Embeddings & Vector Stores

エンベディング（埋め込み）とベクトル検索

Authors: Anant Nawalgaria,
Xiaoqi Ren, and Charles Sugnet

Google

Acknowledgements

Content contributors

Antonio Gulli

Grace Mollison

Ruiqi Guo

Iftekhar Naim

Jinhyuk Lee

Alan Li

Patricia Florissi

Andrew Brook

Omid Fatemieh

Zhuyun Dai

Lee Boonstra

Per Jacobsson

Siddhartha Reddy Jonnalagadda

Xi Cheng

Raphael Hoffmann

Curators and Editors

Antonio Gulli

Anant Nawalgaria

Grace Mollison

Technical Writer

Joey Haymaker

Designer

Michael Lanning

Table of contents

**実世界のデータを表現する
これらの低次元の数値表現は、
元のデータの非可逆圧縮の
手段として機能することにより、
効率的な大規模データ処理および
保存を大幅に支援します。**

はじめに

現代の機械学習は、画像、テキスト、音声など、多様なデータを活用することで発展しています。本書では、これらの異種データを様々なアプリケーションでシームレスに利用できるよう、統一されたベクトル表現へと変換する「埋め込み」の持つ力について解説します。

本書では、以下の内容についてご案内します。

- **埋め込みの理解：**なぜ埋め込みがマルチモーダルデータの扱いに不可欠であり、どのような多様な応用があるのか。
- **埋め込み技術：**様々なデータ型を共通のベクトル空間に射影する手法。
- **効率的な管理：**膨大な埋め込みのコレクションを保存、取得、検索するための技術。
- **ベクトルデータベース：**埋め込みの管理とクエリ発行に特化したシステム。本番環境への導入に関する実践的な考慮事項も扱います。
- **実世界の応用例：**埋め込みとベクトルデータベースを大規模言語モデル（LLM）と組み合わせ、実世界の問題を解決する具体的な事例。

本書全体を通して、コードスニペットを用いて主要な概念を具体的に解説します。

なぜ埋め込みは重要なのか

本質的に、埋め込みとは、テキスト、音声、画像、動画などの実世界のデータを数値で表現したものです。「埋め込み」という名称は、ある空間を別の空間に射影、すなわち埋め込むことができるという数学の類似概念に由来します。例えば、オリジナルの BERT モデル [ref] は、テキストを 768 個の数値からなるベクトルに埋め込みます。これにより、あらゆる文が構成する非常に高次元の空間から、はるかに小さな 768 次元空間への射影が行われます。埋め込みは低次元ベクトルとして表現され、そのベクトル空間における 2 つのベクトル間の幾何学的距離は、それらのベクトルが表現する 2 つの実世界のオブジェクト間の関係性や意味的類似性を反映したものとなります。言い換えれば、埋め込みは、様々な種類のデータをコンパクトに表現するのに役立つと同時に、2 つの異なるデータオブジェクトを比較し、それらが数値スケール上でどの程度類似しているか、あるいは異なっているかを示すことを可能にします。例えば、「コンピューター」という単語は、コンピューターの画像や「ラップトップ」という単語とは類似した意味を持ちますが、「車」という単語とは意味が異なります。実世界のデータを表現するこれらの低次元の数値表現は、元のデータの非可逆圧縮の手段として機能することにより、その重要な意味的特性を保持しつつ、効率的な大規模データ処理および保存を大幅に支援します。

埋め込みに関する直感的な理解のために、地球上の位置を 2 つの数値の組、すなわち長さ 2 のベクトルに射影するために用いられる、お馴染みの緯度と経度を考えてみましょう。緯度と経度は、特定の場所の埋め込みと考えることができます。今となっては自明に思えるかもしれませんが、場所を 2 つの数値の組に射影するというこの単純な方法は、人間のナビゲーションに変革をもたらし、今日においても不可欠なものです。2 つの住所の緯度と経度が与えられれば、それらが互いにどれだけ離れているかを確認したり、近隣の他の場所を検索したりすることは比較的容易です。緯度と経度の場合と同様に、2 つのテキスト埋め込みが埋め込み空間内で近接していれば、それらはテキストとしての意味が意味的に類似していることになります。また、そのベクトル空間内で近傍を探索することにより、意味的に類似した新たなテキストフレーズを見つけることも可能です。ベクトルデータベースを用いて、非常に大規模なデータセットの中から類似アイテムを極めて低いレイテンシで発見できるこの能力は、検索、推薦、広告、不正検知など、今日の多くの本番環境におけるユースケースにとって不可欠です。注意すべき点として、緯度と経度の埋め込みモデルが地球の球形に基づいて設計されたのに対し、テキストの埋め込み空間はニューラルネットワークモデルによって学習されます。重要なこととして、異なるモデルによって学習された埋め込みは相互に比較可能ではなく、実運用においては互換性があり一貫したバージョンの埋め込みが使用されていることを確認することが極めて重要です。

埋め込みの主要な応用分野は検索と推薦であり、これらでは通常、広大な検索空間から結果が選択されます。例えば、Google 検索は、インターネット全体を検索空間とする検索タスクです。今日の検索システムや推薦システムの成功は、以下のステップにかかっています。

1. 検索空間内の何十億ものアイテムに対する埋め込みを事前に計算しておくこと。
2. クエリ埋め込みを同じ埋め込み空間に射影すること。
3. 検索空間内でクエリ埋め込みの最近傍となるアイテムの埋め込みを効率的に計算し、取得すること。

埋め込みは、マルチモーダリティの世界においてもその真価を発揮します。多くのアプリケーションでは、テキスト、音声、画像、動画など、様々なモダリティの大量のデータを扱います。共同埋め込み（ジョイント埋め込み）とは、複数の種類のオブジェクトを同じ埋め込み空間に射影することであり、例えばテキストクエリに基づいて動画を検索するような場合がこれにあたります。これらの埋め込み表現は、元のオブジェクトの特性を可能な限り多く捉えるように設計されています。

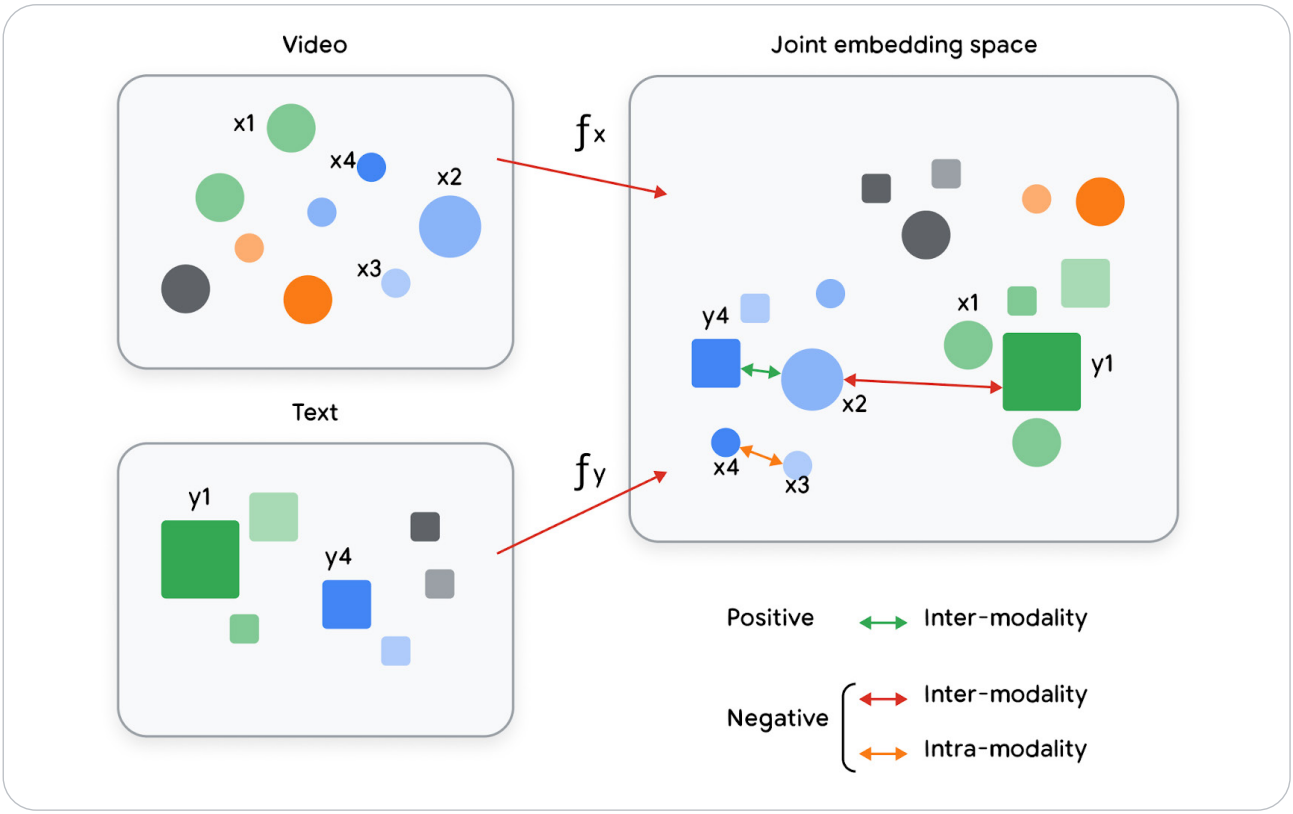

図 1：オブジェクト / コンテンツを意味情報を持つ共通ベクトル空間へ射影する

埋め込みは、類似した意味的特性を持つオブジェクトが埋め込み空間（アイテムを射影できる低次元ベクトル空間）内でより近くなるように設計されています。そして、この埋め込みは、凝縮された意味のある入力として、後続のアプリケーションで使用することができます。例えば、機械学習モデル（ML モデル）の特徴量、推薦システム、検索エンジンなど、その他多くの用途に利用できます。したがって、データはコンパクトな数値表現を得るだけでなく、その表現は特定のタスクや様々なタスクにわたって意味的な内容を保持します。これらの表現がタスク固有であるという事実は、特定のタスクに合わせて最適化された、同じオブジェクトに対する異なる埋め込みを生成できることを意味します。

埋め込みの品質評価

埋め込みモデルは、タスクによって評価方法が異なります。品質を評価するための一般的な評価指標の多くは、類似アイテムを検索し、類似していないアイテムを除外する能力に焦点を当てています。この種の評価には、関連文書または正解文書が既に判明しているラベル付きデータセットが必要です。これは、スニペット 0 で NFCorpus データセットを用いて様々な評価指標を説明している箇所で見られる通りです。前述の検索ユースケースにおいて品質を評価するための 2 つの重要な評価指標は、1) 適合率 (precision)：検索されたすべての文書が関連しているべきである、および 2) 再現率 (recall)：すべての関連文書が検索されるべきである、です。直感的には、最適な埋め込みモデルはすべての関連文書を検索し、関連のない文書は一切検索しないということになりますが、実際には一部の関連文書が除外されたり、一部の無関係な文書が検索されたりすることが多いため、大規模な文書セットや埋め込みモデルに対する品質評価には、より定量的な定義が必要となります。適合率は、検索された関連文書の数を、検索された総文書数で割ることによって定量化されます。これは、特定の検索文書数に対してよく引用されます。例えば、ある埋め込みに対して 10 件の文書が検索され、そのうち 7 件が関連文書で他の 3 件が関連していなかった場合、適合率 @10 (precision@10) は 7/10=0.7 となります。再現率は、関連文書のうちどれだけが検索されたかを示し、検索された関連文書の数をコーパス中の総関連文書数で割ることによって計算されます。再現率もまた、特定の検索文書数に対してよく引用されます。例えば、20 件の文書が検索され、そのうち 3 件が関連文書であったが、コーパス中には合計 6 件の関連文書が存在した場合、再現率 @20 (recall@20) は 3/6=0.5 となります。

適合率と再現率は、関連性スコアが 2 値である場合には非常に有用ですが、文書によって関連性の度合いが異なるケースを捉えることはできません。例えば、検索エンジンを使用する場合、たとえ検索結果がすべて関連文書であったとしても、エンドユーザーはそれらの順序に敏感であるため、最も関連性の高い結果がリストの最上位に表示されることが強く望まれます。データセットに対して文書の関連性の詳細な順序が既知である場合、正規化割引累積ゲイン (nDCG: Normalized Discounted Cumulative Gain) のような評価指標を用いることで、埋め込みモデルによって生成されたランキングの品質を、望ましいランキングと比較して測定することができます。位置 p における DCG (割引累積ゲイン) の式は DCGp=(数式) であり、ここで reli は関連性スコアです。分母はリストの下位にある文書に対してペナルティを課し、DCG は最も関連性の高い文書がリストの最上位にある場合にスコアが最大化されます。正規化版は、DCG スコアを理想的な順序付けのスコアで割ることによって計算され、異なるクエリ間での比較のために 0.0 から 1.0 の範囲の値を取ります。

BEIR[42] のような公開ベンチマークは検索タスクの性能評価に広く用いられており、Massive Text Embedding Benchmark (MTEB)[43] のようなベンチマークは追加のタスクをカバーしています。実務家は、適合率、再現率、nDCG などを計算する際に、Text Retrieval Conference (TREC) から生まれた trec_eval[44] や pytrec_eval[45] のような Python ラッパーなどの標準ライブラリを使用し、他の手法との一貫したベンチマーク測定を行うことが推奨されます。特定のアプリケーションに対する埋め込みモデルの最適な評価方法はアプリケーション固有であるかもしれませんが、より類似したオブジェクトは埋め込み空間内でより近くなるべきであるという直感は、多くの場合、良い出発点となります。モデルサイズ、埋め込み次元数、レイテンシ、全体的なコストといった追加の評価指標も、本番アプリケーションにおける重要な考慮事項です。

検索の例

様々な種類の埋め込みや埋め込みモデル開発の歴史について詳述する前に、前述の検索の例をさらに詳しく見ていきましょう。目標は、ユーザーからのクエリに基づき、大規模なコーパスの中から関連文書を見つけることです。そのための 1 つのアプローチは、質問と回答が埋め込み空間内の類似した位置に射影されるような共同埋め込みモデルを構築することです。質問と回答は、たとえ補完的であっても意味的には異なるため、質問用と文書用にそれぞれ 1 つずつ、共同で学習された 2 つのニューラルネットワークを使用することが有効な場合が多くあります。この視覚的な表現は、図 9(b) の非対称デュアルエンコーダーで見ることができます。これはクエリ用と文書用に別々のネットワークを持つものであり、対照的に図 9(a) ではクエリと文書の両方に単一のニューラルネットワークを使用する、シャムネットワーク (Siamese ネットワーク) とも呼ばれるものが示されています。

図 2 は、検索拡張生成 (RAG) アプローチを用いた検索質問応答アプリケーションの図です。このアプローチでは、埋め込みを使用して関連文書を特定し、それらを LLM のプロンプトに挿入してエンドユーザー向けに要約を生成します。このアプリケーションは、主に 2 つのプロセスに分かれています。第一に、インデックス作成です。ここでは文書がチャンクに分割され、それらを用いて埋め込みが生成され、低レイテンシ検索のためにベクトルデータベースに保存されます。具体的には、これらのチャンクに対して、デュアルエンコーダーニューラルネットワークモデルの文書埋め込み部分が使用されます。第二のフェーズは、ユーザーがシステムに質問をすると、その質問がモデルのクエリ埋め込み部分を用いて埋め込まれ、ベクトルデータベースでの類似性検索によって関連文書にマッピングされるというものです。この第二のフェーズは、エンドユーザーが積極的に応答を待っているため、非常にレイテンシに敏感です。そのため、文書のベクトルデータベースを使用して大規模コーパスからミリ秒単位で関連文書を特定できる能力は、インフラストラクチャの重要な要素となります。

