

Московский государственный университет имени М. В. Ломоносова



Факультет Вычислительной Математики и Кибернетики
Кафедра Математических Методов Прогнозирования

ВЫПУСКНАЯ КВАЛИФИКАЦИОННАЯ РАБОТА

Вероятностные тематические модели на основе данных о со-встречаемости слов

Выполнил:

студент 4 курса 417 группы

Солоткий Михаил

Научный руководитель:

д.ф-м.н., профессор

Воронцов Константин Вячеславович

Москва, 2019

Содержание

1	Введение	3
2	Тематическая модель PLSA	4
3	Аддитивная Регуляризация Тематических Моделей	5
4	Интерпретируемость тем	6
5	Когерентность тем	6
6	Модели со-встречаемостей токенов	7
6.1	PPMI, основанная на частотах пар токенов	7
6.2	PPMI, основанная на частотах документов	8
7	Регуляризатор когерентности	8
8	Алгоритм подсчёта со-встречаемостей	9
8.1	Обработка входной коллекции	10
8.2	Агрегация собранной статистики	11
8.3	Вычисление метрик	12
8.4	Требования к коллекции	13
8.5	Анализ сложности	14
8.5.1	Анализ первого этапа	14
8.5.2	Анализ второго этапа	15
8.5.3	Анализ третьего этапа	16
8.5.4	Выводы и возможные модификации	16
8.6	Реализация алгоритма	16
9	Влияние параметров алгоритма	16
9.1	Параметр batch_size	17
9.1.1	Реальное время работы при разных значениях batch_size	17
9.1.2	Затраченная внешняя память при разных значениях batch_size	18
9.2	Параметр min_merge и константа max_open	19
9.3	Выводы по алгоритму	20
10	Увеличение когерентности	20
10.1	Обучение модели PLSA	20

Аннотация

ToDo: переделай

В данной работе приводится параллельный алгоритм сбора статистики встречаемостей токенов в больших текстовых коллекциях, таких как англоязычная Википедия, а также эксперименты по измерению когерентности тематических моделей.

1 Введение

ToDo: переделай

Тематическое моделирование — это

2 Тематическая модель PLSA

Пусть D — конечное множество документов (коллекция, корпус), W — конечное множество токенов данной коллекции (словарь). Обычно токенами являются слова, но могут быть и словосочетания. Предполагается, что появление произвольного токена w в произвольном документе d связано с некоторой темой t , принадлежащей конечно-му множеству тем T . Следующими модельными предположением является гипотеза «мешка слов», которая утверждает, что для выявления тематики документа не важен порядок токенов в документе, а важны лишь частоты вхождения токенов. Также вводится гипотеза «мешка документов», которая говорит, что для выявления тематики коллекции не важен порядок документов. Эти две гипотезы дают возможность построить вероятностную модель порождения текстовой коллекции. Пространством элементарных исходов является $\Omega = W \times D \times T$, а вся коллекция рассматривается как простая выборка троек (w_i, d_i, t_i) из категориального распределения $P(w, d, t)$, причём темы токенов t_i являются скрытыми, а наблюдаются пары (w_i, d_i) . Предполагается, что вероятность появления токена в документе связана с его темой и не связана с документом, в котором он встретился. Формально это записывается так: $P(w|d, t) = P(w|t)$. Таким образом тема есть вероятностное распределение на W .

Модель PLSA (Probabilistic Latent Semantic Analysis) [3]:

$$P(w|d) = \sum_{t \in T} P(w|t) P(t|d) = \sum_{t \in T} \Phi_{wt} \Theta_{td}$$

Каждой позиции каждого токена в каждом документе присваивается своя скрытая переменная t — тема данного токена на данной позиции в данном документе. Далее будут использоваться следующие обозначения (аналогичные [10]):

n_{dwt} — счётчик, сколько раз токен w был ассоциирован с темой t в документе d ;

$n_{wt} = \sum_d n_{dwt}$ — счётчик, сколько раз в коллекции токен w ассоциирован с темой t ;

$n_{dt} = \sum_w n_{dwt}$ — количество токенов из документа d , ассоциированные с темой t ;

$n_{dw} = \sum_t n_{dwt}$ — количество вхождений токена w в документ d ;

$n_d = \sum_{wt} n_{dwt}$ — длина документа d ;

$n_w = \sum_{dt} n_{dwt}$ — количество вхождений токена w в коллекцию;

$n_t = \sum_{dw} n_{dwt}$ — количество позиций токенов в коллекции, принадлежащих теме t .

Таким образом можно выразить через эти счётчики элементы матриц Φ, Θ :

$$\varphi_{wt} = \frac{n_{wt}}{n_t}, \quad \theta_{td} = \frac{n_{td}}{n_d}$$

При этом мы наблюдаем частотные оценки $\hat{P}(w|d) = \frac{n_{dw}}{\sum_{w \in d} n_{dw}} = \frac{n_{dw}}{n_d}$. Задача состоит в восстановлении матриц Φ и Θ по выборке, то есть в нахождении описания токенов и текстовых документов с помощью смеси тем. Восстановление матриц происходит посредством максимизации неполного правдоподобия с помощью EM-алгоритма:

$$\mathcal{L}(\Phi, \Theta) = \sum_{d,w} n_{dw} \ln \sum_{t \in T} \varphi_{wt} \theta_{td} \rightarrow \max_{\Phi, \Theta}$$

Одной из популярных в компьютерной лингвистике мер качества оценки языковых моделей является перплексия. Применительно к модели PLSA она принимает вид:

$$\mathcal{P}(D) = \exp \left(-\frac{1}{n} \sum_{d \in D} \sum_{w \in d} n_{wd} \ln P(w|d) \right)$$

По сути формула представляет собой усреднённый по всем токенам коллекции логарифм правдоподобия, от которого затем взята обратная экспонента.

