Аннотация

Крупным компаниям с большим количеством накопленных знаний нужно удобно систематизировать свой опыт для быстрого поиска и ответа на вопросы.

Мы предлагаем текстовый редактор с совместным редактированием статей, удобной структурой документов, управлением доступами и умным поиском, который не просто ищет релевантные документы, а именно отвечает на вопрос пользователя.

Ключевые слова— Поиск по документам, совместное редактирование

Содержание

1 Введение

В современном мире инструменты совместной работы стали основой работы и продуктивности. Будь то совместная работа над документом, мозговой штурм или обмен мнениями, возможность беспрепятственно работать вместе имеет первостепенное значение. Целью этого проекта было создание текстового редактора с совместным редактированием, оснащенным умным поиском, который бы не просто выдавал релевантные документы, а именно отвечал на вопросы пользователей. Мы думали над множеством аспектов совместной работы, начиная от эффективного редактирования документов и заканчивая умным поиском информации по всей базе документов.

Одним из ключевых аспектов проекта стала интеграция функциональности умного поиска. Используя передовые технологии, мы разработали надежную модель поиска, которая не только определяет релевантные документы, но и предоставляет точные ответы на запросы пользователей, генерируя ответ на основе базы документов. Такая возможность поиска позволила пользователям быстро находить нужную информацию, повышая производительность и улучшая общий пользовательский опыт.

Кроме того, мы решали задачу обеспечения высокой надежности в среде совместного редактирования. Системы совместной работы часто сталкиваются с проблемами, связанными с одновременным редактированием, конфликтами и согласованностью данных. Благодаря тщательному планированию и реализации мы разработали высоконадежный алгоритм совместного редактирования, который эффектно управляет одновременными правками, разрешает конфликты и поддерживает согласованность данных.

Еще мы создали архитектуру с автоматическим развертыванием для создания кластера с высокой степенью доступности, которая позволяет легко адаптироваться под растущую пользовательскую нагрузку, а если что-то идет не так, то разработаны способы частичной деградации, которые позволят оставить в работе часть системы.

Далее мы углубимся в технические детали и проблемы, с которыми пришлось столкнуться при разработке редактора и поиска.

1.1 Актуальность проблемы

На рынке есть три основных типа текстовых редакторов:

- **Облачные**. Их преимущество в удобном интерфейсе и возможности совместного редактирования статьи: открыл сайт и можешь сразу начать создавать документы, но возникают неудобства, когда документов становится слишком много. Исходя из этого и того, что крупным компаниям важно хранить свои знания у себя локально, такие решения им не подходят.

- **Опенсорсные**. Главные недостаток таких редакторов, что их надо разворачивать и поддерживать самим. Никто не отвечает за качество и работоспособность, поэтому это тоже не подходит крупным компаниями.

- **Локальные**. Крупные компании предпочитают такие решения, потому что есть гарантия качества и можно поднять редактор у себя локально. Но главный недостаток, что у локальных продуктов нет живой обратной связи по продукту с разработчиками, как в клауд решениях, поэтому они развиваются очень медленно как в функциональном плане, так и в интерфейсе.

Мы хотим привнести все преимущества современных клауд редакторов и машинного обучения в корпоративный мир, чтобы экономить время сотрудников на редактировании и поиске.

1.2 Цели и задачи

У нас есть две глобальные цели: создание отказоустойчивого редактора и умного поиска, который бы не просто искал релевантные документы, но и отвечал на вопросы по ним. Более подробно:

- Создание интерфейса редактора, в котором можно будет работать сразу с большой базой документов

- Проектирование микросервисной архитектуры редактора, чтобы при поломке одной из частей функционала, система в целом оставалась бы рабочей

- Написание серверной части редактора, чтобы можно было нескольким людям одновременно редактировать документ

- Поддержка разного типа секций в редакторе: разметка маркдаун, вставка картинок, создание табличек

- Автоподнятие сервисов после падения
- Аунтефикация между сервисами
- Реализация инфраструктуры поиска для выдерживания большой нагрузки
- Оборачивание сервисов в контейнеры, чтобы не зависеть от архитектуры, на которой мы запускам редактор
- Исследование алгоритмов ранжирования документов
- Исследование алгоритмов поиска релевантной информации в документе по запросу
- Исследование возможности перефразирования вырванного из контекста ответа с помощью языковых моделей
- Проектирование микросервиса для сбора размеченных данных
- Сбор размеченных данных для моделей поиска
- Проектирование модульной архитектуры поиска для проведения экспериментов и подбора эффективного алгоритма поиска ответа на запрос

1.3 Аналоги

- Notion
  - \+ Интуитивно понятный и простой интерфейс
  - \+ Удобный и многофункциональный редактор
  - – Нельзя хранить информацию локально
  - – Нет хорошего поиска
  - – Нет системы доступов: все, кто имеют доступ к документу, могут его редактировать
- Wiki
  - \+ Можно хостить локально
  - – Неудобный интерфейс

  - Неудобный редактор
  - Нет качественного поиска
  - Нельзя совместно редактировать статью

- Confluence
  + Удобный интерфейс
  + Удобный редактор
  + Можно хостить локально
  - Нет качественного поиска
  - Нельзя совместно редактировать статью

2 Редактор

2.1 Дизайн

Мы хотим сделать редактор с совместным редактированием статей. Нужно, чтобы он работал бесперебойно в том числе при нестабильной сети. Должно быть автоматическое восстановление системы как при поломке на клиентской стороне, так и на серверной. Система должна быть готова к частичному отказу и не падать целиком.

Важная часть – это доступы, чтобы можно было создавать разные воркспейсы, приглашать в них других участников, удалять, а также выдавать доступы на сами статьи, не только на весь воркспейс.

Структура документов должна быть древовидная, чтобы можно было создавать статьи по подтемам внутри одного воркспейса.

В самом редакторе должен поддерживаться как просто текст, так и заголовки, списки и ту-ду листы.

2.2 Подход к проектированию

Архитектура проекта должна быть совместима как с локальным поднятием сервиса у клиента, так и с развертыванием инфраструктуры в облаке. Мы решили строить архитектуру сразу

клауд-нэйтив, чтобы заложить возможность быстрого масштабирования при увеличении нагрузки. Для этого мы выбрали микросервисную архитектуру.

Микросервисная архитектура хороша тем, что падение одного сервиса не рушит всю систему в целом, становится недоступна только одна небольшая часть. Более того это очень удобно в разработке, потому что у каждой команды есть свой сервис, за который она отвечает, и нет никаких конфликтов, когда несколько человек редактируют один и тот же участок кода.

Сервисы должны быть изолированы друг от друга и общаться через апи. Это позволяет при разработке одного сервиса не поднимать остальные и не мучать свой компьютер. Для этого все сервисы должны быть обложены тестами на каждый эндпоинт апи. Так можно будет гарантировать работоспособность сервиса без поднятия всей системы.

У каждого сервиса должна быть своя база данных, к которой имеет доступ только он. Так мы сможем обезопасить данные, потому что на уровне серсиса сможем контролировать, кому выдавать доступ к данным, а кому нет. И в случае если у сервиса слишком много данных, которые не умещаются в одну базу и нужно шардирование, то сервис сам должен за это отвечать и понимать, на каком шарде какие данные лежат.

Также важная особенность, которой должны обладать микросервисы, – отсутствие состояния, то есть, например, нельзя в сервисе иметь какую-то глобальную изменяемую переменную, которая влияет на поведение сервиса. Если нужно что-то сохранить, то делать это надо через базу данных в любом виде: будь-то реляционная база типа постгреса или какое-то key-value хранилище в виде редиса.

Все сервисы должны одинаково работать вне зависимости от того, где его запустили, в каком количестве и сколько он уже работает. Это нужно для того, чтобы мы могли легко масштабировать сервисы и сделать их отказоустойчивыми. Например, если у нас есть 10 реплик и половина из них упала, то после поднятия упавших система должна работать точно так же, как и до падения. Это и обеспечивает отсутствие состояния.

Заметим, что при падении, скажем, половины реплик у нас может не хватить ресурсов для обслуживания всех клиентов. В этом случае возможно несколько исходов при плохом проектировании

- Клиенты, на которых не хватило ресурсов просто получат ошибку на свой запрос
- Живые реплики от огромной на них нагрузки просто не будут обрабатывать ни один запрос, лягут, и в итоге у нас не останется ни одной живой реплики

Реальность скорее ближе ко второму исходу, потому что клиенты умные, и если сервер им не ответил или запрос упал по таймауту, то они перепошлют запрос и увеличат нагрузку на пустом месте в и так уже полумертвой системе, поэтому нам надо продумать, как спроектировать систему так, чтобы избежать коллапса при нехватке ресурсов.

Для этого используют очереди. Мы обрабатываем не напрямую запросы от пользователей, а вычитываем их из очереди, в которую балансировщик положил нам запросов. Таким образом оставшиеся живые реплики будут обрабатывать ровно столько запросов, сколько они способны обаботать, потому что они сами будут вычитывать задачи из очереди и контролировать свою нагрузку.

Главное по пути не сильно увлечься микросервисами и не делать на каждую функцию по своему сервису, как любят делать в некоторых компаниях. Подход должен быть выстроен вокруг одного типа данных, например, как в нашем случае, – статьи, картинки, пользователи. Если же дробить еще сильнее, то все преимущества микросервисов перекроет их главный недостаток – передача данных. Большее даже не сама передача, которая в рамках одного дата-центра достаточно быстрая, как подготовка и распаковка данных.

Сошлюсь на свой опыт работы в крупнейшей русской технологической компании, в которой я в том числе занимался оптимизацией микросерсива. Там, сняв флейм граф [5], я выяснил, что в зависимости от полезной нагрузки микросервиса распаковка и упаковка джейсонов после получение или перед отправлениям запроса занимает около 10-50% от всех потраченных ресурсов на обработку запроса. Это всегда важно понимать перед тем как дробить сервис или выделять новую сущность в отдельный микросервис, иначе мы будем жечь энергию впустую.

2.3 Архитектура сервисов

2.3.1 API

Мы пишем наши сервисы на FastAPI[7] – это Python фреймворк для написания вебсервисов совместимых с ASGI [3] протоколом, который создан для асинхронной работы. Запускаем сервисы через uvicorn [18] – это библиотека, реализующая в себе этот самый ASGI протокол, однако есть проблема с тем, что python позволяет запускать приложения только на одном ядре. Чтобы решить эту проблемы, мы просто поднимает несколько uvicorn воркеров и есть два вида их оркестрирования:

- Запускать сервис через gunicorn [9] с фиксированным количеством uvicorn воркеров
- Запускать все в одном потоке через uvicorn, но создать несколько реплик сервисов и реплицировать их Кубернетисом [11].

Первый способ очень хорош своей простотой. Сейчас мы используем его, пока нам хватает одной машины, дальше будем переходить на кубернетис. Из недостатков первого способа – фиксированное количество воркеров, которые мы должны как-то сами задавать в зависимости от того, сколько у нас ядер на машине и сколько мы хотим добавить мощностей в наш кластер воркеров. Также мы упираем в сам gunicorn балансировщик. И если у нас есть несколько инстансов гуникор пула, то надо добавлять нджинкс балансировщик между ними и его репликацию.

Прелесть второго подхода в том, что мы мыслим в терминах реплик, которые занимают только одно ядро, поэтому мы можем очень гибко контролировать нагрузку и по необходимости увеличивать и уменьшать на сколько угодно количество воркеров.

