第五章：嵌入式的技巧 - Embeddings

很多行业希望拥有 LLMs 的能力，希望 LLMs 能解决自己的企业内部问题。这就包括员工相关的内容如入职须知，请假和报销流程，还有福利查询等，企业业务流相关的内容包括相关文档，法规，执行流程等，也有一些面向客户的查询。虽然 LLMs 有强大的知识能力，但是基于行业的数据和知识是没办法获取的。那如何注入这些基于行业的知识内容呢？这也是让 LLMs 迈入企业化重要的一步。本章我们就会和大家讲讲如何注入行业的数据和知识，让 LLMs 变得更专业。也就是我们创建 RAG 应用的基础。

从自然语言中的向量谈起

在自然语言领域，我们知道最细的粒度是词，词组成句，句构成段落，篇章和最后的文档。计算机是不认识词的，所以我们需要对词转换为数学上的表示。这个表示就是向量，也就是 Vector。向量是一个数学上的概念，它是一个有方向的量，有大小和方向。 有了向量，我们可以有效地对文本进行向量化，这也是计算机自然语言领域的基础。在自然语言处理领域，我们有很多向量化的方法，比如 One-hot，TF-IDF，Word2Vec，Glove，ELMO，GPT，BERT 等。这些向量化的方法都有各自的优缺点，但是都是基于词的向量化，也就是词向量。词向量是自然语言处理的基础，也是 OpenAI 所有模型的基础。我们分别看看词向量里面的几种常见方法。

One-hot 编码

One-hot 编码，是用 0 和 1 的编码方式来表示词。比如我们有 4 个词，分别是：我，爱，北京，天安门。那么我们可以用 4 个向量来表示这 4 个词，分别是：

```
我 = [1, 0, 0, 0]
爱 = [0, 1, 0, 0]
北京 = [0, 0, 1, 0]
天安门 = [0, 0, 0, 1]
```

在传统的自然语言应用场景中，我们把每个词看成用 One-Hot 向量表示，作为唯一的离散符号。我们的词库中有单词的数量就是向量的维度，如上述的例子总共包含了四个词，所以我们可以用一个四维的向量来表示。在这个向量中，每个词都是唯一的，也就是说每个词都是独立的，没有任何关系。这样的向量我们称为 One-Hot 向量。One-Hot 向量的优点是简单，容易理解，而且每个词都是唯一的，没有任何关系。但是 One-Hot 向量的缺点也很明显，就是向量的维度会随着词的增加而增加。比如我们有 1000 个词，那么我们的向量就是 1000 维的。这样的向量是非常稀疏的，也就是大部分的值都是 0。这样的向量会导致计算机的计算量非常大，而且也不利于计算机的计算。所以 One-Hot 编码的缺点就是向量维度大，计算量大，计算效率低。

TF-IDF 编码

TF-IDF 是一个统计学，通过评估一个词对一个语料的重要程度。TF-IDF 是 Term Frequency - Inverse Document Frequency 的缩写，中文叫做词频-逆文档频率。TF-IDF 的主要思想是：如果某个词在一篇文章中出现的频率高，并且在其他文章中很少出现，那么这个词就是这篇文章的关键词。一般我们习惯把这个概念拆分，分为 TF 和 IDF 两个部分。

TF - 词频

词频（Term Frequency）指的是某个词在文章中出现的频率。词频的计算公式如下：

```
TF = 某个词在文章中出现的次数 / 文章的总词数
```

TF 有一个问题，就是如果一个词在文章中出现的次数很多，那么这个词的 TF 值就会很大。这样的话，我们就会认为这个词是这篇文章的关键词。但是这样的话，我们会发现，很多词都是这篇文章的关键词，这样的话，我们就无法区分哪些词是这篇文章的关键词了。所以我们需要对 TF 进行一些调整，这个调整就是 IDF。

IDF - 逆文档频率

逆文档频率（Inverse Document Frequency）指的是某个词在所有文章中出现的频率。逆文档频率的计算公式如下：

```
IDF = log(语料库的文档总数 / (包含该词的文档数 + 1))
```

IDF 的计算公式中，分母加 1 是为了避免分母为 0 的情况。IDF 的计算公式中，语料库的文档总数是固定的，所以我们只需要计算包含该词的文档数就可以了。如果一个词在很多文章中都出现，那么这个词的 IDF 值就会很小。如果一个词在很少的文章中出现，那么这个词的 IDF 值就会很大。这样的话，我们就可以通过 TF 和 IDF 的乘积来计算一个词的 TF-IDF 值。TF-IDF 的计算公式如下：

```
TF-IDF = TF * IDF
```

