

МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ
РОССИЙСКОЙ ФЕДЕРАЦИИ

Федеральное государственное бюджетное образовательное учреждение высшего образования

«ПЕНЗЕНСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ»

Факультет

вычислительной техники

Кафедра

«Вычислительная техника»

Направление подготовки: 09.04.01 «Информатика и вычислительная техника»

Магистерская программа: «Прикладной искусственный интеллект»

МАГИСТЕРСКАЯ ДИССЕРТАЦИЯ

на тему

ПРИМЕНЕНИЕ МЕТОДОВ МАШИННОГО ОБУЧЕНИЯ
ДЛЯ АВТОМАТИЧЕСКОГО ИЗВЛЕЧЕНИЯ
КЛЮЧЕВЫХ СЛОВ ИЗ ВЕБ-СТРАНИЦ

Студент

Споршев Н.О. 17.06.24 Споршев Сергей Александрович
(подпись, дата) (ФИО полностью)

Руководитель

Бикиташев Р.А. 17.06.24 Бикиташев Р.А.
(подпись, дата) (фамилия, инициалы)

Нормоконтролёр

Бикиташев А.С. 24.06.24 Бикиташев А.С.
(подпись, дата) (фамилия, инициалы)

Рецензент

Васекаев А.В. 25.06.24 25.06.24
(должность, место работы) (подпись, дата) (фамилия, инициалы)

Работа допущена к защите (протокол зас. кафедры от 27.05.24 № 11)

Заведующий кафедрой

М.А. Митрохин
(подпись)

М.А. Митрохин
(фамилия, инициалы)

Работа защищена с отметкой отлично (протокол заседания ГЭК от

25.06.2024 № 38)

Секретарь ГЭК

Е.И. Гурин
(подпись)

Е.И. Гурин
(фамилия, инициалы)

Пенза 2024

Федеральное государственное бюджетное образовательное учреждение
высшего образования
«ПЕНЗЕНСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ»

Факультет

вычислительной техники

Кафедра

«Вычислительная техника»

«Утверждаю»

Заведующий кафедрой

М.А. Митрохин

« 16 » 04 20 24 г.

**ЗАДАНИЕ
НА МАГИСТЕРСКУЮ ДИССЕРТАЦИЮ**

1. Студент _____ гр. 22ВВИм2 факультета ВТ

Споршев Сергей Александрович

2. Тема работы Применение методов машинного обучения для автоматического
извлечения ключевых слов из веб-страниц

Тема утверждена приказом ПГУ № 383/о от " 16 " апреля 20 24 г.

3. Руководитель магистерской диссертации Бикитов Равиль Айдарович

4. Задание на работу (назначение разработки, исходные данные и
т.п.)

Разработать программное решение для автоматического извлечения ключевых слов из веб-страниц с применением алгоритмов машинного обучения. Исходные данные собираются в процессе работы и могут включать HTML документы и целевые ключевые слова.

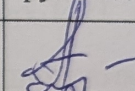
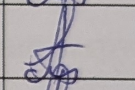
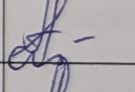
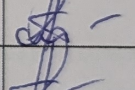
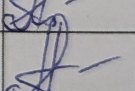
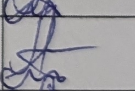
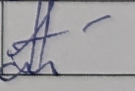
Назначение разработки: Создание системы, способной автоматически извлекать ключевые слова из веб-страниц с использованием различных методов анализа текста. Разработка предназначена для улучшения поисковой оптимизации (SEO). В рамках разработки провести исследование существующих методов извлечения ключевых слов, их применимость к веб-контенту, особенности структуры веб-страниц.

Проектирование архитектуры программного решения выполнить с соблюдением принципов масштабируемости, устойчивости, модульности.

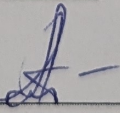
5. Перечень подлежащих разработке вопросов

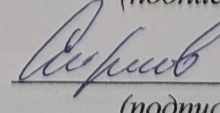
- а) Анализ существующих методов и алгоритмов извлечения ключевых слов из текста
- б) Исследование особенностей веб-контента и их влияния на процесс выделения ключевых слов
- в) Проектирование архитектуры программного решения для автоматического извлечения ключевых слов
- г) Разработка программного решения для автоматического извлечения ключевых слов
- д) Оценка эффективности предложенных методов на реальных данных

6. Календарный график выполнения работы

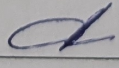
№ п/п	Наименование этапов работы	Объем работы	Срок выполнения	Подпись руководителя
1.	Анализ предметной области семантики веб-страниц	10%	16.04.2024 – 20.04.2024	
2.	Анализ предметной области методов выделения ключевых слов	10%	20.04.2024 – 24.04.2024	
3.	Формирование массива данных для обучения и тестирования модели	10%	24.04.2024 – 28.04.2024	
4.	Проектирование архитектуры системы выделения ключевых слов	10%	29.04.2024 – 03.05.2024	
5.	Разработка всех компонентов программной системы	30%	03.05.2024 – 16.05.2024	
6.	Отладка разработанной системы машинного обучения	10%	16.05.2024 – 21.05.2024	
7.	Проведение экспериментов и анализ результатов	10%	22.05.2024 – 27.05.2024	
8.	Оформление выпускной работы	10%	27.05.2024 – 01.06.2024	

Дата выдачи задания " 16 " 04 20 24 г.

Руководитель магистерской диссертации  16.04.24
(подпись, дата)

Задание к исполнению принял студент  18.04.24
(подпись, дата)

Работу к защите допустить
Декан факультета

 А.Р. Рязанов
24.06.2024 (подпись, дата)

РЕФЕРАТ

Магистерская работа содержит 136 с., 3 рис., 10 табл., 42 источн., 13 прил.

МАШИННОЕ ОБУЧЕНИЕ, КЛЮЧЕВЫЕ СЛОВА, СЕМАНТИКА, ВЕБ-СТРАНИЦА, HTML

Целью работы является разработка средства извлечения ключевых слов из веб-страниц с применением алгоритмов машинного обучения.

Объектом разработки является программное решение, предназначенное для автоматического извлечения ключевых слов из веб-страниц с использованием различных методов анализа текста.

В результате проделанной работы проведено исследование предметной области, проектирование архитектуры программного решения, позволяющего пользователю получить список ключевых слов, соответствующих URL сайта или HTML документу, разработаны алгоритмы работы программного решения. В рамках программного решения реализованы сценарии загрузки и разбиения HTML-документов, управление базой данных, выделение ключевых слов и стратегии их вывода.

Результаты данной работы могут быть применены в области поисковой оптимизации (SEO), контент-менеджмента, автоматического аннотирования документов, создания систем рекомендаций и других задач, связанных с обработкой и анализом текстовых данных.

					ПГУ1.090401.07.001 ПЗ			
Изм.	Лист	№ докум.	Подпись	Дата	Применение методов машинного обучения для автоматического извлечения ключевых слов из веб-страниц Пояснительная записка	Лит.	Лист	Листов
Разраб.		Споршев С.А.		24.06.24				
Провер.		Бикташев Р.А.		24.06.24			2	136
Н. Контр		Бычков А.С.		24.06.24		ФВТ, каф. ВТ		
Утверд.		Митрохин М.А.		24.06.24		гр. 22ВВИМ2		

СОДЕРЖАНИЕ

ПЕРЕЧЕНЬ СОКРАЩЕНИЙ И ОБОЗНАЧЕНИЙ	6
ВВЕДЕНИЕ	7
1 Анализ предметной области.....	9
1.1 Введение в семантику веб-страниц.....	9
1.2 Обзор существующих решений.....	11
1.3 Цель и задачи исследования	12
1.4 Тематики исследования.....	12
1.5 Методы машинного обучения, применяемые в задачах подобного рода.....	14
1.6 Частотный метод извлечения ключевых слов	16
1.7 Метод извлечения ключевых слов на основе TF-IDF	18
1.8 Метод извлечения ключевых слов на основе TextRank.....	21
1.9 Метод извлечения ключевых слов на основе FastText и кластеризации DBSCAN.....	23
1.10 Метод извлечения ключевых слов с использованием TextRazor	25
2 Сбор данных	27
2.1 Методы и средства формирования данных	27
2.2 Получение множества URL для обработки.....	28
2.3 Выбор СУБД.....	28
2.4 Разработка таблиц в базе данных	29
3 Архитектура решения для извлечения ключевых слов из HTML документа.....	33
3.1 Пользовательский интерфейс	33
3.2 Модуль загрузки HTML	34
3.3 Модуль обработки HTML	35
3.4 Модуль выделения ключевых слов.....	35
3.5 Взаимодействие между модулями	37
4 Алгоритм работы приложения для извлечения ключевых слов	38
4.1 Сценарий взаимодействия с пользователем.....	38

4.2	Описание пользовательского интерфейса	39
4.3	Этап загрузки начальных данных	40
4.4	Этап чтения и обработки HTML-кода	41
4.5	Этап выделения ключевых слов	45
4.6	Алгоритм сбора данных для обучения и тестирования модели.....	48
5	Реализация программного решения	49
5.1	Файловая структура проекта.....	49
5.2	Сценарий загрузки HTML-документов	51
5.3	Сценарий разбиения HTML документа	52
5.4	Сценарий выделения ключевых слов из HTML документа	53
5.5	Управление базой данных.....	54
5.5.1	Интерфейс DatabaseManager.....	54
5.5.2	Класс SQLiteDatabaseManager	55
5.6	Входная и выходная обработка	56
5.6.1	Входные обработчики данных	56
5.6.2	Выходные обработчики данных.....	57
5.7	Обработка данных.....	57
5.7.1	Интерфейс DataProcessor.....	57
5.7.2	ElementWeightAssigner	58
5.7.3	HTMLDataProcessor	58
5.8	Выделение ключевых слов.....	59
5.8.1	Интерфейс KeywordExtractor.....	59
5.8.2	SimpleKeywordExtractor	61
5.8.3	TextRazorKeywordExtractor	62
5.8.4	KeywordExtractorWithDB	63
5.8.5	TFIDFKeywordExtractor.....	63
5.8.6	TextRankKeywordExtractor	64
5.8.7	FastTextExtractor.....	66
5.9	Конвейеры.....	67
5.9.1	Определение понятия	67

5.9.2 Интерфейс Pipeline.....	68
5.9.3 CommonPipeline.....	69
5.9.4 CommonFilePipeline	69
5.9.5 DatabaseKeywordsPipeline	70
5.9.6 TextRazorPipeline.....	70
5.10 API и веб-интерфейс	71
6 Оценка качества результатов.....	75
6.1 Интерфейс KeywordComparator.....	75
6.2 FastTextComparator.....	76
6.3 Сценарий количественной оценки методов	77
6.4 Экспериментальная оценка алгоритмов	78
ЗАКЛЮЧЕНИЕ	82
СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ	84
ПРИЛОЖЕНИЕ А Листинг программы	88
ПРИЛОЖЕНИЕ А.1 Сценарий загрузки документов	89
ПРИЛОЖЕНИЕ А.2 Сценарий разбиения HTML документов	90
ПРИЛОЖЕНИЕ А.3 Сценарий выделения ключевых слов.....	91
ПРИЛОЖЕНИЕ А.4 Класс управления базой данных.....	92
ПРИЛОЖЕНИЕ А.5 Классы входной обработки данных	95
ПРИЛОЖЕНИЕ А.6 Классы выходной обработки данных.....	97
ПРИЛОЖЕНИЕ А.7 Классы обработки документов	99
ПРИЛОЖЕНИЕ А.8 Классы выделения ключевых слов.....	102
ПРИЛОЖЕНИЕ А.9 Классы конвейеров.....	110
ПРИЛОЖЕНИЕ А.10 Пользовательский интерфейс	112
ПРИЛОЖЕНИЕ А.11 Сценарий сравнения ключевых слов	119
ПРИЛОЖЕНИЕ Б UML – диаграмма.....	123
ПРИЛОЖЕНИЕ В Презентация.....	125

ПЕРЕЧЕНЬ СОКРАЩЕНИЙ И ОБОЗНАЧЕНИЙ

В данной работе приняты следующие сокращения и обозначения

ACID (Atomicity, Consistency, Isolation, Durability) – атомарность, консистентность, изолированность и устойчивость, набор требований к транзакционной системе

API (Application Programming Interface) – программный интерфейс приложения

CSS (Cascading Style Sheets) – каскадные таблицы стилей

DBSCAN (Density-Based Spatial Clustering of Applications with Noise) – алгоритм кластеризации, основанный на плотности данных

DMOZ (Directory Mozilla) – многоязычный свободный каталог сайтов

GloVe (Global Vectors for Word Representation) – глобальные векторы для представления слов

HTML(5) (Hyper Text Markup Language) – язык гипертекстовой разметки

HTTP/HTTPS (HyperText Transfer Protocol (Secure)) – протокол для передачи гипертекстовых документов

IDF (Inverse Document Frequency) – обратная частота документа

MDN (Mozilla Developer Network) – ресурс для веб-разработчиков, предоставляющий документацию по веб-технологиям

NLP (Natural Language Processing) – обработка естественного языка

NLTK (Natural Language Toolkit) – набор инструментов для обработки естественного языка

ODP (Open Directory Project) – проект «Открытая директория»

SEO (Search Engine Optimization) – оптимизация в поисковой системе

TF (Term Frequency) – частота появления термина в документе

TREC (Text REtrieval Conference) – Ежегодная конференция по вопросам текстового поиска

URL (Uniform Resource Locator) – унифицированный указатель ресурса

XML (Extensible Markup Language) – расширяемый язык разметки

YFCC100M (Yahoo Flickr Creative Commons 100 Million) – публично доступный набор данных для исследований в области мультимедиа

ВВЕДЕНИЕ

В современном интернет-пространстве важность эффективного извлечения ключевой информации из большого объема данных не перестает быть актуальной. Объем данных и количество сайтов в интернете постоянно растет, автоматическое извлечение семантической информации из веб-страниц – это задача, которая постоянно стоит перед разработчиками и является быстро развивающейся областью знаний. Современные алгоритмы и технологии машинного обучения, в частности модели обработки естественного языка (NLP), и генеративные модели, такие как ChatGPT, непрерывно совершенствуются для обеспечения более точного и эффективного анализа огромных массивов данных. Это позволяет улучшать качество поиска, персонализированных рекомендаций и других приложений, зависящих от понимания и интерпретации семантического содержания исходного кода сайтов. Методы машинного обучения предоставляют мощные инструменты для решения подобных задач, позволяя создавать интеллектуальные системы, способные адаптироваться к изменяющейся структуре и содержанию веб-ресурсов.

Цель данной магистерской диссертации заключается в исследовании и разработке методов машинного обучения для автоматического извлечения ключевых слов из веб-страниц. Среди объектов исследования внимание уделяется не только способам повышения точности извлечения ключевых слов, но и особенностям формата данных в веб-документах, таким как неоднородность структуры страниц, наличие разнообразных форматов информации, а также возможности динамических изменений в содержании.

В ходе работы осуществляется анализ существующих инструментов выявления ключевых фраз, их применимость к веб-контенту и выявление возможных ограничений. На основе этого анализа разрабатываются новые подходы к семантической обработке документов, учитывающие особенности интернет-ресурсов. Результаты исследования могут иметь практическую значимость для построения более эффективных систем автоматического

извлечения ключевой информации из интернет-ресурсов, что, в свою очередь, способствует улучшению качества информационного поиска и анализа в сети Интернет.

Предмет исследования: извлечение ключевых слов из веб-страниц.

Объект исследования: методы машинного обучения, применяемые для автоматического извлечения ключевых слов из веб-страниц.

В рамках теоретического исследования и практического проектирования поставим следующие задачи:

- анализ существующих методов и алгоритмов поиска значимых фраз в тексте,
- исследование особенностей веб-контента и их влияния на процесс выделения ключевых слов,
- проектирование архитектуры и разработка программного решения для автоматического определения центральных терминов,
- оценка эффективности предложенных методов на реальных данных.

1 Анализ предметной области

1.1 Введение в семантику веб-страниц

Веб-страницы представляют собой многофункциональные документы, предназначенные для отображения информации в браузерах пользователя и создания интерактивного интерфейса.

Основным “строительным материалом” для сайтов являются HTML (HyperText Markup Language) документы. HTML выступает в роли языка разметки, позволяя определить структуру и визуальное оформление веб-содержимого [1]. С его помощью определяются компоненты страниц, такие как заголовки, параграфы, изображения, ссылки и многие другие, что обеспечивает понятность и удобство восприятия информации [2].

Семантика играет важную роль, добавляя смысл к разметке и делая ее более понятной для поисковых систем и других автоматизированных инструментов. Внедрение семантических практик, таких как применение семантических тэгов `<header>`, `<nav>`, `<article>`, `<footer>` и другие, помогает не только оформить страницу визуально, но и ясно указать ее структуру и содержание [3]. Для HTML-документа нормальной ситуацией является низкая доля значимого содержимого в общем объеме документа, гораздо больший объем могут занимать скрипты и стили, если они размещаются в документе, SVG-иллюстрации, структуры, выполняющие роль в вёрстке и так далее.

Проиллюстрируем это, исследовав главную страницу сайта Пензенского государственного университета <https://pnzgu.ru/>. Доля значимого содержимого здесь составляет 10% - 8536 из 85566 символов [4]. И это при том, что в структуре этого документа размещено сравнительно мало скриптов и стилей.

На рисунке 1 данные участки текста выделены зеленым. Выделение значимого содержимого из HTML просто необходимо – если применять традиционные средства обработки текста, они покажут плохой результат из-за загруженности документа техническими конструкциями, присущими HTML-документам.

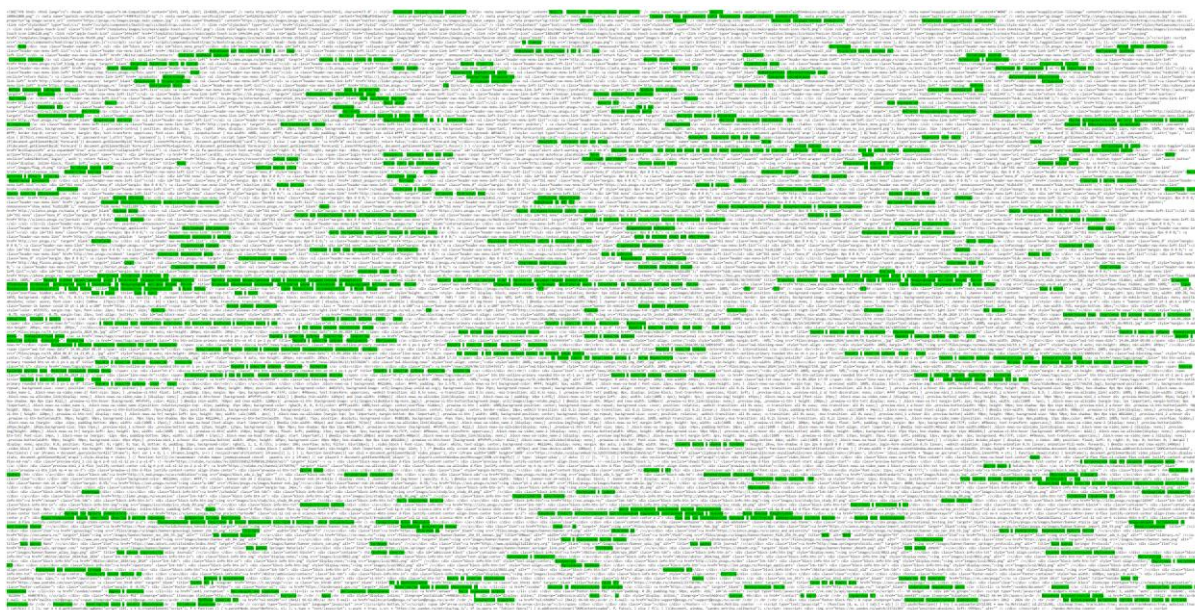


Рисунок 1 – Значимое содержимое в структуре документа

Одним из важных аспектов разработки и обслуживания сайтов является оптимизация для поисковых систем (SEO), которая направлена на улучшение видимости сайта в результатах поиска. Эксперты в SEO сходятся на примерно одинаковом наборе рекомендаций для улучшения результатов индексации, и одним из важнейших пунктов является правильное использование HTML-тегов [5]. В отличие от скриптов JavaScript и стилей CSS, HTML-документы играют главную роль именно в передаче информации, что важно для поисковых систем [6]. Это особенно актуально для бизнеса, ориентированного на интернет-аудиторию.

Модели машинного обучения, такие как BERT и GPT, ознаменовали собой прорыв в обработке естественного языка, демонстрируя способности, свойственные человеческому мышлению. Исследования показывают, что GPT, созданная OpenAI, является одной из самых популярных моделей для генерации текстов, хоть и качество и информативность контента иногда оставляют желать лучшего [7]. Однако применение машинного обучения для извлечения ключевых фраз с веб-страниц может стать очень полезным инструментом для владельцев сайтов, стремящихся продвинуть свой сайт в поисковой выдаче.

Машинное обучение имеет несколько преимуществ перед традиционными алгоритмическими методами. Во-первых, оно может обрабатывать большие объемы данных и адаптироваться к новому контенту без участия человека. Во-вторых, модели могут учитывать плавающие особенности веб-контента, включая неструктурированные страницы, ошибки или свободный стиль разработчиков. В-третьих, обучение на больших наборах данных повышает точность извлечения ключевых слов [8].

И, наконец, новая программа с использованием МО может стать бесплатным и доступным решением для владельцев сайтов. Существующие аналоги инструментов подбора маркерных терминов практически все недоступны для небольших компаний или частных лиц ввиду высокой стоимости, особенно для корпоративных клиентов, а то могут и вообще быть заблокированными в регионе пользователя. Разработка новой открытой программы, использующей машинное обучение, может быть использовано любым владельцем сайта для улучшения своего SEO.

В данном исследовании изучается вопрос применения методов машинного обучения для автоматического извлечения ключевых слов из кода сайтов. Этот анализ охватывает не только технические аспекты формата HTML и семантики, но и внедряет машинное обучение в инструменты анализа HTML-документов с целью улучшить инструменты SEO аудита, обеспечивая более эффективное восприятие контента как человеком, так и поисковыми системами.

1.2 Обзор существующих решений

Рассмотрим список наиболее распространенных инструментов, которыми пользуются SEO- и SEM-специалисты [9].

Google Keyword Planner – бесплатный инструмент от Google Ads, предназначенный для исследования ключевых слов и оценки объема поискового трафика [10].

Wordstat Yandex –сервис для оценки пользовательского интереса к конкретным тематикам и семантического анализа [11].

Semrush – инструмент для анализа ключевых слов, конкурентов и оценки органического поиска. Позволяет выявить топики, по которым сайт ранжируется [12].

Ahrefs – популярный инструмент для исследования ключевых фраз, анализа обратных ссылок, аудита сайта и многого другого [13].

Moz – инструменты для анализа центральных тематик, отслеживания рейтинга, анализа обратных ссылок и других аспектов SEO [14].

TextRazor – API для обработки естественного языка, который можно использовать для извлечения ключевых слов из текста [15].

WebCEO – платформа для SEO-анализа, которая включает в себя инструменты для исследования соответствия по тематикам, анализа конкурентов и многое другое [16].

1.3 Цель и задачи исследования

Основная цель работы – разработка средства извлечения ключевых слов из веб-страниц с применением алгоритмов машинного обучения.

В рамках данной цели будут выполнены следующие задачи:

- обзор предметной области, включая семантический анализ простого текста и HTML-документов,
- анализ существующих способов извлечения ключевых слов и выявление их недостатков,
- разработка и реализация нескольких практик машинного обучения для автоматического выделения центральных понятий из веб-страниц,
- проведение экспериментального сравнения различных методов с использованием реальных данных, полученных из Интернета,
- оценка эффективности на основе метрик качества извлечения центральных токенов.

1.4 Тематики исследования

На начальном этапе исследования можно сформулировать следующие вопросы для проработки в ходе последующей работы:

- обзор традиционных путей и последних тенденций в области текстового анализа, включая статистические, лингвистические характеристики и новейшие исследования в области обработки естественного языка.

- исследование применения методов машинного обучения, таких как алгоритмы классификации, кластеризации и глубокого обучения, применяемых в таких задачах. Описание и сравнение эффективности и применимости этих способов.

- выбор и подготовка данных, разработка методологии для сбора и подготовки данных, необходимых для обучения и тестирования моделей. Анализ источников данных, включая ресурсы разных типов, исследование проблем, связанных с разнообразием контента,

- оценка качества извлечения ключевых слов, разработка метрик и критериев для оценки точности и полноты результата. Сравнение результатов работы различных моделей машинного обучения и оценка возможности улучшений в работе системы,

- семантика HTML документа, исследование правил семантического анализа HTML5 – оценка роли семантических атрибутов, приоритетов в структуре страниц, использование полученных сведений в процессе подготовки данных для анализа,

- учет контекста и семантики, рассмотрение методов, которые учитывают контекст и семантику текста в процессе извлечения ключевых слов. Это может включать в себя использование векторных представлений токенов, а также контекстуальных трансформерных моделей, таких как BERT [17], ChatGPT и другие. BERT основывается на недавних работах по предварительному обучению контекстуальных представлений – включая полу-контролируемое обучение последовательностей, генеративное предварительное обучение, ELMo и ULMFit. Однако, в отличие от этих предыдущих моделей, BERT является первой глубоко двунаправленной,

неконтролируемой языковой моделью, предварительно обученной с использованием только простого текстового корпуса [18],

- исследование возможностей применения разработанных методов в реальных сценариях, таких как поиск информации в поисковых системах, автоматическая категоризация контента и улучшение пользовательских интерфейсов веб-платформ,

- вопрос сайтов с динамической генерацией содержимого и многостраничных сайтов и разработка возможных практических решений,

- вопрос анализа сайтов без использования семантических тэгов и с ошибками в семантике,

- разработка программного решения, основанного на разработанных методах, которое может быть адаптировано под использование в различных условиях – как скрипт или приложение с пользовательским интерфейсом,

- рассмотрение текущих ограничений предложенных решений и предложение путей исследования для их устранения. Обсуждение перспектив развития данной области и возможных направлений для будущих исследований.

