Оглавление

Добавление собственных данных в LLM с помощью RAG

Идеи RAG могут быть реализованы разными способами, но на концептуальном уровне использование RAG в приложении, основанном на технологиях искусственного интеллекта, включает в себя следующие шаги:

1. Пользователь вводит вопрос.
2. Система ищет подходящие документы, которые могут содержать ответ на этот вопрос. Эти документы часто включают в себя собственные данные компании, которая разработала систему, а хранятся они обычно в некоем каталоге документов.
3. Система создаёт промпт для LLM, в котором скомбинированы данные, введённые пользователем, подходящие документы и инструкции для LLM. Модель должна ответить на вопрос пользователя, применив предоставленные ей документы.
4. Система отправляет промпт LLM.
5. LLM возвращает ответ на вопрос пользователя, основанный на предоставленных контекстных сведениях. Это — выходные данные системы.

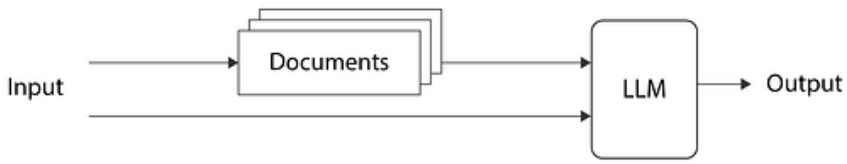


Рисунок 1. Диаграмма, отражающая общую идею, лежащую в основе RAG

Термин «RAG» был предложен в 2021 году, в публикации «Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks», подготовленной командой FAIR и сотрудничавшими с ней учёными. Идеи, предложенные авторами этой работы, оказали огромное воздействие на ИИ-решения, которыми мы пользуемся сегодня.

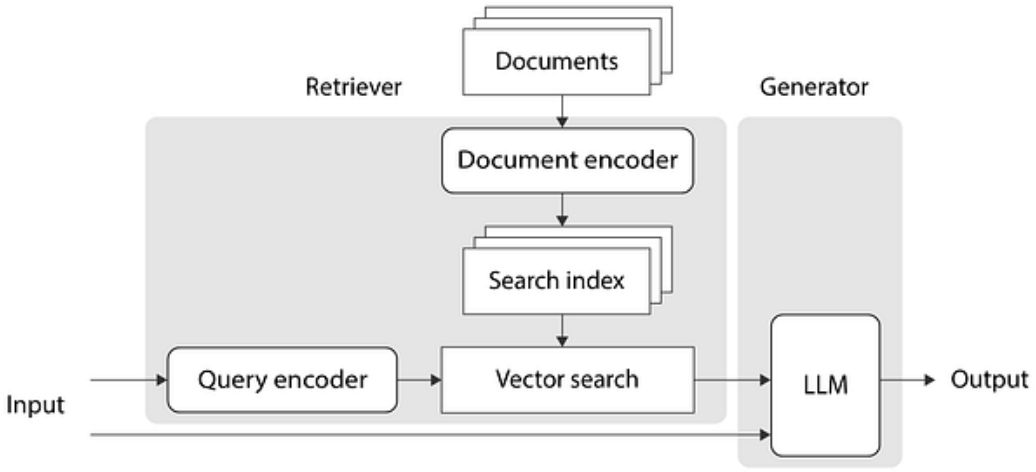


Рисунок 2. Архитектура ИИ-системы, использующей RAG

Если смотреть в общем, то предложенная в работе структура системы составлена из двух компонентов: из поискового модуля и из модуля генератора. Поисковый компонент преобразует входной текст в последовательность чисел с плавающей запятой (вектор), используя кодировщик запроса. Далее, он, используя единый подход, трансформирует каждый из документов, применяя кодировщик документов, после чего сохраняет закодированные документы в виде поискового индекса. Затем, в поисковом индексе, поисковый компонент выполняет поиск векторов документов, которые имеют отношение к входному вектору. После этого он преобразует векторы документов обратно в их текстовое представление и возвращает эти тексты в качестве результата своей работы. Теперь в дело вступает генератор, который принимает текст, введённый пользователем, и соответствующие ему документы, комбинирует всё это в единый промпт и предлагает LLM дать ответ на вопрос пользователя с учётом информации, имеющейся в найденных ранее документах. То, что выдаст после этого LLM, и будет выходными данными всей этой системы.

Кодировщик запроса, кодировщик документов и LLM на рисунке 2 показаны с использованием похожих фигур. Это из-за того, что все они реализованы с использованием трансформеров. Традиционные трансформеры состоят из двух частей: из кодировщика и декодера.

Кодировщик отвечает за трансформацию входного текста в вектор (или в последовательность векторов), которая, в общих чертах, отражает смысл слов. А цель декодера — сгенерировать новый текст, основываясь на входных данных. В архитектуре системы, предложенной в вышеупомянутой публикации, кодировщик запроса и кодировщик документов реализованы с помощью трансформеров, в состав которых входит лишь кодировщик. Дело в том, что этим компонентам системы нужно лишь преобразовывать фрагменты текста в векторы, состоящие из чисел. А LLM в модуле генератора реализована на базе традиционного трансформера, содержащего и кодировщик, и декодер.

В публикации предлагается использовать предварительно обученные трансформеры и проводить совместное дообучение только кодировщика запросов и LLM из блока генератора. Это дообучение выполняется с использованием пар фрагментов данных, один из которых — это то, что ввёл пользователь, а второй — это то, что ожидается получить на выходе LLM. Кодировщик документов не дообучают, так как это может быть достаточно затратным, и из-за того, что авторы работы обнаружили, что в этом нет необходимости для обеспечения хороших результатов работы системы.

В публикации предлагается два подхода для реализации этой архитектуры:

- RAG-последовательность: получают k документов и используют их для генерации всех выходных токенов, которые служат ответом на запрос пользователя.
- RAG-токен: получают k документов, используют их для генерации следующего токена, затем получают ещё k документов, которые используют для генерации следующего токена, и так далее. Это значит, что всё может свестись к нахождению нескольких разных наборов

документов при генерировании единственного ответа на запрос пользователя.

Подобный шаблон проектирования достаточно широко распространён в ИИ-индустрии.

Использование RAG на практике

На практике реализации RAG, которые широко используются в различных организациях, основаны на том, что было предложено в вышеупомянутой публикации, но при этом для них характерны некоторые особенности:

- Из двух подходов к реализации архитектуры RAG, предложенных в публикации, почти всегда выбирают RAG-последовательность. Дело в том, что этот подход оказывается дешевле и проще, чем другой. При этом он даёт отличные результаты.
- На практике обычно не занимаются дообучением каких-либо трансформеров, входящих в состав системы. Те предварительно обученные LLM, которыми мы можем пользоваться в наши дни, достаточно хороши для того чтобы пользоваться ими в их исходном виде. Их, кроме того, слишком затратно дообучать самостоятельно.

