Московский авиационный институт (национальный исследовательский университет)

Факультет информационных технологий и прикладной математики

Кафедра вычислительной математики и программирования

Лабораторные работы по курсу «Информационный поиск»

Студент: А. С. Чернаткин
Преподаватель: А. А. Кухтичев
Группа: М8О-401Б

Дата:
Оценка:
Подпись:

Москва, 2025

Содержание

Задание и постановка задачи

В данной лабораторной работе требуется подготовить полный набор инструментов для построения базовой поисковой системы на собственных структурах данных. Задача включает добычу корпуса документов, реализацию поискового робота для его наполнения, построение и проверку компонентов предобработки текста: токенизации и стемминга, сбор статистики по частотам слов и проверку закона Ципфа, построение булевого инвертированного индекса и реализацию булевого поиска как командной утилиты и веб-интерфейса. В отчёте необходимо подробно описать архитектуру решения, форматы данных, алгоритмы и принятые инженерные решения, а также привести результаты тестирования и автоматическую оценку качества поиска. Требуется обеспечить воспроизводимость запуска: указать точную последовательность команд для сборки и запуска, а также включить скрипты для генерации оценочных файлов (qrels/results). Итоговый документ должен содержать исходные тексты, скрипт сборки и инструкцию по запуску тестовой процедуры. В тексте отчёта должны быть приведены численные характеристики корпуса и результаты метрической оценки (P, NDCG, ERR) — либо по ручной разметке, либо по автоматически сгенерированным qrels.

1 Добыча корпуса документов

1.1 Описание

Этап добычи корпуса предназначен для формирования единого набора документов, который будет использоваться дальше для индексации и тестов качества поиска. На практике корпус должен представлять собой структурированный набор текстовых файлов в кодировке UTF-8, каждый файл при этом — отдельный документ с явным заголовком и телом. В нашей реализации краулер ориентирован на получение больших тематических коллекций, при этом уделяется внимание одинаковой кодировке всех файлов, надёжному логированию и возможности возобновления работы после сбоев. Важным требованием является соблюдение прав сайта и корректная обработка `robots.txt`; краулер проверяет правила доступа и кеширует их для каждого домена. Кроме этого реализована механика politeness — паузы между запросами к одному домену для снижения нагрузки. Для реальной работы необходимо заранее определить пороги по минимальному количеству слов в документе, чтобы отсечь короткие страницы и шум.

1.2 Архитектура решения

Архитектура краулера делится на логические модули: менеджер очереди, загрузчик HTTP, парсер HTML, модуль выделения основного текста и модуль хранения файлов. Менеджер очереди обеспечивает устойчивое хранение списка URL и состояний обработки; при масштабировании очередь можно вынести в внешнюю СУБД или систему сообщений. Загрузчик выполняет HTTP-запросы с поддержкой таймаутов и повтора при ошибках; он также анализирует заголовки и кодировки ответа. Парсер HTML находит основные семантические теги и применяет эвристики для выбора основного текстового блока; этот этап критичен для качества корпуса. Файловый модуль сохраняет документы в стандартизированном виде и фиксирует meta-данные (исходный URL, время загрузки). Важная деталь — способность периодически сохранять состояние очереди и списка посещённых URL, что даёт возможность безопасно возобновлять процесс после сбоев.

