Параллельные и распределённые реализации алгоритмов тематического моделирования

Мурат Апишев
great-mel@yandex.ru

7 апреля 2015

1 Введение

Тематическое моделирование — одно из приложений машинного обучения к анализу текстов. Оно определяет тематическую структуру каждого документа из текстовой коллекции, а также тематический профиль всех слов этой коллекции. Современный анализ текстов практически всегда работает с большими объемами данных, которые можно обработать лишь с помощью параллельных или распределенных реализаций алгоритмов тематического моделирования. В этой лекции дается обзор нескольких подобных реализаций для модели LDA. Кроме того, обсуждается модель APTM (аддитивной регуляризации тематических моделей) и основанная на ней библиотека BigARTM.

§1.1 Обозначения

Пусть есть коллекция документов D, её словарь W и набор тематик T. Опишем ряд обозначений, которые понадобятся далее:

$w \in \{1, \dots, \lvert W\rvert\}$	— номер слова в словаре
$t \in \{1, \dots, \lvert T\rvert\}$	— номер тематики
$d \in \{1, \dots, \lvert D\rvert\}$	— номер документа
N_d	— число слов в документе d
$\boldsymbol{w}_d = [w_{d,1}, \dots, w_{d,N_d}]$	— слова в документе d, $w_{d,n} \in \{1, \dots, \lvert W\rvert\}$
$\boldsymbol{z}_d = [z_{d,1}, \dots, z_{d,N_d}]$	— тематики слов в документе d, $z_{d,n} \in \{1, \dots, \lvert T\rvert\}$
$\boldsymbol{\theta}_d = [\theta_{d,1}, \dots, \theta_{d,\lvert T\rvert}]$	— вероятности тематик в документе d
$\boldsymbol{\varphi}_t = [\varphi_{t,1}, \dots, \varphi_{t,\lvert W\rvert}]$	— вероятности слов тематике t
$\Theta = [\boldsymbol{\theta}_1, \dots, \boldsymbol{\theta}_{\lvert D\rvert}]^T \in \mathbb{R}^{\lvert D\rvert \times \lvert T\rvert}$	— вероятности тематик во всех документах
$\Phi = [\boldsymbol{\varphi}_1, \dots, \boldsymbol{\varphi}_{\lvert T\rvert}]^T \in \mathbb{R}^{\lvert T\rvert \times \lvert W\rvert}$	— вероятности всех слов во всех тематиках
$\mathcal{W} = \{\boldsymbol{w}_1, \dots, \boldsymbol{w}_{\lvert D\rvert}\}$	— набор всех слововхождений в корпусе
$\mathcal{Z} = \{\boldsymbol{z}_1, \dots, \boldsymbol{z}_{\lvert D\rvert}\}$	— разбиение всех слововхождений по тематикам
N_{wt}	— счётчик отнесения слова w к теме t
N_{td}	— счётчик отнесения слова из документа d к теме t

§1.2 LDA

Вероятностная модель *латентного размещения Дирихле* (LDA) [1] имеет вид

$$p(\mathcal{W}, \mathcal{Z}, \Theta, \Phi|\alpha, \beta) = \prod_{t=1}^{|T|} p(\boldsymbol{\varphi}_t|\beta) \prod_{d=1}^{|D|} p(\boldsymbol{\theta}_d|\alpha) \prod_{n=1}^{N_d} p(w_{d,n}|z_{d,n}, \Phi) p(z_{d,n}|\boldsymbol{\theta}_d),$$

$$p(\boldsymbol{\varphi}_t|\beta) = \mathrm{Dir}(\boldsymbol{\varphi}_t|\beta), \quad p(\boldsymbol{\theta}_d|\alpha) = \mathrm{Dir}(\boldsymbol{\theta}_d|\alpha),$$

$$p(w_{d,n}|z_{d,n}, \Phi) = \Phi_{z_{d,n},w_{d,n}}, \quad p(z_{d,n}|\boldsymbol{\theta}_d) = \Theta_{d,z_{d,n}}.$$

Существует два наиболее распространённых способа вывода для модели LDA — VB (вариационный вывод) и MCMC-методы (сэмплирование Гиббса). Суть первого состоит в использовании вариационного EM-алгоритма. Матрицы Φ и Θ получаются по формулам

$$\varphi_{wt} = \frac{\sum_{(d,n):[w_{d,n}=w]} \mu_{d,n,t}}{\sum_{(d,n)} \mu_{d,n,t}}, \quad \theta_{td} = \frac{\gamma_{d,t}}{\sum_{s=1}^{|T|} \gamma_{d,s}},$$

а распределение q, используемое в качестве вариационного приближения, имеет вид

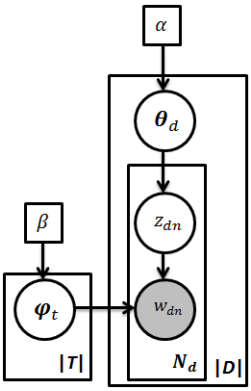

Рис. 1. Графическая модель LDA.

