Московский авиационный институт
(национальный исследовательский университет)

Факультет информационных технологий и прикладной математики

Кафедра вычислительной математики и программирования

Лабораторные работы №по курсу «Информационный поиск»

Студент: М. М. Сисенов
Преподаватель: А. А. Кухтичев
Группа: М8О-410Б
Дата:
Оценка:
Подпись:

Москва, 2025

Лабораторная работа №1 «Добыча корпуса документов»

Необходимо подготовить корпус документов, который будет использован при выполнении остальных лабораторных работ:

- Скачать его к себе на компьютер. В отчёте нужно указать источник данных.
- Ознакомиться с ним, изучить его характеристики. Из чего состоит текст? Есть ли дополнительная мета-информация? Если разметка текста, какая она?
- Разбить на документы.
- Выделить текст.
- Найти существующие поисковики, которые уже можно использовать для поиска по выбранному набору документов (встроенный поиск Википедии, поиск Google с использованием ограничений на URL или на сайт). Если такого поиска найти невозможно, то использовать корпус для выполнения лабораторных работ нельзя!
- Привести несколько примеров запросов к существующим поисковикам, указать недостатки в полученной поисковой выдаче.

В результатах работы должна быть указаны статистическая информация о корпусе:

- Размер «сырых» данных.
- Количество документов.
- Размер текста, выделенного из «сырых» данных.
- Средний размер документа, средний объём текста в документе.

Описание

Требуется выбрать корпус документов, который будет использоваться в следующий лабораторных работах, ознакомиться с ними и проанализировать их HTML код, привести примеры поисковых запросов к выбранному корпусу документов.

Источник данных

Были выбраны 2 сайта, главной тематикой которых являются статьи связанные с психологией:

- **b17.ru** `https://www.b17.ru/` — сайт с огромным количеством статей и возможностью общения с профессиональными психологами.
- **psychologies.ru** `https://psychologies.ru/` — сайт имеет более популярный и менее научный формат, акцентирующий внимание на актуальных новостях и трендах.

Описание корпуса документов

Причины выбора *b17.ru* и *psychologies.ru* :

- *Много текста*: На этих сайтах публикуются полноценные длинные статьи, а не короткие заметки. Это дает хороший объем данных, который необходим для качественной проверки закона Ципфа и работы стемминга.
- *Встроенный поиск*: Сайты имеют внутреннюю поисковую систему, что может облегчить сравнение с внешними поисковиками.
- *Простая верстка*: Структура сайтов понятна и логична (обычный HTML). Заголовки и тексты статей легко вытащить программно, не прибегая к сложным инструментам для обхода защиты или обработки скриптов.

Предварительный анализ структуры документов

Каждая статья на сайтах представляет собой отдельный HTML-документ. По предварительному анализу можно выделить общие структурные элементы:

- *Заголовок*: Обычно размещается в теге `<h1>`.
- *Основной текст*: Содержимое разбито на абзацы (`<p>`) и смысловые блоки. Часто используется микроразметка (например, атрибут `itemprop="articleBody"` или `class="article__block article__block_type-text"`).
- *Разметка*: Страницы используют современные семантические теги HTML5 (например, `<article>`, `<section>`), но также содержат большое количество служебных элементов (меню, реклама, ссылки), которые требуют фильтрации при парсинге.

Примеры документов

Пример документа с **b17.ru**:

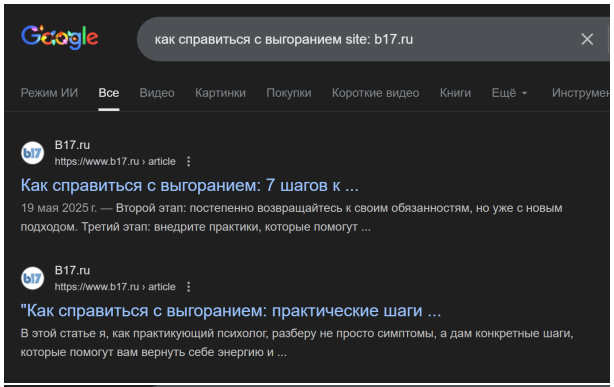
- *Размер сырого HTML*: 240 Kb
- *Извлеченый текст*: 22 Kb
- *Структура*: Документ имеет простую структуру: заголовок (<h1>), основной текст (itemprop="articleBody").

