Day2

Explore text similarity with embeddings

Today's Assignments

Complete Unit 2: "Embeddings and Vector Stores/Databases", which is:

- [Optional] Listen to the summary podcast episode for this unit (created by NotebookLM).
- Read the "Embeddings and Vector Stores/Databases" whitepaper.
- Complete these code labs on Kaggle:
  1. Build a RAG question-answering system over custom documents
  2. Explore text similarity with embeddings
  3. Build a neural classification network with Keras using embeddings

What You'll Learn

Today you will learn about the conceptual underpinning of embeddings and vector databases and how they can be used to bring live or specialist data into your LLM application. You'll also explore their geometrical powers for classifying and comparing textual data.

Reminders and Announcements

📋

- Here is the recording from this morning's livestream. Fortunately our recording did not have any of the technical glitches as today's livestream.
- The 2nd livestream is tomorrow with Paige Bailey and special guests: Omid Fatemieh, Jinhyuk Lee, Alan Li, Iftekhar Naim, Anant Nawalgaria, Yan Qiao, and Xiaoqi Ren.
- Unfortunately, to ensure a fix to our livestream issues moving forward, we need to push back our broadcast time. We'll send another email with updated livestream info soon.
- Be sure to ask all your questions about the podcast, readings, and code lab in the #5dgai-q-and-a channel on Discord. You'll get Kaggle swag if your question is chosen for discussion during the livestream!

Happy learning and see you tomorrow,

The Kaggle Team

A. Introduction (Giới thiệu)

Học máy hiện đại phát triển dựa trên dữ liệu đa dạng—hình ảnh, văn bản, âm thanh, và nhiều loại khác. Tài liệu này khám phá sức mạnh của embedding, một kỹ thuật biến đổi dữ liệu không đồng nhất này thành dạng biểu diễn vector thống nhất để sử dụng dễ dàng trong các ứng dụng khác nhau.

Chúng tôi sẽ hướng dẫn bạn qua các nội dung sau:

- Hiểu về Embedding: Tại sao chúng lại quan trọng trong việc xử lý dữ liệu đa phương thức và các ứng dụng đa dạng của chúng.
- Kỹ thuật Embedding: Các phương pháp ánh xạ các loại dữ liệu khác nhau vào không gian vector chung.
- Quản lý Hiệu quả: Các kỹ thuật lưu trữ, truy xuất và tìm kiếm các bộ embedding lớn.
- Cơ sở Dữ liệu Vector: Các hệ thống chuyên biệt để quản lý và truy vấn embedding, bao gồm các cân nhắc thực tế khi triển khai trong môi trường sản xuất.
- Ứng dụng Thực tế: Các ví dụ cụ thể về cách embedding và cơ sở dữ liệu vector kết hợp với các mô hình ngôn ngữ lớn (LLM) để giải quyết các vấn đề thực tế.

Trong suốt tài liệu, các đoạn mã minh họa sẽ cung cấp ví dụ thực hành về các khái niệm chính.

B. Why embeddings are important (Tại sao embeddings lại quan trọng)

Về bản chất, embedding là các biểu diễn số của dữ liệu thực tế như văn bản, giọng nói, hình ảnh hoặc video. Chúng được biểu diễn dưới dạng vector có số chiều thấp, trong đó khoảng cách hình học giữa hai vector trong không gian vector thể hiện mối quan hệ giữa hai đối tượng thực tế mà các vector đó đại diện. Nói cách khác, embedding giúp cung cấp các biểu diễn gọn nhẹ của dữ liệu với các loại khác nhau, đồng thời cho phép so sánh hai đối tượng dữ liệu khác nhau để xác định mức độ tương đồng hoặc khác biệt của chúng trên thang điểm số. Ví dụ: từ "computer" có nghĩa tương đồng với hình ảnh của một chiếc máy tính, cũng như từ "laptop" nhưng không tương đồng với từ "car". Các biểu diễn số có số chiều thấp này của dữ liệu thực tế giúp việc xử lý và lưu trữ dữ liệu quy mô lớn hiệu quả hơn bằng cách nén mất dữ liệu gốc nhưng vẫn giữ được các thuộc tính quan trọng của nó.

Một trong những ứng dụng chính của embedding là truy xuất và đề xuất, nơi mà kết quả thường đến từ một không gian tìm kiếm khổng lồ. Ví dụ, Google Search là một hệ thống truy xuất với không gian tìm kiếm là toàn bộ Internet. Thành công của các hệ thống truy xuất và đề xuất ngày nay phụ thuộc vào các yếu tố sau:

1. Tính toán trước các embedding cho hàng tỷ mục trong không gian tìm kiếm.
2. Ánh xạ các embedding truy vấn vào cùng một không gian embedding.
3. Tính toán và truy xuất hiệu quả các lân cận gần nhất của embedding truy vấn trong không gian tìm kiếm.

Embedding cũng nổi bật trong lĩnh vực đa phương thức. Hầu hết các ứng dụng làm việc với lượng lớn dữ liệu từ các loại phương thức khác nhau: văn bản, giọng nói, hình ảnh và video, để kể một vài loại. Vì mỗi thực thể hoặc đối tượng được biểu diễn dưới dạng riêng biệt của nó, rất khó để chiếu các đối tượng này vào cùng một không gian vector vừa gọn nhẹ vừa chứa đủ thông tin. Lý tưởng nhất, một biểu diễn như vậy sẽ nắm bắt được nhiều đặc điểm của đối tượng gốc nhất có thể. Embedding là vector chiếu của một đối tượng từ không gian đầu vào vào một không gian vector có số chiều thấp hơn. Mỗi vector là một danh sách các số thập phân.

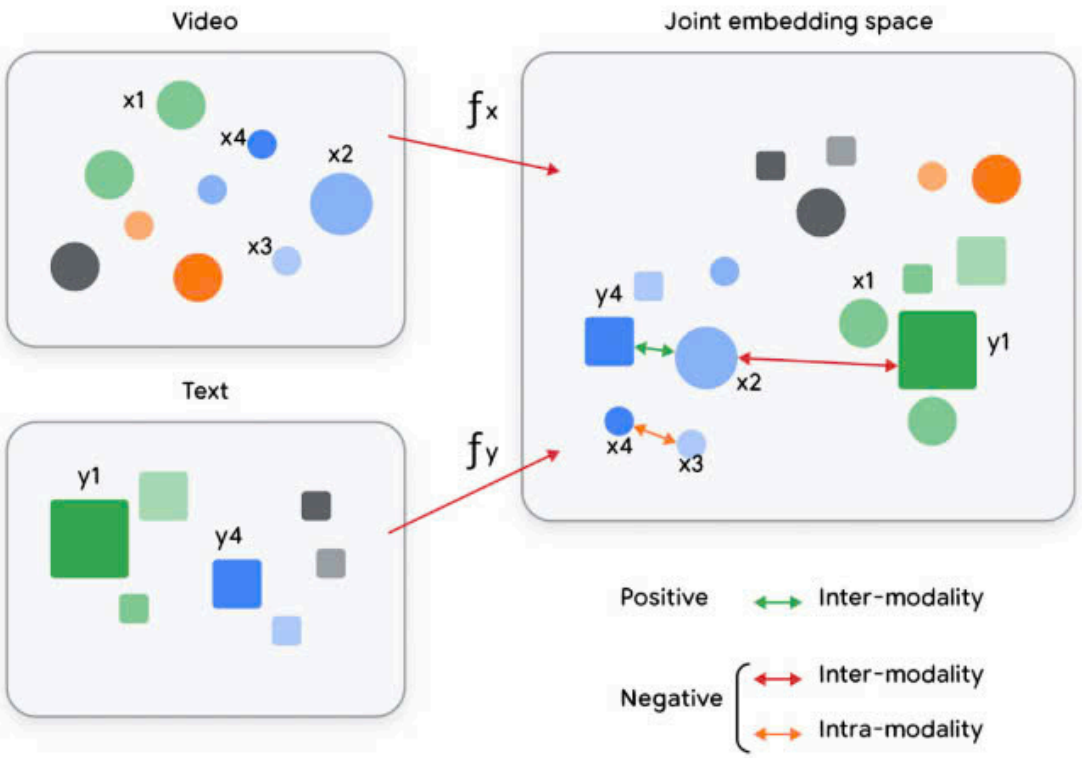


Figure 1. Chiếu các đối tượng/nội dung vào không gian vectơ chung có ý nghĩa ngữ nghĩa

Lý tưởng nhất, các embedding được tạo ra sao cho chúng đặt các đối tượng có thuộc tính ngữ nghĩa tương tự gần nhau hơn trong không gian embedding (một không gian vector có số chiều thấp, nơi các đối tượng có thể được chiếu tới). Embedding sau đó có thể được sử dụng như một đầu vào cô đọng và ý nghĩa cho các ứng dụng phía sau.

Ví dụ, bạn có thể sử dụng embedding làm đặc trưng (features) cho các mô hình học máy (ML), hệ thống gợi ý, công cụ tìm kiếm và nhiều ứng dụng khác. Nhờ đó, dữ liệu của bạn không chỉ được biểu diễn dưới dạng số gọn nhẹ mà còn giữ được ý nghĩa ngữ nghĩa cho một nhiệm vụ cụ thể hoặc trên nhiều nhiệm vụ khác nhau.

Việc các biểu diễn này mang tính đặc thù cho từng nhiệm vụ đồng nghĩa với việc bạn có thể tạo ra các embedding khác nhau cho cùng một đối tượng, được tối ưu hóa cho nhiệm vụ hiện tại.

1. Types of embeddings (Các loại embeddings)

Embedding nhằm mục đích tạo ra một biểu diễn dữ liệu có số chiều thấp nhưng vẫn giữ được hầu hết thông tin "cốt lõi" của dữ liệu gốc. Loại dữ liệu mà embedding đại diện có thể có nhiều hình thức khác nhau. Dưới đây là một số kỹ thuật tiêu chuẩn được sử dụng cho các loại dữ liệu khác nhau, bao gồm văn bản và hình ảnh.

1.1 Text embeddings

Embedding văn bản được sử dụng rộng rãi trong lĩnh vực xử lý ngôn ngữ tự nhiên (NLP). Chúng thường được dùng để biểu diễn ý nghĩa của ngôn ngữ tự nhiên trong các mô hình học máy nhằm xử lý cho các ứng dụng phía sau như tạo văn bản, phân loại, phân tích cảm xúc, và nhiều hơn nữa. Các embedding này thường được chia thành hai loại chính: embedding cho token/từ và embedding cho tài liệu.

Trước khi đi sâu vào hai loại này, điều quan trọng là cần hiểu toàn bộ vòng đời của văn bản: từ khi nó được nhập vào bởi người dùng đến khi nó được chuyển đổi thành embedding.