2.3.2 Данные

Для хранения данных мы выбрали Постгрес [14]. Эта база хороша тем, что она написана на Си, то есть быстрая, и для большинства новых проектов является серебряной пулей, которая умеет все, ее легко поднять и есть большое сообщество, которое всегда может помочь.

База данных – самое узкое место среди всего пайплайна обработки запросов, потому что она хранится на диске и доступ к нему на порядки медленнее, чем к оперативной памяти, поэтому мы хотим как можно реже лезть в базу и работать с ней как можно более эффективно. В частности мы хотим асинхронно взаимодействовать с базой, потому что чтение с HDD измеряется в милисекундах, соответственно, мы в каждый момент времени сможем обсудить на каждому воркере не больше одного клиента, потому что будет все время блокироваться на синхронном чтении с диска.

Если же мы будем читать асинхронно, то пока мы пошли на HDD за данным для одного клиента, то можем принять запрос от следующего и так далее. Для питона есть очень мощный драйвер asyncpg [4], который даже быстрее драйверов на го и выдает около двух миллионов строк в секунду. Его мы и решили взять.

Для работы с драйвером мы использовали ORM SQLAlchemy [16] и мигратор Alembic [2]. Первое нужно для самой работы с данным, формирования SQL запросов в базу данных и парсинга ответа от базы в структуру, с которой можно работать как с обычным классом. Эти

структуры называются моделями, в которых мы описываем, какие колонки хотим создать и использовать: указываем их имена и типы.

Но с помощью ORM мы можем только работать с данными и отсылать запрос. Нам нужно еще создать необходимы таблицы в самой базе по нашим моделям. Для этого мы используем утилиту Alembic, которая можем сама генерировать миграции и понимать, какие колонки мы изменили в моделях, чтобы создать и применить миграцию с поправками. Миграции – это просто SQL скрипты, которые создают или изменяют текущую структуры базы данных.

Как мы увидим дальше, на каждую статью у нас уходит много мета информации для совместного редактирования. Поэтому запросы и ответы содержащие данные мы должны сжимать.

2.4 Сервисы

2.4.1 Фронтенд

Входная точка редактора – фронтенд. Подробно на нем останавливаться не будем, это тема другой работы, расскажем только принцип доставки самих фронтендовских скриптов.

Сам фронтенд у нас написан на реакте. Для локального дебага мы просто поднимаем фронтненд через встроенный в npm http серевер, но такой подход не масштабируется, нам надо было выбрать супер быстрый и эффективный метод. Для этого мы весь наш фронтенд собираем в единый оптимизированный скрипт через npm build и дальше мы его можем раздавать клиентам через любой балансировщик как статик файл. Мы выбрали nginx.

В итоге мы сначала на одной машине разворачиваем nginx, кладем в него наш собранный фронтенд и раздаем его как статик файл. Так как по сути nginx для этого и создан, поэтому так мы можем выдержать сотни тысяч запросов на фронтенд, потому что на серверной части мы ничего не делаем для отображения фронтенда, все построение страницы происходит уже на клиента.

Дальше к нам подключаются преимущества микросервисной архитектуры. Чтобы что-то показать в редакторе, нам нужные данные, которых у нас пока нет. Чтобы их получить, мы идем в другие сервисы, но так как мы уже получили фронетнд, то мы можем пользователю отрисовать, например, иконку загрузки, и тем самым у пользователя не будет тупо белого экрана, пока мы ходим за его статьями. И такой подход у нас со всеми частями страницы: поиском, деревом статей и самим редактором.

Если же одной машины для фронтенда нам уже не хватает, то мы разворачиваем вторую

такую же и дополнительный nginx для балансировки между фронтенд машинами. Этим мы получаем бесконечную масштабируемость нашего фронтенда.

2.4.2 Пользователи и аунтефикация

Чтобы построить систему безопасных доступов, нам нужен сервис, который будет отвечать за пользователей. Мы хотим отличать запросы, которые послал наш клиент от злодейских. Для этого мы будем пользоваться стандартной практикой в виде JWT токена [37].

При регистрации от пользователя мы получаем логин и пароль, предварительно на фронтенде проверив их валидность. Далее мы хешируем пароль алгоритмом Argon2 [35], который сейчас считается самым лучшим и даже не требует дополнительнй соли, все потому что этому алгоритму нужно очень много памяти по сравнению с другими для хеширования, что сильно усложняет задачу брут-форса злоумышленникам.

При логине мы снова получаем почту с паролем и опять хешируем пароль, чтобы проверить с тем хешем, который есть в базе. Если хеше совпадают, то мы выдаем JWT токен с айдишником пользователя, кладем его в куку или хедеры, и теперь можем подтвержать, что пользователь авторизован именно нашим бекендом, а не кем-либо другим. Токен подписывается приватным ключом, а проверяется публичным по схеме RSA [43].

Так, при походе в другие наши сервисы, мы будем получать токен, смотреть, от какого пользователя пришел запросы, как-то по внутренне для сервиса логике проверять авторизацию и выдавать в зависимости от этого ответ. Например, в сервисе хранилища статей у нас есть база в воркспейсами и айдишниками пользователей, которые добавлены в этот воркспейсы. И при вызове эндопоинта по получению статей в этом воркспейсе мы достанем айди пользователя из токена, сходим в нашу базу, проверим, что у этого пользователй есть доступ, и только тогда выдадим ответ. Если же доступа нет или токен невалидный, то вернем ошибку с тем, что пользователь неавторизован.

Дополнительно у нас еще есть два эндопоинта для получения почты по айди пользователя и наоборот, они нужны для хранилища статей, чтобы по почте можно было добавить друга в свой воркспейс. Эти эндопоинты уже защищены проверкой токена по схеме выше.

Однако защита эндпоинтов позволит нам только лишь не увидеть то, к чему у нас нет доступа, но также нам важно, что наши данные не украдут по пути от сервера для клиента. В этом деле нам поможет https – это обычный http протокол, но поверх которого навешен TLS –

это протокол шифрования, который с помощью сертификата, который выдается специальными авторизованным организациями, гарантирует нам, что данные будут передаваться по зашифрованному каналу.

Важно заметить, что все запросы шифровать нет смысла. Нужно шифровать только то, что выходит за пределы дата-центра, то есть только сервис API Gateway, а все остальные запросы между сервисами внутри дата-центра мы можем передавать по обычном http, чтобы экономить на шифровании, которое далеко не бесплатное по времени.

2.4.3 Хранилище статей

Основная часть редактора – это хранилище статей. У этого сервиса есть всевозможное эндпоинты для CRUD [34] статей, воркспейсов и дерева статей. Работа со всеми данными относящимися к статьям должны происходит через этот сервис, то есть мы должны обеспечить безопасный доступ на чтение и изменение ко всем статьям и воркспейсам: проверять, что у пользователя есть доступ к запрашиваемым ресурсам. Это делается через токен, о котором мы рассказали выше. В токене хранится информация в айди пользователя, а в нашей базе хранится сопоставление на айди воркспейса и айди пользователя. С помощью этой таблички мы и понимаем, у кого к чему есть доступ.

Формат хранения статей мы рассмотрим в разделе про совместное редактирование, где увидим, что на каждую статью на пару страниц нам нужно порядка десятка тысяч строк записей в базу. Каждая строка у нас весит порядка полу килобайта или в сжатом виде около ста байт. Получаем, что на статью нам надо порядка мегабайта. Значит при $100 * 10^6$ статей у нас уже будет 10^9 строк в базе, которые будут занимать порядка сотен гигабайт. Тут уже постгрес начнет тужится и ему потребуется помощь. Мы будем понемногу добавлять ему шардом по ключу в виде айди статьи. Тогда при каждому запросе в базу мы сначала посчитаем хеш от айди статьи и поймем в какой из шардом на нужно будет пойти за статьей.

2.4.4 Веб-сокет менеджер

Этот сервис отвечает за прием и рассылку обновлений статьи. Когда пользователь открывает сайт, то с ним устанавливается веб-сокет соединение, через которое он отсылает и получает обновления статей. Делается это для того, чтобы в с друзьями вместе могли редактировать одни и те же документы в живом режиме и видели бы на своих экранах одно и то же. Автоматически

это добавляется автоматическое сохранение всех действий без надобности каждый раз тыкать кнопку сохранениея.

Сам интаснс веб-сокет менеджера хранит в себе словарь, где он знает по айди статьи, какие клиенты на нее подписались. Однако возникает проблема, когда мы запускаем несколько инстансов этого сервиса. Тогда у каждого их них будет свой словарик с клиентами, которые не пересекаются. Например, если пользователь А открыл какую-то статью и установил соединение с воркером 1, и клиент Б открыл эту же статью, но его запрос обработался воркером 2, тогда обновления статьи от пользователя А не дойдут до Б, потому что воркер 1 ничего не знает о соединении в пользователем Б на воркере 2, потому что каждый воркер независим. Это сделано из-за парадигмы стейт-лесс в микросервисах, которая позволяет легко увеличить количество инстансов сервиса для удержания большой нагрузки.

Для решения этой проблемы мы ввели посредника в виде key-value базы данных с каналами Redis [15], на которую можно подписаться. Каждый воркер подписывается на каналы статей, за которыми он сейчас следит, и тем самым получает обновления от других пользователей. Аналогично он шлет обновления в канал, когда сам получает сообщение по вебсокету от пользователя.

Однако там мы делаем наш канал узким местом, потому он только один, и при падении или его перезагрузке мы теряем сервис. Для этого мы сделаем кластер из редис нод и будем все наши обновления слать по стратегии Round-Robin, то есть сначала шлем обновление на первую ноду, потом на вторую и так далее. Это позволит снизить нагрузку на каждую конкретную ноду, но что самое удивительное, производительно будет расти линейно. В этом [42] эксперименте было показано, что одна нода редиса в режиме подписки выдерживает до 100 тысяч запросов в секунду и при увеличении кластера до 20 нод по схеме выше пронзительность растет линейно. В дальнейшем эксперименте у автора просто не было смысла.

2.4.5 Картинки

Хранилище картинок выделено в отдельный сервис, потому что ему нужно постоянное хранилище и много диска, что есть не на каждой машине. Сейчас для простоты мы храним картинки просто как файлы на диске и дополнительно у нас есть табличка, где мы для каждой картинки знаем ее воркспейс айди, чтобы аунтефицировать запросы и случайные пользователи не смогли бы получить доступ к чужим картинкам.

Самым массовым решением решением для хранения файлов являются S3 подобные храни-

лища, например, самое популярное открытое решение – MinIO [12], но есть огромное количество практические проблем с его разворачиванием и поддержанием [46], все-таки этот продукт все еще сырой.

Наш упор – это текст, а не картинки, нам вся мощь и быстрота S3 не нужна, поэтому мы просто будем сжимать картинки и хранить их на диске. В редакторах такого типа, как наш, пользователи не создают весь документ только из картинок. Мы предполагаем, что среднее количество картинок на каждый документ меньше или равно одному, поэтому на загрузку картинки можно дать до 1-2 секунд, потому что они подгружаются отдельно от текста, плюс мы знаем размер этой картинки и можем, пока она грузится пару секунд, сделать заглушку.

Дополнительно сделаем балансировщик, чтобы искать, на какой машине, какая картинка лежит, и нам этого хватит.

