Министерство образования и науки Российской Федерации
Федеральное государственное автономное образовательное учреждение
высшего образования
«Южно-Уральский государственный университет
(национальный исследовательский университет)»
Институт естественных и точных наук
Факультет математики, механики и компьютерных технологий
Кафедра прикладной математики и программирования
Направление подготовки Прикладная математика и информатика

РАБОТА ПРОВЕРЕНА
Рецензент, директор Челябинского подразделения ООО «Компас-плюс».
_____/Ю. В. Гасников
«____»_____2018г.

ДОПУСТИТЬ К ЗАЩИТЕ
Заведующий кафедрой, д.ф.-м.н., доцент
_____/А.А. Замышляева
«____»_____2018 г.

Разработка и исследование алгоритмов повышения релевантности результатов поиска в локальной поисковой системе

ПОЯСНИТЕЛЬНАЯ ЗАПИСКА
К ВЫПУСКНОЙ КВАЛИФИКАЦИОННОЙ РАБОТЕ
ЮУрГУ–01.04.02.2018.183.ПЗ ВКР

Руководитель работы, доцент
_____/Т.Ю. Оленчикова
«____»_____2018 г.
Автор работы
Студент группы ЕТ-222
_____/ А.Д. Марченко
«____»_____2018 г.
Нормоконтролер, доцент
_____/Д.А. Дрозин
«____»_____2018 г.

Челябинск 2018

АННОТАЦИЯ

Марченко А.Д. Разработка и исследование алгоритмов повышения релевантности результатов поиска в локальной поисковой системе. – Челябинск: ЮУрГУ, ЕТ-222, 82 с., 18 ил., 1 табл., библиогр. список – 21 наим., 1 прил.

Данная выпускная квалификационная работа посвящена исследованию и разработке методик повышения релевантности результатов поиска в локальных поисковых системах. Работа выполнена по заказу компании ООО «Компас Плюс» с целью внедрения в программные комплексы компании.

В работе рассматриваются существующие методики повышения релевантности поиска, а также распространённые подходы к оценке качества работы поисковых систем, после чего приводится обоснование их сложно применимости к локальным поисковым системам.

В ходе работы приводятся модели контекстной синонимической связи на основе обученной тематической модели, как способ решения задачи поиска синонимов в локальной поисковой системе. Также описывается алгоритм расширения словаря тематической модели за счёт терминов, отсутствующих в словаре поискового индекса, и алгоритм взаимодействия поисковой системы с тематической моделью.

Также в работе приводятся результаты внедрения в поисковую систему всех разработанных моделей и алгоритмов, оценка результатов производится посредством описанных в работе критериев.

ОГЛАВЛЕНИЕ

ВВЕДЕНИЕ

В наше время объём накапливаемой и обрабатываемой информации неуклонно растёт, постепенно превышая пределы, в которых возможно вручную в массиве накопленной информации найти необходимые данные. Конечно, в распоряжении человека всегда есть большие поисковые системы: Google, Yandex и др., однако не вся информация доступна из Сети Интернет.

Особенно ярко эта проблема проявляется в пределах корпоративных баз знаний. Большие объёмы накапливающейся в них информации зачастую являются конфиденциальными, следовательно, нет возможности опубликовать её на каком-либо сайте в Сети Интернет и пользоваться возможностями глобальных поисковых систем. Таким образом, возникает необходимость в создании своей поисковой системы, нацеленной на поиск исключительно по внутрикорпоративным базам знаний.

Длительное время для таких целей компании использовали обычный полнотекстовый поиск, который реализуется исключительно возможностями операционных систем, либо, в случае если информация хранится в базе данных, SQL-запросами различной степени сложности. Однако, со временем объём накопленной информации становится всё больше, а её разнородность не повышается, ведь компании накапливают информацию, преимущественно, из области своей деятельности. Таким образом, даже при условии использования баз данных для хранения всей накопленной информации, даже если эта информация хорошо структурирована, найти именно то, что нужно, становится всё сложнее. Следствием из описанной ситуации и становится необходимость в разработке локальной поисковой системы, которая бы обладала функционалом, максимально похожим на функционал глобальных поисковых систем, однако действовала бы лишь в пределах базы знаний отдельно взятой компании.

Характерными чертами локальных поисковых систем можно назвать относительно низкие доступные вычислительные мощности, которые компания

может выделить для обучения вспомогательных модулей поисковой системы (классификаторов, агрегаторов, нейронных сетей и т.д.), а также «средний» объём информации в области поиска, то есть слишком большой, чтобы пользоваться обычным полнотекстовым поиском, и слишком маленький для того, чтобы использовать некоторые технологии, свободно распространяемые крупными поисковиками.

К тому же, в отличие от глобальных поисковиков, для оценки которых применяются подходы, связанные с посещаемостью поисковика, его рейтингами в различных опросах, и т.д., чтобы оценить локальную поисковую систему, требуются корректные количественно-качественные критерии, специально разработанные под данную систему и имеющие строгое математическое обоснование.

1 ОБЗОР СУЩЕСТВУЮЩИХ МЕТОДОВ ПОСТРОЕНИЯ, ОЦЕНКИ И УЛУЧШЕНИЯ КАЧЕСТВА РАБОТЫ ПОИСКОВЫХ СИСТЕМ

1.1 Поисковая система

В условиях постоянно возрастающего количества накопленной информации растёт также и потребность в системах, ориентированных на поиск по хранилищам данных. Такие системы называются поисковыми. За годы разработки и совершенствования поисковых систем их структура приняла следующий стандартный вид:

1.1.1 Поисковый индекс

Поисковым индексом называется особым образом структурированное хранилище информации о документах, по которым ведётся поиск. Существует множество способов организации структуры поискового индекса, среди них:

1) обычный индекс – представляет собой сильно разреженную матрицу размера , где n – количество терминов в словаре поисковой системы, а m – количество документов в этой системе. Значение на i,j позиции показывает, сколько раз i-й термин встречается в j-м документе:

где

значение, соответствующее числу вхождений термина

в документ

словарь поискового индекса,

-количество терминов в словаре

-коллекция документов в области поиска,

количество документов в коллекции.

2) инвертированный индекс – представляет собой структуру данных, в которой каждому слову из словаря коллекции документов

посредством списка сопоставляются все документы d из коллекции документов , в которых оно встречается. Инвертированный индекс наиболее хорошо подходит для реализации поиска по текстам.

Существует два основных варианта устройства инвертированного индекса поисковой системы:

a. индекс, в котором для каждого слова содержатся исключительно документы, в которых оно встречается;
b. индекс, дополнительно включающий позицию слова в каждом документе.

1.1.2 Поисковая строка

Поисковая строка – это не только поле для ввода поискового запроса, но и огромное количество инструментов, существующих для распознавания запроса, разбиения его для отдельных блоков и преобразования в форму, понятную поисковой машине (рисунок 1.1).

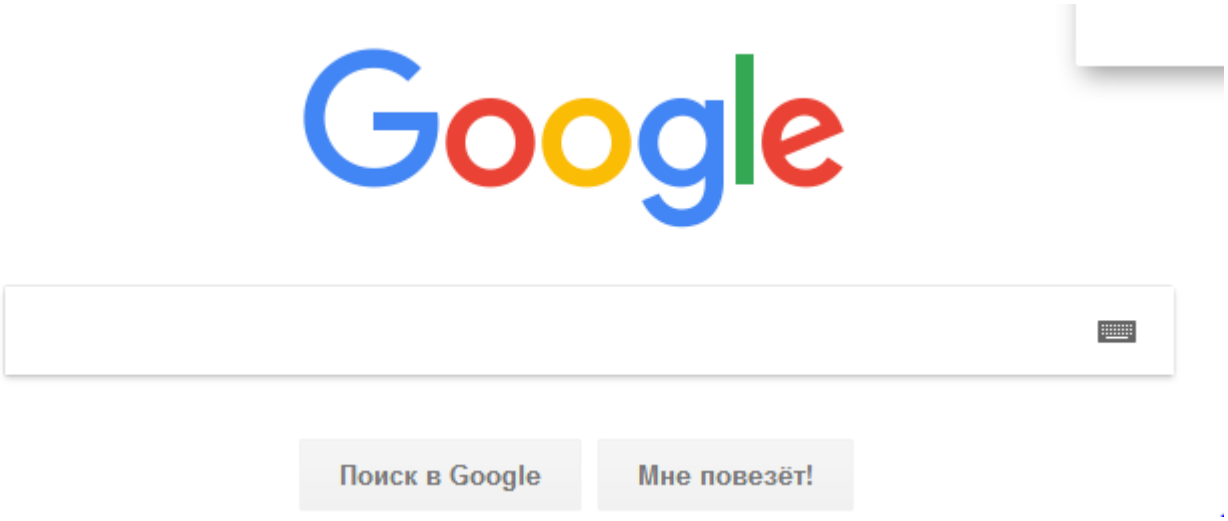

Рисунок 1.1 – Интерфейс поисковой строкии Google Search

1.1.3 Поисковая машина

Поисковой машиной называется часть поисковой системы, реализующая взаимодействие всех остальных его модулей. Поисковая машина отвечает за

создание и дополнение поискового индекса, его чтение и поиск с задействованием алгоритмов ранжирования.

1.1.4 Ранжирование

Под ранжированием понимается процесс нахождения ранга документов и упорядочивание их в список соответственно их рангу.

Рангом документа называется посчитанный специальным алгоритмом рейтинг документа относительно поступившего в поисковую систему запроса. Посчитанный рейтинг показывает, насколько документ соответствует поступившему запросу. При подсчёте рейтинга обычно учитываются следующие критерии:

1) количество вхождений термина из запроса в документ;
2) расстояние между терминами из запроса в документе;
3) нахождение термина из запроса в теле или в заголовке документа;
4) наличие к тексте документа терминов, обладающих синонимической связью с исходным запросом;

и т.д.

1.2 Оценка качества работы поисковой системы

Оценка качества работы каких-либо систем является основополагающим фактором на пути к достижению положительных результатов при разработке этих систем. К сожалению, на данный момент не существует оптимального критерия для общей оценки поисковой системы. В связи с этим различными людьми придумываются концептуально различные способы оценки качества поисковой системы. Рассмотрим некоторые подходы к оценке качества [1].

1.2.1 Удовлетворенность пользователей поисковой системой

Удовлетворенность пользователей поисковика является важным аспектом качества поиска.

Удовлетворенность пользователей оценивается через такие метрики как «число сессий в неделю», «число визитов в месяц» и другие метрики, связанные с активностью и интересом пользователя к поисковой системе. Исходные сырые необработанные данные – публичные счетчики, такие как например liveinternet или top.mail.ru.

Преимущества.

Удовлетворенность учитывает многие важные аспекты качества поиска, такие как персонализация, актуальность и свежесть. Метрика удовлетворенности может зарегистрировать вред от избыточной рекламы, с учетом способности / неспособности пользователей отличать рекламу от выдачи.

Недостатки.

К недостаткам оценки по удовлетворенности можно отнести зависимость от сезонности (день недели, время года), праздников и смещения в пользователях.

Если удовлетворенность оценивается через данные liveinternet, то есть зависимость от среднего присутствия счетчика в результатах поиска.

Скрытая активность на компьютерах пользователей в виде специальный программ также может зашумлять метрику удовлетворенности.

Если качество поиска резко выросло или упало, то удовлетворенность меняется не сразу и не резко, а постепенно, в течение нескольких месяцев.

Часть недостатков метрики можно устранить: коррекция на месяц года, переход на отношение метрик, коррекция на присутствие счетчика, помесячные наблюдения для смягчения эффектов праздников. На графике ниже (рисунок 1.2), данные корректировки уже сделаны.

К сожалению рост или падение удовлетворенности не отвечает на вопрос, в чем была причина изменения удовлетворенности [2].

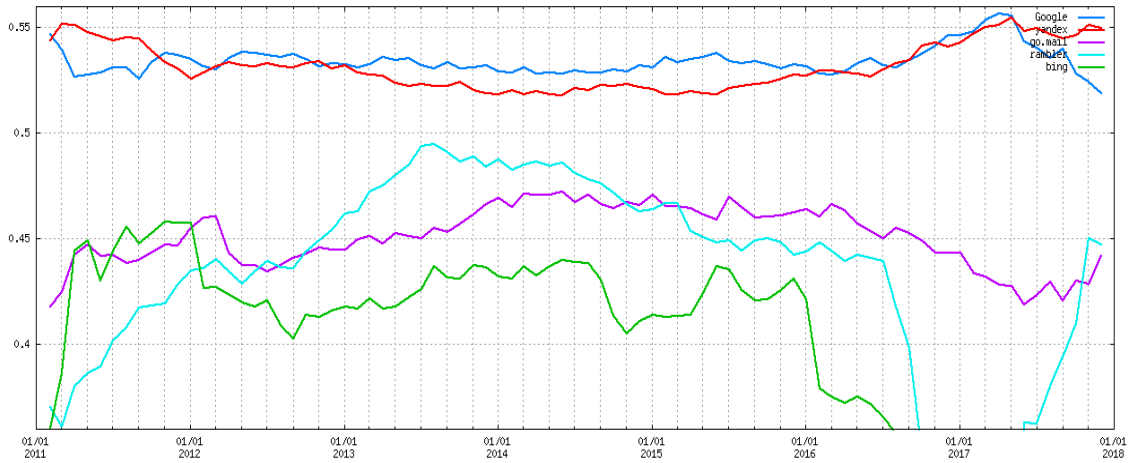

Рисунок 1.2 – Качество популярных поисковых систем по критерию удовлетворенности пользователя

1.2.2 Скорость полнота и точность

Поисковые системы постоянно оценивают качество результатов поиска и вносят необходимые поправки в алгоритм. Так, в Google идут активные разработки персонализированного поиска, основанного на данных о предпочтениях пользователя. Перед сменой своих алгоритмов ранжирования Яндекс и Рамблер активно используют динамическую выдачу, когда разным пользователям по одинаковым запросам показываются разные результаты поиска для оценки популярности элементов.

Среди основных критериев оценки выдачи выделяются:

1) скорость поиска – как быстро пользователь получает ответ на свой вопрос, поскольку время ожидания значительно влияет на лояльность пользователей;
2) полнота ответа – все ли ответы представлены, поскольку часть запросов имеет более одного значения, а другие запросы направлены на получение нетекстовой информации;
3) точность ответа – полностью ли отвечают на вопрос пользователя документы, присутствующие в результатах поиска [3].

1.2.3 Контент

Для данного подхода к оценке необходимо определить специальные тесты, сгенерировать определенные входные запросы и на основе ответов поисковой системы проводить качественный анализ. Можно, например, определить следующие виды тестов:

1) навигационный поиск – сравнивается способность поисковиков находить известные сайты;
2) тематический – сравнивается способность поисковиков формировать выдачу, близкую к ручной экспертной подборке ссылок;
3) подсказки – сравнивается способность поисковиков замечать ошибки при наборе запроса и подсказывать пользователям правильные варианты;
4) опечатки – сравнивается способность поисковиков не реагировать на явные опечатки при наборе запроса и выдавать результаты как для запросов без опечаток;
5) цитаты – сравнивается способность поисковиков находить источники известных цитат;
6) оригиналы – сравнивается способность поисковиков находить первоисточники;
7) синонимы – сравнивается способность поисковиков правильно распознавать одинаковые по смыслу запросы с разными формулировками.
8) спам – сравнивается способность поисковиков удалять спам из результатов поиска;
9) SEO-прессинг (монотематичности выдачи) - сравнивается способность поисковиков противостоять попыткам оптимизаторов в коммерчески значимых тематиках превратить выдачу в набор однотипных рекламных ссылок;
10) цензура – сравнивается способность поисковиков фильтровать ссылки на нежелательный контент, по запросам, не относящимся к этой тематике;

11) полнота – сравнивается способность поисковиков отвечать на редкие запросы.