$$q(\mathcal{Z}, \Theta, \Phi) = \prod_{t=1}^{|T|} \mathrm{Dir}(\boldsymbol{\varphi}_t|\boldsymbol{\lambda}_t) \prod_{d=1}^{|D|} \mathrm{Dir}(\boldsymbol{\theta}_d|\boldsymbol{\gamma}_d) \prod_{n=1}^{N_d} \mathrm{Mult}(z_{d,n}|\boldsymbol{\mu}_{d,n}). \tag{1.1}$$

При сэмплировании обычно применяется *коллапсированный* вариант схемы Гиббса, без учёта Φ и Θ, в котором выборка генерируется из распределения $p(\mathcal{Z}|\mathcal{W}, \alpha, \beta)$. Для этого требуется формула рассчёта условного распределения

$$p(z_{d,n} = t|\mathcal{Z}^{\setminus(d,n)}, \mathcal{W}, \alpha, \beta) \propto \frac{N_{wt}^{\setminus(d,n)} + \beta}{\sum_w N_{wt}^{\setminus(d,n)} + |W|\beta} (N_{td}^{\setminus(d,n)} + \alpha). \tag{1.2}$$

Эта выборка может быть использована для оценки наиболее вероятного разбиения слововхождений $\mathcal{W}$ по тематикам (т.е. $\mathcal{Z}$), распределения тематик для каждого документа (Θ), а также вероятностей слов в каждой тематике (Φ).

Алгоритм 2.1: AD-LDA

повторять
  для каждого *процессора p одновременно* **выполнять**
    Скопировать глобальные счётчики $N_{wtp} := N_{wt}$;
    Сэмплировать $\boldsymbol{z}_p$: LDA-Gibbs-Sampling-Iter($\boldsymbol{w}_d$, $\boldsymbol{z}_d$, N_{tdp}, N_{wtp}, α, β);
  Синхронизация;
  Обновление глобальных счётчиков $N_{wt} := N_{wt} + \sum_{p\in\{1,\dots,P\}}(N_{wtp} - N_{wt})$;
до тех пор, пока *не выполнен критерий останова*;

2 Алгоритм AD-LDA

§2.1 Описание

Approximate Distributed LDA (AD-LDA) был предложен в [2]. Алгоритм основан на коллапсированной схеме Гиббса и предназначен для распараллеливания по ядрам.

Сложность параллелизма заключается в том, что сэмплирование Гиббса — сугубо последовательный процесс, его параллельная реализация в общем случае приводит к нарушениям необходимых требований к марковской цепи, что влечёт за собой генерацию выборки из неверного распределения. Поэтому обновление любого $z_{d,n}$ не может производиться одновременно с обновлением любого другого z_{d^1,n^1}. Но в случае, когда число слововхождений сильно превосходит число процессоров, на которых предполагается параллельная обработка, обновления $z_{d,n}$ и z_{d^1,n^1} будут слабо влиять друг на друга. Значит, можно ослабить требование последовательности сэмплирования и получить неплохую модель [1] .

В AD-LDA коллекция D распределяется по P процессорам примерно в равных пропорциях. Документы распределяются случайным образом, без предварительной кластеризации. Слововхождения $\mathcal{W}$ и соответствующие им $\mathcal{Z}$ разделяются на P частей и распределяются по процессорам в соответствии со своим d. Счетчики N_{td} также распределены (обозначим N_{tdp}). Аналогично каждый процессор имеет свою локальную копию N_{wtp} глобальных счётчиков N_{wt}.

Псевдокод AD-LDA показан на листинге 2.1. После распределения данных и параметров по процессорам, алгоритм производит одновременное сэмплирование на каждом из них. Когда процессор обработал свои данные и обновил локальные присваивания тем $\boldsymbol{z}_p$, он обновляет локальные счётчики N_{tdp} и N_{wtp}. Как видно, счётчики темы-документы N_{tdp} не пересекаются и соответствуют глобальной $\mathcal{Z}$ в силу того, что документы без повторений разнесены по ядрам-обработчикам. А вот с счётчиками N_{wtp} ситуация совершенно иная. В общем случае они будут отличаться для различных p и не будут соответствовать глобальной $\mathcal{Z}$. Для того, чтобы слить все результаты с разных процессоров и привести локальные счётчики в соответствие глобальному состоянию модели, производится операция сбора всех счётчиков N_{wtp}. Полученные результаты прибавляются к глобальному N_{wt}, после чего производится

[1] Качество тематической модели обычно оценивается т.н. *перплексией*, определяемой как $\mathcal{P}(D;p) = \exp(-\frac{1}{N}L(\Phi,\Theta)) = \exp(-\frac{1}{N}\sum_{d\in D}\sum_{w\in d}N_{dw}\ln p(w|d))$ (L — логарифм правдоподобия). Эта величина характеризует степень «удивлённости» модели словам нового документа, чем она меньше, тем лучше.

его копирование на все процессоры для обработки следующей порции данных. Важно учитывать, что на каждом процессоре хранится полная матрица счётчиков N_{wt}, т.е $\sum_{(w,t)} N_{wtp} = N, \quad \forall p \in P$.