TF-IDF 经常用于文本分类的场景，这也是 TF-IDF 最常用的场景。TF-IDF 的优点是简单，容易理解，而且计算量也不大。TF-IDF 的缺点是没有考虑词的顺序，而且没有考虑词与词之间的关系。所以 TF-IDF 适合用于文本分类的场景，而不适合用于文本生成的场景。

Word2Vec 编码

Word2Vec 我们也叫它做 Word Embeddings，中文叫做词嵌入。Word2Vec 的主要思想是：一个词的语义可以通过它的上下文来确定。Word2Vec 有两种模型，分别是 CBOW 和 Skip-Gram。CBOW 是 Continuous Bag-of-Words 的缩写，中文叫做连续词袋模型。Skip-Gram 是 Skip-Gram Model 的缩写，中文叫做跳字模型。CBOW 模型的思想是通过一个词的上下文来预测这个词。Skip-Gram 模型的思想是通过一个词来预测这个词的上下文。Word2Vec 的优点是可以得到词的语义，而且可以得到词与词之间的关系。对比起 One-Hot 编码和 TF-IDF 编码，Word2Vec 编码的优点是可以得到词的语义，而且可以得到词与词之间的关系。Word2Vec 编码的缺点是计算量大，而且需要大量的语料库。

之前我们提及过，One-Hot 编码的维度是词的个数，而 Word2Vec 编码的维度是可以指定的。一般我们会指定为 100 维或者 300 维。Word2Vec 编码的维度越高，词与词之间的关系就越丰富，但是计算量也就越大。Word2Vec 编码的维度越低，词与词之间的关系就越简单，但是计算量也就越小。Word2Vec 编码的维度一般是 100 维或者 300 维，这样的维度可以满足大部分的应用场景。

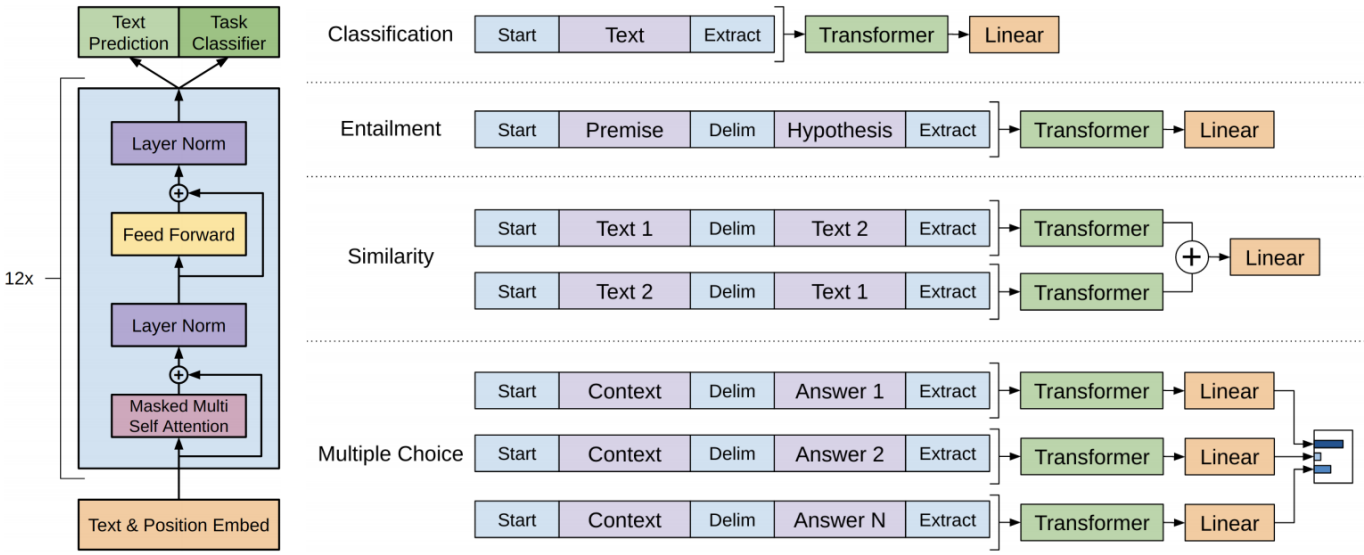
Word2Vec 编码的计算公式非常简单，就是 Word Embeddings。Word Embeddings 是一个词向量，它的维度是可以指定的。Word Embeddings 的维度一般是 100 维或者 300 维，这样的维度可以满足大部分的应用场景。Word Embeddings 的计算公式如下：

```
Word Embeddings = 词的语义 + 词与词之间的关系
```

