Машинное обучение и большие данные

Эффективные алгоритмы обучения тематических моделей

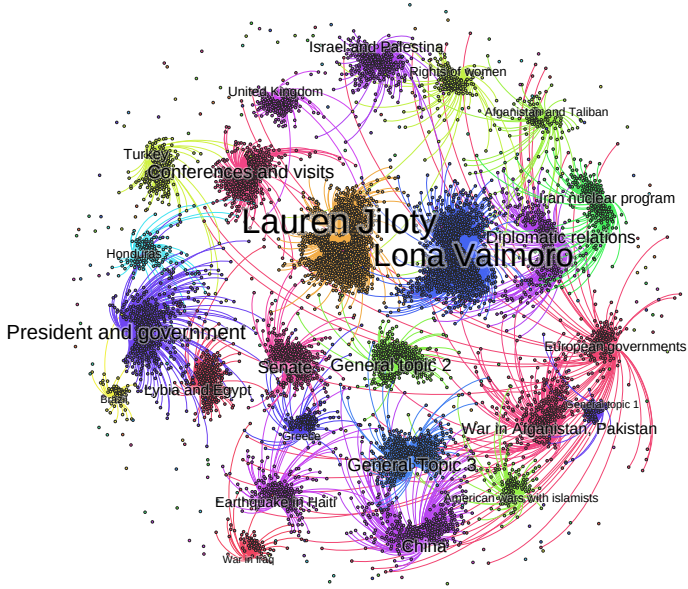
Мурат Апишев (mel-lain@yandex.ru)

Декабрь, 2020

Тематическое моделирование

- Тематическое моделирование (Topic Modeling) — приложение машинного обучения к статистическому анализу текстов
- Тема — терминология предметной области, набор терминов (униграм или $n-$грам) часто встречающихся вместе в документах
- Тематическая модель исследует скрытую тематическую структуру коллекции текстов:
  - тема t — это вероятностное распределение $p(w|t)$ над терминами w
  - документ d — это вероятностное распределение $p(t|d)$ над темами t
- В тематическом моделировании производится би-кластеризация текстовой коллекции
- Тема неформально описывается набором слов, которые относятся к одной предметной области

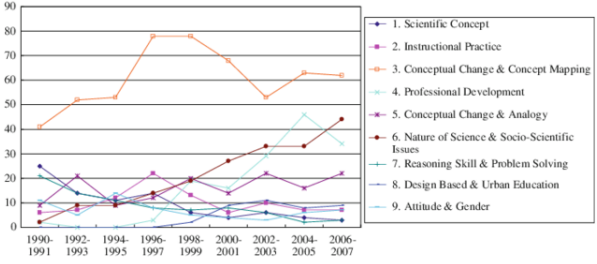
Базовая задача: выявление тематической структуры



Основные приложения тематических моделей

- Векторный семантический поиск (разведочный поиск)

- Трендовая аналитика, анализ новостных потоков



- Рубрикация и категоризация документов

http://www.machinelearning.ru/wiki/images/1/12/YaninaMMPR201929.pdf
Y. Chang, C. Chang, Y. Tseng. Trends of Science Education Research: An Automatic Content Analysis, 2010

Обозначения и гипотезы

- Пусть дана коллекция документов D со словарём W
- Обозначим за T искомое множество тем
- Объединим распределения $p(w|t)$ в матрицу Φ, а $p(t|d)$ — в Θ
- Матрицы Φ и Θ — стохастические, их столбцы — вероятностные распределения ϕ_t и θ_d соответственно
- Порядок слов в документе не имеет значения («мешок слов»)
- Порядок документов в коллекции не имеет значения
- Появление каждого слова в документе связано с некоторой темой $t \in T$
- Коллекция D — это выборка троек $(d_i, w_i, t_i)_{i=1}^n \sim p(d, w, t)$ в пространстве $D \times W \times T$
- n_{dw} — число вхождений слова w в документ d
- Гипотеза условной независимости: $p(w|t, d) = p(w|t)$

Вероятностная модель коллекции текстов

- Запишем модель с учётом гипотезы условной независимости:

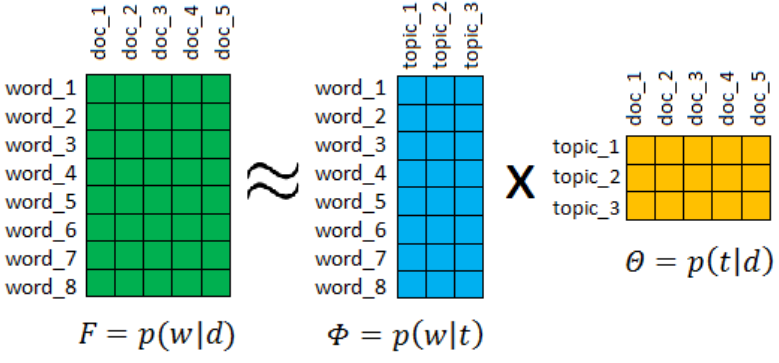
$$p(w|d) = \sum_{t \in T} p(w|t,d)\, p(t|d) = \sum_{t \in T} p(w|t)\, p(t|d) = \sum_{t \in T} \phi_{wt}\theta_{td}$$

- Она описывает процесс появления слов в документах темами
- Пусть распределения известны ϕ_{wt}, $\forall t \in T$ и θ_{td}, $\forall d \in D$
- Из них можно сгенерировать каждое слово в каждом документе коллекции:
  1. для позиции слова в документе d сэмплируем тему t из θ_d
  2. сэмплируем из распределения ϕ_t для этой темы итоговое слово w
- Тематические моделирование решает обратную задачу — по распределениям $p(w|d)$ восстановить неизвестные ϕ_{wt} и θ_{td}

Модель PLSA

- Вероятностная модель коллекции текстов:

$$p(w|d) = \sum_{t \in T} p(w|t)\, p(t|d) = \sum_{t \in T} \phi_{wt} \theta_{td}$$



- Вероятностный латентный семантический анализ (Probabilistic Latent Semantic Analysis) — исторически первая тематическая модель

Оптимизационная задача PLSA

- Задачу поиска параметров распределений по выборке можно решать методом максимального правдоподобия (MLE):

$$\prod_{i=1}^{n} p(w_i, d_i) = \prod_{i=1}^{n} p(w_i|d_i)p(d_i) = \prod_{d\in D}\prod_{w\in W} p(w|d)^{n_{dw}} p(d)^{n_{dw}} \to \max_{\Phi,\Theta}$$

- Прологарифмируем выражение и выбросим константы:

$$\sum_{d\in D}\sum_{w\in W} n_{dw} \ln p(w|d) = \sum_{d\in D}\sum_{w\in W} n_{dw} \ln \sum_{t\in T} \phi_{wt}\theta_{td} \to \max_{\Phi,\Theta}$$

- Добавим к результирующему функционалу ограничения на параметры:

$$\sum_{w\in W} \phi_{wt} = 1,\ \phi_{wt} \geqslant 0; \qquad \sum_{t\in T} \theta_{td} = 1,\ \theta_{td} \geqslant 0;$$

- Получили итоговую постановку задачи PLSA

EM-алгоритм

- Полученная задача оптимизации является невыпуклой, можем искать только локальный экстремум
- Обычный подход в MLE с латентными переменными — EM-алгоритм
- **Идея:** вместо прямого вычисления параметров модели, введём вспомогательные переменные, такие, что:
  - Их легко посчитать через параметры модели
  - Параметры модели легко посчитать через них
- Имея гарантии сходимости, можно организовать итерационный процесс:
  - E-шаг (expectation) — считаем переменные через параметры
  - M-шаг (maximization) — считаем парметры через переменные
- Схожий подход используется в задаче разделения смеси гауссиан:
  - **Параметры:** параметры распределений-компонент и их веса
  - **Переменные:** вероятность отнесения объекта к компоненте

Интуиция EM-алгоритма для PLSA

- Пусть дана коллекция D из одного документа d, ищем ϕ_{wt} и θ_{td}
- Допустим, что кроме текста есть ещё темы, присвоенные каждому слову:

Pooh rubbed his nose again, and said that he hadn't thought of that. And then he brightened up, and said that, if it were raining already, the Heffalump would be looking at the sky wondering if it would clear up, and so he wouldn't see the Very Deep Pit until he was half-way down...

- В этом случае значения $\phi_{wt} = p(w|t)$ и $\theta_{td} = p(t|d)$ легко посчитать:

$$p(w = sky|t) = \frac{n_{wt}}{\sum_w n_{wt}} = \frac{1}{4} \qquad p(t = t|d) = \frac{n_{td}}{\sum_s n_{sd}} = \frac{4}{54}$$

- Здесь
  - n_{wt} — сколько раз слово w в коллекции было отнесено к теме t
  - n_{td} — сколько слов документа d были отнесены к теме t

Интуиция EM-алгоритма для PLSA

- Если известная тема для каждого слова, то оценить параметры легко
- Но в общем случае на вход модели даётся только текст
- Вероятность $p_{tdw} \equiv p(t|d,w)$ отнесения слова w к теме t в документе d можно оценить
- Пусть даны какие-то (даже случайные) значения параметров Φ и Θ
- Тогда значения p_{tdw} можно посчитать по формуле

$$p(t|d,w) = \frac{p(w,t|d)}{p(w|d)} = \frac{p(w|d,t)p(t|d)}{p(w|d)} = \frac{p(w|t)p(t|d)}{p(w|d)} = \frac{\phi_{wt}\theta_{td}}{\sum_{s\in T}\phi_{ws}\theta_{sd}}$$

- Осталось собрать всё вместе

EM-алгоритм для модели PLSA

$$\sum_{d\in D}\sum_{w\in d} n_{dw}\ln\sum_{t\in T}\phi_{wt}\theta_{td} \to \max_{\Phi,\Theta}$$

$$\sum_{w\in W}\phi_{wt}=1,\ \phi_{wt}\geqslant 0; \qquad \sum_{d\in D}\theta_{td}=1,\ \theta_{td}\geqslant 0$$

Теорема: *если* (Φ,Θ) *является точкой локального максимума, то выполняется система уравнений (с исключением нулевых* ϕ_t *и* θ_d*)*:

$$p_{tdw}=\underset{t\in T}{\text{norm}}\big(\phi_{wt}\theta_{td}\big) \Rightarrow \quad \begin{aligned} n_{wt}&=\textstyle\sum_{d\in D} n_{dw}p_{tdw} \\ n_{td}&=\textstyle\sum_{w\in d} n_{dw}p_{tdw}\end{aligned} \qquad \text{(E-шаг)}$$

$$\phi_{wt}=\underset{w\in W}{\text{norm}}\big(n_{wt}\big) \qquad \theta_{td}=\underset{t\in T}{\text{norm}}\big(n_{td}\big) \qquad \text{(M-шаг)}$$

где $\underset{i\in I}{\text{norm}}(x_i)=\dfrac{\max\{0,x_i\}}{\sum\limits_{j\in I}\max\{0,x_j\}}$, *для всех* $i\in I$

EM-алгоритм для модели PLSA

- Алгоритм многократно проходит по коллекции D
- Перед стартом задаются случайные приближения параметров Φ и Θ
- По ним на E-шаге вычисляются вспомогательные переменные, из которых на лету формируются счётчики n_{wt} и n_{td}
- Нормировка этих счётчиков на E-шаге даёт новые значения параметров
- Процесс продолжается до сходимости оптимизируемого функционала
- Конкретное значение локального максимума существенно зависит от начальных приближений Φ и Θ
- Часто пробуют запускать обучение модели несколько раз и выбирают наилучший результат с точки зрения выбранной метрики

Метрики качества моделирования

- Часто вместо значения лог-правдоподобия используется перплексия — монотонная функция от него:

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

- Средняя когерентность тем в модели оценивает её интерпретируемость:

$$\mathcal{C}(\Phi) = \frac{1}{|T|}\sum_{t\in T}\mathcal{C}_t(\Phi), \quad \mathcal{C}_t(\Phi) = \frac{2}{k(k-1)}\sum_{i=1}^{k-1}\sum_{j=i+1}^{k}\mathrm{PPMI}(w_i, w_j),$$

где

- w_i — i-й терм в списке из k топ-термов в распределении темы t
- $\mathrm{PPMI}(u,v) = \max\left\{0, \ln\frac{|D|N_{uv}}{N_u N_v}\right\}$
- N_{uv} — число документов, которые содержат оба терма u и v
- N_u — число документов, содержащих терм u

Регуляризация тематических моделей

- В PLSA решается задача приближённого матричного разложения $F \approx \Phi\Theta$, где F — матрица вероятностей $p(w|t)$
- Даже для одного значения локального экстремума это разложение может иметь бесконечно много решений
- Пусть $S \in \mathbb{R}^{|T|\times|T|}$ — любая невырожденная квадратная матрица, тогда

$$F \approx \Phi\Theta = (\Phi S)(S^{-1}\Theta) = \Phi'\Theta'$$

- Задача PLSA является некорректно поставленной
- Для таких задач можно применять регуляризацию — накладывать на решения дополнительные ограничения
- Это позволяет сузить пространство решений и получить матрицы параметров с полезными свойствами