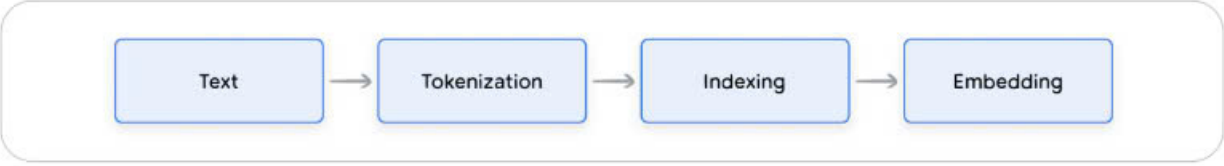


Figure 2: quá trình chuyển đổi text sang embeddings

Quá trình này bắt đầu với chuỗi đầu vào, được chia nhỏ thành các phần có ý nghĩa gọi là **token**. Quá trình này được gọi là **tokenization** (phân tách từ). Thông thường, các token này bao gồm các đoạn từ, ký tự, từ, số và dấu câu, được tạo ra bằng cách sử dụng một trong nhiều kỹ thuật phân tách từ hiện có.

Sau khi chuỗi được phân tách, mỗi token sẽ được gán một giá trị số nguyên duy nhất, thường nằm trong phạm vi: `[0, tổng số token trong tập dữ liệu]`. Ví dụ, với một từ vựng gồm 16 từ, các ID sẽ nằm trong khoảng từ 0-15. Giá trị này còn được gọi là **token ID**. Những token này có thể được sử dụng để biểu diễn mỗi chuỗi dưới dạng vector số thưa thớt phục vụ trực tiếp cho các nhiệm vụ phía sau, hoặc sau khi được mã hóa **one-hot encoding**.

One-hot encoding là một dạng biểu diễn nhị phân của các giá trị phân loại, trong đó sự xuất hiện của một từ được biểu thị bằng giá trị `1`, và sự vắng mặt của nó là

`0`. Điều này đảm bảo rằng các token ID được coi là giá trị phân loại, nhưng thường dẫn đến một vector dày đặc có kích thước bằng từ vựng của tập dữ liệu. Đoạn mã Snippet 1 và Hình 3 minh họa cách thực hiện điều này bằng TensorFlow.

```
# 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',
split=' ')
tokenizer.fit_on_texts(data)
# Translate each sentence into its word-level IDs, and then one-hot encode thos
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 )
```

Snippet 1. Tokenizing, indexing and one-hot encoding strings

```
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.]]
```

Figure 3. Output of Snippet 1

Tuy nhiên, vì các ID số nguyên này (hoặc các vector one-hot encoding tương ứng) được gán ngẫu nhiên cho các từ, chúng không mang ý nghĩa ngữ nghĩa vốn có nào. Đây là lúc embedding trở nên hữu ích hơn nhiều. Mặc dù cũng có thể tạo embedding cho các token ở cấp ký tự hoặc cấp dưới từ, chúng ta hãy xem xét

embedding cho từ và tài liệu để hiểu rõ hơn về một số phương pháp đằng sau chúng.

Word embeddings

Trong phần này, bạn sẽ thấy một số kỹ thuật và thuật toán embedding từ để huấn luyện và sử dụng embedding từ. Mặc dù có nhiều thuật toán dựa trên học máy được phát triển theo thời gian nhằm tối ưu hóa cho các mục tiêu khác nhau, nhưng các phương pháp phổ biến nhất là **GloVe**, **SWIVEL**, và **Word2Vec**. Các embedding từ hoặc dưới từ (sub-word) cũng có thể được lấy trực tiếp từ các lớp ẩn của các mô hình ngôn ngữ. Tuy nhiên, các embedding này sẽ khác nhau đối với cùng một từ trong các ngữ cảnh khác nhau của văn bản. Phần này tập trung vào embedding từ gọn nhẹ và không phụ thuộc ngữ cảnh, còn các embedding tài liệu nhận biết ngữ cảnh sẽ được thảo luận ở phần về embedding tài liệu. Embedding từ có thể được áp dụng trực tiếp vào các nhiệm vụ phía sau như trích xuất thực thể được đặt tên (named entity extraction) và mô hình hóa chủ đề (topic modeling).

Word2Vec là một họ các kiến trúc mô hình hoạt động dựa trên nguyên tắc **"nghĩa ngữ nghĩa của một từ được xác định bởi các từ lân cận"**, tức là các từ thường xuất hiện gần nhau trong tập dữ liệu huấn luyện. Phương pháp này có thể được sử dụng để tự huấn luyện embedding từ các tập dữ liệu lớn hoặc tích hợp nhanh chóng thông qua các embedding đã được huấn luyện sẵn có trực tuyến.

Embedding cho mỗi từ - thực chất là các vector có độ dài cố định - được khởi tạo ngẫu nhiên để bắt đầu quá trình, dẫn đến một ma trận có dạng `(kích_thước_từ_vựng, kích_thước_embedding)`. Ma trận này có thể được sử dụng như một bảng tra cứu sau khi hoàn tất quá trình huấn luyện bằng một trong các phương pháp sau (xem Hình 4):

- Phương pháp Continuous Bag of Words (CBOW): Dự đoán từ ở giữa, sử dụng embedding của các từ xung quanh làm đầu vào. Phương pháp này không quan tâm đến thứ tự của các từ xung quanh trong ngữ cảnh. CBOW nhanh để huấn luyện và chính xác hơn một chút với các từ xuất hiện thường xuyên.
- Phương pháp Skip-gram: Thiết lập ngược lại với CBOW, sử dụng từ ở giữa để dự đoán các từ xung quanh trong một phạm vi nhất định. Phương pháp này chậm hơn để huấn luyện nhưng hoạt động tốt với dữ liệu nhỏ và chính xác hơn với các từ hiếm.

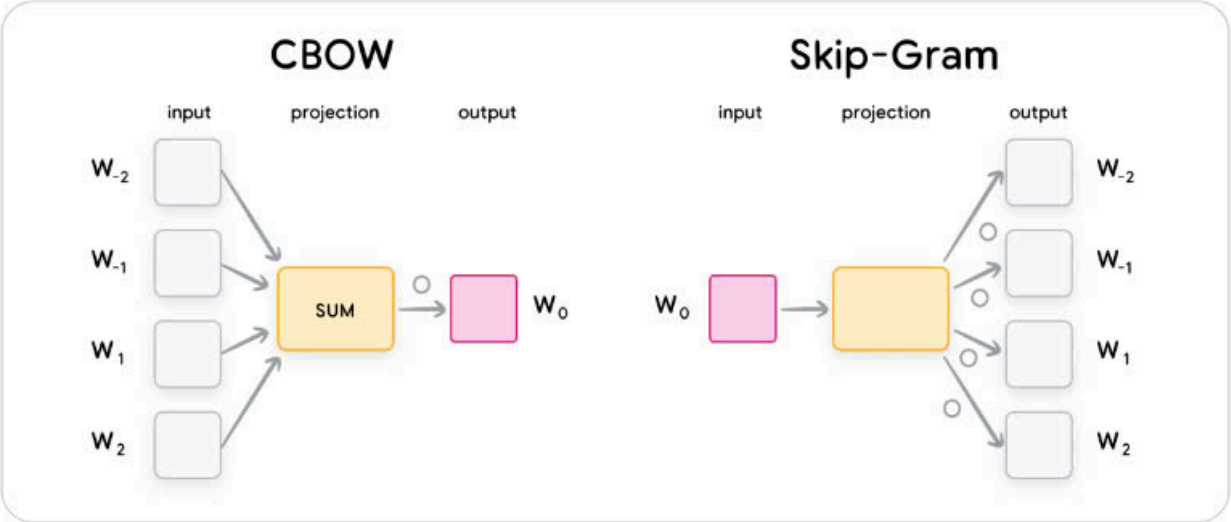


Figure 4. Sơ đồ giải thích cách phương thức CBOW và Skip-Gram hoạt động

Các thuật toán Word2Vec cũng có thể được mở rộng xuống cấp độ dưới từ (sub-word), điều này đã truyền cảm hứng cho các thuật toán như **FastText**.

Tuy nhiên, một trong những nhược điểm lớn của Word2Vec là mặc dù nó xử lý tốt các thống kê cục bộ của từ trong một cửa sổ trượt nhất định, nhưng nó không nắm bắt được các thống kê toàn cục (các từ trong toàn bộ tập dữ liệu). Hạn chế này chính là điều mà các phương pháp như thuật toán **GloVe** nhằm khắc phục.

GloVe là một kỹ thuật embedding từ tận dụng cả thống kê toàn cục và cục bộ của các từ. Nó thực hiện điều này bằng cách đầu tiên tạo ra một **ma trận đồng xuất hiện (co-occurrence matrix)**, biểu thị mối quan hệ giữa các từ. Sau đó, GloVe sử dụng một kỹ thuật phân rã ma trận (factorization) để học biểu diễn từ từ ma trận đồng xuất hiện này. Các biểu diễn từ kết quả có khả năng nắm bắt cả thông tin toàn cục lẫn cục bộ của từ và rất hữu ích cho nhiều nhiệm vụ NLP.

Ngoài GloVe, **SWIVEL** cũng là một phương pháp dựa trên ma trận đồng xuất hiện để học embedding từ. SWIVEL là viết tắt của **Skip-Window Vectors with Negative Sampling**. Khác với GloVe, SWIVEL sử dụng các cửa sổ cục bộ để học vector từ bằng cách xét đến sự đồng xuất hiện của các từ trong một cửa sổ cố định của các từ lân cận.

Hơn nữa, SWIVEL cũng xem xét các đồng xuất hiện chưa được quan sát và xử lý chúng bằng cách sử dụng một hàm mất mát đặc biệt theo từng phần (piecewise loss), giúp cải thiện hiệu suất với các từ hiếm. Phương pháp này thường được coi là chỉ kém chính xác hơn GloVe một chút, nhưng lại nhanh hơn đáng kể trong huấn

luyện. Điều này là do SWIVEL tận dụng huấn luyện phân tán bằng cách chia nhỏ các vector embedding thành các ma trận con nhỏ hơn và thực hiện phân rã ma trận song song trên nhiều máy.