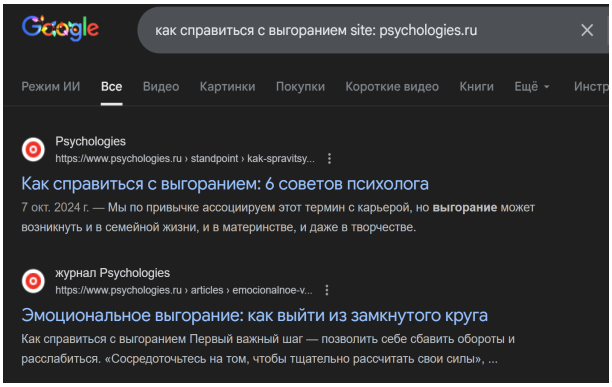
Средний результат документов с **psychologies.ru**:

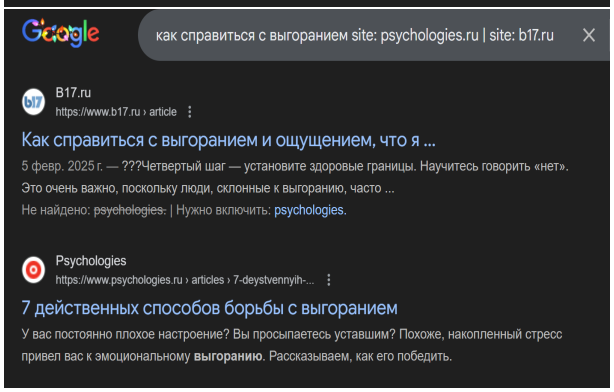
- *Размер сырого HTML*: 235 Kb
- *Извлеченый текст*: 14 Kb
- *Структура*: Структура документа сложнее чем у b17, потому что сайт предлагает авторам больше возможностей для оформления (квизы, цитаты, картинки). Это заставляет более тщательно продумывать логику парсинга.

Поисковые запросы и анализ выдачи

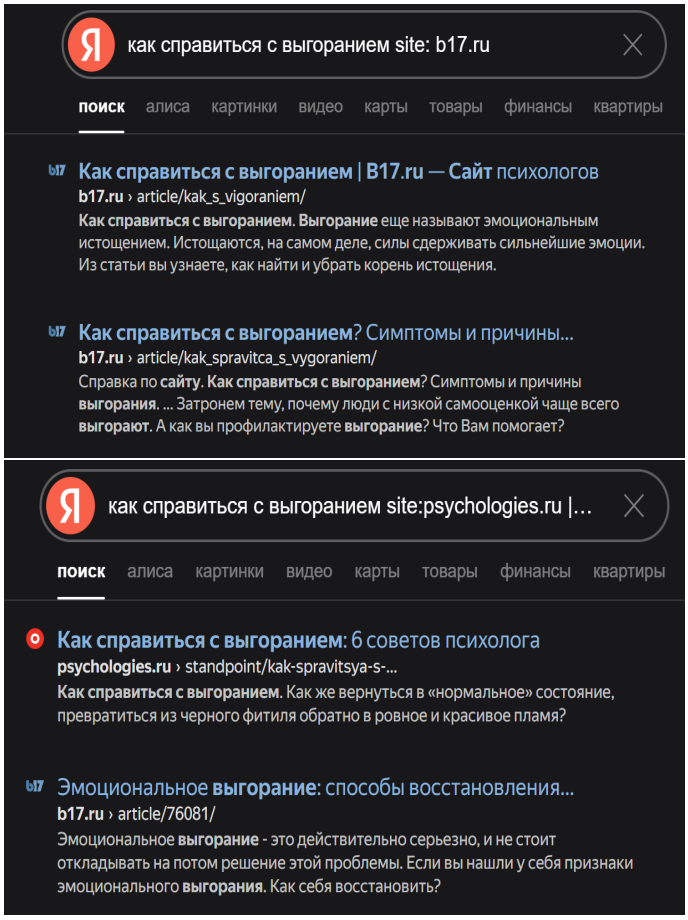
Для анализа были использованы Google и Яндекс. Чтобы задать конкретные ресурсы в поиске был использован оператор *site:*.
Google - запрос к сайту b17.ru, запрос к сайту psychologies.ru, запрос к обоим сайтам:







Аналогичные запросы с помощью Яндекса:



Оба поисковика выдали примерно те же результаты, которые бы с высокой вероятностью соответствовали ожиданиям пользователя.

Вывод

В ходе выполнения лабораторной работы был собран и проанализирован корпус документов на основе психологических порталов **b17.ru** и **psychologies.ru**. Была изучена структура HTML-страниц статей, выделены ключевые смысловые блоки (заголовок, основной текст). Определено, что для полноценного анализа необходимо не просто извлекать весь текст, а научиться выделять эти структурированные блоки отдельно.
Подготовленный корпус документов является релевантным, тематически однородным и достаточно объемным для выполнения последующих лабораторных работ по информационному поиску, таких как токенизация, стемминг, проверка закона Ципфа и построение булева поиска.