Модель LDA

- Латентное размещение Дирихле (Latent Dirichlet Allocation) — старый и популярный метод регуляризации тематических моделей
- В качестве регуляризаторов используются априорные распределения Дирихле
- Предполагается, что
  - вектор-столбцы ϕ_t генерируются из распределения Дирихле с параметрами $\beta \in \mathbb{R}^{|T|}$:

$$\mathrm{Dir}(\phi_t|\beta) = \frac{\Gamma(\sum_{w\in W}\beta_w)}{\prod_{w\in W}\Gamma(\beta_w)} \prod_{w\in W} \phi_{wt}^{\beta_w - 1}, \qquad \phi_{wt} > 0, \quad \beta_w > 0$$

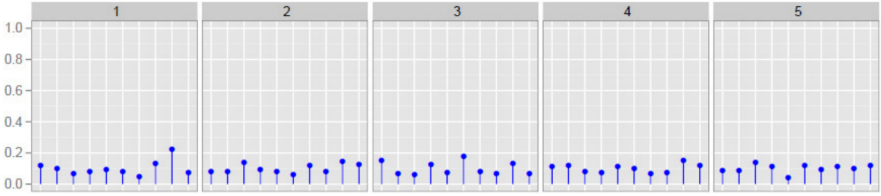
  - вектор-столбцы θ_d генерируются из распределения Дирихле с параметрами $\alpha \in \mathbb{R}^{|W|}$:

$$\mathrm{Dir}(\theta_d|\alpha) = \frac{\Gamma(\sum_{t\in T}\alpha_t)}{\prod_{t\in T}\Gamma(\alpha_t)} \prod_{t\in T} \theta_{td}^{\alpha_t - 1}, \qquad \theta_{td} > 0, \quad \alpha_t > 0$$

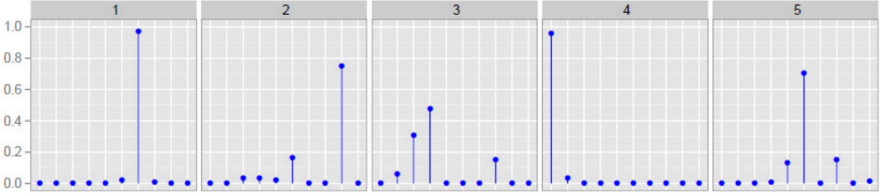
Почему выбрано распределение Дирихле

- Распределение Дирихле позволяет порождать разреженные векторы и имеет параметры для управления степенью разреженности

Все параметры равны 10:



Все параметры равны 0.1:



- Сопряжённость к мультиномиальному распределению упрощает байесовский вывод

Обучение LDA

- Для обучения модели LDA используются различные подходы:
  - Вариационный вывод (Variational Bayes, VB)
  - Сэмплирование Гиббса (Gibbs Sampling, GS)
  - Метод максимума апостериорной вероятности (MAP)
- VB и GS основываются на байесовском выводе
  - Используется сложный математический аппарат (особенно в VB)
  - Параметры модели Φ и Θ не ищутся напрямую
  - Вместо этого строятся распределения на их всевозможные значения
  - Для тематического моделирования такой подход кажется избыточным
  - Итоговым результатом всё равно становятся не сами распределения, а их точечные оценки
- MAP, как и MLE, сразу ищет точечную оценку распределений параметров
- По сути, метод MAP для LDA эквивалентен модели PLSA с регуляризаторами сглаживания/разреживания

MAP для модели LDA

- Распишем апостериорное распределение на параметры по формуле Байеса:

$$p(\Phi,\Theta|D,\alpha,\beta) = \frac{p(D|\Phi,\Theta)\;p(\Phi,\Theta|\alpha,\beta)}{p(D|\alpha,\beta)} \propto p(D|\Phi,\Theta)\;p(\Phi,\Theta|\alpha,\beta) \to \max_{\Phi,\Theta}$$

- Прологарифмируем, подставим априорные распределения, удалим константы:

$$\ln\big(p(D|\Phi,\Theta)\;p(\Phi,\Theta|\alpha,\beta)\big) = \ln\left(\prod_{i=1}^{n} p(d_i,w_i|\Phi,\Theta)\;p(\Phi|\beta)\;p(\Theta|\alpha)\right) =$$

$$= \ln\left(\prod_{d\in D}\prod_{w\in d} p(d,w|\Phi,\Theta)^{n_{dw}} \prod_{t\in T}\mathrm{Dir}\,(\phi_t|\beta)\prod_{d\in D}\mathrm{Dir}\,(\theta_d|\alpha)\right) \propto$$

$$\propto \underbrace{\sum_{d\in D}\sum_{w\in d} n_{dw}\ln\left(\sum_{t\in T}\phi_{wt}\theta_{td}\right)}_{\text{Лог-правдоподобие}} + \underbrace{\sum_{t\in T}\sum_{w\in W}\ln\left(\phi_{wt}^{\beta_w-1}\right) + \sum_{d\in D}\sum_{t\in T}\ln\left(\theta_{td}^{\alpha_t-1}\right)}_{\text{Регуляризаторы}} \to \max_{\Phi,\Theta}$$

MAP для модели LDA

- Оптимизационная задача для LDA MAP:

$$\underbrace{\sum_{d\in D}\sum_{w\in d} n_{dw} \ln\left(\sum_{t\in T}\phi_{wt}\theta_{td}\right)}_{\text{Лог-правдоподобие}} + \underbrace{\sum_{t\in T}\sum_{w\in W}\ln\left(\phi_{wt}^{\beta_w-1}\right) + \sum_{d\in D}\sum_{t\in T}\ln\left(\theta_{td}^{\alpha_t-1}\right)}_{\text{Регуляризаторы}} \to \max_{\Phi,\Theta}$$

$$\sum_{w\in W}\phi_{wt} = 1,\ \phi_{wt} \geqslant 0; \qquad \sum_{d\in D}\theta_{td} = 1,\ \theta_{td} \geqslant 0$$

- Обучение LDA MAP производится таким же EM-алгоритмом, как и PLSA, но с добавлением поправок на M-шаге:

$$\phi_{wt} = \underset{w\in W}{\text{norm}}\left(n_{wt} + \beta_w - 1\right), \qquad \theta_{td} = \underset{t\in T}{\text{norm}}\left(n_{td} + \alpha_t - 1\right)$$

- В зависимости от значений гиперпараметров происходит
  - сглаживание $(\alpha_t, \beta_w > 1)$
  - разреживание $(1 > \alpha_t, \beta_w > 0)$

GS для модели LDA

- Пусть Z — вектор тем t_i (скрытые переменные), присвоенных всем словопозициям (d_i, w_i) в D (одно выбранное значение из каждого $p_{td_iw_i}$)
- Алгоритм оценивает распределение $p(Z|D, \alpha, \beta)$, после этого они используются для вычисления Φ и Θ
- GS для LDA даёт EM-подобный алгоритм с сэмплированием на E-шаге:

$$Z \sim p(Z|D, \beta, \alpha) = \int_\Phi \int_\Theta \underbrace{p(\Phi, \Theta, Z|D, \beta, \alpha)}_{\substack{\text{Совместное распределение} \\ \text{на параметры}}} d\Phi\, d\Theta = \frac{p(D, Z|\alpha, \beta)}{\sum_Z p(D, Z|\alpha, \beta)}$$

- Числитель можно посчитать аналитически, знаменатель нельзя из-за суммирования по всевозможным присваиваниям тем
- Можно сгенерировать выборку из распределения, известного с точностью до нормировки, с помощью сэмплирования Гиббса и оценить его эмпирически

GS для модели LDA

- На каждом шаге для одной словопозиции генерируется одна тема из распределения:

$$t_i \sim p_{td_i w_i} = \underset{t \in T}{\text{norm}} \left(\frac{n_{w_i t} + \beta_{w_i} - 1}{\sum_w (n_{wt} + \beta_w) - 1} \cdot \frac{n_{td_i} + \alpha_t - 1}{\sum_t (n_{td_i} + \alpha_t) - 1} \right)$$

- Старое значение темы для словопозиции должно игнорироваться при сэмплировании нового (вычитаться из счётчиков)
- В процессе происходит уточнение счётчиков n_{wt} и n_{td}, из которых получаются итоговые параметры Φ и Θ:

$$\phi_{wt} = \underset{w \in W}{\text{norm}} \big(n_{wt} + \beta_w \big); \qquad n_{wt} = \sum_{i=1}^{n} [w_i = w][t_i = t]$$

$$\theta_{td} = \underset{t \in T}{\text{norm}} \big(n_{td} + \alpha_t \big); \qquad n_{td} = \sum_{i=1}^{n} [d_i = d][t_i = t]$$

- Сэмплирование требует большого числа маленьких итераций с постоянным обновлением счётчиков

VB для модели LDA

- LDA VB основан на поиске совместного апостериорного распределения параметров модели и скрытых переменных $p(\Phi, \Theta, Z|D, \alpha, \beta)$
- Оно приближается произведением независимых апостериорных распределений по переменным z_{di}, ϕ_t, θ_d
- С учётом взятия матожидания от распределений на параметры и отсутствия оптимизации гиперпараметров получаем EM-подобный алгоритм

$$p_{tdw} = \underset{t \in T}{\text{norm}} \left(\frac{E(n_{wt} + \beta_w)}{E(\sum_w (n_{wt} + \beta_w))} \cdot \frac{E(n_{td} + \alpha_t)}{E(\sum_t (n_{td} + \alpha_t))} \right)$$

$$\phi_{wt} = \underset{w \in W}{\text{norm}}(n_{wt} + \beta_w); \qquad n_{wt} = \sum_{d \in D} n_{dw} p_{tdw}$$

$$\theta_{td} = \underset{t \in T}{\text{norm}}(n_{td} + \alpha_t); \qquad n_{td} = \sum_{w \in d} n_{dw} p_{tdw}$$

где $E(x) = \exp(\psi(x)) \approx x - \frac{1}{2}$ — экспонента от дигамма-функции $\psi(x) = \frac{\Gamma'(x)}{\Gamma(x)}$

Эффективность обучения тематических моделей

- Все описанные методы обучения по своей сути являются EM-подобными
- Во всех требуется
  - подсчитывать распределения p_{tdw} или сэмплировать из них
  - подсчитывать и обновлять счётчики n_{wt} и n_{td}
  - нормировать их с поправками для получения параметров Φ и Θ
- Алгоритмы в чистом «математическом» виде на практике плохи:
  - плохая масштабируемость по данным
  - плохая масштабируемость по параметрам
  - отсутствие учёта аппаратных проблем
- В силу схожести методов различные подходы к повышению эффективности одного из них оказываются полезными для прочих
- Рассмотрим преимущества и недостатки разных предложенных подходов

Распределённые хранение и обработка коллекции

- Объём данных может быть настолько большим, что обработка их на одном компьютере может занять слишком много времени
- Кроме того, документные параметры могут не влезть в оперативную память одной машины
- Хранить и обрабатывать данные можно распределённо на кластере (MPI, Hadoop, Spark)
- Это возможно, поскольку E-шаг (и его аналоги) могут выполняться параллельно для разных документов/слов
- Даже в случае сэмплирования Гиббса это можно делать с минимальным ухудшением сходимости, поскольку данных и уникальных слов много

Распределённые хранение и обработка коллекции

- Каждый вычислительный узел получает/хранит:
  - документы
  - соответствующие счётчики n_{td}
  - соответствующие им переменные Z или p_{tdw}
- Счётчики n_{wt} и параметры Ф являются разделяемыми ресурсами, рассмотрим различные подходы работы с ними
- Увеличение числа машин и ядер может обеспечивать рост производительности до определённого момента
- Вообще, параллельный E-шаг возможен и на одном компьютере
- Для этого нужен достаточный объём оперативной памяти
- Онлайновые алгоритмы позволяют обучать любые объёмы данных с константным пиковым потреблением памяти

Синхронная параллельная обработка

- Счётчики n_{wt} и параметры Φ являются разделяемыми ресурсами
- Текущая версия n_{wt} копируется в память каждого процесса перед E-шагом и доступна ему на чтение
- Обновления счётчиков, получаемые на E-шаге, локальны, и должны быть как-то агрегированы между параллельными процессами/потоками
- В синхронном подходе E-шаг и сбор счётчиков работают поочерёдно:
  1. Все параллельные обработчики завершают E-шаг на текущей итерации
  2. Обновления счётчиков n_{wt} агрегируются в одну матрицу
  3. При необходимости выполняется M-шаг и обновление Φ
  4. Версия n_{wt} в памяти процесса обновляется перед новой итерацией
- Такой подход прост в реализации, но имеет недостатки:
  - длительный простой обработчиков
  - скорость выполнения итерации параллельного E-шага определяется самым медленным процессом/потоком

Асинхронная параллельная обработка

- Асинхронные алгоритмы не имеют выделенного шага синхронизации
- Счётчики n_{wt} и матрица Φ обновляются одновременно с обработкой документов на E-шаге
- Такие архитектуры обычно сложнее в реализации и настройке, но более производительные
- Есть разные способы организации асинхронности:
  - обмен обновлениями n_{wt} между случайной парой обработчиков
  - запись обновлений в фоне в глобальное хранилище счётчиков
  - обучение с запаздыванием обновления параметров
- Все эти подходы имеют реализацию и будут рассмотрены далее