Embedding từ có thể được sử dụng trực tiếp trong các nhiệm vụ phía sau như **Nhận diện thực thể được đặt tên (Named Entity Recognition - NER)**.

```
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 plot 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=250
        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()

# Load pre-trained models
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])

# Sample common words from both models
pprint.pprint("2D projection of some common words of both models")
sample_common_words = list(set(v2w_model.index_to_key[100:10000]) & set(glo

# Create and display the TSNE plot
tsne_plot([v2w_model, glove_model], sample_common_words)
```

Snippet 2. Tải và vẽ GloVe và Word2Vec embeddings trong 2D

Hình 5 Hiển thị các từ giống nhau về mặt ngữ nghĩa được phân cụm khác nhau cho hai thuật toán

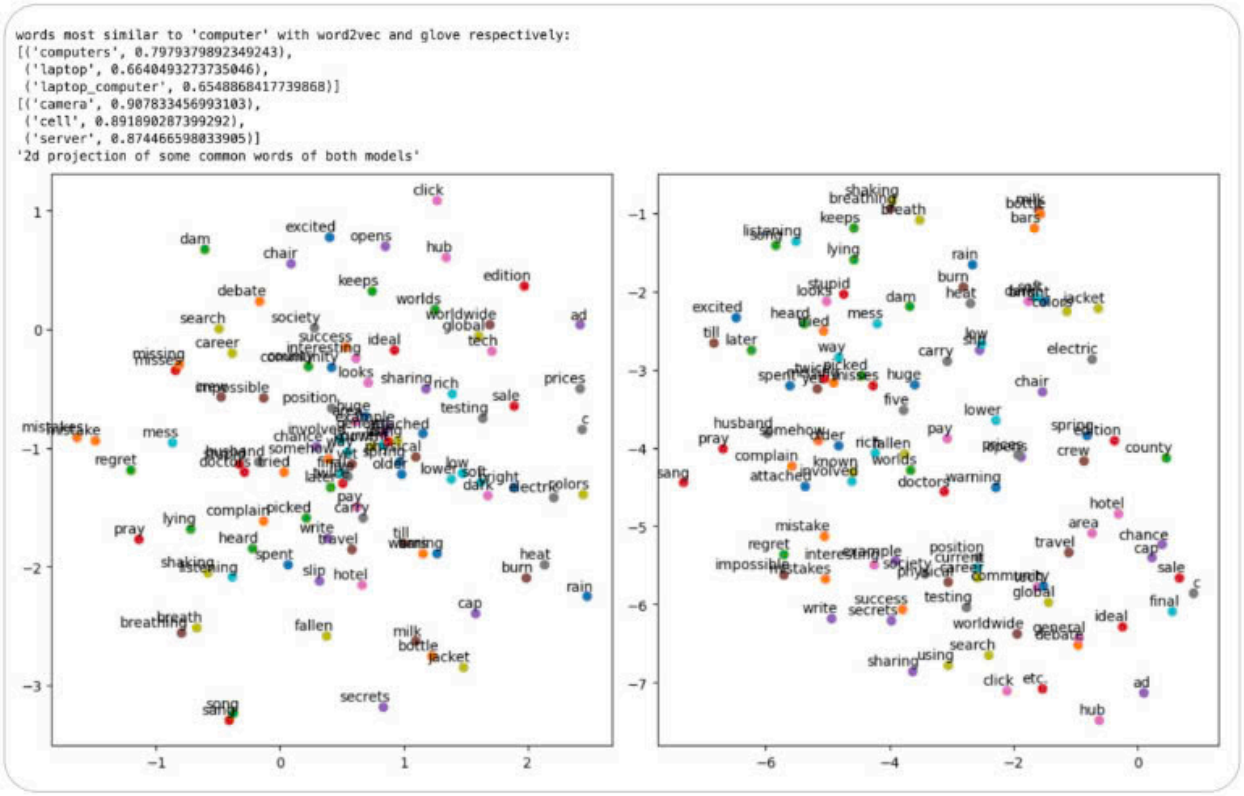


Figure 5. 2D visualization of pre-trained GloVe and Word2Vec word embeddings

Document embeddings

Embedding tài liệu vào không gian embedding dày đặc có số chiều thấp đã thu hút sự quan tâm lâu dài kể từ những năm 1980. Embedding tài liệu có thể được sử dụng trong nhiều ứng dụng, bao gồm tìm kiếm ngữ nghĩa, khám phá chủ đề, phân loại và phân cụm để biểu diễn ý nghĩa của một chuỗi từ trong các đoạn văn và tài liệu, và sử dụng nó cho nhiều ứng dụng phía sau. Sự phát triển của các mô hình embedding chủ yếu được phân loại thành hai giai đoạn: các mô hình Bag-of-words (BoW) nông và các mô hình ngôn ngữ lớn được huấn luyện sâu.

Shallow BoW models

Các công trình embedding tài liệu ban đầu tuân theo mô hình bag-of-words (BoW), giả định rằng một tài liệu là một tập hợp từ không có thứ tự. Các công trình ban đầu này bao gồm phân tích ngữ nghĩa tiềm ẩn (Latent Semantic Analysis - LSA) và phân bổ Dirichlet tiềm ẩn (Latent Dirichlet Allocation - LDA). Phân tích ngữ nghĩa tiềm ẩn (LSA) sử dụng ma trận đồng xuất hiện của các từ trong tài liệu, còn phân bổ Dirichlet tiềm ẩn (LDA) sử dụng một mạng Bayesian để mô hình hóa embedding tài liệu. Một họ nổi tiếng khác trong các mô hình bag-of-words là các

mô hình dựa trên TF-IDF (tần suất từ - nghịch đảo tần suất tài liệu), đây là các mô hình thống kê sử dụng tần suất từ để biểu diễn embedding tài liệu. Các mô hình dựa trên TF-IDF có thể là embedding thưa thớt, đại diện cho mức độ quan trọng của từng thuật ngữ, hoặc có thể được kết hợp với embedding từ như một yếu tố trọng số để tạo ra embedding dày đặc cho tài liệu. Ví dụ, BM25, một mô hình bag-of-words dựa trên TF-IDF, vẫn là một chuẩn cơ sở mạnh mẽ trong các tiêu chuẩn đánh giá truy xuất ngày nay.

Tuy nhiên, mô hình bag-of-words cũng có hai điểm yếu lớn: cả thứ tự từ và ý nghĩa ngữ nghĩa đều bị bỏ qua. Các mô hình BoW không thể nắm bắt được mối quan hệ tuần tự giữa các từ, điều này rất quan trọng để hiểu được ý nghĩa và ngữ cảnh. Lấy cảm hứng từ Word2Vec, Doc2Vec đã được đề xuất vào năm 2014 để tạo embedding tài liệu bằng cách sử dụng các mạng nơ-ron (nông). Mô hình Doc2Vec bổ sung thêm một embedding 'đoạn văn' hoặc, nói cách khác, embedding tài liệu vào mô hình Word2Vec như minh họa trong Hình 6. Embedding đoạn văn này được nối hoặc trung bình với các embedding từ khác để dự đoán một từ ngẫu nhiên trong đoạn văn. Sau khi huấn luyện, đối với các đoạn văn hoặc tài liệu hiện có, embedding đã học có thể được sử dụng trực tiếp trong các nhiệm vụ phía sau. Đối với một đoạn văn hoặc tài liệu mới, cần thực hiện các bước suy luận bổ sung để tạo ra embedding đoạn văn hoặc tài liệu.

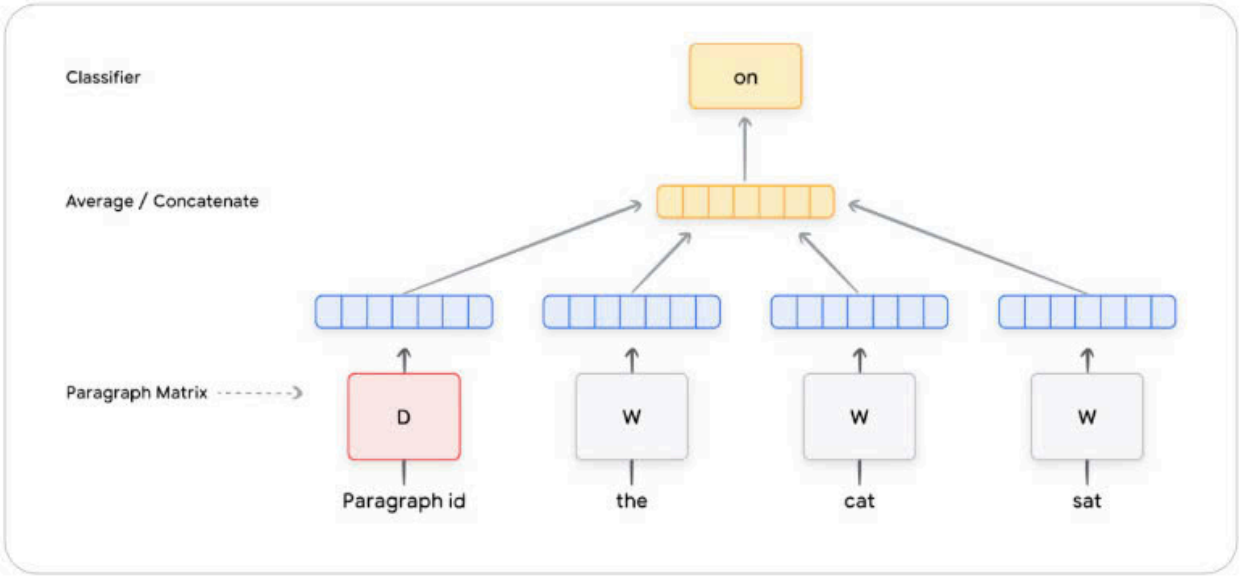


Figure 6. Mô hình Doc2vec CBOW

Đoạn 3 bên dưới cho thấy cách bạn có thể huấn luyện các mô hình doc2Vec của riêng mình trên kho văn bản tùy chỉnh:

```
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=
# 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"])
```

Snippet 3. Self-supervised Training and inference using Doc2Vec on private corpus

Thành công của việc áp dụng mạng nơ-ron trong lĩnh vực embedding đã truyền cảm hứng cho sự quan tâm ngày càng tăng đến việc sử dụng mạng nơ-ron sâu để tạo ra các embedding.

Deeper pretrained large language models

Lấy cảm hứng từ sự phát triển của mạng nơ-ron sâu, nhiều mô hình và kỹ thuật embedding khác nhau đã được đề xuất, và các mô hình tiên tiến nhất (state-of-the-art) liên tục được cập nhật. Những thay đổi chính của các mô hình này bao gồm:

1. Sử dụng các mô hình học phức tạp hơn, đặc biệt là các mô hình mạng nơ-ron sâu hai chiều (bi-directional deep neural networks).
2. Sử dụng việc huấn luyện trước quy mô lớn trên văn bản không gán nhãn.
3. Sử dụng bộ phân tách từ con (subword tokenizer).
4. Tinh chỉnh (fine-tuning) để thực hiện các nhiệm vụ NLP khác nhau.

