a master component returned by either `ArtmCreateMasterComponent()` or `ArtmCreateMasterProxy()` method.
- **master_component_config** (*const_char**) – Serialized *MasterComponentConfig* message, describing the new configuration of the master component.
- **length** (*int*) – The length in bytes of the *master_component_config* message.

**Returns** A zero value if operation succeeded, otherwise one of the *error codes*.

### 8.4.5 ArtmDisposeMasterComponent

int **ArtmDisposeMasterComponent**(int *master_id*)

Disposes master component together with all its models, regularizers and dictionaries.

**Parameters**

- **master_id** (*int*) – The ID of a master component or a master proxy, returned by either `ArtmCreateMasterComponent()` or `ArtmCreateMasterProxy()` method. If *master_id* represents a master proxy, then this operation will dispose both remote master component and local proxy object.

**Returns** This operation always returns `ARTM_SUCCESS`.

This operation releases memory and other unmanaged resources, used by the master component. If the master component had been launched in the network modus operandi, then this operation also releases resources on

the remote nodes and closes all network connections. If the *master_id* parameter represents a proxy-object to a remote master component, then both the remote master component and the local proxy are disposed.

After this operation the *master_id* value becames invalid and must not be used in other operations.

### 8.4.6 ArtmCreateNodeController

int **ArtmCreateNodeController**(int *length*, const char* *node_controller_config*)
Creates a node controller component.

**Parameters**

- **node_controller_config** (*const_char**) – Serialized *NodeControllerConfig* message, describing the configuration of the node controller.
- **length** (*int*) – The length in bytes of the *node_controller_config* message.

**Returns** Returns `ARTM_SUCCESS` value if operation succeeded, otherwise returns one of the *error codes*.

### 8.4.7 ArtmDisposeNodeController

int **ArtmDisposeNodeController**(int *node_controller_id*)
Disposes node controller and releases all unmanaged resources.

**Parameters**

- **master_id** (*int*) – The ID of a node controller, returned by `ArtmCreateNodeController()` method.

**Returns** This operation always returns `ARTM_SUCCESS`.

### 8.4.8 ArtmCreateModel

int **ArtmCreateModel**(int *master_id*, int *length*, const char* *model_config*)
Defines a new topic model.

**Parameters**

- **master_id** (*int*) – The ID of a master component or a master proxy, returned by either `ArtmCreateMasterComponent()` or `ArtmCreateMasterProxy()` method.
- **model_config** (*const_char**) – Serialized *ModelConfig* message, describing the configuration of the topic model.
- **length** (*int*) – The length in bytes of the *model_config* message.

**Returns** Returns `ARTM_SUCCESS` value if operation succeeded, otherwise returns one of the *error codes*.

Note that this method only defines the configuration a topic model, but does not tune it. Use `ArtmInvokeIteration()` method to process the collection of textual documents, and then `ArtmRequestTopicModel()` to retrieve the resulting topic model.

It is important to notice that *model_config* must have a unique value in the `ModelConfig.name` field, that can be further used to identify the model (for example in `ArtmRequestTopicModel()` call).

### 8.4.9 ArtmReconfigureModel

int **ArtmReconfigureModel**(int *master_id*, int *length*, const char* *model_config*)
Updates the configuration of topic model.

**Parameters**

- **master_id** (*int*) – The ID of a master component or a master proxy, returned by either ArtmCreateMasterComponent() or ArtmCreateMasterProxy() method.
- **model_config** (*const_char**) – Serialized *ModelConfig* message, describing the new configuration of the topic model.
- **length** (*int*) – The length in bytes of the *model_config* message.

**Returns** Returns ARTM_SUCCESS value if operation succeeded, otherwise returns one of the *error codes*.

### 8.4.10 ArtmDisposeModel

int **ArtmDisposeModel**(int *master_id*, const char* *model_name*)
Explicitly delete a specific topic model. All regularizers within specific master component are also deleted automatically by ArtmDisposeMasterComponent().

After ArtmDisposeModel() the *model_name* became invalid and shell not be used in ArtmRequestScore(), ArtmRequestTopicModel(), ArtmRequestThetaMatrix() or any other method (or protobuf message) that require *model_name*.

**Parameters**

- **master_id** (*int*) – The ID of a master component or a master proxy, returned by either ArtmCreateMasterComponent() or ArtmCreateMasterProxy() method.
- **model_name** (*const_char**) – A string identified of the model that should be deleted.