Каждый из этих тестов условно проводится на 100 входных, специальных запросах, для которых ожидаются определенные попадания. В качестве численной оценки для каждого теста можно посчитать процент попадания с ожидаемые сайты. Общая оценка сложится из среднего результата между оценками за каждый тест [4].

1.3 Методы улучшения работы поисковых систем

Поисковые системы могут использоваться для разных целей. Направление развития качества работы поисковой системы необходимо определять, исходя из потребностей пользователей поисковой системы. Выделим несколько методик, с помощью которых можно повысить качество работы поисковых систем.

1.3.1 Индивидуальная настройка параметров поисковой системы

Индивидуальный подход применяется при конфигурации каждого блока поисковой системы.

1) При индексации текст токенизируется и проходит через ряд фильтров (морфология, синонимы и др.). Оригинальное слово при этом может быть приведено к нормальной форме, заменено, или вообще удалено. Какие бы фильтры не использовались, необходимо сохранять оригинальное слово в той же позиции, что и замененное/измененное. Пример того, что может произойти, если не сохранить оригинальное слово, показано на «рисунке 1.3»:

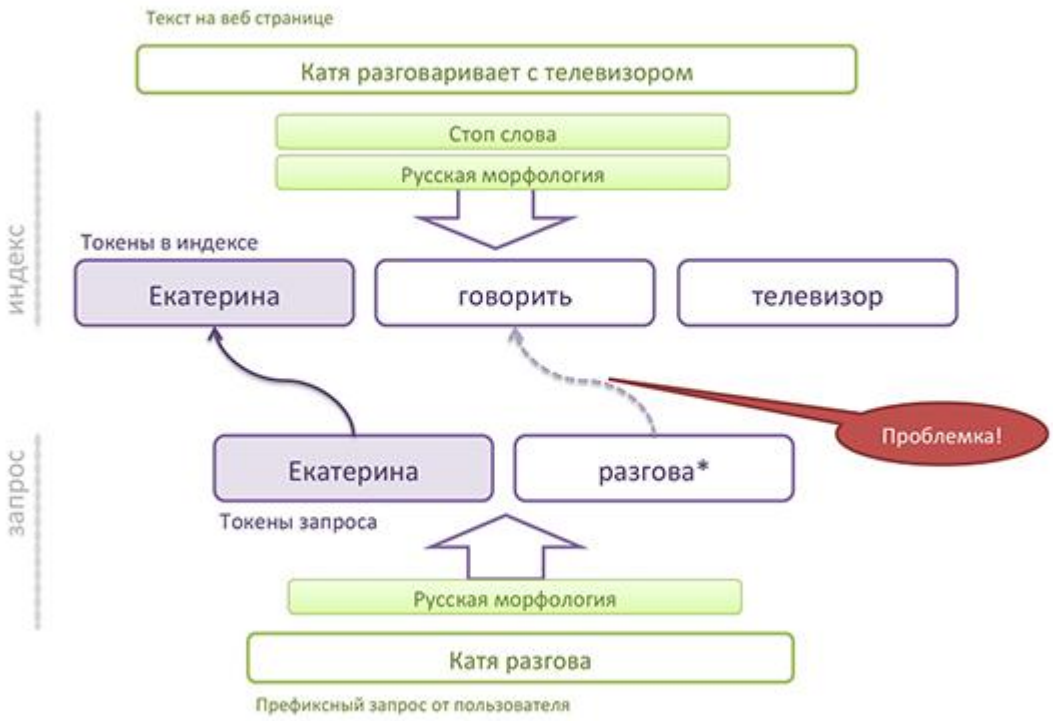

Рисунок 1.3 – Пример несоответсвия токена и нормализованного слова

2) Нельзя использовать анализаторы, которые «чистят» запрос, например, фильтр стоп-слов, удаляющий междометия и частицы. Пример использования подобного анализатора приведен на рисунке 1.4.

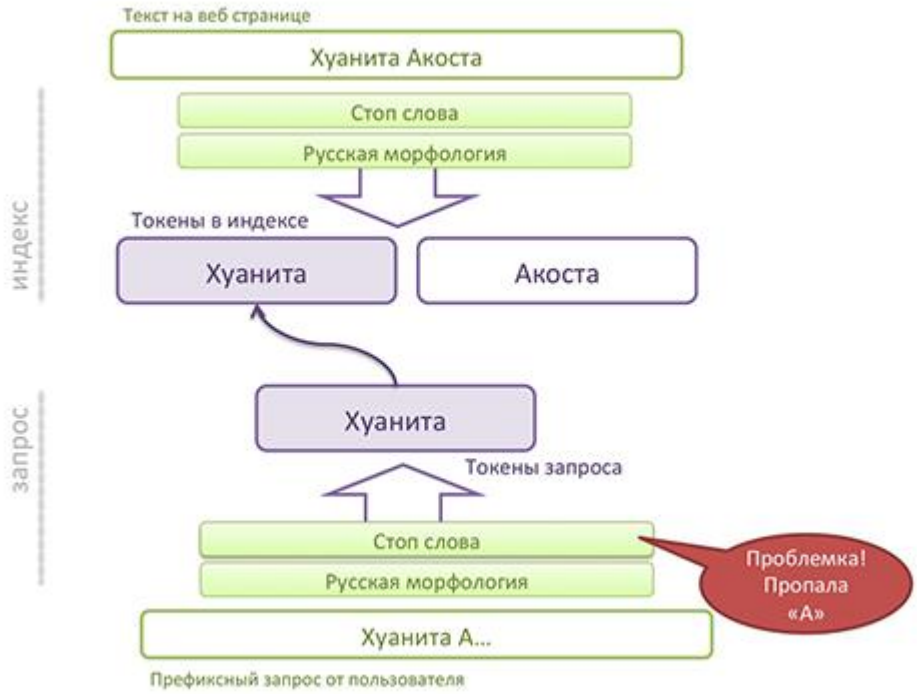

Рисунок 1.4 – Пример ситуации, в которой из запроса удалены короткие термы

3) Как показано на рисунке 1.4, «А» - это так называемое «стоп-слово», анализатор запроса его удалил. Это обозначает, что произойдет «скачок», то есть человек создал запрос «Хуанита А» и увидит документ по запросу «Хуанита» (без А). При этом, если в выданном результате много документов, то нет никакой гарантии, что нужный будет в начале списка. Чтобы этого избежать, убираем фильтр стоп-слов из запроса (рисунок 1.5).

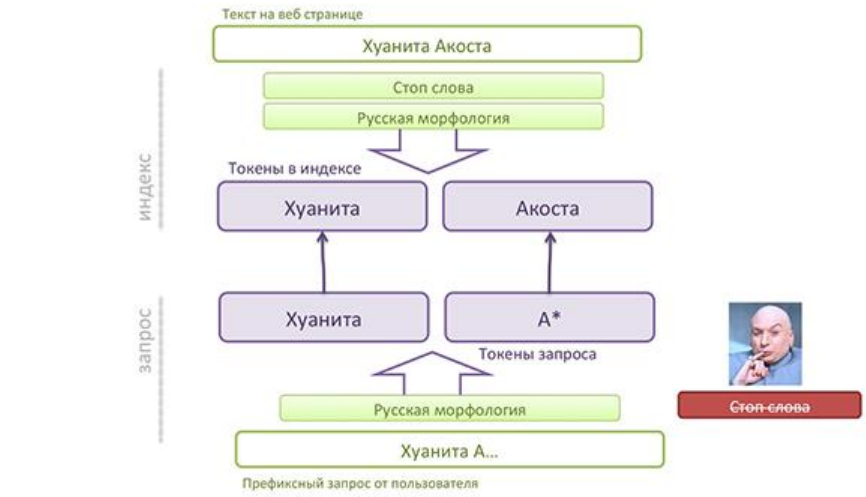

Рисунок 1.5 – Поисковая машина, из которой исключен фильтр коротких слов

4) Не стоит пытаться решить задачу сразу, использовав один запутанный гигантский запрос. Необходимо разделить задачу на подзадачи, после чего использовать dismax-запросы и булевы запросы, с помощью которых соединить результат работы нескольких запросов в один. Для увеличения «веса» точного совпадения в заголовке статьи по сравнению с совпадением в теле статьи можно так же использовать boost-фактор.
5) Поисковый запрос составляется из множества небольших запросов с различными boost-факторами. Boost-фактор — это множитель релевантности, он применяется после того как подсчитана базовая текстовая релевантность.
6) Далее применяется поиск префиксного совпадения. Например, точные совпадения слова «как» будут выведены первым запросом, после чего происходит поиск слова «кактусы». Эти результаты будут выведены нище чем точные совпадения за счет меньшего «веса». Здесь так же используется параметр slop, который определяет, что слова в фразе могут отстоять друг от друга на несколько позиций в любую сторону, при этом по умолчанию префиксный поиск выполняется только «вперед».

Сначала идет поиск точного совпадения фразы в заголовке статьи. При точном совпадении префикс не используется. Например, запрос «КАК» выведет в начале списка статьи с заголовком «Как правильно…», а не «Выращиваем кактус». Документы, найденные по такому запросу, получают boost.

7) Совпадения в заголовке были найдены и получили «буст». После этого происходит переход к телу статьи. В первую очередь происходит поиск точного совпадения, и с меньшим boost-фактором – префиксных совпадений.

   В результате поиска, совпадения в заголовке будут отображены в начале списка, а совпадения в теле статьи — ниже. Необходимо учесть, что стандартными алгоритмами учитывается длина текста в котором найдено совпадение и чем короче текст, тем выше релевантность. Практика показывает, что при префиксном поиске в теле статьи будет найдено много совпадений. Например, если набрать 2-3 буквы, в теле статьи будет уже 200-300 совпадений, и, несмотря на длину текста, они превысят релевантность совпадений в заголовке. Именно поэтому используется boost-фактор и сортировка не отдается на откуп текстовой релевантности.

8) Последним (за счет «буста») идет нечеткий поиск. Нечеткий поиск позволяет «простить» некоторые опечатки в поисковом запросе, введенном пользователем. Внутри запроса используется расстояние Левенштейна. Нечеткий запрос имеет несколько настроек, которые выступают в качестве параметров к алгоритму, также есть возможность указать минимальное необходимое совпадение префикса [5].

1.3.2 Внедрение улучшенных алгоритмов ранжирования

1.3.2.1 Google PageRank

Алгоритм использует специализированный параметр *значимости* страниц для расчета её ранга. Значимость определяется суммой отношений рангов ссылающихся страниц к общему числу ссылок на ссылающихся страницах. Теоретически алгоритм рассчитывает вероятность того, что человек, случайно переходящий по ссылкам, доберется до некоторой страницы. Чем больше ссылок ведет на данную страницу с других популярных страниц, тем выше вероятность, что экспериментатор чисто случайно наткнется на нее. Разумеется, если пользователь будет щелкать по ссылкам бесконечно долго, то в конце концов он посетит каждую страницу, но большинство людей в какой-то момент останавливаются. Чтобы учесть это, в алгоритм PageRank введен коэффициент затухания 0,85, означающий, что пользователь продолжит переходить по ссылкам, имеющимся на текущей странице, с вероятностью 0,85. Например, существует 4 страницы A, B, C, D. Если для страниц B, C, D известны ранги – 0.5, 0,7 и 0.2 соответственно, B имеет 4 ссылки, из них одна на A, C имеет 5 ссылки, из них одна на A, D имеет единственную ссылку на A, то для A можно посчитать значимость как — — — (рисунок 1.6).

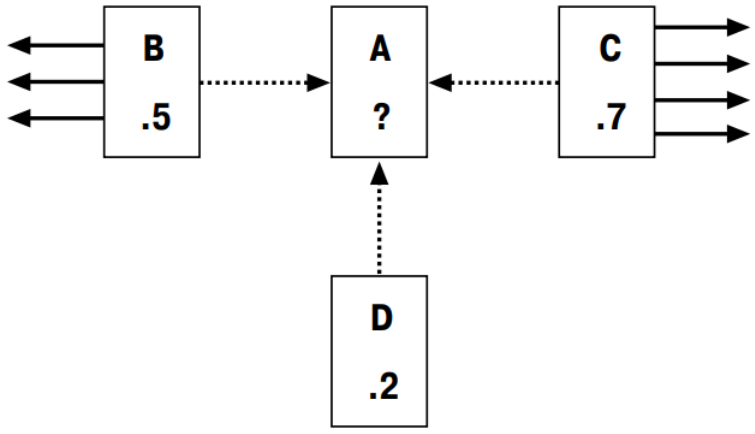

Рисунок 1.6 – Пример вычисления ранга страницы A

В данном примере для всех страниц, ссылающихся на A, уже вычислен ранг. Но невозможно вычислить ранг страницы, пока неизвестны ранги ссылающихся на нее страниц, а эти ранги можно вычислить, только зная ранги страницы, которые ссылаются на них. Для вычисления нулевых рангов необходимо всем страницам произвольный начальный ранг и провести несколько итераций. После каждой итерации ранг каждой страницы будет все ближе к истинному значению PageRank [6].

1.3.2.2 Google Penguin, Google Panda

Такие алгоритмы проводят тестирование содержимого сайта на качественность его контента, производя оценку на наличие различных триггеров. Google Panda например отслеживает следующие триггеры:

– слабое соответствие контента и описания сайта;

– наличие дубликатов контента;

– доверенность источника;

– наличие слишком разностороннего контента;

– низкокачественный текст, сгенерированный пользователями;

– наличие большого количества рекламы;

– и др.

Для увеличения значимости сайтов (PageRank), организовываются спам-ссылки, «невидимый» текст (человек на сайте не видит, а поисковые машины индексируют это содержимое), недостоверное описание с популярными тегами и др. Чтобы избежать недостоверной оценки значимости страницы в Google был придуман алгоритм Penguin.

Алгоритмы Panda и Penguin анализируют содержимое сайтов и понижают ранг некачественных [7].

1.3.3 Тематическое моделирование

На текущий момент разработано и реализовано достаточно большое количество тематических моделей. Рассмотрим наиболее распространенные и основополагающие из них [8, 20].

1.3.3.1 Латентный семантический анализ

Латентный семантический анализ (ЛСА) — это метод обработки информации, представленной на естественном языке, основанный на анализе взаимосвязей между коллекцией документов и встречающимися в ней терминами, в котором некоторым факторам (тематикам) сопоставляются все документы и термины.

В основе этого метода лежит выявление латентных связей изучаемых объектов или явлений.

ЛСА был разработан и запатентован Скоттом Дирвестером, Сьюзен Дюмаи, Георгом Фарнасом, Ричардом Харшманом, Томасом Ландауэром, Карэн Лохбаум и Лин Стритер в 1988. Также этот метод называют латентно-семантическим индексированием (ЛСИ).

