Глава 7. Ключевая роль векторов и векторных хранилищ в RAG

Это продолжение перевода книги Кит Борн. Раскрытие потенциала данных с помощью генеративного ИИ и технологии RAG. Векторы являются ключевым компонентом генерации, дополненной поиском (RAG). В этой главе мы вернемся к коду из предыдущих глав, уделяя внимание тому, как векторы влияют на него. Мы поговорим о том, что такое вектор, как они создаются и где их хранить. Мы сосредоточимся на том, как векторы связаны с RAG, но мы призываем вас потратить больше времени на более глубокое понимания векторов. Чем лучше вы понимаете векторы, тем эффективнее будут ваши конвейеры RAG. Обсуждение векторов настолько важно, что мы посвятим ему две главы. В этой главе мы рассмотрим вектора и векторные хранилища, а в главе 8 – векторный поиск, то есть то, как векторы используются в системе RAG. В этой главе мы рассмотрим следующие темы:

- Основы векторов
- Где в вашем коде скрываются векторы
- Объем текста, который вы векторизуете, имеет значение!
- Не все семантики одинаково хороши!
- Лаборатория кода 7.1 – Распространенные методы векторизации
- Факторы выбора варианта векторизации
- Начало работы с векторными хранилищами
- Векторные хранилища
- Выбор векторного хранилища

Предыдущая глава Содержание Следующая глава

Код для этой главы размещен в репозитории GitHub.

Эта глава посвящена только одной строке кода:

vectorstore = Chroma.from_documents(documents=splits, embedding=OpenAIEmbeddings())

А глава 8 будет сосредоточена на другой строке кода:

retriever = vectorstore.as_retriever()

Только эти две строки кода для двух глав? Да! Это показывает, насколько важны векторы для системы RAG. И чтобы понять векторы, мы начинаем с основ.

Основы векторов в RAG

В этом разделе мы рассмотрим несколько важных тем, связанных с векторами и эмбеддингами в контексте обработки естественного языка (NLP) и RAG. Мы начнем с уточнения взаимосвязи между векторами и эмбеддингами, поясняя, что эмбеддинги являются определенным типом векторного представления, используемым в NLP. Затем мы обсудим свойства векторов, такие как их размерность и масштаб, а также как эти характеристики влияют на точность и эффективность текстового поиска и сравнения по схожести.

В чем разница между эмбеддингами и векторами?

Векторы и эмбеддинги являются ключевыми концепциями в области обработки естественного языка (NLP). Но что они собой представляют и как они взаимосвязаны? Эмбеддинги можно считать специфическим типом векторного представления. Когда речь идет о LLM и RAG, термины *эмбеддинги*, *векторы*, *векторные эмбеддинги* и *векторные представления* используются как синонимы!

Но что такое вектор в более широком контексте?

Что такое вектор?

О чем вы думаете в первую очередь, когда слышите слово «вектор»? Многие люди сказали бы, что о математике. И это верно! Векторы — это математическое представление текста, и они позволяют применять математические операции к данным . Слово *вектор* также может навести вас на мысль о скорости. Это тоже верно! С помощью векторов мы можем выполнять текстовый поиск со значительно более высокой скоростью, чем с любой другой технологией, которая была ранее. Еще одно понятие, которое ассоциируется со словом вектор – это точность. Преобразуя текст в эмбеддинги, мы можем повысить точность поисковых систем. И наконец, вектор — это не просто набор чисел. Это математический объект, представляющий как величину, так и направление. Вот

почему он лучше справляется с представлением текста и сходств между текстами, поскольку он фиксирует более сложную форму текста, чем просто цифры.

Размерность и длина вектора

Персонаж Вектор из мультфильма "Гадкий я" говорил, что вектор — это «величина, представленная стрелкой». Такое представление, где вектор изображается стрелкой на двумерной или трехмерной плоскости, действительно помогает лучше понять его сущность. Однако в реальных задачах векторы часто имеют гораздо больше измерений, чем два или три. Это количество измерений называют размерностью вектора или его размером. Чтобы увидеть это в действии, добавим новую ячейку в код, после определения переменных

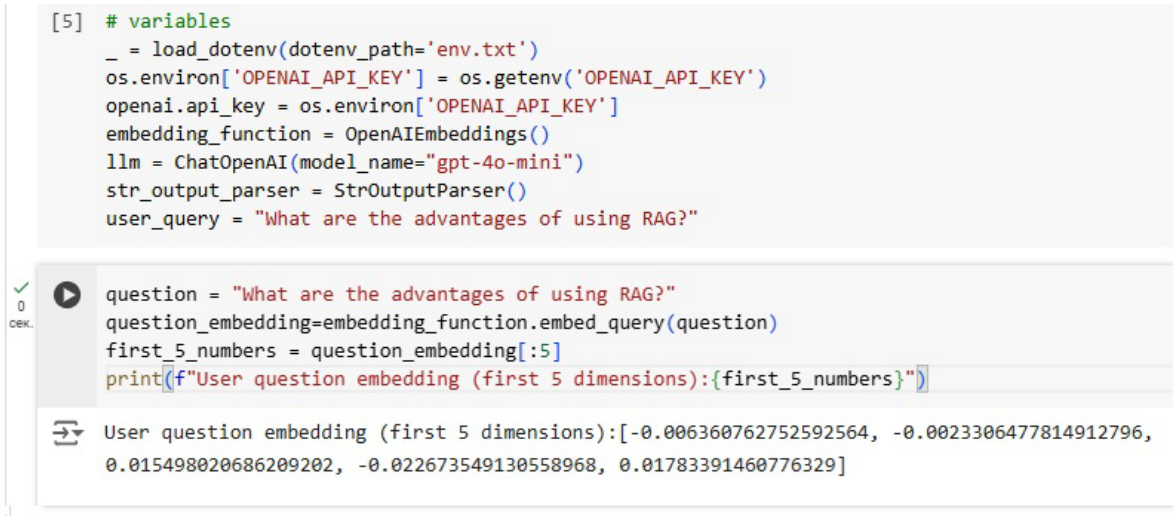

Рис. 7.1. Место в коде для вставки новой ячейки

```
question = "What are the advantages of using RAG?"
question_embedding=embedding_function.embed_query(question)
first_5_numbers = question_embedding[:5]
print(f"User question embedding (first 5 dimensions):{first_5_numbers}")
```

В этом коде мы используем вопрос, который встречался во всех примерах, — *«Какие преимущества использования RAG?»* — и применяем API эмбеддинг OpenAI для преобразования его в векторное представление. Переменная question_embedding представляет собой этот эмбеддинг. С помощью среза [0:5] мы извлекаем первые пять чисел из question_embedding, которые соответствуют первым пяти измерениям вектора, и выводим их на экран. Полный вектор состоит из 1,536 чисел с плавающей точкой, каждое из которых содержит 17–20 знаков. Чтобы упростить чтение, мы минимизируем объем выводимой информации. Результат выполнения этой ячейки представлен на рис. 7.1.

Мы выводим только первые пять измерений, но эмбеддинг значительно больше. Вскоре мы поговорим о практическом способе определить общее количество измерений, но сначала обратим внимание на длину каждого числа.

Все числа в этих эмбеддингах имеют формат +/-0 с десятичной точкой, так что давайте обсудим количество цифр после этой точки. Первое число, -0.006319054113595048, содержит 18 знаков после точки, второе — 19, а четвертое — 17. Это количество знаков связано с точностью представления чисел с плавающей запятой, используемой моделью эмбеддингов OpenAI, **OpenAIEmbeddings**. Эта модель использует формат представления чисел с высокой точностью на основе 64-разрядных чисел (также известных как числа с двойной точностью). Такая высокая точность позволяет различать очень мелкие детали и точно представлять семантическую информацию, которую фиксирует модель эмбеддингов.

Вывод эмбеддингов напоминает список чисел с плавающей точкой в Python. На самом деле, в данном случае это действительно список Python, так как OpenAI возвращает именно его из своего API для работы с эмбеддингами. Вероятно, это сделано для обеспечения большей совместимости с экосистемой Python. Однако, чтобы избежать путаницы, важно понимать, что в мире машинного обучения подобные данные, предназначенные для обработки, обычно представляются массивами

NumPy. Несмотря на то, что список чисел и массив NumPy выглядят одинаково в распечатанном виде, как в этом случае, их функциональность и использование различаются.

Здесь мы выводим только первые пять измерений, но вложение намного больше. О практическом способе определения общего количества измерений мы поговорим в один миг, но сначала я хочу обратить ваше внимание на длину каждого числа.

Все числа в этих вложениях будут иметь значение +/-0 с десятичной запятой, поэтому давайте поговорим о количестве цифр, которые идут после этой запятой. Первое число здесь, -0,006319054113595048, имеет 18 цифр после запятой, второе число имеет 19, а четвертое число имеет 17. Эти длины цифр связаны с точностью представления с плавающей запятой, используемого моделью встраивания OpenAI, OpenAIEmbeddings. Эта модель использует так называемый высокоточный формат с плавающей запятой, обеспечивающий 64-битные числа (также известные как двойная точность). Такая высокая точность дает возможность очень тонкого различения и точного представления семантической информации, полученной с помощью модели встраивания.