Машинное обучение позволяет автоматизированно обрабатывать и интерпретировать информацию, содержащуюся в HTML документах, что полезно в ряде задач обработки текста, в частности, извлечения ключевых слов, категоризации контента, анализа тематики документов, краткого изложения и так далее [8].

1.5 Методы машинного обучения, применяемые в задачах подобного рода

Для извлечения ключевых слов из документа можно использовать различные методы машинного анализа, успешно применяемые в других задачах семантического анализа текста. Рассмотрим примеры:

- TF-IDF (Term Frequency-Inverse Document Frequency): Этот вариант разметки текста оценивает важность термина в документе относительно корпуса текстов [19]. Высокие значения TF-IDF свидетельствуют о том, что

термин часто встречается в данном документе, но редко встречается в других документах. Применение TF-IDF к текстовому содержимому HTML-документа позволяет выделить ключевые термины, но требует корпуса текстов либо многостраничной обработки сайта,

- модели векторного представления слов (Word Embeddings): Word2Vec, GloVe, FastText и другие аналогичные модели позволяют представить лексемы в виде векторов в многомерном пространстве [19]. Применение этих методов к текстовой информации из HTML-документа позволяет в выделении семантически связанных основных понятий,

- обучение с учителем применимо для построения модели выделения наиболее репрезентативных фраз, но потребует сбора большого объема данных,

- алгоритмы обработки естественного языка (Natural Language Processing, NLP): NLP-практики, такие как выделение именованных сущностей (Named Entity Recognition) или анализ синтаксиса, могут помочь выявить важные термины и фразы в тексте HTML [20].

Учет семантических тегов, таких как <title>, <meta>, <h1> и других, позволяет более точно определить ключевые слова, предоставляя контекст и структуру страницы [21].

В анализе документов поисковые системы оценивают, находятся ли поисковые термины в важных областях документа – заголовке, метаданных, тегах заголовков и основном тексте. Они также пытаются автоматически оценить качество документа на основе анализа документа, а также множества других факторов [22].

Опиаться только на анализ документов недостаточно для современных поисковых систем, поэтому они также учитывают семантическую связанность. Семантическая связанность относится к словам или фразам, которые обычно ассоциируются друг с другом. Поисковые системы активно создают собственный тезаурус и словарь, чтобы помочь им определить, как определенные термины и темы связаны. Просто сканируя свои огромные базы

данных контента в Интернете, они могут использовать теорию нечетких множеств и определенные уравнения [22].

Эффективный метод часто включает в себя комбинацию различных подходов и признаков, таких как частота встречаемости токенов, их семантическая близость, структура HTML и другие.

1.6 Частотный метод извлечения ключевых слов

Частотный метод извлечения основывается на допущении, что чем чаще слово встречается в тексте, тем выше вероятность, что оно является ключевым. Сначала текст разбивают на отдельные единицы, или токены.

Затем все результаты приводят к нижнему регистру и удаляют пунктуацию, чтобы избежать дублирования. Так, "обучение" и "Обучение" будут рассматриваться как один токен. После этого из анализа исключают часто встречающиеся и малоинформативные лексемы, такие как "и", "в", "на", "что". Это позволяет сосредоточиться на значимых фразах [23].

Далее подсчитывают количество вхождений каждого оставшегося токена в текст. Затем они сортируются по частоте появления в тексте: чем чаще слово встречается, тем выше оно в списке.

Наконец, из отсортированного списка выбираются значения с наибольшей частотой. Это может быть фиксированное количество, например, топ-10, или определяться пороговым значением частоты.

Итого, последовательность шагов алгоритма:

- а) токенизация,
- б) нормализация,
- в) удаление стоп-слов,
- г) подсчет частотности,
- д) сортировка по частоте,
- е) формирование списка ключевых слов.

Рассмотрим последовательность шагов этого метода на примере. Предположим, у нас есть следующий текст:

"Машинное обучение – это область искусственного интеллекта, которая использует алгоритмы для анализа данных и создания предсказательных моделей."

После токенизации получаем множество токенов: ["Машинное", "обучение", "—", "это", "область", "искусственного", "интеллекта", "которая", "использует", "алгоритмы", "для", "анализа", "данных", "и", "создания", "предсказательных", "моделей"],

Нормализация приводит токены к единому виду: ["машинное", "обучение", "это", "область", "искусственного", "интеллекта", "которая", "использует", "алгоритмы", "для", "анализа", "данных", "и", "создания", "предсказательных", "моделей"],

После удаления стоп-слов: ["машинное", "обучение", "область", "искусственного", "интеллекта", "использует", "алгоритмы", "анализа", "данных", "создания", "предсказательных", "моделей"],

Подсчет частот (шаг г): {"машинное": 1, "обучение": 1, "область": 1, "искусственного": 1, "интеллекта": 1, "использует": 1, "алгоритмы": 1, "анализа": 1, "данных": 1, "создания": 1, "предсказательных": 1, "моделей": 1},

Шаг д –так как все значения встречаются по одному разу, сортировка не изменит порядок,

Получаем результат: ["машинное", "обучение", "область", "искусственного", "интеллекта", "использует", "алгоритмы", "анализа", "данных", "создания", "предсказательных", "моделей"].

Преимущества:

- простота реализации и вычислительная эффективность,
- хорошо работает для текстов, где центральные фразы действительно часто встречаются.

Недостатки:

- игнорирует контекст и семантику,
- может не учитывать важные лексемы, которые встречаются редко,

– стоп-слова могут не полностью исключаться, если их список не исчерпывающий.

1.7 Метод извлечения ключевых слов на основе TF-IDF

TF-IDF (Term Frequency-Inverse Document Frequency) – это статистический метод, используемый для оценки важности слова в документе относительно коллекции документов (корпуса) [20]. TF-IDF помогает выявлять главные лексемы, которые часто встречаются в конкретном документе, но редко в других документах, что делает их уникальными и значимыми [24].

TF (term frequency – частота термина) – отношение числа вхождений некоторого термина к общему числу слов документа. Таким образом, оценивается важность лексемы t в пределах отдельного документа.

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

где n_t – число вхождений лексемы t в документ,

$\sum_k n_k$ – общее число слов в документе d .

IDF (inverse document frequency – обратная частота документа) – инверсия частоты, с которой некоторое понятие встречается в документах коллекции. Основоположником данной концепции является Карен Спарк Джонс. Учёт IDF уменьшает вес широкоупотребительных лексем. Для каждого уникального слова в пределах конкретной коллекции документов существует только одно значение IDF.

$$IDF(t, D) = \log \frac{|D|}{|\{d_i \in D \mid t \in d_i\}|},$$

где $|D|$ – число документов в коллекции;

$|\{d_i \in D \mid t \in d_i\}|$ – число документов из коллекции D , в которых встречается t (когда $n_t \neq 0$).

Выбор основания логарифма в формуле не имеет значения, поскольку изменение основания приводит к изменению веса каждого слова на постоянный множитель, что не влияет на соотношение весов.

Таким образом, мера TF-IDF является произведением двух сомножителей [25]:

$$TFIDF(t, d, D) = TF(t, d) \times IDF(t, D).$$

Основные шаги работы метода:

- а) токенизация текста: разделение текста на отдельные лексемы,
- б) нормализация текста: приведение токенов к нижнему регистру, удаление пунктуации,
- в) удаление стоп-слов: исключение часто встречающихся и малоинформативных единиц текста. Шаги а-в аналогичны соответствующим шагам в предыдущем методе
- г) вычисление частоты термина (TF). Определение частоты каждого термина в документе. TF вычисляется как отношение количества вхождений термина к общему числу слов в документе. Например, если слово "обучение" встречается 3 раза в документе из 100 текстовых единиц, TF будет $3/100 = 0.03$,
- д) вычисление обратной частоты документа (IDF). Определение важности лексемы в корпусе документов. IDF вычисляется как логарифм отношения общего числа документов к числу документов, содержащих данное слово. Например, если "обучение" встречается в 10 из 1000 документов, IDF будет $\log(1000/10) = 2$,
- е) вычисление значения TF-IDF. Умножение значения TF на значение IDF для каждой лексемы. Это дает оценку важности текстовой единицы в контексте конкретного документа и всего корпуса. Например, для слова "обучение" значение TF-IDF будет $0.03 * 2 = 0.06$,

ж) формирование списка ключевых слов. На основе значений TF-IDF формируется список терминов, где значений с наибольшей оценкой считаются наиболее значимыми.

Пример работы метода TF-IDF. Рассмотрим работу с текстом из предыдущей подглавы.

"Машинное обучение – это область искусственного интеллекта, которая использует алгоритмы для анализа данных и создания предсказательных моделей."

Результаты шагов а – в (токенизация, нормализация, удаление стоп-слов) аналогичны шагам а – в перечисления в разделе 1.6.

Токены: ["машинное", "обучение", "область", "искусственного", "интеллекта", "использует", "алгоритмы", "анализа", "данных", "создания", "предсказательных", "моделей"],

Вычисление TF – например, "обучение" встречается 1 раз в документе из 15 слов, TF будет $1/15 = 0.0667$,

Вычисление IDF – например, если "обучение" встречается в 50 из 1000 документов, IDF будет $\log(1000/50) \approx 1.3$,

Вычисление значения TF-IDF для каждого токена – например, для "обучение" значение TF-IDF будет $0.0667 * 1.3 \approx 0.087$,

Сортируя по убыванию значения, получаем список ключевых слов: ["обучение", "данных", "алгоритмы", "анализ", "интеллекта", "машинное", "область", "искусственного", "использует", "создания"].

Преимущества:

- учитывает как частотность токенов в документе, так и их распространенность в корпусе,
- способен выявлять уникальные и значимые токены, которые отличают документ от остальных.

Недостатки:

- требует наличие корпуса документов для вычисления IDF,

– может не учитывать контекст и семантические связи между тематиками.

1.8 Метод извлечения ключевых слов на основе TextRank

TextRank – это метод, основанный на графовой модели, аналогичный алгоритму PageRank, используемому для ранжирования веб-страниц.

TextRank строит граф, где вершины представляют собой токены, а ребра – связи между ними. Затем алгоритм разбирает понятия по важности в зависимости от их связей [26].

Основные шаги работы:

- а) токенизация текста – разделение текста на отдельные токены,
- б) нормализация текста – приведение лексем к нижнему регистру, удаление пунктуации,
- в) удаление стоп-слов – исключение часто встречающихся и малоинформативных лексем. Шаги а-в аналогичны предыдущим методам.
- г) построение графа. Создание графа, где вершинами являются токены, а ребра связывают точки, которые находятся в пределах фиксированного окна (окна контекста) друг от друга. Например, для окна размером 2, слова "машинное" и "обучение" будут связаны,
- д) взвешивание ребер. Ребра могут взвешиваться в зависимости от частоты совместного появления в окне контекста или могут быть равнозначными,
- е) применение алгоритма PageRank к построенному графу для разделения слов. Алгоритм итеративно обновляет веса вершин в зависимости от их связей с другими вершинами, до тех пор, пока не стабилизируется группировка,
- ж) формирование списка. На основе полученных рангов формируется список ключевых слов, где единицы с наибольшими рангами считаются наиболее значимыми.

Рассмотрим пример работы с текстом из предыдущих пунктов.

"Машинное обучение – это область искусственного интеллекта, которая использует алгоритмы для анализа данных и создания предсказательных моделей."

Результаты шагов а – в (токенизация, нормализация, удаление стоп-слов) аналогичны шагам а – в перечисления в разделе 1.6.

Токены: ["машинное", "обучение", "область", "искусственного", "интеллекта", "использует", "алгоритмы", "анализа", "данных", "создания", "предсказательных", "моделей"],

Построив граф, получаем вершины: ["машинное", "обучение", "область", "искусственного", "интеллекта", "использует", "алгоритмы", "анализа", "данных", "создания", "предсказательных", "моделей"] Ребра (для окна контекста 2): ("машинное", "обучение"), ("обучение", "область"), ("область", "искусственного"), и т.д, Ребра могут взвешиваться равнозначно, например, вес 1 для каждого ребра,

Применяем алгоритм PageRank. Алгоритм назначает ранги вершинам в графе. Например, "обучение" и "данных" могут получить высокие ранги из-за множества связей,

Сортируя вершины по убыванию значений ранга, получаем результат, например: ["обучение", "данных", "алгоритмы", "анализ", "модели", "интеллекта", "машинное", "область", "искусственного", "использует"].

Преимущества:

- учитывает контекстные связи между лексемами,
- способен выявлять важные слова и фразы, даже если они встречаются редко.

Недостатки:

- вычислительно более сложен, чем частотный метод,
- может требовать настройки параметров, таких как размер окна контекста и веса ребер.

1.9 Метод извлечения ключевых слов на основе FastText и кластеризации DBSCAN

FastText – это метод, использующийся для представления слов в виде векторов и анализа их сходства на основе нейронных сетей [27]. В данном случае FastText используется для создания векторных представлений, а затем применяется кластеризация DBSCAN для выделения основных понятий.

DBSCAN (density-based spatial clustering of applications with noise – плотностный алгоритм кластеризации пространственных данных с присутствием шума) имеет несколько ключевых преимуществ. Во-первых, пользователю не нужно заранее определять количество кластеров. Во-вторых, алгоритм способен выявлять кластеры сложных форм и определять точки, которые не принадлежат ни одному кластеру. Несмотря на то, что DBSCAN работает немного медленнее, чем агломеративная кластеризация и алгоритм k-средних, он способен масштабироваться на достаточно большие наборы данных [28].

DBSCAN выделяет точки, находящиеся в "густонаселенных" областях пространства характеристик, где многие точки данных расположены близко друг к другу. Эти области называют плотными (dense) областями пространства характеристик. Суть алгоритма DBSCAN заключается в том, что кластеры образуются из плотных областей данных, которые разделены относительно пустыми пространствами.

Точки, расположенные в плотных областях, называются ядовыми примерами (core samples) или ядовыми точками (core points). Алгоритм DBSCAN использует два параметра: min_samples и eps. Если по крайней мере min_samples точек находятся в радиусе окрестности eps от рассматриваемой точки, эта точка классифицируется как ядовая. Ядовые точки, расстояния между которыми не превышают радиус окрестности eps, объединяются алгоритмом DBSCAN в один кластер [28].

Основные шаги работы:

а) токенизация,

- б) нормализация,
- в) удаление стоп-слов. Шаги а – в аналогичны предыдущим методам,
- г) представление с помощью FastText. Обучение или применение обученной модели FastText на предобработанном тексте для получения векторных представлений. Каждая текстовая единица представляется вектором фиксированной размерности,
- д) применение алгоритма DBSCAN (Density-Based Spatial Clustering of Applications with Noise) для кластеризации векторных представлений. DBSCAN использует плотность точек в пространстве векторов для определения кластеров [29],
- е) выделение ключевых терминов: для каждого кластера выбираются точки с наибольшей плотностью векторных представлений. Точки из различных кластеров считаются ключевыми,
- ж) формирование списка ключевых слов: на основе выбранных точек формируется список наиболее важных фраз.

Рассмотрим работу с текстом из предыдущих пунктов: "Машинное обучение – это область искусственного интеллекта, которая использует алгоритмы для анализа данных и создания предсказательных моделей."

Результаты шагов а – в (токенизация, нормализация, удаление стоп-слов) аналогичны шагам а – в перечисления в разделе 1.6.

Получаем токены: ["машинное", "обучение", "область", "искусственного", "интеллекта", "использует", "алгоритмы", "анализа", "данных", "создания", "предсказательных", "моделей"],

FastText обучается на тексте, чтобы получить векторные представления слов, затем применяется для получения векторов (стандартная размерность эмбедингов – 300),

DBSCAN применяется к векторам для выделения кластеров. Например, два кластера могут быть сформированы: один для перечня "машинное", "обучение", "алгоритмы", "анализ", "данных", "создания",

"предсказательных", и другой для перечня "область", "искусственного", "интеллекта", "использует",

Из каждого кластера выбираются наиболее репрезентативные значения. Ключевыми могут быть "машинное обучение", "алгоритмы", "анализ", "искусственного интеллекта". Получаем результат: ["машинное обучение", "алгоритмы", "анализ", "искусственного интеллекта"]. Есть возможность оценки значимости точки по расстоянию от центра кластера.

Преимущества:

- учитывает семантическую близость между лексемами,
- способен выявлять ключевые слова на основе их смыслового содержания.

Недостатки:

- требует больших вычислительных ресурсов для обучения модели FastText и вычисления векторных представлений токенов,
- в случае использования обученной модели может обладать пропусками в лексике,
- не всегда учитывает контекстуальную семантику понятий.

1.10 Метод извлечения ключевых слов с использованием TextRazor

TextRazor – это внешний сервис для извлечения ключевых слов из текста с учетом контекста и семантического анализа [15]. В данном случае, метод основывается на использовании API TextRazor для получения ключевых понятий из текста или URL.

Основные шаги работы:

- подготовка URL для отправки на анализ через API TextRazor,
- отправка текста или адреса на сервер TextRazor через API. Сервис TextRazor анализирует текст с учетом контекста и семантики для выделения ключевых слов,
- обработка результатов. Получение ответа от TextRazor, содержащего список фраз, которые наиболее точно отражают содержание текста.

Результаты могут включать как однословные, так и многословные конструкции.

Преимущества метода:

- использование мощных алгоритмов семантического анализа,
- учет контекста и семантической близости текстовых единиц,
- соответствие общей модели модуля выделения – «На входе текст или

URL, на выходе список ключевых слов».

Недостатки метода:

- необходимость использования внешнего сервиса,
- зависимость от доступности и качества работы API TextRazor,
- закрытость внутреннего строения и подробностей работы системы.

2 Сбор данных

2.1 Методы и средства формирования данных

В ходе исследования выделены следующие возможные методы набора данных, которые можно объединить в две группы:

– открытые наборы данных:

1) Kaggle и HuggingFace. Эти платформы предлагают предобработанные данные различных форматов и банки решений от пользователей. Примеры: Ashkangoharfar/sites-information-data-from-alexacom-dataset на Kaggle [30], Mathurinache/mostvisitedwebsites на HuggingFace [31];

2) Common Crawl: в рамках проекта создаются и распространяются публичные наборы данных веб-страниц, включая их HTML-код [32]. Инструменты для извлечения ключевых слов из содержимого страниц могут быть применены для анализа этих данных;

3) ClueWeb09 и ClueWeb12: Корпусы данных интернета, созданные в рамках проекта TREC (Text REtrieval Conference). Они содержат миллиарды записей и могут использоваться для задач в рамках семантического анализа;

4) Open Directory Project (ODP) – это каталог веб-ресурсов, также известный как DMOZ [33]. Данные о ключевых словах могут быть извлечены из описаний сайтов;

5) Web Data Commons предоставляет наборы данных, включая извлеченные данные из веб-страниц, такие как метаданные, текстовое содержание и теги. Проект специализируется на выделении структурированных данных из Common Crawl [34];

6) YFCC100M (Yahoo Flickr Creative Commons 100 Million): хотя это в основном набор данных изображений, он также содержит текстовые описания и теги, которые могут быть использованы для анализа содержания вебсайтов [35] [36];

– инструменты и API для SEO анализа:

1) Google Search Console позволяет собирать информацию о производительности сайта, включая данные о ключевых словах и поисковых запросах;

2) SEMrush – это платформа для анализа главных тэгов, конкуренции, органического поиска и других аспектов SEO [12];

3) Moz предоставляет инструменты для анализа ключевых слов, мониторинга активности, аудита сайтов и конкурентного анализа [14].

2.2 Получение множества URL для обработки

Для автоматического извлечения ключевых слов с вебсайтов необходимо сначала составить список URL-адресов, которые будут служить основой для дальнейшего анализа. На данном этапе целесообразно использовать два основных источника – рейтинг интернет-ресурсов по посещаемости Ahrefs [37] и выбор случайных адресов из большего числа документов, доступных через DMOZ [33].

Ahrefs – популярный инструмент для анализа посещаемости доменов, предоставляющий данные о наиболее популярных сайтах в различных категориях. С помощью данных Ahrefs можно выбрать 1000 сайтов с наибольшим трафиком [37], что даст представление о наиболее активных и значимых онлайн-ресурсах для анализа

DMOZ (каталог Mozilla) – один из старейших и наиболее обширных каталогов в Интернете, классифицирующий сайты по категориям и подкатегориям [33]. Выбор случайных ссылок из DMOZ расширит географический охват и разнообразие тематики ресурсов, что обеспечит разнообразие анализируемого материала.

Собранные адреса будут сохранены в файле urls.txt, который станет источником данных для загрузки, обработки сырых документов и извлечения ключевых слов.

2.3 Выбор СУБД

Для хранения загруженных HTML-документов, извлеченных ключевых слов, целевых значений и данных об ошибках имеет смысл использовать

реляционную базу данных. Об этом говорит то, что поля в основном являются простыми строками, таблицы имеют много записей, во всех случаях понятен ключ для доступа к данным – URL, большинство запросов будут на получение. Оптимальным выбором в данном случае является SQLite:

Преимущества SQLite:

- легкость и простота использования ввиду того, что это сервер баз данных, который работает с памятью или одним файлом на диске. Он не требует отдельного сервера баз данных, что упрощает установку и использование [38],
- SQLite поддерживается большинством языков программирования и операционных систем, обеспечивая свободу интеграции с различными технологиями и средами разработки,
- многопоточность и параллельное использование,
- архитектура SQLite обеспечивает простоту создания резервных копий и восстановления данных. Для резервного копирования достаточно копировать файл с расширением .db,
- SQLite поддерживает принципы ACID (Atomicity, Consistency, Isolation, Durability), что гарантирует целостность данных даже в случае сбоя системы или сети. Это повышает устойчивость программного решения [38],
- несмотря на свою простоту, SQLite обладает полноценными возможностями SQL, позволяя выполнять сложные запросы и агрегации данных.

Выбор SQLite для данного проекта обусловлен простотой его использования, универсальностью, надежностью и функциональностью, что делает его хорошим выбором в данном случае и для разработки и поддержки приложений, связанных с анализом данных и машинным обучением в целом.

2.4 Разработка таблиц в базе данных

Для задач данного проекта, а именно хранения ключевых слов, выбранных из обработанных открытых данных, хранения множества пар [блок

текста, вес], записи истории ошибок доступа к ресурсу и т.д., спроектируем таблицы в базе данных database/keywords.db, используемой с СУБД SQLite.

Таблица keywords (таблица 1) предназначена для хранения целевых понятий для каждого URL, их приоритетов и связанных данных. В таблице 2 продемонстрирован пример заполнения данными для понимания.

Таблица 1 – Структура таблицы keywords

Поле	Тип	Описание
url	Text	Адрес
keyword	Text	Ключевое слово
priority	Real	Приоритет слова (вещественное число)

Таблица 2 – Пример данных keywords

url	keyword	priority
http://url1	keyword1	0.98
http://url1	keyword2	0.5
https://site2	keyword3	0.7

Выбор такой структуры позволяет выполнять запрос на получение множества пар [слово, приоритет] по URL. Это важно для задач обучения с учителем или тестирования работы модуля, где используются данные именно такого формата. Также это позволяет легко обновлять приоритеты, выполняя поиск по паре URL + keyword.

Таблица url_text (таблица 3) хранит исходный код страниц, полученный загрузкой по URL, чтобы уменьшить количество запросов в интернет и уменьшить задержку, связанную с этим. В таблице 4 представлен пример заполнения данными.

Таблица 3 – Структура таблицы url_text

Поле	Тип	Описание
url	Text	URL
html_content	Text	Исходный код страницы в формате текста

Таблица 4 – Пример данных url_text

url	html_content
http://url1	<html> ... </html>
https://url2	<!DOCTYPE html>...

Таблица text_blocks (таблица 5) необходима для хранения результатов выделения текстовых блоков из HTML-документов. В таблице 6 представлено заполнение данными.

Таблица 5 – Структура таблицы text_blocks

Поле	Тип	Описание
url	Text	Адрес
text_block	Text	Содержимое выделенного элемента
weight		Семантический вес элемента

Таблица 6 – Пример данных text_blocks

id	url	text_block	weight
1	http://url1	title	50
2	http://url1	просто содержимое	1
3	https://site2	заголовок	30

Последняя таблица, которую мы заложим в базу данных – history (таблица 7), используется для отслеживания истории загрузки кода страниц. Запись информации о каждом адресе и результатах его загрузки позволяет отслеживать успешность операций и быстро выявлять проблемы с доступом к веб-страницам.

Таблица 7 – Структура таблицы history

Поле	Тип	Описание
url	Text	Адрес сайта
error	Text	Ошибка при загрузке (NULL, если успешно)

В таблице 8 можно видеть пример данных.

Таблица 8 – Пример данных history

url: text	error: text
http://url1	NULL
http://site2	Error 403
https://resource3	NULL

В случае, если запрос проходит успешно, в базу записывается значение NULL, что позволяет отследить также и успешные запросы. Это важно обеспечения стабильности процесса сбора данных.

Выбор такой структуры таблиц в SQLite обеспечивает гибкость и масштабируемость для проекта, простоту и понятность процессов хранения данных.

3 Архитектура решения для извлечения ключевых слов из HTML документа

3.1 Пользовательский интерфейс

Архитектуру системы можно разделить на четыре модуля, каждый из которых выполняет определенную функцию: загрузка HTML (по URL или файлу), первичная обработка данных, собственно выделение ключевых слов и пользовательский интерфейс. В ходе использования программы эти модули взаимодействуют, чтобы обеспечить достижение пользователем цели использования ПО и качество его службы.

Пользовательский интерфейс – компонент программной системы, обеспечивающий передачу информации между человеком и программными компонентами. В данной работе мы делаем фокус на исследовании моделей машинного обучения, поэтому интерфейс оставим минималистичным, но верно выполняющим свою функцию.

