Оглавление

ЭТАП 1

Теоретические сведения:

Информационный поиск – это процесс поиска в большой коллекции (хранящейся, как правило, в памяти компьютеров) некоего неструктурированного материала (обычно – документа), удовлетворяющего информационные потребности.

Поисковый образ документа (ПОД) – описание документа, в виде перечня ключевых слов, которые могут дополняться их весами, связями и указателями роли. По этому описанию внутри системы составляются структуры данных, служащие для поиска документов и выдачи их из хранилищ. Такое же описание строится для пользовательского запроса.

Поисковый образ запроса (ПОЗ) – описание пользовательского запроса, в виде удобном для поисковой системы. Структура поисковых образов для разных поисковых систем может быть различной, однако поисковый образ запроса и поисковый образ документа должны иметь одинаковую структуру в пределах одной поисковой системы.

Основная задача любой поисковой системы – дать пользователю ответ на его запрос или, другими словами, предоставить список документов релевантных запросу пользователя.

Основными задачами, решаемыми информационно-поисковыми системами, являются:

- приём информации и её предварительная обработка;
- анализ документов и данных, и, возможно, хранение;
- анализ и организация информационных запросов;
- поиск релевантной информации;
- выдача запрашиваемой информации.

Таким образом, поисковые системы обычно состоят из трех компонентов:

- агент (паук, кроулер, робот), который собирает информацию о документах, среди которых будет осуществляться поиск;
- база данных, которая содержит всю информацию, собираемую пауками;
- поисковый механизм, который пользователи используют как интерфейс для взаимодействия с базой данных.

Описание структуры разработанной системы:

Вся разработанная система состоит из трех компонентов. Два из них (микросервер и субклиент) обязательные и являются основой системы. Третий компонент (клиент) является необязательным и представляет из себя оболочку над субклиентом которую могут создать другие пользователи, используя API субклиента. Общая структура системы представлена на рисунке 1.

Рисунок 1. Структура системы.

Субклиент является концентратором информации, находящейся на разных микросерверах. Количество микросерверов не ограничено, чем их больше, тем больше информации можно распределить в системе. Когда на субклиент приходит запрос поиска документов он рассылает данный запрос на микросервера, которые к нему подключены. Настройка подключения микросерверов производится вручную в файле variables.py. После рассылки запросов, субклиент собирает ответы и сортирует в порядке убывания релевантности документов. Так же в субклиенте присутствует стандартный клиент (далее сайт) для взаимодействия, что облегчает работу с системой. Файловая структура субклиента представлена на рисунке 2.

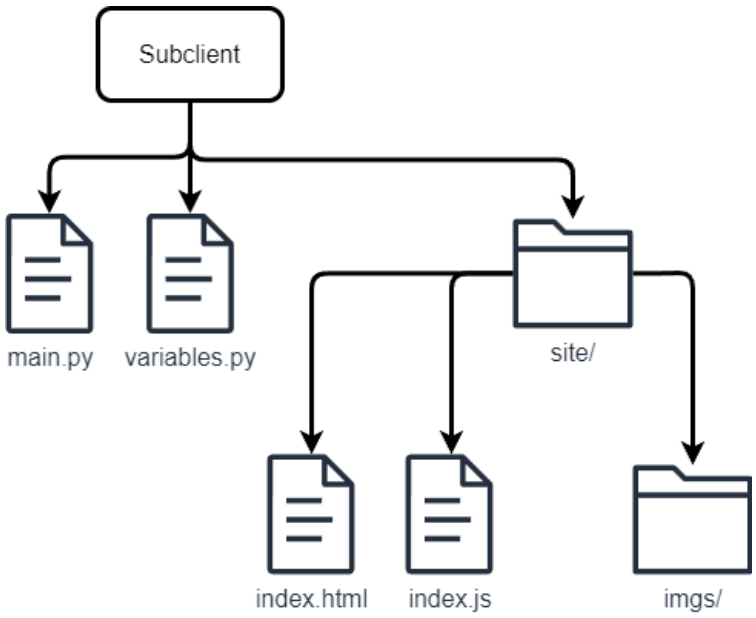

Рисунок 2. Файловая структура субклиента.

Основная логика субклиента находится в файле main.py. Данный файл используется для запуска субклиента. Команда запуска представлена в коде 1.

```
python main.py
```

Код 1. Команда для запуска субклиента.

Для поиска документов в системе и получения доступа к сайту используется настраиваемый HTTP сервер. Все доступные переменные находятся в файле variables.py. Субклиент поддерживает две конечные точки:

1. http://localhost:13000/site/ – позволяет получить доступ к сайту;
2. http://localhost:13000/search?request_content="request" – производит поиск по строке, заданной в параметре request_content (в примере такой строкой является "request").

Так же субклиент составляет метрики по ответам микросерверов к созданным пользователями запросам. Однако метрики составляются исключительно на сайте субклиента и не существует API endpoint-а для взаимодействия с ними. Вся логика составления метрик находится в файле index.js.

Общая файловая структура микросервера представлена на рисунке 3.

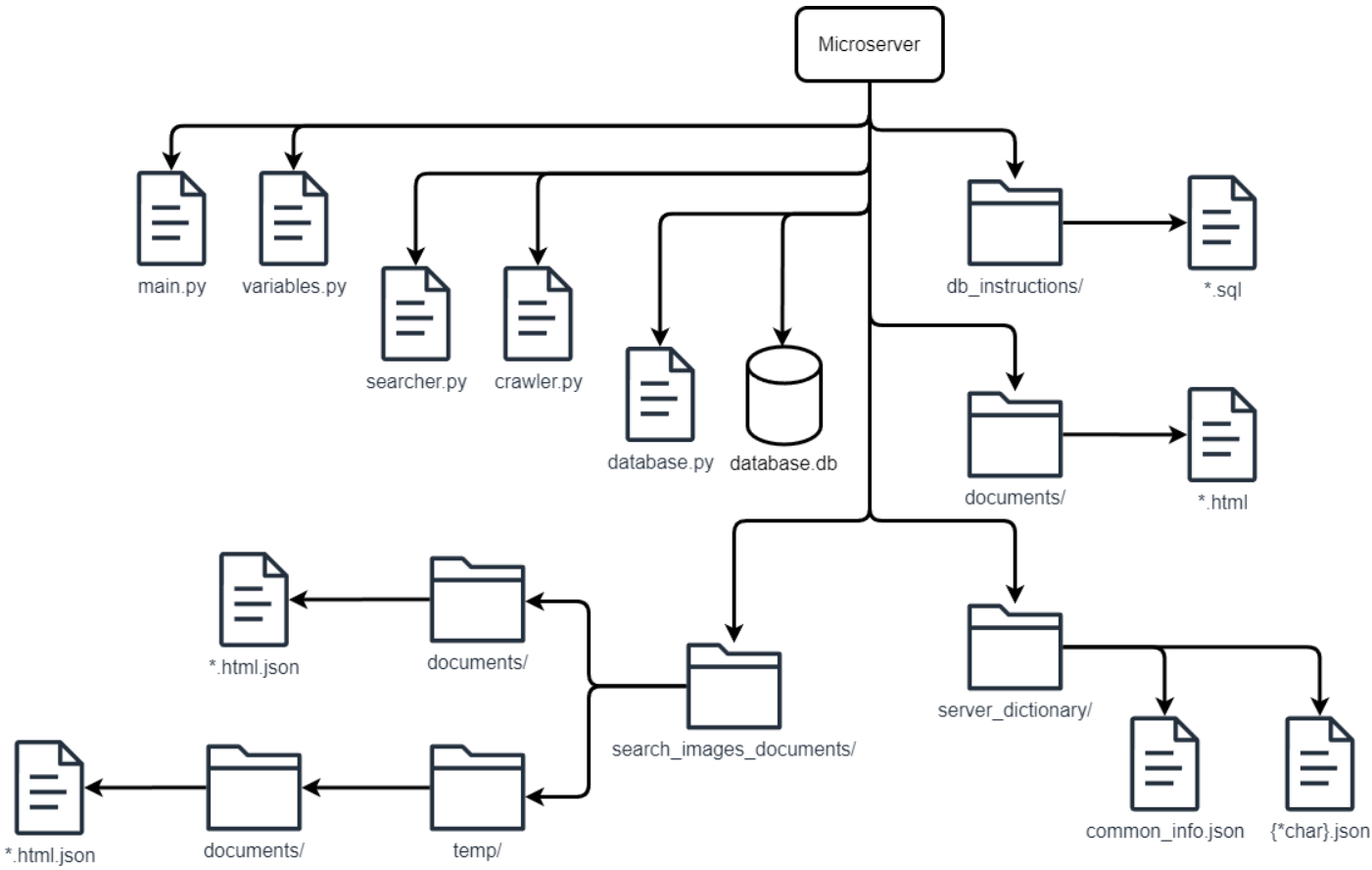

Рисунок 3. Файловая структура микросервера.

