ПРАВИТЕЛЬСТВО РОССИЙСКОЙ ФЕДЕРАЦИИ
ФГАОУ ВО НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ
«ВЫСШАЯ ШКОЛА ЭКОНОМИКИ»

Факультет компьютерных наук
Образовательная программа «Прикладная математика и информатика»

Отчет о программном проекте на тему:

Многофункциональный поисковой движок для индексирования документов

Выполнил студент:

группы #БПМИ238, 2 курса
Поляков Иван Андреевич 01.04.2025
(дата)

Принял руководитель проекта:

Садуллаев Музаффар Тимурович
Приглашенный преподаватель,
Департамент больших данных и информационного поиска
Факультет компьютерных наук НИУ ВШЭ

Москва 2025

Содержание

Аннотация

Данный проект представляет собой реализацию поискового движка, разработанного на языке Golang с использованием современной архитектуры и фронтенд-фреймворка Vue.js. Основной задачей работы является обеспечение эффективного поиска по документам посредством обработки текстовых данных с применением методов стемминга. Текстовые файлы разбиваются на токены, которые предварительно нормализуются, очищаются от стоп-слов и подвергаются обработке с помощью алгоритма Портера для русского и английского языков.

Полученные токены используются для построения инвертированного индекса, что позволяет быстро сопоставлять каждому токену список документов, в которых он встречается. Для оценки релевантности результатов поиска применяется метрика TF-IDF. Ранжирование документов осуществляется по косинусному расстоянию между вектором поискового запроса и векторами документов. Важным аспектом реализации является использование конкурентных вычислений с помощью горутин, что значительно ускоряет процессы индексирования и обработки запросов.

Архитектура системы включает интеграцию серверной части, отвечающей за парсинг, индексацию и обработку поисковых запросов, с базой данных MongoDB для хранения обратного индекса, а также графическую оболочку на Vue.js для взаимодействия с пользователем. Дополнительно, протестированы ключевые модули приложения (Parser, Indexer, Search).

1 Введение

1.1 Цель

Цель работы заключается в разработке легковесного и многофункционального поискового движка для работы с различными данными.

1.2 Задачи

- Исследовательская часть
  - Изучить существующие поисковые движки
  - Изучить их устройство и используемые технологии
  - Изучить процесс токенизации, существующие opensource-аналоги
  - Изучить процесс ранжирования результатов поиска
- Программная часть
  - Разработать алгоритм токенизации, позволяющий за короткий временной промежуток обрабатывать большие объемы документов
  - Разработать алгоритм для преобработки текста, который будет включать в себя такие части как токенизация, нормализация и стемминг
  - Построить инвертированный индекс для последующего поиска
  - Реализовать хранение индекса в базе данных
  - Разработать алгоритм ранжирования результатов поиска
  - Реализовать графическую оболочку для вывода результатов поиска

1.3 Актуальность работы

На рынке представлено множество различных поисковых движков, однако большая часть отличается обширной кодовой базой и большими системными требованиями. Среди крупных игроков рынка можно выделить Google, Yandex, Baidu, Yahoo, Bing. Все эти компании разработали собственные поисковые движки, которые являются готовыми продуктами и ими ежедневно пользуются миллиарды человек. Однако их главная проблема все еще в том, что их работа требует огромных производительных мощностей. Мой движок должен стать компромиссным решением, которое будет сочетать в себе многофункциональный поиск по

нескольким типам файлов и обладать достаточной легковестностью. Таким образом можно считать, что актуальность работы продемонстрирована.