2.4.6 Поисковая строка

Этот сервис – прослойка между клиентом и поисковым движком. На вход он принимает вопрос, и отсылает его движку. После получения ответов может потребоваться склейка ответов, если они пересекаются, и добавления контекста слева и справа, чтобы было понятно, откуда вообще взялся ответ.

Также у этого сервиса есть изящное падение [6]. Сам поисковый движок очень массивный, у него сложная инфраструктура и ему требуется много ресурсов, поэтому выше риски его падения, но пользователю мы поломки поиска показывать не хотим. Для этого у поисковой строки есть ручка с пингом, которую дергает фронтенд, а поисковая строка посылает пинг запрос поисковому движку, и если ответ не пришел, то на фронтенде не отображаем поисковую строку. Этим мы скрываем от пользователя проблемы с поиском и не рушим остальную архитектуру. В этом и есть главное преимущество микросервисов – поломка части системы не влияет на ее работоспособность в целом.

2.4.7 API Gateway

Для всех сервисов нам нужно связующее звено, которое называют API Gateway. Все сервисмы могут находится на разных машинах, в том числе и один и тот же сервис может быть реплицирован по разные машинам. Соответственно, нам нужен кто-то, кто будет знать где и кто находится.

Заметим, что в кубернетисе есть встроенный инструмент оркестрации, который настраивается автоматически, но мы пока используем ручную вариант, потому что для кубернетис нужно хотя бы 3 ноды в кластере, а у нас всю нагрузку все еще может выдержать одна машина.

При поднятии очередной реплики любого из сервисов выше мы вносим в key-value хранилище хост и порт данной реплики. Далее эта запись прорастает в сам API Gateway, который у нас сделан на базе нджинкса, который будет балансировать нагрузку между всеми инстансами серсисов. Этим мы можем сделать распределенный и отказоустойчивый API Gateway.

2.5 Совместное редактирование

2.5.1 Базовые подходы

В совместном редактировании есть два основных подхода – Conflict-free replicated data type CRDT и Operational transformation OT. В отличие от классического сохранения документа, когда мы буквально берем все тело статьи и сохраняем его в базу, в совместном редактировании применяется техника обновлений.

На каждый изменение пользователем статьи мы формируем обновление какого-то формата и отсылаем его остальным пользователям и серверу. Для простоты далее будем считать сервер как дополнительного клиента. После того, как клиент получил обновление от другого пользователя, от применят его к своему документу и получает измененную версию другим пользователем. От обновлений мы хотим два свойства:

- Коммутативность
- Идемпотентность

Первое из них очень важно, потому что сеть не идеальна, и обновления спокойно могут переупорядочиваться, пока идут от одного клиента к другому. По схожим причинам нам нужна и идемпотентность, потому что обновления могут могут теряться, тогда мы его переотрпавим, оно дойдет, но тут вдруг дойдет и потерявшеейся обновление, и мы на клиенте применим два одинаковых обновления.

От системы в целом мы хотим согласованность в конечном счете, то есть если много пользователей редактируют один и тот же документ, то все они в конце концов увидят один и тот же результат. Самые наивные подходы такого не обеспечивают. Например, если у нас была строка

‘АББ‘ и пользователь 1 добавил букву Х после А и пользователь 2 добавил букву У после А, тогда если мы будем формировать обновления в абсолютных значениях, то есть первый вставил Х на позицию 1 и второй вставил У на позицию 1, то в конце концов первый увидит у себя документ ‘АУХББ‘, а второй ‘АХУББ‘.

2.5.2 CRDT

CRDT – это тип распределенной структуры данных, которая разработана для реплицирования и одновременного обновления без необходимости координации. Фундаментальный принцип CRDT заключается в том, что обновления структуры данных являются коммутативными и ассоциативными, что означает, что порядок применения обновлений не влияет на конечное состояние структуры данных. CRDT можно использовать для создания различных приложений для совместной работы, включая текстовые редакторы. Вообще говоря, текстовые редакторы – это всего лишь очень частный случай всей мощи CRDT. Как увидим дальше, мы будем хранить только одну букву в этой структуре, но можно хранить и произвольные структуры данных.

2.5.3 OT

OT, с другой стороны, представляет собой технику, используемую для синхронизации изменений, вносимых несколькими пользователями в общий документ. Она работает путем преобразования операций каждого пользователя таким образом, чтобы они могли быть применены к общему документу. Системы OT требуют наличия центрального сервера для координации действий пользователей. Этот сервер получает обновления от каждого пользователя и обеспечивает их преобразование и применение таким образом, чтобы сохранить намерения всех пользователей. Системы на основе OT были использованы для создания нескольких совместных текстовых редакторов, таких как Google Docs, однако, как оказалось, большинство статей посвещенных OT, оказались провальными, потому что не обеспечивали согласованность в коненом счете [39].

2.5.4 PeriText

CRDT более масштабируем и отказоустойчив, хоть и более сложен в реализации, и мы выбрали его. В качестве алгоритма мы взяли работу PeriText [40], которая как раз была написана для текстовых редакторов, в которых не только сам текст, но и разное форматирование типа жирного или курсивного шрифта, заголовки и тому подобное. И это очень важная делать, потому

что все классические подходы не работают с такого вида редакторами. Приведем пример.

Допустим у нас был html текст `Я купил ежа`. Пусть пользователь А выделил ежа курсивом, а Б жирным шрифтом, тогда все обычные алгоритмы нам выдадут ответ вида `Я купил <i>ежа</i><b>ежа</b>`, но мы хотели получить `Я купил <i><b>ежа</b></i>`, поэтому авторы статьи предлагают рассматривать форматирование как отдельный вид данных отличный от текста.

PeriText был специально разработан, чтобы можно было совместно редактировать документ даже после долгого разрыва связи. Например, у вас есть документ, который вы редактируете вместе с другом, но не хотите, чтобы он в живом режиме видел все ваши изменения, а хотите сначала отредактировать все в режиме черновика. С упором на эту особенность и был разработан алгоритм, но важно, что это не несет недостатков для случая, когда вы изменяете документ онлайн.

2.5.5 Структура данных

В базе данных мы будем хранить не всю статью, а просто список из всех ее обновления. Такой вид структуры данных называется иммутабельной, потому что из нее мы ничего не удаляем, а только добавляем, то есть у нас получается так называемый полный лог изменения. Это дает нам большое преимущество в том, что любой пользователь может откатить свои изменения назад и посмотреть всю историю изменения документа, но также привносит трудности в виде большого объема хранимых данных для каждой статьи, но это проблему мы уже рассмотрели в разделе про хранилище.

Вообще говоря, мы не можем удалять старые обновления для согласованности, но если учесть, что мы делаем онлайн редактор и все обновления доходят за разумное время и пользователи не отказывают изменения на год назад, то можно удалять обновления, которые хотя бы недельной давности. Таким образом мы сможем эффективно поддержать более большие документы.

2.5.6 Изменение текста

Для начала рассмотрим формирование обновлений по изменению текста статьи. На каждый клик клавиатуры мы смотрим, чем текущая версия отличается от предыдущей и в зависимости от того, что мы сделали: удалили символ, добавили, либо и то, и другое, формируем обновления

двух типов:

```
{ action: "insert", opId: "2@alice", afterId: "1@alice", character: "x" }
```

```
{ action: "remove", opId: "5@alice", removedId: "2@alice" }
```

Здесь и далее у нас opId формируется из двух частей: номер операции и айди клинета. Второе всегда фиксированное, мы просто берем айди нашего пользователя, который нам выдал сервис пользователей, а номер операции мы инициализируем как максимальный номер операции из существующий обновления плюс один, а дальше с каждой нашей операцией увеличиваем его на один. Еще при каждом новом обновлении от других пользователей проверяем, что их номер операции не превосходит наш минус один, иначе мы увеличиваем наш счетчик.

Как видим, в обновлении мы никак не используем абсолютную позицию измененного символа, как это делает в OT, а используем только условные идентификаторы, по которым однозначно можем понять, куда нам вставлять, или какой из символов удалить.

2.5.7 Изменение форматирования

После того, как мы поняли, какие буквы изменились, нам надо получить форматирование нового документа. Для этого на клиенте мы просто проходимся по всему тексту и формируем словарь из форматирований и позиций, куда применено это форматирование. Далее смотрим, что поменялось и формируем обновления вида:

```
{
    action: "addMark",
    opId: "18@A",
    start: { type: "before", opId: "5@A" },
    end: { type: "before", opId: "17@B" },
    markType: "bold"
}
```

2.5.8 Применение обновлений

Все изменения от одного клиента через сервер рассылаются остальным, которые подписаны на ту же статью. Чтобы мы могли генерировать и применять обновления у клиента дополнительно хранится карта на индекс в текущем документе и opId соответствующего символа.

Допустим к нам пришло обновление со вставкой символа с айди 19@B после айди 18@A. Мы находим индекс элемента, которому соответствует следующему за 18@A, и далее идем вправо до тех пор, пока 19@B будет больше следующего за текущим. Порядок у нас лексиграфический по частям айди, то есть сначала сравниваем номер операции, если они равные, то сравниваем айдишники пользователей. Посмотрим на примере:

Пусть изначально был текст “ABC“ с айди `[1@A, 2@A, 3@A]`

- **Пользователь B**
  - Вставил D с айди 3@B после айди 2A
  - Локально видит текст “ABDC“ с айдишниками `[1@A, 2@A, 3@B 3@A]`, а у C будет “ABEC“ с айди `[1@A, 2@A, 3@C 3@A]`
  - Пришло обновление от C со вставкой E с айди 3@C после 2@A
  - Видим, что после 2@A стоит 3@B, но $3@C > 3@B \Leftrightarrow 3 = 3 \& C > B$, значит оставляем E после C
  - Получаем итоговый текст “ABEDC“ с айдишниками `[1@A, 2@A, 3@C, 3@B 3@A]`
- **Пользователь C**
  - Вставил E с айди 3@C после айди 2A
  - Локально видит текст “ABEC“ с айдишниками `[1@A, 2@A, 3@C 3@A]`
  - Пришло обновление от B со вставкой D с айди 3@B после 2@A
  - Видим, что после 2@A стоит 3@C, но $3@B < 3@C \Leftrightarrow 3 = 3 \& B < C$, значит двигаемся дальше
  - Видим, что после 3@C стоит 3@A, но $3@B > 3@A \Leftrightarrow 3 = 3 \& B > A$, значит оставляем D после E
  - Получаем итоговый текст “ABEDC“ с айдишниками `[1@A, 2@A, 3@C, 3@B 3@A]`

Как видим, оба пользователя в конечном счете сошлись к одному и тому же документу.

2.5.9 Формирование обновлений

С применением обновлений никаких проблем нет, мы показали однозначность и простоту применения. Самое сложное – это сформировать обновления.

Мы используем на фронтенде редактор TinyMCE [17], который выдает нам документ в html формате. Нам надо отделить текст от форматирование, чтобы применить алгоритм. Для этого мы написали свой парсер, который по заданному абзацу выдает сопоставления на все типы форматирований и позиции, на которых оно применено. На этой основе далее мы формируем обновления по форматированию.

2.5.10 Разрыв соединения

Если соединение разорвалось, то до клиента могла не дойти часть обновления от другого пользователя, и тогда документ может быть у разных пользователей разный, чего мы не хотим. Для этого раз в какое-то время клиент запрашивает от сервера все обновления для открытой статьи, и применяет те, которых еще нет.