Субклиент отправляет запросы на все микросервера. А те в свою очередь их принимают и создают ответы на них. Для реализации данной идеи на микросервере так же присутствует HTTP сервер со следующими endpoint-ами:

1. http://localhost:3000/documents/ – позволяет получать документы с микросервера;
2. http://localhost:3000/search?request_content="request" – производит поиск по строке, заданной в параметре request_content (в примере такой строкой является "request").

Так как система предназначена для локальных сетей в ней отсутствует прокси система для документов. Подразумевается наличие прямого доступа пользователей системы к микросерверам. При работе с сайтом системы пользователь может переходить по ссылкам на документы, которые используют в качестве адреса хоста адрес микросервера на котором хранится данный файл, а не адрес субклиента.

Как и субклиент микросервер является настраиваемый, все возможные переменные для настройки находятся в файле variables.py, а для запуска используется файл main.py. Команда запуска микросервера представлена в коде 2.

```
python main.py
```

Код 2. Команда для запуска субклиента.

Данная команда запускает HTTP сервер для раздачи файлов, поисковик и кроулер в асинхронном режиме. После заданного в настройках таймаута кроулер производит индексирование всех файлов в папке documents и создает поисковые образы документов в search_images_documents/temp. В зависимости от наличия и времени обновления записей в базе данных documents.db, кроулер добавляет, обновляет или удаляет записи. Так же в его функции входит обновление серверного словаря хранящегося в server_dictionary. Словарь разбит на json файлы, каждый файл называется определенным символом из доступного словаря в переменных и хранит только слова, начинающиеся с данного символа. Так же кроулер высчитывает весовые коэффициенты для каждого слова в поисковых образах документов, и инверсную частоту слов в базе для серверного словаря. Операции, связанные с поисковыми образами документов, производятся транзакционно в отдельных функция кроулера. Изначально высчитывается вся информация, а после записывается.

Для работы кроулера и поисковика с базой данных был реализован интерфейс в файле database.py. Данный интерфейс поддерживает все команды записанные в sql файлы и хранящиеся в папке db_instructions. Архитектура базы данных представлена на рисунке 4. В нее входит не только таблица хранящихся документов, но и структура словаря сервера.

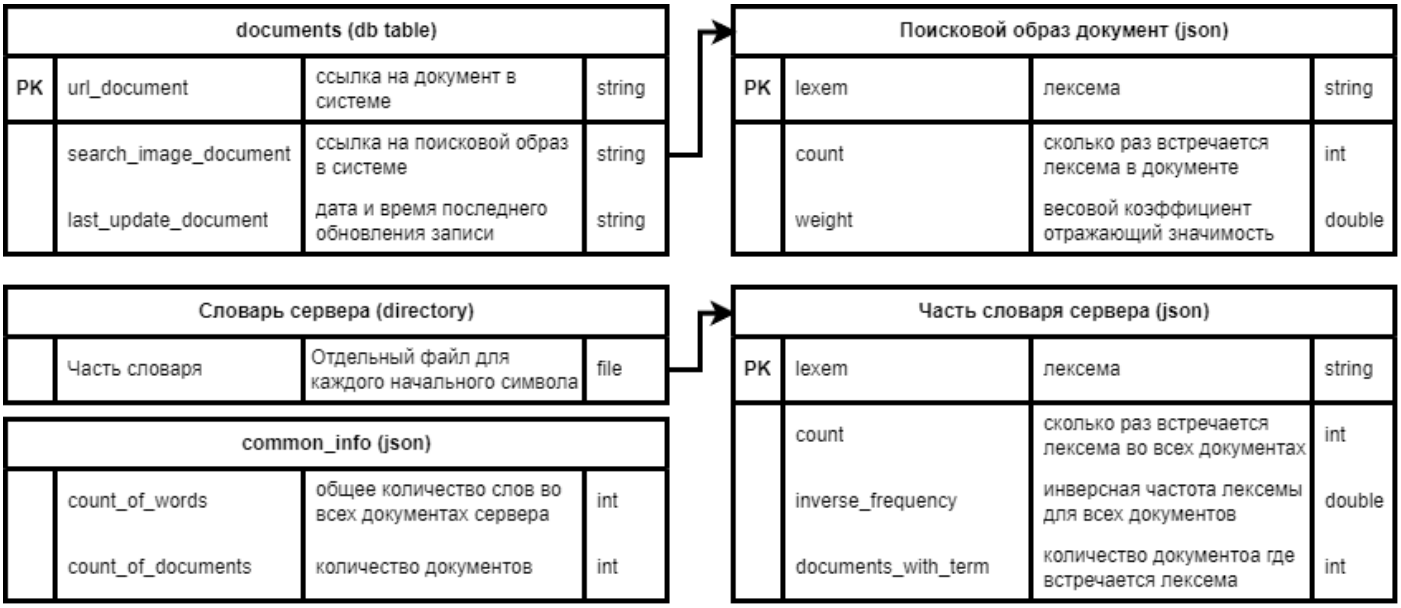

Рисунок 4. Архитектура базы данных микросервера.

Основные алгоритмы реализации компонентов системы

Поиск

Расчет весов кроулером осуществляется согласно статистической мере TF-IDF. Вес некоторого слова пропорционален частоте употребления этого слова в документе и обратно пропорционален частоте употребления слова во всех документах коллекции.

При получении запроса, поисковик разбивает запрос на токены точно так же, как это делает кроулер с документами, а после для каждого документа рассчитывает схожесть с запросом. Код, высчитывающий схожесть документа и запроса, представлен ниже в коде 3.

```
def __calculateSimilarityOfVectors(self, vector_request, vector_document):
    lexems = vector_request.keys()
    def scalarFunc(vector_request, vector_document, lexems):
        answer_scalar = 0
        for lexem in lexems:
            answer_scalar += vector_request[lexem] * vector_document[lexem]
        return answer_scalar
    def moduleFunc(vector, lexems):
        answer_module = 0
        for lexem in lexems:
            answer_module += vector[lexem] ** 2
        return sqrt(answer_module)
    module = (moduleFunc(vector_request,lexems)*moduleFunc(vector_document,lexems))
    if module == 0:
        return 0
    similarity = scalarFunc(vector_request, vector_document, lexems) / module
    return similarity
```

Код 3. Расчет косинуса для двух векторов.

Расчет схожести производится на основе векторной модели поиска. Для поискового запроса строится вектор с мощностью количества выделенных лексем, заполненный единицами. Далее для каждого поискового образа документа так же строится вектор той же мерности, что и вектор запроса. В соответствие каждой лексеме ставится ее вес из документа, для которого строится вектор. После того как вектор документа и вектор запроса построены происходит само вычисление схожести, вычисляется косинус между полученными векторами. Значение косинуса варьируется в пределах от 0 до 1, чем это значение выше, тем меньше расстояние между векторами, а значит запрос и документ более схожи.