1.4 План работы

- Проанализировать существующие поисковые движки, изучить их устройство и используемые технологии
- Изучить процесс токенизации, рассмотреть существующие библиотеки
- Имплементировать алгоритм токенизации, позволяющий за короткий временной промежуток обрабатывать большие объемы документов
- Реализовать хранение индекса в базе данных, проработать ее эффективное взаимодействие с остальной программой
- Имплементировать поиск данных в индексе на основе поискового запроса пользователя
- Реализовать алгоритм ранжирования полученных результатов поиска
- Разработать графическую оболочку для просмотров результатов поиска

2 Обзор литературы

2.1 Существующие поисковые движки

Если мы говорим про крупнейшие поисковые движки, то в силу закрытого исходного кода, то мы не можем знать, какие именно алгоритмы токенизации и ранжирования они используют, однако существует масса решение с открытым исходным кодом. Среди самых используемых можно выделить Elasticsearch [4] и Solr [10]. Рассмотрим каждый из них и выделим, какие их части можно интегрировать в мой движок.

2.1.1 Elasticsearch

В первую очередь это поисковой движок для поиска по документам. Этот движок использует построение инвертированного индекса - индекса, который сопоставляет токенам соответствующие документы, в которых они встретились. Для этого используется библиотека Apache Lucene. Также в Elasticsearch может использоваться распределенный индекс - он хранится на нескольких серверах, что позволяет организовать горизонтальное масштабирование. Для эффективного поиска этот движок использует специальные метрики узлов - базы данных для хранения индекса - позволяющие распределять нагрузку по нескольким серверам.

Приведу примеры использования в различных проектах:

- Stack Overflow использует Elasticsearch как средство для полнотекстового поиска по вопросам и ответам для пользователей, а также для поиска похожих вопросов и подсказок при создании нового вопроса. С помощью Elasticsearch сервис предоставляет поиск по точному совпадению (например, поиск строки кода) и нечёткий поиск с большим количеством настроек.

- GitHub обеспечивает пользователей возможностями полнотекстового поиска и поиска по отдельным критериям среди 8 миллионов репозиториев кода благодаря Elasticsearch. Например, можно найти проект на языке Clojure, который был активен в течение последнего месяца.

2.1.2 Solr

Начиная с 2010 года Solr [10] и Lucene были объединены. Solr это Apache Lucene с дополнительным функционалом - поиском. Он использует всю ту же библиотеку Apache Lucene

для построения инвертированного индекса, но добавляет алгоритм поиска и ранжирования результатов.

Приведу примеры использования в различных проектах:

- Известное медиа-издание «The Guardian» использует Solr для обеспечения быстрого и релевантного поиска по огромному объёму новостных материалов.
- GitHub обеспечивает пользователей возможностями полнотекстового поиска и поиска по отдельным критериям среди 8 миллионов репозиториев кода благодаря Elasticsearch. Например, можно найти проект на языке Clojure, который был активен в течение последнего месяца.

2.1.3 Вывод

Из анализа существующих решений с открытым исходным кодом можно сделать вывод, что в моем поисковом движке следует также использовать инвертированный индекс, так как он позволяет осуществлять полнотекстовый поиск и сразу по токенам получать id релевантных документов. Это замедляет процесс индексирования, но кратно уменьшает время поиска.

2.2 Токенизация

Сама по себе Токенизация это процесс разбиение текста на токены. В зависимости от задачи это можно делать разными способами, например, в качестве разделителей использовать только пробелы, или же пробелы и запятые, или же токенизировать текст по слогам. В моем проекте необходимо будет не только разбивать тексты различных документов на токены, но еще и производить с ними какие-то манипуляции для эффективного индексирования.

2.2.1 Нормализация

Самый первых шаг это нормализация. В нее входит приведение всех символов к нижнему регистру и преобразование некоторых специфических букв различных языков к своим латинским аналогам: å –> a. Далее необходимо производить удаление так называемых стоп слов-слов, которые очень часто встречаются в языке, но не несут в себе какой-то смысл, в русском это местоимения, предлоги, частицы, междометия и прочее. Они будут встречаться в каждом документе и не будут при этом отличать его от других при поиске. После удаления стоп-слов следует провести либо стемминг, либо лемматизацию для уменьшения потенциального количества токенов и более качественно поиска. Разберем этот момент подробнее.

