大模型 RAG 应用开发基础及入门
本章介绍

大模型出现幻觉及缓解方案

- 了解大语言模型出现幻觉的原因，涵盖数据层面、模型层面的相关解释，以及幻觉的评估基准。

- 了解常见的大语言模型幻觉缓解方案，涵盖了调整模型参数、外挂知识、多智能体互动、加入诚实样本等。

- 学习什么是 RAG 以及基础架构，并尝试在 ChatGPT 上手动模拟 RAG 的运行流程加深理解。

AI 应用开发利器——向量数据库

- 学习了解什么是向量数据库，以及使用场景，为什么需要向量数据库，与传统数据库之间的差异。
- 了解什么是词向量，在 LLM 应用开发中的用途，掌握几种常见的词向量相似度计算方法，了解向量数据库的索引和搜索，底层如何加速相似度搜索过程。
- 掌握各类向量数据库的配置部署与使用，涵盖商用类型、云端数据库、开源本地部署等。

文本嵌入模型与 RAG 应用初步开发

- 了解什么是文本嵌入模型，掌握利用文本嵌入模型提取非结构化数据的特征。
- 了解几种不同类型的文本嵌入模型使用技巧，不同维度的词向量使用的场景与优缺点。
- 利用 LangChain 基于向量数据库构建第一个 RAG 应用，实现一个基于知识库问答的聊天机器人。

大语言模型出现幻觉的原因及缓解方案

大语言模型出现幻觉的原因及缓解方案

01. 幻觉的定义及出现的原因

1.1 幻觉的定义

大语言模型在处理自然语言时，有时会出现幻觉，表现为回答不准确或前后不一致的问题。这些幻觉可以分为两类：

1. **事实性幻觉**：指模型生成的内容与可验证的现实事实不一致。比如提问"第一个在月球上行走的人是谁？"，模型回复"Charles Lindbergh在1951年月球先驱任务中第一个登上月球"，而实际上，第一个登上月球的人是Neil Armstrong。而事实性幻觉又分为事实不一致（与现实世界信息相矛盾）和事实捏造（压根没有，无法根据现实信息验证）。
2. **忠实性幻觉**：指模型生成的内容与用户的指令或上下文不一致。比如让模型总结今年10月的新闻，结果模型却在说2006年10月的事。忠实性幻觉也可以细分，分为指令不一致（输出偏离用户指令）、上下文不一致（输出与上下文信息不符）、逻辑不一致（推理步骤以及与最终答案之间不一致）三类。

例如下方左侧为 `事实性幻觉`，右侧为 `忠实性幻觉`：

1.2 产生幻觉的原因

那么致使大模型产生幻觉的原因都有哪些？其实可以划分成三大来源：`数据源`、`训练过程` 和 `推理`。

1.2.1 数据源导致的幻觉

首先病从口入，大模型的粮食**数据**，是致使它产生幻觉的一大原因。这里面就包括 `数据缺陷` 和数据中捕获的事实知识的 `利用率` 低。

具体来说，数据缺陷分为错误信息和偏见（重复偏见、社会偏见），此外大模型也存在知识边界，所以存在领域知识缺陷和过时的事实知识。即便大模型吃掉了大量的数据，也会在利用时出现问题。

除此之外，大模型可能会过度依赖训练数据中的一些模式，如位置接近性、共现统计数据和相关文档计数，从而导致幻觉，比如：如果训练数据中频繁出现"加拿大"和"多伦多"，那么大模型可能会错误地将多伦多识别为加拿大的首都。

1.2.2 训练过程导致的幻觉

在模型的**预训练阶段**（大模型学习通用表示并获取世界知识）、**对齐阶段**（微调大模型使其更好地与人类偏好一致）两个阶段产生的问题也会导致幻觉的发生。

预训练阶段可能会存在：

- **架构缺陷**：基于前一个 token 预测下一个 token，这种单向建模阻碍了模型捕获复杂的上下文关系的能力；自注意力模块存在缺陷，随着 token 长度增加，不同位置的注意力被稀释。
- **暴露偏差**：训练策略也有缺陷，模型推理时依赖于自己生成的 token 进行后续预测，模型生成的错误 token 会在整个后续 token 中产生级联错误。

对齐阶段可能会存在：

- **能力错位**：大模型内在能力与标注数据中描述的功能之间可能存在错位。当对齐数据需求超出这些预定义的能力边界时，大模型会被训练来生成超出其自身知识边界的内容，从而放大幻觉的风险。
- **信念错位**：基于 RLHF 等的微调，使大模型的输出更符合人类偏好，但有时模型会倾向于迎合人类偏好，从而牺牲信息真实性。

1.2.3 推理导致的幻觉

大模型产生幻觉的第三个关键因素是**推理**，存在两个问题：

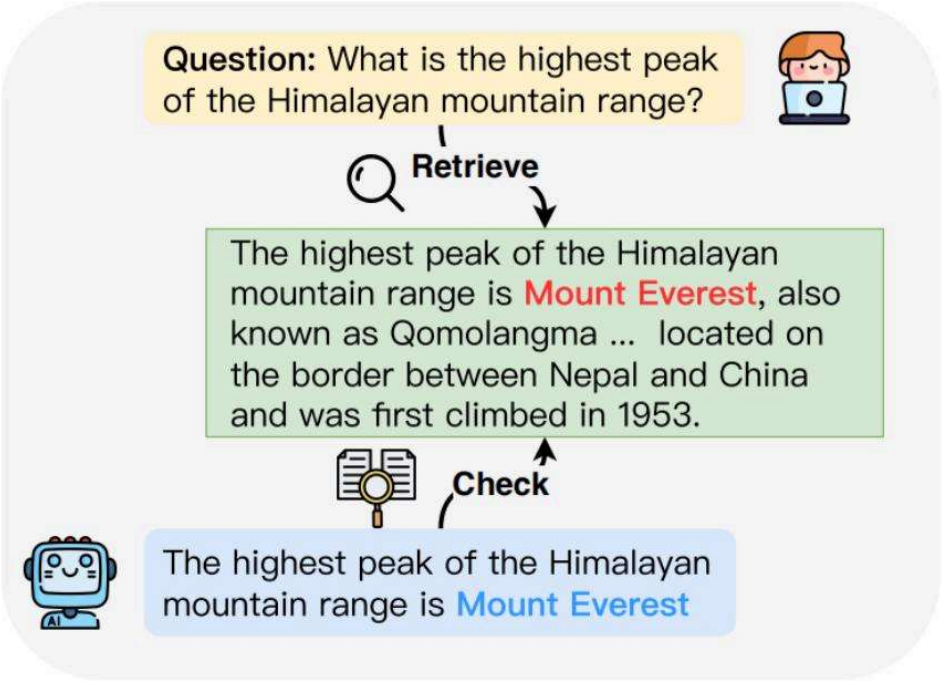
1. 固有的抽样随机性：在生成内容时根据概率随机生成。
2. 不完美的解码表示：上下文关注不足（过度关注相邻文本而忽视了源上下文）和 softmax 瓶颈（输出概率分布的表达能力受限）。

02. 大模型幻觉的评估

目前对于大模型幻觉的评估，对于 `事实性幻觉` 和 `忠实性幻觉` 有不同的评估方法。

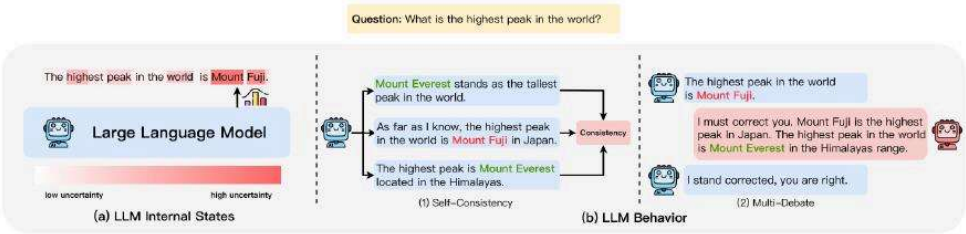
其中 `事实性幻觉`，有 `检索外部事实` 和 `不确定性估计` 两种方法。

`检索外部事实` 是将模型生成的内容与可靠的知识来源进行比较，例如同一个问题利用大语言模型生成内容，同时和外部检索到真实的信息进行对比校验。



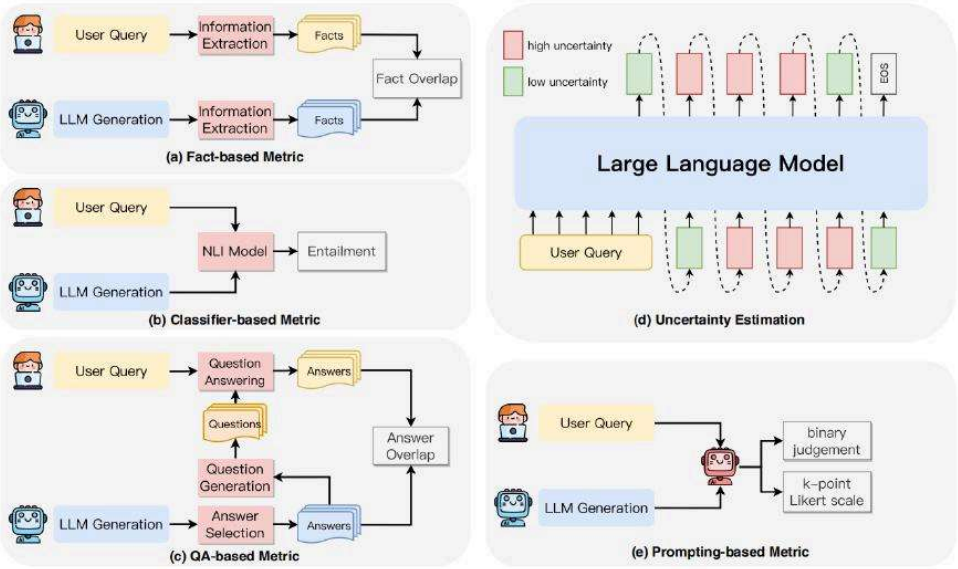
而 不确定性估计 的幻觉检测方法有两类：基于**内部状态**的方法和基于**行为**的方法。