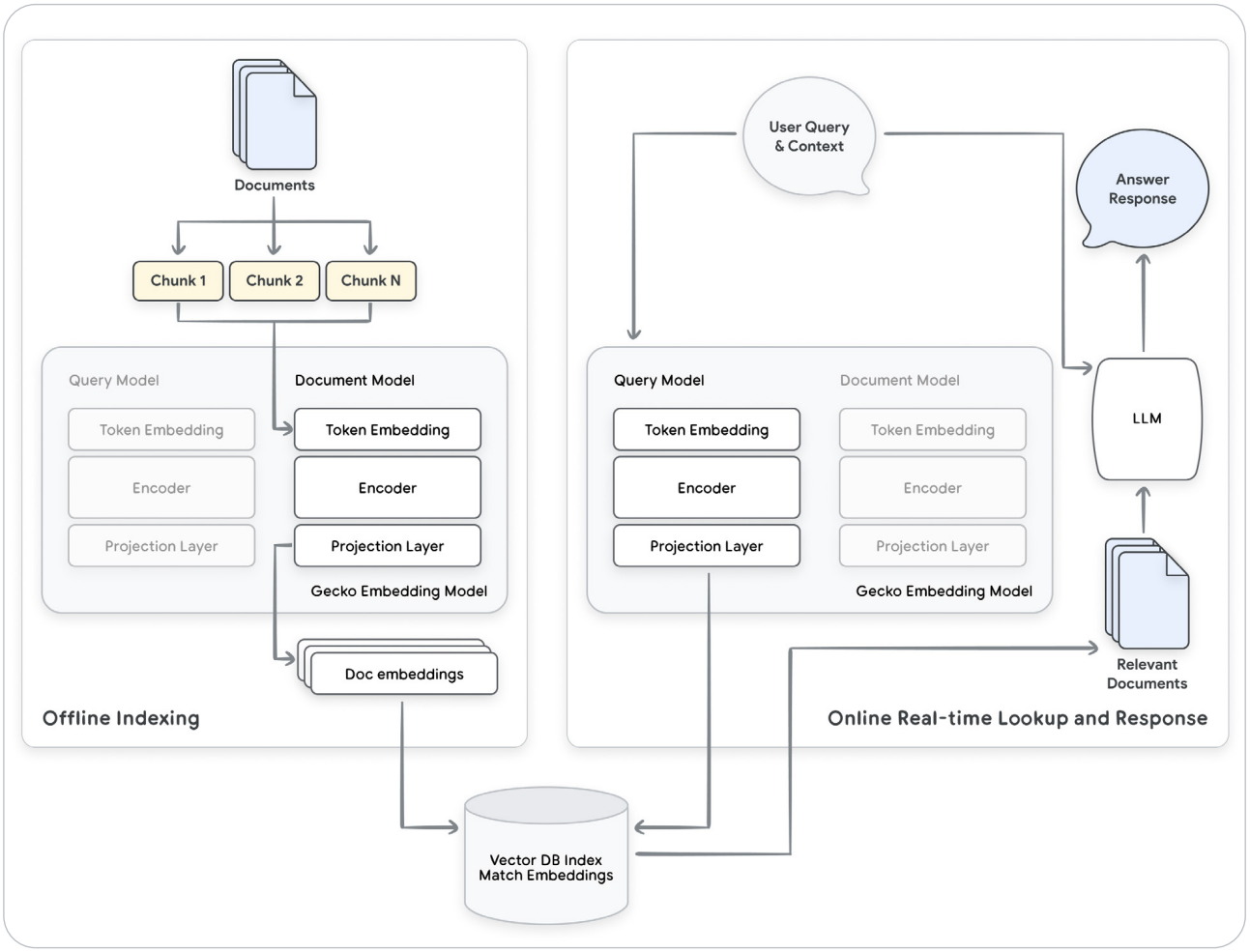

図 2：埋め込みに焦点を当てた RAG 検索アプリケーションのフロー例。文書埋め込みはバックグラウンドで生成され、ベクトルデータベースに保存されます。ユーザーがクエリを入力すると、デュアルエンコーダーのクエリ埋め込み部分を使用して埋め込みが生成され、関連文書の検索に使用されます。これらの文書は LLM のプロンプトに挿入され、ユーザー向けの関連性の高い要約応答が生成されます。

埋め込みモデルの品質は、BERT の登場以来急速に向上しており、その勢いは当面衰える気配がありません。近年、AI 分野では LLM が多くの注目を集めていますが、情報検索と埋め込みモデルの進歩もまた変革的でした。オリジナルの BERT モデルは当時としては飛躍的な進歩であり、BEIR ベンチマークで平均スコア 10.6 でしたが、Google による現在の (2025 年の) 埋め込みは、単純な API 呼び出しで AI の知識を必要とせずに、現在では平均 BEIR スコア 55.7 を達成しています。モデルは急速に改善し続けているため、埋め込みモデルを本番環境に導入する際には、モデルのアップグレードを念頭に置いた設計を心がけてください。特定のアプリケーション向けに設計された優れた評価スイートは、スムーズなアップグレードを保証するために不可欠です。アップグレードパスが整備されているプラットフォームで埋め込みモデルを選択することは、開発者の時間を節約し、AI の深い専門知識を持たないチームの運用オーバーヘッドを削減するのに役立ちます。例えば、後述のスニペット 0 では、Google Vertex 経由の単純な API 呼び出しを使用しています。

スニペット 1 には、健康関連の質問と文書を含む NFCorpus データセット [46] を使用した埋め込みに関して、上記で取り上げた重要な概念のいくつかを説明するための基本的な埋め込みコードサンプルが含まれています。

- クエリに関連する情報を含むテキスト文書は、高品質と運用容易性の両方を実現するために Google Vertex API を使用して埋め込まれます。質問と回答はしばしば異なる表現で記述されるため、RETRIEVAL_DOCUMENT タスクタイプが使用されます。意味的類似性を持つ単一モデルを使用すると、文書とクエリの共同埋め込みと比較してパフォーマンスが低下する可能性があります。
- 埋め込みは、効率的な類似性検索のために faiss ライブラリ [47] を使用して保存されます。
- 特定のクエリに対しては、RETRIEVAL_QUERY タスクタイプを使用してテキスト埋め込みが生成されます。
- クエリ埋め込みは、faiss ライブラリによって、デフォルトのユークリッド距離メトリックを使用して埋め込みが近い文書の ID を検索するために使用されます。
- すべてのクエリに対する埋め込みが生成され、最も類似した文書が検索されます。検索品質は、pytrec ライブラリを使用して適合率 @1、再現率 @10、nDCG@10 といった評価指標を測定し、「正解」値と比較して評価されます。

```
from beir import util
from beir.datasets.data_loader import GenericDataLoader
import faiss
import vertexai
from vertexai.language_models import TextEmbeddingInput, TextEmbeddingModel
import numpy as np
import pandas as pd
import pytrec_eval

def embed_text(texts, model, task, batch_size=5) :
 embed_mat = np.zeros((len(texts),768))
 for batch_start in range(0,len(texts),batch_size):
   size = min(len(texts) - batch_start, batch_size)
   inputs = [TextEmbeddingInput(texts[batch_start+i], task_type=task) for i in range(size)]
   embeddings = model.get_embeddings(inputs)
   for i in range(size) :
     embed_mat[batch_start + i, :] = embeddings[i].values
 return embed_mat

# Download smallish NFCorpus dataset of questions and document text
url = "https://public.ukp.informatik.tu-darmstadt.de/thakur/BEIR/datasets/nfcorpus.zip"
data_path = util.download_and_unzip(url, "datasets")
# Corpus of text chunks, text queries and “gold” set of query to relevant documents dict
corpus, queries, qrels = GenericDataLoader("datasets/nfcorpus").load(split="test")

# Note need to setup Google Cloud project and fill in id & location below
vertexai.init(project="PROJECT_ID", location="LOCATION")
model = TextEmbeddingModel.from_pretrained("text-embedding-005")
doc_ids,docs = zip(*[(doc_id, doc['text']) for doc_id,doc in corpus.items()])
q_ids,questions = zip(*[(q_id, q) for q_id,q in queries.items()])
```

次のページに続きます

```
# Embed the documents and queries jointly using different models
doc_embeddings = embed_text(docs, model, "RETRIEVAL_DOCUMENT")
index = faiss.IndexFlatL2(doc_embeddings.shape[1])
index.add(doc_embeddings)

# Example look up example query to find relevant doc - note using 'RETRIEVAL_QUERY'
example_embed = embed_text(['Is Caffeinated Tea Really Dehydrating?'],
model, 'RETRIEVAL_QUERY')
s,q = index.search(example_embed,1)
print(f'Score: {s[0][0]:.2f}, Text: "{docs[q[0][0]]}"')
# Score: 0.49, Text: "There is a belief that caffeinated drinks, such as tea,
# may adversely affect hydration. This was investigated in a randomised
# controlled trial ... revealed no significant differences
# between tea and water for any of the mean blood or urine measurements…"

# Embed all queries to evaluate quality compared to "gold" answers
query_embeddings = embed_text(questions, model, "RETRIEVAL_QUERY")
q_scores, q_doc_ids = index.search(query_embeddings, 10)
# Create a dict of query to document scores dict for pytrec evaluation
# Multiply scores by -1 for sorting as smaller distance is better score for pytrec eval
search_qrels = { q_ids[i] : { doc_ids[_id] : -1*s.item() for _id, s in zip(q_doc_ids[i], q_
scores[i])} for i in range(len(q_ids))}
evaluator = pytrec_eval.RelevanceEvaluator(qrels, {'ndcg_cut.10','P_1','recall_10'})
eval_results = evaluator.evaluate(search_qrels)
df = pd.DataFrame.from_dict(eval_results, orient='index')
df.mean()
#P_1            0.517028 // precision@1
#recall_10      0.203507 // recall@10
#ndcg_cut_10    0.402624 // nDCG@10
```

スニペット 1：テキスト埋め込みを用いたセマンティック検索と検索文書の品質評価の例。

ニューラルネットワークの学習と評価の両方には、スニペット 0 で使用されている NFCorpus のような、質問と関連文書のペアを含むデータセットが必要です。特定のアプリケーションの学習または評価に最適なデータセットは、そのアプリケーションの性質によって異なります。例えば、医療アプリケーションは、法務ユースケースに焦点を当てたアプリケーションとは異なる専門用語や慣習を使用します。これらのラベル付きデータセットは、人間の専門家を使用して生成するには費用と時間がかかる場合があります。Google DeepMind による Gecko 埋め込みモデルの論文 [48] では、LLM を使用して学習用に大量の合成的な質問と文書のペアを生成し、モデルの改善と多くのベンチマークでのパフォーマンス向上につながった経緯が詳述されています。LLM を利用して専門家による学習データの生成を支援したり、回答の評価を行ったりすることは、学習、チューニング、評価用データセットをコスト効率よく大規模に拡張するための効果的な方法となり得ます。

埋め込みの種類

埋め込みは、元のデータの「本質的な情報」のほとんどを保持しつつ、その低次元表現を得ることを目指しています。埋め込みが表現するデータの種類は、様々な異なる形式を取り得ます。以下では、テキストや画像を含む様々な種類のデータに対して用いられる、いくつかの標準的な手法を紹介します。

テキスト埋め込み

テキスト埋め込みは、自然言語処理 (NLP) の一部として広く利用されています。これらは、テキスト生成、分類、感情分析などの様々な後続アプリケーションで処理するために、機械学習において自然言語の意味を埋め込むためによく使用されます。これらの埋め込みは、大まかにトークン / 単語埋め込みと文書埋め込みの 2 つのカテゴリに分類されます。

これらのカテゴリをより深く掘り下げる前に、ユーザーによる入力から埋め込みへの変換に至るまでの、テキストのライフサイクル全体を理解することが重要です。

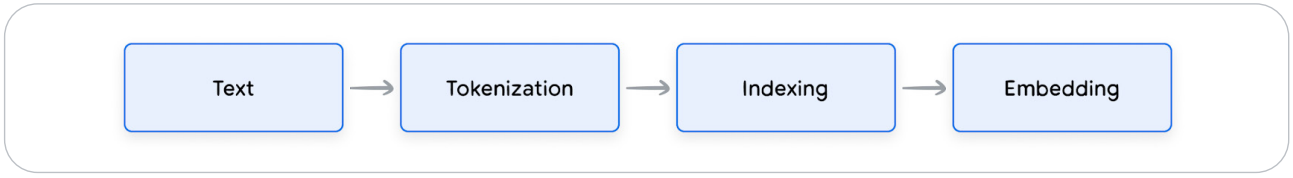

図 3：テキストを埋め込みに変換するプロセス

すべては入力文字列から始まり、これはトークンと呼ばれるより小さな意味のある断片に分割されます。このプロセスはトークン化と呼ばれます。一般的に、これらのトークンは、既存の多くのトークン化手法のいずれかを用いて [1]、ワードピース、文字、単語、数値、句読点となります。文字列がトークン化された後、これらの各トークンには通常、[0, コーパス内の総語彙数 − 1] の範囲で一意の整数値が割り当てられます。例えば、16 単語の語彙の場合、ID は 0 から 15 の範囲になります。この値はトークン ID とも呼ばれます。これらのトークンは、各文字列を、後続タスクで直接使用されるか、または one-hot エンコーディング後に使用される、スパースな数値ベクトル表現として表すために用いることができます。

one-hot エンコーディングはカテゴリ値の 2 値表現であり、単語の存在を 1 で、非存在を 0 で表します。これにより、トークン ID がそのままカテゴリ値として扱われることが保証されますが、多くの場合、コーパスの語彙サイズを持つ密なベクトルが生成されます。スニペット 2 と図 4 は、これを TensorFlow を使用して行う方法の例を示しています。

```
# Tokenize the input string data
from tensorflow.keras.preprocessing.text import Tokenizer
data = [
  "The earth is spherical.",
  "The earth is a planet.",
  "I like to eat at a restaurant."]

# Filter the punctiations, tokenize the words and index them to integers
tokenizer = Tokenizer(num_words=15, filters='!"#$%&()*+,-./:;<=>?[\\]^_'{|}~\t\n', lower=True,
split=' ')
tokenizer.fit_on_texts(data)

# Translate each sentence into its word-level IDs, and then one-hot encode those IDs
ID_sequences = tokenizer.texts_to_sequences(data)
binary_sequences = tokenizer.sequences_to_matrix(ID_sequences)
print("ID dictionary:\n", tokenizer.word_index)
print("\nID sequences:\n", ID_sequences)
print("\n One-hot encoded sequences:\n", binary_sequences)
```

スニペット 2：文字列のトークン化、インデックス化、および one-hot エンコーディング 図 4. スニペット 2 の出力

```
ID dictionary:
 {'the': 1, 'earth': 2, 'is': 3, 'a': 4, 'spherical': 5, 'planet': 6, 'i': 7, 'like': 8, 'to': 9, 'eat': 10, 'at': 11, 'restaurant': 12}

ID sequences:
 [[1, 2, 3, 5], [1, 2, 3, 4, 6], [7, 8, 9, 10, 11, 4, 12]]

 One-hot encoded sequences:
 [[0. 1. 1. 1. 0. 1. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 1. 1. 1. 1. 0. 1. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 1. 0. 0. 1. 1. 1. 1. 1. 1. 0. 0.]]
```

図 4：スニペット 2 の出力

しかし、これらの整数 ID（または対応する one-hot エンコードされたベクトル）はランダムに単語に割り当てられるため、固有の意味的内容を持ちません。ここで埋め込みがはるかに有用になります。文字レベルやサブワードレベルのトークンを埋め込むことも可能ですが、ここでは単語埋め込みと文書埋め込みを取り上げ、それらの背後にあるいくつかの手法を理解しましょう。

単語埋め込み

本節では、現在使用されている最新のテキスト埋め込みの前身となった、単語埋め込みを学習および使用するためのいくつかの単語埋め込み技術とアルゴリズムを紹介します。長年にわたり様々な目的に合わせて最適化された多くの機械学習ベースのアルゴリズムが開発されてきましたが、最も一般的なものには GloVe[2]、SWIVEL[3]、および Word2Vec[4] がありました。単語埋め込みやサブワード埋め込みは、言語モデルの隠れ層から直接得ることもできます。しかし、その場合、同じ単語であってもテキスト内の異なる文脈では埋め込み表現が異なります。本節では、軽量で文脈に依存しない単語埋め込みに焦点を当て、文脈を考慮した文書埋め込みについては文書埋め込みの節で扱います。

単語埋め込みは、固有表現抽出やトピックモデリングのような後続タスクに直接適用することができます。

Word2Vec は、「単語の意味的意味はその隣接語によって定義される」、すなわち学習コーパス中で互いに近接して頻繁に出現する単語によって定義されるという原則に基づいて動作するモデルアーキテクチャ群です。この手法は、大規模データセットから独自の埋め込みを学習するために使用することも、オンラインで入手可能な多数の事前学習済み埋め込みのいずれかを通じて迅速に統合することも可能です [5]