Страница 2 – форма ввода данных. На первой странице пользовательский интерфейс представляет собой форму ввода данных, которая позволяет пользователю загрузить HTML документ или ввести адрес веб-страницы для последующего анализа.

Элементы интерфейса:

- заголовок: "Извлечение ключевых слов из HTML документа";
- форма ввода:
 - 1) поле ввода для URL;
 - 2) кнопка "Выбрать файл" для загрузки файла с устройства;
 - 3) кнопка "Анализировать" для запуска процесса анализа.

Страница 2 – форма вывода данных. На второй странице пользовательский интерфейс отображает результаты выполнения анализа, включая список результатов.

Элементы интерфейса:

- описание полученных данных: информация о введенном адресе или имени загруженного файла, описание выполненного анализа и предварительной обработки данных,
- блок вывода результатов: список ключевых слов и фраз, извлеченных из HTML контента,
- кнопка "Назад": для возвращения на первую страницу,
- обработка ошибок: если произошла ошибка при загрузке или обработке документа, сообщение об ошибке отображается на второй странице.

3.2 Модуль загрузки HTML

Модуль загрузки отвечает за получение HTML документа из указанного URL или загруженного файла.

Функциональность модуля включает два пункта:

- загрузка по URL – выполняет HTTP запрос к указанному адресу для получения содержимого:
 - 1) если ресурс доступен и запрос выполнен успешно, модуль загружает документ для дальнейшей обработки;
 - 2) в случае возникновения сетевых ошибок (например, недоступность сервера, ошибка 404 и т.д.), модуль возвращает сообщение об ошибке;
- загрузка из файла – если пользователь загружает HTML файл с устройства, модуль обрабатывает этот файл:
 - 1) проверяет корректность формата и содержимое файла;
 - 2) в случае ошибок, таких как неверный формат файла или недопустимое содержимое, возвращает сообщение об ошибке.

Обработка ошибок:

- сетевые ошибки при загрузке по ссылке: выводится сообщение, что указанный URL недоступен или возникли проблемы с сетевым соединением,
- ошибки при загрузке файла: если файл не удалось загрузить из-за неверного формата или других проблем, пользователю выводится сообщение о проблеме с загрузкой файла,

- отображение ошибок в интерфейсе: Ошибки, возникшие в модуле загрузки, отображаются в пользовательском интерфейсе для уведомления пользователя о возникших проблемах.

3.3 Модуль обработки HTML

Модуль обработки выполняет два важных этапа предварительной обработки сырого HTML, выделяя важные отрывки текста и подготавливая их для последующего извлечения ключевых слов.

Функциональность модуля включает два пункта:

- предварительная обработка:

- 1) извлечение основного содержимого: Модуль использует алгоритмы обработки HTML документа для извлечения основного содержимого страницы, удаляя при этом ненужные элементы, такие как скрипты (`<script>`), стили (`<style>`), комментарии и другие [39];

- 2) токенизация текста: текст разбивается на текстовые единицы для последующей обработки [40];

- 3) нормализация текста: текст нормализуется для устранения нерелевантных элементов и улучшения качества результатов;

- значимость отрывков текста определяется на причастности к важным элементам (`<title>`, `<b>`, `<h1>...<h6>` и т.д.) или частоте повторения слов в рассматриваемом документе и других документах, если речь идет о TF-IDF и мультистраничном анализе.

3.4 Модуль выделения ключевых слов

Модуль выделения ключевых слов является главным компонентом разрабатываемой нами системы. В данной работе мы исследуем несколько методов извлечения, поэтому для каждого из них требуется реализовать собственный класс, поэтому в целях совместимости их нужно будет объединить одним интерфейсом. Роль модуля выделения заключается в принятии на вход предварительно обработанных данных и выдаче результата в виде множества ключевых слов, а также оценки значимости, если реализация поддерживает такую оценку.

Как уже сказано, модуль будет иметь несколько воплощений в виде уже исследованных методов:

Метод частотного анализа подсчитывает количество вхождений каждой текстовой единицы и определяет ключевыми те, которые встречаются наиболее часто. Входными данными здесь может быть просто массив токенов.

Алгоритм TextRank строит графовую модель слов и выявляет наиболее значимые точки по количеству связей. Здесь на вход требуются уже исходные структуры – предложения, параграфы.

Алгоритм TF-IDF использует две оценки значимости термина в рамках множества документов и одного конкретного. Отношение частоты единицы в одном документе к оценке по множеству документов показывает значимость слова в документе, снижая оценку общеупотребительных лексем.

Модель, использующая кластеризацию на основе векторного представления является наиболее вычислительно сложной и заглядывающей в глубокие признаки текста, но зато выявляющей семантические связи. Моделей векторизации, как и кластеризации, существует несколько, среди них мы используем многоязычную модель FastText и кластеризацию DBSCAN.

Последним методом будет использование внешнего сервиса, выполняющего задачу выделения ключевых слов. Среди таких выбран TextRazor, имеющий определенную популярность и вместе с тем бесплатный тариф на 500 запросов. Входными данными здесь должен быть массив URL.

Каждый из вариантов должен выдавать список ключевых слов ограниченной длины, то есть наиболее значимые понятия должны отделяться по принципу причастности в перцентиль, например, 1% наиболее значимых понятий, либо по отсечению оценки, например, > 0.3 доли единицы.

Также в функциональность модуля должна входить обработка ошибок, которые могут возникнуть в процессе работы каждого метода. Например, это могут быть ошибки токенизации, проблемы с доступом к внешнему сервису в случае TextRazor, или непредвиденные ошибки в работе алгоритмов. Ошибки

и исключения снабжаются информативными сообщениями для вывода в интерфейс пользователя или отражения в логах.

3.5 Взаимодействие между модулями

Описанные прежде модули взаимодействуют между собой по следующей схеме:

а) пользовательский интерфейс и модуль загрузки HTML. Пользователь вводит URL или загружает файл через интерфейс. Модуль загрузки определяет тип данных и передает документ на обработку в модуль обработки,

б) модуль загрузки и модуль обработки ключевых слов. Модуль загрузки получает содержимое по URL или из загруженного файла, затем передает его на обработку в модуль обработки. Обработка может завершиться ошибкой, и в этом случае модуль загрузки возвращает соответствующее сообщение об ошибке,

в) модуль обработки и модуль выделения ключевых слов. Модуль обработки HTML предоставляет обработанные текстовые отрывки модулю извлечения для анализа и извлечения ключевых элементов,

г) модуль выделения и пользовательский интерфейс. Результаты извлечения возвращаются обратно в пользовательский интерфейс для отображения пользователю на экране результатов.

Вся цепочка шагов а – г от модуля загрузки и до модуля извлечения (или модуля выходной обработки после выделения ключевых слов) может быть объединена в конвейер – унифицированную структуру, обладающую своим интерфейсом и вследствие этого легко заменяемую другими конвейерами.

4 Алгоритм работы приложения для извлечения ключевых слов

4.1 Сценарий взаимодействия с пользователем

Описание шагов алгоритма работы программы извлечения ключевых слов сформулировано ниже. Можно выделить шаги:

а) инициализация программы:

- 1) пользователь запускает программу;
- 2) программа отображает интерфейс;
- 3) стадия загрузки данных. Пользователь выбирает метод выделения из предложенных в списке;

б) альтернатива 1. Пользователь вводит адрес сайта, для которого хочет получить список ключевых слов:

- 1) программа выполняет HTTP запрос для загрузки кода по заданному URL;

- 2) ресурс загружается и сохраняется для дальнейшей обработки;

в) альтернатива 2. Пользователь загрузил файл:

- 1) программа принимает загруженный файл;
- 2) код HTML из файла сохраняется для дальнейшей обработки;

г) стадия предварительной обработки данных. Программа использует алгоритм контентного извлечения для выделения основного содержимого страницы, исключает навигационные элементы, рекламные блоки и другие несущественные части страницы;

д) программа разбивает текст на отдельные слова и энграммы;

е) стадия нормализации текста. Программа удаляет лишние пробелы, специальные символы и т.д.;

ж) извлечение структурированной информации. Программа извлекает ключевые фразы из заголовков страницы (тэг title), мета-описания (тэг <meta name="description">) и заголовков (тэги H1/H2);

з) генерация кандидатов на основе текстовой обработки:

- 1) программа применяет чанкер для выделения именных групп из текста;

2) программа рассматривает выделенные именные группы как потенциальные ключевые фразы;

и) стадия оценки кандидатов с использованием модели машинного обучения. Программа передает всех кандидатов в модель машинного обучения:

1) модель обучена на размеченных данных для предсказания релевантности каждого кандидата;

2) применение признаков в зависимости от применяемой модели;

3) стадия извлечения. Модель предсказывает вероятность каждого кандидата быть ключевым словом;

4) программа выбирает топ-N кандидатов с наивысшими вероятностями релевантности;

к) программа выдает результат пользователю.

4.2 Описание пользовательского интерфейса

Примерный сценарий действий пользователя при работе с интерфейсом:

– ввод адреса: Пользователь вводит URL страницы в текстовое поле. Это позволяет программе получить доступ к содержимому указанной страницы,

– загрузка HTML-документа. Вместо ссылки пользователь может загрузить файл с локального компьютера. Этот подход полезен, если ресурс находится за защищенной сетью или в случае, когда требуется обработать локальный файл *.html,

– пользователь выбирает метод выделения ключевых слов,

– инициирование процесса. Нажатие на кнопку запускает процесс извлечения ключевых слов по введенному адресу или из загруженного файла,

– отображение результатов. После обработки данных пользователь перенаправляется на страницу с результатами, где отображается информация о запросе (URL или имя файла) и список извлеченных ключевых слов.

Опишем примерный внешний вид страниц. Верхняя часть страницы ввода содержит заголовок "Извлечение ключевых слов из веб-страницы". Под

заголовком расположена форма с текстовым полем для ввода и элементом для загрузки HTML-файла. Кнопка "Продолжить" является основным элементом, привлекающим внимание пользователя.

Верхняя часть страницы результатов содержит заголовок "Результаты извлечения ключевых слов". Информация о запросе (URL или имя файла) отображается в отдельном блоке. Список результатов представлен в виде упорядоченного списка или таблицы.

4.3 Этап загрузки начальных данных

Этот этап работы программы включает в себя процесс загрузки документа, который может быть выполнен двумя способами, в соответствии с двумя типами входных данных программы: через адрес или напрямую с HTML-кодом. Перечисление шагов этапа включает ссылки на пункты сценария из раздела 4.1:

- получение входных данных. Программа начинает с получения входных данных, которые могут быть в двух форматах:

- 1) Адрес ресурса: Программа получает исходный код, сделав запрос на указанный адрес;

- 2) HTML-документ: Программа начинает обработку текста напрямую, без необходимости отправки запроса;

- загрузка документа по ссылке (шаг б). Если входные данные представляют собой ссылку, программа выполняет следующие шаги:

- 1) формирование HTTP/HTTPS запроса: программа формирует запрос к указанному домену с использованием протокола HTTP или HTTPS;

- 2) отправка запроса на сервер: запрос отправляется на сервер, который хостит сайт;

- 3) получение ответа от сервера: программа ожидает ответ от сервера;

- 4) проверка статуса ответа: проверяет статус ответа от сервера. Например, статус 200 означает успешный запрос, а другие статусы могут указывать на различные ошибки (404 - страница не найдена и т.д.);

5) чтение HTML-кода: если запрос выполнен успешно (статус 200 OK), программа считывает исходный код страницы из тела ответа. Если запрос завершается ошибкой (например, статус 404 Not Found), программа должна обработать ошибку и предпринять соответствующие действия (например, выдать сообщение об ошибке или выполнить альтернативные действия);

- работа с HTML-документом (шаг в). Если входные данные – это документ, программа пропускает шаги, связанные с отправкой запроса и получением ответа от сервера. Вместо этого она сразу начинает обработку предоставленного HTML-кода.

Общие шаги для обоих типов входных данных:

- определение типа входных данных:
- программа определяет тип полученных входных данных (URL или файл с кодом).

4.4 Этап чтения и обработки HTML-кода

Программа считывает код и производит его нормализацию (удаление HTML-тегов, очистка от комментариев и других ненужных элементов).

После загрузки содержимого страницы (по ссылке или из файла), программа производит обработку документа для выявления и извлечения ключевых фраз. Процесс включает выделение основного содержимого, очистку незначимых элементов и применение системы взвешивания сегментов текста.

Шаги обработки:

- выделение основного содержимого страницы. Программа использует алгоритм, который извлекает основное текстовое содержимое страницы, включая содержимое тегов <p>, <div>, <article> и т.д. Inline теги (например, , , <i>) учитываются как часть параграфа, а не отдельная сущность;

- очистка незначимого содержимого:

- 1) удаляются незначимые атрибуты HTML элементов, такие как id, class, style и другие, которые используются только в технических целях;

2) элементы `<script>` и `<style>` полностью удаляются вместе с их содержимым, так как они не несут смысловой нагрузки;

3) содержимое тегов `<aside>`, `<nav>` и аналогичных уменьшается в значимости, поскольку они обычно содержат вспомогательную информацию;

– токенизация и нормализация текста. Токенизация текста включает разделение его на отдельные токены (термины или фразы), что является первым шагом обработки. В этом процессе учитываются следующие особенности:

1) текст разбивается на отдельные слова и фразы, учитывая пунктуацию и пробелы;

2) слова, встречающиеся внутри тегов (например, `<b>`, `<strong>`, `<i>`) и в домене, рассматриваются как единые токены;

3) приведение всех символов к нижнему регистру для унификации лексем с разным регистром;

4) удаление лишних символов, таких как знаки пунктуации, пробелы в начале и конце строк;

5) лемматизация или стемминг, чтобы свести разные формы слов к основной форме (например, "running" к "run"). В совокупности шаги 3-5 называют нормализацией. Она направлена на приведение его к стандартизированному виду, чтобы улучшить его обработку и анализ;

– применение системы приоритезации сегментов текста:

1) title и meta description: Заголовок страницы (title) и мета-описание (meta description) получают высокий приоритет, так как часто содержат ключевую информацию о контенте страницы;

2) заголовки (`<h1>`, `<h2>`, ...): Заголовки страницы также получают повышенный приоритет, поскольку они выделяют ключевые темы на странице;

3) SEO-теги `<b>`, `<strong>` и аналогичные: Теги, выделяющие текст жирным или сильным, также имеют повышенный вес, поскольку они могут выделять ключевые фразы и концепции;

4) основной текст страницы учитывается с обычным приоритетом.

Алгоритм построения набора ключевых слов состоит из двух независимых модулей:

- построение списка терминов и соответствующих весов, где вес – действительное число, которое характеризует значимость лексемы внутри документа,
- построение фраз (сочетаний из 2 и более слов) на основе списка терминов и их весов.

Алгоритм первого модуля, набирающего список слов с весами, основан на определенной структуре HTML-документа. Язык разметки HTML предусматривает опциональное наличие различных тегов. Чтобы содержимое страницы было легко читаемо и понятно для пользователя, определенные части документа принято выделять соответствующими тегами.

Например, название документа обычно заключают внутри тега `<title></title>`, заголовок большого параграфа выделяют с помощью `<h1></h1>`. Также внутри веб-страницы могут быть скрытые элементы, которые не видны пользователю и обычно служат дополнительной информацией для поисковых систем.

Применение системы ранжирования:

- заголовок страницы (`<title>`): Содержимое тега `<title>` считается наиболее значимым для определения ключевых фраз страницы. Этот тег обычно содержит краткое, но информативное описание содержимого страницы,
- мета-описание (`<meta description>`): Мета-тег `<meta description>` также считается важным, поскольку он содержит описание страницы, которое может включать ключевые фразы и ключевые слова,
- заголовки (`<h1>`, `<h2>`, ...): Заголовки страницы обычно отражают основные темы и разделы контента. Так как они являются визуально выделенными элементами на странице, они получают повышенный вес при назначении веса ключевых фраз,

– SEO-теги (, , <i>,): Теги, отмечающие текст жирным (,) или курсивом (<i>,), также могут выделять ключевые фразы и фрагменты текста. Эти теги получают увеличенный вес,

– основной текст страницы: Основной текст содержит основное содержимое страницы. В процессе рангования учитывается положение фраз в тексте, частота использования, наличие в заголовках и мета-тегах, а также влияние в зависимости от использования.

Каждому тегу можно сопоставить действительное число, которое характеризует значимость данного тэга [41]. Используем систему, приведенную в таблице 9. Данные числа были получены на основе анализа большой выборки веб- страниц.

Таблица 9 – Оценка значимости элементов документа

Элемент документа	Оценка значимости
<title>	50
meta keywords	40
meta description	40
h1-h6	30
	30
	20
<i>, <u>, , <cite>	10
Основной текст	1

Наибольший вес получили метаэлементы и тэг <title></title>. Далее мы будем рассматривать в документе не слова, а лексемы. Например, два слова «окно» и «окна» относятся к одной и той же лексеме. При этом у каждой лексемы есть основная словоформа, в данном случае «окно». Для каждой лексемы внутри документа подсчитывается его частотность внутри каждого тэга и на основе этой частности и веса тэга рассчитывается действительное число внутри промежутка [0;1]. Математически это выглядит следующим образом:

$$W_j = \sum w_{tag}$$

$$revelance_score_j = \frac{W_j}{W_{max}},$$

где w_{tag} – оценка значимости тэга,

W_j – совокупный вес слова j ,

W_{max} – максимальный вес среди слов,

$revelance_score_j$ – оценка значимости слова.

Это число считаем весом данной лексемы внутри документа [41]. Если полученное число оказывается меньше порогового значения, то слово не включается в результирующий список. Например, можно использовать следующую формулу для фильтрации:

$$r = 0.001 \text{ revelance}_{max},$$

где $revelance_{max}$ – максимальная значимость слова.

r – пороговое значение.

Полученный вектор пар («основная_словоформа», k), где k – действительное число в $[0;1]$, подается в четвертый модуль программы.

4.5 Этап выделения ключевых слов

Для извлечения используются несколько вариантов, каждый из которых реализован в отдельном классе с общим интерфейсом. В этом разделе описаны общие шаги алгоритма и особенности каждого метода.

Общие шаги алгоритма:

а) инициализация и подготовка:

1) программа инициализирует соответствующий функционал извлечения ключевых слов;

2) загружаются необходимые модели и данные, такие как векторные модели или предобученные веса;

б) предварительная обработка текста:

1) текст разбивается на предложения или блоки текста;

- 2) токенизация – разбиение текста на отдельные токены;
- 3) удаляются стоп-слова и применяются лемматизация или стемминг для нормализации лексем;
- в) выделение признаков и генерация кандидатов ключевых слов:
 - 1) каждый метод использует свой подход для извлечения признаков из текста;
 - 2) генерация списка кандидатов на основе этих признаков;
- г) ранжирование кандидатов:
 - 1) оценка релевантности кандидатов на ключевые слова с использованием различных критериев;
 - 2) применяются соответствующие модели или алгоритмы для оценки кандидатов;
- д) извлечение ключевых слов: из кандидатов выбираются топ-N с наивысшими оценками релевантности.

Все классы далее по тексту выполняют вышеизложенные шаги, но обладают своими особенностями. Рассмотрим частности, касающиеся тех или иных шагов алгоритма для каждого из них. Пункты перечислений ниже содержат ссылки на соответствующие пункты перечисления общих шагов.

Частотный метод – базовый метод выделения ключевых слов на основе частотного анализа.

Шаги:

- токенизация текста и удаление стоп-слов (пункт б общего алгоритма),
- подсчет частоты каждого токена (подпункт 1 пункта в общего алгоритма),
- выбор значений с наибольшей частотой как ключевых (подпункт 2 пункта в общего алгоритма).

Метод на основе алгоритма TextRank, использующий графовую модель для извлечения ключевых слов.

Шаги:

- токенизация текста и удаление стоп-слов (пункт б),

- построение графа, где ребра обозначают совместное появление в окне фиксированного размера (подпункт 1 пункта в),

- применение алгоритма PageRank для оценки значимости терминов (подпункт 2 пункта в),

- выбор ключевых лексем на основе ранговых оценок (пункт д).

TF-IDF – метод, основанный на использовании взвешенной частоты термина, умноженной на обратную частоту документа.

Шаги:

- токенизация текста и удаление стоп-слов (пункт б),

- подсчет частоты каждого термина в документе (TF) (подпункт 1 пункта в),

- вычисление обратной частоты документа для каждого термина (IDF) (подпункт 1 пункта в),

- умножение TF на IDF для получения взвешенных значений (подпункт 1 пункта в),

- выбор записей с наивысшими значениями TF-IDF как результатов (пункт д).

TextRazor – путь, использующий внешний сервис для извлечения ключевых слов с учетом контекста и семантического анализа.

Шаги:

- отправка текста в сервис TextRazor (пункт б),

- получение семантически значимых фраз и понятий (подпункт 1 пункта в),

- оценка и выбор ключевых слов на основе семантической релевантности (пункт д).

Метод кластеризации векторов – использует модель FastText для представления лексем в виде векторов и анализа их сходства.

Шаги:

- токенизация текста и удаление стоп-слов (пункт б),

- преобразование токенов в векторы с помощью модели FastText (подпункт 1 пункта в),
- анализ сходства векторов для выявления центральных понятий (подпункт 2 пункта в),
- оценка слов по их значимости на основе векторных представлений (пункт д).

4.6 Алгоритм сбора данных для обучения и тестирования модели

В число алгоритмов программного решения также стоит отнести механизм формирования базы данных для проведения обучения и тестирования модели. Для этих целей необходимо набрать множество записей с признаками: URL (необязательно), текст HTML документа, перечень ключевых слов. Формирование такого множества можно проводить несколькими путями:

- воспользоваться подготовленными наборами данных из банков Kaggle и HuggingFace,
- автоматический парсинг в обратном к цели работы направлении: использование поисковых систем для поиска по ключевым словам и исследование результатов поиска на предмет наиболее успешных в соответствии этим ключевым словам сайтов, изучение используемых в них методов SEO,
- набор базы данных путем использования API поисковых машин.

Этот путь позволит наиболее точно решить задачу семантического анализа с точки зрения потенциального пользователя, так как он заинтересован именно в наличии механизма, наиболее точно соответствующего работе поисковых движков.

5 Реализация программного решения

5.1 Файловая структура проекта

Проект структурирован в виде иерархической файловой структуры для обеспечения удобства управления и разработки. Основная часть кода находится в директории `src`, которая включает в себя несколько разделов, объединяющих файлы по функциональности. Листинг файлов проекта представлен в приложении А.

В корне `src` расположены файлы `__init__.py`, которые служат для инициализации модулей, и файлы, предназначенные для общих утилит и скриптов, такие как `comparison.py`, `comparison_from_file.py`, `use_urls.py` и `script.py`. Они обеспечивают базовые операции с данными и управление URL-адресами.

Директория `comparator` содержит модуль `keyword_comparator.py`, предоставляющий функционал для сравнения ключевых слов между различными источниками данных. Этот модуль используется для анализа схожести наборов лексем и оценки их релевантности.

В `database_manager` находятся модули, отвечающие за управление базой данных. `database_manager.py` определяет интерфейс для работы с базой данных, а `sqlite_database_manager.py` предоставляет конкретную реализацию для SQLite. Эти компоненты отвечают за сохранение ключевых слов, управление историей запросов и обработку текстовых блоков.

Директория `data_input` включает файлы `data_input.py` и `url_input.py`, предназначенные для чтения входных данных из различных источников, включая файлы и URL-адреса, и передачи их на следующий этап обработки.

Папка `data_output` содержит модули для вывода обработанных данных: `database_output.py`, `data_output.py`, `db_and_terminal.py`, `terminal_output.py` и `transform_output.py`. Они обеспечивают выдачу результатов обработки данных в различных форматах и для различных целей использования.

В папке `data_processing` расположены файлы для обработки данных, такие как `data_processor.py`, `element_weight_assigner.py`, `html_data_processor.py`

и `html_processor_db.py`. Эти компоненты отвечают за структурирование и анализ входных данных, включая разбиение на взвешенные блоки текста.

Модули `keyword_extractor.py`, `extractor.py`, `textrank_extractor.py`, `tfidf_extractor.py`, `fasttext_extractor.py` в директории `keyword_extractor` являются реализациями различных алгоритмов извлечения, включая TF-IDF и TextRank. Классы в файлах `textrazor_keyword_extractor.py` и `keyword_extractor_with_db.py` соответствует общему интерфейсу, но получают ключевые слова из API сервиса TextRazor и базы данных `keywords` соответственно. Все они предоставляют методы для извлечения ключевых слов из текстовых данных.

Директория `pipeline` содержит конвейеры обработки данных, такие как `common_pipeline.py`, `database_pipeline.py`, `keyword_extraction_pipeline.py`, `pipeline.py` и `textrazor_pipeline.py`. Они обеспечивают упрощенную организацию обработки данных и предоставляют единый интерфейс для преобразования и передачи данных между различными этапами.

Директория `util` включает утилитарные скрипты, такие как `comparison.py`, `comparison_from_file.py`, `script.py`, `url_text.py` и `use_urls.py`, которые реализуют сценарии множественного заполнения таблицы `url_text`, обработки URL из файла `urls.txt`, сравнения вариантов.

В папке `web` находятся материалы веб-интерфейса, включая `keyword_extraction_methods.json` для конфигурации методов, `main.py` и `run.py` для серверного кода, `static/css/style.css` для стилей, а также шаблоны в `templates` (`error.html`, `index.html` и `results.html`). Это компоненты сервера, обеспечивающие взаимодействие с пользователем через веб-интерфейс, включая ввод данных, отображение результатов и обработку ошибок.

Такая структура позволяет легко разделять функциональные компоненты проекта на логически обособленные модули, что улучшает поддержку, расширяемость и понимание кода

5.2 Сценарий загрузки HTML-документов

Сценарий загрузки выполняет несколько задач, связанных с извлечением исходного кода ресурсов. Исходный код представлен в приложении А.1. Далее подробно опишем работу скрипта.