Внешнее хранение параметров документов

- Хранение счётчиков n_{td}, переменных Z или параметров Θ приводит к линейному росту потребления оперативной памяти с ростом числа документов
- Это существенно ухудшает масштабируемость алгоритма
- Есть два основных метода борьбы с проблемой:
  - хранить все связанные с документами счётчики и параметры на диске, подгружая их в память по мере необходимости
  - производить вычисление всех величин, связанных с документом, на лету, удаляя их после получения обновлений n_{wt} для этого документа
- Второй подход требует модификации EM-алгоритма — нужно перенести обновление n_{td} и θ_d на E-шаг
- Он используется в онлайновых алгоритмах и будет рассмотрен далее

Онлайновая (потоковая) обработка

- Оффлайновые алгоритмы делают на каждой итерации полный проход коллекции, накапливая счётчики n_{wt}
- Матрица n_{wt} (или Φ) обновляется в памяти обработчиков раз за итерацию по коллекции
- Такой подход удобен, если коллекция достаточно мала, иначе алгоритм будет долго сходиться
- Онлайновые алгоритмы выполняют M-шаг после обработки каждого документа (или пакета документов)
- Это ускоряет сходимость алгоритма и на большой коллекции позволяет обойтись одним проходом по коллекции
- Поэтому онлайновая обработка позволяет строить модели на потоках данных, если алгоритм не хранит в памяти данные всех документов

Распределённое или оптимизированное хранение модели

- Если число тем в модели и объём словаря велики, то n_{wt} и Ф могут не помещаться в оперативную память одной машины
- Проблему можно решать двумя способами:
  - хранить большую часть модели на диске и подгружать нужные части в ОЗУ по мере необходимости
  - распределить модель по ОЗУ машин кластера
- В первом случае можно продолжить работать на одном компьютере, но скорость вычислений может упасть
- Для борьбы с этим параметры, связанные с наиболее частыми словами держат в памяти постоянно
- Во втором способе модель некоторым способом разбивается на части, которые могут перемещаться между узлами при необходимости
- Различные методы разбиения будут рассмотрены далее

Разреженные хранение и инициализация модели

- Тематические модели часто имеют очень высокую разреженность, использование этого позволяет сильно сэкономить память
- Можно использовать разные форматы хранения, например, CSR или хэш-таблицы (допустимы и гибридные форматы)
- Обратной стороной разреженного хранения может стать снижение производительности из-за замены прямого доступа к элементам матриц на последовательный
- Для борьбы с этим параметры, связанные с наиболее частыми словами хранят в памяти в плотном виде, а прочие — в разреженном
- Для того, чтобы память экономилась всегда, а не с некоторого момента, используют разреженную инициализацию
- Например, можно случайно занулить часть элементов Φ или сузить множество допустимых тем при сэмплировании Z

Динамическое изменение размера модели

- Сэкономить память на старте обучения можно за счёт сильного уменьшения числа тем
- После того, как модель частично сошлась и стала разреженной, можно расширить множество тем
- Постепенно число тем в модели можно увеличить до нужного, не увеличивая расхода памяти за счёт разреженности
- При обработке потоковой коллекции темы имеет смысл добавлять при обработке порции новых документов
- Если коллекция, например, новостная, то такой подход выглядит логичным и с прикладной точки зрения

Разреженная обработка термов

- Разные слова могут иметь существенно различную частоту в коллекции
- Слова, которые часто встречаются в документах, обрабатываются чаще, связанные с ними параметры и счётчики сходятся быстрее
- С какого-то момента обработка частых слов привносит мало информации и тратит на себя много ресурсов
- Можно хранить для каждого слова предшествующие результаты сэмплирования или E-шага и оценивать, насколько сильно они изменились
- Слова для которых изменения систематически оказываются незначительными, исключаются из дальнейшей обработки
- Исключение можно делать безусловно или с заданной вероятностью

Обеспечение отказоустойчивости

- Отказоустойчивость критически важна для любого длительного процесса
- Сложность её реализации сильно зависит от платформы, лежащей в основе реализации
- Hadoop и Spark обеспечивают её самостоятельно, а MPI возлагает эту задачу на разработчика
- Для EM-подобных алгоритмов основной метод защиты от сбоев — периодическое сохранение на диск текущих счётчиков n_{wt} и, при необходимости, состояния модели Φ и переменных Z
- Можно восстановить состояние системы и продолжить обучение с места остановки при сбоях, не затрагивающих диск

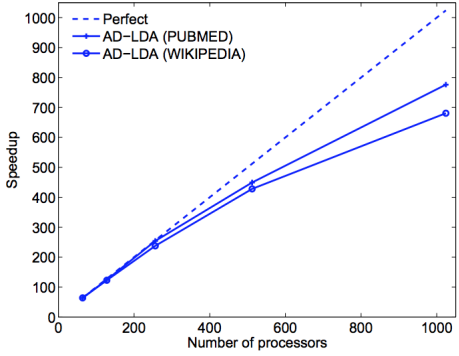
Обзор реализации AD-LDA

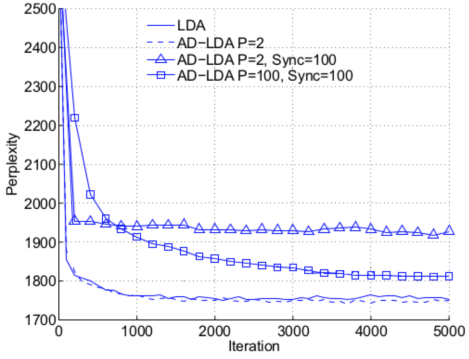
- Реализует LDA GS на MPI кластере
- В основе синхронная многопроцессорная архитектура без общей памяти
- Z получается случайной инициализацией, по ним рассчитываются стартовые значения n_{wt}
- D, Z и n_{td} случайно распределяются по P процессам-сэмплерам
- Счётчики n_{wt} целиком копируются на каждый процессор перед очередным шагом сэмплирования
- После окончания работы последнего сэмплера локальные счётчики n_{wt}^p собираются со всех обработчиков и прибавляются к глобальному состоянию:

$$n_{wt} := n_{wt} + \sum_{p=1,\dots,P} (n_{wt}^p - n_{wt})$$

- Затем обновлённая n_{wt} рассылается на каждый процессор и запускается следующая итерация сэмплирования

Производительность AD-LDA

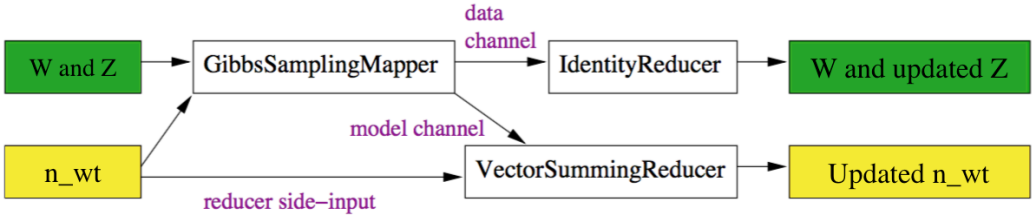




- Частота синхронизации: сходимость vs. скорость работы
- Скорость работы определяется самым медленным процессором
- Нагрузка на вычислительные и сетевые ресурсы неравномерная
- Высокие требования к объёму памяти из-за большого числа копий n_{wt}

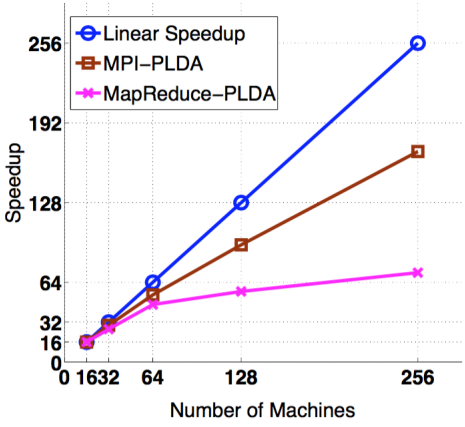
Обзор реализации PLDA

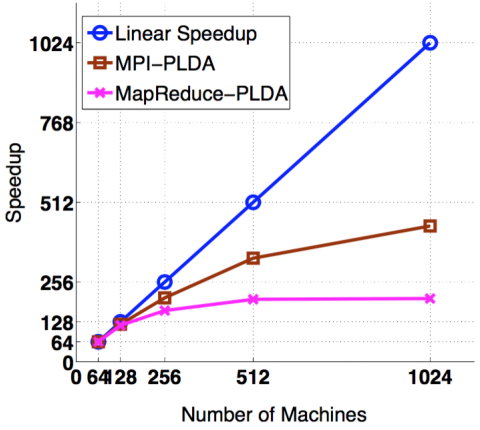
- Реализует AD-LDA на MPI и Hadoop кластерах
- Реализация на MPI обеспечивает отказоустойчивость
- В реализации на Hadoop используется MapReduce:



- Каждый `mapper` получает свою часть коллекции D (на картинке W), скрытых переменных Z и копию глобальных счётчиков n_{wt}
- На выходе два вида `reducer`-ов агрегации и обновления n_{wt} и Z

Производительность PLDA





- Реализация на MapReduce работает и масштабируется ощутимо хуже, чем на MPI — сказывается существенно более частая работа с диском
- PLDA унаследовал все проблемы AD-LDA

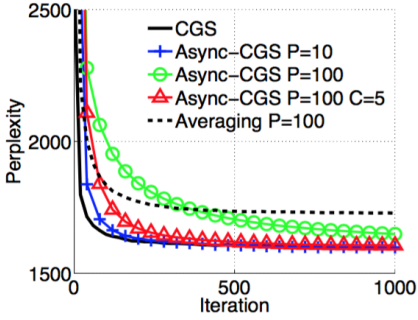
Обзор реализации AS-LDA

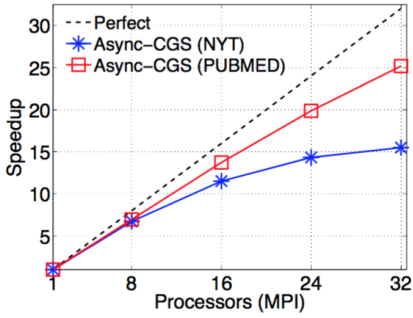
- Реализует LDA GS на MPI кластере
- В основе асинхронная многопроцессорная архитектура без общей памяти
- Каждый $p-$й процесс-сэмплер перед началом работы получает свою часть D, Z и копию n_{wt} (обозначим $n_{wt}^{p-global}$) и собирает обновления n_{wt}^{p}
- После окончания итерации сэмплирования обработчик p коммутирует с другим случайным процессом g, который тоже завершил свою работу
- При этом возможны два варианта:
  - если процессы до этого не коммутировали — производится обмен обновлениями n_{wt}^{g}, которые прибавляются к глобальной копии счётчиков:

$$n_{wt}^{p-global} := n_{wt}^{p-global} + n_{wt}^{g}$$

  - если коммутировали — вычитаются старые счётчики n_{wt}^{g} и прибавляются новые
- Вместо хранения старых счётчиков всех коммутировавших процессов используется алгоритм для подсчёта их аппроксимации $\hat{n}_{wt}^{g}$, которая вычитается из $n_{wt}^{p-global}$

Производительность AS-LDA





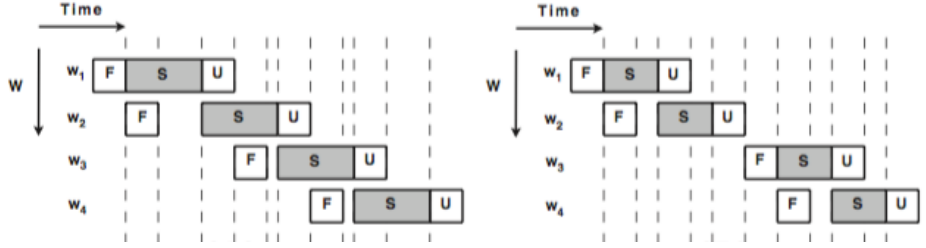
- При небольшом числе процессов сходимость такая же, как у LDA GS (Collapsed GS)
- Если число процессов растёт, то сходимость ухудшается, хотя остаётся лучше простого параллельного сэмплирования с синхронным усреднением в конце итерации
- Эффект можно нивелировать хранением процессом обновления $\hat{n}^g_{wt}$ от 5 самых частых процессов, с которыми он коммутировал (x5 потребления памяти)
- Эти обновления процесс передаёт при коммутации вместе со своими
- В целом реализация получилась сложная и не очень эффективная

Обзор реализации PLDA+

- Реализует LDA GS на кластере с помощью RPC-вызовов
- В основе асинхронная многопроцессорная архитектура без общей памяти
- Процессоры делятся на два подмножества:
  - рабочие — производят сэмплирование
  - транспортные — доставляют и обновляют счётчики n_{wt}
- Это позволяет сэмплировать и обновлять параметры одновременно
- Документы делятся по рабочим процессорам, вхождения одного слова обрабатываются сразу во всех документах процессора
- Матрица n_{wt} перед стартом распределяется по рабочим процессорам в соответствии с тем, какие слова они будут обрабатывать
- Каждый транспортный процессор получает свою часть счётчиков и далее занимается их обновлением и доставкой рабочим процессорам