各単語の埋め込み（本質的には固定長ベクトル）は、プロセスを開始するためにランダムに初期化され、結果として形状が (語彙サイズ , 各埋め込みの次元数) の行列が生成されます。この行列は、以下のいずれかの手法を用いて学習プロセスが完了した後、ルックアップテーブルとして使用することができます（[図 5] 参照）。

- **Continuous Bag-of-Words (CBOW) アプローチ：**周辺単語の埋め込みを入力として使用し、中央の単語を予測しようとします。この手法は、文脈における周辺単語の順序に依存しません。このアプローチは学習が速く、高頻度語に対して若干精度が高くなります。
- **Skip-gram アプローチ:**CBOW とは逆の構成で、中央の単語を使用して特定の範囲内の周辺単語を予測します。このアプローチは学習が遅いですが、少量データでもうまく機能し、低頻度語に対して精度が高くなります。

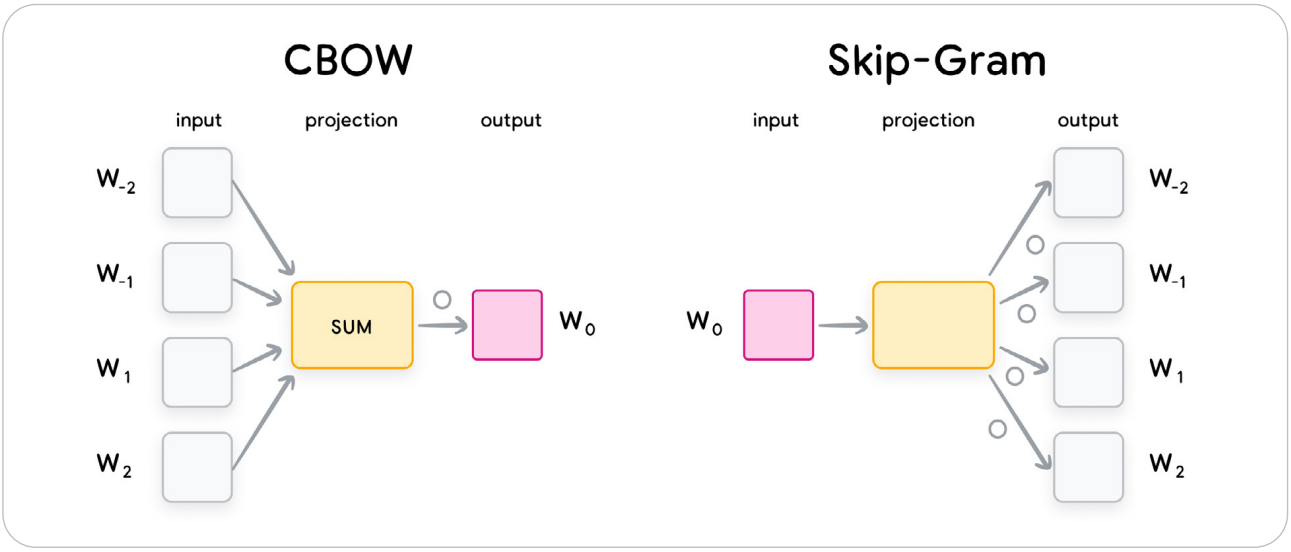

図 5：CBOW および Skip-gram 手法の仕組みを説明する図

Word2Vec アルゴリズムはサブワードレベルにも拡張可能であり、これは FastText[6] のようなアルゴリズムに着想を与えました。しかし、Word2Vec の主な注意点の 1 つは、特定のスライディングウィンドウ内の単語の局所的な統計情報はうまく考慮するものの、大域的な統計情報（コーパス全体の単語）を捉えられないことです。この欠点を解決するのが、GloVe アルゴリズムのような手法です。

GloVe は、単語の大域的および局所的な両方の統計情報を活用する単語埋め込み技術です。これは、まず単語間の関係を表す共起行列を作成することによって行われます。次に、GloVe は分解手法を用いて共起行列から単語表現を学習します。結果として得られる単語表現は、単語に関する大域的および局所的な情報の両方を捉えることができ、様々な NLP タスクに役立ちます。

GloVe に加えて、SWIVEL も共起行列を活用して単語埋め込みを学習する別のアプローチです。SWIVEL は Skip-Window Vectors with Negative Sampling の略です。GloVe とは異なり、SWIVEL は隣接語の固定ウィンドウ内での単語の共起を考慮して、ローカルウィンドウを使用して単語ベクトルを学習します。さらに、SWIVEL は観測されていない共起も考慮し、特殊な区分的損失関数を用いて処理することで、低頻度語に対するパフォーマンスを向上させます。一般的に、平均して GloVe よりもわずかに精度が劣るとされていますが、学習は大幅に速いです。これは、埋め込みベクトルをより小さな部分行列に細分化し、複数のマシンで並列に行列分解を実行する分散学習を活用しているためです。以下の [スニペット 3] は、Word2Vec と GloVe の両方の事前学習済み単語埋め込みをロードし、それらを 2 次元空間で可視化し、最近傍を計算する例を示しています。

```
from gensim.models import Word2Vec
import gensim.downloader as api
import pprint
import matplotlib.pyplot as plt
from sklearn.manifold import TSNE
import numpy as np
def tsne_plot(models, words, seed=23):
  "Creates a TSNE models & plots for multiple word models for the given words"

  plt.figure(figsize=(len(models)*30, len(models)*30))
  model_ix = 0
  for model in models:
    labels = []
    tokens = []

    for word in words:
      tokens.append(model[word])
      labels.append(word)

    tsne_model = TSNE(perplexity=40, n_components=2, init='pca', n_iter=2500, random_state=seed)
    new_values = tsne_model.fit_transform(np.array(tokens))
    x = []
    y = []
    for value in new_values:
      x.append(value[0])
      y.append(value[1])

    model_ix +=1
    plt.subplot(10, 10, model_ix)
    for i in range(len(x)):
      plt.scatter(x[i],y[i])
      plt.annotate(labels[i],
            xy=(x[i], y[i]),
            xytext=(5, 2),
            textcoords='offset points',
            ha='right',
            va='bottom')
  plt.tight_layout()
  plt.show()
v2w_model = api.load('word2vec-google-news-300')
glove_model = api.load('glove-twitter-25')
print("words most similar to 'computer' with word2vec and glove respectively:")
pprint.pprint(v2w_model.most_similar("computer")[:3])
pprint.pprint(glove_model.most_similar("computer")[:3])
pprint.pprint("2d projection of some common words of both models")
sample_common_words= list(set(v2w_model.index_to_key[100:10000])
                      & set(glove_model.index_to_key[100:10000]))[:100]
tsne_plot([v2w_model, glove_model], sample_common_words)
```

スニペット 3：GloVe および Word2Vec 埋め込みの 2 次元でのロードとプロット

図 6 は、2 つのアルゴリズムで意味的に類似した単語が異なる形でクラスタリングされることを示しています。

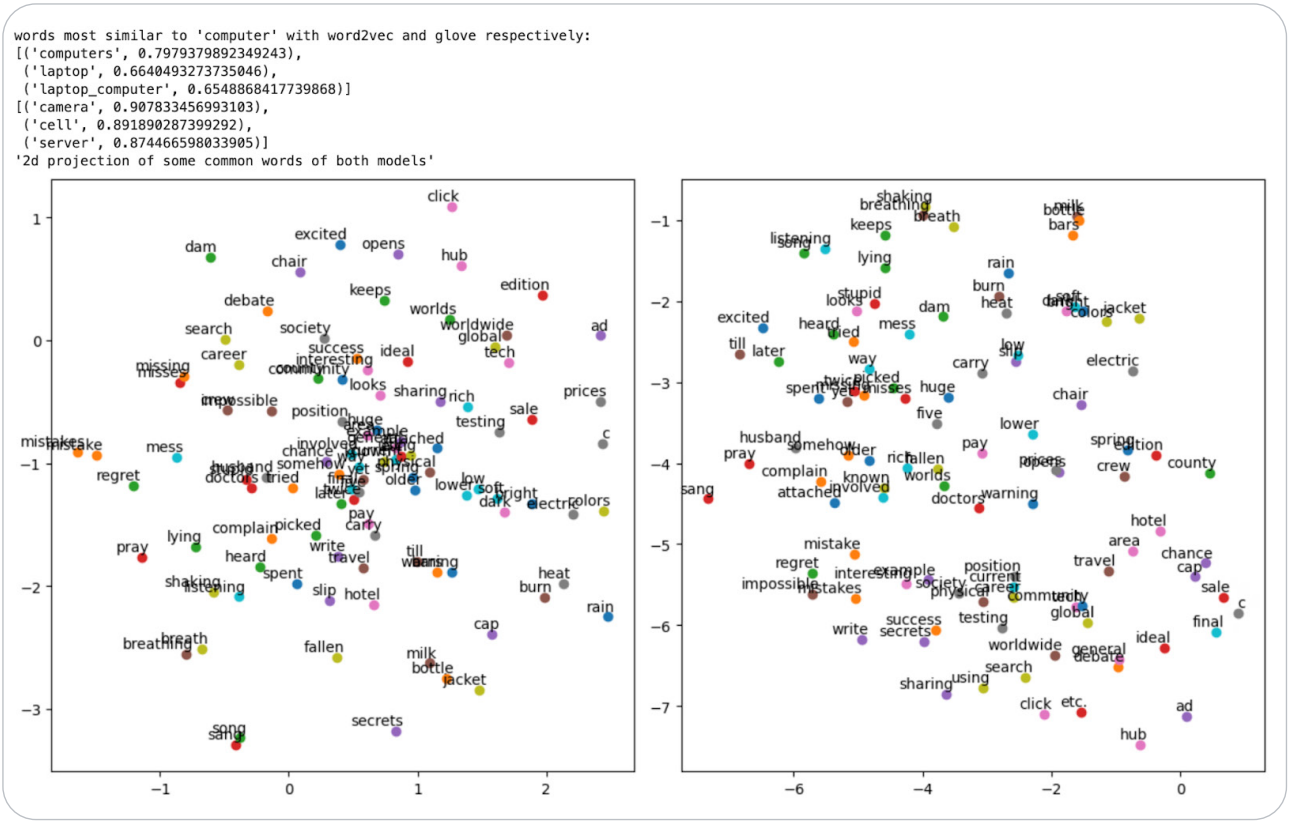

図 6：事前学習済み GloVe および Word2Vec 単語埋め込みの 2 次元可視化

文書埋め込み

文書を低次元の密な埋め込みへと変換する技術は、1980 年代から長年にわたり関心を集めてきました。文書埋め込みは、段落や文書内の一連の単語の意味を埋め込み、それをセマンティック検索、トピック発見、分類、クラスタリングといった様々な後続アプリケーションに利用することができます。埋め込みモデルの進化は、主に、浅い Bag-of-Words (BoW) モデルと、より深い事前学習済み大規模言語モデルという 2 つの段階に分類できます。

浅い BoW モデル

初期の文書埋め込み研究は、文書を順序なしの単語の集合と仮定する Bag-of-Words (BoW) パラダイムに従っています。これらの初期の研究には、潜在意味解析 (LSA)[7] や潜在ディリクレ配分法 (LDA)[8] などがあります。潜在意味解析 (LSA) は文書中の単語の共起行列を使用し、潜在ディリクレ配分法 (LDA) はベイジアンネットワークを使用して文書埋め込みをモデル化します。文書埋め込みのもう一つのよく知られた Bag-of-Words ファミリーは、TF-IDF (ターム頻度 - 逆文書頻度) ベースのモデルです。これらは、単語頻度を使用して文書埋め込みを表現する統計モデルです。TF-IDF ベースのモデルは、単語レベルの重要度を表すスパースな埋め込みとすることも、単語埋め込みと組み合わせて重み付け係数として使用し、文書の密な埋め込みを生成することもできます。例えば、TF-IDF ベースの Bag-of-Words モデルである BM25[49] は、今日の検索ベンチマークにおいても依然として強力なベースラインです [9]。

しかし、Bag-of-Words パラダイムには、単語の順序と意味的内容の両方が無視されるという 2 つの大きな弱点もあります。BoW モデルは、意味や文脈を理解する上で不可欠な単語間の順序関係を捉えることができません。

Word2Vec に着想を得て、（浅い）ニューラルネットワークを使用して文書埋め込みを生成する Doc2Vec[10] が 2014 年に提案されました。Doc2Vec モデルは、図 6 に示されるように、Word2Vec のモデルに「段落」埋め込み、言い換えれば文書埋め込みを追加します。段落埋め込みは、他の単語埋め込みと連結または平均化され、段落内のランダムな単語を予測するために使用されます。学習後、既存の段落や文書については、学習済み埋め込みを後続タスクに直接使用できます。新規の段落や文書については、段落埋め込みまたは文書埋め込みを生成するために追加の推論ステップを実行する必要があります。

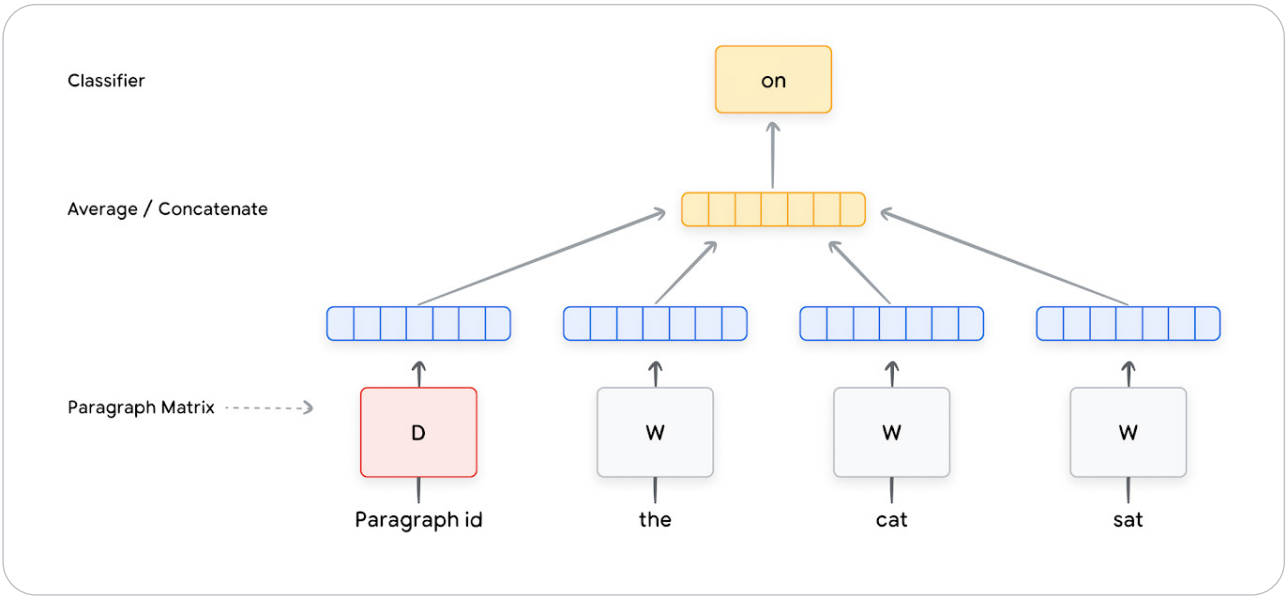

図 7：Doc2Vec CBOW モデル

以下のスニペット 4 は、カスタムコーパスで独自の Doc2Vec モデルを学習する方法を示しています。

```
from gensim.test.utils import common_texts
from gensim.models.Doc2Vec import Doc2Vec, TaggedDocument
from gensim.test.utils import get_tmpfile
#train model on a sequence of documents tagged with their IDs
documents = [TaggedDocument(doc, [i]) for i, doc in enumerate(common_texts)]
model = Doc2Vec(documents, vector_size=8, window=3, min_count=1, workers=6)
# persist model to disk, and load it to infer on new documents
model_file = get_tmpfile("Doc2Vec_v1")
model.save(model_file)
model = Doc2Vec.load(model_file)
model.infer_vector(["human", "interface"])
```

スニペット 4：プライベートコーパスにおける Doc2Vec を用いた自己教師あり学習と推論

より深い事前学習済み大規模言語モデル

ディープニューラルネットワークの発展に動機付けられ、様々な埋め込みモデルと技術が提案され、最先端のモデルは急速に進歩しています。モデルの主な変更点には以下が含まれます。