Кроме того, давайте вернемся к вопросу, высказанному в главе 1, о том, что предыдущий вывод очень похож на список с плавающей запятой на Python. В данном случае это на самом деле список Python, так как именно его OpenAI возвращает из своего API встраивания. Вероятно, это решение сделать его более совместимым с миром программирования на Python. Но чтобы избежать путаницы, важно понимать, что, как правило, в мире машинного обучения, когда вы видите что-то подобное в использовании, которое будет использоваться для обработки, связанной с машинным обучением, это, как правило, массив NumPy, даже если список чисел и массив NumPy выглядят одинаково при печати в качестве вывода, как мы только что сделали.

Интересный факт. Если вы работаете с генеративным ИИ, рано или поздно вы столкнетесь с понятием **квантования** (quantization). Подобно эмбеддингам, квантование связано с числами с плавающей точкой высокой точности. Однако суть квантования заключается в преобразовании параметров модели, таких как веса и активации, из их исходного высокоточного формата с плавающей точкой в формат с меньшей точностью.

Это позволяет уменьшить объем памяти, необходимый для модели, и снизить вычислительные затраты на работу с LLM. Применение квантования делает предобучение, обучение и дообучение языковой модели (LLM) более экономичным. Кроме того, квантование помогает снизить затраты на процесс **инференса** — использование модели для получения ответов. Когда я говорю об экономичности в этом контексте, я имею в виду возможность выполнения операций на более простом и недорогом оборудовании.

Однако квантование имеет свои недостатки. Это метод сжатия с потерями, что означает, что часть информации теряется в процессе преобразования. Уменьшенная точность квантованных моделей LLM может привести к снижению их точности по сравнению с исходными моделями высокой точности.

Когда вы используете RAG и рассматриваете различные алгоритмы для преобразования текста в эмбеддинги, обратите внимание на длину значений эмбеддингов, чтобы убедиться, что вы используете формат чисел с плавающей точкой высокой точности, если точность и качество ответов имеют высокий приоритет в вашей системе RAG.

Но сколько измерений представляют эти эмбеддинги? В предыдущем примере мы показали только пять, но могли бы вывести их все и сосчитать вручную. Конечно, это выглядит непрактично. Вместо этого мы воспользуемся функцией len(), чтобы произвести подсчет. В приведенном ниже коде вы увидите, как эта полезная функция позволяет узнать количество измерений в эмбеддинге:

```
embedding_size = len(question_embedding)
print(f"Embedding size: {embedding_size}")
```

Вывод этого кода выглядит следующим образом:

```
Embedding size: 1536
```

Данный эмбеддинг имеет 1536 измерений! Представить это в уме сложно, так как мы обычно мыслим максимум в трех измерениях, но дополнительные 1533 измерения существенно влияют на

точность семантических представлений текста, связанных с эмбеддингом. В большинстве современных алгоритмов векторизации векторы часто имеют сотни или даже тысячи измерений. Количество измерений соответствует количеству чисел с плавающей точкой, представляющих эмбеддинг. Это означает, что вектор размером 1 024 представлен 1 024 числами с плавающей точкой.

Хотя нет жестких ограничений на длину эмбеддинга, многие современные алгоритмы векторизации имеют предустановленные размеры. Например, модель эмбеддингов OpenAI **ada**, которую мы используем, по умолчанию создает векторы размером 1 536. Это связано с тем, что она обучена генерировать эмбеддинги определенного размера, и если попытаться сократить этот размер, это изменит контекст, отраженный в эмбеддинге.

Однако ситуация меняется. Появляются новые векторизаторы (например, модель OpenAI **text-embedding-3-large**), которые позволяют изменять размеры векторов. Эти модели эмбеддингов обучены предоставлять одинаковый контекст при различных размерах векторных измерений. Это открывает возможность использования техники, называемой **адаптивным поиском**.

С помощью адаптивного поиска вы генерируете несколько наборов эмбеддингов разного размера. Сначала поиск выполняется по векторам с меньшим количеством измерений, чтобы приблизиться к финальному результату, поскольку поиск по низкоразмерным векторным пространствам значительно быстрее, чем по высокоразмерным. После того как поиск в низкоразмерных векторах приблизит вас к контенту, наиболее похожему на ваш запрос, поиск адаптируется к работе с более медленными высокоразмерными векторами, чтобы найти самый релевантный контент и завершить поиск по схожести.

В целом, это может ускорить поиск на 30–90%, в зависимости от того, как настроен процесс. Эмбеддинги, созданные с использованием этой техники, называются **матрешечными эмбеддингами**, по аналогии с русскими матрешками, отражая, что эмбеддинги, как и матрешки, относительно одинаковы, но различаются по размеру. Если вам когда-либо потребуется оптимизировать RAG-пайплайн для работы в условиях высокой нагрузки в производственной среде, стоит рассмотреть этот подход.

Следующий важный концепт, который вам нужно понять, — это место в коде, где хранятся ваши векторы. Это поможет вам применить изученные концепции работы с векторами непосредственно к задачам в рамках RAG.

Где в вашем коде скрываются векторы

Один из способов показать ценность векторов в системе RAG — это рассмотреть все места, где они используются. Процесс начинается с текстовых данных, которые преобразуются в векторы на этапе векторизации. Это происходит на стадии индексирования системы RAG. Вам необходимо определить, где именно будут храниться векторы, что приводит нас к понятию хранилища векторов (vector store).

На этапе извлечения данных (retrieval) система получает вопрос от пользователя, который сначала преобразуется в вектор, а затем начинается процесс поиска. Этот процесс использует алгоритм похожести, чтобы определить степень близости между вектором вопроса и всеми векторами, сохраненными в хранилище.

Есть еще одна область, где векторы могут быть полезны: это оценка ответов, полученных от системы RAG. Этот аспект мы рассмотрим в главе 9, где будут описаны методы оценки. А пока давайте углубимся в остальные концепции, начиная с векторизации.

Векторизация происходит в двух местах

В самом начале процесса RAG пользователь вводит свой запрос, который передается механизму извлечения данных. Этот процесс в коде можно увидеть следующим образом:

```
rag_chain_with_source = RunnableParallel(
   {"context": retriever,
    "question":RunnablePassthrough()}
).assign(answer=rag_chain_from_docs)
```

Ретривер — это объект LangChain, который выполняет поиск по схожести и извлечение релевантных векторов на основе пользовательского запроса. Векторизация данных происходит в двух ключевых моментах:

1. Векторизация исходных данных, которые будут использоваться в системе генерации, дополненной поиском (RAG).
2. Векторизация пользовательского запроса.

Обе эти операции связаны тем, что они используются в процессе поиска по схожести. Перед тем как рассматривать процесс поиска, важно понять, где хранятся векторы, полученные из исходных данных. Эти векторы сохраняются в хранилище векторов.

Базы данных/хранилища векторов

Хранилище векторов — это, как правило, векторная база данных, оптимизированная для хранения и обработки векторов. Она играет ключевую роль в эффективной работе системы RAG. Технически, возможно построить систему RAG без использования векторной базы данных, но при этом теряются многие преимущества, заложенные в эти инструменты. Это может негативно повлиять на требования к памяти, вычислительные ресурсы и точность поиска.

В сфере работы с векторами часто используют термин **векторная база данных**, подразумевая оптимизированные структуры для хранения векторов. Однако существуют инструменты и механизмы, которые выполняют схожие функции, но не являются базами данных в полном смысле. Поэтому все такие решения объединяются под общим термином **хранилища векторов**. Этот термин соответствует документации LangChain, где под хранилищами векторов понимаются все типы механизмов, предназначенных для хранения и обработки векторов. Тем не менее, термины "векторная база данных" и "хранилище векторов" часто используются взаимозаменяемо, причем первый из них остается более популярным. Для точности и соответствия терминологии LangChain, в этой книге мы будем использовать термин **хранилище векторов**.

Векторное хранилище является основным местом, где сохраняются векторы, созданные в вашем коде. Когда вы векторизуете данные, полученные эмбеддинги помещаются в векторное хранилище. При выполнении поиска по сходству эмбеддинги, представляющие данные, извлекаются из этого хранилища. Теперь, когда мы знаем, где хранятся эмбеддинги исходных данных, давайте рассмотрим, как они используются в связи с эмбеддингами пользовательских запросов.

Сходство векторов сравнивает ваши векторы

У нас есть два основных процесса векторизации:

- Эмбеддинг пользовательского запроса
- Векторные эмбеддинги, представляющие все данные в нашем векторном хранилище

Рассмотрим, как эти два процесса связаны. Когда мы выполняем поиск по сходству векторов, являющийся основой нашего процесса извлечения, фактически мы выполняем математическую операцию, измеряющую расстояние между эмбеддингом пользовательского запроса и исходными эмбеддингами данных.

Для выполнения вычисления расстояния могут использоваться разные математические алгоритмы, которые мы рассмотрим в главе 8. Однако сейчас важно понимать, что это вычисление позволяет определить ближайшие эмбеддинги исходных данных к эмбеддингу пользовательского запроса и возвращает их список, отсортированный по расстоянию (от ближайшего к наиболее удаленному). В нашем коде это работает немного проще: эмбеддинг представляет точки данных (фрагменты) в соотношении 1:1.