Стоит добавить, что применяемые методы поиска документов не всегда реализованы именно так, как предложено в публикации. Поиск обычно выполняется с применением некоего поискового сервиса — наподобие FAISS или Azure Cognitive Search. Такие сервисы поддерживают различные способы поиска документов, которые хорошо сочетаются с RAG. В работе поисковых сервис обычно можно выделить два следующих шага:

- Поиск информации. На этом шаге производится сопоставление запроса пользователя с документами, находящимися в поисковом индексе. После этого осуществляется извлечение наиболее подходящих

документов. Существуют три самых распространённых подхода к поиску документов: поиск по ключевым словам, векторный поиск, гибридный поиск.

- Ранжирование информации. Это — необязательный шаг, который следует за шагом поиска информации. При его выполнении берётся список документов, которые были сочтены подходящими при поиске информации, после чего, по более точным алгоритмам, чем ранее, определяется их соответствие запросу пользователя.

Поиск по ключевым словам

Самый простой способ поиска документов, имеющих отношение к запросу пользователя — это так называемый «поиск по ключевым словам» (ещё известный как «полнотекстовый поиск»). Поиск по ключевым словам использует те ключевые слова, которые ввёл пользователь, для поиска по индексу документов, содержащих эти слова. Сравнение содержимого документов с запросом выполняется исключительно на основе текста, без применения векторов. Этот подход к поиску существует уже давно, но он не потерял актуальности и в наши дни. Он весьма полезен в тех случаях, когда ищут идентификаторы пользователей, коды продуктов, адреса и любые другие данные, при поиске которых важно точное совпадение с текстом запроса. Вот — общая схема системы, в которой используется поиск по ключевым словам.

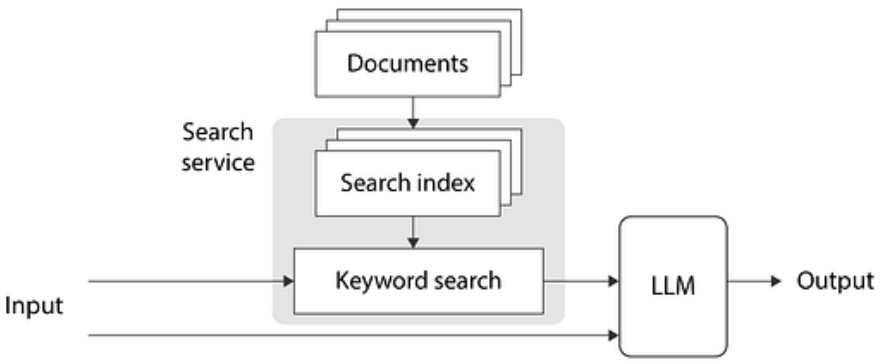


Рисунок 3. Система, в которой используется поиск по ключевым словам

При таком подходе поисковый сервис поддерживает работу инвертированного индекса, хранящего сведения о соответствии слов и документов, в которых используются эти слова. Текстовые данные, введённые пользователем, разбирают, извлекая из них ключевые слова. Эти слова анализируют, находя их базовые формы. Затем выполняют поиск по инвертированному индексу, каждому совпадению назначают оценку, а после этого наиболее релевантные документы, соответствующие запросу, возвращают из поискового сервиса.

При таком подходе поисковый сервис поддерживает работу инвертированного индекса, хранящего сведения о соответствии слов и документов, в которых используются эти слова. Текстовые данные, введённые пользователем, разбирают, извлекая из них ключевые слова. Эти слова анализируют, находя их базовые формы. Затем выполняют поиск по инвертированному индексу, каждому совпадению назначают оценку, а после этого наиболее релевантные документы, соответствующие запросу, возвращают из поискового сервиса.

Векторный поиск

«Векторный поиск», который ещё называют «плотным поиском информации» («dense retrieval»), отличается от поиска по ключевым словам. Один из аспектов этого отличия заключается в том, что при векторном поиске возможно нахождение соответствия поисковому запросу в документах, в которых нет ключевых слов из запроса, но общий смысл которых близок к смыслу запроса. Например, представьте, что создаёте чат-бота, который планируется использовать в службе поддержки сайта для сдачи недвижимости в аренду. Пользователь задаёт боту вопрос: «Do you have recommendations for a spacious apartment close to the sea», интересуясь,

может ли он порекомендовать просторное жильё, расположенное близко к морю. В документе, содержащем сведения о подходящем жилище, имеется следующий текст: «4000 sq ft home with ocean view» — тут описан дом площадью 4000 квадратных футов с видом на океан. При поиске по ключевым словам такой документ найден не будет. А вот система векторного поиска его найдёт. Векторный поиск лучше всего показывает себя в тех случаях, когда в неструктурированном тексте ищут некие общие идеи, а не точные ключевые слова.

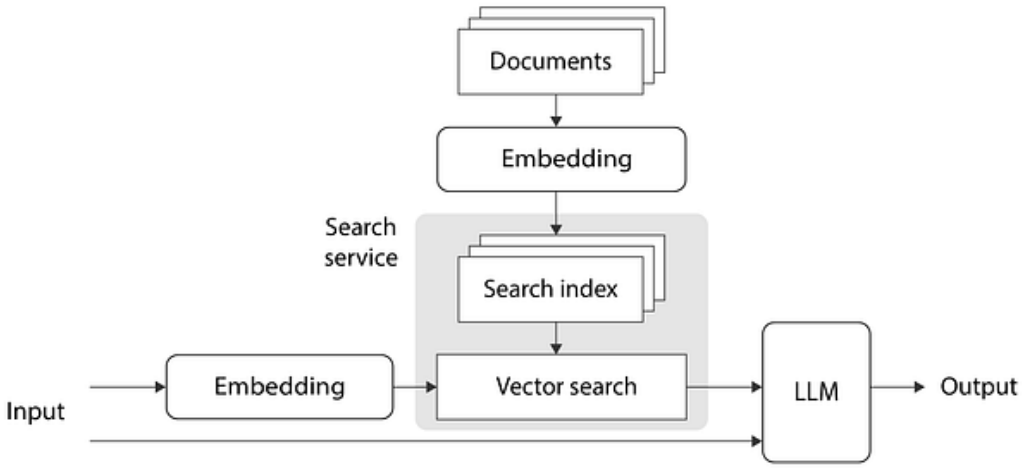


Рисунок 4. Система, в которой используется векторный поиск

Взглянув на схему, изображенную на рисунке 4, можно понять, что векторный поиск, используемый на практике, это — практически то же самое, что было предложено в публикации по RAG. За исключением того, что в данном случае не проводится дообучение трансформеров.