1. 基于内部状态的方法主要依赖于大模型的内部状态。例如，通过考虑关键概念的最小标记概率来确定模型的不确定性。
2. 基于行为的方法主要依赖观察大模型的行为，不需要访问其内部状态。例如，通过采样多个响应并评估事实陈述的一致性来检测幻觉。



而检测 忠实性幻觉 也有 5 种主流方法：

1. **基于事实的度量**：测量生成内容和源内容之间事实的重叠程度来评估忠实性。
2. **分类器度量**：使用训练过的分类器来区分模型生成的忠实内容和幻觉内容。
3. **问答度量**：使用问答系统来验证源内容和生成内容之间的信息一致性。
4. **不确定度估计**：测量模型对其生成输出的置信度来评估忠实性。
5. **提示度量**：让大模型作为评估者，通过特定的提示策略来评估生成内容的忠实性。



03. 大语言模型幻觉的缓解方案

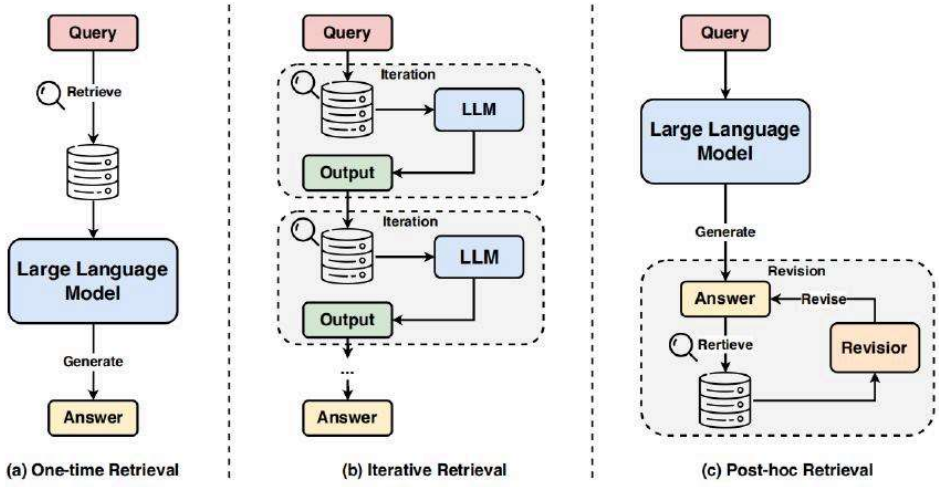
幻觉和创造/创新/涌现其实只有一线之隔，大模型如果没有幻觉，那就永远无法产生新内容。所以，从涌现/创新的角度来说，大模型的幻觉永远不会被解决，在某些场合下只可能被缓解。

大语言模型幻觉的缓解方案也有**数据、预训练、对齐、推理**相关的方案，与 LLM 应用开发相关的主要在数据方面的处理。

3.1 缓解数据相关幻觉(重点)

减少错误信息和偏见，最直观的方法是收集高质量的事实数据，并进行数据清理以消除偏见。对于大语言模型知识边界的问题，有两种流行方法。一种是**知识编辑**，直接编辑模型参数弥合知识差距。另一种通过**检索增强生成（RAG）**利用非参数知识源。

检索增强生成具体分为三种类型：**一次性检索**、**迭代检索**和**事后检索**。



一次性检索是将从单次检索中获得的外部知识直接预置到大模型的提示中；迭代检索允许在整个生成过程中不断收集知识；事后检索是基于检索的修订来完善大模型输出。

3.2 缓解预训练相关幻觉

1. **改进模型架构**：使用双向自回归模型和注意力锐化技术，增强模型对上下文的理解。
2. **优化训练目标**：通过引入事实性增强的训练方法和上下文预训练，提升模型的事实关联和逻辑一致性。
3. **减少曝光偏差**：采用新的监督信号和解码策略，减少训练与推理过程中的幻觉。

3.3 缓解对齐相关幻觉

1. **减少能力错位**：通过改进人类偏好判断，确保模型生成内容在其知识范围内。
2. **减少信念错位**：聚合人类偏好和调整模型内部激活，减少模型迎合行为，避免生成与模型自身认知相悖的内容。

3.4 缓解推理相关幻觉

1. **增强事实性解码**：动态调整解码策略，利用模型内部结构引导事实性回答。
2. **增强忠实度解码**：通过上下文和逻辑一致性策略，确保模型输出与用户指令或上下文保持一致。

检索增强生成 RAG 基础架构与手动模拟

检索增强生成 RAG 基础架构与手动模拟

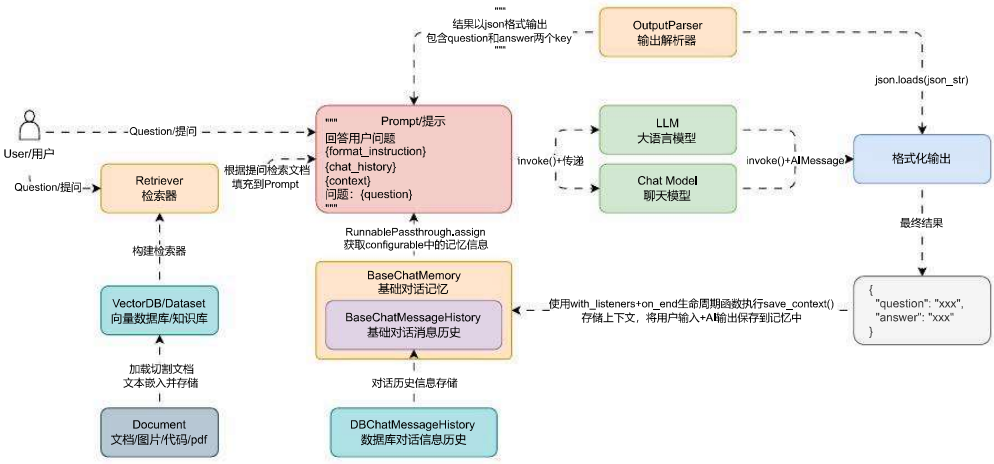
01. 检索增强生成 RAG

1.1 什么是 RAG?

检索增强生成（RAG）是指对大型语言模型输出进行优化，使其能够在生成响应之前引用训练数据来源之外的权威知识库。大型语言模型（LLM）用海量数据进行训练，使用数十亿个参数为回答问题、翻译语言和完成句子等任务生成原始输出。在 LLM 本就强大的功能基础上，RAG 将其扩展为能访问特定领域或组织的内部知识库，所有都无需重新训练模型。是一种经济高效地改进 LLM 输出的方法，让它在各种情境下都能保持相关性、准确性和实用性。

简单理解：**RAG 就是从外部先检索对应的知识内容，和用户的提问一起构成 Prompt，再让 LLM 生成内容。**

如果为前面开发的聊天机器人架构添加上 RAG 模块，更新后的运行流程如下：



1.2 RAG 的重要性及优点

我们可以将 LLM 看成是一个过于热情的员工，而且**这个员工拒绝了解任何时事，但是他总是会很自信地回答每一个问题**，更不幸的是这个员工**回答态度非常好，内容非常流畅，一般情况下还很难看出是真是假**！所以单纯利用 LLM 进行开发，存在非常大的缺陷：

1. LLM 的训练数据是静态的，这意味着 LLM 掌握的知识是有时间限制的，对于新知识不了解。
2. 当用户需要特定或者即时的数据时，LLM 往往提供通用或者过时的数据。
3. LLM 回答的内容可能是从非权威来源创建响应。
4. 由于术语混淆，不同的培训来源使用相同的术语来谈论不同的事情，因此会产生不确定的响应。