3 Аддитивная Регуляризация Тематических Моделей

Исходная модель PLSA обладает неединственностью решения, так как для матриц Φ, Θ можно подобрать такую стохастическую матрицу S ранга $|T|$ и объявить матрицы $\Phi' = \Phi S, \Theta' = S^{-1} \Theta$ новым решением. Это некорректно поставленная задача, которую можно попытаться подменить корректно поставленной с помощью добавления регуляризаторов к исходному функционалу [10]:

$$\mathcal{L}(\Phi, \Theta) = \sum_{d,w} n_{dw} \ln \sum_{t \in T} \varphi_{wt} \theta_{td} + \sum_{i=1}^k R_i(\Phi, \Theta) \rightarrow \max_{\Phi, \Theta}$$

В литературе в основном решают проблему неединственности решения PLSA с помощью введения априорных распределений на матрицы Φ и Θ и используются техники приближённого байесовского вывода для обучения моделей. Модель называется LDA (Latent Dirichlet Allocation) [1]. Большинство известных тематических

моделей, в том числе LDA, допускают переформулировку в терминах аддитивной регуляризации, и могут обучаться посредством максимизации неполного регуляризованного правдоподобия,

4 Интерпретируемость тем

Недостатком рассмотренных тематических моделей является то, что они никак не учитывают взаимное расположение токенов в коллекции. Согласно гипотезе дистрибутивности [11]: токены близки семантически, если совместно часто встречаются близко в тексте. Информация о том, как часто некоторые токены встречаются рядом в коллекции могла бы помочь построить более точную модель с более интерпретируемыми темами. Интерпретируемость — субъективное понятие. Процесс оценивания темы выглядит следующим образом: эксперт в некоторой предметной области получает топ k токенов некоторой темы (обычно $k = 10$) и должен без труда понять, что связывает полученные токены и дать теме адекватное название. Затем по некоторой шкале выставляет оценку: насколько тема интерпретируема.

5 Когерентность тем

В [8] проводились попытки подобрать автоматически вычисляемую меру качества темы, которая бы коррелировала с оценками, выставленными экспертами (в смысле корреляции Спирмена). Лучшей мерой качества оказалась когерентность — средняя PMI (Pointwise Mutual Information) [4] по парам среди m наиболее вероятным различным токенам темы.

$$C_t = \frac{2}{m(m-1)} \sum_{i=1}^{m-1} \sum_{j=i+1}^m \text{PMI}(w_i, w_j)$$

$$\text{PMI}(A, B) = \log \frac{\text{P}(A \cap B)}{\text{P}(A) \text{P}(B)}$$

Также в [2, 5] было показано, что использование Positive PMI вместо PMI и Shifted Positive PMI улучшает качество в задачах семантической близости слов.

$$\text{PPMI}(A, B) = \max(0, \text{PMI}(A, B))$$

$$\text{SPPMI}_k(A, B) = \max(0, \text{PMI}(A, B) - \ln k)$$

Если события A и B независимы, то вероятность пересечения событий факторизуется в произведение вероятностей по отдельным событиям и РРМІ равна 0. То есть данная мера близости присваивает значения больше 0 парам, которые встречаются вместе чаще, чем если бы их появление было независимо. Идея Shifted РМІ заключается в удалении из рассмотрения пар, которые чуть-чуть чаще встречаются вместе, чем случайно. Такие пары неинтересны, и за счёт их удаления можно повысить эффективность вычисления когерентности и уменьшить затраты по памяти.

Ясно, что понятие когерентности определено неоднозначно, так как совместную вероятность токенов может рассчитывать по-разному в зависимости от модели. Далее рассматриваются некоторые модели со-встречаемостей токенов.

6 Модели со-встречаемостей токенов

6.1 РРМІ, основанная на частотах пар токенов

Рассмотрим в документе d множество позиций токенов. Скажем, что пара позиций (i, j) в одном и том же документе находится в некотором окне ширины k , если $0 < |i - j| \leq k$. Через w_{di} обозначим токен, находящийся на позиции i в документе d . Через n_{uv} - счётчик со-встречаемости токенов u и v обозначим количество позиций (i, j) суммарно во всех документах коллекции, принадлежащих некоторому окну и таких, что $w_{di} = u$, $w_{dj} = v$. Формально можно записать следующим образом:

$$n_{uv} = \sum_{d=1}^{|D|} \sum_{i=1}^{n_d} \sum_{j=1}^{n_d} [0 < |i - j| \leq k] [w_{di} = u] [w_{dj} = v]$$

Заметим, что $n_{uv} = n_{vu}$.

Введём вероятность на множестве пар токенов:

$$P(u, v) \propto n_{uv}$$

$P(u, v)$ — совместная вероятность наблюдения пары токенов (u, v) , она же вероятность пересечения событий — наблюдение токена u и наблюдение токена v . Обозначив через $n_u = \sum_v n_{uv}$ и через $n = \sum_u n_u$, получаем:

$$P(u, v) = \frac{n_{uv}}{n} \quad P(u) = \sum_v P(u, v) = \frac{n_u}{n}$$

$P(u)$ примерно равна доле токена u среди всех токенов коллекции, равенство не точное за счёт краевых эффектов на границах документов. Итоговая формула для RPMI:

$$\text{RPMI}(u, v) = \left[\ln \frac{P(u, v)}{P(u) P(v)} \right]_+ = \max \left(0, \ln \left[\frac{n_{uv} \cdot n}{n_u \cdot n_v} \right] \right)$$

Данная модель с небольшими отличиями использовалась в [2, 5, 6].

6.2 RPMI, основанная на частотах документов

Ещё одним популярным способом задания вероятностей для RPMI является учёт количества документов, в которых фигурировала хотя бы раз данная пара токенов. В оригинальной статье [7] не использовались окна фиксированной ширины, а учитывались попадание пары токенов в 1 документ. Этот способ можно обобщить введением окон фиксированной ширины по аналогии с предыдущей моделью, основанной на частотах пар токенов.