Изначально ЛСА применялся для автоматического индексирования текстов, выявления их смысловой структуры и получения псевдо-документов. Затем применение этого метода успешно расширили до использования его в представлениях баз знаний и построения с его помощью когнитивных моделей.

В последнее время основным применением ЛСА стал поиск информации, классификация документов, построение моделей понимания, а также другие области, в которых требуется выявление основных качеств из большого массива информации.

Устройство.

ЛСА часто сравнивают с простой нейронной сетью, состоящей из трёх слоёв:

1) слой, содержащий множество слов (термов);
2) слой, содержащий множество документов;

3) слой, в котором содержатся весовые коэффициенты, связывающие первый и второй слои. Обычно этот слой является скрытым.

Для обучения ЛСА использует матрицу термы-на-документы, в которой описывается исходный набор данных. Как правило, в элементах этой матрицы хранятся веса, характеризующие частоты использования каждого терма в каждом документе и участие трема во всех документах. В основном в ЛСА используется разложение диагональной матрицы по сингулярным значениям (SVD – Singular Value Decomposition). С помощью данного разложения удаётся представить исходную матрицу как множество ортогональных матриц, линейная комбинация которых достаточно точно отображает исходную.

Применение.

С помощью ЛСА в общем случае решаются три основных задачи:

1) сравнение двух термов;
2) сравнение документа с термом;
3) сравнение документов между собой.

К достоинствам метода можно отнести следующие качества:

1) ЛСА является наилучшим из разработанных методов для выявления латентных зависимостей внутри и между документами;
12) используются так называемые матрицы «близости», составленные на основе частотных характеристик документов и лексических единиц.

К недостаткам ЛСА можно отнести:

1) при увеличении объёма исходных данных наблюдается существенное снижение скорости работы алгоритма;
13) вероятностная модель метода основана на нормальном распределении, что не совсем соответствует реальности.

1.3.3.2 Вероятностный латентно-семантический анализ

Вероятностный латентно-семантический анализ (ВЛСА), или вероятностное латентно-семантическое индексирование (ВЛСИ) – статистический метод анализа

соответствий двух типов данных. Этот метод является развитием и логическим продолжением описанного выше метода ЛСА и применяется в поиске информации, обработке данных на естественных языках и т.д. Автором данного метода является Томас Хоффман.

В отличие от ЛСА, который основан на линейной алгебре и различных способах снижения размерности матрицы, ВЛСА основывается на смешанном разложении, берущем начало из модели скрытых классов. Использование данного подхода гораздо лучше обосновано статистически.

Пусть – множество (коллекция) текстовых документов, – множество (словарь) всех употребляемых в них терминов (слов или словосочетаний). Каждый документ представляет собой последовательность терминов из словаря . Термин может повторяться в документе много раз.

Пусть существует конечное множество тем , и каждое употребление термина в каждом документе связано с некоторой темой , которая не известна. Формально тема определяется как дискретное (мультиномиальное) вероятностное распределение в пространстве слов заданного словаря .

Введем дискретное вероятностное пространство . Тогда коллекция документов может быть рассмотрена как множество троек , выбранных случайно и независимо из дискретного распределения . При этом документы и термины являются наблюдаемыми переменными, тема является латентной (скрытой) переменной.

Требуется найти распределения терминов в темах для всех тем и распределения тем в документах для всех документов . С учетом гипотезы условной независимости по формуле полной вероятности получаем вероятностную модель порождения документа :

При этом делается ряд допущений аналогичный допущениям латентно-семантического анализа.

Существуют следующие расширения ВЛСА:

1) иерархические, которые в свою очередь делятся на:
2) асимметричный или полиномиальный асимметричный иерархический анализ;
3) симметричное или иерархический вероятностный латентно-семантический анализ;
4) генеративные модели;
5) скрытое распределение Дирихле;
6) данные высшего порядка, т.е. применение ВЛСА для трёх и более переменных.

1.3.3.3 Латентное распределение Дирихле

Метод латентного размещения Дирихле (latent Dirichlet allocation, LDA) предложен Дэвидом Блеем в 2003 году. В этом методе устранены основные недостатки PLSA.

Метод LDA основан на той же вероятностной модели

при дополнительных предположениях:

1) векторы документов порождаются одним и тем же вероятностным распределением на нормированных |T| - мерных векторах; это распределение удобно взять из параметрического семейства распределений Дирихле ;

2) векторы тем порождаются одним и тем же вероятностным распределением на нормированных векторах размерности |W|; это распределение удобно взять из параметрического семейства распределений Дирихле .

1.4 Вычисление релевантности

Каждая поисковая система имеет в своей основе алгоритмы расчёта рейтинга документа в соответствии с полученным поисковым запросом. Ярким примером такого алгоритмов является Lucene TF-IDF Similarity, основанный на Okapi BM-25.

1.4.1 Lucene TF-IDF Similarity

Lucene TF-IDF Similarity определяет компоненты ранжирования Lucene [13].

Ниже описывается, как подсчет Lucene эволюционирует от базовых моделей поиска информации до эффективной реализации. Сначала мы кратко расскажем о VSM Score, затем проиллюстрируем его концептуальную формулу оценки Lucene, из которой, наконец, развивается практическая функция оценки Lucene (последняя связана непосредственно с классами и методами Lucene).

Lucene объединяет булевую модель (BM) информационного поиска с векторной космической моделью (VSM) информационного поиска - документы, одобренные BM, оцениваются VSM.

В VSM документы и запросы представлены в виде взвешенных векторов в многомерном пространстве, где каждый отдельный индексный индекс является измерением, а веса - значениями Tf-idf.

VSM не требует, чтобы весы были значениями Tf-idf, но считается, что значения Tf-idf дают результаты поиска высокого качества, поэтому Lucene использует Tf-idf. Tf и Idf описаны более подробно ниже, но пока, для завершения, просто скажем, что для заданного термина t и документа (или запроса) x Tf (t, x) изменяется с количеством вхождений термина t в x (когда один увеличивается, а другой - другой), а idf (t) также изменяется с обратным к числу индексных документов, содержащих термин t.

VSM-оценка документа d для запроса q - это сходство косинусов векторов взвешенных запросов V (q) и V (d) (рисунок 1.7):

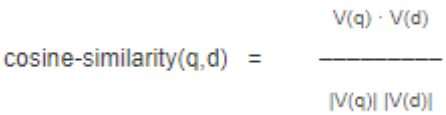

Рисунок 1.7 – Косинусальная близость запроса и документа

где V (q) · V (d) - точечное произведение взвешенных векторов, а | V (q) | и | V (d) | являются их евклидовыми нормами.

Примечание: приведенное выше уравнение можно рассматривать как точечное произведение нормированных взвешенных векторов в том смысле, что деление V (q) на его евклидову норму нормализует его на единичный вектор.

Lucene уточняет оценку VSM как для качества поиска, так и для удобства использования.

Известно, что нормализация V(d) на единичном векторе является проблематичной, поскольку она удаляет всю информацию о длине документа. Для некоторых документов удаление этой информации возможно нормально, например, документ, сделанный путем дублирования определенного пункта 10 раз, особенно если этот пункт составлен из отдельных терминов. Но для документа, который не содержит дублированных абзацев, это может быть неправильно. Чтобы избежать этой проблемы, используется другой коэффициент нормализации длины документа, который нормализуется к вектору, равному или большему, чем единичный вектор: doc-len-norm (d).

При индексировании пользователи могут указать, что определенные документы важнее других, назначая повышение документа. Для этого оценка каждого документа также умножается на его значение boost doc-boost (d).

Lucene - это полевая база, поэтому каждый термин запроса применяется к одному полю, нормализация длины документа - по длине определенного поля, и в дополнение к увеличению документа также увеличиваются поля документа.

Одно и то же поле может быть добавлено в документ во время индексации несколько раз, и поэтому повышение этого поля - это умножение повышений

отдельных дополнений (или частей) этого поля в документе.

Во время поиска пользователи могут указывать boosts для каждого запроса, подзапроса и каждого термина запроса, поэтому вклад термина запроса в партитуру документа умножается на увеличение этого запроса query-boost (q).

Документ может соответствовать многопользовательскому запросу, не содержащему все условия этого запроса (это верно для некоторых запросов), и пользователи могут дополнительно вознаграждать документы, соответствующие более количеству запросов, посредством коэффициента координации, который обычно больше, когда больше условий согласованный: координатный фактор (q, d).

В упрощенном предположении о единственном поле в индексе мы получаем формулу Lucene «Концептуальная оценка» (рисунок 1.8):

$$score(q,d) = coord\text{-}factor(q,d) \cdot query\text{-}boost(q) \cdot \frac{V(q) \cdot V(d)}{|V(q)|} \cdot doc\text{-}len\text{-}norm(d) \cdot doc\text{-}boost(d)$$

Рисунок 1.8 – Концептуальный вид функции подсчёта рейтинга Lucene

Концептуальная формула – это упрощение в том смысле, что термины и документы выставляются, и повышение обычно выполняется для каждого запроса, а не для запроса.

Теперь мы опишем, как Lucene реализует эту концептуальную формулу оценки и извлекает из нее практическую функцию подсчета Lucene.

Для эффективного расчета баллов некоторые скоринговые компоненты вычисляются и агрегируются заранее:

1) текст запроса (фактически для каждого термина запроса) известен при запуске поиска.
2) Евклидова норма запроса |V(q)| может быть вычислена при запуске поиска, поскольку он не зависит от введённого термина. С точки зрения оптимизации поиска, это правильный вопрос, зачем вообще

нормализовать запрос, потому что все введённые термины будут умножаться на те же |V(q)|, и, следовательно, ранжирование документов (их порядок по баллам) не будет затронут по этой нормировке.

Есть две веские причины для сохранения этой нормализации.

Напомним, что сходство Косинуса можно использовать, чтобы найти, как похожи два документа. Можно использовать Lucene для, например, кластеризации и использовать документ в качестве запроса для вычисления его сходства с другими документами. В этом случае важно, чтобы оценка документа d3 для запроса d1 была сопоставима с оценкой документа d3 для запроса d2. Другими словами, количество документов для двух разных запросов должно быть сопоставимым. Для этого могут потребоваться другие приложения. И это именно то, что нормализует вектор запроса V (q), обеспечивает: сопоставимость (в определенной степени) двух или более запросов.

Применение нормализации запросов в баллах помогает сохранить оценки вокруг единичного вектора, что предотвращает потерю данных оценки из-за ограничений точности с плавающей запятой.

Стандарт длины документа doc-len-norm (d) и ускорение документа doc-boost (d) известны во время индексации. Они вычисляются заранее, и их умножение сохраняется как одно значение в индексе: norm (d). (В приведенных ниже уравнениях норма (t в d) означает норму (поле (t) в doc d), где поле (t) - поле, связанное с термом t).

Практическая функция подсчета Lucene вытекает из вышесказанного. Цветовые коды демонстрируют, как они соотносятся с концептуальной формулой (рисунок 1.9):

$$score(q,d) = coord(q,d) \cdot queryNorm(q) \cdot \sum_{t \text{ in } q} \left(tf(t \text{ in } d) \cdot idf(t)^2 \cdot t.getBoost() \cdot norm(t,d) \right)$$

Рисунок 1.9 – Применяемая на практике функция подсчёта рейтинга Lucene

где

1) tf (t in d) коррелирует с частотой этого члена, определяемой как количество раз, которое t появляется в текущем набранном документе d. Документы, которые имеют больше случаев определенного срока, получают более высокий балл. Заметим, что tf (t в q) считается равным 1, и поэтому оно не фигурирует в этом уравнении. Однако, если запрос содержит в два раза один и тот же термин, будут два терм-запроса с этим же термином, и, следовательно, вычисление будет по-прежнему быть правильным (хотя и не очень эффективным). Вычисление по умолчанию для tf (t в d) в DefaultSimilarity:

2) idf (t) означает обратную частоту документа. Это значение коррелирует с обратным значением docFreq (количество документов, в которых появляется термин t). Это означает, что более редкие термины дают более высокий вклад в общий балл. idf (t) появляется для t как в запросе, так и в документе, поэтому он равен квадрату в уравнении. Вычисление по умолчанию для idf (t) в DefaultSimilarity:

3) Координата (q, d) - это коэффициент оценки, основанный на том, сколько из условий запроса найдено в указанном документе. Как правило, документ, содержащий больше условий запроса, получит более высокий балл, чем другой документ с меньшим количеством запросов. Это коэффициент времени поиска, вычисляемый в координатах (q, d) по сходству, действующему во время поиска.
4) queryNorm (q) - нормализующий фактор, используемый для сопоставления оценок между запросами. Этот фактор не влияет на ранжирование документа (поскольку все ранжированные документы умножаются на один и тот же фактор), а скорее просто попытки сопоставить оценки из разных запросов (или даже разных индексов). Это коэффициент времени

поиска, рассчитанный по сходству, действующему во время поиска. Вычисление по умолчанию в DefaultSimilarity дает евклидову норму:

$$\frac{}{\sqrt{}}$$

5) t.getBoost () - это увеличение времени поиска термина t в запросе q, как указано в тексте запроса (см. синтаксис запроса), или как установлено вызовами приложения для setBoost (). Обратите внимание, что на самом деле нет прямого API для доступа к увеличению одного термина в многопользовательском запросе, но несколько терминов представлены в запросе как несколько объектов TermQuery, и поэтому повышение термина в запросе доступно путем вызова подзапрос getBoost ().
6) norm (t, d) инкапсулирует несколько факторов (индексации времени) и факторы длины:
    a. Улучшение поля - задается вызовом поля.setBoost () перед добавлением поля в документ.
    b. lengthNorm - вычисляется, когда документ добавляется к индексу в соответствии с количеством токенов этого поля в документе, так что короткие поля вносят больший вклад в счет. LengthNorm вычисляется классом подобия, действующим при индексировании.

Метод computeNorm (org.apache.lucene.index.FieldInvertState) отвечает за объединение всех этих факторов в один поплавок.

Когда документ добавляется к индексу, все вышеупомянутые факторы умножаются. Если в документе есть несколько полей с тем же именем, все их повышения умножаются вместе:

Однако приведенное значение нормы кодируется как один байт перед

сохранением. Во время поиска значение стандартного байта считывается из индексной директории и декодируется обратно до значения поплавка. Это кодирование / декодирование при уменьшении размера индекса происходит с ценой потери точности - не гарантируется, что декодирование (encode (x)) = x. Например, декодировать (кодировать (0.89)) = 0.75.

Сжатие значений нормы в один байт сохраняет память во время поиска, потому что, как только поле ссылается на время поиска, его нормы - для всех документов – сохраняются в памяти.

Обоснование поддержки такого сжатия с потерями значений нормы заключается в том, что, учитывая трудности (и неточность) пользователей, чтобы выразить свою истинную потребность в информации по запросу, имеют значение только большие различия.

Наконец, обратите внимание, что во время поиска будет слишком поздно, чтобы изменить эту стандартную часть оценки, например, используя другое сходство для поиска.

1.5 Программное обеспечение, реализующее или использующее тематические модели

1.5.1 WEKA

WEKA (Waikato Environment for Knowledge Analysis) – свободное программное обеспечение, которое используется для анализа данных. Написано на языке Java в университете Уайкато (Новая Зеландия), распространяется по лицензии GNU.

WEKA является набором средств визуализации и алгоритмов, использующимся для интеллектуального анализа данных и решения задач прогнозирования, вместе с графической пользовательской оболочкой для доступа к нему.