Năm 2018, **BERT** (viết tắt của *Bidirectional Encoder Representations from Transformers*) được đề xuất và đạt được những kết quả đột phá trên 11 nhiệm vụ NLP. Transformer, mô hình nền tảng mà BERT dựa trên, đã trở thành mô hình chủ

đạo cho đến nay. Bên cạnh việc sử dụng transformer làm xương sống, một yếu tố quan trọng khác mang lại thành công cho BERT là huấn luyện trước (pre-training) trên một tập dữ liệu không gán nhãn khổng lồ. Trong quá trình huấn luyện trước, BERT sử dụng mục tiêu **masked language model (MLM)**, bằng cách ngẫu nhiên che đi một số token trong đầu vào và sử dụng ID của các token bị che để làm mục tiêu dự đoán. Điều này cho phép mô hình tận dụng cả ngữ cảnh bên trái và bên phải để huấn luyện trước một transformer hai chiều sâu. BERT cũng sử dụng nhiệm vụ dự đoán câu tiếp theo (*next sentence prediction*) trong quá trình huấn luyện trước.

BERT tạo ra embedding ngữ cảnh hóa cho từng token trong đầu vào. Thông thường, embedding của token đầu tiên (một token đặc biệt có tên là [CLS]) được sử dụng làm embedding cho toàn bộ đầu vào.

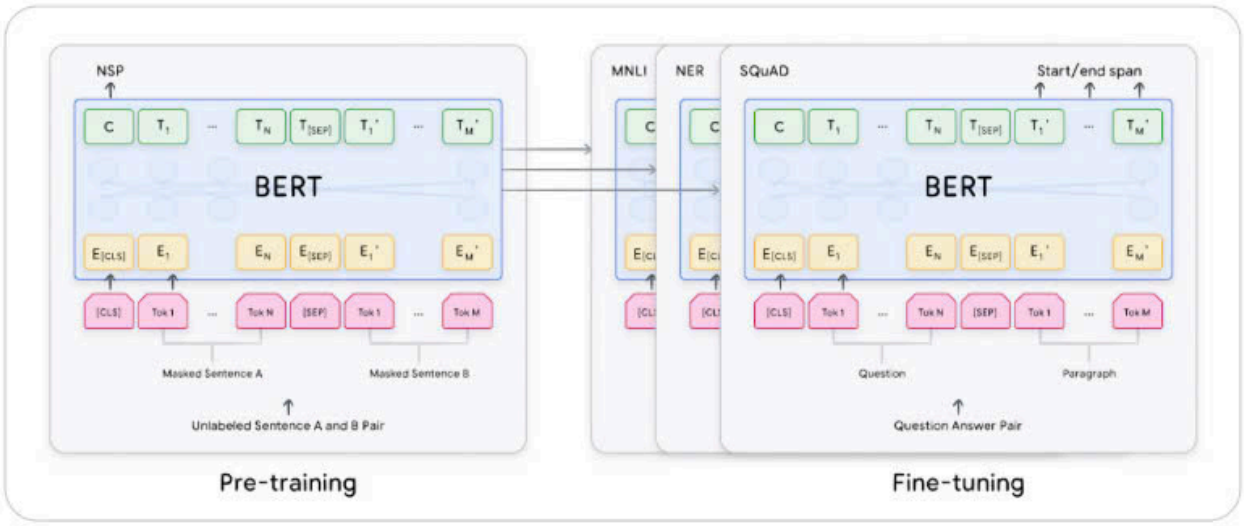


Figure 7. The BERT architecture

BERT đã trở thành mô hình nền tảng cho nhiều mô hình embedding khác, bao gồm **Sentence-BERT**, **SimCSE**, và **E5**. Trong khi đó, sự phát triển của các mô hình ngôn ngữ - đặc biệt là các mô hình ngôn ngữ lớn (LLMs) - không ngừng tiến xa hơn. Năm 2019, **T5** được đề xuất với quy mô lên đến 11 tỷ tham số. Năm 2022, **PaLM** được đề xuất với quy mô đáng kinh ngạc 540 tỷ tham số. Các mô hình như **Gemini** của Google, **GPT** của OpenAI, và **Llama** của Meta cũng liên tục được phát triển với tốc độ đáng kinh ngạc. Vui lòng tham khảo tài liệu nền tảng về các mô hình để biết thêm thông tin về một số LLM phổ biến.

Các mô hình embedding mới dựa trên các mô hình ngôn ngữ lớn cũng đã được đề xuất. Ví dụ, **GTR** và **Sentence-T5** cho thấy hiệu suất tốt hơn trên các nhiệm vụ truy xuất (retrieval) và đo độ tương đồng câu (sentence similarity) so với các mô hình thuộc họ BERT.

Một cách tiếp cận khác trong việc phát triển các mô hình embedding mới là tạo các embedding đa vector (*multi-vector embeddings*) thay vì chỉ một vector duy nhất, nhằm tăng cường khả năng biểu diễn của các mô hình. Các mô hình embedding thuộc nhóm này bao gồm **ColBERT** và **XTR**.

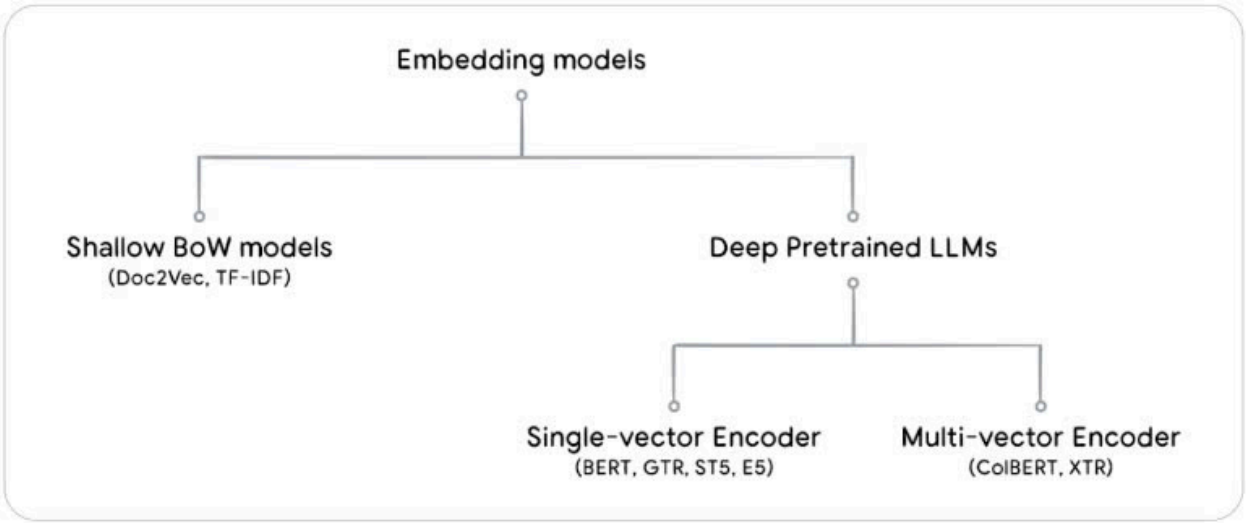


Figure 8. Hình minh họa sơ đồ phân loại của các mô hình embedding

Mặc dù các mô hình mạng nơ-ron sâu yêu cầu nhiều dữ liệu hơn và thời gian tính toán lâu hơn để huấn luyện, chúng có hiệu suất tốt hơn nhiều so với các mô hình sử dụng các phương pháp bag-of-words. Ví dụ, đối với cùng một từ, embedding sẽ khác nhau trong các ngữ cảnh khác nhau.

Snippet 4 minh họa cách sử dụng các mô hình embedding tài liệu đã được huấn luyện trước từ TensorFlow Hub (ví dụ: Sentence T5) và Vertex AI để huấn luyện các mô hình với Keras và TF datasets. Embedding văn bản từ Vertex Generative AI có thể được sử dụng với Vertex AI SDK, LangChain và Google BigQuery (**Snippet 5**) cho các embedding và quy trình công việc nâng cao.

A. Note: not all models on https://tfhub.dev/ can be commercially used. Please check the licenses of the models
and the training datasets and consult the legal team before commercial usage.
B. Note: not all models on

https://tfhub.dev/ can be commercially used. Please check the licenses of the models
and the training datasets and consult the legal team before commercial usage.

```
import vertexai
from vertexai.language_models import TextEmbeddingInput, TextEmbeddingMod
import numpy as np
import tensorflow as tf
import tensorflow_hub as hub

# 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

def LLM_embed(text):
    def embed_text(text):
        text_inp = TextEmbeddingInput(task_type="CLASSIFICATION", text=text.nur
        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, trainabl

# 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_
history = model.fit(train_data.shuffle(100).batch(8))
```

Snippet 4. Tạo và tích hợp text embeddings (Vertex, Tfhub) into keras text classification models

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

Snippet 5. Creating LLM based text embeddings in BigQuery for selected columns in a table

1.2 Image & multimodal embeddings

Cũng giống như văn bản, việc tạo các embedding cho hình ảnh và đa phương thức (multimodal) là hoàn toàn khả thi.

Embedding hình ảnh đơn phương thức (unimodal) có thể được tạo theo nhiều cách, một trong số đó là huấn luyện một mô hình CNN hoặc Vision Transformer trên một tác vụ phân loại hình ảnh quy mô lớn (ví dụ: ImageNet), sau đó sử dụng lớp áp chót làm embedding hình ảnh. Lớp này đã học được một số bản đồ đặc trưng phân biệt quan trọng cho nhiệm vụ huấn luyện. Nó chứa một tập hợp các bản đồ đặc trưng phân biệt cho nhiệm vụ hiện tại và có thể được mở rộng sang các nhiệm vụ khác.

Để có được embedding đa phương thức (multimodal), bạn kết hợp embedding văn bản và hình ảnh đơn phương thức, cùng với các mối quan hệ ngữ nghĩa của chúng được học thông qua một quá trình huấn luyện khác. Điều này mang lại một biểu diễn ngữ nghĩa cố định có kích thước trong cùng một không gian tiềm ẩn (latent space).

Snippet dưới đây (**Snippet 6**) có thể được sử dụng để tính embedding hình ảnh và đa phương thức cho hình ảnh và văn bản, và có thể được sử dụng trực tiếp với một mô hình Keras (tương tự như ví dụ về embedding văn bản).