Введём величину n_{uv} , которая равна количеству документов, в которых встретились токены u и v хотя бы раз внутри некоторого окна ширины k :

$$n_{uv} = \left| \left\{ d \in D \mid \exists (i, j) : w_{di} = u, w_{dj} = v, 0 < |i - j| \leq k \right\} \right|$$

Элементарным исходом является документ, на множестве документов введём равномерное распределение. Тогда вероятность совместно встретить токены u и v внутри некоторого окна в коллекции будет равна доле документов с данным свойством, а соответственно вероятность появления токена будет также выражаться в терминах документов — доля документов, в которых встречался данный токен хотя бы 1 раз:

$$P(u, v) = \frac{n_{uv}}{|D|} \quad P(u) = \frac{n_u}{|D|}$$

То есть количество документов нормируется на общий размер коллекции. Тогда RPMI можно переписать следующим образом:

$$\text{RPMI}(u, v) = \frac{n_{uv} \cdot |D|}{n_u \cdot n_v}$$

7 Регуляризатор когерентности

В [7] была предложена процедура оптимизации когерентности. Идеи, описанные в данной статье были заложены в основу регуляризатора когерентности, который был

описан в [10]. Процедура оптимизации из статьи [7] может быть приближённо сформулирована в виде следующего регуляризатора:

$$R_{doc_coher}(\Phi) = \tau \sum_{t \in T} \sum_{(u,v) \in W^2} \frac{N_{uv}}{N_v} n_{vt} \ln \varphi_{vt}$$

где N_{uv} — количество документов, в которых встретились вместе токены u и v (не в некотором окне, а в принципе в документе), N_u — количество документов, в которых встречается токен u , n_{vt} — количество раз, сколько токен v был связан с темой t в коллекции. Данный регуляризатор является частным случаем более общего:

$$\begin{aligned} R(\Phi) &= -\tau \sum_{t \in T} n_t \text{KL}_u(\hat{P}(u|t) \parallel \varphi_{ut}) = -\tau \sum_{t \in T} n_t \sum_{u \in W} \hat{P}(u|t) \ln \frac{\hat{P}(u|t)}{\varphi_{ut}} \\ \arg \max_{\Phi} R(\Phi) &= \arg \max_{\Phi} \left[\tau \sum_{t \in T} n_t \sum_{u \in W} \hat{P}(u|t) \ln \varphi_{ut} \right] = \\ &= \arg \max_{\Phi} \left[\tau \sum_{t \in T} n_t \sum_{u \in W} \hat{P}(u|t) \ln \varphi_{ut} \right] = \\ &\left\{ \hat{P}(u|t) = \sum_{v \in W} P_{doc}(u|v, t) P(v|t), \quad P_{doc}(u|v, t) = P_{doc}(u|v) \right\} = \\ &= \arg \max_{\Phi} \left[\tau \sum_{t \in T} n_t \sum_{(u,v) \in W^2} P_{doc}(u|v) P(v|t) \ln \varphi_{ut} \right] = \\ &= \arg \max_{\Phi} \left[\tau \sum_{t \in T} n_t \sum_{(u,v) \in W^2} P_{doc}(u|v) \frac{n_{vt}}{n_t} \ln \varphi_{ut} \right] = \arg \max_{\Phi} \left[\tau \sum_{t \in T} \sum_{(u,v) \in W^2} \frac{N_{uv}}{N_v} n_{vt} \ln \varphi_{ut} \right] \end{aligned}$$

Ничего не мешает заменить $P_{doc}(u|v)$ на условные вероятности другой вероятностной модели, главное — определить в соответствующей модели счётчики n_{uv} , так как

$$P(u|v) = \frac{P(u, v)}{P(v)} = \frac{n_{uv}}{\sum_u n_{uv}}$$

8 Алгоритм подсчёта со-встречаемостей

В данной работе предложен алгоритм обработки текстовых коллекций и подсчёта статистики со-встречаемостей пар токенов. Особенности данного алгоритма:

- асинхронная обработка входной коллекции и промежуточных данных;

- возможность обработки коллекций потенциально неограниченных по числу документов;
- возможность улучшения производительности за счёт увеличения числа потоков-обработчиков на 1 вычислительном узле

Условно алгоритм можно разбить на 3 этапа:

- обработка входной коллекции;
- агрегация статистики, собранной по коллекции;
- вычисление метрик на основе статистики со-встречаемости.

8.1 Обработка входной коллекции

Обработка входной коллекции происходит по батчам, т.е. в оперативную память считывается *batch_size* документов коллекции и считаются со-встречаемости на этом подмножестве документов, как будто они составляют всю коллекцию. Предполагается, что размер каждого документа ограничен общей константой, иначе нельзя было бы гарантировать, что каждый документ можно считать в оперативную память. Параметр *batch_size* указывается пользователем. Также пользователь может указать некоторый словарь релевантных для него токенов — *vocab*, и со-встречаемости будут считаться только между токенами из данного множества. *vocab* может представлять собой словарь коллекции W , очищенный от стоп-слов и низкочастотных токенов, или он может состоять из токенов некоторого узкого набора тем. Для больших коллекций рекомендуется задавать *vocab*. Дело в том, что множество всех токенов некоторого естественного языка может быть довольно большим — около миллиона токенов или больше, и на хранение статистики по каждой паре в батче будет расходоваться много памяти. Таким образом, если задать большой *vocab* или не задавать его вовсе, не получится использовать большие батчи, так как при этом не хватило бы памяти для хранения большого количества различных пар токенов, встретившихся в батче. Статистика со-встречаемостей по каждому батчу сохраняется в файл в отсортированном формате для того, чтобы потом можно было легко объединить файлы, соответствующие всем различным батчам в один. Также для вычисления RPI нужно значение n — общее количество рассмотренных пар в коллекции. Это значение можно вычислить на первом этапе алгоритма. Обработка батчей происходит параллельно асинхронно в n_{jobs} потоков по 1 батчу на поток. Для каждого документа подсчитываются частоты со-встречаемостей пар токенов внутри некоторого окна ширины *window_width*,

которая является параметром алгоритма. Если некоторое окно выходит за границы документа, оно усекается до соответствующих границ.