**Returns** Returns ARTM_SUCCESS value if operation succeeded, otherwise returns one of the *error codes*.

### 8.4.11 ArtmCreateRegularizer

int **ArtmCreateRegularizer**(int *master_id*, int *length*, const char* *regularizer_config*)
Creates a new regularizer.

**Parameters**

- **master_id** (*int*) – The ID of a master component or a master proxy, returned by either ArtmCreateMasterComponent() or ArtmCreateMasterProxy() method.
- **regularizer_config** (*const_char**) – Serialized *RegularizerConfig* message, describing the configuration of a new regularizer.
- **length** (*int*) – The length in bytes of the *regularizer_config* message.

**Returns** Returns ARTM_SUCCESS value if operation succeeded, otherwise returns one of the *error codes*.

This operation only creates the regularizer so that it can be used by topic models. To actually apply the regularizer you should include its name in ModelConfig.regularizer_name list of a model config.

### 8.4.12 ArtmReconfigureRegularizer

int **ArtmReconfigureRegularizer**(int *master_id*, int *length*, const char* *regularizer_config*)
Updates the configuration of the regularizer.

**Parameters**

- **master_id** (*int*) – The ID of a master component or a master proxy, returned by either ArtmCreateMasterComponent() or ArtmCreateMasterProxy() method.
- **regularizer_config** (*const_char**) – Serialized *RegularizerConfig* message, describing the configuration of a new regularizer.
- **length** (*int*) – The length in bytes of the *regularizer_config* message.

**Returns** Returns ARTM_SUCCESS value if operation succeeded, otherwise returns one of the *error codes*.

### 8.4.13 ArtmDisposeRegularizer

int **ArtmDisposeRegularizer**(int *master_id*, const char* *regularizer_name*)
Explicitly delete a specific regularizer. All regularizers within specific master component are also deleted automatically by ArtmDisposeMasterComponent().

**Parameters**

- **master_id** (*int*) – The ID of a master component or a master proxy, returned by either ArtmCreateMasterComponent() or ArtmCreateMasterProxy() method.
- **regularizer_name** (*const_char**) – A string identified of the regularizer that should be deleted.

**Returns** Returns ARTM_SUCCESS value if operation succeeded, otherwise returns one of the *error codes*.

### 8.4.14 ArtmCreateDictionary

int **ArtmCreateDictionary**(int *master_id*, int *length*, const char* *dictionary_config*)
Creates a new dictionary.

**Parameters**

- **master_id** (*int*) – The ID of a master component or a master proxy, returned by either ArtmCreateMasterComponent() or ArtmCreateMasterProxy() method.
- **dictionary_config** (*const_char**) – Serialized *DictionaryConfig* message, describing the configuration of a new dictionary.
- **length** (*int*) – The length in bytes of the *dictionary_config* message.

**Returns** Returns ARTM_SUCCESS value if operation succeeded, otherwise returns one of the *error codes*.

### 8.4.15 ArtmReconfigureDictionary

int **ArtmReconfigureDictionary**(int *master_id*, int *length*, const char* *dictionary_config*)
Updates the dictionary.

**Parameters**

- **master_id** (*int*) – The ID of a master component or a master proxy, returned by either `ArtmCreateMasterComponent()` or `ArtmCreateMasterProxy()` method.
- **dictionary_config** (*const_char**) – Serialized *DictionaryConfig* message, describing the new configuration of the dictionary.
- **length** (*int*) – The length in bytes of the *dictionary_config* message.

**Returns** Returns `ARTM_SUCCESS` value if operation succeeded, otherwise returns one of the *error codes*.

### 8.4.16 ArtmDisposeDictionary

int **`ArtmDisposeDictionary`**(int *master_id*, const char* *dictionary_name*)

Explicitly delete a specific dictionary. All dictionaries within specific master component are also deleted automatically by `ArtmDisposeMasterComponent()`.

**Parameters**

- **master_id** (*int*) – The ID of a master component or a master proxy, returned by either `ArtmCreateMasterComponent()` or `ArtmCreateMasterProxy()` method.
- **dictionary_name** (*const_char**) – A string identified of the dictionary that should be deleted.

**Returns** Returns `ARTM_SUCCESS` value if operation succeeded, otherwise returns one of the *error codes*.

### 8.4.17 ArtmAddBatch

int **`ArtmAddBatch`**(int *master_id*, int *length*, const char* *add_batch_args*)

Adds batch for processing.