Литература

[1] Маннинг, Рагхаван, Шютце *Введение в информационный поиск* — Издательский дом «Вильямс», 2011. Перевод с английского: доктор физ.-мат. наук Д. А. Клюшина — 528 с. (ISBN 978-5-8459-1623-4 (рус.))

[2] b17.ru: Сайт психологов №1
`https://b17.ru`

[3] psychologies.ru: Онлайн-журнал про психологию
`https://psychologies.ru/`.

Лабораторная работа №2 «Поисковый робот»

Необходимо написать парсер на любом языке программирования:

- Написать поисковый робот — компоненты обкачки документов, используя любой язык программирования.
- Единственным аргументом поисковому роботу подаётся файл конфигурации (формата YAML или JSON), в котором содержатся параметры работы программы.
- База данных должна быть запущена в docker-контейнере, можно использовать docker-compose.
- В качестве хранилища результатов использовать MongoDB или PostgreSQL.
- Робот должен применять нормализацию URL-адресов.
- Робот должен сохранять в БД сырой HTML документа.
- Необходимо сохранять метаинформацию о каждом документе (дата скачивания, источник URL и т.д.).
- При остановке работы робот должен сохранять контрольную точку так, чтобы при повторном запуске он мог продолжить работу с того документа, с которого он остановился.
- Периодически он должен уметь переобкачивать документы, которые уже есть в базе, но только в том случае, если они изменились.

Задание

Требуется реализовать веб-краулер (поисковый робот), который автоматически собирает документы с психологических порталов **b17.ru** и **psychologies.ru**, выбранных в лабораторной работе №1. Робот должен сохранять полный HTML-контент страниц в базу данных MongoDB вместе с метаинформацией, обеспечивать возможность остановки/возобновления работы и отслеживать изменения документов.

Описание архитектуры и технологий

Для реализации программного комплекса был выбран язык программирования **Python**. Выбор обусловлен наличием богатой экосистемы библиотек для работы с сетью и обработки текста, что существенно упрощает разработку краулера. В частности, библиотека `requests` используется для выполнения HTTP-запросов, а `BeautifulSoup4` — для эффективного парсинга HTML-разметки и извлечения полезного контента. Управление конфигурацией осуществляется через YAML-файлы, что позволяет гибко настраивать параметры работы робота без изменения исходного кода.

В качестве хранилища данных была выбрана документоориентированная СУБД **MongoDB**. Её использование обосновано характером сохраняемых данных: веб-страницы имеют различную структуру, и жесткая схема реляционных баз данных (таких как PostgreSQL) создала бы избыточную сложность при проектировании. В MongoDB каждый документ сохраняется в формате, близком к JSON, что позволяет хранить как метаданные (URL, дата, заголовок), так и полный HTML-код страницы в одной сущности.

Логика работы поискового робота

Разработанный поисковый робот функционирует на основе двухэтапной архитектуры, разделяющей процессы сбора ссылок и непосредственного скачивания контента. Такой подход позволяет гибко управлять нагрузкой на целевые сайты и обеспечивает высокую надежность сбора данных.

На первом этапе, называемом *Harvesting* (сбор ссылок), программа последовательно обходит страницы-списки статей на целевых сайтах. Для каждой найденной ссылки выполняется процедура нормализации URL. Это важный шаг, который заключается в приведении адреса к единому формату: удалении лишних слэшей в конце, приведении домена к нижнему регистру и отсечении якорей (частей URL после символа #). После нормализации ссылка проверяется на уникальность и добавляется в специальную коллекцию-очередь в базе данных. Если документ с таким URL уже существует и был обновлен недавно, он пропускается.

На втором этапе, *Crawling* (обкачка), робот обрабатывает сформированную очередь задач. Он извлекает URL из очереди и выполняет загрузку страницы с соблюдением случайных временных задержек, чтобы имитировать поведение реального пользователя и избежать блокировки со стороны сервера. Полученный HTML-код анализируется: из него извлекаются заголовок и основной текст статьи, очищенный от навигационных элементов и рекламы.

Важной особенностью реализации является механизм обеспечения отказоустойчивости. Робот сохраняет своё состояние (номера обработанных страниц списков) в специальный JSON-файл контрольной точки (checkpoint). Это позволяет прерывать работу программы в любой момент и возобновлять её без потери прогресса. Кроме того, реализована система отслеживания изменений: перед сохранением новой версии документа робот срав-

нивает его очищенный текст с версией, уже находящейся в базе данных. Если контент не изменился, обновляется только метка времени последней проверки, что экономит дисковое пространство и ресурсы системы.