Также при разрыве у нас возникает проблема с отправкой наших обновлений. В этом случае при ошибке при отправке мы записываем наше изменение в очередь цикла событий на клиенте с определенным таймаутом. То есть если сейчас мы не смогли отправить обновления из-за ошибки вебсокета, то мы говорим браузеру, что запомни это сообщения и попытайся отправить его еще раз через определенное время. Если и тогда не получилось, то пытайся дальше.

Для удобства пользователей мы показываем статус сохранения, который активируется тогда, когда мы начинаем отправлять сообщение, и убирается тогда, когда закончили. В случае ошибки отправки мы не убираем статус и ждем успешной отправки.

2.6 Локализация

Мы хотим, чтобы нашим редактором могли пользоваться люди из любой страны, поэтому нам нужно поддержать версию для нескольких языков.

Для этого есть специальная утилита gettext [8], которая принимает специальный скомпилированный файл `*.mo`, в котором заданы переводы всех нужных фраз на заданный язык, и она может эффективно искать перевод.

Чтобы получить этот файл сначала нам нужно найти, какие фразы нам в принципе нужно перевести. Для этого есть специальный парсер, который ищет конструции со специальным

синтаксисом, например:

```
_ = get_translator(Language.RUS)


workspace = Workspace(
    name=_("New workspace"),
)
```

В данном случае парсер выделит фразу ‘New workspace‘ и добавит ее в файл `*.po`, в котором ее нужно будет перевести на нужные языки, чтобы в итоге получить файл `*.mo` для работы gettext.

Синтаксис с нижним подчеркиванием – это специальное соглашение для вызова gettext, по которому можно понять, что мы хотим перевести. В нашем случае мы определяем нижнее подчеркивание через нашу функцию `get_translator`, которая инициализирует gettext по заданному языку и передет ему нужный `*.mo` файл.

Чтобы понять, какой язык нам выбрать, мы попросим, чтобы фронтенд добавлял заголовок `Accept-Language` в http запросы к бекенду с выбранным пользователем языком, а фронтенд в свою очередь поймет, какой язык ставить, с помощью обычного переключателя в редакторе.

Далее при получении запроса от клиента, мы будем смотреть на этот http заголовок и понимать, какой язык для gettext нам нужно выбрать.

2.7 Испытания

После добавление нового или изменения функционала мы хотим убедиться, что система по отдельности и в целом продолжает работать. Для этого нам нужно сделать инфраструктуру для испытаний как всех сервисов по отдельности, так и всего приложения целиком.

2.7.1 Единичные испытания

Для тестирования мы используем PyTest. Единичные испытания нужны для проверки логической части отдельных функций. Например, мы написали алгоритм для совместного редактирования статей, и хотим проверить, что он корректен. Для этого мы пишем серию проверок для каждой из функций этого алгоритма. Важно, что на этом этапе испытаний мы тестируем сервис независимо. Если есть какие-то запросы в базу данных или к другим сервисам, то их

мы мокаем, то есть выдаем либо случайный ответ, либо какой-то очень упрощенный, но важно соблюсти формат моков точно такой же, как у реального ответа, иначе тестирование будет бесполезным.

Такая изолированность единичной стадии дает разработчику сервиса быстро писать код и проверять результат без зависимости от других сервисов.

2.7.2 Соединительные испытания

После того, как мы проверили, что логические части нашего сервиса работают корректно по отдельности, надо убедиться, что сам сервис в целом работает тоже верно. Мы хотим прогнать запросы на все ручки через весь сервис. Для этого у FastAPI есть тестовый клиент, который симулирует реальное разворачивание нашего сервиса через uvicorn, к которому мы делаем обычные http запросы из PyTest.

На этой стадии мы от сервиса должны дать гарантию, что по модулю работоспособности других сервисов и соблюдения ими обговоренного формата данных, который мы зашили в моки, наш сервис будет работать корректно и стабильно.

2.7.3 Испытательный стенд

Когда все сервисы независимо друг от друга нам дали гарантию, что они корректны, мы хотим проверить, что система целиком работает тоже корректно, то есть что соблюдаются форматы данных, инфраструктура выдерживает нагрузку и никто не падает.

Для этого мы разворачиваем тестовое окружения, которое полностью дублирует настоящее производственное, но просто в меньших масштабах. Там каждый разработчик может выкатить свое протестированное изменение и проверить, что при реальном взаимодействие между сервисами, его сервис не падает и сайт целиком работает так, как и ожидалось.

2.8 Контейнеризация

После того, как все изменения были протестированы и независимо друг от друга по сервисам, и все вместе на тестовом стенде, мы готовы выкатить новый релиз в продакшн. Мы хотим, чтобы релиз для пользователей прошел незаметно и в случае чего мы могли бы откатиться назад. В этом нам поможет контейнеризация.

У каждого из наших сервисов есть свой докер-контейнер, в котором уже скачаны все нужные зависимости и собран полный пакет для запуска. Виртуализацией мы гарантируем, что наш сервис точно запустится и будет работать одинаково на любых машинах.

При выкатке новой версии мы запускаем ее рядом с текущей, и когда новый контейнер запустился, перенаправляем трафик на него и удаляем контейнер с предыдущей версией. Аналогично мы можем откатиться на старую версию в случае ошибки в новой.

Контейнеры мы используем и для тестирования, потому что у всех разработчиком разные операционные системы и, чтобы не подстраиваться подо всех, проще всем разрабатывать сразу в контейнере, в котором у всех все будет одинаково и не будет неожиданностей связанных с тем, что локально все работает, но на сервере почему-то код не хочет запускаться.

Для тестирования и простоты развертки всей системы у себя локально у нас есть докер-компоуз файлы, который позволяет одной командой запусить все сервисы. Но в продакщене мы этого сделать не сможем, потому что нам нужно контролировать количество реплик, поэтому все контейнеры приходится запускать и контролировать их количество вручную, пока мы не переехали в кубернетис.

2.9 Итоги

На этой стадии мы создали пользовательскую и серверную части редактора.

В пользовательской части мы проработали интерфейс, продумали сценарии взаимодействия, чтобы было удобно пользоваться. Сделали так, чтобы ошибки бекенда не крушили бы весь сайт, а либо отключали часть необязательного функционала, как поиска, либо контролируемо выдавали пользователю сообщение с ошибкой и просьбой подождать или перезагрузить сайт, если, например, отказало хранилище статей, но оба этих случая не рушат фронтенд, а только не показывают часть информации.

Чтобы доставлять пользовательскую часть максимально быстро и код бы исполнялся эффективно, мы собираем весь пользовательский код в один оптимизированный файл, который потом раздаем балансировщиком. Прелесть подхода в том, что балансировщик очень легковесный и его можно легко реплицировать. Этим мы получаем то, что даже при падении сложной серверной части, пользовательская всегда будет доступна, и, когда все будет плохо, мы будем показывать не пустой экран, а заранее продуманный сценарий.

На серверной части мы спроектировали и реализовали всю необходимую инфраструкту-

ру для совместного редактирования статей и взаимодействия с поисковым движком. Сделали архитектуру отказоустойчивой, масштабируемой и легко выкатываемой в производство. Для поверкок мы настроили тестовое окружения с тремя видами испытаний, которые могут нам гарантировать, что при выкатке новой версии в производство у нас все будет работать так, как мы ожидаем, но в случае провала у нас есть инструменты для отката на предыдущую версию.

Архитектуру серверной части мы выбрали микросервисной, чтобы распределить риски ошибок между сервисами и ломать не всю систему, а ее небольшую часть. Маленькие сервисы более удобны для больших проектов, где есть несколько команд. Так мы можем каждой доверить свой серсвис и она будет за нее отвечать, проектировать, тестировать, выкатывать и не зависеть от остальных команд, как в случае монолитной архитектуры. Также микросервисы хороши тем, что они достаточно небольшие, что позволяет их выкатывать быстро в производство. Так мы сможем быстрее выкатить срочное изменение, потому что надо будет как меньше согласований, так и процесс сборки будет быстрее, чем в мнонолитном приложении.

Дальше расскажем о разработке поискового движка.

3 Поиск

3.1 Дизайн

Мы хотим реализовать микросервис, ответственный за систему поиска. Кроме того, нам требуется разработать удобный интерфейс для проведения исследований и экспериментов. В связи с тем, что процесс поиска состоит из нескольких этапов, включая обработку документов, обработку поисковых запросов и формирование окончательных ответов, мы решили использовать модульную систему, в которой каждый метод представлен отдельным классом. Затем эти модули будут объединяться в ациклический граф вычислений. Такой подход позволяет нам создать легко масштабируемую систему, которая позволяет изменять различные этапы поиска и встраивать новые компоненты в общую цепочку обработки. Кроме того, такая изолированная структура обеспечивает простоту тестирования всех компонентов системы.

3.2 Обзор используемых библиотек

3.2.1 Haystack [10]

Для реализации поиска мы выбрали библиотеку Haystack, она представляет из себя инструментарий для построения систем вопросов и ответов с использованием методов извлечения информации (Information Retrieval) и машинного чтения (Machine Reading).

Основные компоненты Haystack включают:

- **Ретривер (Retriever):** Компонент, отвечающий за поиск релевантных документов в коллекции данных. В Haystack реализовована часть базовых алгоритмов ранжирования.
- **Ридер (Reader):** Компонент, который анализирует релевантные документы и пытается извлечь ответы на заданные вопросы. Haystack предоставляет интерфейс для использования готовых языковых моделей в ридерах.
- **Пайплайны (Pipelines) 1:** Haystack предоставляет возможность создания пайплайнов, которые объединяют ретриверы и ридеры для выполнения комплексных операций поиска и ответа на вопросы. Пайплайны позволяют оптимизировать, настраивать и добавлять собственные модули в процесс обработки текстовых данных.
- **Интерфейсы для баз данных:** Haystack предлагает удобные интерфейсы для работы с популярными базами данных, такими как Elasticsearch, SQL и другими. Это упрощает интеграцию существующих систем хранения данных в проекты Haystack.
- **Утилиты для обработки текстов:** Haystack предоставляет набор утилит для предварительной обработки текстовых данных, включая токенизацию, удаление стоп-слов, лемматизацию и другие операции, которые помогают улучшить качество поиска и извлечения информации.

Таким образом, Haystack предоставляет нам удобный интерфейс для реализации всех наших требований к архитектуре поиска и базе для проведения экспериментов.

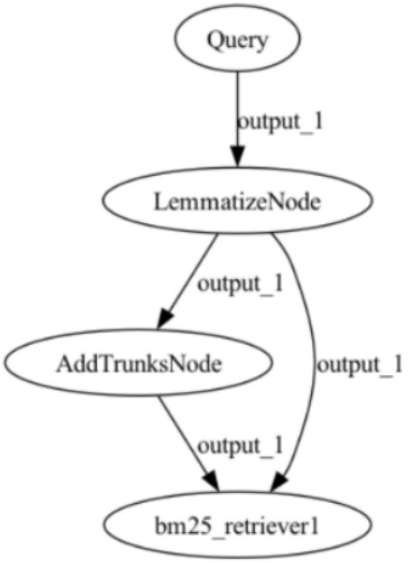

Рис. 1: Пример пайплайна

3.2.2 Natasha [13]

Набор библиотек Natasha - это коллекция инструментов для обработки естественного языка на русском языке. Он предоставляет функциональность для выполнения различных задач, связанных с обработкой и анализом текста, включая разбор слов, извлечение именованных сущностей, лемматизацию и морфологический анализ.

