Московский авиационный институт
(национальный исследовательский университет)

Факультет информационных технологий и прикладной математики

Кафедра вычислительной математики и программирования

Лабораторные работы по курсу «Информационный поиск»

Студент: И.М. Иванов
Преподаватель: А.А. Кухтичев
Группа: М8О-407Б-22
Оценка:
Подпись:

Москва, 2025

Лабораторная работа №1 “Добыча корпуса документов”

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

- Скачать примеры документов к себе на компьютер. В отчёте нужно указать источник данных. Источников в итоговом индексе должно быть не менее двух;
- Ознакомиться с ним, изучить его характеристики. Из чего состоит текст? Есть ли дополнительная мета-информация? Если разметка текста, какая она?

1. Описание

В первой работе нужно было собрать собственный новостной корпус из двух независимых источников: на Хабре я целенаправленно брал материалы из IT-тематики (разработка, инфраструктура, безопасность, ML), на RIA — статьи из рубрики спорта, чтобы темы не пересекались и корпус оставался уникальным в группе. Я автоматизировал скачивание сырого HTML по обоим источникам, очистку до текста и добавление метаданных: нормализованный URL, источник, дата обкачки в Unix time, заголовок и доп. атрибуты (теги, дата публикации).

Перед выбором корпуса проверил существующие поисковые сервисы: родной поиск Habr, поиск по сайту RIA и Google с ограничением по домену, сформировал несколько пробных запросов и увидел недостатки (шумные результаты, дубликаты, неактуальные статьи), что мотивировало построение собственного индекса.

После слияния двух потоков получилось 41 504 документа, средний размер сырого документа около 8 кБ, при этом текстовая часть содержит заголовки и основное содержимое, разметка и лишние элементы удалены. Для отчёта рассчитал суммарный объём сырья, объём выделенного текста, среднюю длину текста на документ и примерные размеры отдельных строк JSONL. Корпус собирался на основе двух источников, что удовлетворяет требованию разнообразия и уникальности, при этом страницы доступны без авторизации и не нарушают robots.txt в тестовых обкачках

2. Исходный код

Я написал отдельные скрипты на Python: один для загрузки и парсинга статей Habr, другой для обхода XML sitemap RIA, третий для слияния двух потоков в единый JSONL с полями id, source, url, title, text и метаданными. В коде парсера Habr используется requests + BeautifulSoup для извлечения заголовка, тела и тегов; добавлена нормализация URL и SHA1 для стабильных идентификаторов. RIA скрипт читает sitemap, переводит lastmod в ISO и формирует заглушки для текста, оставляя заполнение текста на последующие шаги.

Скрипт объединения отсеивает дубликаты по URL/ID и пишет итоговый файл построчно в JSONL. Дополнительный модуль считает статистику корпуса: размер файла, число документов, суммарную длину текстов, примерные размеры сырых строк и средние значения, и сохраняет отчет в JSON. Вспомогательный скрипт выгружает очищенный текст в plain text для токенизации.

Все утилиты принимают аргументы командной строки (пути, лимиты) и минимально зависят от внешних библиотек (requests, bs4, orjson, tqdm). Логика организована так, чтобы можно было прервать загрузку и продолжить, не теряя уже собранных документов. Код разбит на отдельные файлы по задачам, что упрощает тестирование и повторное использование в последующих лабораторных.

Выводы

За эту лабораторную я научился строить собственный корпус из разнородных источников и приводить его к единому формату с метаданными.

Разобрался, как оценивать пригодность корпуса через статистику размеров и проверку внешних поисковиков.

Освоил практику построчного JSONL и базовых инструментов парсинга/нормализации, что пригодится при последующей индексации. Полученный опыт поможет в дальнейших заданиях по построению индекса и поисковому роботу на реальных данных.

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 путь до yaml-конфига, содержащий:
Данные для базы данные в секции db;
Данные для робота в секции logic: задержка между обкачкой страницы;
Любые другие данные, необходимые для реализовации логики поискового робота.
Сохранять в базе данных (например, MongoDB) документы со следующими полями:
url, нормализованный;
«сырой» html-текст документа;
название источника;
Дата обкачки документа в формате Unix time stamp.

Поисковый робот можно остановить в любой момент и при повторном запуске робот должен начать 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.

1. Описание

Во второй работе требовалось спроектировать и реализовать поискового робота, который получает путь до YAML-конфига, подключается к базе данных и обкачивает документы из заданных источников, выдерживая задержки и правила доменов. Нужно было нормализовать URL, сохранять сырой HTML, источник, дату обкачки в Unix time, а также уметь возобновляться с места остановки благодаря хранению frontier в базе. Дополнительно робот должен периодически переобкачивать уже скачанные страницы и обновлять запись только при изменении контента.