1. より複雑な学習モデル、特に双方向ディープニューラルネットワークモデルの使用。
2. ラベルなしテキストでの大規模な事前学習の利用。
3. サブワードトークナイザーの使用。
4. 様々な後続の NLP タスクのためのファインチューニングの利用。

2018 年、BERT (Bidirectional Encoder Representations from Transformers)[11] が提案され、11 の NLP タスクで画期的な結果を示しました。BERT が基づいているモデルパラダイムである Transformer は、今日に至るまで主流のモデルパラダイムとなっています。Transformer をモデルのバックボーンとして使用することに加えて、BERT の成功のもう 1 つの鍵は、大規模なラベルなしコーパスでの事前学習です。事前学習において、BERT はマスク化言語モデル (MLM) を事前学習の目的として利用しました。これは、入力の一部のトークンをランダムにマスクし、マスクされたトークン ID を予測目的として使用することによって行われました。これにより、モデルは右文脈と左文脈の両方を利用して、深層双方向 Transformer を事前学習することができます。BERT はまた、事前学習において次文予測タスクも利用します。BERT は、入力内のすべてのトークンに対して文脈化された埋め込みを出力します。通常、最初のトークン（[CLS] という名前の特殊トークン）の埋め込みが、入力全体の埋め込みとして使用されます。

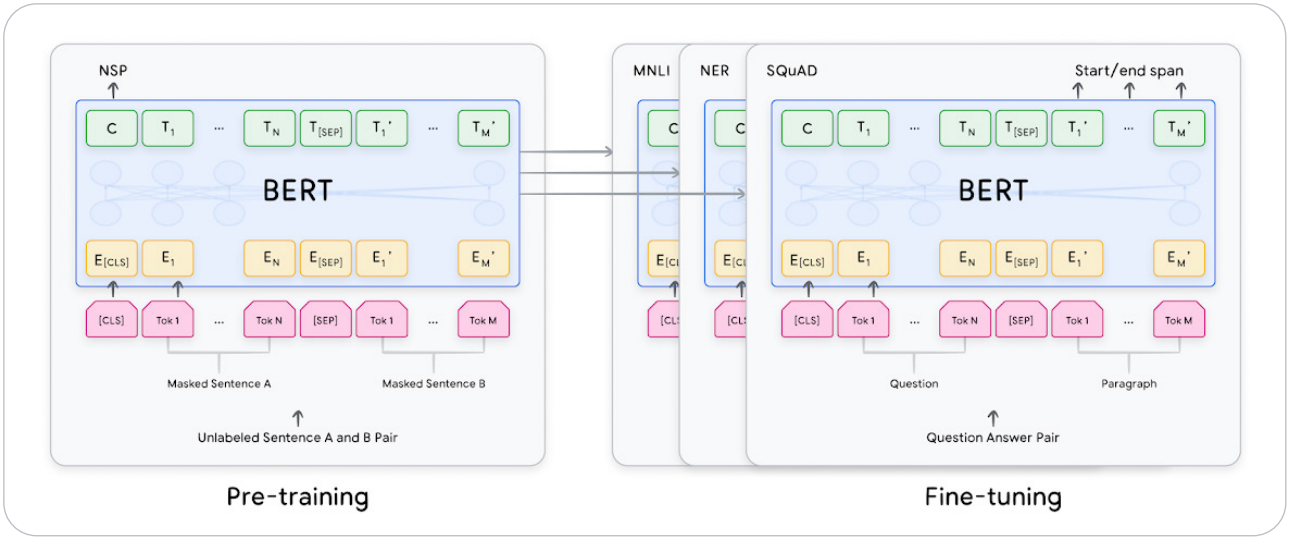

図 8：BERT アーキテクチャ

BERT は、Sentence-BERT[12]、SimCSE[13]、E5[14] など、複数の埋め込みモデルのベースモデルとなりました。一方、言語モデル、特に大規模言語モデルの進化は止まりません。T5[50] は 2019 年に最大 110 億パラメータで提案されました。PaLM[51] は 2022 年に提案され、大規模言語モデルを驚異的な 5400 億パラメータにまで押し上げました。Google の Gemini[52]、OpenAI の GPT モデル群 [53]、Meta の Llama モデル群 [54] のようなモデルも、驚異的なスピードで新しい世代へと進化しています。一般的な LLM に関する詳細については、基盤モデルに関するホワイトペーパーを参照してください。

大規模言語モデルに基づいた新しい埋め込みモデルが提案されています。例えば、GTR と Sentence-T5 は、BERT ファミリーのモデルよりも検索と文類似度（それぞれ）において優れたパフォーマンスを示しています。最近、Gemini モデルのバックボーンを利用した新しい埋め込みモデルが Vertex AI でリリースされ、すべての公開ベンチマークで優れた結果を達成しています。Matryoshka Embeddings[55,56] を使用すると、後続のユーザーは、可能な場合にストレージやインデックス作成に必要なデータを削減するために、自身のタスクに適した次元数を選択できます。

新しい埋め込みモデル開発への別のアプローチは、モデルの表現力を高めるために、単一ベクトルの代わりにマルチベクトル埋め込みを生成することです。このファミリーの埋め込みモデルには、ColBERT[15] や XTR[16] などがあります。ColPali[57] もマルチベクトルを使用するアプローチですが、その応用をテキストのみから拡張し、マルチモーダル文書のためにテキストと画像を共同で埋め込みます。

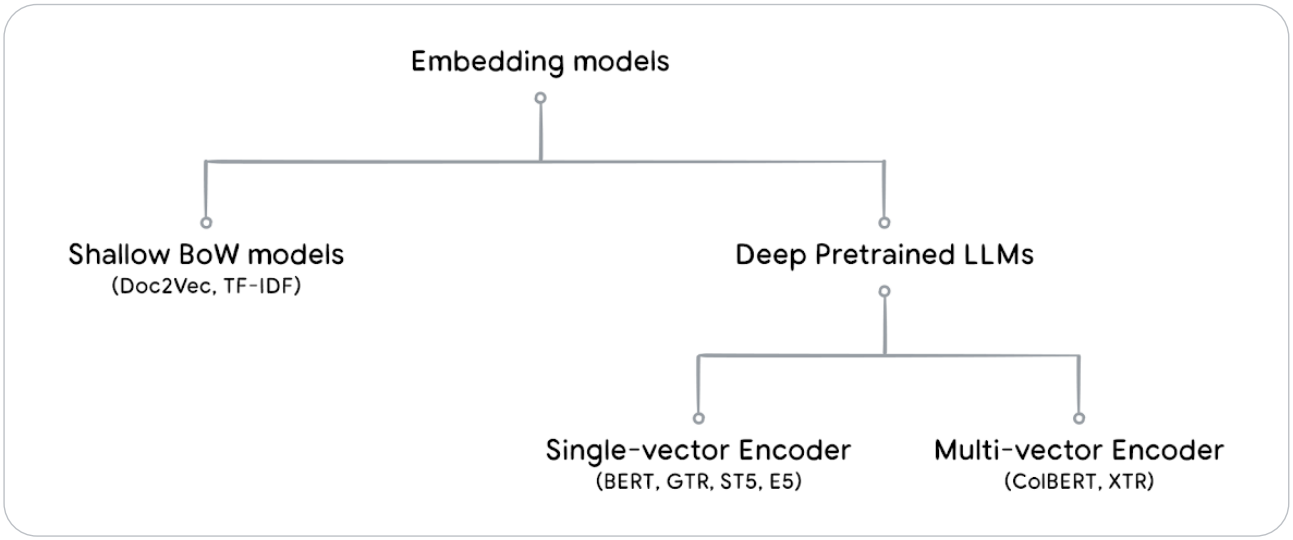

図 9：埋め込みモデルの分類図のイラスト

ディープニューラルネットワークモデルは、学習に多くのデータと計算時間を必要としますが、Bag-of-Words パラダイムを使用するモデルと比較してはるかに優れたパフォーマンスを発揮します。例えば、同じ単語でも文脈が異なれば埋め込みも異なりますが、定義上、Bag-of-Words ではそうはなりません。スニペット 4 は、TensorFlow Hub[17]（例えば Sentence T5）^A および Vertex AI^B からの事前学習済み文書埋め込みモデルを、Keras および TF データセットでモデルを学習するためにどのように使用できるかを示しています。Vertex AI 生成 AI テキスト埋め込みは、Vertex AI SDK、LangChain、および Google BigQuery（スニペット 5）と共に、埋め込みや高度なワークフローに使用できます [18]。

```
import vertexai
from vertexai.language_models import TextEmbeddingInput, TextEmbeddingModel

# Set the model name. For multilingual: use "text-multilingual-embedding-002"
MODEL_NAME = "text-embedding-004"
# Set the task_type, text and optional title as the model inputs.
# Available task_types are "RETRIEVAL_QUERY", "RETRIEVAL_DOCUMENT",
# "SEMANTIC_SIMILARITY", # "CLASSIFICATION", and "CLUSTERING"
TASK_TYPE = "RETRIEVAL_DOCUMENT"
TITLE = "Google"
TEXT = "Embed text."

# Use Vertex LLM text embeddings
embeddings_vx = TextEmbeddingModel.from_pretrained("textembedding-gecko@004")

def LLM_embed(text):
    def embed_text(text):
        text_inp = TextEmbeddingInput(task_type="CLASSIFICATION",    text=text.numpy())
        return np.array(embeddings_vx.get_embeddings([text_inp])[0].values)
    output = tf.py_function(func=embed_text, inp=[text], Tout=tf.float32)
    output.set_shape((768,))
    return output

# Embed strings using vertex LLMs
LLM_embeddings=train_data.map(lambda x,y: ((LLM_embed(x), y))
# Embed strings in the tf.dataset using one of the tf hub models
embedding = "https://tfhub.dev/google/sentence-t5/st5-base/1"
hub_layer = hub.KerasLayer(embedding, input_shape=[],dtype=tf.string, trainable=True)

# Train model
model = tf.keras.Sequential()
model.add(hub_layer) # omit this layer if using Vertex LLM embeddings
model.add(tf.keras.layers.Dense(16, activation='relu'))
model.add(tf.keras.layers.Dense(1))
model.compile(optimizer='adam',loss=tf.keras.losses.BinaryCrossentropy(from_logits=True),
  metrics=['accuracy'])
history = model.fit(train_data.shuffle(100).batch(8))
```

スニペット 4：テキスト埋め込み（Vertex, Tfhub）を作成し、Keras テキスト分類モデルに統合する

```
SELECT * FROM ML.GENERATE_TEXT_EMBEDDING(
MODEL my_project.my_company.llm_embedding_model,
(
SELECT review as content
FROM bigquery-public-data.imdb.reviews));
```

スニペット 5：テーブル内の選択された列に対して BigQuery で LLM ベースのテキスト埋め込みを作成する

画像およびマルチモーダル埋め込み

テキストと同様に、画像埋め込みとマルチモーダル埋め込みの両方を作成することも可能です。ユニモーダル画像埋め込みは、例えば大規模な画像分類タスク（例：ImageNet）で CNN または Vision Transformer モデルを学習させ、その最後から 2 番目の層を画像埋め込みとして使用するなど、多くの方法で導出できます。この層は、学習タスクにとって重要な識別的な特徴マップを学習しています。これには、当面のタスクに対して識別的であり、他のタスクにも拡張可能な特徴マップのセットが含まれています。

マルチモーダル埋め込み [19] を得るには、個々のユニモーダルなテキスト埋め込みと画像埋め込みを取得し、別の学習プロセスを通じて学習されたそれらの意味的関係の共同埋め込みを作成します。これにより、同じ潜在空間内に固定長の意味表現が得られます。スニペット 6 は、画像とテキストに対する画像埋め込みおよびマルチモーダル埋め込みを計算し、（テキスト埋め込みの例とよく似て）Keras モデルで直接使用することができます。ColPali[57] のようなマルチモーダル埋め込みアプローチは、画像モデルを使用して、複雑な OCR やレイアウト前処理なしに、マルチモーダル文書に対するテキストクエリからの検索を可能にします。モデルは、インデックス作成のためにテキストのみの形式に変換する必要なく、ウェブブラウザや PDF ビューアでユーザーに表示されるのと同じように画像を検索します。

```
import base64
import tensorflow as tf
from google.cloud import aiplatform
from google.protobuf import struct_pb2

#fine-tunable layer for image embeddings which can be used for downstream keras modelimage_
embed=hub.KerasLayer("https://tfhub.dev/google/imagenet/efficientnet_v2_imagenet21k_ft1k_s/feature_
vector/2",trainable=False)

class EmbeddingPredictionClient:
  """Wrapper around Prediction Service Client."""
  def __init__(self, project : str,
    location : str = "us-central1",
    api_regional_endpoint: str = "us-central1-aiplatform.googleapis.com"):
    client_options = {"api_endpoint": api_regional_endpoint}
    self.client = aiplatform.gapic.PredictionServiceClient(client_options=client_options)
    self.location = location
    self.project = project

  def get_embedding(self, text : str = None, gs_image_path : str = None):
   #load the image from a bucket in google cloud storage
   with tf.io.gfile.GFile(gs_image_path, "rb") as f:
     image_bytes = f.read()
   if not text and not image_bytes:
    raise ValueError('At least one of text or image_bytes must be specified.')
   #Initialize a protobuf data struct with the text and image inputs
   instance = struct_pb2.Struct()
    if text:
      instance.fields['text'].string_value = text
      if image_bytes:
      encoded_content = base64.b64encode(image_bytes).decode("utf-8")
      image_struct = instance.fields['image'].struct_value
      image_struct.fields['bytesBase64Encoded'].string_value = .string_value = encoded_content

     #Make predictions using the multimodal embedding model
     instances = [instance]
     endpoint = (f"projects/{self.project}/locations/{self.location}"
         "/publishers/google/models/multimodalembedding@001")
     response = self.client.predict(endpoint=endpoint, instances=instances)

     text_embedding = None
     if text:
      text_emb_value = response.predictions[0]['textEmbedding']
      text_embedding = [v for v in text_emb_value]

     image_embedding = None
     if image_bytes:
      image_emb_value = response.predictions[0]['textEmbedding']
      image_embedding = [v for v in image_emb_value]
```

次のページに続きます

```
 return EmbeddingResponse (text_embedding=text_embedding, image_embedding=image_embedding)
#compute multimodal embeddings for text and images
client.get_embedding(text="sample_test", gs_image_path="gs://bucket_name../image_filename..")
```

スニペット 6：Vertex API を使用したマルチモーダル埋め込みの作成 (グラフ埋め込み)

構造化データ埋め込み

構造化データとは、個々のフィールドが既知の型と定義を持つデータベース内のテーブルのように、定義されたスキーマを持つデータを指します。通常、事前学習済みの埋め込みモデルが利用可能な非構造化テキストデータや画像データとは異なり、構造化データの場合は、特定のアプリケーションに固有となるため、その埋め込みモデルを作成する必要があります。

一般的な構造化データ

一般的な構造化データテーブルが与えられた場合、各行に対して埋め込みを作成することができます。これは、PCA モデルのような次元削減カテゴリの ML モデルによって行うことができます。

これらの埋め込みの 1 つのユースケースは異常検知です。例えば、異常な発生を特定するラベル付きセンサー情報の大規模データセットを使用して、異常検知のための埋め込みを作成することができます [20]。もう 1 つのユースケースは、これらの埋め込みを分類などの後続の ML タスクに入力することです。

元の高次元データを使用する場合と比較して、埋め込みを使用して教師ありモデルを学習する方が、より少ないデータで済みます。これは、学習データが十分ではない場合に特に重要です。

ユーザー / アイテム構造化データ

入力は、上記のような一般的な構造化データテーブルではありません。代わりに、入力にはユーザーデータ、アイテム / 商品データに加えて、評価スコアのようなユーザーとアイテム / 商品間のインタラクションを記述するデータが含まれます。

このカテゴリは、2 つのデータセット（ユーザーデータセットとアイテム / 商品 / その他データセット）を同一の埋め込み空間に射影することから、推薦を目的としています。

推薦システムでは、商品や記事などの様々なエンティティと相関する構造化データから埋め込みを作成することができます。この場合も、独自の埋め込みモデルを作成する必要があります。