Основные компоненты набора библиотек Natasha включают:

1. **Natasha NER (Named Entity Recognition):** Этот компонент предназначен для извлечения именованных сущностей из текста, таких как имена людей, организации, местоположения и другие важные сущности. Он использует модели машинного обучения для распознавания и классификации именованных сущностей.

2. **Natasha Morph:** Этот компонент предоставляет возможности морфологического анализа текста на русском языке. Он позволяет разбирать слова на составные части, такие как основа, часть речи, падеж, число и т. д.

3. **Natasha Doc:** Этот компонент обеспечивает функции для обработки текстовых документов. Он позволяет извлекать предложения, разбивать текст на абзацы, а также выполнять простые операции поиск и замена.

4. **Natasha Lemmatizer:** Этот компонент предоставляет возможность лемматизации слов

на русском языке. Лемматизация - это процесс приведения слова к его базовой форме (лемме), что упрощает анализ и сравнение текста.

5. **Natasha Navec:** Этот компонент предоставляет собой нейросетевую модель для векторного представления слов на русском языке и простой интерфейс для работы с ними. Он позволяет получить векторное представление для отдельного слова или выполнить операции над векторами, например нахождение наиболее похожих слов.

Мы используем данную библиотеку для обработки документов и предобработки поисковых запросов, а также для генерации качественных русскоязычных векторных представлений

3.3 Архитектура

Наш микросервис взаимодействует с редактором с использованием API, реализованного с помощью библиотеки FastAPI [7], о которой было упомянуто ранее.

3.3.1 `NeuroSearch`

Главный класс проекта, который вызывается в обработчиках API. Он инициализируется с помощью следующих компонентов: `document_store`, `documents_preprocess`, `base_retriever`, `navec_retri` `reader`, `paraphraser`. Класс реализует логику взаимодействия между этими компонентами, которая не может быть выражена в виде пайплайнов из библиотеки Haystack из-за их ограниченных возможностей.

- `document_store`: представляет собой наше хранилище информации. Он инициализируется отдельно, т.к. для локального тестирования мы не хотим использовать.
- `documents_preprocess`: В различных фазах поиска может потребоваться хранение различных подструктур документов в нашем хранилище, таких как названия, слова или подмножества строк. С помощью этого метода мы получаем один документ и возвращаем набор необходимых нам подструктур. `ElasticsearchDocumentStore`, и удобнее использовать `InMemoryDocumentStore` или `SQLDocumentStore`.
- `base_retriever`: это пайплайн для обработки запроса, поиска релевантных документов и постобработки документов для фазы ридера.

- `navec_retriever`: необходим для обработки документов при добавлении их в `document_store`, а также для создания эмбеддингов слов.
- `reader`: экземпляр ридера, который извлекает релвентный отсносительно запроса контекст из документов, полученных ретривером.
- `paraphraser`: компонент, ответственный за переформулировку извлеченного ридером контекста.

Таким образом, мы получили удобную модульную структуру поискового движка, которая позволяет собирать пайплайн поиска из готовых частей, комбинировать их между собой и легко проводить A/B-тестирование на каждую из компонент в будущем.

3.3.2 API

В API доступны три эндпоинта.

- `GET /answer_question`: принимает вопрос и источник, в котором нужно найти ответ. В ответ возвращается список структур, содержащих переформулированный ответ, контекст из документа, сам документ и некоторую служебную информацию
- `POST /add_entry`: принимает документ, который описывается следующими полями: id, content, url, title, subtitles, source. Документ добавляется в указанный источник. Если переданный документ уже существует в базе данных, он будет обновлен.
- `POST /remove_entry`: принимает id документа и удаляет его из хранилища.

3.3.3 `NavecNodes`

Во многих модулях нашей системы используется библиотека `Navec` и несколько из ее ресурсоемких компонентов. Для предотвращения дублирования кода и облегчения его поддержки мы разработали класс `NavecPreprocessor`. Этот класс обеспечивает доступ к ресурсоемким компонентам и реализует все необходимые методы, которые используют компоненты библиотеки `Natasha`.

Поимимо этого, реализован класс `NavecRetriever`, которые наследуется от `NavecPreprocessor` и `DenseRetriever`. Он используется для поиска информации с использованием эмбеддингов из

библиотеки Navec. Первое наследование необходимо для доступа ко всем функциям, предоставляемым этим классом, а второе - для интеграции нашего ретривера в пайплайны обработки данных.

Также, мы передаем экземпляр класса NavecRetriever во многие модули для обработки данных

3.3.4 Nodes

В ходе нашей работы мы экспериментировали с разными методами обработки текстовых данных. Для удобства формирования пайплайнов мы создали отдельные классы для каждого метода. Чтобы можно было включать эти классы в пайплайны, они должны быть унаследованы от класса BaseComponent, который представлен в библиотеке Haystack, и реализовывать метод run.

- LemmatizeNode – лемматизирует текст
  - __init__: принимает экземпляр NavecRetriever
  - run: принимает текст и лемматизирует его
- UpdBestWindowNode – осуществляет постобработку документов для фазы ридера. Обновляет документы и добавляет новое поле, содержащее лучшую подстроку относительно релевантности к запросу.
  - __init__: принимает экземпляр NavecRetriever
  - run: принимает документ и вопрос, выполняет постобработку
- AddTrunksNode – добавляет префикс с указанной длины рядом с каждым словом в тексте.
  - __init__: принимает длину подстроки
  - run: принмиает текст и добавляет префиксы
- AddSynonymsNode – расширяет текст синонимами, которые ищутся с помощью эмбеддингов из Navec
  - __init__: принимает NavecRetriever и количество синонимов которое необходимо добавить
  - run: принимает текст и дополняет его синонимами

3.4 Представление документов

Будем хранить векторное представление наших документов, которое отображает их в пространство признаков. Эти векторы будем создавать при помощи нейронных сетей или алгоритмов обработки естественного языка, например рекурентные нейронные сети, BERT [36] или TF-IDF [41]. Для поиска будем использовать две основные компоненты - Reader [29] и Retriever [30].

3.5 Хранение данных

При поиске мы не можем просто взять, пройтись по всем документам и найти в них нужный ответ. Нам нужна индексация, некий предпосчет, который нам позволит быстро искать ответ по большому объему документов. Для этого мы используем популярное решение ElasticSearch [28], которое позволяет нам быстро выполнить процедуру Retriever для последующей обработки топа релевантных документов с помощью машинного обучения.

3.6 Retriever

Retriever – это компонент, который находит наиболее релевантные документы, связанные с запросом пользователя. Он выполняет первоначальную фильтрацию документов и сокращает объем данных, которые должны быть обработаны более мощными инструментами, такими как Reader.

Retriever использует различные методы для поиска наиболее релевантных документов. Один из таких методов – это поиск по ключевым словам, когда Retriever ищет документы, которые содержат определенные ключевые слова, связанные с запросом пользователя. Этот метод является наиболее простым и быстрым, но не всегда является самым эффективным, так как некоторые документы могут не содержать ключевых слов, но всё же могут быть релевантными.

Другой метод, используемый Retriever, – это поиск по векторному представлению. Векторное представление – это математическое представление слов, фраз и документов в виде векторов чисел. Этот метод позволяет Retriever находить более релевантные документы, несмотря на то, что они могут не содержать ключевых слов, но иметь схожий контекст и смысл.

Retriever может использовать различные модели машинного обучения для создания векторных представлений документов, такие как TF-IDF (term frequency-inverse document frequency), BM25 [44] (best match 25), а также более сложные модели, основанные на нейронных сетях,

такие как DENSE [38] (document expansion via neural sentence embeddings), тут не эффективно использовать большие модели, например BERT (Bidirectional Encoder Representations from Transformers), так как они работают сильно дольше.

Для того, чтобы Retriever мог правильно выбрать наиболее релевантные документы, он должен быть обучен на большом наборе данных, содержащем запросы и связанные с ними документы. Во время обучения Retriever пытается научиться правильно выбирать наиболее релевантные документы для каждого запроса.

3.6.1 Тестирование

Для тестирования Retriever'а в качестве базы данных был выбран Уголовный кодекс Российской Федерации, что обосновано несколькими причинами:

- Ограниченный объем базы данных позволяет успешно проводить тесты на персональных устройствах или недорогих серверах.
- Существует доступный сайт, с которого легко получить эту базу данных.
- Эта база данных популярна и позволяет легко найти вопросы, связанные с ней.
- Вопросы могут быть сформулированы таким образом, что ответ на каждый из них содержится только в одной статье Уголовного кодекса, что удобно для тестирования алгоритмов.
- Уголовный кодекс обладает удобной структурой документов.

На основе данной базы данных был сформирован размеченный датасет с вопросами и статьями в качестве таргета, содержащий примерно 500 вопросов, собранных из следующих источников:

- Генерация ChatGPT.
- Юридические тесты.
- Ручная разметка.

Для формирования размеченной выборки вопросов и ответов были разработаны специальные инструменты, которые позволяли обрабатывать различные форматы вопросов и ответов и приводить их к необходимому виду. Мы также проверили все вопросы из внешних источников следующим образом: рассмотрели некоторый процент вопросов с ответами и, если все было верно, оставляли всю выборку. Если были обнаружены ошибки, мы вручную проверяли каждый вопрос. Мы также обращали внимание на вопросы, на которых модель показывала плохие результаты, и проверяли их на наличие ошибок.

3.6.2 `BM25Retriever`

Наш первый метод для ранжирования документов, который мы рассмотрели. Он имплементирован в используемой библиотеке haystack. Пусть наш запрос q состоит из слов q_i, где $1 \leq i \leq n$, тогда будем рассчитывать расстояние между запросом и документом по следующей формуле

$$\text{BM25}(q, d) = \sum_{i=1}^{n} \text{IDF}(q_i) \frac{\text{tf}(q_i, d)(k_1 + 1)}{\text{tf}(q_i, d) + k_1(1 - b + b\frac{|D|}{\bar{n}_D})}$$

$$\text{IDF}(q_i) = \log \frac{|D|}{|\{d \in D | q_i \in d\}|}$$

Где $\text{IDF}(q_i)$ – логарифм обратной доли документов содержащих данное слово, $\text{tf}(q_i, d)$ – число вхождений q_i в документ d, $|D|$ – число документов в выборке, $\bar{n}_D$ – средняя длина документа в коллекции, а k_1 и b являются гиперпараметрами.

Интуитивно мы считаем, что релевантность документа зависит от количества совпадающих терминов и их значимости в документе, также мы учитываем длину документа, чтобы балансировать длинные документы в которых термины могут появляться чаще из-за объема. tf – измеряет частоту термина, чем чаще термин встречается, тем больше его вес, IDF – отвечает за важность термина в документах, если термин редкий и встречается только в некоторых документах, его IDF будет высоким, что увеличит его вес.

Top 1 accuracy	Top 3 accuracy	Top 5 accuracy	Mean resposne time
0.33	0.52	0.59	1.5ms

3.6.3 `EmbeddingRetriever`

Также один из встроенных в используемую библиотеку методов ранжирования. Он извлекает релевантные документов из большой коллекции на основе векторного представления текста. Для его работы можно использовать предобученные модели такие как BERT, RoBERTa или любые другие, способные преобразовывать текст в векторные представления. Здесь важно не переборщить с количеством параметров модели, т.к. фаза отбора документов не должна слишком сильно затягиваться.

