



什么是embedding?

Vicki Boykis

译: weedge

Abstract

在过去的十年中，`embeddings`（作为深度学习模型输入的机器学习特征的数值表示）已经成为工业机器学习系统中的基础数据结构。TF-IDF、PCA和one-hot编码一直是机器学习系统中的关键工具，用于压缩和理解大量文本数据。然而，随着现代应用程序捕获的数据量、速度和多样性的爆炸性增长，传统方法在处理日益增长的数据量时所能推理的上下文量是有限的，因此创建专门针对扩展的方法变得越来越重要。

Google的Word2Vec论文在从简单的统计表示转向词语的语义含义方面迈出了重要的一步。随后Transformer架构和转移学习的出现，以及生成型方法的最新兴起，促进了embeddings作为基础机器学习数据结构的发展。这篇综述论文旨在深入探讨embeddings是什么、它们的历史以及在工业界的使用模式。

后记

本文使用 `LATEX`排版。封面艺术品为康定斯基的《圆中的圆》，1923年。使用 ChatGPT 生成了部分图表。

代码、`LATEX`和网站

本文和代码示例的最新版本 [可在此处获取](#)。该项目的 [网站在此处](#)。

关于作者

Vicki Boykis 是一名机器学习工程师。她的个人网站是 vickiboykis.com，她的语义搜索副业是 viberary.pizza。

致谢

我要感谢所有慷慨提供技术反馈的人，特别是Nicola Barbieri，Peter Baumgartner，Luca Belli，James Kirk和Ravi Mody。所有剩余的错误、打字错误和冷笑话都是我的。感谢丹的耐心、鼓励，感谢他在我处于潜在空间时做家长，也感谢他曾经随意地问过：“你到底是怎么生成这些‘embeddings’的呢？”

许可证

本作品采用 Creative Commons “署名-非商业性使用-相同方式共享 3.0 中国大陆”许可协议进行许可。



Contents

1	引言	4
2	作为商业问题的推荐	9
2.1	构建Web应用程序	11
2.2	基于规则的系统与机器学习	12
2.3	使用机器学习构建Web应用	14
2.4	制定机器学习问题	16
2.4.1	推荐任务	18
2.4.2	机器学习特征	20
2.5	数值特征向量	21
2.6	从词语到向量的三个基本要素	22
3	历史编码方法	23
3.1	早期方法	23
3.2	编码	24
3.2.1	指示器和独热编码	24
3.2.2	TF-IDF	28
3.2.3	SVD 和 PCA	33
3.3	LDA 和 LSA	34
3.4	传统方法的局限性	34
3.4.1	维数灾难	35
3.4.2	计算复杂性	35
3.5	支持向量机	36
3.6	Word2Vec	37
4	现代embeddings方法	44
4.1	神经网络	44
4.1.1	神经网络架构	45
4.2	Transformers	46
4.2.1	编码器/解码器与注意力机制	47
4.3	BERT	51
4.4	GPT	51
5	生产中的embeddings	52
5.1	实践中的embeddings	52
5.1.1	Pinterest	53
5.1.2	YouTube 和 Google Play Store	53
5.1.3	Twitter	56
5.2	embeddings作为工程问题	58
5.2.1	embeddings生成	60
5.2.2	存储和检索	60
5.2.3	漂移检测、版本控制和可解释性	62
5.2.4	推理和延迟	63
5.2.5	在线和离线模型评估	63
5.2.6	什么使embeddings项目成功	64
6	总结	64

1 引言

实现深度学习模型已成为越来越重要的机器学习策略¹，对于希望建立数据驱动产品的公司而言尤为重要。为了构建和驱动深度学习模型，公司收集并向深度学习模型输入了数以亿计的多模态²数据。因此，**embeddings**—深度学习模型对其输入数据的内部表示—迅速成为构建机器学习系统的关键组成部分。

例如，它们在 Spotify 的项目推荐系统中占据了重要地位 [24]，YouTube 视频推荐系统中起着重要作用 [10]，以及 Pinterest 的视觉搜索中也有应用 [28]。即使它们没有被明确呈现给用户通过推荐系统的用户界面，embeddings 也被内部使用，比如 Netflix 根据用户喜好的流行程度做出内容决策，决定开发哪些节目。

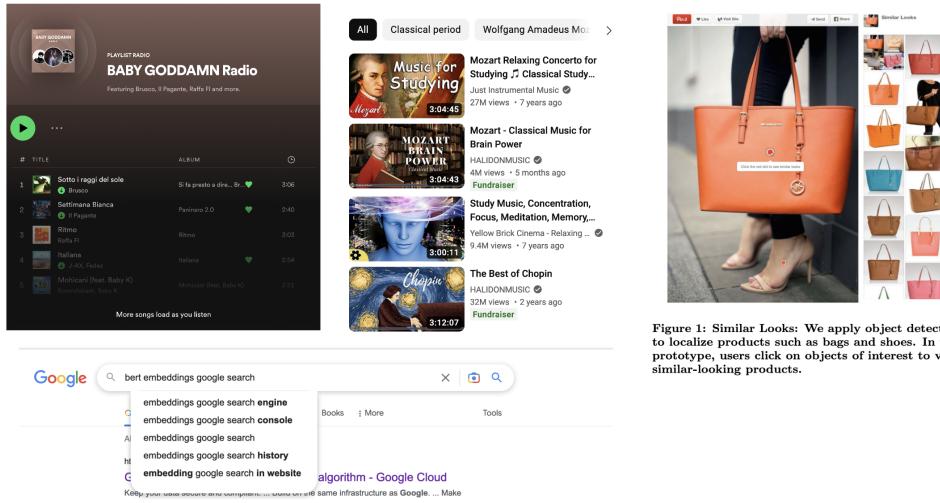


Figure 1: 从左到右：使用**embeddings**生成推荐项的产品：*Spotify Radio*、*YouTube* 视频推荐、*Pinterest* 的视觉推荐、*BERT embeddings*在建议的 *Google* 搜索结果中

embeddings用于生成内容的压缩、特定上下文表示的使用在 *Google* 的 Word2Vec 论文发表后迅速流行起来 [42]。

¹查看每年汇总的机器学习工业观点 Matt Turck [发布的报告](#)，其规模已经大幅扩展。

²多模态意味着各种数据，通常包括文本、视频、音频等，最近还包括 Meta 的 ImageBind 中展示的深度、热量和 IMU 数据等。

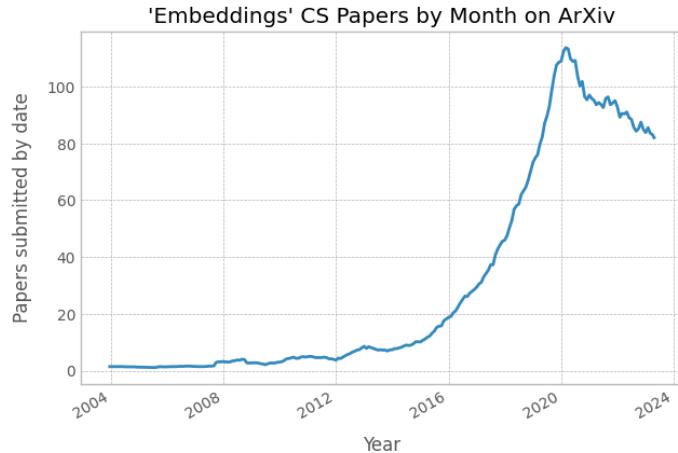


Figure 2: *arXiv* 上关于 *embeddings* 的论文数量按月统计。值得注意的是 *embeddings* 特定论文的频率下降，可能与 GPT 等深度学习架构的兴起同时进行 [来源](#)

在 Word2Vec 的基础上构建和扩展，具有自注意机制的 Transformer [57] 架构已成为学术界和工业界学习不断增长的多模态词汇表示的事实标准，并且其在学术界和工业界的流行程度提高导致 *embeddings* 成为深度学习工作流程的重要组成部分。

然而，*embeddings* 的概念可能会让人困惑，因为它们既不是数据流的输入，也不是输出结果 - 它们是在机器学习服务中存在的中间元素，用于优化模型。因此，从一开始就明确定义它们会很有帮助。

作为一个通用的定义，*embeddings* 是被转换为 n 维矩阵以用于深度学习计算的数据。*embeddings* 的过程（作为动词）包括：

- 将多模态输入转换为更易于进行密集计算的表示形式，以向量、张量或图的形式存在[45]。对于机器学习来说，我们可以将向量视为数字的列表（或数组）。
- 压缩输入信息以用于机器学习任务 — 用于解决特定问题的机器学习方法 — 例如总结文档、识别社交媒体帖子的标签或标签，或在大型文本语料库上执行语义搜索。压缩的过程将可变的特征维度转换为固定的输入，使其能够有效地传递到机器学习系统的下游组件中。
- 创建 *embeddings* 空间，该空间特定于 *embeddings* 训练的数据，但在深度学习表示的情况下，也可以通过迁移学习 — 能够切换上下文 — 推广到其他任务和领域，这也是 *embeddings* 在机器学习应用中爆炸性增长的原因之一。

embeddings 实际上是什么样子呢？这里是一个单一的 *embeddings*，也称为向量，在三个维度中。我们可以将其视为数据集中单个元素的表示。例如，这个假设的 *embeddings* 表示了一个单词“fly”，在三个维度上。通常，我们将单个 *embeddings* 表示为行向量。

$$\begin{bmatrix} 1 & 4 & 9 \end{bmatrix} \tag{1}$$

而这是一个张量，也称为矩阵³，它是多个元素的向量表示的多维组合。例如，这可能是“fly”和“bird”的表示。

³矩阵和张量的区别在于，如果你在做线性代数，那么它就是一个矩阵；如果你是一个人工智能研究者，那么它就是一个张量。

$$\begin{bmatrix} 1 & 4 & 9 \\ 4 & 5 & 6 \end{bmatrix} \quad (2)$$

这些embeddings是通过学习embeddings的过程的输出，我们通过将原始输入数据传递给机器学习模型来进行这一过程。我们通过本文讨论的算法将这个多维输入数据压缩到一个低维空间中，从而进行了转换。结果是在一个embeddings空间中的一组向量。



Figure 3: embeddings的过程。

我们经常谈论项目embeddings在X维空间中，范围从100到1000不等，在将其用于机器学习问题时，在200-300之后，其效用会递减⁴。这意味着每个项目（图像、歌曲、单词等）都由长度为X的向量表示，其中每个值都是X维空间中的一个坐标。

我们刚刚为“bird”编了一个embeddings，但让我们看看在语录中为词“hold”生成的真实embeddings是什么样子，这是由BERT深度学习模型生成的，

“Hold fast to dreams, for if dreams die, life is a broken-winged bird that cannot fly.” — Langston Hughes

我们突出了这个语录，因为我们将在本文中将这句话作为我们的输入示例进行处理。

⁴embeddings大小可以调整为超参数，但到目前为止，关于最佳embeddings大小的研究还不多，大多数embeddings的大小是通过魔法和猜测来确定的。

```
1 import torch
2 from transformers import BertTokenizer, BertModel
3
4 # Load pre-trained model tokenizer (vocabulary)
5 tokenizer = BertTokenizer.from_pretrained('bert-base-uncased')
6
7 text = """Hold fast to dreams, for if dreams die, life is a broken-winged bird
8     ↳ that cannot fly."""
9
10 # Tokenize the sentence with the BERT tokenizer.
11 tokenized_text = tokenizer.tokenize(text)
12
13 # Print out the tokens.
14 print (tokenized_text)
15
16 ['[CLS]', 'hold', 'fast', 'to', 'dreams', ',', 'for', 'if', 'dreams', 'die',
17     ↳ ',', 'life', 'is', 'a', 'broken', '-', 'winged', 'bird', 'that', 'cannot',
18     ↳ 'fly', '.', '[SEP]']
19
20 # BERT code truncated to show the final output, an embedding
21
22 [tensor([-3.0241e-01, -1.5066e+00, -9.6222e-01,  1.7986e-01, -2.7384e+00,
23          -1.6749e-01,  7.4106e-01,  1.9655e+00,  4.9202e-01,
24          -2.0871e+00,
25          -5.8469e-01,  1.5016e+00,  8.2666e-01,  8.7033e-01,
26          -8.5101e-01,
27          5.5919e-01, -1.4336e+00,  2.4679e+00,  1.3920e+00,
28          -3.9291e-01,
29          -1.2054e+00,  1.4637e+00,  1.9681e+00,  3.6572e-01,
30          -3.1503e+00,
31          -4.4693e-01, -1.1637e+00,  2.8804e-01, -8.3749e-01,
32          -1.5026e+00,
33          -2.1318e+00,  1.9633e+00, -4.5096e-01, -1.8215e+00,
34          -3.2744e+00,
35          5.2591e-01,  1.0686e+00,  3.7893e-01, -1.0792e-01,
36          -5.1342e-01,
37          -1.0443e+00,  1.7513e+00,  1.3895e-01, -6.6757e-01,
38          -4.8434e-01,
39          -2.1621e+00, -1.5593e+01,  1.5249e+00,  1.6911e+00,
40          -1.2916e+00,
41          1.2339e+00, -3.6064e-01, -9.6036e-01,  1.3226e+00,
42          -1.6427e+00,
43          1.4588e+00, -1.8806e+00,  6.3620e-01,  1.1713e+00,
44          -1.1050e+00, ...
45          2.1277e+00])
```

Figure 4: 使用BERT分析embeddings。查看完整的笔记本 [源代码](#)

我们可以看到，这个embeddings是一个PyTorch张量对象，一个包含多

个embeddings层级的多维矩阵，这是因为在BERT的embeddings表示中，我们有13个不同的层。每一层的神经网络都计算一个embeddings层。每个级别表示我们给定的标记的不同视图—或者简单地说是一个字符序列。通过对几个层进行汇总，我们可以得到最终的embeddings，这些细节将在我们逐步理解使用BERT生成的embeddings时深入讨论。

当我们为一个单词、句子或图像创建一个表示该艺术品的embeddings，我们可以对这个embeddings进行任意数量的操作。例如，对于在机器学习中专注于内容理解的任务，我们通常感兴趣的是比较两个给定项，看它们有多相似。将文本投影为向量使我们能够以数学严谨的方式进行比较，并在共享的embeddings空间中比较单词。

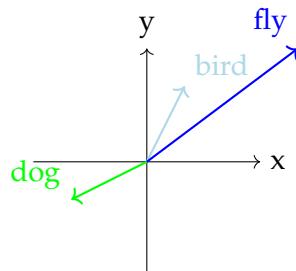


Figure 5: 将单词投影到共享embeddings空间

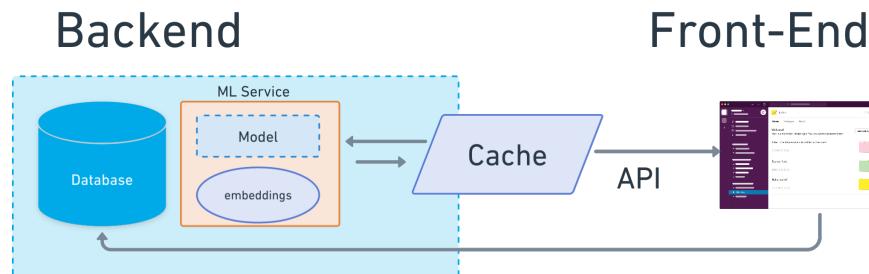


Figure 6: 在应用程序上下文中的embeddings。

基于embeddings的工程系统在构建和维护方面可能会非常昂贵 [53]。需要创建、存储和管理embeddings的需求最近导致了整个相关产品生态系统的爆炸。例如，最近对矢量数据库的开发大幅增加，以促进在机器学习系统中进行最近邻语义查询的生产就绪使用⁵，以及embeddings作为一项服务的崛起⁶。

因此，重要的是要了解它们的上下文，无论是作为最终消费者、产品管理团队，还是作为与它们一起工作的开发人员。但是在我深入研究embeddings参考资料时，我发现有两种资源：非常深入的技术学术论文，适用于已经是NLP专家的人，以及表面级的营销垃圾，适用于寻求购买基于embeddings的技术的人，而这两者在所涵盖的内容上没有重叠。

在系统思维中，Donella Meadows写道：“你认为因为你理解了‘一个’，所以你必须理解了‘二’，因为一个加一个等于二。但你忘了你还必须理解‘和’。”[40] 为了理解当前的embeddings架构并能够决定如何构建它们，我们必须了解它们是如何产生的。在构建自己的理解时，我想要一个足够技

⁵有关今天矢量数据库空间的调查，请参阅 [这篇文章](#)

⁶embeddings现在是按需ML服务定价的一个关键区别因素

术的资源，以对机器学习从业者有所帮助，但同时也要将embeddings放入它们在ML架构堆栈中更频繁使用的正确业务和工程上下文中。这就是这篇文章的目的。

在这篇文章中，我们将从三个角度来审视embeddings，从最高级别的视角到最技术化的视角。我们将从业务环境开始，然后是工程实现，最后是机器学习理论，重点是它们的工作原理的细节。在一个平行的轴上，我们还将穿越时间，调查最早的方法，然后移向现代的embeddings方法。

在撰写本文时，我努力平衡了对概念进行精确的技术和数学定义的需求，以及我希望避免让人们的眼睛发直的解释。我在首次出现技术术语时对其进行定义，以建立上下文。我包含了代码作为从业者的参考框架，但没有像代码教程那样深入⁷。因此，读者最好对编程和机器学习基础有一些了解，特别是在讨论业务背景的部分之后。但是，最终的目标是教育任何愿意阅读本文的人，无论他们的技术水平如何。

值得一提的是，这篇文章并不试图解释GPT和生成模型的最新进展，也不试图完全解释transformers，也不试图覆盖所有爆炸性增长的矢量数据库和语义搜索领域。我尽力使它简单化，并专注于真正理解embeddings的核心概念。

2 作为商业问题的推荐

在深入讨论实施细节之前，让我们退后一步，通过一个具体的例子来看看更大的背景。让我们构建一个社交媒体网络，Flutter，这是一款面向所有有翅膀的事物的首要社交网络。Flutter是一个网站和移动应用程序，鸟类可以在上面发布短文本、视频、图片和声音，让周围的其他鸟类、昆虫和蝙蝠知道发生了什么。它的商业模式基于定向广告，其应用架构包括一个基于用户关注的“主页”反向时间线，由称为“flits”的小型多媒体内容组成，可以是文本、视频或照片。主页反向时间线默认按用户策划的时间顺序排列。但我们还想提供个性化的推荐flits，以便用户在我们的平台上发现他们以前可能不知道的有趣内容。

⁷换句话说，我希望跨越Diátaxis框架的“解释”和“参考”象限。



Figure 7: *Flutter*的内容时间轴，其中包含了有机关注内容、广告和推荐内容的社交反馈。

在这里，我们如何解决时间轴中要显示什么的问题，以使我们的用户发现内容相关且有趣，并平衡广告商和业务合作伙伴的需求？

在许多情况下，我们可以在不涉及机器学习的情况下解决工程问题。实际上，我们绝对应该从不涉及机器学习的情况开始，因为机器学习给我们的工作应用程序增加了巨大的复杂性。然而，在*Flutter*主页反向时间线的情况下，机器学习形成了产品提供的业务关键功能的一部分。从业务产品的角度来看，目标是向*Flutter*的用户提供相关的内容⁸、有趣且新颖，以便他们继续使用该平台。如果我们不在以内容为中心的产品中构建发现和个性化，*Flutter*用户将无法发现更多的内容来消费，并将退出该平台。

这是许多基于内容的业务的情况，所有这些业务都有推荐类似的表面领域，包括Netflix、Pinterest、Spotify和Reddit。它还涵盖了电子商务平台，必须向用户呈现相关的项目，以及信息检索平台，必须在关键字查询时向用户提供相关的答案。由于GPT系列模型的工作，还出现了一个涉及语义搜索上下文中的问答混合应用程序的新类别，但为了简化起见，以及因为这个领域每周都在变化，我们将专注于理解基本的基础概念。

在订阅式平台⁹中，有一个明确的与底线直接相关的业务目标，正如这篇关于Netflix的推荐系统的2015年论文中所述：

⁸在推荐领域，相关项目的具体定义因人而异，并且在学术界和工业界的激烈辩论下变化，但通常意味着对用户感兴趣的项目

⁹在基于广告的服务中，留存和收入之间的界限有点模糊，我们通常会遇到所谓的多利益相关者问题，其中实际上优化的函数是在满足用户需求和广告主需求之间取得平衡[65]。在现实生活中，这往往会导致平台的恶化过程[14]，从而导致极其次优的最终用户体验。因此，当我们创建*Flutter*时，我们必须非常小心地平衡这些问题，出于简化的目的，我们也假设*Flutter*是一个爱我们作为用户并希望我们幸福的良好服务。

Netflix推荐系统的主要任务是帮助我们的会员发现他们将观看和喜欢的内容，以最大程度地满足他们的长期满意度。由于每个人都是独一无二的，他们有各种各样的兴趣，这些兴趣在不同的情境下可能会有所不同，并且在不确定自己想要观看什么时，需要一个推荐系统。做到这一点意味着每个会员都会获得一个独特的体验，使他们能够充分利用Netflix。作为一个月度订阅服务，会员满意度与一个人留存到我们的服务中的可能性密切相关，这直接影响到我们的收入。

在这个业务背景下，个性化内容通常更相关，并且通常在线平台上获得更高的参与率^[27]，相较于非个性化的推荐形式¹⁰。因此，我们可能会考虑如何在Flutter的机器学习工作流程中使用嵌入来向用户展示对他们个人感兴趣的 flit。要理解这一点，我们首先需要了解Web应用程序的工作原理以及嵌入其中的作用。

2.1 构建Web应用程序

我们今天使用的大多数应用程序—Spotify、Gmail、Reddit、Slack和Flutter—都是基于相同的基础软件工程模式设计的。它们都是可以在Web和移动客户端上使用的应用程序。它们都有一个前端，用户可以与应用程序的各种产品功能进行交互，一个API将前端与后端元素连接起来，一个数据库处理数据并记住状态。

作为重要说明，功能在机器学习和工程领域有很多不同的定义。在这个特定的情况下，我们指的是构成某些前端元素的代码集合，例如按钮或推荐面板。我们将这些称为产品功能，与机器学习功能相对，后者是机器学习模型的输入数据。

这种应用程序架构通常称为模型-视图-控制器模式^[18]，或者在通用行业术语中，称为CRUD应用程序，因为其API允许执行管理应用程序状态的基本操作：创建、读取、更新和删除。