Индексирование

Для индексирования файлов используется кроулер. Он находит все файлы в заданной директории. Так же получает данные из базы данных о записанных документах. Далее он составляет три списка документов: add – документы которых нет в базе но есть в директории, их надо добавить, readd – документы которые есть и в базе и в директории, их надо обновить в случае если прошло заданное количество времени, delete – документы запись о которых есть в базе, но которых нет в директории, данные о таких документах надо удалить из системы.

После этого для всех выделенных документов кроулер создает поисковые образы и записывает их во временные файлы. Для того что бы создать поисковой образ требуется разбить документ на токены, делается это по алгоритму, представленному в коде 4.

```
html_document_with_tags=""
current_path: str = os.path.join(self.__working_directory, document_url)
if os.path.isfile(current_path):
    with codecs.open(current_path, "r", encoding="utf-8") as file:
        html_document_with_tags = file.read()
pattern = re.compile('<.*?>')
text_from_document = self.__keepCharactersInStringWithRegex(
    input_string=re.sub(pattern, '', html_document_with_tags).replace('\n', ' '),
    reference_string=ALLOWED_DICTIONARY)

list_of_lexems = re.sub(r'(?:(?!\u0301)[\W\d_])+', ' ', ("".join(character for
character in text_from_document if character in ALLOWED_DICTIONARY)))
list_of_lexems = [lexem for lexem in list_of_lexems.split(" ") if len(lexem) > 2]

dict_of_lexems = {lexem: {
    "count": list_of_lexems.count(lexem),
    "weight": 0} for lexem in list_of_lexems
}
search_image_document = {
    "count_of_words": len(list_of_lexems),
    "dict_of_lexems": dict_of_lexems
}
```

Код 4. Токенизация документ, разбиение на лексемы.

После того как поисковые образы созданы для всех документов, кроулер начинает менять словарь сервера, проверяя все созданные списки документов. Если документ в списке на добавление, то в общей информации о словаре (common_information) добавляется количество документов и количество слов, в противном случае эти же значения уменьшаются. Так же если документ в списке на добавление, то во все части словаря кроулер добавляет новые лексемы из документа. Или, если лексема уже существует в словаре, кроулер добавляет в запись о лексеме количество самих слов и количество документов, в которых эти слова используются. Если документ в списке на удаление, описанные значения уменьшаются, и если какое-то из них становится равным нулю (они должны одновременно стать нулями), то лексема удаляется из словаря. Для документов, которые должны быть обновлены, система сначала все удаляет по их старым образам, а потом заново все добавляет уже исходя из новых.

После того как словарь сервера обновлен кроулер работает с базой данных и перемещает созданные поисковые образы документов из временной папки в папку со всеми образами микросервера.

Расчет метрик

Основной задачей сайта системы является отобразить найденные документы, однако у него есть расширенный функционал – расчет метрик для каждого созданного запроса. Для того что б рассчитать метрики мало знать какие из полученных документов являются релевантными, так же требуется знать сколько всего релевантных документов находится в базе данных. Так что данный функционал предназначен только для разработчика системы, и только для тестовых выборок документов, где проведена аналитика и известны нужные для расчетов значения. В коде 5 расчет метрик.

```
let count = main_store.documents.length
let relevant_count = relevant_documents.size
let unrelevant_count = count - relevant_count
let recall = relevant_count / Math.max(relevant_count_in_db, 1)
let precision_request = relevant_count / Math.max(count, 1)
let error = unrelevant_count /  Math.max(count, 1)
let f_measure = 2 /((1 / Math.max(precision_request, 1)) + (1 / Math.max(error,1)))
let precision = 0, n_precision = 0, r_precision = 0
let precisions_sum = 0
for (const [num_in_queue, document_in_list] of relevant_documents) {
    let num_in_queue_id = parseInt(num_in_queue)
    if (num_in_queue_id + 1 <= 10) { n_precision += 1; }
    if (num_in_queue_id + 1 <= relevant_count) { r_precision += 1; }
    precision += 1
    precisions_sum += (precision / Math.max((num_in_queue_id + 1), 1))
}
n_precision /= Math.max(n, 1)
r_precision /= Math.max(relevant_count, 1)
let average_precision = (1 / Math.max(relevant_count_in_db, 1) * precisions_sum)
```

Код 5. Расчет основных метрик для запроса.

Интеллектуальная составляющая интерфейса

По условию задания стандартный описанный субклиент (интерфейс) должен иметь интеллектуальную составляющая. Для реализации данного требования сайт поддерживает голосовой ввод, для определения слов и предложений используется встроенная функция JavaScript – SpeechRecognition. Данная функция является разрабатываемой и поддерживается малым количеством браузеров. Корректная работа была выявлена в браузере Edge [Версия 119.0.2151.72 (Официальная сборка) (64-разрядная версия)].

При предоставлении разрешении сайта доступа к микрофону, он постоянно слушает что говорит пользователь и выполняет команды, соответствующие таким функциям как “Поиск”, “Метрики”, “Пересчет метрик”, “Очистка”, “Заполнение количество релевантных документов”.

Благодаря использованию стандартных средств и гибкой реализации логики, добавление команд является простой задачей. Так же поддерживаются несколько режимов команд, когда текст сказанный пользователем полностью совпадает с заданной фразой или когда начинается с нее.

Тестирования системы и анализ метрик

Для тестирования системы было отобрано 25 текстов в предметной области «Компьютерные игры». Документы были помещены в папку documents и был запущен микросервер в локальной сети. Общее количество слов, полученное после индексирования выбранных документов, составило 30 тысяч. На обработку данных текстов ушло примерно 5 секунд, что говорит о довольно быстрой работе системы. После этого был запущен субклиент, а в списке микросерверов был указан созданный ранее микросервер. При помощи сайта были сформированы и протестированы сотни запросов, все запросы выполнялись с различных устройств, проверялось выполнение различных функций системы. Для тестирования работы выбрались браузеры – Google Chrome [Версия 119.0.6045.163, ОС Android 13], для мобильных устройств, Firefox [Версия 120.0 (64-разрядная версия)] и Edge – для компьютеров. Исходя из полученных результатов, в качестве общей рекомендации для составления запросов к системе, в которой хранится выборка документов о конкретной предметной области, можно сказать следующее:

1. Надо стараться избегать использовать вопросы с общими словами предметной области. К примеру, в случае предметной области “Компьютерные игры” таковыми словами являются: игра, персонаж, компьютер, разработчик.
2. Использовать как можно более узкие термины для получения более конкретных результатов.
3. Делать запросы как можно длиннее, чтобы получать больше вариантов документов.

Для той же выборки при создании запросов были получены следующие метрики:

- полнота (recall)
- точность (precision)
- ошибка (error)
- F-мера (F-measure)
- точность на уровне 10 документов (precision(10))
- R-точность (R-precision)
- средняя точность (average precision)
- 11-точечный график полноты/точности, измеренный по методике TREC (11-point matrix (TREC))

На рисунках 5 и 6 приведены примеры полученных метрик для заданных запросов с различных устройств в локальной сети.

Что бы получить метрики требуется выполнить запрос и указать какие из полученных в ответе документов являются релевантными, а также требуется указать общее количество релевантных документов в системе. Исходя из метрик и результатов поиска для разных запросов можно сказать, что система работает тем точнее, чем конкретнее слова для документа.

Точность системы зависит от качества получаемых запросов. Полнота практически всегда очень высока, потому что система возвращает все документы с найденными в запросе словами. Точность от n и r-точность довольно слабые метрики, потому как не учитывают все документы, которые вернула система в качестве ответа. Хорошей метрикой точности является «Средняя точность» так как она практически на все запросы выдавала оценку близкую к настоящей точности системы. 11-точечный график полноты/точности, измеренный по методике TREC хорош тем, что дает более общую картинку качества работы систему, отображает постепенное снижение точности в упорядоченном ответе системы: чем дальше правая граница области индексов документов ответа, тем больше полнота и тем меньше точность.