Мы рассмотрели несколько моделей, однако использование крупных трансформеров приводило к значительному времени обработки, поэтому мы отказались от них и искали модели Sentence Transform. На платформе Hugging Face мы обнаружили одну потенциально подходящую модель для решения данной задачи на русском языке - rubert-base-cased-sentence [33].

Top 1 accuracy	Top 3 accuracy	Top 5 accuracy	Mean resposne time
0.08	0.2	0.25	393ms

Такие результаты несколько удивили. Модель работает значительно дольше, чем обычный BM25. Мы полагаем, что результаты хуже из-за того, что УКРФ представляет собой довольно хорошо формализованный документ, в котором содержится относительно небольшое количество сложностей в интерпретации, и все статьи достаточно хорошо характеризуются определенным набором слов. Поэтому на данном датасете мы решили не использовать данную модель ретривера. Однако в будущем мы можем вернуться к ней для проведения дальнейших исследований.

3.6.4 `NavecRetriever`

Еще одна из версий ретривера, которую мы попытались запустить в первую очередь, при этом имея ограниченные знания в области глубинного обучения и не полностью освоившись в данной теме. Результаты предыдущего ретривера, основанного на эмбеддингах, не удовлетворили нас из-за его низкого качества и длительного времени работы. В ходе исследования мы обнаружили набор предварительно обученных векторных представлений для русского языка и пришли к выводу, что они могут быть более эффективными. Тем более процедура получения векторного представления предложения с помощью которой мы получали векторные представления документов была вычислительно легкой.

Наш алгоритм был довольно простым: мы проходили по каждому слову в документе, суммировали векторные представления слов и нормализовали полученный вектор, что служило векторным представлением документа. Аналогичным образом мы получали векторное представление для запроса. Затем, используя алгоритм, зависящий от базы данных документов, мы находили наиболее схожие документы с нашим запросом.

Top 1 accuracy	Top 3 accuracy	Top 5 accuracy	Mean resposne time
0.12	0.15	0.2	45ms

В итоге стало очевидно, что структура документов не настолько проста, чтобы просто складывать векторные представления слов и получать итоговое векторное представление документа. Хотя мы сократили время вычислений, по-прежнему уступаем классическому методу BM25 в качестве результатов.

3.6.5 Эвристики и этапы обработки

В итоговом решении, мы решили улучшить производительность ретривера BM25 и ридера с помощью применения различных эвристик и этапов обработки запросов и документов, основанных на библиотеке Natasha.

Мы опробовали следующее

1. Поиск только по названиям документов, с применением всех этапов обработки и эвристик.

2. Лемматизация документов и поисковых запросов с целью приведения всех слов к единой форме.

3. Добавление префиксов к словам. Мы обнаружили, что в некоторых случаях в документах преобладают определенные части речи, в то время как в запросах преобладают другие. Например, при наличии частого использования слова ”наркотический” в документах, а в поисковых запросах - ”наркотик”, ретривер BM25 проявляет недостаточную эффективность. Для решения этой проблемы мы решили добавить рядом с каждым словом префикс, состоящий из n букв. Значение параметра n было подобрано с помощью метода перебора.

4. Расширение документов и поисковых запросов синонимами, найденными на основе эмбеддингов Navec.

5. Определение наилучшего окна в каждом документе на основе мер соответствия из алгоритма BM25 для облегчения работы ридера.

Method	Top 1 accuracy	Top 3 accuracy	Top 5 accuracy	Mean time
BM25 titles	0.47	0.61	0.64	1ms
BM25 lemma	0.48	0.73	0.8	3ms
BM25 lemma title	0.66	0.81	0.84	3ms
BM25 lemma trunks	0.53	0.69	0.78	3ms
BM25 lemma title trunks	0.64	0.78	0.84	2ms
BM25 lemma synonyms	0.6	0.75	0.82	2ms
BM25 lemma title synonyms	0.67	0.8	0.86	2ms

3.6.6 Итоги

Удалось успешно реализовать полный цикл разработки модели ранжирования и подготовить основу для дальнейших исследований в области поиска релевантоного контекста в документе и переформулировки ответа в более естественную форму.

В рамках этого проекта мы:

1. Подготовили инструментов для тестирования, которые были необходимы для последующих этапов работы.

2. Собрали датасет с использованием этих тестовых инструментов, что позволило создать надлежащую основу для анализа и экспериментов.

3. Спроектировали архитектуры системы, которая обеспечивает эффективный поиск и ранжирование документов.

4. Развернули микросервиса для поиска, позволяющего проводить тестирование и оценку моделей.

5. Провели исследования и эксперименты, результатом которых стало улучшение качества базовых моделей и повышение общей производительности системы.

3.7 Reader

После того, как Retriever выбрал наиболее релевантные документы, они передаются Reader для дальнейшей обработки.

Reader отвечает за поиск ответов в найденных документах. Reader обрабатывает каждый документ, используя алгоритмы машинного обучения, чтобы найти наиболее релевантную информацию для поискового запроса.

Для того, чтобы Reader мог правильно извлекать ответы, он использует контекстуальное векторное представление, которое позволяет учитывать контекст и семантику при извлечении ответов. Контекстуальное векторное представление - это представление слов, учитывающее контекст, в котором они используются. Например, слово "банка"может иметь различные значения в разных контекстах, например, "банка консервов"или "банка для хранения денег". Контекстуальное векторное представление учитывает такие различия и позволяет модели правильно понимать смысл слов в контексте.

Для извлечения ответа на вопрос из текста Reader использует так называемые “span selection“ модели. Такие модели предсказывают начальную и конечную позицию в тексте, которая содержит ответ на вопрос пользователя. Например, для вопроса “Какой цвет у яблока?“ модель должна предсказать позиции в тексте, которые содержат ответ “красный“ или “зеленый“

Для обучения модели Reader используется набор данных, который содержит пары вопросов и ответов. Этот набор данных может быть создан путем аннотации текстовых документов вручную или автоматически с помощью алгоритмов извлечения ответов из документов.

После обучения модели Reader, ее можно использовать для извлечения ответов на новые вопросы из выбранных документов. Reader может быть интегрирован с другими инструментами и сервисами, чтобы создать полнофункциональный поисковый движок для поиска ответов на вопросы в текстовых документах. В частности, в качестве Reader можно взять модели такие как BERT, XLNet [45] и многие другие.

3.7.1 Трансформер

Трансформер (англ. **transformer**) — архитектура глубоких нейронных сетей, использующая механизм внутреннего внимания, и не полагается на рекуррентные нейронные сети. Эту архитектуру мы можем использовать в поиске. Устройство трансформера состоит из кодирующего и декодирующего элементов. На вход принимается последовательность токенов, создается ее

векторное представление (англ. **embedding**), добавляется вектор позиционного кодирования, после чего вектора без учета порядка в последовательности поступают в кодирующий компонент, а затем декодирующий компонент получает на вход часть этой последовательности и выход кодирующего. В результате получается новая выходная последовательность. Трансформер показывает хорошие результаты при работе с текстами, так как обладает способностью обрабатывать долгосрочные зависимости во входной последовательности, что достигается за счет механизма внутреннего внимания. Это позволяет модели фиксировать отношения между токенами, которые находятся далеко друг от друга во входной последовательности, что сложно для традиционных рекуррентных нейронных сетей.

3.7.2 BERT

Архитектура BERT [36] состоит из стека кодеров трансформера, которые обучаются двунаправленным образом. Это означает, что во время обучения модель учится предсказывать слово на основе как предшествующего, так и последующего контекста.

Во время исследования моделей мы тестировали модели архитектуры DEBERTA. Основные отличия DEBERT от BERT:

- механизм распутанного внимания, в котором каждое слово представлено с помощью двух векторов, которые кодируют его содержимое и позицию соответственно, а весовые коэффициенты внимания среди слов вычисляются с использованием распутанных матриц их содержания и относительных позиций соответственно.

- совершенствованный декодер маски используется для включения абсолютных позиций в слой декодирования для прогнозирования замаскированных токенов при предварительном обучении модели.

3.7.3 Выбор модели

Модели брали с Hugging Face, это сообщество с открытым исходным кодом, которые внесли значительный вклад в области обработки естественного языка и машинного обучения. Наиболее известен разработкой библиотеки Transformers, которая стала популярной основой для работы с моделями на основе трансформеров. Предоставляет широкий спектр предварительно обученных моделей, включая BERT, GPT, RoBERTa и многие другие. Эти модели доступны на разных

языках и могут быть точно настроены для конкретных задач, таких как классификация текста, ответы на вопросы и анализ тональности.

Тестирование каждой из модели было произведено с помощью внешних ассесоров. Этот метод включает в себя представление результатов работы модели человеку-оценщику (ассесора) и просьбу их предоставить отзыв о точности выданного ответа. Так как это отдельная значительная часть работы, расскажем про это подробнее.

У нас есть данные вида - вопрос и наиболее релевантные к нему документы, полученные при помощи Retriver, к этому моменту мы выбрали оптимальный алгоритм для поиска документов и пользовались им. Нам требуется оценить работу Reader - модели, которая в наиболее подходящих документах находит ответ на поставленный вопрос. Для этого мы привлекли людей, которые будут оценивать результат работы Redear'a при помощи меток 0 или 1, 1 - если ответ их устраивает и 0 в другом случае. У этого метода есть некоторые следующая проблема. Оценщики-люди могут иметь разные точки зрения или интерпретации одного и того же текста, что приводит к субъективным оценкам. Несмотря на это, идея данного подхода состоит в том, что при увеличении количества людей оценщиков общее представление о работе модели будет объективным. Для устранения наиболее субъективных ответов можно применить EM алгоритм [1].

Для повышения качества Reader мы собрали данные вида: вопрос - контекст - ответ. Таким образом, мы получили размеченные данные для нашей задачи. Это нужно для того, чтобы обучить модель на этих данных, и тогда она будет лучше понимать устройство текста в уголовном кодексе Российской Федерации, так как изначально все модели, используемые в качестве Reader обучались на общих данных на русском, а не целенаправленно под каждый набор документа. То есть таким образом, наша дообученная модель будет лучше понимать стиль и логику исходного документа. У этого подхода есть минус - на документах, не относящихся к статьям из уголовного кодекса, ожидается, что модель будет хуже работать, так как будет вносить предположение о том, что данные имеют логику и структуру построения текста такую же, как в уголовном кодексе. Чтобы такого не происходило можно использовать следующие техники:

- использовать исходную модель на новых документа
- собирать размеченные данные на новые документы и дообучить исходную модель на них
- при дообучении модели использовать подсказки, то есть в размеченную выборку по ста-

тьям из уголовного кодекса добавлять начало: "Ответь на вопрос по уголовному кодексу"

3.7.4 Сбор и обработка оценок

Для сбора оценок результаты работа Redeara'a и разметки данных был использован телеграмм бот. Пользователю предлагалось оценить выход модели по 50 вопросам по статьям из уголовного кодекса Российской Федерации. Мы использовали вопросы подготовленные для Retrivier. Так же телеграмм бот использовался для сбора данных для дообучения модели по статьям из уголовного кодекса Российской Федерации. Для получения размеченных данных пользователям выдавались вопросы, не пересекающиеся с 50 предыдущими вопросами, контекст и предлагалось ввести ответ.