WEKA используется для выполнения различных задач, таких как анализ данных, подготовка данных (preprocessing), отбор признаков (англ. *feature*

selection), кластеризация, классификация, регрессионный анализ и визуализация результатов [15].

1.5.2 MALLET

MALLET (MAchine Learning for LanguagE Toolkit) – это разработанный на Java пакет для обработки естественного языка, классификации документов, кластеризации, тематического моделирования, извлечения информации и других приложений машинного обучения на основе текста [16].

MALLET включает усовершенствованные средства для классификации документов: эффективные процедуры для извлечения «особенностей» текста, широкий спектр алгоритмов и код для оценки производительности классификатора на основе нескольких широко распространённых метрик.

В дополнение к классификации, MALLET включает средства для разметки приложений, таких как поимённое извлечение сущностей из текста. Включены алгоритмы: скрытых Марковских моделей, Марковских моделей максимальной энтропии и Марковских случайных полей. Эти методы реализованы как расширяемая система для конечных преобразователей.

Тематические модели удобны для анализа больших коллекций непомеченных текстов. Набор для тематического моделирования MALLET включает в себя реализации Латентного Размещения Дирихле, Распределения Пачинко и Иерархического Латентного Размещения Дирихле.

Многие алгоритмы в пакете MALLET опираются на численную оптимизацию. MALLET включает эффективную реализацию алгоритма ограниченной памяти Бройдена-Флетчера-Голдфарба-Шанно (limited memory BFGS), а также множество других оптимизационных методов.

В дополнение к усовершенствованным приложениям для машинного обучения, пакет MALLET включает в себя процедуры для преобразования текстовых документов в их численные представления для большей эффективности обработки. Данный процесс реализован с помощью гибкой системы «трубок»,

через каждую из которых производятся различные действия, такие как маркировка строк, удаления стоп-слов и преобразования предложений в вектора весов.

Пакет MALLET представляет собой библиотеку, предназначенную для Data Mining и является подключаемым пакетом для java, в следствие чего здесь не приведены никакие изображения, иллюстрирующие данное ПО.

1.5.3 Gensim

Gensim – это подключаемая библиотека, разработанная для языка для языка python, которая начиналась в 2008 году, как набор скриптов на языке python, написанных для Чешской Библиотеки Цифровой математики. На текущий момент Gensim является одним из наиболее робастных, эффективных и безпроблемных программных обеспечений для реализации не требующих контроля семантических моделей для необработанного текста. Он расположился где-то между немасштабируемыми и нестабильными самодельными приложениями с одной стороны и робастными java-подобными алгоритмами, которым требуется вечность только лишь для запуска “Hello, World!” – с другой.

Gensim – мультиплатформенный, масштабируемый open-source пакет, обладающий рядом эффективных усовершенствований для реализованных в нём алгоритмов [17].

Так как Gensim не является отдельным приложением, здесь не приводятся никакие изображения, которые могли бы послужить иллюстрацией к процессу работы с ним.

1.5.4 Infer.NET

Infer.Net – фреймворк, который включает в себя реализацию многих распространённых алгоритмов обработки информации. Среди реализованных алгоритмов: разрешенные байесовские сети, факторный анализ, алгоритмы нахождения решений, а также тематические модели, включая Латентное Распределение Дирихле.

Infer.Net был разработан группой машинного обучения и восприятия в Кембриджском Исследовательском отделе Майкрософт.

Infer.Net мультиплатформенный, т.е. может быть как надстройка к множеству популярных языков программирования [18].

1.6 Программное обеспечение, реализующее или использующее поисковые системы

1.6.1 Apache Lucene

Apache LuceneTM – высокопроизводительная полнофункциональная текстовая поисковая библиотека, полностью написанная на Java. Это технология, подходящая практически для любого приложения, которое требует полнотекстового поиска, особенно межплатформенного.

Apache Lucene – это проект с открытым исходным кодом, доступный для бесплатной загрузки. Пожалуйста, используйте ссылки справа для доступа к Lucene.

Особенности LuceneTM:

1) масштабируемая высокопроизводительная индексация:
    a. более 150 ГБ / час на современном оборудовании;
    b. Требования к небольшой ОЗУ - всего 1 Мб;
2) инкрементное индексирование так же быстро, как пакетная индексация
    a. размер индекса примерно 20-30% размер проиндексированного текста;
3) мощные, точные и эффективные алгоритмы поиска:
    a. ранжированный поиск - лучшие результаты возвращены;
    b. много мощных типов запросов: фразовые запросы, подстановочные запросы, запросы близости, запросы диапазона и многое другое (например, название, автор, содержимое);
    c. сортировка по любому полю;

d. поиск по нескольким индексам с объединенными результатами;
e. позволяет одновременно обновлять и выполнять поиск;
f. гибкая огранка, выделение, объединение и группировка результатов;
g. быстрые, эффективные по памяти и типично-терпимые предложения;
h. сменные модели ранжирования, включая векторную космическую модель и Okapi BM25;
i. настраиваемый механизм хранения (кодеки);

4) межплатформенное решение:
   a. доступно как программное обеспечение с открытым исходным кодом под лицензией Apache, которое позволяет использовать Lucene как в коммерческих, так и в программах с открытым исходным кодом;
   b. 100% -ная Java-версия;
   c. реализации других доступных языков программирования, которые совместимы с индексами, созданными Lucene на любом языке программирования.

Apache Software Foundation обеспечивает поддержку сообщества Apache проектов с открытым исходным кодом. Проекты Apache определяются совместными процессами на основе консенсуса, открытой, прагматичной лицензией на программное обеспечение и желанием создать высококачественное программное обеспечение, которое лидирует в своей области. Apache Lucene, Apache Solr, Apache PyLucene, Apache Open Relevance Project и их соответствующие логотипы являются товарными знаками The Apache Software Foundation. Все остальные упоминаемые знаки могут быть товарными знаками или зарегистрированными товарными знаками соответствующих владельцев [9].

1.6.2 Sphinx

Sphinx (англ. SQL Phrase Index) – система полнотекстового поиска,

разработанная Андреем Аксёновым и распространяемая по лицензии GNU GPL. Отличительной особенностью является высокая скорость индексации и поиска, а также интеграция с существующими СУБД (MySQL, PostgreSQL) и API для распространённых языков веб-программирования (официально поддерживаются PHP, Python, Java; существуют реализованные сообществом API для Perl, Ruby,.NET[1] и C++).

Основные возможности:

1) высокая скорость индексации (до 10-15 МБ/сек на каждое процессорное ядро);
2) высокая скорость поиска (до 150—250 запросов в секунду на каждое процессорное ядро с 1 000 000 документов);
3) высокая масштабируемость (крупнейший известный кластер индексирует до 3 000 000 000 документов и поддерживает более 50 миллионов запросов в день);
4) поддержка распределенного поиска;
5) поддержка нескольких полей полнотекстового поиска в документе (до 32 по умолчанию);
6) поддержка нескольких дополнительных атрибутов для каждого документа (то есть группы, временные метки и т. д.);
7) поддержка стоп-слов;
8) поддержка однобайтовых кодировок и UTF-8;
9) поддержка морфологического поиска — имеются встроенные модули для английского, русского и чешского языков; доступны модули для французского, испанского, португальского, итальянского, румынского, немецкого, голландского, шведского, норвежского, датского, финского, венгерского языков;
10) нативная поддержка PostgreSQL и MySQL;
11) ноддержка ODBC совместимых баз данных (MS SQL, Oracle и т. д.) [10].

1.6.3 Microsoft Windows 10

Windows 10 – это операционная система для персональных компьютеров, разработанная и выпущенная Microsoft как часть семейства операционных систем Windows NT. Она была выпущена 29 июля 2015 года. Это первая версия Windows, которая постоянно получает свежие обновления своего функционала. Устройства в корпоративных средах могут получать эти обновления медленнее или использовать долгосрочные этапы поддержки, которые получают критические обновления, такие как исправления безопасности.

Windows 10 использует современные поисковые технологии. Поисковик встроен в меню «Пуск» и позволяет искать любые объекты, находящиеся в пределах видимости операционной системы. Поиск может вестись по названиям файлов, их содержимому, типам и многому другому [11] (рисунок 1.10).

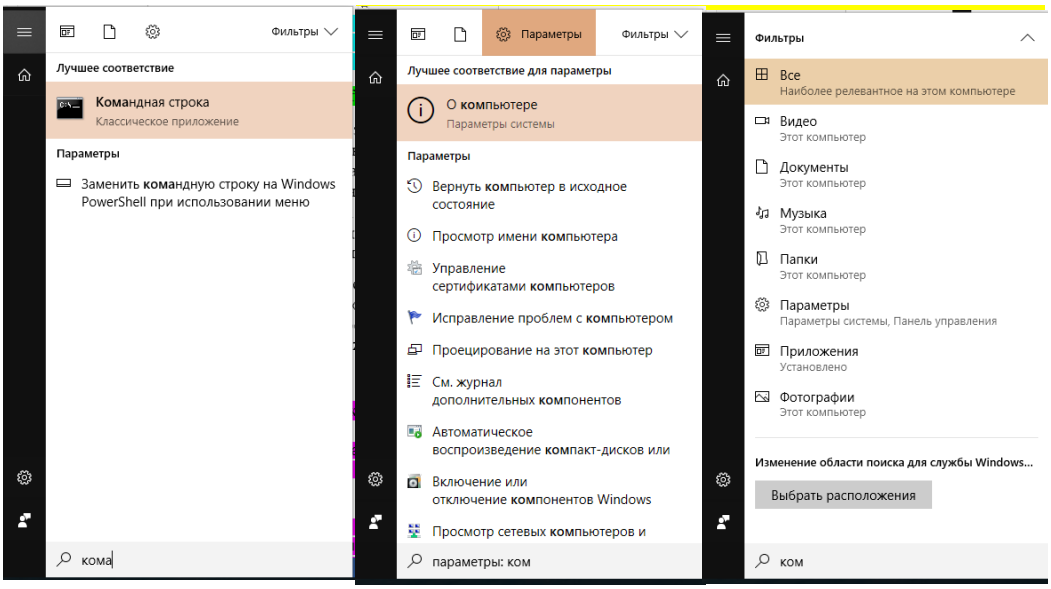

Рисунок 1.10 – Изображения интерфейса поиска в меню «Пуск» ОС Windows 10

Выводы по разделу

В данном разделе был произведён обзор научных литературных источников, связанных с поисковыми системами и смежными областями, проведено исследование существующих способов оценки и улучшения качества работы

поисковых систем, а также выполнен обзор программного обеспечения, использующего поисковые и смежные с ними технологии, либо предоставляющие возможности для их реализации.

2 РАЗРАБОТКА МАТЕМАТИЧЕСКОЙ МОДЕЛИ И АЛГОРИТМОВ ДЛЯ УЛУЧШЕНИЯ И ОЦЕНКИ КАЧЕСТВА РАБОТЫ ЛОКАЛЬНОЙ ПОИСКОВОЙ СИСТЕМЫ

2.1 Постановка задачи

Целью ВКР является разработка и исследование алгоритмов повышения релевантности результатов поиска с учетом синонимической близости запросов и предпочтений пользователей локальной поисковой системы.

Для достижения поставленной цели необходимо решить следующие задачи:

1) на основе проведённого анализа выбрать способ повышения качества поиска локальной поисковой системы;
2) разработать формальное представление модели поисковой системы;
3) разработать математическую модель синонимов на основе данных, предоставляемых обученной тематической моделью;
4) разработать алгоритм взаимодействия тематической модели и поисковой системы со встроенной искусственной нейронной сетью;
5) разработать алгоритм модификации словаря и документов коллекции с целью учёта при поиске слов, отсутствующих в словаре поисковой системы;
6) реализовать и отладить разработанные алгоритмы, внедрить их в поисковую систему;
7) произвести оценку результатов внедрения разработанных алгоритмов в поисковую систему.

2.2 Математические модели

2.2.1 Вероятностная модель коллекции документов

Пусть D – коллекция текстовых документов, W – словарь всех употребляемых

терминов (слов или их сочетаний). Каждый документ - последовательность терминов (словаря *W*. Один термин может встречаться не единожды в одном и том же документе.

2.2.2 Вероятностное пространство

Предположим, что *T* – конечное множество тем, причём каждый встреченный термин *w* в любом документе *d* связано с некоторой темой . Коллекция документов рассматривается как множество троек , которые выбираются случайным образом из дискретного распределения независимо друг от друга. Указанное распределение задаётся на конечном множестве . Наблюдаемыми переменными являются термины и документы , тогда как тема считается скрытой (латентной) переменной.

2.2.3 Гипотеза о независимости элементов выборки

Предположим, что порядок терминов в документе не является значительным фактором для определения темы, тогда тему документа можно определить, использовав не сам текст, а так называемый «мешок слов», в который помещены все термин документа в произвольном или специально упорядоченном порядке.

Также в эту гипотезу входит понятие «мешка документов», которое эквивалентно «мешку слов», но применяется ко всей исследуемой коллекции документов.

На основе гипотезы «мешка слов» можно построить такое представление документа в виде подмножества , где каждому элементу сопоставляется некое число , равное числу вхождений термина *w* в документ *d*.

2.2.4 Гипотеза об условной независимости

Предположим, что появление в документе d слов, которые имеют отношение к теме t, можно описать общим для коллекции распределением , которое не

зависит от конкретного документа.

Это предположение можно описать тремя эквивалентными представлениями:

(2.1)

2.2.5 Вероятностная модель порождения данных

Исходя из определения условной вероятности, а также формулы полной вероятности и гипотезы условной независимости

(2.2)

В случае выполнения условия известности распределений и , вероятностная модель описывает порождение коллекции D.

Обратной данной задачей является построение тематической модели. Когда по известной коллекции D нужно получить распределения на основе которых она была построена.

2.2.6 Гипотеза о разреженности

Логично предположим, что каждый документ и каждый термин имеет связь с некоторым количеством тем. В этом случае значительное число вероятностей и принимает нулевое значение.

В случае принадлежности документа к множеству различных тематик такой документ стоит разбить на некоторое число отдельных фрагментов, каждый из которых относится к меньшему возможному числу тем.

Если же подобное верно для какого-либо термина, высока вероятность того, что данное слово является общеупотребительным и не имеет особой ценности для построения модели.

2.2.7 Частотные оценки вероятностей

Вероятности, относящиеся к исследуемым переменным d и w, будем оценивать по выборке как частоты (обозначение выборочных вероятностей).

—, —, —, —, (2.3)

- число вхождений термина w в документ d;

- длина документа d в терминах;

- число вхождений термина w во все документы коллекции;

- длина коллекции в терминах.

Если коллекцию документов рассмотреть в виде , то вероятности, относящиеся к скрытой пременнной t, тоже можно рассматривать в качестве частот.

—, —, —, —, (2.4)

- число троек документа d, связанного с темой t, в которых есть термин w;

- число троек, в которых некий термин документа d связан с тематикой t;

- число троек, в которых термин w связан с темой t;

- общее число троек, связанных с темой t;

Если перейти к пределу при , частотные характеристики из формул (2.3-2.4) устремляются к соответствующим вероятностям, основываясь на законе больших чисел, однако частотная интерпретация даёт лучшее понимание условных вероятностей, что пригодится в дальнейшем построении модели

2.2.8 Латентное размещение Дирихле

Метод латентного размещения Дирихле (latent Dirichlet allocation, LDA) предложен Дэвидом Блеем в 2003 году. В этом методе устранены основные недостатки PLSA.