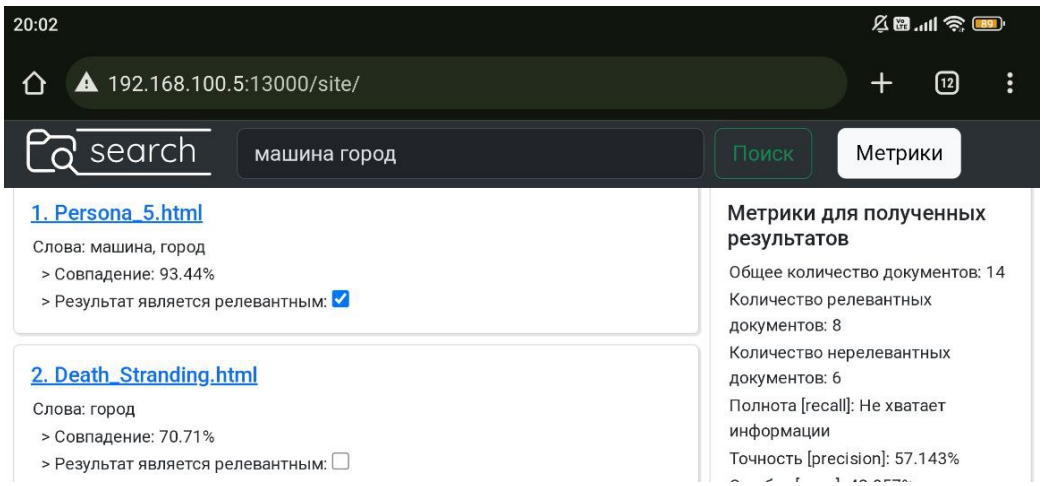

Рисунок 5. Получение результатов работы системы в локальной сети.

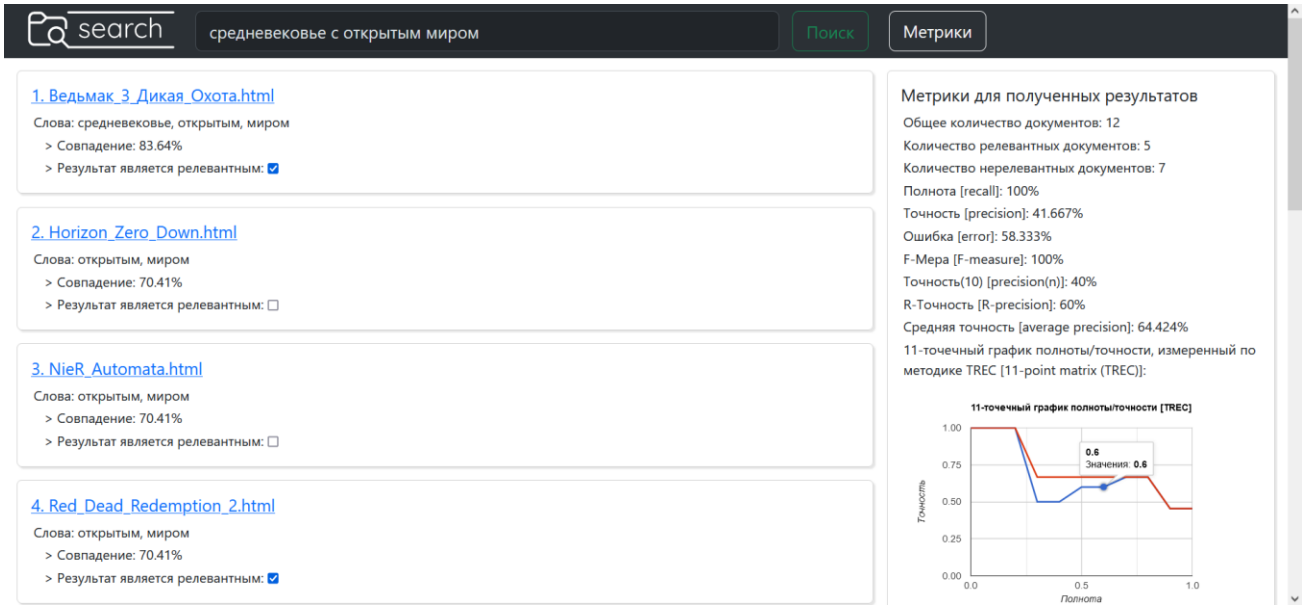

Рисунок 6. Результат выполнения поиска в системе и нахождение метрик.

На основании результатов анализа данных метрик можно предложить следующие улучшения для созданной системы:

- изменить метод токенизации текстов, убрать формы лексем, оставить только начальные формы слов
- учитывать положение слов в контексте при расчете весов, пытаться понять контекст запроса
- добавить отрицательные слова в запрос, слова, которых не должно быть в ответных документах

ЭТАП 2

Теоретические сведения:

N-грамм метод основывается на подсчете частот N-грамм (подстрок длины не более N) и предположении, что примерно 300 самых часто используемых N-грамм сильно зависят от языка. В основе этого метода лежит Закон Зипфа – эмпирическая закономерность распределения частоты слов естественного языка: если все слова языка (или просто достаточно длинного текста) упорядочить по убыванию частоты их использования, то частота n-го слова в таком списке окажется приблизительно обратно пропорциональной его порядковому номеру n (так называемому рангу этого слова). Например, первое по используемости слово встречается примерно в два раза чаще, чем второе, и в три раза чаще, чем третье.

Алфавитный метод заключается в определении языка на основании обнаруженных в анализируемом тексте характерных диакритических знаков – специальных значков, добавляемых к буквам того или иного алфавита с целью обозначить изменение их стандартного чтения или же указать на какую-либо особую роль, которую звук, обозначенный буквой с диакритикой, играет в слове.

Нейросетевой метод заключается в построении математической модели в виде сети нейронной сети, обучении ее на заготовленных тестовых текстах, а затем получение предсказаний от разработанного алгоритма о языке текста, передаваемого на входной слой сети.

Описание структуры разработанной системы:

Структура разработанной системы описана в этапе 1. Запуск и развертывание системы аналогично. На данном этапе был добавлен класс Определитель (Definer). От него в своего очередь были наследованы еще три класса для каждого метода. Для того чтобы текст документа распознавался внутри системы, было добавлен вызов каждого метода в веб-краулер. Теперь система распознает язык каждого документа в системе и хранит в поисковом образе документа результаты распознавания для каждого метода. Так же были добавлены необходимые переменные в микросервере и общий обработчик HTML-текстов.

Для обучения системы требуются корректные изменения в файле *microserver/variables.py* (при добавлении языков использовать один порядок для всех связанных с данным процессом переменных – *ALLOWED_DEICTIONARY*, *LANGUAGES_TO_DEFINE*, *LANGUAGES_ALPHABETS*). Так же возможна настройка методов распознавания при помощи переменных в данном файле.

Кроме изменения переменных для обучения так же требуется тестовая выборка документов, которая должна быть помещена в директорию *documents_for_language_definer/documents_sources/*lang*/*source_doc_name*.html*.

Каждый текст должен быть в директории своего языка, рекомендуетсяс использовать одинаковое количество тектстов одинакового размера для всех языков.

Изменения файловой структуры микросервиса представлены на рисунке 7.