1.3 Реализация

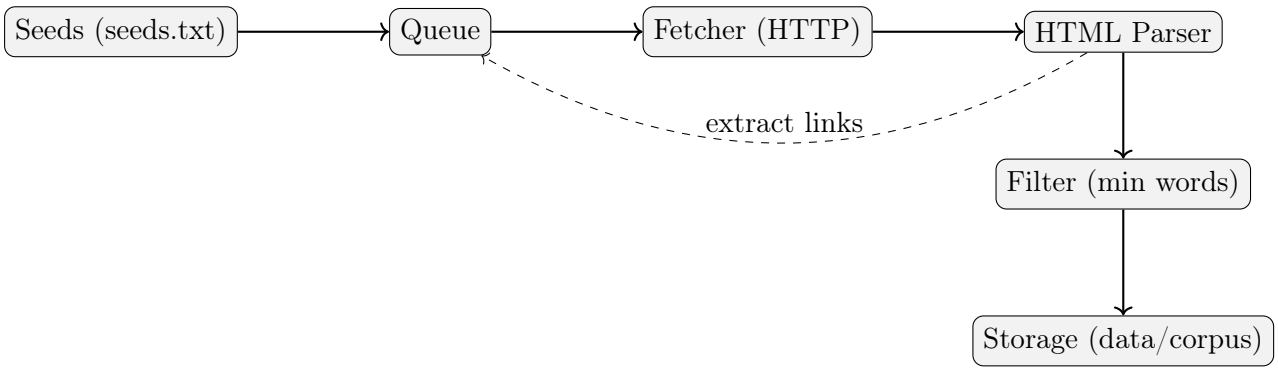
Реализация выполнена в виде Python-скрипта, использующего requests и BeautifulSoup для загрузки и парсинга. Код организован в виде функций с чёткими контрактами: fetch(url) -> `raw_html`, `extract_text(html)` -> `cleaned_text`, `save_document(id, title, text)` и т.д. Для выхода из повторяющихся ошибок реализована стратегия экспоненциального бэкоффа, а также максимальное число попыток для одного URL. Для совместимости с разными сайтами предусмотрены эвристики определения кодировки, а затем чистая конвертация в UTF-8 перед сохранением. Для минимизации коллизий имён файлов применяется порядковая нумерация с префиксом `doc_`. При

масштабировании полезно добавить параллельные воркеры с разделением по доменам, чтобы не нарушать politeness.

1.4 Тестирование и контроль качества

Тестирование включает несколько уровней: unit-тесты для функций парсинга, интеграционные тесты, имитирующие скачивание набора страниц, и прогон по небольшой тестовой коллекции с ручной проверкой результатов. В тестах проверяется корректность кодировки, отсутствие пустых файлов, соблюдение минимального порога слов и корректность сохранённых meta-данных. Также проверяется устойчивость к ошибкам сети: сценарии с долгим тайм-аутом, 5xx ответами и редиректами. Результаты тестирования фиксируются в лог-файле, в котором указываются URL, статус обработки и причина пропуска при наличии.

1.5 Диаграмма потока краулера



2 Поисковый робот

2.1 Описание

Поисковый робот — это компонент, который отвечает за регулярное и корректное наполнение корпуса документами. В учебном проекте робот реализован как Python-утилита с модульной структурой, где каждая функциональная часть вынесена в отдельную функцию для удобства тестирования и замены. Робот должен уметь корректно управлять очередью обхода, учитывать правила доступа (`robots.txt`), обрабатывать ошибки сети и сохранять результаты в стандартизированном формате. Важная часть — логирование и мониторинг: робот фиксирует успешные и неуспешные попытки, что позволяет анализировать причины потерь документов и улучшать эвристику фильтрации. Политика politeness встроена в ядро робота, поэтому он не будет посылать много запросов к одному домену за короткий промежуток времени.

2.2 Архитектура и взаимодействие компонентов

Архитектура робота строится по принципу «одно назначение — один модуль»: менеджер очереди, проверка robots, загрузчик, парсер, фильтр и модуль хранения. Менеджер очереди отвечает также за дедупликацию URL и контроль глубины обхода; это убирает проблему бесконечных циклов. Модуль проверки robots кеширует парсеры правил, чтобы избежать лишних сетевых вызовов, а загрузчик управляет сессией HTTP и повторными попытками. Парсинг HTML и очистка текста выполняются в отдельном модуле, что даёт возможность легко заменить алгоритм выделения основного текста (например, подключить Readability). Модуль хранения отвечает за конвертацию в UTF-8 и безопасное сохранение на диск с резервным логом.