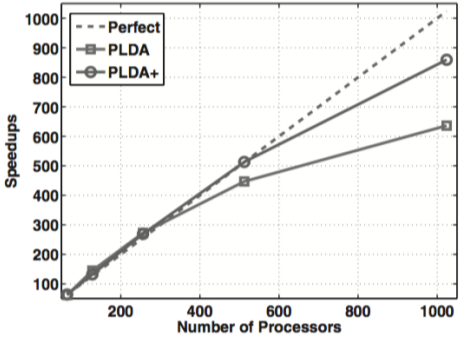
Обзор реализации PLDA+

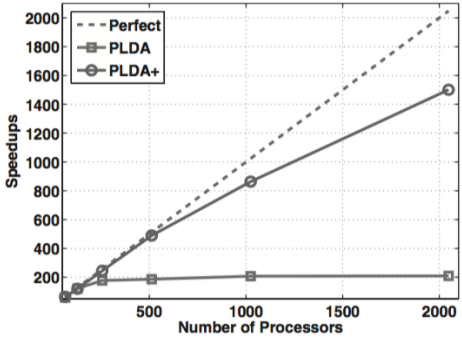
- Для балансирования нагрузки на процессоры редкие слова объединяются в блоки с частыми и обрабатываются одновременно
- Для минимизации вероятности одновременной обработки одного слова w двумя рабочими процессорами создаётся циклическая очередь обрабатываемых слов и расписание для неё
- Каждый процессор стартует с некоторым сдвигом по словарю, это позволяет не загрузить одновременно транспортный процессор, ответственный за w
- Примеры оптимальной и не оптимальной загрузки (F — подгрузка данных и счётчиков, S — сэмплирование, U — обновление счётчиков):



Производительность PLDA+

- Сравнение масштабируемости для коллекций с $|W| = 20K$ и $|W| = 200K$





- Масштабируемость получилась сильно лучше, чем у PLDA за счёт сокращения времени не-фоновых коммуникаций
- Архитектура получилась довольно сложной и хрупкой

Обзор реализации Y!LDA

- Реализует LDA GS на кластере, архитектура «сервер параметров»
- В основе асинхронная многопроцессорная архитектура с общей памятью
- Глобальные счётчики n_{wt} хранятся в распределённом хранилище в `memcached`
- Обработка делится на два логических уровня:
  - в рамках кластера обработка параллелится по узлам
  - в рамках одного узла выполняется параллельное сэмплирование
- На каждому узле хранится копия матрицы n_{wt}; запускается несколько потоков-сэмплеров, работающих непрерывно
- Также запускается выделенный поток, асинхронно сливающий полученные обновления $\tilde{n}_{wt}$ в глобальную n_{wt} по одному слову за раз
- **Проблема:** синхронизация глобальной (в рамках кластера) матрицы n_{wt} между всеми узлами-обработчиками

Обзор реализации Y!LDA

- Схема обновления n_{wt}:
  - n_{wtp} – текущие локальные счётчики p–го узла
  - n_{wtp}^{old} – их копия на момент последней синхронизации с n_{wt}

---

1 Инициализировать $n_{wt} = n_{wtp} = n_{wtp}^{old}$, для всех нодов p;
2 **repeat**
3 | Заблокировать глобально n_{wt} для некоторого слова;
4 | Заблокировать локально n_{wtp} для данного слова;
5 | Обновить глобальное состояние: $n_{wt} := n_{wt} + (n_{wtp} - n_{wtp}^{old})$;
6 | Обновить локальное состояние: $n_{wtp}^{old} = n_{wtp} = n_{wt}$;
7 | Разблокировать n_{wtp};
8 | Разблокировать n_{wt};
9 **until** производится сэмплирование;

- За счёт блокирования одного слова узлы не мешают работе друг друга до определённого числа параллельных узлов

Производительность Y!LDA

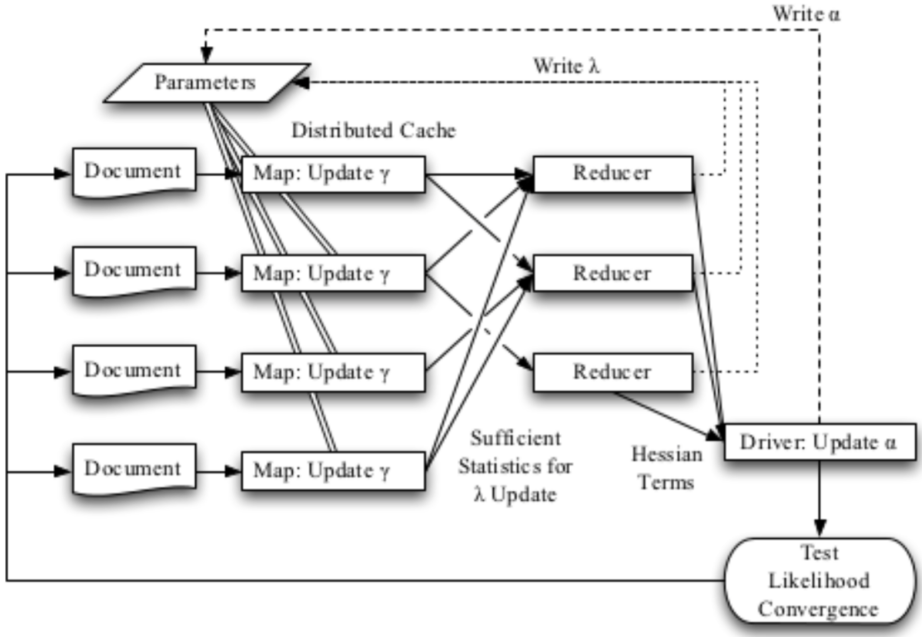
Pubmed	Число узлов	1	2	5	10	20	41	
	Часы работы	3.2	4.2	4.1	4.2	4.4	4.8	
	Обработано за час	62.8K	47.4K	49.3K	47.4K	45.2K	41.7K	
News	Число узлов	1	2	5	10	20	50	100
	Часы работы	4.6	7.5	7.9	8.1	9.0	10.9	12.5
	Обработано за час	43.8K	26.8K	25.2K	24.6K	22.3K	18.3K	16.3K

- Каждый новый узел обрабатывает 200K новых документов
- Резкий скачок от 1 до 2 узлов из-за появления сетевых коммуникаций между узлами
- Алгоритм лучше масштабируется при обработке коротких документов (Pubmed), чем длинных (News), каждый из которых требует большего числа коммуникаций с n_{wt}
- Для обеспечения этих коммуникаций приходится отдавать больше потоков под обновления, уменьшая число сэмплеров
- Суммарный расход памяти по сравнению с предшественниками падает за счёт использования общих локальных счётчиков для всех потоков в рамках одного узла

Обзор реализации Mr.LDA

- Реализует LDA VB на Hadoop кластере с помощью MapReduce
- Схема MapReduce в принятых выше терминах:
  - На каждый документ коллекции создаётся `mapper`, вычисляющий распределения p_{tdw}, счётчики n_{td} и обновления счётчиков n_{wt}
  - На каждую тему создается `reducer`, выполняющий слияние счётчиков n_{wt}
- Глобальные параметры n_{wt} хранятся в общей и доступной всем на чтение памяти в распределённом кэше
- Для ускорения работы `mapper`-ов вместе со счётчиками n_{wt} считаются и сохраняются в кэше нормировочные константы для них
- В каноническом вариационном EM-алгоритме все описанные операции считаются E-шагом, опциональный M-шаг заключается в оптимизации гиперпараметров α и β, в Mr.LDA его выполняет компонент `driver`

Обзор реализации Mr.LDA

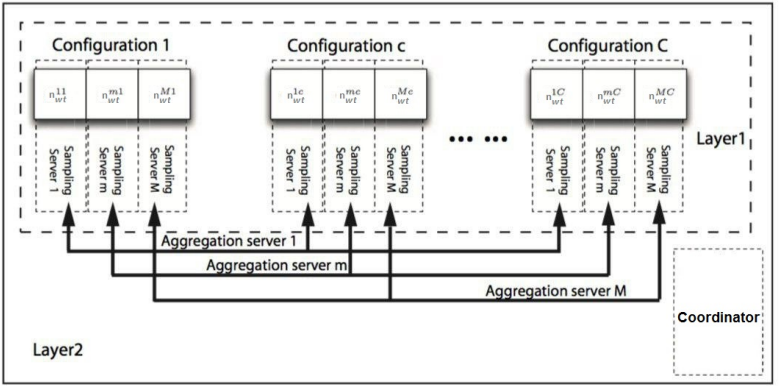


Обзор реализации Peacock

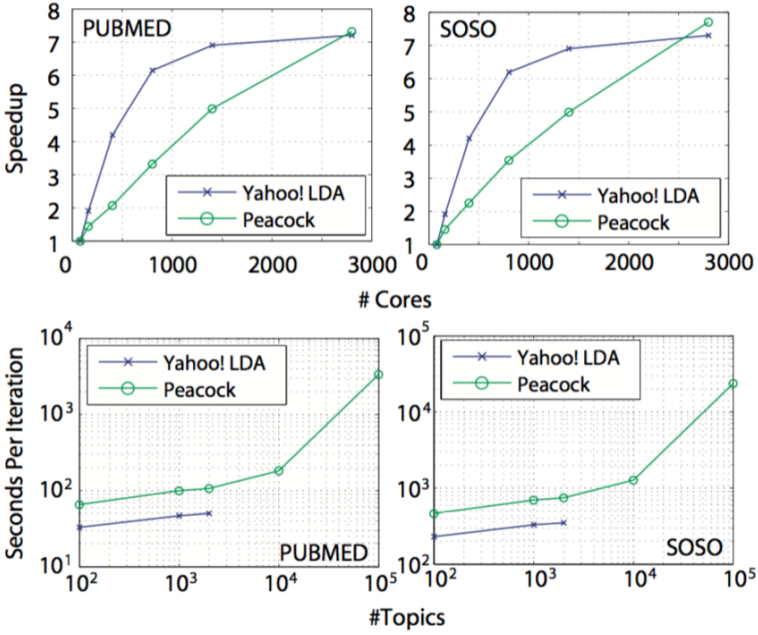
- Реализует LDA GS на кластере с помощью RPC-вызовов
- В основе синхронная многопроцессорная архитектура без общей памяти с возможностью распределённого хранения не только данных, но и модели
- Все ядра делятся на три группы:
  - серверы сэмплирования
  - серверы данных
  - серверы синхронизации
- Счётчики n_{dw} и переменные Z разбиваются на M блоков по документам, каждый связывается с некоторым сервером данных
- Счётчики n_{wt} разбиваются на M блоков по словам, каждый связывается с некоторым сервером сэмплирования
- При обработке сервер данных посылает каждому серверу сэмплирования части своих документов, в которых есть связанные с ним слова
- После обработки блока, сервер сэмплирования обновляет свои счётчики n_{wt} и отправляет обновлённую часть Z на соответствующий сервер данных

Обзор реализации Peacock

- **Проблема:** множество документов и слов нужно делить на одинаковое количество частей, но $|D| \gg |W| \Rightarrow$ перегрузка серверов данных
- Для решения D делится на C частей, для каждой из которых производится своё разделение на блоки
- Агрегирование и синхронизацию счётчиков n_{wt} между серверами сэмплирования из разных наборов блоков обеспечивают серверы синхронизации



Производительность Peacock



Производительность Peacock

- С определённого момента Y!LDA перестаёт масштабироваться из-за частых обращений к хранилищу n_{wt}
- В Peacock больше сетевых операций, но масштабируемость лучше и по ядрам, и по темам
- В целях балансирования нагрузки данные подвергаются случайным перестановка строк и столбцов для достижения максимальной близости числа словопозиций в разных блоках
- Серверы данных сохраняют переменные Z на диск для обеспечения отказоустойчивости
- Качество моделей с точки зрения правдоподобия при обучении с помощью Peacock на одной машине и на кластере почти не отличается

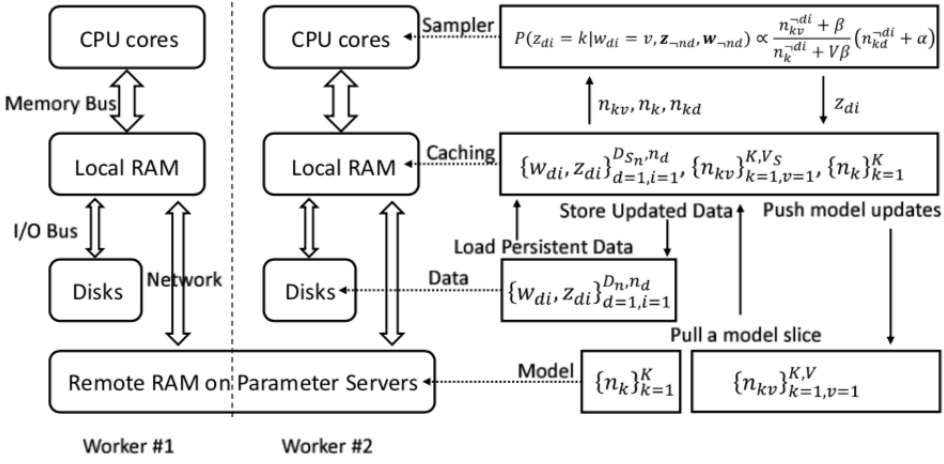
Обзор реализации Light LDA

- Реализует алгоритм на основе LDA GS на кластере с помощью фреймворка Petuum, архитектура «сервер параметров»
- В основе асинхронная многопроцессорная архитектура с общей памятью и возможностью распределённого хранения данных и модели
- В отличие от Peacock, данные являются статичными, а параметры модели перемещаются между узлами-сэмплерами
- С каждым узлом связан свой фрагмент данных, для которого сохраняется список уникальных слов, содержащихся в нём
- На каждом шаге сэмплер выбирает небольшой набор (срез) слов, которые будет обрабатывать
- В его память с диска загружается набор фрагментов документов, содержащих только слова из среза
- Из распределённого хранилища подгружается соответствующие словам из среза счётчики n_{wt}