Этап 1. Обработка входной коллекции.

Вход: коллекция D , $vocab$, $batch_size$, $window_width$, n_{jobs} ;

Выход: набор отсортированных файлов со-встречаемостей F , общее количество пар n ;

```

1  $n \leftarrow 0$ ;
2 пока не конец коллекции
3   для всех  $job = 1, \dots, n_{jobs}$ 
4     инициализировать пустой сортирующий контейнер  $C$ ;
5     считать  $batch\_size$  документов в  $batch$ ;
6     для всех  $d \in batch$ 
7       для всех  $i = 1, \dots, n_d - 1$ 
8         для всех  $j = 1, \dots, window\_width : i + j \leq n_d$ 
9            $r = i + j$ ;
10           $C[w_{di}, w_{dr}] \leftarrow C[w_{di}, w_{dr}] + 1$ ;
11           $C[w_{dr}, w_{di}] \leftarrow C[w_{dr}, w_{di}] + 1$ ;
12           $n \leftarrow n + 2$ 
13   сохранить  $C$  во внешнюю память;
```

В алгоритме используется сортирующий контейнер C . Для эффективной обработки коллекции необходимо, чтобы была возможность быстрого доступа к его элементам и их изменение. Для реализации было выбрано двухуровневое красно-чёрное дерево: вначале по ключу (токену u) находятся все токены, с которыми он встречался в коллекции внутри некоторого окна, а затем снова по ключу (токену v) во внутреннем красно-чёрном дереве ищется значение со-встречаемости пары (u, v) . Файлы построены по тому же принципу, то есть сначала указан первый токен пары, а затем все, с которыми он встретился и значение со-встречаемости. Красно-чёрное дерево было выбрано, так как оно позволяет иметь быстрый доступ к данным и поддерживает их отсортированными.

8.2 Агрегация собранной статистики

Основой второго этапа является сортировка во внешней памяти с помощью алгоритма k -way merge ¹. Алгоритм в случае $k = 2$ совпадает со слиянием 2 отсортированных массивов, а в случае $k > 2$ строит бинарную кучу на минимальных

¹https://en.wikipedia.org/wiki/K-way_merge_algorithm

элементах массивов. В данном случае массивы — это файлы, а элементу массива соответствует запись в файле, которая содержит некоторый токен u и все токены, с которыми он встречался в некотором батче, а также значение со-встречаемости. Данный алгоритм не требует хранения содержимого файлов целиком в оперативной памяти, поэтому идеально подходит для сортировки во внешней памяти. В оперативной памяти нужно хранить лишь 1 запись для каждого файла. Так как число файлов потенциально неограниченно, алгоритм в явном виде нельзя применять, и используется его модифицированная версия: если количество файлов слишком большое, вначале происходит серия последовательных слияний малых порций файлов с помощью стандартного k -way merge до тех пор, пока количество массивов не станет меньше заданной константы. Ситуация, когда количество файлов слишком велико и происходит предварительное слияние некоторых порций файлов, на практике реализуется в случаях больших коллекций или малых значений $batch_size$. Так как для слияния необходимо держать файлы открытыми, количество одновременно объединяемых файлов не может превосходить максимального количества открытых файлов в одном процессе. Эта константа выставлена в ядре операционной системы. Если бы количество файлов превышало эту константу, пришлось бы производить открытие-закрытие файлов, что сильно сильно замедлило бы работу алгоритма, так системные вызовы отрабатывают долго.

Все слияния можно производить многопоточно: каждый поток будет иметь свой локальный набор файлов. Многопоточное слияние нужно прекратить по достижении какого-то критически малого количества файлов и произвести окончательное однопоточное слияние оставшихся файлов. Также на этом этапе алгоритма удобно посчитать частоты токенов n_u , которые будут использованы для вычисления RPI. Во время последнего слияния файлов можно не записывать в итоговый файл те пары, значения со-встречаемости которых ниже некоторого заданного порога, так как низкочастотные пары не несут статистически значимой информации. Также это поможет сэкономить время на 3 этапе алгоритма. Также для экономии внешней памяти по завершении слияния некоторой порции файлов и записи объединённых данных в новый, предыдущие файлы можно сразу удалить.

8.3 Вычисление метрик

Последний этап алгоритма заключается в вычислении метрик по имеющейся статистике со-встречаемости. В текущей реализации считается только RPI, однако добавление других метрик не составляет труда.

Этап 2. Слияние отсортированных файлов.

Вход: набор отсортированных файлов со-встречаемостей F ,
минимальное число файлов для окончательного слияния min_merge ,
максимальное число открытых файлов max_open ,
число потоков n_jobs , минимальное значение со-встречаемости min_cooc ;

Выход: файл со-встречаемостей f , частоты токенов n_u ;

```
1 пока  $|F| > min\_merge$ 
2    $F' \leftarrow \{\}$ ;
3   пока  $|F| > 0$ 
4     для всех  $i = 1, \dots, n\_jobs$ 
5        $batch\_size \leftarrow \min(\lfloor \frac{max\_open}{n\_jobs} \rfloor, |F|)$ ;
6        $batch \leftarrow F_1, \dots, F_{batch\_size}$ ;
7        $F \leftarrow F \setminus batch$ ;
8        $f \leftarrow k\text{-way merge}(batch)$ ;
9       Добавить  $f$  в  $F'$ ;
10   $F \leftarrow F'$ 
11  $f \leftarrow k\text{-way merge}(F, min\_cooc)$ ;
```

Этап 3. Вычисление RPMI.