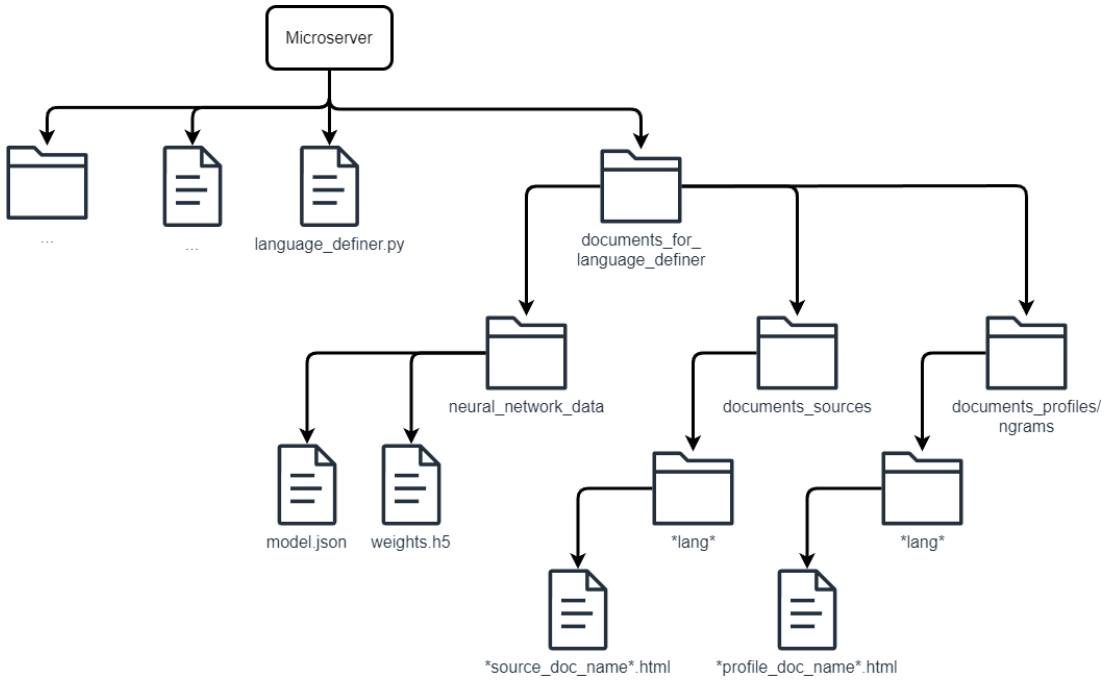

Рисунок 7. Изменения в архитектуре микросервера.

Основные алгоритмы реализации компонентов системы

Метод N-грамм

Для всех языков каждого документа в директориях *documents_for_language_definer/documents_sources/*lang*/* строятся профили документов хранящие n-граммы в порядке убывания частоты их появления для и сохраняются в директорию *documents_for_language_definer/documents_profiles/ngrams/*lang*/*. Максимальная длина n-грамма указывается в настройках микросервера. После того как были построены все профили доукментов система начинает принимать запросы на определение языка. Для каждого текста поступаемого на вход в определитель строится n-грамм профиль, а после находится расстояния между построеным профилем и всеми сохраненными профилями документов.

Расстояние между статистиками подсчитывается следующим образом: все n-граммы сортируются в порядке убывания частоты их появления, затем для каждой n-граммы вычисляется разница её позиций в отсортированном списке n-грамм тестового и тестируемого документов. Расстояние между статистиками определяется как сумма разниц позиций каждой n-граммы. Если n-грамма отсутствует в профиле категории (языка), то ей назначается максимальная величина оценки несовпадения позиций N-грамм.

```
@staticmethod
    def __calculatingTheOutOfPlaceMeasureBetweenTwoProfiles(ngram_profile_1,
ngram_profile_2):
        distance_measure = 0
        max_diff = len(ngram_profile_1)
        for ngram in ngram_profile_1:
            if ngram in ngram_profile_2:
                distance_measure += abs(ngram_profile_1.index(ngram) -
ngram_profile_2.index(ngram))
            else:
                distance_measure += max_diff
        return distance_measure
```

Код 6. Расчет расстояния между двумя профилями.

Алфавитный метод

Алфавитный метод построен на определении самых часто встречаемых символов в документе и сравнении всех алфавитов доступных системе. Язык алфавит, которого является наиболее близким к алфавиту текста и считается языком текста.

```
@staticmethod
def define(text: str) -> str:
    text = TextProcessor.makeClearedTextFromRawHtmlText(text)
    chars = {}
    for char in text:
        if char == " ": continue
        if not chars.get(char): chars[char] = 1
            else: chars[char] += 1

    languages_weights = {language: 0 for language in LANGUAGES_TO_DEFINE}
    for char in chars:
        for language in LANGUAGES_TO_DEFINE:
            if char in LANGUAGES_ALPHABETS[language]:
                languages_weights[language] += chars[char]
    result = max(languages_weights, key=languages_weights.get)
    if languages_weights[result] == 0:
        return ""

    return result
```

Код 7. Алфавитный метод определения языка текста.

Нейросетевой метод

Для реализации данного метода использовались сторонние библиотеки для языка python – библиотека tensorflow и ее обертка keras. Так же была использована библиотека numpy для обработки и подготовки данных для обучения сети. Построенная нейронная сеть является LSTM-сетью. Качество нейронной сети на определяется критерием точности “accuracy”, на тестовой выборке данное значение доходило до 90% при трех

эпохах обучения. На рисунке 8 представлен график точности в зависимости от эпохи обучения нейросети.

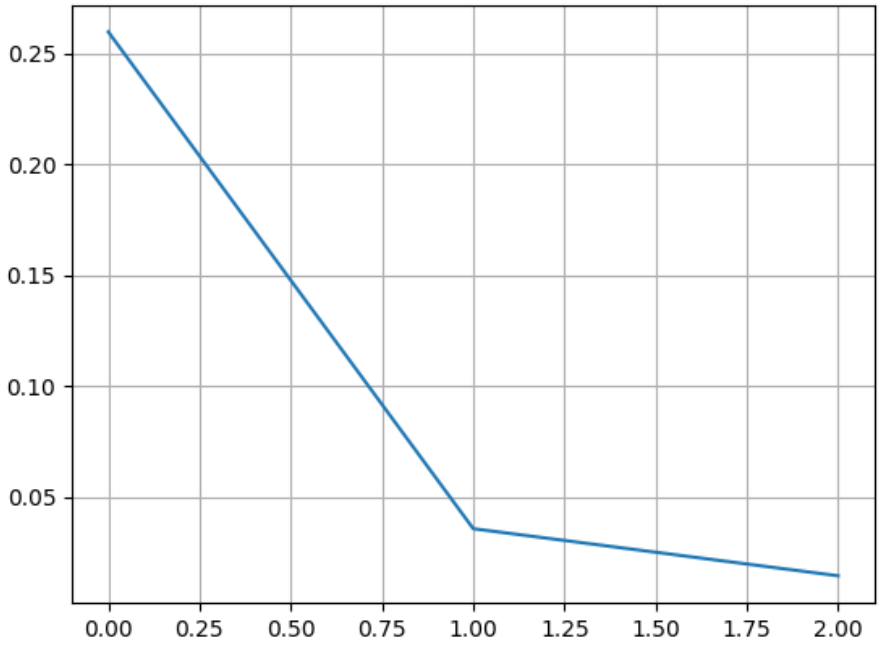

Рисунок 8. График точности сети при обучении.

Перед тем как обучить нейросеть идет обработка и подготовка данных. Каждый текст для необходимых языков токенезируется и разбивается на последовательности необходимой длины. После этого идет обучение. После обучения система принимает запросы на определения языка нейросетевым методом. Каждый входной текст также токенезируется и разбивается на последовательности необходимой длины.

```
@staticmethod
def trainNetwork(data_to_train) -> None:
    model = DefinerNeuralNetworkMethod.NeuralNetwork.createNetwork()

    batch_size = 32
    epochs = 3
    history = model.fit(data_to_train["input"],
                        data_to_train["output"],
                        batch_size=batch_size,
                    epochs=epochs,
                    verbose=True)
```

Код 8. Обучение нейросети на подготовленных данных.