Метод LDA основан на той же вероятностной модели

при дополнительных предположениях:

1) векторы документов порождаются одним и тем же вероятностным распределением на нормированных |T| - мерных векторах; это распределение удобно взять из параметрического семейства распределений Дирихле ;

2) векторы тем порождаются одним и тем же вероятностным распределением на нормированных векторах размерности |W|; это распределение удобно взять из параметрического семейства распределений Дирихле .

2.2.9 Расстояние Кульбака-Лейблера

В силу гипотез о «мешке слов» и «мешке документов», действующих в рамках тематических моделей, использовалась формула расчёта дивергенции Кульбака-Лейблера для дискретного случая:

— , (2.5)

где

Также стоит учитывать крайнюю важность предварительной обработки документов в коллекции посредством того же стеммера, который используется поисковой системой для обработки терминов, входящих в поисковый запрос пользователя.

2.2.10 Поисковая система

В рамках описания поисковой системы обозначим:

-поисковый запрос,

значение функции подсчёта рейтинга документа в соответствии с

запросом

В качестве результата поиска обозначим:

(2.6)

число документов в области поиска,

номер -го документа в рейтинге .

Множество документов, оцененных экспертом, как релевантные запросу , обозначим как , .

2.2.11 Собираемая поисковой системой статистика

В дальнейшем нам потребуется математически сформулированное описание статистики, которую собирает поисковая система о происходящих поисковых сессиях.

Одна запись в статистике поисковой системы представляет собой пару:

(2.7)

где

заголовок записи,

поступивший в поисковую систему запрос,

время начала поисковой сессии,

идентификатор пользователя,обратившегося к поисковой системе

тело записи,

строка в теле записи,

документ-результат из области поиска,

номер документа в текущем ранжированном списке результатов,

время выбора документа пользователем

значение функции подсчёта рейтинга документа

в соответствии с запросом

Целиком множество статистики определим так:

(2.8)

2.2.12 Контекстные синонимы

Исходя из того, что поисковый запрос состоит из терминов , можно определить тематическое распределение поискового запроса как сумму тематических распределений входящих в него терминов, поделенную на количество терминов в запросе. Покажем это.

Пусть оператор расширения вектора запроса

(2.9)

где

тогда

—— тематическое распределение запроса.

Таки образом, в силу того, что представляет собой вектор условных вероятностей, то есть, фактически, вероятностное тематическое распределение, близость двух запросов можно определить через близость их тематических распределений. Воспользуемся расстоянием Кульбака-Лейблера для определения критерия.

Будем говорить, что запросы и обладают синонимической связью, если

(2.10)

где

пороговое значение, определяемое экспериментально, в зависимости от требуемой степени близости распределений.

2.2.13 Множество пар (запрос, его тематическое распределение) для уникальных запросов из обучающей выборки ИНС

Определим обучающую выборку ИНС:

(2.11)

где

-поисковый запрос из j-й записи в статистике,

множество документов, выбранных пользователем в

поисковую сессию, соответствующую -й записи в статистике.

Тогда множество уникальных запросов из обучающей выборки ИНС определим, как:

(2.12)

Следовательно, множество пар запросов и их тематических распределений обозначим:

(2.13)

2.2.14 Оператор ИНС

Оператор ИНС определим как вектор вероятности соответствия документа поисковому запросу :

(2.14)

2.2.15 Пороговый оператор для результатов ИНС

Пороговый оператор для результатов ИНС определим как

, (2.16)

где

Также обозначим оператор как вектор сумм пороговых корректировок:

(2.17)

2.2.15 Lucene TF-IDF Similarity

Используемая Lucene функция подсчёта рейтинга документа относительно запроса имеет следующий вид:

(2.18)

где

количество раз, которое термин встречается в документе

обратная частота документа, то есть значение, обратно пропорциональное тому, в скольких документах коллекции встречается термин

коэффициент, связанный с числом терминов содержащихся в документе

нормирующий коэффициент

это значение множителя рейтинга для термина которое отдельно указывается в поисковом запросе,

сборный модификатор, обобщающий ряд значимых параметров, таких как длина поля, в котором найден термин в документе значение его множителя и др.

2.2.16 Модификация значения функции подсчёта рейтинга Lucene TF-IDF Similarity на основе результатов совместной работы Нейронной Сети и Тематической Модели

Обозначим оператор модификации значения функции подсчёта:

——— (2.19)

где

Тогда итоговый рейтинг релевантности документа запросу определим:

——— (2.20)

Тогда итоговое множество отранжированных документов принимает вид:

(2.21)

2.2.17 Критерии оценки релевантности поиска на основе собранной поисковой системой статистики

2.2.17.1 Метрика релевантности по позиции

Обозначим релевантность поиска как:

(2.22)

Далее, чтобы избежать ошибочных оценок, обозначим:

(2.23)

где

количество запросов q за промежуток времени

Тогда среднюю релевантность поиска обозначим:

(2.24)

где

количество поисковых запросов, поступивших в поисковую систему за промежуток времени t,

номер запроса в собранной поисковой системой статистике.

2.2.17.2 Метрика релевантности по времени

Критерий оценки качества работы поисковой системы, основанный на времени поступления запроса в поисковую систему зададим следующим образом:

(2.25)

количество поисковых сессий за время ,

время выбора ,

время поступления запроса в систему.

2.3 Алгоритмы

2.3.1 Обучение модели

2.3.1.1 Данные для обучения модели

В качестве коллекции документов D для обучения модели используются тексты документов из области поиска поисковой системы. В рамках рассматриваемой локальной поисковой системы документов.

В качестве множества слов W используются все термины, входящие в эти документы. В рамках рассматриваемой локальной поисковой системы терминов.

Количество тем T задаётся произвольным образом, исходя из количества документов в коллекции, а также требуемой точности определения наличия или отсутствия синонимической связи. В данном случае количество тем .

2.3.2 Расширение словаря «незнакомыми» терминами и дообучение Тематической Модели

Определим множество терминов, которые были включены в поисковые запросы, но отсутствуют в словаре Тематической Модели:

(2.26)

Также определим множество записей из статистики поисковой системы, в которых в результате поиска выбирается документ

(2.27)

Также определим множество записей из статистики поисковой системы, в которых документ выбирается как релевантный запросу , который включает в себя термин :

(2.28)

Определим число вхождений термина в документ как .

Также определим «Мешок Слов» документа как множество пар из термина и числа вхождений термина в документ , нормированного длиной документа :

(2.29)

Также определим дополнение к «Мешку Слов» документа как множество пар из термина и числа сопоставлений документа термину в статистике поисковой системы, нормированное общим числом упоминаний документа в статистике поисковой системы:

(2.30)

Тогда алгоритм расширения словаря коллекции «незнакомыми» терминами, расширения его «Мешка Документов» и дообучения модели с целью обнаружения скрытой связи между «незнакомыми» терминами и документами коллекции принимает вид:

1)
2)
3) Повторно запускаем процесс обучения тематической модели для новых множеств , сохраняя то же самое число тем T.

2.3.3 Взаимодействие поисковой системы, в которую встроена ИНС, с тематической моделью

При поступлении в поисковую систему нового запроса запускается следующая последовательность действий:

1) запрос передаётся модулю синонимизации, где:
   a. вычисляется ;
   b. строится множество синонимических для запроса запросов вида:

(2.31)

где расстояние Кульбака-Лейблера,
определённое экспериментальным путём пороговое значение;

c. строится множество ;
d. множество передаётся модулю преобразования запросов.

2) в модуле преобразования запросов

a. строится вектор ;
b. строится множество ;
c. множество передаётся модулю нейронной сети;

3) в модуле НС:

a. : вычислим ;
b. построим – вектор корректировок для поискового запроса на основе рекомендации ИНС для исходного запроса и запросов, обладающих с ним синонимической связью;
c. G передаётся модулю модификации ранжирования;

4) также в модуле базового ранжирования строим множество и передаём его в модуль модификации ранжирования;

5) в модуле модификации ранжирования:

a. на основе полученного множества и вектора G строим множество ;

6) выводим множество .

2.4 Схемы алгоритмов

2.4.1 Дообучение тематической модели

Схема алгоритма имеет следующий вид (рисунок 2.1):

Рисунок 2.1 – Схема алгоритма модификации словаря и коллекции документов с переобучением Тематической Модели

2.4.2 Взаимодействие поисковой системы со встроенной ИНС с тематической моделью

Схема алгоритма имеет следующий вид (рисунок 2.2):

Рисунок 3.2 – Схема алгоритма взаимодействия поисковой системы с тематической моделью

Выводы по разделу

В данном разделе были выполнены следующие работы:

1) на основе проведённого анализа выбран способ повышения качества поиска локальной поисковой системы;
2) разработано формальное представление модели поисковой системы, собираемой ей статистики, прочих, необходимы для описания алгоритмов, элементов;
3) разработана математическая модель синонимов на основе данных, предоставляемых обученной тематической моделью;
4) разработан алгоритм взаимодействия тематической модели и поисковой системы со встроенной искусственной нейронной сетью;
5) разработан алгоритм модификации словаря и документов коллекции с целью учёта при поиске слов, отсутствующих в словаре поисковой системы;
6) для разработанных алгоритмов были построены схемы, с целью иллюстрации их работы.
7) все разработанные модели и алгоритмы в дальнейшем были реализованы и внедрены в существующую локальную поисковую систему.

3 РЕАЛИЗАЦИЯ ПОИСКОВОЙ СИСТЕМЫ И ОЦЕНКА ЕЁ КАЧЕСТВА

3.1 Поисковая система

3.1.1 Структура поисковой системы

3.1.1.1 Индекс

Процедура построения и хранения индекса выполняется полностью средствами библиотеки Lucene. Для построения индекса используются следующие пакеты:

1) lucene-analyzers-common-6.2.0.jar – пакет, включающий в себя большое количество стеммеров для различных естественных языков. Из данного пакета был взят класс org.apache.lucene.analysis.ru.RussianAnalyzer, предоставляющий функционал стемминга для русского языка;
2) lucene-core-6.2.0.jar – пакет, включающий в себя все основные модули, реализующие функции поисковой системы. Для индексирования из данного пакета были взяты следующие классы:
    a. org.apache.lucene.document.Document – класс, реализующий внутреннее представление документа для его последующего индексирования;
    b. org.apache.lucene.document.Field и org.apache.lucene.document.FieldType – классы, выполняющие вспомогательную роль для класса Document, а именно отвечающие создание и хранение полей документа и их свойств;
3) классы org.apache.lucene.index. DirectoryReader, org.apache.lucene.index. IndexReader, org.apache.lucene. index.IndexWriter, org.apache.lucene.index. IndexWriterConfig, org.apache.lucene.store .Directory, org.apache.lucene.store .FSDirectory, отвечающие, собственно, за прочтение директории, в которой хранится коллекция документов для индексации а также создание и

конфигурацию индекса в выбранной директории.

Процесс построения индекса обязательно включает в себя этап построения модели документа из коллекции, которую предполагается индексировать. На этом этапе требуется описать средствами API Lucene все поля документов, которые предполагается индексировать, а также разработать метод, который считает документы в соответствующую структуру и после передаст её для построения индекса.

В случае нашей прикладной задачи документы имеют следующую структуру:

<приоритет>; <описание, представляющее из себя набор тегов, разделённых вертикальной чертой>; <зашифрованный ключ>

Поле «приоритет» не требует хранения.

Поле «описание» представляет собой тексты на русском языке с большим количеством аббревиатур, состоящих из русских и латинских букв. Поиск планируется вести именно по данному тексту, соответственно требуется при индексации сохранить исходную форму каждого слова, а также его стемм.

Поле «ключ» представляет собой шифр, который не требуется обрабатывать, только хранить.

В качестве структуры, соответствующей нашему представлению, был разработан класс Item, имеющий конструктор, включающий в себя строковые поля desc и key. При чтении каждого нового документа создаётся экземпляр класса Item, в поля которого помещаются описание и ключ документа.

Также средствами Lucene был создан шаблон документа, включающий в себя соответствующие поля, отвечающие описанным выше требованиям.

После выполнения подготовительной работы коллекция документов была считана, преобразована в документы Lucene по созданному шаблону и подана на вход классу, который построил на её основе индекс. На этом этап построения индекса был закончен.

3.1.1.2 Поисковая строка

С поисковой строкой связано несколько задач, а именно:

1) парсинг поисковой строки на слова;
2) преобразование каждого слова в вид, понятный модулю, который будет выполнять поиск. В данном случае поисковую строку вида «мама мыла раму» следует преобразовать в строку «tags:мам tags:мыл tags:рам». Делается это следующим образом:
    a. строка разбивается на слова;
    b. каждое слово обрабатывается стеммером;
    c. к каждому слову приписывается префикс «tags:», указывающий на то, что искать данную информацию мы будем в поле «tags» докуметов, содержащихся в индексе;
    d. преобразованные слова снова объединяются в общую строку;
3) также задачи нечёткого поиска и поиска в глубину решаются при помощи построения запросов специальным образом. Так, для выполнения «нечёткого» поиска требуется активировать функцию, разрешающую использование специального языка поисковых запросов, встроенного в Lucene. Данный язык включает в себя возможности по группировке запросов, выставление приоритетов для отдельных элементов запроса, явно определение логических операций между словами в запросе, а также задание степени «нечёткости» слова в запросе, которое подразумевает под собой максимальное число несовпадающих в слове из запроса и слове из документа букв.

Определяется это посредством расстояния Левенштейна, а с точки зрения задачи построения поискового запроса выглядит следующим образом: в уже построенном запросе «tags:мам tags:мыл tags:рам» мы дописываем ~<число символов>, например, для слов из трёх букв это будет 1, то есть на выходе получим такой поисковый запрос: «tags:мам~1 tags:мыл~1 tags:рам~1».

Под поиском в глубину понимается такой метод поиска, когда мы хотим найти что-то в уже найденном ранее результате. Для решения этой задачи также используются возможности специального языка поисковых запросов Lucene, а именно – возможность группировать слова в запросе, а также вручную определять логические операции между словами в запросе. Таким образом, для того, чтобы выполнить поиск в результатах предыдущего поиска нам достаточно построить запрос следующим образом:

1) предположим, что у нас есть запрос query1, с результатом поиска res1. Запрос query1 выглядит следующим образом: tags:pan tags:предприят~2 ;
2) в результатах поиска по первому запросу мы хотим найти также слово «владелец», таким образом мы к запросу tags:pan tags:предприят~2 должны пристыковать tags:владел~2 так, чтобы при поиске выводились результаты пересечения множеств результатов поиска как по первому, так и по второму запросу. Следовательно, мы составляем запрос следующим образом:

   (tags:pan tags:предприят~2) AND (tags:владел~2), после чего передаём строку парсеру запросов, который преобразовывает её к следующему виду:

   +(tags:pan tags:предприят~2) + tags:владел~2
3) в дальнейшем, если мы хотим ещё сильнее уточнить результаты поиска, скажем, при помощи слова «изготовитель», нам следует преобразовать его к виду «tags:изготовит~2», после чего построить запрос следующим образом:

((tags:pan tags:предприят~2) AND (tags:владел~2)) AND (tags:изготовит~2) ,

после чего передать строку парсеру запросов. Как итог получим следующий запрос:

+(tags:pan tags:предприят~2) + tags:владел~2 + tags:изготовит~2

Весь функционал построения запросов нечёткого поиска и поиска в глубину реализован программно и конечному пользователю не виден, а доступ

выполняется посредством отдельных кнопок интерфейса.