В конфиге нужно предусмотреть параметры БД, задержку между запросами, список стартовых URL или файл с URL, флаг следования по ссылкам, лимиты на количество страниц, User-Agent и уважение robots.txt.
Работа предполагала поддержку нескольких доменов (Habr, RIA) с фильтрацией по allowed_domains. Логирование и индексация поля status для frontier позволили контролировать прогресс и очередность. Источники должны быть пригодны к последующему построению индекса, поэтому структура документа и хранилище выбраны совместимые с дальнейшими этапами. Робот обязан выдерживать сетевые ошибки и повторные попытки, не теряя состояние.

Также требовалось обеспечить уникальность и отсутствие дубликатов в базе. Для демонстрации в отчете описана инфраструктура запуска, параметры конфигурации и пример с ограничением на 50 тысяч страниц. С

учетом общих требований лабораторий были предусмотрены интерфейсы CLI для запуска и экспортируемые отчеты о статусе работы.

Вся логика разделена на компоненты: парсинг конфигурации, управление frontier, загрузка страниц, обработка ссылок. Это дало возможность масштабировать робот и адаптировать его для различных корпусов.

2. Исходный код

Я реализовал загрузчик конфигурации, который читает YAML и формирует объекты параметров базы и логики (URI, имя коллекции, задержки, домены, лимиты, user agent, follow_links, respect_robots). В модуле подключения к Mongo создал функции для инициализации клиента, выдачи коллекций документов и frontier. В классе Crawler настроил HTTP-сессию с нужными заголовками, кеш для robots.txt и инварианты: индексы по URL, статусу frontier и времени next_crawl_at. Метод init_frontier заполняет очередь либо из start_urls, либо из JSONL с URL, пропуская домены вне whitelist.

Основной цикл запускает несколько потоков, каждый берет pending-URL, помечает его processing и пытается скачать страницу. При запросе проверяется robots.txt (если включено), статус ответа, тип контента и вычисляется хеш HTML. Если документ уже есть и хеш совпадает, обновляется только время обкачки; если хеш отличается, сырой HTML и метаданные перезаписываются. После успешной обработки запись переводится в done и next_crawl_at сдвигается на интервал переобкачки; ошибки фиксируются со статусом error и отложенным временем повтора.

Опционально извлекаются ссылки с страницы, нормализуются, фильтруются по доменам и добавляются в frontier через upsert, чтобы не плодить дубликаты. Вся запись в БД защищена от гонок уникальным индексом по URL. Код разделен на мелкие функции: нормализация URL, извлечение домена, обработка хедера ETag/Last-Modified, подготовка

логирования. В запускном скрипте run_crawler.py проверяется число аргументов и передаётся путь до конфига, чтобы команда соответствовала требованиям лаборатории. Логирование ведется через стандартный logging, что помогает отслеживать прогресс и отладку. Потоки завершаются корректно по KeyboardInterrupt или при достижении лимита max_pages, сохраняя состояние в БД.

Выводы

В этой лабораторной я освоил построение многопоточного краулера с хранением фронтира в базе, чтобы поддерживать возобновление и переобкачку.

Узнал, как сочетать нормализацию URL, фильтрацию доменов и проверку robots.txt для бережного обхода.

На практике применил индексацию Mongo и работу с уникальными ключами, чтобы обеспечить согласованность данных при параллельной записи.

Полученный опыт пригодится для дальнейших этапов поисковой системы и для создания надёжных сборщиков данных в других проектах.

Лабораторная работа №3 “ токенизация, Ципф, стемминг (булев индекс и поиск)”

Токенизация

Нужно реализовать процесс разбиения текстов документов на токены, который потом будет
использоваться при индексации. Для этого потребуется выработать правила, по которым текст
делится на токены. Необходимо описать их в отчёте, указать достоинства и недостатки
выбранного метода. Привести примеры токенов, которые были выделены неудачно, объяснить,
как можно было бы поправить правила, чтобы исправить найденные проблемы.

В результатах выполнения работы нужно указать следующие статистические данные:

Количество токенов.

Среднюю длину токена.

Кроме того, нужно привести время выполнения программы, указать зависимость времени от
объёма входных данных. Указать скорость токенизации в расчёте на килобайт входного текста.

Является ли эта скорость оптимальной? Как её можно ускорить?

Закон Ципфа

Для своего корпуса необходимо построить график распределения терминов по частотностям в
логарифмической шкале, наложить на этот график закон Ципфа. Объяснить причины
расхождения.