Токенизация для нейросетевого метода

Для разбиения исходного текста на токены используется функция word_tokenize из библиотеки nltk. Далее для каждого токена полученного данным метолом вычисляется индекс по формуле:

$$Index(Token) = \sum_{i=1}^{len(Token)} (ADi(Token[i]) * i)$$

где $Token$ – строка токен, i – индекс символа в токене, $len[Token]$ – длинна токена, $Token[i]$ – символ токена под индексом i, $ADi(Char)$ – индекс символа $Char$ в разрешенном словаре $AllowedDictionary$

Пример для токена “longword” в английском языке:

$AllowedDictionary$ = “abcdefghijklmnopqrstuvwxyz” – символы, разрешенные для использования в токенах

$MaxTokenLength$ = 30 – максимальная возможная длинна токена

Token = “longword” – пример токена

$ADi(l) = 12,\ ADi(o) = 15, ADi\ (n) = 14,\ ADi\ (g) = 7,\ ADi\ (w) = 23,\ ADi\ (o) = 15,\ ADi\ (r) = 18,\ ADi\ (d) = 4$

$$Index(longword) = 1 * 12 + 2 * 15 + 3 * 14 + 4 * 7 + 5 * 23 + 6 * 15 + 7 * 18 + 8 * 4 = 12 + 30 + 42 + 28 + 115 + 90 + 126 + 32 = 475$$

Тестирования системы и анализ методов распознавания языка текстов

Для тестирования системы было отобрано 10 текстов в предметной области «Информатика». Документы были помещены в папку documents и был запущен микросервер в локальной сети. Для каждого документа система определила язык исходя из всех трех реализованных методов распознавания.

Для обучения системы были использованы документы из той же предметной области по 10 штук для каждого языка.

Исходя из полученных результатов, в качестве общих рекомендаций и замечаний по обучению и использованию различных методов определения языка текстов можно выделить следующие пункты:

4. Каждый из методов не идеален и может давать сбой, поэтому система вычисляет язык на основе всех трех методов
5. Для обучения и распознавания лучше использовать чистый текст без каких-либо повторяющихся элементов (к примеру названия кнопок при скачивании html-страницы из интернет)

Результаты работы системы представлены на рисунке 9.

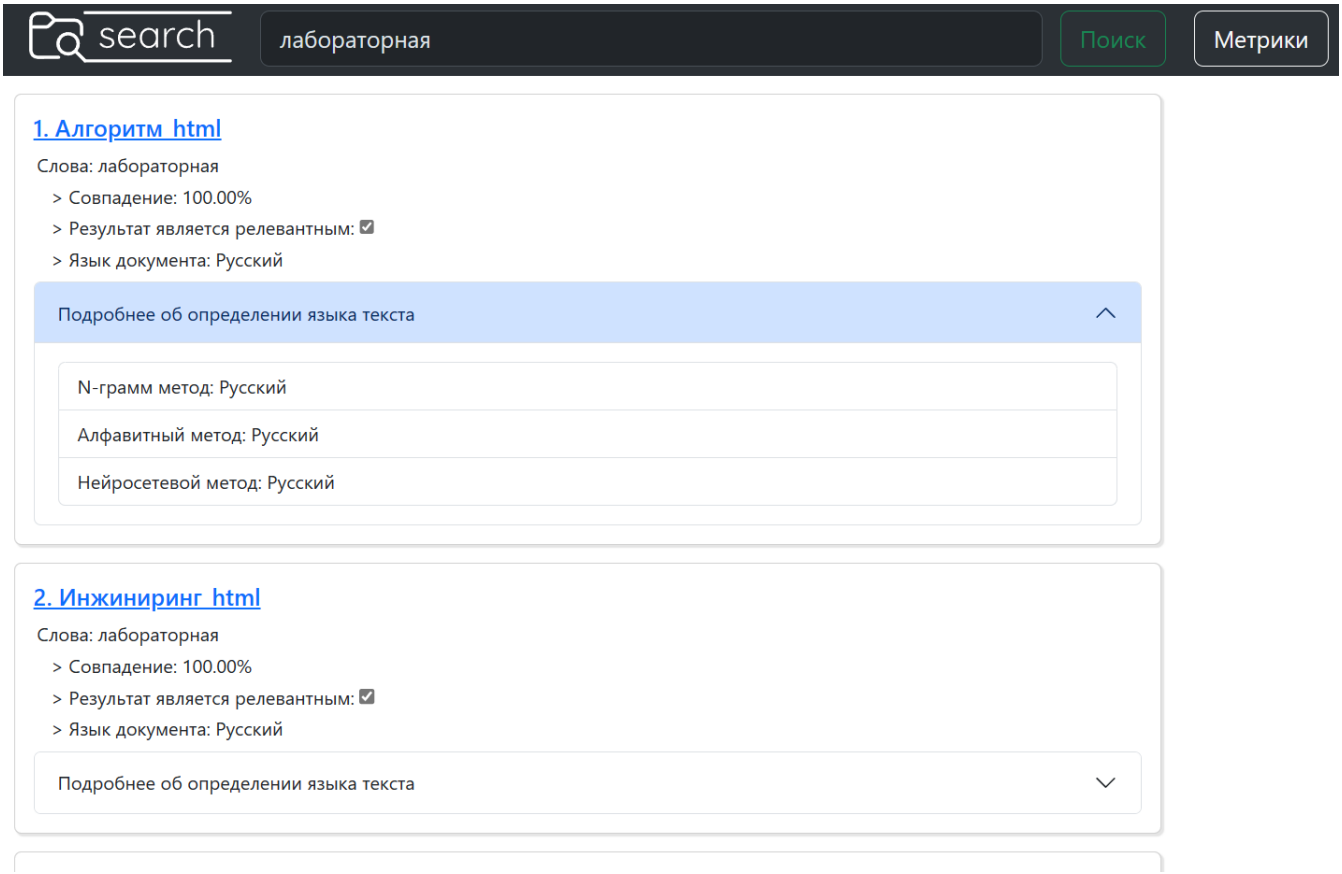

Рисунок 9. Результат выполнения поиска в системе и определения языка документов.

На основании результатов тестирования системы можно предложить следующие улучшения для созданной системы:

- изменить метод токенизации текстов (существует высокая вероятность создания одинаковых индексов для разных токенов)
- добавить лемматизацию в систему
- добавить больше методов определения языка (к примеру метод частотных слов или метод коротких слов)
- добавить фильтры поиска по языку

ЭТАП 3

Теоретические сведения:

Весовые коэффициенты значимости терминов. Функция автоматического реферирования – дать сжатое представление текстовой информации, позволяющее пользователю экономить время при поиске и отборе необходимой информации, т.е. «отсеивать» менее значимую информацию. Можно выделить следующие типы рефератов:

- реферат в виде списка ключевых слов
- классический реферат
- структурированный реферат
- запросно-ориентированный реферат

Реферат в виде ключевых слов – список, возможно иерархический (в виде дерева) наиболее информативных слов и словосочетаний (именных групп) обрабатываемого документа. Такой реферат позволяет пользователю понять основные темы, описанные в документе. Например: лазер, лазерный луч, синий лазер, красный лазер, устройство и т.д.

Классический реферат – это набор наиболее информативных предложений текста, возможно трансформированных (удаление вводных конструкций, замена анафоричных местоимений и т.д. с целью улучшения связности реферата и уменьшения его объема).

При построении классического реферата методом sentence extraction необходимо:

- Вычислить веса слов документа. При этом слова из латинских букв, числа, стоп-слова - не учитываются. Базовый вес слова вычисляется по формуле TF*IDF.
- Вычислить веса предложений согласно формулам, приведенным ниже.
- Осуществить генерацию реферата.

Этап генерации представляет собой выбор из исходного текста определенного количества предложений с наибольшим весом в той последовательности, в которой они идут в тексте. Рекомендуемый размер реферата 10 предложений. Вес $Weight(S_i)$ каждого предложения S_i вычисляется произведением значений функций, приведенных ниже.