Все использованные возможности предоставляются классами пакета lucene-queryparser-6.2.0.jar

3.1.1.3 Поисковая машина

Весь механизм поиска полностью реализован пакетом Lucene, достаточно просто использовать API классов org.apache.lucene.search.* .

У этих классов существует большое количество различных методов, предоставляющих возможность выбирать тип ранжирования, количество выводимых результатов, номер результата, с которого следует начать вывод и многое другое.

Процесс поиска заключается в чтении движком Lucene индекса и сравнении слов в нём с поисковой строкой в соответствии с параметрами, которые задаются различными префиксами и суффиксами на этапе построения запроса.

3.1.1.4 Ранжирование результатов поиска

У пакета Lucene есть несколько доступных функций ранжирования результатов, также можно изменить ранжирование путём перегрузки данных функций. Математическая модель функций ранжирования подробно описана в документации к библиотеке Lucene, модификации же к ней на текущий момент не потребовались, следовательно, разработаны не были.

3.1.1.5 Внедрение Тематической Модели в поисковую систему

Общая схема поисковой системы имеет следующий вид (рисунок 3.1):

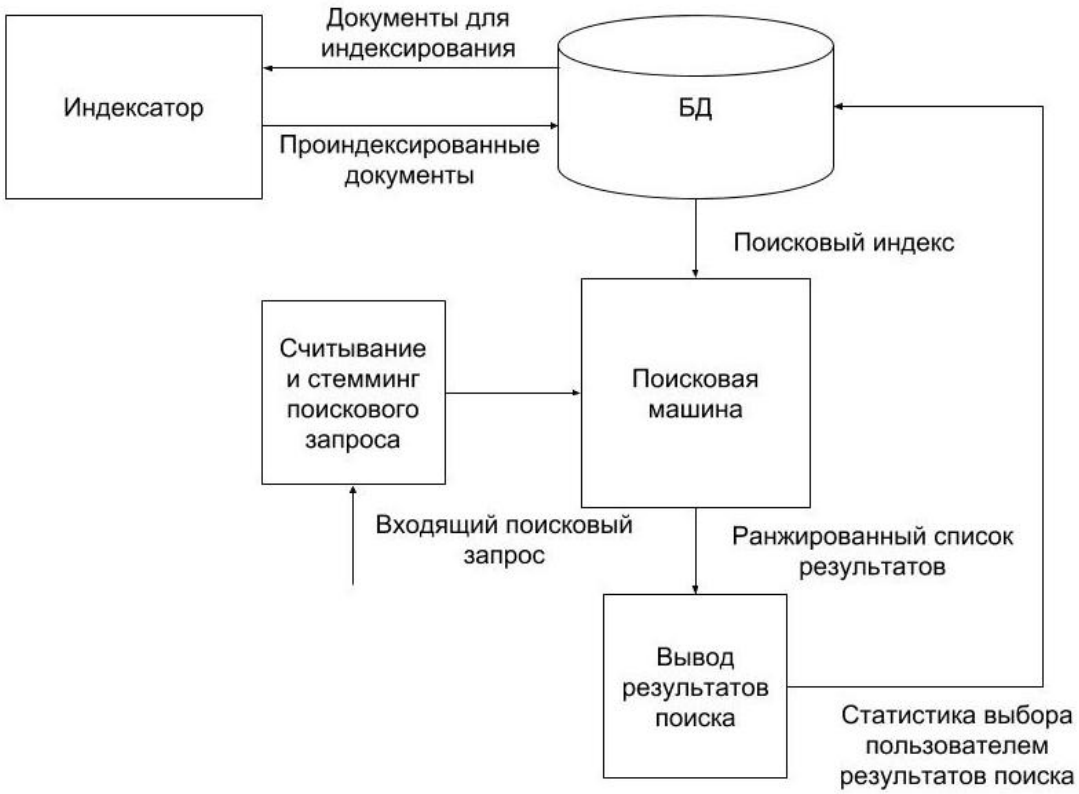

Рисунок 3.1 – Общая схема взаимодействия модулей поисковой системы

После внедрения в поисковую систему искусственной нейронной сети, обученной для рекомендации наиболее часто выбираемых пользователями результатов по схожим запросам, модуль Поисковая Машина принимает следующий вид (рисунок 3.2):

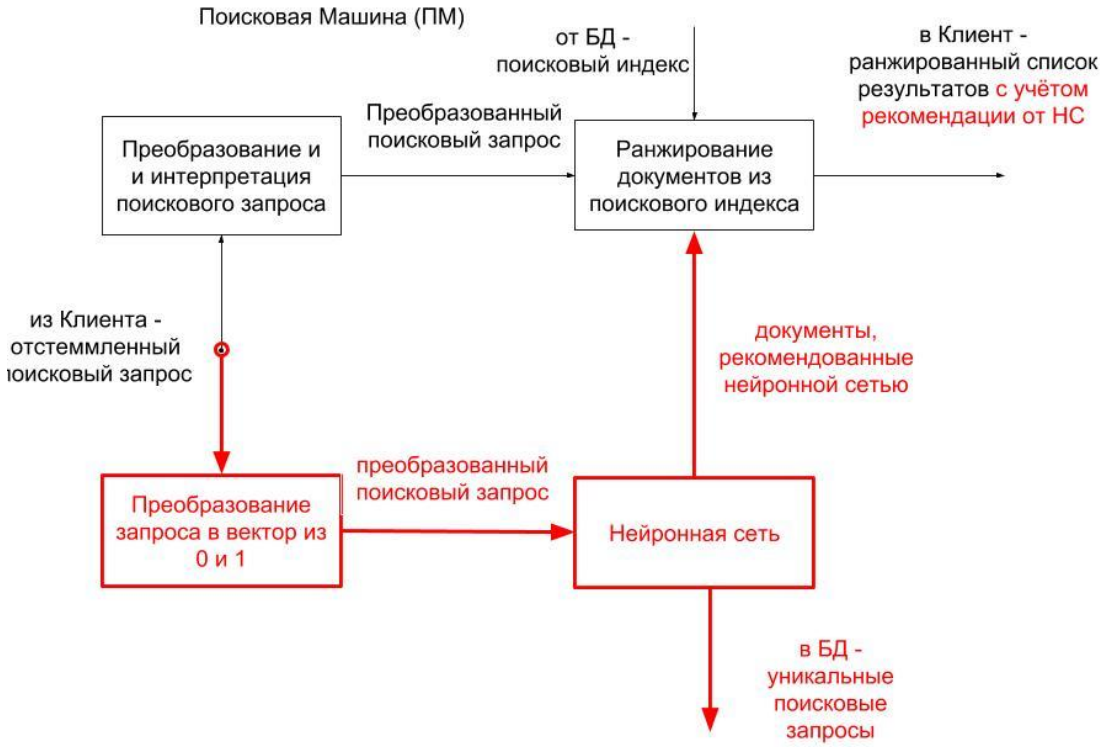

Рисунок 3.2 – Схема взаимодействия подмодулей модуля Поисковая Машина после внедрения искуственной нейронной сети

Также после внедрения модуль База Данных выглядит следующим образом (рисунок 3.3):

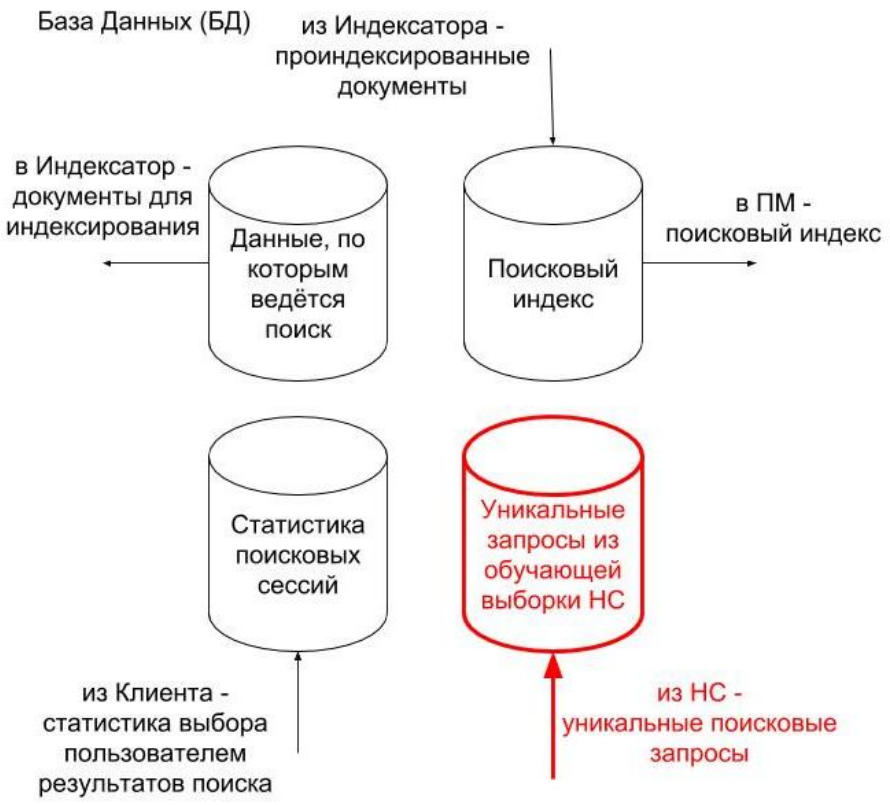

Рисунок 3.3 – Схема взаимодействия с подмодулями модуля База Данных после внедрения в поисковую систему искуственной нейронной сети

Следующим шагом в поисковую систему внедряются элементы, связанные с Тематической моделью. После внедрения Поисковая Машина принимает следующий вид (рисунок 3.4):

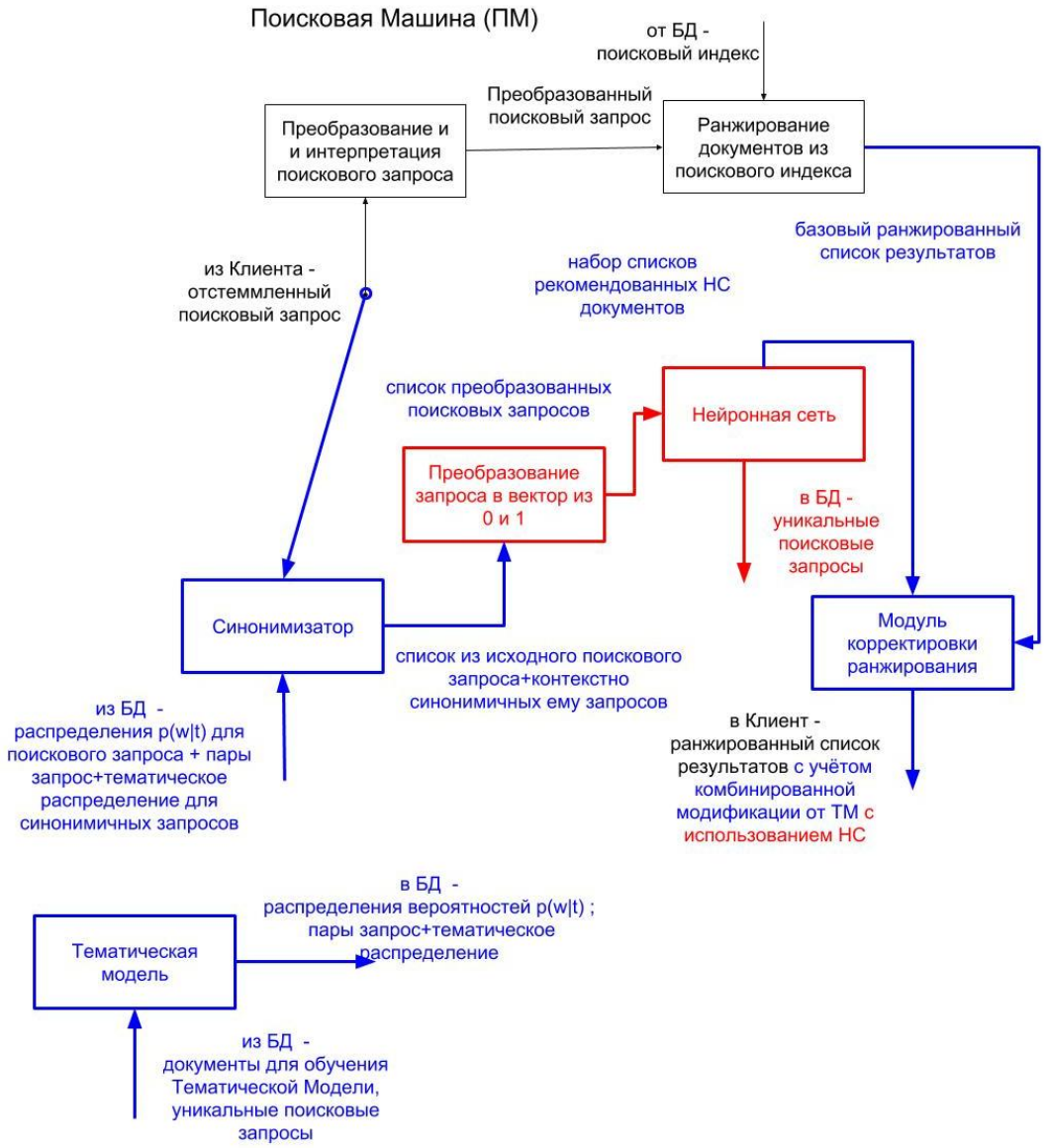

Рисунок 3.4 - Схема взаимодействия подмодулей модуля Поисковая Машина после внедрения тематической модели

Также итоговый вид, к которому приходит модуль База Данных, следующий (рисунок 3.5):