2.2.2 Стемминг

Стемминг [6] это обрезка слова для поиска его основы - она не всегда совпадает с морфологическим корнем. Основа - неизменяемая часть слова, которая выражает его. В некоторых реализациях не требуется явного использования алгоритмов машинного обучения, предполагается возможность обойтись раннее изученными алгоритмами, которые могут решить эту задачу. Также рассмотрим варианты реализации на русском и английском языках

2.2.3 Лемматизация

Лемматизация [7] [2] это процесс приведения слова к лемме - его нормальной форме. Если говорить проще - то это начальная, словарная форма слова. То есть мы не всегда обрезаем слово, а например меняем его окончание/суффикс. Безусловно, лемматизация дает более точный по смыслу результат, но ее недостатки заключаются в том, что она требует намного больше ресурсов и ее использование зачастую предполагает ML.

2.3 Индексирование

После стемминга или лемматизации наш документ превращается в набор токенов, каждый из которых был нормализован и сокращен с помощью специальных алгоритмов. Построение инвертированного индекса заключается в сопоставлении токену документов, в которых встретился этот токен. Для эффективной выдачи результатов также необходимо отсортировать документы по убыванию количества вхождения в них определенного токена, чтобы при необходимости можно было быстро искать топ-k релевантных документов по поисковому запросу.

2.4 Поиск

Существует две базовые метрики оценки релевантности документов в информационном поиске — TF-IDF [1], [12], [13] и семейство метрик BM25 [8] [5], которые по своей сути являются усовершенствованными версиями первой. Остальные алгоритмы в большинстве своем используют дополнительные механизмы для повышения точности ранжирования. Современные системы информационного поиска часто комбинируют статистические признаки, подобные тем, что используются в TF-IDF, с более сложными подходами, учитывающими контекст запроса и поведения пользователей. Рассмотрим их.

2.4.1 TF-IDF

TF-IDF - статистическая мера, используемая для оценки важности слова в контексте документа, являющегося частью коллекции документов. Вес некоторого токена пропорционален частоте употребления этого токена в документе и обратно пропорционален частоте употребления токена во всех документах коллекции. Она состоит из двух частей - TF и IDF. TF - term frequency, частота вхождения слова в документ, формула для ее расчета:

$$\mathrm{TF}(t, d) := \frac{n_t}{\sum_k n_k}$$

где n_i число вхождений токена i в документ d.

IDF - inverse document frequency, формула для ее расчета:

$$\mathrm{IDF(t, D)} = \ln\left(\frac{|D|}{|\{d_i \mid t \in d_i\}|}\right)$$

где D - набор документов, а знаменатель - мощность множества документов, в которые входит токен t.

Таким образом, $\mathrm{TF|IDF}(t, d, D) = \mathrm{TF}(t, d) \cdot \mathrm{IDF}(t, D)$. Эта метрика оценивает важность вхождения токена в документ, причем если какой-то токен встречается во всех документах, то TF-IDF для этого токена будет равна нулю.

Поиск осуществляется следуюшим образом - для поступающего поискового запроса Q формируется вектор TF-IDF, TF которого рассчитывается на основе его собственных токенов, а IDF - на основе вхождения токенов во всю коллекцию документов. Далее для всех файлов формируется матрица TF-IDF, строки которой являются векторами TF-IDF соответствующих документов, а столбцы соответсвуют токенам запроса. Документы упорядочиваются с помощью сравнения косинуса угла между вектором запроса и векторами документов.