对比其他解决 LLM 幻觉的方案，RAG 带来的好处也非常明显：

1. **经济高效**：预训练和微调模型的成本很高，相比之下，RAG 是一种经济高效将新输入引入 LLM 的方案。
2. **信息即时**：使用 RAG 可以为 LLM 提供最新的研究、统计数据或新闻，确保数据的即时性。
3. **增强用户信任度**：RAG 允许 LLM 通过来源归属来呈现准确的信息。输出可以包括对来源的引文或引用。如果需要进一步说明或更详细的信息，用户也可以自己查找源文档。这可以增加对您的生成式人工智能解决方案的信任和信心。

4. **开发人员拥有更多控制权**：借助 RAG，开发人员可以更高效地测试和改进他们的聊天应用程序。他们可以控制和更改 LLM 的信息来源，以适应不断变化的需求或跨职能使用。开发人员还可以将敏感信息的检索限制在不同的授权级别内，并确保 LLM 生成适当的响应。此外，如果 LLM 针对特定问题引用了错误的信息来源，他们还可以进行故障排除并进行修复。组织可以更自信地为更广泛的应用程序实施生成式人工智能技术。

02. ChatGPT 手动模拟

人类与大语言模型的主要交接方式就是通过 Prompt，所以通过 Playground/ChatGPT 手动模拟 RAG 的过程其实也非常简单，使用用户的提问 `query` 进行搜索，得到搜索相关的内容，将搜索的内容与预设的 Prompt 模板、用户的 query 拼接成最终提示词，传递给大语言模型即可模拟最基础的 RAG 运行流程。

例如用户提问："**公司有销售什么产品么？**"，会触发一下流程：

① 调用 `检索器` 并传递 `公司有销售什么产品么？` 作为搜索语句进行检索得到对应文档，将这些文档整理合并得到对应的文本，输出：

```
1. 潮汕手工牛肉丸
产品名称：潮汕手工牛肉丸
电商网址：shop.example.com/beefballs
产品描述：潮汕手工牛肉丸选用优质牛肉，纯手工捶打制作，口感Q弹有嚼劲。全程无添加防腐剂和人工色素，确保天然健康，适合家庭火锅、煮汤等多种烹饪方式。
原材料：优质牛肉、生姜、盐、胡椒粉
制作工艺：传统手工捶打
口感：Q弹鲜美，肉质紧实
净重：500克/袋、1000克/袋
保质期：6个月（冷冻保存）
发货方式：顺丰冷链配送，确保新鲜
物流信息：24小时内发货，预计2-3天到货
推荐菜系：
- 牛肉丸火锅：搭配蔬菜、菌类，煮至牛肉丸浮起即可享用。
- 牛肉丸煮汤：与青菜、萝卜等食材同煮，营养丰富。
价格:500克：68元/袋、1000克：128元/袋

2. 潮汕猪肉卷
产品名称：潮汕猪肉卷
电商网址：shop.example.com/porkroll
产品描述：潮汕猪肉卷采用猪后腿肉为主要原料，配以特制香料腌制，手工卷制而成。口感鲜嫩多汁，香味四溢，是潮汕传统名菜之一。
原材料：猪后腿肉、香料、盐、糖
制作工艺：精细切割，手工卷制
口感：鲜嫩多汁，咸香可口
净重：400克/袋、800克/袋
保质期：3个月（冷冻保存）
发货方式：顺丰冷链配送，确保新鲜
物流信息：24小时内发货，预计2-3天到货
推荐菜系：
- 猪肉卷煎烤：切片后煎至金黄，外脆里嫩。
- 猪肉卷炖煮：切块后与蔬菜同炖，风味更佳。
价格:400克：58元/袋、800克：108元/袋

3. 潮汕三宝（酱油、甜醋、虾酱）
产品名称：潮汕三宝
电商网址：shop.example.com/chaoshanthree
```

产品描述：潮汕三宝包含酱油、甜醋和虾酱。酱油由大豆、麦子自然发酵而成，甜醋以糯米酿制，虾酱选用新鲜海虾发酵，是潮汕菜肴必备调味品。
酱油：大豆、麦子自然发酵，500ml/瓶
甜醋：糯米酿制，500ml/瓶
虾酱：新鲜海虾发酵，200克/瓶
保质期：酱油和甜醋12个月，虾酱6个月
发货方式：顺丰配送，确保完好
物流信息：24小时内发货，预计2-3天到货
推荐菜系：
- 酱油：适合调味、蘸料、炒菜。
- 甜醋：用于凉拌菜、蘸料。
- 虾酱：适合炒菜、做蘸料。
价格：128元/套（含酱油、甜醋、虾酱各一瓶）

4. 潮汕鸭母捻
产品名称：潮汕鸭母捻
电商网址：shop.example.com/duckegg
产品描述：潮汕鸭母捻是一种传统甜点，使用糯米粉制作，内馅有花生、芝麻、红豆等多种口味，外皮软糯，汤底清甜。
原材料：糯米粉、花生、芝麻、红豆、糖
制作工艺：手工包制
口感：软糯香甜，馅料丰富
净重：500克/袋
保质期：3个月（冷冻保存）
发货方式：顺丰冷链配送，确保新鲜
物流信息：24小时内发货，预计2-3天到货
推荐菜系：
- 甜汤：加入红糖水煮沸，香甜可口。
- 咸汤：搭配咸菜、肉片，别有风味。
价格：45元/袋

② 接下来将用户的输入 `query` 和检索得到的文档文本 `context` 合并到预设的提示模板中，如下：

```
你是一个由OpenAI开发的聊天机器人，善于根据上下文内容帮助用户解决问题，回复的内容尽可能简洁，如果需要用户提供额外的信息，请进行引导，如果不知道就说不知道。

<context>
{context}
<context>

用户的提问是：{query}
```

③ 将构建好的提示词传递给大语言模型，得到对应的输出内容如下：

```
公司销售以下产品：
1. 潮汕手工牛肉丸
2. 潮汕猪肉卷
3. 潮汕三宝（酱油、甜醋、虾酱）
4. 潮汕鸭母捻

每种产品都有详细的描述、价格和购买信息。
```

这样就可以完成一个手动 RAG 的过程模拟，实际在代码中，无论多么复杂的 RAG、无论如何进行 RAG 优化，**本质上都是执行外部检索，然后将外部检索的内容和用户原始提问合并成最终 Prompt，再向大语言模型发起提问，最终得到对应的内容。**

AI 应用开发新宠——向量数据库的介绍与用途

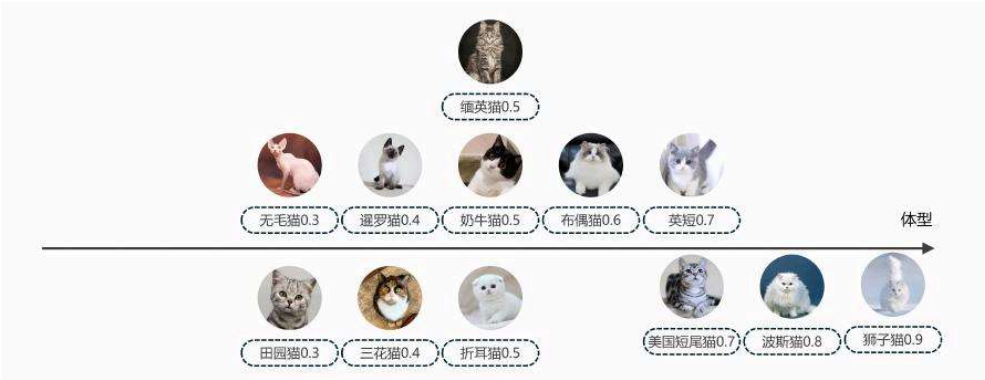
AI应用开发新宠——向量数据库的介绍与用途

01. “猫”与向量的关系与衍生

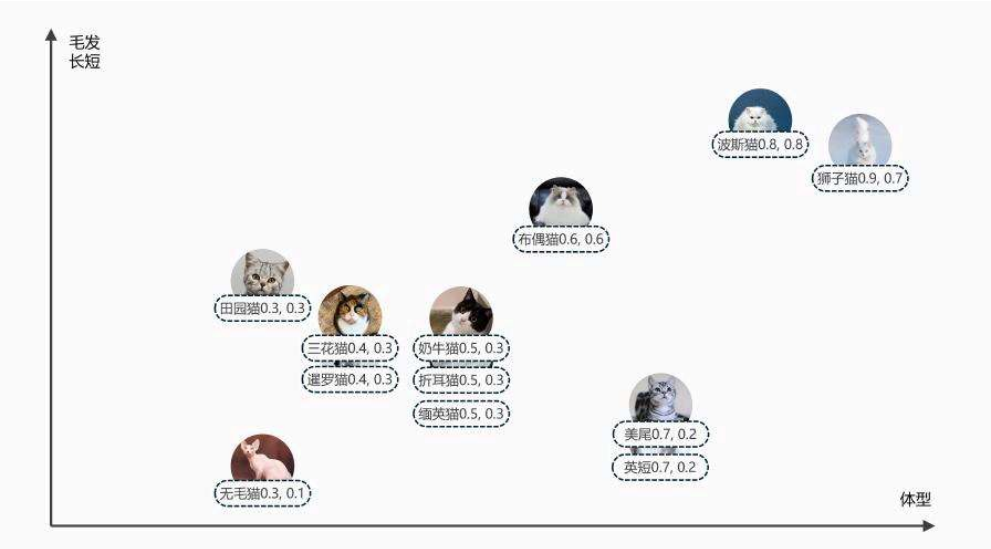
这里我们以 `猫` 引出 `词向量`，再引出 `向量数据库` 的概念，通过这一个小小的演示，大家就能快速掌握词向量与向量数据库。