§2.2 Эксперименты

Производительность AD-LDA была протестирована на коллекциях pubmed и wikipedia, каждая из которых содержит по несколько миллионов документов. Результаты показаны на рис. 2. Реализация была сделана на суперкомпьютере с помощью MPI. Эксперименты на качество построения модели показали, что AD-LDA строит модель с такими же скоростью убывания и финальным значением перплексии, что и обычный LDA.

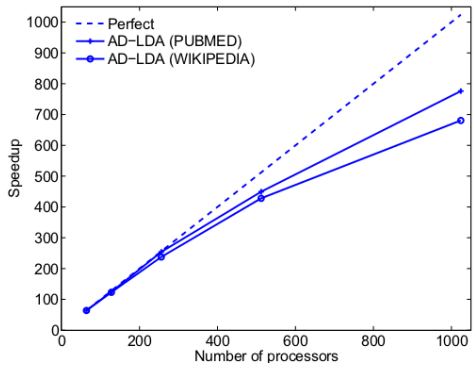

Рис. 2. Масштабируемость по числу процессоров MPI-реализации AD-LDA на кластере.

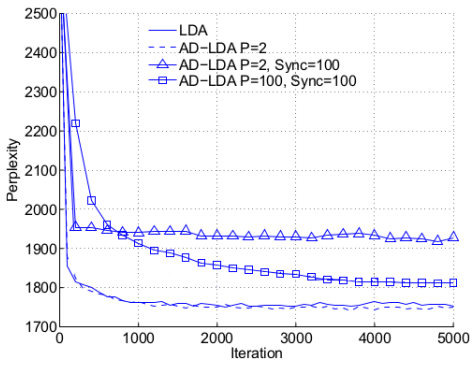

Рис. 3. Сходимость AD-LDA при построении модели коллекции KOS с 16 темами при разных P и частотах синхронизации.

§2.3 Выводы

AD-LDA послужил отправной точкой для многих последующих параллельных реализаций LDA [2] . Но сам по себе он имеет ряд серьёзных недостатков. Получающиеся модели являются хорошими только в том случае, если синхронизации производятся достаточно часто и используется как можно большее число процессоров. В противном случае алгоритм может сойтись к неоптимальному решению. На рис. 3 показана перплексия модели на небольшой коллекции KOS с $|T| = 16$. Видно, что небольшое число процессоров при редких синхронизациях дают плохое качество результирующей модели в сравнении с обычным LDA. Отсюда следует, что синхронизироваться нужно как можно чаще, в идеале — после каждой итерации сэмплирования Гиббса. Но это приводит к тому, что скорость работы всей совокупность ядер определяется самым медленным из них. Кроме того, во время итераций сеть простаивает, а во время синхронизаций — перегружена. Всё это сказывается на производительности и масштабируемости. Кроме того, серьёзным недостатком AD-LDA

[2] По схожим принципам реализован известный пакет для тематического моделирования Mallet (http://www.cs.umass.edu/mccallum/mallet).

является большое потребление памяти — копия глобальных счётчиков N_{wt} хранится на каждом ядре (а их в экспериментах было до 1024).

3 Алгоритм Y!LDA

§3.1 Описание

Следующий рассматриваемый алгоритм называется Y!LDA. Он был описан в [3]. Эта реализация так же основана на коллапсированной схеме Гиббса, но она, как мы увидим далее, имеет целый ряд преимуществ по сравнению с AD-LDA. У Y!LDA есть две ключевые особенности реализации — асинхронная обработка данных в пределах одного нода и т.н. *архитектура классной доски* для взаимодействия разных нодов кластера. Рассмотрим каждую из них подробнее.

3.1.1 Мультипроцессорный параллелизм

Основной идеей, допускающей параллелизм, является отмеченное ранее предположение о том, что глобальное (в рамках нода) состояние, представленное счётчиками N_{wt} [3] слабо изменится во время сэмплирования тем для одного документа. Отсюда следует, что добавление обновлений, получаемых в процессе обработки этого документа, можно отложить до конца его полной обработки. А это означает, что процесс обновления можно вообще делегировать отдельному потоку, общему для всех потоков-сэмплеров в пределах одного нода. Кроме того, в Y!LDA глобальное состояние храниться не на каждом ядре, а является общим для всех ядер нода.

На рис. 4 показана схема работы сэмплеров. Слова $\boldsymbol{w}_d$ и соответсвующие им присваивания тем $\boldsymbol{z}_d$, для всех d, обрабатываемых на этом ноде, хранятся отдельно и сливаются в первом фильтре. Полученные документы обрабатываются параллельно потоками-сэмплерами. Поскольку таблица N_{wt} общая, сэмплер инициирует блокировку на чтение нужного ему значения, когда хочет получить к нему доступ. После обработки всего документа, полученные инкременты отправляются в общий поток, обновляющий счётчики. [4] Во время обработки новых документов можно производить диагностику текущей модели (например, считать перплексию). Наконец, последний фильтр записывает новые значения $\boldsymbol{z}_d$ в файл.