Для работы с базой данных использовалась библиотека SQLite, которая обеспечивает реализацию системы управления реляционными базами данных. SQLite следует без серверной архитектуре, что означает отсутствие отдельного запущенного серверного процесса.

Описание работы телеграмм бота для сбора оценок:

- Пользователь выбирал режим оценивания выход модели
- По очереди отправлялось 50 вопросов
- Пользователь выбирал отметку 1, если ответ удовлетворял вопросу и 0 в другом случае
- данные сохранялись в базе данных в виде строк со следующими значениями: индентификатор пользователя, номер вопроса, оценка

Описание работы телеграмм бота для разметки данных:

- Пользователь выбирал режим разметки данных
- Каждому пользователю по очереди отправлялись свои 10 вопросов.
- Пользователь отвечал на поставленный вопрос отрывком текста из контекста
- данные сохранялись в базе данных в виде строк со следующими значениями: индентификатор пользователя, номер вопроса, начало ответа, конец ответа. Последние 2 значения получались при помощи обычного поиска подстроки в строке.

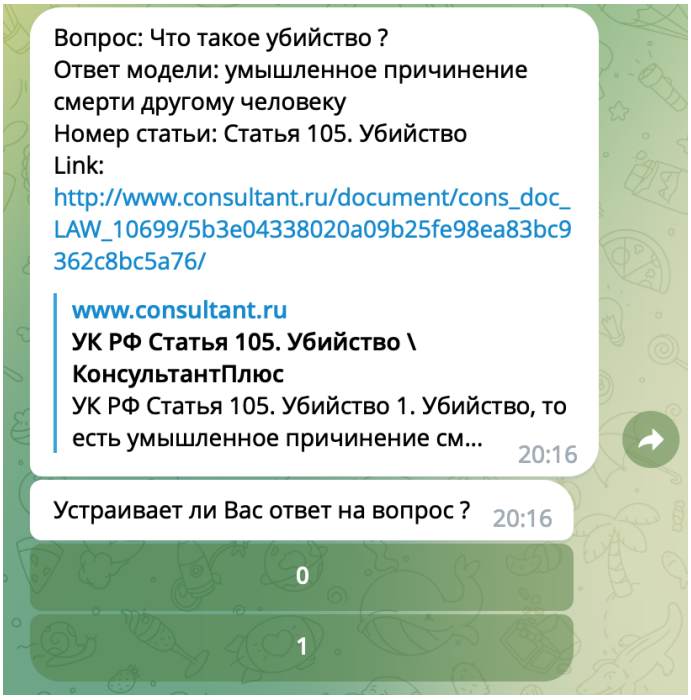

Рис. 2: Пример работы бота для сбора оценок

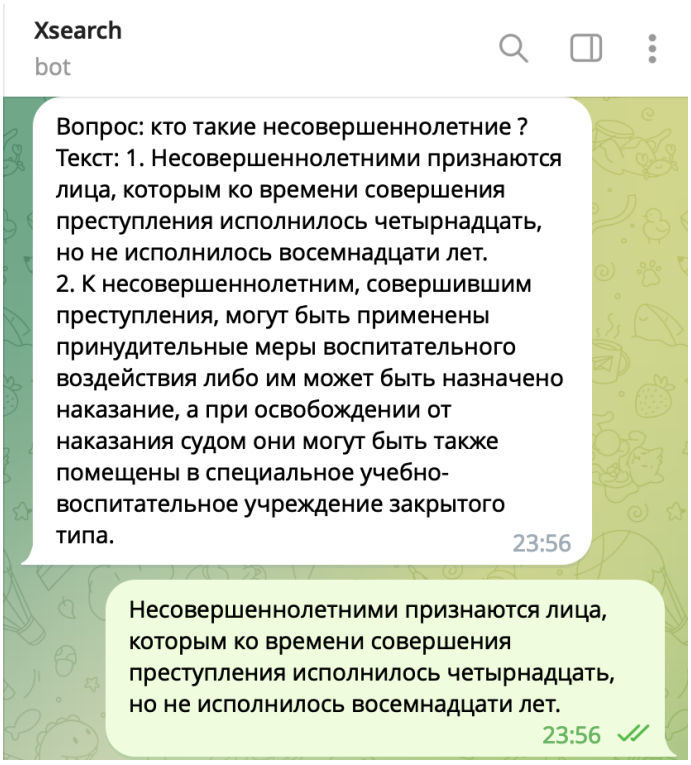

Рис. 3: Пример работы бота для разметки данных

3.7.5 Дообучение модели

Благодаря собранным размеченными данным удалось дообучить модель на них. График ошибки представлен здесь 4. По нашим субъективным оценкам модель стала лучше находить ответ в документе на поставленный вопрос.

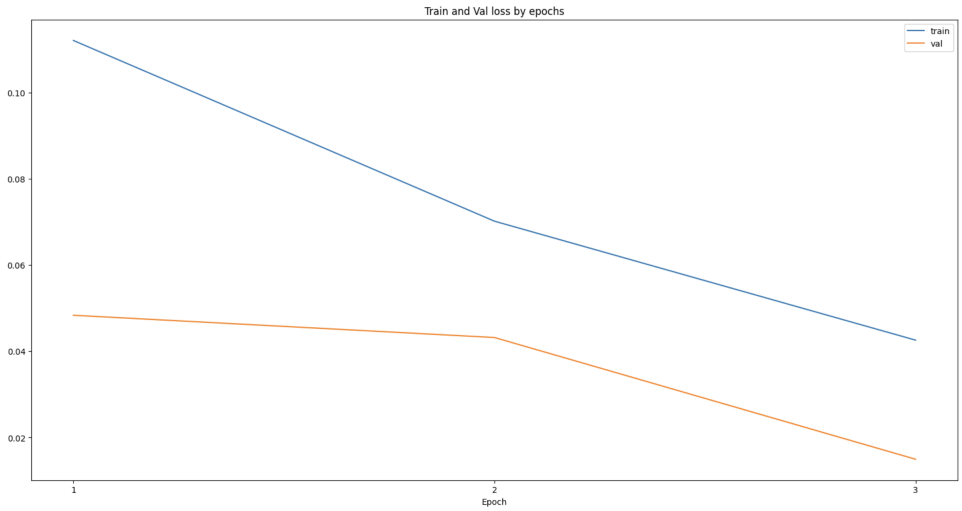

Рис. 4: График ошибки при дообучении модели на размеченных данных

3.7.6 Итог сбора и оценок работы

Сбор размеченных данных имеет решающее значение для оценки качества работы моделей. Размеченные данные относятся к набору данных, в котором каждый пример снабжен аннотацией правильного или ожидаемого результата, что позволяет модели учиться и делать точные прогнозы. Таким образом, получилось достичь следующих результатов:

- Реализация архитектуры микросервиса для сбора размеченных данных по документам
- Собраны размеченные данные по уголовному кодексу Российской Федерации
- Получены оценки внешних ассесеров на работу модели Reaeder
- При помощи оценонок внешних ассесеров была получена оценка на для модели Reader

3.8 Перефразирование

После того, как Reader выделил контекст, в котором наиболее вероятен ответ на поставленный вопрос пользователя. Мы берем контекст и просим модель для перефразирования ответить на

этот же вопрос по данному контексту. Например, если Reader на вопрос «Какого цвета майка зебры ?» нашел контекст «Вчерашним днем зебра Артура вышла погулять и решила надеть зеленую майку», то в модель для перефразирования подается следующий текст: «Ответь на вопрос по тексту: Какого цвета майка зебры ? Текст: Вчерашним днем зебра Артура вышла погулять и решила надеть зеленую майку» такой метод называется подсказка (англ. **prompt**). Подсказки играют решающую роль в языковых моделях, поскольку они обеспечивают отправную точку или контекст для создания текста. Подсказка — это конкретный ввод или инструкция, данные языковой модели для направления ее вывода. Это может быть несколько слов, предложение или даже более длинный отрывок текста. Языковые модели, обучены генерировать текст на основе шаблонов и информации, которую они извлекли из больших наборов данных. Однако у них нет врожденного понимания или знаний о желаемом результате. Предоставляя подсказку, мы можем направлять сгенерированый текст модели в определенном направлении или просить ее выполнить определенные задачи.

Кроме того, выбор подсказки может существенно повлиять на реакцию модели. Небольшие изменения в формулировке или формулировке подсказки могут привести к разным результатам, демонстрируя чувствительность языковых моделей к изменениям ввода.

Почему не остановится на ответе Reader ? Потому что в качестве Reader можно использовать только модели, которые используются для ответов на вопросы. Поэтому с помощью перефразирования мы расширяем область используемых моделей. Таким образом мы сильно повышаем качество поиска, но тратим на перефразирование лишнее время. В секции 3.8.2 предоставлены данные о скорости работы моделей.

3.8.1 Модели для перефразирования

Для данной задачи подходят все те же трансформеры, например T5 [32] - модель состоит из энкодеров и декодеров, обучение происходит на большем наборе данных с подсказками (префикс строки), что позволяет использовать модель в разных задачах. Также рассматривались модели, дообученные из исходной LLaMa [31].

3.8.2 Выбор модели

За последние полгода появилось много новых моделей, обученных на корпусе русских текстов, мы протестировали те из них, что подходили именно под нашу задачу. Так как времени было недостаточно для того, чтобы оценивать модель при помощи внешних ассесоров, то модели подбирались по субъективным нашим оценкам. Мы рассмотрели следующие модели:

1. `instruct_rugptlarge` [20]
2. `rugpt_medium_turbo_instructed` [25]
3. `rut5_large_turbo_instructed` [27]
4. `mt0_xxl_ru_turbo_alpaca_lora` [22]
5. `llama_7b_ru_turbo_alpaca_lora` [21]
6. `FRED-T5-large` [19]
7. `ruT5-base` [26]
8. `rugpt3medium_based_on_gpt2` [24]
9. `ruBert-large` [23]

Тестирование производили при помощи размеченных данных. То есть на вход моделям подавался текст с подсказкой, мы сравнивали глазами выход модели и правильный ответ.
Например: «Ответь на вопрос по тексту: Что такое судебный штраф? Текст: 1. Судебный штраф есть денежное взыскание, назначаемое судом при освобождении лица от уголовной ответственности в случаях, предусмотренных статьей 76.2 настоящего Кодекса. 2. В случае неуплаты судебного штрафа в установленный судом срок судебный штраф отменяется и лицо привлекается к уголовной ответственности по соответствующей статье Особенной части настоящего Кодекса.»
Ответ ассесера был: «Судебный штраф есть денежное взыскание, назначаемое судом при освобождении лица от уголовной ответственности».
Ответ модели: «Судебный штраф это денежное взыскание».
Для такой оценки моделей мы использовали 15 наборов вопрос - ответ.

name model	Mean resposne time on gpu M2	Mean resposne time on cpu
instruct_rugptlarge	14s	16s
rugpt_medium_turbo_instructed	7s	8s
rut5_large_turbo_instructed	9s	11s
mt0_xxl_ru_turbo_alpaca_lora	8s	9s
llama_7b_ru_turbo_alpaca_lora	11s	13s
FRED-T5-large	20s	25s
ruT5-base	19s	23s
rugpt3medium_based_on_gpt2	15s	18s
ruBert-large	14s	17s

Такое большое время перефразирования обусловено квадратичной сложностью алгоритма от длины подающегося текста.