画像やテキスト記述が見つかる場合には、これが非構造化埋め込み手法と組み合わされることもあります。

グラフ埋め込み

グラフ埋め込みは、特定のオブジェクトに関する情報だけでなく、その近傍（すなわち、それらのグラフ表現）も表現することを可能にする、もう 1 つの埋め込み技術です。

各個人がノードであり、人々の間のつながりがエッジとして定義されるソーシャルネットワークを例に考えてみましょう。グラフ埋め込みを使用すると、各ノードを埋め込みとしてモデル化できます。この埋め込みは、その人物自身に関する意味情報だけでなく、その関係性や関連性も捉えるため、埋め込み表現がより豊かになります。

例えば、2 つのノードがエッジで接続されている場合、それらのノードのベクトルは類似したものになります。そうすることで、その人が誰に最も似ているかを予測し、新しいつながりを推薦することができるようになるかもしれません。

グラフ埋め込みはまた、ノード分類、グラフ分類、リンク予測、クラスタリング、検索、推薦システムなど、様々なタスクに使用することができます。

グラフ埋め込みのための一般的なアルゴリズム [21,22] には、DeepWalk、Node2vec、LINE、および GraphSAGE[23] などがあります。

埋め込みの学習

現在の埋め込みモデルは通常、デュアルエンコーダー（Two-Tower）アーキテクチャを使用します。例えば、質問応答で使用されるテキスト埋め込みモデルの場合、一方のタワーがクエリをエンコードするために使用され、もう一方のタワーが文書をエンコードするために使用されます。画像・テキスト埋め込みモデルの場合、一方のタワーが画像をエンコードするために使用され、もう一方のタワーがテキストをエンコードするために使用されます。モデルは、2 つのタワー間でモデルコンポーネントがどのように共有されるかに応じて、様々なサブアーキテクチャを持つことができます。以下の図は、デュアルエンコーダーのいくつかのアーキテクチャを示しています [24]。

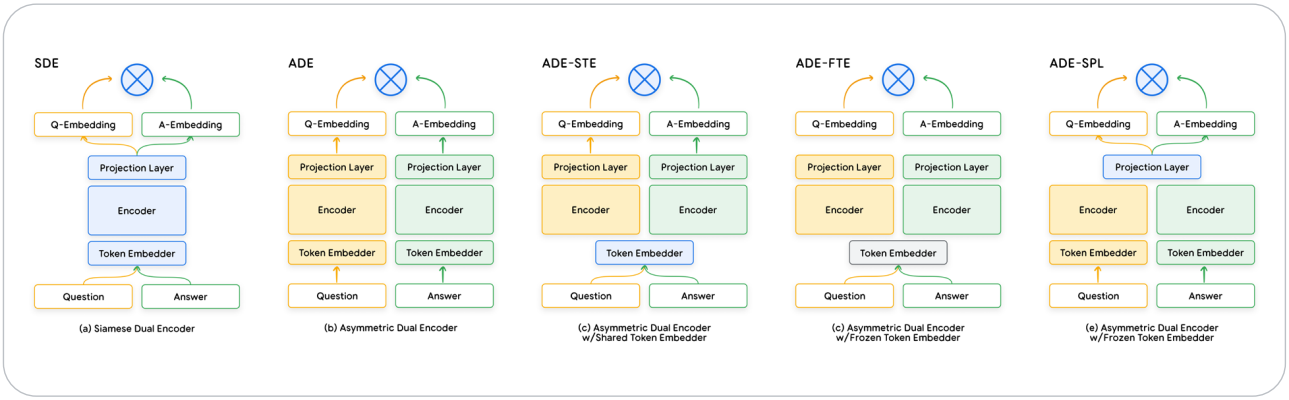

図 10：デュアルエンコーダーのいくつかのアーキテクチャ

埋め込みモデルの学習で使用される損失は通常、対照損失（contrastive loss）の一種であり、これは < 入力、正例ターゲット、[任意] 負例ターゲット > のタプルを入力として取ります。対照損失を用いた学習は、正例をより近くに、負例をより遠くに引き離します。

基盤モデルの学習と同様に、埋め込みモデルをスクラッチから学習するには、通常、事前学習（教師なし学習）とファインチューニング（教師あり学習）の 2 つの段階が含まれます。今日では、埋め込みモデルは通常、BERT、T5、GPT、Gemini、CoCa などの基盤モデルから直接初期化されます。これらのベースモデルを使用することで、基盤モデルの大規模事前学習から得られた膨大な知識を活用できます。埋め込みモデルのファインチューニングは、1 つ以上のフェーズを持つことができます。ファインチューニング用データセットは、人手によるラベリング、合成データセット生成、モデル蒸留、ハードネガティブマイニングなど、様々な方法で作成できます。

分類や固有表現認識のような後続タスクに埋め込みを使用するために、埋め込みモデルの上に（例えば、softmax 分類層のような）追加の層を加えることができます。埋め込みモデルは、（特に学習データセットが小さい場合には）固定する（フリーズする）か、スクラッチから学習するか、あるいは後続タスクと共にファインチューニングすることができます。

Vertex AI は、Vertex AI テキスト埋め込みモデルをカスタマイズする機能を提供しています [25]。ユーザーはモデルを直接ファインチューニングすることも選択できます。例としては、TensorFlow Model Garden[26] を使用した BERT モデルのファインチューニングがあります。また、TensorFlow Hub から埋め込みモデルを直接ロードし、そのモデル上でファインチューニングすることも可能です。スニペット 7 は、TensorFlow Hub のモデルに基づいて分類器を構築する方法の例を示しています。

```
# Can switch the embedding to different embeddings from different modalities on #
tfhub. Here we use the BERT model as an example.
tfhub_link = "https://tfhub.dev/tensorflow/bert_en_uncased_L-12_H-768_A-12/4"

class Classifier(tf.keras.Model):
    def __init__(self, num_classes):
      super(Classifier, self).__init__(name="prediction")
        self.encoder = hub.KerasLayer(tfhub_link, trainable=True)
      self.dropout = tf.keras.layers.Dropout(0.1)
      self.dense = tf.keras.layers.Dense(num_classes)

    def call(self, preprocessed_text):
      encoder_outputs = self.encoder(preprocessed_text)
      pooled_output = encoder_outputs["pooled_output"]
   x = self.dropout(pooled_output)
   x = self.dense(x)
   return x
```

スニペット 7：学習可能な TensorFlow Hub レイヤーを使用した Keras モデルの作成

ここまでで、様々な種類の埋め込み、様々なデータモダリティに対してそれらを学習させるための技術とベストプラクティス、そしてそのいくつかの応用例を見てきました。次のセクションでは、作成された埋め込みを、本番ワークロードのために高速かつスケーラブルな方法で永続化し検索する方法について説明します。

ベクトル検索

全文キーワード検索は、長年にわたり現代のITシステムの中核を担ってきました。全文検索エンジンやデータベース（リレーショナルおよび非リレーショナル）は、多くの場合、明示的なキーワード照合に依存しています。例えば、「カプチーノ」と検索すると、検索エンジンやデータベースは、タグやテキスト記述にその検索語句と完全に一致する言及があるすべての文書を返します。しかし、キーワードにスペルミスがあったり、異なる言葉で記述されていたりすると、従来のキーワード検索では不正確な結果が返されたり、結果が全く得られなかったりします。スペルミスやその他の誤植に対して寛容な従来のアプローチも存在します。しかし、それでもなお、クエリの根底にある意味内容に最も近い結果を見つけることはできません。ここでベクトル検索が非常に強力な理由があります。それは、文書のベクトル、すなわち埋め込まれた意味表現を使用するからです。ベクトル検索はあらゆる種類の埋め込みに対して機能するため、テキストに加えて画像、動画、その他のデータ型での検索も可能にします。

ベクトル検索を使用すると、検索語句そのものによる検索を超えて、様々なデータモダリティにわたって意味を検索することができます。これにより、言葉遣いが異なっていても関連性の高い結果を見つけることができます。様々なアイテムの埋め込みを計算できる関数を用意した後、対象アイテムの埋め込みを計算し、この埋め込みをデータベースに保存します。次に、入力されたクエリをアイテムと同じベクトル空間に埋め込みます。そして、クエリに最も一致するアイテムを見つけ出す必要があります。このプロセスは、検索可能なベクトルの全コレクションの中から最も「類似した」一致を見つけることに似ています。ベクトル間の類似性は、ユークリッド距離、コサイン類似度、内積などの評価指標を使用して計算できます。

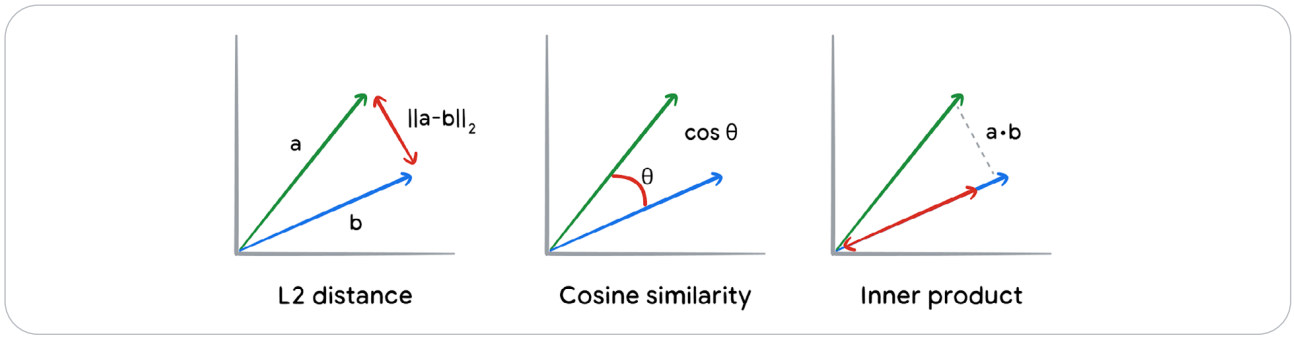

図 11：様々な評価指標がベクトル類似度をどのように計算するかの可視化

ユークリッド距離（すなわちL2距離）は、ベクトル空間内の2点間の距離の幾何学的尺度です。これは低次元の場合にうまく機能します。コサイン類似度は、2つのベクトル間の角度の尺度です。そして内積（ドット積）は、あるベクトルを別のベクトルに射影したものです。これらは、ベクトルノルムが1の場合に等価になります。これは、より高次元のデータに対してより効果的に機能するようです。ベクトルデータベースは、大規模なベクトル検索の複雑さを保存し、管理し、運用可能にするのに役立つと同時に、一般的なデータベースのニーズにも対応します。

主要なベクトル検索アルゴリズム

最も類似した一致を見つける最も簡単な方法は、クエリベクトルを各文書ベクトルと比較し、最も類似性の高いものを返すという従来の線形検索を実行することです。しかし、このアプローチの実行時間は、検索対象の文書またはアイテムの量に応じて線形に (O(N)) 増加するため、数百万以上の文書を含むほとんどのユースケースでは許容できないほど遅くなります。

その目的のためには、近似最近傍 (ANN) 検索を使用する方がより実用的です。ANN は、データセット内の与えられた点に対して最も近い点群を、わずかな誤差の範囲で発見する技術ですが、検索空間が O(logN) に大幅に削減されるため、必要となる計算量ははるかに少なくなります。

規模、インデックス作成時間、パフォーマンス、単純さなどに関して様々なトレードオフを持つ多くのアプローチが存在します [26]。これらのアプローチは、量子化、ハッシング、クラスタリング、ツリー構造などの技術の 1 つ以上の実装を使用しており、そのうち最も一般的なもののいくつかを以下で説明します。

局所性鋭敏型ハッシュ（LSH）とツリー

局所性鋭敏型ハッシュ (LSH)[27] は、大規模データセット内で類似アイテムを発見するための技術です。これは、類似したアイテムを高い確率で同じハッシュバケットにマッピングする 1 つ以上のハッシュ関数を作成することによって行われます。これにより、特定のアイテムに対して、同じハッシュバケット（または隣接するバケット）内の候補アイテムのみを調べ、それらの候補ペア間で線形検索を行うだけで、すべての類似アイテムを迅速に見つけることができます。これにより、特定の半径内での検索が大幅に高速化されます。

ハッシュ関数 / テーブルとバケットの数は、検索の再現率と速度のトレードオフ、さらには偽陽性と真陽性のトレードオフを決定します。ハッシュ関数が多すぎると類似アイテムが異なるバケットに入ってしまう可能性があり、少なすぎると多くのアイテムが誤って同じバケットにハッシュされてしまい線形検索の回数が増加する可能性があります。

LSH について直感的に理解するもう 1 つの方法は、住居を郵便番号や近隣地域名でグループ化することを考えることです。そして、誰かが引っ越す場所に基づいて、その近隣地域の住居のみを調べて最も近い一致を見つけます。

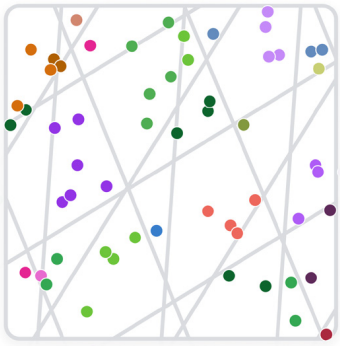

図 12：LSH がランダムな超平面を使用してベクトル空間をどのように分割するかの可視化

ツリーベースのアルゴリズムも同様に機能します。例えば、Kd-tree アプローチは、最初の次元の値の中央値を計算し、次に 2 番目の次元の中央値を計算する、というようにして決定境界を作成することで機能します。このアプローチは決定木によく似ています。当然ながら、検索可能なベクトルが高次元である場合、これは効果的でない可能性があります。その場合、Ball-tree アルゴリズムの方が適しています。機能的には似ていますが、次元ごとの中央値で分割する代わりに、中心からのデータ点の半径方向の距離に基づいてバケットを作成する点が異なります。以下に、これらの 3 つのアプローチの実装例を示します。

```
from sklearn.neighbors import NearestNeighbors
from vertexai.language_models import TextEmbeddingModel
from lshashing import LSHRandom
import numpy as np

model = TextEmbeddingModel.from_pretrained("textembedding-gecko@004")
test_items= [
  "The earth is spherical.",
  "The earth is a planet.",
  "I like to eat at a restaurant."]
query = "the shape of earth"
embedded_test_items = np.array([embedding.values for embedding in model.get_embeddings(test_items)])
embedded_query = np.array(model.get_embeddings([query])[0].values)

#Naive brute force search
n_neighbors=2
nbrs = NearestNeighbors(n_neighbors=n_neighbors, algorithm='brute').fit(embedded_test_items)
naive_distances, naive_indices = nbrs.kneighbors(np.expand_dims(embedded_query, axis = 0))

#algorithm- ball_tree due to high dimensional vectors or kd_tree otherwise
nbrs = NearestNeighbors(n_neighbors=n_neighbors, algorithm='ball_tree').fit(embedded_test_items)
distances, indices = nbrs.kneighbors(np.expand_dims(embedded_query, axis = 0))

#LSH
lsh_random_parallel = LSHRandom(embedded_test_items, 4, parallel = True)
lsh_random_parallel.knn_search(embedded_test_items, embedded_query, n_neighbors, 3, parallel = True)

#output for all 3 indices = [0, 1] , distances [0.66840428, 0.71048843] for the first 2 neighbours
#ANN retrieved the same ranking of items as brute force in a much scalable manner
```

スニペット 8：scikit-learn[28] および lshashing[29] を使用した、LSH、KD 木 /Ball-tree、および線形検索による ANN

ハッシングとツリーベースのアプローチは、検索アルゴリズムの再現率とレイテンシの間の最適なトレードオフを得るために、組み合わせて拡張することもできます。HNSW（階層的ナビゲーブルスモールワールド）を用いた FAISS や ScaNN[32,33] が良い例です。