Подключение к базе данных – скрипт начинает с подключения к базе данных SQLite keywords.db, расположенной в директории database. Создается объект conn для управления соединением с базой данных и объект cursor, который используется для выполнения SQL-запросов. Если таблица url_html еще не существует, скрипт создает ее с двумя столбцами: url типа TEXT, который является первичным ключом, и html_content типа TEXT, который будет хранить код веб-страницы. Это делается с помощью команды CREATE TABLE IF NOT EXISTS.

Функция fetch_and_store_html принимает URL веб-страницы в качестве аргумента и выполняет следующие действия:

Функция сначала проверяет, существует ли уже запись с данным адресом в таблице. Это делается с помощью запроса SELECT COUNT(*) FROM url_html WHERE url = ?, который считает количество строк.

Если запись отсутствует (count == 0), функция пытается загрузить исходный код страницы с помощью requests.get(url) и проверяет, что ответ успешный (response.raise_for_status()).

Далее функция выполняет запись в таблицу url_html с использованием запроса INSERT OR REPLACE INTO url_html (url, html_content) VALUES (?, ?). Это позволяет избежать дублирования данных, заменяя существующие записи новыми значениями, если они уже присутствуют.

Скрипт извлекает список уникальных адресов из таблицы keywords, используя запрос SELECT DISTINCT url FROM keywords. Полученный список передается в цикл.

Цикл: для каждого адреса в списке вызывается функция fetch_and_store_html, которая либо загружает и сохраняет исходный код, либо пропускает итерацию, если он уже присутствует в базе данных.

5.3 Сценарий разбиения HTML документа

Сценарий `extract_text_blocks_and_save.py` предназначен для извлечения текстовых блоков из кода и сохранения этих блоков в базу данных SQLite. Исходный код представлен в приложении А.2.

Описание работы скрипта:

- подготовка к работе,
- инициализация объектов,
- извлечение содержимого,
- обработка контента,
- сохранение текстовых блоков,
- завершение работы.

Скрипт начинает с импорта необходимых модулей и классов: `SQLiteDatabaseManager` для работы с базой данных SQLite, `HTMLDataProcessor` для обработки HTML-контента, и `ElementWeightAssigner` для определения весов элементов.

Создаются экземпляры `ElementWeightAssigner` и `HTMLDataProcessor`, которые будут использоваться для определения весов элементов HTML и извлечения текстовых блоков соответственно.

Вызывается функция `get_url_html_data()` у объекта `db_manager`, который возвращает список кортежей с доменом и кодом всех сохраненных записей.

Для каждой записи (URL, код) из полученного списка вызывается метод `process(html_content)` объекта `html_processor`. Он использует `BeautifulSoup` для парсинга HTML и извлечения текстовых блоков с помощью метода `extract_text_blocks()`.

Извлеченные текстовые блоки вместе с ссылкой и их весами собираются в список `text_blocks_to_insert`. Этот список передается в `save_text_blocks(text_blocks_to_insert)` объекта `db_manager`, который вставляет их в таблицу `text_blocks` базы данных SQLite.

По завершении операций соединение с базой данных закрывается вызовом метода `close()` у объекта `db_manager`.

В конце скрипт выводит сообщение о количестве записанных текстовых блоков.

Используемые классы:

- `SQLiteDatabaseManager`: Класс для работы с базой данных SQLite. Включает `get_url_html_data()` для извлечения URL и текста страницы, и `save_text_blocks()` для сохранения текстовых блоков.
- `HTMLDataProcessor`: Класс для обработки HTML-контента. Включает `process(html)`, который использует BeautifulSoup для извлечения текстовых блоков.
- `ElementWeightAssigner`: Класс для определения весов элементов HTML. Содержит словарь `weights`, где ключи - типы элементов, а значения – их веса. Используется метод `get_weight(element_name)` для получения веса элемента.

5.4 Сценарий выделения ключевых слов из HTML документа

Скрипт в файле `use_urls.py` предназначен для извлечения ключевых слов из веб- страниц, чьи URL содержатся в файле `urls.txt`, и сохранения результатов как в базу данных, так и на консоль. Исходный код представлен в приложении А.3.

Процесс обработки включает в себя несколько шагов, которые входят в функцию `process_urls`:

- инициализирует менеджер базы данных `SQLiteDatabaseManager`;
- создает объект `KeywordExtractorWithDB`, который использует `TextRazorKeywordExtractor` для извлечения ключевых слов и `SQLiteDatabaseManager` для управления базой данных;
- определяет вывод через `DbAndTerminalOutput`, который сочетает печать в терминал (`TerminalOutput`) и запись в базу данных (`DatabaseOutput`);
- создает экземпляр `CommonPipeline` с `KeywordExtractorWithDB` и `DbAndTerminalOutput`;
- чтение URL-адресов: чтение файла `urls.txt` и считывание URL-адресов с удалением лишних пробелов с каждой строки;

- обработка URL-адресов. Для каждого URL в списке:
 - 1) выводит сообщение о текущей обработке URL;
 - 2) запускает пайплайн для данного URL с помощью метода run;
 - 3) в случае возникновения ошибки, логирует сообщение об ошибке с указанием URL.

5.5 Управление базой данных

5.5.1 Интерфейс DatabaseManager

Файл `database_manager.py` содержит абстрактный базовый класс `DatabaseManager`, который определяет интерфейс классов для манипуляций с базой данных. Все методы в этом классе являются абстрактными и должны быть реализованы в подклассах. Исходный код представлен в приложении А.4. Опишем каждый атрибут интерфейса, реализуемый в наследующих классах:

- `url_exists(url: str) -> bool` – функция проверяет, существует ли запись с данным адресом в базе данных. Параметры: `url` – строка, представляющая адрес для проверки. Возвращаемое значение: Булево значение, указывающее, существует ли запись в базе данных.

- `save_keywords(url: str, keywords: dict)` – функция сохраняет результаты для данного адреса в базе данных. Параметры:

- 1) `url` – адрес, для которого сохраняются ключевые слова.
- 2) `keywords` – словарь, где ключи – это лексемы, а значения – их значимость.

- `save_history(url: str)` – сохраняет запись об успешной обработке. Параметры: `url` – строка, представляющая успешно обработанный запрос.

- `save_error(url: str, error_message: str)`. Сохраняет сообщение об ошибке. Параметры:

- 1) `url` – строка, представляющая ссылку на ресурс, для которого сохраняется ошибка.
- 2) `error_message` – строка, представляющая сообщение об ошибке.

- `get_text_blocks(url: str) -> list`. Возвращает текстовые блоки для данного адреса из базы данных. Параметры: `url` – , для которого извлекаются

текстовые блоки. Возвращаемое значение: Список кортежей, где каждый кортеж состоит из текстового блока и его веса.

- `save_text_blocks(url: str, text_blocks: list)`. Сохраняет текстовые блоки для данного URL в базе данных. Параметры:

- 1) `url` – строка, представляющая URL, для которого сохраняются текстовые блоки.

- 2) `text_blocks` – список кортежей, где каждый кортеж состоит из текстового блока и его веса.

- `get_keywords_by_url(url: str) -> list`. Возвращает ключевые слова для данного URL из базы данных. Параметры: `url` – строка, представляющая URL, для которого извлекаются ключевые слова. Возвращаемое значение: Список кортежей, где каждый кортеж состоит из ключевого слова и его приоритета.

- `get_urls_chunk(chunk_size: int = 1) -> list`. Возвращает часть URL из базы данных. Параметры: `chunk_size` – целое число, представляющее количество URL для извлечения за один раз (по умолчанию 1). Возвращаемое значение: Список строк, каждая из которых представляет URL.

- `get_url_html_data() -> list`. Возвращает HTML-контент для всех URL из базы данных. Параметры: не принимает параметров. Возвращаемое значение: Список кортежей, где каждый кортеж состоит из URL и HTML-контента.

5.5.2 Класс `SQLiteDatabaseManager`

Класс `SQLiteDatabaseManager` наследуется от интерфейса `DatabaseManager` и предоставляет функции для создания необходимых таблиц при инициализации, использует стандартные SQL-запросы для выполнения операций с базой данных. Исходный код представлен в приложении А.4. В частности, он создает четыре таблицы: `keywords`, `history`, `text_blocks` и `url_html`. Эти таблицы обеспечивают хранение ключевых слов, истории обработанных URL, текстовых блоков с их весом и исходного кода HTML соответственно.

При реализации методов для взаимодействия с базой данных, класс использует параметризованные запросы для предотвращения SQL-инъекций и

обеспечивает коммит всех изменений в базу данных для сохранения консистентности данных. Кроме того, класс включает `close` для закрытия соединения с базой данных и освобождения ресурсов.

Использование SQLite в разработке системы позволяет легко управлять и хранить данные без необходимости настройки и обслуживания полноценной базы данных, такой как MySQL или PostgreSQL.

5.6 Входная и выходная обработка

5.6.1 Входные обработчики данных

Для того, чтобы гибко использовать классы-выявители ключевых слов в различных обстоятельствах, например, при использовании приложения пользователем или в скрипте сравнения классов, необходимо использовать модульные входные и выходные обработчики данных, которые были бы заменимы и использовались для взаимодействия с внешними источниками данных и последующей обработки или вывода результатов. Исходный код входных обработчиков представлен в приложении А.5.

Интерфейс `DataInput` (приложение А.5) определяет метод `get_data`, который должен быть реализован в каждом классе, представляющем входной обработчик данных. Он используется для загрузки данных из различных источников, таких как файлы или веб-страницы. Каждый конкретный класс, реализующий `DataInput`, будет возвращать данные в виде строки.

Входные обработчики данных `FileInput` и `URLInput` участвуют в процессе загрузки и подготовки данных. Класс `FileInput` предоставляет чтение данных из файла, а `URLInput` получает данные с использованием HTTP-запросов. `FileInput` обеспечивает возможность загрузки текстовых данных из файлов, что полезно при обработке предварительно сохраненных данных, таких как локальные текстовые файлы или файлы с данными. `URLInput` используется для загрузки данных с веб-страниц. Он отправляет HTTP-запросы на указанный URL и возвращает HTML-контент страницы.

Все входные обработчики наследуют абстрактный базовый класс `DataInput`, который определяет общий интерфейс для загрузки данных. Это

позволяет легко подменять и добавлять новые источники данных без изменения основного кода обработки.

5.6.2 Выходные обработчики данных

Интерфейс `DataOutput` (приложение А.6) определяет метод `output_data`, который должен быть реализован в каждом классе, представляющем выходной обработчик данных. Он принимает словарь `keywords` и строку `source`, указывающую источник данных. Каждый конкретный класс, реализующий `DataOutput`, будет выполнять операции с данными, такие как сохранение в базу данных, вывод на терминал или их преобразование перед выводом, и возвращать результаты операций:

- `DatabaseOutput` использует `DatabaseManager` для сохранения результатов в таблице `keywords`. Он предоставляет возможность сохранения данных в виде словаря,
- `TerminalOutput` – класс для вывода полученных результатов на терминал. Он принимает данные в форме словаря или списка и выводит их в удобном для чтения формате,
- `DBAndTerminalOutput` комбинирует функциональность `DatabaseOutput` и `TerminalOutput`, обеспечивая сохранение в базу данных и вывод на терминал в одной операции,
- `TransformOutput` – класс, который преобразует данные с использованием предоставленной функции `transform_func`.

5.7 Обработка данных

5.7.1 Интерфейс `DataProcessor`

Раздел `data_processing` предоставляет необходимую функциональность для преобразования HTML-кода в структурированный формат данных, удобный для использования моделями машинного обучения. Классы и интерфейсы этого раздела обеспечивают очистку данных, извлечение текстовых блоков и присвоение им весов, что приводит данные в обрабатываемый моделью МО формат и обеспечивает более высокую

точность прогнозирования. Исходный код обработчиков представлен в приложении А.7.

Класс `DataProcessor` является абстрактным базовым классом для всех обработчиков данных. Он определяет единственный абстрактный метод `process`, который должен быть реализован в дочерних классах. `process` принимает HTML-код в виде строки и возвращает список кортежей, представляющих текстовые блоки и их веса.

Функция `process(html: str) -> list` принимает на вход HTML-код страницы в виде строки и возвращает список кортежей. Каждый кортеж представляет собой текстовый блок и его вес. Реализация этого атрибута в подклассах определяет конкретные шаги обработки данных.

5.7.2 `ElementWeightAssigner`

`ElementWeightAssigner` определяет веса для различных элементов HTML, таких как заголовки, мета-теги и другие элементы. Эти веса используются в `HTMLDataProcessor` для определения важности каждого текстового блока. Использование весов позволяет моделям машинного обучения учитывать значимость каждого элемента при анализе страницы.

Описание:

- `__init__()` – конструктор класса, инициализирующий словарь весов для различных элементов HTML,
- `get_weight(element_name: str) -> int`: возвращает вес элемента HTML по его имени.

5.7.3 `HTMLDataProcessor`

`HTMLDataProcessor` выполняет преобразование HTML-кода в структурированный формат данных, готовый для использования моделями машинного обучения. Он использует `ElementWeightAssigner` и выделяет текстовые блоки, назначая им веса.

Описание:

- `__init__(weight_assigner: ElementWeightAssigner)`: конструктор класса, который принимает объект `ElementWeightAssigner`,

- `process(html: str) -> list`: принимает на вход HTML- код страницы, очищает его от скриптов и стилей, извлекает текстовые блоки, назначая им веса с помощью `ElementWeightAssigner` и возвращает список кортежей,
- `clean_html(html: str) -> BeautifulSoup`: Приватный, очищает HTML-код от скриптов и стилей с использованием библиотеки `BeautifulSoup`,
- `extract_text_blocks(soup: BeautifulSoup) -> list`: Приватный, извлекает текстовые блоки и их веса из объекта `BeautifulSoup`.

5.7.4 HTMLDataProcessorWrapper

`HTMLDataProcessorWrapper` представляет собой обертку вокруг `HTMLDataProcessor`, которая добавляет функциональность работы с базой данных `SQLite` для сохранения и получения текстовых блоков.

- `__init__()`: конструктор класса, который инициализирует `HTMLDataProcessor` и `SQLiteDatabaseManager`,
- `process(url: str, html: str) -> list`: обрабатывает HTML- код страницы, используя `HTMLDataProcessor`, и сохраняет полученные текстовые блоки в базу данных `SQLite`, если они не были найдены ранее,
- `process_html(html: str) -> list`: обрабатывает HTML-код страницы, используя `HTMLDataProcessor`, без сохранения в базу данных.

5.8 Выделение ключевых слов

5.8.1 Интерфейс `KeywordExtractor`

В `extractor.py` определен абстрактный базовый класс `KeywordExtractor`, который определяет интерфейс для всех классов, используемых для извлечения важнейших терминов. Этот класс описывает `extract_keywords`, который должен быть обязательно реализован в наследнике, который принимает URL-адрес и возвращает словарь с соответствующими значениями веса. Кроме того, класс предоставляет два метода, которые могут быть переопределены в подклассах: `train` и `extract_keywords_from_html_text`. `Train` по умолчанию вызывает ошибку `NotImplementedError` и сигнализирует, что обучение модели не поддерживается, если только этот метод не переопределен реализацией подкласса. `Extract_keywords_from_html_text` также по умолчанию

генерирует исключение и предназначен для прямого извлечения ключевых слов из текста HTML.

Описание класса `KeywordExtractor` (исходный код представлен в приложении А.8):

- `extract_keywords(self, url: str) -> dict`: абстрактный метод должен быть реализован в подклассах. Он предназначен для получения результатов для заданного адреса,

- `train(self, data: list)`: по умолчанию вызывает исключение `NotImplementedError`. Он может быть переопределен в подклассах, если требуется поддержка обучения модели на наборе данных,

- `extract_keywords_from_html_text(self, html: str)`: по умолчанию вызывает исключение `NotImplementedError`. Он может быть переопределен в подклассах, если требуется поддержка извлечения ключевых слов напрямую из HTML текста.

Наследующие классы:

- `src/keyword_extractor/extractor.py` содержит реализацию класса `SimpleKeywordExtractor`, который реализует алгоритмы извлечения с использованием библиотеки NLTK. Ключевые слова извлекаются путем маркировки текста, фильтрации стоп-слов и выбора важных частей речи (существительных, глаголов, прилагательных). Вес каждого значения определяется его частотой в тексте и весом текстового блока, в котором оно встречается,

- `src/keyword_extractor/textrank_extractor.py` содержит реализацию класса `TextRankExtractor`, который использует алгоритм `TextRank` для извлечения ключевых слов. Он строит диаграмму на основе блоков текста и вычисляет важность узлов, используя алгоритм `PageRank`,

- `src/keyword_extractor/tfidf_extractor.py` содержит реализацию класса `TFIDFKeywordExtractor`, использующего алгоритм TF-IDF для извлечения с применением библиотеки `scikit-learn` для вычисления значений TF-IDF и выбора наиболее значимых терминов из текстовых блоков,

- `src/keyword_extractor/fasttext_extractor.py` содержит реализацию класса `FastTextKeywordExtractor`, использующего векторизацию токенизированного текста с применением модели `fasttext` и кластеризацию методом `DBSCAN`. Среди кластеров выделяются наиболее характеристичные точки, которые объявляются ключевыми словами, геометрически оценивается их значимость,

- `src/keyword_extractor/textrazor_keyword_extractor.py` содержит реализацию класса `TextRazorKeywordExtractor`, использующего внешний сервис `TextRazor` применяя API через поставляемую библиотеку,

- `src/keyword_extractor/keyword_extractor_with_db.py` содержит класс `KeywordExtractorWithDb`, который добавляет функции базы данных к существующему классу-экстрактору, передаваемому в качестве аргумента конструктора, обёртывая его. Этот класс позволяет извлекать ключевые слова из базы данных в случае их наличия там, без необходимости проводить вычисления обёртываемым классом. В случае отсутствия записей в базе данных, выделение производится основным классом, после чего результат сохраняется в таблицу `keywords`.

5.8.2 SimpleKeywordExtractor

`SimpleKeywordExtractor` – это класс для извлечения ключевых слов из текста, который использует частотный анализ.

В этом классе реализован метод, основанный на анализе частоты текстовых единиц и их значимости в тексте. Исходный код представлен в Приложении А.8.

Параметры инициализации и атрибуты класса:

- `fetcher` (тип `DataInput`): Компонент для получения данных. По умолчанию используется `URLInput` для загрузки данных по адресу,

- `weight_assigner` (тип `ElementWeightAssigner`): Класс для назначения весов элементам текста,

- `html_processor` (тип `HTMLDataProcessorWrapper`): Компонент для обработки сырого исходного кода.

Методы:

- `fetch_html(self, url)`: используется для загрузки ресурса с указанного URL,
- `extract_and_filter_keywords(self, text_blocks)`: извлекает и фильтрует ключевые слова из текстовых блоков, удаляя стоп-слова и нерелевантные части речи. Возвращает словарь с терминами и их вещественными весами,
- `normalize_keyword_weights(self, keywords)`: нормализует веса результатов, чтобы привести их к единой шкале [0.0; 1.0],
- `generate_keywords(self, text_blocks)`: ключевые слова выделяются на основании их частого повторения в тексте, при этом учитывается вес блока, в которым термин встречается,
- `extract_keywords(self, url)`: основной метод для извлечения ключевых слов из веб-страницы. Загружает контент, обрабатывает его и генерирует список результатов,
- `extract_keywords_from_html_text(self, html)`: извлекает ключевые слова из HTML-текста, минуя этап загрузки данных.

5.8.3 TextRazorKeywordExtractor

`TextRazorKeywordExtractor` – это класс для извлечения ключевых слов с использованием сервиса `TextRazor`, который предоставляет API для обработки текстов и извлечения сущностей, тем и других ключевых данных. Исходный код представлен в приложении А.8.

Основные методы:

- `__init__`. Конструктор инициализирует клиент `TextRazor`, используя предоставленный API ключ. Настраивает извлечение сущностей, тем и классификации новостных кодов,
- `extract_keywords(url)` выполняет извлечение из ресурса по указанному URL. Анализирует содержимое страницы с помощью `TextRazor`, извлекает сущности из ответа, сортируя их по уровню релевантности, возвращает словарь в формате `{entity_id: relevance_score}`,

– `extract_keywords_from_html_text(html)` выполняет аналогичные операции, что и `extract_keywords`, но вместо URL использует непосредственно HTML-код.

5.8.4. KeywordExtractorWithDB

`KeywordExtractorWithDB` – это класс, который расширяет функциональность извлечения ключевых слов, добавляя функции сохранения истории и ошибок в базу данных при обработке. Исходный код представлен в приложении А.8.

Основные методы

– `__init__(self, keyword_extractor, db_manager, overwrite=False)`

Конструктор инициализирует объект `KeywordExtractorWithDB` с заданными параметрами:

1) `keyword_extractor` – объект, который используется для извлечения ключевых слов;

2) `db_manager` – менеджер базы данных, который используется для сохранения истории и ошибок;

3) `overwrite` – флаг, указывающий, следует ли перезаписывать данные в случае их существования;

– `extract_keywords(self, url)` выполняет извлечение из заданного URL. Проверяет, существует ли адрес в базе данных. Если запись уже существует и флаг `overwrite` равен `False`, функция выводит сообщение об этом и прерывает выполнение. Иначе, пытается извлечь ключевые слова с помощью `keyword_extractor`. При успехе сохраняет запись в истории. В случае ошибки логирует сообщение и сохраняет информацию об ошибке в базе данных.

5.8.5 TFIDFKeywordExtractor

`TFIDFKeywordExtractor` является классом, который использует TF-IDF для извлечения ключевых слов из текстовых данных.

Основные методы

– `__init__(self)`. Конструктор инициализирует объект

`TFIDFKeywordExtractor` и его зависимости:

- 1) `weight_assigner`: объект для присвоения весов элементам;
 - 2) `html_processor`: обертка для обработки HTML данных;
 - 3) `fetcher`: объект для получения данных по URL;
 - 4) `vectorizer`: векторизатор TF-IDF, настроенный для токенизации текста и удаления стоп-слов английского языка;
- `tokenize_text(self, text)` – используется для токенизации текста. Преобразует текст в нижний регистр и извлекает только алфавитно-цифровые символы;
 - `extract_keywords(self, url)` извлекает ключевые слова из заданного URL. Получает исходный код по адресу с помощью `fetch_html`. Обработывает текстовые блоки с использованием `html_processor`. Генерирует ключевые слова с помощью `generate_keywords`;
 - `extract_keywords_from_html_text(self, html)` выполняет те же шаги, что и `extract_keywords`, кроме получения документа по URL;
 - `fetch_html(self, url)` получает содержимое по указанному URL с помощью `fetcher`;
 - `extract_and_filter_keywords(self, text_blocks)` извлекает и фильтрует значимые термины из текстовых блоков. Использует стоп-слова из NLTK, удаляет токены, не являющиеся алфавитно-цифровыми, и отфильтровывает по частям речи (существительные, глаголы, прилагательные). Нормализует веса результатов с помощью `normalize_keyword_weights`;
 - `normalize_keyword_weights(self, keywords)` нормализует веса ключевых слов по максимальному приоритету;
 - `generate_keywords(self, text_blocks)` извлекает и фильтрует полученные значения с помощью `extract_and_filter_keywords`. Нормализует веса результатов с помощью `normalize_keyword_weights`.

5.8.6 TextRankKeywordExtractor

`TextRankExtractor` использует алгоритм `TextRank` для автоматического извлечения ключевых слов из текста. Алгоритм строит граф на основе текстовых блоков, где узлы представляют собой токены, а ребра - связи между

токенами на основе совместного появления в текстах. Затем применяется алгоритм PageRank для вычисления важности каждого токена. Ключевые слова выбираются на основе их взвешенных оценок. Исходный код представлен в приложении А.8.

Этот метод подходит для извлечения наиболее важных фраз из текстов, особенно в тех случаях, когда необходимо учитывать связи между различными частями текста для определения важности токенов.

Методы:

– `__init__(self)`. Конструктор инициализирует объект `TextRankKeywordExtractor` и его зависимости:

- 1) `weight_assigner`: объект для присвоения весов элементам;
- 2) `html_processor`: обертка для обработки HTML данных;
- 3) `fetcher`: объект для получения данных;

– `build_graph(text_blocks)`. Используется для построения графа на основе текстовых блоков. Создает узлы для каждого уникального токена и добавляет ребра на основе совместного появления токенов. Возвращает объект графа `G` типа `nx.Graph`;

– `tokenize_and_filter(text)`. Предназначен для токенизации и фильтрации текста. Использует `word_tokenize` для токенизации текста и `pos_tag` для определения частей речи. Возвращает список отфильтрованных токенов;

– `extract_keywords(url)`. Извлечение ключевых слов по URL. Получает контент с помощью `fetcher`, обрабатывает его с использованием `html_processor` и вызывает `generate_keywords`;

– `extract_keywords_from_html_text(html)`. Извлечение ключевых слов из текста. Разбирает HTML с помощью `html_processor` и вызывает `generate_keywords`;

– `generate_keywords(text_blocks)`. Генерирует набора значений из текстовых блоков. Строит граф `G` с помощью `build_graph`, вычисляет веса вершин с помощью PageRank (`nx.pagerank(G)`). Возвращает словарь `keywords`, где ключи – это токены, а значения – взвешенные оценки их значимости,

- функция `predict_text_vectors` принимает текст в качестве входных данных, токенизирует его, удаляет стоп-слова и предсказывает векторы для каждого слова в тексте. Результатом является список векторов, составленный на основе предсказаний модели.

5.8.7 FastTextExtractor

Класс `FastTextKeywordExtractor` представляет собой реализацию интерфейса `KeywordExtractor` для извлечения ключевых слов из текста с использованием модели `FastText` и алгоритма кластеризации `KMeans`. Давайте подробно опишем этот класс. Исходный код представлен в приложении А.8.