3.1.2 Кластерный параллелизм

Основной проблемой при реализации LDA на кластере является синхронизация глобальной (в рамках кластера) таблицы N_{wt} между всеми нодами-обработчиками. Одним из способов решения этой проблемы может быть синхронизация всех локальных счётчиков после каждого прохода по коллекции. Такой подход плох тем, что время обработки будет определяться самым медленным сэмплером. Альтернативой

[3] На самом деле, помимо N_{wt}, на ноде хранится также набор счётчиков N_t, получаемый из первых суммированием по w. Стоит отметить, что на уровне кластера эти счётчики не хранятся, а вычисляются непосредственно по новым N_{wt}.

[4] Может возникнуть сомнение в том, что один поток справится с обновлениями от всех сэмплеров, но авторы алгоритма утверждают, что обновление глобального состояния для нода — достаточно легковесная операция. Однако однопоточная реализация не является принципиальной, допустима и многопоточная.

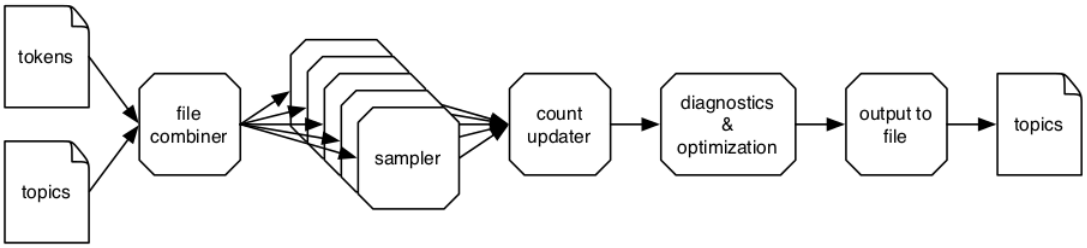

Рис. 4. Схема обработки данных в Y!LDA.

Алгоритм 3.1: Алгоритм синхронизации состояний в Y!LDA

1 Инициализировать $N_{wt} = N_{wt}^i = N_{wt}^{i\ old}$, для всех нодов i;
2 **до тех пор, пока** *производится сэмплирование* **выполнять**
3 Заблокировать глобально N_{wt} для некоторого слова;
4 Заблокировать локально N_{wt}^i для данного слова;
5 Обновить глобальное состояние: N_{wt} := N_{wt} + (N_{wt}^i - $N_{wt}^{i\ old}$);
6 Обновить локальное состояние: $N_{wt}^{i\ old} = N_{wt}^i = N_{wt}$;
7 Разблокировать N_{wt}^i;
8 Разблокировать N_{wt};

является упомянутая ранее архитектура классной доски. Суть её в следующем: глобальная таблица счётчиков N_{wt} хранится в единственном экземпляре и является общей для всех нодов, её обновление производится сэмплерами асинхронно по одному слову w за один раз.

Предположим, что в некоторый момент времени глобальные кластерные счётчики и локальные счётчики нодов находятся в одном и том же состоянии (о том, как этого добиться, будет написано ниже в 3.1.4). Пусть N_{wt} — глобальное состояние, N_{wt}^i — текущее локальное состояние и $N_{wt}^{i\ old}$ — копия локального состояния на момент последней синхронизации с N_{wt}. Тогда алгоритм, описанный в листинге 3.1 будет поддерживать синхронизацию счётчиков. Он периодически вносит локальные изменения в глобальное состояние, после чего приводит локальные счётчики в соответствие глобальным. Поскольку единицей обновления является слово, алгоритм практически не вносит дедлоков в работу сэмплеров.

3.1.3 Технические детали реализации

- Все данные внутри программы реализуются и передаются с помощью технологии `Google protocol buffers` [5]. Она позволяет описывать структуры данных на псевдо-языке, компилировать его в код на C++/Python/Java, сериализовывать/десериализовывать эти структуры.

- Для хранения глобального N_{wt} используется `memcached` [6] — сервис, реализующий в оперативной памяти хранилище на основе хеш-таблицы. Данные хранятся в виде списка (тема, счётчик) для каждого слова w в порядке убывания

[5] http://code.google.com/p/protobuf/
[6] http://www.danga.com/memcached/

значений счётчиков. Это позволяет реализовывать сэмплеры эффективнее — наиболее редкие слова с большой вероятностью не будут достигаться. Обновления, приходящие от потоков-сэмплеров к потоку обновлений, имеют вид (слово, id старой темы, id новой темы). Если тема не изменилась, то обновление можно производить вообще без блокировок за счёт атомарности обновления 32-хбитных целых чисел на современных x86-архитектурах.

- Алгоритм был реализован на кластере рабочих станций с сервером и на Hadoop-кластере с машинами аналогичной мощности.