Обзор реализации Light LDA

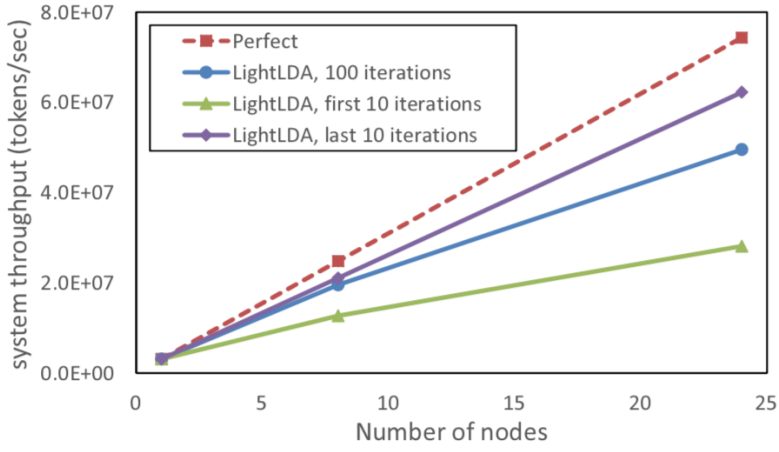
- Простые разбиение и обработка документов работают хуже — небольшой набор длинных документов содержит значительную часть слов из W и требует больших затрат на хранение локальной копии n_{wt}
- При завершении сэмплирования для слов текущего среза узел загружает следующий и продолжает
- Сетевые коммуникации и взаимодействия с диском происходят в фоне
- Обновления n_{wt} вычисляются локально и тоже отправляются в хранилище асинхронно, как в Y!LDA
- Для хранения глобальной n_{wt} используется гибридное решение — для 10% самых частых термов строки матриц хранятся в плотном виде, для остальных — в разреженном в виде хэш-таблицы
- Система отказоусточивая за счёт хранения на диске Z

Обзор реализации Light LDA



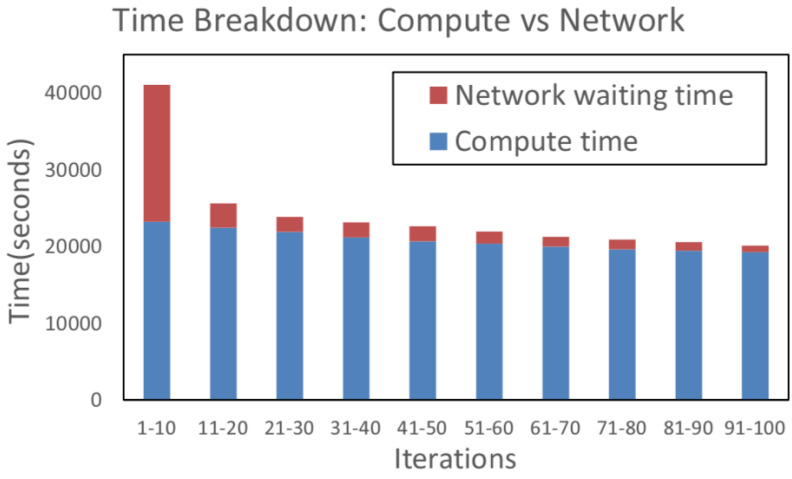
- Каждый узел сэмплирует в многопоточном режиме
- Фрагмент n_{wt} — общий для всех потоков с доступом на чтение

Производительность Light LDA



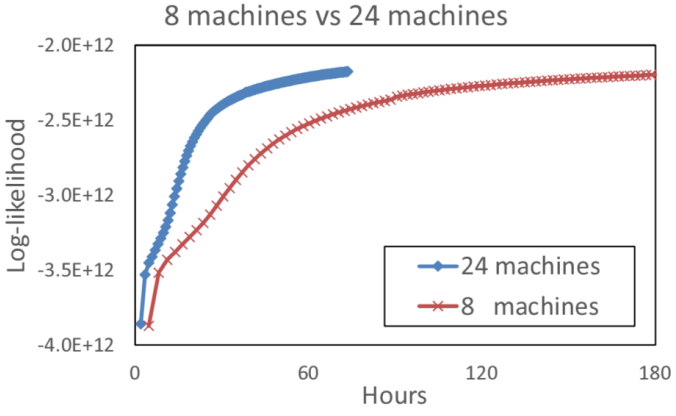
- Масштабируемость сильно хуже на первых итерациях в силу высокой плотности модели n_{wt} — сетевые операции не удаётся производить полностью в фоне

Производительность Light LDA



- После первых итераций разреженность n_{wt} растёт, затраты времени на сетевые коммуникации сильно уменьшаются

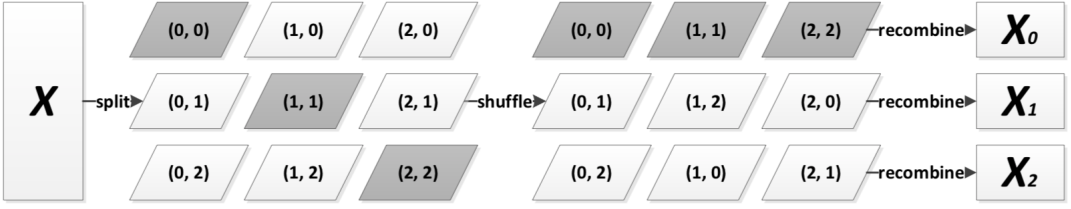
Производительность Light LDA



- При запуске на наборе данных Bing WebC ($|D| = 1.2$MM, $|W| = 1$M, $|T| = 1$M, $n = 200$MM) обучение на 8 машинах заняло 5 дней, на 24 — 2 (при аналогичном значении лог-правдоподобия)

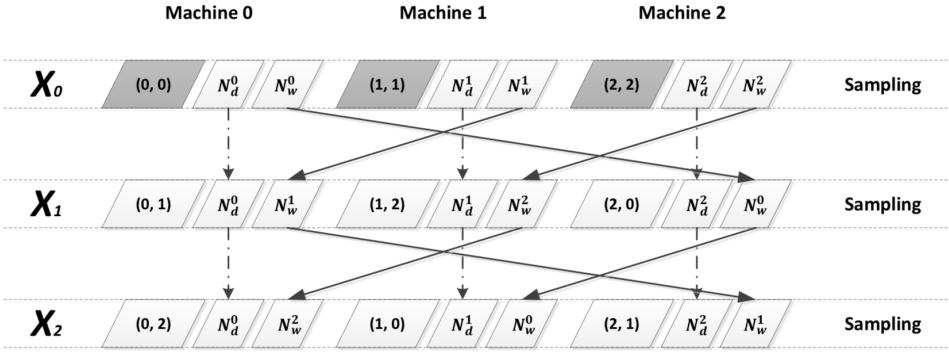
Обзор реализации Spark-LDA

- Реализует алгоритм LDA GS на Spark кластере
- В основе синхронная многопроцессорная архитектура без общей памяти с возможностью распределённого хранения данных и модели
- Данные n_{dw} и счётчики n_{wt} нарезаются по словам на P частей
- Затем на столько же частей данные и счётчики n_{td} нарезаются по документам
- Получается $P \times P$ блоков данных, из которых набирается P наборов по P блоков так, чтобы блоки не пересекались друг с другом по словам и документам:



Обзор реализации Spark-LDA

- Для каждого набора выполняется параллельное сэмплирование и обновление локальных счётчиков n_{wt} и n_{td}, n_{wt} перемещаются между машинами при обработке очередного блока данных



- Сплошная линия — данные передаются, пунктирная — хранятся локально
- Сходимость алгоритма не хуже, чем у непараллельного, но серьёзных экспериментов по сравнению с другими реализациями нет

Обзор реализации ZenLDA

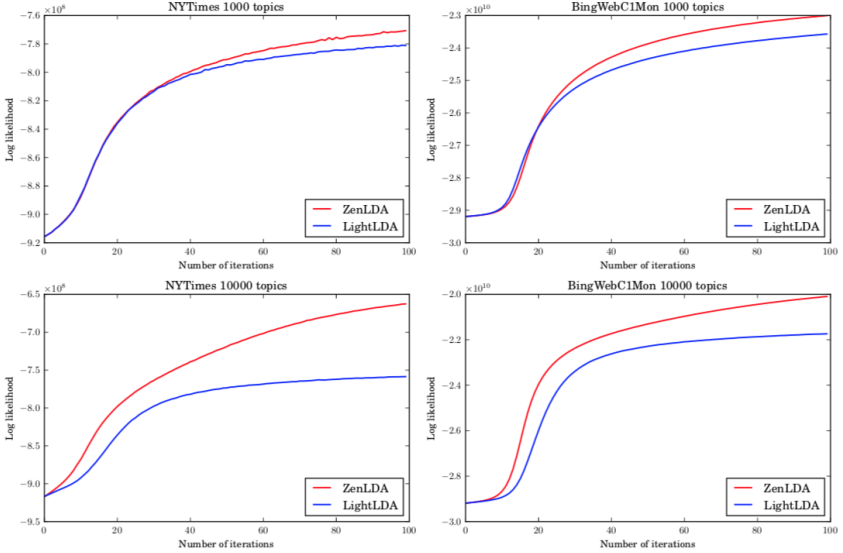
- Реализует алгоритм LDA GS на Spark кластере
- В основе синхронная многопроцессорная архитектура без общей памяти с возможностью распределённого хранения данных и модели
- Данные и модель представляются в виде двудольного ненаправленного графа с вершинам-словами и вершинами-документами
- Ребро обозначает наличие слова в документе
- Счётчики n_{wt} $[n_{td}]$ хранятся в виде разреженных векторов, привязанных к своим вершинам-словам [вершинам-документам]
- Метод хранения $n_w t$ схож с LightLDA: для частых слов представление плотное, для редких — разреженное
- Значения Z привязаны к рёбрам между соответствующими вершинами-термами и вершинами-документами
- Векторы n_t агрегируются после итерации и хранятся отдельно

Обзор реализации ZenLDA

- Параллелизм вычислений обеспечивается разбиением графа на части, обрабатываемые узлами одновременно и независимо
- В реализации используется модификация алгоритма «degree-based hashing», производящего разделение графа по вершинам
- После того, как все обработчики выполнят итерации сэмплирования, обновления счётчиков отправляются на выделенный мастер-узел
- Компонент `driver` подсчитывает и обновляет константы n_t
- Мастер рассылает обновлённые счётчики обратно на узлы-сэмплеры
- Для оптимизации из обработки с некоторой вероятностью исключаются слова, у которых присвоенная тема не изменилась на последней итерации

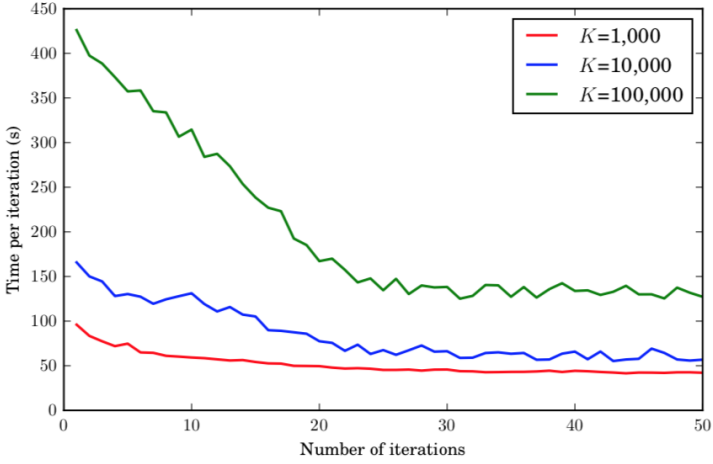
Производительность ZenLDA

- ZenLDA показывает более высокую скорость сходимости, чем LightLDA по итерациям, причём каждая итерация при этом в 2-6 раз быстрее



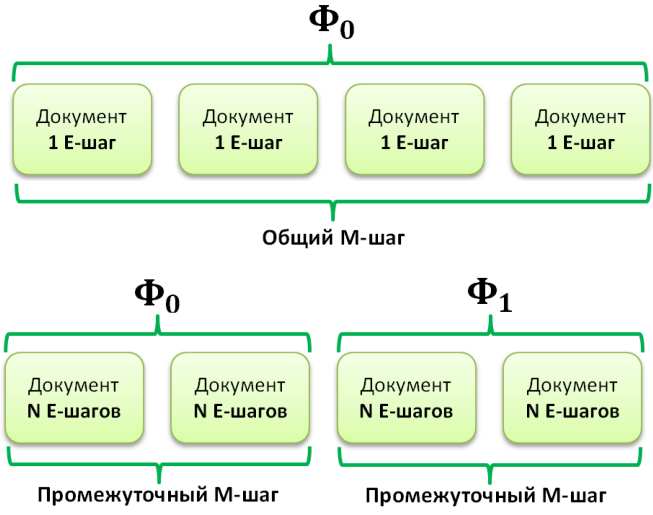
Производительность ZenLDA

- Реализация использует разреженную инициализацию Z путём случайного сужения множества возможных тем для сэмплирования
- Это даёт более разреженную n_{wt} на первых итерациях и слабый рост времени обработки одной итерации при росте числа тем



Напоминание: онлайновый алгоритм

- Онлайновые алгоритмы обновляют модель (выполняют M-шаг) после обработки каждого документа (пакета докумнетов), а не всех документов коллекции
- Это ускоряет сходимость алгоритма и на большой коллекции позволяет обойтись одним проходом по всему множеству документов
- Поэтому онлайновая обработка позволяет строить модели на потоках данных, если алгоритм не хранит в памяти данные всех документов



Обзор реализаций Vowpal Wabbit LDA и Gensim

- Обе библиотеки реализуют онлайновый вариационный EM-алгоритм
- VW.LDA работает в однопоточном режиме, эффективен за счёт реализации на чистом C++
- Gensim — пакет для NLP, берёт начало от реализации алгоритма обучения LDA
- В Gensim реализуется две версии обучения LDA:
  - LdaModel — однопоточная реализация
  - LdaMulticore — многопоточная реализация (архитектурно схожа с Y!LDA)
- Gensim реализует алгоритмы обучения на Python, это сказывается на скорости и масштабируемости
- Но его просто устанавливать и использовать, поэтому он удобен для обработки небольших коллекций
- В Gensim есть интерфейсы для использования VW.LDA из Python

https://github.com/JohnLangford/vowpal_wabbit/wiki
R. Rehurek, P. Sojka, Software Framework for Topic Modelling with Large Corpora, 2010.