Однако во многих приложениях, таких как чат-боты для вопросов и ответов, где вопросы или ответы длинные и разбиваются на меньшие фрагменты, вы можете встретить ситуации, когда у этих фрагментов есть внешний ключ (ID), ссылающийся на более крупный элемент контента. Это позволяет извлекать полный контент, а не только его фрагмент. Такая архитектура может варьироваться в зависимости от задачи, которую решает система RAG.

Итак, мы рассмотрели основные места, где встречаются векторы в системе RAG: где они появляются, где хранятся и как используются. В следующем разделе мы обсудим, как размер текстовых данных, используемых в поиске для системы RAG, может варьироваться. Вам придется принимать решения в коде, определяющие этот размер. Но, зная основы работы с векторами, вы можете задуматься: если

мы векторизуем контент разных размеров, как это влияет на возможность их сравнения и на построение максимально эффективного процесса извлечения? Этот вопрос абсолютно уместен, и далее мы обсудим, как размер контента, преобразуемого в эмбеддинги, влияет на процесс.

Количество текста, которое вы векторизуете, имеет значение!

Вектор, который мы рассмотрели ранее, был создан на основе текста *What are the advantages of using RAG?* (Каковы преимущества использования генерации, дополненной поиском?). Это относительно небольшой объем текста, что позволяет вектору с размерностью 1 536 эффективно представлять контекст внутри этого текста. Однако, если мы вернемся к коду, данные, которые мы векторизуем для представления информации, берутся из следующего участка:

```
loader = WebBaseLoader(
   web_paths=("https://kbourne.github.io/chapter1.html",),
   bs_kwargs=dict(
      parse_only=bs4.SoupStrainer(
         class_=("post-content", "post-title",
            "post-header")
      )
   ),
)
docs = loader.load()
```

Этот код извлекает веб-страницу, которую мы рассматривали в предыдущих главах, и она значительно длиннее, чем текст вопроса. Чтобы сделать эти данные более удобными для обработки, мы разбиваем содержимое на части с помощью текстового разбиения, реализованного в следующем коде:

```
text_splitter = SemanticChunker(embedding_function)
splits = text_splitter.split_documents(docs)
```

Если бы вы вытащили третий чанк с помощью splits[2], то увидели бы:

> There are also generative models that generate images from text prompts, while others generate video from text prompts. There are other models that generate text descriptions from images. We will talk about these other types of models in Chapter 16, Going Beyond the LLM. But for most of the book, I felt it would keep things simple and let you focus on the core principles of RAG if we focus on the type of model that most RAG pipelines use, the LLM. But I did want to make sure it was clear, that while the book focuses primarily on LLMs, RAG can also be applied to other types of generative models, such as those for images and videos. Some popular examples of LLMs are the OpenAI ChatGPT models, the Meta LLaMA models, Google's PaLM and Gemini models, and Anthropic's Claude models. Foundation model\nA foundation model is the base model for most LLMs. In the case of ChatGPT, the foundation model is based on the GPT (Generative Pretrained Transformer) architecture, and it was fine-tuned for Chat. The specific model used for ChatGPT is not publicly disclosed. The base GPT model cannot talk with you in chatbot-style like ChatGPT does. It had to get further trained to gain that skill.

Я выбрал третий фрагмент для демонстрации, так как он относительно короткий. Большинство фрагментов намного длиннее. Используемый нами текстовый сплиттер Semantic Chunker старается учитывать семантику для определения того, как разбивать текст, применяя эмбеддинги для анализа семантики. Теоретически, это позволяет нам создавать фрагменты, которые лучше структурируют данные в зависимости от контекста, а не по произвольному размеру.

Однако есть важный аспект, связанный с эмбеддингами, который влияет на выбор сплиттера и размер эмбеддингов в целом. Все дело в том, что независимо от длины текста, передаваемого в алгоритм векторизации, результат будет иметь ту же размерность, что и любые другие эмбеддинги. В данном случае это означает, что эмбеддинг пользовательского запроса будет иметь размерность 1 536, и все длинные текстовые фрагменты в хранилище векторов также будут иметь размерность 1 536, несмотря на разницу в их реальной длине в текстовом формате. Это может показаться неинтуитивным, но, что удивительно, такой подход отлично работает!

При выполнении поиска по запросу пользователя в векторном хранилище математические представления эмбеддинга пользовательского запроса и других эмбеддингов создаются таким образом, что мы можем выявлять семантические сходства между ними, несмотря на значительную разницу в их размере. Этот аспект поиска по векторному сходству — одна из тех вещей, которые заставляют математиков любить математику. Кажется совершенно нелогичным, что можно преобразовать текст разного объема в числа и при этом обнаруживать их сходства.

Однако есть еще один аспект, который следует учитывать: когда вы сравниваете результаты только по фрагментам, на которые вы разбиваете свои данные, размер этих фрагментов имеет значение. В данном случае, чем больше объем контента, который подвергается векторизации, тем более "размытым" будет эмбеддинг. С другой стороны, чем меньше объем контента, который представляет эмбеддинг, тем меньше контекста будет доступно для сопоставления при выполнении поиска по векторному сходству. Для каждой реализации RAG необходимо найти тонкий баланс между размером фрагмента и представлением контекста.

Понимание этого поможет принимать более обоснованные решения о том, как разбивать данные и какие алгоритмы векторизации использовать при улучшении системы RAG. Мы рассмотрим дополнительные техники, которые помогут извлечь больше пользы из вашей стратегии разбиения на фрагменты, в главе 11, где речь пойдет о сплиттерах LangChain. Далее мы обсудим важность тестирования различных моделей векторизации.

Не все семантики одинаково хороши!

Распространенная ошибка в приложениях, основанных на генерации, дополненной поиском (RAG), — это использование первого попавшегося алгоритма векторизации и предположение, что он обеспечивает наилучшие результаты. Эти алгоритмы преобразуют семантический смысл текста в математические представления. Однако такие алгоритмы, как правило, сами по себе являются крупными NLP-моделями, и их возможности и качество могут значительно различаться, так же как и у больших языковых моделей (LLM).

Как и люди, эти модели часто сталкиваются с трудностями в понимании тонкостей и нюансов текста. Например, модели прошлого не могли различить значение слова *bark* (лай собаки) и *bark* (кора дерева), но современные модели способны определять это на основе контекста, в котором используется слово. Эта область развивается и адаптируется столь же быстро, как и другие направления в ИИ.

В некоторых случаях специализированная векторизационная модель, обученная на данных конкретной области, например научных статьях, может показать лучшие результаты в приложении, ориентированном на научные тексты, чем универсальная модель. Ученые используют специфический язык, который значительно отличается от того, что можно встретить в социальных сетях. Поэтому большая модель, обученная на общем веб-контенте, может не справляться с задачами в узкоспециализированной области.

Забавный факт. Вы часто слышите о том, как можно дообучать большие языковые модели для улучшения результатов в вашей конкретной области. Но знаете ли вы, что можно также дообучать модели эмбеддингов? Дообучение модели эмбеддингов может улучшить ее понимание данных вашей области и, следовательно, повысить качество поиска по сходству. Это, в свою очередь, может существенно улучшить всю вашу систему RAG, адаптировав ее к специфике вашего домена.

Было бы неправильно говорить о важности алгоритма векторизации, не упомянув, какие из них доступны! Поэтому в следующем разделе мы рассмотрим список некоторых из самых популярных техник векторизации. И даже сделаем это с примерами кода!

Лаборатория кода 7.1 – Распространенные методы векторизации

Алгоритмы векторизации значительно эволюционировали за последние десятилетия. Понимание того, как и почему произошли эти изменения, поможет вам лучше разобраться, какой из алгоритмов лучше всего подходит для ваших нужд.

Давайте рассмотрим некоторые из этих алгоритмов векторизации, начиная с самых ранних и заканчивая самыми современными и продвинутыми вариантами. Это далеко не исчерпывающий список, но выбранные примеры дадут общее представление о том, как развивалась эта часть области и куда она движется.

Прежде чем начать, установим и импортируем несколько новых Python-пакетов, которые играют важную роль в изучении техник векторизации:

```
%pip install gensim --user
%pip install transformers
%pip install torch
```

Этот код должен располагаться в верхней части кода из главы 6 в той же ячейке, что и установка других пакетов.

Частота термина – обратная частота документа (TF-IDF)

1972 год, вероятно, кажется неожиданно ранним для книги о сравнительно новой технологии, такой как генерация, дополненная поиском (RAG). Однако именно тогда были заложены основы обсуждаемых нами техник векторизации. Карен Ида Боалт Сперк Джонс, самоучка-программист и пионер британской компьютерной науки, опубликовала несколько работ, посвященных обработке естественного языка (NLP). В 1972 году она внесла один из самых значимых вкладов, введя концепцию обратной частоты документа (IDF). Основная идея, по ее словам, заключалась в том, что «специфичность термина можно количественно оценить как обратную функцию числа документов, в которых он встречается».

