# Использование библиотеки для тематического моделирования BigARTM

Мурат Апишев
great-mel@yandex.ru
MelLain@github.com

МГУ им М.В. Ломоносова

16 октября 2015

# Основные особенности

BigARTM — open source библиотека для тематического моделирования, в основе — теория APTM.

- Эффективная параллельная обработка данных — *одна из самых быстрых реализация алгоритма TM*
- Онлайновый алгоритм — *возможность обработки данных в потоковом режиме*
- Кроссплатформенность — *Windows, Linix, Mac OS*
- Библиотека регуляризаторов
- Библиотека функционалов качества
- Пользовательские API на разных языках — *C++, Python, ...*

# Онлайновый пакетный EM-алгоритм

**Algorithm 1** BigARTM's algorithm

Initialize $\phi^0_{wt}$ for all $w \in W$ and $t \in T$;
**for all** $i = 1, \dots, I$ **do**
  $n^i_{wt} := 0, n^i_t := 0$ for all $w \in W$ and $t \in T$;
  **for all** batches $D_j$, j = 1,...,J **do**
    $\tilde{n}_{wt} := 0, \tilde{n}_t := 0$ for all $w \in W$ and $t \in T$;
    **for all** $d \in D_j$ **do**
      initialize $\theta_{td}$ for all $t \in T$;
      **repeat**
        $Z_w := \sum_{t \in T} \phi^{i-1}_{wt} \theta_{td}$ for all $w \in d$;
        $\theta_{td} := \frac{1}{n_d} \sum_{w \in d} n_{dw} \phi^{i-1}_{wt} \theta_{td} / Z_w$ for all $t \in T$;
      **until** $\theta_d$ converges;
      increment $\tilde{n}_{wt}, \tilde{n}_t$ by $n_{dw} \phi^{i-1}_{wt} \theta_{td} / Z_w$ for all $w \in W$ and $t \in T$;
    $n^i_{wt} := n^i_{wt} + \tilde{n}^i_{wt}$ for all $w \in W$ and $t \in T$;
    $n^i_t := n^i_t + \tilde{n}_t$ for all $t \in T$;
  $\phi^i_{wt} := \frac{n^i_{wt}}{n^i_t}$ for all $w \in W$ and $t \in T$;

# Компоненты библиотеки

1. **Ядро библиотеки** — параллельная (многопоточная) реализация онлайнового EM-алгоритма
2. **Регуляризаторы** — плагины, которые можно добавлять к библиотеке без структурных изменений ядра
3. **Функционалы** — плагины, во многом похожи на регуляризаторы
4. **Словари** — внешние объекты с различными данными о коллекции
5. **Парсер** — компонент, отвечающий за преобразование входных данных в формат BigARTM, поддерживает форматы UCI-BoW, Vowpal Wabbit, обычный текст.

# Регуляризаторы

- Сглаживание/разреживание $\Phi$ (+ частичное обучение)
- Сглаживание/разреживание $\Theta$ (+ частичное обучение)
- Декорреляция тем в $\Phi$
- Заданное разреживание $\Phi$
- Балансирование классов в $\Phi$
- Повышение когерентности тем в $\Phi$

# Функционалы качества

- Перплексия
- Разреженность $\Phi$
- Разреженность $\Theta$
- Характеристики ядер тем
- Самые вероятные слова
- Срез матрицы $\Theta$
- Число обработанных документов
- Доля фоновых тем в модели

# Словари

Словарь (в пользовательском смысле) — это файл, в котором лежит информация о коллекции, подготовленная парсером.

Словарь можно сделать вручную (**опционально, большинству простых пользователей не нужно!**).

**Оффтоп**: Google Protocol Buffers

1. Позволяет описывать структуры данных (*сообщения*) на псевдоязыке и компилировать их в классы на C++/Python/Java/...
2. Предоставляет механизм сериализации сообщений в строки и наоборот. Сериализованную в, например, C++, строку можно передать в Python и десериализовать в аналогичное сообщение, но уже на Python. И наоборот.

# Словари

Поля сообщения описываются в формате `qualifier type name = number`

`qualifier` — квалификатор, принимает два значения:
`optional` — скалярное поле, `repeated` — массив.

`type` — тип поля, может быть одним из базовых типов, либо другим сообщением.