平时有养猫或者熟悉猫的小伙伴对于下面这一张猫的品类图，很快就能区分出它们的品种，之所以能做到这点，是因为我们会从不同的角度来观察这些猫的特征，如下：

如果我们使用一个水平轴来表示 `体型大小` 这个特征，这些不同品种的猫将落在不同的坐标点上，这样就可以通过体型的大小区分出一些品种，如下：

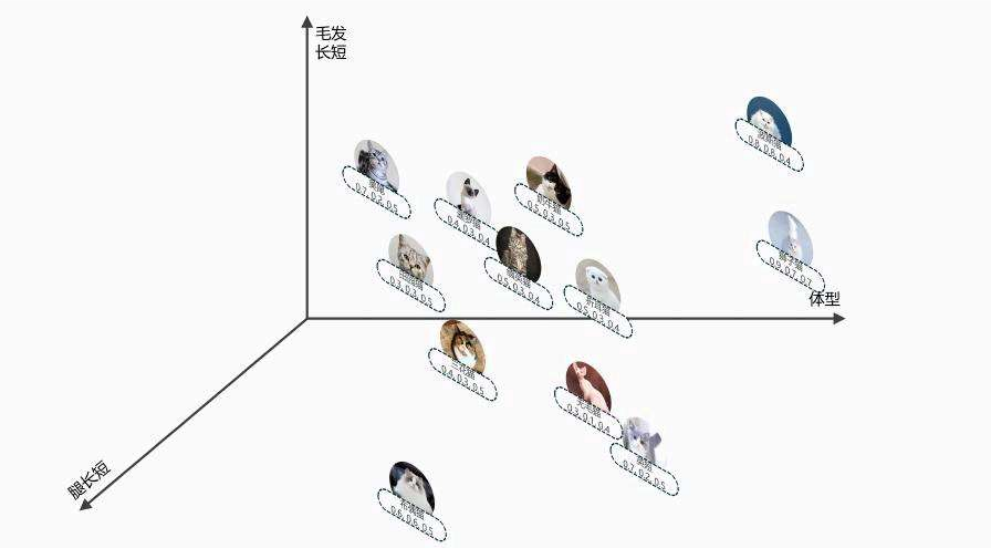


然而，如果仅仅靠体型一个特征，依旧会有很多品种的猫特征相近，比如 `缅因猫`、`奶牛猫` 和 `折耳猫` 就非常接近，所以我们继续添加多一个特征，比如毛发的长短，继续建立一个毛发的垂直轴，这样子就可以区分出更多品种。



现在每个品种的猫就可以表示为一个二维的坐标点，但是哪怕有两个特征，仍然会有一些品种无法区分。

所以我们需要从更多的角度来观察，例如 `腿的长短`，这个时候在建立一个 `腿的长短` 的 `z轴`，又有更多的品种被区分出来，现在每个品种的猫就可以表示为一个三维的坐标点，如下：



如果这个时候还想引入更多的特征进行区分，比如：`眼睛大小`、`尾巴长短`、`毛发颜色`、`声音大小`、`耳朵形状` 等等，虽然在坐标图上没法展示出来，但是我们却可以很轻松地将这些特征使用数值的方式展示出来，例如下方：

```
1. 暹罗猫：(0.4, 0.3, 0.4, 0.5, 0.3, 0.4, 0.5, ...)
2. 英国短毛猫：(0.7, 0.2, 0.5, 0.5, 0.5, 0.5, 0.5, ...)
3. 缅甸猫：(0.5, 0.3, 0.4, 0.5, 0.3, 0.4, 0.5, ...)
4. 波斯猫：(0.8, 0.8, 0.4, 0.8, 0.7, 0.4, 0.6, ...)
5. 布偶猫：(0.7, 0.6, 0.5, 0.8, 0.5, 0.4, 0.5, ...)
6. 无毛猫：(0.6, 0.1, 0.4, 0.5, 0.3, 0.4, 0.5, ...)
7. 中华田园猫：(0.5, 0.3, 0.5, 0.5, 0.5, 0.4, 0.5, ...)
8. 折耳猫：(0.6, 0.3, 0.4, 0.5, 0.3, 0.4, 0.7, ...)
9. 三花猫：(0.5, 0.3, 0.5, 0.5, 0.5, 0.5, 0.5, ...)
10. 美国短毛猫：(0.7, 0.2, 0.5, 0.5, 0.5, 0.5, 0.5, ...)
11. 狮子猫：(0.9, 0.7, 0.7, 0.8, 0.7, 0.8, 0.5, ...)
12. 奶牛猫：(0.6, 0.3, 0.5, 0.5, 0.5, 0.4, 0.5, ...)
```

当记录的**特征足够大，维度足够大时，区分的程度也会越高**，当看到一只猫，只需要将它转换成对应的`多维坐标数据/向量`，就可以很轻松地找到这只猫的分类归属，而且不仅仅是猫，几乎所有的事物都可以使用这一套方式进行表达。

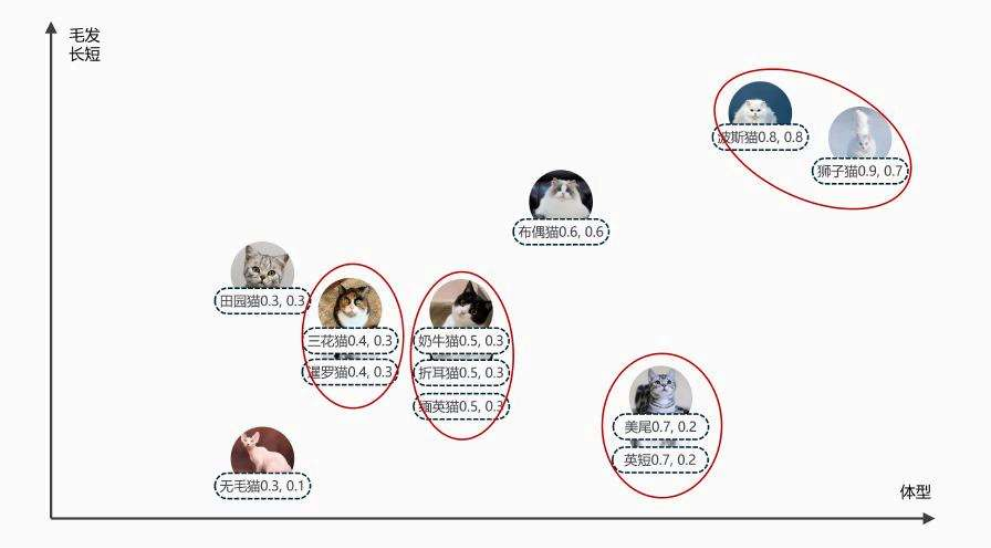
所以一个字、一个词、一句话、一篇文本、甚至一张图片都可以用这样一个`多维坐标数据`，亦或者说`向量`记录对应的特征，而将`文本`转换为记录特征的`向量`就可以被称为词向量。

02. 向量数据库概念与用途

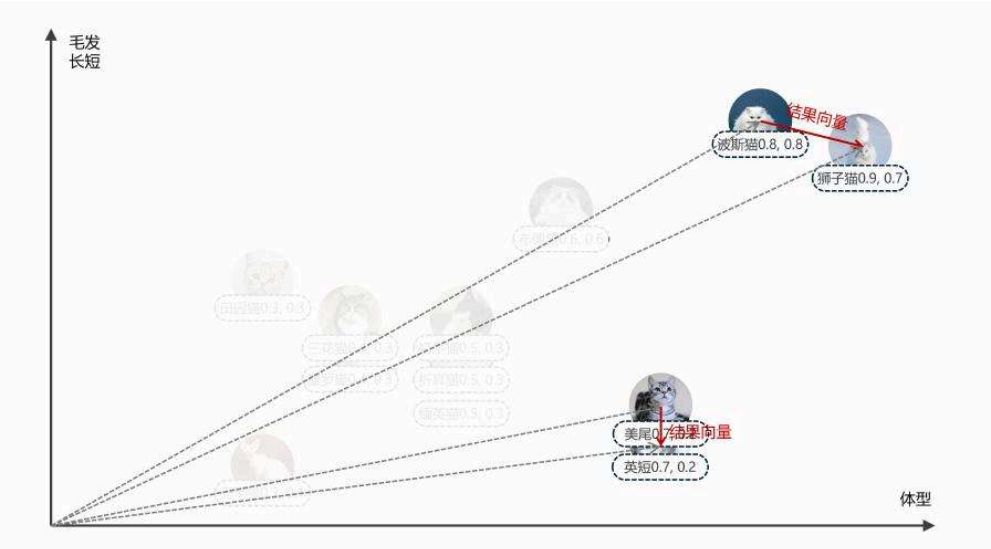
向量数据库就是一种专门用于存储和处理向量数据的数据库系统，传统的关系型数据库通常不擅长处理向量数据，因为它们需要将数据映射为结构化的表格形式，而向量数据的维度较高、结构复杂，导致存储和查询效率低下，所以向量数据库应运而生。