2.3 Реализация и инженерные детали

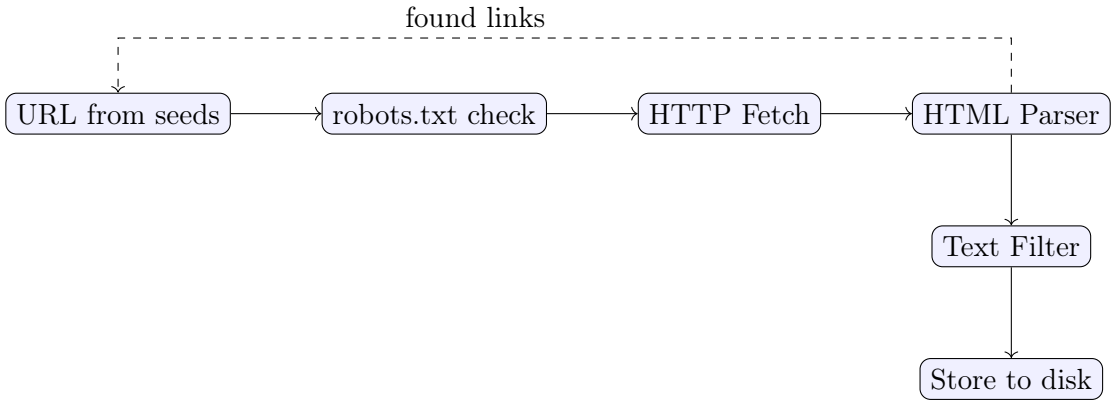
Реализация ориентирована на надёжность: для каждого домена хранится таймер последней загрузки, что позволяет держать паузу между запросами. Загрузчик использует заголовок User-Agent, идентифицирующий проект, и умеет обрабатывать редиректы и chunked-ответы. Для оптимизации при долгих запусках реализовано сохранение состояния очереди и списка посещённых URL в файл; это позволяет возобновлять работу с места остановки без повторной загрузки уже обработанных страниц. Данные о скачанных страницах и проблемах сохраняются в логах с отметкой времени и кодом ошибки.

2.4 Тестирование и контроль

При тестировании проверяется поведение робота на страницах с разными типами контента, на сайтах с ограничениями robots, а также реакции на ошибки сети. Те-

стовые сценарии включают проверку: корректности парсинга HTML, выделения текста, соблюдения задержек, корректности сохранения файлов и возможности восстановления состояния после аварии. Для контроля качества рекомендуется запускать робота сначала на небольшом наборе сайтов и ручной проверкой подтверждать корректность извлечённого текста, после чего масштабировать сбор на основной корпус.

2.5 Диаграмма потока (повторно)



3 Токенизация

3.1 Описание

Токенизация превращает текст в набор лексических токенов, на которых далее базируются стемминг и индексирование. Важной задачей здесь является корректная работа с кодировками и сохранение смысловой единицы (слова) в виде токена, пригодного для стеммера. В реализованном модуле токенизация состоит из нормализации регистра, фильтрации нежелательных символов и разбиения по пробелам. В простейшей конфигурации все небуквенно-цифровые символы заменяются на пробелы, что обеспечивает детерминированное поведение на большинстве страниц. Для специализированных корпусов можно расширять правила допустимых символов (дефисы, точки в номерах версий и т.п.), но это делается локально, не ломая общую логику. Токенизация интегрирована с логикой тестирования, где проверяется поведение на наборе контрольных строк.

3.2 Архитектура и интерфейс

Токенизатор реализован как функция с чистым интерфейсом: на вход подаётся строка в UTF-8, на выходе — динамический массив токенов в формате `SchVector<SchString>`. Такой интерфейс позволяет легко подменять внутренние правила без изменения остального кода. Важная деталь — возврат оригинальной позиции токена в тексте (offset) при необходимости отображения сниппетов на выдаче. Алгоритм работает линейно по длине текста, что обеспечивает хорошую производительность при больших документах.

3.3 Реализация и тонкая настройка

Реализация использует простые и быстрые операции над байт-строкой: проверка isalnum, tolower, запись в промежуточный буфер и последующее чтение через stringstream или ручной парсинг. Минимальная длина токена настраиваема, также есть возможность подключить стоп-лист и фильтр служебных символов. Для производительных сценариев вместо stringstream предпочтительнее использовать ручной разбор, что снижает расходы на аллокации и копирования. В проекте по умолчанию разрешено использовать std::string только во внутренней области токенизации, затем данные переводятся в `SchString` для ядра индекса.

3.4 Тестирование

Тесты токенизатора включают набор входных строк с разной пунктуацией, смешанными кодировками и техническими маркерами. Проверяется корректность количества токенов, соответствие ожидаемым лексемам и устойчивость к краевым случаям

(пустой ввод, строки с одной буквой, длинные последовательности спецсимволов). Тесты автоматизированы и включены в тестовый сценарий проекта, что упрощает контроль регрессий после изменения правил.