階層的ナビゲーブルスモールワールド

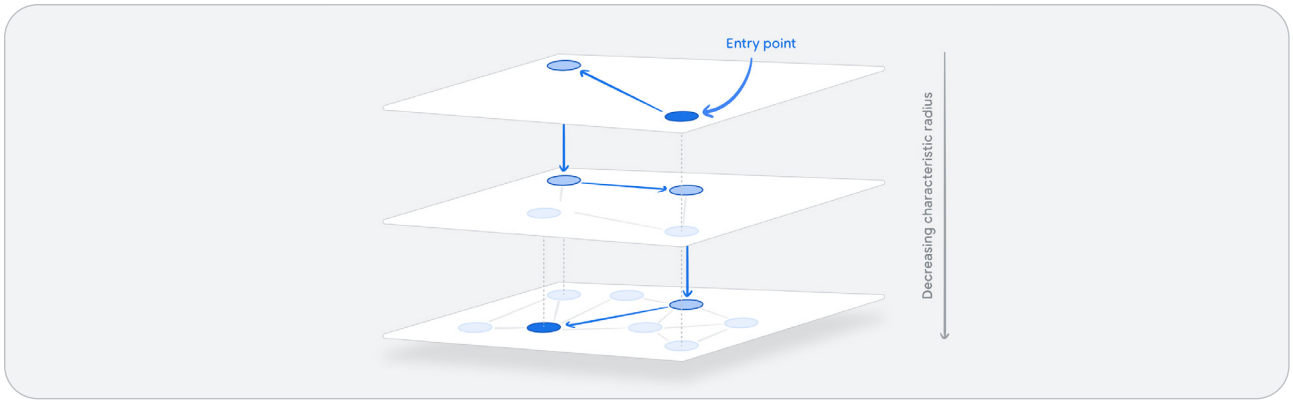

図 13：HNSW が ANN を実行するためにどのように「ズームイン」するかを示す図

FAISS (Facebook AI Similarity Search) の実装の 1 つは、階層的ナビゲーブルスモールワールド (HNSW)[30] の概念を活用して、高い精度で準線形 (O(logN)) の実行時間でベクトル類似性検索を実行します。

HNSW は、グラフリンクが異なる層に分散された階層構造を持つ近接グラフです。最上位層は最も長いリンクを持ち、最下位層は最も短いリンクを持ちます。図 13 に示すように、検索は最上位層から開始され、アルゴリズムはグラフを貪欲に探索してクエリに最も意味的に類似した頂点を見つけます。その層での局所最適解が見つかると、次に下の層の最も近い頂点に対応するグラフに切り替えます。このプロセスは、最下位層での局所最適解が見つかるまで反復的に継続され、アルゴリズムは K 最近傍を返すために探索したすべての頂点を追跡します。

このアルゴリズムは、速度とメモリ効率を向上させるために、オプションで量子化とベクトルインデックス作成によって補強することができます。

```
# Create an endpoint
my_index_endpoint = aiplatform.MatchingEngineIndexEndpoint.create(
    display_name=f"{DISPLAY_NAME}-endpoint", public_endpoint_enabled=True
)

# NOTE : This operation can take upto 20 minutes
my_index_endpoint = my_index_endpoint.deploy_index(
    index=my_index, deployed_index_id=DEPLOYED_INDEX_ID
)

# retrieve the id of the most recently deployed index or manually look up the index
deployed above
index_id=my_index_endpoint.deployed_indexes[-1].index.split("/")[-1]
endpoint_id= my_index_endpoint.name

# TODO : replace 1234567890123456789 with your acutial index ID
my_index = aiplatform.MatchingEngineIndex(index_id)

# TODO : replace 1234567890123456789 with your acutial endpoint ID
# Be aware that the Index ID differs from the endpoint ID
my_index_endpoint = aiplatform.MatchingEngineIndexEndpoint(endpoint_id)

# Input texts
texts= [
   "The earth is spherical.",
   "The earth is a planet.",
   "I like to eat at a restaurant.",
]

# Create a Vector Store
vector_store = VectorSearchVectorStore.from_components(
    project_id=PROJECT_ID,
    region=REGION,
    gcs_bucket_name=BUCKET,
    index_id=my_index.name,
    endpoint_id=my_index_endpoint.name,
    embedding=embedding_model,
    stream_update=True,
)

# Add vectors and mapped text chunks to your vectore store
vector_store.add_texts(texts=texts)

# Initialize the vectore_store as retriever
retriever = vector_store.as_retriever()
```

次のページに続きます

```
retriever=vector_store.as_retriever(search_kwargs={'k':1 })

#create custom prompt for your use case
prompt_template="""You are David, an AI knowledge bot.
Answer the questions using the facts provided. Use the provided pieces of context to answer
the users question.
If you don't know the answer, just say that "I don't know", don't try to make up an answer.
{summaries}"""

messages = [
    SystemMessagePromptTemplate.from_template(prompt_template),
    HumanMessagePromptTemplate.from_template("{question}")
]
prompt = ChatPromptTemplate.from_messages(messages)

chain_type_kwargs = {"question": prompt}

#initialize your llm model
llm = VertexAI(model_name="gemini-pro")

#build your chain for RAG+C
chain= RetrievalQA.from_chain_type(llm=llm, chain_type="stuff",
retriever=retriever, return_source_documents=True)

#print your results with Markup language
def print_result(result):
  output_text = f"""### Question:
  {query}
  ### Answer:
  {result['result']}
  ### Source:
  {' '.join(list(set([doc.page_content for doc in result['source_documents']])))}
  """
  return(output_text)

chain= "What shape is the planet where humans live?"
result = chain(query)
display(Markdown(print_result(result)))
```

スニペット 9：Vertex AI ベクトル検索のための ANN インデックスの構築 / デプロイと、LLM プロンプトを用いた RAG による根拠のある結果 / ソースの生成

```
import faiss
M=32 #creating high degree graph:higher recall for larger index & searching time
d=768 # dimensions of the vectors/embeddings
index = faiss.IndexHNSWFlat(d, M)
index.add(embedded_test_items) #build the index using the embeddings in Snippet 9
#execute the ANN search
index.search(np.expand_dims(embedded_query, axis=0), k=2)
```

スニペット 10：HNSW を使用した FAISS ライブラリによる ANN 検索のインデックス作成と実行

ScaNN（スケーラブル近似最近傍）

Google は、多くの自社製品やサービスで使用されているスケーラブル近似最近傍 (ScaNN)[31,32] アプローチを開発しました。これには、Vertex AI ベクトル検索や、AlloyDB、Cloud Spanner、Cloud SQL MySQL を含む Google Cloud データベースを通じて、すべての Google Cloud の顧客が外部から利用可能になっていることも含まれます。以下は、ScaNN が効率的なベクトル検索を実行するために一連のステップをどのように使用するかを示したものであり、各ステップにはそれぞれ独自のパラメータサブセットがあります。

最初のステップは、学習中のオプションのパーティション分割ステップです。ここでは、利用可能な複数のアルゴリズムの 1 つを使用してベクトルストアを論理パーティション / クラスタに分割し、意味的に関連するものをグループ化します。パーティション分割ステップは、小規模なデータセットではオプションです。しかし、10 万を超える埋め込みベクトルを持つ大規模なデータセットの場合、検索空間の枝刈りによって検索空間を桁違いに削減し、それによってクエリを大幅に高速化するため、パーティション分割ステップは不可欠です。空間枝刈りは、パーティション数と検索対象のパーティション数を通じて設定されます。これらの数が多いほど再現率は向上しますが、パーティション作成時間は長くなります。良いヒューリスティックとしては、パーティション数をベクトル数の平方根に設定することです。

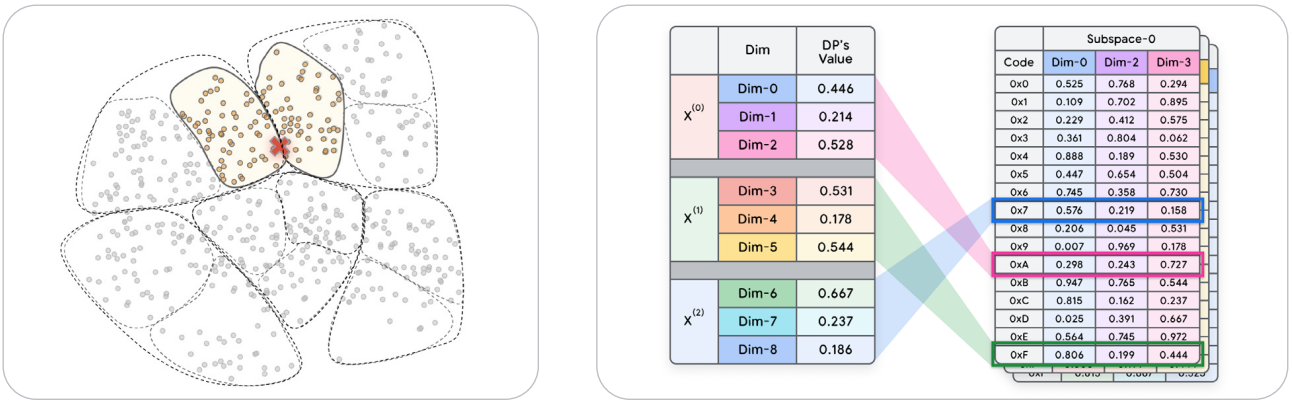

図 14：検索空間のパーティション分割と枝刈り（左）および近似スコアリング（右）

クエリ実行時、ScaNN はユーザー指定の距離尺度を使用して指定された数の上位パーティション（ユーザーが指定した値）を選択し、次にスコアリングステップを実行します。このステップでは、ScaNN はクエリを上位パーティション内のすべての点と比較し、上位 K’個を選択します。この距離計算は、正確な距離または近似距離として設定できます。近似距離計算は、標準的な積量子化または異方性量子化技術のいずれかを活用します。後者は ScaNN によって採用されている特定の手法であり、より優れた速度と精度のトレードオフを提供します。

最後に、最終ステップとして、ユーザーはオプションで、指定した上位 K 個の結果をより正確に再スコアリングすることを選択できます。これにより、図 14 からも推測できるように、ScaNN が知られている業界トップクラスの速度 / 精度のトレードオフが実現されます。スニペット 11 にコード例を示します。

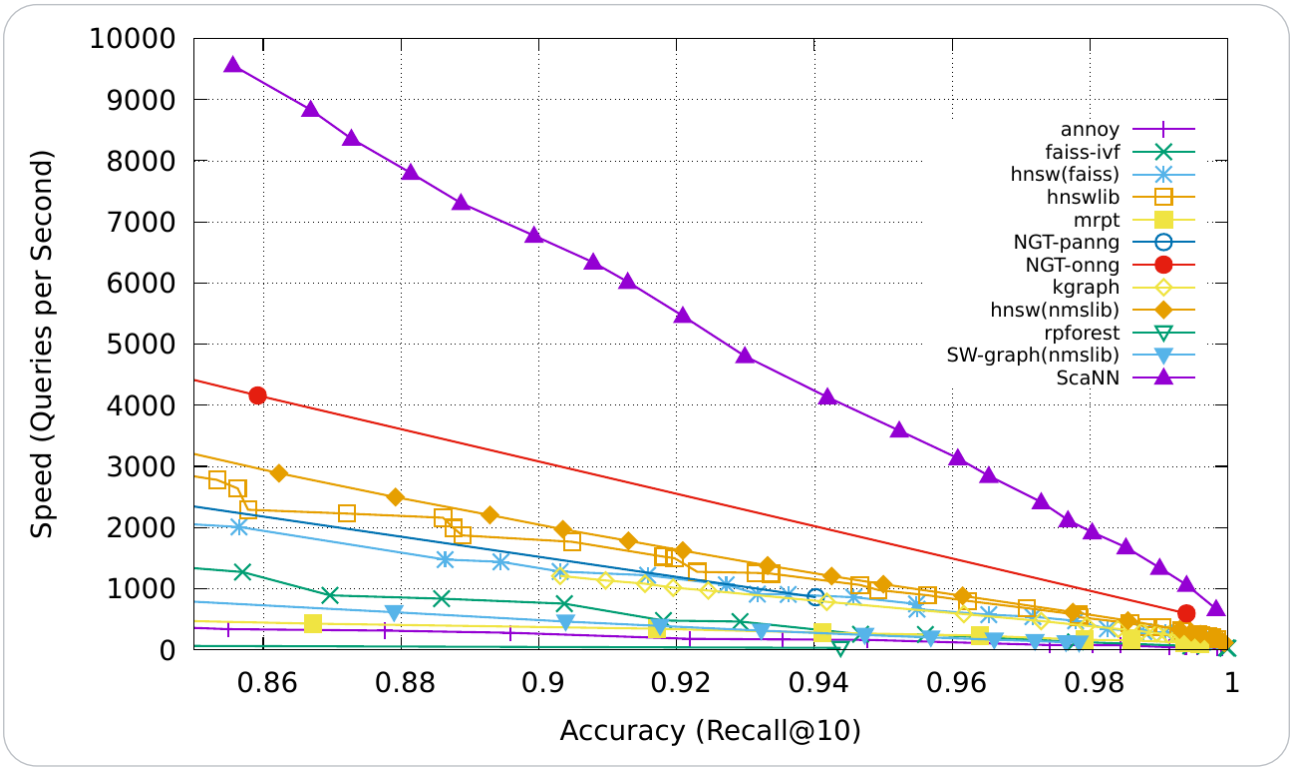

図 15：様々な SOTA (最先端) ANN 検索アルゴリズムにおける精度 / 速度のトレードオフ [58]

```
import tensorflow as tf
import tensorflow_recommenders as tfrs
from vertexai.language_models import TextEmbeddingModel, TextEmbeddingInput

# Embed documents & query(from snip 9.) and convert them to tensors and tf.datasets
embedded_query = tf.constant((LM_embed(query, "RETRIEVAL_QUERY")))
embedded_docs = [LM_embed(doc, "RETIREVAL_DOCUMENT") for doc in searchable_docs]
embedded_docs = tf.data.Dataset.from_tensor_slices(embedded_docs).enumerate().batch(1)

# Build index from tensorflow dataset and execute ANN search based on dot product metric
scann = tfrs.layers.factorized_top_k.ScaNN(
  distance_measure= 'dot_product',
  num_leaves = 4, #increase for higher number of partitions / latency for increased recall
  num_leaves_to_search= 2) # increase for higher recall but increased latency
scann = scann.index_from_dataset(embedded_docs)
scann(embedded_query, k=2)
```

スニペット 11：TensorFlow Recommenders[33] を使用した ScaNN アルゴリズムによる ANN 検索の実行

本稿では、現在の ANN 検索アルゴリズムと従来の ANN 検索アルゴリズムの両方（ScaNN、FAISS、LSH、KD 木、Ball-tree）を見ていき、それらが提供する優れた速度 / 精度のトレードオフを検証しました。しかし、これらのアルゴリズムを使用するには、スケーラブルで安全、かつ本番環境に対応した方法でデプロイする必要があります。そのためには、ベクトルデータベースが必要です。

ベクトルデータベース

ベクトル埋め込みはデータの意味的内容を具体化し、ベクトル検索アルゴリズムはそれらを効率的にクエリするための手段を提供します。歴史的に、従来のデータベースは意味的内容と効率的なクエリ発行を組み合わせる手段を欠いていました。これがベクトルデータベースの台頭につながりました。ベクトルデータベースは、本番シナリオのためにこれらの埋め込みを管理するべくゼロから構築されています。近年の生成 AI の人気により、従来の検索機能に加えてベクトル検索機能（「ハイブリッド検索」）のサポートを組み込む従来のデータベースが増えつつあります。ハイブリッド検索機能を備えた単純なベクトルデータベースのワークフローを見てみましょう。

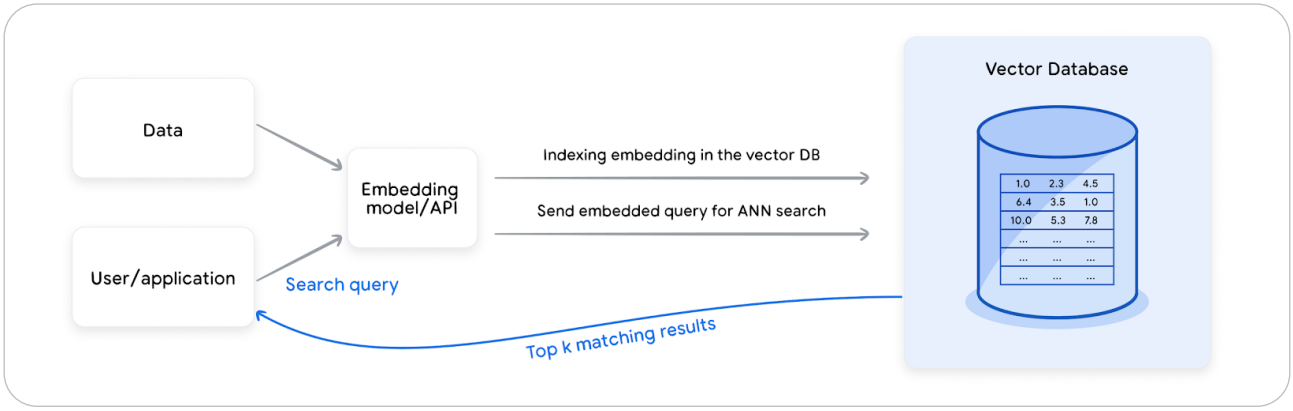

図 16：ベクトルデータベースへのデータ投入とクエリ発行

各ベクトルデータベースは実装が異なりますが、一般的なフローは図 16 に示されています。

1. 適切な学習済み埋め込みモデルを使用して、関連するデータ点を固定次元のベクトルとして埋め込みます。
2. 次に、ベクトルに適切なメタデータと補足情報（タグなど）が付加され、効率的な検索のために指定されたアルゴリズムを使用してインデックス化されます。
3. 入力クエリが適切なモデルで埋め込まれ、最も意味的に類似したアイテムとその関連する元のコンテンツ / メタデータの検索に使用されます。