可以把 Word2Vec 看作是简单化的神经网络。

GPT 模型

GPT 模型的全称是 Generative Pre-Training，中文叫做预训练生成模型。GPT 模型是 OpenAI 在 2018 年提出的，它的主要思想是：一个词的语义可以通过它的上下文来确定。GPT 模型的优点是可以得到词的语义，而且可以得到词与词之间的关系。GPT 模型的缺点是计算量大，而且需要大量的语料库。GPT 模型的结构是一个多层单向的 Transformer 结构，它的结构如下图所示：



训练过程是两个阶段，第一个阶段是预训练，第二个阶段是微调。预训练的语料是维基百科和 BookCorpus，微调的语料是不同的自然语言任务。预训练的目标是通过一个词的上下文来预测这个词，微调的目标是根据不同的自然语言任务，如文本分类、文本生成、问答系统等，对语义模型进行微调，得到不同的模型。

GPT 模型已经经历了 4 个阶段，最为出名的就是 ChatGPT 所使用的 GPT-3.5 以及 GPT 4 。GPT 开启了全新的时代，它的出现让我们看到了自然语言处理的无限可能。GPT 模型的优点是可以得到词的语义，而且可以得到词与词之间的关系。GPT 模型的缺点是计算量大，而且需要大量的语料库。很多人希望拥有自己行业对标的 GPT ，这也是我们本章需要解决的问题。

BERT 编码

BERT 是 Bidirectional Encoder Representations from Transformers 的缩写，中文叫做双向编码器的 Transformer。BERT 是一个预训练的模型，它的训练语料是维基百科和 BookCorpus。BERT 的主要思想是：一个词的语义可以通过它的上下文来确定。BERT 的优点是可以得到词的语义，而且可以得到词与词之间的关系。对比起 One-Hot 编码、TF-IDF 编码和 Word2Vec 编码，BERT 编码的优点是可以得到词的语义，而且可以得到词与词之间的关系。BERT 编码的缺点是计算量大，而且需要大量的语料库。

Embeddings 向量嵌入技术

我们提及了 One-Hot 编码、TF-IDF 编码、Word2Vec 编码、BERT 编码和 GPT 模型。这些编码和模型都是 Embeddings 嵌入技术的一种。Embeddings 嵌入技术的主要思想是：一个词的语义可以通过它的上下文来确定。Embeddings 嵌入技术的优点是可以得到词的语义，而且可以得到词与词之间的关系。Embeddings 是作为自然语言深度学习的基础，它的出现让我们看到了自然语言处理的无限可能。

对于文本内容的 Embeddings 方法，我们结合上一节，你会发现从 word2vec 技术诞生后，文本内容的 Embeddings 就不断得到加强，从 word2vec 到 GPT 再到 BERT ,Embeddings 技术的效果越来越好 。Embeddings 技术的本质就是“压缩”，用更少的维度来表示更多的信息。这样的好处是可以节省存储空间，提高计算效率。

在 Azure OpenAI Service 中，Embeddings 技术的应用非常广泛，将文本字符串转换为浮点向量，通过向量之间的距离来衡量文本之间的相似度。不同行业希望加入自己的数据 我们就可以把这些企业级的数据通过 OpenAI Embeddings - text-embedding-ada-002 模型查询出向量，并通过映射进行保存，在使用时将问题也转换为向量，通过相似度的算法对比，找出最接近的 TopN 结果，从而找到与问题相关联的企业内容。

我们可以通过向量数据库将企业数据向量化后保存，结合 text-embedding-ada-002 模型通过向量的相似度进行查询，从而找到与问题相关联的企业内容。现在常用的向量数据库就包括 Qdrant, Milvus, Faiss, Annoy, NMSLIB 等。

Open AI 的 Embeddings 模型

OpenAI 的文本嵌入向量文本字符串的相关性。 嵌入通常用于以下场景

- 搜索（结果按与查询字符串的相关性排序）
- 聚类（其中文本字符串按相似性分组）
- 推荐（推荐具有相关文本字符串的项目）
- 异常检测（识别出相关性很小的异常值）
- 多样性测量（分析相似性分布）
- 分类（其中文本字符串按其最相似的标签分类）

嵌入是浮点数的向量（列表）。 两个向量之间的距离衡量它们的相关性。 小距离表示高相关性，大距离表示低相关性。 例如，如果您有一个嵌入为[0.1,0.2,0.3]的字符串“狗”，则该字符串与嵌入为[0.2,0.3,0.4]的字符串“猫”比与嵌入为[0.9,0.8,0.7]的字符串“汽车”更相关。

Semantic Kernel 的 Embeddings

在 Semantic Kernel 中对 Embeddings 的支持非常好，除了支持 text-embedding-ada-002 外，也对向量数据库进行支持。 Semantic Kernel 对向量数据库做了抽象，开发人员可以用一致的 API 进行向量数据库的调用。 ***本次案例以 Qdrant 为例***，为了您顺利运行例子，请先安装好 Docker，并安装好 Qdrant 容器并运行，运行脚本如下：

```
docker pull qdrant/qdrant

docker run -p 6333:6333 qdrant/qdrant
```