4 Стемминг

4.1 Описание

Стемминг уменьшает словоформы до базовой формы, что даёт преимущество в полноте поиска, объединяя формы слова в одну запись в индексе. В проекте реализована компактная версия алгоритма Портера, которая покрывает наиболее частые случаи и снабжена защитными проверками, исключающими избыточную агрегацию различных по смыслу слов. Функция стемминга детерминирована и работает линейно по длине слова, что важно при массовой обработке большого корпуса. В стеммер включены проверки measure, `contains_vowel` и шаблонов CVC, что соответствует стандартной логике Portera. Стеммер интегрирован в пайплайн: каждый токен после токенизации передаётся в стеммер, а результат идёт в индекс. Для практики важно сохранять и оригинальную форму слова, чтобы можно было при выдаче показать читабельный фрагмент.

4.2 Архитектура и шаги

Алгоритм разделён на последовательные шаги: начальные простые трансформации, затем более сложные замены суффиксов и завершающие корректировки. Каждый шаг применяет набор правил, которые выполняются только при выполнении условий по мере слова. Структурно реализация оформлена как набор вспомогательных функций (`is_consonant`, measure, `contains_vowel`, cvc) и блоков применения правил. Такой подход упрощает тестирование каждого шага и упрощает понимание логики при отладке. Для некоторых кейсов добавлены комментарии в коде с примерами вход-выход, что облегчает поддержку и развитие.

4.3 Диаграмма шагов

4.4 Тестирование и практические замечания

Стеммер покрыт широким набором тестов; в них включены как стандартные пары (running -> run), так и редкие исключения. Тесты помогают обнаружить случаи over-stemming и скорректировать правила так, чтобы минимизировать нежелательные объединения. Для многоязычности потребуется иное решение — лемматизация и POS-tagging, но для английского корпуса Porter's stemmer даёт хороший компромисс между сложностью и качеством. Рекомендуется анализировать частые стеммы и вручную проверять пары «стем -> слова», чтобы понять, нет ли систематических ошибок.

5 Закон Ципфа

5.1 Описание и цель

Проверка закона Ципфа используется как инструмент контроля качества корпуса и корректности предобработки текстов. Закон утверждает обратную зависимость частоты слова от его ранга, и проверка этой закономерности на реальном корпусе помогает выявить ошибки токенизации, чрезмерную долю стоп-слов или проблемы с нормализацией. Для этого в процессе индексации собираются частоты всех стемм, после чего данные экспортируются в CSV и визуализируются в лог-лог шкале. Анализ графика даёт информацию о поведении распределения в вершине и хвосте, что важно для принятия инженерных решений на предыдущих этапах обработки. Проверка Zipf — это не самоцель, а способ быстро увидеть структурные проблемы корпуса и скорректировать пайплайн.

5.2 Реализация экспорта

Частоты собираются в словаре при индексации; после завершения индексации все пары «терм, частота» экспортируются в CSV-файл. Экспорт выполняется в отдельной функции, которая формирует файл с заголовком и затем списком пар, упрощая последующую обработку в Python. Сортировка по убыванию частоты выполняется либо на C++ (в собственном алгоритме сортировки), либо в Python-скрипте визуализации. Экспортируемый файл затем подаётся на вход визуализатору, который генерирует PNG с лог-лог графиком.

5.3 Интерпретация результатов

На корректно обработанном корпусе ожидается приближенная линейность на лог-лог графике; отклонения в топе часто объясняются стоп-словами, а искривления в хвосте — шумом и редкими токенами. Для практического анализа полезно смотреть графики по диапазонам рангов (топ-10, топ-100, топ-1000), поскольку глобальная картина может скрывать локальные аномалии. Если наблюдается сильное отклонение, рекомендуется вернуться к токенизации/стеммингу и проанализировать наиболее частотные стеммы вручную. Также полезно строить гистограммы длин токенов и распределение частот по документам.