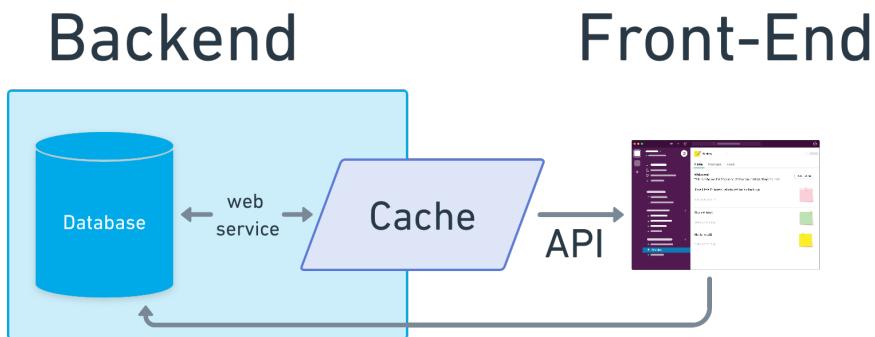


Figure 8: 典型的CRUD Web应用程序架构

当我们考虑这些应用程序的体系结构中的结构性组件时，我们可能首先会从产品功能的角度考虑。例如，在Slack这样的应用程序中，我们可以发布和

¹⁰有关更多信息，请参阅[这个案例研究](#)关于个性化推荐以及[这篇论文的介绍部分](#)，其中涵盖了许许多个性化使用案例。

阅读消息、管理通知，并添加自定义表情符号。这些都可以看作是应用程序功能。为了创建功能，我们必须组合常见的元素，如数据库、缓存和Web服务。所有这些都是在客户端与API进行通信时发生的，API将数据传递给数据库以处理。

在更精细、特定于程序的层面上，我们可能会考虑到诸如数组或哈希映射之类的基本数据结构，而更低的层次可能是关于内存管理和网络拓扑的考虑。这些都是现代编程的基础要素。

在功能级别上，我们看到它不仅包括典型的CRUD操作，如发布和阅读Slack消息，还包括更多的元素，这些元素不仅仅是改变数据库状态的操作。一些功能，如个性化频道建议、通过搜索查询返回相关结果和预测Slack连接邀请，需要使用机器学习。

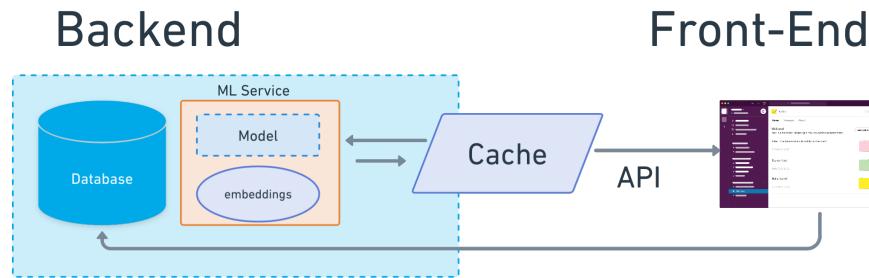


Figure 9: 具有机器学习服务的CRUD应用程序

2.2 基于规则的系统与机器学习

要理解embeddings在这些系统中的位置，首先有必要了解机器学习在Flutter或任何给定公司中作为整体的位置。在典型的消费类公司中，用户界面应用程序由编写的产品功能组成，通常作为服务或服务的部分编写。要添加一个新的Web应用程序功能，我们根据一组业务逻辑要求编写代码。这些代码作用于应用程序中的数据，以开发我们的新功能。

在典型的数据中心软件开发生命周期中，我们从业务逻辑开始。例如，让我们考虑发布消息的能力。我们希望用户能够以他们选择的语言输入文本和表情符号，在消息按时间顺序排序，并在Web和移动设备上正确呈现。这些是业务要求。我们使用输入数据，例如用户消息，在UI中以低延迟正确格式化和按时间顺序排序。



Figure 10: 典型的应用程序开发生命周期

基于机器学习的系统通常也是Web应用程序后端的服务。它们被整合到生产工作流程中。但是，它们处理数据的方式有很大不同。在这些系统中，我们不是从业务逻辑开始。我们从用于构建将为我们建议业务逻辑的模型的输入数据开始。有关如何考虑这些数据中心工程系统的具体信息，请参阅Kleppmann[32]。

这需要稍微不同的应用程序开发方式，当我们编写一个包含机器学习模型作为输入的应用程序时，我们实际上是颠倒了传统的应用程序生命周期。我们现在有的是数据加上我们想要的结果。数据被组合成一个模型，而这个模型则生成我们的业务逻辑，以构建功能。

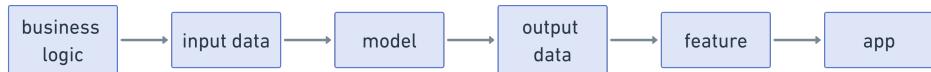


Figure 11: 机器学习开发生命周期

简而言之，编程和机器学习开发之间的区别在于，我们不是通过业务规则生成答案，而是通过数据生成业务规则。然后将这些规则重新整合到应用程序中。

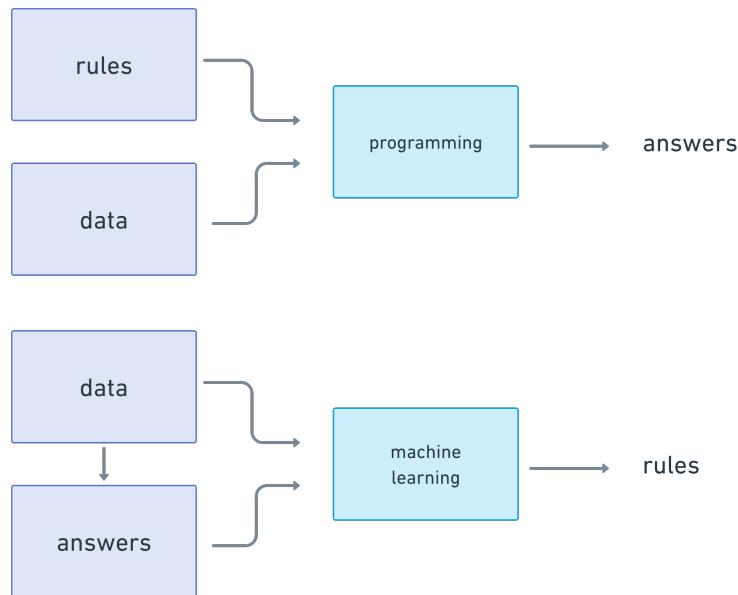


Figure 12: 通过机器学习生成答案。顶部图表显示了一种经典的编程方法，其中规则和数据作为输入，而底部图表显示了一种机器学习方法，其中数据和答案作为输入。[\[7\]](#)

例如，对于Slack的频道推荐产品功能，我们并不是在硬编码一个需要从组织API中调用的频道列表。我们正在输入关于组织用户的数据（他们加入的其他频道、他们成为用户的时间长短、他们与Slack中最常互动的人的频道），并在该数据上构建一个模型，该模型为每个用户推荐一个非确定性的、个性化的频道列表，然后通过UI显示出来。

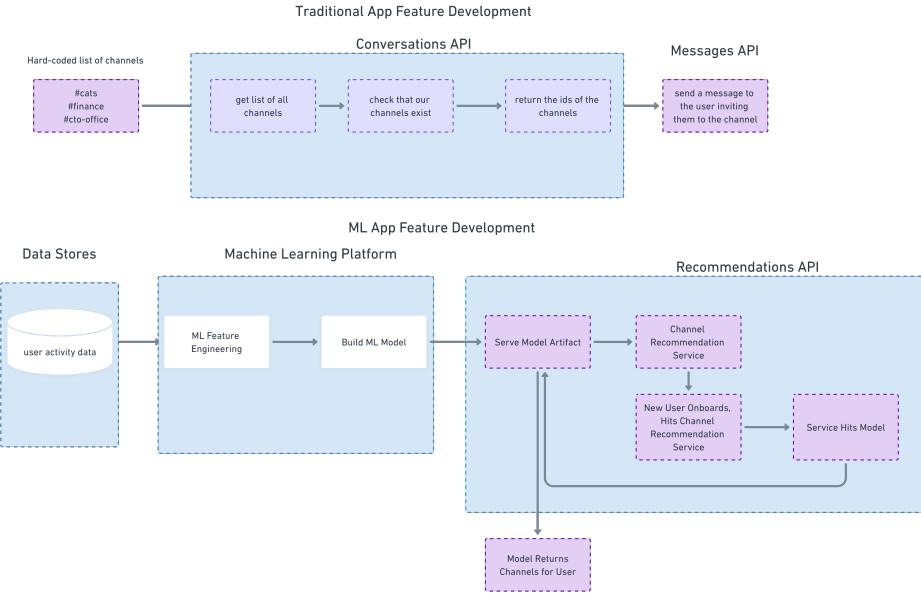


Figure 13: 传统与机器学习架构和基础设施

2.3 使用机器学习构建Web应用

所有机器学习系统都可以通过它们完成这四个步骤来进行审视。当我们构建模型时，我们的关键问题应该是，“我们有什么样的输入数据以及它的格式是什么”，以及“我们得到了什么样的结果”。我们将对我们所看到的每种方法都问这个问题。当我们构建一个机器学习系统时，我们从处理数据开始，最终提供一个学习到的模型产物。

机器学习系统的四个组成部分是¹¹：

- 输入数据 - 从数据库中处理数据或从生产应用程序流式传输以用于建模
- 特征工程和选择 - 检查数据并对其进行清理以选择特征的过程。在这种情况下，我们指的是用作机器学习输入的任何给定元素的属性。特征的例子包括：用户名称、地理位置、过去5天点击按钮的次数以及收入。这个步骤在任何给定的机器学习系统中通常花费最长的时间，并且也被称为找到最适合机器学习算法的数据的表示 [4]。在新的模型架构中，这是我们使用embeddings作为输入的地方。
- 模型构建 - 我们选择重要的特征并训练我们的模型，不断迭代不同的性能指标，直到我们有一个可接受的模型可以使用。**embeddings**也是这一步的输出，我们可以在其他下游步骤中使用它们。
- 模型提供 - 现在我们有了一个喜欢的模型，我们将其提供给生产环境，在那里它会访问一个Web服务，可能会进行缓存，并在API中进行传播，然后传播到前端，供用户作为我们Web应用程序的一部分来消费

¹¹机器学习系统中有无数层的可怕之处 [33]。这些仍然是基础组件。

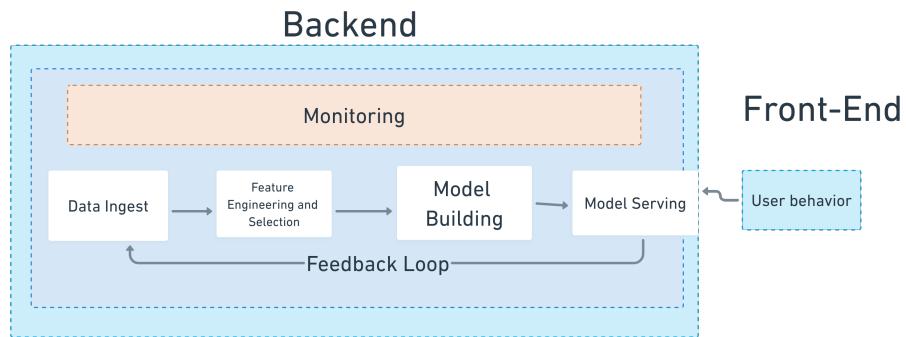


Figure 14: 带有机器学习的CRUD应用

在机器学习中，我们可以使用许多方法来适应不同的任务。最有效的机器学习工作流程被制定为解决特定的业务需求和机器学习任务。任务可以最好地被认为是解决方案空间中的建模方法。例如，学习回归模型是一个特定的任务案例。其他任务包括聚类、机器翻译、异常检测、相似度匹配或语义搜索。ML任务的三个最高级别类型是监督学习，在这种情况下，我们有训练数据可以告诉我们模型预测的结果是否正确，根据某种模型的世界。第二个是无监督学习，这里没有单一的地面真实答案。这里的一个例子是我们客户群的聚类。聚类模型可以检测数据中的模式，但不会明确标记这些模式是什么。第三个是强化学习，它与这两个类别分开，并且被公式化为一个博弈论问题：我们有一个代理通过环境移动，我们想要了解如何使用探索-利用技术最优地将他们通过给定环境移动。我们将专注于监督学习，并查看PCA和Word2Vec中的无监督学习。

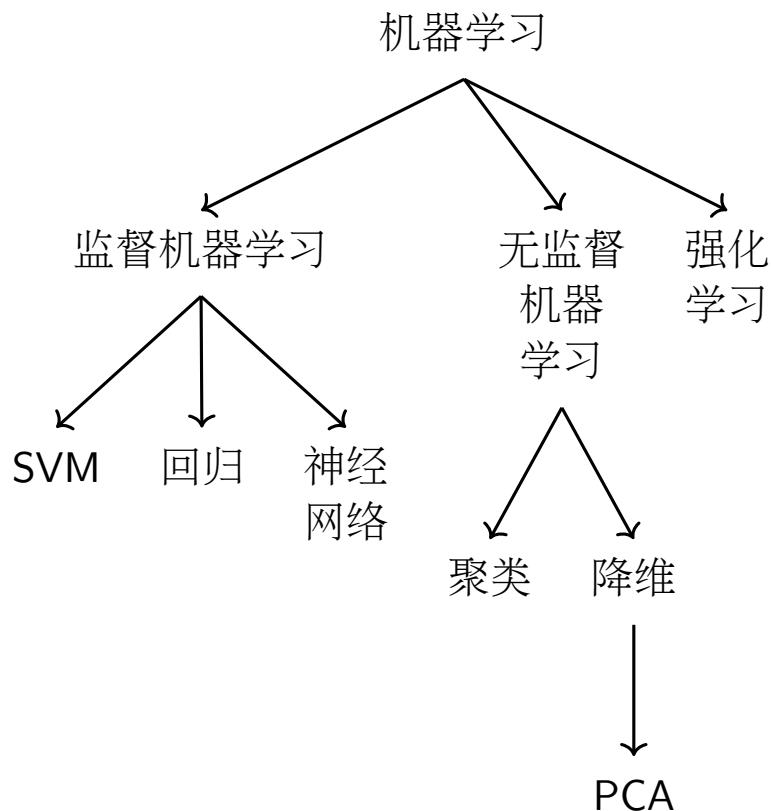


Figure 15: 机器学习任务解决方案空间和模型家族

2.4 制定机器学习问题

正如我们在上一节中看到的，机器学习是一个过程，它将数据作为输入，生成关于如何对某事进行分类、过滤或推荐的规则，具体取决于手头的任务。在任何这些情况下，例如生成一组潜在的候选者，我们需要构建一个模型。

机器学习模型是从数据中生成给定输出的一组指令。这些指令是从输入数据本身的特征中学习的。对于Flutter，我们想要构建的一个示例模型是一个候选生成器，它选择类似于我们的鸟已经喜欢的鸟飞行片段，因为我们认为用户也会喜欢这些片段。为了建立机器学习工作流程的直观感觉，让我们选择一个非常简单的示例，与我们的业务问题无关，即线性回归，它给出了一个连续变量作为响应。

例如，假设我们想要根据用户发布的帖子数量和他们喜欢的帖子数量来预测他们在Flutter上可能继续停留的天数。对于使用表格数据的传统监督建模方法，我们从我们的输入数据开始，或者在处理文本的机器学习问题中，通常被称为**NLP**（自然语言处理）领域中的语料库。

不过，我们还没有做NLP，所以我们的输入数据可能如下所示，其中包含UID（用户ID）和该用户的一些属性，例如他们发布的次数和喜欢的帖子数量。这些是我们的机器学习特征。

Table 1: Flutter用户的表格输入数据

鸟ID	鸟帖子数	鸟喜欢数
012	2	5
013	0	4
056	57	70
612	0	120

我们需要将这些数据的一部分用作训练模型，一部分用于测试我们训练的模型的准确性，一部分用于调整模型的元信息。这些被称为超参数。

我们将这些数据分成两部分，不参与模型输入。第一部分是测试集，我们使用它来验证模型对它之前从未见过的数据的最终性能。我们使用第二个分割，称为验证集，来在模型训练阶段检查我们的超参数。对于线性回归，没有真正的超参数，但我们需要记住，我们将需要调整模型的元数据以应对更复杂的模型。

假设我们有100个这样的值。通常接受的分割是使用80%的数据进行训练，20%进行测试。理由是我们希望我们的模型能够尽可能多地访问数据，以便学习更准确的表示。

总的来说，我们的目标是将输入数据输入到模型中，通过我们选择的函数进行处理，并获得一些预测输出， $f(X) \rightarrow y$ 。

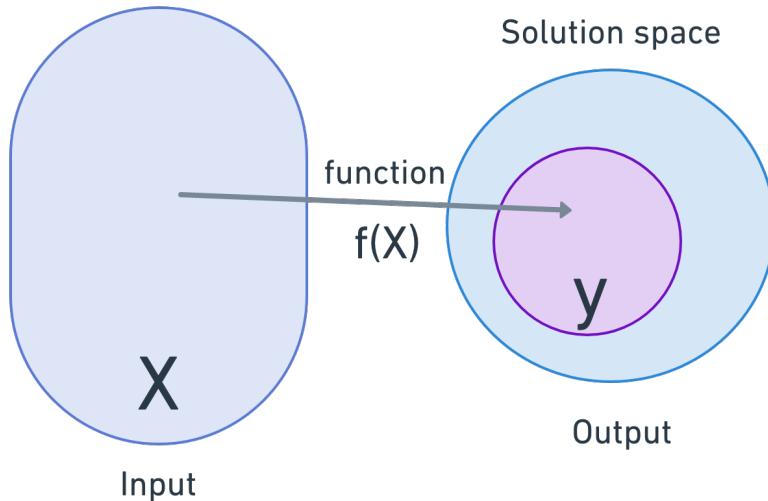


Figure 16: ML函数中输入到输出的映射 [31]

对于我们简单的数据集，我们可以使用线性回归方程：

$$y = x_1\beta_1 + x_2\beta_2 + \varepsilon \quad (3)$$

这告诉我们输出 y 可以通过两个输入变量 x_1 （鸟帖子）和 x_2 （鸟喜欢）以及它们的权重 β_1 和 β_2 加上一个误差项 ε 来预测，或者说是每个数据点与由该方程生成的回归线之间的距离。我们的任务是找到每个点与线之间的最小平方差，换句话说是最小化误差，因为这意味着在每个点处，我们的预测 y 与实际 y 尽可能接近，考虑到其他点。

$$y = x_1\beta_1 + x_2\beta_2 + \varepsilon \quad (4)$$

机器学习的核心是训练阶段，这是找到一组模型指令和数据的过程，它们可以准确地表示我们的真实数据，在监督学习中，我们可以通过检查来自测试集的正确“答案”来验证它们是否准确。

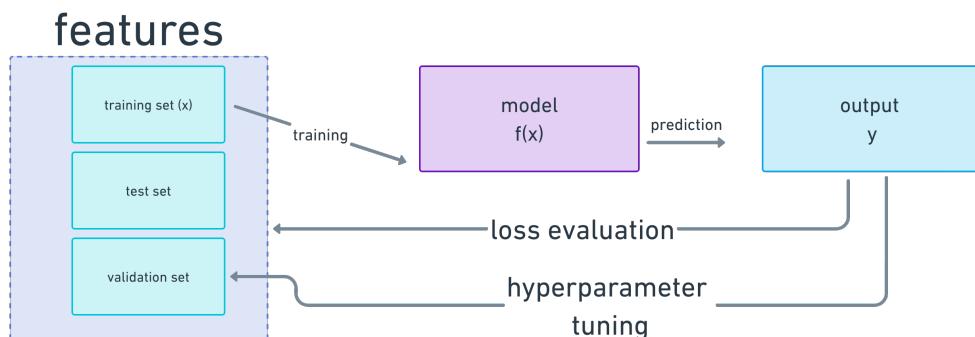


Figure 17: 机器学习模型开发循环

随着第一轮训练的开始，我们有了我们的数据。我们通过使用来自训练数据的一组输入 X 来训练—或构建—我们的模型。 β_1 和 β_2 通过设置为零或随机初始化（根据模型，不同的方法效果最好）来初始化，并且我们计算 \hat{y} ，我们模型的预测值。 ε 是根据数据和估计的系数得出的。

$$y = 2\beta_1 + 5\beta_2 + \varepsilon \quad (5)$$

我们如何知道我们的模型是好的呢？我们用一些值、权重初始化它，并通过最小化代价函数来迭代这些权重。代价函数是一个模型的预测值和训练数据的实际输出之间差异的函数。第一个输出可能不是最优的，所以我们在模型空间中进行多次迭代，优化使模型尽可能地代表真实情况，并最小化实际值和预测值之间的差异。因此，在我们的情况下，我们比较 \hat{y} 和 y 。每个观察值的实际值与预测值之间的平均平方差是成本，也称为**MSE** - 平均平方误差。

$$MSE = \frac{1}{N} \sum_{i=1}^n (y_i - (mx_i + b))^2 \quad (6)$$

我们希望最小化这个成本，我们使用梯度下降来实现这一目标。当我们说模型学习时，我们的意思是“我们可以通过一个迭代过程来了解模型的正确输入，我们通过这个过程向模型提供数据，评估输出，并查看通过梯度下降过程是否改进了它的预测。我们会知道，因为我们的损失应该在每次训练迭代中逐渐减少。

我们终于训练好了我们的模型。现在，我们在20个我们用作保留集的值上测试模型的预测；也就是说，模型之前没有见过这些值，我们可以有信心地假设它们不会影响训练数据。我们比较模型能够正确预测的保留集中的元素数量，以查看模型的准确性。

2.4.1 推荐任务

我们刚刚看到了一个与预测连续响应变量相关的机器学习简单示例。当我们的业务问题是“向用户展示什么内容比较好”时，我们面临的是推荐任务。推荐系统是为信息检索而设立的系统，这是与NLP密切相关的一个领域，专注于在大量文档集合中找到相关信息。信息检索的目标是综合大量的非结构化文本文档。在信息检索中，我们可以通过搜索和推荐两种互补的方式来向用户提供正确的答案：

搜索是有针对性的信息寻求问题，即用户提供系统一个特定查询，希望得到一组精炼的结果。搜索引擎在这一点上是该领域一个成熟的传统解决方案。

推荐是一个“人是查询”的问题。在这里，我们不知道人们确切地在寻找什么，但我们希望推断出他们喜欢什么，并基于他们学到的口味和偏好推荐项目。

最早的工业推荐系统是在施乐帕洛阿尔托研究中心为了过滤电子邮件和新闻组中的信息而创建的，这是由于在Web上过滤传入信息的需求不断增加。如今最常见的推荐系统是Netflix、YouTube等大型平台上的推荐系统，它们需要一种方式来向用户呈现相关的内容。

推荐系统的根本目标是向用户呈现相关的项目。在推荐的机器学习方法框架中，主要的机器学习任务是确定在特定情况下向用户展示哪些项目。[\[5\]](#)。解决推荐问题有几种常见的方法。