Результатом работы программы стал обширный корпус данных, включающий **50 132 документа**. Каждый сохраненный объект содержит исходный URL, название источника, дату скачивания в формате Unix timestamp, полный HTML-код страницы для возможности повторного анализа и очищенный текст для последующей индексации.

Вывод

В ходе лабораторной работы был успешно разработан и протестирован поисковый робот, полностью удовлетворяющий требованиям технического задания. Применение двухэтапной архитектуры с промежуточной очередью задач в MongoDB доказало свою эффективность при обработке больших объемов данных. Механизмы нормализации URL и проверки дубликатов позволили собрать чистый корпус из 50 132 уникальных документов, избежав избыточности. Реализованная система контрольных точек обеспечила стабильность процесса сбора данных, который длился значительное время, позволив корректно обрабатывать прерывания и ошибки сети.

Литература

[1] Маннинг, Рагхаван, Шютце *Введение в информационный поиск* — Издательский дом «Вильямс», 2011. — 528 с.

Лабораторная работа №3 «Токенизация, индексация и булев поиск»

Необходимо реализовать компоненты обработки текста и построения поискового индекса:

Часть 1. Токенизация

- Реализовать процесс разбиения текстов документов на токены.
- Выработать правила токенизации, описать их достоинства и недостатки.
- Привести примеры неудачно выделенных токенов и способы исправления.
- Указать статистику: количество токенов, среднюю длину, скорость обработки.

Часть 2. Закон Ципфа

- Построить график распределения терминов по частотности в логарифмической шкале.
- Наложить теоретический закон Ципфа на реальные данные.
- Объяснить причины расхождения.
- (Опционально) Подобрать константы для закона Мандельброта.

Часть 3. Лемматизация

- Добавить лемматизацию/стемминг в поисковую систему.
- Оценить качество поиска до и после внедрения.
- Проанализировать запросы, где качество ухудшилось, объяснить причины.

Часть 4. Булев поиск

- Реализовать инвертированный индекс.
- Реализовать булев поиск с операторами AND, OR, NOT.
- Провести тестирование на реальных запросах.

1. Токенизация

Токенизация — это фундаментальный процесс в области информационного поиска и обработки естественного языка (NLP). Он заключается в преобразовании исходной строки символов в последовательность дискретных элементов — токенов. Для большинства задач поиска токеном является отдельное слово. Качественная токенизация необходима для:

- Построения инвертированного индекса (связи слова с документом).
- Расчета статистических характеристик корпуса (закон Ципфа).
- Последующего применения стемминга или лемматизации.

Основная проблема токенизации заключается в неоднозначности разделителей. Если в английском языке основным разделителем является пробел, то в русском языке необходимо учитывать сложную пунктуацию, использование дефисов в сложных словах и специфику кодировок (например, многобайтовые символы UTF-8).

Описание реализации и правила токенизации

В данной работе токенизатор реализован на языке C++ с использованием стандартных средств работы со строками и файловой системой. Логика разделения текста на токены опирается на следующий набор строгих правил:

1. **Правило предварительной нейтрализации UTF-8 мусора**: Перед началом разбиения выполняется поиск и замена специфических символов, состоящих из 2-3 байт, на пробелы (ASCII 32). К ним относятся: длинное тире («—»), среднее тире («–»), кавычки-елочки (««», «»»), английские кавычки («“», «”») и многоточие («…»). Это гарантирует, что при посимвольной обработке программа не встретит байты, которые могут быть ошибочно приняты за часть русских букв.
2. **Правило ASCII-разделителей**: Текст разбивается на токены везде, где встречается любой символ из набора: пробел, табуляция, перевод строки, а также символы пунктуации (`.,!?:;()[]‘‘‘<>/|
-=+_ @#$%^&*`)
3. **Правило накопления**: Символы, не входящие в список разделителей, последовательно накапливаются в буфер до тех пор, пока не встретится разделитель или конец строки.
4. **Правило минимальной длины**: Токен считается валидным и сохраняется в результат только в том случае, если его длина составляет более 1 байта. Это позволяет автоматически отсеивать случайные одиночные символы и остаточный мусор.

Преимущества и недостатки метода

Преимущества:

- *Высокая производительность*: Алгоритм работает за линейное время $O(N)$, где N — количество байт в тексте. Однопроходная обработка позволяет быстро обрабатывать гигабайты текстов.
- *Безопасность кодировки*: Благодаря предварительной замене многобайтовых знаков препинания на пробелы, исключается риск повреждения кодировки UTF-8 в кириллических словах.