2.4.2 BM25

BM25 - метрика, также основанная на модели «мешка слов», которая оценивает релевантность документа запросу, исходя из встречаемости токенов запроса в документе без учёта их порядка. Эта функция представляет собой семейство методов с настраиваемыми параметрами, и одна из распространенных форм определяется следующим образом:

Пусть дан запрос Q, состоящий из токенов $q_1, q_2, \ldots, q_n$. Тогда оценка релевантности до-

кумента D вычисляется по формуле:

$$\text{score}(D, Q) = \sum_{i=1}^{n} \text{IDF}(q_i) \cdot \frac{f(q_i, D) \cdot (k_1 + 1)}{f(q_i, D) + k_1 \cdot \left(1 - b + b \cdot \frac{|D|}{\text{avgdl}}\right)}$$

где:

- $f(q_i, D)$ — частота слова q_i в документе D (TF);
- $|D|$ — длина документа (количество слов);
- avgdl — средняя длина документа в коллекции;
- k_1 и b — настраиваемые коэффициенты.

TF и IDF(q_i) определяются также, как и в случае TF-IDF.

Таким образом, BM25 учитывает как частоту появления слова в документе, так и его распространённость в коллекции, корректируя оценку за счет нормировки по длине документа и введения настраиваемых параметров.

2.5 Анализ готовых решений

Существуют библиотеки для различной работы с текстом, его токенизацией, нормализацией и стеммингом. Можно выделить такие решения, как NLTK и SpaCy.

2.5.1 NLTK

NLTK - Natural Language toolkit [3], это Python библиотека для обработки естественного языка. Она продоставляет широкую функциональность, начиная от базовой токенизацией и заканчивая готовыми списками стоп слов на разных языках. Предполагается изучение некоторых алгоритмов, которые используются в этой библиотеке и их реализация на Golang.

Среди минусов можно отметить:

- NLTK не оптимизирован для высокопроизводительных систем — в задачах, где требуется быстрый анализ больших объёмов данных, он может оказаться медленным.
- Построение эффективного обратного индекса является основой любой поисковой системы. NLTK предоставляет базовые инструменты для токенизации и обработки текста, но не имеет встроенных оптимизированных решений для создания и управления поисковыми индексами.

2.5.2 SpaCy

SpaCy [9] - эта библиотека Python является аналогом NLTK, однако она быстрее справляется с некоторыми задачами в силу того, что написана на CPython. В процессе реализации алгоритмов стемминга также предполагается изучить их реализацию в этой библиотеке.

Среди минусов можно отметить:

- spaCy использует особую систему хеширования строк для экономии памяти [11]. Все строки кодируются в хеш-значения, и внутренне spaCy оперирует только этими значениями. Хотя это эффективно для экономии памяти, такая система имеет ограничения:

  1 Хеши невозможно преобразовать обратно в строки без доступа к словарю (Vocab)

  2 Необходимо обеспечить, чтобы все объекты имели доступ к одному и тому же словарю, иначе spaCy не сможет найти нужные строки

  Эти особенности могут создать проблемы при распределенном индексировании и поиске, когда разные части системы работают с разными экземплярами словаря.

3 Реализация

3.1 Исходный код

С исходным кодом проекта можно ознакомиться по ссылке: Multi-functional-Search-Engine

3.2 Структура проекта

Приведу структуру моего проекта:

```
.
|-- Dockerfile
|-- build
|-- config.yaml
|-- docker-compose.yml
|-- frontend
|   |-- README.md
|   |-- index.html
|   |-- jsconfig.json
|   |-- package-lock.json
|   |-- package.json
|   |-- public
|   |-- src
|   |   |-- App.vue
|   |   |-- assets
|   |   |-- components
|   |   |   |-- Background.vue
|   |   |   |-- FileUploadButton.vue
|   |   |   |-- SearchBar.vue
|   |   |   \-- SearchResults.vue
|   |   \-- main.js
|   |-- vite.config.js
|   \-- vue.config.js
|-- go.mod
|-- go.sum
|-- main.go
|-- pkg
|   |-- API
|   |   \-- search_API.go
|   |-- config
|   |   \-- config.go
|   |-- indexer
|   |   |-- indexer.go
|   |   \-- indexer_test.go
|   |-- models
|   |   \-- index.go
|   |-- mongodb
|   |   |-- db.go
|   |   \-- query.go
|   |-- parser
|   |   |-- file_readers.go
|   |   |-- parser.go
|   |   |-- stop_words.go
|   |   |-- tokenizer.go
|   |   |-- parser_test.go
|   |   \-- utils
|   |       |-- english_stop_words.txt
|   |       \-- russian_stop_words.txt
|   \-- search
|       |-- search.go
|       |-- search_tokenizer.go
|       |-- tf-idf.go
|       \-- search_test.go
\-- search_test
```