Вход: файл со-встречаемостей f ;

Выход: файл RPMI f' ;

```
1 для всех  $(u, v, n_{uv}) \in f$ 
2    $ratio \leftarrow \frac{n_{uv}n}{n_u n_v}$ ;
3   если  $ratio > 1$  то
4      $RPMI \leftarrow \log(ratio)$ ;
5      $f' \leftarrow RPMI$ ;
```

8.4 Требования к коллекции

Для корректной работы алгоритма коллекция должна быть представлена в формате Wowpal Wabbit. Также на коллекцию и словарь *vocab* накладываются следующие ограничения:

- (1) оперативной памяти хватает для хранения словаря *vocab*;
- (2) любой документ целиком можно записать в оперативную память;
- (3) оперативной памяти хватает для хранения записей из промежуточных файлов (по одной записи на каждый открытый файл);
- (4) внешней памяти хватает для хранения всех промежуточных файлов;

- (5) оперативной памяти хватает для хранения всех пар токенов любого батча и счётчиков со-встречаемостей в двухуровневом красно-чёрном дереве;
- (6) оперативной памяти хватает для хранения всех пар токенов словаря и счётчиков со-встречаемостей в двухуровневом красно-чёрном дереве.

Достаточно выполнения ограничений (1 — 4) и любого из (5), (6).

8.5 Анализ сложности

8.5.1 Анализ первого этапа

В самом начале алгоритма в память записывается словарь *vocab* для отслеживания релевантных токенов. Также каждому токеноу из *vocab* сопоставляется число и после обработки коллекции работа происходит не с токенами, а с числами для эффективно-го хранения статистики со-встречаемостей. Отображение токенов в числа и обратно хранятся в хеш-таблице и в массиве соответственно. Построение этих структур занимает $\mathcal{O}(|vocab|)$ времени, а обращение к данным по ключу — $\mathcal{O}(1)$. Дальнейшая часть первого первого этапа алгоритма работает за линейное время от общей длины коллекции и ширины окна, причём эти параметры входят как множители в оценку, так как для почти всех токенов коллекции (за исключением первых и последних *window_width* токенов в каждом документе) необходимо просмотреть *window_width* токенов, стоящих справа. Также для каждой найденной пары токенов необходимо изменить счётчики со-встречаемости в красно-чёрном дереве, то есть время работы первого этапа есть $\mathcal{O}(N \cdot window_width \cdot \log(n_{pairs}))$, где N — длина коллекции в токенах, а n_{pairs} — верхняя оценка на количество пар в красно-чёрном дереве. Так как одновременно в память пишется *batch_size* документов, можно утверждать $n_{pairs} = window_width \cdot \sum_{i=1}^{batch_size} n_d = \{\text{ограничение (4)}\} = const$. С другой стороны, можно получить другую верхнюю оценку $\hat{n}_{pairs} = |vocab|^2 = \{\text{ограничение (5)}\} = const$. В итоге время оценивается как $\mathcal{O}(|vocab| + N \cdot window_width)$. Можно также оценивать через $\mathcal{O}(|vocab| + |D| \cdot window_width)$, так как размеры всех документов ограничены одной константой.

Оперативная память на первом этапе тратится на содержание батча документов и красно-чёрного дерева. Отсюда условия (2), (5), (6). Также так как в самом начале этапа в память был записан словарь, создана хеш-таблица и массив обратных индексов, памяти требуется как минимум $\mathcal{O}(|vocab|)$. При выполнении условий (5) или (6) можно гарантировать, что затраты по памяти будут $\mathcal{O}(|vocab| + n_{jobs} \cdot batch_size)$, так как $\mathcal{O}(|vocab|)$ тратится на содержание *vocab*, а $\mathcal{O}(n_{jobs} \cdot batch_size)$ — на хранение документов в памяти, а расходы на хранение красно-чёрного дерева занимают

константу. Выполнения условия (5) можно добиться, если предполагать, что отдельный документ мал по размеру, что как раз реализуется на практике: обычно текстовые документы представляют собой статьи из Википедии, новости или посты в социальных сетях.

Суммарный объём файлов на внешнем устройстве можно оценить как $\mathcal{O}(|D|)$, так как размер каждого документа заранее ограничен.

8.5.2 Анализ второго этапа

В стандартном k -way merge время доступа к минимальному элементу есть $\mathcal{O}(1)$, а время перестроения после изменения положения указателя в одном файле есть $\mathcal{O}(\log k)$, где k — количество файлов. Общее время работы есть $\mathcal{O}(N \cdot \log k \cdot |vocab|)$, так как для слияния может понадобиться $\mathcal{O}(|vocab|)$ операций. Здесь N — верхняя оценка на количество записей в файлах. В данном случае записью является последовательность из пар с одинаковым первым токеном. Так как в данном алгоритме используется модификация k -way merge, оценки сложности немного отличаются от стандартного случая. На первом проходе алгоритма всё множество файлов разбивается на непересекающиеся части, которые обрабатываются параллельно. Всего частей $\frac{|D| \cdot n_{jobs}}{batch_size \cdot max_open} + \mathcal{O}(1)$, и n_{jobs} частей можно обрабатывать одновременно. То есть время работы на первом этапе оценивается как $\mathcal{O}(\frac{|D| \cdot n_{jobs}}{batch_size \cdot max_open} \cdot \frac{1}{n_{jobs}} \cdot T)$, где T — время обработки одной части, которое оценивается для стандартного k -way merge как $\mathcal{O}(|vocab| \cdot \log \frac{max_open}{n_{jobs}} \cdot |vocab|)$, и итоговая сложность на первом этапе равна $\mathcal{O}(\frac{|D| \cdot |vocab|^2}{batch_size})$. Остальные переменные в формуле оцениваются сверху константой. Всего проходов по файлам до достижения минимального количества происходит $\lceil \log_{max_open} \frac{|F|}{min_merge} \rceil$, поэтому общее время работы можно оценить как $\mathcal{O}(\frac{|D| \cdot \log |D| \cdot |vocab|^2}{batch_size})$.