Обзор реализации FOEM-LDA

- Реализует синхронный однопоточный онлайновый алгоритм LDA MAP
- Обновление матрицы Φ производится после каждого документа
- Матрица Θ никогда не хранится полностью в явном виде
- Счётчики и параметры модели хранятся на диске и подгружаются в память по мере необходимости
- При обработке пакета документов в память загружаются параметры для наиболее частых слов
- Затем список обновляется и в память загружаются параметры для слов, для которых $r_w = \Delta p_{tdw} \cdot n_{dw}$ в сумме по пакету наиболее значительно
- При загрузке недостающего вектора ϕ_{wt} для слова из памяти удаляется вектор для слова с наименьшим r_w
- На тематизацию коллекции с $\approx 8M$ документов и $T = 10000$ с ограничением памяти в 2Гб уходит 29 часов (размер модели — 10Гб)

Регуляризация тематических моделей

- В PLSA решается задача приближённого матричного разложения $F \approx \Phi\Theta$, где F — матрица вероятностей $p(w|t)$
- Даже для одного значения локального экстремума это разложение может иметь бесконечно много решений
- Пусть $S \in \mathbb{R}^{|T|\times|T|}$ — любая невырожденная квадратная матрица, тогда

$$F \approx \Phi\Theta = (\Phi S)(S^{-1}\Theta) = \Phi'\Theta'$$

- Задача PLSA является некорректно поставленной
- Регуляризация является способом наложить дополнительные ограничения на модель
- LDA накладывает регуляризацию в виде априорных распределений Дирихле, но это далеко не единственный способ

Модель ARTM

- Аддитивная регуляризация тематических моделей (Additive Regularization of Topic Models) обобщает модель PLSA добавлением взвешенной суммы ограничений-регуляризаторов R:

$$\sum_{d\in D}\sum_{w\in d} n_{dw}\ln\sum_{t\in T}\phi_{wt}\theta_{td} + R(\Phi,\Theta) \to \max_{\Phi,\Theta}, \qquad R(\Phi,\Theta) = \sum_i \tau_i R_i(\Phi,\Theta)$$

- R_i — ограничение, выраженное в виде функционала, τ_i — его весовой гиперпараметр
- Обучать модели ARTM можно с помощью регуляризованного EM-алгоритма, похожего на алгоритм для PLSA
- ARTM может комбинировать множество ограничений в одной модели:
  - разреженность
  - интерпретируемость
  - иерархичность
  - наличие фона
  - выделение заданных тем
  - ...

EM-алгоритм для модели ARTM

$$\sum_{d\in D}\sum_{w\in d} n_{dw}\ln\sum_{t\in T}\phi_{wt}\theta_{td} + R(\Phi,\Theta) \to \max_{\Phi,\Theta}, \qquad R(\Phi,\Theta) = \sum_i \tau_i R_i(\Phi,\Theta)$$

$$\sum_{w\in W}\phi_{wt} = 1,\ \phi_{wt} \geqslant 0; \qquad \sum_{d\in D}\theta_{td} = 1,\ \theta_{td} \geqslant 0$$

Теорема: *если функция $R(\Phi,\Theta)$ непрерывно дифференцируема, а (Φ,Θ) является точкой локального максимума, то выполняется система уравнений (с исключением нулевых ϕ_t и θ_d)*:

$$p_{tdw} = \underset{t\in T}{\text{norm}}\left(\phi_{wt}\theta_{td}\right) \Rightarrow \qquad n_{wt} = \textstyle\sum_{d\in D} n_{dw}p_{tdw} \qquad \text{(E-шаг)}$$

$$n_{td} = \textstyle\sum_{w\in d} n_{dw}p_{tdw}$$

$$\phi_{wt} = \underset{w\in W}{\text{norm}}\left(n_{wt} + \phi_{wt}\frac{\partial R}{\partial \phi_{wt}}\right) \qquad \theta_{td} = \underset{t\in T}{\text{norm}}\left(n_{td} + \theta_{td}\frac{\partial R}{\partial \theta_{td}}\right) \qquad \text{(M-шаг)}$$

Пример: модель с фоновыми темами

- **Идея:** выделить подмножество разреженных интерпретируемых тем, оставив модели свободу для описания общих тем и лексики
- Множество выделяемых тем T делится на две подмножества: предметные темы S и фоновые темы B
- Для предметных тем в матрице Φ включается декоррелирование
- Для фоновых тем в матрице Φ включается равномерное сглаживание

$$R(\Phi) = -\frac{\tau_1'}{2}\sum_{t\in S}\sum_{s\in S\setminus t}\sum_{w\in W}\phi_{wt}\phi_{ws} + \tau_2'\sum_{t\in B}\sum_{w\in W}\ln\phi_{wt}$$

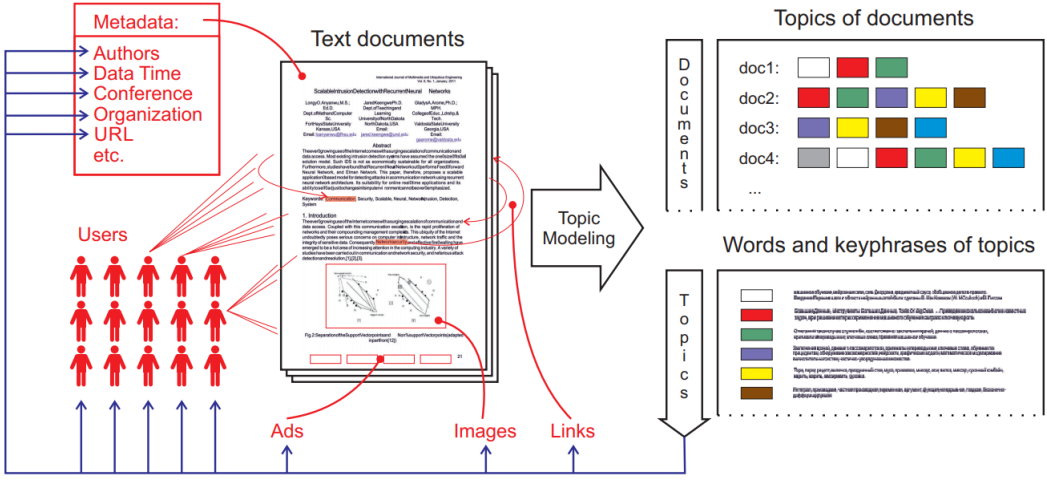
- В результате множество S содержит темы более высокого качества, а значение перплексии у полученной модели не ухудшается

Мультимодальные текстовые данные

- Описанные модели используют представление данных в виде общего для всех элементов документа «мешка слов»
- Это лишает модель полезной информации, поскольку слов в различных частях документ могут иметь существенно разную важность
- Помимо обычных слов в текстах могут встречаться другие типы (модальности) слов, например
  - метки классов и категорий
  - имена авторов
  - ссылки
  - ключевые слова и теги
  - дата публикации
- Использование для разных модальностей общего «мешка слов» как снижает качество моделирования, так и сужает круг решаемых задач

Модель M-ARTM

Идея: использовать для каждой модальности свой функционал правдоподобия и обучать их совместно в общем EM-алгоритме



Модель M-ARTM

- Модель M-ARTM обобщает модель ARTM на случай множества модальностей M:

$$\sum_{m \in M} \tau_m \sum_{d \in D} \sum_{w \in W^m} n_{dw} \ln \sum_{t \in T} \phi_{wt} \theta_{td} + R(\Phi, \Theta) \to \max_{\Phi, \Theta}$$

где

  - W^m — словарь модальности $m \in M$
  - τ_m — веса, регулирующие значимость слагаемых в функционале

- Ограничения в такой постановке задачи примут вид

$$\sum_{w \in W^m} \phi_{wt} = 1,\ \phi_{wt} \geqslant 0,\ \forall m \in M; \qquad \sum_{d \in D} \theta_{td} = 1,\ \theta_{td} \geqslant 0$$

- Матрица Φ для M-ARTM — это блочная матрица, полученная конкатенацией всех Φ^m

K. Vorontsov, O. Frei et al. Non-bayesian additive regularization for multimodal topic modeling of large collections, 2015.

EM-алгоритм для модели M-ARTM

$$\sum_{m\in M}\tau_m\sum_{d\in D}\sum_{w\in W^m} n_{dw}\ln\sum_{t\in T}\phi_{wt}\theta_{td} + R(\Phi,\Theta)\to\max_{\Phi,\Theta}$$

$$\sum_{w\in W^m}\phi_{wt}=1,\ \phi_{wt}\geqslant 0,\ \forall m\in M; \qquad \sum_{d\in D}\theta_{td}=1,\ \theta_{td}\geqslant 0$$

Теорема: *если функция $R(\Phi,\Theta)$ непрерывно дифференцируема, а (Φ,Θ) является точкой локального максимума, то выполняется система уравнений (с исключением нулевых ϕ_t и θ_d)*:

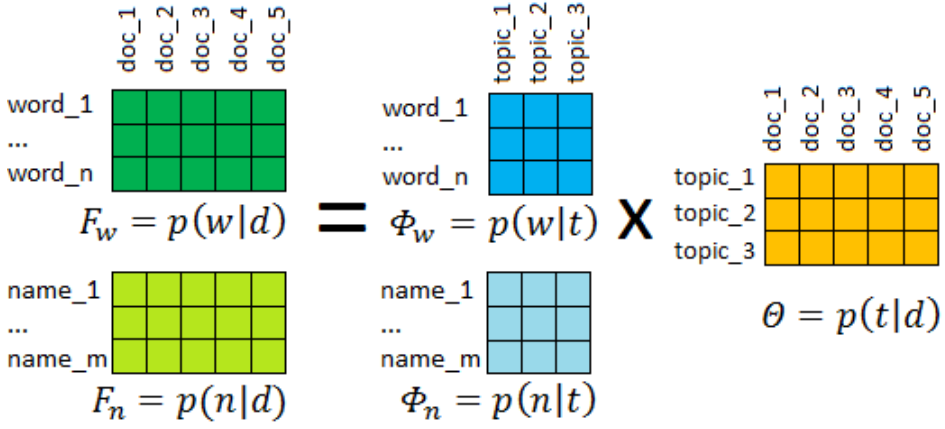
$$p_{tdw}=\underset{t\in T}{\text{norm}}\left(\phi_{wt}\theta_{td}\right)\Rightarrow \qquad \begin{aligned} n_{wt}&=\textstyle\sum_{d\in D}\tau_{m(w)}n_{dw}p_{tdw}\\ n_{td}&=\textstyle\sum_{w\in d}\tau_{m(w)}n_{dw}p_{tdw}\end{aligned} \qquad \text{(E-шаг)}$$

$$\phi_{wt}=\underset{w\in W^m}{\text{norm}}\left(n_{wt}+\phi_{wt}\frac{\partial R}{\partial\phi_{wt}}\right) \quad \theta_{td}=\underset{t\in T}{\text{norm}}\left(n_{td}+\theta_{td}\frac{\partial R}{\partial\theta_{td}}\right) \qquad \text{(M-шаг)}$$

где $\tau_{m(w)}$ — вес модальности слова w

Модель M-ARTM

- По факту на одной коллекции строится $|M|$ тематических моделей
- Матрица Θ является общей для всех моделей