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

# Fine-tunable layer for image embeddings which can be used for downstream K
embed = hub.KerasLayer("https://tfhub.dev/google/imagenet/efficientnet_v2_ima

class EmbeddingPredictionClient:
    """Wrapper around Prediction Service Client."""

    def __init__(self, project: str, location: str = "us-central1", api_regional_endpoint
        client_options = {"api_endpoint": api_regional_endpoint}
        self.client = aiplatform.gapic.PredictionServiceClient(client_options=client_o
        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 = encoded_conte

        # Make predictions using the multimodal embedding model
        instances = [instance]
        endpoint = f"projects/{self.project}/locations/{self.location}/publishers/goog
        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]['imageEmbedding']
            image_embedding = [v for v in image_emb_value]

        return EmbeddingResponse(text_embedding=text_embedding, image_embe

# Compute multimodal embeddings for text and images
client = EmbeddingPredictionClient(project="your_project_id")
client.get_embedding(text="sample_test", gs_image_path="gs://bucket_name/im
```

Snippet 6. Using Vertex API to create Multimodal embeddings Graph embeddings

1.3 Structured data embeddings

Có hai cách phổ biến để tạo embedding cho dữ liệu có cấu trúc, một cách mang tính tổng quát hơn và cách còn lại được tùy chỉnh đặc biệt cho các ứng dụng gợi ý.

Không giống như dữ liệu phi cấu trúc, nơi thường có sẵn các mô hình embedding được huấn luyện trước, chúng ta phải tự tạo mô hình embedding cho dữ liệu có cấu trúc vì nó sẽ đặc thù cho một ứng dụng cụ thể.

General structured data

Đối với một bảng dữ liệu có cấu trúc tổng quát, chúng ta có thể tạo embedding cho từng hàng. Điều này có thể được thực hiện bằng cách sử dụng các mô hình học máy trong danh mục giảm chiều dữ liệu, chẳng hạn như mô hình PCA.

Một trường hợp sử dụng của các embedding này là phát hiện bất thường. Ví dụ, chúng ta có thể tạo embedding để phát hiện bất thường bằng cách sử dụng các tập dữ liệu lớn chứa thông tin cảm biến có gán nhãn để xác định các hiện tượng bất thường. Một trường hợp sử dụng khác là cung cấp các embedding này cho các tác vụ học máy phía sau, chẳng hạn như phân loại.

So với việc sử dụng dữ liệu gốc có số chiều cao, việc sử dụng embedding để huấn luyện mô hình có giám sát yêu cầu ít dữ liệu hơn. Điều này đặc biệt quan trọng trong các trường hợp dữ liệu huấn luyện không đủ.

User/item structured data

Đầu vào không còn là một bảng dữ liệu có cấu trúc tổng quát như trên. Thay vào đó, đầu vào bao gồm dữ liệu người dùng, dữ liệu mục/sản phẩm, cùng với dữ liệu mô tả sự tương tác giữa người dùng và mục/sản phẩm, chẳng hạn như điểm đánh giá.

Danh mục này nhằm mục đích hỗ trợ các hệ thống gợi ý, vì nó ánh xạ hai tập dữ liệu (tập dữ liệu người dùng và tập dữ liệu mục/sản phẩm/v.v.) vào cùng một không gian embedding. Đối với các hệ thống gợi ý, chúng ta có thể tạo embedding từ dữ liệu có cấu trúc liên quan đến các thực thể khác nhau, chẳng hạn như sản phẩm, bài viết, v.v. Một lần nữa, chúng ta cần tự tạo mô hình embedding của mình. Đôi khi, phương pháp này có thể được kết hợp với các phương pháp embedding phi cấu trúc khi có hình ảnh hoặc mô tả văn bản.

1.4 Graph embeddings

Graph embedding là một kỹ thuật embedding khác, cho phép bạn biểu diễn không chỉ thông tin về một đối tượng cụ thể mà còn cả các đối tượng lân cận của nó (tức là biểu diễn đồ thị của chúng). Lấy ví dụ về một mạng xã hội, nơi mỗi người là một

nút (node), và các kết nối giữa mọi người được định nghĩa là các cạnh (edge). Sử dụng graph embedding, bạn có thể mô hình hóa mỗi nút dưới dạng một embedding, sao cho embedding này không chỉ nắm bắt thông tin ngữ nghĩa về bản thân đối tượng mà còn cả các mối quan hệ và liên kết của nó, từ đó làm phong phú thêm embedding.

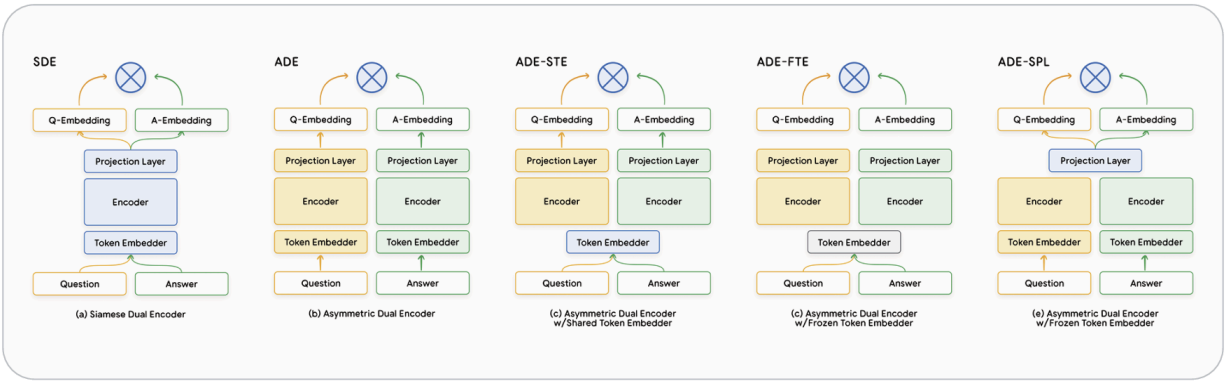
Ví dụ, nếu hai nút được kết nối bởi một cạnh, các vector cho các nút đó sẽ có sự tương đồng. Dựa trên điều này, bạn có thể dự đoán người nào giống nhất với người khác và gợi ý các kết nối mới. Graph embedding cũng có thể được sử dụng cho nhiều nhiệm vụ khác nhau, bao gồm phân loại nút (node classification), phân loại đồ thị (graph classification), dự đoán liên kết (link prediction), phân cụm (clustering), tìm kiếm, hệ thống gợi ý, và nhiều ứng dụng khác.

Các thuật toán phổ biến cho graph embedding bao gồm **DeepWalk**, **Node2vec**, **LINE**, và **GraphSAGE**.

2. Training Embeddings

Các mô hình embedding hiện tại thường sử dụng kiến trúc dual encoder (hai tháp). Ví dụ, đối với mô hình embedding văn bản được sử dụng trong hỏi-đáp, một tháp được dùng để mã hóa các truy vấn, và tháp còn lại được dùng để mã hóa các tài liệu. Đối với mô hình embedding hình ảnh và văn bản, một tháp được sử dụng để mã hóa hình ảnh, trong khi tháp còn lại được dùng để mã hóa văn bản.

Mô hình có thể có các kiến trúc phụ khác nhau, tùy thuộc vào cách các thành phần của mô hình được chia sẻ giữa hai tháp. Hình dưới đây minh họa một số kiến trúc của các dual encoders.



Hàm mất mát (loss) được sử dụng trong huấn luyện các mô hình embedding thường là một biến thể của **contrastive loss**, với đầu vào là một tuple bao gồm `<inputs, positive targets, [optional] negative targets>`. Huấn luyện với contrastive loss sẽ kéo các ví dụ tích cực lại gần nhau và đẩy các ví dụ tiêu cực ra xa.

Tương tự như quá trình huấn luyện mô hình nền tảng (foundation models), việc huấn luyện một mô hình embedding từ đầu thường bao gồm hai giai đoạn: **huấn luyện trước** (học không giám sát) và **tinh chỉnh** (học có giám sát). Hiện nay, các mô hình embedding thường được khởi tạo trực tiếp từ các mô hình nền tảng như **BERT**, **T5**, **GPT**, **Gemini**, và **CoCa**. Bạn có thể sử dụng các mô hình nền tảng này để tận dụng lượng lớn kiến thức đã được học từ quá trình huấn luyện trước quy mô lớn.

Quá trình tinh chỉnh (fine-tuning) của các mô hình embedding có thể bao gồm một hoặc nhiều giai đoạn. Các tập dữ liệu tinh chỉnh có thể được tạo theo nhiều cách khác nhau, bao gồm: gán nhãn thủ công, tạo tập dữ liệu tổng hợp, trích xuất từ mô hình (model distillation), và khai thác các ví dụ tiêu cực khó (hard negative mining).

Để sử dụng embedding cho các tác vụ phía sau như phân loại hoặc nhận diện thực thể có tên (named entity recognition), các lớp bổ sung (ví dụ: lớp phân loại softmax) có thể được thêm vào phía trên các mô hình embedding. Mô hình embedding có thể được giữ nguyên (đóng băng) - đặc biệt khi tập dữ liệu huấn luyện nhỏ, được huấn luyện từ đầu, hoặc được tinh chỉnh cùng với các tác vụ phía sau.

Vertex AI cung cấp khả năng tùy chỉnh các mô hình embedding văn bản trên Vertex AI. Người dùng cũng có thể chọn tinh chỉnh trực tiếp các mô hình này. Xem tài liệu để biết ví dụ về tinh chỉnh mô hình **BERT** bằng **TensorFlow Model Garden**. Ngoài ra, bạn cũng có thể tải trực tiếp các mô hình embedding từ **tfhub** và thực hiện tinh chỉnh trên các mô hình này. **Snippet 7** minh họa cách xây dựng một bộ phân loại dựa trên các mô hình từ **tfhub**.