**Parameters**

- **master_id** (*int*) – The ID of a master component or a master proxy, returned by either `ArtmCreateMasterComponent()` or `ArtmCreateMasterProxy()` method.
- **add_batch_args** (*const_char**) – Serialized *AddBatchArgs* message, describing the arguments of this operation.
- **length** (*int*) – The length in bytes of the *add_batch_args* message.

**Returns** Returns `ARTM_SUCCESS` value if operation succeeded, otherwise returns one of the *error codes*.

This operation is only allowed when `MasterComponentConfig.online_batch_processing` is set to *True* and `MasterComponentConfig.modus_operandi` is set to *Local*.

### 8.4.18 ArtmInvokeIteration

int **`ArtmInvokeIteration`**(int *master_id*, int *length*, const char* *invoke_iteration_args*)

Invokes several iterations over the collection.

**Parameters**

- **master_id** (*int*) – The ID of a master component or a master proxy, returned by either `ArtmCreateMasterComponent()` or `ArtmCreateMasterProxy()` method.

- **char* invoke_iteration_args** (*const*) – Serialized *InvokeIterationArgs* message, describing the arguments of this operation.
- **length** (*int*) – The length in bytes of the *invoke_iteration_args* message.

**Returns** Returns `ARTM_SUCCESS` value if operation succeeded, otherwise returns one of the *error codes*.

### 8.4.19 ArtmSynchronizeModel

int **`ArtmSynchronizeModel`**(int *master_id*, int *length*, const char* *sync_model_args*)
Synchronizes topic model.

**Parameters**

- **master_id** (*int*) – The ID of a master component or a master proxy, returned by either `ArtmCreateMasterComponent()` or `ArtmCreateMasterProxy()` method.
- **sync_model_args** (*const_char**) – Serialized *SynchronizeModelArgs* message, describing the arguments of this operation.
- **length** (*int*) – The length in bytes of the *sync_model_args* message.

**Returns** Returns `ARTM_SUCCESS` value if operation succeeded, otherwise returns one of the *error codes*.

This operation updates the Phi matrix of the topic model with all model increments, collected since last call to ArtmSynchronizeModel. In addition, this operation invokes all Phi-regularizers for the requested topic model.

### 8.4.20 ArtmInitializeModel

int **`ArtmSynchronizeModel`**(int *master_id*, int *length*, const char* *init_model_args*)
Initializes the phi matrix of a topic model with some random initial approximation.

**Parameters**

- **master_id** (*int*) – The ID of a master component or a master proxy, returned by either `ArtmCreateMasterComponent()` or `ArtmCreateMasterProxy()` method.
- **init_model_args** (*const_char**) – Serialized *InitializeModelArgs* message, describing the arguments of this operation.
- **length** (*int*) – The length in bytes of the *init_model_args* message.

**Returns** Returns `ARTM_SUCCESS` value if operation succeeded, otherwise returns one of the *error codes*.

### 8.4.21 ArtmWaitIdle

int **`ArtmWaitIdle`**(int *master_id*, int *length*, const char* *wait_idle_args*)
Awaits for ongoing iterations.

**Parameters**

- **master_id** (*int*) – The ID of a master component or a master proxy, returned by either `ArtmCreateMasterComponent()` or `ArtmCreateMasterProxy()` method.
- **wait_idle_args** (*const_char**) – Serialized *WaitIdleArgs* message, describing the arguments of this operation.

- **length** (*int*) – The length in bytes of the *wait_idle_args* message.

**Returns** Returns `ARTM_SUCCESS` value if operation succeeded, otherwise returns one of the *error codes*.

### 8.4.22 ArtmOverwriteTopicModel

int **`ArtmOverwriteTopicModel`**(int *master_id*, int *length*, const char* *topic_model*)

This operation schedules an update of an entire topic model or of it subpart.

**Parameters**

- **master_id** (*int*) – The ID of a master component or a master proxy, returned by either `ArtmCreateMasterComponent()` or `ArtmCreateMasterProxy()` method.
- **topic_model** (*const_char**) – Serialized *TopicModel* message, describing the new topic model.
- **length** (*int*) – The length in bytes of the *topic_model* message.

**Returns** Returns `ARTM_SUCCESS` value if operation succeeded, otherwise returns one of the *error codes*.

Note that this operation only schedules the update of a topic model. To make sure the update is completed you must call `ArtmWaitIdle()` and `ArtmSynchronizeModel()`. Remember that by default `ArtmSynchronizeModel()` will calculate all regularizers enabled in the configuration of the topic model. The may result in a different topic model than the one you passed as *topic_model* parameter. To avoid this behavior set `SynchronizeModelArgs.invoke_regularizers` to `false`.