由于高维的向量我们在三维空间没法绘图，这里我们以二维向量的形式扩展到多维，来一起看下向量的神通广大之处！

例如下图，在二维坐标上，概念上更接近的点，在图表上也更聚集，而那些概念上不同的点则一般距离比较远。

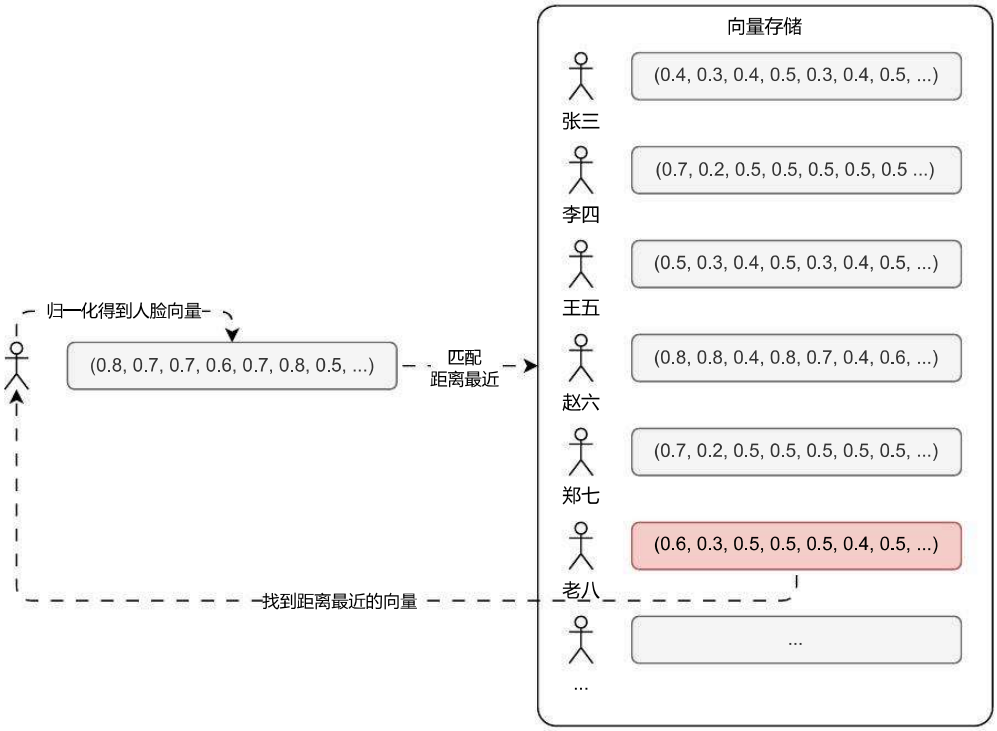


如果将两个点之间作差，得到一条新的向量，甚至我们可以使用这条`结果向量`来表示两个点之间的关联（越短表示关联越大），如下：



所以最常见的应用就是人脸识别，将不同的人脸归一化（固定大小）后生成对应的向量，然后将千千万万张人脸的向量进行存储。

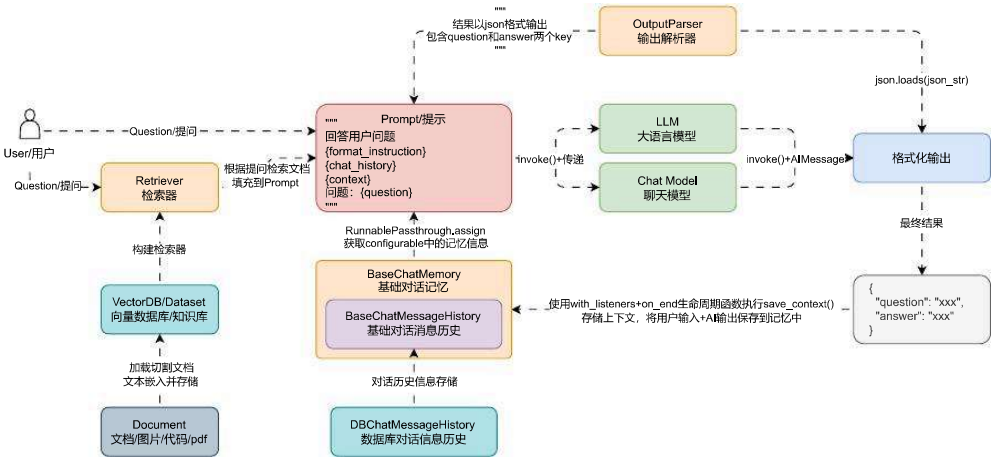
进行人脸识别时，只需要将这张人脸按照统一的标准归一化并转换成向量，接下来在向量数据库中搜索与这个向量最接近的向量，就可以巧妙实现人脸识别。



这也是向量数据库的最常使用场景：**人脸识别、图像搜索、音频识别、智能推荐系统**等。

而在 RAG 中，我们将对应的知识文档按照特定的规则拆分成合适的大小，再转换成向量存储到向量数据库中，当人类提问时，将人类提问 `query` 转换成向量并进行搜索，找到在特征上更接近的文本块，这些文本块就可以看成和 `query` 具有强关联或者说有因果关系。

这样就可以将这些 `文本块` 作为这次提问的额外补充知识，让 LLM 基于补充知识+提问生成对应的内容，从而实现知识库问答。


"""
结果以json格式输出
包含question和answer两个key
"""
OutputParser
输出解析器
json.loads(json_str)
User/用户
Question/提问
"""
Prompt/提示
回答用户问题
{format_instruction}
{chat_history}
{context}
问题：{question}
"""
LLM
大语言模型
invoke()+传递
invoke()+AIMessage
格式化输出
Question/提问
Retriever
检索器
根据提问检索文档
填充到Prompt
Chat Model
聊天模型
最终结果
构建检索器
RunnablePassthrough.assign
获取configurable中的记忆信息
{
"question": "xxx",
"answer": "xxx"
}
BaseChatMemory
基础对话记忆
使用with_listeners+on_end生命周期函数执行save_context()
存储上下文，将用户输入+AI输出保存到记忆中
VectorDB/Dataset
向量数据库/知识库
BaseChatMessageHistory
基础对话消息历史
加载切割文档
文本嵌入并存储
对话历史信息存储
Document
文档/图片/代码/pdf
DBChatMessageHistory
数据库对话信息历史

Embedding 嵌入模型介绍与使用

传统数据库与向量数据库的使用差异

传统数据库与向量数据库的使用差异

01. 两种数据库的差异

1.1 搜索方式差异

传统数据库，比如关系型数据库，擅长处理结构化数据，如存储在表格中的文本和数字等。它们通过预定义的查询语言（如SQL）来进行**精确匹配或条件搜索**。这种方式在处理银行交易、客户信息等数据时效果显著，但在处理复杂的模式识别问题时就显得力不从心。

例如，通过 `SELECT` + `WHERE` 可以精准查询到 id 为 `e0d13c78-870b-46df-b2f5-693ae9d5d727` 的用户。

```
SELECT * FROM account WHERE id='e0d13c78-870b-46df-b2f5-693ae9d5d727'
```

但是想通过 SQL 来查询和 `我喜欢打篮球游泳与编程` 这句话语义相近的内容，就无能力为了。

相比之下，向量数据库不是通过匹配确切的数值，而是通过一种称为 `相似性搜索` 的方法来工作。它们可以快速找到与查询向量最相似的数据点（目前绝大部分向量数据库都支持在 `相似性搜索` 的基础上添加筛选条件），即使这些数据点在数值上并不完全相同。

例如，在一个向量数据库中，即使没有完全相同的照片，我们仍然可以找到风格相似的图片。通过这种方式，向量数据库打破了传统数据库的局限，为处理和分析大规模、复杂的数据提供了更灵活和强大的解决方案。

1.2 数据处理与存储差异

传统数据库采用基于行的存储方式，传统数据库将数据存储为行记录，每一行包含多个字段，并且每个字段都有固定的列。传统数据库通常使用索引来提高查询性能，例如下方就是一个典型的传统数据库表格：

电影ID	电影名称	导演	类型	评分
1	阿凡达	詹姆斯·卡梅隆	科幻/冒险	8.8
2	飞屋环游记	皮特·道格特	动画/冒险	8.2
3	泰坦尼克号	詹姆斯·卡梅隆	爱情/剧情	7.8

这种方式在处理结构化数据时非常高效，但在处理非结构化或半结构化数据时效率低下。

向量数据库将数据以列形式存储，即每个列都有一个独立的存储空间，这使得向量数据库可以更加灵活地处理复杂的数据结构。向量数据库还可以进行列压缩（**稀疏矩阵**），以减少存储空间和提高数据的访问速度。

并且在向量数据库中，将数据表示为**高维向量**，其中**每个向量对应于数据点。这些向量之间的距离表示它们之间的相似性**。这种方式使得非结构化或半结构化数据的存储和检索变得更加高效。

以电影数据库为例，我们可以将每部电影表示为一个特征向量。假设我们使用四个特征来描述每部电影：**动作、冒险、爱情、科幻**。每个特征都可以在0到1的范围内进行标准化，表示该电影在该特征上的强度。

例如，电影"阿凡达"的向量表示可以是 `[0.9, 0.8, 0.2, 0.9]`，其中数字分别表示动作、冒险、爱情、科幻的特征强度。其他电影也可以用类似的方式表示。这些向量可以存储在向量数据库中，如下所示：

电影ID	特征向量
1	[0.9, 0.8, 0.2, 0.9]
2	[0.7, 0.9, 0.1, 0.2]
3	[0.2, 0.1, 0.9, 0.1]
...	...