5.4 График

5.5 Практические рекомендации

После первичной проверки Zipf рекомендуется настроить стоп-лист и пороги частот, а также провести чистку корпуса от служебных и навигационных блоков. При итера-

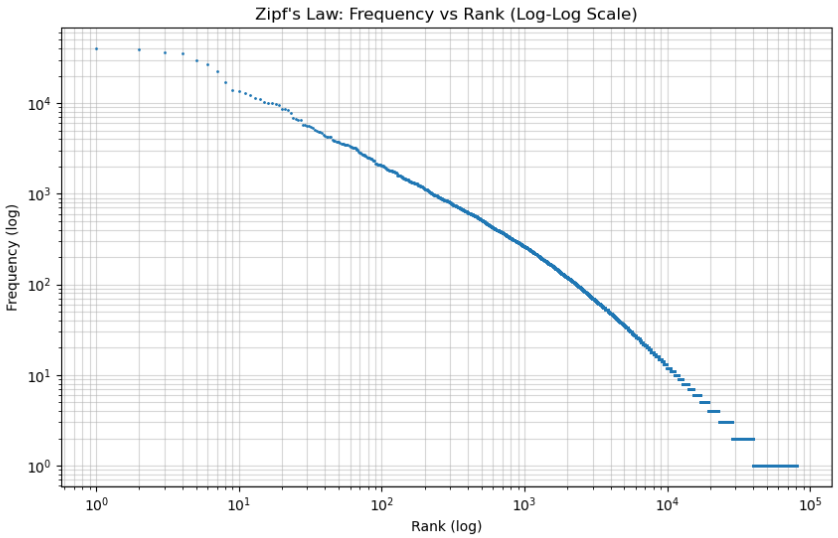


Рис. 1: Log-Log график частоты слов против ранга (Zipf).

тивной доработке пайплайна повторная проверка Zipf служит индикатором улучшений или регрессий. Этот процесс является частью контроля качества данных перед проведением полноценной оценки поисковой выдачи по qrels.

6 Булев индекс

6.1 Описание

Инвертированный индекс — это основная структура для булевого поиска: для каждого терма хранится список документов, в которых он встречается, что позволяет быстро вычислять булевы выражения. В проекте индекс реализован с нуля, без использования STL: собственная строка, динамический массив и хэш-таблица с цепочками обеспечивают полный контроль представления и сериализации. Такое решение полезно для учебной задачи, поскольку демонстрирует внутренние механизмы и даёт возможность оптимизировать конкретные узкие места. Индекс сохраняется в бинарном формате, что даёт быструю загрузку в память при запуске поисковой утилиты. Помимо базовой структуры, предусмотрены мероприятия по предотвращению дубликатов внутри posting lists и сортировка списков перед сохранением для корректной работы алгоритмов пересечения.

6.2 Структуры и реализация

Ключевой тип `SchString` инкапсулирует управление памятью строк, а `SchVector<T>` реализует динамический массив с изменяемой ёмкостью. Хэш-таблица выполнена методом цепочек, где узлы содержат C-строки ключей и значение — posting list. Posting list — это `SchVector<int>`, хранящий id документов; при добавлении мы проверяем последний элемент, чтобы не добавлять один и тот же id дважды. Сохранение индекса организовано детерминированно: сначала запись списка имён документов, затем запись термов и их posting lists. Такая схема облегчает восстановление индекса и делает формат стабильным.

6.3 Диаграмма структуры posting list

PostingList хранит отсортированные id документов; пересечение — два указателя.

6.4 Проблемы масштаба и компрессии

При больших корпусов хранение списков в виде массивов int может занимать много памяти; для реального проекта следует применить дельта-кодирование и переменную длину кодирования (variable-byte), а также skip-поинтеры для ускорения пересечений. Кроме того, имеет смысл разделять индекс на сегменты и загружать только активные части при обслуживании запросов. В учебной реализации эти оптимизации

отложены как дальнейшая работа, но архитектура рассчитана на их последующую интеграцию.

6.5 Тестирование и валидность индекса

При тестировании индекса проверяется корректность добавления документов, отсутствие дубликатов, корректность сортировки posting lists и корректная сериализация/десериализация. Также проверяется работоспособность базовых операций: получение posting list по терму, пересечение и объединение списков, и целостность данных после сохранения и повторной загрузки. Эти тесты помогают гарантировать, что индекс готов к использованию поисковой утилитой.