- 协同过滤 - 创建推荐的最常见方法是将我们的数据构建成在给定一组用户-项目互动历史中找到缺失的用户-项目互动的问题。我们首先收集明确（评级）数据或隐式用户互动数据，如点击、页面浏览或在项目上花费的时间，然后计算。交互的最简单形式是邻域模型，其中初始通过查找与给定目标用户相似的用户来预测评级。我们使用相似性函数来计算用户之间的接近程度。另一种常见的方法是使用矩阵分解等方法，矩阵分解的过程是在由低维因子向量组成的特征矩阵中表示用户和项目，并通过最小化成本函数的过程来学习这些特征向量。在我们的情况下，这些特征向量也被称为embeddings，并学习这些特征向量。这个过程可以被视为与Word2Vec [38] 类似的过程，Word2Vec是一个深度学习模型，我们将在本文档中深入讨论。协同过滤有许多不同的方法，包括矩阵分解和分解机。
- 内容过滤 - 此方法使用关于我们项目的可用元数据（例如电影或音乐中的标题、发行年份、流派等）作为模型的初始或附加特征输入，并且当我们没有关于用户活动的太多信息时效果很好，尽管它们通常与协同过滤方法结合使用。许多embeddings架构属于此类别，因为它们帮助我们对项目的文本特征进行建模。
- 学习排名 - 学习排名方法侧重于基于已知的首选排序集对项目进行排序，并且错误是在排名一对或一组项目时排名不正确的情况数。在这里，问题不是呈现单个项目，而是一组项目以及它们之间的相互作用。这一步通常发生在生成候选的候选生成步骤之后，因为对极大列表进行排序在计算上是很昂贵的。
- 神经推荐 - 使用神经网络来捕捉与矩阵分解相同的关系，而不需要显式地创建用户/项目矩阵，并根据输入数据的形状。这是深度学习网络以及最近的大型语言模型发挥作用的地方。用于推荐的深度学习架构的示例包括Word2Vec和BERT，我们将在本文中讨论这些架构，以及用于顺序推荐的卷积和循环神经网络（例如音乐播放列表中发现的）。深度学习使我们能够更好地对基于内容的推荐进行建模，并为我们的项目提供embeddings空间中的表示。[64]

推荐系统已经发展出自己独特的架构¹²，它们通常包括构建由多个机器学习模型组成的四阶段推荐系统，每个模型执行不同的机器学习任务。



Figure 18: 作为机器学习问题的推荐系统

- 候选生成(**Candidate Generation**) - 首先，我们从Web应用程序中获取数据。这些数据进入初始部分，这个部分生成候选推荐的第一次模型。这是协同过滤的地方，我们将我们的潜在候选人名单从数百万人中缩减到数千人或数百人。
- 排名(**Ranking**) - 最后，我们需要一种方法来根据我们认为用户最喜欢的内容对过滤后的推荐列表进行排序，因此下一阶段是排名，然后我们在时间轴或我们正在处理的ML产品界面中为他们提供服务。

¹²关于搜索和推荐之间的相似性和差异的良好调查，请阅读[这篇关于系统设计的精彩文章](#)

- **过滤(Filtering)** - 一旦我们生成了候选人名单，我们希望继续对它们进行过滤，使用业务逻辑（例如，我们不想看到不安全的内容，或者不想看到不打折的项目，例如）。这通常是一个严重基于启发式的步骤。
- **检索(Retrieval)** - 这是Web应用程序通常打到模型端点以获取通过产品UI向用户提供的最终项目列表的部分。

数据库已经成为构建执行数据查找的后端基础设施的基本工具。`embeddings`已经成为许多现代搜索和推荐产品架构创建的类似构建块。`embeddings`是一种机器学习特征 — 或模型输入数据 — 我们首先将其作为输入到特征工程阶段，然后将我们的候选生成阶段的第一组结果，它们随后被合并到排名和检索的下游处理步骤中，以产生用户看到的最终项目。

2.4.2 机器学习特征

现在我们对机器学习和推荐系统的工作原理有了一个高层次的概念性视图，让我们朝着一个候选生成模型的方向构建，该模型将提供相关的 `flits`。

让我们从建模一个传统的机器学习问题开始，并将其与我们的NLP问题进行对比。例如，假设我们的一个业务问题是预测一只鸟是否可能会继续留在Flutter平台上，还是流失¹³ — 与平台断开联系并离开。

当我们预测流失时，我们对于每个用户都有一组给定的机器学习特征输入，并且从模型中得到最终的二进制输出，如果鸟可能会流失，则为 1，如果用户可能会留在平台上，则为 0。

我们可能有以下输入：

- 过去一个月鸟点击了多少篇帖子（我们将其称为我们的输入数据中的 `bird_posts`）
- 鸟的地理位置（从浏览器头部获得）（`bird_geo`）
- 过去一个月鸟喜欢了多少篇帖子（`bird_likes`）

Table 2: Flutter 用户的表格输入数据

bird_id	bird_posts	bird_geo	bird_likes
012	2	美国	5
013	0	英国	4
056	57	新西兰	70
612	0	英国	120

我们首先选择我们的模型特征，并将它们按表格格式排列。我们可以根据鸟 ID 和我们的鸟特征的行来构建这个数据表（如果我们仔细看，这也是一个矩阵）。

表格数据是任何结构化数据。例如，对于给定的 Flutter 用户，我们有他们的用户 ID、他们喜欢的帖子数量、账户的年龄等。这种方法对于我们认为的传统机器学习方法很有效，因为它们处理的是表格数据。通常情况下，正确制定输入数据的过程可能是机器学习的核心。也就是说，如果我们有不良

¹³这是几乎每个行业都要解决的一个非常普遍的业务问题，无论是客户群体还是基于收入的订阅。

的输入，我们将得到不良的输出。因此，在所有情况下，我们都希望花费时间非常谨慎地组织我们的输入数据集和特征工程。

这些都是我们可以将其馈送到模型并从中学习权重的离散特征，只要我们有数字特征，就是相当简单的。但是，这里需要注意的一点是，在我们的鸟互动数据中，我们既有数字特征，也有文本特征（鸟的地理位置）。那么我们该如何处理这些文本特征？我们如何比较“美国”和“英国”？

将数据正确格式化以馈送到模型中的过程称为特征工程。当我们有单个连续的、数值特征时，比如“*flit* 的年龄（天）”，我们可以很容易地将这些特征馈送到模型中。但是，当我们有文本数据时，我们需要将其转换为数字表示，以便我们可以比较这些表示。

2.5 数值特征向量

在处理文本的机器学习领域，我们将特征表示为数值向量。我们可以将我们表格特征数据中的每一行视为一个向量。一组特征，或我们的表格表示，就是一个矩阵。例如，在我们的第一个用户的向量中，[012, 2, 'US', 5]，我们可以看到这个特定值由四个特征表示。当我们创建向量时，我们可以对它们进行数学计算，并将它们作为输入馈送到我们需要的数值形式的 ML 模型中。

数学上，向量是一组坐标，告诉我们一个给定点在空间中的位置在许多维度中。例如，在二维空间中，我们有一个点 [2, 5]，表示 `bird_posts` 和 `bird_likes`。

在三维空间中，有三个特征，包括鸟 ID，我们将有一个向量

$$[12 \ 2 \ 5] \quad (7)$$

它告诉我们该用户在所有三个轴上的位置。

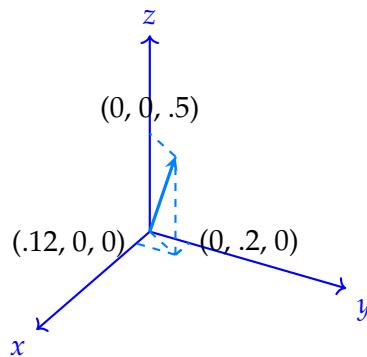


Figure 19: 将向量投影到三维空间中

但是我们如何在这个空间中表示“US”或“UK”？因为现代模型通过对矩阵执行操作收敛 [35]，我们需要将地理位置编码为某种数值，以便模型可以将其计算为输入¹⁴。因此，一旦我们有一组向量的组合，我们就可以将其与其他点进行比较。所以在我们的情况下，数据的每一行告诉我们如何将每只鸟相对于其他给定鸟的位置基于特征的组合。这确实是我们的数值特征允许我们做的事情。

¹⁴有些模型，特别是决策树，不需要进行文本编码，因为树可以直接学习分类变量，但是实现有所不同，例如最流行的两个实现，scikit-learn 和 XGBoost [1]，不能做到这一点。

2.6 从词语到向量的三个基本要素

在《Operating Systems: Three Easy Pieces》中，作者写道：“与任何由人类构建的系统一样，操作系统中的好思想随着时间的推移而积累，工程师们学会了设计中的重要性。”^[3]同样地，今天的大型语言模型也是基于数十年间数百个基础思想的累积而建立起来的。同样地，构成将单词转换为数值表示的工作的几个基本概念一次又一次地出现在每个深度学习架构和每个与 NLP 相关的任务中¹⁵：

- 编码 - 我们需要将我们的非数值、多模态数据表示为数字，以便我们可以基于它们创建模型。有很多不同的方法来做这件事。
- 向量 - 我们需要一种存储我们已编码数据并能够在其上以优化的方式执行数学函数的方法。我们通常将编码存储为向量，通常是浮点表示。
- 查找矩阵 - 在很多情况下，我们从编码和embeddings方法中得到的最终结果是对我们的文本的形状和格式的某种近似，我们需要能够快速地在大量文本中将数字转换为单词表示。因此，我们使用查找表，也称为哈希表，也称为注意力，来帮助我们在单词和数字之间进行映射。

随着我们逐步深入文档的历史背景，我们将从编码到 BERT 等更高级内容构建我们的直觉¹⁶。随着我们深入文档，我们会发现每个概念的解释会越来越简短，因为我们已经在开始时理解了构建模块的基本工作。

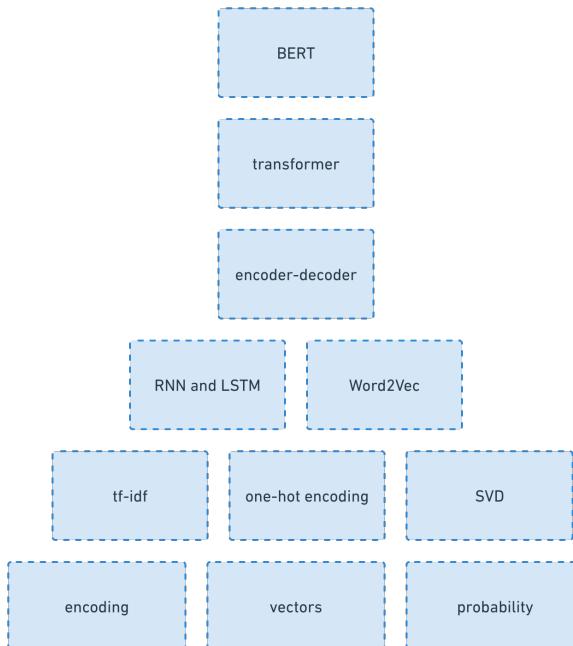


Figure 20: 从基本概念到 BERT 的金字塔

¹⁵当我们在 NLP 基于机器学习的任务中谈论任务时，我们的意思非常明确，即机器学习问题是如何制定的。例如，我们有排名、推荐、翻译、文本摘要等任务。

¹⁶原始图表来自 [关于 BERT 的这篇优秀指南](#)

3 历史编码方法

将内容压缩到更低维度以获取紧凑的数值表示并进行计算并不是一个新的想法。自从人类被信息淹没以来，我们一直在试图合成它，以便我们可以基于它做出决策。早期的方法包括一热编码、TF-IDF、词袋、LSA 和 LDA。

早期的方法是基于计数的方法。它们专注于计算单词相对于其他单词出现的次数，并基于此生成编码。LDA 和 LSA 可以被认为是统计方法，但它们仍然关注通过启发式方法推断数据集的属性，而不是建模。后来出现的基于预测的方法，通过支持向量机、Word2Vec、BERT 和 GPT 系列等模型，学习给定文本的属性。

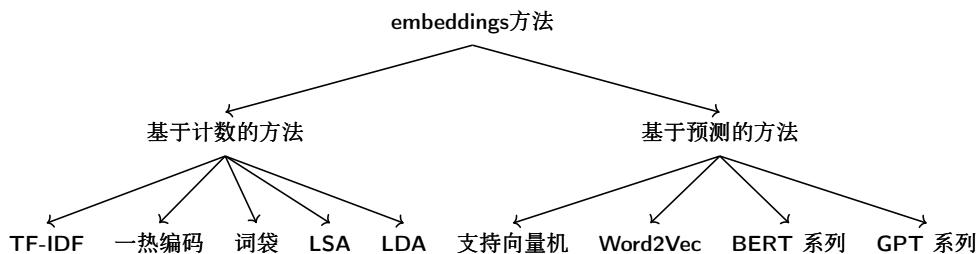


Figure 21: *embeddings*方法解决空间

关于代码的说明 在以编程方式研究这些方法时，我们将首先使用 `scikit-learn`，这是用于较小数据集的事实标准机器学习库，一些实现则是在原生 Python 中，以便更清楚地理解 `scikit-learn` 包装的功能。随着我们进入深度学习领域，我们将转向 PyTorch，这是一个深度学习库，它正在迅速成为深度学习实现的行业标准。有很多不同的方法来实现我们在这里讨论的概念，这些只是使用 Python 的 ML 通用语言库来说明的最简单的方法。

3.1 早期方法

生成文本特征的第一种方法是基于计数的，依赖于简单的计数或对统计属性的高级理解：它们是描述性的，而不是模型，模型是预测性的，试图根据一组输入值猜测一个值。最初的方法是编码方法，是 `embeddings` 的前身。编码通常是数据准备的第一阶段，用于更复杂的建模方法的输入。有几种方法可以使用称为编码的过程来创建文本特征，以便我们可以将地理特征映射到向量空间中：

- 序数编码
- 指示器编码
- 一热编码

在所有这些情况下，我们所做的是创建一个新特征，它映射到文本特征列，但是是变量的数值表示，以便我们可以将其投影到该空间以用于建模目的。我们将通过来自 `scikit-learn` 的简单代码片段来说明这些示例，该库是演示基本 ML 概念的最常见库。我们将从基于计数的方法开始。

3.2 编码

序数编码 让我们再次回到我们的 flit 数据集。我们使用连续的数字对数据进行编码。例如，“1”是“finch”，“2”是“bluejay”，依此类推。只有在变量之间存在自然排序关系时，我们才能使用此方法。例如，在这种情况下，“bluejay”不是“比”“finch”更多的，因此在我们的模型中会错误地表示。如果在我们的 flit 数据中，我们将“US”编码为 1，将“UK”编码为 2，情况也是一样的。

Table 3: 鸟类地理位置编码

bird_id	bird_posts	bird_geo	bird_likes	enc_bird_geo
012	2	US	5	2
013	0	UK	4	1
056	57	NZ	70	0
612	0	UK	120	1

```
1 from sklearn.preprocessing import OrdinalEncoder
2
3 data = [['US'], ['UK'], ['NZ']]
4 >>> print(data)
5 [['US']
6 ['UK']
7 ['NZ']]]
8
9 # our label features
10 encoder = OrdinalEncoder()
11 result = encoder.fit_transform(data)
12 >>> print(result)
13 [[2.]
14 [1.]
15 [0.]]
```

Figure 22: Scikit-Learn 中的序数编码 [来源](#)

3.2.1 指示器和独热编码

指示器编码（Indicator encoding）将 n 个类别（例如“US”、“UK”和“NZ”）编码为 $n - 1$ 个类别，为每个类别创建一个新特征。那么为什么要这样做呢？因为通常情况下，如果类别是相互独立的，如时间点的地理位置估计，如果有人在美国，我们可以确定他们不在英国也不在新西兰，因此可以减少计算开销。

如果我们使用所有变量，并且它们之间非常密切相关，就有可能会陷入所谓的指示变量陷阱。我们可以从其他变量预测一个变量，这意味着我们不再具有特征的独立性。对于地理位置来说，通常不会出现这种风险，因为地理位置超过 2 或 3 个，如果不在美国，则不能保证在英国。因此，如果我们有 $US = 1$ 、 $UK = 2$ 和 $NZ = 3$ ，并且更喜欢更紧凑的表示，我们可以使用指示器编码。然而，许多现代机器学习方法不要求线性特征独立性，并使用 L1 正则

化¹⁷来修剪不最小化误差的特征输入，因此只使用独热编码。

独热编码是基于计数的方法中最常用的方法。此过程为我们拥有的每个特征创建一个新变量。在句子的每个存在的位置，我们都放置一个“1”在向量中。我们正在创建特征空间中所有元素的映射，其中 0 表示不匹配，1 表示匹配，并比较这些向量的相似性。

```
1 from sklearn.preprocessing import OneHotEncoder
2 import numpy as np
3
4 enc = OneHotEncoder(handle_unknown='ignore')
5 data = np.asarray([['US'], ['UK'], ['NZ']])
6 enc.fit(data)
7 enc.categories_
8 >>> [array(['NZ', 'UK', 'US'], dtype='|<U2')]
9 onehotlabels = enc.transform(data).toarray()
10 onehotlabels
11 >>>
12 array([[0., 0., 1.],
13        [0., 1., 0.],
14        [1., 0., 0.]])
```

Figure 23: scikit-learn 中的独热编码[来源](#)

Table 4: 我们的独热编码数据与标签

bird_id	US	UK	NZ
012	1	0	0
013	0	1	0
056	0	0	1

现在，我们已经将文本特征编码为向量，我们可以将它们输入到我们正在开发的模型中以预测流失。我们学习的函数将通过预测每个特征的正确参数来最小化模型的损失，或者说模型的预测与实际值之间的距离。然后，学习的模型将返回一个介于 1 和 0 之间的值，表示特定鸟类的事件发生的概率，即流失或不流失。由于这是一个监督模型，我们通过将测试数据输入模型并将模型的预测与实际数据进行比较来评估模型的准确性，这告诉我们鸟类是否已流失。

我们构建的是一个标准的逻辑回归模型。一般来说，现在的机器学习社区已经开始使用梯度提升决策树方法处理表格数据，但我们将看到神经网络是建立在简单的线性和逻辑回归模型之上生成输出的，因此这是一个很好的起点。

将embeddings作为更大的特征输入

一旦我们对特征数据进行了编码，我们就可以将此输入用于接受表格特征的任何类型的模型。在我们的机器学习任务中，我们希望输出表明鸟类是否可

¹⁷正则化是一种防止我们的模型过度拟合的方法。过度拟合意味着我们的模型可以根据训练数据准确预测结果，但不能学习新的输入，这意味着它无法泛化

能基于其位置和一些使用数据离开平台。现在，我们想专门聚焦于展示与用户已经互动过的其他 **flit** 相似的 **flit**，因此我们需要用户或内容的特征表示。

让我们回到我们在本文开头提出的原始业务问题：鉴于我们知道用户消费过的过去内容（即喜欢和分享），我们如何为 Flutter 用户推荐有趣的新内容？

在传统的协同过滤推荐方法中，我们首先基于我们的输入数据构建一个用户-项目矩阵，当进行因子化时，我们可以得到每个 **flit** 的潜在属性，并允许我们推荐相似的 **flit**。

在我们的情况下，我们有 Flutter 用户可能喜欢的给定 **flit**。在考虑到该 **flit** 的文本属性后，我们会推荐哪些其他 **flit**？

下面是一个例子。我们有一条我们的鸟用户喜欢的 **flit**。

"Hold fast to dreams, for if dreams die, life is a broken-winged bird that cannot fly."

我们还有其他一些我们可能或可能不希望展示在我们鸟的 feed 中的 **flit**。

"No bird soars too high if he soars with his own wings."

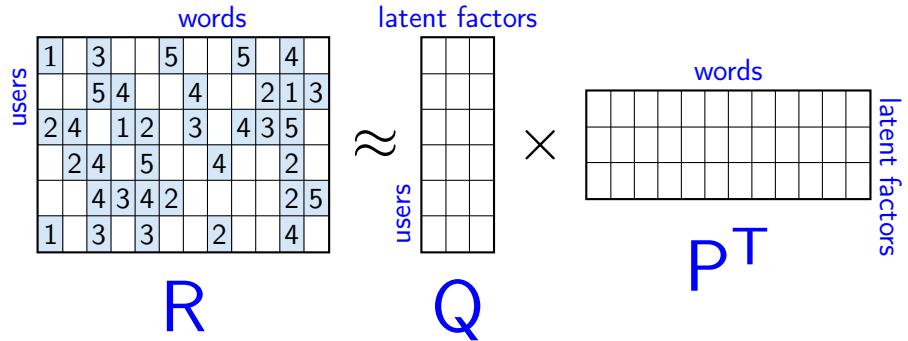
"A bird does not sing because it has an answer, it sings because it has a song."