现在，如果我们想要查找与电影"阿凡达"相似的电影，我们可以计算向量之间的距离，找到最接近的向量，从而实现相似性匹配，而无需复杂的SQL查询。这就像使用地图找到两个地点之间的最短路径一样简单。

1.3 优缺点横向对比

尽管向量数据库在处理高维数据和实现快速检索方面有着显著优势，但它并不是一种“一刀切”的解决方案。在某些应用场景中，其他类型的数据库可能更合适，而且向量数据库与传统关系数据库协同发展、相互补充。

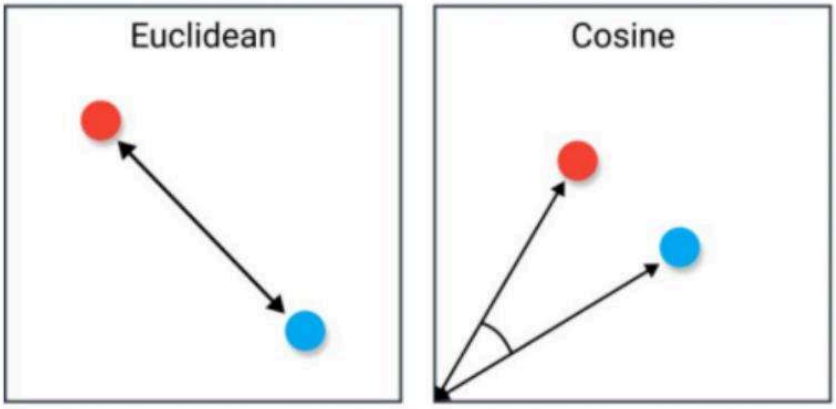
	传统的关系型数据库	向量数据库
数据类型	数值、字符串、时间等传统数据类型	新的数据类型：向量数据 不存储原始数据（有的也支持）
数据规模	小，1亿条数据对于关系型数据库来说规模很大	大，最少千亿数据是底线
数据组织方式	基于表格，按照行和列组织	基于向量，按照向量维度组织
查找方式	精确查找：点查/范围查 查询结果要么符合条件要么不符合条件	近似查找 查询结果是与输入条件最相似的，近似计算能力要就较高
低延时，高并发	否	是
上层应用	较弱	对外提供统一的API，更适合大规模AI应用程序的部署和使用
下游应用场景	央企、国企。央企国企工作因工作内容要求 容错率低，传统数据库能够提供更为准确 的搜索结果。	互联网公司。向量数据库的结果正确率相对较低， 成本低，互联网公司的场景容错率较高， 能够包容这一缺陷。

02. 相似性搜索算法

2.1 余弦相似度与欧式距离

在向量数据库中，支持通过多种方式来计算两个向量的相似度，例如：**余弦相似度、欧式距离、曼哈顿距离、闵可夫斯基距离、汉明距离、Jaccard相似度**等多种。其中最常见的就是 `余弦相似度` 和 `欧式距离`。

例如下图，左侧就是 `欧式距离`，右侧就是 `余弦相似度`。



① 余弦相似度主要用于衡量向量在方向上的相似性，特别适用于文本、图像和高维空间中的向量。它不受向量长度的影响，只考虑方向的相似程度，余弦相似度的计算公式如下（计算两个向量夹角的余弦值，取值范围为[-1, 1]）：

$$\text{similarity}(\mathbf{A}, \mathbf{B}) = \frac{\mathbf{A} \cdot \mathbf{B}}{\|\mathbf{A}\|\|\mathbf{B}\|} = \frac{\sum_{i=1}^{n} A_i \cdot B_i}{\sqrt{\sum_{i=1}^{n} A_i^2} \cdot \sqrt{\sum_{i=1}^{n} B_i^2}}$$

② 欧式距离衡量向量之间的直线距离，得到的值可能很大，最小为 0，通常用于低维空间或需要考虑向量各个维度之间差异的情况。欧氏距离较小的向量被认为更相似，欧式距离的计算公式如下：

$$\text{distance}(\mathbf{A}, \mathbf{B}) = \sqrt{\sum_{i=1}^{n} (A_i - B_i)^2}$$

2.2 相似性搜索加速计算

在向量数据库中，数据按列进行存储，通常会将多个向量组织成一个 M×N 的矩阵，其中 M 是向量的维度（特征数），N 是向量的数量（数据库中的条目数），这个矩阵可以是稠密或者稀疏的，取决于向量的稀疏性和具体的存储优化策略。

这样计算相似性搜索时，本质上就变成了向量与 M×N 矩阵的每一行进行相似度计算，这里可以用到大量成熟的加速算法：

1. **矩阵分解方法**：
    - **SVD（奇异值分解）**：可以通过奇异值分解将原始矩阵转换为更低秩的矩阵表示，从而减少计算量。
    - **PCA（主成分分析）**：类似地，可以通过主成分分析将高维矩阵映射到低维空间，减少计算复杂度。

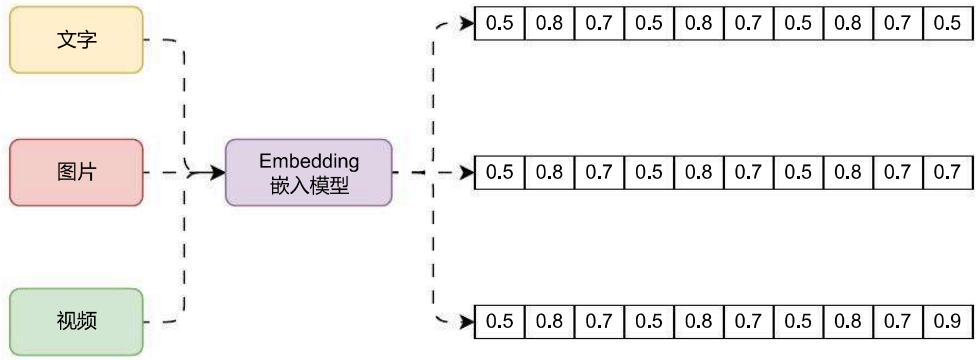
2. **索引结构和近似算法**：
    - **LSH（局部敏感哈希）**：LSH 可以在近似相似度匹配中加速计算，特别适用于高维稀疏向量的情况。
    - **ANN（近似最近邻）算法**：ANN 算法如KD-Tree、Ball-Tree等可以用来加速对最近邻搜索的计算，虽然主要用于向量空间，但也可以部分应用于相似度计算中。
3. **GPU 加速**：使用图形处理单元（GPU）进行并行计算可以显著提高相似度计算的速度，尤其是对于大规模数据和高维度向量。
4. **分布式计算**：由于行与行之间独立，所以可以很便捷地支持分布式计算每行与向量的相似度，从而加速整体计算过程。

向量数据库底层除了在算法层面上针对相似性搜索做了大量优化，在存储结构、索引机制等方面均做了大量的优化，这才使得向量数据库在处理高维数据和实现快速相似性搜索上展示出巨大的优势。

Embedding 嵌入模型介绍与使用

01. 什么是 Embedding?