В подобных системах для кодирования запроса и документов обычно используют предварительно обученную модель эмбеддингов, такую, как text-embedding-ada-002 от OpenAI. А для формирования итогового результата применяют предварительно обученные LLM, например — gpt-35-turbo (ChatGPT), тоже созданную OpenAI. Модель эмбеддингов используется для преобразования входного текста и текста документов в соответствующие «эмбеддинги». Что такое «эмбеддинг»? Это — вектор чисел с плавающей запятой. Он, в общих чертах, «схватывает» смысл

текста, который в нём закодирован. Если два фрагмента текста как-то связаны, тогда можно предположить, что и соответствующие им эмбеддинги будут похожи друг на друга.

Как определить то, что векторы похожи? Для ответа на этот вопрос рассмотрим пример. Мы исходим из предположения о том, что для нахождения следующих векторов эмбеддингов используется наша собственная модель эмбеддингов:

- a = (0, 1) представляет «Do you have recommendations for a spacious apartment close to the sea?» — «Можете ли вы порекомендовать просторное жильё, расположенное близко к морю?».
- b = (0.12, 0.99) представляет «4000 sq ft home with ocean view» — «дом площадью 4000 квадратных футов с видом на океан»
- c = (0.96, 0.26) представляет «I want a donut» — «Я хочу пончик».

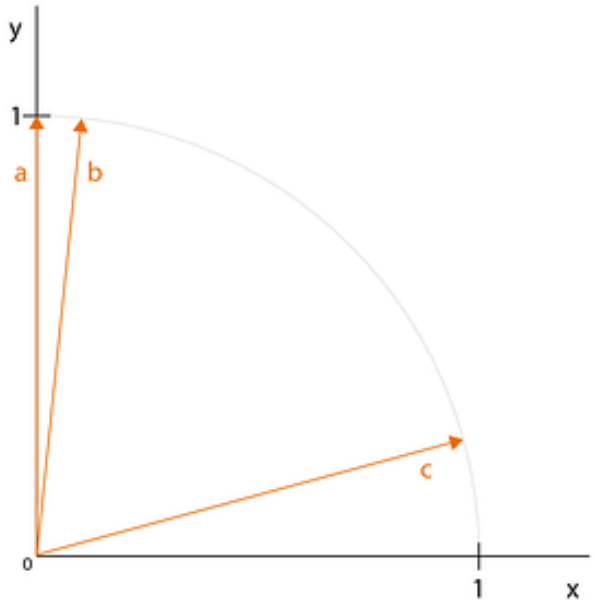


Рисунок 5. Графическое представление векторов

Глядя на рисунок 5, можно догадаться о том, что вектор, который сильнее всего похож на вектор a — это b (а не c). Если прибегнуть к математическим методам выявления схожести векторов, то можно сказать, что существуют три распространённых подхода для решения этой задачи: скалярное произведение векторов, косинусный коэффициент векторов и

евклидово расстояние между векторами. Найдём показатели, характеризующие схожесть векторов, используя эти методы, и посмотрим, подтвердят ли они нашу догадку.

При применении метода скалярного произведения векторов, как можно ожидать из его названия, просто вычисляют скалярное произведение (его ещё называют внутренним произведением) векторов. Чем больше результат — тем ближе друг к другу векторы.

$$\vec{u} = (u_1, u_2)$$
$$\vec{v} = (v_1, v_2)$$
$$\vec{u} \cdot \vec{v} = u_1 * v_1 + u_2 * v_2$$
$$\vec{a} \cdot \vec{b} = (0, 1) \cdot (0.12, 0.99) = 0 * 0.12 + 1 * 0.99 = 0.99$$
$$\vec{a} \cdot \vec{c} = (0, 1) \cdot (0.96, 0.26) = 0 * 0.96 + 1 * 0.26 = 0.26$$

Скалярное произведение векторов a и b больше, чем скалярное произведение a и c. Это подтверждает нашу догадку относительно того, что векторы a и b больше похожи друг на друга, чем a и c. Применяя этот метод, надо помнить о том, что скалярное произведение очень сильно зависит от длины векторов. Оно применимо только для сравнения схожести векторов, имеющих одинаковую длину. Эмбеддинги OpenAI всегда представлены единичными векторами — векторами, имеющими длину, равную единице. Чтобы наши эмбеддинги согласовывались бы с теми, что используются в OpenAI, они тоже должны быть единичными.

Косинусный коэффициент векторов вычисляют, находя косинус угла между двумя векторами. Чем этот коэффициент больше — тем сильнее векторы похожи друг на друга. Формула для вычисления этого коэффициента предусматривает вычисление скалярного произведения векторов, которое делят на произведение длин векторов. Это дополнение предыдущей формулы, нормализующее результаты, позволяет эффективно измерять схожесть векторов, учитывая их направления и не обращая внимания на их длины.

$$\vec{u} = (u_1, u_2)$$
$$\vec{v} = (v_1, v_2)$$
$$S_c(\vec{u}, \vec{v}) = \frac{\vec{u} \cdot \vec{v}}{\|\vec{u}\|\|\vec{v}\|} = \frac{u_1 * v_1 + u_2 * v_2}{\sqrt{u_1^2 + u_2^2} * \sqrt{v_1^2 + v_2^2}}$$
$$S_c(\vec{a}, \vec{b}) = (0 * 0.12 + 1 * 0.99)/1 = 0.99$$
$$S_c(\vec{a}, \vec{c}) = (0 * 0.96 + 1 * 0.26)/1 = 0.26$$

Так как мы пользуемся единичными векторами, их скалярное произведение и косинусный коэффициент дают в точности одинаковые результаты.

Вычисляя евклидово расстояние между векторами, находят расстояние между ними — в том смысле, в котором понятие «расстояние» используется в обычной жизни. Чем меньше расстояние между двумя векторами — тем сильнее они друг на друга похожи. В отличие от косинусного коэффициента, при вычислении евклидового расстояния во внимание принимаются длина и направление векторов.

$$\vec{u} = (u_1, u_2)$$
$$\vec{v} = (v_1, v_2)$$
$$d(\vec{u}, \vec{v}) = \sqrt{(u_1 - v_1)^2 + (u_2 - v_2)^2}$$
$$d(\vec{a}, \vec{b}) = \sqrt{(0 - 0.12)^2 + (1 - 0.99)^2} = 0.12$$
$$d(\vec{a}, \vec{c}) = \sqrt{(0 - 0.96)^2 + (1 - 0.26)^2} = 1.21$$

Видно, что расстояние между векторами a и b меньше, чем между векторами a и c. Это говорит нам о том, что векторы a и b больше похожи друг на друга, чем a и c.