7 Булев поиск

7.1 Описание

Булевый поиск — это базовый режим, в котором запросы интерпретируются как булевы формулы над термами и возвращают множество документов, удовлетворяющих формуле. В проекте реализована командная утилита `search_cli`, которая загружает индекс, читает запросы со стандартного ввода и печатает найденные документы, а также лёгкий веб-интерфейс, который оборачивает CLI и отображает результаты в браузере. Такой подход даёт простоту реализации и удобство тестирования: ядро поиска остаётся на C++ для скорости, а веб-интерфейс служит фронтендом. Булевый режим удобен для формальных тестов и для случаев, когда пользователю нужно строго выразить логическую комбинацию условий.

7.2 Парсинг запросов и исполнение

Запросы разбиваются по пробелам; поддерживаются операторы `AND` и `OR`; при отсутствии оператора применяется `AND`. Каждый терм токенизируется и стеммируется, затем по стемме извлекается posting list. Для выполнения операций используются классические алгоритмы: пересечение выполняется через два указателя на отсортированные списки, объединение — через merge. Эти алгоритмы имеют линейную сложность по сумме длин списков, что делает их эффективными при умеренных размерах posting lists. Для долгих списков можно применять оптимизации: сначала выбирать операцию с наименьшим списком и итеративно пересекать с более длинными, тем самым уменьшая объём работы.

7.3 Интеграция с веб-интерфейсом и примеры

Веб-интерфейс реализован на Flask; он запускает `search_cli` как подпроцесс и передаёт запросы через stdin, затем парсит stdout и формирует HTML. Такой подход минимизирует дублирование логики и упрощает сопровождение. Ниже приведены реальные скриншоты: один показывает вывод CLI в терминале, второй — страницу веб-интерфейса с результатами. Скриншоты подтверждают, что веб-обёртка отображает те же документы, что и CLI, и обеспечивает удобный интерфейс для проверки и демонстрации.