我们如何将这转换为一个机器学习问题，它以特征作为输入，并以预测作为输出，考虑到我们已经知道如何做到这一点？首先，为了构建此矩阵，我们需要将每个单词转换为一个列值特征，每个用户保持为一个行值。

在 tabular 和自由形式的模型输入表示之间的最佳区别方式是，tabular 数据的一行看起来像这样，`[012, 2, "US", 5]`，而文本数据的一行或文档看起来像这样，`["No bird soars too high if he soars with his own wings."]` 在两种情况下，这些都是向量，或者说是表示单个鸟的值列表。

在传统的机器学习中，行是关于单个鸟的用户数据，列是关于鸟的特征。在推荐系统中，我们的行是关于每个用户的单个数据，我们的列数据代表关于每个 **flit** 的给定数据。如果我们可以对此矩阵进行因子化，即将其分解为两个矩阵 (Q 和 P^T)，当相乘时，乘积就是我们的原始矩阵 (R)，我们可以学习到“潜在因子”，即允许我们将相似的用户和项目分组以推荐它们的特征。

另一种思考这个问题的方法是，在传统的 ML 中，我们必须积极地工程特征，但然后它们作为矩阵对我们可用。在文本和深度学习方法中，我们不需要进行特征工程，但需要执行额外的步骤来生成有价值的数值特征。



我们的特征矩阵的因子化成这两个矩阵，其中 Q 中的行实际上是用户的embeddings[38]，矩阵 P 中的行是 flit 的embeddings，允许我们填充 Flutter 用户尚未明确喜欢的 flit 的值，然后在矩阵中进行搜索，找到他们可能感兴趣的其他单词。最终结果是我们生成的推荐候选项，然后我们在下游对其进行过滤和展示给用户，因为推荐问题的核心是向用户推荐项目。

在这种基本情况下，每一列都可以是我们拥有的每个 flit 的整个词汇表中的单词，并且我们创建的向量，如矩阵频率表所示，将是一个非常大且稀疏的向量，其中包含我们词汇表中的词的出现 0。我们可以朝着这个表示构建的方法是从一个称为词袋的结构开始，或者简单地说，是给定文档中文本出现的频率（在我们的情况下，每个 flit 是一个文档）。该矩阵是许多embeddings的早期方法的输入数据结构。

在 scikit-learn 中，我们可以使用 ‘CountVectorizer’ 创建我们文档之间的初始输入矩阵。

```

1 from sklearn.feature_extraction.text import CountVectorizer
2 import pandas as pd
3
4 vect = CountVectorizer(binary=True)
5 vects = vect.fit_transform(flists)
6
7 responses = ["Hold fast to dreams, for if dreams die, life is a broken-winged
    ↵ bird that cannot fly.", "No bird soars too high if he soars with his own
    ↵ wings.", "A bird does not sing because it has an answer, it sings because
    ↵ it has a song."]
8
9 doc = pd.DataFrame(list(zip(responses)))
10
11 td = pd.DataFrame(vects.todense()).iloc[:5]
12 td.columns = vect.get_feature_names_out()
13 term_document_matrix = td.T
14 term_document_matrix.columns = ['flit '+str(i) for i in range(1, 4)]
15 term_document_matrix['total_count'] = term_document_matrix.sum(axis=1)
16
17 print(term_document_matrix.drop(columns=['total_count']).head(10))
18
19         flit_1  flit_2  flit_3
20 an          0      0      1
21 answer       0      0      1
22 because      0      0      1
23 bird          1      1      1
24 broken        1      0      0
25 cannot        1      0      0
26 die           1      0      0
27 does          0      0      1
28 dreams        1      0      0
29 fast           1      0      0
30

```

Figure 24: 创建一个文档-词频率表以创建用户-项目矩阵 [来源](#)

3.2.2 TF-IDF

独热编码只处理单个文档中单个术语的存在与否。然而，当我们有大量数据时，我们希望考虑每个术语在文档集合中与其他所有术语的权重。

为了解决独热编码的局限性，TF-IDF（词项频率-逆文档频率）被开发出来。TF-IDF是上世纪70年代引入的¹⁸，它通过对文档的所有单词权重进行平均来创建文档的向量表示。在许多情况下，它的表现非常好。例如，最常用的搜索功能之一，BM25，使用TF-IDF作为基线[49]，作为Elasticsearch/OpenSearch中的默认搜索策略¹⁹。它将TF-IDF扩展到开发与文档中每对单词相关的相关性概率，并且在当前的神经搜索中仍在应用[56]。

¹⁸由Karen Spärck Jones提出，她的论文“[Synonymy and semantic classification](#)”对自然语言处理领域具有重要意义

¹⁹您可以在此处了解Elasticsearch如何实现BM25[here](#)

TF-IDF将告诉您单词在语料库中的重要性，为其分配权重，并同时降低常见单词（例如，“a”、“and”和“the”）的权重。这个计算出的权重为我们提供了单个单词TF-IDF的特征，以及词汇表中各个特征的相关性。

我们将所有结构化为句子的输入数据拆分为单个单词，并对其值执行计数，生成词袋。TF是术语频率，或术语在文档中出现的次数相对于文档中其他术语的数量。

$$\text{tf}(t, d) = \frac{f_{t,d}}{\sum_{t' \in d} f_{t',d}} \quad (8)$$

而IDF是术语在词汇表中所有文档中的逆文档频率。

$$\text{idf}(t, D) = \log \frac{N}{|\{d \in D : t \in d\}|} \quad (9)$$

让我们看一下如何从头开始实现它：

```

1
2 import math
3
4 # 将文档处理为单个单词
5 documentA = ['Hold', 'fast', 'to', 'dreams', 'for', 'if', 'dreams', 'die',
   ↪ , 'life', 'is', 'a', 'broken-winged', 'bird', 'that', 'cannot', 'fly']
6 documentB = ['No', 'bird', 'soars', 'too', 'high', 'if',
   ↪ 'he', 'soars', 'with', 'his', 'own', 'wings']
7
8 def tf(doc_dict: dict, doc_elements: list[str]) -> dict:
9     """单词在文档中的术语频率，除以文档中的总词数"""
10    tf_dict = {}
11    corpus_count = len(doc_elements)
12    for word, count in doc_dict.items():
13        tf_dict[word] = count / float(corpus_count)
14    return tf_dict
15
16 def idf(doc_list: list[str]) -> dict:
17     """每个术语在文档中出现的文档数"""
18     idf_dict = {}
19     N = len(doc_list)
20     idf_dict = dict.fromkeys(doc_list[0].keys(), 0)
21     for word, val in idf_dict.items():
22         idf_dict[word] = math.log10(N / (float(val) + 1))
23     return idf_dict
24
25 # 所有单词的逆文档频率
26 # 字典是每个文档中单词的频率计数，例如dict.fromkeys(corpus, 0)
27 idfs = idf([dict_a, dict_b])
28
29 def tfidf(doc_elements: list[str], idfs)-> dict:
30     """给定单词和术语出现在多少个文档中，计算TF * IDF"""
31     tfidf_dict = {}
32     for word, val in doc_elements.items():
33         tfidf_dict[word] = val * idfs[word]
34     return tfidf_dict
35
36 # 分别计算每个文档的术语频率
37 tf_a = tf(dict_a, document_a)
38 tf_b = tf(dict_b, document_b)
39
40 # 给定每个术语频率，计算逆文档频率
41 tfidf_a = tfidf(tf_a, idfs)
42 tfidf_b = tfidf(tf_b, idfs)
43
44 # 返回每个文档中每个单词的权重
45 document_tfidf = pd.DataFrame([tfidf_a, tfidf_b])
46 document_tfidf.T
47 #          doc 0      doc 1
48 a          0.018814  0.000000
49 dreams    0.037629  0.000000
50 No         0.000000  0.025086

```

Figure 25: 截断的TF-IDF实现，查看完整的实现[source](#)

一旦我们理解了底层的基本概念，我们就可以使用scikit-learn实现，它做了相同的事情，并且还展示了词汇表中每个单词的TF-IDF。

```
1 from sklearn.feature_extraction.text import TfidfVectorizer
2 corpus = [
3
4
5     "Hold fast to dreams, for if dreams die, life is a broken-winged bird that
6         → cannot fly.",
7     "No bird soars too high if he soars with his own wings.",
8 ]
9 text_titles = ["quote_langstonhughes", "quote_william_blake"]
10
11 vectorizer = TfidfVectorizer()
12 vector = vectorizer.fit_transform(corpus)
13 dict(zip(vectorizer.get_feature_names_out(), vector.toarray()[0]))
14
15 tfidf_df = pd.DataFrame(vector.toarray(), index=text_titles,
16     → columns=vectorizer.get_feature_names_out())
17
18 tfidf_df['00_Document Frequency'] = (tfidf_df > 0).sum()
19 tfidf_df.T
20
21 # 单词在给定文档中的普遍性或独特性, 相对于词汇表
22 dreams_langstonhughes    quote_william_blake    00_Document Frequency
23 bird                      0.172503                0.197242            2.0
24 broken                     0.242447                0.000000            1.0
25 cannot                     0.242447                0.000000            1.0
26 die                        0.242447                0.000000            1.0
```

Figure 26: 在scikit-learn中实现TF-IDF *source*

由于逆文档频率是衡量单词在文档中的普遍性或独特性的指标，我们可以看到“dreams”很重要，因为它在文档中很少出现，因此对我们来说比“bird”更有趣。我们可以看到每个实现中给定单词的tf-idf稍有不同，这是因为Scikit-learn对分母进行了归一化，并使用了稍微不同的公式。您还会注意到，在第一个实现中，我们自己分隔了语料库中的单词，没有删除任何停用词，并且没有将所有内容转换为小写。这些步骤在scikit-learn中是自动完成的，或者可以作为参数设置到处理管道中。我们稍后会看到，这些是每次处理文本时执行的关键NLP步骤。

TF-IDF对我们的文本语料库施加了几个重要的排序规则：

- 当术语在少数文档中多次出现时，提高术语频率
- 当术语在许多文档中多次出现时，降低术语频率，也称为不相关
- 当术语出现在整个文档库中时，真的会降低术语频率[49]。

有许多计算和创建TF-IDF中单个单词权重的方法。在每种情况下，我们计算每个单词的得分，以告诉我们在我们的语料库中其他单词中，该单词的重要性如何，从而给它一个权重。一旦我们弄清楚了每个单词在所有可能

的flit中的出现频率，以及对整个句子进行了加权得分，我们就可以获得向量化的句子，并创建每个引用的向量以评估它们的相似性。

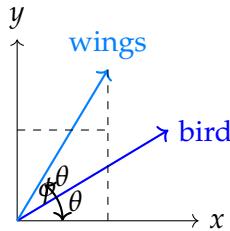


Figure 27: 两个向量之间的余弦相似度的示意图

让我们看看余弦相似度的实际方程。我们从两个向量的点积开始，这只是它们的每个值与第二个向量中相应值的乘积之和，然后我们除以归一化的点积。

```

1 v1 = [0,3,4,5,6]
2 v2 = [4,5,6,7,8]
3
4 def dot(v1, v2):
5     dot_product = sum((a * b) for a,b in zip(v1,v2))
6     return dot_product
7
8 def cosine_similarity(v1, v2):
9     ...
10    (v1 dot v2)/||v1|| *||v2||)
11    ...
12    products = dot(v1,v2)
13    denominator = ( (dot(v1,v1) **.5) * (dot(v2,v2) ** .5) )
14    similarity = products / denominator
15    return similarity
16
17 print(cosine_similarity(v1, v2))
18 # 0.9544074144996451

```

Figure 28: 从头开始实现余弦相似度 *source*

或者，再次在 scikit-learn 中，作为成对度量：

```

1 from sklearn.metrics import pairwise
2
3 v1 = [0,3,4,5,6]
4 v2 = [4,5,6,7,8]
5
6 # 需要以numpy数据格式
7 pairwise.cosine_similarity([v1],[v2])
8 # array([[0.95440741]])
9

```

Figure 29: 在scikit-learn中实现余弦相似度 *source*

在语义相似性和推荐中，其他常用的距离度量包括：

- 欧几里得距离 - 计算两点之间的直线距离
- 曼哈顿距离 - 通过求坐标的绝对差的和来测量两点之间的距离
- 杰卡德距离 - 通过将它们的交集大小除以它们的并集大小来计算两个集合之间的不相似性
- 海明距离 - 通过计算两个字符串在它们不同的位置上的个数来测量两个字符串之间的不相似性

3.2.3 SVD 和 PCA

我们在使用独热编码和 TF-IDF 创建的向量存在一个问题：它们是稀疏的。稀疏向量是指大部分元素都是零的向量。它们是稀疏的，因为大多数句子不包含与其他句子相同的所有单词。例如，在我们的 flit 中，可能同时在两个句子中遇到单词 "bird"，但其余单词完全不同。

```

1 sparse_vector = [1,0,0,0,0,0,0,0,0]
2 dense_vector = [1,2,2,3,0,4,5,8,8,5]

```

Figure 30: 文本处理中的两种向量类型

稀疏向量导致了一些问题，其中包括冷启动—即我们不知道推荐未被互动的项目，或者对于新用户。相反，我们希望创建密集向量，它将为我们提供有关数据的更多信息，其中最重要的是考虑到给定单词相对于其他单词的权重。这就是我们离开独热编码和 TD-IDF 转向旨在解决这种稀疏性的方法。密集向量只是具有大多数非零值的向量。我们将这些密集表示称为动态表示[59]。

除了 TF-IDF 外，还使用了几种其他相关的早期方法来创建项目的紧凑表示：主成分分析（PCA）和奇异值分解（SVD）。

SVD 和 PCA 都是降维技术，通过矩阵转换应用于我们原始的文本输入数据，通过矩阵变换将项目分解成潜在组件，从而显示两个项目之间的潜在关系。

SVD 是一种矩阵分解类型，将给定的输入特征矩阵表示为三个矩阵的乘积。然后，它使用组件矩阵创建彼此之间最大差异的线性组合，这些组合

基于给定线的点群的方差而方向不同。这些群集代表了压缩特征的“特征群集”。

在执行 SVD 并分解这些矩阵的过程中，我们生成了一个包括特征向量和特征值对或样本协方差对的矩阵表示。

PCA 使用相同的初始输入特征矩阵，但是，与独热编码仅将文本特征转换为可以处理的数值特征不同，PCA 还执行了压缩并将我们的项目投影到二维特征空间中。第一个主成分是数据的缩放后的特征向量，最能描述您的数据的变量的权重，第二个主成分是最能描述您的数据的下一组变量的权重。

结果模型是所有单词的投影，根据这些维度聚合到单个空间中的。虽然我们不能获得所有这些组件的单个含义，但很明显，单词群集，也就是特征，是语义上相似的，也就是它们在含义上彼此接近²⁰。

两者之间的区别经常令人困惑（人们在 80 年代还在解决这些方法时承认了这一点[19]），对于本次调查论文的目的，我们将说 PCA 经常可以使用 SVD 实现²¹。

3.3 LDA 和 LSA

因为 PCA 对每一对特征执行计算以生成两个维度，所以随着特征数量的增长，它变得非常计算密集。许多早期方法，如 PCA，对于较小的数据集效果良好，比如许多传统 NLP 研究中使用的数据集，但随着数据集的继续增长，它们并不太适用。

其他方法是基于 TF-IDF 和 PCA 发展出来的，以解决它们的局限性，包括潜在语义分析（LSA）和潜在狄利克雷分配（LDA）[11]。这两种方法的基本原理都是，出现在彼此附近的单词具有更重要的关系。LSA 使用与 TF-IDF 相同的单词加权，试图将该矩阵组合成一个较低秩的矩阵，即余弦相似度矩阵。在矩阵中，单元格的值范围是[-1,1]，其中-1 表示完全相反的文档，而 1 表示文档是相同的。LSA 然后在矩阵上运行并将项目分组在一起。

LDA 采用稍微不同的方法。虽然它使用相同的输入矩阵，但它输出的矩阵的行是单词，列是文档。距离度量，而不是余弦相似度，是单词和文档交集提供的主题的数值值。假设是，我们输入的任何句子都包含一组主题，这取决于相对于输入语料库的表示比例，而且有一些主题可以用来对给定的句子进行分类。我们通过假设每个单词都有可能出现在一个主题中来初始化算法。LDA 最初将单词随机分配到主题中，然后迭代直到收敛到使当前单词分配到当前主题的概率最大化的点。为了进行单词到主题的映射，LDA 生成了一个embeddings，创建了一个词汇或句子集合的空间，这些词汇或句子在语义上一起工作。

3.4 传统方法的局限性

所有这些传统方法都试图以不同的方式在潜在空间中生成我们语料库中项目之间的关系 - 单词之间未明确说明的关系，但我们可以根据如何对数据进行建模来推断出来的关系。

然而，在所有这些情况下，随着我们的语料库的增长，我们开始遇到两个问题：维数灾难和计算规模。

²⁰语义相似性有许多定义 - “king” 和 “queen” 之间的接近意味着什么？ - 但是一个高层次的方法涉及使用词典和词库等原始来源来创建一个结构化的知识库，并基于节点和边缘（即它们多频地在一起出现的频率）提供术语和概念的结构化表示。

²¹这是在 scikit-learn 软件包中实现的方式

3.4.1 维数灾难

随着我们对更多特征进行独热编码，我们的表格数据集变得越来越大。回到我们的流失模型，一旦我们有了 181 个国家而不是两三个呢？我们将不得不将每个国家都编码为其自己的向量表示。例如，如果我们有数百万个词汇，例如成千上万只鸟每天发布数百万条消息怎么办？我们的 tf-idf 稀疏矩阵变得计算密集，以至于难以因子化。

而我们用于表格机器学习和朴素文本方法的输入向量只有三个条目，因为我们只使用了三个特征，多模态数据的实际维数是存在的单词数量，图像数据的实际维数是像素的高度乘以宽度，每个给定图像。视频和音频数据具有类似的指数属性。我们可以使用大 **O** 表示法来评估我们编写的任何代码的性能，这将对算法的运行时间进行分类。根据程序处理的元素数量，有些程序性能更差，有些性能更好。这意味着在计算性能方面，独热编码在最坏情况下的复杂度是 $O(n)$ 。因此，如果我们的文本是一个包含一百万个独特词汇的语料库，我们将得到一百万列或向量，其中每个都将是稀疏的，因为大多数句子不会包含其他句子的单词。

让我们以更具体的案例来说明。即使在我们初始的简单案例中，即我们的初始鸟类引用中，我们有 28 个特征，每个特征对应句子中的一个单词，假设我们不删除和处理最常见的停用词 - 非常常见的词，如“the”、“who”和“is”，它们出现在大多数文本中，但不添加语义含义。我们如何创建一个具有 28 个特征的模型？如果我们每个单词都编码为一个数字值，这其实相当简单但又繁琐。

Table 5: 独热编码及其对我们的 *flit* 增长维度灾难的影响

flit_id	bird_id	hold	fast	dreams	die	life	bird
9823420	012	1	1	1	1	1	1
9823421	013	1	0	0	0	0	1

一旦我们开始生成大量特征（列），我们开始遇到维数灾难，这意味着我们积累的特征越多，我们就需要更多的数据才能准确地统计上对它们做出任何说明，这导致模型可能不会准确地代表我们的数据[26]，特别是在推荐中用户/项目互动中特征非常稀疏的情况下。

3.4.2 计算复杂性

在生产机器学习系统中，我们算法的统计属性很重要。但同样关键的是我们的模型返回数据的速度，或者系统的效率。系统效率可以用许多方式来衡量，而且在任何表现良好的系统中找到导致延迟的性能瓶颈是至关重要的，或者在执行操作之前花费的时间[23]。如果您在生产中拥有推荐系统，您不能冒着向用户显示空白或需要花费几毫秒以上时间才能渲染的推荐流的风险。如果您有一个搜索系统，在电子商务环境中特别是您不能冒险结果需要花费几毫秒以上的时间才能返回[2]。因此，从整体系统的角度来看，生成数据的时间，读取数据和训练模型也可能存在延迟。

导致延迟的两个主要驱动因素是：

- I/O 处理 - 我们只能以网络速度允许的速度发送项目
- CPU 处理 - 我们只能处理我们在任何给定系统中可用内存的项目数量²²

²²在过去几年中关于现代数据密集型应用程序中 IO 还是 CPU 瓶颈的讨论。

一般来说，TF-IDF 在识别文档中的关键术语方面表现良好。但是，由于该算法处理给定语料库中的所有元素，所以在等式的分子和分母中，时间复杂度都会增加，总体上，计算所有文档中所有术语的 TF-IDF 权重的时间复杂度为 $O(Nd)$ ，其中 N 是语料库中术语的总数， d 是语料库中的文档数。另外，由于 TF-IDF 输出一个矩阵，我们最终要做的是处理巨大的状态矩阵。例如，如果您有大约 100k 个文档，并且需要存储出现在这些文档中前五千个单词的频率计数和特征，我们将得到一个大小为 $100000 * 5000$ 的矩阵。这种复杂性只会增长。

这种线性时间复杂度的增长在我们试图处理数百万或数亿个标记时成为问题 - 通常是单词的同义词，但也可以是音节等子词。这是一个随着时间推移在行业中变得尤为普遍的问题，存储变得便宜的情况下。

从新闻组到电子邮件，最终到公共互联网文本，我们开始产生大量的数字废料，公司以追加记录的形式收集它们，这是按时间顺序排序的一系列记录，配置为连续追加记录。²³

公司开始发出、保留和使用这些无休止的日志流进行数据分析和机器学习。突然之间，在不到一百万个文档的集合中表现良好的算法开始跟不上。

在规模上捕获日志数据开始了大数据时代的兴起，这导致了数据移动的大量变化、速度和数量。数据量的增加与数据存储变得更便宜同时发生，使公司能够在大量的商用硬件上存储他们收集的所有数据。

公司已经在关系数据库中保留了运行关键业务操作所需的分析数据，但对该数据的访问是结构化的，并且以每天或每周批量增量方式处理。这种新的日志文件数据移动迅速，并且具有传统数据库所缺乏的多样性水平。

由此产生的 NLP、搜索和推荐问题的语料库也爆炸性增长，导致人们寻找更高性能的解决方案。

3.5 支持向量机

最初的建模方法是浅层模型—只使用一层权重和偏置执行机器学习任务的模型[8]。支持向量机 (SVM) 是在上世纪90年代中期在贝尔实验室开发的，被用于高维空间的自然语言处理任务，比如文本分类[29]。SVM将数据集群分离成线性可分的点通过一个超平面，这是一个决策边界，将元素分隔成不同的类别。在二维向量空间中，超平面是一条线，在三维或更高维度空间中，分隔器也有很多维度。

SVM的目标是找到最优的超平面，使得新对象（在我们的案例中是单词）投影到空间中的距离最大化平面和元素之间的距离，从而降低错误分类的可能性。

²³Jay Kreps 关于日志如何工作的[经典文章](#)是必读的。

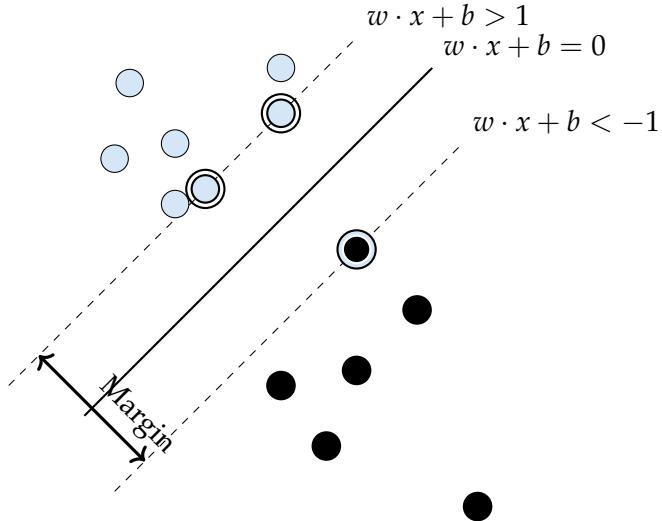


Figure 31: 在SVM中分隔向量空间中的点的示例，通过超平面分隔

使用SVM执行的监督机器学习任务的示例包括下一个单词预测、预测给定序列中缺失的单词，并预测在窗口中出现的单词。例如，经典的词embeddings推理任务是在我们手机上输入时的自动校正。我们键入一个单词，根据单词本身和句子中的周围上下文，自动校正的任务是预测正确的单词。因此，它需要学习一个embeddings词汇，为它提供选择正确单词的概率。

然而，与其他情况一样，当我们达到高维度时，SVM完全无法处理稀疏数据，因为它依赖于计算点之间的距离来确定决策边界。因为在我们的元素的稀疏向量表示中，大多数距离都是零，所以超平面将无法清晰地分隔边界并将单词错误分类。

3.6 Word2Vec

为了克服早期文本处理方法的局限性，并跟上文本语料库的增长，2013年，谷歌的研究人员提出了一种使用神经网络的解决方案，称为Word2Vec[42]。

到目前为止，我们已经从简单的启发式方法如热独编码，转向了像LSA和LDA这样的机器学习方法，这些方法旨在学习数据集的模型特征。以前，就像我们最初的热独编码一样，所有embeddings的方法都集中在生成可以指示两个单词相关性的稀疏向量上，但却没有说明它们之间存在语义关系。例如，“狗追逐猫”和“猫追逐狗”在向量空间中的距离是相同的，尽管它们是两个完全不同的句子。

Word2Vec是一个系列模型，有几种实现方式，每种方式都专注于将整个输入数据集转换为向量表示，并且更重要的是，不仅关注于个别单词的固有标签，还关注这些表示之间的关系。