使用方式如下：

.NET 场景

添加 Nuget 库

```
#r "nuget: Microsoft.SemanticKernel.Connectors.Qdrant, *-*"
```

引用库

```
using Microsoft.SemanticKernel.Memory;
using Microsoft.SemanticKernel.Connectors.Qdrant;
```

创建实例和 Memory 绑定

```
var textEmbedding = new AzureOpenAITextEmbeddingGenerationService("Your
Azure OpenAI Service Embedding Models Deployment Name" , "Your Azure
OpenAI Service Endpoint", "Your Azure OpenAI Service API Key");


var qdrantMemoryBuilder = new MemoryBuilder();
qdrantMemoryBuilder.WithTextEmbeddingGeneration(textEmbedding);
qdrantMemoryBuilder.WithQdrantMemoryStore("http://localhost:6333", 1536);

var qdrantBuilder = qdrantMemoryBuilder.Build();
```

注意： Memory 组件还在调整阶段所以你需要注意接口有改变风险，还需要忽略以下信息

```
#pragma warning disable SKEXP0003
#pragma warning disable SKEXP0011
```

```
#pragma warning disable SKEXP0026
```

Python 场景

引用库

```
from semantic_kernel.connectors.ai.open_ai import AzureChatCompletion,
AzureTextEmbedding
from semantic_kernel.connectors.memory.qdrant import QdrantMemoryStore
```

添加模块支持

```
kernel.add_text_embedding_generation_service(
        "embeddings_services", AzureTextEmbedding("EmbeddingModel",
endpoint,api_key=api_key,api_version = "2023-07-01-preview")
)
```

添加 Memory 链接

```
qdrant_store = QdrantMemoryStore(vector_size=1536,
url="http://localhost",port=6333)
await qdrant_store.create_collection_async('aboutMe')
kernel.register_memory_store(memory_store=qdrant_store)
```

注意： Memory 组件还在调整阶段所以你需要注意接口有改变风险

Semantic Kernel 中保存和搜索您的向量

在 Semantic Kernel 中，通过抽象的方式把不同向量数据的方法进行了统一，您可以很方便地保存和搜索您的向量

.NET 场景

保存向量数据

```
await qdrantBuilder.SaveInformationAsync(conceptCollectionName, id:
```

```
"info1", text: "Kinfey is Microsoft Cloud Advocate");
await qdrantBuilder.SaveInformationAsync(conceptCollectionName, id:
"info2", text: "Kinfey is ex-Microsoft MVP");
await qdrantBuilder.SaveInformationAsync(conceptCollectionName, id:
"info3", text: "Kinfey is AI Expert");
await qdrantBuilder.SaveInformationAsync(conceptCollectionName, id:
"info4", text: "OpenAI is a company that is developing artificial general
intelligence (AGI) with widely distributed economic benefits.");
```

查询向量数据

```
string questionText = "Do you know kinfey ?";
var searchResults =  qdrantBuilder.SearchAsync(conceptCollectionName,
questionText, limit: 3, minRelevanceScore: 0.7);

await foreach (var item in searchResults)
{
    Console.WriteLine(item.Metadata.Text + " : " + item.Relevance);
}
```

Python 场景

保存向量数据

```
await kernel.memory.save_information_async(base_vectordb, id="info1",
text="Kinfey is Microsoft Cloud Advocate")
await kernel.memory.save_information_async(base_vectordb, id="info2",
text="Kinfey is ex-Microsoft MVP")
await kernel.memory.save_information_async(base_vectordb, id="info3",
text="Kinfei is AI Expert")
await kernel.memory.save_information_async(base_vectordb, id="info4",
text="OpenAI is a company that is developing artificial general
intelligence (AGI) with widely distributed economic benefits.")
```

查询向量数据

```
ask = "who is kinfey?"

memories = await kernel.memory.search_async(
```

```
    base_vectordb, ask, limit=3, min_relevance_score=0.8
)

i = 0
for memory in memories:
    i = i + 1
    print(f"Top {i} Result: {memory.text} with score {memory.relevance}")
```

您可以非常简单方便地接入任意的向量数据库来完成相关的操作，也意味着可以非常简单地构建 RAG 应用。

例子

.NET 例子 请点击访问这里

Python 例子 请点击访问这里

小结

现在很多的企业数据进入到 LLMs 使用的都是通过 Embeddings 的方式构建 RAG 应用。 Semantic Kernel 无论在 Python 还是 .NET 都给了我们非常简单的方式来完成相关的功能，所以对于一些希望在工程增加 RAG 应用的人来说是非常大的帮助。