Пример: тематическая модель классификации

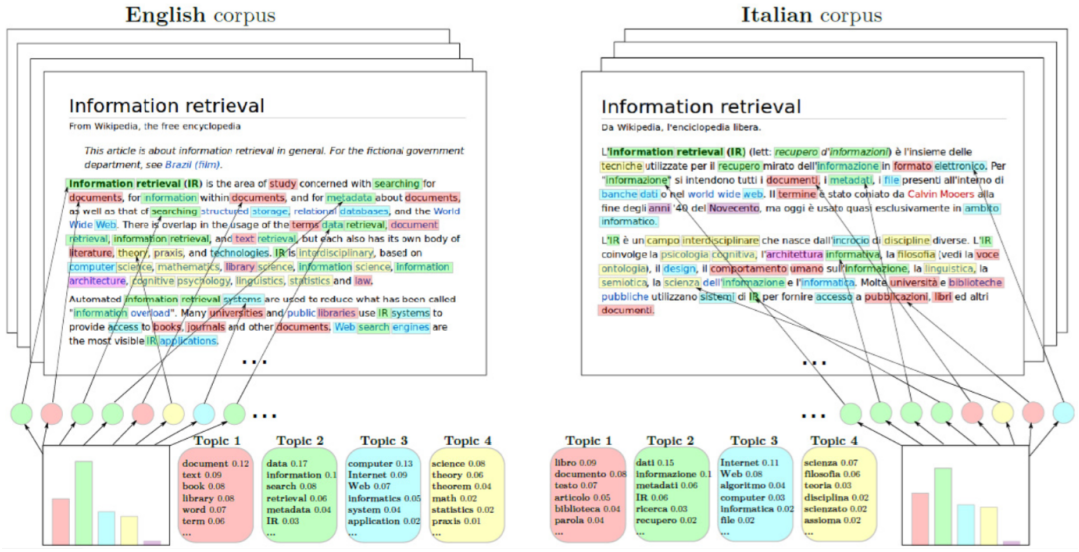
- Модель с двумя модальностями:
  - обычные слова или N-граммы документа
  - слова-метки классов
- При обучении вес модальности слов-меток обычно на 1-2 порядка больше, чем у обычных слов
- После обучения модель запускается на новых документах с фиксированной Φ и выдаèт для них векторы θ_d
- Имея такой вектор для документа d можно посчитать вероятность его отнесения к классу c по формуле

$$p(c|d) = \sum_{t \in T} p(c|t)\, p(t|d)$$

Пример: мультиязычная тематическая модель

- Модель с K модальностями:
  - слова или N-граммы документа d на языке 1
  - ...
  - слова или N-граммы документа d на языке K
- Параллельные тексты должны быть похожи по темам, но не обязаны быть точными переводами друг друга
- Множества наиболее вероятных слов в разных модальностях внутри одной темы часто получаются переводами друг друга

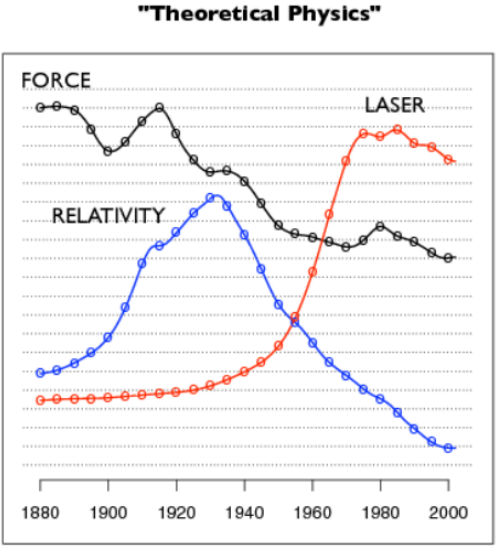
Пример: мультиязычная тематическая модель



I. Vulic et al. Probabilistic topic modeling in multilingual settings: a short overview of its methodology with applications, 2012.

Пример: динамическая тематическая модель

- Модель с двумя модальностями:
  - обычные слова или N-граммы документа
  - слова-метки времени
- При обучении вес модальности слов-меток обычно на 1-2 порядка больше, чем у обычных слов
- Можно следить за изменением популярности темы во времени



Библиотека BigARTM

- Реализует параллельный онлайновый асинхронный EM-алгоритм для обучения модели M-ARTM
- Работает в разы быстрее ближайших конкурентов
- Доступна под Linux и Mac OS

- Имеет гибкий интерфейс на Python для многоэтапного моделирования
- Содержит встроенную библиотеку регуляризаторов и метрик качества
- Может строить иерархические регуляризованные модели
- Библиотека TopicNet, надстраиваемая на BigARTM, снижает порог входа и упрощает проведение экспериментов и настройку моделей

Функция ProcessDocument

Input: документ $d \in D$, матрица $\Phi = (\phi_{wt})$;
Output: матрица $(\tilde{n}_{wt})$, вектор θ_{td};

1 инициализировать $\theta_{td} := \frac{1}{|T|}$ для всех $t \in T$;
2 **repeat**
3 $\quad p_{tdw} := \underset{t \in T}{\text{norm}}\left(\phi_{wt}\theta_{td}\right)$ для всех $w \in d$ и $t \in T$;
4 $\quad \theta_{td} := \underset{t \in T}{\text{norm}}\left(\sum_{w \in d} n_{dw} p_{tdw} + \theta_{td}\frac{\partial R}{\partial \theta_{td}}\right)$ для всех $t \in T$;
5 **until** до сходимости θ_d;
6 $\tilde{n}_{wt} := n_{dw} p_{tdw}$ для всех $w \in d$ и $t \in T$;

- `ProcessDocument` представляет собой E-шаг с пересчетом Θ
- Он запускается для каждого $d \in D$ и возвращает инкременты счётчиков $\tilde{n}_{wt}$, при необходимости итоговый вектор θ_d тоже можно сохранить

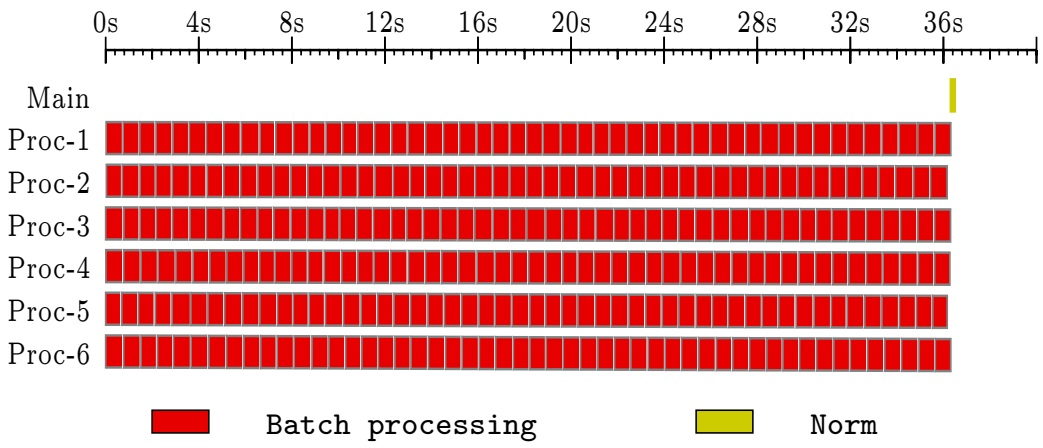
Оффлайновый алгоритм

Input: коллекция D;
Output: матрица $\Phi = (\phi_{wt})$;

1 инициализировать (ϕ_{wt});
2 создать батчи $D := D_1 \sqcup D_2 \sqcup \cdots \sqcup D_B$;
3 **repeat**
4 $\quad (n_{wt}) := \sum\limits_{b=1,\ldots,B} \sum\limits_{d \in D_b} \text{ProcessDocument}(d, \Phi)$;
5 $\quad (\phi_{wt}) := \underset{w \in W}{\text{norm}}(n_{wt} + \phi_{wt} \frac{\partial R}{\partial \phi_{wt}})$;
6 **until** до сходимости (ϕ_{wt});

- Синхронный многопоточный пакетный EM-алгоритм для M-ARTM
- Несколько рабочих потоков занимаются обработкой документов и сбором счётчиков n_{wt}, во время синхронизации один выделенный поток выполняет M-шаг
- Один пакет (батч) обрабатывается рабочим потоком за раз

Оффлайновый алгоритм: диаграмма Гантта



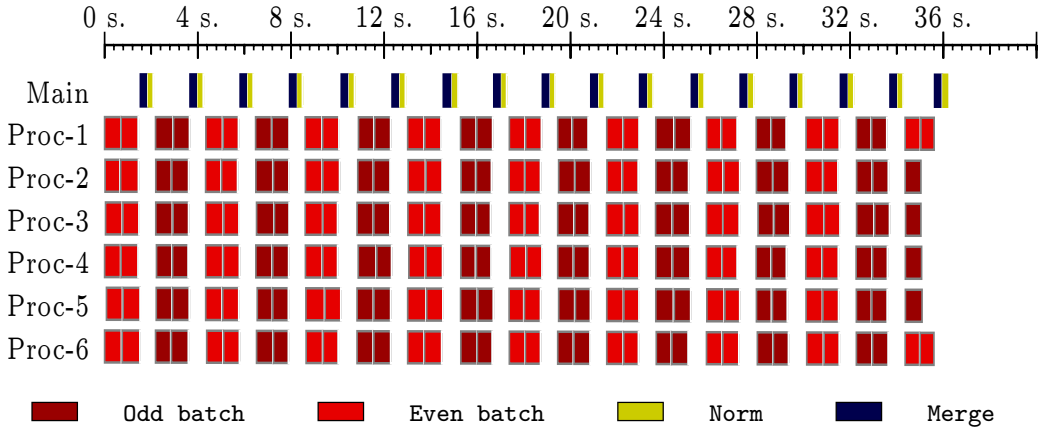
Синхронный онлайновый алгоритм

Input: коллекция D, параметры η, τ_0, κ;
Output: матрица $\Phi = (\phi_{wt})$;

создать батчи $D := D_1 \sqcup D_2 \sqcup \cdots \sqcup D_B$;
инициализация (ϕ^0_{wt});
for all обновить $i = 1, \ldots, \lfloor B/\eta \rfloor$
    $(\tilde{n}^i_{wt}) := \text{ProcessBatches}(\{D_{\eta(i-1)+1}, \ldots, D_{\eta i}\}, \Phi^{i-1})$;
    $\rho_i := (\tau_0 + i)^{-\kappa}$;
    $(n^i_{wt}) := (1 - \rho_i) \cdot (n^{i-1}_{wt}) + \rho_i \cdot (\tilde{n}^i_{wt})$;
    $(\phi^i_{wt}) := \underset{w \in W}{\text{norm}}(n^i_{wt} + \phi^{i-1}_{wt} \frac{\partial R}{\partial \phi_{wt}})$;

- В онлайновом алгоритме синхронизации производятся через каждые η батчей
- Вместо зануления n_{wt} между M-шагами используется скользящее среднее
- Рабочие потоки чаще простаивают из большого числа синхронизаций

Синхронный онлайновый алгоритм: диаграмма Гантта

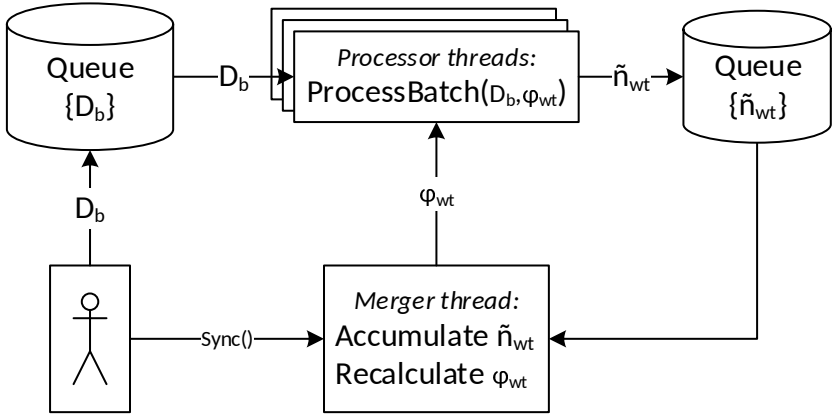


Асинхронный алгоритм Async

- Выделенный поток `DataLoader` загружает загружает батчи с диска в очередь обработки `Processor queue`
- Каждый рабочий поток по мере необходимости извлекает из очереди обработки один батч и выполняет E-шаг
- Завершив обработку, он складывает матрицу инкрементов $\tilde{n}_{wt}$ в очередь слияния `Merger queue` (если в ней есть место) и начинает обработку следующего батча
- Когда число матриц в очереди `Merger queue` становится равным параметру η, выделенный поток слияния `Merger` извлекает инкременты, складывает их в одну n_{wt} и обновляет Φ
- Все потоки работают параллельно, что обеспечивает асинхронность
- Введём элементарную операцию для упрощения нотации:

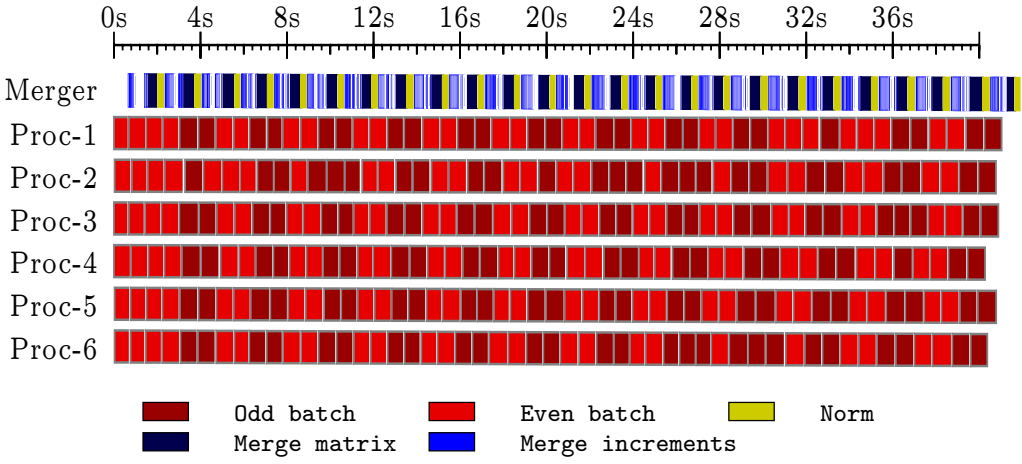
$$\texttt{ProcessBatch}(D_b, \Phi) = \sum_{d \in D_b} \texttt{ProcessDocument}(d, \Phi)$$