В примере используются двумерные векторы, что позволяет нам, просто глядя на их изображения, строить догадки об их схожести. Но в случае с эмбеддингами часто бывает так, что речь идёт о векторах гораздо большей размерности. Например, модель text-embedding-ada-002, созданная OpenAI, генерирует векторы, размерность которых равна 1536. Не забывайте о том, что количество измерений, используемых для

представления эмбеддингов, не зависит от длины входного текста. Поэтому и короткий запрос, и длинный документ будут преобразованы в векторы эмбеддингов одинаковой размерности.

Большинство поисковых сервисов, существующих в наши дни, поддерживают все три рассмотренные нами метода определения схожести векторов. Они позволяют пользователю выбрать метод, который ему нужен. Какой из этих методов лучше выбрать?

- Если ваши эмбеддинги имеют разную длину, и вы хотите учитывать их длину при определении их схожести, тогда лучше всего будет выбрать евклидово расстояние между векторами, так как при его вычислении учитывается и длина векторов, и их направление.
- Если все ваши эмбеддинги нормализованы, приведены к единичной длине, тогда все три рассмотренных нами подхода дадут, как вы уже видели, схожие результаты. Но при этом скалярное произведение векторов вычисляется быстрее. Предположим, вы работаете с системой (с приложение или сервисом), которая знает о том, что имеет дело с единичными векторами. В такой ситуации вычисление косинусного коэффициента векторов, скорее всего, будет оптимизировано и сведено к вычислению их скалярного произведения. Поэтому, скорее всего, в подобном случае вычислительная нагрузка на систему при использовании косинусного коэффициента векторов будет столь же скромной, как и при использовании скалярного произведения.

Для того чтобы найти векторы документов, лучше всего соответствующие вектору запроса, поисковый сервис может пойти самым простым и неэффективным, в плане траты вычислительных ресурсов, путём. А именно, он определит схожесть входного вектора с вектором каждого из документов, отсортирует список документов в соответствии с их оценкой, и вернёт верхнюю часть этого списка. Этот простой подход,

правда, не масштабируется. Он не подходит для больших корпоративных приложений, в которых используется очень много документов. В результате поисковые сервисы обычно используют некую разновидность алгоритма ANN (Approximate Nearest Neighbor, приближённый поиск ближайшего соседа). В этом алгоритме применяются продуманные оптимизации, которые позволяют ему возвращать приблизительные результаты быстрее, чем при использовании других методов. Одной из популярных реализаций ANN является алгоритм HNSW (Hierarchical Navigable Small World, иерархический маленький мир).

Гибридный поиск подразумевает применение и поиска по ключевым словам, и векторного поиска. Например, представим себе ситуацию, когда имеется идентификатор клиента и текстовый запрос. Нужно провести поиск, учитывающий и точные данные в виде идентификатора, и общий смысл текста, введённого пользователем. Это — отличный сценарий для применения гибридного поиска. Два типа поиска выполняются самостоятельно, после чего полученные данные комбинируются с помощью специального алгоритма, который выбирает лучшее из того, что дали разные подходы к поиску. Этот метод часто используется на практике, особенно — в достаточно сложных приложениях.

Семантическое ранжирование

Семантическое ранжирование (или переранжирование) — это необязательный шаг работы RAG-системы, который следует за шагом нахождения документов. На шаге нахождения документов система делает всё возможное для ранжирования возвращённых документов на основании их релевантности запросу пользователя. А семантическое ранжирование часто способно улучшить результаты первоначальной оценки документов. При его выполнении берётся подмножество документов, возвращённое после их поиска, после чего, с помощью LLM, обученной специально для

выполнения этой задачи, вычисляются коэффициенты релевантности более высокого качества, чем ранее. Эти коэффициенты применяются для переранжирования документов.

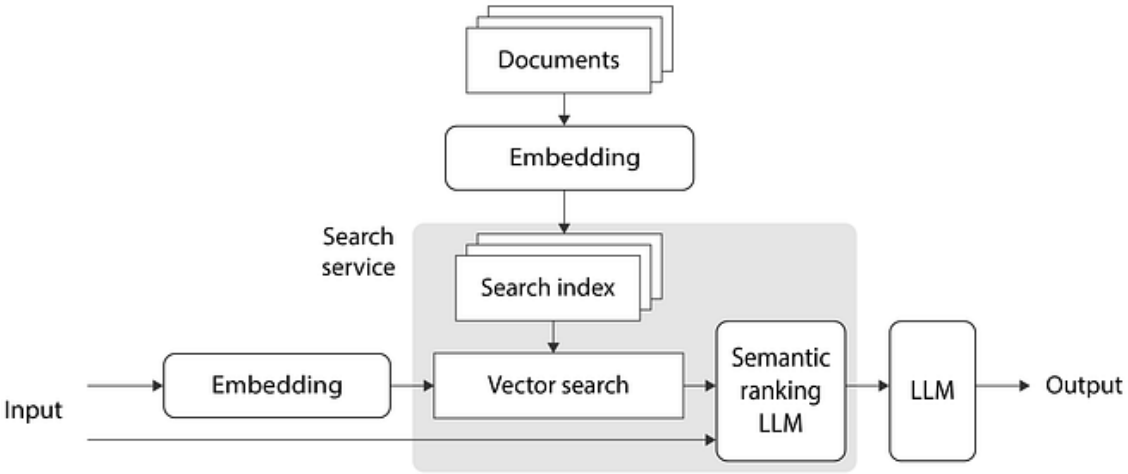


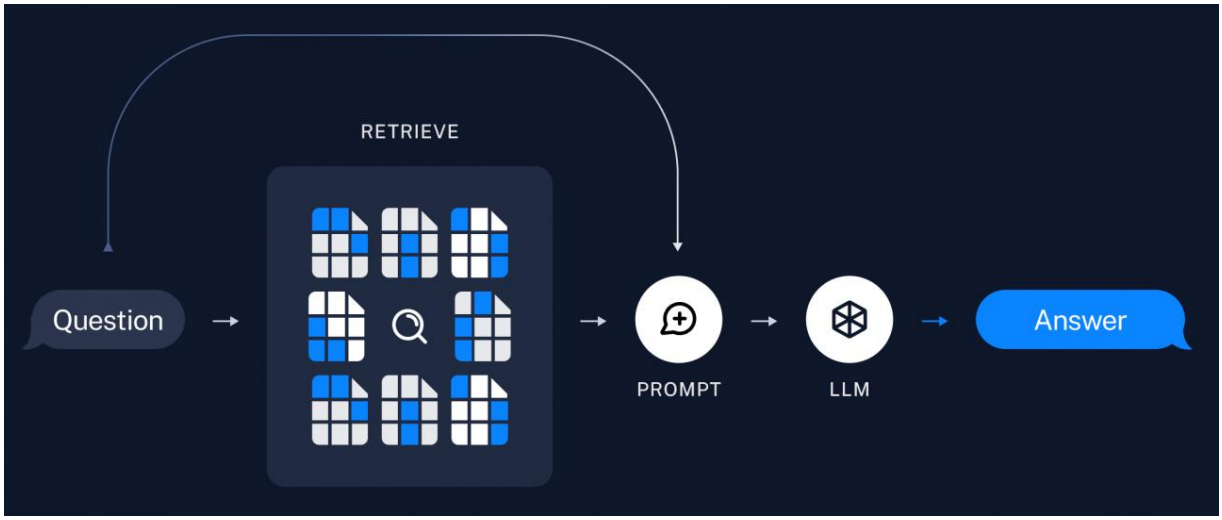
Рисунок 6. RAG-система, в которой используется модуль семантического ранжирования документов

На рисунке 6 показан модуль семантического ранжирования, скомбинированный с модулем векторного поиска, но этот модуль хорошо сочетается и с подсистемой поиска по ключевым словам. Я решил показать здесь совместную работу семантического ранжирования и векторного поиска из-за того, что именно так сделано в примере реализации RAG-системы, подготовленном для этой статьи. Рассмотрим этот пример.