3.3 Архитектура проекта

Приведу архитектуру моего приложения:

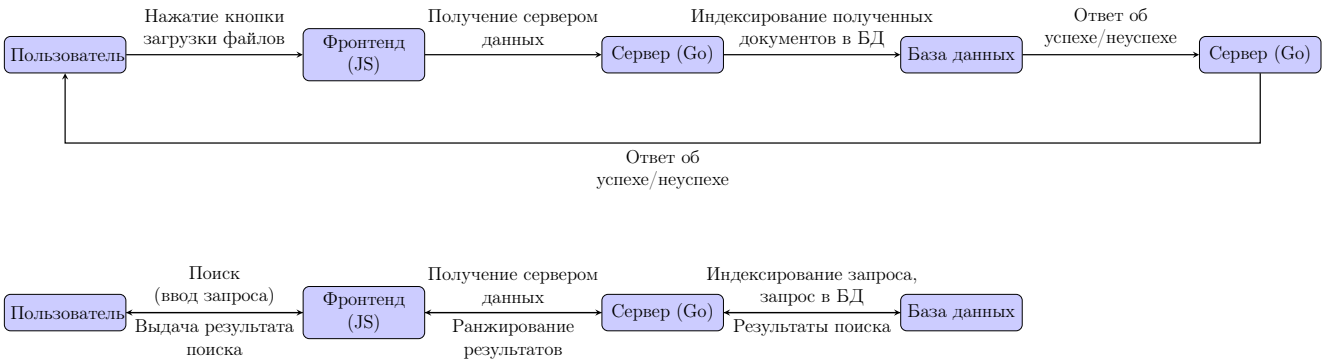

Схема архитектуры поискового движка

3.4 Parser package

Как мы уже знаем, токенизация - это процесс разбиения текста на токены. Токен же представляет собой нормализованную строку. Я опишу, как происходит разбиение на токены txt-файла. PDF документы в свою очередь обрабатываются с помощью преобразования в txt.

Основным функционалом данного пакета является чтение файла и его последующая обработка, заключающаяся в токенизации. В связи с тем, что размер файла может превышать размер оперативной памяти, необходимо читать файл поблочно, причем для повышения скорости обработки документов используются горутины - благодаря чему обработка текста происходит конкурентно, а следовательно - быстрее.

Каждая горутина обрабатывает блок файла фиклированной длины. Принято решение, что оптимальный размер блока с учетом скорости последующего стемминга составляет 2^{20} байт, что позволяет не создавать слишком много горутин, но при этом ускоряет обработку файла. После чтения определенного блока файла из памяти каждая горутина вызыват функцию токенизации.

Пошаговое описание токенизации документа

Токенизация файла происходит в несколько шагов:

- Нормализация блока текста
- Разбиение на токены

- Удаление стоп-слов
- Стемминг токенов
- Подсчет вхождения каждого токена в документ

Нормализация просходит с помощью функции ToLower пакета strings. Разбиение на токены же происходит по пробелам, запятым и всем остальным знакам препинания.

Подробно опишу, как происходит обработка наличия стоп-слов. Напомню, что стоп-слова - это такие слова, которые встречаются в большинстве документов и не несут в себе никакой дополнительной информации. Если не производить их отсеивание, то индекс становится кратно больше, нагружая всю систему, а качество поиска не улучнается. Стоп-слова хранятся в txt-файлах в специальных директориях проекта, при запуске исполняемого файла проекта происходит чтение этих файлов (Английских и Русских стоп-слов), а затем контейнер с набором этих слов сохраняется в глобальную переменную пакета для корректного конкурентного доступа из каждой горутины, которая обрабытывает блок текста.