На втором этапе требуется хранить для каждого открытого файла по одной записи из файла, что потенциально может занимать $\mathcal{O}(|vocab| \cdot max_open)$ оперативной памяти. Здесь необходимо воспользоваться условием (3).

Так как на втором этапе файлы объединяются посредством слияния, а слияние может происходить одновременно только по max_open файлам, можно для дополнительно затраченной внешней памяти ввести оценку $\mathcal{O}(|vocab|^2 \cdot max_open)$, так как каждый новый файл будет содержать не более $|vocab|^2$ записей константной длины. Также справедлива оценка с первого этапа: $\mathcal{O}(|D|)$, так как после слияния суммарный размер полученных дополнительных файлов не будет превышать суммарного размера исходных файлов.

8.5.3 Анализ третьего этапа

Третий этап алгоритма тривиальный, требует $\mathcal{O}(|vocab|^2)$ времени, $\mathcal{O}(1)$ оперативной памяти и $\mathcal{O}(|vocab|^2)$ внешней памяти.

8.5.4 Выводы и возможные модификации

Алгоритм имеет время работы $\mathcal{O}(|D| \cdot \log|D|)$ от размера коллекции и $\mathcal{O}(|vocab|^2)$ от размера *vocab* и требует $\mathcal{O}(1)$ оперативной памяти, а также $\mathcal{O}(|D|)$ внешней памяти, что делает возможным обработку больших коллекций при соблюдении условий (1 — 4) и одного из (5) или (6). За счёт того, что внешней памяти требуется не константа, алгоритм имеет ограничение на объём обрабатываемых коллекций, однако можно его модифицировать: применять к коллекции некоторого размера по частям, получать словари со-встречаемостей на больших частях коллекции и потом делать слияние получившихся файлов со-встречаемостей, тогда размер потребляемой внешней памяти не будет зависеть от длины коллекции, а будет зависеть от размера *vocab* как $\mathcal{O}(|vocab|^2)$

8.6 Реализация алгоритма

Описанный выше алгоритм был реализован в библиотеке тематического моделирования с открытым кодом BigARTM [9]^{2 3 4} на языке C++. Язык был выбран из соображений эффективности и простоты использования стандартных структур данных. Например, в качестве красно-чёрного дерева использовался контейнер `std::map`.

9 Влияние параметров алгоритма

Для некоторых параметров алгоритма тяжело сказать, исходя из описания алгоритма и его анализа, как эти параметры будут влиять на время работы и затраченное место на внешнем носителе. В данном разделе приводятся результаты экспериментов по измерению потребления ресурсов при различных значениях параметров. Все эксперименты проводились на ноутбуке с SSD и 8-ядерным процессором 2,3 GHz Intel Core i5.

²<http://bigartm.org> — сайт проекта BigARTM.

³http://docs.bigartm.org/en/master/tutorials/python_userguide/coherence.html — тьюториал по измерению когерентности в BigARTM.

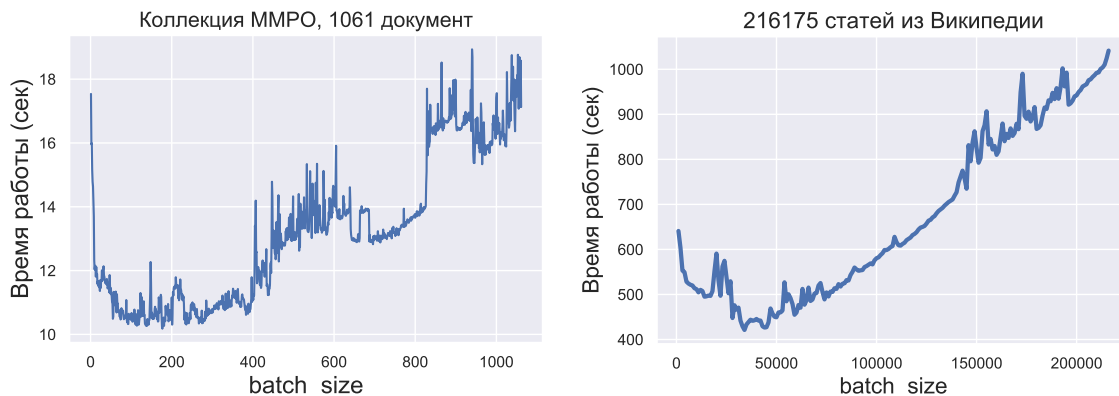
⁴http://docs.bigartm.org/en/master/tutorials/bigartm_cli.html — документация к CLI BigARTM и инструкция по запуску алгоритма.

9.1 Параметр `batch_size`

Некоторые пары токенов встречаются в большом количестве документов, поэтому если пара (i, j) встретилась в разных батчах, она будет записана в разные файлы несколько раз. Чтобы избежать подобного дублирования, стоит увеличивать параметр `batch_size`, однако вместе с его увеличением растёт затрачиваемая оперативная память, и при больших значениях `batch_size` есть риск, что очередной `batch` документов не поместится в оперативную память, и программа завершится с ошибкой. С другой стороны, если коллекцию можно записать целиком в оперативную память, для минимизации времени работы лучше не брать значение `batch_size = |D|`, так как никакого эффекта от параллелизма не будет.