В итоге была выбрана модель FRED-T5-large [19]. Со следующими гиперпараметрами: 5

```
gen_kwargs = {
    "min_length": 20,
    "top_k": 50,
    "top_p": 0.6,
    "do_sample": True,
    "max_new_tokens": 150,
    "early_stopping": True,
    "no_repeat_ngram_size": 2,
    "use_cache": True,
    "repetition_penalty": 1.5,
    "length_penalty": 1.2,
    "num_beams": 4,
    "num_return_sequences": 2,
    "renormalize_logits" : True
}
```

Рис. 5: Значения гиперпараметров модели

3.9 Итог выбора моделей

Во-первых, мы проанализировали характер документов и конкретные поисковые требования. Это помогло нам определить наиболее подходящие модели для обработки естественного языка и извлечения соответствующей информации. Мы рассмотрели модели основанные на трансформерах, такие как DEBERTA, GPT, LLaMa, T5, которые понимают семантику текстов.

Затем мы оценили вычислительные требования и масштабируемость моделей. Поскольку наш текстовый редактор предназначен для работы с большими объемами документов, нам нужны были модели, которые могли бы эффективно обрабатывать и выполнять поиск в больших объемах текста.

Кроме того, мы учитывали наличие и доступность предварительно обученных моделей и связанных с ними ресурсов. Использование предварительно обученных моделей позволило нам извлечь выгоду из существующих знаний и точно настроить их для нашего конкретного случая использования.

Наконец, мы провели тщательное тестирование и сравнительный анализ, чтобы оценить производительность различных моделей. Мы оценили такие критерии, как точность поиска, ответ модели, время работы, чтобы выбрать наиболее эффективные модели для нашей функции интеллектуального поиска. Мы неоднократно уточняли наш выбор на основе этих оценок, пока не достигли желаемого уровня производительности.

Таким образом, мы определили и интегрировали наиболее подходящие модели, обеспечив точный и эффективный поиск в нашем текстовом редакторе.

4 Распределение задач

- Беляев Артём
  - Создание микросервиса для сбора оценок внешних ассесеров для EM алгоритма и для разметки данных для дообучения
  - Подготовка архитектуры для модели Reader и модели перефразирования
  - Поиск подходящей модели для Reader и оценка его работы
  - Реализация взаимодействия Reader с другими компонентами поиска

  - Дообучение Reader на полученных данных
  - Поиск подходящей модели для перефразирования и оценки его работы
  - Реализация взаимодействия модели перефразирования с другими компонентами поиска

- Гимранов Артур
  - Проектирование и реализация архитектуры поиска
  - Подготовка инструментов для тестирования
  - Сбор и разметка тестирующей выборки для оценки модели Retriever
  - Поиск подходящей модели для Retriever
  - Реализация необходимых методов обработки данных для стадии ранжирования документов
  - Поиск подходящего алгоритма и модели для поиска документов
  - Написание взаимодействия основной базы документов с индексируемым хранилищем
  - Реализация взаимодействия между пользователем и моделью машинного обучения
  - Репрезентация ответа модели в вид, который ожидает фронтенд

- Прокопчук Леонид
  - Проектирование архитектуры сервисов
  - Контейнеризация сервисов
  - Написание хранилища статей для обработки запросов на изменение, создание и удаление статей
  - Реализация функционала для совместного редактирования статьи
  - Масштабирование и изоляция сервисов для удобного разворачивания в кластерах
  - Проектирование и разработка инфраструктуры для тестирования сервисов
  - Разработка аунтефикации для безопасного общения между сервисами
  - Развертывание и поддержка баз данных

5 Заключение

Мы создали редактор для высокой продуктивности как в больших компаниях, так и в маленьких командах, которая обеспечивается умным поиском с возможностью ответов на вопросы по всей базе документов и совместным редактированием, с которым можно вместе вести заметки на встречах, проводить мозговые штурмы, писать отчеты и многое другое.

Также мы позаботились над удобным интерфейсом с большим функционалом нашего редактора, чтобы у пользователей не было преград на пути к воплощению своих идей в текст. Мы создали удобную древовидную структуру, в которой можно сортировать свои документы по темам или командам. Добавили возможность создавать несколько рабочих пространств под разные задачи. В этих пространствах можно управлять, кто к чему имеет доступ, для обеспечения приватности данных.

Особенность нашего редактора – умный поиск. Мы выстроили пайплайн так, чтобы можно было легко менять все этапы поиска для проведения экспериментов и выбора лучшего решения. Таким образом мы создали сервис, который быстро ищет релевантные документы на запрос, а также генерирует простой и понятный ответ, чтобы пользователю не пришлось искать по всем ссылкам детали ответа не его вопрос.

В совместном редактировании мы исследовали разные алгоритмы. Отличие нашего случае – это разные типы форматированной, картинки и таблицы. Для таких случаев нужен особый подход к классическим алгоритмам совместного редактирования. Нам была важна устойчивость алгоритма и согласованность всех документов, чтобы в конечном счете у каждого пользователя был виден один и тот же документ, причем как можно быстрее.

Важной частью проекта была высокая доступность. Мы создали микросервисную архитектуру, которая может масштабироваться в зависимости от пользовательской нагрузки, а в случае отказа падать лишь частично, чтобы пользователь не заметил проблем.

Список литературы

[1] EM algorithm. URL: https://ru.wikipedia.org/wiki/EM.

[2] Alembic. May 2023. URL: https://github.com/sqlalchemy/alembic.

[3] Asgi. May 2023. URL: https://en.wikipedia.org/wiki/Asynchronous_Server_Gateway_Interface.

[4] asyncpg. May 2023. URL: https://gistpreview.github.io/?b8eac294ac85da177ff82f784ff2cb60.

[5] Cpu flame graphs. May 2023. URL: https://www.brendangregg.com/FlameGraphs/cpuflamegraphs.html.

[6] Elegant degradation. May 2023. URL: https://en.wikipedia.org/wiki/Elegant_degradation.

[7] Fastapi 0.89.1. January 2023. URL: https://github.com/tiangolo/fastapi.

[8] gettext. May 2023. URL: https://www.gnu.org/software/gettext/.

[9] Gunicorn. May 2023. URL: https://github.com/benoitc/gunicorn.

[10] Haystack 1.13.2. January 2023. URL: https://haystack.deepset.ai.

[11] Kubernetes. May 2023. URL: https://github.com/kubernetes/kubernetes.

[12] Minio. May 2023. URL: https://min.io/.

[13] Natasha 1.5.0. January 2023. URL: https://natasha.github.io.

[14] Postgresql. May 2023. URL: https://www.postgresql.org/.

[15] Redis. May 2023. URL: https://redis.io/.

[16] Sqlalchemy. May 2023. URL: https://github.com/sqlalchemy/sqlalchemy.

[17] Tinymce. May 2023. URL: https://github.com/tinymce/tinymce.

[18] Uvicorn. May 2023. URL: https://github.com/encode/uvicorn.

[19] Fred-t5-large. April, 2023. URL: https://huggingface.co/ai-forever/FRED-T5-large.

[20] instruct_rugptlarge. April, 2023. URL: https://huggingface.co/AlexWortega/instruct_rugptlarge.

[21] llama_7b_ru_turbo_alpaca_lora. April, 2023. URL: https://huggingface.co/IlyaGusev/llama_7b_ru_turbo_alpaca_lora.

[22] mt0_xxl_ru_turbo_alpaca_lora. April, 2023. URL: https://huggingface.co/IlyaGusev/mt0_xxl_ru_turbo_alpaca_lora.

[23] rubert-large. April, 2023. URL: https://huggingface.co/ai-forever/ruBert-large.

[24] rugpt3medium_based_on_gpt2. April, 2023. URL: https://huggingface.co/ai-forever/rugpt3medium_based_on_gpt2.

[25] rugpt_medium_turbo_instructed. April, 2023. URL: https://huggingface.co/IlyaGusev/rugpt_medium_turbo_instructed.

[26] rut5-base. April, 2023. URL: https://huggingface.co/ai-forever/ruT5-base.

[27] rut5_large_turbo_instructed. April, 2023. URL: https://huggingface.co/IlyaGusev/rut5_large_turbo_instructed.

[28] Elastic search. February 15, 2023. URL: https://en.wikipedia.org/wiki/Elasticsearch.

[29] Reader. February 15, 2023. URL: https://docs.haystack.deepset.ai/docs/reader.

[30] Retriever. February 15, 2023. URL: https://docs.haystack.deepset.ai/docs/retriever.

[31] Llama. February 27, 2023. URL: https://arxiv.org/abs/2302.13971.pdf.

[32] T5. Jule 28, 2020. URL: https://arxiv.org/pdf/1910.10683.pdf.

[33] rubert-base-cased-sentence. March, 2020. URL: https://huggingface.co/DeepPavlov/rubert-base-cased-sentence.

[34] Crud. May 10, 2023. URL: https://en.wikipedia.org/wiki/Create,_read,_update_and_delete.

[35] Alex Biryukov, Daniel Dinu, and Dmitry Khovratovich. Argon2. 2015. URL: https://www.password-hashing.net/argon2-specs.pdf.

[36] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert. 2018. URL: https://arxiv.org/abs/1810.04805.

[37] Obinna Ethelbert, Faraz Fatemi Moghaddam, Philipp Wieder, and Ramin Yahyapour. Jwt. 2017. URL: https://arxiv.org/abs/1710.08281.

[38] Vladimir Karpukhin, Barlas Oğuz, Sewon Min, Patrick Lewis, Ledell Wu, Sergey Edunov, Danqi Chen, and Wen tau Yih. Dense passage retrieval for open-domain question answering. 2020. URL: https://arxiv.org/abs/2004.04906.

[39] Martin Kleppmann. Crdts and the quest for distributed consistency. 2018. URL: https://www.youtube.com/watch?v=B5NULPSiOGw.

[40] Geoffrey Litt, Sarah Lim, Martin Kleppmann, and Peter van Hardenberg. Peritext: A CRDT for collaborative rich text editing. *Proceedings of the ACM on Human-Computer Interaction*, 6(CSCW2):1–36, November 2022. doi:10.1145/3555644.

[41] Hans Peter Luhn and Karen Spärck Jones. Tf-idf. 1957-1972. URL: https://en.wikipedia.org/wiki/Tf-idf.

[42] Shahar Mor. Scaling redis pubsub. March 2018. URL: https://www.youtube.com/watch?v=6G22a5Iooqk.

[43] R.L. Rivest, A. Shamir, and L. Adleman. Rsa. 1977. URL: https://people.csail.mit.edu/rivest/Rsapaper.pdf.

[44] Stephen Robertson and Hugo Zaragoza. Bm25. 1970-1980. URL: http://www.staff.city.ac.uk/~sbrp622/papers/foundations_bm25_review.pdf.

[45] Zhilin Yang, Zihang Dai, Yiming Yang, Jaime Carbonell, Ruslan Salakhutdinov, and Quoc V. Le. Xlnet. 2019. URL: https://arxiv.org/abs/1906.08237.

[46] Алексей Плетнёв. Свой распределённый s3 на базе minio. 2022. URL: https://www.youtube.com/watch?v=XiJVC9nzAW4.