一部のデータベースでは、クエリの速度とパフォーマンスをさらに向上させるために、キャッシングや（タグに基づく）事前フィルタリング、および（より精度の高い別のモデルを使用したリランキングなどの）事後フィルタリング機能が提供される場合があります。

今日ではかなりの数のベクトルデータベースが利用可能であり、それぞれが異なるビジネスニーズや考慮事項に合わせて調整されています。商用管理されているベクトルデータベースの良い例としては、Google Cloud の Vertex AI ベクトル検索 [34]、Google Cloud の AlloyDB および Cloud SQL Postgres 向け Elasticsearch[35]、Pinecone[36] などが挙げられます。Vertex AI ベクトル検索は Google によって構築されたベクトルデータベースであり、高速なベクトル検索のために ScaNN アルゴリズムを使用しつつ、Google Cloud のすべてのセキュリティおよびアクセス保証を維持しています。AlloyDB および Cloud SQL Postgres は、OSS の pgvector[37] 拡張機能を通じてベクトル検索をサポートしており、これにより SQL クエリで ANN 検索を従来の述語や ANN 検索インデックスのための通常のトランザクションセマンティクスと組み合わせることができます。AlloyDB にはまた、ScaNN のネイティブ実装であり pgvector 互換の ScaNN インデックス拡張機能もあります。同様に、他の多くの従来のデータベースも、ベクトル検索を可能にするためのプラグインを追加し始めています。Pinecone[37] と Weaviate[39] は、従来の検索を使用してデータをフィルタリングする機能に加えて、高速なベクトル検索のために HNSW を活用しています。これらのオープンソースの同等製品の中では、Weaviate[38] や ChromaDB[39] がデプロイ時に包括的な機能を提供し、プロトタイピングフェーズではメモリ上でテストすることも可能です。

運用上の考慮事項

ベクトルデータベースは、埋め込みを大規模に保存しクエリする際に生じる技術的課題の大部分を管理する上で不可欠です。これらの課題のいくつかはベクトルストアの性質に固有のものですが、他のものは従来のデータベースの課題と重複しています。これらには、水平および垂直スケーラビリティ、可用性、データ一貫性、リアルタイム更新、バックアップ、アクセス制御、コンプライアンスなどが含まれますが、これらに限りません。しかし、埋め込みやベクトルストアを使用する際には、考慮に入れるべきさらに多くの課題や検討事項があります。

第一に、埋め込みは、従来のコンテンツとは異なり、時間とともに変化する可能性があります。これは、同じテキスト、画像、動画、またはその他のコンテンツであっても、後続アプリケーションのパフォーマンスを最適化するために、異なる埋め込みモデルを使用して埋め込むことが可能であり、またそうすべきであることを意味します。これは、様々なドリフトや目的の変更に対応するためにモデルが再学習された後の、教師ありモデルの埋め込みにおいて特に当てはまります。同様に、教師なしモデルがより新しいモデルに更新された場合にも同じことが言えます。しかし、埋め込みを頻繁に更新すること、特に大量のデータで学習されたものを更新することは、法外に高価になる可能性があります。したがって、バランスを取る必要があります。このため、予算を考慮に入れつつ、ベクトルデータベースから埋め込みを保存、管理、そして場合によってはパージするための、明確に定義された自動化プロセスが必要となります。

第二に、埋め込みは意味情報を表現するのに優れていますが、リテラル情報や構文情報の表現においては最適ではない場合があります。これは、ドメイン固有の単語や ID において特に当てはまります。これらの値は、埋め込みモデルが学習されたデータの中に潜在的に欠落しているか、十分に表現されていない可能性があります。例えば、ユーザーが大量のテキストと共に特定の番号の ID を含むクエリを入力した場合、モデルはテキストの意味にはよく一致するが、この文脈で最も重要な要素である ID には一致しない、意味的に類似した近傍を見つけてしまうかもしれません。この課題は、セマンティック検索モジュールに渡す前に、フルテキスト検索を組み合わせて検索空間を事前フィルタリングまたは事後フィルタリングすることによって克服できます。

考慮すべきもう 1 つの重要な点は、セマンティッククエリが発生するワークロードの性質に応じて、異なるベクトルデータベースに依存する価値があるかもしれないということです。例えば、頻繁な読み取り / 書き込み操作を必要とする OLTP (オンライントランザクション処理) ワークロードの場合、AlloyDB、Spanner、Postgres、または CloudSQL のようなオペレーショナルデータベースが最良の選択です。大規模な OLAP (オンライン分析処理) 分析ワークロードやバッチユースケースの場合、BigQuery のベクトル検索を使用する方が好ましいです。

結論として、ベクトルデータベースを選択する際には、様々な要因を考慮する必要があります。これらの要因には、データセットのサイズと種類（一部はスパースなデータに、その他は密なデータに適している）、ビジネスニーズ、ワークロードの性質、予算、セキュリティ、プライバシー保証、セマンティック検索および構文検索のニーズ、ならびにすでに使用されているデータベースシステムが含まれます。本節では、様々な ANN 検索アプローチ、ならびにベクトルデータベースの必要性と利点について見てきました。次のセクションでは、セマンティック検索のために Vertex AI ベクトル検索を使用する例を示します。

応用

埋め込みモデルは、様々なアプリケーションの原動力となる基本的な機械学習モデルの 1 つです。以下の表に、いくつかの一般的な応用をまとめます。

タスク	説明
検索 (Retrieval)	クエリとオブジェクトのセット（例：文書、画像、動画）が与えられた場合に、最も関連性の高いオブジェクトを検索します。関連オブジェクトの定義に基づき、サブタスクには質問応答や推薦が含まれます。
セマンティックテキスト類似度 (Semantic text	2 つの文が同じ意味的内容を持つかどうかを判断します。サブタスクには、言い換え、重複検出、バイテキストマイニングが含まれます。
分類 (Classification)	オブジェクトを可能性のあるカテゴリに分類します。ラベルの数に基づき、サブタスクには 2 値分類、多クラス分類、マルチラベル分類が含まれます。
クラスタリング (Clustering)	類似したオブジェクトをまとめてクラスタリングします。
リランキング (Reranking)	特定のクエリに基づいてオブジェクトのセットを再ランキングします。

埋め込みは、ANN (近似最近傍) 検索を提供するベクトルストアと共に、様々な応用に利用できる強力なツールです。これらには、LLM のための検索拡張生成 (RAG)、検索、推薦システム、異常検知、フューショット分類などが含まれますが、これらに限りません。

検索や推薦のようなランキング問題では、埋め込みは通常、プロセスの最初の段階で使用されます。埋め込みは、意味的に類似した潜在的に良い候補を取得し、結果として検索結果の関連性を向上させます。選別すべき情報量は非常に多くなる可能性があるため（場合によっては数百万から数十億にも及ぶ）、ScaNN のような ANN 技術は、検索空間をスケーラブルに絞り込む上で大いに役立ちます。この最初の結果セットは、このより小さな候補セットに対して、より高度なモデルを用いてさらに洗練させることができます。

LLM と RAG の両方を組み合わせて質問応答を支援する応用例を見てみましょう。

出典付き Q&A（検索拡張生成：RAG）

Q&A のための検索拡張生成 (RAG) は、検索と生成の両者の長所を組み合わせた技術です。まず知識ベースから関連文書を検索し、次にプロンプト拡張を使用してそれらの文書から回答を生成します。プロンプト拡張は、データベース検索と組み合わせることで非常に強力になり得る技術です。プロンプト拡張では、モデルはデータベースから関連情報を（主にセマンティック検索とビジネスルールの組み合わせを使用して）取得し、それを用いて元のプロンプトを拡張します。モデルはこの拡張されたプロンプトを使用して、検索のみまたは生成のみの場合よりもはるかに興味深く、事実に基づいた有益なコンテンツを生成します。

RAG は、LLM に関する 2 つの一般的な問題、すなわち、1)「ハルシネーション」を起こし、事実と異なるがもっともらしい応答を生成する傾向、および 2) 新しいデータはモデル学習時ではなくプロンプト経由で供給できるため、最新情報に追随するための再学習コストが高いこと、の解決に役立ちます。RAG はハルシネーションを削減できますが、完全に排除することはできません。この問題をさらに軽減するのに役立つのは、検索から得られた出典も返し、人間または LLM によって簡単な一貫性チェックを行うことです。これにより、LLM の応答が意味的に関連のある出典と一致していることが保証されます。LangChain[40] のようなライブラリと連携して Vertex AI LLM テキスト埋め込みおよび Vertex AI ベクトル検索を使用することでスケーラブルに実装できる、出典付き RAG の例 (スニペット 12) を見てみましょう。

```
# Before you start run this command:
# pip install --upgrade --user --quiet google-cloud-aiplatform langchain_google_vertexai
# after running pip install make sure you restart your kernel

# TODO : Set values as per your requirements
# Project and Storage Constants
PROJECT_ID = "<my_project_id>"
REGION = "<my_region>"
BUCKET = "<my_gcs_bucket>"
BUCKET_URI = f"gs://{BUCKET}"

# The number of dimensions for the text-embedding-005 is 768
# If other embedder is used, the dimensions would probably need to change.
DIMENSIONS = 768

# Index Constants
DISPLAY_NAME = "<my_matching_engine_index_id>"
DEPLOYED_INDEX_ID = "yourname01" # you set this. Start with a letter.

from google.cloud import aiplatform
from langchain_google_vertexai import VertexAIEmbeddings
from langchain_google_vertexai import VertexAI
from langchain_google_vertexai import (
    VectorSearchVectorStore,
    VectorSearchVectorStoreDatastore,
)
from langchain.chains import RetrievalQA
from langchain.prompts.chat import (
    ChatPromptTemplate,
    SystemMessagePromptTemplate,
    HumanMessagePromptTemplate,
)
from IPython.display import display, Markdown

aiplatform.init(project=PROJECT_ID, location=REGION, staging_bucket=BUCKET_URI)
embedding_model = VertexAIEmbeddings(model_name="text-embedding-005")

# NOTE : This operation can take upto 30 seconds
my_index = aiplatform.MatchingEngineIndtex.create_tree_ah_index(
    display_name=DISPLAY_NAME,
    dimensions=DIMENSIONS,
    approximate_neighbors_count=150,
    distance_measure_type="DOT_PRODUCT_DISTANCE",
    index_update_method="STREAM_UPDATE",  # allowed values BATCH_UPDATE , STREAM_UPDATE
)
```

次のページに続きます

```
# Create an endpoint
my_index_endpoint = aiplatform.MatchingEngineIndexEndpoint.create(
    display_name=f"{DISPLAY_NAME}-endpoint", public_endpoint_enabled=True
)

# NOTE : This operation can take upto 20 minutes
my_index_endpoint = my_index_endpoint.deploy_index(
    index=my_index, deployed_index_id=DEPLOYED_INDEX_ID
)

my_index_endpoint = my_index_endpoint.deploy_index(
    index=my_index, deployed_index_id=DEPLOYED_INDEX_ID
)
my_index_endpoint.deployed_indexes

# TODO : replace 1234567890123456789 with your acutial index ID
my_index = aiplatform.MatchingEngineIndex("1234567890123456789")

# TODO : replace 1234567890123456789 with your acutial endpoint ID
# Be aware that the Index ID differs from the endpoint ID
my_index_endpoint = aiplatform.MatchingEngineIndexEndpoint("1234567890123456789")

from langchain_google_vertexai import (
    VectorSearchVectorStore,
    VectorSearchVectorStoreDatastore,
)

# Input texts
texts = [
    "The cat sat on",
    "the mat.",
    "I like to",
    "eat pizza for",
    "dinner.",
    "The sun sets",
    "in the west.",
]
```

次のページに続きます

```
# Create a Vector Store
vector_store = VectorSearchVectorStore.from_components(
    project_id=PROJECT_ID,
    region=REGION,
    gcs_bucket_name=BUCKET,
    index_id=my_index.name,
    endpoint_id=my_index_endpoint.name,
    embedding=embedding_model,
    stream_update=True,
)

# Add vectors and mapped text chunks to your vectore store
vector_store.add_texts(texts=texts)

# Initialize the vectore_store as retriever
retriever = vector_store.as_retriever()

# perform simple similarity search on retriever
retriever.invoke("What are my options in breathable fabric?")
```

スニペット 12：Vertex AI ベクトル検索のための ANN インデックスの構築 / デプロイと、LLM プロンプトを用いた RAG による根拠のある結果 / ソースの生成

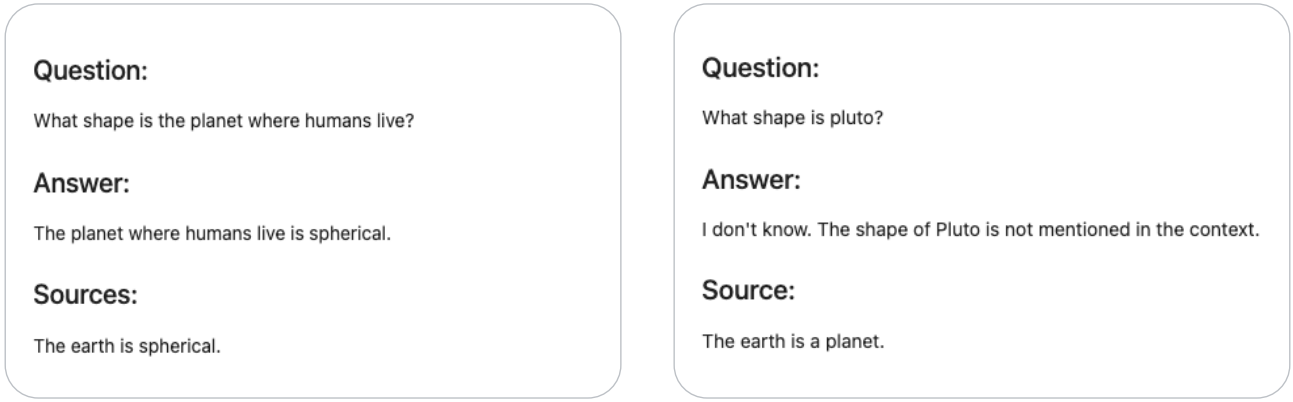

図 17：LLM がデータベースに根差していることを示す、出典付きのモデル応答

[図 17] から推測できるように、出力は LLM をデータベースから取得された意味的に類似した結果に根差させるだけでなく（そのため、データベースで文脈が見つからない場合は回答を拒否します）、ハルシネーションを大幅に削減し、さらに人間または別の LLM による検証のための出典も提供します。

まとめ

本稿では、本番グレードのアプリケーションの文脈において、様々なデータモダリティの埋め込みを効果的に作成、管理、保存、検索するための様々な手法について説明しました。後続アプリケーションのために埋め込みを作成、維持、使用することは、組織内の複数の役割が関与する複雑なタスクとなり得ます。しかし、その利用を徹底的に運用化および自動化することにより、最も重要なアプリケーションのいくつかにおいて、埋め込みが提供する素晴らしい利点を安全に活用することができます。本稿の主要なポイントには以下が含まれます。

1. データとユースケースに合わせて埋め込みモデルを賢く選択すること。推論で使用されるデータが学習で使用されたデータと一致していることを確認してください。学習から推論への分布シフトは、ドメイン分布シフトや後続タスク分布シフトなど、様々な要因から生じる可能性があります。既存の埋め込みモデルが現在の推論データ分布に適合しない場合、既存モデルのファインチューニングがパフォーマンスの大幅な向上に役立ちます。もう 1 つのトレードオフはモデルサイズから生じます。大規模ディープニューラルネットワーク（大規模マルチモーダルモデル）ベースのモデルは通常、より優れたパフォーマンスを発揮しますが、より長いサービングレイテンシというコストを伴う可能性があります。クラウドベースの埋め込みサービスを使用することで、高品質かつ低レイテンシの埋め込みサービスを提供することにより、上記の問題を克服できます。ほとんどのビジネスアプリケーションでは、事前学習済み埋め込みモデルを使用することで良好なベースラインが得られ、これはさらにファインチューニングしたり、後続モデルに統合したりすることができます。データが固有のグラフ構造を持つ場合、グラフ埋め込みが優れたパフォーマンスを提供できます。

2. 埋め込み戦略が定義されたら、予算とビジネスニーズに適合する適切なベクトルデータベースを選択することが重要です。利用可能なオープンソースの代替手段でプロトタイプを作成する方が迅速に思えるかもしれませんが、より安全でスケーラブル、かつ実戦で検証済みのマネージドベクトルデータベースを選択することで、開発者の時間を大幅に節約できます。多くの強力な ANN ベクトル検索アルゴリズムのいずれかを使用した様々なオープンソースの代替手段がありますが、ScaNN と HNSW は最高の精度とパフォーマンスのトレードオフのいくつかを提供することが証明されています。

3. 埋め込みを適切な ANN を利用したベクトルデータベースと組み合わせることは非常に強力なツールであり、検索、推薦システム、LLM のための検索拡張生成など、様々な応用に活用できます。このアプローチは、ハルシネーション問題を軽減し、LLM ベースシステムの検証可能性と信頼性を強化することができます。

卷末注

1. Rai, A., 2020, Study of various methods for tokenization. In Advances in Natural Language Processing. Available at: https://doi.org/10.1007/978-981-15-6198-6_18

2. Pennington, J., Socher, R. & Manning, C., 2014, GloVe: Global Vectors for Word Representation. [online] Available at: https://nlp.stanford.edu/pubs/glove.pdf.