Атрибуты класса

- `model`: модель `FastText` для векторизации лексем,
- `fetcher`: объект, реализующий интерфейс `DataInput`, используемый для получения данных по URL или HTML,
- `processor`: экземпляр класса `HTMLDataProcessorWrapper`, обеспечивающий предобработку кода и текстовых данных.

Методы класса

- конструктор `__init__(self, fetcher: DataInput, model)` инициализирует атрибуты `fetcher` и `processor`. Если модель `FastText` не передана, загружает модель "fasttext-wiki-news-subwords-300" с помощью `gensim.downloader`,
- `get_word_vector(self, word) -> np.ndarray`. Принимает лексему и возвращает его векторное представление с использованием модели `FastText`. Если значение отсутствует в модели, возвращает нулевой вектор,
- `extract_keywords(self, url: str) -> Dict[str, float]`. Принимает адрес и извлекает результаты из содержимого страницы по этому URL. Использует `fetcher.get_data` и `processor.process` для получения текстовых блоков и их предварительной обработки, `generate_keywords` для извлечения ключевых слов. Возвращает словарь результатов с оценкой их значимости,
- `extract_keywords_from_html_text(self, html: str) -> Dict[str, float]`. Аналогичен методу `extract_keywords`, но в качестве параметров принимает текст документа, пропуская стадию получения ресурса по URL,

– `generate_keywords(self, text_blocks: List[Tuple[str, float]]) -> Dict[str, float]`. Принимает список текстовых блоков с весами. Фильтрует незначимые текстовые единицы с использованием стоп-слов, вычисляет вектора для каждого токена в текстовых блоках и их взвешенное среднее значение. Производит кластеризацию векторов с использованием алгоритма DBSCAN, вычисляет оценки точек на основе их расстояния до центров кластеров, нормализует результаты для получения значений в диапазоне от 0 до 1. Возвращает словарь лексем с оценкой их значимости в тексте,

– `filter_words(self, words: list[str]) -> List[str]`. Принимает список токенов и фильтрует его, удаляя стоп-слова и ненужные символы. Возвращает отфильтрованный список.

5.9 Конвейеры

5.9.1 Определение понятия

Конвейеры (или пайплайны, от англ. pipeline) представляют собой структурированные процессы, которые последовательно объединяют этапы обработки данных для достижения цели извлечения ключевых слов. Эти этапы включают загрузку входных данных, предварительную обработку, извлечение, постобработку и вывод результатов. Основная идея использования конвейеров заключается в унификации контракта от адреса или файла до словаря лексем в результате.

Первая ступень конвейера – загрузка данных из источников в виде URL, текста или СУБД. Здесь используются классы интерфейса `DataInput`, `CommonFilePipeline` обрабатывает текст из файлов, `CommonPipeline`, использует адреса. В случае конвейера `DatabaseKeywordsPipeline`, параметром также является URL, но данные извлекаются просто из базы данных.

Далее следует этап обработки, где входные данные перерабатываются в ключевые слова. Главным элементом здесь является тот или иной `KeywordExtractor`, но также может включаться стадия очистки и нормализации данных. Ненужные элементы, такие как `script` и `style` в HTML, при этом удаляются, токены фильтруются, а текст разделяется на взвешенные блоки.

Затем данные анализируются для извлечения ключевых слов. Обработка данных для выделения требуется не всегда, TextRazorPipeline или DatabaseKeywordsPipeline не предполагают этого в своей работе и не принимают в своих конструкторах.

Заключительный этап – вывод обработанных данных. Ключевые слова или фразы выводятся в формате словаря (термин, значимость), что унифицирует и выходную часть контракта. Результаты могут быть сохранены в базе данных, выведены на терминал или представлены в файловом отчете.

5.9.2 Интерфейс Pipeline

Интерфейс Pipeline служит абстрактным базовым классом, который определяет общий контракт для всех конкретных реализаций пайплайнов.

Этот контракт задает метод, который должны реализовать все производные классы, обеспечивая согласованность и предсказуемость поведения различных пайплайнов. Исходный код представлен в приложении А.9.

Интерфейс Pipeline включает в себя один абстрактный метод `run(url: str) -> Dict[str, float]`, который обязаны реализовать все подклассы. Он принимает на вход строку `url`, представляющую адрес сайта, и возвращает словарь, где ключами являются ключевые слова (в виде строк), а значениями - их веса (в виде чисел с плавающей точкой).

Аргументы: `url (str)`: адрес страницы, данные которой будут обрабатываться.

Возвращаемое значение: `Dict[str, float]`: словарь, где ключи – это извлеченные ключевые термины, а значения – их соответствующие веса.

Run определяет основную функциональность каждого пайплайна, выполняя полный цикл обработки данных от их извлечения и предварительной обработки до анализа и получения ключевых слов.

Реализация этого в конкретных подклассах обеспечивает конкретную логику обработки и позволяет интегрировать различные инструменты и методы анализа данных.

5.9.3 CommonPipeline

Класс `CommonPipeline`, унаследованный от класса `Pipeline`, обеспечивает унифицированный конвейер для классов выделения ключевых слов.

Параметры инициализации:

- `keyword_extractor` (тип `KeywordExtractor`) – основной компонент класса, отвечающий за извлечение ключевых слов из входных данных. Этот параметр позволяет использовать различные методы и алгоритмы, обеспечивая гибкость конвейера,
- `output_method` (тип `DataOutput`) – объект, определяющий путь вывода результатов конвейера. Это позволяет настраивать вывод в соответствии с конкретными требованиями, будь то сохранение в файл, вывод на экран или дальнейшая обработка.

Методы: `run(self, input_source)`: принимает входные данные, иницирует процесс извлечения ключевых слов с помощью `keyword_extractor` и настраивает вывод результатов с помощью `output_method`.

5.9.4 CommonFilePipeline

`CommonFilePipeline` предназначен для работы с текстом, загруженным из файлов. Он аналогичен `CommonPipeline`, но ориентирован на обработку текстов из локальных источников.

Методы:

- `__init__(self, keyword_extractor: KeywordExtractor, output_method: DataOutput)`: пайплайн инициализируется с экземплярами `KeywordExtractor` и `DataOutput`,
- `run(self, text)`: принимает текстовое содержимое файла в качестве входного параметра. В ходе выполнения конвейера выполняет прямую обработку текстового содержимого без необходимости получения HTML по URL, разделение текста на блоки и назначение весов с использованием `HTMLDataProcessor`, извлечение ключевых слов через

`extract_keywords_from_html_text` из `KeywordExtractor`, вывод результатов через `DataOutput`.

5.9.5 DatabaseKeywordsPipeline

`DatabaseKeywordsPipeline` используется для извлечения ключевых слов, уже сохраненных в базе данных. Этот пайплайн не выполняет стандартные стадии извлечения и обработки HTML, а работает исключительно с данными, хранящимися в базе. Исходный код представлен в приложении А.9.

- `__init__(self, db_manager: SQLiteDatabaseManager)` Пайплайн инициализируется с экземпляром `SQLiteDatabaseManager`;

- `run(self, url: str) -> Dict[str, float]`: принимает URL в качестве входного параметра. В ходе выполнения конвейера выполняет:

- 1) проверку наличия записей для данного ключа в базе данных;
- 2) извлечение данных из таблицы `keywords`;
- 3) возвращение ключевых слов в виде словаря.

5.9.6 TextRazorPipeline

Класс `TextRazorPipeline`, унаследованный от класса `Pipeline`, позволяет извлечь ключевые слова из URL с использованием `TextRazorKeywordExtractor`.

Параметры инициализации: `output_method` (тип `DataOutput`): объект, определяющий процесс вывода результатов конвейера, что позволяет настраивать вывод в соответствии с конкретными потребностями.

Атрибуты: `keyword_extractor` (тип `TextRazorKeywordExtractor`): объект, отвечающий за извлечение ключевых слов из входных данных, используя `TextRazor API` для эффективного и точного извлечения ключевой информации.

Методы: `run(self, input_source)`: извлекает ключевые слова с использованием API `TextRazor` и выводит результаты в виде словаря.

Эти пайплайны обеспечивают гибкость и возможность адаптации для различных задач извлечения и анализа ключевых терминов, используя как локальные ресурсы, так и внешние API, а также интеграцию с базой данных

для хранения и последующего извлечения результатов. Они используются как в работе приложения с фронтенда, обеспечивая универсальный процесс и простую заменяемость, так и при использовании класса-сравнителя KeywordComparator.

5.10 API и веб-интерфейс

Пользовательский интерфейс реализован с использованием FastAPI для создания веб-приложения, позволяющего пользователям извлекать ключевые слова из URL или загруженных файлов, выбирая из списка.

Форма на главной странице включает несколько компонентов: обработка загрузки данных, выбор метода извлечения, отображение результатов и обработка ошибок.

Исходный код представлен в приложении А.10.

Файловая структура сервера:

- src\web\keyword_extraction_methods.json,
- src\web\main.py,
- src\web\run.py,
- src\web\static\css\style.css,
- src\web\templates\error.html,
- src\web\templates\index.html,
- src\web\templates\results.html.

Файл конфигурации keyword_extraction_methods.json содержит список методов извлечения, доступных для использования пользователем. Каждый из них описывается именем, значением и флагом, указывающим на возможность использования файлового ввода,

Основной модуль main.py содержит маршруты и логику для обработки запросов, получения и отображения результатов,

Маршрут главной страницы отвечает за отображение начальной формы, где пользователь может ввести URL или загрузить файл, а также выбрать вариант из списка.

```
@app.get("/", response_class=HTMLResponse)
```

```
async def read_form(request: Request)
```

Маршрут `extract_keywords` обрабатывает форму, отправленную пользователем. В зависимости от выбранного метода и типа ввода (URL или файл), извлекаются ключевые слова и возвращаются результаты

```
@app.post("/extract_keywords/",
response_class=HTMLResponse)

async def extract_keywords(request: Request, url: str =
Form(None), file:
UploadFile = File(None), method: str = Form(...))
```

Шаблон `index.html` используется для отображения формы ввода, где пользователь может ввести адрес или загрузить файл, а также выбрать метод извлечения ключевых слов. Скрипт на JavaScript обрабатывает изменения в выборе и показывает или скрывает поле для загрузки файла в зависимости от выбранного в элементе `select` варианта,

Шаблон `results.html` отображает результаты извлечения ключевых слов. Результаты отображаются в виде списка вместе с их весами,

Шаблон `error.html` используется для отображения сообщений об ошибках, возникающих в процессе извлечения, например, при неверном формате файла или внутренних ошибках сервера,

Статический файл стилей `style.css` содержит стили для всех HTML-шаблонов, обеспечивая единый внешний вид интерфейса. Включает стили для формы ввода, кнопок, списка ключевых слов и сообщений об ошибках.

Скрипт `run.py` используется для запуска FastAPI приложения. С помощью `Uvicorn`, приложение запускается на локальном сервере и готово к приему запросов на порт 8000.

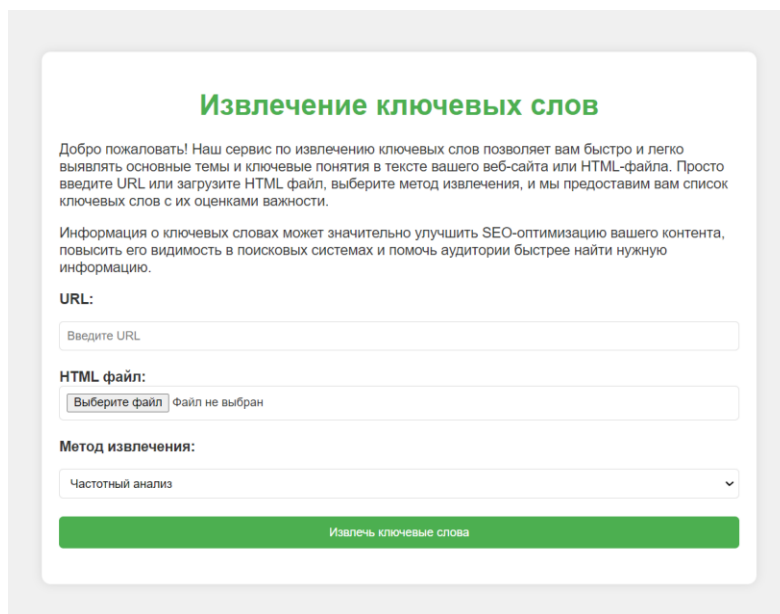
Сценарий пользовательского процесса:

- пользователь переходит на главную страницу, где видит форму для ввода ссылки или загрузки HTML-файла и выбора метода извлечения,
- пользователь вводит URL или загружает файл, выбирает вариант из списка и нажимает кнопку для начала процесса,

- приложение обрабатывает введенные данные, извлекает ключевые слова с использованием выбранного и отображает результаты на новой странице,
- если происходит ошибка (например, файл неверного формата), пользователю показывается страница с сообщением об ошибке.

Если скрипт запущен напрямую, в конце файла вызывается `uvicorn.run` для запуска сервера FastAPI на порту 8000 с автоматической перезагрузкой при изменении исходного кода.

На рисунке 2 представлен главный экран пользовательского интерфейса. Можно заметить краткое описание сервиса в двух параграфах.



The screenshot shows a web application titled "Извлечение ключевых слов" (Extraction of key words) in green. Below the title, there is a welcome message in Russian explaining the service's purpose: to quickly and easily identify main topics and key concepts in a website's text or HTML file. It instructs the user to enter a URL or upload an HTML file, choose an extraction method, and receive a list of key words with importance ratings. A second paragraph states that this information can significantly improve SEO optimization, increase visibility in search engines, and help the audience find needed information faster. The form includes three input sections: "URL:" with a text box labeled "Введите URL"; "HTML файл:" with a file selection button labeled "Выберите файл" and a status indicator "Файл не выбран"; and "Метод извлечения:" with a dropdown menu currently set to "Частотный анализ". At the bottom, there is a prominent green button labeled "Извлечь ключевые слова".

Рисунок 2 – Главная страница "/"

На рисунке 3 представлен экран результатов по запросу для <https://pnzgu.ru>. На экране присутствует краткое описание использованного метода.

Ключевые слова для <https://pnzgu.ru>

Использованный метод: FastText

Метод, основанный на использовании предобученной модели FastText для извлечения ключевых слов.

университет:	1.0
государственный:	0.9834051686016908
новости:	0.8498297851687031
качества:	0.6853976517911459
другие:	0.6835812565249713
комиссия:	0.6821326317238854
сайте:	0.6688535621551657
наука:	0.6481093436410672
образование:	0.6382007338473465
специалистов:	0.6290069398175487

Назад

Рисунок 3 – Страница результатов `"/extract_keywords"`

6 Оценка качества результатов

6.1 Интерфейс KeywordComparator

Директория comparator содержит компоненты, предназначенные для количественного сравнения и оценки схожести двух наборов ключевых слов, полученных различными методами выделения. Основная цель этого процесса – определить, насколько сходны результаты, полученные разными методами, при обработке одного и того же источника. Листинг файлов и исходный код представлены в приложении А.11.

Процесс количественного сравнения и оценки схожести двух наборов ключевых слов заключается в вычислении числового значения, которое отражает степень их сходства. Это может быть полезно для нескольких целей:

- сравнение различных методов позволяет понять, насколько разные способы выделения (например, TF-IDF, TextRank, TextRazor и др.) дают схожие результаты при анализе одного и того же текста. Может использоваться для оценки точности и согласованности выходных данных,
- валидация и тестирование новых методов помогает разработчикам оценивать новые реализации выделения ключевых слов, сравнивая их с уже существующими и проверенными. Упрощает процесс валидации новых алгоритмов,
- идентификация ключевых различий позволяет выявить ключевые различия между методами, что может быть полезно для дальнейшего улучшения и оптимизации алгоритмов.

Для реализации сравнения был разработан абстрактный класс KeywordComparator, который определяет необходимый интерфейс для всех классов-компараторов (приложение 7).

Описание интерфейса: compare(self, url: str) -> float – абстрактный метод, который должен быть реализован в подклассах. Принимает адрес в качестве входного параметра. Возвращает числовое значение типа float, представляющее степень схожести между двумя наборами ключевых слов.

Этот интерфейс служит шаблоном для создания различных компараторов, обеспечивая гибкость и расширяемость системы. Любой новый способ сравнения может быть легко интегрирован в систему путем создания нового класса, который реализует этот интерфейс.

6.2 FastTextComparator

FastTextComparator это класс, предназначенный для сравнения наборов взвешенных лексем, извлеченных двумя различными пайплайнами, с использованием косинусного сходства между их взвешенными средними векторами. Класс позволяет оценить степень схожести двух наборов ключевых слов, предоставленных различными источниками или методами извлечения.

Интерфейс KeywordComparator не определяет алгоритм, по которому будет подсчитана числовая оценка близости лексем, это остается на наследниках. В этом классе, чтобы получить числовое значение, которое отражало бы степень близости по смыслу, мы можем воспользуемся зарекомендовавшей себя метрикой в виде косинусного сходства векторов. Это предполагает выполнение двух операций: векторизации и вычисления близости. Векторизатор можно использовать любой, среди тех, которые демонстрируют способность отражать семантические особенности слов, и которые подходят к условиям вашей работы. В нашем случае это мультиязычность, чему соответствует уже знакомая нам fastText.

Имея два вектора, которые отражают смысл терминов, можно воспользоваться линейной алгеброй: угол между этими векторами в n -мерном пространстве является мерой их расхождения, а косинус этого угла – числовым выражением близости векторов. Дополнительным удобством этого метода является тот факт, что значение косинуса для угла между совпадающими векторами равно 1.0, а для ортогональных (что можно считать полной несвязностью исходных слов) – 0.0 [42], что и является контрактом, определенным интерфейсом KeywordExtractor. Нам не потребуется осуществлять дополнительную нормализацию значения как в случае, если мы

воспользуемся альтернативным способом, например, скалярным произведением векторов [42].

Основные компоненты:

- `def __init__(self, pipeline1: Pipeline, pipeline2: Pipeline)`. При инициализации класс принимает два объекта типа `Pipeline`, которые будут использоваться для извлечения ключевых слов из одного и того же источника. Также загружается предобученная модель `FastText` из библиотеки `Gensim` для представления лексем в виде векторов,

- `def get_word_vector(self, word: str) -> np.ndarray`. Метод `get_word_vector` принимает слово и возвращает его векторное представление, используя предобученную модель `FastText`. Если значение отсутствует в словаре модели, возвращается вектор из нулей той же размерности, что и вектора модели,

- `def compute_weighted_average_vector(self, keywords: Dict[str, float]) -> np.ndarray`. Принимает словарь ключевых слов и их весов и вычисляет взвешенный средний вектор для всего набора. Если все вектора слов оказались пустыми (нулевыми), возвращается нулевой вектор,

- `def compare(self, url: str) -> float`. `Compare` принимает URL в качестве аргумента и извлекает ключевые слова, используя оба пайплайна (`pipeline1` и `pipeline2`). Затем вычисляются взвешенные средние вектора для каждого набора. Если один из векторов пустой, возвращается ошибка.

В противном случае, вычисляется косинусное сходство между двумя векторами и возвращается результат.

6.3 Сценарий количественной оценки методов

Для сравнения результатов извлечения ключевых слов различными методами, разработанными в главе 5, используется скрипт `comparison.py`, Этапы выполнения сценария:

- инициализация логов: настройка регистрации ошибок в файле `logging/errors.log` с помощью модуля `logging`;

- загружается модель FastText ("fasttext-wiki-news-subwords-300") с помощью gensim.downloader;
- создается экземпляр SQLiteDatabaseManager для управления базой данных;
- создается экземпляр TextRankExtractor с DB_URL_Input для извлечения по URL;
- создается конвейер CommonPipeline с использованием TextRankExtractor и TranformOutput для преобразования выходных данных;
- создается конвейер DatabaseKeywordsPipeline для работы с ключевыми словами из базы данных;
- создается экземпляр FastTextComparator для сравнения ключевых слов с использованием конвейеров и модели FastText;
- итеративная обработка:
 - 1) цикл извлекает порции записей из базы данных с использованием db_manager.get_urls_chunk;
 - 2) для каждого адреса выполняется сравнение с использованием comparator.compare;
 - 3) вычисляется среднее значение similarity_score для всех обработанных записей;
 - 4) выводится информация о методе, количестве обработанных адресов и среднем значении similarity_score.

6.4 Экспериментальная оценка алгоритмов

Целью данного эксперимента является сравнение различных методов выделения ключевых слов на основе их способности идентифицировать релевантные понятия из веб-страниц и сопоставить их с целевыми значениями из базы данных. В качестве метрики для оценки используется среднее значение косинусного сходства между результатами, полученными тем или иным методом, и записями из базы данных.

В эксперименте были использованы следующие классы для выделения ключевых слов:

- SimpleKeywordExtractor базовый метод выделения на основе частотного анализа,
- TextRankKeywordExtractor: метод на основе алгоритма TextRank, который использует графовую модель,
- TFIDFKeywordExtractor: основанный на использовании взвешенной частоты термина, умноженной на обратную частоту документа (TF-IDF),
- TextRazorKeywordExtractor: использующий сервис TextRazor для извлечения с учетом контекста и семантического анализа,
- FastTextKeywordExtractor: использующий модель FastText для представления лексем в виде векторов и анализа их сходства.

Описание эксперимента

Эксперимент был проведен с использованием скрипта `src/util/comparison.py`, который обрабатывал URL-адреса, извлекая ключевые слова с помощью каждого из упомянутых методов и сравнивая их с записями, хранящимися в базе данных. Для сравнения использовалась метрика косинусного сходства, вычисляемая с помощью класса `FastTextComparator`.

Ход эксперимента:

- инициализация моделей и загрузка данных:
 - 1) модель FastText была загружена для векторизации лексем;
 - 2) менеджер базы данных SQLite использовался для получения данных из таблицы `keywords`;
 - 3) для каждого метода выделения ключевых слов был создан соответствующий экземпляр класса;
 - обработка:
 - 1) URL из базы данных обрабатывались партиями по 10;
 - 2) для каждой записи извлекались ключевые слова;
 - 3) вычислялось косинусное сходство между массивами, извлеченными исследуемым классом, и из базы данных;
 - 4) результаты сходства сохранялись для дальнейшего анализа;
 - анализ результатов:

1) после обработки всех адресов вычислялось среднее значение косинусного сходства для каждого метода;

2) также учитывалось количество успешных сравнений, где удалось извлечь и сопоставить ключевые слова.

Результаты эксперимента для каждого из способов извлечения представлены в таблице 10:

Таблица 10 – Результаты эксперимента

Метод извлечения	Число успешных результатов	Среднее косинусное сходство
Частотный анализ	736	0.5758
TextRank	680	0.6322
TFIDF	723	0.5995
TextRazor	568	0.3247
FastText	723	0.6523

Исходя из результатов эксперимента, можно сделать следующие выводы.

FastText показал самое высокое среднее косинусное сходство (0.6523), что указывает на его высокую способность к идентификации релевантных ключевых слов. Это может быть обусловлено использованием модели FastText для представления лексем в виде векторов и анализа их сходства.

TextRank и TFIDF показали сопоставимые результаты среднего косинусного сходства со значениями 0.6322 и 0.5995 соответственно.

Это говорит о том, что оба метода эффективны в выделении ключевых слов из HTML кода.

Метод частотного анализа также продемонстрировал неплохие результаты со средним косинусным сходством 0.5758, что говорит о его относительной эффективности на данной выборке с данными значениям весов по тэгам (таблица 1).

TextRazor показал наименьшее среднее косинусное сходство (0.3247), что может указывать на его ограниченную способность к выделению

релевантных понятий. Это можно объяснить тем, что сервис предназначен для выделения ключевых слов из текста, а функционал загрузки документа по URL хоть и включает очистку HTML и выдает адекватные результаты, не так точна при анализе семантически неравных участков текста.

ЗАКЛЮЧЕНИЕ

В ходе настоящего исследования была достигнута основная цель – разработка и анализ методов машинного обучения для автоматического извлечения ключевых слов из веб-страниц. Исследование включало всесторонний обзор существующих решений в данной области, а также оценку их применимости и ограничений в контексте специфики веб-контента.

В первой части работы был проведен обзор предметной области, в том числе существующих решений по извлечению ключевых слов. Рассмотрены цели и задачи исследования, темы, а также методы машинного обучения, применяемые в подобных задачах.

На этапе формирования данных были выбраны и подготовлены необходимые наборы данных, организована структура таблиц в базе данных. Важной частью работы стало проектирование архитектуры решения для извлечения ключевых слов из HTML документов, включающей пользовательский интерфейс, модули загрузки и обработки, выделения ключевых слов и взаимодействие между этими модулями.

Разработанный алгоритм работы приложения включает этапы загрузки начальных данных, чтения и обработки кода, выделения ключевых слов, а также алгоритм сбора данных для обучения и тестирования модели. Эти этапы были реализованы в программном решении, структура которого охватывает сценарии загрузки и разбиения документов, управление базой данных, выделение ключевых слов и стратегии их вывода. Разработана классовая модель, обеспечивающая модульность компонентов системы и применимость функционала как посредством пользовательского интерфейса, так и скриптами (приложение Б).

Были рассмотрены и проанализированы существующие алгоритмы, такие как частотный метод, TF-IDF, TextRank. Выполнена адаптация существующего сервиса по выделению ключевых слов с подключением по API. Разработан новый метод извлечения ключевых слов с использованием векторизации FastText и кластеризации DBSCAN. Проведен сравнительный

анализ этих методов, выявлены их преимущества и недостатки в контексте обработки веб-контента.

Проведенная оценка качества предложенных методов показала, что разработанное решение позволяет значительно улучшить точность и полноту извлечения ключевых слов по сравнению с традиционными подходами. Экспериментальные результаты подтвердили эффективность использования методов машинного обучения для данной задачи и продемонстрировали потенциал их практического применения.