Word2Vec有两种建模方法——连续词袋（CBOW）和skipgrams，两者都生成密集的embeddings向量，但对问题的建模方式略有不同。Word2Vec模型的最终目标是学习最大化给定单词或单词组成为准确预测的概率的参数[20]。

在训练skipgrams时，我们从初始输入语料库中取一个词，并预测围绕它的一组单词的概率。在我们最初的flit引用中，“Hold fast to dreams for if dreams die, life is a broken-winged bird that cannot fly”，模型的中间步骤

生成了一个embeddings的集合，即整个短语的所有词之间的距离，并使用单词“fast”作为输入，填充了整个短语的下一个几个概率。

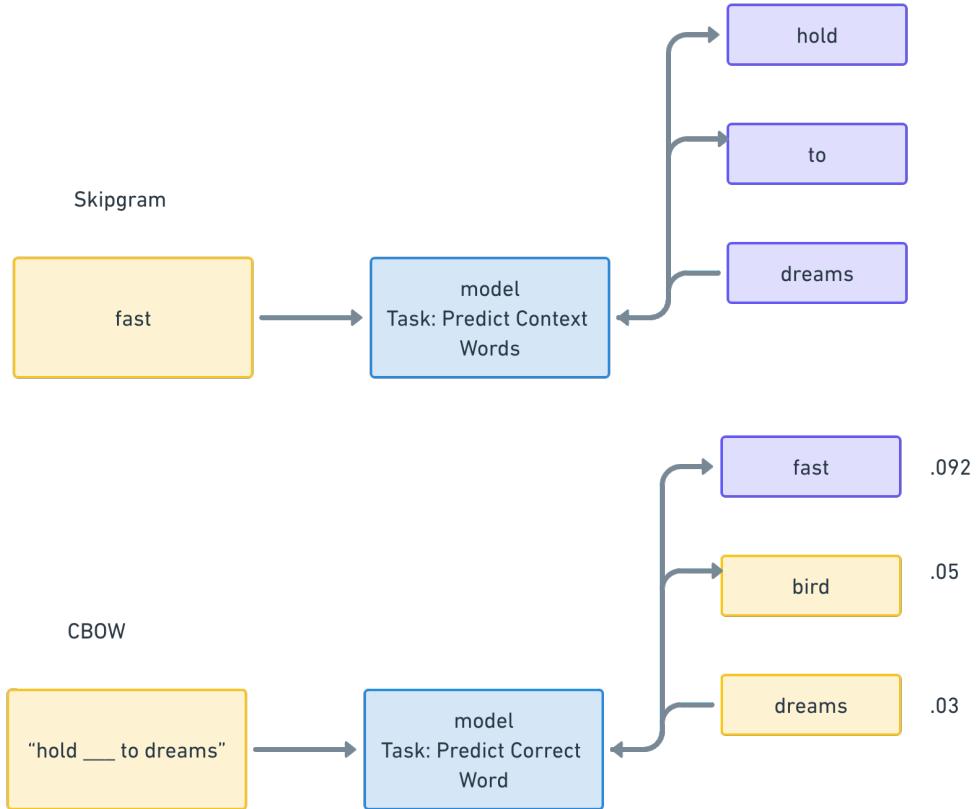


Figure 32: Word2Vec架构

在CBOW训练中，我们相反地：我们从一个被称为上下文窗口的短语中删除一个词，并训练模型来预测填充空白的给定单词的概率，如下方方程式所示，我们试图最大化。

$$\arg \max_{\theta} \prod_{w \in Text} \left[\prod_{c \in C(w)} p(c|w; \theta) \right] \quad (10)$$

如果我们优化这些参数 - theta - 并最大化单词属于句子的概率，我们将为我们的输入语料库学习到良好的embeddings。

让我们专注于CBOW的详细实现，以更好地理解它是如何工作的。这次，对于代码部分，我们将从适用于较小数据集的scikit-learn转移到PyTorch进行神经网络操作。

在高层次上，我们有一个输入词的列表，经过第二层处理，即embeddings层，然后通过输出层，这只是一个返回概率的线性模型。

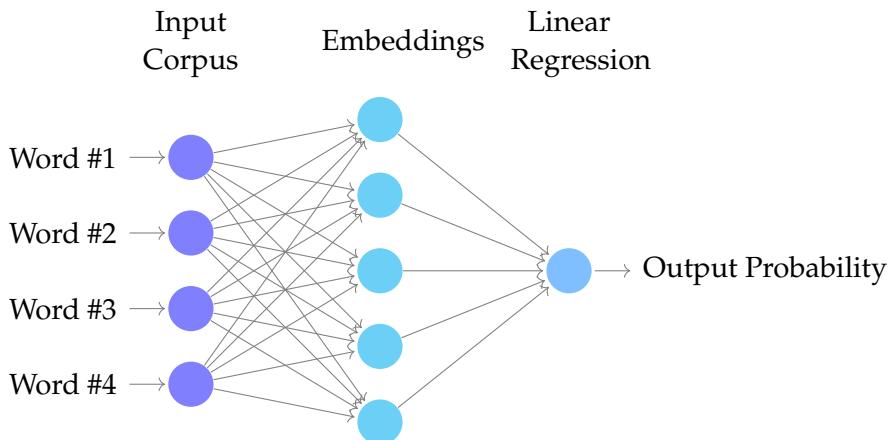


Figure 33: Word2Vec CBOW Neural Network architecture

我们将在PyTorch中运行这个实现，PyTorch是一个流行的用于构建神经网络模型的库。实现Word2Vec的最佳方法，特别是当你处理较小的数据集时，是使用Gensim，但是Gensim将层抽象成内部类，这使得用户体验非常好。但是，由于我们只是学习它们，我们想更清楚地看到它们的内部工作，而PyTorch虽然没有Word2Vec的本地实现，但让我们可以更清晰地看到内部的运作方式。

要在PyTorch中对我们的问题建模，我们将使用与任何机器学习问题相同的方法：

- 检查和清理我们的输入数据。
- 构建我们模型的层（对于传统ML，我们只有一个）。
- 将输入数据馈送到模型中并跟踪损失曲线。
- 检索经过训练的模型工件，并将其用于对我们分析的新项目进行预测。

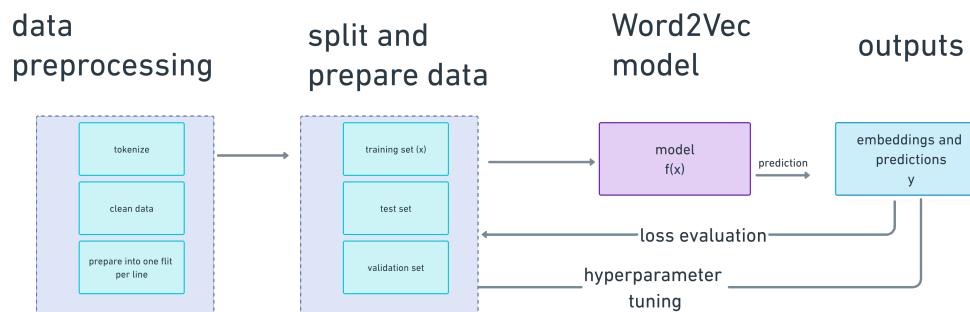


Figure 34: 创建Word2Vec模型的步骤

让我们从我们的输入数据开始。在这种情况下，我们的语料库是我们收集到的所有flit。我们首先需要将它们处理为我们模型的输入。

```
responses = ["Hold fast to dreams, for if dreams die, life is a broken-winged  
→ bird that cannot fly.", "No bird soars too high if he soars with his own  
→ wings.", "A bird does not sing because it has an answer, it sings because  
→ it has a song."]
```

Figure 35: 我们的Word2Vec输入数据集

让我们从我们的输入训练数据开始，这是我们的flit列表。为了准备PyTorch的输入数据，我们可以使用DataLoader或Vocab类，它将我们的文本拆分成标记并对其进行标记化——或创建每个句子的较小的单词级表示以进行处理。对于文件中的每一行，我们通过将每一行拆分为单个单词，去除空格和标点符号，并将每个单词转换为小写形式来生成标记。

这种处理管道在NLP中非常常见，花费时间来正确完成这一步骤非常关键，这样我们就可以得到干净、正确的输入数据。它通常包括以下步骤[43]：

- 标记化——通过拆分将句子或单词转换为其组成字符
- 去除噪音——包括URL、标点符号和文本中与手头任务无关的任何内容
- 分词——将我们的句子拆分为单个单词
- 纠正拼写错误

```
class TextPreProcessor:  
    def __init__(self) -> None:  
        self.input_file = input_file  
  
    def generate_tokens(self):  
        with open(self.input_file, encoding="utf-8") as f:  
            for line in f:  
                line = line.replace("\\\\", "")  
                yield line.strip().split()  
  
    def build_vocab(self) -> Vocab:  
        vocab = build_vocab_from_iterator(  
            self.generate_tokens(), specials=["<unk>"], min_freq=100  
        )  
        return vocab
```

Figure 36: 在PyTorch中处理我们的输入词汇表并从数据集构建一个Vocabulary对象
source

现在我们有了一个可以使用的输入词汇表对象，下一步是为每个单词创建一个独热编码到数值位置的映射，以及将每个位置返回到一个单词，这样就可以轻松地引用我们的单词和向量。我们的目标是在进行查找和检索时能够进行单词和向量之间的相互转换。

在embeddings层中进行此操作。在PyTorch的embeddings层中，我们根据我们指定的大小和词汇表的大小初始化一个embeddings矩阵，并将该层将词汇表索引为一个字典以进行检索。embeddings层是一个查找表²⁴，它

²⁴PyTorch文档中的embeddings层

按索引匹配一个单词与相应的单词向量。最初，我们创建了我们的独热编码的词语到项字典。然后，我们为每个单词创建了一个到字典条目的映射，以及一个字典条目到每个单词的映射。这被称为双射。通过这种方式，`embeddings`层类似于一个独热编码矩阵，并允许我们执行查找。这个层中的查找值被初始化为一组随机权重，接下来我们将它们传递到线性层。

`embeddings`类似于哈希映射，并且也有它们自己的性能特征 ($O(1)$ 检索和插入时间)，这就是为什么它们在其他方法无法扩展时可以轻松扩展的原因。在`embeddings`层中，Word2Vec中的每个向量值代表特定维度上的单词，更重要的是，与许多其他方法不同，每个向量的值与输入数据集中的其他单词直接相关。

```
1 class CBOW(torch.nn.Module):
2     def __init__(self): # 我们将vocab_size和embedding_dim作为超参数传入
3         super(CBOW, self).__init__()
4         self.num_epochs = 3
5         self.context_size = 2 # 从左侧选择2个单词，从右侧选择2个单词
6         self.embedding_dim = 100 # embeddings向量的大小
7         self.learning_rate = 0.001
8         self.device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
9
10        self.vocab = TextPreProcessor().build_vocab()
11        self.word_to_ix = self.vocab.get_stoi()
12        self.ix_to_word = self.vocab.get_itos()
13        self.vocab_list = list(self.vocab.get_stoi().keys())
14        self.vocab_size = len(self.vocab)
15
16        self.model = None
17
18        # 输出: 1 x embedding_dim
19        self.embeddings = nn.Embedding(
20            self.vocab_size, self.embedding_dim
21        ) # 根据我们的输入初始化一个embeddings矩阵
22        self.linear1 = nn.Linear(self.embedding_dim, 128)
23        self.activation_function1 = nn.ReLU()
24
25        # 输出: 1 x vocab_size
26        self.linear2 = nn.Linear(128, self.vocab_size)
27        self.activation_function2 = nn.LogSoftmax(dim=-1)
```

Figure 37: PyTorch中的Word2Vec CBOW实现。[source](#)

一旦我们有了查找值，我们就可以处理所有的单词。对于CBOW，我们选择一个单词，并选择一个滑动窗口，在我们的例子中，在它之前的两个单词和之后的两个单词，并尝试推断实际的单词是什么。这被称为上下文向量，在其他情况下，我们会看到它被称为注意力。例如，如果我们有短语“没有鸟[空白]太高”，我们试图预测答案是“飞翔”，并给出一个给定的softmax概率，即根据其他单词的排名。一旦我们有了上下文向量，我们查看损失——真实单词和根据概率排名的预测单词之间的差异——然后我们继续。

我们训练这个模型的方法是通过上下文窗口。对于模型中的每个给定单词，我们创建一个包含该单词和它之前的2个单词以及之后的2个单词的滑动窗口。

我们使用**ReLU**激活函数激活线性层，它决定一个给定权重是否重要。在这种情况下，ReLU将我们初始化**embeddings**层的所有

负值压缩为零，因为我们不能有逆词关系，我们通过学习模型的单词关系的权重执行线性回归。然后，对于每个批次，我们检查损失——真实单词与我们预测的应该在那里的单词之间的差异——然后我们将其最小化。

在每个时期结束时，或者通过模型的一次遍历之后，我们将权重或反向传播回线性层，然后再次根据概率更新每个单词的权重。概率是通过**softmax**函数计算的，它将一组实数向量转换为概率分布——即向量中的每个数字，即每个单词的概率值，在0和1之间的区间内，所有单词的数字加起来等于1。作为反向传播到**embeddings**表的距离应该收敛或收缩，具体取决于模型理解特定单词有多接近。

```
1     def make_context_vector(self, context, word_to_ix) -> torch.LongTensor:
2         """
3             对于词汇表中的每个单词，找到与单词位置相关的[-2,1,0,1,2]索引的滑动窗口
4             :param vocab: 词汇表中的单词列表
5             :return: torch.LongTensor
6         """
7
8         idxs = [word_to_ix[w] for w in context]
9         tensor = torch.LongTensor(idxs)
10
11
12     def train_model(self):
13
14         # 损失和优化器
15         self.model = CBOW().to(self.device)
16         optimizer = optim.Adam(self.model.parameters(), lr=self.learning_rate)
17         loss_function = nn.NLLLoss()
18
19         logging.warning('Building training data')
20         data = self.build_training_data()
21
22         logging.warning('Starting forward pass')
23         for epoch in tqdm(range(self.num_epochs)):
24             # 我们开始跟踪我们的初始单词有多准确
25             total_loss = 0
26
27             # 对于训练数据中的x, y:
28             for context, target in data:
29                 context_vector = self.make_context_vector(context, self.word_to_ix)
30
31                 # 我们看损失
32                 log_probs = self.model(context_vector)
33
34                 # 比较损失
35                 total_loss += loss_function(
36                     log_probs, torch.tensor([self.word_to_ix[target]]))
37
38             # 在每个时期结束时进行优化
39             optimizer.zero_grad()
40             total_loss.backward()
41             optimizer.step()
42
43             # 记录一些指标以查看损失是否减少
44             logging.warning("end of epoch {} | loss {:.3f}".format(epoch, total_loss))
45
46             torch.save(self.model.state_dict(), self.model_path)
47             logging.warning(f'Save model to {self.model_path}')
```

Figure 38: 在PyTorch中的Word2Vec CBOW实现
看完整实现

一旦我们完成了对训练集的迭代，我们就学会了一个模型，该模型检索给定单词是正确单词的概率，以及我们词汇表的整个embeddings空间。

4 现代embeddings方法

Word2Vec成为了第一个利用embeddings概念创建固定特征词汇表的神经网络架构之一。但整体而言，神经网络因为几个关键因素而在自然语言建模中变得越来越受欢迎。首先，在1980年代，研究人员在神经网络学习中使用反向传播技术取得了进展[47]。反向传播是模型学习通过使用链式法则来计算损失函数对神经网络权重的梯度，这是微积分中的一个概念，它允许我们计算由多个函数组成的函数的导数。这个机制使得模型能够理解何时达到了损失的全局最小值，并通过梯度下降选择正确的模型参数权重，从而收敛。早期的方法，如感知器学习规则，尝试做到这一点，但存在一些限制，比如只能处理简单的层架构，收敛需要很长时间，并且经历了消失的梯度问题，这使得有效更新模型权重变得困难。

这些进步催生了第一种多级神经网络，即前馈神经网络。1998年，一篇论文使用反向传播在多层感知机上正确执行了识别手写数字图像的任务[36]，展示了从业者和研究人员可以应用的实际用例。这个MNIST数据集现在是深度学习的标准“Hello World”示例之一。

其次，在2000年代，聚合的日志数据的增长导致了从互联网上抓取的大规模多模态输入数据的数据库的创建。这使得可能进行广泛的实验，以证明神经网络在大量数据上的工作。例如，ImageNet是由斯坦福大学的研究人员开发的，他们希望通过创建一组黄金标准的神经网络输入数据集来专注于改进模型性能，这是处理的第一步。FeiFei Li召集了一组学生和来自Amazon Turk的付费兼职工作者，正确标记了一组从互联网上抓取的320万张图像，并根据WordNet这个1970年代研究人员组织的分类法将它们组织起来[48]。

研究人员看到使用标准数据集的力量。2015年，Alex Krizhevsky与Ilya Sutskever合作，后者现在是OpenAI的一位主要研究人员，是当前生成式AI浪潮的基础，提交了一篇名为AlexNet的论文参加ImageNet竞赛。这个模型是一个卷积神经网络，超越了许多其他方法。关于AlexNet有两件事情是重要的。第一，它有八层堆叠的权重和偏置，这在当时是不寻常的。如今，像BERT和其他transformers这样的12层神经网络是完全正常的，但在当时，超过两层是革命性的。第二，它在GPU上运行，这是一个新的架构概念，因为GPU主要用于游戏。

神经网络开始变得流行，作为生成词汇表表示的方法。特别是，神经网络架构，如循环神经网络（RNN）和后来的长短期记忆网络（LSTM）也出现了，作为处理文本数据的方式，用于从自然语言处理到计算机视觉等各种机器学习任务。

4.1 神经网络

神经网络是传统机器学习模型的扩展，但它们具有一些关键的特性。让我们回顾一下我们在正式化机器学习问题时对模型的定义。模型是一个具有一组可学习输入参数的函数，它接受一组输入和一个表格化的输入特征集，并给出输出。在传统的机器学习方法中，有一个可学习参数集或层和一个模型。如果我们的数据没有复杂的交互作用，我们的模型可以相当容易地学习特征空间并做出准确的预测。

然而，当我们开始处理非常大的、隐式的特征空间，比如文本、音频或视频时，我们将无法得出如果我们手动创建它们，就不会显而易见的具体特征。通过堆叠神经元，每个神经元代表模型的某个方面，神经网络可以提取这些潜在的表示。神经网络非常擅长学习数据的表示，网络的每一层都将一个学习到的表示转换为更高层次，直到我们得到了对我们的数据的清晰认识[37]。

4.1.1 神经网络架构

我们已经遇到了我们的第一个神经网络，Word2Vec，它旨在理解文本中单词之间的关系，而这些关系单凭单词本身是无法告诉我们的。在神经网络空间中，有几种流行的架构：

- 前馈网络，从固定长度的输入中提取含义。这些模型的结果不会反馈到模型中进行迭代
- 卷积神经网络（CNNs） - 主要用于图像处理，包括由滤波器组成的卷积层，该滤波器在图像上移动以检查特征表示，然后通过点积与滤波器相乘以提取特定特征
- 循环神经网络，它接受一个项目序列并产生一个总结该序列的向量

RNNs和CNNs主要用于特征提取-它们通常不代表整个建模流程，但后来会被馈送到进行分类、摘要等最终工作的前馈模型中。

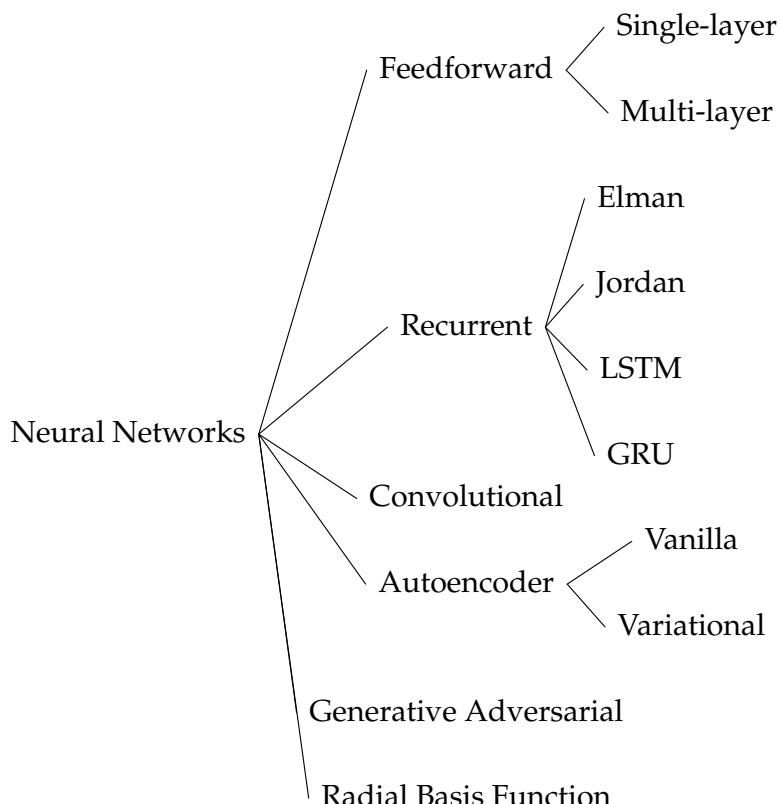


Figure 39: Types of Neural Networks

由于各种原因，构建和管理神经网络是复杂的。首先，它们需要大量的干净、标记良好的数据来进行优化。它们还需要用于处理的特殊GPU架构，正

如我们将在生产部分中看到的那样，它们有自己的元数据管理和延迟考虑。最后，在网络内部，我们需要完成大量的传递，使用我们的训练数据

的批次来使其收敛。我们需要运行计算的特征矩阵的数量²⁵，因此，我们必须在模型的生命周期中累积并需要进行大量性能调优的数据。

这些特点使得直到最近十五年左右，开发和运行神经网络成本高昂。首先，增长的商品硬件存储空间的指数增长意味着我们现在可以将数据存储起来进行计算，而日志数据的爆炸性增长给了谷歌等公司大量的训练数据来处理。其次，GPU作为利用神经网络进行尴尬并行计算的工具的崛起，尴尬并行是计算的一个特性，当将步骤容易分离成可以并行执行的步骤时，比如词频统计。在神经网络中，我们通常可以以任意数量的方式并行化计算，包括在单个神经元的级别上。

虽然GPU最初用于处理计算机图形，在21世纪初，研究人员发现了将其用于通用计算的潜力，并且英伟达通过引入CUDA（一种在GPU上的API层）对这种计算进行了巨大的投资。这反过来又促成了像PyTorch和Tensorflow这样的高级流行深度学习框架的创建和发展。

神经网络现在可以大规模地进行训练和实验。回到与以前方法的比较，当我们计算TF-IDF时，我们需要循环遍历每个单词并在整个数据集上顺序执行我们的计算，以获得与所有其他单词成比例的得分，这意味着我们的计算复杂度将是 $O(ND)$ [9]。

然而，使用神经网络，我们可以将模型训练分布在不同的GPU上，这个过程被称为模型并行，或者计算批次 - 将训练数据的大小馈送到模型中，在训练循环中使用，在每个小批量结束时并行更新其超参数并更新，这被称为数据并行[52]。