3. Shazeer, N., Mirhoseini, A., Maziarz, K., Davis, A., Le, Q. V. & Hinton, G., 2016, Swivel: Improving embeddings by noticing what's missing. ArXiv, abs/1602.02215. Available at: https://arxiv.org/abs/1602.02215.

4. Mikolov, T., Sutskever, I., Chen, K., Corrado, G. & Dean, J., 2013, Efficient estimation of word representations in vector space. ArXiv, abs/1301.3781. Available at: https://arxiv.org/pdf/1301.3781.pdf.

5. Rehurek, R., 2021, Gensim: open source python library for word and document embeddings. Available at: https://radimrehurek.com/gensim/intro.html.

6. Bojanowski, P., Grave, E., Joulin, A. & Mikolov, T., 2016, Enriching word vectors with subword information. ArXiv, abs/1607.04606. Available at: https://arxiv.org/abs/1607.04606.

7. Deerwester, S., Dumais, S. T., Furnas, G. W., Landauer, T. K., & Harshman, R., 1990, Indexing by latent semantic analysis. Journal of the American Society for Information Science, 41(6), pp. 391-407.

8. Blei, D. M., Ng, A. Y., & Jordan, M. I., 2001, Latent Dirichlet allocation. In T. G. Dietterich, S. Becker, & Z. Ghahramani (Eds.), Advances in Neural Information Processing Systems 14. MIT Press, pp. 601-608. Available at: https://proceedings.neurips.cc/paper/2001/hash/296472c9542ad4d4788d543508116cbc-Abstract.html.

9. Muennighoff, N., Tazi, N., Magne, L., & Reimers, N., 2022, Mteb: Massive text embedding benchmark. ArXiv, abs/2210.07316. Available at: https://arxiv.org/abs/2210.07316.

10. Le, Q. V., Mikolov, T., 2014, Distributed representations of sentences and documents. ArXiv, abs/1405.4053. Available at: https://arxiv.org/abs/1405.4053.

11. Devlin, J., Chang, M. W., Lee, K., & Toutanova, K., 2019, BERT: Pre-training deep Bidirectional Transformers for Language Understanding. In Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers), pp. 4171-4186. Available at: https://www.aclweb.org/anthology/N19-1423/.

12. Reimers, N. & Gurevych, I., 2020, Making monolingual sentence embeddings multilingual using knowledge distillation. In Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP), pp. 254-265. Available at: https://www.aclweb.org/anthology/2020.emnlp-main.21/.

13. Gao, T., Yao, X. & Chen, D., 2021, Simcse: Simple contrastive learning of sentence embeddings. ArXiv, abs/2104.08821. Available at: https://arxiv.org/abs/2104.08821.
14. Wang, L., Yang, N., Huang, X., Jiao, B., Yang, L., Jiang, D., Majumder, R. & Wei, F., 2022, Text embeddings by weakly supervised contrastive pre-training. ArXiv. Available at: https://arxiv.org/abs/2201.01279.
15. Khattab, O. & Zaharia, M., 2020, colBERT: Efficient and effective passage search via contextualized late interaction over BERT. In Proceedings of the 43rd International ACM SIGIR Conference on Research and Development in Information Retrieval, pp. 39-48. Available at: https://dl.acm.org/doi/10.1145/3397271.3401025.
16. Lee, J., Dai, Z., Duddu, S. M. K., Lei, T., Naim, I., Chang, M. W. & Zhao, V. Y., 2023, Rethinking the role of token retrieval in multi-vector retrieval. ArXiv, abs/2304.01982. Available at: https://arxiv.org/abs/2304.01982.
17. TensorFlow, 2021, TensorFlow hub, a model zoo with several easy to use pre-trained models. Available at: https://tfhub.dev/.
18. Zhang, W., Xiong, C., & Zhao, H., 2023, Introducing BigQuery text embeddings for NLP tasks. Google Cloud Blog. Available at: https://cloud.google.com/blog/products/data-analytics/introducing-bigquery-text-embeddings.
19. Google Cloud, 2024, Get multimodal embeddings. Available at: https://cloud.google.com/vertex-ai/generative-ai/docs/embeddings/get-multimodal-embeddings.
20. Pinecone, 2024, IT Threat Detection. [online] Available at: https://docs.pinecone.io/docs/it-threat-detection.
21. Cai, H., Zheng, V. W., & Chang, K. C., 2020, A survey of algorithms and applications related with graph embedding. In Proceedings of the 29th ACM International Conference on Information & Knowledge Management. Available at: https://dl.acm.org/doi/10.1145/3444370.3444568.
22. Cai, H., Zheng, V. W., & Chang, K. C., 2017, A comprehensive survey of graph embedding: problems, techniques and applications. ArXiv, abs/1709.07604. Available at: https://arxiv.org/pdf/1709.07604.pdf.
23. Hamilton, W. L., Ying, R. & Leskovec, J., 2017, Inductive representation learning on large graphs. In Advances in Neural Information Processing Systems 30. Available at: https://cs.stanford.edu/people/jure/pubs/graphsage -nips17.pdf.
24. Dong, Z., Ni, J., Bikel, D. M., Alfonseca, E., Wang, Y., Qu, C. & Zitouni, I., 2022, Exploring dual encoder architectures for question answering. ArXiv, abs/2204.07120. Available at: https://arxiv.org/abs/2204.07120.
25. Google Cloud, 2021, Vertex AI Generative AI: Tune Embeddings. Available at: https://cloud.google.com/vertex-ai/docs/generative-ai/models/tune-embeddings.

26. Matsui, Y., 2020, Survey on approximate nearest neighbor methods. ACM Computing Surveys (CSUR), 53(6), Article 123. Available at: https://wangzwhu.github.io/home/file/acmmm-t-part3-ann.pdf.
27. Friedman, J. H., Bentley, J. L. & Finkel, R. A., 1977, An algorithm for finding best matches in logarithmic expected time. ACM Transactions on Mathematical Software (TOMS), 3(3), pp. 209-226. Available at: https://dl.acm.org/doi/pdf/10.1145/355744.355745.
28. Scikit-learn, 2021, Scikit-learn, a library for unsupervised and supervised neighbors-based learning methods. Available at: https://scikit-learn.org/.
29. lshashing, 2021, An open source python library to perform locality sensitive hashing. Available at: https://pypi.org/project/lshashing/.
30. Malkov, Y. A., Yashunin, D. A., 2016, Efficient and robust approximate nearest neighbor search using hierarchical navigable small world graphs. ArXiv, abs/1603.09320. Available at: https://arxiv.org/pdf/1603.09320.pdf.
31. Google Research, 2021, A library for fast ANN by Google using the ScaNN algorithm. Available at: https://github.com/google-research/google-research/tree/master/scann.
32. Guo, R., Zhang, L., Hinton, G. & Zoph, B., 2020, Accelerating large-scale inference with anisotropic vector quantization. ArXiv, abs/1908.10396. Available at: https://arxiv.org/pdf/1908.10396.pdf.
33. TensorFlow, 2021, TensorFlow Recommenders, an open source library for building ranking & recommender system models. Available at: https://www.tensorflow.org/recommenders.
34. Google Cloud, 2021, Vertex AI Vector Search, Google Cloud's high-scale low latency vector database. Available at: https://cloud.google.com/vertex-ai/docs/vector-search/overview.
35. Elasticsearch, 2021, Elasticsearch: a RESTful search and analytics engine. Available at: https://www.elastic.co/elasticsearch/.
36. Pinecone, 2021, Pinecone, a commercial fully managed vector database. Available at: https://www.pinecone.io.
37. pgvector, 2021, Open Source vector similarity search for Postgres. Available at: https://github.com/pgvector/pgvector.
38. Weaviate, 2021, Weaviate, an open source vector database. Available at: https://weaviate.io/.
39. ChromaDB, 2021, ChromaDB, an open source vector database. Available at: https://www.trychroma.com/.

40. LangChain, 2021.,LangChain, an open source framework for developing applications powered by language model. Available at: https://langchain.com.

42. Thakur, N., Reimers, N., Ruckl’ e, A., Srivastava, A., & Gurevych, I. (2021). BEIR: A Heterogenous Benchmark for Zero-shot Evaluation of Information Retrieval Models. ArXiv, abs/2104.08663.
Available at: https://github.com/beir-cellar/beir

43. Niklas Muennighoff, Nouamane Tazi, Loic Magne, and Nils Reimers. 2023. MTEB: Massive Text Embedding Benchmark. In Proceedings of the 17th Conference of the European Chapter of the Association for Computational Linguistics, pages 2014–2037, Dubrovnik, Croatia. Association for Computational Linguistics.
Available at: https://github.com/embeddings-benchmark/mteb

44. Chris Buckley. trec_eval IR evaluation package. Available from https://github.com/usnistgov/trec_eval

45. Christophe Van Gysel and Maarten de Rijke. 2018. Pytrec_eval: An Extremely Fast Python Interface to trec_eval. In The 41st International ACM SIGIR Conference on Research & Development in Information Retrieval (SIGIR ‘18). Association for Computing Machinery, New York, NY, USA, 873–876.
Availalbe at: https://doi.org/10.1145/3209978.3210065

46. Boteva, Vera & Gholipour Ghalandari, Demian & Sokolov, Artem & Riezler, Stefan. (2016). A Full-Text Learning to Rank Dataset for Medical Information Retrieval. 9626. 716-722. 10.1007/978-3-319-30671-1_58. Available at https://www.cl.uni-heidelberg.de/statnlpgroup/nfcorpus/

47. Douze, M., Guzhva, A., Deng, C., Johnson, J., Szilvasy, G., Mazaré, P.E., Lomeli, M., Hosseini, L. and Jégou, H., 2024. The Faiss library. arXiv preprint arXiv:2401.08281. Available at https://arxiv.org/abs/2401.08281

48. Lee, J., Dai, Z., Ren, X., Chen, B., Cer, D., Cole, J.R., Hui, K., Boratko, M., Kapadia, R., Ding, W. and Luan, Y., 2024. Gecko: Versatile text embeddings distilled from large language models. arXiv preprint arXiv:2403.20327.
Available at: https://arxiv.org/abs/2403.20327

49. Okapi BM25: a non-binary model” Christopher D. Manning, Prabhakar Raghavan, Hinrich Schütze. An Introduction to Information Retrieval, Cambridge University Press, 2009, p. 232.

50. Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J. Liu. 2020. Exploring the limits of transfer learning with a unified text-to-text transformer. J. Mach.
Learn. Res. 21, 1, Article 140 (January 2020), 67 pages.
Available at https://dl.acm.org/doi/abs/10.5555/3455716.3455856

51. Aakanksha Chowdhery, Sharan Narang, Jacob Devlin, Maarten Bosma, Gaurav Mishra, Adam Roberts, Paul Barham, Hyung Won Chung, Charles Sutton, Sebastian Gehrmann, Parker Schuh, Kensen Shi, Sashank Tsvyashchenko, Joshua Maynez, Abhishek Rao, Parker Barnes, Yi Tay, Noam Shazeer, Vinodkumar Prabhakaran, Emily Reif, Nan Du, Ben Hutchinson, Reiner Pope, James Bradbury, Jacob Austin, Michael Isard, Guy Gur-Ari, Pengcheng Yin, Toju Duke, Anselm Levskaya, Sanjay Ghemawat, Sunipa Dev, Henryk Michalewski, Xavier Garcia, Vedant Misra, Kevin Robinson, Liam Fedus, Denny Zhou, Daphne Ippolito, David Luan, Hyeontaek Lim, Barret Zoph, Alexander Spiridonov, Ryan Sepassi, David Dohan, Shivani Agrawal, Mark Omernick, Andrew M. Dai, Thanumalayan Sankaranarayana Pillai, Marie Pellat, Aitor Lewkowycz, Erica Moreira, Rewon Child, Oleksandr Polozov, Katherine Lee, Zongwei Zhou, Xuezhi Wang, Brennan Saeta, Mark Diaz, Orhan Firat, Michele Catasta, Jason Wei, Kathy Meier-Hellstern, Douglas Eck, Jeff Dean, Slav Petrov, and Noah Fiedel. 2023. PaLM: scaling language modeling with pathways. J. Mach. Learn. Res. 24, 1, Article 240 (January 2023), 113 pages. Available at https://dl.acm.org/doi/10.5555/3648699.3648939

52. Gemini: A Family of Highly Capable Multimodal Models, Gemini Team, Dec 2023. Available at: https://storage.googleapis.com/deepmind-media/gemini/gemini_1_report.pdf

53. Radford, Alec and Karthik Narasimhan. "Improving Language Understanding by Generative Pre-Training." (2018). Available at: https://cdn.openai.com/research-covers/language-unsupervised/language_understanding_paper.pdf

54. Touvron, H., Lavril, T., Izacard, G., Martinet, X., Lachaux, M.A., Lacroix, T., Rozière, B., Goyal, N., Hambro, E., Azhar, F. and Rodriguez, A., 2023. Llama: Open and efficient foundation language models. arXiv preprint arXiv:2302.13971. Available at: https://arxiv.org/abs/2302.13971

55. Kusupati, A., Bhatt, G., Rege, A., Wallingford, M., Sinha, A., Ramanujan, V., Howard-Snyder, W., Chen, K., Kakade, S., Jain, P. and Farhadi, A., 2022. Matryoshka representation learning. Advances in Neural Information Processing Systems, 35, pp.30233-30249. Available at: https://proceedings.neurips.cc/paper_files/paper/2022/file/c32319f4868da7613d78af9993100e42-Paper-Conference.pdf

56. Nair, P., Datta, P., Dean, J., Jain, P. and Kusupati, A., 2025. Matryoshka Quantization. arXiv preprint arXiv:2502.06786. Available at: https://arxiv.org/abs/2502.06786

57. Faysse, M., Sibille, H., Wu, T., Omrani, B., Viaud, G., Hudelot, C. and Colombo, P., 2024. Colpali: Efficient document retrieval with vision language models. arXiv preprint arXiv:2407.01449. Available at: https://arxiv.org/abs/2407.01449

58. Aumüller, M., Bernhardsson, E. and Faithfull, A., 2020. ANN-Benchmarks: A benchmarking tool for approximate nearest neighbor algorithms. Information Systems, 87, p.101374.