Постановка задачи тестирования

Использование больших языковых моделей (LLM) для обработки текстовых документов является классическим подходом, основанным на промптах, который называется RAG (Retrieval-Augmented Generation). До наступления эры LLM модели часто дополняли новыми данными, просто проводя их дообучение. Но теперь, когда используемые модели стали гораздо масштабнее, когда обучать их стали на гораздо больших объёмах данных, дообучение моделей подходит лишь для совсем немногих сценариев их использования. Дообучение особенно хорошо подходит для

тех случаев, когда нужно сделать так, чтобы модель взаимодействовала бы с пользователем, используя стиль и тональность высказываний, отличающиеся от изначальных. Один из отличных примеров успешного применения дообучения — это когда компания OpenAI доработала свои старые модели GPT-3.5, превратив их в модели GPT-3.5-turbo (ChatGPT). Первая группа моделей была нацелена на завершение предложений, а вторая — на общение с пользователем в чате. Если модели, завершающей предложения, передавали промпт наподобие «Можешь рассказать мне о палатках для холодной погоды», она могла выдать ответ, расширяющий этот промпт: «и о любом другом походном снаряжении для холодной погоды?». А модель, ориентированная на общение в чате, отреагировала бы на подобный промпт чем-то вроде такого ответа: «Конечно! Они придуманы так, чтобы выдерживать низкие температуры, сильный ветер и снег благодаря…». В данном случае цель компании OpenAI была не в том, чтобы расширить информацию, доступную модели, а в том, чтобы изменить способ её общения с пользователями. Но дообучение уже не так хорошо себя показывает при необходимости добавления новых данных в модель. Это, как мне удалось выяснить, гораздо более распространённый бизнес-сценарий. Кроме того, дообучение LLM требует больших объёмов высококачественных данных, серьёзных затрат на вычислительные ресурсы и много времени. Для большинства пользователей LLM всё это относится к разряду ограниченных ресурсов.



Основной принцип заключается в разбиении большого документа, который нельзя поместить в промпт, на мелкие части или чанки. Затем, когда задается вопрос, система ищет наиболее релевантные куски текста, где может находиться ответ. Из них формируется финальный промпт.

Реализация этого механизма включает несколько этапов:

1) Разбить текст на чанки.
2) Перевести эти чанки в векторный вид для последующего выбора наиболее релевантных с помощью косинусной близости.
3) Сформировать промпт из этих небольших частей.
4) Отправить промпт в выбранную модель и получить ответ.

Таким образом, для создания миварных баз знаний с помощью большой языковой модели нужно выполнить следующие действия:

1) Подобрать алгоритм разбития текста на чанки (куски).
2) Подобрать токензайзер для получения эмбедингов по этим кускам.
3) Подобрать промпт.
4) Подобрать модель.

Классическое описание данного механизма, описанное в Langchain, предполагает:

- Делить текст по переносам строк, стараясь сохранить слова целыми, с ограничением по количеству символов или токенов.

- Для токенизации предлагается использовать модель ada-002 от OpenAI.
- Использование OpenAI для генерации ответа.

Это подход отлично работает, если не стоит задача использовать локальные решения и нет каких-либо ограничений на использование исходных данных, таких как коммерческая тайна. Возьмем модель на базе GPT4-Turbo (10К контекст) с их родным токенайзером ada-002 в качестве референса.

Исходя из поставленной задачи, нужно последовательно перебрать различные модели и токенайзеры, чтобы выбрать наиболее близкие к референсу. В итоге для сравнения были отобраны:

1) Один алгоритм чанкования.
2) Четыре вида токенайзеров: ada-002, RuBert, YandexEmbeding, RuBert Finetuned.
3) Один промпт.
4) 7 видов моделей: GPT4, GPT3.5, YandexGPT, GigaChat, Saiga, FineTuned Saiga.

Обработка текста

В классическом подходе предлагается разбивать текст на абзацы или, где это невозможно, по словам с некоторым перекрытием. Этот метод хорош, когда каждый абзац содержит законченную мысль, абзацы схожи по размеру и структуре. Однако, в случае с юридическими документами или медицинскими инструкциями, некоторые абзацы могут быть подпунктами более крупных разделов. Например, пункт 3.3.3 описывает случаи, когда запрещено использовать температурную перевозку, с подпунктами 3.3.3.1 и 3.3.3.2, детализирующими отдельные случаи. Важно не разделять такие связанные пункты.

Также в результате тестов было выявлено, что чем длиннее чанк, тем хуже качество токенизации – она становится более обобщенной и происходят потери параметров при финальной генерации миварной базы знаний. Поэтому было принято решение, что оптимальный чанк – это не сугубо техническое решение, прежде всего должен сохранятся контекст. В итоге был подобран размер чанков до 150 слов, сгруппированные в соответствии с иерархией документа или иного исходноготекста. Для этого был разработан скрипт, который анализирует структуру документа, создавая чанки, максимально крупные в рамках ограничений.

Токенизатор (получение эмбедингов)

Каждый чанк необходимо преобразовать в вектор, который затем записывался в PSQL. Были рассмотрели различные варианты: облачные сервисы OpenApi (ada-002) и YandexEmbeding (с ограничением до 10 запросов в секунду), а также множество локальных вариантов, включая Word2Vec, RuBert, rugpt и даже слои эмбеддингов у Llama. Облачные сервисы работали отлично, хотя у Яндекса была низкая квота по умолчанию.

С локальными вариантами процесс отбора подходящих моделей сложнее. Пришлось использовать подобие автотеста. Было выбрано порядка 10 вопросов и соответствующие части текста, где точно есть ответ на них. Далее меняя разные токенайзеры отбирали те, какие покажут хорошую косинусную близость между вопросом и куском исходного текста.