4.2 Transformers

Word2Vec是一个前馈网络。模型的权重和信息只从编码状态流向隐藏的embeddings层，再流向输出概率层。第二层和第三层之间没有反馈，这意味着每一层都不知道其后层的状态。它无法提供超出上下文窗口范围的推理建议。这在我们对单一、静态词汇量的机器学习问题中效果很好。

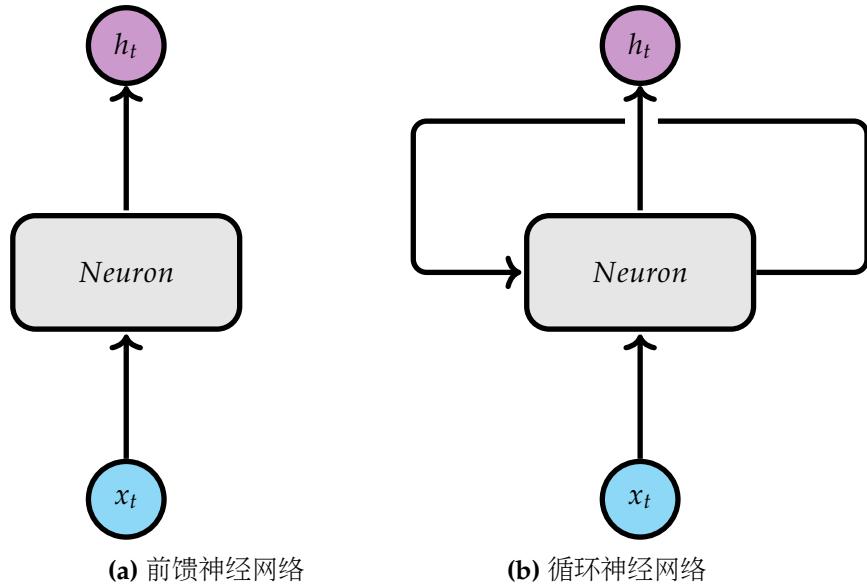
然而，在需要理解单词在彼此语境中的长距离联系的长文本上效果不佳。例如，在对话过程中，我们可能会说：“我读到了朗斯顿·休斯的那段话。我喜欢它，但并没有真正读过他的后期作品。”我们理解“它”指的是引文，前一个句子的上下文，以及“他”的指代是两个句子前提到的“朗斯顿·休斯”。

另一个限制是Word2Vec无法处理超出词汇表的单词，即模型没有经过训练需要进行泛化的单词。这意味着如果我们的用户搜索一个新的流行术语，或者我们想要推荐一个在我们的模型训练之后编写的文本，他们将无法从我们的模型中看到任何相关结果[13]，除非模型经常重新训练。

另一个问题是Word2Vec在多义性周围遇到上下文崩溃的情况——同一短语有许多可能的含义共存：例如，如果你在同一个句子中有“监狱牢房”和“手机电池”，它无法理解两个单词的上下文是不同的。基于深度学习的NLP工作的许多工作都集中在理解和保留上下文，以便通过模型传播并提取语义含义。

为了克服这些限制，提出了不同的方法。研究人员尝试使用循环神经网络(RNNs)。RNN建立在传统的前馈网络上，其区别在于模型的各层向前馈反馈到先前的层。这使得模型能够记住句子中单词周围的上下文。

²⁵没有一个很好的单一资源来计算神经网络的计算复杂度，因为存在许多广泛的架构，但是这篇文章在阐述这个案例方面做得很好，基本上是 $O(n^5)$



传统RNN的一个问题是，在反向传播期间，权重必须传递到先前层的神经元，这导致了梯度消失的问题。当我们持续地取导数时，使得反向传播中链式规则中使用的偏导数接近于零。一旦接近零，神经网络就会假设它已经达到了局部最优点，并停止训练，而不是收敛。

一种非常流行的RNN变体，可以解决这个问题，叫做长短期记忆网络（LSTM），最初由Schmidhuber开发²⁶，在文本应用、语音识别和图像字幕中广泛使用[30]。然而，虽然LSTM效果相当不错，但它们也有自己的局限性。因为它们的架构复杂，所以训练时间更长，计算成本更高，因为它们不能并行训练。

4.2.1 编码器/解码器与注意力机制

两个概念帮助研究人员克服了在RNNs和之前的Word2Vec中记住长向量的计算成本高昂的问题：编码器/解码器架构和注意力机制。

编码器/解码器架构是由两个神经网络组成的神经网络架构，一个编码器从我们的数据中获取输入向量并创建固定长度的embeddings，另一个解码器也是一个神经网络，它将编码的embeddings作为输入并生成静态的输出集，例如翻译文本或文本摘要。在这两种类型的层之间是注意力机制，一种通过持续执行加权矩阵乘法来保持整个输入状态的方式，以突出显示词汇表中特定术语之间的相关性。我们可以将注意力视为一个非常大、复杂的哈希表，它跟踪文本中的单词以及它们在输入和输出中如何映射到不同的表示。

²⁶如果你读过Schmidhuber的著作，你就会理解到深度学习中的一切都最初是由Schmidhuber开发的

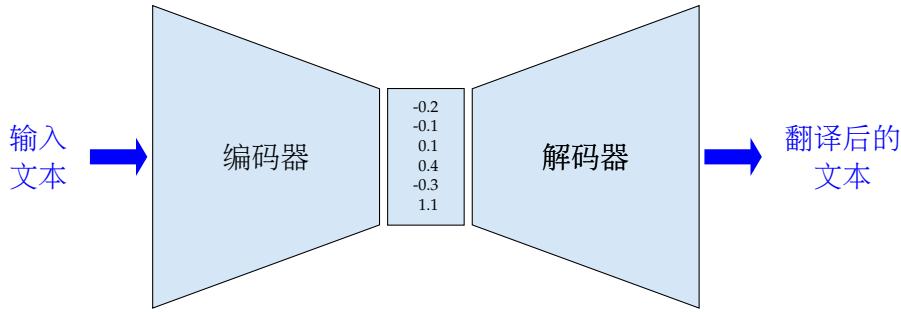


Figure 41: 编码器/解码器架构

```

1  class EncoderDecoder(nn.Module):
2      """
3          定义编码器/解码器步骤
4      """
5
6      def __init__(self, encoder, decoder, src_embed, tgt_embed, generator):
7          super(EncoderDecoder, self).__init__()
8          self.encoder = encoder
9          self.decoder = decoder
10         self.src_embed = src_embed
11         self.tgt_embed = tgt_embed
12         self.generator = generator
13
14     def forward(self, src, tgt, src_mask, tgt_mask):
15         """接收并处理蒙版的源和目标序列。"""
16         return self.decode(self.encode(src, src_mask), src_mask,
17                            tgt, tgt_mask)
18
19     def encode(self, src, src_mask):
20         return self.encoder(self.src_embed(src), src_mask)
21
22     def decode(self, memory, src_mask, tgt, tgt_mask):
23         return self.decoder(self.tgt_embed(tgt), memory, src_mask, tgt_mask)
24
25     class Generator(nn.Module):
26         """定义标准线性+softmax生成步骤。"""
27         def __init__(self, d_model, vocab):
28             super(Generator, self).__init__()
29             self.proj = nn.Linear(d_model, vocab)
30
31         def forward(self, x):
32             return F.log_softmax(self.proj(x), dim=-1)

```

Figure 42: 典型的编码器/解码器架构 来自*Annotated Transformer*

《Attention is All You Need》[57]，于2017年发布，将这两个概念合并到了单一的架构中。这篇论文立即取得了巨大的成功，如今Transformer是用于自然语言任务的事实标准模型之一。

基于原始模型的成功，发布了许多变体的Transformer架构，随后在2018年推出了GPT和BERT，2019年推出了Distilbert，这是BERT的一个更小、更紧凑的版本，2020年推出了GPT-3²⁷。

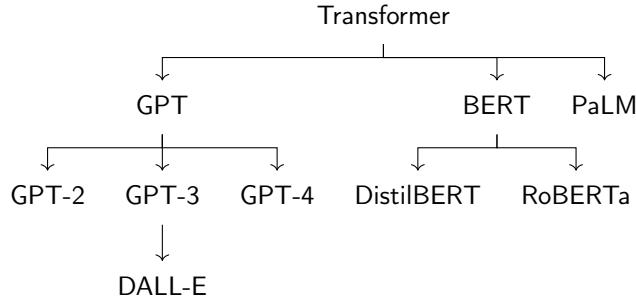


Figure 43: Transformer 模型的时间线

Transformer 架构本身并不是新的，但它们包含了我们到目前为止讨论的所有概念：向量、编码和哈希映射。Transformer 模型的目标是接收一个多模态内容，并通过创建输入语料库中单词组的多个视图（多个上下文窗口）来学习潜在的关系。自注意力机制，在 Transformer 论文中实现为缩放点积注意力，通过六个编码器层和六个解码器层多次创建数据的不同上下文窗口。输出是特定的机器学习任务的结果——一个翻译的句子，一个摘要的段落——倒数第二层是模型的embeddings，我们可以用于下游工作。

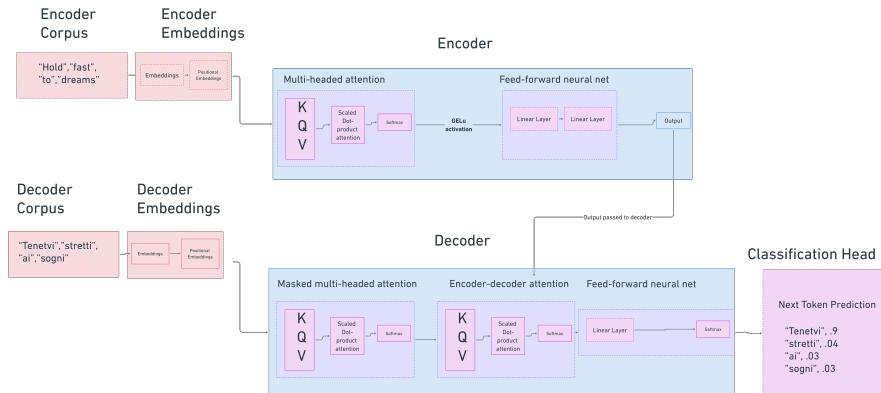


Figure 44: Transformer 层的视图，受多个来源的启发，包括此图 [链接](#)

在论文中描述的 Transformer 模型以文本语料库作为输入²⁸。我们首先通过将每个单词或子单词标记化并映射到索引来将文本转换为标记embeddings。这与 Word2Vec 中的过程相同：我们只需将每个单词分配给矩阵中的一个元素。然而，这些单独的embeddings不会帮助我们理解上下文，因此我们还使用正弦或余弦函数学习位置embeddings，这些embeddings考虑了词汇表中所有其他单词的位置，并映射和压缩为矩阵。该过程的最终输出是位置向量或单词编码。

接下来，这些位置向量被并行传递到模型中。在 Transformer 论文中，模型由六个执行编码和六个执行解码的层组成。我们从编码器层开始，它包

²⁷要查看完整的视觉Transformer时间线，请查看[此链接](#)

²⁸理论上，您可以使用任何形式的模态进行转换，而无需修改输入，只需用给定的模态标记数据 [62]，但早期的工作，如机器翻译，主要集中在文本上，所以我们将关注文本

括两个子层：自注意力层和一个前馈神经网络。自注意力层是关键部分，它通过缩放点积注意力执行学习每个术语与其他术语之间的关系的过程。我们可以从几个角度来看待自注意力：作为可微查找表，或作为一个大型查找字典，其中包含了术语和它们的位置，每个术语的权重是由前面的层获得的。

缩放点积注意力是三个矩阵的乘积：关键、查询和值。这些最初都是相同的值，是之前层的输出 — 在模型的第一次通过中，它们最初都是相同的，随机初始化，并在每个步骤中通过梯度下降进行调整。对于每个embeddings，我们根据这些学习到的注意力权重生成加权平均值。我们计算查询和

关键之间的点积，最后通过 softmax 对权重进行归一化。多头注意力意味着我们在并行计算缩放点积注意力的过程多次，并将结果连接成一个向量。

$$\text{Attention}(Q, K, V) = \text{softmax} \left(\frac{QK^T}{\sqrt{d_k}} \right) V \quad (11)$$

缩放点积注意力（以及编码器中的所有层）之所以如此出色，是因为可以在我们代码库中的所有标记上并行进行工作：我们不需要像在 Word2Vec 中那样等待一个单词完成处理，以便处理下一个单词，因此无论词汇量有多大，输入步骤的数量都保持不变。

解码器部分与编码器略有不同。它以不同的输入数据集开始：在 Transformer 论文中，它是我们想要将文本翻译成的目标语言数据集。例如，如果我们将 Flit 从英语翻译成意大利语，我们将期望对意大利语语料库进行训练。否则，我们执行所有相同的操作：我们创建索引embeddings，然后将其转换为位置embeddings。然后，我们将目标文本的位置embeddings馈送到具有三个部分的层：掩码多头注意力、多头注意力和前馈神经网络。掩码多头注意力组件与自注意力类似，只是多了一个额外的部分：在此步骤中引入的掩码矩阵 $acts$ 作为过滤器，以防止注意力头查看未来的标记，因为解码器的输入词汇表是我们的 "答案"，即翻译文本应该是什么。来自掩码多头自注意力层的输出传递到编码器-解码器注意力部分，它接受初始六个编码器层的最终输入作为键和值，并将前一个解码器层的输入作为查询，然后在此上执行缩放点积。然后，每个输出都被馈送到前馈层，以获得最终的embeddings。

一旦我们有了每个标记的隐藏状态，我们就可以附加任务头。在我们的情况下，这是预测单词应该是什么。在过程的每一步中，解码器都会查看前面的步骤，并基于这些步骤生成，因此我们形成一个完整的句子。然后，我们得到了预测的单词，就像在 Word2Vec 中一样。

Transformers 之所以革命性，是因为它们解决了几个人们一直在解决的问题：

- 并行化 - 模型中的每个步骤都是可并行化的，这意味着我们不需要等待了解一个单词的位置embeddings，以便处理另一个单词，因为每个embeddings查找矩阵都将注意力集中在特定的单词上，具有该单词的所有其他单词的查找表 - 每个单词的每个矩阵都携带整个输入文本的上下文窗口。
- 消失的梯度 - 以前的模型如 RNN 可能会遭受消失或爆炸的梯度问题，这意味着模型在完全训练之前就达到了局部最小值，使得捕捉长期依赖性变得具有挑战性。Transformer 通过允许序列中任意两个位置之间的直接连接来缓解这个问题，在前向和反向传播期间更有效地流动信息。
- 自注意力 - 注意力机制允许我们学习整个文本的上下文，这个文本比 2 或 3 个单词的滑动上下文窗口更长，从而使我们能够在不同的上下文中学习不同的单词，并更准确地预测答案。

4.3 BERT

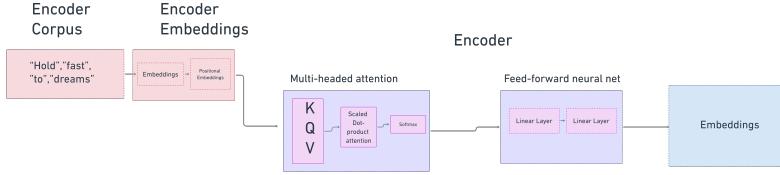


Figure 45: 仅编码器架构

在“注意力机制就是一切”的爆炸性成功之后，各种变体的Transformer架构涌现了出来，深度学习中对这种架构的研究和实现也迅速蓬勃发展。被视为显著进步的下一个Transformer架构是**BERT**。BERT代表双向编码器，于2018年发布[12]，是由Google发布的一篇论文的基础，旨在解决常见的自然语言任务，如情感分析、问答和文本摘要。BERT是一个Transformer模型，也基于注意力机制，但其架构仅包括编码器部分。它最显著的用途是在Google搜索中，它是支持提供相关搜索结果的算法。在2019年他们发布的关于将BERT用于搜索排名的博文中，Google明确讨论了为查询添加上下文作为替代基于关键词的方法的原因²⁹。

BERT作为一个掩码语言模型工作。掩码就是我们在实现Word2Vec时所做的事情，即删除单词并构建我们的上下文窗口。当我们使用Word2Vec创建表示时，我们只考虑向前移动的滑动窗口。Bert中的B代表双向，这意味着它通过缩放点积注意力双向关注单词。BERT有12个Transformer层。它首先使用**WordPiece**，一种将单词分割为子词、标记的算法。对BERT进行训练的目标是根据上下文预测一个标记。

BERT的输出是单词及其上下文的潜在表示——一组embeddings。BERT本质上是一个巨大的并行化的Word2Vec，可以记住更长的上下文窗口。由于BERT非常灵活，可以用于许多任务，从翻译、摘要到自动完成。因为它没有解码器组件，所以无法生成文本，为GPT模型接替BERT留下了道路。

4.4 GPT

在开发BERT的同时，OpenAI开发了另一种Transformer架构，即GPT系列。GPT与BERT的不同之处在于，它既对embeddings进行编码，又对解码文本进行解码，因此可以用于概率推理。

原始的、第一版的GPT模型是一个12层、12头的Transformer，只有一个解码器部分，基于来自Book Corpus的数据进行训练。随后的版本在此基础上进行了改进，以尝试提高上下文理解能力。最大的突破是在GPT-4中，它使用人类反馈的强化学习进行训练，这使得它能够进行更接近人类写作的文本推理。

我们现在已经达到了embeddings可能实现的前沿。随着生成式方法的兴起以及基于人类反馈的强化学习方法，例如OpenAI的ChatGPT以及初创的开源项目Llama、Alpaca等模型，任何在本文中写下的东西都将在出版时已经过时了³⁰。

²⁹ BERT搜索公告

³⁰ 已经有一些关于利用LLM进行推荐的可能用途的研究，包括会话式推荐系统，但目前仍处于早期阶段。有关更多信息，请参阅[此文](#)

5 生产中的embeddings

随着Transformer模型的出现，更重要的是BERT，为了在各种机器学习任务中使用大型、多模态对象的表示变得更加容易，这些表示变得更加准确，而且如果公司有GPU可用，计算现在可以并行加速进行。现在我们了解了embeddings是什么，那么我们应该如何使用它们呢？毕竟，我们不是为了进行数学练习而这样做。如果有一件事可以从整个文本中得出，那就是：

所有工业机器学习（ML）项目的最终目标是开发ML产品，并迅速将其投入生产。[\[33\]](#)

部署的模型总是比仅仅是原型的模型更好、更准确。我们在这里已经经历了端到端训练embeddings的过程，但是有几种处理embeddings的模态。我们可以：

- 训练自己的**embeddings**模型 - 我们可以从头开始训练BERT或BERT的某个变种。BERT使用了大量的训练数据，所以这对我们来说并不是很有利，除非我们想更好地理解内部原理并且拥有大量的GPU。
- 使用预训练的**embeddings**并微调 - 有许多BERT模型的变种，它们都已经被用来生成**embeddings**，作为许多推荐系统和信息检索系统的下游输入。变种的BERT都被用来生成**embeddings**，用作许多推荐和信息检索系统的下游输入。变换器架构赋予我们的最大礼物之一就是能够进行迁移学习。

在此之前，当我们在预变换器架构中学习**embeddings**时，我们对手头的任何数据集的表示是固定的一—我们无法改变TF-IDF中单词的权重，而不重新生成整个数据集。

现在，我们可以将BERT的层的输出视为我们自己定制模型的下一个神经网络层的输入。除了迁移学习之外，还有许多更紧凑的BERT模型，如Distilbert和RoBERTA等等，这些模型都可以在HuggingFace模型库等地方找到³¹。

有了这些知识，我们可以想象出几种**embeddings**的用例，考虑到它们作为数据结构的灵活性。

- 将它们馈送到另一个模型中 - 例如，我们现在可以使用从我们的数据中学习到的用户和项目**embeddings**执行协同过滤，而不是编写用户和项目本身。
- 直接使用它们 - 我们可以直接使用项目**embeddings**进行内容过滤—找到与其他项目最接近的项目，这是推荐任务与搜索共享的任务。有许多算法用于通过将项目投影到我们的**embeddings**空间并使用faiss和HNSW等算法执行相似性搜索来执行向量相似性查找。

5.1 实践中的embeddings

今天，许多公司都在这些领域使用**embeddings**，涵盖了信息检索的各个方面。使用深度学习模型生成的**embeddings**正在用于谷歌Play的应用程序。

³¹关于开源机器学习深度学习的发展，可以参考“[A Call to Build Models Like We Build Open-Source Software](#)”

用商店推荐的宽深模型[64]，在Overstock进行产品互补内容推荐的双重embeddings[34]，通过实时排名个性化搜索结果的Airbnb[22]，Netflix的内容理解[15]，Shutterstock的视觉风格理解[21]等等。

5.1.1 Pinterest

一个值得注意的例子是Pinterest。Pinterest作为一个应用程序拥有各种各样的内容，需要对其进行个性化和分类，以便在多个界面上向用户推荐，尤其是主页和购物选项卡。生成的内容规模—每月3.5亿用户和20亿个项目—Pins—或由文本描述的图片的卡片—需要一个强大的过滤和排名策略。

为了表示用户的兴趣并展示有趣的内容，Pinterest开发了PinnerSage[44]，通过多个256维度embeddings来表示用户的兴趣，并根据相似性进行聚类，并由中介—作为给定兴趣聚类中心的代表—来表示。

该系统的基础是一组通过名为PinSage的算法[63]开发的embeddings。PinSage使用图卷积神经网络生成embeddings，这是一个神经网络，考虑了网络中节点之间的关系图结构。该算法查看一个Pin的最近邻居，并根据相关的邻域访问从附近的Pins中对其进行采样。输入是Pin的embeddings：图像embeddings和文本embeddings，并找到最近的邻居。

然后，Pinsageembeddings传递给PinnerSage，后者获取用户在过去90天内采取行动的Pins，并对它们进行聚类。它计算中介并根据重要性选择前3个中介，并且在给定用户查询是中介的情况下，使用HNSW执行近似最近邻搜索来在embeddings空间中找到与查询最接近的Pins。

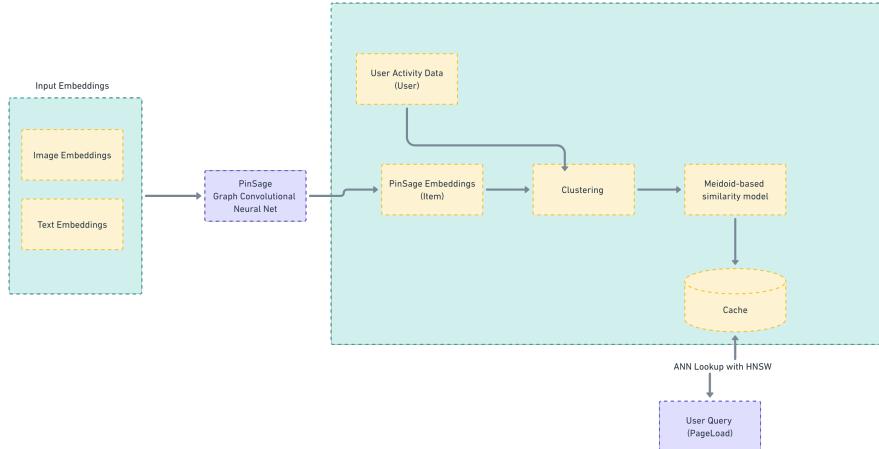


Figure 46: 基于Pinnersage和Pinsageembeddings的相似性检索

5.1.2 YouTube 和 Google Play Store

YouTube

YouTube 是最早公开分享他们在生产推荐系统中使用embeddings的工作的大公司之一，他们在 "Deep Neural Networks for YouTube Recommendations" 中介绍了这一工作。