要想使用向量数据库的相似性搜索，存储的数据必须是向量，那么如何将高维度的文字、图片、视频等非结构化数据转换成向量呢？这个时候就需要使用到 Embedding 嵌入模型了，例如下方就是 Embedding 嵌入模型的运行流程：



Embedding 模型是一种在机器学习和自然语言处理中广泛应用的技术，它旨在将高纬度的数据（如文字、图片、视频）映射到低纬度的空间。Embedding 向量是一个 N 维的实值向量，它将输入的数据表示成一个连续的数值空间中的点。这种嵌入可以是一个词、一个类别特征（如商品、电影、物品等）或时间序列特征等。

而且通过学习，**Embedding 向量可以更准确地表示对应特征的内在含义，使几何距离相近的向量对应的物体有相近的含义**，甚至对向量进行加减乘除算法都有意义！

一句话理解 Embedding：**一种模型生成方法，可以将非结构化的数据，例如文本/图片/视频等数据映射成有意义的向量数据。**

目前生成 embedding 方法的模型有以下 4 类：

1. **Word2Vec（词嵌入模型）**：这个模型通过学习将单词转化为连续的向量表示，以便计算机更好地理解和处理文本。Word2Vec 模型基于两种主要算法 `CBOW` 和 `Skip-gram`。
2. **Glove**：一种用于自然语言处理的词嵌入模型，它与其他常见的词嵌入模型（如 Word2Vec 和 FastText）类似，可以将单词转化为连续的向量表示。GloVe 模型的原理是通过观察单词在语料库中的共现关系，学习得到单词之间的语义关系。具体来说，GloVe 模型将共现概率矩阵表示为两个词向量之间的点积和偏差的关系，然后通过迭代优化来训练得到最佳的词向量表示。

   GloVe 模型的优点是它能够在大规模语料库上进行有损压缩，得到较小维度的词向量，同时保持了单词之间的语义关系。这些词向量可以被用于多种自然语言处理任务，如词义相似度计算、情感分析、文本分类等。
3. **FastText**：一种基于词袋模型的词嵌入技术，与其他常见的词嵌入模型（如 Word2Vec 和 GloVe）不同之处在于，FastText考虑了单词的子词信息。其核心思想是将单词视为字符的 n-grams 的集合，在训练过程中，模型会同时学习单词级别和n-gram级别的表示。这样可以捕捉到单词内部的细粒度信息，从而更好地处理各种形态和变体的单词。
4. **大模型 Embeddings（重点）**：和大模型相关的嵌入模型，如 OpenAI 官方发布的第二代模型：text-embedding-ada-002。它最长的输入是 8191 个tokens，输出的维度是 1536。

02. Embedding 带来的价值

1. **降维**：在许多实际问题中，原始数据的维度往往非常高。例如，在自然语言处理中，如果使用 Token 词表编码来表示词汇，其维度等于词汇表的大小，可能达到数十万甚至更高。通过 Embedding，我们可以将这些高维数据映射到一个低维空间，大大减少了模型的复杂度。
2. **捕捉语义信息**：Embedding 不仅仅是降维，更重要的是，它能够捕捉到数据的语义信息。例如，在词嵌入中，语义上相近的词在向量空间中也会相近。这意味着Embedding可以保留并利用原始数据的一些重要信息。
3. **适应性**： 与一些传统的特征提取方法相比，Embedding 是通过数据驱动的方式学习的。这意味着它能够自动适应数据的特性，而无需人工设计特征。
4. **泛化能力**：在实际问题中，我们经常需要处理一些在训练数据中没有出现过的数据。由于Embedding能够捕捉到数据的一些内在规律，因此对于这些未见过的数据，Embedding仍然能够给出合理的表示。
5. **可解释性**：尽管 Embedding 是高维的，但我们可以通过一些可视化工具（如t-SNE）来观察和理解 Embedding 的结构。这对于理解模型的行为，以及发现数据的一些潜在规律是非常有用的。

03. Embedding 的运算法则

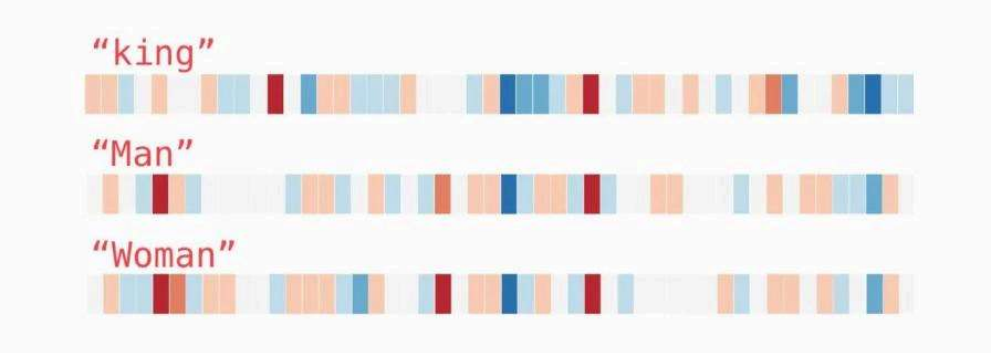
通过对上文的理解，我们继续看看训练好的词向量实例（也被称为词嵌入），并探索他们的一些有趣属性（一个当成是玩笑的例子，主要用于帮助大家更好去理解 Embedding，该例子在其他 Embedding 模型下不一定能复现）。

这是一个"king"词的嵌入（使用 GloVe 方法生成的向量）：

```
[0.50451, 0.68607, -0.59517, -0.022801, 0.60046, -0.13498, -0.08813, 0.47377,
-0.61798, -0.31012, -0.076666, 1.493, -0.034189, -0.98173, 0.68229, 0.81722,
-0.51874, -0.31503, -0.55809, 0.66421, 0.1961, -0.13495, -0.11476, -0.30344,
0.41177, -2.223, -1.0756, -1.0783, -0.34354, 0.33505, 1.9927, -0.04234, -0.64319,
0.71125, 0.49159, 0.16754, 0.34344, -0.25663, -0.8523, 0.1661, 0.40102, 1.1685,
-1.0137, -0.21585, -0.15155, 0.78321, -0.91241, -1.6106, -0.64426, -0.51042]
```

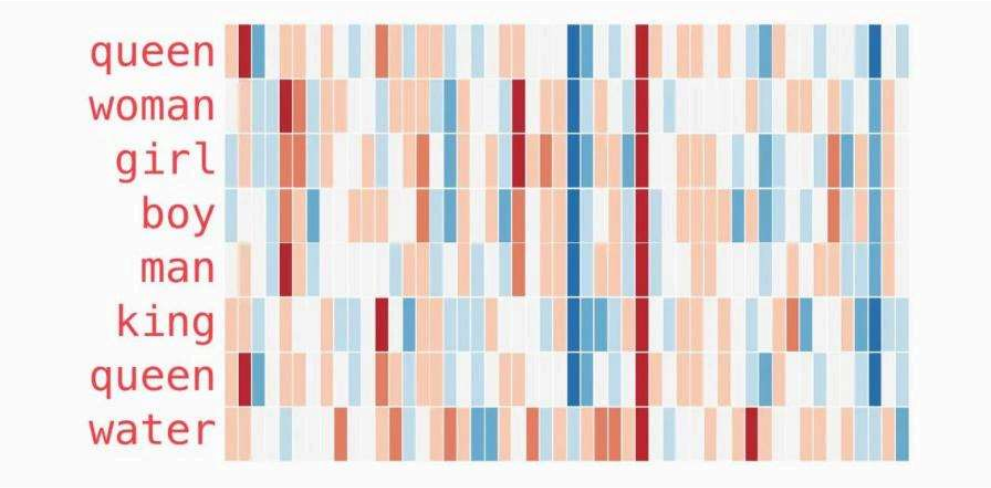
这是一个 50 维的向量，通过数字我们很难观察其规律，但是我们可以**将这个向量的每一个元素进行可视化颜色编码，由于范围是 [-2, 2]，越靠近 -2 则设置成蓝色，越靠近 2 则设置成红色**，那么这一串向量就可以表示成这样：

我们忽略数字仅查看颜色，并且将"King"与其他单词进行比较（King 国王、Man 男人、Woman 女人）：



从色块分布图中，可以很容易的看出 King、Man、Woman 这三组向量的结构其实非常接近，色块颜色非常集中，这是否能意味着国王、男人、女人这三个词之间是具有强关联甚至是子集的因果关系。

接下来看下另外一个示例列表，这个示例展示了更多单词的向量分布的规律：



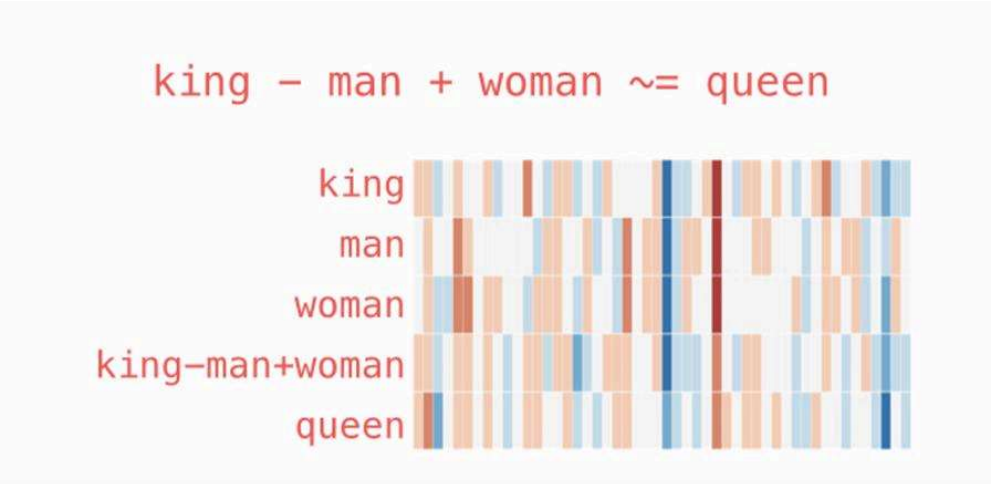
在这张分布图中也可以轻松看出一些信息：

1. 这几个单词的词向量在中间都有一条直的红色列，代表它们在这个维度上是相似的，但是从数值来看我们看不出这个维度是什么。
2. woman 和 girl 在很多地方都是相似的，man 和 boy 也一样。
3. girl 和 boy 也有一些彼此相似的地方，但这些和 woman/girl 不同，这些差异是否可以总结出“年龄”这个维度？
4. 上面的单词中，除了 water 其他的都和人相关，并且向量的倒数第三个元素是一个蓝色的色块，来到 water 这里却消失了。
5. king 和 queen 非常接近，但是存在一些比较明显的差异，这部分差异是否能体现出“性别”这个维度？