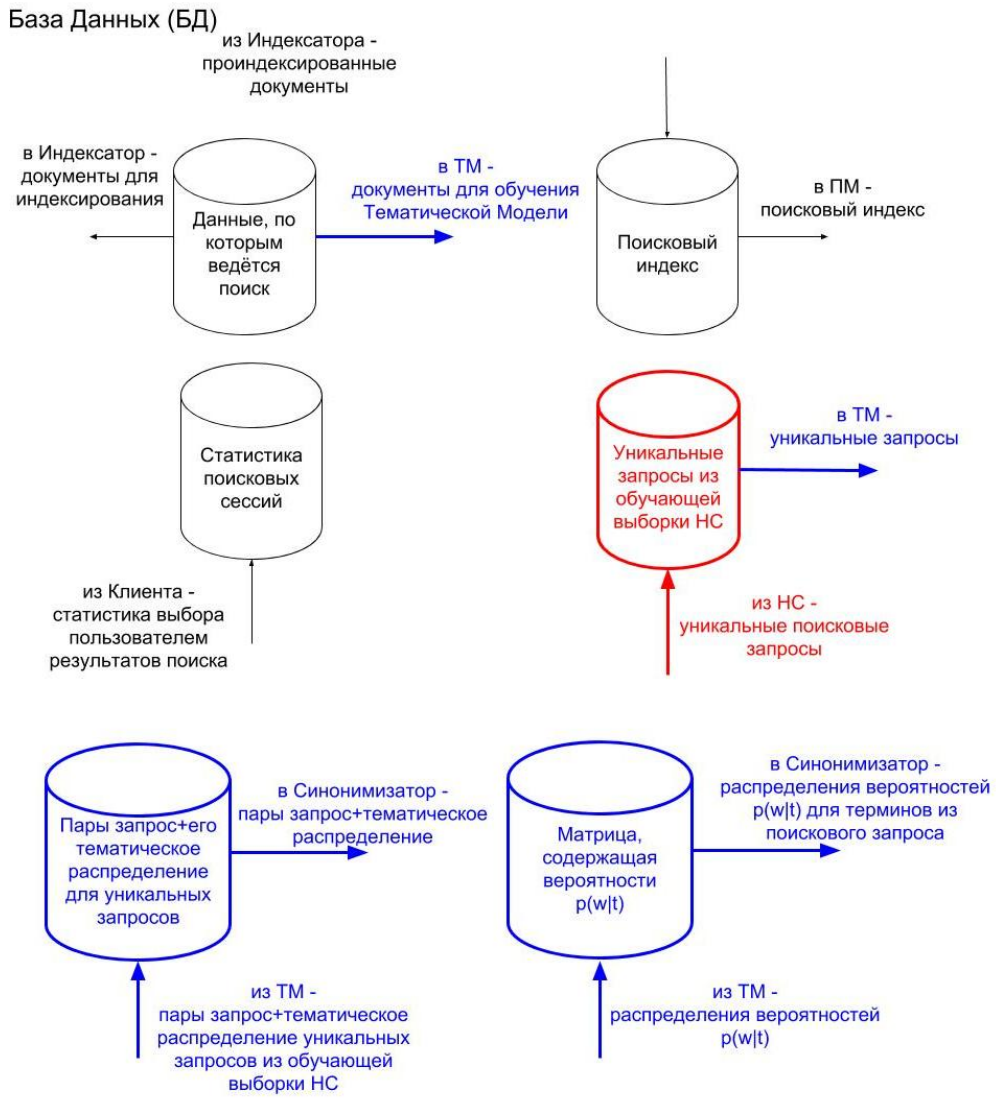

Рисунок 3.5 – Схема взаимодействия с подмодулями модуля База Данных после внедрения в систему тематической модели

3.2 Методика эксперимента

В работающую поисковую систему на длительный промежуток времени внедряется тематическая модель, схема работы поисковой системы изменяется в соответствии с разработанной в разделе 3 схемой.

С целью оценить результаты внесённых в поисковую систему изменений,

собирается статистика.

По истечению заданного промежутка времени собранная статистика анализируется в соответствии с критериями.

3.3 Результаты внедрения тематической модели в общую систему

По итогам эксперимента была составлена следующая таблица.

Таблица 3.1

Оценка результатов модификации локальной поисковой системы

Вид локальной поисковой системы	количество примеров в статистике, поисковых сессий	время использования	качество по позиционному критерию	качество по временному критерию, сек
Базовая локальная поисковая система	26149	первый месяц	0,61	61 сек
Базовая локальная поисковая система	27478	второй месяц	0,6257	57 сек
Базовая локальная поисковая система	28642	третий месяц	0,6156	56 сек
Базовая локальная поисковая система (итого)	82269	3 месяца	0,61	56 сек
Локальная поисковая система с использованием нейронной сети	31238	1 месяц	0,69	52 сек
Локальная поисковая система с	33457	1 месяц	0,78	47 сек

тематической моделью				
Локальная поисковая система после переобучения тематической модели и нейронной сети	35631	1 месяц	0,81	44 сек

Выводы по разделу

В данном разделе была описана разработанная поисковая система, а также показано, к какому виду пришла поисковая система после внедрения в неё всех модификаций, связанных с тематической моделью

Также в данном разделе была определена методика эксперимента по оценке внесённых в поисковую систему изменений.

В соответствии с разработанной методикой эксперимента была произведена оценка изменения качества поиска после внедрения в систему тематической модели.

По результатам произведённой оценки можно сделать вывод о том, что внесённые изменения вывели работу локальной поисковой системы на качественно новый уровень, что говорит о правильности выбранного подхода к улучшению поисковой системы, а также о корректности работы разработанных моделей и алгоритмов.

ЗАКЛЮЧЕНИЕ

В процессе выполнения работы был выполнен обзор литературных источников на следующие темы:

1) общие подходы к построению поисковых систем;
2) распространённые подходы к повышению качества поиска;
3) основные подходы к оценке поисковых систем;
4) программное обеспечение, использующее тематические модели;
5) программное обеспечение, использующее поисковые системы.

В процессе выполнения работы были разработаны следующие математические модели и алгоритмы:

8) формальное представление модели поисковой системы, собираемой ей статистики, прочих, необходимы для описания алгоритмов, элементов;
9) математическая модель синонимов на основе данных, предоставляемых обученной тематической моделью;
10) алгоритм взаимодействия тематической модели и поисковой системы со встроенной искусственной нейронной сетью;
11) алгоритм модификации словаря и документов коллекции с целью учёта при поиске слов, отсутствующих в словаре поисковой системы.

В процессе выполнения работы была произведена разработка и реализация локальной поисковой системы, с последующим внедрением в программы RadixWare и TranzWare компании ООО «Компас Плюс».

В дальнейшем в разработанную поисковую систему была внедрена тематическая модель.

Также были разработаны критерии для оценки качества работы поисковой системы и произведена оценка работы после внедрения в систему тематической модели.

Также в ходе выполнения работы были написаны и опубликованы научные статьи [18, 19].

БИБЛИОГРАФИЧЕСКИЙ СПИСОК

1) Маннинг, К. Введение в информационный поиск / К. Маннинг, П. Рагхаван, Х. Штюнце, Д. Клюшин / Москва, Санкт-Петербург, Киев, изд. дом Вильямс, 2011 - 528 с.
2) Протасов, С. Удовлетворенность пользователей Yandex/ Google/ Go.Mail/ Rambler/Bing (2011-2017) / С. Протасов / URL: http://sz.ru/search-quality/ (Дата обращения: 05.05.2018).
3) Результаты поиска и их оценка / URL: http://www.sembook.ru/ book/poiskovye_sistemy/rezultaty-poiska-i-ikh-otsenka/ (Дата обращения: 05.05.2018).
4) Качество поиска – одной цифрой, размышления о сводном показателе / URL: http://optimization.ru/neiron/021.html (Дата обращения: 05.05.2018).
5) Как это сделано: префиксный поиск / URL: https://habr.com/company /mailru /blog/204654/ (Дата обращения: 08.05.2018).
6) Google PageRank algorithm / https://web.stanford.edu/class/cs54n/handouts/24-GooglePageRankAlgorithm.pdf (Дата обращения: 05.05.2018).
7) Google Panda, Penguin & Hummingbird: Everything You Need to Know /
8) URL : https://www.searchenginejournal.com/seo-guide/panda-penguin-hummingbird/ (Дата обращения: 04.05.2018).
9) Тематическое моделирование / URL: http://www.machinelearning.ru /wiki/index.php?titleТематическое моделирование (Дата обращения: 06.04.2018).
10) Apache Lucene Core / URL: https://lucene.apache.org/core/ (Дата обращения: 10.04.2018).
11) Sphinx overview / URL: http://sphinxsearch.com/about/sphinx/ (Дата обращения: 06.05.2018).
12) Windows 10 / URL: https://www.microsoft.com/ru-ru/windows (Дата

обращения: 03.04.2018).

13) Lucene TF-IDF Similarity / URL: https://lucene.apache.org/core/7_2_0/core/ org/apache/lucene/search/similarities/TFIDFSimilarity.html (Дата обращения: 24.04.2018).

14) Prisma / URL: http://prisma-ai.com/ (Дата обращения: 07.05.2018).

15) Waikato Weka Introduction / URL: http://www.cs.waikato.ac.nz/ml/weka/ (Дата обращения 05.05.2018).

16) Mallet introduction / URL: http://mallet.cs.umass.edu/ (Дата обращения 05.05.2018).

17) An introduction to Gensim / URL: http://radimrehurek.com/gensim/ (Дата обращения: 02.05.2018).

18) Latent Dirichlet Allocation URL: http://research.microsoft.com/ en-us/um/cambridge/projects/ infernet/ docs /Latent%20Dirichlet%20Allocation.aspx (Дата обращения 02.05.2018).

19) Марченко, А. Применение тематического моделирования в решении задачи определения синонимов в локальных поисковых системах / А. Марченко, Д. Масленников, Т. Оленчикова / Естественнонаучный журнал «Точная наука» №26, Кемерово, изд. дом. Плутон, 2018 - С. 8-12.

20) Масленников, Д. Разработка критериев оценки качества работы локальной поисковой системы/ Д. Масленников, А. Марченко, Т. Оленчикова / Естественнонаучный журнал «Точная наука» №26, Кемерово, изд. дом. Плутон, 2018 - С. 84-87.

21) Воронцов, К. Вероятностное тематическое моделирование: обзор моделей и аддитивная регуляризация / К. Воронцов / URL: http://www.machinelearning.ru/wiki/images/d/d5/Voron17survey-artm.pdf (Дата обращения: 20.04.2018).

ПРИЛОЖЕНИЕ 1

Листинг 1.1 – Пакет Searcher

Листинг 1.1.1 – класс Result

```
package Searcher;

import java.util.List;
import org.apache.lucene.search.Query;
import org.apache.lucene.search.TopDocs;
import org.apache.lucene.search.highlight.TextFragment;

//Класс, реализующий две возможные структуры объекта типа Result, необходимого для работы
//поисковика с подержкой режима "подсветки" слов-результатов поиска в фрагментах документов
public class Result {

    public TopDocs hits;
    public Query query;
    public List<List<TextFragment>> fragList;
//  Констурктор без списка фрагметов
    Result(TopDocs hits_in, Query query_in) {
        this.hits = hits_in;
        this.query = query_in;
    }
//  Конструктор со списком фрагментов
    Result(TopDocs hits_in, Query query_in, List<List<TextFragment>> fragList_in) {
        this.hits = hits_in;
        this.query = query_in;
        this.fragList = fragList_in;
    }
}
```

Листинг 1.1.2 – класс Searcher

```
package Searcher;

import org.apache.lucene.analysis.ru.RussianAnalyzer;
import org.apache.lucene.document.*;
import org.apache.lucene.index.*;
import org.apache.lucene.queryparser.classic.ParseException;
import org.apache.lucene.queryparser.classic.QueryParser;
import org.apache.lucene.search.*;
import org.apache.lucene.store.Directory;
import org.apache.lucene.store.FSDirectory;
import org.apache.lucene.search.highlight.Highlighter;
import org.apache.lucene.search.highlight.InvalidTokenOffsetsException;
import org.apache.lucene.search.highlight.QueryScorer;
import org.apache.lucene.search.highlight.SimpleHTMLFormatter;
import org.apache.lucene.search.highlight.TextFragment;
import org.apache.lucene.search.highlight.TokenSources;
```

```
   import java.io.IOException;
   import java.nio.file.Paths;
   import java.util.ArrayList;
   import java.util.List;
   import java.util.Scanner;
   import java.io.File;
   import java.io.FileNotFoundException;
   import java.util.Arrays;
   import org.apache.lucene.analysis.Analyzer;
   import org.apache.lucene.analysis.TokenStream;

   public class Searcher {

   //  Здесь представлен блок методов, отвечающих за создание объекта типа
IndexSearcher, выполняющего поиск по индексу

   //    Метод, создающий объект-чтец индекса
   //    На входе: путь к дирректории хранения индекса
   //    На выходе: объект типа IndexReader
       public static IndexReader createReader(String index_dir) throws
IOException, FileNotFoundException {
           Directory dir = FSDirectory.open(Paths.get(index_dir));
           IndexReader reader = DirectoryReader.open(dir);
           return reader;
       }

   //    Метод для создания поисковика по одиночному индексу
   //    На входе: объект типа IndexReader
   Продолжение листинга 1.2.2
   //    На выходе: объект типа IndexSearcher для поиска по одному индексу
       public static IndexSearcher createSearcher(IndexReader reader) throws
IOException, FileNotFoundException {
           IndexSearcher searcher = new IndexSearcher(reader);
           return searcher;
       }

   //    Метод для создания поисковика по нескольким индексам
   //    На входе: объект типа IndexReader
   //    На выходе: объект типа IndexSearcher для поиска по нескольким индексам
       public static IndexSearcher createMultiSearcher(IndexReader[] subReaders)
throws IOException, FileNotFoundException {
           MultiReader mReader = new MultiReader(subReaders, true);
           IndexSearcher searcher = new IndexSearcher(mReader);
           return searcher;


   }

   //    Ниже представлена группа методов, реализующих механизм парсинга запроса
из поисковой строки и создание на его основе
   //    запроса, в соответсвии с которым выполняется "нечёткий" поиск

   //    Базовый метод для создания "нечётких" поисковых запросов, созданный для
проверки работы механизма "нечёткого" поиска,
   //    используется для базовой обработки слова из строки запроса
   //    На входе: текст запроса и путь к директории со словарём коротких терминов
   //    На выходе: обработанная строка запроса, понятная объекту IndexSearcher
```

```
       public static String QueryMaker(String queryLine, String dict_path) throws
FileNotFoundException, ParseException {
```