Стемминг токенов осуществляется с помощью стороннего модуля, содержашего переписанный на Golang алгоритм Портера для Русского и Английского языков.

Теперь опишем, как происходило построение индекса каждого документа. Сначала создавался отдельный контейнер, в котором хранилось соответсвие вида (токен - количество вхождений). Далее каждая горутина, обработав свой блок текста и разбив его на токены, обращалась к специальной структуре данных и с помощью комбинации нескольких вызовов методов StoreOrLoad атомарно изменяла глобальный счетчик вхождений каждого токена в документ.

3.5 Indexer package

Индексация - процесс построения обратного индекса, то есть соответствия вида (токен - список документов, в которые входит этот токен). Это нужно для эффективного поиска документов, отвечающих поисковому запросу пользователя.

Пакет Indexer обрабатывает список файлов, он парсит каждый файл из списка а далее заносит изменения в базу данных. Индексирование также происходит конкурентно, список файлов делится на блоки фиксированного размера так, чтобы каждый блок файлов был примерно одинакового размера. Далее каждой горутине передается блок файлов и она обрабатывает его с помощью пакета Parser. То есть внутри пакета Indexer создается общий sync.Map для индекса, а далее создается структура данных, описывающая обратный индекс.

Обратный индекс формируется с помощью линейного перебора всех элементов индекса, затем документ добавляется в список соответсвующего токена.

Также в базе данных для каждого файла сохраняется его индекс - соответсвие вида (токен - количество вхождений) - для последующего использования при поиске.

3.6 MongoDB, Models, Config packages

После обработки блока файлов пакет Indexer вносит изменение в базу данных, это взаимодействие происходит благодаря пакетам MongoDB и Config. Их функционал заключается в обращении к базе данных и загрузке в нее обратного индекса. На вход пакет MongoDB принимает срез структур, описывающих обратный индекс, в них прописан сам токен и список документов, в которых содержится этот токен.

Функция, записыващая обратный индекс в базу данных, также проверяет, была ли создана коллекция, отвечающая определенному токену. Если нет, то она создает ее и с помощью работы с bson-файлами.

Для корректной работы с базой данных также предусмотрет пакет Config, который создает новый конфиг со всей нужной информацией для подключения - context, URI и прочее.

3.7 Search package

Ключевой аспект поискового движка - ранжирование результатов поиска. Данный пакет отвечает за упорядочивание документов по релевантности поисковому запросу. Приведу подробное описание реализованного алгоритма.

Пакет Search обрабатывает поисковой запрос, формируя для него вектор TF-IDF. Это происходит в несколько этапов:

- Нормализация запроса
- Разбиение на токены
- Удаление стоп слов
- Подсчет частоты употребления токена в запросе
- Подсчет обратной частоты токенов в коллекции документов

Подсчет частот происходит эффективно благодаря хранению в базе данных прямого и обратного индексов, что позволяет быстро вычислять значения метрик. Аналилогично строится матрица TF-IDF для всех документов.

Далее формируется список всех файлов коллекции. Затем происходит ранжирование результатов при помощи подсчета косинуса угла между вектором запроса и векторами документов:

$$\langle v, u \rangle_F = \sum_i v_i u_i \qquad \|v\| = \sqrt{\langle v, v \rangle_F} \qquad \cos_i = \frac{\langle \mathrm{req_{vec}}, (\mathrm{doc_{vec}})_i \rangle_F}{\|\mathrm{req_{vec}}\| \cdot \|(\mathrm{doc_{vec}})_\mathrm{i}\|}$$

Таким образом, пакет Search отвечает за упорядочивание документов по релевантности в соответсвии с поисковым запросом.