Пример:

```
message FloatArray { repeated float value = 1; }
optional FloatArray values = 1;
```

Фрагмент кода сообщения, описывающего словарь:

```
message DictionaryConfig {
  optional string name = 1;
  repeated DictionaryEntry entry = 2;
  optional int32 total_items_count = 4;
  optional DictionaryCoocurenceEntries cooc_entries = 5;
  optional float total_token_weight = 6;
}
```

# Словари

Фрагемент кода сообщения, описывающего один токен в словаре

```
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 float token_weight = 8;
}
```

Кода сообщения, описывающего подсловарь со-встречаемостей:

```
message DictionaryCoocurenceEntries {
  repeated int32 first_index = 1;
  repeated int32 second_index = 2;
  repeated float value = 3;
  optional bool symmetric_cooc_values = 4 [default = false];
}
```

# Создание словаря

Пример:

Дан словарь (набор всех уникальных токенов) коллекции в виде Python dict `vocab_values`.

Для каждого ключа значение является неотрицательным числом, которое нужно вычесть из счётчиков $n_{wt}$ для данного слова при использовании регуляризатора сглаживания/разреживания Ф.

Самый простой вариант:

1. загрузить созданный парсером словарь BigARTM для этой коллекции (в нём уже лежат все токены из `vocab_values`);
2. внести в сообщение необходимые изменения;
3. сохранить словарь обратно на диск и использовать его.

# Создание словаря

Пример:

Допустим, что словарь лежит на диске в файле `dictionary`. Тогда

```
import artm

dictionary = artm.messages.DictionaryConfig()
with open('dictionary', 'rb') as f:
  dictionary.SerializeFromString(f.read())

for entry in dictionary.entry:
  entry.value = vocab_values[entry.key_token]

with open('dictionary', 'wb') as f:
  f.write(dictionary.SerializeToString())
```

# Подготовка данных

По аналогии с sklearn входные данные — объект `BatchVectorizer`:

```
BatchVectorizer(batches=None,
                collection_name=None,
                data_path='',
                data_format='batches',
                target_folder='',
                batch_size=1000,
                dictionary_name='dictionary'):
```

Какие могут быть входные данные:

1. готовые батчи (`batches` = путь к директории с батчами)

# Подготовка данных

Какие могут быть входные данные:

2. мешок слов в формате UCI-Bow (`collection_name` = имя коллекции по названию файла, `data_path` = путь к директории с данными, `target_folder` = путь к директории для батчей и словаря, `data_format = 'bow_uci'`)
3. текст в формате Vowpal Wabbit (`data_path` = путь к директории с данными, `target_folder` = путь к директории для батчей и словаря, `data_format = 'vowpal_wabbit'`)

Пример:

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

# Создание модели

```
ARTM(num_processors=0, num_topics=10,
     topic_names=None, class_ids=None,
     cache_theta=True, scores=None,
     regularizers=None)
```

`num_processors` — число потоков-обработчиков
`num_topics` — число тем (вариант попроще)
`topic_names` — список с именами тем (вариант посложнее)
`class_ids` — dict, ключ – имя модальности, значение – вес
`cache_theta` — флаг кэширования $\Theta$
`scores` — функционалы качества
`regularizers` — регуляризаторы

Пример:

```
topic_names = ['topic_name_{}'.format(n) for n in xrange(15)]
model = artm.ARTM(num_processors=4,
                  topic_names=topic_names)
```

# Инициализация модели

Есть два способа инициализации: по батчам и по словарю.

```
ARTM.initialize(data_path=None, dictionary_name=None)
```

В первом случае нужно указать путь к директории с батчами. Во втором — указать имя словаря, загруженного в ядро.

**Не путать!** Есть имя словаря, хранящееся в нём, имя файла со словарём и имя, которые словарь получает при загрузке в ядро с помощью метода:

```
ARTM.load_dictionary(dictionary_name=None, dictionary_path=None)
```
`dictionary_name` — это самое имя.

Удалить словарь из ядра можно двойственным методом:

```
ARTM.remove_dictionary(dictionary_name=None)
```

Пример инициализации по словарю:
```
model.load_dictionary(dictionary_name='dictionary',
                      dictionary_path='kos/dictionary')
model.initialize(dictionary_name='dictionary')
```

# Регуляризаторы

Добавление регуляризаторов:

```
model.regularizers.add(Regularizer(parameters))
```

Вместо `Regularizer` могут быть:

- artm.SmoothSparsePhiRegularizer()
- artm.SmoothSparseThetaRegularizer()
- artm.DecorrelatorPhiRegularizer()
- artm.LabelRegularizationPhiRegularizer()
- artm.SpecifiedSparsePhiRegularizer()
- artm.ImproveCoherencePhiRegularizer()
- artm.SmoothPtdwRegularizer()

# Регуляризаторы

У каждого регуляризатора есть свои параметры (см. док-строки `bigartm/python/artm/regularizers.py`)

Пример: регуляризатор сглаживания/разреживания Φ:

1. `name` — имя регуляризатора, строка
2. `tau` — коэффициент регуляризации, вещественное число
3. `topic_names` — список имён регуляризуемых тем, список строк
4. `class_ids` — список имён регуляризуемых модальностей, список строк
5. `dictionary_name` — имя словаря, который нужен регуляризатору, строка

Все параметры опциональные.