Продолжение Листинга 1.1.2

```
   // reading dictionary from file to an array

           List<String> stopWords = new ArrayList();
           Scanner input = new Scanner(new File(dict_path + File.separator +
"dict.txt"));

   //        обращаемся к словарю коротких терминов
           while (input.hasNext()) {
               stopWords.add(input.nextLine());
           }
           //reading query and converting for fuzzy query with cheking if term is
in the dictionary of abbreviates and shor words
           StringBuilder sb = new StringBuilder(1000);

   //        убираем из запроса все спецсимволы
           String uq = queryLine;
           uq = uq.replace(",", " ");
           uq = uq.replace(".", " ");
           uq = uq.replace(";", " ");
           uq = uq.replace(":", " ");
           uq = uq.replace("(", " ");
           uq = uq.replace(")", " ");
           uq = uq.replace("{", " ");
   uq = uq.replace("}", " ");
           uq = uq.replace("-", " ");
           uq = uq.replace("_", " ");
           uq = uq.replace("'", " ");
           uq = uq.replace("[", " ");
           uq = uq.replace("]", " ");
           uq = uq.replace("*", " ");
           uq = uq.replace("?", " ");
           uq = uq.replace("/", " ");
   //        далее процесс парсинга строки на термины и "оборачивания" каждого
   //        отдельного термина в теги и спецсиволы для "нечёткого" поиска
           for (String word : uq.split(" ")) {
               if (word.equals("")) {
                   continue;
               }

               if (word.charAt(0) == '+') {
                   sb.append('+');
          word = word.substring(1);
                }
               if (word.charAt(0) == '-') {
                   sb.append('-');
                   word = word.substring(1);
               }
               int fl = 0;
               for (int i = 0; i < stopWords.size(); i++) {
                         if (word.equalsIgnoreCase(stopWords.get(i))) {
                       fl = 1;
                       break;
                   }
               }
               if ((fl == 1) || (word.length() < 3)) {
                   sb.append(word);
```

```
                    if (sb.length() > 0) {
                    sb.append(" ");
                }
            } else {
                if (word.length() < 5) {
                    Term term1 = new Term("body", word);
                    Query q3 = new FuzzyQuery(term1, 1);
                    sb.append(q3.toString() + " ");
                } else {
                    Term term1 = new Term("body", word);
                    Query q3 = new FuzzyQuery(term1, 2);
                    sb.append(q3.toString() + " ");
                }
            }
        }

            return sb.toString();
        }

    //    Метод, в котором создаётся связь между предыдущим запросом и следующим
    //    На входе: предыдущий запрос, следующий запрос и путь к директории со словарём коротких терминов
    //    На выходе: связанные посредством специального оператора AND предыдущий и следующий запрос
        public static String NextQueryMaker(String prevQuery, String query, String dict_path) throws FileNotFoundException, ParseException {

            StringBuilder sb = new StringBuilder(1000);
            sb.append('(');
            sb.append(prevQuery);
            sb.append(") AND (");
            sb.append(QueryMaker(query, dict_path));
            sb.append(')');

            return sb.toString();
        }
            //Query to include ids in searching
        public static Query ID_query_maker(List<String> id_list) throws ParseException {
            QueryParser qp = new QueryParser("id", new StandardAnalyzer());
            qp.setAllowLeadingWildcard(true);
            List<String> strList = null;

            for (String str : id_list) {
                StringBuilder sb = new StringBuilder();
                sb.append(" AND ");
                sb.append('"');
                sb.append(str);
                sb.append('"');
                strList.add(sb.toString());
            }

            StringBuilder sb = new StringBuilder();
            for (String str : strList) {
                sb.append(str);
            }

            Query resQuery = qp.parse(sb.toString());
```

```
      return resQuery;
       }
       //Query to exclude ids from searching
       public static Query NOT_ID_query_maker(List<String> id_list) throws
ParseException {
           QueryParser qp = new QueryParser("id", new StandardAnalyzer());
           qp.setAllowLeadingWildcard(true);
           List<String> strList = null;

           for (String str : id_list) {
               StringBuilder sb = new StringBuilder();
               sb.append(" NOT ");
               sb.append('"');
               sb.append(str);
               sb.append('"');
               strList.add(sb.toString());
           }
           StringBuilder sb = new StringBuilder();
           for (String str : strList) {
               sb.append(str);
           }
           Query resQuery = qp.parse(sb.toString());
           return resQuery;
       }
     public static TopDocs Scope_Searcher(IndexSearcher searcher, List<String>
query_list, List<String> scope_list, int num) throws FileNotFoundException,
ParseException, IOException {
           QueryParser qp = new QueryParser("tags", new RussianAnalyzer());
           qp.setAllowLeadingWildcard(true);

           Query tag_query = null, key_query = null;
           String str1 = null, str2 = null;
           if (query_list.size() == 1) {
               str1 = QueryMaker(query_list.get(0));
           } else {
               List<String> ql = new ArrayList();
               ql.add(0, QueryMaker(query_list.get(0)));

               for (int i = 1; i < query_list.size(); i++) {
                   ql.add(i, NextQueryMaker(ql.get(i - 1), query_list.get(i)));
               }
               str1 = ql.get(ql.size() - 1);
           }
           tag_query = qp.parse(str1);

           QueryParser qp1 = new QueryParser("tabl", new StandardAnalyzer());
           qp1.setAllowLeadingWildcard(true);

           StringBuilder sb = new StringBuilder();
           for (String word : scope_list) {
               if (sb.length() > 0) {
                   sb.append(' ');
               }
               sb.append(word);
           }
           str2 = sb.toString();
           key_query = qp1.parse(str2);
```

```
     return searcher.search(Concatenator(Arrays.asList(tag_query, key_query)),
num);

       }

   //    Метод, в котором реализуется механизм извлечения фрагментов текста из по-
ля hits типа TopDocs объекта Result, созданного без списка фрагментов
   //    На входе: объект типа Result без поля с фрагментами текста, объект-
поисковик типа IndexSearhcer, объект-стеммер типа Analyzer
   //    На выходе: объект типа Result с полем, содержащим искомые фрагменты тек-
ста документа
       public static Result HighlitgherFragmenter(Result result, IndexSearcher
multiSearcher, Analyzer analyzer) throws IOException, InvalidTokenOffsetsException
{
   //        Создаём пустой список фрагментов документа, а также инициализируем
объект типа Highlighter со служебным объектом типа SimpleHTMLFormatter
           List<List<TextFragment>> fragList = new
ArrayList<List<TextFragment>>();
           SimpleHTMLFormatter htmlFormatter = new SimpleHTMLFormatter();
           Highlighter highlighter = new Highlighter(htmlFormatter, new
QueryScorer(result.query));
   //        Двигаясь по циклу результатов поиска объекта типа Result, извлекаем
из поля "body" искомые текстовые фрагменты
           for (ScoreDoc scoreDoc : result.hits.scoreDocs) {
               int id = scoreDoc.doc;
               Document doc = multiSearcher.doc(id);
               String text = doc.get("body");
               TokenStream tokenStream =
TokenSources.getAnyTokenStream(multiSearcher.getIndexReader(), id, "body", analyz-
er);

   fragList.add(Arrays.asList(highlighter.getBestTextFragments(tokenStream, text,
false, 4)));
           }
           return new Result(result.hits, result.query, fragList);
       }

   //   Метод, в котором реализован сам механизм выполнения поискового запроса и
формирования из него объекта типа Result
   //   с полем, содержащим искомые фрагменты текста документа
   //   На входе: объект-поисковик типа IndexSearcher, список запросов, путь к ди-
ректории со словарём коротких терминов, а также количество результатов поиска,
   //   которые требуется сохранить для вывода на экран
   //   На выходе: объект типа Result с полем, содержащим искомые фрагменты текста
документа
       public static Result Searcher(IndexSearcher searcher, List<String> que-
ry_list, String dict_path, int num) throws IOException, FileNotFoundException,
ParseException, InvalidTokenOffsetsException {
           //Инициализируем стеммер
           Analyzer analyzer = new RussianAnalyzer();
   //        Инициализируем парсер запросов
           QueryParser qp = new QueryParser("body", analyzer);
   //        Активируем функцию использования специального языка запросов
           qp.setAllowLeadingWildcard(true);
           String str = null;
   //        На основе поисковой строки генерируем поисковый запрос в терминах
Lucene
           if (query_list.size() == 1) {
               str = QueryMaker(query_list.get(0), dict_path);
```

Продолжение Листинга 1.1.2

```
            } else {

                List<String> ql = new ArrayList();

                for (int i = 0; i < query_list.size(); i++) {
                    if (i == 0) {
                        ql.add(0, QueryMaker(query_list.get(0), dict_path));
                    } else {
                        ql.add(i, NextQueryMaker(ql.get(i - 1), query_list.get(i),
dict_path));
                    }
                }
                str = ql.get(ql.size() - 1);
            }

            Query query = qp.parse(str);
    //        Выполняем ранжирование поискового индекса в соответсвии с полученным
запросом
            TopDocs hits = searcher.search(query, num);

    //        позвращаем объект типа Result, предварительно извлекая методом
HighlitgherFragmenter искомые текстовые фрагменты
            return HighlitgherFragmenter(new Result(hits, query), searcher, analyz-
er);
        }
    }
    //Method to unite the different queries
        public static Query Concatenator(List<Query> query_list) {

            BooleanQuery.Builder bq = new BooleanQuery.Builder();

            for (int i = 0; i < query_list.size(); i++) {
                bq.add(query_list.get(i), BooleanClause.Occur.MUST);
            }

            return bq.build();
        }
        //**  Subgroup of methods, realising ANN-based searching  **//
        //  Method, realising ranking of documents which were predicted by ANN //
        public static TopDocs Neuro_Searcher(IndexSearcher searcher, List<String>
query_list, List<String> id_list, int num) throws ParseException,
FileNotFoundException, IOException {
            QueryParser qp = new QueryParser("tags", new RussianAnalyzer());
            qp.setAllowLeadingWildcard(true);
            String str = null;

            Query idQuery = ID_query_maker(id_list);
            if (query_list.size() == 1) {
                str = QueryMaker(query_list.get(0));
            } else {
                List<String> ql = new ArrayList();
                for (int i = 0; i < query_list.size(); i++) {
                    if (i == 0) {
                        ql.add(0, QueryMaker(query_list.get(0)));
                    } else {
                        ql.add(i, NextQueryMaker(ql.get(i - 1), que-
ry_list.get(i)));
                    }
                }
```

```
                str = ql.get(ql.size() - 1);
```

Продолжение Листинга 1.1.2

```
    }

            List<Query> queriesToConcatenateList = new ArrayList();
            queriesToConcatenateList.add(qp.parse(str));
            queriesToConcatenateList.add(idQuery);

            Query resQuery = Concatenator(queriesToConcatenateList);

            if (id_list.size() <= num) {
                return searcher.search(resQuery, id_list.size());
            } else {
                return searcher.search(resQuery, num);
            }
        }
        // Method, realising ranking of documents, excluding documents wich were
predicted by ANN   //
        public static TopDocs After_Neuro_Searcher(IndexSearcher searcher,
List<String> query_list, List<String> id_list, int num) throws ParseException,
FileNotFoundException, IOException {
            QueryParser qp = new QueryParser("tags", new RussianAnalyzer());
            qp.setAllowLeadingWildcard(true);
            String str = null;

            Query NOTidQuery = NOT_ID_query_maker(id_list);
            if (query_list.size() == 1) {
                str = QueryMaker(query_list.get(0));
            } else {
                List<String> ql = new ArrayList();

                for (int i = 0; i < query_list.size(); i++) {
                    if (i == 0) {
                        ql.add(0, QueryMaker(query_list.get(0)));
                    } else {
                        ql.add(i, NextQueryMaker(ql.get(i - 1), que-
ry_list.get(i)));
                    }
                }

                str = ql.get(ql.size() - 1);
            }

            List<Query> queriesToConcatenateList = new ArrayList();
            queriesToConcatenateList.add(qp.parse(str));
            queriesToConcatenateList.add(NOTidQuery);

            Query resQuery = Concatenator(queriesToConcatenateList);

            return searcher.search(resQuery, num - id_list.size());
        }

        //Method, that makes choice which type of searching system needs to use
based on fact, has ANN predicted smht or not    //
        public static List<TopDocs> SearcherChoicer(SearchList sl, int num,
IndexSearcher searcher) throws IOException, FileNotFoundException, ParseException
{
            List<TopDocs> result = new ArrayList();

            if (sl.ids.size() == 0) {
```

```
                result.add(Searcher(searcher, sl.queries, num));
            } else {
```

Продолжение Листинга 1.1.2

```
                result.add(Neuro_Searcher(searcher, sl.queries, sl.ids, num));
                result.add(After_Neuro_Searcher(searcher, sl.queries, sl.ids,
num));
            }
            return result;
        }
```

Листинг 1.2 – Пакет Indexer

Листинг 1.2.1 – Класс Item

```
package Indexer;

import java.io.BufferedReader;
import java.io.FileInputStream;
import java.io.IOException;
import java.io.InputStreamReader;
import java.util.ArrayList;
import java.util.List;

/**
 * In THIS class we are converting list of tags into structure,
 * that consists of pairs of tags and encrypted keys.
 * @author Dima
 */
public class Item {

    public String key;
    public String desc;



    Item(String key_in, String desc_in) {
        this.key = key_in;
        this.desc = desc_in;
    }

    public static List<Item> DocToList(String path) {
        List<Item> itemlist = new ArrayList();
        if (path == "") {
            path = "C:\\Ltest\\defs_in.txt";
        }

        //Read file and process
        try {
            FileInputStream fstream = new FileInputStream(path);
            BufferedReader br = new BufferedReader(new
InputStreamReader(fstream));
            String strLine;
            try {
                strLine = br.readLine();
            } catch (IOException e) {
```

```
                    System.err.println("Ошибка");
                }
```

Продолжение Листинга 1.2.1

```
    while ((strLine = br.readLine()) != null) {
                    int ind = 0;
                    int interation = -1;
                    for (int i = strLine.length() - 1; i > 0; i--) {

                        if (strLine.charAt(i) == ';') {
                            ind = i;
                            interation++;
                            if (interation == 1) {
                                break;
                            }
                        }
                    }
                    String buf = new String(strLine.substring(strLine.indexOf(';')
+ 1));
                    String desc = new String(strLine.substring(strLine.indexOf(';')
+ 1, ind + 1));
                    String key = new String(strLine.substring(ind + 1,
strLine.length() - 1));

                    itemlist.add(new Item(key, desc));
                }
            } catch (IOException e) {
                System.err.println("Ошибка");
            }

            return itemlist;
        }
    }
```

Листинг 1.2.2 – Класс Indexer

```
    package Indexer;

    import org.apache.lucene.analysis.ru.RussianAnalyzer;
    import org.apache.lucene.document.*;
    import org.apache.lucene.index.*;
    import org.apache.lucene.store.FSDirectory;

    import java.io.IOException;
    import java.nio.file.Paths;
    import java.util.ArrayList;
    import java.util.List;

    public class TestIndexing {

        public static List<Document> createDocs(List<Item> ItemList) {

            List<Document> docs = new ArrayList<>();

            FieldType descType = new FieldType();

descType.setIndexOptions(IndexOptions.DOCS_AND_FREQS_AND_POSITIONS_AND_OFFSETS);
            descType.setStored(true);
            descType.setTokenized(true);
            descType.setStoreTermVectors(true);
```

```
            descType.setStoreTermVectorPositions(true);
            descType.setStoreTermVectorOffsets(true);
```

Продолжение Листинга 1.2.2

```
            descType.setStoreTermVectorPayloads(true);

            FieldType keyType = new FieldType();
            keyType.setIndexOptions(IndexOptions.DOCS);
            keyType.setStored(true);
            keyType.setTokenized(true);

            FieldType tablType = new FieldType();
            tablType.setIndexOptions(IndexOptions.DOCS);
            tablType.setStored(true);
            tablType.setTokenized(true);
            FieldType idType = new FieldType();
            idType.setIndexOptions(IndexOptions.DOCS);
            idType.setStored(true);
            idType.setTokenized(true);
            int k = 0;
            for (Item i : ItemList) {
                Document doc = new Document();
                Field descField = new Field("tags", i.desc, descType);
                doc.add(descField);

                Field keyField = new Field("key", i.key, keyType);
                doc.add(keyField);

                String str = null;
                if (i.key.lastIndexOf('&') > 0) {
                    String buf = i.key.trim();
                    int a = buf.lastIndexOf(' ');
                    int b = buf.lastIndexOf('&');
                    if (b > a) {
                        str = buf.substring(a, b);
                    }
                } else {
                    str = " ";
                }
                Field tablField = new Field("tabl", str, tablType);
                doc.add(tablField);

                Field idField = new Field("id", String.valueOf(k), idType);
                doc.add(idField);
                k++;
                docs.add(doc
            }
            return docs;
        }
        public static void main(String[] args) throws IOException {
            FSDirectory dir = FSDirectory.open(Paths.get(TestConstants.INDEX_DIR));
            IndexWriterConfig config = new IndexWriterConfig(new
RussianAnalyzer());
            try (IndexWriter writer = new IndexWriter(dir, config)) {
                writer.deleteAll();
                List<Item> ItemList = Item.DocToList("");
                List<Document> docs = TestIndexing.createDocs(ItemList);
                writer.addDocuments(docs);
                writer.commit();
            }
        }}
```