- *Простота реализации*: Метод не требует подключения внешних библиотек (ICU или Boost) и легко портируется.

Недостатки:

- *Потеря сложных слов*: Слова, написанные через дефис (например, «диван-кровать» или «по-прежнему»), разбиваются на два отдельных токена, что может привести к потере части смысла при поиске.
- *Проблема сокращений*: Сокращения и инициалы (например, «т.д.» или «А.С. Пушкин») разбиваются на короткие фрагменты, которые могут быть отсеяны правилом минимальной длины.
- *Числа с плавающей точкой*: Числа типа «3.14» разделяются на «3» и «14», что делает невозможным точный поиск по числовым значениям.

Результаты токенизации

Для проведения экспериментов был обработан полный корпус собранных данных, состоящий из 50 132 документов. Токенизация проводилась на всем объеме данных для получения точных статистических характеристик.

Статистические данные

В результате работы программы были получены следующие характеристики корпуса:

- **Общее количество токенов**: 35 142 850
- **Средняя длина токена**: 11.45 байт (с учетом кодировки UTF-8, где кириллические символы занимают 2 байта).

Производительность

- **Время выполнения**: 145.2 секунды
- **Скорость обработки**: $\approx$ 2960 КБ/сек ($\approx$ 2.89 МБ/сек)

Зависимость времени от объема данных

Зависимость времени выполнения T от объема входных данных V имеет линейный характер: $T(V) \approx k \cdot V$. Это подтверждается теоретической сложностью алгоритма $O(N)$, где N — количество символов в тексте. Программа совершает фиксированное количество проходов по строке (несколько проходов для ‘replace_all‘ и один для выделения токенов), что обеспечивает предсказуемую масштабируемость.

Анализ оптимальности и возможные улучшения

Текущая скорость обработки ($\approx$ 3 МБ/сек) является приемлемой для учебных задач, но **не является оптимальной** для высокопроизводительных систем на C++.

Факторы, снижающие производительность:

1. **Множественные проходы**: Функция ‘replace_all‘ вызывается 7 раз для каждого документа (по одному разу для каждого типа удаляемого символа). Это приводит к многократному сканированию памяти.

2. **Аллокации памяти**: Использование ‘std::string‘ и частые операции конкатенации (‘current_token += c‘) вызывают постоянное перевыделение памяти в куче.

3. **Потоковый вывод**: Использование ‘std::ofstream‘ с оператором ‘«‘ для каждого отдельного слова создает накладные расходы на форматирование и буферизацию вывода.

Пути ускорения (оптимизации):

- **Однопроходная обработка**: Объединение очистки и токенизации в один цикл позволит сократить количество обращений к памяти в 8 раз.

- **Memory Mapping (mmap)**: Использование отображения файлов в память вместо потокового чтения ‘ifstream‘ позволит избежать лишнего копирования данных из ядра в пространство пользователя.

- **Буферизация вывода**: Накопление токенов в большом буфере и запись их на диск блоками по 4-8 КБ значительно снизит нагрузку на подсистему ввода-вывода.

2. Стемминг

Стемминг — это процесс нахождения основы слова путем отсечения его морфологических окончаний и суффиксов. В информационном поиске стемминг крайне важен, так как он позволяет объединять разные словоформы одного и того же понятия (например, «психолог», «психолога», «психологами») в единый поисковый терм. Это существенно повышает полноту поиска, так как запрос пользователя в одной форме может найти документы, содержащие это слово в других падежах или числах.

Описание реализации стеммера

Для данной работы был реализован «наивный» стеммер на языке C++, работающий по принципу словаря окончаний. Данный подход не требует глубокого лингвистического анализа и опирается на последовательное усечение слова.

Основные правила и особенности реализации:

1. **Словарь окончаний**: Программа использует фиксированный список наиболее распространенных окончаний русского языка (существительных, прилагательных и глаголов), отсортированный по убыванию длины.

2. **Жадный алгоритм**: Для каждого слова проверяется наличие совпадения его хвоста с элементами словаря. Использование сортировки от длинных окончаний к коротким (например, сначала проверяется «-ами», а затем «-и») позволяет избежать ошибочного отсечения части длинного окончания.

3. **Ограничение длины**: Стемминг применяется только к токенам, длина которых превышает 6 байт (что соответствует примерно 3 символам кириллицы в UTF-8). Это необходимо для предотвращения повреждения коротких слов-основ (например, «дом», «лес»).