3.8 Графическая оболочка

Для написания пользовательского интерфейса был выбрал фреймворк Vue.js. Графическая оболочка позволяет загружать выбранные файлы на сервер, выполнять поисковые запросы и при необходимости скачивать файлы, являющиеся результатами поиска.

3.9 Тестирование

Для тестирования приложения были написаны тесты трех основных пакетов: Parser, Indexer и Search.

3.10 Деплой

Произведен деплой приложения с помощью веб-сервера nginx.

4 Заключение

В ходе выполнения проекта были успешно реализованы все поставленные задачи, направленные на создание высокопроизводительного поискового движка. Основные достижения и функциональные возможности приложения включают:

- Обработка и нормализация текстовых данных. Реализованы этапы токенизации, нормализации текста, удаления стоп-слов, а также стемминга с использованием алгоритма Портера для русского и английского языков.
- Построение инвертированного индекса. Разработана система индексирования, которая формирует обратный индекс, сопоставляющий каждому токену список документов, где он встречается. Такой подход обеспечивает эффективный и быстрый поиск по коллекции текстовых файлов.
- Реализация алгоритмов ранжирования. Для оценки релевантности документов к поисковому запросу была использована метрика TF-IDF. Это позволило проводить точное ранжирование результатов поиска на основе косинусного сходства между вектором запроса и векторами документов.
- Интеграция с базой данных. Приложение взаимодействует с MongoDB, где хранится обратный индекс, что обеспечивает устойчивость и масштабируемость системы.
- Разработан пользовательсткий интерфейсреализованный на Vue.js, он предоставляет графическую оболочку для загрузки файлов, выполнения поисковых запросов и получения результатов, что значительно упрощает взаимодействие с системой.
- Произведен деплой приложения с помощью nginx

Список литературы

[1] Prafulla Bafna, Dhanya Pramod и Anagha Vaidya. «Document clustering: TF-IDF approach». В: (2016). URL: https://clck.ru/3K8iF9.

[2] Balakrishnan, Vimala и Ethel Lloyd-Yemoh. «Stemming and lemmatization: A comparison of retrieval performances». В: (2014). URL: https://goo.su/kVKUZV.

[3] Bird и Steven. «NLTK: the natural language toolkit». В: (2006). URL: https://goo.su/zZUe.

[4] *Elasticsearch*. URL: https://github.com/elastic/elasticsearch (дата обр. 27.03.2025).

[5] Kadhim и Ammar Ismael. «Term weighting for feature extraction on Twitter: A comparison between BM25 and TF-IDF». В: (2019). URL: https://clck.ru/3K9MWA.

[6] Julie Beth Lovins. «Development of a Stemming Algorithm». В: (1968). URL: http://chuvyr.ru/MT-1968-Lovins.pdf.

[7] Plisson и др. «A rule based approach to word lemmatization». В: (2004). URL: https://clck.ru/3K9MdR.

[8] Robertson, Stephen и Hugo Zaragoza. «The probabilistic relevance framework: BM25 and beyond». В: (2009). URL: https://clck.ru/3K9MCh.

[9] Schmitt и др. «A replicable comparison study of NER software: StanfordNLP, NLTK, OpenNLP, SpaCy, Gate». В: (2019). URL: https://goo.su/gCuuA.

[10] *Solr*. URL: https://github.com/apache/solr (дата обр. 27.03.2025).

[11] *spaCy*. URL: https://spacy.io/usage/spacy-101 (дата обр. 27.03.2025).

[12] Михайлов Дмитрий Владимирович, Козлов Александр Павлович и Емельянов Геннадий Мартинович. «Выделение знаний и языковых форм их выражения на множестве тематических текстов: подход на основе меры tf-idf». В: (2015). URL: https://clck.ru/3K8Tnt.

[13] Попова Светлана Владимировна и Данилова Вера Владимировна. «Представление документов в задаче кластеризации аннотаций научных текстов». В: (2014). URL: https://clck.ru/3K8QEA.