Функции, характеризующие положение предложения в документе $Pos_{document}(S_i)$ и положение в абзаце $Pos_{paragrap}(S_i)$.

$$Pos_{document}(S_i) = 1 - \frac{B_{document}(S_i)}{|D|}$$

$$Pos_{paragraph}(S_i) = 1 - \frac{B_{paragraph}(S_i)}{|P|}$$

$$Weight(S_i) = Pos_{document}(S_i) * Pos_{paragraph}(S_i)$$

$|D|$ – число символов в документе D, содержащем предложение S_i;

$B_{document}(S_i)$ – количество символов до S_i в $Document(S_i)$;

$|P|$ – количество символов в абзаце P, содержащем предложение S_i;

$B_{paragraph}(S_i)$ – количество символов до S_i в $Paragraph(S_i)$.

Модифицированная TF*IDF функция

$$Score(S_i) = \sum_{t \in S_i} tf(term, S_i) * w(term, Document)$$

$tf(term, S_i)$ – частота термина t в предложении S_i;

$w(term, Document)$ – вес термина в документе $Document$.

$$w(term, Document) = 0.5 * \left(1 + \frac{tf(term, Document)}{tf_{max}(Document)}\right) * \log\left(\frac{|DocumentCount|}{df(t)}\right)$$

$tf(term, Document)$ – частота термина t в документе $Document$;

$df(t)$ – количество документов, с термином $term$;

$tf_{max}(Document)$ – максимальная частота термина в документе $Document$;

$|DocumentCount|$ – количество документов.

Описание структуры разработанной системы:

Структура разработанной системы описана в этапах 1 и 2. Запуск и развертывание системы аналогично. На данном этапе был добавлен класс Рефератор (Summarizer). От него в своего очередь были наследованы еще три класса для каждого метода. Для того чтобы реферат составлялся внутри системы, были добавлены вызовы каждого метода в веб-краулер. Теперь система создает реферат для каждого документа в системе и хранит в поисковом образе документа результаты реферирования для каждого метода. Так же были добавлены необходимые переменные в микросервере.

Основные алгоритмы реализации компонентов системы

Метод Sentence Extraction (метод из методического пособия)

Данный метод является извлекающим методом реферирования текстов и относится к машинному обучению. В этом подходе статистическая эвристика используется для определения наиболее важных предложений текста. Формулы для определения весов предложений описаны в теоретической части отчета. Извлечение предложений – это недорогой подход по сравнению с более наукоемкими более глубокими подходами, которые требуют дополнительных баз знаний, таких как онтологии или лингвистические знания.

Основным недостатком применения методов извлечения предложений к задаче реферирования является потеря связности итогового резюме. Тем не менее, резюме извлечения предложений могут дать ценные ключи к разгадке основных моментов документа и часто достаточно понятны для читателей.

```
for paragraph_index in range(len(paragraphs)):
    for sentence_index in range(len(paragraphs[paragraph_index]["paragraph"])):
        # calculate weight of sentence
        Pos_document = 1 - (count_of_symbols_before_in_doc / count_of_symbols_in_doc)
        Pos_paragraph = 1 - (count_of_symbols_before_in_paragraph /
count_of_symbols_in_paragraph)
        sentence_weight = Pos_document * Pos_paragraph
        paragraphs[paragraph_index]["paragraph"][sentence_index]["sentence_weight"] =
sentence_weight

        # add sentence to return list
        if sentence_weight > min_weight_in_sentences_to_return or
            len(sentences_to_return) < COUNT_OF_SENTENCES_TO_RETURN:
            ...
            # delete sentence with minimal weight
            if sentence_weight > min_weight_in_sentences_to_return and \
               len(sentences_to_return) >= COUNT_OF_SENTENCES_TO_RETURN:
                sentence_index_to_remove = -1
                # find lowest weight
                ...
            # remove sentence
            if sentence_index_to_remove >= 0:
                ...
            # add new sentence

            sentences_to_return.append(sentence)
            # update weight min
                ...
            return [sentence.capitalize() for sentence in sentences_to_return]
```

Код 9. Код метода Sentence Extraction (метод из методического пособия).

Метод Keywords Extraction

Сегодня, когда доступны большие лингвистические корпуса, значение tf – idf, которое возникло в поиске информации, может успешно применяться для идентификации ключевых слов текста: если, например, слово "кошка" "встречается значительно чаще в тексте, который нужно резюмировать (TF =" частота термина "), чем в корпусе (IDF означает" обратная частота документа "; здесь корпус означает" документ "), то" кошка ", скорее всего, быть важным словом текста; на самом деле текст может быть текстом о кошках.

На данном этапе работы использовались те самые корпуса в виде библиотеки spacy и ее модулей, при добавлении како-либо языка в систему требуется добавлять соответсвующий модуль дял spacy. Система была протестирована на модулях “ru_core_news_lg”, “it_cors_news_lg”. Обратите внимание, что система не будет работать с языками у которых отсутсвует модуль “lang_cors_news_lg”.

```
 @staticmethod
 def summarize(text: str, language) -> list:

     result = []
     cleared_text = TextProcessor.makeClearedTextFromRawHtmlText(text, True, True,
True)
     try:
          nlp = spacy.load(language + "_core_news_lg")
          nlp.add_pipe("yake")
          doc = nlp(cleared_text)
          for keyword, score in doc._.extract_keywords(n=15):
               result.append(f"{keyword}")
     except Exception as ex:
          print(ex)
     return list(result)
```

Код 10. Код метода Keywords Extraction.

Модифицированный Метод Луна (ML-метод)

Модификация, в отличии от первого реализованного метода, заключается в разбиение предложения на чанки, и вычисления их весов. В зависимости от весов этих чанков меняется и вес всего предложения. В данном случае не учитываются положения предложения в тексте, а учитывается его содержимое, токены которые в него входят. В современных терминах алгоритм звучит так:

- Вычисляем значимые слова документа:
    - Делаем стемминг или лемматизацию слов: разные словоформы одной леммы должны считаться как одно слово.
    - Считаем частоты слов, формируем список слов по убыванию частоты.
    - Убираем стоп-слова: частотные слова, у которых нет отдельной смысловой нагрузки, например предлоги и частицы.

    - Убираем слишком редкие слова, например такие, которые встречаются только 1 раз, либо убираем какой-то перцентиль слов по частоте.
    - Все оставшиеся слова считаем значимыми.
- Считаем значимость для предложений:
    - Предложение делим на промежутки, которые начинаются и заканчиваются значимыми словами. В промежутке могут быть и незначимые слова, но не более 4 подряд.
    - Значимость промежутка — квадрат количества значимых слов в промежутке, делённый на размер промежутка.
    - Значимость предложения — максимум из значимостей промежутков.
- Берём в качестве реферата предложения со значимостью выше определённого порога.

Пример вычисления значимости предложения приведён на рисунке 10. Красным обозначены стоп-слова, фиолетовым — незначимые слова, а зелёным — значимые слова.

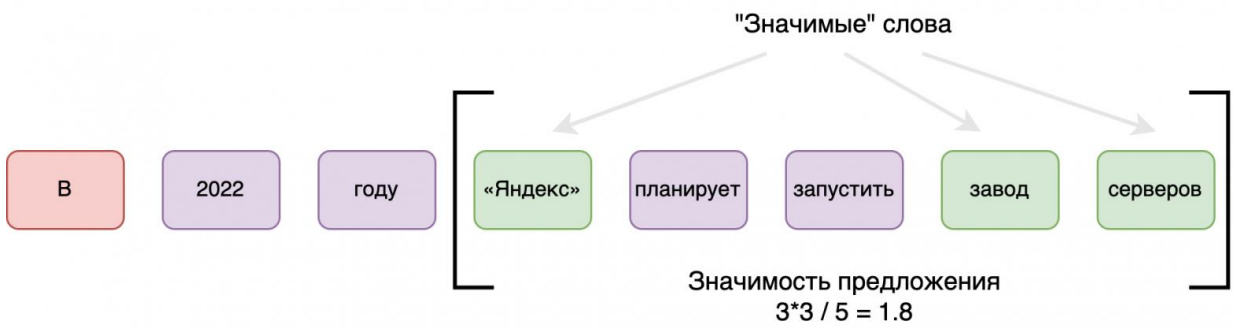

Рисунок 10. Пример вычисления значимости предложения.