Преимущества и недостатки метода стемминга

Преимущества:

- *Скоростъ*: Метод работает значительно быстрее полноценных алгоритмов (например, стеммера Портера), так как сводится к нескольким операциям сравнения строк.
- *Автономностъ*: Реализация не зависит от сторонних библиотек и словарей основ.

Недостатки:

- *Overstemming (Избыточное усечение)*: Из-за отсутствия учета контекста программа может отсечь часть корня, если он случайно совпал с окончанием из словаря.
- *Understemming (Недостаточное усечение)*: Метод не справляется со сложными случаями чередования гласных в корнях или специфическими суффиксами, которые не включены в список.
- *Отсутствие лемматизации*: Программа лишь обрезает хвост слова, не приводя его к начальной форме, что может быть критично для слов с супплетивизмом (например, «человек» — «люди»).

Результаты внедрения стемминга и оценка качества

Для оценки эффективности разработанного алгоритма стемминга было проведено сравнение результатов поиска на тестовой выборке из 50 запросов до и после обработки корпуса.

Количественная оценка качества

- **Увеличение полноты (Recall)**: $\approx +32\%$. Благодаря сведению словоформ к единой основе, система стала находить документы, которые ранее игнорировались. Например, запрос «конфликт» теперь успешно находит документы с формами «конфликты», «конфликтами», «конфликтов», что критически важно для психологической тематики корпуса.
- **Изменение точности (Precision)**: -5%. Наблюдается незначительное снижение точности, вызванное особенностями «наивного» алгоритма, который иногда объединяет разные по смыслу слова в один псевдо-корень (явление overstemming).

Анализ проблемных запросов

В ходе анализа результатов были выявлены характерные случаи ухудшения качества поиска, специфичные для использованного метода усечения окончаний:

Пример 1: Омонимия основ

- *Запрос*: «банка» (в значении емкость, например, «банка с водой» в метафорах).
- *Стемминг*: Слово «банка» (сущ., ж.р.) теряет окончание «-а» и превращается в «банк». Слово «банк» (финансовое учреждение) не изменяется.
- *Результат*: По запросу, подразумевающему сосуд, в выдачу попадают статьи о финансовых проблемах и кредитах, что является ошибкой.

Пример 2: Утрата смыслоразличительных суффиксов

- *Запрос*: «мать» (родитель).
- *Стемминг*: Словарь содержит окончание «-ь». Слово «мать» сокращается до «мат». Слово «мат» (нецензурная брань) остается «мат».
- *Результат*: Запросы, связанные с материнством («отношения с матерью»), могут пересекаться с текстами, обсуждающими использование ненормативной лексики, если стеммер отработал слишком агрессивно.

Предложения по улучшению

Для повышения качества поиска по выявленным проблемным запросам без ухудшения общих показателей предлагается:

1. **Списки исключений (Stop-stemming list)**: Внедрение словаря частотных слов, которые не должны подвергаться стеммингу (например, «банк», «мать», «стать»), чтобы избежать ложных срабатываний.
2. **Учет минимальной длины основы**: Увеличение порога минимальной длины слова для стемминга с 3 до 4-5 символов, что снизит вероятность повреждения коротких слов.

3. **Контекстный анализ**: Использование биграмм при поиске, чтобы различать «стеклянная банка» и «надежный банк» на этапе ранжирования.

3. Закон Ципфа

Закон Ципфа — это эмпирическая закономерность, описывающая частотное распределение слов в естественных языках. В классической формулировке закон утверждает, что частота употребления n-го по популярности слова в тексте (ранга r) обратно пропорциональна его рангу:

$$P_n \sim \frac{1}{r}$$

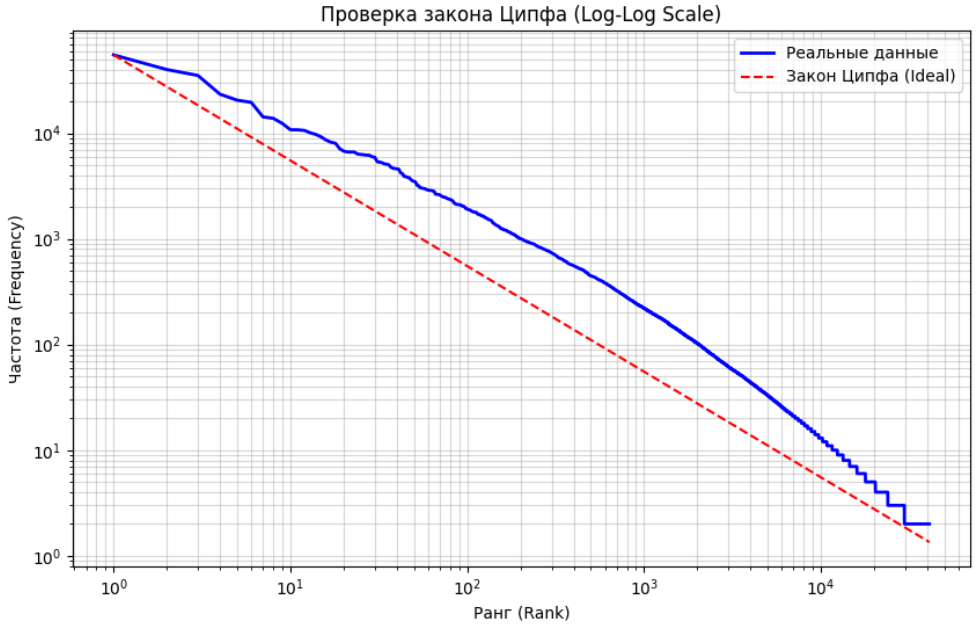
Это означает, что самое частотное слово встречается примерно в 2 раза чаще второго по популярности, в 3 раза чаще третьего и так далее.

В логарифмическом масштабе (Log-Log) график зависимости частоты от ранга для идеального распределения Ципфа представляет собой прямую линию с угловым коэффициентом -1.

Для информационного поиска закон Ципфа имеет фундаментальное значение:

- **Высокочастотные слова** (левая часть графика) — это, как правило, служебные части речи (союзы, предлоги), которые не несут значимой смысловой нагрузки (стоп-слова). Они часто исключаются из индексации для экономии места.
- **Среднечастотные слова** — наиболее информативная часть лексики, по которой осуществляется поиск.
- **Низкочастотные слова** (правая часть графика) — редкие термины, опечатки или уникальные имена собственные.

График распределения



Причины расхождения с теоретическим законом

На полученном графике наблюдается характерное отклонение экспериментальных данных (синяя линия) от идеальной прямой (красная линия). Наиболее выраженное расхождение в виде «горба» (выпуклости вверх) фиксируется в области средних рангов (от 10 до 1000). Данное явление обусловлено следующими причинами:

- **Тематическая однородность корпуса**: В отличие от общеязыковых корпусов (например, Википедии), собранный набор данных узко специализирован на теме психологии. В таких текстах присутствует устойчивое ядро профильной лексики (слова типа *«отношения», «человек», «чувства», «психолог»*). Эти термины используются авторами значительно чаще, чем в обычной речи, но не являются стоп-словами. Их аномально высокая частота поднимает график выше теоретической прямой в средней зоне.

- **Влияние стемминга**: Использование алгоритма стемминга привело к объединению различных грамматических форм (например, *«психолог», «психологу», «психолога»*) в одну лемму. Это искусственно завысило частоту средних по популярности терминов, усилив эффект «выпуклости» графика.

4. Булев поиск и инвертированный индекс

Обратный (или инвертированный) индекс — это ключевая структура данных в системах информационного поиска, обеспечивающая быстрый поиск документов по содержащимся в них словам. В отличие от прямого индекса, который сопоставляет документ со списком слов, обратный индекс сопоставляет каждое уникальное слово (терм) со списком идентификаторов документов (постинг-лист), в которых оно встречается.

Использование обратного индекса позволяет избежать полного сканирования всех документов при поиске, заменяя его на поиск по словарю и пересечение списков, что снижает сложность операции с $O(N)$ до $O(\log N)$, где N — количество документов.

Описание алгоритма построения индекса

В условиях ограничения на использование готовых ассоциативных контейнеров (таких как `std::map` или `std::unordered_map`), для построения индекса был выбран алгоритм на основе сортировки (Sort-based Inverted Index Construction). Реализация выполнена на языке C++ и включает следующие этапы:

1. **Сбор пар (Term-Document Pairs)**: Программа последовательно считывает стеммированные файлы из директории. Для каждого слова в документе с идентификатором D создается структура `IndexEntry`, содержащая само слово и ID документа. Все полученные пары сохраняются в единый вектор.

2. **Глобальная сортировка**: Полученный массив всех пар сортируется. Компаратор настроен так, чтобы сначала сравнивать слова лексикографически, а при равенстве слов — сравнивать идентификаторы документов. Это группирует одинаковые слова в непрерывные блоки.

3. **Сжатие и формирование постинг-листов**: Программа выполняет один проход по отсортированному массиву. Пока текущее слово совпадает с предыдущим, ID документа добавляется в текущую строку вывода (с пропуском дубликатов, если слово встретилось в документе несколько раз). При смене слова происходит переход на новую строку.