3.1.4 Инициализация и восстановление после сбоев

В момент инициализации никаких присваиваний тем $\mathcal{Z}$ нет, их нужно откуда-то получить. Можно произвести случайное присваивание, задействовав сэмплеры. Однако лучшим вариантом является инициализация моделью, обученной на небольшом подмножестве документов из D. По аналогичной схеме производится и процедура восстановления в случае сбоя: система просто загружает последние присваивания тем из файла и производит «инициализацию» модели ими.

Вспомним, что при разговоре о кластерном параллелизме было потребовано синхронное состояние системы в некоторый момент времени для того, чтобы произвести инициализацию. Для этого предлагается следующий протокол:

1. Локальная инициализация (шаг 0). Проверка, что у каждого нода есть список IP-адресов всех остальных нодов. Список всех нодов храниться в `memcached` в виде пар (IP-адрес, «текущий шаг протокола для данного нода»). Каждый сэмплер строит для себя локальную N_{wt}.

2. Объединение счётчиков (шаг 1). Производится объединение всех локальных статистик в глобальной таблице. Для этого текущая машина для каждого слова w из своего словаря блокирует счётчики N_{wt} для этого слова в `memcached`, увеличивает их на свои локальные значения и производит разблокировку. Тут нужно учесть, что индексирование производится по строковым представлениям слов, а не по индексам, поскольку локальные индексы на разных нодах не совпадают с глобальными, а синхронизировать их неудобно и накладно.

3. Локальная синхронизация (шаг 2). После выполнения шага 2 нод ожидает, пока этот шаг будет пройден всеми остальными машинами. После этого все сэмплеры производят обновление своих счётчиков из агрегированной глобальной таблицы.

4. Сэмплирование (шаг 3). После обновления счётчиков нод начинает сэмплирование. Стоит отметить, что для этого ему не требуется ожидать окончания шага локальной синхронизации всех остальных нодов.

5. Завершение работы алгоритма (шаг 4). Терминальное состояние сэмплера.

§3.2 Выводы

Алгоритм выглядит гораздо лучше предыдущего. Потребление памяти существенно сократилось. Локальное состояние всегда остаётся синхронизированным безо всяких выделенных шагов синхронизации. Это увеличивает скорость обработки и эффективность использования ресурсов нодов и сети.

4 Алгоритм Mr. LDA

§4.1 Описание

Реализация Mr. LDA описана в [4]. Она имеет два существенных отличия от предыдущих алгоритмов. Во-первых, в качестве метода вывода используется вариационный вывод, а не сэмплирование Гиббса. Во-вторых, Mr. LDA реализован в рамках парадигмы MapReduce на Hadoop.

4.1.1 Почему вариационный вывод?

Вариационный вывод имеет некоторые примущества по сравнению с MCMC-методами. Это детерминированность результата, более высокая скорость сходимости в многомерных пространствах. Параллельное коллапсированное сэмплирование Гиббса требует постоянной синхронизации состояний. Если синхронизации редкие, это приводит к ухудшению качества, если частые — к затратам времени. Это усугубляется тем, что при сэмплировании производится очень много небольших итераций, после каждой из которых желательна синхронизация. В противоположность этому, вариационный вывод даёт математическое решение того, как нужно масштабировать процесс обучения: вариационное приближение позволяет считать документы независимыми и их вывод можно осуществлять параллельно. При использовании вариационного вывода итерации более сложные, но их гораздо меньше, и синхронизации нужны только между ними. Кроме того, вариационный EM-алгоритм хорошо вписывается в MapReduce. Это сразу и бесплатно даёт отказоустойчивость — алгоритм можно перезапустить с последней успешной итерации и получить детерминированный результат.

4.1.2 Общая схема MapReduce

Все обозначения параметров описаны в формуле 1.1. В алгоритме присутствуют три вида компонентов — mapper, reducer и driver [7] . Схема обработки следующая. На каждый документ коллекции создаётся mapper. Его задача — вычисление обновлений для связанных с документом гиперпараметров $\boldsymbol{\mu}_d$ и $\boldsymbol{\gamma}_d$. Каждый mapper даёт на выходе три вида пар (ключ, значение). Первый вид — обновления, требуемые для reducer, второй — величины, необходимые для последующего обновления гиперпараметра α. Последний вид — обновлённый параметр $\boldsymbol{\gamma}_d$ — пишется в файл. В качестве ключей используются номера тем. Далее, на каждую тематику создается reducer.

[7] С технической точки зрения нужно отметить, что используются так же и combiner с partitioner, но их роли сводятся к дополнительной агрегации промежуточных данных для повышения производительности reducer-ов.