Обычно для определения используется комбинация эвристик. Каждая эвристика присваивает предложению оценку (положительную или отрицательную). После применения всех эвристик предложения с наивысшей оценкой включаются в сводку. Отдельные эвристики оцениваются в зависимости от их важности.

Основные статьи, заложившие основы для многих используемых сегодня методов, были опубликованы Гансом Петером Луном в 1958 и 1969 годах. В своих работах Лун предложил придать больший вес предложениям в начале документа или абзаца.

Эдмундсон подчеркнул важность слов заголовков для резюмирования и был первым, кто использовал стоп-листы для фильтрации неинформативных слов с низким семантическим содержанием (например, большинство грамматических слов, таких как «of», «the», «a»). Он также различал бонусные слова и слова стигмы, то есть слова, которые, вероятно, встречаются вместе с важной (например, словоформа «значительный») или неважной информацией. Его идея использования ключевых слов, то есть слов, которые значительно чаще встречаются в документе, по-прежнему является одной из основных эвристик современных рефератов.

```
def __call__(self, text, target_sentences_count):
    # Считаем значимые токены.
    all_significant_tokens = self._getSignificantTokens(text)
    sentences = TextProcessor.tokenizeTextBySentences(text)

    # Считаем значимости предложений.
    ratings = []
    for sentence_index, sentence in enumerate(sentences):
        # Значимость предложений - максимум из значимостей промежутков.
        sentence_rating = max(self._getChunkRatings(sentence, all_significant_tokens))
        ratings.append((sentence_rating, sentence_index))

        # Сортируем предложения по значимости.
    ratings.sort(reverse=True)

    # Оставляем топовые и собираем реферат.
    ratings = ratings[:target_sentences_count]
    indices = [index for _, index in ratings]
    indices.sort()

    return " ".join([sentences[index] for index in indices])
```

Код 11. Код метода Луна.

Данный метод может не быть реализован с нуля, а использован из библиотеки sumy для языка python.

Тестирования системы и анализ методов реферирования текстов

Для тестирования системы было отобрано 10 текстов в предметной области «Информатика» и «Сочинения по литературе» как на русском так и на итальянском языках. Документы были помещены в папку documents и был запущен микросервер в локальной сети. Для каждого документа система создала краткие рефераты используя все три реализованных метода.

Исходя из полученных результатов, в качестве общих рекомендаций и замечаний по использованию различных методов автоматического реферирования текстов можно выделить следующие пункты:

6. При токенизации часть данных теряется и не восстанавливается, что влияет на качество итоговых рефератов.
7. Методы реферирования хороши для понимания общей мысли текста, но все же могут зачастую упускать действительно важную информацию из документа.
8. Для реферирования лучше использовать чистый текст без каких-либо мешающих элементов (к примеру названия кнопок при скачивании html-страницы из интернет), для этого надо использовать парсеры, токенайзеры и лемматизаторы.

Результаты работы системы представлены на рисунке 11.

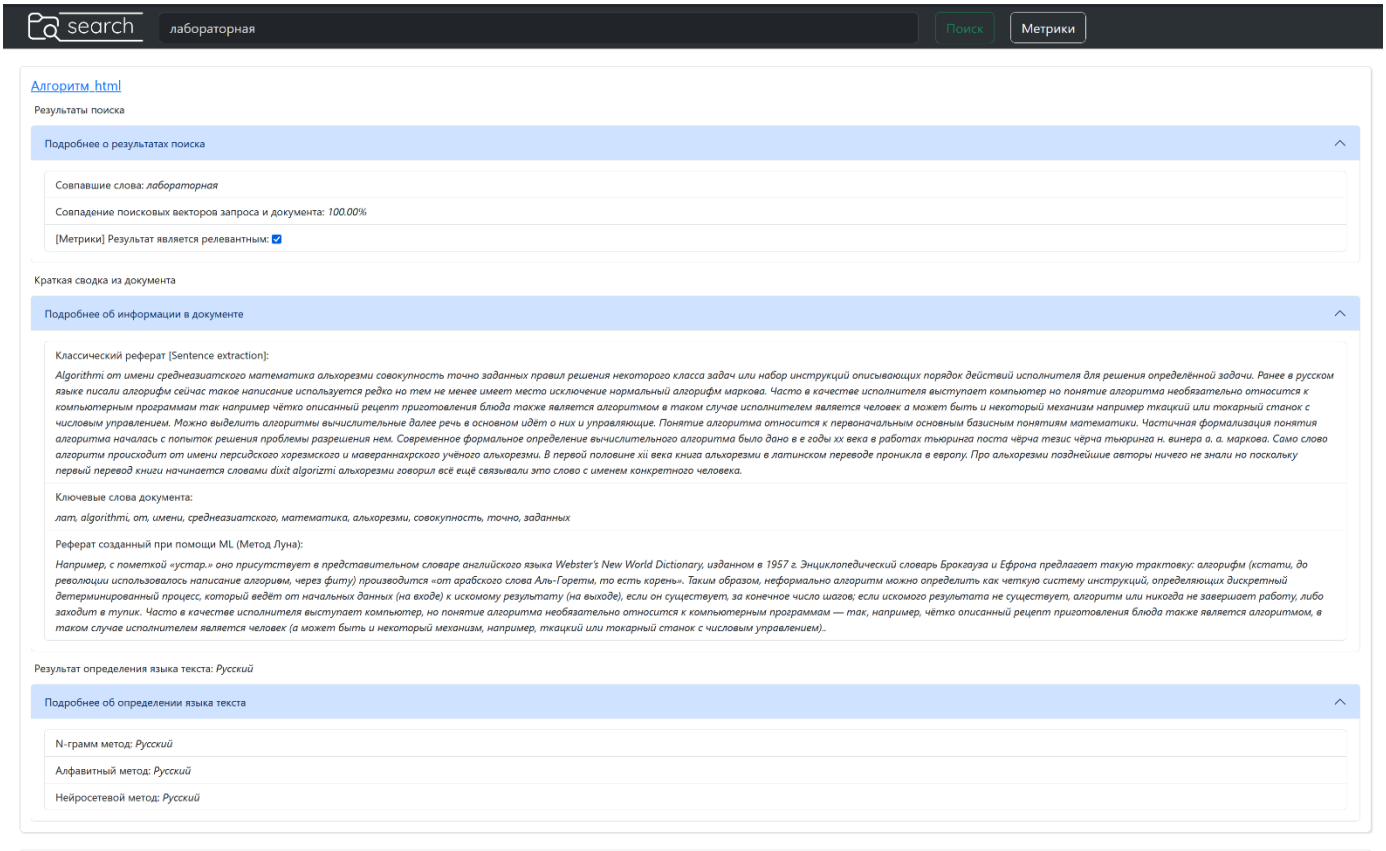

Рисунок 11. Результат выполнения поиска в системе и реферирования документов.

По итогам тестирования можно сравнить все методы реферирования.

1. Метод Sentence extraction – простой в реализации метод, информация, полученная после реферирования, дает представление о содержании документа. Скорость обработки самая высокая по сравнению с двумя другими методами.
2. Метод Keywords extraction – в реализации средний по сложности, требует больше оперативной памяти чем Sentence extraction. Выдает точные ключевые слова текста, однако для понятия всей сути содержания данного метода мало.
3. Метод Луна (ML-метод) – сложный в реализации, по сути своей объединяет Sentence extraction и Keywords extraction. Работает дольше других, однако результат реферирования является наиболее впечатляющим.

На основании результатов тестирования системы можно предложить следующие улучшения для созданной системы:

- попробовать использовать генеративные методы реферирования
- выбрать один лучший метод реферирования, или объединить, что бы по итогу получать один текст, дающий полное представление о содержании документа, а не несколько
- использовать ml-методы для поиска документов