Пример из реального мира. Рассмотрим, как можно применить частоту документа (df) и обратную частоту документа (idf) к некоторым словам из 37 пьес Шекспира. Вы обнаружите, что слово *Romeo* получает самый высокий результат. Это объясняется тем, что оно встречается очень часто, но только в одном документе — в пьесе *Ромео и Джульетта*. Для *Romeo* частота документа (df) будет равна 1, так как оно появляется в одном документе. Обратная частота документа (idf) для *Romeo* составит 1,57, что выше, чем у любого другого слова, благодаря высокой частоте в одном документе.

Напротив, слово *sweet* использовалось Шекспиром время от времени, но в каждой пьесе, что дает ему низкий балл. Для *sweet* df равно 37, а idf равно 0. В своей статье Карен Джонс утверждала, что если слова, такие как *Romeo*, встречаются только в небольшом числе документов, то документы, где они появляются, можно считать очень важными и информативными о содержании. В противоположность этому, слова вроде *sweet* не дают значимой информации ни о самих документах, ни об их важности.

Перейдем к коду! Библиотека scikit-learn предоставляет функцию для векторизации текста с использованием метода TF-IDF. Вот код, где мы определяем переменную splits, которая используется в качестве данных для обучения модели:

```
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.metrics.pairwise import cosine_similarity
tfidf_documents = [split.page_content for split in splits]
tfidf_vectorizer = TfidfVectorizer()
tfidf_matrix = tfidf_vectorizer.fit_transform(tfidf_documents)
vocab = tfidf_vectorizer.get_feature_names_out()
tf_values = tfidf_matrix.toarray()
idf_values = tfidf_vectorizer.idf_
word_stats = list(zip(vocab, tf_values.sum(axis=0), idf_values))
word_stats.sort(key=lambda x: x[2], reverse=True)
print("Word\t\tTF\t\tIDF")
print("----\t\t--\t\t---")
for word, tf, idf in word_stats[:10]: print(f"{word:<12}\t{tf:.2f}\t\t{idf:.2f}")
```

В отличие от модели эмбеддингов OpenAI, эта модель требует обучения на ваших данных, то есть на всем текстовом материале, который у вас есть для этой цели. Этот код в первую очередь демонстрирует, как используется модель TF-IDF по сравнению с текущим ретривером в RAG-конвейере, поэтому мы не будем разбирать его построчно. Однако мы рекомендуем поэкспериментировать с различными настройками кода.

Следует отметить, что векторы, которые создает этот алгоритм, называются разреженными (sparse vectors), а векторы, с которыми мы работали ранее, назывались плотными (dense vectors). Это важное различие, которое мы подробно рассмотрим в главе 8.

Эта модель использует корпус данных для настройки среды, которая затем может рассчитывать эмбеддинги для нового контента, который вы добавляете. Результат должен выглядеть так:

Word	TF	IDF
000	0.16	2.95
1024	0.04	2.95
123	0.02	2.95
13	0.04	2.95
15	0.01	2.95
16	0.07	2.95
192	0.06	2.95
1m	0.08	2.95
200	0.08	2.95
2024	0.01	2.95

Рис. 7.2. Таблица TG–IDF

Мы показали только 10 слов, но у всех у них одинаковое высокое значение idf, и все они представляют собой текст, основанный на числах. Это не кажется особенно полезным, но причиной этого является то, что наш корпус данных слишком мал. Обучение на большем объеме данных того же автора или из той же области может помочь создать модель с более глубоким контекстуальным пониманием содержания.

Теперь, возвращаясь к изначальному вопросу, который мы рассматривали: *Каковы преимущества RAG?*, мы хотим использовать эмбеддинги TF-IDF, чтобы определить, какие документы являются наиболее релевантными.

```
tfidf_user_query = ["What are the advantages of RAG?"]
new_tfidf_matrix = tfidf_vectorizer.transform(
   tfidf_user_query)
tfidf_similarity_scores = cosine_similarity(
   new_tfidf_matrix, tfidf_matrix)
tfidf_top_doc_index = tfidf_similarity_scores.argmax()
print("TF-IDF Top Document:\n",
   tfidf_documents[tfidf_top_doc_index])
```

Это воспроизводит поведение, которое мы наблюдаем у ретривера, где используется алгоритм сходства для поиска ближайшего эмбеддинга по расстоянию. В данном случае мы применяем косинусное сходство, о котором подробнее поговорим в главе 8, но стоит помнить, что существует множество алгоритмов расчета расстояния, которые можно использовать для этой цели. Результат выполнения кода:

> TF-IDF Top Document:
> Can you imagine what you could do with all of the benefits mentioned above, but combined with all of the data within your company, about everything your company has ever done, about your customers and all of their interactions, or about all of your products and services combined with a knowledge of what a specific customer's needs are? You do not have to imagine it, that is what RAG does…[TRUNCATED FOR BREVITY]

Если вы запустите наш оригинальный код, который использует оригинальное хранилище векторов и ретривер, вы увидите следующий вывод:

> Retrieved Document:
> Can you imagine what you could do with all of the benefits mentioned above, but combined with all of the data within your company, about everything your company has ever done, about your customers and all of their interactions, or about all of your products and services combined with a knowledge of what a specific customer's needs are? You do not have to imagine it, that is what RAG does…[TRUNCATED FOR BREVITY]

Они совпадают! Небольшой алгоритм из 1972 года, обученный на наших данных за доли секунды, оказывается таким же эффективным, как и масштабные алгоритмы, разработанные OpenAI, на создание которых потратили миллиарды долларов!

Но давайте притормозим — это определенно НЕ так! В реальных сценариях вы будете работать с гораздо более крупным набором данных и более сложными пользовательскими запросами, где использование современных сложных техник эмбеддингов принесет пользу.

TF-IDF был очень полезен на протяжении многих лет. Но действительно ли было необходимо изучать алгоритм из 1972 года, когда мы обсуждаем самые передовые модели генеративного ИИ? Ответ — BM25.[1] Это только затравка, но в следующей главе вы узнаете больше об этом популярном алгоритме поиска по ключевым словам, который сегодня является одним из самых востребованных. И угадайте что? Он основан на TF-IDF!

Однако у TF-IDF есть проблема: он плохо захватывает контекст и семантику по сравнению с моделями, о которых мы поговорим далее. Перейдем к следующему значительному шагу вперед: Word2Vec и связанным алгоритмам.

Word2Vec, Sentence2Vec и Doc2Vec

Word2Vec и похожие модели представляют собой ранний пример использования обучения без учителя, что стало значительным шагом вперед в области обработки естественного языка (NLP). Существует несколько типов моделей на основе векторов (Word2Vec, Doc2Vec, Sentence2Vec), которые различаются уровнем текста, на котором они обучаются: слова, документы или предложения соответственно.

- **Word2Vec** учится представлять отдельные слова в виде векторов, отражающих их семантическое значение и отношения между словами.
- **Doc2Vec** обучается на уровне целых документов, создавая векторные представления, которые захватывают общий контекст и тематику документа.
- **Sentence2Vec** работает аналогично Doc2Vec, но фокусируется на уровне отдельных предложений, создавая их векторные представления.

Word2Vec полезен для задач, связанных с аналогиями и семантической близостью слов. Doc2Vec и Sentence2Vec лучше подходят для задач на уровне документов, таких как сравнение документов, их классификация или поиск.

Поскольку мы работаем с более крупными документами, а не только словами или предложениями, мы выберем модель **Doc2Vec**. Мы обучим эту модель на наших данных, чтобы посмотреть, как она работает в роли ретривера. Как и модель TF-IDF, эту модель можно обучить на наших данных, после чего передать в нее пользовательский запрос, чтобы проверить, какие данные будут выбраны, как наиболее релевантные. Добавьте следующий код в новую ячейку после кода для TF-IDF:

```
from gensim.models.doc2vec import Doc2Vec, TaggedDocument
from sklearn.metrics.pairwise import cosine_similarity
doc2vec_documents = [
   split.page_content for split in splits]
doc2vec_tokenized_documents = [
   doc.lower().split() for doc in doc2vec_documents]
doc2vec_tagged_documents = [TaggedDocument(words=doc,
   tags=[str(i)]) for i, doc in enumerate(
   doc2vec_tokenized_documents)]
doc2vec_model = Doc2Vec(doc2vec_tagged_documents,
   vector_size=100, window=5, min_count=1, workers=4)
doc2vec_model.save("doc2vec_model.bin")
```

Подобно модели TF-IDF, этот код предназначен в первую очередь для демонстрации того, как используется модель Doc2Vec по сравнению с текущим ретривером в RAG-конвейере. Поэтому мы не будем разбирать его построчно, но рекомендуем вам попробовать код самостоятельно и поэкспериментировать с различными настройками. Этот код сосредоточен на обучении модели Doc2Vec и сохранении ее локально.

Интересный факт. Обучение языковых моделей — это популярная и хорошо оплачиваемая область в наши дни. Вы когда-нибудь обучали языковую модель? Если ваш ответ «нет», то вы ошибаетесь. Вы

[1] Полное название современного алгоритма, основанного на TF-IDF – Best Matching 25. – *Прим. Багузина*

только что обучили языковую модель, более того – две! TF-IDF и Doc2Vec — это языковые модели, которые вы только что обучили. Это относительно простые версии обучения моделей, но с чего-то нужно начинать, и вы только что сделали это!