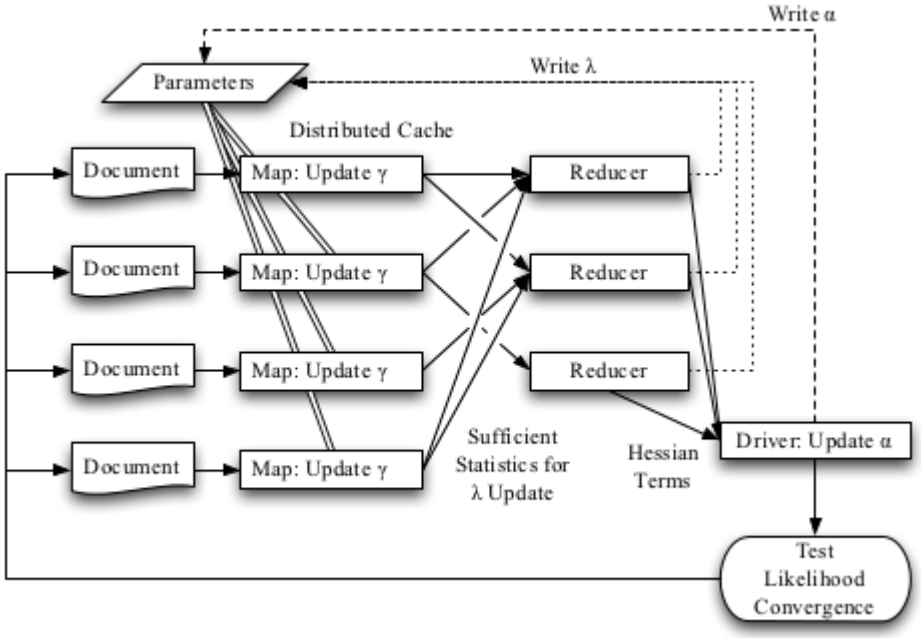

Рис. 5. Схема обработки данных в Mr. LDA.

В его задачи входит пересчёт ассоциированного с темой t параметра $\boldsymbol{\lambda}_t$ с помощью обновлений, посчитанных mapper-ами. Третий компонент — driver — присутствует в системе в единственном экземпляре и управляет всем процессом обучения. Кроме этого, он осуществляет оптимизацию гиперпараметра α, тоже используя результаты работы mapper-ов, и вычисляет вариационную нижнюю оценку обоснованности.

Следует отметить, что глобальные параметры $\boldsymbol{\lambda}$ и α хранятся в специальной, доступной только для чтения и общей для всех mapper-ов памяти, называемой *распределённым кэшем*.

Вся описанная схема графически проиллюстрирована на рис. 5.

4.1.3 Структурные модификации алгоритма

Экспериментальные результаты показали, что производительность Mr. LDA ограничивается двумя существенными факторами: огромным количеством генерируемых reducer-ами промежуточных значений и временем, которое mapper-ам требуется для чтения текущих вариационных параметров перед началом своей работы.

Одним из возможных улучшений является *кэширование выхода* reducer. Алгоритм устроен таким образом, что reducer просто генерирует новые значения вариационных параметров $\boldsymbol{\lambda}_t$, в то время как mapper требует нормированные значения, и поэтому вынужден считать сумму $\boldsymbol{\lambda}_t$ по всем словам для нормировки. Вместо этого разумнее посчитать сумму прямо во время вычисления её компонентов, на reducer, после чего сохранить её в распределённый кэш. Это увеличивает расходы памяти reducer, однако, эти затраты гораздо меньше того объёма памяти, который тратится mapper-ми на операцию суммирования.

Другой модификацией является *слияние файлов*. Большое число reducer-ов порождает много небольших выходных данных, размер каждого из которых существенно меньше блока HDFS, что приводит к засорению памяти. Для борьбы с этим можно

сливать релевантные выходные данные до отправки их в распределённый кэш для следующей итерации.

§4.2 Выводы

Mr. LDA обладает рядом достоинств: оптимизация гиперпараметров, высокая скорость и качество работы в сравнении с Mahout [8] — другим популярным фреймворком такого типа. Алгоритм является расширяемым — примером модификации является мультиязычная модель LDA. Тем не менее нельзя не отметить, что MapReduce на Hadoop плохо подходит для реализации итерационных алгоритмов, во-первых — из-за своей негибкости, во-вторых — из-за необходимости частой работы с относительно медленной постоянной памятью.

5 Библиотека BigARTM

В этом разделе описан фреймворк, основанный на модели *вероятностного латентного семантического анализа* [5] (PLSA) и теории *аддитивной регуляризации тематических моделей* [6, 7] (ARTM). Прежде, чем перейти к техническим деталям, рассмотрим необходимые теоретические сведения.

§5.1 ARTM

Примем гипотезу условной независимости, которая гласит, что появление термина в документе зависит только от тематики этого термина и не зависит от документа. Тогда вероятностная модель PLSA будет иметь вид [9] :

$$p(\mathcal{W}, \mathcal{Z}|\Phi, \Theta) = \prod_{d=1}^{|D|}\prod_{n=1}^{N_d} p(w_{d,n}|z_{d,n}, \Phi)p(z_{d,n}|\Theta) = \prod_{d=1}^{|D|}\prod_{n=1}^{N_d} \varphi_{w_{d,n},z_{d,n}}\theta_{z_{d,n},d}.$$

Задача сводится к поиску разложения матрицы вероятностей слов в документах в произведение Φ и Θ. Она является некорректно поставленной и должна быть регуляризована. В исходной задаче максимизировалось правдоподобие, добавим к нему взвешенную сумму n критериев, называемых *регуляризаторами* и отражающих требования, накладываемые на матрицы Φ и Θ. Решение этой задачи максимизации приводит к обновлённым формулам для вычисления элементов этих матриц. LDA является частным случаем этой модели с регуляризаторами сглаживания.