# Регуляризаторы

Пример подключения регуляризаторов:

```
model.regularizers.add(
    artm.SmoothSparsePhiRegularizer(name='SparsePhi', tau=-1.0,
                                    topic_names=topic_names[: -2],
                                    dictionary_name='dictionary'))
```

Пусть все темы, кроме последних двух, — предметные. Регуляризатор будет разреживать их значениями, сохранёнными в словаре.

Опишем ещё регуляризатор сглаживания последних двух фоновых тем:

```
model.regularizers.add(
    artm.SmoothSparsePhiRegularizer(name='SmoothPhi', tau=0.3,
                                    topic_names=topic_names[-2: ]))
```

**Схема регуляризации:**

$n_{wt}^{new} = n_{wt}^{old} + \tau * d_w$, где $d_w$ — это `dictionary.entry[w].value`. Если имя словаря не было задано, или словарь не был найден, или значение для слова отсутствует $\Rightarrow d_w = 1.0$.

# Функционалы качества

Добавление функционалов качества аналогично добавлению регуляризаторов.

```
model.scores.add(Score(parameters))
```

Вместо `Score` могут быть:

- artm.PerplexityScore()
- artm.SparsityPhiScore()
- artm.SparsityThetaScore()
- artm.TopicKernelScore()
- artm.TopTokensScore()
- artm.ThetaSnippetScore()
- artm.ItemsProcessedScore()
- artm.TopicMassPhiScore()

# Функционалы качества

У каждого функционала есть свои параметры (см. док-строки `bigartm/python/artm/scores.py`)

Пример: функционал самых вероятных слов в каждой теме:

1. `name` — имя функционала, строка
2. `topic_names` — список имён оцениваемых тем, список строк
3. `class_id` — имя оцениваемой модальности, строка
4. `num_tokens` — максимальное число искомых топ-токенов, целое число
5. `dictionary_name` — имя словаря, в данном случае с информацией о совместной встречаемости слов для подсчёта когерентности по топ-токенам, строка

Все параметры опциональные.

# Функционалы качества

Пример подключения функционалов. Подключим общую перплексию, разреженность $\Phi$ и $\Theta$ для предметных тем и топ-токены для всех тем:

```
model.scores.add(artm.PerplexityScore(name='Perplexity')

model.scores.add(artm.SparsityPhiScore(
                 name='SparsityPhi',
                 topic_names=topic_names[: -2]))

model.scores.add(artm.SparsityThetaScore(
                 name='SparsityTheta',
                 topic_names=topic_names[: -2]))

model.scores.add(artm.TopTokensScore(
                 name='TopTokens', num_tokens=10))
```

# Снова конструктор модели

В принципе, все регуляризаторы и функционалы можно описать прямо в конструкторе ARTM:

```
topic_names = ['topic_name_{}'.format(n) for n in xrange(15)]
model = artm.ARTM(num_processors=4,
                  topic_names=topic_names,
                  regularizers=[artm.SmoothSparsePhiRegularizer(
                                    name='SparsePhi',
                                    tau=-1.0,
                                    topic_names=topic_names[: -2],
                                    dictionary_name='dictionary'),
                                artm.SmoothSparsePhiRegularizer(
                                    name='SmoothPhi',
                                    tau=0.3,
                                    topic_names=topic_names[-2: ])
                                ],
                  scores=[artm.PerplexityScore(name='Perplexity'),
                          artm.SparsityPhiScore(
                              name='SparsityPhi',
                              topic_names=topic_names[: -2]),
                          artm.SparsityThetaScore(
                              name='SparsityTheta',
                              topic_names=topic_names[: -2]),
                          artm.TopTokensScore(
                              name='TopTokens',
                              num_tokens=10)
                         ])
```

# Алгоритм обучения

Оффлайн EM-алгоритм

1. Многократное итерирование по коллекции.
2. Однократный проход по документу.
3. Необходимость хранить матрицу $\Theta$.
4. $\Phi$ обновляется в конце каждого прохода по коллекции.
5. Применяется при обработке небольших коллекций.