В следующем коде мы будем использовать модель Doc2Vec для наших данных:

```
loaded_doc2vec_model = Doc2Vec.load("doc2vec_model.bin")
doc2vec_document_vectors = [loaded_doc2vec_model.dv[
   str(i)] for i in range(len(doc2vec_documents))]
doc2vec_user_query = ["What are the advantages of RAG?"]
doc2vec_tokenized_user_query = [content.lower().split() for content in
doc2vec_user_query]
doc2vec_user_query_vector = loaded_doc2vec_model.infer_vector(
   doc2vec_tokenized_user_query[0])
doc2vec_similarity_scores = cosine_similarity([
   doc2vec_user_query_vector], doc2vec_document_vectors)
doc2vec_top_doc_index = doc2vec_similarity_scores.argmax()
print("\nDoc2Vec Top Document:\n",
   doc2vec_documents[doc2vec_top_doc_index])
```

Мы отделили код для создания и сохранения модели от использования модели, чтобы вы могли увидеть, как эту модель можно сохранить и как на нее сослаться позже. Вот результат работы с этим кодом:

> Doc2Vec Top Document:
> Once you have introduced the new knowledge, it will always have it! It is also how the model was originally created, by training with data, right? That sounds right in theory, but in practice, fine-tuning has been more reliable in teaching a model specialized tasks (like teaching a model how to converse in a certain way), and less reliable for factual recall…[TRUNCATED FOR BREVITY]

По сравнению с результатами нашего первоначального ретривера, эта модель возвращает другие результаты. Однако данная модель была настроена с использованием всего лишь 100-мерных векторов в следующей строке:

```
doc2vec_model = Doc2Vec(doc2vec_tagged_documents,
   vector_size=100, window=5, min_count=1, workers=4)
```

Что произойдет, если вы измените vector_size на 1,536 (тот же размер, что и в модели OpenAI)?

Измените определение переменной doc2vec_model на следующее:

```
doc2vec_model = Doc2Vec(doc2vec_tagged_documents,
   vector_size=1536, window=5, min_count=1, workers=4)
```

Результаты будут выглядеть следующим образом:

> Doc2Vec Top Document:
> Can you imagine what you could do with all of the benefits mentioned above, but combined with all of the data within your company, about everything your company has ever done, about your customers and all of their interactions, or about all of your products and services combined with a knowledge of what a specific customer's needs are? You do not have to imagine it, that is what RAG does… [TRUNCATED FOR BREVITY]

Текст оказался таким же, как и в исходных результатах с использованием эмбеддингов OpenAI. Однако результаты этой модели непоследовательны. Если обучить эту модель на большем объеме данных, скорее всего, качество результатов улучшится.

Теоретически преимущество этой модели перед TF-IDF заключается в том, что она основана на нейронной сети и учитывает окружающие слова, тогда как TF-IDF — это просто статистическая мера, оценивающая, насколько релевантно слово документу (поиск по ключевым словам). Но, как мы говорили о модели TF-IDF, существуют еще более мощные модели, чем векторные, которые способны захватывать гораздо больше контекста и семантики текста. Перейдем к следующему поколению моделей — трансформерам.

Двунаправленные представления кодировщика из трансформеров

С использованием BERT (Bidirectional Encoder Representations from Transformers), мы полностью переходим к применению нейронных сетей для более глубокого понимания семантики корпуса, что является еще одним значительным шагом вперед для алгоритмов NLP. BERT также стал одной из первых моделей, применивших трансформеры — специфический тип нейронных сетей, который стал важным этапом в развитии технологий, приведших к появлению больших языковых моделей (LLM), известных нам сегодня. Например, популярные модели ChatGPT от OpenAI также являются трансформерами, но они обучались на гораздо большем корпусе данных и с использованием иных методов по сравнению с BERT.

Тем не менее, BERT остается очень мощной моделью. Ее можно использовать как автономную модель, импортируемую локально, что позволяет не зависеть от API, таких как сервис эмбеддингов OpenAI. Возможность использовать локальную модель в вашем коде может стать большим преимуществом в средах с ограниченным доступом к сети, вместо того чтобы полагаться на облачные API-сервисы.

Одной из ключевых характеристик трансформеров является механизм самовнимания (self-attention), который позволяет учитывать зависимости между словами в тексте. BERT включает в себя несколько слоев трансформеров, что позволяет модели изучать еще более сложные представления. В отличие от модели Doc2Vec, BERT уже предварительно обучен на больших объемах данных, таких как Wikipedia и BookCorpus, с целью предсказания следующего предложения.

Код для сравнения извлеченных результатов с использованием BERT:

```
from transformers import BertTokenizer, BertModel
import torch
from sklearn.metrics.pairwise import cosine_similarity
bert_documents = [split.page_content for split in splits]
Code lab 7.1 – Common vectorization techniques 113
bert_tokenizer = BertTokenizer.from_pretrained(
   'bert-base-uncased')
bert_model = BertModel.from_pretrained('bert-base-uncased')
bert_vector_size = bert_model.config.hidden_size
print(f"Vector size of BERT (base-uncased) embeddings:
   {bert_vector_size}\n")
bert_tokenized_documents = [bert_tokenizer(doc,
   return_tensors='pt', max_length=512, truncation=True)
   for doc in bert_documents]
bert_document_embeddings = []
with torch.no_grad():
   for doc in bert_tokenized_documents:
      bert_outputs = bert_model(**doc)
      bert_doc_embedding =
         bert_outputs.last_hidden_state[0, 0, :].numpy()
      bert_document_embeddings.append(bert_doc_embedding)
bert_user_query = ["What are the advantages of RAG?"]
bert_tokenized_user_query = bert_tokenizer(
   bert_user_query[0], return_tensors='pt',
   max_length=512, truncation=True)
bert_user_query_embedding = []
with torch.no_grad():
   bert_outputs = bert_model(
      **bert_tokenized_user_query)
      bert_user_query_embedding =
         bert_outputs.last_hidden_state[
            0, 0, :].numpy()
bert_similarity_scores = cosine_similarity([
   bert_user_query_embedding], bert_document_embeddings)
```

```
bert_top_doc_index = bert_similarity_scores.argmax()
print("BERT Top Document:\n", bert_documents[
   bert_top_doc_index])
```

В этом коде есть одно очень важное отличие по сравнению с использованием предыдущих моделей. Здесь мы не выполняем настройку модели на наших данных. Модель BERT уже была обучена на большом наборе данных. Тем не менее, возможно дополнительно дообучить модель на наших данных, что рекомендуется, если вы планируете использовать ее для конкретных задач.

Результаты будут отражать отсутствие дообучения, но это не помешает нам показать, как эта модель работает! Код выводит размер вектора для сравнения с другими моделями:

```
Vector size of BERT (base-uncased) embeddings: 768
BERT Top Document:
Or if you are developing in a legal field, you may want it to sound more like a lawyer. Vector Store or Vector Database?
```

Размер вектора составляет внушительные 768. Даже без метрик можно сказать, что лучший найденный документ не является оптимальным фрагментом для ответа на вопрос *«Каковы преимущества RAG?»*.

Эта модель мощная и имеет потенциал для работы лучше, чем предыдущие модели, но для этого потребуется дополнительная настройка (fine-tuning), чтобы улучшить ее работу с нашими данными по сравнению с другими типами моделей эмбеддингов, которые мы обсуждали. Это может быть не обязательно для всех данных, но в специализированных областях, таких как эта, дообучение модели стоит рассмотреть как вариант. Особенно это важно, если вы используете небольшую локальную модель, а не крупный хостинг через API, например, API для эмбеддингов OpenAI.

Рассмотрение этих трех различных моделей демонстрирует, насколько изменились модели эмбеддингов за последние 50 лет. Надеюсь, примеры показали, насколько важным является выбор модели эмбеддингов. Мы завершим обсуждение моделей эмбеддингов, вернувшись к той, с которой начали, — модели эмбеддингов OpenAI, доступной через API OpenAI. Мы также обсудим эту модель и ее аналоги в других облачных сервисах.

OpenAI и другие масштабные сервисы эмбеддинга

Давайте подробнее поговорим о модели BERT, которую только что использовали, в сравнении с моделью эмбеддингов OpenAI. Мы использовали версию *'bert-base-uncased'*, которая представляет собой достаточно мощную трансформерную модель с 110 миллионами параметров, особенно в сравнении с предыдущими рассматренными моделями.

В зависимости от среды, в которой вы работаете, эта модель может проверять пределы ваших вычислительных возможностей. Для моего компьютера это была самая крупная из версий BERT, которую он смог обработать. Однако, если у вас более мощная вычислительная среда, вы можете изменить модель в этих двух строках на *'bert-large-uncased'*:

```
tokenizer = BertTokenizer.from_pretrained(
   'bert-base-uncased')
model = BertModel.from_pretrained('bert-base-uncased')
```

Вы можете ознакомиться с полным списком вариантов BERT здесь.

Модель *'bert-large-uncased'* содержит 340 миллионов параметров, что более чем в три раза превышает размер *'bert-base-uncased'*. Если ваша вычислительная среда не способна обработать такую модель, это приведет к сбою ядра, и вам придется заново загружать все импортированные библиотеки и соответствующие ячейки ноутбука. Это подчеркивает, насколько крупными могут быть такие модели. Однако следует уточнить, что обе модели BERT имеют 110 миллионов и 340 миллионов параметров, то есть речь идет о миллионах, а не миллиардах.