```
# Can switch the embedding to different embeddings from different modalities or
# 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
```

Snippet 7. Creating a Keras model using trainable tfhub layer

Đến nay, bạn đã tìm hiểu về các loại embedding khác nhau, các kỹ thuật và phương pháp tốt nhất để huấn luyện chúng cho các kiểu dữ liệu khác nhau, cũng như một số ứng dụng của chúng. Phần tiếp theo sẽ thảo luận về cách lưu trữ và tìm kiếm các embedding đã được tạo một cách nhanh chóng và mở rộng, phù hợp với khối lượng công việc trong môi trường sản xuất.

C. Vector search

Tìm kiếm toàn văn bằng từ khóa đã trở thành nền tảng của các hệ thống CNTT hiện đại trong nhiều năm qua. Các công cụ tìm kiếm toàn văn và cơ sở dữ liệu (quan hệ và phi quan hệ) thường dựa vào việc khớp chính xác từ khóa. Ví dụ, nếu bạn tìm kiếm từ "cappuccino," công cụ tìm kiếm hoặc cơ sở dữ liệu sẽ trả về tất cả các tài liệu đề cập chính xác đến truy vấn trong thẻ hoặc mô tả văn bản. Tuy nhiên, nếu từ khóa bị viết sai chính tả hoặc được mô tả bằng văn bản khác, tìm kiếm từ khóa truyền thống sẽ trả về kết quả không chính xác hoặc không có kết quả nào.

Có một số phương pháp truyền thống cho phép khắc phục lỗi chính tả hoặc các lỗi đánh máy khác. Tuy nhiên, các phương pháp này vẫn không thể tìm thấy các kết quả có ý nghĩa ngữ nghĩa gần nhất với truy vấn. Đây chính là lúc tìm kiếm vector

thể hiện sức mạnh: nó sử dụng vector hoặc biểu diễn ngữ nghĩa được nhúng (embedded semantic representation) của tài liệu.

Tìm kiếm vector cho phép bạn vượt xa việc tìm kiếm chính xác các từ khóa và tìm kiếm ý nghĩa trên nhiều loại dữ liệu khác nhau. Điều này mang lại kết quả tinh tế và chính xác hơn. Sau khi bạn có một hàm có thể tính toán embedding của các đối tượng khác nhau, bạn tính toán embedding của các đối tượng cần tìm kiếm và lưu trữ embedding này trong cơ sở dữ liệu.

Tiếp theo, bạn nhúng (embed) truy vấn đầu vào vào cùng không gian vector như các đối tượng đó. Sau đó, bạn phải tìm các kết quả phù hợp nhất với truy vấn. Quá trình này tương tự như việc tìm kiếm các kết quả "tương đồng" nhất trong toàn bộ tập hợp vector có thể tìm kiếm. Độ tương đồng giữa các vector có thể được tính toán bằng một số phép đo như **khoảng cách Euclid (euclidean distance)**, **độ tương đồng cosine (cosine similarity)** hoặc **tích vô hướng (dot product)**.

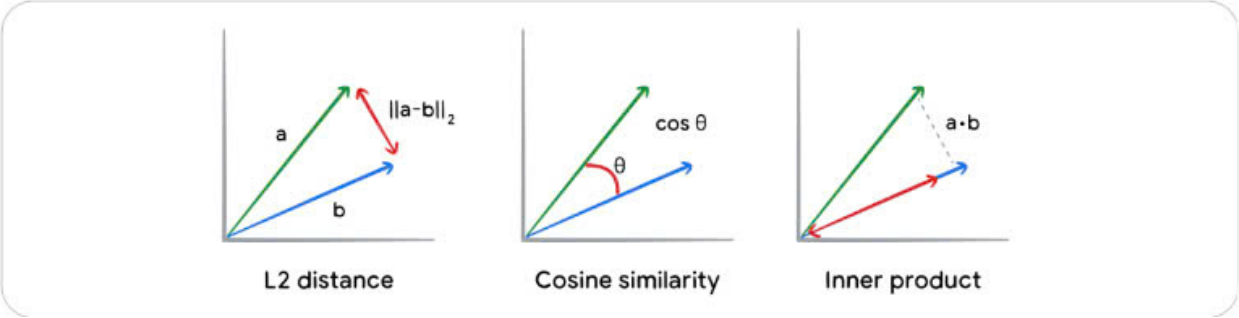


Figure 10. Visualization of how different metrics compute vector similarity

Khoảng cách Euclid (tức là khoảng cách L2) là một phép đo hình học về khoảng cách giữa hai điểm trong không gian vector. Phương pháp này hoạt động tốt với dữ liệu có số chiều thấp. **Độ tương đồng cosine** là một phép đo về góc giữa hai vector. Trong khi đó, **tích vô hướng (inner/dot product)** là phép chiếu của một vector lên vector khác. Chúng tương đương nhau khi chuẩn của các vector là 1. Phương pháp này dường như hoạt động tốt hơn với dữ liệu có số chiều cao.

Cơ sở dữ liệu vector không chỉ lưu trữ mà còn giúp quản lý và vận hành sự phức tạp của tìm kiếm vector ở quy mô lớn, đồng thời đáp ứng các nhu cầu cơ bản của cơ sở dữ liệu.

Important vector search algorithms

Cách đơn giản nhất để tìm kết quả tương đồng nhất là thực hiện tìm kiếm tuyến tính truyền thống bằng cách so sánh vector truy vấn với từng vector tài liệu và trả về vector có độ tương đồng cao nhất. Tuy nhiên, thời gian chạy của phương pháp này tỷ lệ tuyến tính $(O(N)$ với số lượng tài liệu hoặc đối tượng cần tìm kiếm. Phương pháp này không đủ nhanh đối với hầu hết các trường hợp sử dụng liên quan đến hàng triệu tài liệu hoặc nhiều hơn.

Sử dụng **tìm kiếm lân cận gần đúng (Approximate Nearest Neighbor - ANN)** cho mục đích này sẽ thực tế hơn. ANN là một kỹ thuật để tìm các điểm gần nhất với một điểm cho trước trong tập dữ liệu với một biên độ sai số nhỏ - nhưng mang lại sự cải thiện đáng kể về hiệu suất.

Có nhiều phương pháp ANN với các thỏa hiệp khác nhau về quy mô, thời gian lập chỉ mục, hiệu suất, tính đơn giản, và hơn thế nữa. Những phương pháp này thường sử dụng một hoặc nhiều kỹ thuật như: **lượng tử hóa (quantization)**, **băm (hashing)**, **phân cụm (clustering)** và **cây (trees)**, cùng các phương pháp khác. Một số phương pháp phổ biến nhất sẽ được thảo luận dưới đây.

1.1 Locality sensitive hashing & trees

Locality sensitive hashing (LSH) 28 là một kỹ thuật để tìm các mục tương tự trong một tập dữ liệu lớn. Nó thực hiện điều này bằng cách tạo ra một hoặc nhiều hàm băm mà với xác suất cao sẽ ánh xạ các mục tương tự vào cùng một xô băm (hash bucket). Điều này có nghĩa là bạn có thể nhanh chóng tìm thấy tất cả các mục tương tự với một mục đã cho bằng cách chỉ xem xét các mục ứng viên trong cùng xô băm (hoặc các xô liền kề) và thực hiện tìm kiếm tuyến tính giữa các cặp ứng viên đó. Điều này cho phép tra cứu nhanh hơn đáng kể trong một bán kính cụ thể. Số lượng hàm băm/bảng băm và xô băm quyết định sự đánh đổi giữa độ hồi tưởng (recall)/tốc độ tìm kiếm, cũng như tỷ lệ dương tính giả/dương tính thật. Có quá nhiều hàm băm có thể khiến các mục tương tự vào các xô khác nhau, trong khi quá ít có thể dẫn đến quá nhiều mục bị băm vào cùng một xô và số lượng tìm kiếm tuyến tính tăng lên.

Một cách nghĩ trực quan khác về LSH là nhóm các nơi cư trú theo mã bưu điện hoặc tên khu phố. Sau đó, dựa trên nơi mà ai đó muốn chuyển đến, bạn chỉ xem xét các nơi cư trú trong khu vực đó và tìm kiếm kết quả phù hợp nhất.

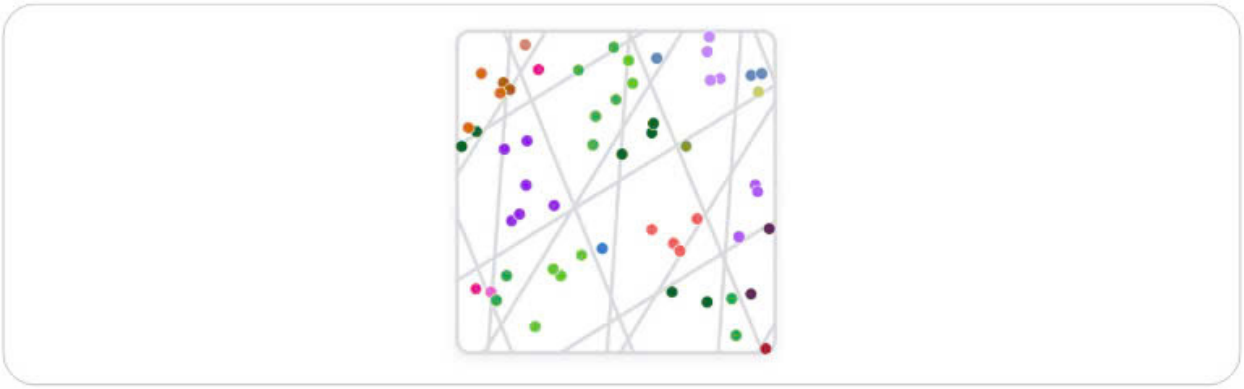


Figure 11. Visualization of how LSH uses random hyperplanes to partition the vector space

Các thuật toán dựa trên cây hoạt động tương tự. Ví dụ, phương pháp **Kd-tree** hoạt động bằng cách tạo các ranh giới quyết định (decision boundaries) bằng cách tính giá trị trung vị của các giá trị trong chiều đầu tiên, sau đó là của chiều thứ hai, và cứ tiếp tục như vậy. Phương pháp này khá giống với cây quyết định (decision tree).

Tuy nhiên, phương pháp này có thể không hiệu quả nếu các vector tìm kiếm có số chiều cao. Trong trường hợp đó, thuật toán **Ball-tree** sẽ phù hợp hơn. Về chức năng, nó tương tự như Kd-tree, nhưng thay vì dựa trên trung vị theo từng chiều, nó tạo các xô (bucket) dựa trên khoảng cách xuyên tâm (radial distance) của các điểm dữ liệu từ tâm.

Dưới đây là một ví dụ về cách triển khai ba phương pháp này:

```
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
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(embe
naive_distances, naive_indices = nbrs.kneighbors(np.expand_dims(embedded_q
#algorithm- ball_tree due to high dimensional vectors or kd_tree otherwise
nbrs = NearestNeighbors(n_neighbors=n_neighbors, algorithm='ball_tree').fit(em
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_neig
#output for all 3 indices = [0, 1] , distances [0.66840428, 0.71048843] for the firs
#ANN retrieved the same ranking of items as brute force in a much scalable man
```

Snippet 8. Using scikit-learn29 and lshashing30 for ANN with LSH, KD/Ball-tree and linear search

Các phương pháp dựa trên băm (hashing) và cây (tree-based) cũng có thể được kết hợp và mở rộng để đạt được sự cân bằng tối ưu giữa độ hồi tưởng (recall) và độ trễ (latency) cho các thuật toán tìm kiếm. **FAISS** với **HNSW** và **ScaNN** là những ví dụ điển hình.

1.2 Hierarchical navigable small worlds

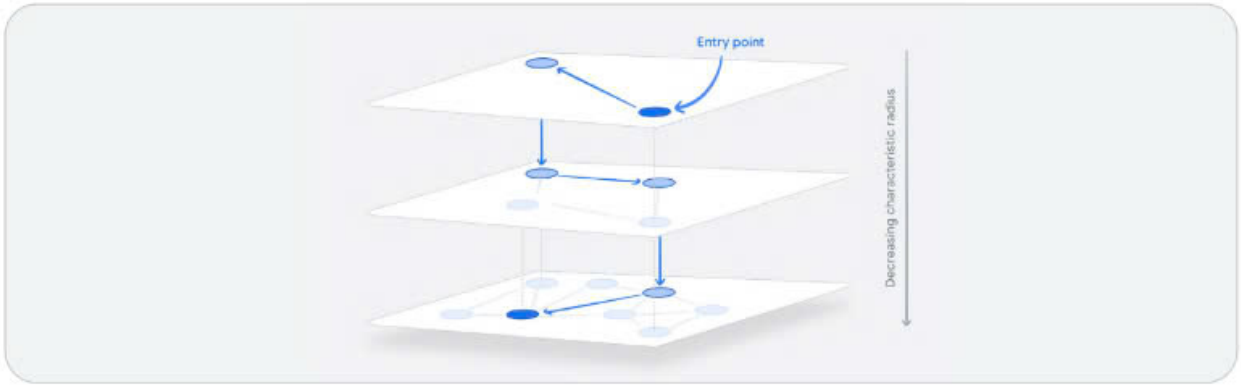


Figure 12. Diagram showing how HNSW 'zooms in' to perform ANN

Một trong các triển khai của **FAISS** (Facebook AI Similarity Search) sử dụng khái niệm **Hierarchical Navigable Small World (HNSW)** để thực hiện tìm kiếm độ tương đồng vector với thời gian chạy dưới tuyến tính $O(\log n)$ và mức độ chính xác cao. **HNSW** là một đồ thị gần đúng có cấu trúc phân cấp, trong đó các liên kết trong đồ thị được phân bổ trên các lớp khác nhau.

Lớp trên cùng có các liên kết dài nhất, trong khi lớp dưới cùng có các liên kết ngắn nhất. Như minh họa trong Hình 9, quá trình tìm kiếm bắt đầu từ lớp trên cùng nhất, nơi thuật toán tham lam (greedy) duyệt qua đồ thị để tìm đỉnh có ý nghĩa ngữ nghĩa gần nhất với truy vấn. Khi tìm thấy giá trị cực tiểu cục bộ (local minimum) cho lớp đó, thuật toán chuyển sang đồ thị của đỉnh gần nhất trên lớp bên dưới.

Quá trình này tiếp tục lặp đi lặp lại cho đến khi tìm thấy cực tiểu cục bộ cho lớp dưới cùng, với thuật toán theo dõi tất cả các đỉnh đã duyệt để trả về K đỉnh gần nhất (*K-nearest neighbors*). Thuật toán này có thể được tăng cường thêm bằng cách sử dụng kỹ thuật lượng tử hóa (quantization) và lập chỉ mục vector (vector indexing) để tăng tốc độ và hiệu quả bộ nhớ.

```
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 Snipp
#execute the ANN search
index.search(np.expand_dims(embedded_query, axis=0), k=2)
```

Snippet 9. Indexing and executing ANN search with the FAISS library using HNSW

1.3 ScaNN

Google đã phát triển phương pháp **Scalable Approximate Nearest Neighbor (ScaNN)**, được sử dụng trong nhiều sản phẩm và dịch vụ của hãng. Phương pháp này cũng được cung cấp cho tất cả khách hàng của Google Cloud thông qua **Vertex AI Vector Search**. Dưới đây là cách ScaNN sử dụng các bước khác nhau để thực hiện tìm kiếm vector hiệu quả, với mỗi bước có tập hợp các tham số riêng.

Bước đầu tiên là bước phân vùng tùy chọn trong quá trình huấn luyện: ScaNN sử dụng một trong nhiều thuật toán sẵn có để phân vùng kho vector thành các phân vùng/cụm logic, trong đó các vector có liên quan ngữ nghĩa được nhóm lại với nhau. Bước phân vùng này là tùy chọn đối với các tập dữ liệu nhỏ. Tuy nhiên, đối với các tập dữ liệu lớn với hơn 100.000 vector embedding, bước phân vùng là rất quan trọng vì bằng cách cắt giảm không gian tìm kiếm, nó giảm đáng kể kích thước không gian tìm kiếm và tăng tốc đáng kể quá trình truy vấn. Việc cắt giảm không gian tìm kiếm được cấu hình thông qua số lượng phân vùng và số phân

vùng cần tìm kiếm. Số lượng phân vùng lớn hơn mang lại độ hồi tưởng (recall) tốt hơn nhưng thời gian tạo phân vùng dài hơn. Một phương pháp ước lượng tốt là đặt số lượng phân vùng bằng căn bậc hai của số lượng vector.

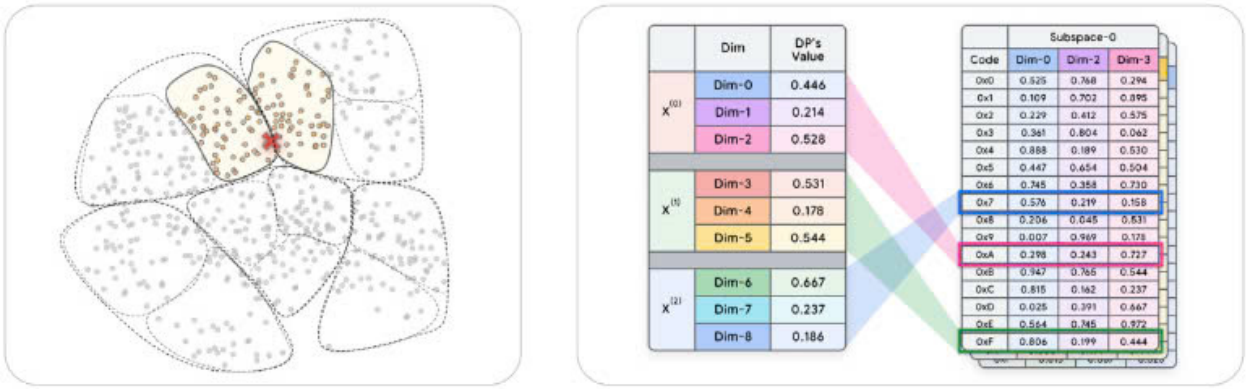


Figure 13. Search space partitioning & pruning(left) & Approximate scoring (right)

Khi truy vấn, ScaNN sử dụng phép đo khoảng cách được người dùng chỉ định để chọn số lượng phân vùng hàng đầu (giá trị do người dùng chỉ định) và sau đó thực hiện bước tính điểm. Trong bước này, ScaNN so sánh truy vấn với tất cả các điểm trong các phân vùng hàng đầu và chọn ra **K'** điểm hàng đầu. Việc tính toán khoảng cách này có thể được cấu hình là khoảng cách chính xác hoặc khoảng cách xấp xỉ. Phép tính khoảng cách xấp xỉ tận dụng kỹ thuật lượng tử hóa sản phẩm tiêu chuẩn (standard product quantization) hoặc lượng tử hóa phi đẳng hướng (anisotropic quantization). Phương pháp thứ hai là một kỹ thuật cụ thể được ScaNN sử dụng, mang lại sự cân bằng tốt hơn giữa tốc độ và độ chính xác.

Cuối cùng, ở bước cuối, người dùng có thể tùy chọn tính điểm lại (rescore) chính xác hơn cho **K** kết quả hàng đầu được chỉ định bởi người dùng. Điều này mang lại sự cân bằng tốc độ/độ chính xác hàng đầu trong ngành mà ScaNN được biết đến, như được minh họa trong Hình 14. **Snippet 10** cung cấp một ví dụ mã.

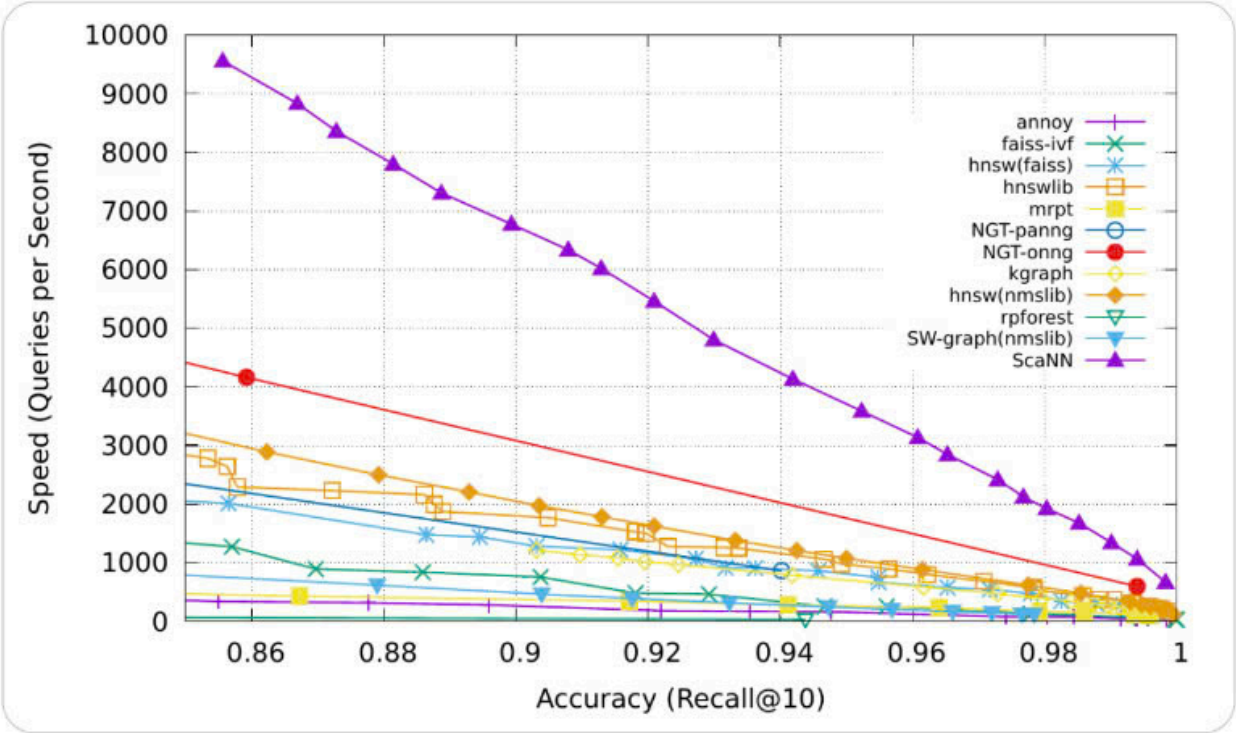


Figure 14. Accuracy/speed tradeoffs for various SOTA ANN search algorithms