9.1.1 Реальное время работы при разных значениях `batch_size`

Проводилось несколько экспериментов по замеру реального времени работы алгоритма на коллекциях малого, среднего и большого размеров. В качестве малой коллекции была взята коллекция записей выступлений на конференции ММРО (1069 документов, 7805 уникальных токенов и 804423 токенов всего), а в качестве коллекции среднего размера — набор статей из русскоязычной и англоязычной Википедии (216175 документов, 196749 уникальных токенов и 70536525 токенов всего). Словари `vocab` обеих коллекций совпадали с полными словарями коллекций W , а параметр ширины окна `window_width = 10`. Для определения зависимости времени работы от параметра `batch_size` алгоритм запускался для каждого значения от 1 до $|D|$ включительно с шагом 1 для маленькой коллекции и с шагом 1000 — для большой. Полученные графики времени работы представлены ниже. Для статей из Википедии на графике указаны замеры для значений `batch_size`, начинающихся с 1001, так как для `batch_size = 1` время работы около 2000 секунд.



Видно, что на заключительном участке время растёт, то есть эффект от параллелизма исчезает, а время обработки коллекции становится временем обработки самого

большого батча. На начальном участке время работы становится меньше с увеличением размера батча, а глобальный минимум достигается на коллекции ММРО при значении $batch_size =$ и на коллекции статей из русскоязычной и англоязычной Википедии при $batch_size = 176$, то есть когда $batch_size \approx \frac{|D|}{8}$, что согласуется с ожиданиями, так как продолжение наращивания $batch_size$ ведёт к тому, что первый этап алгоритма работает дольше. Также эксперименты проводились при параметре $n_{jobs} \in [1, 8]$, результаты такие же: довольно хорошим приближением к оптимальному значению $batch_size$ будет значение $\frac{|D|}{n_{jobs}}$. Соответственно рекомендация состоит в том, чтобы выбирать настолько большое значение, которое позволяет оперативная память, но не больше, чем $\frac{|D|}{n_{jobs}}$. Значения больше, чем $\frac{|D|}{n_{jobs}}$ есть смысл выставлять, только если есть жёсткие ограничения на объём доступного места на внешнем носителе.

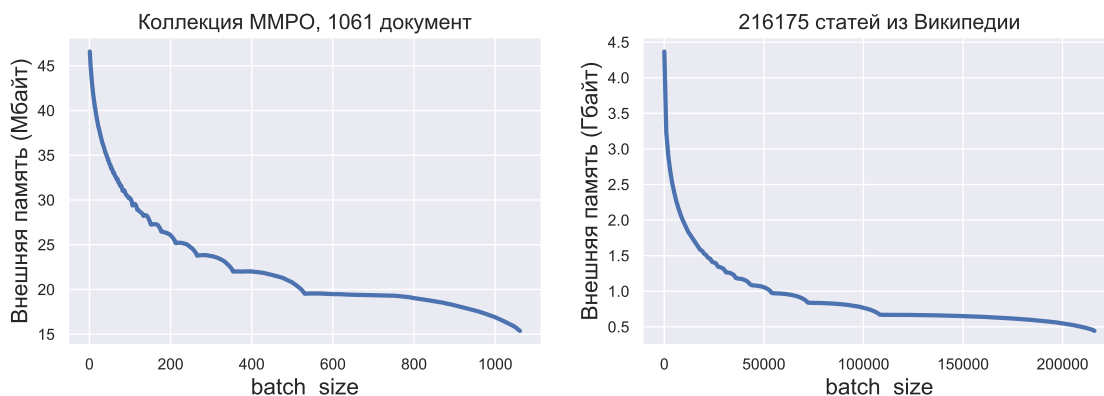
Также было проведено 2 эксперимента с большими коллекциями:

- 8446835 статей англоязычной Википедии, 8272855 уникальных токенов, 3832966193 токенов всего, $window_width = 10$, $batch_size = 2500$
- посты социальных сети (4528512 документов), 87494 уникальных токенов, 1613807215 токенов всего, $window_width = 10$, $batch_size = 10000$

При значениях остальных параметров как в предыдущих экспериментах время работы составило **4 часа 18 минут** на Википедии и **2 часа 45 минут** на постах социальных сетей. Оперативной памяти было затрачено во всех экспериментах не более **16 GB**.

9.1.2 Затраченная внешняя память при разных значениях $batch_size$

Значение $batch_size$ стоит выбирать большим также из соображений экономии памяти на внешнем устройстве. В этом эксперименте замерялся суммарный размер файлов, которые получались в результате работы первого этапа алгоритма. Параметры алгоритма и коллекции те же, что и в экспериментах по замеру времени. Исходный размер коллекции ММРО составляет примерно 14.2 МВ, а коллекция статей из русскоязычной и англоязычной Википедии занимает примерно 795.6 МВ.



Если $vocab = W$, то при запуске на больших коллекциях затраченная на внешнем носителе память может сильно превышать размер исходной коллекции (как в эксперименте со статьями из Википедии, результаты ниже), поэтому рекомендуется урезать $vocab$ посредством, к примеру, удаления низкочастотных токенов.

В экспериментах с большими коллекциями алгоритм запускался с теми же параметрами, как и эксперименте по замеру времени. Статьи Википедии занимают около **21.94 GB**, а при работе алгоритма промежуточные файлы занимают около **131.17 GB**. Коллекция постов социальных сетей занимала исходно около **27.9 GB**, а промежуточные файлы — около **59.2 GB**. Оперативной памяти во всех экспериментах было затрачено не больше **16 GB**.