Модель эмбеддингов OpenAI, которую мы использовали, основана на архитектуре GPT-3 и имеет 175 миллиардов параметров. Да, это миллиарды. Позже в этой главе мы поговорим о новых моделях эмбеддингов OpenAI, основанных на архитектуре GPT-4, которые содержат уже один триллион параметров. Эти модели огромны и значительно превосходят любую из обсуждаемых нами ранее.

BERT и OpenAI — обе трансформерные модели, но BERT был обучен на 3,3 миллиарда слов, тогда как полный корпус GPT-3 оценивается примерно в 17 триллионов слов.

В настоящее время OpenAI предлагает три разные модели эмбеддингов. Мы использовали более старую модель *'text-embedding-ada-002'*, основанную на GPT-3, чтобы снизить затраты на API, но она по-прежнему является весьма мощной моделью. Две другие, более новые модели, основаны на GPT-4: *'text-embedding-3-small'* и *'text-embedding-3-large'*. Обе эти модели поддерживают матрешечные эмбеддинги, позволяя использовать адаптивный подход к извлечению данных.

Однако OpenAI — не единственный облачный провайдер, предлагающий API для текстовых эмбеддингов. Google Cloud Platform (GCP) предоставляет услуги API для текстовых эмбеддингов, а последняя версия модели была выпущена 9 апреля 2024 года под названием *'text-embedding-preview-0409'*. Эта модель, наряду с новыми моделями OpenAI, является одной из немногих крупных облачных моделей, поддерживающих матрешечные эмбеддинги.

Amazon Web Services (AWS) также предлагает модели эмбеддингов на основе своей модели Titan, а также модели компании Cohere. Ожидается, что версия Titan Text Embeddings V2 будет выпущена в ближайшее время и также будет поддерживать матрешечные эмбеддинги.

Мы рассмотрели модели, которые отражают ключевые этапы прогресса в этой области за последние 50 лет, но это лишь малая часть всех доступных методов для генерации эмбеддингов. Теперь, когда ваше понимание возможностей эмбеддингов расширилось, давайте перейдем к факторам, которые следует учитывать при выборе модели.

Факторы выбора варианта векторизации

Выбор подходящей опции векторизации — это ключевое решение при построении системы генерации, дополненной поиском (RAG). Основные аспекты, которые следует учитывать: качество эмбеддингов для вашего приложения, связанные с этим расходы, доступность сети, скорость генерации эмбеддингов и совместимость между различными моделями эмбеддингов. Помимо рассмотренных выше моделей, существуют и другие варианты, которые можно изучить в зависимости от ваших потребностей. Давайте рассмотрим эти аспекты подробнее.

Качество эмбеддингов

При оценке качества эмбеддингов нельзя полагаться только на общие метрики, представленные для каждой модели. Например, модель OpenAI *'text-embedding-ada-002'* набрала 61,0% в тесте Massive Text Embedding Benchmark (MTEB), тогда как модель *'text-embedding-3-large'* получила 64,6%. Эти метрики могут быть полезными, особенно когда вы ищете модель определенного уровня качества, но это не означает, что одна модель будет на 3,6% лучше в вашем конкретном случае. Более того, это вообще не гарантирует, что одна модель будет лучше другой.

Не полагайтесь исключительно на общие тесты. Важно то, насколько хорошо эмбеддинги работают в рамках вашего конкретного приложения. Это касается и моделей эмбеддингов, которые вы обучаете на своих данных. Если вы работаете над приложением, связанным с определенной областью, например наукой, юриспруденцией или технологиями, весьма вероятно, что вы сможете найти или обучить модель, которая будет работать лучше с вашими специализированными данными.

Когда вы начинаете проект, попробуйте несколько моделей эмбеддингов в рамках вашей системы RAG, а затем используйте методы оценки, которые мы рассмотрим в главе 9, чтобы сравнить результаты. Это поможет определить, какая модель лучше всего подходит для вашего приложения.

Стоимость

Стоимость услуг для создания эмбеддингов варьируется от бесплатной до относительно высокой. Например, самая дорогая модель эмбеддингов OpenAI стоит $0.13 за миллион токенов. Это означает, что обработка одной страницы текста, содержащей 800 токенов, обойдется вам в $0.000104, или чуть больше 1% от одного цента. На первый взгляд это кажется незначительным, но в большинстве случаев, особенно в корпоративных приложениях, эти затраты быстро масштабируются, достигая тысяч или даже десятков тысяч долларов даже для небольших проектов. Однако другие API для эмбеддингов стоят дешевле и могут удовлетворять вашим требованиям не хуже. Более того, если вы создаете собственную модель, ваши расходы будут ограничены стоимостью оборудования или хостинга для этой модели. Со временем это может оказаться гораздо дешевле и при этом подходить для ваших нужд.

Доступность сети

При выборе модели эмбеддингов важно учитывать доступность сети. Практически у всех приложений могут возникнуть ситуации, когда сеть будет недоступна. Это может повлиять как на доступ пользователей к интерфейсу приложения, так и на сетевые вызовы, которые ваше приложение отправляет в другие сервисы.

Например, может сложиться ситуация, когда пользователи могут получить доступ к интерфейсу вашего приложения, но само приложение не может подключиться к сервису эмбеддингов OpenAI для генерации эмбеддингов по запросу пользователя. Как вы будете поступать в таком случае? Если вы используете модель, которая работает в вашей локальной среде, такая проблема не возникает. Это важно учитывать с точки зрения доступности и влияния на опыт пользователей.

Обратите внимание, что вы не можете просто так заменить модель эмбеддингов для пользовательского запроса. Если вы думаете, что сможете использовать локальную модель как резервный вариант при недоступности сети, это не всегда работает. Если вы используете проприетарную модель, доступную только через API, ваша система RAG будет зависеть от доступности этого API. OpenAI не предоставляет свои модели эмбеддингов для локального использования. Подробнее об этом в разделе *Совместимость эмбеддингов*.

Скорость

Скорость генерации эмбеддингов — важный фактор, который влияет на пользовательский опыт. При использовании хостинг-сервиса API OpenAI запросы для генерации эмбеддингов отправляются через сеть. Хотя такие сетевые вызовы обычно выполняются быстро, они все же вносят задержку по сравнению с генерацией эмбеддингов локально.

Однако локальная генерация не всегда быстрее, так как скорость также зависит от используемой модели. Некоторые модели могут иметь медленное время вывода, что сводит на нет преимущества локальной обработки. Основные аспекты, которые следует учитывать при выборе модели с точки зрения скорости: задержка сети, время вывода модели, доступные аппаратные ресурсы и возможность пакетной генерации эмбеддингов для оптимизации работы.

Совместимость эмбеддингов

Это очень важный аспект работы с эмбеддингами! При сравнении эмбеддингов, например, для выявления сходства между эмбеддингом пользовательского запроса и эмбеддингами, хранящимися в векторном хранилище, их необходимо создавать с использованием одной и той же модели. Эти модели генерируют уникальные векторные сигнатуры, которые специфичны только для данной модели.

Это правило справедливо даже для моделей одного поставщика услуг. Например, в OpenAI три модели эмбеддингов несовместимы друг с другом. Если вы используете одну из моделей OpenAI для векторизации эмбеддингов, хранящихся в вашем векторном хранилище, вам придется обращаться к API OpenAI и использовать ту же модель для векторизации пользовательского запроса.

По мере роста вашего приложения изменение или обновление модели эмбеддингов может иметь серьезные финансовые последствия, поскольку это потребует генерации новых эмбеддингов для новой модели. Это может стать причиной выбора локальной модели вместо хостинг-сервиса API, так как генерация эмбеддингов с помощью модели, находящейся под вашим контролем, обычно обходится дешевле.

Ключевые выводы

- Генерация эмбеддингов на основе универсальных тестов может быть полезной, но для вашего приложения важно проводить тестирование в конкретном домене.
- Затраты могут сильно варьироваться в зависимости от поставщика услуг и объема требуемых эмбеддингов.
- Доступность сети и скорость генерации — важные аспекты, особенно при использовании хостинг-сервисов API.
- Совместимость между моделями критически важна, так как эмбеддинги, созданные разными моделями, невозможно сравнивать напрямую.
- По мере роста вашего приложения смена или обновление модели эмбеддингов потребует значительных затрат.

Локальная генерация эмбеддингов предоставляет больший контроль и потенциально снижает затраты, но скорость будет зависеть от используемой модели и доступных аппаратных ресурсов. Тщательное тестирование и бенчмаркинг необходимы для нахождения оптимального баланса между качеством, стоимостью, скоростью и другими важными факторами для вашего приложения.

Теперь, когда мы рассмотрели аспекты выбора опции векторизации, перейдем к теме их хранения в векторных хранилищах.