Полученные результаты имеют высокую практическую значимость и могут быть использованы для создания интеллектуальных систем автоматического извлечения ключевой информации из интернет-ресурсов. Это, в свою очередь, способствует улучшению качества информационного поиска и анализа в сети Интернет, повышая удобство и эффективность работы пользователей с большими объемами данных.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

- 1 Основы HTML - Изучение веб-разработки | MDN [Электронный ресурс] – URL: https://developer.mozilla.org/ru/docs/Learn/Getting_started_with_the_web/HTML_basics (дата обращения 28.12.2023)
- 2 Сычев А.В. Web-технологии [Электронный ресурс] / Сычев А.В.– Электрон. текстовые данные.– М.: Интернет-Университет Информационных Технологий (ИНТУИТ), 2016.– 184 с.
- 3 MDN Web Docs (HTML Element Reference) - Mozilla Developer Network. [Электронный ресурс]. – URL: <https://developer.mozilla.org/en-US/docs/Web/HTML/Element> (3)
- 4 Пензенский государственный университет [Электронный ресурс] – URL: <https://pnzgu.ru/> (дата обращения 01.06.2023)
- 5 SEO-копирайтинг 2.0. Как писать тексты в эру семантического поиска : руководство / И. С. Шамина, А. С. Ушакова, Е. А. Ткаченко [и др.] ; под общей редакцией И. С. Шаминой. – Вологда : Инфра-Инженерия, 2018. – 260 с.
- 6 Углубленное руководство по принципам работы Google Поиска | Центр Google Поиска | Документация | Google for Developers. – [Электронный ресурс]. URL: <https://developers.google.com/search/docs/fundamentals/how-search-works>
- 7 SEO-Optimization using Transformer Models - Squadra Machine Learning Company – [Электронный ресурс]. URL: <https://machine-learning-company.nl/en/technisch-en/seo-optimization-using-transformer-models/>
- 8 Ванюшкин А.С., Гращенко Л.А., Методы и алгоритмы извлечения ключевых слов. – Новые информационные технологии в автоматизированных системах, 2016 – 85–93 с.
- 9 Неелова, Н. SEMBOOK. Энциклопедия поискового продвижения / Н. Неелова, К. Шпорт, А. Моргачева и др. – Спб.: Питер, 2014. – 520 с.
- 10 Находите подходящие ключевые слова с помощью Планировщика ключевых слов – Google Реклама [Электронный ресурс] – URL:

<https://ads.google.com/intl/ru/home/tools/keyword-planner/> (дата обращения 04.01.2024)

11 Вордстат [Электронный ресурс] – URL: <https://wordstat.yandex.ru/> (дата обращения 04.01.2024)

12 Features | Semrush [Электронный ресурс] – URL: <https://www.semrush.com/features/> (дата обращения 04.01.2024)

13 Ahrefs - SEO Tools & Resources To Grow Your Search Traffic [Электронный ресурс] – URL: <https://ahrefs.com/> (дата обращения 04.01.2024)

14 The SEO Keyword Research Master Guide - Moz [Электронный ресурс] – URL: <https://moz.com/keyword-research-guide> (дата обращения 04.01.2024)

15 TextRazor - Python API Documentation [Электронный ресурс] – URL: <https://www.textrazor.com/docs/python> (дата обращения 06.01.2024)

16 WebCEO | SEO & Digital Marketing Tools [Электронный ресурс] – URL: <https://www.webceo.com/> (дата обращения 04.01.2024)

17 Anna Rogers, Olga Kovaleva, и Anna Rumshisky. 2020. A Primer in BERTology: What We Know About How BERT Works. Transactions of the Association for Computational Linguistics, 842–866 с.

18 Open Sourcing BERT: State-of-the-Art Pre-training for Natural Language Processing – [Электронный ресурс]. URL: <https://research.google/blog/open-sourcing-bert-state-of-the-art-pre-training-for-natural-language-processing/>

19 Рашка, С. Python и машинное обучение: крайне необходимое пособие по новейшей предсказательной аналитике, обязательное для более глубокого понимания методологии машинного обучения : руководство / С. Рашка ; перевод с английского А. В. Логунова. – Москва : ДМК Пресс, 2017. – 418 с.

20 Автоматическая обработка текстов на естественном языке и анализ данных : учеб. пособие / Большакова Е.И., Воронцов К.В., Ефремова Н.Э., Клышинский Э.С., Лукашевич Н.В., Сапин А.С. – М.: Изд-во НИУ ВШЭ, 2017. – 269 с.

- 21 Semantics - Глоссарий MDN Web Docs: определения веб-терминов | MDN [Электронный ресурс] – URL: <https://developer.mozilla.org/ru/docs/Glossary/Semantics> (дата обращения 10.01.2024)
- 22 Энг, Э., Спенсер, С., Стриккиола, Дж. The Art of SEO: Mastering Search Engine Optimization / Manning; First Edition., 2019. – 544 с.
- 23 Feature extraction – scikit-learn 1.4.2 documentation – [Электронный ресурс]. URL: https://scikit-learn.org/stable/modules/feature_extraction.html#text-feature-extraction
- 24 Лейн, Г., Хапке, Х., Ховард, К. Natural Language Processing in Action: Understanding, analyzing, and generating text with Python / O'Reilly Media, Inc., 2015. – 994 с.
- 25 Н.Ю. Рязанова. Анализ вопросов автоматизации поиска информации – Вестник МГТУ им. Н.Э. Баумана: электронное издание. 2013 – 1–5 с.
- 26 Шереметьева, С.О. МЕТОДЫ И МОДЕЛИ АВТОМАТИЧЕСКОГО ИЗВЛЕЧЕНИЯ КЛЮЧЕВЫХ СЛОВ / С.О. Шереметьева, П.Г. Осминин // Вестник Южно-Уральского государственного университета. Серия: Лингвистика. – 2015. – № 1. – С. 76-81.
- 27 Enriching Word Vectors with Subword Information Piotr Bojanowski* and Edouard Grave* and Armand Joulin and Tomas Mikolo Facebook AI Research[Электронный ресурс] – URL: <https://arxiv.org/pdf/1607.04606v2> (дата обращения 12.05.2024)
- 28 Мюллер А., Гвидо С. Введение в машинное обучение с помощью Python. – СПб.: Вильямс, 2019. – 480 с.
- 29 Рашка С., Мирджалили В. Python и машинное обучение: машинное и глубокое обучение с использованием Python, scikit-learn и TensorFlow 2. – СПб.: ООО "Диалектика", 2020. – 848 с.
- 30 Effective SEO parameters for all types of websites [Электронный ресурс]. – URL: <https://www.kaggle.com/datasets/ashkangoharfar/sites-information-data-from-alexacom-dataset>

31 Most Visited websites | Kaggle [Электронный ресурс] – URL:<https://www.kaggle.com/datasets/mathurinache/mostvisitedwebsites> (дата обращения 22.02.2024)

32 Common Crawl - Get Started [Электронный ресурс] – URL: <https://commoncrawl.org/get-started> (дата обращения 14.04.2024)

33 The Directory of the Web [Электронный ресурс] – URL:<https://dmoztools.net/> (дата обращения 14.04.2024)

34 Web Data Commons [Электронный ресурс] – URL: <https://webdatacommons.org/> (дата обращения 14.04.2024)

35 Multimedia Commons [Электронный ресурс] – URL:<http://www.multimediacommons.org/> (дата обращения 18.04.2024)

36 YFCC100M – Communications of the ACM [Электронный ресурс] – URL:<https://cacm.acm.org/research/yfcc100m/#FNA> (дата обращения 23.05.2024)

37 Top Websites Ranking in the World, June 2024 [Электронный ресурс] – URL:<https://ahrefs.com/top> (дата обращения 03.06.2024)

38 Белугина, С. В. Разработка программных модулей программного обеспечения для компьютерных систем. Прикладное программирование : учебное пособие / С. В. Белугина. – Санкт-Петербург : Лань, 2020.

39 HTML Standard. [Электронный ресурс]. – URL: <https://html.spec.whatwg.org/> (дата обращения 03.01.2024)

40 Машинное обучение (курс лекций, К.В.Воронцов) [Электронный ресурс] – URL: [http://www.machinelearning.ru/wiki/index.php?title=Машинное_обучение_\(курс_лекций%2C_К.В.Воронцов\)](http://www.machinelearning.ru/wiki/index.php?title=Машинное_обучение_(курс_лекций%2C_К.В.Воронцов)) (дата обращения 11.05.2024)

41 Журавель Александр Сергеевич Программа автоматизированной генерации ключевых слов на основе коллекции веб-документов // Образовательные ресурсы и технологии. 2015. №1 (9).

42 Singhal, Amit. "Modern Information Retrieval: A Brief Overview" – Bulletin of the IEEE Computer Society Technical Committee on Data Engineering 24 (4), 2001 – 35–43 с.

ПРИЛОЖЕНИЕ А

(обязательное)

Применение методов машинного обучения для автоматического извлечения
ключевых слов из веб-страниц

Листинг программы

ПРИЛОЖЕНИЕ А.1

Сценарий загрузки документов

Текст файла src/util/url_text.py

```
import sqlite3
import requests

conn = sqlite3.connect('database/keywords.db')
cursor = conn.cursor()

cursor.execute('''
CREATE TABLE IF NOT EXISTS url_html (
    url TEXT PRIMARY KEY,
    html_content TEXT
)
''')
conn.commit()

def fetch_and_store_html(url):
    cursor.execute('SELECT COUNT(*) FROM url_html WHERE url =
?', (url,))
    count = cursor.fetchone()[0]

    if count == 0:
        try:
            response = requests.get(url)
            response.raise_for_status()
            html_content = response.text

            cursor.execute('''
INSERT OR REPLACE INTO url_html (url, html_content)
VALUES (?, ?)
''', (url, html_content))
            conn.commit()
        except requests.RequestException as e:
            print(f"Error fetching {url}: {e}")
        else:
            print(f'Skipping {url}')

cursor.execute('SELECT DISTINCT url FROM keywords')
urls = cursor.fetchall()

for url in urls:
    fetch_and_store_html(url[0])
```

ПРИЛОЖЕНИЕ А.2

Сценарий разбиения HTML документов

Текст файла src/util/extract_text_blocks.py

```
from src.database_manager.sqlite_database_manager import
SQLiteDatabaseManager
from src.data_processing.html_data_processor import
HTMLDataProcessor
from src.data_processing.element_weight_assigner import
ElementWeightAssigner

def extract_text_blocks_and_save(db_manager):
    weight_assigner = ElementWeightAssigner()
    html_processor = HTMLDataProcessor(weight_assigner)

    url_html_data = db_manager.get_url_html_data()

    text_blocks_to_insert = []
    for url, html_content in url_html_data:
        text_blocks = html_processor.process(html_content)
        for text_block, weight in text_blocks:
            text_blocks_to_insert.append((url, text_block,
weight))

    db_manager.save_text_blocks(text_blocks_to_insert)

    print(f"Записаны {len(text_blocks_to_insert)} блоков.")

if __name__ == '__main__':
    db_manager = SQLiteDatabaseManager('database/keywords.db')
    extract_text_blocks_and_save(db_manager)
    db_manager.close()
```

ПРИЛОЖЕНИЕ А.3

Сценарий выделения ключевых слов

Текст файла src\util\use_urls.py

```
import os
import logging
from data_output.terminal_output import TerminalOutput
from data_output.database_output import DatabaseOutput
from data_output.db_and_terminal import DbAndTerminalOutput
from src.keyword_extractor.textrazor_keyword_extractor import
TextRazorKeywordExtractor
from src.keyword_extractor.keyword_extractor_with_db import
KeywordExtractorWithDB
from src.database_manager.sqlite_database_manager import
SQLiteDatabaseManager
from src.pipeline.common_pipeline import CommonPipeline

if not os.path.exists('logs'):
    os.makedirs('logs')

logging.basicConfig(filename='logs/errors.log',
level=logging.ERROR,
format='%(asctime)s: %(levelname)s: %(message)s')

def process_urls():
    db_manager = SQLiteDatabaseManager()
    keyword_extractor =
KeywordExtractorWithDB(TextRazorKeywordExtractor(), db_manager)
    output_method =
DbAndTerminalOutput(terminal_output=TerminalOutput(),
db_output=DatabaseOutput(db_manager))
    pipeline = CommonPipeline(keyword_extractor, output_method)

    with open('urls.txt') as file:
        urls = [line.strip() for line in file]
        for url in urls:
            try:
                print(f"Processing URL: {url}")
                pipeline.run(url)
            except Exception as e:
                logging.error(f"Error processing URL {url}:
{e}")

if __name__ == "__main__":
    process_urls()
```

ПРИЛОЖЕНИЕ А.4

Класс управления базой данных

Текст файла src\database_manager\database_manager.py

```
from abc import ABC, abstractmethod

class DatabaseManager(ABC):
    @abstractmethod
    def url_exists(self, url: str) -> bool:
        pass

    @abstractmethod
    def save_keywords(self, url: str, keywords: dict):
        pass

    @abstractmethod
    def save_history(self, url: str):
        pass

    @abstractmethod
    def save_error(self, url: str, error_message: str):
        pass

    @abstractmethod
    def get_text_blocks(self, url: str) -> list:
        pass

    @abstractmethod
    def save_text_blocks(self, url: str, text_blocks: list):
        pass

    @abstractmethod
    def get_keywords_by_url(self, url: str) -> list:
        pass

    @abstractmethod
    def get_urls_chunk(self, chunk_size: int = 1) -> list:
        pass

    @abstractmethod
    def get_url_html_data(self) -> list:
        pass

    @abstractmethod
    def get_html_content(self, url: str) -> str:
        pass
```

Текст файла src\database_manager\sqlite_database_manager.py

```
import sqlite3
from src.database_manager.database_manager import
DatabaseManager

class SQLiteDatabaseManager(DatabaseManager):
    def __init__(self, db_name='database/keywords.db'):
        self.conn = sqlite3.connect(database=db_name)
        self.c = self.conn.cursor()
        self.create_tables()

    def create_tables(self):
        self.c.execute('''
            CREATE TABLE IF NOT EXISTS keywords (
                url TEXT,
                keyword TEXT,
                priority REAL DEFAULT NULL
            )
        ''')
        self.c.execute('''
            CREATE TABLE IF NOT EXISTS history (
                url TEXT PRIMARY KEY,
                error TEXT
            )
        ''')
        self.c.execute('''
            CREATE TABLE IF NOT EXISTS text_blocks (
                id INTEGER PRIMARY KEY,
                url TEXT,
                text_block TEXT,
                weight REAL
            )
        ''')
        self.c.execute('''
            CREATE TABLE IF NOT EXISTS url_html (
                url TEXT PRIMARY KEY,
                html_content TEXT
            )
        ''')

    def url_exists(self, url):
        self.c.execute('SELECT 1 FROM history WHERE url = ?
LIMIT 1', (url,))
        return self.c.fetchone() is not None

    def save_keywords(self, url, keywords):
        sorted_keywords = sorted(keywords.items(), key=lambda
item: item[1], reverse=True)[:10]
        valid_keywords = [group for group in sorted_keywords if
group[1] > 0.2]
        if not valid_keywords:
            raise ValueError("Ключевые слова не найдены")
```

```

        for keyword, priority in valid_keywords:
            self.c.execute('INSERT INTO keywords (url, keyword,
priority) VALUES (?, ?, ?)', (url, keyword, priority))
            self.conn.commit()

    def save_history(self, url):
        self.c.execute('INSERT OR IGNORE INTO history (url)
VALUES (?)', (url,))
        self.conn.commit()

    def save_error(self, url, error_message):
        self.c.execute('INSERT OR REPLACE INTO history (url,
error) VALUES (?, ?)', (url, error_message))
        self.conn.commit()

    def get_text_blocks(self, url):
        self.c.execute('SELECT text_block, weight FROM
text_blocks WHERE url = ?', (url,))
        return self.c.fetchall()

    def save_text_blocks(self, url, text_blocks):
        self.c.executemany('INSERT INTO text_blocks (url,
text_block, weight) VALUES (?, ?, ?)',
                           [(url, text, weight) for text, weight
in text_blocks])
        self.conn.commit()

    def get_keywords_by_url(self, url):
        self.c.execute('SELECT keyword, priority FROM keywords
WHERE url = ?', (url,))
        return self.c.fetchall()

    def get_urls_chunk(self, chunk_size, offset=0):
        self.c.execute('SELECT DISTINCT url FROM keywords LIMIT
? OFFSET ?', (chunk_size, offset))
        return [row[0] for row in self.c.fetchall()]

    def get_url_html_data(self):
        self.c.execute('SELECT url, html_content FROM url_html')
        return self.c.fetchall()

    def get_html_content(self, url):
        self.c.execute('SELECT html_content FROM url_html WHERE
url = ?', (url,))
        result = self.c.fetchone()
        if result:
            return result[0]
        else:
            raise ValueError(f"HTML документ не найден для URL:
{url}")

    def close(self):
        self.conn.close()

```

ПРИЛОЖЕНИЕ А.5

Классы входной обработки данных

Текст файла src\data_input\data_input.py

```
from abc import ABC, abstractmethod

class DataInput(ABC):
    @abstractmethod
    def get_data(self, source: str) -> str:
        pass
```

Текст файла src\data_input\file_input.py

```
from src.data_input.data_input import DataInput

class FileInput(DataInput):
    def get_data(self, file_path: str) -> str:
        try:
            with open(file_path, 'r', encoding='utf-8') as file:
                return file.read()
        except FileNotFoundError:
            raise ValueError(f"Файл не найден по пути: {file_path}")
        except Exception as e:
            raise ValueError(f"Ошибка при чтении файла по пути {file_path}: {e}")
```

Текст файла src\data_input\url_input.py

```
from src.data_input.data_input import DataInput
import requests

class URLInput(DataInput):
    def get_data(self, url: str) -> str:
        try:
            response = requests.get(url)
            response.raise_for_status()
            return response.text
        except requests.RequestException as e:
            raise ValueError(f"Ошибка при получении данных по URL {url}: {e}")
```

Текст файла src\data_input\db_url_input.py

```
from src.data_input.data_input import DataInput
from src.database_manager.sqlite_database_manager import
SQLiteDatabaseManager
```

```
class DB_URL_Input(DataInput):
    def __init__(self, db_manager):
        self.db_manager = db_manager

    def get_data(self, url):
        try:
            html_content = self.db_manager.get_html_content(url)
            return html_content
        except ValueError as e:
            print(f"Error: {e}")
            return None
```

ПРИЛОЖЕНИЕ А.6

Классы выходной обработки данных

Текст файла src\data_output\data_output.py

```
from abc import ABC, abstractmethod

class DataOutput(ABC):
    @abstractmethod
    def output_data(self, keywords: dict, source: str) -> dict:
        pass
```

Текст файла src\data_output\database_output.py

```
from src.data_output.data_output import DataOutput

class DatabaseOutput(DataOutput):
    def __init__(self, db_manager):
        self.db_manager = db_manager

    def output_data(self, keywords: dict, source: str):
        self.db_manager.save_keywords(source, keywords)
        return keywords
```

Текст файла src\data_output\db_and_terminal.py

```
from src.data_output.data_output import DataOutput
from src.data_output.database_output import DatabaseOutput
from src.data_output.terminal_output import TerminalOutput

class DbAndTerminalOutput(DataOutput):
    def __init__(self, db_output: DatabaseOutput,
terminal_output: TerminalOutput):
        self.db_output = db_output
        self.terminal_output = terminal_output

    def output_data(self, keywords: dict, source: str):
        self.db_output.output_data(keywords, source)
        self.terminal_output.output_data(keywords, source)
        return keywords
```

Текст файла src\data_output\terminal_output.py

```
from data_output.data_output import DataOutput

class TerminalOutput(DataOutput):
    def output_data(self, keywords: dict, source: str):
        print(f"Keywords for {source}:")
        for keyword, score in sorted(list(keywords.items()),
key=lambda x: x[1], reverse=True)[:10]):
```

```
        print(f"{keyword}: {score}")
    return keywords
```

Текст файла src\data_output\transform_output.py

```
from .data_output import DataOutput
from typing import Callable

class TranformOutput(DataOutput):
    def __init__(self, transform: Callable[[dict], dict] =
lambda x: x):
        self.transform = transform

    def output_data(self, keywords: dict, source: str):
        return self.transform(keywords)
```

ПРИЛОЖЕНИЕ А.7

Классы обработки документов

Текст файла src\data_processing\data_processor.py

```
from abc import ABC, abstractmethod

class DataProcessor(ABC):
    @abstractmethod
    def process(self, html: str) -> list:
        pass
```

Текст файла src\data_processing\element_weight_assigner.py

```
class ElementWeightAssigner:
    def __init__(self):
        self.weights = {
            'title': 50,
            'meta_keywords': 40,
            'meta_description': 40,
            'h1': 30, 'h2': 30, 'h3': 30, 'h4': 30, 'h5': 30,
            'h6': 30,
            'b': 30,
            'strong': 20,
            'i': 10, 'u': 10, 'em': 10, 'cite': 10,
            'text': 1
        }

    def get_weight(self, element_name):
        return self.weights.get(element_name,
            self.weights['text'])
```

Текст файла src\data_processing\html_data_processor.py

```
from bs4 import BeautifulSoup, Tag, Comment
from src.data_processing.element_weight_assigner import
ElementWeightAssigner
from src.data_processing.data_processor import DataProcessor

class HTMLDataProcessor(DataProcessor):
    def __init__(self, weight_assigner: ElementWeightAssigner):
        self.weight_assigner = weight_assigner

    def process(self, html: str) -> list:
        soup = self.clean_html(html)
        text_blocks = self.extract_text_blocks(soup)
        return text_blocks
```

```

def clean_html(self, html: str) -> BeautifulSoup:
    soup = BeautifulSoup(html, 'html.parser')

    for script_or_style in soup(['script', 'style', 'svg']):
        script_or_style.decompose()

    comments = soup.find_all(string=lambda text:
isinstance(text, Comment))
    for comment in comments:
        comment.extract()

    return soup

def extract_text_blocks(self, soup: BeautifulSoup) -> list:
    text_blocks = []

    if soup.title:
        text_blocks.append((soup.title.get_text(strip=True),
self.weight_assigner.get_weight('title')))

    meta_keywords = soup.find('meta', attrs={'name':
'keywords'})
    if isinstance(meta_keywords, Tag):
        content = meta_keywords.get('content')
        if content:
            text_blocks.append((content,
self.weight_assigner.get_weight('meta_keywords')))

    meta_description = soup.find('meta', attrs={'name':
'description'})
    if isinstance(meta_description, Tag):
        content = meta_description.get('content')
        if content:
            text_blocks.append((content,
self.weight_assigner.get_weight('meta_description')))

    def extract_text_recursively(element, weight=1):
        if isinstance(element, Tag):
            if element.name in ['script', 'style', 'svg',
'aside', 'nav']:
                return

            weight =
self.weight_assigner.get_weight(element.name)
            text_content = element.find(text=True,
recursive=False)

            if weight != 1 or text_content:
                text_blocks.append((text_content, weight))

            for child in element.children:
                extract_text_recursively(child, weight)

```

```
extract_text_recursively(soup.body)

return text_blocks
```

Текст файла src\data_processing\html_processor_db.py

```
from src.data_processing.element_weight_assigner import
ElementWeightAssigner
from src.data_processing.html_data_processor import
HTMLDataProcessor
from src.database_manager.sqlite_database_manager import
SQLiteDatabaseManager

class HTMLDataProcessorWrapper:
    def __init__(self):
        self.html_processor =
HTMLDataProcessor(weight_assigner=ElementWeightAssigner())
        self.db_manager = SQLiteDatabaseManager()

    def process(self, url: str, html: str) -> list:
        text_blocks = self.db_manager.get_text_blocks(url)
        if text_blocks:
            return text_blocks

        text_blocks = self.html_processor.process(html)

        self.db_manager.save_text_blocks(url, text_blocks)

        return text_blocks

    def process_html(self, html: str) -> list:
        text_blocks = self.html_processor.process(html)
        return text_blocks
```

ПРИЛОЖЕНИЕ А.8

Классы выделения ключевых слов

Текст файла src\keyword_extractor\keyword_extractor.py

```
from abc import ABC, abstractmethod

class KeywordExtractor(ABC):
    @abstractmethod
    def extract_keywords(self, url: str) -> dict:
        pass

    def train(self, data: list):
        raise NotImplementedError("Обучение не поддерживается")

    def extract_keywords_from_html_text(self, html: str):
        raise NotImplementedError("Извлечение из html документа не поддерживается, используйте URL")
```

Текст файла src\keyword_extractor\extractor.py

```
from collections import defaultdict
import nltk
from nltk.corpus import stopwords
from nltk.tokenize import word_tokenize
from nltk.tag import pos_tag
from src.data_input.data_input import DataInput
from src.data_input.url_input import URLInput
from src.data_processing.element_weight_assigner import ElementWeightAssigner
from src.data_processing.html_processor_db import HTMLDataProcessorWrapper
from src.keyword_extractor.keyword_extractor import KeywordExtractor

nltk.download('punkt')
nltk.download('stopwords')
nltk.download('averaged_perceptron_tagger')

class SimpleKeywordExtractor(KeywordExtractor):
    def __init__(self, fetcher: DataInput = URLInput()):
        self.weight_assigner = ElementWeightAssigner()
        self.fetcher = fetcher
        self.html_processor = HTMLDataProcessorWrapper()

    def fetch_html(self, url):
        return self.fetcher.get_data(url)

    def extract_and_filter_keywords(self, text_blocks):
        stop_words = set(stopwords.words(stopwords.fileids()))
```

```

keyword_dict = defaultdict(int)

for text, weight in text_blocks:
    words = word_tokenize(text.lower())
    words_pos = pos_tag(words)
    filtered_words = [
        word for word, pos in words_pos
        if word.isalnum() and word not in stop_words and
pos.startswith(('NN', 'VB', 'JJ'))
    ]

    for word in filtered_words:
        keyword_dict[word] += weight

return keyword_dict

def normalize_keyword_weights(self, keywords):
    max_priority = max(keywords.values(), default=1)
    return {k: v / max_priority for k, v in
keywords.items()}

def generate_keywords(self, text_blocks):
    keywords = self.extract_and_filter_keywords(text_blocks)
    normalized_keywords =
self.normalize_keyword_weights(keywords)
    return normalized_keywords

def extract_keywords(self, url):
    html = self.fetch_html(url)
    text_blocks = self.html_processor.process(url, html)
    keywords = self.generate_keywords(text_blocks)
    return keywords

def extract_keywords_from_html_text(self, html):
    text_blocks = self.html_processor.process_html(html)
    keywords = self.generate_keywords(text_blocks)
    return keywords