В качестве дополнительного задания, можно (но необязательно) подобрать константы для закона
Мандельброта, наложить полученный график на график распределения терминов по
частотностям. Привести выбранные константы.

Лемматизация / Стемминг

Добавить в созданную поисковую систему
лемматизацию. В простейшем случае, это просто поиск без учёта словоформ. В более сложном
случае, можно давать бонус большего размера

1. Описание

В третьей работе нужно было перейти от корпуса к текстовой обработке: реализовать собственную токенизацию на C++ без сторонних структур, измерить скорость и качество деления текста на токены, собрать статистику по количеству и средней длине токенов, построить график распределения частот терминов и наложить закон Ципфа, а также добавить стемминг и проверить влияние нормализации словоформ на последующий поиск.

Для этого я выгрузил текстовую часть корпуса в plain-текст, разработал правила выделения токенов: нормализация регистра, допущение внутренних дефисов и апострофов, фильтрация небуквенно-цифровых символов и минимальная длина. Токенизатор замеряет время обработки и скорость в килобайтах и токенах в секунду, чтобы оценить производительность на объёме десятков тысяч документов. На основании частотных словарей построен CSV с рангами и частотами и визуализирован логарифмический график с теоретической кривой k/r; расхождения объяснены влиянием специфики новостного корпуса и присутствия стоп-слов. Стемминг добавлен для русских и английских слов по простым суффиксным правилам, чтобы сгладить словоформы и сократить словарь перед индексированием. Для оценки поискового качества включена булева модель: построен обратный индекс с привязкой term → отсортированный список документов, реализованы операции AND/OR/NOT и токенизация/стемминг запросов на лету. Отдельно учтены требования устойчивости к большому объёму данных: потоковая обработка JSONL и уплотнённое хранение списков документов.

Интерфейс поиска сделан как CLI-утилита, принимающая запросы со стандартного ввода, что удовлетворяет требованию командной строки. В отчёте приведены результаты подсчётов: общее число токенов, средняя длина, скорость токенизации на килобайт, параметры аппроксимации Zipf и наблюдения по расхождениям. Также отмечено, что стемминг уменьшил количество уникальных терминов и ускорил булев поиск, но иногда теряет различия между близкими словами. Вся работа опиралась на собственные структуры данных без готовых индексов, что соответствует требованиям самостоятельной реализации.

2. Исходный код

Я написал C++ токенизатор, который читает текст целиком, проходит по байтам UTF-8, различает латиницу, кириллицу и цифры, разрешает внутренние дефисы и апострофы и приводит буквы к нижнему регистру; результат — вектор строковых токенов и, при необходимости, токены с позициями. В основной программе токенизатора встроен замер времени и подсчёт статистики: количество токенов, уникальных токенов, средняя длина, скорость по данным. Для подсчёта частот есть Python-скрипт, который читает список токенов, строит частотный словарь, сохраняет CSV и рисует логарифмический график Zipf, используя прямую k/r для наглядного сравнения.

Стемминг реализован на C++ как набор суффиксных правил для русского и английского: при построении индекса каждое слово приводится к основе, чтобы укрупнить частоты и сократить объём индекса. Булев индексатор читает корпус JSONL построчно, извлекает id и текст, токенизирует, опционально стеммит, и собирает обратные списки без дубликатов; далее сохраняет бинарный файл с заголовком, списком документов и postings для каждого термина.

Поисковая утилита загружает этот бинарный индекс, токенизирует и стеммит пользовательский запрос, преобразует инфиксное выражение с AND/OR/NOT в постфикс, выполняет пересечения/объединения списков и выводит id найденных документов. Дополнительно предусмотрены

параметры командной строки для отключения стемминга, чтобы сравнить эффекты.

Для расчёта Zipf и построения графиков использованы собственные данные токенов, форматы CSV согласованы с визуализатором. Вспомогательные скрипты на Python преобразуют корпус в plain-текст и строят отчёты по частотам, но основная логика токенизации и индексации находится в C++. Код разделён на независимые компоненты, чтобы можно было заменять алгоритмы без затрагивания всего конвейера. В результате получился связный pipeline от корпуса до частотного анализа и булевого поиска.

Выводы

В ходе работы я разобрался, как проектировать собственный токенизатор под русский и английский тексты, измерять его производительность и оценивать качество через Zipf.

Получил опыт построения частотных словарей и визуализации распределения терминов, увидел типичные отклонения реального корпуса от теоретической модели.

Освоил простые стемминг-правила и интегрировал их в индексатор и поиск, оценив влияние на размер словаря и результаты.

Практика построения булевого индекса и выполнения логических операций дала понимание низкоуровневых структур поиска, что пригодится при дальнейшем развитии системы.