Обучение можно производить обычным EM-алгоритмом, ARTM добавит в его формулы M-шага новые слагаемые, получающиеся в результате применения регуляризации. Но для обработки больших коллекций лучше использовать *пакетный онлайновый* вариант алгоритма. В нём пересчёт $\boldsymbol{\theta}_d$ вынесен в E-шаг и обработка документов производится по-пакетно. Такому алгоритму достаточно одного прохода по всей коллекции и нескольких итераций по каждому документу.

[8] http://mloss.org/software/view/144

[9] Для PLSA удобнее считать, что $\Phi \in \mathbb{R}^{|W|\times|T|}$ и $\Theta \in \mathbb{R}^{|T|\times|D|}$

§5.2 Архитектура библиотеки

BigARTM — библиотека для эффективной онлайновой параллельной обработки прежде всего в пределах одной машины [10] . Одной из основных целей, преследуемых при проектировании фреймворка, было достижение независимости объёма используемой оперативной памяти от размера обрабатываемой коллекции. Достигается это следующим образом — вся коллекция D делится на части (*батчи*) D_b, сохраняется на диск в отдельных файлах, и в каждый момент времени в памяти находится только часть из них. Кроме того, вся матрица Θ никогда не хранится в памяти, онлайн алгоритм позволяет в данный момент времени хранить только те её куски, которые соответствуют обрабатываемым документам, а после окончания обработки удалять их.

Параллелизм BigARTM основан на многопоточности. Общую схему можно увидеть на рис. 6. Существует множество *обработчиков* (Processor) и один *поток слияния* (Merger). Задача первых — производить параллельную обработку батчей, выводя элементы матрицы Θ и рассчитывая инкременты для матрицы Φ, второго — получать от первых асинхронно эти инкременты и обновлять матрицу Φ (похожий подход уже наблюдался в Y!LDA). Для обработчиков существует т.н. *очередь заданий*, в которых хранятся подгруженные библиотекой в память батчи. Оттуда обработчики берут задания. Так же существует *очередь слияния*, в которую обработчики отправляют по мере необходимости обновления для матрицы Φ. В свою очередь, поток Merger извлекает из этой очереди данные, которые нужны ему для работы. Тут важно учитывать, что все данные хранятся в памяти статично и не копируются, «перемещения» производятся с помощью указателей. В каждый момент времени Merger хранит две копии матрицы Φ — *базовую* и *активную*. Первая из них является доступной для чтения обработчикам, вторая — всегда доступна для записи потоку слияния. Каждый обработчик перед началом работы берёт себе указатель на текущую активную матрицу Φ и использует её для вывода векторов в $\boldsymbol{\theta}_d$. А Merger, получая обновления из очереди слияния, обновляет базовую матрицу. Как только все текущие инкременты были к ней применены, она становится активной, и Merger создаёт новую базовую матрицу [11]. Обновление матрицы Φ (синхронизацию) можно инициировать в любой момент из пользовательского кода.

5.2.1 Технические детали реализации

- Все данные внутри программы реализуются и передаются с помощью технологии `Google protocol buffers`, как это делалось в Y!LDA.

- Код ядра написан на C++, для организации многопоточности (и многого другого) используются библиотеки `Boost C++ Libraries` [12].

[10] Возможность организации обработки на кластере имеется и на текущий момент реализована с помощью механизма rpc. Однако в будущем предполагается использование специализированных средств, таких как Hadoop, Spark и т.д.

[11] На самом деле, в некоторые моменты времени существует *три* матрицы Φ — старая активная, которая существует до тех пор, пока есть обработчики, использующие её, новая активная, которая передаётся новым обработчикам и базовая.

[12] www.boost.org/

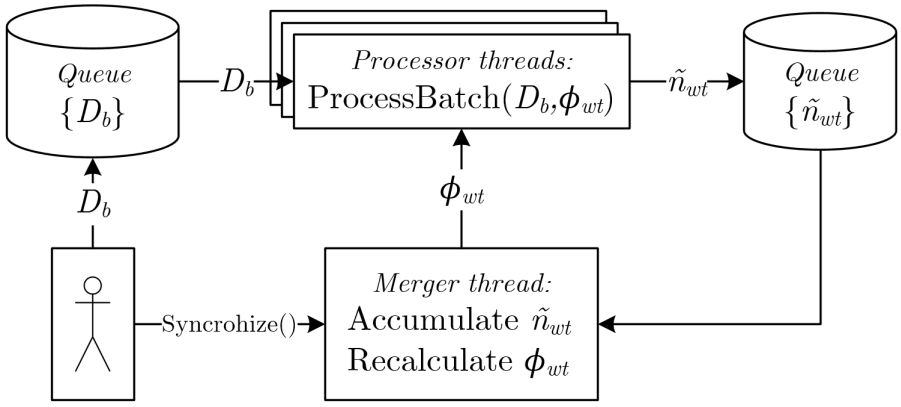

Рис. 6. Схема обработки данных в BigARTM.