### 8.4.23 ArtmRequestThetaMatrix

int **`ArtmRequestThetaMatrix`**(int *master_id*, int *length*, const char* *get_theta_args*)

Requests theta matrix. Use `ArtmCopyRequestResult()` to copy the resulting message.

**Parameters**

- **master_id** (*int*) – The ID of a master component or a master proxy, returned by either `ArtmCreateMasterComponent()` or `ArtmCreateMasterProxy()` method.
- **get_theta_args** (*const_char**) – Serialized *GetThetaMatrixArgs* message, describing the arguments of this operation.
- **length** (*int*) – The length in bytes of the *get_theta_args* message.

**Returns** In case of success, returns the length in bytes of a buffer that should be allocated on callers site and then passed to `ArtmCopyRequestResult()` method. This will populate the buffer with *ThetaMatrix* message, carrying the requested information. In case of a failure, returns one of the *error codes*.

### 8.4.24 ArtmRequestTopicModel

int **`ArtmRequestTopicModel`**(int *master_id*, int *length*, const char* *get_model_args*)

Requests topic model.

**Parameters**

- **master_id** (*int*) – The ID of a master component or a master proxy, returned by either `ArtmCreateMasterComponent()` or `ArtmCreateMasterProxy()` method.

- **get_model_args** (*const_char**) – Serialized *GetTopicModelArgs* message, describing the arguments of this operation.
- **length** (*int*) – The length in bytes of the *get_model_args* message.

**Returns** In case of success, returns the length in bytes of a buffer that should be allocated on callers site and then passed to `ArtmCopyRequestResult()` method. This will populate the buffer with *TopicModel* message, carrying the requested information. In case of a failure, returns one of the *error codes*.

### 8.4.25 ArtmRequestRegularizerState

int **`ArtmRequestRegularizerState`**(int *master_id*, const char* *regularizer_name*)

Requests state of a specific regularizer.

**Parameters**

- **master_id** (*int*) – The ID of a master component or a master proxy, returned by either `ArtmCreateMasterComponent()` or `ArtmCreateMasterProxy()` method.
- **regularizer_name** (*const_char**) – A string identified of the regularizer.

**Returns** In case of success, returns the length in bytes of a buffer that should be allocated on callers site and then passed to `ArtmCopyRequestResult()` method. This will populate the buffer with *RegularizerInternalState* message, carrying the requested information. In case of a failure, returns one of the *error codes*.

### 8.4.26 ArtmRequestScore

int **`ArtmRequestScore`**(int *master_id*, int *length*, const char* *get_score_args*)

Request the result of score calculation.

**Parameters**

- **master_id** (*int*) – The ID of a master component or a master proxy, returned by either `ArtmCreateMasterComponent()` or `ArtmCreateMasterProxy()` method.
- **const_char*** – get_score_args: Serialized *GetScoreValueArgs* message, describing the arguments of this operation.
- **length** (*int*) – The length in bytes of the *get_score_args* message.

**Returns** In case of success, returns the length in bytes of a buffer that should be allocated on callers site and then passed to `ArtmCopyRequestResult()` method. This will populate the buffer with *ScoreData* message, carrying the requested information. In case of a failure, returns one of the *error codes*.

### 8.4.27 ArtmRequestParseCollection

int **`ArtmRequestParseCollection`**(int *length*, const char* *collection_parser_config*)

Parses a text collection into a set of batches and stores them on disk. Returns a *DictionaryConfig* message that lists all tokens, occured in the collection.

Check the description of *CollectionParserConfig* message for more details about this operation.

**Parameters**

- **const_char*** – collection_parser_config: Serialized *CollectionParserConfig* message, describing the configuration the collection parser.
- **length** (*int*) – The length in bytes of the *collection_parser_config* message.

**Returns** In case of success, returns the length in bytes of a buffer that should be allocated on callers site and then passed to `ArtmCopyRequestResult()` method. The buffer will contain *DictionaryConfig* message, that lists all unique tokens from the collection being parsed. In case of a failure, returns one of the *error codes*.

**Warning:** The following error most likelly indicate that you are trying to parse a very large file in 32 bit version of BigARTM.
`InternalError : failed mapping view: The parameter is incorrect`
Try to use 64 bit BigARTM to workaround this issue.

### 8.4.28 ArtmRequestLoadDictionary

int **`ArtmRequestLoadDictionary`**(const char* *filename*)
Loads a *DictionaryConfig* message from disk.