除了单个词的向量能展示出一些有意思的信息，对向量进行 `加减乘除` 也能得到一些有意思的结果。在 NLP 界有一个非常著名的公式：

```
Queen = King - Man + Woman
女王 = 国王 - 男人 + 女人
```

对这几个向量进行运算后，同样可视化出来，结果如下：



你会发现 `king-man+woman` 和 `queen` 的向量特征分布非常接近，也是 GloVe 集合中 40w 个字嵌入中最接近它的词。

这一个示例虽然不一定是一个正确的思路，但是从这个示例中仍然可以看到将对应的词转换成 Embedding 向量后，嵌入模型能够捕捉到数据的语义信息，并且语义上相近的词在向量空间中也会相近，这也为语义搜索提供了一个可能。

OpenAI Embedding 接口使用实践测试

OpenAI Embedding 接口使用实践测试

01. OpenAI Embedding 嵌入模型

OpenAI 服务提供商提供了线上的 Embeddings 服务，涵盖了 3 种模型，信息对比如下：

模型	1美元支持嵌入的文档数 每个文档 800 个 Token	性能评估	最大输入	向量维度
text-embedding-3-small	62,500	62.3%	8191	1536（支持修改）
text-embedding-3-large	9,615	64.6%	8191	3072（支持修改）
text-embedding-ada-002	12,500	61.0%	8191	1536

OpenAI 的 Embeddings 嵌入模型虽然是市面上效果最好的嵌入模型，虽然价格非常低廉，不过由于没有本地版本，而且接口响应速度相对较慢，在需要对大量文档进行嵌入时，效率非常低，所以国内使用得并不多（国内一般会使用对应的本地或者本地服务提供商的嵌入模型）。

OpenAI Embeddings 官网文档：https://platform.openai.com/docs/guides/embeddings

02. Embedding 组件使用

在 LangChain 中，Embeddings 类是一个专为与文本嵌入模型交互而设计的类，这个类为许多嵌入模型提供商（如 OpenAI、Cohere、Hugging Face 等）提供一个标准的接口。

LangChain 中 Embeddings 类提供了两种方法：

1. `embed_documents`：用于嵌入文档列表，传入一个文档列表，得到这个文档列表对应的向量列表。
2. `embed_query`：用于嵌入单个查询，传入一个字符串，得到这个字符串对应的向量。

并且 Embeddings 类并不是一个 Runnable 组件，所以并不能直接接入到 Runnable 序列链中，需要额外的 RunnableLambda 函数进行转换，核心基类代码也非常简单：

```python
class Embeddings(ABC):
    """Interface for embedding models."""

    @abstractmethod
    def embed_documents(self, texts: List[str]) -> List[List[float]]:
        """Embed search docs."""

    @abstractmethod
    def embed_query(self, text: str) -> List[float]:
        """Embed query text."""

    async def aembed_documents(self, texts: List[str]) -> List[List[float]]:
        """Asynchronous Embed search docs."""
        return await run_in_executor(None, self.embed_documents, texts)
```

```
    async def aembed_query(self, text: str) -> List[float]:
        """Asynchronous Embed query text."""
        return await run_in_executor(None, self.embed_query, text)
```

如果如果想对接自定义的 Embedding 嵌入模型，其实非常简单，只需实现 `embed_documents` 和 `embed_query` 这两个方法即可。

LangChain 使用 OpenAI Embeddings 嵌入模型示例：

```
import dotenv
import numpy as np
from langchain_openai import OpenAIEmbeddings
from numpy.linalg import norm

dotenv.load_dotenv()


def cosine_similarity(vector1: list, vector2: list) -> float:
    """计算传入两个向量的余弦相似度"""
    # 1.计算内积/点积
    dot_product = np.dot(vector1, vector2)

    # 2.计算向量的范数/长度
    norm_vec1 = norm(vector1)
    norm_vec2 = norm(vector2)

    # 3.计算余弦相似度
    return dot_product / (norm_vec1 * norm_vec2)


embeddings = OpenAIEmbeddings()

query_vector = embeddings.embed_query("你好，我是慕小课，我喜欢打篮球")
documents_vector = embeddings.embed_documents([
    "你好，我是慕小课，我喜欢打篮球",
    "这个喜欢打篮球的人叫慕小课",
    "求知若渴，虚心若愚"
])

print(query_vector)
print(len(query_vector))

print("============")

print(len(documents_vector))
print("vector1与vector2的余弦相似度:", cosine_similarity(documents_vector[0],
documents_vector[1]))
print("vector2与vector3的余弦相似度:", cosine_similarity(documents_vector[0],
documents_vector[2]))
```

CacheBackEmbedding 组件的使用与注意事项

CacheBackEmbedding 组件的使用与注意事项

01. CacheBackEmbedding 使用与场景

通过嵌入模型计算传递数据的向量需要昂贵的算力，对于重复的内容，Embeddings 计算的结果肯定是一致的，如果数据重复仍然二次计算，会导致效率非常低，而且增加无用功。

所以在 LangChain 中提供了一个叫 `CacheBackEmbedding` 的包装类，一般通过类方法 `from_bytes_store` 进行实例化，它接受以下参数：

1. `underlying_embedder`：用于嵌入的嵌入模型。
2. `document_embedding_cache`：用于缓存文档嵌入的任何存储库（ByteStore）。
3. `batch_size`：可选参数，默认为 None，在存储更新之间要嵌入的文档数量。
4. `namespace`：可选参数，默认为""，用于文档缓存的命名空间。此命名空间用于避免与其他缓存发生冲突。例如，将其设置为所使用的嵌入模型的名称。
5. `query_embedding_cache`：可选默认为 None 或者不缓存，用于缓存查询/文本嵌入的 ByteStore，或这是为 True 以使用与 `document_embedding_cache` 相同的存储。

> 注意事项：CacheBackEmbedding 默认不缓存 embed_query 生成的向量，如果要缓存，需要设置 query_embedding_cache 的值，另外请尽可能设置 namespace，以避免使用不同嵌入模型嵌入的相同文本发生冲突。

使用示例：

```
import dotenv
import numpy as np
from langchain.embeddings import CacheBackedEmbeddings
from langchain.storage import LocalFileStore
from langchain_openai import OpenAIEmbeddings
from numpy.linalg import norm

dotenv.load_dotenv()


def cosine_similarity(vector1: list, vector2: list) -> float:
    """计算传入两个向量的余弦相似度"""
    # 1.计算内积/点积
    dot_product = np.dot(vector1, vector2)

    # 2.计算向量的范数/长度
    norm_vec1 = norm(vector1)
    norm_vec2 = norm(vector2)

    # 3.计算余弦相似度
    return dot_product / (norm_vec1 * norm_vec2)


embeddings = OpenAIEmbeddings(model="text-embedding-3-small")
embeddings_with_cache = CacheBackedEmbeddings.from_bytes_store(
    embeddings,
```

```
    LocalFileStore("./cache/"),
    namespace=embeddings.model,
    query_embedding_cache=True,
)

query_vector = embeddings_with_cache.embed_query("你好，我是慕小课，我喜欢打篮球")
documents_vector = embeddings_with_cache.embed_documents([
    "你好，我是慕小课，我喜欢打篮球",
    "这个喜欢打篮球的人叫慕小课",
    "求知若渴，虚心若愚"
])

print(query_vector)
print(len(query_vector))

print("============")

print(len(documents_vector))
print("vector1与vector2的余弦相似度:", cosine_similarity(documents_vector[0],
documents_vector[1]))
print("vector2与vector3的余弦相似度:", cosine_similarity(documents_vector[0],
documents_vector[2]))
```

02. CacheBackEmbedding 运行流程

CacheBackEmbedding 底层的运行流程非常简单，本质上就是**封装了一个持久化存储的数据存储仓库**，在每次进行数据嵌入前，会从前数据存储仓库中检索对对应的向量，然后逐个匹配对应的数据是否相等，找到缓存中没有的文本，然后将这些文本调用嵌入生成向量，最后将生成的新向量存储到数据仓库中，从而完成对数据的存储。

运行流程如下：