Асинхронный алгоритм Async

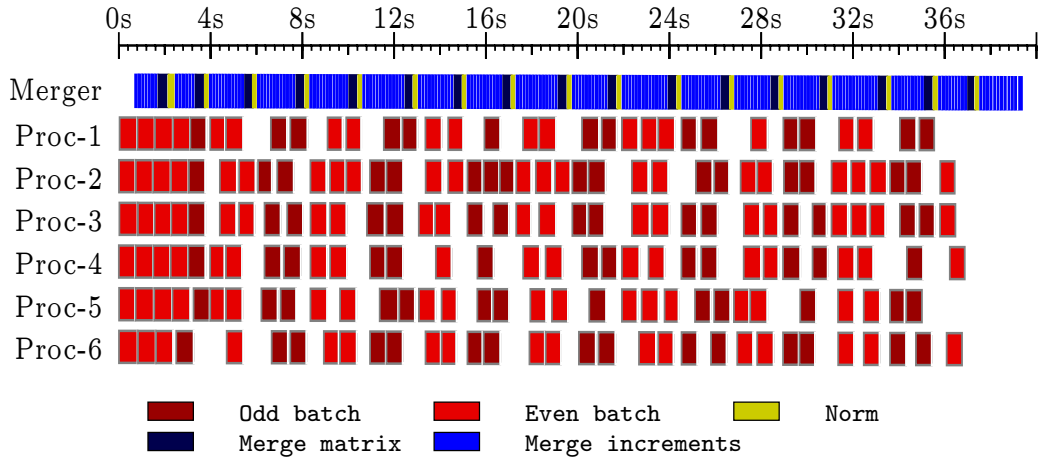


- После очередного M-шага появляется новая версия матрицы Φ
- Рабочие потоки переключаются на неё при старте обработки новых батчей
- Старая версия Φ удаляется после того, как ей перестаёт пользоваться последний рабочий поток

Алгоритм Async: диаграмма Гантта



Алгоритм Async: диаграмма Гантта (плохой случай)



- Очередь слияния — узкое место, много матриц инкрементов потребляют память и тратят процессорное время на своё суммирование
- **Пример**: маленькие батчи, мало повторений E-шага, долгая регуляризация

Асинхронный алгоритм DetAsync

- Основные проблемы Async:
  - Хранение инкрементов для каждого батча в виде отдельной матрицы требует много памяти
  - Слияние матриц инкрементов для набора батчей требует много времени
  - Порядок обработки пакетов может не совпадать с порядком помещения обновлений $\tilde{n}_{wt}$ в очередь слияния, из-за чего алгоритм является недетрминированным
- Для решения этих проблем предлагается:
  - Ликвидировать очередь и поток слияния, собирать все обновления в одну общую матрицу n_{wt}, доступную на запись построчно (как в Y!LDA)
  - Обновлять модель с запаздыванием на один шаг — это позволяет нормировать модель по $\tilde{n}_{wt}$ для i-го набора батчей одновременно с обработкой $(i+1)$-го

Асинхронный алгоритм DetAsync

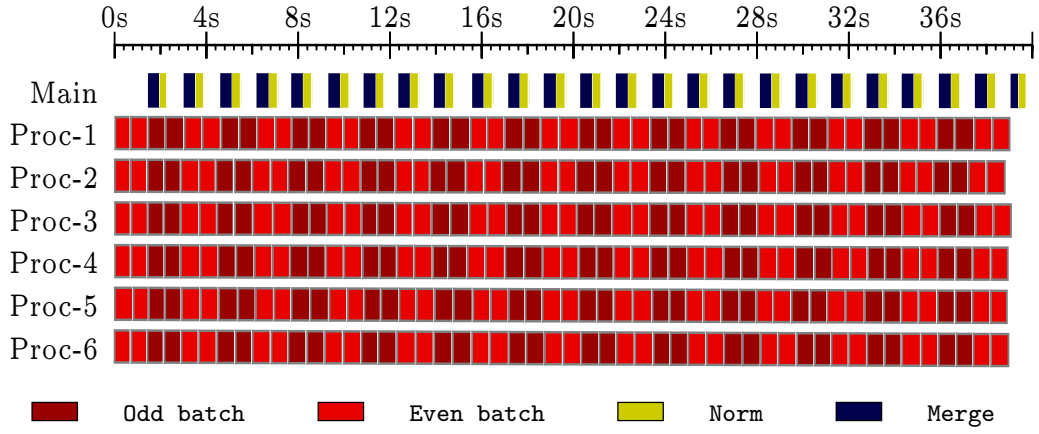
- Вводятся две новые операции:
  - `AsyncProcessBatches`$(\{D_b\}, \Phi) = \sum_{D_b}$ `ProcessBatch`(D_b, Φ) — сразу после запуска операция возвращает управление в вызвавший поток, результат ожидается в `future`-объекте
  - `Await` — блокирует вызвавший поток и дожидается результатов заданного `future`-объекта
- Устраняется поток `DataLoader`, рабочие потоки сами загружают батчи с диска на обработку по одному за раз
- Доступ к n_{wt} на запись обеспечивается с помощью спин-локов, связанных с каждой строкой матрицы
- Поток, вынужденный ждать очереди, не засыпает
- Тройка операций «lock-update-release» производится в цикле по всем словам документа $w \in d$ в конце операции `ProcessDocument`

Асинхронный алгоритм DetAsync

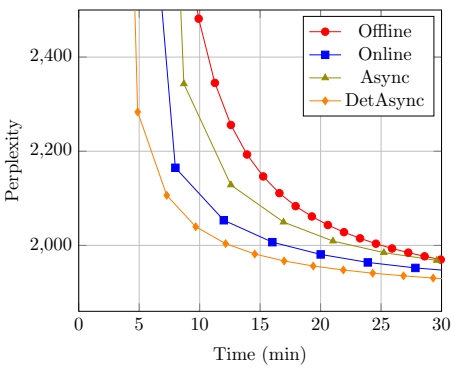
Input: коллекция D, параметры η, τ_0, κ;
Output: матрица $\Phi = (\phi_{wt})$;

создать батчи $D := D_1 \sqcup D_2 \sqcup \cdots \sqcup D_B$;
инициализировать (ϕ^0_{wt});
$F^1 := \text{AsyncProcessBatches}(\{D_1, \ldots, D_\eta\}, \Phi^0)$;
for all обновления $i = 1, \ldots, \lfloor B/\eta \rfloor$
    if $i \neq \lfloor B/\eta \rfloor$ **then**
        $F^{i+1} := \text{AsyncProcessBatches}(\{D_{\eta i+1}, \ldots, D_{\eta i+\eta}\}, \Phi^{i-1})$;
    $(\hat{n}^i_{wt}) := \text{Await}(F^i)$;
    $\rho_i := (\tau_0 + i)^{-\kappa}$;
    $(n^i_{wt}) := (1 - \rho_i) \cdot (n^{i-1}_{wt}) + \rho_i \cdot (\hat{n}^i_{wt})$;
    $(\phi^i_{wt}) := \underset{w \in W}{\text{norm}}(n^i_{wt} + \phi^{i-1}_{wt} \frac{\partial R}{\partial \phi_{wt}})$;

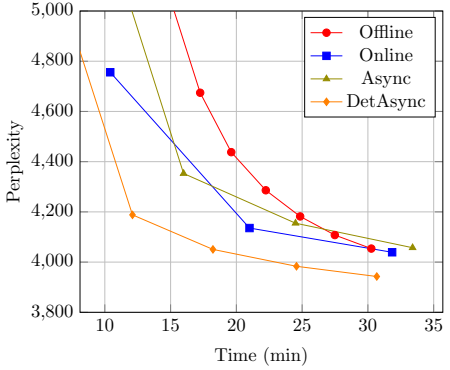
Асинхронный алгоритм DetAsync: диаграмма Гантта



Сравнение алгоритмов в BigARTM



Pubmed



Wikipedia

- Сравнение перплексии, полученной за выделенный промежуток времени
- Коллекции Pubmed ($|D| = 8.2M$, $|W| = 141K$), Wikipedia ($|D| = 3.7M$, $|W| = 100K$)
- Пиковое потребление памяти у DetAsync в 1.5-2 раза ниже, чем у Async

Сравнение BigARTM vs. Gensim vs. Vowpal Wabbit LDA

	Gensim	VW-LDA	BigARTM (Online ARTM)	BigARTM (DetAsync)
P = 1, T= 50	142m (4945)	50m (5413)	42m (5117)	25m (5131)
P = 1, T= 100	287m (3969)	91m (4592)	52m (4093)	32m (4133)
P = 1, T= 200	637m (3241)	154m (3960)	83m (3347)	53m (3362)
P = 2, T= 50	89m (5056)		22m (5092)	13m (5160)
P = 2, T= 100	143m (4012)		29m (4107)	19m (4144)
P = 2, T= 200	325m (3297)		47m (3347)	28m (3380)
P = 4, T= 50	88m (5311)		12m (5216)	7m (5353)
P = 4, T= 100	104m (4338)		16m (4233)	10m (4357)
P = 4, T= 200	315m (3583)		26m (3520)	16m (3634)
P = 8, T= 50	88m (6344)		8m (5648)	5m (6220)
P = 8, T= 100	107m (5380)		10m (4660)	6m (5119)
P = 8, T= 200	288m (4263)		15m (3929)	10m (4309)

Таблица 5: Сравнение BigARTM с Vowpal Wabbit.LDA и Gensim (P – число потоков)

Фреймворк / T	2000	5000
BigARTM (Online ARTM)	166m (2377)	399m (1942)
BigARTM (DetAsync)	119m (2645)	281m (2216)

Таблица 6: Запуск BigARTM с большим числом тем.

Хранение и обработка параметров в BigARTM

- BigARTM хранит три вида Ф-подобных матриц: Ф, n_{wt} и поправки регуляризаторов r_{wt} (при их наличии)
- Каждая матрица хранится построчно, строка — непрерывная область памяти с подряд идущими значениями
- Это полезно при выполнении E-шага с плотными параметрами — сохраняется локальность при подсчете нормировочной константы
- Для разреженных параметров это подходит хуже:
  - многие слагаемые в скалярном произведении равны нулю
  - все нулевые значения матриц хранятся в памяти
- Имеет смысл изменить формат хранения так, чтобы
  - сохранить локальность вычислений
  - удалить из подсчёта нулевые элементы
  - не тратить память на нулевые элементы

Хранение и обработка параметров в BigARTM

- Хранение каждой строки S длины $m = |T|$ с k ненулевыми элементами по-отдельности в плотном или разреженном режиме
- Выбор режима зависит от доли $\frac{k}{m}$ ненулевых элементов в строке
- Плотный режим — непрерывный массив
- Разреженный режим — три массива:
  - V — все ненулевые элементы S в исходном порядке
  - I — индексы в S элементов из V
  - M — битовая маска длины m, индикатор ненулевых элементов
- Смысл этих массивов:
  - V — хранение ненулевых элементов
  - I — доступ к ненулевым элементам за $O(\log_2 k)$
  - I — быстрое скалярное умножение на плотный массив
  - M — доступ к нулевым элементам за $O(1)$
- Дополнительно E-шаг адаптируется к V (ϕ_w) и I (индексы для θ_d)

Эффект от использования разреженности

- **Оффлайновый алгоритм:**
  - Ускорение обучения больших разреженных моделей (выигрыш до 30%)
  - Если разреженность достигается регуляризацией, то растёт потребление памяти из-за наличия r_{wt}
- **Синхронный онлайновый алгоритм:**
  - Ускорение обучения больших разреженных моделей (выигрыш до 30%)
  - Есть две версии n_{wt}, экономия на них скрывает затраты на r_{wt} (потребление памяти аналогично плотной модели без регуляризации)
- **Алгоритм DetAsync:**
  - Ускорение обучения больших разреженных моделей (выигрыш до 25%)
  - Есть три версии n_{wt}, экономия на них скрывает затраты на r_{wt} и даёт выигрыш в памяти до 23%

Итоги занятия

- Тематическое моделирование — инструмент для кластеризации и анализа тематической структуры текстов
- Существуют различные постановки задачи и регуляризации, все они приводят к похожим моделям, обучаемым с помощью итеративных алгоритмов
- Существует набор различных методов повышения эффективности реализаций таких алгоритмов, обычно они не зависят от выбора конкретной модели
- Реализации делятся на следующие основные типы:
  - Параллельные или нет
  - С общей памятью или нет
  - Онлайновые или оффлайновые
  - С хранением Θ (n_{td}) или без
  - С распределённым хранением Φ или нет
- Библиотека BigARTM — эффективный и гибкий инструмент для построения различных моделей на одной машине