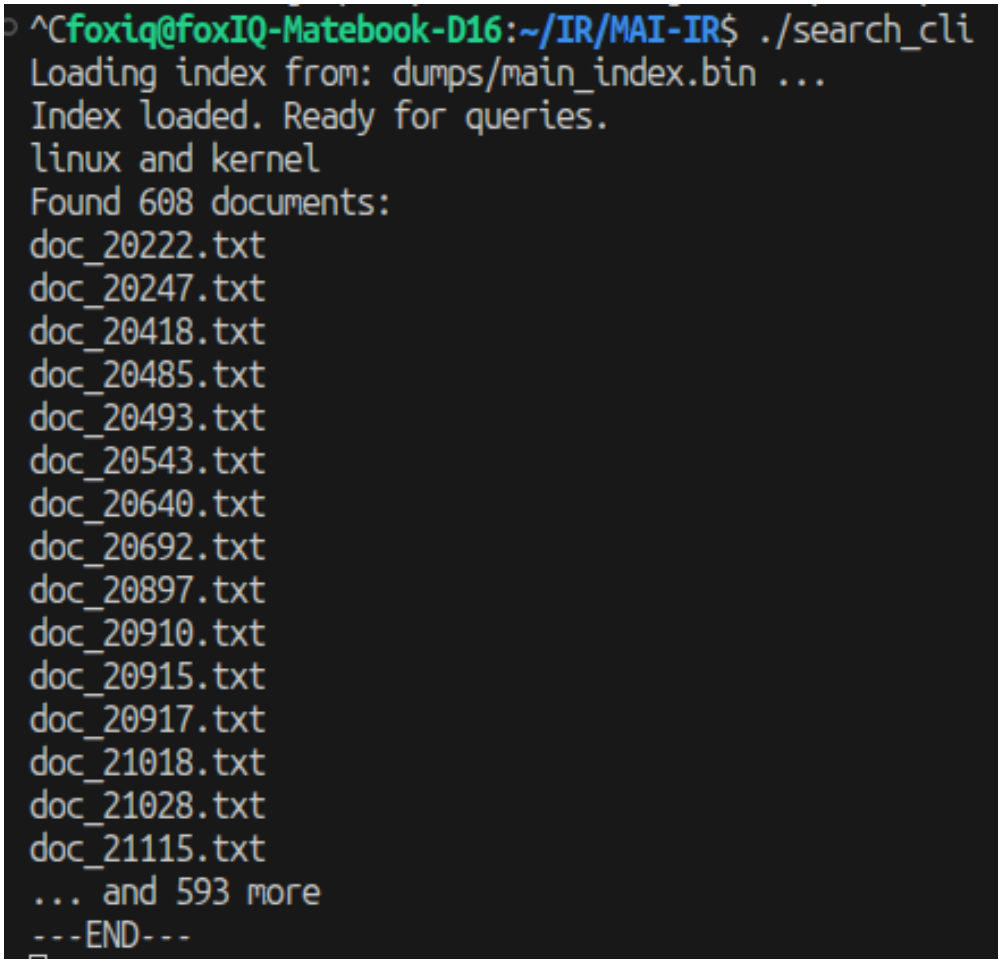


Рис. 2: Пример вывода `search_cli` в терминале.

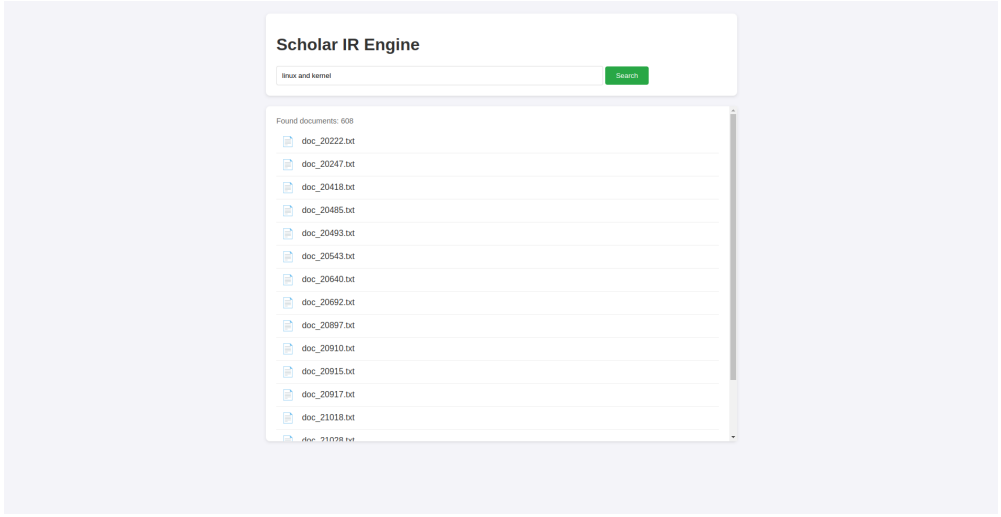


Рис. 3: Пример страницы веб-интерфейса с результатами поиска.

7.4 Ограничения и дальнейшие шаги

Булевый поиск не ранжирует результаты и возвращает их в произвольном порядке (либо в порядке id). Для практических приложений требуется ранжирование по релевантности, в частности внедрение TF–IDF или BM25, чтобы полезные документы шли первыми. Также полезно добавить поддержку фразового поиска и proximity queries. В дополнение к булевой логике в проекте подготовлены скрипты для генерации qrels и results и модуль для расчёта P и NDCG, что позволит количественно оценивать качество при вводе ранжирования.

Выводы

В ходе выполнения работы создана воспроизводимая цепочка от добычи корпуса до базового поискового сервиса. Реализация всех компонентов на собственной низкоуровневой инфраструктуре дала ценные знания о внутреннем устройстве индекса и о том, какие компромиссы приходится принимать между скоростью, памятью и качеством. Проверка закона Ципфа и визуализация распределений помогли быстро находить и устранять ошибки на этапе предобработки. В дальнейшем полезно интегрировать ранжирование BM25, добавить компрессию posting lists и развить пайплайн для масштабной индексации. Работа оставляет пространство для множества улучшений, но уже в текущем виде она обеспечивает ясную и воспроизводимую платформу для экспериментов с информационным поиском.

Литература

Список литературы

[1] C. D. Manning, P. Raghavan, H. Schütze. *Introduction to Information Retrieval.* Cambridge University Press, 2008.

[2] M. Porter. *An algorithm for suffix stripping.* Program, 14(3):130-137, 1980.

[3] G. K. Zipf. *Human Behavior and the Principle of Least Effort.* Addison-Wesley, 1949.