RuBert-Large оказался наиболее адекватным для нашей задачи, с использованием среднего пулинга (mean pooling). Скрытый слой Llama оказался полностью бесполезным. RuGPT, Fred и прочие от ai-forever - оказались хуже.

Промт

От промпта многое зависит, но для эксперимента было решено оставить классический вариант: Ты … . Твоя задача … используя части текста из инструкции. Ответь на вопрос: [вопрос] Используя сведения из следующих кусков текста: [текстовые куски]. Отвечай кратко. Ничего не придумывай. Если ответ не знаешь, то так и скажи.

Размер промпта был ограничен до 10000 символов, так как этот размер хорошо подходил под все рассматриваемые модели, несмотря на разное количество занимаемых токенов для русских символов в разных моделях (например, 2-3 символа на каждый из 4096 токенов у Llama или 1 токен = 1 символ для GPT4).

Модели

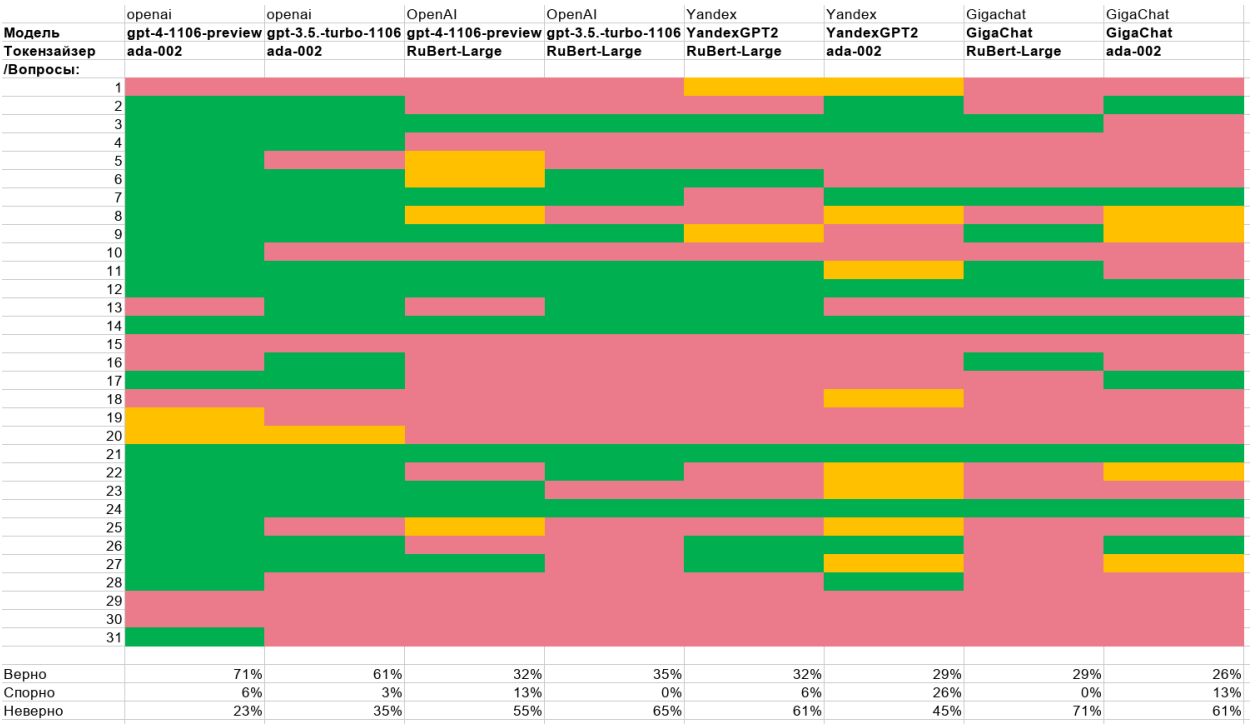
В исследовании были использованы несколько источников: доступ к OpenAI, YandexGPT, GigaChat, а также мощную видеокарту 4090 для запуска и обучения локальных моделей. В качестве базовых моделей были выбраны Llama и Saiga.

Для работы с облачными моделями температура генерации ответов бала установлена на ноль, чтобы повысить точность и уменьшить креативность ответов. Доступы к моделям были получены через API.

Результаты тестирования

Для тестирования использовался 31 запрос, так или иначе связанный с обработкой текста, используемого для создания баз знаний, на который система отвечала, используя разные комбинации моделей и токенайзеров. Каждый ответ затем был оценен вручную по следующей градации: верно (зелёный), спорно (желтый), неверно (красный).

По горизонтали расположены вопросы, по вертикали – комбинации параметров. В ячейках таблицы указан цвет, соответствующий оценке ответа.



	openai	openai	OpenAI	OpenAI	Yandex	Yandex	Gigachat	GigaChat
Модель	**gpt-4-1106-preview**	**gpt-3.5.-turbo-1106**	**gpt-4-1106-preview**	**gpt-3.5.-turbo-1106**	**YandexGPT2**	**YandexGPT2**	**GigaChat**	**GigaChat**
Токензайзер	**ada-002**	**ada-002**	**RuBert-Large**	**RuBert-Large**	**RuBert-Large**	**ada-002**	**RuBert-Large**	**ada-002**
/Вопросы:								
Верно	71%	61%	32%	35%	32%	29%	29%	26%
Спорно	6%	3%	13%	0%	6%	26%	0%	13%
Неверно	23%	35%	55%	65%	61%	45%	71%	61%

Из результатов тестирования видно, что GPT4-Turbo с использованием родного токенайзера ada-02 показал лучшие результаты: 71% ответов оценены как верные и только 23% как неверные. Наименее эффективной оказалась модель GigaChat с локальным токенайзером RuBert-Large. Результаты YandexGPT были чуть лучше, но разница несущественна. Важно отметить, что модели OpenAI также показали плохие результаты на токенайзере RuBert. Это подтверждает предположение, что неправильный выбор частей текста (ошибка токенайзера) для формирования промпта приводит к неверным ответам. Почти во всех случаях, когда модели OpenAI давали неверные ответы, другие модели также показывали низкие результаты.

На основе первых данных стало ясно, что стоит сосредоточиться на разработке локализованных под конкретную задачу токенайзеров и

использовать ada-002 в качестве эталонного токенайзера. Это подчеркивает важность правильного выбора инструментов для обработки и подготовки текста перед использованием языковых моделей для задачи создания миварных баз знаний.

Выбор токенайзера

Правильный выбор токенайзера в RAG очень важен. Чтобы избежать перебора всех возможных вариантов и не нагружать проверкой человека-эксперта, решено было провести предварительную квалификацию. В RAG промпт задачи формируется динамически. Вопрос (A) конвертируется в вектор, и выбираются наиболее близкие ему математические векторы фрагментов регламента, сортируемые по убыванию "близости". Затем из них формируется запрос. Например, для вопроса A наиболее близкими являются фрагменты текста P, D, F, G, при этом дистанция между векторами A-P больше, чем A-D. С этой задачей лучше всего справляется ADA-02 от OpenAI.