Начало работы с векторными хранилищами

Векторные хранилища, наряду с другими типами хранилищ данных, являются топливом для вашей системы генерации, дополненной поиском (RAG). Очевидно, что без места для хранения данных, ориентированных на RAG, которые обычно включают создание, управление, фильтрацию и поиск векторов, вы не сможете построить полноценную систему RAG. Выбор хранилища и способ его реализации будут иметь значительное влияние на производительность всей вашей системы, поэтому это важное решение.

Чтобы начать этот раздел, давайте вернемся к понятию базы данных.

Источники данных (не векторные)

В нашем базовом примере RAG мы пока упрощаем процесс и не подключаем дополнительных ресурсов баз данных. Можно рассматривать веб-страницу, с которой извлекается контент, как базу данных, хотя в данном контексте точнее будет назвать ее неструктурированным источником данных.

Тем не менее, весьма вероятно, что ваше приложение со временем расширится до уровня, на котором потребуется поддержка базы данных. Это может быть традиционная SQL-база данных или огромный дата-лейк (хранилище, содержащее все типы исходных данных, преимущественно неструктурированных), где данные предварительно обрабатываются для создания более удобного формата, поддерживающего вашу систему RAG.

Архитектура вашего хранилища данных может быть основана на реляционной системе управления базами данных (RDBMS), разнообразных типах NoSQL, NewSQL (объединяющей лучшие качества SQL и NoSQL), или различных версиях дата-вуагхаузов и дата-лейков.

С точки зрения этой книги, мы рассматриваем представленные этими системами источники данных как абстрактное понятие. Однако важно понимать, что ваш выбор векторного хранилища, скорее всего, будет сильно зависеть от существующей архитектуры ваших источников данных. Текущие технические навыки вашей команды также будут играть ключевую роль в этом решении.

Например, если вы используете PostgreSQL в качестве RDBMS, а ваша команда имеет значительный опыт работы с этой системой и оптимизации ее производительности, вам стоит рассмотреть использование расширения **pgvector** для PostgreSQL. Это расширение превращает таблицы PostgreSQL в векторные хранилища, позволяя использовать многие возможности PostgreSQL, с которыми ваша команда уже знакома, в контексте работы с векторами. Концепции, такие как индексация и написание SQL-запросов для PostgreSQL, уже будут знакомы вашей команде, что ускорит процесс освоения pgvector.

Если же вы строите всю свою инфраструктуру данных с нуля, что редко встречается в корпоративной среде, вы можете выбрать другой подход, оптимизированный по скорости, стоимости, точности или всем этим критериям сразу. Однако для большинства компаний важно учитывать совместимость с существующей инфраструктурой при выборе векторного хранилища.

Забавный факт. Что насчет таких приложений, как SharePoint?

SharePoint обычно считается системой управления контентом (CMS) и не всегда строго вписывается в определения других источников данных, упомянутых ранее. Однако SharePoint и подобные приложения содержат огромные массивы неструктурированных данных, включая документы PDF, Word, Excel и PowerPoint, которые составляют значительную часть базы знаний компании, особенно в крупных корпоративных средах. Учитывая, что генеративный ИИ продемонстрировал уникальную способность работать с неструктурированными данными, превосходя предшествующие технологии, такие приложения становятся отличным источником данных для систем RAG.

Подобные приложения также обладают развитым API, который позволяет извлекать данные в процессе загрузки документов. Например, можно извлечь текст из документа Word и поместить его в

базу данных до этапа векторизации. Во многих крупных компаниях из-за высокой ценности данных в таких приложениях и относительной легкости их извлечения с помощью API это становится одним из первых источников данных для систем RAG. Таким образом, да, SharePoint и аналогичные приложения можно и нужно включать в список потенциальных источников данных!

Мы скоро подробнее поговорим о *pgvector* и других вариантах векторных хранилищ. Важно понимать, что выбор векторного хранилища может быть специфичным для каждой ситуации, а также что факторы, не связанные напрямую с самим хранилищем, будут играть важную роль в принятии решения. Независимо от выбранной или уже используемой опции, векторное хранилище является ключевым компонентом, который поставляет данные вашей системе RAG.

Векторные хранилища

Векторные хранилища, также известные как векторные базы данных или векторные поисковые движки, — это специализированные системы хранения, разработанные для эффективного хранения, управления и извлечения векторных представлений данных. В отличие от традиционных баз данных, которые организуют данные в строках и столбцах, векторные хранилища оптимизированы для операций в многомерных векторных пространствах. Они играют решающую роль в эффективной системе RAG, предоставляя быстрый поиск по сходству, необходимый для выявления наиболее релевантной информации в ответ на векторизированный запрос.

Архитектура векторного хранилища обычно включает три основные компонента:

- **Слой индексации**. Организует векторы так, чтобы ускорить поисковые запросы. Используются методы индексации, такие как разбиение на деревья (например, KD-деревья) или хэширование (например, хэширование, чувствительное к локальности), для быстрого извлечения векторов, близких друг к другу в векторном пространстве.
- **Слой хранения**. Эффективно управляет хранением данных на диске или в памяти, обеспечивая оптимальную производительность и масштабируемость.
- **Слой обработки (опционально)**. Некоторые векторные хранилища включают слой обработки для трансформации векторов, вычислений сходства и других аналитических операций в реальном времени.

Хотя технически возможно построить систему RAG без использования векторного хранилища, это приведет к снижению производительности и масштабируемости. Векторные хранилища специально разработаны для решения уникальных задач хранения и обслуживания многомерных векторов, предлагая оптимизации, которые значительно улучшают использование памяти, вычислительные ресурсы и точность поиска.

Как уже упоминалось, термины *векторная база данных* и *векторное хранилище* часто используются как синонимы, но не все векторные хранилища являются базами данных. Существуют и другие инструменты и механизмы, которые выполняют ту же или похожую функцию, что и векторные базы данных. Для точности и соответствия документации LangChain мы будем использовать термин *векторное хранилище* для обозначения всех механизмов, которые хранят и предоставляют векторы, включая векторные базы данных и другие решения, не являющиеся базами данных.

Далее мы рассмотрим доступные опции векторных хранилищ, чтобы лучше понять, что можно использовать.

Обзор популярных векторных хранилищ

При выборе векторного хранилища учитывайте такие факторы, как требования к масштабируемости, простота настройки и обслуживания, потребности в производительности, бюджетные ограничения, а также уровень контроля и гибкости, которые вы хотите иметь над базовой инфраструктурой. Также важно оценить варианты интеграции и поддерживаемые языки программирования, чтобы убедиться в совместимости с вашей текущей технологической средой.

Существует множество векторных хранилищ: часть из них разработана известными компаниями и сообществами в области баз данных, многие являются продуктами новых стартапов, и ежедневно появляются новые решения. В то же время вероятно, что некоторые из них могут выйти из бизнеса к тому моменту, когда вы будете читать этот текст. Это очень динамичная сфера!

Будьте внимательны и используйте информацию из этой главы, чтобы определить аспекты, наиболее важные для ваших конкретных приложений RAG. Затем изучите текущий рынок, чтобы определить, какой вариант подходит вам лучше всего.

Мы сосредоточимся на тех векторных хранилищах, которые имеют установленную интеграцию с LangChain. При этом мы ограничим список, чтобы не перегружать вас, но дадим достаточно вариантов, чтобы вы смогли понять, какие типы решений существуют. Имейте в виду, что эти векторные хранилища постоянно добавляют новые функции и улучшения. Перед выбором обязательно ознакомьтесь с их последними версиями! Это может существенно повлиять на ваше решение и помочь сделать более удачный выбор.

В следующих подразделах мы рассмотрим некоторые распространенные варианты векторных хранилищ, интегрирующихся с LangChain, и то, что следует учитывать при выборе каждого из них.

Chroma — это векторная база данных с открытым исходным кодом. Она обеспечивает высокую скорость поиска и легкую интеграцию с LangChain через Python SDK. Chroma выделяется своей простотой и удобством использования благодаря понятному API и поддержке динамической фильтрации коллекций во время поиска. Она также предлагает встроенную поддержку разбиения документов на фрагменты (чанки, chunking) и индексацию, что делает ее удобной для работы с большими текстовыми наборами данных.

Chroma — хороший выбор, если вы цените простоту и хотите использовать решение с открытым исходным кодом, которое можно развернуть локально. Однако она может не обладать такими продвинутыми функциями, как распределенный поиск, поддержка множества алгоритмов индексации или гибридный поиск, совмещающий сходство векторов с фильтрацией по метаданным.

LanceDB — это векторная база данных, разработанная для эффективного поиска по сходству и извлечения данных. Она выделяется своими гибридными возможностями поиска, объединяя поиск по сходству векторов с традиционным поиском по ключевым словам.

LanceDB поддерживает различные метрики расстояний и алгоритмы индексации, включая HNSW (Hierarchical Navigable Small World) для эффективного поиска приблизительных ближайших соседей. LanceDB интегрируется с LangChain, обеспечивает высокую производительность поиска и поддерживает разнообразные техники индексации.

LanceDB подойдет, если вам нужна специализированная векторная база данных с хорошей производительностью и интеграцией с LangChain. Однако у нее может быть меньшее сообщество и экосистема по сравнению с другими вариантами.

Milvus — это векторная база данных с открытым исходным кодом, предоставляющая масштабируемый поиск по сходству и поддержку различных алгоритмов индексации. Она имеет облачно-ориентированную архитектуру и поддерживает развертывание на основе Kubernetes для обеспечения масштабируемости и высокой доступности.