Онлайн EM-алгоритм

1. Однократный проход по коллекции.
2. Многократное итерирование по документу.
3. Нет необходимости хранить матрицу $\Theta$.
4. $\Phi$ обновляется через определённое число обработанных документов.
5. Применяется при обработке больших коллекций в потоковом режиме.

# Оффлайн алгоритм

```
ARTM.fit_offline(batch_vectorizer=None,
                 num_collection_passes=20,
                 num_document_passes=1,
                 reuse_theta=True,
                 dictionary_filename='dictionary')
```

| | |
|---|---|
| `batch_vectorizer` | – объект с данными |
| `num_collection_passes` | – число проходов по коллекции |
| `num_document_passes` | – число проходов по документу |
| `reuse_theta` | – использование $\Theta$ с прошлой итерации |
| `dictionary_filename` | – имя словаря для авто-инициализации |

Пример:

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

# Онлайн алгоритм

```
ARTM.fit_online(batch_vectorizer=None,
                tau0=1024.0,
                kappa=0.7,
                update_every=1,
                num_document_passes=10,
                reset_theta_scores=False,
                dictionary_filename='dictionary')
```

`batch_vectorizer` – объект с данными
`tau0, kappa` – элементы расчётной формулы
`update_every` – частота обновлений $\Phi$ в батчах
`num_document_passes` – число проходов по документу
`reset_theta_scores` – сбрасывание функционалов $\Theta$
`dictionary_filename` – имя словаря для авто-инициализации

```
update_count = current_processed_docs / (batch_size * update_every)
rho = pow(tau0 + update_count, -kappa)
```

`decay_weight = 1-rho` – вес старых счётчиков $n_{wt}$ при обновлении
`apply_weight = rho` – вес новых счётчиков $n_{wt}$ при обновлении

# Извлечение функционалов качества

ARTM хранит информацию о всех значениях всех подключенных функционалов на момент каждого обновления матрицы Φ.

Доступ к функционалам унифицирован:

```
ARTM.score_tracker[<score_name>].<field>
```

<field> — различные поля функционала, например, для разреженности Φ: value, zero_tokens, total_tokens. Для каждого поля есть аналог с приставкой last_ — возвращает значение поля на последней синхронизации.

**Вся информация о полях всех функционалов описана в док-строках в файле bigartm/python/artm/score_tracker.py!**

Пример извлечения значений перплексии за всё время обучения:

```
model.score_tracker['Perplexity'].value
```

Выход — список значений перплексии на обучающей коллекции на каждой синхронизации.

# Извлечение Φ

Всю матрицу Φ можно извлечь в виде `pandas.DataFrame` с помощью метода `ARTM.phi_`.

Но! Этот метод — соответствие формальным правилам `sklearn`. Лучше пользуйтесь `ARTM.get_phi()`, больше возможностей:

```
ARTM.get_phi(topic_names=None, class_ids=None)
```

`topic_names` – имена тем, которые надо извлечь
`class_ids` – имена модальностей, которые надо извлечь

Пример:

```
model.get_phi(topic_names=topic_names[1: 10])
```

# Извлечение $\Theta$, построение $\theta_d$ для новых документов

```
ARTM.fit_transform(topic_names=None, remove_theta=False)
```

`topic_names` – имена тем, которые надо извлечь
`remove_theta` – удалять $\Theta$ из кэша (True $\Rightarrow$ больше извлечь эту матрицу до новых итераций нельзя!)

Получение столбцов $\theta_d$ для новых документов:

```
ARTM.transform(batch_vectorizer=None,
               num_document_passes=1)
```

`batch_vectorizer` – объект с данными
`num_document_passes` – число итераций прохода по документу (вывод осуществляется с помощью итераций EM-алгоритма с фиксированной $\Phi$).

# Сохранение и загрузка модели

Модель можно выгружать в бинарном виде и загружать обратно.

При этом сохраняется только информация о матрице $\Phi$.

Все данные о $\Theta$, о числе пройденных итераций и значениях функционалов качества стирается!

```
ARTM.save(filename='artm_model')
```

```
ARTM.load(filename)
```

# Полезные ссылки

Официальный сайт проекта:

www.bigartm.org

Адрес GitHub-репозитория:

www.github.com/bigartm/bigartm

IPython Notebook с примером использования Python API (модельный эксперимент):

www.github.com/bigartm/bigartm-book/blob/master/BigARTM_example_RU.ipynb

Все вопросы можно адресовать сюда:
great-mel@yandex.ru