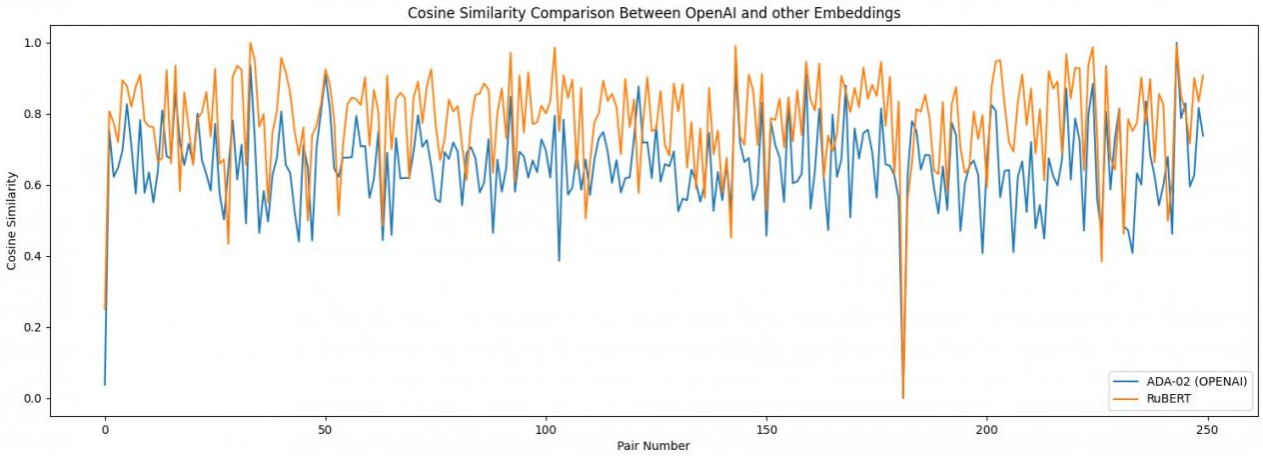
Таким образом, можно создать собственный токенайзер и сравнить, какие фрагменты текста он выдаст для такого же запроса. Если их последовательность совпадает с ADA-02, это отлично. В противном случае, эту ошибку можно измерить и анализировать.

Методология поиска лучшего токенайзера

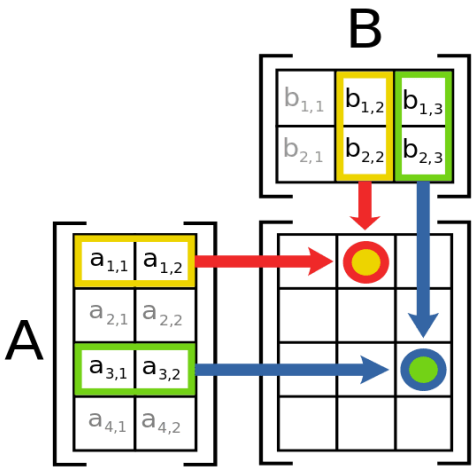
В качестве референса было использовано около 5000 текстовых фрагментов различного содержания: примеры вопросов, выдержки из регламента. Из этих данных случайным образом формировались 250 пар текстов для расчета косинусной близости для выбранного токенизайзера. Далее рассчитывал отклонение между значением эталонного ADA-02 и выбранного варианта. Колебание этого отклонения и есть наш параметр.

Если токензайзеры идентичны, то и среднеквадратичное отклонение между ними = 0.

Среднее квадратичное отклонение между каждой точкой позволило определить точность выбранного токенизатора. RuBert – Large демонстрировал погрешность около 10-11% в отношении ADA-02, sbert-nlu-ru-mt – 9-11%, что является улучшением, хоть и незначительным.

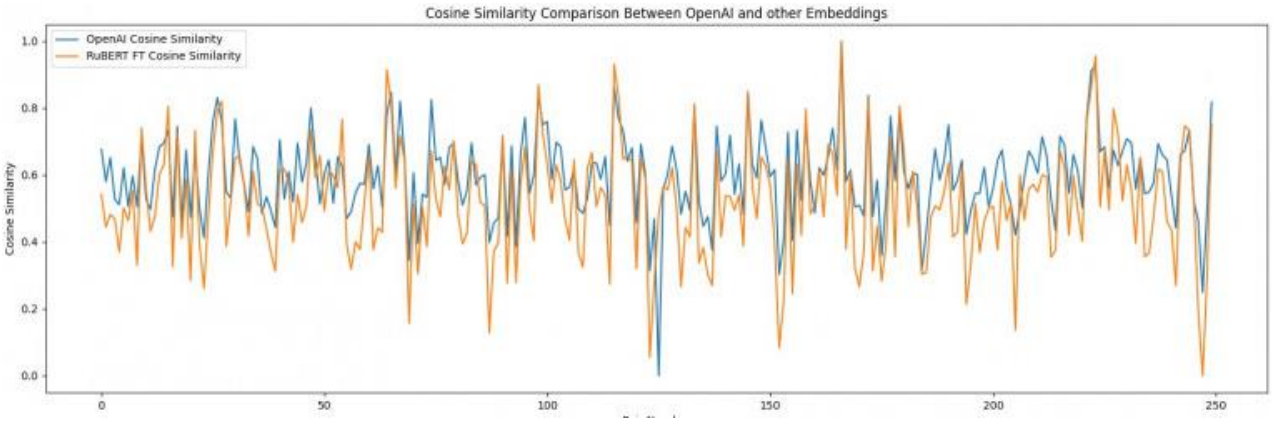


9Ембеддинг – это матрица чисел, поэтому задача адаптации локального токенизатора, такого как RuBERT или sbert-nlu-ru-mt, заключается в преобразовании одной матрицы в другую. Классическая ML задача.



После исследований промпта в ChatGPT было принято решение остановиться на трехслойной сети TensorFlow, без специальной настройки и подбора гиперпараметров (всё по умолчанию, что наверное неправильно).

30 эпох обучения и был создан адаптер. Он переводит вектор RuBert (1024 параметра) в вектор OpenAI (1526 параметров). Этот адаптер будет эффективно работать только с текстами, близкими к определенной тематике.



Синтетические тесты показали, что ошибка снизилась с 9-11% до 6%. При этом использование sbert-nlu-ru-mt не дало прироста, поэтому был выбран RuBert-large в качестве основы. Таким образом, был получен локально работающий токенизатор, близкий по эффективности к облачному ADA-02, что теоретически должно улучшить качество ответов.

Проверка на практике, как работает FineTuned токенайзер.