§5.3 Возможности и расширяемость BigARTM

Как показывают эксперименты, BigARTM позволяет очень эфективно обрабатывать данные (производительность выше, чем у популярных фреймворков VW.LDA и Gensim [см. Приложение A]), получая на выходе хорошие модели. Поддержка ARTM организована на модульном принципе, т.е. пользователь всегда может самостоятельно написать тот регуляризатор, который ему нужен, и включить его в библиотеку. Аналогичная ситуация с функционалами качества тематической модели. У библиотеки есть пользовательские API на C++ и Python, которые позволяют писать программы, очень тонко описывающие и постоянно контролирующие процесс обучения.

Также BigARTM поддерживает т.н. *мультимодальные тематические модели* (MTM), которые позволяют использовать мета-информацию, связанную с документами, строить мультиязычные модели. В рамках теории MTM расписываются многие сложные байесовские модели, основанные на LDA.

§5.4 Выводы

При разработке архитектуры BigARTM был учтён опыт прошлых реализаций. В целом, библиотека является эффективным инструментом для тематического моделирования на одном компьютере. В будущем планируется сделать его настолько же эффективным для кластера. Основным её функциональным преимуществом является поддержка ARTM и MTM.

Список литературы

[1] *D. M. Blei, A. Ng, and M. Jordan.* (2003). Latent Dirichlet allocation, // Journal of Machine Learning Research, vol. 3, pp. 993–1022.

[2] *D. Newman, A. Asuncion, P. Smyth, and M. Welling.* (2009). Distributed algorithms for topic models, // NIPS.

[3] *Alexander Smola and Shravan Narayanamurthy.* (2010). An architecture for parallel topic models, // VLDB.

[4] *Ke Zhai, Jordan Boyd-Graber, Nima Asadi, Mohamad Alkhouja.* (2012). Mr. LDA: A Flexible Large Scale Topic Modeling Package using Variational Inference in MapReduce, // ACM.

[5] *T. Hofmann..* Probabilistic latent semantic indexing, // Proceedings of the 22nd annual international ACM SIGIR conference on Research and development in information retrieval. — New York, NY, USA: ACM, 1999. — Pp. 50–57.

[6] *Воронцов К. В..* Аддитивная регуляризация тематических моделей коллекций текстовых документов, // Доклады РАН. 2014. — Т. 455., No3. 268–271.

[7] *Vorontsov K. V., Potapenko A. A.* (2014). Additive Regularization of Topic Models, // Machine Learning Journal. Special Issue «Data Analysis and Intelligent Optimization with Applications».

[8] *Matthew D. Hoffman, David M. Blei, Francis Bach.* (2010). Online Learning for Latent Dirichlet Allocation, // NIPS.

A Приложение

Несмотря на то, что алгоритм LDA в составе библиотеки Vowpal Wabbit [13] (VW.LDA) не является ни параллельным, ни распределенным, его имеет смысл упомянуть, поскольку эта реализация является одной из самых распространенных и часто используемых для обработки больших данных.

В основе VW.LDA лежит онлайновый LDA [8], который принципиально ничем не отличается от онлайнового PLSA, о котором говорилось ранее. Эффективная реализация этого алгоритма на C++ без использования STL и сторонних библиотек сделали VW.LDA производительным инструментом для тематического моделирования, позволяющим на одной машине обрабатывать в потоковом режиме большие коллекции документов.

Пакет для тематического моделирования Gensim [14], аналогично, основан на онлайновом LDA. Однако в отличии от VW.LDA, он поддерживает как обычный однопоточный вариант алгоритма (LdaModel), так и многопоточную реализацию (LdaMulticore). Также имеется возможность запуска этого фреймворка на кластере. Параллельная архитектура во много схожа с той, которая реализована в BigARTM. Реализован на `Python`. Gensim является активно используемой библиотекой для тематического моделирования, его скорость обработки считается высокой.

Тем не менее, сравнительные эксперименты на коллекции Wikipedia [15] на машине с 8-ю ядрами с $|T| = 100$, результаты которых приведены в таблице ниже, показали, что BigARTM производительнее как VW.LDA, так и Gensim [16] .

Библиотека	Число процессоров	Время обучения модели
BigARTM	1	35 минут
LdaModel	1	369 минут
VW.LDA	1	73 минуты
BigARTM	4	9 минут
LdaMulticore	4	60 минут
BigARTM	8	4.5 минуты
LdaMulticore	8	57 минут

[13] https://github.com/JohnLangford/vowpal_wabbit/wiki/Latent-Dirichlet-Allocation

[14] https://radimrehurek.com/gensim/

[15] Коллекция статей английской Википедии, $|D| \approx 3.7 \times 10^6$.

[16] BigARTM запускался без регуляризации. Качество итоговых моделей с точки зрения тестовой перплексии для всех случаев было примерно равным.