**Parameters**

- **const_char*** – filename: A full file name of a file that contains a serialized DictionaryConfig message.

**Returns** In case of success, returns the length in bytes of a buffer that should be allocated on callers site and then passed to `ArtmCopyRequestResult()` method. The buffer will contain the resulting *DictionaryConfig* message. In case of a failure, returns one of the *error codes*.

This method can be used to load `CollectionParserConfig.dictionary_file_name` or `CollectionParserConfig.cooccurrence_file_name` dictionaries, saved by *ArtmRequestParseCollection* method.

### 8.4.29 ArtmRequestLoadBatch

int **`ArtmRequestLoadBatch`**(const char* *filename*)
Loads a *Batch* message from disk.

**Parameters**

- **const_char*** – filename: A full file name of a file that contains a serialized Batch message.

**Returns** In case of success, returns the length in bytes of a buffer that should be allocated on callers site and then passed to `ArtmCopyRequestResult()` method. The buffer will contain the resulting *Batch* message. In case of a failure, returns one of the *error codes*.

This method can be used to load batches saved by *ArtmRequestParseCollection* method or *ArtmSaveBatch* method.

### 8.4.30 ArtmCopyRequestResult

int **`ArtmCopyRequestResult`**(int *length*, char* *address*)
Copies the result of the last request.

**Parameters**

- **const_char*** – address: Target memory location to copy the data.

- **length** (*int*) – The length in bytes of the *address* buffer.

**Returns** Returns `ARTM_SUCCESS` value if operation succeeded, otherwise returns one of the *error codes*.

### 8.4.31 ArtmSaveBatch

int **ArtmSaveBatch**(const char* *disk_path*, int *length*, const char* *batch*)
Saves a *Batch* message to disk.

**Parameters**

- **const_char*** – disk_path: A floder where to save the batch.
- **batch** (*const_char**) – Serialized *Batch* message to save.
- **length** (*int*) – The length in bytes of the *batch* message.

**Returns** Returns `ARTM_SUCCESS` value if operation succeeded, otherwise returns one of the *error codes*.

### 8.4.32 ArtmGetLastErrorMessage

const char* **ArtmGetLastErrorMessage**()
Retrieves the textual error message, occured during the last failing request.

### 8.4.33 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
#define ARTM_NETWORK_ERROR -9
```

**ARTM_SUCCESS**
The API call succeeded.

**ARTM_STILL_WORKING**
This error code is applicable only to `ArtmWaitIdle()`. 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 neither MasterComponent nor MasterProxy exist with given ID.

**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.

**ARTM_NETWORK_ERROR**
Network call did not reply withing the specified timeout.

## 8.5 C++ interface

This document explains C++ interface of BigARTM library.

In addition to this page consider to look at *Plain C interface of BigARTM*, *Python Interface* or *Messages*. These documentation files are also to certain degree relevant for C++ interface, because C++ interface is quite similar to Python interface and share the same Protobuf messages.

### 8.5.1 MasterComponent

**class MasterComponent**

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

**MasterComponent**(const MasterProxyConfig& *config*)
Creates a proxy to remote master component with configuration defined by *MasterProxyConfig* 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 **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*).

### 8.5.2 NodeController

**class NodeController**

**NodeController**(const NodeControllerConfig& *config*)
Creates a node controller with configuration defined by *NodeControllerConfig* message.

### 8.5.3 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 **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*.

### 8.5.4 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.

### 8.5.5 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.

### 8.5.6 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.

## 8.6 BigARTM command line utility

This document provides an overview of `cpp_client`, a simple command-line utility shipped with BigARTM.

To run *cpp_client* you need to download input data (a textual collection represented in bag-of-words format). We recommend to download *vocab* and *docword* files by links provided in *Parse collection* section of the tutorial. Then you can use *cpp_client* as follows:

```
cpp_client -d docword.kos.txt -v vocab.kos.txt
```

You may append the following options to customize the resulting topic model:

- `-t` or `--num_topic` sets the number of topics in the resulting topic model.
- `-i` or `--num_iters` sets the number of iterative scans over the collection.
- `--num_inner_iters` sets the number of updates of theta matrix performed on each iteration.
- `--reuse_theta` enables caching of Theta matrix and re-uses last Theta matrix from the previous iteration as initial approximation on the next iteration. The default alternative without `--reuse_theta` switch is to generate random approximation of Theta matrix on each iteration.
- `--tau_phi`, `--tau_theta` and `--tau_decor` allows you to specify weights of different regularizers. Currently *cpp_client* does not allow you to customize regularizer weights for different topics and for different iterations. This limitation is only related to *cpp_client*, and you can simply achieve this by using BigARTM interface (either in Python or in C++).
- `--update_every` is a parameter of the online algorithm. When specified, the model will be updated every *update_every* documents.

You may also apply the following optimizations that should not change the resulting model

- `--reuse_batches` skips parsing of *docword* and *vocab* files, and tries to use batches located in `--batch_folder`. You may download pre-parsed batches by links provided in *Parse collection* section of the tutorial.
- `-p` allows you to specify number of concurrent processors. The recommended value is to use the number of logical cores on your machine.
- `--no_scores` disables calculation and visualization of all scores. This is a clean way of measuring pure performance of BigARTM, because at the moment some scores takes unnecessary long time to calculate.
- `--disk_cache_folder` applies only together with `--reuse_theta`. This parameter allows you to specify a writable disk location where BigARTM can cache Theta matrix between iterations to avoid storing it in main memory.
- `--merger_queue_size` limits the size of the merger queue. Decrease the size of the queue might reduce memory usage, but decrease CPU utilization of the processors.

```
>cpp_client --help
BigARTM - library for advanced topic modeling (http://bigartm.org):

Basic options:
  -h [ --help ]                         display this help message
  -d [ --docword ] arg                  docword file in UCI format
  -v [ --vocab ] arg                    vocab file in UCI format
  -t [ --num_topic ] arg (=16)          number of topics
  -p [ --num_processors ] arg (=2)      number of concurrent processors
  -i [ --num_iters ] arg (=10)          number of outer iterations
  --num_inner_iters arg (=10)           number of inner iterations
  --reuse_theta                         reuse theta between iterations
  --batch_folder arg (=batches)         temporary folder to store batches
  --dictionary_file arg (=filename of dictionary file)
  --reuse_batches                       reuse batches found in batch_folder
                                        (default = false)
  --items_per_batch arg (=500)          number of items per batch
  --tau_phi arg (=0)                    regularization coefficient for PHI
                                        matrix
  --tau_theta arg (=0)                  regularization coefficient for THETA
                                        matrix
  --tau_decor arg (=0)                  regularization coefficient for topics
                                        decorrelation (use with care, since
                                        this value heavily depends on the size
                                        of the dataset)
  --paused                              wait for keystroke (allows to attach a
                                        debugger)
  --no_scores                           disable calculation of all scores
  --update_every arg (=0)               [online algorithm] requests an update
                                        of the model after update_every
                                        document
  --parsing_format arg (=0)             parsing format (0 - UCI, 1 - matrix
                                        market)
  --disk_cache_folder arg               disk cache folder
  --merger_queue_size arg               size of the merger queue

Networking options (experimental):
  --nodes arg                   endpoints of the remote nodes (enables network
                                modus operandi)
  --localhost arg (=localhost) DNS name or the IP address of the localhost
  --port arg (=5550)            port to use for master node
  --proxy arg                   proxy endpoint
  --timeout arg (=1000)         network communication timeout in milliseconds

Examples:
        cpp_client -d docword.kos.txt -v vocab.kos.txt
        set GLOG_logtostderr=1 & cpp_client -d docword.kos.txt -v vocab.kos.txt
```

For further details please refer to the source code of cpp_client.

CHAPTER 9

# Publications


1. Vorontsov K. V. Additive Regularization for Topic Models of Text Collections // Doklady Mathematics. 2014, Pleiades Publishing, Ltd. — Vol. 89, No. 3, pp. 301–304. PDF in English, PDF in Russian.
2. Vorontsov K. V., Potapenko A. A. Tutorial on Probabilistic Topic Modeling: Additive Regularization for Stochastic Matrix Factorization // AIST'2014, Analysis of Images, Social networks and Texts. Springer International Publishing Switzerland, 2014. Communications in Computer and Information Science (CCIS). Vol. 436. pp. 29–46. PDF in English.
3. Vorontsov K. V., Potapenko A. A. Additive Regularization of Topic Models // Machine Learning Journal, Special Issue "Data Analysis and Intelligent Optimization", Springer, 2014. (to appear). PDF in English, PDF in Russian.

CHAPTER 10

# Indices and tables

- *genindex*
- *modindex*
- *search*

## a

## A



## B



## C

## D



## E



## F



## G

## I



## K



## L



## M

## N



## O



## P



## R



## S

## T



## U



## V



## W



## Z