Возьмем наш эталон в виде GPT-4 и пропустим через нашу машинку вопросов/ответов и отдадим на оценку ответов экспертам. И мы получаем следующие результаты:

GPT-4	**ADA-02**	**RuBert - Large**	**RuBert-Large FineTuned**
Верно	71%	32%	55%
Спорно	6%	13%	19%
Ошибочно	23%	55%	26%

Значения правильных ответов на эталонной модели значительно выросли (с 32% до 55%). Ошибочных ответов уменьшилось в двое и равно

эталонным (уменьшение с 55% до 26%, при 23% эталонных). Иными словами – наша стратегия сработала:

1. Правильный токенайзер повышает качество. Улучшение точности ембедингов с 11% до 6% даёт значимый результат.
2. Fine Tune токенайзера возможен.

Аналогичный результат мы видим и для моделей YandexGPT. Количество правильных ответов растет, ошибочных падает:

YandexGPT	**ADA-02**	**RuBert - Large**	**RuBert-Large FineTuned**
Верно	29%	32%	45%
Спорно	26%	6%	13%
Ошибочно	45%	61%	42%

Локальные модели

Развернем локальные модели на базе Saiga2 (LLama2). Решение не тестировать на токенизаторе от OpenAI и обычном RuBert было обоснованным, так как использование локальной модели с облачным токенизатором от OpenAI кажется абсурдным, а применение обычного RuBert нецелесообразно из-за его низкого качества.

Для сравнения возьмем модели с двумя разными квантованиями. На 5 и 8. Первая может поместиться в видеокарту 3060 (версия 12GB), вторая уже требует карты 3090 или 4090 (версия 24GB).

Обе модели показали почти идентичные: 35% верных ответов, 52% ошибочных и 13% спорных ответов. Это немного лучше, чем у YandexGPT и GigaChat на базе обычного RuBert, но хуже, чем у RuBert-fine tuned. В целом, YandexGPT немного превосходит Saiga LLama2, что является положительным моментом.

Итоговая сравнительная таблица:

Верных ответов

	gpt-4-1106-preview	gpt-3.5.-turbo-1106	YandexGPT2	GigaChat	Saiga/LLama2-13BQ8_0	Saiga/LLama2-13BQ5_K
Ada-02	71%	61%	29%	26%		
RuBert-Large	32%	35%	32%	29%		
RuBert-FineTuned	55%		45%		35%	32%

Спорные ответы

	gpt-4-1106-preview	gpt-3.5.-turbo-1106	YandexGPT2	GigaChat	Saiga/LLama2-13BQ8_0	Saiga/LLama2-13BQ5_K
Ada-02	6%	3%	26%	13%		
RuBert-Large	13%	0%	6%	0%		
RuBert-FineTuned	19%	n/a	13%		13%	16%

Ошибочные

	gpt-4-1106-preview	gpt-3.5.-turbo-1106	YandexGPT2	GigaChat	Saiga/LLama2-13BQ8_0	Saiga/LLama2-13BQ5_K
Ada-02	23%	35%	45%	61%		
RuBert-Large	55%	65%	61%	71%		
RuBert-FineTuned	26%	n/a	42%		52%	52%

С fine-tune моделями пока что не всё получается как надо. Много галлюцинаций. Поэтому нужно проводить дополнительные исследования по этой теме.

Вывод

На данный момент задача RAG с использованием локальных моделей (в закрытом контуре, без доступа к интернету и интерфейсу других моделей) не решается в полной мере в лоб. Одновременно эту задачу можно решать, если есть доступ к OpenAI GPT4 с широким окном контекста, достигая 70-80% правильных ответов без особых усилий. Обратная сторона медали, кроме отсутствия контура безопасности – это стоимость. Сейчас запрос будет обходиться где-то в 5-10 центов.

Основные ограничения локальных моделей связаны в первую очередь с правильным формированием частей первичного текста токенайзерами и в меньшей степени – с самими моделями. Недостатки моделей можно компенсировать более точными промптами и fine-tuning. Альтернативно, можно подождать развития технологий.

Работа над нарезкой и векторизацией данных – это задача, требующая внимания и которая не будет решена сама по себе без дополнительных

исследований. Доработка FineTune токенайзера представляется технически выполнимой. Нужно больше экспериментировать с гиперпараметрами модели.

По вопросу нарезки данных (частей текста) возможны три пути. Первый, немного примитивный, но правильный с точки зрения продуктового подхода – ограничить RAG задачу информацией, которая умещается в 1-2 абзаца, возможно с ручной нарезкой. Второй – изучить возможности и технологии графового представления данных для более сложной обработки. Третий – ждать, пока появятся достойные локальные модели с длиной контекста (эффективной длинной) до 128К символов или обучать их самостоятельно.

Оценка качества ответов производилась вручную экспертами, что придало исследованию практическую значимость. Одним из ключевых выводов является то, что значительную роль в доле качественных ответов играла модель, создававшая первичные эмбеддинги документа. Хороший эмбединг придавал больше роста в качестве, чем хорошая LLM.

Список литературы

1. Vanderplaats G. N.. Very large scale optimization. National Aeronautics and Space Administration (NASA), Langley Research Center, 2002.
2. Bo Jiang, Ning Wang, Cooperative bare-bone particle swarm optimization for data clustering, Soft Comput. 18 (6) (2014). Pp. 1079–1091.
3. Lin Lin, Mitsuo Gen, Yan Liang, A hybrid EA for high-dimensional subspace clustering problem, in: 2014 IEEE Congress on Evolutionary Computation (CEC), IEEE, 2014, pp. 2855–2860.
4. Bäck T. Evolutionary Algorithms in Theory and Practice: Evolution Strategies, Evolutionary Programming, Genetic Algorithms. Oxford University Press, Dec. 1995.

5. Storn R., Price K. V. “Differential evolution: A simple and efficient adaptive scheme for global optimization over continuous spaces,” ICSI, USA, Tech. Rep. TR-95-012, 1995.
6. Zhenyu Yang, Ke Tang, Xin Yao. Self-adaptive differential evolution with neighborhood search, in: IEEE Congress on Evolutionary Computation, 2008. CEC 2008 (IEEE World Congress on Computational Intelligence), IEEE, 2008. Pp. 1110–1116.
7. Mitchell A. Potter, Kenneth A. De Jong, A cooperative coevolutionary approach to function optimization, in: Parallel Problem Solving from NaturePPSN III, Springer, 1994. Pp. 249–257.
8. Li X., Tang K., Omidvar M. N., Yang Z., Qin K., Benchmark Functions for the CEC’2013 Special Session and Competition on Large-Scale Global Optimization, Tech. Rep., 2013.
9. Yang, Q., Chen, W. N., Da Deng, J., Li, Y., Gu, T., & Zhang, J. A Level-based Learning Swarm Optimizer for Large Scale Optimization. IEEE Transactions on Evolutionary Computation, 2017.