YouTube 拥有超过 8 亿个内容（视频）和约 26 亿活跃用户，他们希望向这些用户推荐这些视频。该应用程序需要向用户推荐现有内容，同时也需要

推荐新内容，而这些新内容经常会被上传。他们需要能够在推断时间 — 用户加载新页面时 — 以低延迟提供这些推荐。

在这篇论文中[10]，YouTube 分享了他们为视频创建的基于两个深度学习模型的两阶段推荐系统。机器学习任务是在YouTube推荐中预测在给定时间向用户显示的正确下一个视频，以便用户点击。最终输出被构造为一个分类问题：给定用户的输入特征和视频的输入特征，我们能否预测一个类别，该类别包括用户观看特定视频的预测观看时间和特定概率。

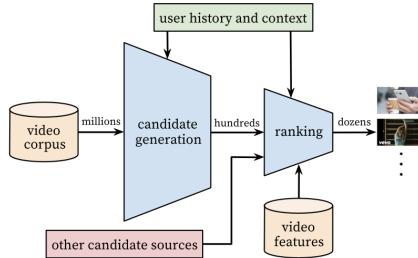


Figure 47: YouTube 的端到端视频推荐系统，包括候选生成器和排序器 [10]

我们设定了这个任务，给定一个用户 U 和上下文 C ³²

鉴于输入语料库的大小，我们需要将问题形式化为两阶段推荐器：第一阶段是候选生成器，将候选视频集减少到数百个项，并且第二个模型，大小和形状相似，称为排序器，将这数百个视频按用户点击和观看的概率进行排序。

候选生成器是一个具有多个层的**softmax**深度学习模型，所有这些层都使用**ReLU**激活函数 – 修正线性单元激活函数，如果输入为正，则直接输出输入；否则，输出为零。使用**embeddings**和表格学习特征，所有这些特征都被组合起来。

为了构建模型，我们使用两组**embeddings**作为输入数据：一个是用户加上上下文作为特征的**embeddings**，另一个是视频项。该模型具有数百个特征，包括表格和基于**embeddings**的特征。对于基于**embeddings**的特征，我们包括以下元素：

- 用户的观看历史记录 - 由稀疏视频ID元素的向量表示，并映射到稠密向量表示
- 用户的搜索历史记录 - 将搜索词映射到从搜索词中点击的视频，也是稀疏向量，映射到与用户观看历史相同的空间
- 用户的地理位置、年龄和性别 - 作为表格特征映射
- 视频以前展示给用户的次数，根据时间归一化

所有这些都被组合成一个单一的项**embeddings**，对于用户来说，一个融合了所有用户**embeddings**特征的单一**embeddings**，并被馈送到模型的**softmax**层，该层比较了**softmax**层的输出，即用户点击项目的概率，以及一组实际项目，即用户已经与之交互的项目。项目的对数概率是两个n维向量的点积，即查询和项目**embeddings**之间的点积。

我们认为这是隐式反馈的一个例子 - 用户没有明确给出的反馈，例如评分，但我们可以从我们的日志数据中捕获到。每个类别的响应，约有一百万个，都会输出一个概率。

³²在推荐系统中，我们通常考虑 **四个相关项目** 来构建我们的推荐问题 - 用户、物品、上下文和查询。上下文通常是环境，例如推断时间的时间或用户的地理位置

DNN是我们之前讨论过的矩阵分解模型的一种泛化。

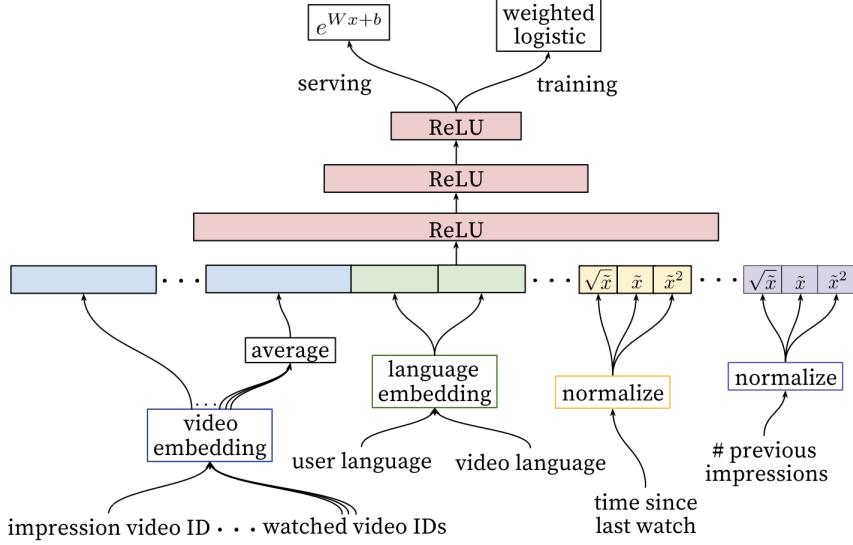


Figure 48: YouTube 使用输入embeddings的视频推荐的多步骤神经网络模型 [10]

Google Play 应用商店

在 Google Play 应用商店中，也进行了类似的工作，尽管采用了不同的架构，这是在 "Wide and Deep Learning for Recommender Systems" [6] 中完成的。这个工作跨越了搜索和推荐领域，因为它返回了正确的排名和个性化应用推荐，作为搜索查询的结果。输入是当用户访问应用商店时收集的点击流数据。

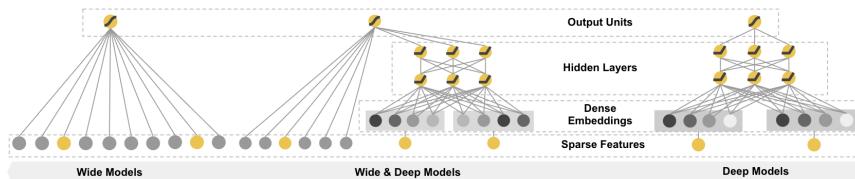


Figure 49: Wide and deep [6]

这里将推荐问题制定为两个联合训练的模型。在迭代之间，权重在两个模型之间共享和交叉传播。

当我们尝试构建推荐项目的模型时，存在两个问题：记忆 - 模型需要通过学习历史数据中项目的共现情况来学习模式，以及泛化 - 模型需要能够给出用户以前没有看到过的新推荐，但仍然与用户相关联，提高推荐的多样性。通常情况下，单个模型无法涵盖这两个权衡。

Wide and deep 由两个相互补充的模型组成：

- 一个宽模型，它使用传统的表格特征来改进模型的记忆。这是一个通用的线性模型，训练在稀疏的、独热编码的特征上，例如 `user_installed_app=netflix`，跨数千个应用。在这里，通过创建二进制特征来工作的记忆通过特征的组合，比如

`AND(user_installed_app=netflix, impression_app_pandora,` 让我们看到与目标的不同共现组合，即安装应用 Y 的可能性。然而，如果它得到训练数据之外的新值，这个模型不能泛化。

- 一个深模型，支持对模型以前没有见过的项目的泛化，使用由分类特征组成的前馈神经网络，这些特征被转换为embeddings，例如用户语言、设备类别以及一个给定应用是否有印象。这些embeddings的每一个都在 0-100 的维度范围内。它们联合地被组合成一个 1200 维的稠密向量的embeddings空间。并随机初始化。embeddings值被训练以最小化最终函数的损失，该函数是两个深和宽模型共同的逻辑损失函数。

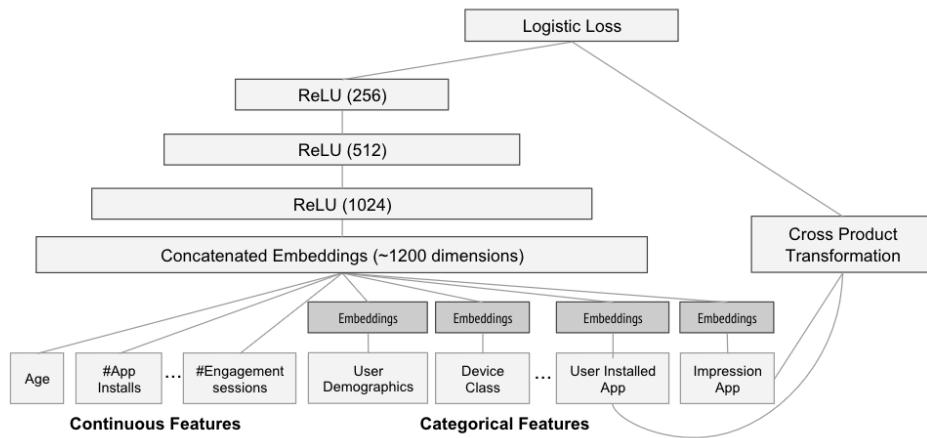


Figure 50: *Wide and deep* 模型的深层部分 [6]

该模型在 5000 亿个示例上进行训练，并且使用 AUC 进行离线评估，并使用应用获取率进行在线评估，即人们下载应用的速率。根据论文，使用这种方法，相对于对照组，提高了应用商店主页的应用获取率 3.9%。

5.1.3 Twitter

在Twitter，预先计算的embeddings是许多应用表面领域推荐的一个关键部分，包括用户入门主题兴趣预测、推荐推文、主页时间线构建、推荐关注的用户以及推荐广告。

Twitter有许多基于embeddings的模型，但我们在*这里*将介绍两个项目：Twice [39]，用于推文的内容embeddings，旨在为在主页时间线、通知和主题中呈现推文提供既包括文本又包括视觉数据的丰富表示。Twitter还开发了TwHIN [16]，Twitter异构信息网络，一组基于图的embeddings，用于个性化广告排名、账户关注推荐、有害内容检测和搜索排名等任务，基于节点（如用户和广告商）和代表实体交互的边。

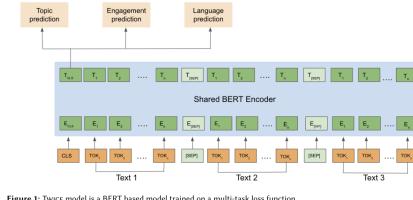


Figure 1: Twice model is a BERT based model trained on a multi-task loss function.

Figure 51: Twitter的Twiceembeddings，一个经过训练的BERT模型 [39]

Twice是一个从头开始在包括用户自身在内的样本了90天的用户参与的2亿推文的输入语料库上训练的BERT模型。该模型的目标是优化多个任务：主题预测（即与推文关联的主题，可能有多个）、参与预测（用户与推文互动的可能性）以及语言预测，以将相同语言的推文聚类到更接近的位置。

与仅关注推文内容不同，TwHIN认为Twitter环境中的所有实体（推文、用户、广告实体）都属于联合embeddings空间图中的一部分。

通过使用用户-推文互动、广告和关注数据，创建多模型embeddings，进行联合embeddings。TWIn用于候选生成。候选生成器使用HNSW或Faiss检索候选项，找到要关注的用户或要与之互动的推文。然后使用TwHinembeddings查询候选项，并增加候选池中的多样性。

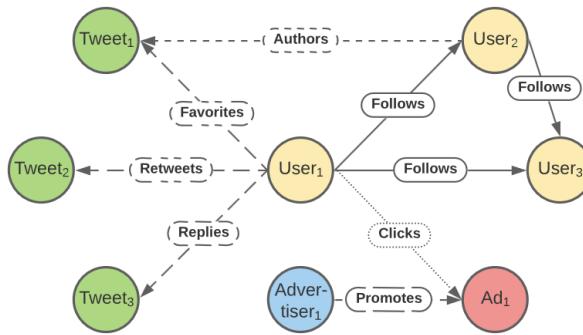


Figure 2: An example heterogeneous information network (HIN) where $|V| = 8$ and $|E| = 9$. There are four entity types (\mathcal{T}): ‘User’, ‘Tweet’, ‘Advertiser’, and ‘Ad’. There are seven types of relationship (\mathcal{R}): ‘Follows’, ‘Authors’, ‘Favorites’, ‘Replies’, ‘Retweets’, ‘Promotes’, and ‘Clicks’. See Section 3 for more details.

Figure 52: Twitter应用异构信息网络模型 [16]

Flutter中的embeddings

一旦我们综合了足够多的这些架构，我们就会看到一些模式开始出现，我们可以考虑将其调整为在Flutter中开发我们相关推荐系统所需的模式。

首先，我们需要大量的输入数据才能进行准确的预测，并且该数据应该包含有关明确或更可能是隐式数据的信息，例如用户的点击和购买，以便我们可以构建用户偏好的模型。我们需要大量数据的原因有两个。首先，神经网络需要大量的训练数据才能正确推断出关系，而传统模型相比要求更多。其次，大量数据需要大量的管道。

如果我们没有大量的数据，那么一个更简单的模型就足够好，因此我们需要确保我们实际上处于embeddings和神经网络有助于解决我们业务问题的规模。很可能情况是我们可以从简单的开始。事实上，最近一位最初开发因子分解机的研究人员在一篇论文中提出，作为推荐中重要方法的简单点积优于神经网络[46]。其次，为了获得良好的embeddings，我们需要花费大量时间清理和处理数据并创建特征，就像我们在YouTube论文中所做的那样，因此结果必须值得花费的时间。

其次，我们需要能够理解用户与他们交互的项目之间的潜在关系。在传统的推荐系统中，我们可以使用TF-IDF找到作为特定flit的一部分的加权词特征，并比较不同的文档，只要我们的语料库不会太大。在更先进的推荐系统中，我们可以通过查看简单的关联规则或将推荐作为基于交互的协同过滤问题来执行相同任务，类似于Word2Vec来生成用户和项目的潜在特征，即embeddings。事实上，这正是Levy和Goldberg在"Neural Word Embedding as Implicit Matrix Factorization"一文中所提出的[38]。他们查看了Word2Vec的skipgram实现，并发现在隐式情况下，它因子化了一个单词-上下文矩阵。

我们也可以将表格特征作为输入用于我们的协同过滤问题，但使用神经网络[25]而不是简单的点积来收敛于正确的关系和模型的下游排序。

有了关于embeddings和推荐系统工作原理的新知识，我们现在可以将embeddings融入到我们在Flutter中提供的flits的推荐中。如果我们想要推荐相关内容，我们可能会以不同的方式进行，具体取决于我们的业务要求。在我们的语料库中，我们有数亿条消息需要过滤到数百条以显示给用户。因此，我们可以从Word2Vec或类似的基线开始，然后转向任何BERT或其他神经网络方法来开发模型输入特征、向量相似性搜索和排序，通过embeddings的强大功能。

embeddings是无穷灵活和无穷有用的，可以增强和改进我们的多模式机器学习工作流的性能。然而，正如我们刚刚看到的，如果我们决定使用它们，有一些事情需要记住。

5.2 embeddings作为工程问题

总的来说，机器学习工作流给我们的工程系统增加了巨大的复杂性和开销，原因有很多 [50]。首先，它们混合了数据，然后需要在下游监视数据漂移。其次，它们的输出是非确定性的，这意味着它们需要被极其谨慎地跟踪为工件，因为我们通常不对数据进行版本控制。第三，它们导致了处理管道的混乱。

作为粘合代码的特殊情况，处理管道混乱通常出现在数据准备中。这些管道可能是有机地演变而来，当新的信号被识别出并且新的信息源被添加时。如果不小心处理，为了以便机器学习友好的格式准备数据的系统可能会变成一团由爬取、连接和采样步骤组成的混乱，通常会有中间文件输出。

从几个系统图中可以看出，包括PinnerSage和Wide and Deep模型，**使用embeddings的生产推荐系统**有许多移动组件。



Figure 53: *PinnerSage*模型架构 [44]

Figure 54: *Wide and Deep*模型架构 [6]

您可能还记得我们在此图中讨论的推荐系统的简单阶段。



Figure 55: 上下文中处理`embeddings`的通用系统

考虑到我们对于成功的推荐系统的实际生产级要求，我们的实际生产系统通常看起来更像是这样的：

- 生成`embeddings`
- 存储`embeddings`
- `embeddings`特征工程和迭代
- 工件检索
- 更新`embeddings`
- 对`embeddings`和数据漂移进行版本控制
- 推断和延迟
- 在线（A/B测试）和离线（度量扫描）模型评估

考虑到我们的关注范围，任何给定生产系统的图表都会更像是这样的：

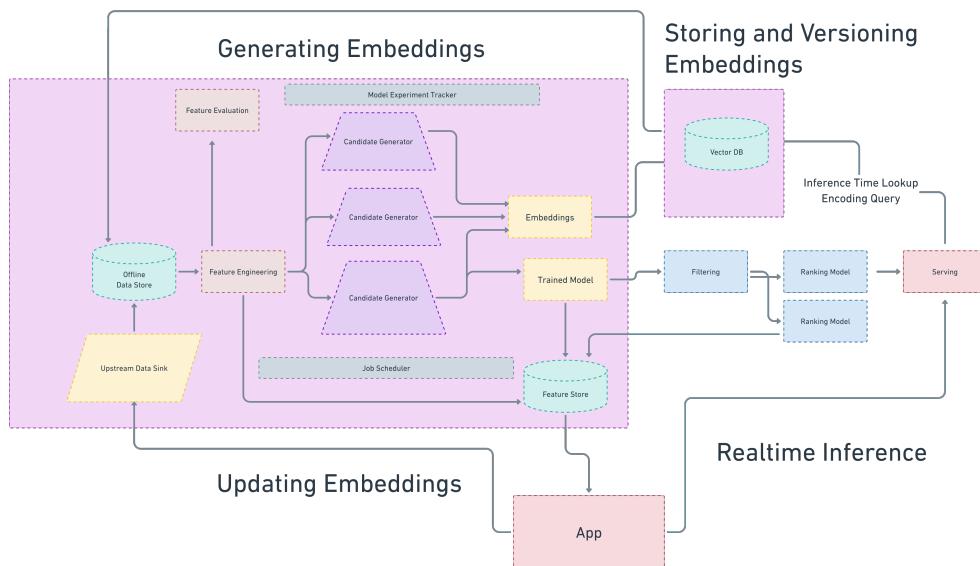


Figure 56: Recommender systems as a machine learning problem

5.2.1 embeddings生成

我们已经知道，embeddings通常作为训练神经网络模型的副产品生成，最常见的就是在添加用于分类或回归的最终输出层之前的倒数第二层。我们有两种方式来构建它们。我们可以像YouTube、Pinterest和Twitter一样训练自己的模型。在LLM领域，也越来越有兴趣能够在内部训练大型语言模型。

然而，深度学习模型的一个巨大优势是我们也可以使用预训练模型。预训练模型是指已经在巨大的训练数据集上训练过的与我们考虑的任务相似的模型，可以用于下游任务。BERT就是一个已经预训练的模型示例，通过微调的过程可以用于任意数量的机器学习任务。在微调中，我们采用已经预训练在通用数据集上的模型。例如，BERT是在包含11k本书和8亿字的BookCorpus以及包含25亿字的英文维基百科上进行训练的。

关于训练数据的一点

训练数据是任何给定模型最重要的部分。预训练的大型语言模型的训练数据来自哪里？通常是从互联网上大量抓取。出于竞争优势的考虑，这些训练数据集的构建方式通常不会被公开，因此存在大量的逆向工程和猜测。例如，“What’s in my AI”深入研究了GPT系列模型背后的训练数据，发现GPT-3是在Books1和Books2上训练的。Books1很可能是bookcorpus，而Books2很可能是libgen。GPT还包括Common Crawl，一个大型的开源网站索引数据集，WebText2和维基百科。这些信息很重要，因为当我们选择一个模型时，我们至少需要高层次地了解它是如何被训练成通用模型的，以便我们能够解释在微调它时发生了什么变化。

在微调模型时，我们执行与从头训练相同的所有步骤。我们有训练数据，有一个模型，并最小化损失函数。但是，存在几个不同之处。当我们创建我们的新模型时，我们复制现有的预训练模型，唯一的例外是最终输出层，我们根据新任务从头初始化它。当我们训练模型时，我们随机初始化这些参数，并且只继续调整前一层的参数，使它们专注于这个任务，而不是从头开始训练。通过这种方式，如果我们有一个像BERT这样的模型，它是训练成可以在整个互联网上进行泛化的，但是我们的Flutter语料库非常敏感于热门话题，并且需要每天更新，我们可以重新聚焦模型，而不必训练一个新模型，仅仅使用10k个样本而不是最初的数亿个。

同样，也有可以微调的BERTembeddings。还有其他通用的语料库可用，例如GloVe、Word2Vec和FastText（也是用CBOW训练的）。我们需要决定是使用这些，从头开始训练一个模型，还是第三种选择，从API查询可用的embeddings，就像OpenAiembeddings的情况一样，尽管这样做可能会产生更高的成本，相对于训练或微调我们自己的模型。当我们开始一个项目时，所有这些都取决于我们特定的用例，并且在评估时非常重要。

5.2.2 存储和检索

一旦我们训练了我们的模型，我们就需要从训练对象中提取embeddings。通常情况下，当模型训练完成时，结果输出是一个包含模型所有参数的数据结构，包括模型的权重、偏差、层和学习率。embeddings作为模型对象的一部分存在于这个模型对象中作为一个层，并且最初存储在内存中。当我们将模型写入磁盘时

，我们将其作为一个模型对象传播，并将其序列化到内存中，并在重新训练或推理时加载。

最简单的embeddings存储形式可以是一个内存中的numpy数组³³。

但是，如果我们正在迭代构建一个带有embeddings的模型，我们希望能够对它们进行一些操作：

- 在推理时批量和逐个访问它们
- 对embeddings的质量进行离线分析
- embeddings特征工程
- 使用新模型更新embeddings
- 版本化embeddings
- 对新文档进行编码以生成新的embeddings

处理许多这些用例的最复杂和可定制化的软件是向量数据库，在向量数据库和内存存储之间的某处是现有存储的向量搜索插件，如Postgres和SQLite，以及Redis等缓存。

我们希望执行的最重要的操作之一是向量搜索，它允许我们找到与给定embeddings相似的embeddings，以便我们可以返回项目相似性。如果我们想要搜索embeddings，我们需要一种机制来优化我们的矩阵数据结构的搜索，并以与传统关系数据库优化行为类似的方式执行最近邻比较。关系数据库使用B树结构来通过在节点层次结构内按升序对项目进行排序来优化读取，该层次结构构建在数据库中的索引列之上。我们不能有效地对我们的向量进行行列查找，因此我们需要为它们创建不同的结构。例如，许多向量存储基于倒排索引。