Milvus предлагает такие функции, как мультивекторная индексация, позволяющая искать сразу по нескольким векторным полям, и систему плагинов для расширения функциональности. Она хорошо интегрируется с LangChain, поддерживает распределенное развертывание и горизонтальную масштабируемость.

Milvus подойдет, если вам нужно масштабируемое и функционально богатое векторное хранилище с открытым исходным кодом. Однако оно может потребовать больше настроек и управления по сравнению с облачными сервисами.

Pgvector — это расширение для PostgreSQL, добавляющее поддержку поиска по векторному сходству и интегрирующееся с LangChain как векторное хранилище. Оно использует мощность и надежность PostgreSQL, самой продвинутой реляционной базы данных с открытым исходным кодом, а также преимущества зрелой экосистемы PostgreSQL, обширной документации и сильной поддержки сообщества. pgvector позволяет бесшовно интегрировать поиск по векторному сходству с традиционными функциями реляционных баз данных, что дает возможность выполнять гибридный поиск. Недавние обновления улучшили производительность pgvector, что позволило ей конкурировать с другими специализированными векторными базами данных.

Учитывая, что PostgreSQL является одной из самых популярных баз данных в мире (проверенная временем зрелая технология с огромным сообществом), а расширение pgvector добавляет возможности, аналогичные другим векторным базам данных, эта комбинация является отличным решением для любой компании, уже использующей PostgreSQL.

Pinecone — это полностью управляемый сервис векторной базы данных, предоставляющий высокую производительность, масштабируемость и легкую интеграцию с LangChain. Pinecone обладает такими функциями, как индексирование в реальном времени, что позволяет обновлять и искать векторы с минимальной задержкой. Он также поддерживает гибридный поиск, совмещающий сходство векторов с фильтрацией по метаданным, и предоставляет такие возможности, как распределенный поиск и поддержку множества алгоритмов индексации.

Pinecone является отличным выбором, если вам нужно управляемое решение с хорошей производительностью и минимальными настройками. Однако он может оказаться дороже, чем решения, развернутые локально.

Weaviate — это поисковый движок с открытым исходным кодом для работы с векторами, который поддерживает различные алгоритмы индексации и поиска по сходству. Он использует подход на основе схем (schema-based), позволяя задавать семантическую модель данных для ваших векторов.

Weaviate поддерживает операции CRUD, проверку данных, механизмы авторизации и предлагает модули для распространенных задач машинного обучения, таких как классификация текста и поиск сходства изображений. Он интегрируется с LangChain и предоставляет такие функции, как управление схемами, индексирование в реальном времени и GraphQL API.

Weaviate подойдет, если вам нужен поисковый движок с открытым исходным кодом, обладающий расширенными функциями и гибкостью. Однако его использование может потребовать более сложной настройки и конфигурации по сравнению с управляемыми сервисами.

В предыдущих разделах мы рассмотрели различные варианты векторных хранилищ, интегрирующихся с LangChain, и дали общий обзор их функций, сильных сторон и факторов, которые следует учитывать при выборе. Это подчеркивает важность оценки таких аспектов, как масштабируемость, простота использования, производительность, бюджет и совместимость с существующими технологиями, при выборе векторного хранилища.

Несмотря на широкий охват, этот список остается кратким по сравнению с общим количеством доступных вариантов для интеграции с LangChain и использования в качестве векторных хранилищ.

Упомянутые векторные хранилища предлагают широкий спектр возможностей, включая быстрый поиск по сходству, поддержку различных алгоритмов индексации, распределенные архитектуры, гибридный поиск (сочетание сходства векторов с фильтрацией по метаданным) и интеграцию с другими сервисами и базами данных.

С учетом быстрого развития ландшафта векторных хранилищ новые решения появляются регулярно. Используйте эту информацию как основу, но перед созданием вашей следующей системы RAG настоятельно рекомендуем обратиться к документации LangChain, чтобы изучить доступные варианты и выбрать тот, который лучше всего соответствует вашим потребностям на данный момент.

В следующем разделе мы подробно обсудим аспекты, которые следует учитывать при выборе векторного хранилища для вашей системы RAG.

Выбор векторного хранилища

Выбор подходящего векторного хранилища для системы RAG требует учета нескольких факторов, включая объем данных, требования к производительности поиска (скорость и точность) и сложность операций с векторами.

- **Масштабируемость** является ключевым аспектом для приложений, работающих с большими наборами данных, так как требуется механизм, который сможет эффективно управлять векторами и извлекать их из растущего корпуса данных.
- **Производительность** включает скорость поиска и способность базы данных возвращать высоко релевантные результаты.

Кроме того, важны легкость интеграции с существующими моделями RAG и гибкость, необходимая для поддержки различных операций с векторами. Разработчики должны искать векторные хранилища с надежными API, обширной документацией и сильной поддержкой сообщества или вендора.

Как упоминалось ранее, существует множество популярных векторных хранилищ, каждое из которых предлагает уникальные функции и оптимизации, подходящие для различных сценариев использования и требований производительности.

Ключевые аспекты при выборе векторного хранилища:
Совместимость с существующей инфраструктурой. При оценке векторных хранилищ важно учитывать, насколько хорошо они интегрируются с вашей существующей инфраструктурой данных. Оцените совместимость векторного хранилища с вашей текущей технологической средой и навыками вашей команды. Например, если ваша команда имеет значительный опыт работы с PostgreSQL, расширение для векторов, такое как **pgvector**, может быть подходящим выбором, поскольку оно позволяет бесшовно интегрироваться и использовать знания вашей команды.

Масштабируемость и производительность. Какое хранилище способно справиться с ожидаемым ростом данных и удовлетворить требования к производительности вашей системы RAG? Оцените возможности индексации и поиска в векторном хранилище, убедившись, что оно может обеспечить требуемый уровень производительности и точности.
Если вы планируете развертывание в крупном масштабе, распределенные базы данных, такие как **Milvus** или **Elasticsearch** с плагинами для работы с векторами, могут быть более подходящими, так как они предназначены для обработки больших объемов данных и обеспечения высокой пропускной способности поиска.

Простота использования и обслуживания. Какова кривая обучения при работе с векторным хранилищем, учитывая доступную документацию, поддержку сообщества и вендора?
Оцените усилия, необходимые для настройки, конфигурации и постоянного обслуживания хранилища. Полностью управляемые сервисы, такие как **Pinecone**, упрощают развертывание и управление, снижая операционную нагрузку на вашу команду. С другой стороны, решения с собственным размещением, такие как **Weaviate**, обеспечивают больший контроль и гибкость, позволяя настраивать и адаптировать хранилище для удовлетворения ваших специфических требований.

Безопасность данных и соответствие требованиям. Оцените функции безопасности и контроля доступа, предоставляемые векторным хранилищем, чтобы убедиться, что они соответствуют требованиям вашей отрасли. Если вы работаете с конфиденциальными данными, проанализируйте возможности хранилища по шифрованию и защите данных. Убедитесь, что хранилище соответствует требованиям нормативных актов и стандартов конфиденциальности, таких как GDPR или HIPAA, в зависимости от ваших нужд.

Стоимость и лицензирование. Какова модель ценообразования векторного хранилища? Основана ли она на объеме данных, операциях поиска или комбинации факторов? Оцените долгосрочную экономическую эффективность хранилища, принимая во внимание масштабируемость и прогнозируемый рост вашей системы RAG. Учитывайте лицензионные сборы, затраты на инфраструктуру и эксплуатационные расходы. Решения с открытым исходным кодом могут иметь более низкие первоначальные затраты, но требуют больше внутренних ресурсов и экспертизы для обслуживания. Управляемые сервисы, напротив, могут иметь более высокую абонентскую плату, но упрощают управление и предоставляют поддержку.

Экосистема и интеграции. При выборе векторного хранилища важно учитывать поддерживаемую экосистему и интеграции. Обратите внимание на наличие клиентских библиотек, SDK и API для различных языков программирования, так как это значительно упростит процесс разработки и обеспечит бесшовную интеграцию с существующим кодом. Оцените совместимость хранилища с другими инструментами и фреймворками, часто используемыми в системах RAG, такими как библиотеки NLP или фреймворки машинного обучения. Размер поддерживающего сообщества также имеет значение: убедитесь, что сообщество достаточно велико, чтобы развиваться и процветать. Векторное хранилище с развитой экосистемой и обширными интеграциями обеспечит большую гибкость и возможности для расширения функциональности вашей системы RAG.

Тщательно оценивайте эти факторы и сопоставляйте их с вашими конкретными требованиями, чтобы принять обоснованное решение при выборе векторного хранилища для вашей системы RAG. Проведите всестороннее исследование, сравните различные варианты и учтите долгосрочные последствия вашего выбора с точки зрения масштабируемости, производительности и поддержки.

Помните, что выбор векторного хранилища — это не универсальное решение, и оно может меняться по мере роста вашей системы RAG и изменения ваших требований. Регулярно пересматривайте выбор хранилища и вносите корректировки, чтобы обеспечить оптимальную производительность и соответствие архитектуре вашей системы.