```

Текст файла src\keyword_extractor\keyword_extractor_with_db.py

```

import logging
from src.keyword_extractor.keyword_extractor import
KeywordExtractor

class KeywordExtractorWithDB(KeywordExtractor):
    def __init__(self, keyword_extractor, db_manager,
overwrite=False):
        self.keyword_extractor = keyword_extractor
        self.db_manager = db_manager
        self.overwrite = overwrite

    def extract_keywords(self, url):

```

```

        if not self.overwrite and
self.db_manager.url_exists(url):
            print(f"Skipping URL {url} as it already exists in
the database or caused an error previously.")
            raise

        try:
            keywords =
self.keyword_extractor.extract_keywords(url)
            self.db_manager.save_history(url)
            return keywords
        except Exception as e:
            error_message = str(e)
            logging.error(f"Error processing URL {url}:
{error_message}")
            self.db_manager.save_error(url, error_message)
            raise

```

Текст файла src\keyword_extractor\textrank_extractor.py

```

from collections import defaultdict
import networkx as nx
from nltk.tokenize import word_tokenize
from nltk.tag import pos_tag
from src.data_input.url_input import URLInput
from src.data_processing.html_processor_db import
HTMLDataProcessorWrapper
from src.data_processing.element_weight_assigner import
ElementWeightAssigner
from src.keyword_extractor.keyword_extractor import
KeywordExtractor

class TextRankExtractor(KeywordExtractor):
    def __init__(self):
        self.weight_assigner = ElementWeightAssigner()
        self.fetcher = URLInput()
        self.html_processor = HTMLDataProcessorWrapper()

    def build_graph(self, text_blocks):
        G = nx.Graph()

        for text, weight in text_blocks:
            tokens = self.tokenize_and_filter(text)
            for token in tokens:
                G.add_node(token)

        for i in range(len(text_blocks)):
            text_i, weight_i = text_blocks[i]
            tokens_i = set(self.tokenize_and_filter(text_i))

            for j in range(i + 1, len(text_blocks)):

```

```

        text_j, weight_j = text_blocks[j]
        tokens_j = set(self.tokenize_and_filter(text_j))

        common_tokens = tokens_i & tokens_j
        if common_tokens:
            for token in common_tokens:
                G.add_edge(token, token)

    return G

    def tokenize_and_filter(self, text):
        words = word_tokenize(text.lower())
        words_pos = pos_tag(words)
        filtered_words = [
            word for word, pos in words_pos
            if word.isalnum() and pos.startswith(('NN', 'VB',
'JJ'))
        ]
        return filtered_words

    def extract_keywords(self, url):
        html = self.fetcher.get_data(url)
        text_blocks = self.html_processor.process(url, html)
        return self.generate_keywords(text_blocks)

    def extract_keywords_from_html_text(self, html):
        text_blocks = self.html_processor.process_html(html)
        return self.generate_keywords(text_blocks)

    def generate_keywords(self, text_blocks):
        G = self.build_graph(text_blocks)
        scores = nx.pagerank(G)

        keywords = defaultdict(float)
        for token, score in scores.items():
            for text, weight in text_blocks:
                if token in self.tokenize_and_filter(text):
                    keywords[token] += score * weight

    return keywords

```

Текст файла src\keyword_extractor\textrazor_keyword_extractor.py

```

import textrazor
import logging
from src.keyword_extractor.keyword_extractor import
KeywordExtractor

class TextRazorKeywordExtractor(KeywordExtractor):
    def __init__(self, ):
        textrazor.api_key = <secret>

```

```

        self.client =
textrazor.TextRazor(extractors=["entities", "topics"])
        self.client.set_cleanup_mode("cleanHTML")
        self.client.set_classifiers(["textrazor_newscodes"])

    def extract_keywords(self, url):
        try:
            response = self.client.analyze_url(url)
            entities = list(response.entities())
            if not entities:
                raise ValueError("No keywords found in the
response")
            entities.sort(key=lambda x: x.relevance_score,
reverse=True)
            return {entity.id: entity.relevance_score for entity
in entities}
        except Exception as e:
            error_message = str(e)
            logging.error(f"Error processing URL {url}:
{error_message}")
            raise

```

Текст файла src\keyword_extractor\tfidf_extractor.py

```

from collections import defaultdict
from nltk.corpus import stopwords
from nltk.tag import pos_tag
from nltk.tokenize import word_tokenize
from sklearn.feature_extraction.text import TfidfVectorizer
from src.data_processing.html_processor_db import
HTMLDataProcessorWrapper
from src.data_processing.element_weight_assigner import
ElementWeightAssigner
from src.keyword_extractor.keyword_extractor import
KeywordExtractor
from src.data_input.url_input import URLInput

class TFIDFKeywordExtractor(KeywordExtractor):
    def __init__(self):
        self.weight_assigner = ElementWeightAssigner()
        self.html_processor = HTMLDataProcessorWrapper()
        self.fetcher = URLInput()
        self.vectorizer =
TfidfVectorizer(tokenizer=self.tokenize_text,
stop_words='english')

    def tokenize_text(self, text):
        words = word_tokenize(text.lower())
        return [word for word in words if word.isalnum()]

    def extract_keywords(self, url):
        html = self.fetch_html(url)

```

```

        text_blocks = self.html_processor.process(url, html)
        keywords = self.generate_keywords(text_blocks)
        return keywords

    def extract_keywords_from_html_text(self, html):
        text_blocks = self.html_processor.process_html(html)
        keywords = self.generate_keywords(text_blocks)
        return keywords

    def fetch_html(self, url):
        return self.fetcher.get_data(url)

    def extract_and_filter_keywords(self, text_blocks):
        stop_words = set(stopwords.words(stopwords.fileids()))
        keyword_dict = defaultdict(int)

        for text, weight in text_blocks:
            words = word_tokenize(text.lower())
            words_pos = pos_tag(words)
            filtered_words = [
                word for word, pos in words_pos
                if word.isalnum() and word not in stop_words and
pos.startswith(('NN', 'VB', 'JJ'))
            ]

            for word in filtered_words:
                keyword_dict[word] += weight

        return keyword_dict

    def normalize_keyword_weights(self, keywords):
        max_priority = max(keywords.values(), default=1)
        return {k: v / max_priority for k, v in
keywords.items()}

    def generate_keywords(self, text_blocks):
        keywords = self.extract_and_filter_keywords(text_blocks)
        normalized_keywords =
self.normalize_keyword_weights(keywords)
        return normalized_keywords

```

Текст файла src\keyword_extractor\fasttext_extractor.py

```

from typing import List, Tuple, Dict
import numpy as np
import gensim.downloader as api
import re
from nltk.corpus import stopwords
import nltk
from typing import Any
from src.data_input.data_input import DataInput
from src.data_input.url_input import URLInput

```

```

from src.data_processing.html_processor_db import
HTMLDataProcessorWrapper
from src.keyword_extractor.keyword_extractor import
KeywordExtractor
from sklearn.cluster import DBSCAN
from sklearn.metrics.pairwise import cosine_similarity

nltk.download('stopwords')
STOPWORDS = set(stopwords.words(stopwords.fileids()))

class FastTextKeywordExtractor(KeywordExtractor):
    def __init__(self, fetcher: DataInput = URLInput(), model:
Any = None):
        self.model = model
        if not self.model:
            self.model = api.load("fasttext-wiki-news-subwords-
300")
        self.fetcher = fetcher
        self.processor = HTMLDataProcessorWrapper()

    def get_word_vector(self, word) -> np.ndarray:
        try:
            return self.model[word]
        except KeyError:
            return np.zeros(self.model.vector_size)

    def extract_keywords(self, url: str) -> Dict[str, float]:
        html_text = self.fetcher.get_data(url)
        text_blocks = self.processor.process(url, html_text)
        keywords = self.generate_keywords(text_blocks)
        return keywords

    def extract_keywords_from_html_text(self, html: str) ->
Dict[str, float]:
        text_blocks = self.processor.process_html(html)
        keywords = self.generate_keywords(text_blocks)
        return keywords

    def generate_keywords(self, text_blocks: List[Tuple[str,
float]]) -> Dict[str, float]:
        word_weights = {}
        for text, weight in text_blocks:
            for word in self.filter_words(text.split()):
                if word:
                    if word in word_weights:
                        word_weights[word] = max([weight,
word_weights[word]])
                    else:
                        word_weights[word] = weight

        words = list(word_weights.keys())
        word_vectors = np.array([self.get_word_vector(word) for
word in words])

```

```

        word_weights_array = np.array([word_weights[word] for
word in words])

        dbscan = DBSCAN(eps=0.5, min_samples=10,
metric='cosine')
        clusters = dbscan.fit_predict(word_vectors)

        cluster_centers = []
        unique_clusters = set(clusters)
        for cluster_id in unique_clusters:
            cluster_points = word_vectors[clusters ==
cluster_id]
            cluster_center = np.mean(cluster_points, axis=0)
            cluster_centers.append(cluster_center)

        cluster_centers = np.array(cluster_centers)

        word_scores = {}
        for i, word in enumerate(words):
            word_vector = self.get_word_vector(word)
            word_weight = word_weights_array[i]
            max_similarity = -1

            for cluster_center in cluster_centers:
                similarity = cosine_similarity([word_vector],
[cluster_center])[0][0]
                if similarity > max_similarity:
                    max_similarity = similarity

            word_scores[word] = max_similarity * word_weight

        min_score = min(word_scores.values())
        max_score = max(word_scores.values())
        normalized_word_scores = {word: (score - min_score) /
(max_score - min_score) for word, score in word_scores.items()}

        sorted_normalized_word_scores = {k: v for k, v in
sorted(normalized_word_scores.items(), key=lambda item: item[1],
reverse=True)}

        top_keywords =
list(sorted(normalized_word_scores.items()))[:10]
        keywords = {word: score for word, score in top_keywords}

        return keywords

    def filter_words(self, words: List[str]) -> List[str]:
        return [re.sub(r'\W+', ' ', word).lower().strip() for
word in words if word.lower() not in STOPWORDS]

```

ПРИЛОЖЕНИЕ А.9

Классы конвейеров

Текст файла src\pipeline\pipeline.py

```
from abc import ABC, abstractmethod
from typing import Dict

class Pipeline(ABC):
    @abstractmethod
    def run(self, url: str) -> Dict[str, float]:
        pass
```

Текст файла src\pipeline\common_file_pipeline.py

```
from src.keyword_extractor.keyword_extractor import
KeywordExtractor
from src.data_output.data_output import DataOutput
from src.keyword_extractor.extractor import
SimpleKeywordExtractor
from src.pipeline.pipeline import Pipeline
from datetime import datetime

class CommonFilePipeline(Pipeline):
    def __init__(self, keyword_extractor: KeywordExtractor,
output_method: DataOutput):
        self.keyword_extractor = keyword_extractor
        self.output_method = output_method

    def run(self, text):
        time = datetime.now()
        keywords =
self.keyword_extractor.extract_keywords_from_html_text(text)
        return self.output_method.output_data(keywords,
source=f"file upload at {time}")
```

Текст файла src\pipeline\common_pipeline.py

```
from src.keyword_extractor.keyword_extractor import
KeywordExtractor
from src.data_output.data_output import DataOutput
from src.keyword_extractor.extractor import
SimpleKeywordExtractor
from src.pipeline.pipeline import Pipeline

class CommonPipeline(Pipeline):
    def __init__(self, keyword_extractor: KeywordExtractor,
output_method: DataOutput):
        self.keyword_extractor = keyword_extractor
        self.output_method = output_method
```

```

    def run(self, input_source):
        keywords =
self.keyword_extractor.extract_keywords(input_source)
        return self.output_method.output_data(keywords,
source=input_source)

```

Текст файла src\pipeline\database_pipeline.py

```

from src.pipeline.pipeline import Pipeline
from src.database_manager.sqlite_database_manager import
SQLiteDatabaseManager
from typing import Dict

class DatabaseKeywordsPipeline(Pipeline):
    def __init__(self, db_manager: SQLiteDatabaseManager):
        self.db_manager = db_manager

    def run(self, url: str) -> Dict[str, float]:
        keywords = self.db_manager.get_keywords_by_url(url)
        return {k: w for k, w in keywords}

```

Текст файла src\pipeline\textrazor_pipeline.py

```

from src.keyword_extractor.keyword_extractor import
KeywordExtractor
from src.data_output.data_output import DataOutput
from src.keyword_extractor.textrazor_keyword_extractor import
TextRazorKeywordExtractor
from src.pipeline.pipeline import Pipeline

class TextRazorPipeline(Pipeline):
    def __init__(self, output_method: DataOutput):
        self.keyword_extractor = TextRazorKeywordExtractor()
        self.output_method = output_method

    def run(self, input_source):
        keywords =
self.keyword_extractor.extract_keywords(input_source)
        return self.output_method.output_data(keywords,
source=input_source)

```

ПРИЛОЖЕНИЕ А.10

Пользовательский интерфейс

Текст файла src\web\keyword_extraction_methods.json

```
[
  {
    "name": "Частотный анализ",
    "value": "simple",
    "file_input": true
  },
  {
    "name": "TF-IDF",
    "value": "tfidf",
    "file_input": true
  },
  {
    "name": "TextRank",
    "value": "textrank",
    "file_input": true
  },
  {
    "name": "TextRazor",
    "value": "textrazor",
    "file_input": false
  },
  {
    "name": "FastText",
    "value": "fasttext",
    "file_input": true
  }
]
```

Текст файла src\web\main.py

```
import json
import logging
from fastapi import FastAPI, UploadFile, File, Form, Request, HTTPException
from fastapi.responses import HTMLResponse
from fastapi.staticfiles import StaticFiles
from fastapi.templating import Jinja2Templates
from src.pipeline.common_file_pipeline import CommonFilePipeline
from src.keyword_extractor.textrazor_keyword_extractor import TextRazorKeywordExtractor
from src.keyword_extractor.tfidf_extractor import TFIDFKeywordExtractor
from src.keyword_extractor.textrank_extractor import TextRankExtractor
from src.keyword_extractor.fasttext_extractor import FastTextKeywordExtractor
```

```

from src.pipeline.common_pipeline import CommonPipeline
from src.keyword_extractor.extractor import
SimpleKeywordExtractor
from src.data_output.transform_output import TransformOutput
import gensim.downloader as api

app = FastAPI()
templates = Jinja2Templates(directory="src/web/templates")
app.mount("/static", StaticFiles(directory="src/web/static"),
name="static")

model = api.load('fasttext-wiki-news-subwords-300')

def load_extraction_methods():
    with open("src/web/keyword_extraction_methods.json",
encoding='utf-8') as f:
        return json.load(f)

def get_pipeline(method: str, isFile: bool):
    output_method = TransformOutput(lambda x: {word: value for
word, value in x.items() if value > 0.3})

    if method == "tfidf":
        extractor = TFIDFKeywordExtractor()
    elif method == "textrank":
        extractor = TextRankExtractor()
    elif method == "textrazor":
        extractor = TextRazorKeywordExtractor()
    if method == "fasttext":
        extractor = FastTextKeywordExtractor(model=model)
    else:
        extractor = SimpleKeywordExtractor()

    if isFile:
        return CommonFilePipeline(extractor, output_method)
    else:
        return CommonPipeline(extractor, output_method)

@app.get("/", response_class=HTMLResponse)
async def read_form(request: Request):
    extraction_methods = load_extraction_methods()
    return templates.TemplateResponse("index.html", {"request":
request, "extraction_methods": extraction_methods})

@app.post("/extract_keywords/", response_class=HTMLResponse)
async def extract_keywords(request: Request, url: str =
Form(None), file: UploadFile = File(None), method: str =
Form(...)):
    try:
        if url:
            isFile = False
            pipeline = get_pipeline(method, isFile)
            keywords = pipeline.run(url)

```

```

        return templates.TemplateResponse("results.html",
{"request": request, "keywords": keywords, "source": url})
    elif file:
        isFile = True
        if file.content_type != "text/html":
            raise HTTPException(status_code=400,
detail="Неверный тип файла. Поддерживаются только HTML файлы.")
        content = await file.read()
        pipeline = get_pipeline(method, isFile)
        keywords = pipeline.run(content.decode('utf-8'))
        return templates.TemplateResponse("results.html",
{"request": request, "keywords": keywords, "source":
file.filename})
    else:
        raise HTTPException(status_code=400, detail="Нет
входных данных.")
    except ValueError as e:
        return templates.TemplateResponse("error.html",
{"request": request, "error_message": str(e)}, status_code=400)
    except Exception as e:
        logging.error(f"Ошибка при обработке запроса: {e}")
        return templates.TemplateResponse("error.html",
{"request": request, "error_message": "Внутренняя ошибка
сервера."}, status_code=500)

```

Текст файла src\web\run.py

```

import uvicorn

if __name__ == "__main__":
    uvicorn.run("main:app", host="0.0.0.0", port=8000,
reload=True)

```

Текст файла src\web\templates\error.html

```

<!DOCTYPE html>
<html lang="ru">
<head>
    <meta charset="UTF-8">
    <title>Ошибка</title>
    <link rel="stylesheet" href="/static/css/style.css">
</head>
<body>
    <div class="container">
        <h1>Произошла ошибка</h1>
        <p>{{ error_message }}</p>
        <a href="/" class="button">Назад</a>
    </div>
</body>
</html>

```

Текст файла src\web\templates\index.html

```
<!DOCTYPE html>
<html lang="ru">
<head>
  <meta charset="UTF-8">
  <title>Извлечение ключевых слов</title>
  <link rel="stylesheet" href="/static/css/style.css">
  <script>
    function updateFileInput() {
      const methodSelect =
document.getElementById('method');
      const selectedOption =
methodSelect.options[methodSelect.selectedIndex];
      const fileInputDiv = document.getElementById('file-
input-div');
      const fileInputWarning =
document.getElementById('file-input-warning');

      if (selectedOption.getAttribute('data-file-input')
=== 'False') {
        fileInputDiv.style.display = 'none';
        fileInputWarning.style.display = 'block';
      } else {
        fileInputDiv.style.display = 'block';
        fileInputWarning.style.display = 'none';
      }
    }

    window.onload = function() {
      updateFileInput(); // Initialize on page load
    };
  </script>
</head>
<body>
  <div class="container">
    <h1>Извлечение ключевых слов</h1>
    <p>Добро пожаловать! Наш сервис по извлечению ключевых
слов позволяет вам быстро и легко выявлять основные темы и
ключевые понятия в тексте вашего веб-сайта или HTML-файла.
Просто введите URL или загрузите HTML файл, выберите метод
извлечения, и мы предоставим вам список ключевых слов с их
оценками важности.</p>
    <p>Информация о ключевых словах может значительно
улучшить SEO-оптимизацию вашего контента, повысить его видимость
в поисковых системах и помочь аудитории быстрее найти нужную
информацию.</p>
    <form action="/extract_keywords/" method="post"
enctype="multipart/form-data" class="form">
      <label for="url">URL:</label>
```

```

        <input type="text" id="url" name="url"
placeholder="Введите URL">
        <div class="file-input-div" id="file-input-div">
            <label for="file">HTML файл:</label>
            <input type="file" id="file" name="file">
        </div>
        <label for="method">Метод извлечения:</label>
        <select id="method" name="method"
onchange="updateFileInput()">
            {% for method in extraction_methods %}
                <option value="{{ method.value }}" data-
file-input="{{ method.file_input }}">{{ method.name }}</option>
            {% endfor %}
        </select>
        <div class="file-input-warning" id="file-input-
warning" style="color:red;">Ввод файла недоступен для выбранного
метода.</div>
        <button type="submit">Извлечь ключевые
слова</button>
    </form>
</div>
</body>
</html>

```

Текст файла src\web\templates\results.html

```

<!DOCTYPE html>
<html lang="ru">
<head>
    <meta charset="UTF-8">
    <title>Результаты ключевых слов</title>
    <link rel="stylesheet" href="/static/css/style.css">
</head>
<body>
    <div class="container">
        <h1>Ключевые слова для {{ source }}</h1>
        <ul class="keywords-list">
            {% for keyword, score in keywords.items() %}
                <li>{{ keyword }}: {{ score }}</li>
            {% endfor %}
        </ul>
        <a href="/" class="button">Назад</a>
    </div>
</body>
</html>

```

Текст файла src\web\static\css\style.css

```

* {
    box-sizing: border-box;

```

```

}

body {
  font-family: Arial, sans-serif;
  background-color: #f0f0f0;
  color: #333;
  margin: 0;
  padding: 0;
}

.container {
  max-width: 800px;
  margin: 50px auto;
  padding: 20px;
  background-color: #fff;
  border-radius: 10px;
  box-shadow: 0 0 10px rgba(0, 0, 0, 0.1);
}

h1 {
  text-align: center;
  color: #4CAF50;
}

.form {
  display: flex;
  flex-direction: column;
  align-items: stretch;
  margin-bottom: 20px;
  gap: 10px;
}

.form label {
  margin-bottom: 5px;
  font-weight: bold;
}

.form input[type="text"], .form input[type="file"], .form select
{
  margin-bottom: 10px;
  padding: 8px;
  width: 100%;
  border: 1px solid #ddd;
  border-radius: 5px;
}

.form button {
  padding: 10px 20px;
  background-color: #4CAF50;
  color: white;
  border: none;
  border-radius: 5px;
  cursor: pointer;
}

```

```

    transition: background-color 0.3s ease;
}

.form button:hover {
    background-color: #45a049;
}

.keywords-list {
    list-style-type: none;
    padding: 0;
}

.keywords-list li {
    background-color: #f9f9f9;
    margin: 5px 0;
    padding: 10px;
    border-radius: 5px;
    border: 1px solid #ddd;
}

.button {
    display: inline-block;
    padding: 10px 20px;
    background-color: #4CAF50;
    color: white;
    border: none;
    border-radius: 5px;
    text-decoration: none;
    text-align: center;
    cursor: pointer;
    transition: background-color 0.3s ease;
}

.button:hover {
    background-color: #45a049;
}

```

ПРИЛОЖЕНИЕ А.11

Сценарий сравнения ключевых слов

Текст файла src\util\comparison.py

```
import os
import logging
from src.comparator.keyword_comparator import FastTextComparator
from src.data_output.transform_output import TransformOutput
from src.keyword_extractor.extractor import
SimpleKeywordExtractor
from src.keyword_extractor.textrank_extractor import
TextRankExtractor
from src.database_manager.sqlite_database_manager import
SQLiteDatabaseManager
from src.pipeline.textrazor_pipeline import TextRazorPipeline
from src.pipeline.common_pipeline import CommonPipeline
from src.pipeline.database_pipeline import
DatabaseKeywordsPipeline

if not os.path.exists('logs'):
    os.makedirs('logs')

logging.basicConfig(filename='logs/errors.log',
                    level=logging.ERROR,
                    format='%(asctime)s: %(levelname)s: %(message)s')

def process_urls():
    db_manager = SQLiteDatabaseManager()
    keyword_extractor = TextRankExtractor()
    pipeline = CommonPipeline(
        keyword_extractor,
        TransformOutput(lambda x: {word: value for word, value in
x.items() if value > 0.3})
    )
    db_keywords_pipeline = DatabaseKeywordsPipeline(db_manager)

    comparator = FastTextComparator(pipeline,
db_keywords_pipeline)

    chunk_size = 10
    offset = 0
    while True:
        urls = db_manager.get_urls_chunk(chunk_size, offset)
        if not urls:
            break
        for url in urls:
            try:
                print(f"\nProcessing URL: {url}")
                similarity_score = comparator.compare(url)
                print(f"Similarity Score: {similarity_score}")
```

```

        except Exception as e:
            logging.error(f"Error processing URL {url}:"
{e}")
        offset += chunk_size

if __name__ == "__main__":
    process_urls()

```

Текст файла src\util\comparison_from_file.py

```

import os
import logging
from src.comparator.keyword_comparator import FastTextComparator
from src.data_output.transform_output import TranformOutput
from src.keyword_extractor.extractor import
SimpleKeywordExtractor
from src.database_manager.sqlite_database_manager import
SQLiteDatabaseManager
from src.pipeline.textrazor_pipeline import TextRazorPipeline
from src.pipeline.common_pipeline import CommonPipeline
from src.pipeline.database_pipeline import
DatabaseKeywordsPipeline

if not os.path.exists('logs'):
    os.makedirs('logs')

logging.basicConfig(filename='logs/errors.log',
level=logging.ERROR,
format='%(asctime)s: %(levelname)s: %(message)s')

def process_urls():
    db_manager = SQLiteDatabaseManager()
    keyword_extractor = SimpleKeywordExtractor()
    pipeline = CommonPipeline(
        keyword_extractor,
        TranformOutput(lambda x: {word: value for word, value in
x.items() if value > 0.3})
    )
    db_keywords_pipeline = DatabaseKeywordsPipeline(db_manager)

    comparator = FastTextComparator (pipeline,
db_keywords_pipeline)

    with open('urls.txt') as file:
        urls = [line.strip() for line in file]
        for url in urls:
            try:
                print(f"\nProcessing URL: {url}")
                similarity_score = comparator.compare(url)
                print(f"Similarity Score: {similarity_score}")
            except Exception as e:

```

```

        logging.error(f"Error processing URL {url}:
{e}")

if __name__ == "__main__":
    process_urls()