通用形式的embeddings存储包含embeddings本身，一个索引将它们从潜在空间映射回单词、图片或文本，以及一种通过各种最近邻算法在不同类型的embeddings之间进行相似性比较的方法。我们之前谈到过余弦相似度作为比较潜在空间表示的重要方法。当元素数量达到数百万个集合时，这可能会变得计算昂贵，因为我们需要对每对进行成对比较。为了解决这个问题，开发了近似最近邻（ANN）算法，例如在推荐系统中，通过向量的元素创建邻域并找到向量的k个最近邻。最常用的算法包括HNSW（分层可导航小世界）和Faiss，它们都是独立的库，也作为许多现有向量存储的一部分实现。

全面搜索和最近邻搜索之间的权衡是后者不太精确，但速度更快。在评估中在精度和召回率之间切换时，我们需要意识到这种权衡，并考虑我们对于embeddings的准确性以及推理延迟的要求是什么。

这是Twitter正是这个用例构建的embeddings存储系统的一个示例，早在向量数据库出现之前。[\[54\]](#)。

³³根据Andrej Karpathy在2023年对于他如何存储小型电影推荐项目的向量embeddings的一条推文，“np.array人们在如今追求的东西上过于追求花哨了。[推特](#)

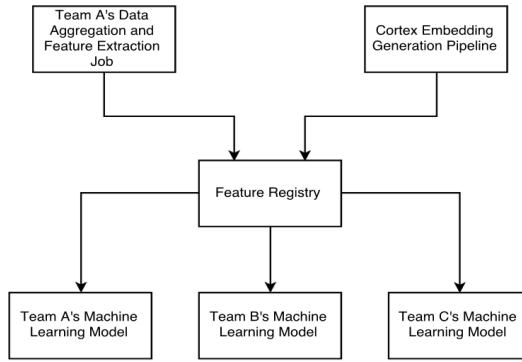


Figure 57: Twitter的embeddings管道 [54]

考虑到Twitter在之前部分中描述的多个来源中使用embeddings，Twitter通过创建一个集中式平台，对数据进行重新处理，并将embeddings生成到特征注册表中，使embeddings成为“一等公民”。

5.2.3 漂移检测、版本控制和可解释性

一旦我们训练了embeddings，我们可能会认为一切都完成了。但是，与任何机器学习管道一样，embeddings需要进行刷新，因为我们可能会遇到概念漂移。概念漂移发生在支撑我们模型的数据发生变化时。例如，假设我们的模型包含一个二进制特征，即是否拥有座机电话。在2023年，这在世界大部分地区不再是一个相关的特征，因为大多数人已经将手机作为主要电话，因此该模型会失去准确性。

对于用于分类的embeddings，这种现象更加普遍。例如，假设我们使用embeddings进行热门话题检测。模型必须像宽而深的模型一样泛化以检测新的类别，但如果我们的类别变化迅速，它可能无法做到这一点，因此我们需要频繁地重新训练embeddings。或者，例如，如果我们在图中对embeddings进行建模，就像Pinterest一样，并且图中节点之间的关系发生变化，我们可能需要更新它们[60]。我们还可能遇到垃圾邮件或损坏内容的涌入，这会改变embeddings中的关系，因此我们需要重新训练。

embeddings可能很难理解（以至于有些人甚至写了整篇论文来讨论它们），而且更难以解释。国王接近女王但远离骑士意味着什么？在投影的embeddings空间中，两个Flits彼此接近意味着什么？

我们可以通过两种方式来思考这个问题，即内在和外在评估。对于embeddings本身（外在评估），我们可以通过UMAP（均匀流形逼近和降维）或t-SNE（t分布随机邻域embeddings）等算法将它们可视化，这些算法允许我们以二维或三维的方式可视化高维数据，就像PCA一样。或者我们可以将embeddings适配到我们的下游任务中（例如，摘要或分类），并且分析方式与离线指标相同。有许多不同的方法[58]，但总结起来就是embeddings可能是客观难以评估的，我们需要将执行此评估所需的时间纳入建模时间。

一旦我们重新训练了初始的基准模型，我们现在有了一个次要问题：我们如何将第一组embeddings与第二组进行比较；也就是说，我们如何评估它们是否是数据的良好表示，考虑到embeddings通常是无监督的；也就是说—我

们怎么知道"国王"应该接近"女王"? 单独作为第一次尝试, `embeddings`可能很难解释, 因为在多维空间中很难理解向量的哪个维度对应于决定将项目放在一起的决策[55]。当与其他一组`embeddings`进行比较时, 我们可能会使用最终模型任务的离线指标, 如精确度和召回率, 或者我们可以通过比较概率分布的距离来测量`embeddings`在

潜在空间中的分布, 使用一种称为**Kullback–Leibler**散度的度量。

最后, 现在假设我们有两组`embeddings`, 您需要对它们进行版本控制并保留两组, 以防新模型效果不佳时可以回退到旧模型。这与ML操作中模型版本控制的问题密切相关, 但在这种情况下, 我们需要对模型和输出数据进行版本控制。

有许多不同的模型和数据版本控制方法, 涉及构建一个跟踪元数据和保存在辅助数据存储中的资产位置的系统。还要记住的是, 特别是对于大词汇量, `embeddings`层可能会体积急剧膨胀, 因此现在我们必须考虑存储成本。

5.2.4 推理和延迟

当涉及到`embeddings`时, 我们不仅在理论上操作, 还在实际的工程环境中操作。任何在生产环境中运行的机器学习系统中最关键的工程部分是推理时间—查询模型资产并将结果返回给最终用户需要多长时间。

对此, 我们关心延迟, 我们可以粗略地定义为任何等待的时间, 这是任何生产系统中关键的性能指标[23]。一般来说, 它是任何操作完成的时间—应用请求、数据库查询等。Web服务层次的延迟通常以毫秒为单位进行测量, 并且会尽一切努力将其尽可能接近零。对于搜索和加载内容流的使用情况, 体验需要是即时的, 否则用户体验会降低, 我们甚至可能会损失收入。在一项较早的研究中, 亚马逊发现, 每增加100毫秒的延迟会减少1% 的利润[17]。

鉴于这一点, 我们需要考虑如何减少模型及其所有层的足迹, 以便对用户的响应是即时的。我们通过在我们的机器学习系统中创建可观察性来实现这一点, 从运行系统的硬件开始, 到CPU和GPU的利用率, 我们的模型架构的性能, 以及该模型与其他组件的交互方式。例如, 当我们进行最近邻查询时, 我们进行该查询的方式以及我们使用的算法, 编写该算法的编程语言, 都会影响延迟问题。

举个例子, 在宽而深的论文中, 推荐排序模型每秒评分超过1000万个应用。该应用最初是单线程的, 所有候选项耗时31毫秒。通过实施多线程, 他们能够将客户端延迟降低到14毫秒[6]。

机器学习系统的操作是一门完全不同的研究艺术和手艺, 最好还是留给另一篇论文[33]。

5.2.5 在线和离线模型评估

我们几乎没有涉及模型中最关键的部分之一: 它在离线和在线测试中的表现。当我们谈论离线测试时, 我们是指分析模型的统计特性, 以了解模型是否是有效的模型—即我们的损失函数是否收敛? 模型是否过拟合或欠拟合? 精确度和召回率是多少? 我们是否经历了任何漂移? 如果是推荐排行模型, 我们是否使用类似**NDCG** (归一化折现累积增益) 的指标来了解我们的新模型是否比以前的迭代排列项目更好?

然后, 有在线评估, 也就是模型在实际生产环境中的实际成功情况。通常, 通过A/B测试进行评估, 在这种情况下, 一组用户使用旧模型或系统, 另一组用户使用新系统, 并查看诸如点击率、提供的项目和在站点的特定区域花费的时间等指标的统计显著性。

5.2.6 什么使embeddings项目成功

最后，一旦我们所有的算法和工程问题都解决了，还有最后一件事要考虑，那就是从商业角度来看我们的项目如何成功。我们应该承认，对于我们的机器学习问题，我们可能并不总是需要embeddings，或者最初我们可能根本不需要机器学习，如果我们的项目完全基于少数可以由人类确定和分析的启发式规则[66]。

如果我们得出结论说我们正在操作一个数据丰富的领域，在该领域中自动推断实体之间的语义关系是正确的，我们需要问自己是否愿意花大量精力生成干净的数据集，这是任何良好的机器学习模型的基础，即使在大语言模型的情况下也是如此。实际上，干净的领域数据如此重要，以至于许多这里讨论的公司最终都训练了自己的embeddings模型，最近，像彭博社[61]和Replit[51]这样的公司甚至正在训练自己的大型语言模型，以提高其特定业务领域的准确性。

关键是，要达到一个具有embeddings式的推荐系统在生产中的状态，我们需要一个团队，他们在需要完成的工作上有多层次的对齐。在较大的公司中，这个团队的规模将会更大，但是最重要的是，embeddings式的工作需要一个可以专门讨论用例的人，一个支持用例并将其优先级提高的人，以及一个可以完成工作的技术人员[41]。

如果所有这些组成部分都齐全，现在我们在生产中有了一个基于embeddings的推荐系统。

6 总结

我们已经走过了一个端到端的示例，介绍了embeddings是什么。我们从高层次概述了embeddings是如何融入机器学习应用的背景中的。然后，我们深入研究了早期编码方法，逐步建立了对Word2Vec中embeddings的直觉，然后转向了transformers和BERT。尽管降低维度作为一个概念在机器学习系统中一直很重要，以减少计算和存储复杂性，但在现代多模态数据表示的爆炸式增长中，压缩变得更加重要，这些数据来自应用日志文件、图像、视频和音频，以及Transformer、生成式和扩散模型的爆炸式增长，结合了廉价的存储和数据爆炸，已经修正为越来越多地使用embeddings的架构。

我们理解了为什么我们可能在我们的应用程序中包含机器学习模型的工程背景，它们是如何工作的，如何将它们合并到我们的应用程序中，以及embeddings—深度学习模型输入和输出数据的密集表示—可以最好地发挥作用的地方。embeddings是任何机器学习系统中的强大工具，但其维护和可解释性成本高昂。使用正确的方法、正确的指标和硬件和软件生成embeddings，是一个需要深思熟虑的项目。现在，我们希望对embeddings的基础知识有扎实的理解，可以在我们的下一个项目中利用它们—或解释为什么不利用它们。

祝你在embeddings中航行顺利，在潜在空间见！

References

1. Gabriel Altay. Categorical variables in decision trees, Mar 2020. URL <https://www.kaggle.com/code/gabrielaltay/categorical-variables-in-decision-trees>.
2. Ioannis Arapakis, Xiao Bai, and B Barla Cambazoglu. Impact of response latency on user behavior in web search. In *Proceedings of the 37th international ACM SIGIR conference on Research & development in information retrieval*, pages 103–112, 2014.
3. Remzi H Arpacı-Dusseau and Andrea C Arpacı-Dusseau. *Operating systems: Three easy pieces*. Arpacı-Dusseau Books, LLC, 2018.
4. Yoshua Bengio, Aaron Courville, and Pascal Vincent. Representation learning: A review and new perspectives. *IEEE transactions on pattern analysis and machine intelligence*, 35(8):1798–1828, 2013.
5. Pablo Castells and Dietmar Jannach. Recommender systems: A primer. *arXiv preprint arXiv:2302.02579*, 2023.
6. Heng-Tze Cheng, Levent Koc, Jeremiah Harmsen, Tal Shaked, Tushar Chandra, Hrishi Aradhye, Glen Anderson, Greg Corrado, Wei Chai, Mustafa Ispir, et al. Wide & deep learning for recommender systems. In *Proceedings of the 1st workshop on deep learning for recommender systems*, pages 7–10, 2016.
7. Francois Chollet. *Deep learning with Python*. Simon and Schuster, 2021.
8. Ronan Collobert and Jason Weston. A unified architecture for natural language processing: Deep neural networks with multitask learning. In *Proceedings of the 25th international conference on Machine learning*, pages 160–167, 2008.
9. Yingnan Cong, Yao-ban Chan, and Mark A Ragan. A novel alignment-free method for detection of lateral genetic transfer based on tf-idf. *Scientific reports*, 6(1):1–13, 2016.
10. Paul Covington, Jay Adams, and Emre Sargin. Deep neural networks for youtube recommendations. In *Proceedings of the 10th ACM conference on recommender systems*, pages 191–198, 2016.
11. Toni Cvitanic, Bumsoo Lee, Hyeyon Ik Song, Katherine Fu, and David Rosen. Lda v. lsa: A comparison of two computational text analysis tools for the functional categorization of patents. In *International Conference on Case-Based Reasoning*, 2016.
12. Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.
13. Giovanni Di Gennaro, Amedeo Buonanno, and Francesco AN Palmieri. Considerations about learning word2vec. *The Journal of Supercomputing*, pages 1–16, 2021.

14. Author Cory Doctorow. Pluralistic: Tiktok’s enshittification (21 jan 2023), Feb 2023. URL <https://pluralistic.net/2023/01/21/potemkin-ai/#hey-guys>.
15. Melody Dye, Chaitanya Ekandham, Avneesh Saluja, and Ashish Rastogi. Supporting content decision makers with machine learning, Dec 2020. URL <https://netflixtechblog.com/supporting-content-decision-makers-with-machine-learning-995b7b76006f>.
16. Ahmed El-Kishky, Thomas Markovich, Serim Park, Chetan Verma, Baekjin Kim, Ramy Eskander, Yury Malkov, Frank Portman, Sofía Samaniego, Ying Xiao, et al. Twhin: Embedding the twitter heterogeneous information network for personalized recommendation. In *Proceedings of the 28th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, pages 2842–2850, 2022.
17. Tobias Flach, Nandita Dukkipati, Andreas Terzis, Barath Raghavan, Neal Cardwell, Yuchung Cheng, Ankur Jain, Shuai Hao, Ethan Katz-Bassett, and Ramesh Govindan. Reducing web latency: the virtue of gentle aggression. In *Proceedings of the ACM SIGCOMM 2013 conference on SIGCOMM*, pages 159–170, 2013.
18. Martin Fowler. *Patterns of Enterprise Application Architecture: Pattern Enterprise Applica Arch.* Addison-Wesley, 2012.
19. Jan J Gerbrands. On the relationships between svd, klt and pca. *Pattern recognition*, 14(1-6):375–381, 1981.
20. Yoav Goldberg and Omer Levy. word2vec explained: deriving mikolov et al.’s negative-sampling word-embedding method. *arXiv preprint arXiv:1402.3722*, 2014.
21. Raul Gomez Bruballa, Lauren Burnham-King, and Alessandra Sala. Learning users’ preferred visual styles in an image marketplace. In *Proceedings of the 16th ACM Conference on Recommender Systems*, pages 466–468, 2022.
22. Mihajlo Grbovic and Haibin Cheng. Real-time personalization using embeddings for search ranking at airbnb. In *Proceedings of the 24th ACM SIGKDD international conference on knowledge discovery & data mining*, pages 311–320, 2018.
23. Brendan Gregg. *Systems performance: enterprise and the cloud*. Pearson Education, 2014.
24. Casper Hansen, Christian Hansen, Lucas Maystre, Rishabh Mehrotra, Brian Brost, Federico Tomasi, and Mounia Lalmas. Contextual and sequential user embeddings for large-scale music recommendation. In *Proceedings of the 14th ACM Conference on Recommender Systems*, pages 53–62, 2020.
25. Xiangnan He, Lizi Liao, Hanwang Zhang, Liqiang Nie, Xia Hu, and Tat-Seng Chua. Neural collaborative filtering. In *Proceedings of the 26th international conference on world wide web*, pages 173–182, 2017.

26. Michael E Houle, Hans-Peter Kriegel, Peer Kröger, Erich Schubert, and Arthur Zimek. Can shared-neighbor distances defeat the curse of dimensionality? In *Scientific and Statistical Database Management: 22nd International Conference, SSDBM 2010, Heidelberg, Germany, June 30–July 2, 2010. Proceedings* 22, pages 482–500. Springer, 2010.
27. Dietmar Jannach, Markus Zanker, Alexander Felfernig, and Gerhard Friedrich. *Recommender systems: an introduction*. Cambridge University Press, 2010.
28. Yushi Jing, David Liu, Dmitry Kislyuk, Andrew Zhai, Jiajing Xu, Jeff Donahue, and Sarah Tavel. Visual search at pinterest. In *Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 1889–1898, 2015.
29. Thorsten Joachims. Text categorization with support vector machines: Learning with many relevant features. In *Machine Learning: ECML-98: 10th European Conference on Machine Learning Chemnitz, Germany, April 21–23, 1998 Proceedings*, pages 137–142. Springer, 2005.
30. Andrej Karpathy. The unreasonable effectiveness of recurrent neural networks, May 2015. URL <https://karpathy.github.io/2015/05/21/rnn-effectiveness/>.
31. P.N. Klein. *Coding the Matrix: Linear Algebra Through Applications to Computer Science*. Newtonian Press, 2013. ISBN 9780615880990. URL <https://books.google.com/books?id=3AA4nwEACAAJ>.
32. Martin Kleppmann. *Designing data-intensive applications: The big ideas behind reliable, scalable, and maintainable systems*. " O'Reilly Media, Inc.", 2017.
33. Dominik Kreuzberger, Niklas Kühl, and Sebastian Hirschl. Machine learning operations (mlops): Overview, definition, and architecture. *arXiv preprint arXiv:2205.02302*, 2022.
34. Giorgi Kvernadze, Putu Ayu G Sudyanti, Nishan Subedi, and Mohammad Hajiaghayi. Two is better than one: Dual embeddings for complementary product recommendations. *arXiv preprint arXiv:2211.14982*, 2022.
35. Valliappa Lakshmanan, Sara Robinson, and Michael Munn. *Machine learning design patterns*. O'Reilly Media, 2020.
36. Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.
37. Yann LeCun, Yoshua Bengio, Geoffrey Hinton, et al. Deep learning. *nature*, 521 (7553), 436-444. *Google Scholar Google Scholar Cross Ref Cross Ref*, page 25, 2015.
38. Omer Levy and Yoav Goldberg. Neural word embedding as implicit matrix factorization. *Advances in neural information processing systems*, 27, 2014.

39. Xianjing Liu, Behzad Golshan, Kenny Leung, Aman Saini, Vivek Kulkarni, Ali Mollahosseini, and Jeff Mo. Twice-twitter content embeddings. In *CIKM 2022*, 2022.
40. Donella H Meadows. *Thinking in systems: A primer*. chelsea green publishing, 2008.
41. Doug Meil. Ai in the enterprise. *Communications of the ACM*, 66(6):6–7, 2023.
42. Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781*, 2013.
43. Usman Naseem, Imran Razzak, Shah Khalid Khan, and Mukesh Prasad. A comprehensive survey on word representation models: From classical to state-of-the-art word representation language models. *Transactions on Asian and Low-Resource Language Information Processing*, 20(5):1–35, 2021.
44. Aditya Pal, Chantat Eksombatchai, Yitong Zhou, Bo Zhao, Charles Rosenberg, and Jure Leskovec. Pinnersage: Multi-modal user embedding framework for recommendations at pinterest. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pages 2311–2320, 2020.
45. Delip Rao and Brian McMahan. *Natural language processing with PyTorch: build intelligent language applications using deep learning*. " O'Reilly Media, Inc.", 2019.
46. Steffen Rendle, Walid Krichene, Li Zhang, and John Anderson. Neural collaborative filtering vs. matrix factorization revisited. In *Proceedings of the 14th ACM Conference on Recommender Systems*, pages 240–248, 2020.
47. David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. Learning representations by back-propagating errors. *nature*, 323(6088):533–536, 1986.
48. Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, et al. Imagenet large scale visual recognition challenge. *International journal of computer vision*, 115:211–252, 2015.
49. Hinrich Schütze, Christopher D Manning, and Prabhakar Raghavan. *Introduction to information retrieval*, volume 39. Cambridge University Press Cambridge, 2008.
50. David Sculley, Gary Holt, Daniel Golovin, Eugene Davydov, Todd Phillips, Dietmar Ebner, Vinay Chaudhary, and Michael Young. Machine learning: The high interest credit card of technical debt.(2014), 2014.
51. Reza Shabani. How to train your own large language models, Apr 2023. URL <https://blog.replit.com/llm-training>.

52. Christopher J Shallue, Jaehoon Lee, Joseph Antognini, Jascha Sohl-Dickstein, Roy Frostig, and George E Dahl. Measuring the effects of data parallelism on neural network training. *arXiv preprint arXiv:1811.03600*, 2018.
53. Or Sharir, Barak Peleg, and Yoav Shoham. The cost of training nlp models: A concise overview. *arXiv preprint arXiv:2004.08900*, 2020.
54. Dan Shiebler and Abhishek Tayal. Making machine learning easy with embeddings. *SysML* <http://www.sysml.cc/doc/115.pdf>, 2010.
55. Adi Simhi and Shaul Markovitch. Interpreting embedding spaces by conceptualization. *arXiv preprint arXiv:2209.00445*, 2022.
56. Krysta M Svore and Christopher JC Burges. A machine learning approach for improved bm25 retrieval. In *Proceedings of the 18th ACM conference on Information and knowledge management*, pages 1811–1814, 2009.
57. Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.
58. Bin Wang, Angela Wang, Fenxiao Chen, Yuncheng Wang, and C-C Jay Kuo. Evaluating word embedding models: Methods and experimental results. *APSIPA transactions on signal and information processing*, 8:e19, 2019.
59. Yuxuan Wang, Yutai Hou, Wanxiang Che, and Ting Liu. From static to dynamic word representations: a survey. *International Journal of Machine Learning and Cybernetics*, 11:1611–1630, 2020.
60. Christopher Wewer, Florian Lemmerich, and Michael Cochez. Updating embeddings for dynamic knowledge graphs. *arXiv preprint arXiv:2109.10896*, 2021.
61. Shijie Wu, Ozan Irsoy, Steven Lu, Vadim Dabrowski, Mark Dredze, Sebastian Gehrmann, Prabhanjan Kambadur, David Rosenberg, and Gideon Mann. Bloomberggpt: A large language model for finance. *arXiv preprint arXiv:2303.17564*, 2023.
62. Peng Xu, Xiatian Zhu, and David A Clifton. Multimodal learning with transformers: A survey. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2023.
63. Rex Ying, Ruining He, Kaifeng Chen, Pong Eksombatchai, William L Hamilton, and Jure Leskovec. Graph convolutional neural networks for web-scale recommender systems. In *Proceedings of the 24th ACM SIGKDD international conference on knowledge discovery & data mining*, pages 974–983, 2018.
64. Shuai Zhang, Lina Yao, Aixin Sun, and Yi Tay. Deep learning based recommender system: A survey and new perspectives. *ACM computing surveys (CSUR)*, 52(1):1–38, 2019.

65. Yong Zheng. Multi-stakeholder recommendation: Applications and challenges. *arXiv preprint arXiv:1707.08913*, 2017.
66. Martin Zinkevich. Rules of machine learning: Best practices for ml engineering. URL: <https://developers.google.com/machine-learning/guides/rules-of-ml>, 2017.