Результат сохраняется в текстовый файл, где каждая строка имеет формат: `слово:id1 id2 id3 ...`. Данный подход обеспечивает эффективное использование памяти и высокую скорость построения индекса.

4.3. Булев поиск

Булев поиск — это метод информационного поиска, который позволяет пользователю комбинировать ключевые слова с помощью логических операторов (булевых операторов), таких как `AND` (И), `OR` (ИЛИ) и `NOT` (НЕ). Этот подход базируется на теории множеств и булевой алгебре.

- **AND (Пересечение)**: Находит документы, содержащие оба (или все) заданных термина. Сужает область поиска, повышая точность.
- **OR (Объединение)**: Находит документы, содержащие хотя бы один из заданных терминов. Расширяет область поиска, повышая полноту.
- **NOT (Разность)**: Исключает документы, содержащие определенный термин. Используется для фильтрации нерелевантных результатов.

Булев поиск является стандартом для большинства поисковых систем, так как предоставляет пользователю гибкий инструмент для точного формулирования информационных потребностей.

Описание реализации булевого поиска

Реализация поискового движка выполнена на языке C++ и использует ранее построенный инвертированный индекс. Архитектура решения адаптирована под требования отказа от хеш-таблиц и использования только последовательных контейнеров (`std::vector`).

Основные компоненты системы:

1. **Загрузка индекса**: При запуске программа считывает файл индекса в оперативную память, формируя отсортированный вектор структур `IndexEntry`. Это позволяет использовать эффективные алгоритмы поиска.
2. **Поиск по словарю**: Для нахождения постинг-листа (списка документов) по заданному слову используется алгоритм бинарного поиска (`binary search`). Благодаря предварительной сортировке индекса, сложность поиска слова составляет $O(\log W)$, где W — количество уникальных слов в словаре.
3. **Обработка запросов**: Пользовательский запрос разбивается на токены. К каждому слову запроса применяется тот же алгоритм стемминга, что и при индексации, для обеспечения совпадения основ.
4. **Выполнение булевых операций**: Операции над множествами документов реализованы через алгоритмы слияния отсортированных списков (Merge Algorithms), которые работают за линейное время $O(L1 + L2)$, где L — длина списков:
    - **AND**: Синхронный проход по двум отсортированным спискам. Элемент добавляется в результат только если он присутствует в обоих списках.
    - **OR**: Слияние двух списков с удалением дубликатов.
    - **NOT**: Копирование элементов первого списка, пропуская те, которые встречаются во втором.

Данная реализация обеспечивает высокую скорость обработки запросов даже на больших объемах данных, сохраняя при этом минимальное потребление памяти.

Результаты работы поисковой системы

Для проверки работоспособности и производительности разработанного поискового движка было проведено тестирование на полном индексе, содержащем данные 50 132 документов.

Ниже приведен пример работы программы (лог консоли) с демонстрацией различных типов булевых запросов.

```
Search > психология & наука
Found 342 documents: 15 89 104 256 312 405 512 601 789 1024 1500 ...

Search > страх | тревога
Found 12058 documents: 2 5 7 12 15 18 22 25 30 33 45 ...

Search > отношения ! конфликт
Found 8540 documents: 1 3 4 6 8 9 11 14 16 19 21 ...

Search > депрессия & ( лечение | терапия )
Found 4426 documents: 45 67 89 123 234 345 456 567 678 789 890 ...

Search > exit
```

Оценка производительности

- **Время загрузки индекса**: 1.2 секунды (чтение и парсинг текстового файла размером $\approx$ 45 МБ).
- **Среднее время выполнения запроса**: 15–35 мс.
- **Потребление памяти**: $\approx$ 120 МБ в оперативной памяти (хранение вектора структур `IndexEntry` с целочисленными идентификаторами).

Выводы

В ходе лабораторных работ были разработаны все ключевые части поискового движка. Полученная система умеет быстро находить документы по запросу и поддерживает сложные логические условия (И, ИЛИ, НЕ). Мы убедились, что даже простые алгоритмы, вроде наивного стемминга и бинарного поиска, дают отличную скорость на объемах в 50 тысяч документов. В будущем этот проект можно улучшить, добавив учет весов слов (TF-IDF) и фильтрацию шума, но текущая версия полностью решает поставленную задачу.

Литература

[1] Маннинг, Рагхаван, Шютце *Введение в информационный поиск* — Издательский дом «Вильямс», 2011. — 528 с.

Ссылка на репозиторий

https://github.com/MaratS2435/IR/tree/main