```

Текст файла src\comparator\comparator.py

```

from abc import ABC, abstractmethod

class KeywordComparator(ABC):
    @abstractmethod
    def compare(self, url: str) -> float:
        pass

```

Текст файла src\comparator\fasttext_comparator.py

```

from typing import Dict
from sklearn.metrics.pairwise import cosine_similarity
import numpy as np
from src.pipeline.pipeline import Pipeline
from src.comparator.comparator import KeywordComparator
import gensim.downloader as api
from typing import Any

class FastTextComparator(KeywordComparator):
    def __init__(self, pipeline1: Pipeline, pipeline2: Pipeline,
model: Any = None):
        self.pipeline1 = pipeline1
        self.pipeline2 = pipeline2
        self.model = model
        if(not self.model):
            self.model = api.load("fasttext-wiki-news-subwords-
300")

    def get_word_vector(self, word):
        try:
            return self.model[word]
        except KeyError:
            return np.zeros(self.model.vector_size)

    def compute_weighted_average_vector(self, keywords:
Dict[str, float]):
        weighted_vectors = [
            weight * self.get_word_vector(word)
            for word, weight in keywords.items()
            if np.any(self.get_word_vector(word))
        ]
        if not weighted_vectors:
            return np.zeros(self.model.vector_size)

```

```

        return np.mean(weighted_vectors, axis=0)

    def compare(self, url: str) -> float:
        keywords1 = self.pipeline1.run(url)
        print(f"{keywords1=}")
        keywords2 = self.pipeline2.run(url)
        print(f"{keywords2=}")
        vector1 =
self.compute_weighted_average_vector(keywords1)
        vector2 =
self.compute_weighted_average_vector(keywords2)

        if vector1.shape[0] == 0 or vector2.shape[0] == 0:
            raise ValueError("Пустые векторы")

        similarity = cosine_similarity([vector1],
[vector2])[0][0]
        return similarity

```

ПРИЛОЖЕНИЕ Б

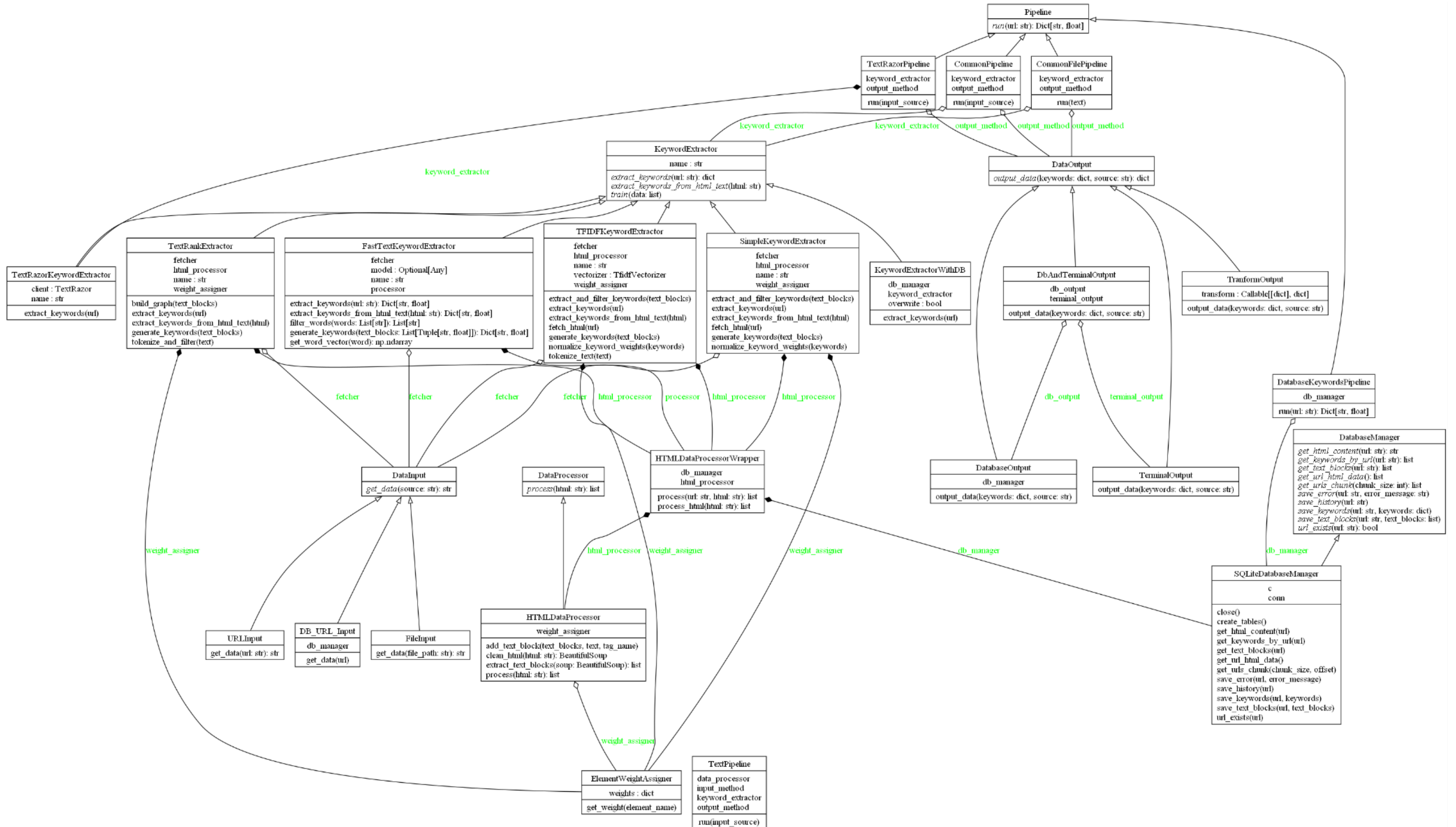
(обязательное)

Применение методов машинного обучения для автоматического извлечения

ключевых слов из веб-страниц

UML – диаграмма

Диаграмма классов



ПРИЛОЖЕНИЕ В

(обязательное)

Применение методов машинного обучения для автоматического извлечения
ключевых слов из веб-страниц

Презентация

ПРИМЕНЕНИЕ МЕТОДОВ МАШИННОГО ОБУЧЕНИЯ ДЛЯ АВТОМАТИЧЕСКОГО ИЗВЛЕЧЕНИЯ КЛЮЧЕВЫХ СЛОВ ИЗ ВЕБ-СТРАНИЦ

Автор работы:

магистрант группы 22ВВИм2 Споршев С. А.

Научный руководитель:

к.т.н. доцент Бикташев Р. А.

Слайд №1

Предмет работы

Цель, объект и предмет работы

Цель работы: разработка средства извлечения ключевых слов из веб-страниц с применением алгоритмов машинного обучения

Объект работы: программное решение, предназначенное для автоматического извлечения ключевых слов из веб-страниц с использованием различных методов анализа текста

Предмет работы: методы и алгоритмы машинного обучения, применяемые для автоматического выделения центральных понятий из веб-страниц

Слайд №2

Актуальность работы

Немного статистики:

- 1.Органический трафик:** По данным, 53% всего отслеживаемого веб-трафика поступает из органического поиска.
- 2.Глубина:** Первые пять результатов поиска получают в среднем 67.6% кликов. Результаты со второй страницы – 0.78% кликов.
- 3.Мобильный поиск:** 87% пользователей мобильных телефонов пользуются поисковиком ежедневно. При этом первое органическое место в мобильных результатах получает в среднем 27,4% всех кликов
- 4.Локальный поиск:** Примерно 46% всех поисковых запросов в Google имеют локальный характер, и 76% людей, ищущих что-то поблизости на своих смартфонах, посещают бизнес в течение дня.
- 5.Влияние на покупки:** Локальные поисковые запросы приводят к покупке в 28% случаев.

Данные - [Respona](#)

Слайд №3

Введение в предметную область

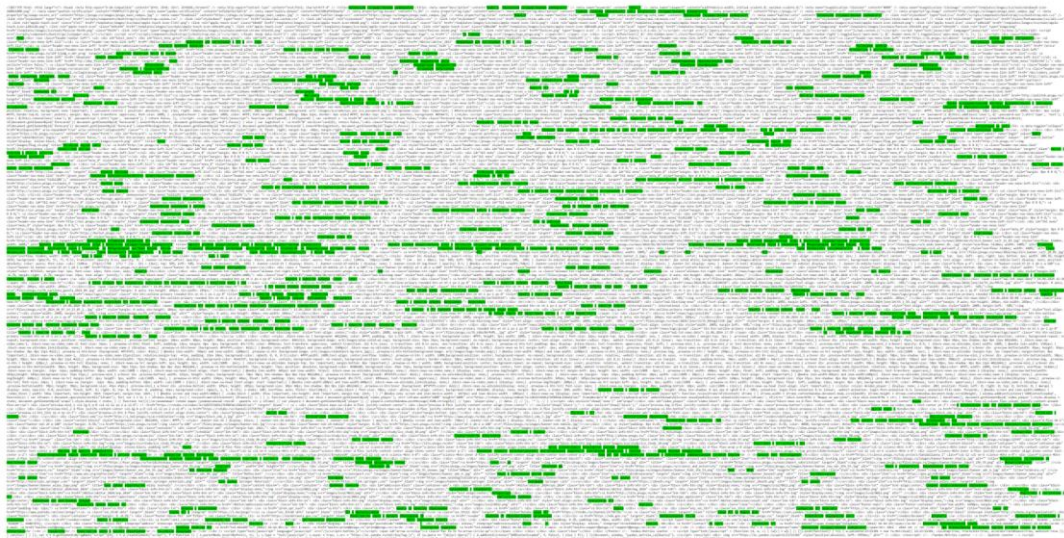
HTML (HyperText Markup Language)

– это язык гипертекстовой разметки, который применяется для структурирования и корректного отображения контента, размещенного на веб-странице, включая текст, таблицы, изображения, видео и другие элементы.

Содержимое HTML-документа играет **главную роль именно в передаче информации**, что важно для поисковых систем

При этом для HTML-документа нормальной ситуацией является **низкая доля значимого содержимого** в общем объеме документа, гораздо больший объем могут занимать скрипты и стили, если они размещаются в документе, SVG-иллюстрации, структуры, выполняющие роль в вёрстке и так далее

Слайд №4



Текстовое содержимое сайта ПГУ выделено зелёным

Слайд №5

Что должна делать
программа?

Извлекать ключевые слова
для веб-страницы

Слайд №6

Что должна делать программа?

Извлекать ключевые слова для веб-страницы

Какими данными программа обладает на входе?

В простейшем случае – исходным кодом сайта в формате HTML. Но для пользователя наиболее удобным способом будет простой ввод URL сайта. Нужно предусмотреть оба способа.

Слайд №7

Что должна делать программа?

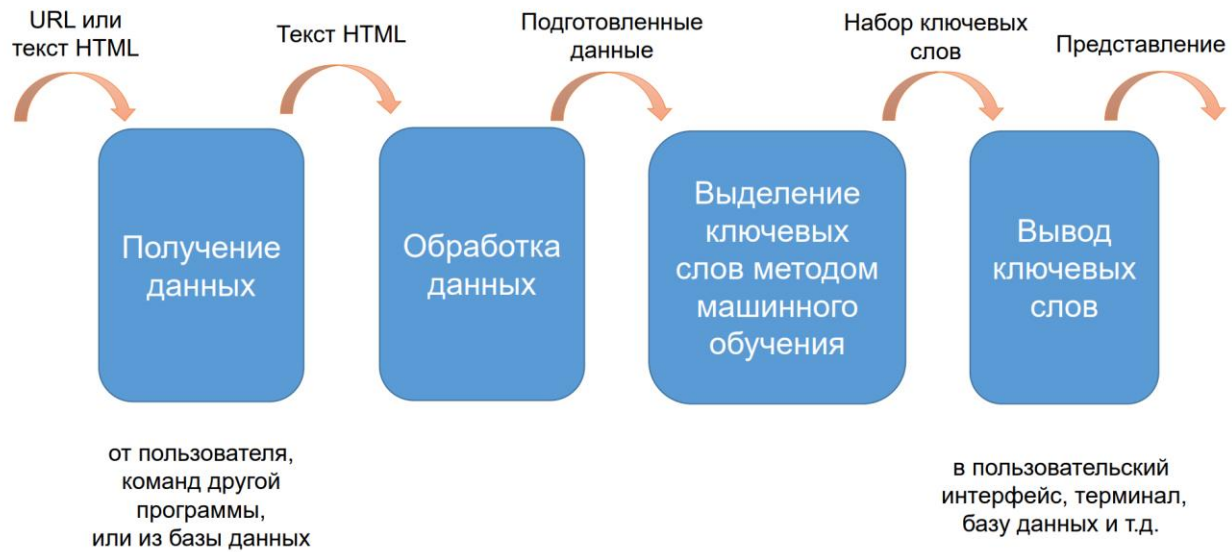
Извлекать ключевые слова для веб-страницы

Какими данными программа обладает на входе?

В простейшем случае – исходным кодом сайта в формате HTML. Но для пользователя наиболее удобным способом будет простой ввод URL сайта. Нужно предусмотреть оба способа.

Какая последовательность действий позволит достичь результата?

Слайд №8



Слайд №9

Модуль обработки данных

```
<!DOCTYPE html>
<html>

<head>
<title>Название сайта</title>
<meta name="description" content="Описание сайта" />
<link href="style.css" rel="stylesheet" />
</head>

<body>
<h1>Заголовок</h1>
<p>
Обычный текст, в котором есть <b>ключевые</b> <b>слова</b>
</p>

<p>Совсем обычный текст</p>

<script>
alert("А скрипт нам совсем не нужен")
</script>
</body>

</html>
```

Элемент документа	Оценка значимости
<title>	50
meta keywords	40
meta description	40
h1-h6	30
	30
	20
<i>, <u>, , <cite>	10
Основной текст	1

Слайд №10

```

<!DOCTYPE html>
<html>

<head>
  <title>Название сайта</title>
  <meta name="description" content="Описание сайта" />
  <link href="style.css" rel="stylesheet" />
</head>

<body>
  <h1>Заголовок</h1>
  <p>
    Обычный текст, в котором есть <b>ключевые</b> <b>слова</b>
  </p>

  <p>Совсем обычный текст</p>

  <script>
    alert("А скрипт нам совсем не нужен")
  </script>
</body>

</html>

```

Элемент документа	Оценка значимости
<title>	50
meta keywords	40
meta description	40
h1-h6	30
	30
	20
<i>, <u>, , <cite>	10
Основной текст	1

Слайд №11

```

<!DOCTYPE html>
<html>

<head>
  <title>Название сайта</title>
  <meta name="description" content="Описание сайта" />
  <link href="style.css" rel="stylesheet" />
</head>

<body>
  <h1>Заголовок</h1>
  <p>
    Обычный текст, в котором есть <b>ключевые</b> <b>слова</b>
  </p>

  <p>Совсем обычный текст</p>

  <script>
    alert("А скрипт нам совсем не нужен")
  </script>
</body>

</html>

```


 text_blocks = [
 ["Название сайта", 50]
]

Слайд №12

```

<!DOCTYPE html>
<html>

<head>
  <title>Название сайта</title>
  <meta name="description" content="Описание сайта" />
  <link href="style.css" rel="stylesheet" />
</head>

<body>
  <h1>Заголовок</h1>
  <p>
    Обычный текст, в котором есть <b>ключевые</b> <b>слова</b>
  </p>

  <p>Совсем обычный текст</p>

  <script>
    alert("А скрипт нам совсем не нужен")
  </script>
</body>

</html>

```

text_blocks = [
 ("Название сайта", 50),
 ("Описание сайта", 40),
]



Слайд №13

```

<!DOCTYPE html>
<html>

<head>
  <title>Название сайта</title>
  <meta name="description" content="Описание сайта" />
  <link href="style.css" rel="stylesheet" />
</head>

<body>
  <h1>Заголовок</h1>
  <p>
    Обычный текст, в котором есть <b>ключевые</b> <b>слова</b>
  </p>

  <p>Совсем обычный текст</p>

  <script>
    alert("А скрипт нам совсем не нужен")
  </script>
</body>

</html>

```

text_blocks = [
 ("Название сайта", 50),
 ("Описание сайта", 40),
 ("Заголовок", 30),
 ("Обычный текст, в котором есть
ключевые слова", 1),
 ("ключевые", 30),
 ("слова", 30),
 ("Совсем обычный текст", 1)
]

Слайд №14

Частотный метод

Для извлечения ключевых слов из документов можно использовать различные методы машинного анализа, успешно применяемые в других задачах семантического анализа текста. Рассмотрим пример:

- TF-IDF (Term Frequency-Inverse Document Frequency): Этот метод разметки текста оценивает важность термина в документе относительно корпуса текстов [19]. Высокие значения TF-IDF свидетельствуют о том, что термин часто встречается в данном документе, но редко встречается в других документах. Применение TF-IDF к текстовому содержанию HTML-документа позволяет выделить ключевые термины, но требует корпуса текстов либо многостраничной обработки сайта.
- Модели векторного представления слов (Word Embeddings): Word2Vec, GloVe, FastText и другие модели векторного представления слов позволяют представить слова в виде векторов в многомерном пространстве [19]. Применение этих методов к текстовой информации из HTML-документа позволяет в выделения семантически связанных ключевых слов.
- Обучение с учителем применимо для построения модели выделения ключевых слов, но потребует сбора большого объема данных.
- Алгоритмы обработки естественного языка (Natural Language Processing, NLP): NLP-методы, такие как выделение именованных сущностей (Named Entity Recognition) или анализ синтаксиса, могут помочь выявить важные термины и фразы в тексте HTML [20].

Учет семантических тем, таких как , , и других, позволяет более точно определить ключевые слова, предоставляя контекст и структуру страницы [21].

В анализе документов поисковые системы оценивают, находятся ли поисковые термины в важных областях документа — заголовке, метаданных, тегах заголовков и основном тексте. Они также пытаются автоматически оценить качество документа на основе анализа документа, а также множества других факторов [22].

Опираясь только на анализ документов недостаточно для современных поисковых систем, поэтому они также учитывают семантическую связанность. Семантическая связанность относится к словам или фразам, которые обычно ассоциируются друг с другом. Поисковые системы активно создают собственный тезаурус и словарь, чтобы помочь им определить, как определенные термины и темы связаны. Просто сканируя свои огромные базы данных контента в Интернете, они могут использовать теорию нечетких множеств и определенные уравнения [22].

Эффективный метод часто включает в себя комбинацию различных подходов и признаков, таких как частота встречаемости, их семантическая близость, структура HTML и другие.

1.6 Частотный метод извлечения ключевых слов

Частотный метод извлечения ключевых слов основывается на допущении, что чем чаще слово встречается в тексте, тем выше вероятность, что оно является ключевым. Для этого метод использует частотный анализ, чтобы выявить наиболее значимые слова. Сначала текст разбивают на отдельные слова или токены. Затем все слова приводят к нижнему регистру и удаляют пунктуацию, чтобы избежать дублирования. Так, "обучение" и "Обучение" будут

$$R = \frac{N_{\text{слова}}}{N}$$

R — значимость слова
 $N_{\text{слова}}$ — частота вхождения слова
 N — общее число слов

Слайд №15

Метод TF-IDF

TF (term frequency — частота термина) — отношение числа вхождений некоторого термина к общему числу слов документа

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

n_t есть число вхождений лексемы t в документ

IDF (inverse document frequency) — обратная частота документа) — инверсия частоты, с которой некоторое понятие встречается в документах коллекции

$$IDF(t, D) = \log \frac{|D|}{|\{d_i \in D \mid t \in d_i\}|},$$

$|D|$ — число документов в коллекции;

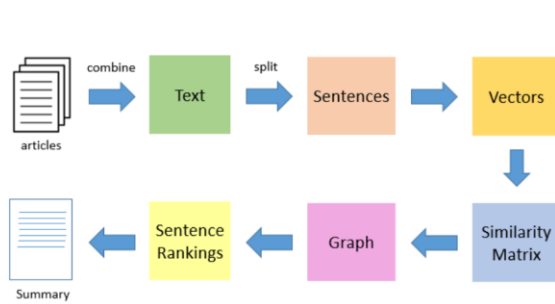
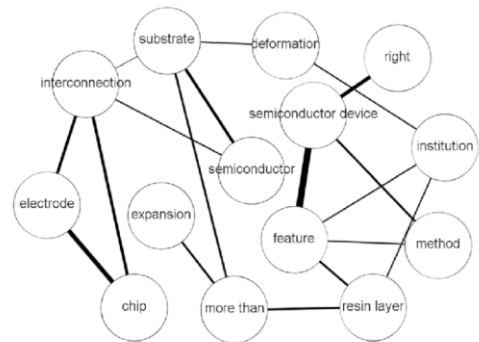
$|\{d_i \in D \mid t \in d_i\}|$ — число документов из коллекции D , в которых встречается t (когда $n_t \neq 0$).

Мера TF-IDF является произведением двух сомножителей

$$TFIDF(t, d, D) = TF(t, d) \times IDF(t, D)$$

Слайд №16

Метод TextRank

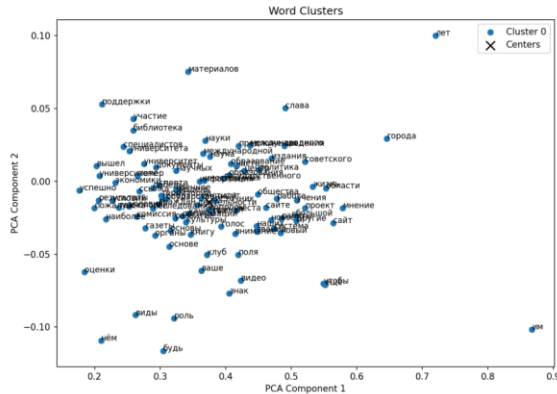


Алгоритм строит граф, где вершины представляют собой токены, а ребра — связи между ними.

Затем разбирает понятия по важности в зависимости от их связей

Слайд №17

Метод кластеризации векторов



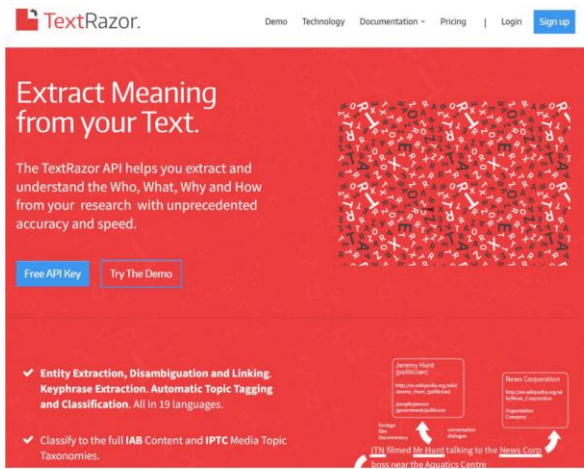
Включает два этапа:

Векторизация – представление лексем в виде вектора. Векторы можно складывать, вычитать, сравнивать. В данной работе используем мультязычную модель **fastText**

Кластеризация – группировка векторов таким образом, чтобы объекты из одного кластера были более похожи друг на друга, чем на объекты из других кластеров. Используем метод DBSCAN.

Слайд №18

Метод с использованием TextRazor



```
import textrazor

textrazor.api_key = "API_KEY_GOES_HERE"

client = textrazor.TextRazor(extractors=["entities", "topics"])
client.set_cleanup_mode("cleanHTML")
response = client.analyze_url("https://pnzgu.ru")

for entity in response.entities():
    print(entity.id, entity.relevance_score,
          entity.confidence_score, entity.freebase_types)
```

Слайд №19

Экспериментальная оценка алгоритмов

Чтобы получить числовое значение, которое отражало бы близость слов по смыслу, мы можем воспользуемся зарекомендовавшей себя метрикой в виде косинусного сходства векторов

Дополнительным удобством этого метода является тот факт, что значение косинуса для угла между совпадающими векторами равно 1.0, а для ортогональных (что можно считать полной несвязностью исходных слов) – 0.0

Имея два вектора, которые отражают смысл терминов, можно воспользоваться линейной алгеброй: угол между этими векторами в n-мерном пространстве является мерой их расхождения, а косинус этого угла – числовым выражением близости векторов.

Метод извлечения	Число успешных результатов	Среднее косинусное сходство
Частотный анализ	736	0.5758
TextRank	680	0.6322
TFIDF	723	0.5995
TextRazor	568	0.3247
FastText	723	0.6523

Результаты сравнения методов

Слайд №20

Пользовательский интерфейс

Извлечение ключевых слов

Добро пожаловать! Наш сервис по извлечению ключевых слов позволяет вам быстро и легко выявлять основные темы и ключевые понятия в тексте вашего веб-сайта или HTML-файла. Просто введите URL или загрузите HTML-файл, выберите метод извлечения, и мы предоставим вам список ключевых слов с их оценками важности.

Информация о ключевых словах может значительно улучшить SEO-оптимизацию вашего контента, повысить его видимость в поисковых системах и помочь аудитории быстрее найти нужную информацию.

URL:

HTML файл:

файл не выбран

Метод извлечения:

Ключевые слова для https://pnzgu.ru

Использованный метод: FastText

Метод, основанный на использовании предобученной модели FastText для извлечения ключевых слов.

университет:	1.0
государственный:	0.9834051686016908
новости:	0.8498297851687031
качества:	0.6853976517911459
другие:	0.6835812565249713
комиссия:	0.6821326317238854
сайте:	0.6688535621551657
наука:	0.6481093436410672
образование:	0.6382007338473465
специалистов:	0.6290069398175487

Слайд №21

Результаты

По итогам работы реализовано функциональное программное решение, выполняющее задачу выделения ключевых слов несколькими методами на выбор.

Выводы, полученные в ходе исследования могут быть применены в области

- поисковой оптимизации (SEO),
- контент-менеджмента,
- краткого изложения текста,
- автоматического аннотирования документов,
- создания систем рекомендаций

и других задач, связанных с обработкой и анализом текстовых данных.

Слайд №22