9.2 Параметр `min_merge` и константа `max_open`

Параметр `min_merge` отвечает за то минимальное количество файлов, при котором слияние производится параллельно. Если количество файлов меньше данного, слияние производится последовательно. Выбор оптимального значения параметра `min_merge` неочевиден: с одной стороны использование вычислительных мощностей может помочь уменьшить время слияния файлов, с другой стороны, файлы будут больше раз записываться во внешнюю память и считываться из неё. Здесь всё зависит от скорости записи и чтения с внешнего носителя, от общего объёма данных, количества ядер процессора. Также наши эксперименты показывают, что на больших коллекциях есть смысл выставить этот параметр максимально возможным, поэтому в реализации в BigARTM этот параметр зафиксирован. Значение параметра `min_merge` не является фактически максимальным, которое позволяет операционная система, но это некоторая нижняя оценка на максимальное значение. Само максимальное значение не используется, так как процесс при этом может обрабатывать другие файлы.

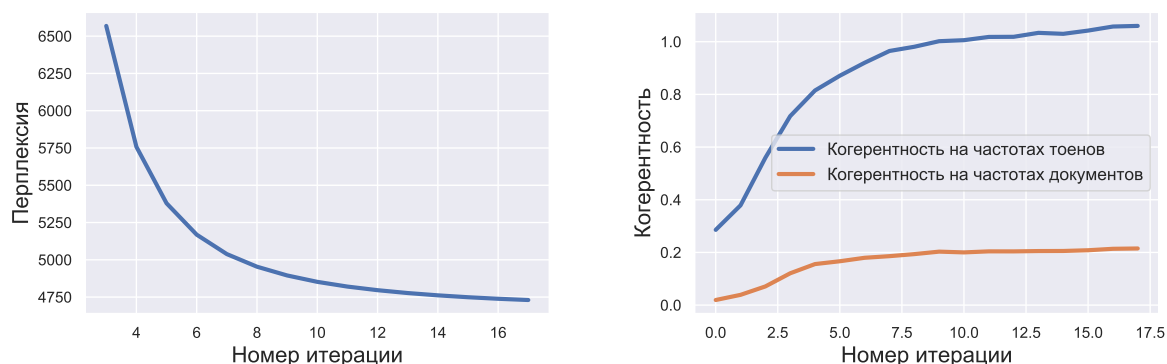
9.3 Выводы по алгоритму

Алгоритм можно запускать на ноутбуке, и за приемлемое время получать результат. На практике со-встречаемости по большой коллекции подсчитываются один раз перед построением тематической модели или проведением других экспериментов с использованием данной информации. Относительно времени, которое ушло на сбор статистики по коллекции, дальнейшие исследования занимают, как правило, намного больше времени. Также дальнейшее обновление статистики в случае пополнения коллекции не вызывает сложностей: надо лишь запустить алгоритм ещё раз на дополнительной части коллекции, объединить файлы со-встречаемостей и пересчитать PPMI.

10 Увеличение когерентности

10.1 Обучение модели PLSA

Была обучена модель PLSA на коллекции новостных статей «NY Times». В коллекции 216175 документов. Была произведена предварительная токенизация, лемматизация и удаление стоп-слов. С помощью описанного выше алгоритма была предварительно собрана статистика со-встречаемостей на англоязычной википедии и посчитаны PPMI. Обучение проводилось с помощью библиотеки BigARTM. Количество тем было взято равным 25. Это значение было взято произвольно, точное количество «тем» в коллекции неизвестно. После каждой итерации EM-алгоритма замерялась средняя по всем темам когерентность и перплексия модели. Ниже представлены графики перплексии и двух видов когерентности (основанная на частотах пар токенов и на частотах документов соответственно) после каждой итерации EM-алгоритма. Когерентность считалась по 10 наиболее вероятным токенам каждой темы.



Видно, что оба вида когерентности растут по мере увеличения правдоподобия.

Список литературы

- [1] Blei, D.M., Ng,A.Y., Jordan,M.I.: Latent dirichlet allocation. the Journal of machine Learning research 3, 993–1022 (2003)
- [2] Bullinaria, J.A. & Levy, J.P. Behavior Research Methods (2007) 39: 510. <https://doi.org/10.3758/BF03193020>
- [3] Hofmann, T.: Probabilistic latent semantic analysis. In: Proceedings of the Fifteenth Conference on Uncertainty in Artificial Intelligence. pp. 289–296. UAI’99, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA (1999)
- [4] Kenneth Ward Church and Patrick Hanks. 1990. Word association norms, mutual information, and lexicography. Comput. Linguist. 16, 1 (March 1990), 22–29.
- [5] O. Levy and Y. Goldberg. Neural word embedding as implicit matrix factorization. In Advances in Neural Information Processing Systems, pages 2177–2185, 2014.
- [6] Levy, O., Goldberg, Y., & Dagan, I. (2015). Improving Distributional Similarity with Lessons Learned from Word Embeddings. Transactions Of The Association For Computational Linguistics, 3, 211–225.
- [7] Mimno, D., Wallach, H.M., Talley, E., Leenders, M., McCallum, A.: Optimizing semantic coherence in topic models. In: Proceedings of the Conference on Empirical Methods in Natural Language Processing. pp. 262–272. EMNLP ’11, Association for Computational Linguistics, Stroudsburg, PA, USA (2011)
- [8] Newman, D., Lau, J.H., Grieser, K., Baldwin, T.: Automatic evaluation of topic coherence. In: Human Language Technologies: The 2010 Annual Conference of the North American Chapter of the Association for Computational Linguistics. pp. 100–108. HLT ’10, Association for Computational Linguistics, Stroudsburg, PA, USA (2010)
- [9] Vorontsov K., Frei O., Apishev M., Romov P., Dudarenko M. BigARTM: Open Source Library for Regularized Multimodal Topic Modeling of Large Collections // Analysis of Images, Social Networks and Texts. 2015.
- [10] Vorontsov, K., Potapenko, A.: Additive regularization of topic models. Machine Learning 101(1), 303–323 (2015)
- [11] Zellig Harris. Distributional structure. Word, 10(23):146–162, 1954.