```
import tensorflow as tf
import tensorflow_recommenders as tfrs
from vertexai.language_models import TextEmbeddingModel, TextEmbeddingInp
# Embed documents & query(from snip 9.) and convert them to tensors and tf.da
embedded_query = tf.constant((LM_embed(query, "RETRIEVAL_QUERY")))
embedded_docs = [LM_embed(doc, "RETIREVAL_DOCUMENT") for doc in searc
embedded_docs = tf.data.Dataset.from_tensor_slices(embedded_docs).enumera
# Build index from tensorflow dataset and execute ANN search based on dot pro
scann = tfrs.layers.factorized_top_k.ScaNN(
distance_measure= 'dot_product',
num_leaves = 4, #increase for higher number of partitions / latency for increasec
num_leaves_to_search= 2) # increase for higher recall but increased latency
scann = scann.index_from_dataset(embedded_docs)
scann(embedded_query, k=2)
```

Snippet 10. Using Tensorflow Recommenders34 to perform ANN search using the ScaNN algorithm

Trong tài liệu này, chúng ta đã tìm hiểu cả các thuật toán tìm kiếm ANN hiện đại (State-of-the-Art - SOTA) và truyền thống, bao gồm **ScaNN**, **FAISS**, **LSH**, **KD-Tree**, và **Ball-tree**, cũng như xem xét sự cân bằng tốc độ/độ chính xác tuyệt vời mà chúng mang lại. Tuy nhiên, để sử dụng các thuật toán này, chúng cần được triển khai theo cách có thể mở rộng, an toàn và sẵn sàng cho môi trường sản xuất. Để làm được điều đó, chúng ta cần các **cơ sở dữ liệu vector (vector databases)**.

D. Vector databases

Vector embeddings thể hiện ý nghĩa ngữ nghĩa của dữ liệu, trong khi các thuật toán tìm kiếm vector cung cấp phương tiện để truy vấn chúng một cách hiệu quả. Lịch sử cho thấy các cơ sở dữ liệu truyền thống thiếu khả năng kết hợp ý nghĩa ngữ nghĩa và truy vấn hiệu quả để lưu trữ, truy xuất và tìm kiếm các embedding liên quan nhất theo cách an toàn, có thể mở rộng và linh hoạt, phục vụ cho phân tích phức tạp và các ứng dụng doanh nghiệp theo thời gian thực. Đây chính là lý do ra đời của các **cơ sở dữ liệu vector (vector databases)**, được xây dựng từ đầu để quản lý các embedding này trong các kịch bản sản xuất.

Với sự phổ biến gần đây của AI tạo sinh (Generative AI), ngày càng có nhiều cơ sở dữ liệu truyền thống bắt đầu tích hợp chức năng tìm kiếm vector bên cạnh chức năng tìm kiếm truyền thống (**tìm kiếm lai - hybrid search**). Hãy cùng xem quy trình làm việc của một cơ sở dữ liệu vector đơn giản với khả năng tìm kiếm lai.

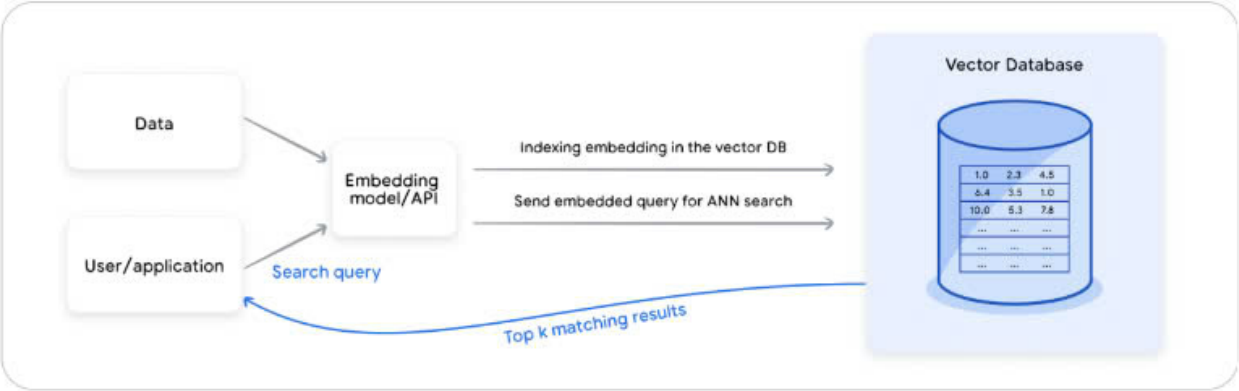


Figure 15. Populating and querying vector databases

Mỗi cơ sở dữ liệu vector có cách triển khai khác nhau, nhưng quy trình chung được minh họa trong Hình 15:

1. Sử dụng một mô hình embedding được huấn luyện phù hợp để nhúng các điểm dữ liệu liên quan thành các vector có kích thước cố định.
2. Các vector sau đó được bổ sung với siêu dữ liệu (metadata) và thông tin bổ sung phù hợp (chẳng hạn như thẻ) và được lập chỉ mục bằng thuật toán đã chỉ định để tìm kiếm hiệu quả.
3. Một truy vấn đầu vào được nhúng bằng cùng một mô hình, sau đó được sử dụng để truy vấn và trả về các mục có ý nghĩa ngữ nghĩa gần nhất cùng với nội dung/siêu dữ liệu liên quan chưa được nhúng. Một số cơ sở dữ liệu có thể cung cấp khả năng lưu đệm (caching) và tiền lọc (pre-filtering) dựa trên thẻ, cũng như hậu lọc (post-filtering) để sắp xếp lại thứ hạng bằng một mô hình chính xác hơn nhằm nâng cao tốc độ và hiệu suất truy vấn.

Hiện nay, có khá nhiều cơ sở dữ liệu vector sẵn có, mỗi loại được tùy chỉnh để đáp ứng các nhu cầu kinh doanh và cân nhắc khác nhau. Một số ví dụ tiêu biểu về các cơ sở dữ liệu vector được quản lý thương mại bao gồm: **Vertex AI Vector Search**, **AlloyDB** & **Cloud SQL Postgres ElasticSearch** của Google Cloud, và **Pinecone**.

- **Vertex AI Vector Search** là một cơ sở dữ liệu vector được Google xây dựng, sử dụng thuật toán **ScaNN** để tìm kiếm vector nhanh chóng, đồng thời vẫn duy trì các đảm bảo về bảo mật và truy cập của Google Cloud.
- **AlloyDB** & **Cloud SQL Postgres** hỗ trợ tìm kiếm vector thông qua tiện ích mở rộng OSS **pgvector**, cho phép các truy vấn SQL kết hợp tìm kiếm ANN với các điều kiện truyền thống và ngữ nghĩa giao dịch thông thường cho chỉ mục tìm kiếm ANN. **AlloyDB** cũng có một tiện ích mở rộng chỉ mục ScaNN, là một triển khai nguyên bản của ScaNN và tương thích với **pgvector**.
- Tương tự, nhiều cơ sở dữ liệu truyền thống khác cũng đã bắt đầu bổ sung các plugin để kích hoạt tìm kiếm vector.

Pinecone và **Weaviate** tận dụng **HNSW** cho tìm kiếm vector nhanh, đồng thời cung cấp khả năng lọc dữ liệu bằng tìm kiếm truyền thống. Trong số các đối thủ nguồn mở: **Weaviate** và **ChromaDB** cung cấp đầy đủ chức năng khi triển khai và có thể được thử nghiệm trong bộ nhớ (in-memory) trong giai đoạn tạo nguyên mẫu.

Operational considerations

Cơ sở dữ liệu vector (Vector Databases) đóng vai trò quan trọng trong việc giải quyết hầu hết các thách thức kỹ thuật nảy sinh khi lưu trữ và truy vấn các embedding ở quy mô lớn. Một số thách thức này là đặc thù của bản chất vector stores, trong khi những thách thức khác lại tương đồng với cơ sở dữ liệu truyền thống. Các thách thức bao gồm khả năng mở rộng ngang và dọc, tính sẵn sàng, tính nhất quán dữ liệu, cập nhật thời gian thực, sao lưu, kiểm soát truy cập, tuân thủ quy định, và nhiều yếu tố khác.

Tuy nhiên, cũng có nhiều thách thức và cân nhắc khác cần được xem xét khi sử dụng embedding và vector stores:

Thứ nhất, **embedding**, không giống như nội dung truyền thống, có thể thay đổi theo thời gian. Điều này có nghĩa là cùng một văn bản, hình ảnh, video hoặc nội dung khác có thể (và nên) được nhúng lại bằng các mô hình embedding khác nhau để tối ưu hóa hiệu suất cho các ứng dụng phía sau. Điều này đặc biệt đúng với embedding của các mô hình có giám sát khi mô hình được huấn luyện lại để xử lý các sự thay đổi (drift) hoặc mục tiêu thay đổi. Tương tự, điều này cũng áp dụng với các mô hình không giám sát khi chúng được cập nhật lên phiên bản mới hơn. Tuy nhiên, việc thường xuyên cập nhật các embedding - đặc biệt là những embedding được huấn luyện trên lượng dữ liệu lớn - có thể rất tốn kém. Do đó, cần đạt được sự cân bằng. Điều này đòi hỏi một quy trình tự động được xác định rõ ràng để lưu trữ, quản lý, và có thể xóa bỏ các embedding khỏi cơ sở dữ liệu vector dựa trên ngân sách cho phép.

Thứ hai, mặc dù embedding rất tốt trong việc biểu diễn thông tin ngữ nghĩa, đôi khi chúng lại không tối ưu trong việc biểu diễn thông tin ngữ pháp hoặc cú pháp. Điều này đặc biệt đúng với các từ hoặc ID đặc thù của một lĩnh vực. Các giá trị này có thể bị thiếu hoặc không được đại diện đầy đủ trong dữ liệu mà các mô hình embedding đã được huấn luyện. Ví dụ, nếu một người dùng nhập một truy vấn chứa một ID cụ thể cùng với nhiều văn bản, mô hình có thể tìm thấy các lân cận ngữ nghĩa tương tự phù hợp với ý nghĩa của văn bản nhưng không khớp với ID - yếu tố quan trọng nhất trong ngữ cảnh này. Bạn có thể giải quyết vấn đề này bằng cách kết hợp tìm kiếm toàn văn (full-text search) để tiền lọc (pre-filter) hoặc hậu lọc (post-filter) không gian tìm kiếm trước khi chuyển nó đến mô-đun tìm kiếm ngữ nghĩa.

Một điểm quan trọng khác cần xem xét là tùy thuộc vào tính chất của khối lượng công việc mà truy vấn ngữ nghĩa diễn ra, việc dựa vào các cơ sở dữ liệu vector

khác nhau có thể mang lại hiệu quả tốt hơn. Ví dụ, đối với khối lượng công việc OLTP (xử lý giao dịch trực tuyến) yêu cầu các thao tác đọc/ghi thường xuyên, một cơ sở dữ liệu hoạt động như **Postgres** hoặc **CloudSQL** là lựa chọn tốt nhất. Đối với khối lượng công việc OLAP (phân tích trực tuyến) quy mô lớn và các trường hợp sử dụng theo lô, sử dụng tìm kiếm vector của **BigQuery** sẽ phù hợp hơn.

Tóm lại, có nhiều yếu tố cần được xem xét khi chọn một cơ sở dữ liệu vector. Những yếu tố này bao gồm kích thước và loại tập dữ liệu của bạn (một số phù hợp với dữ liệu thưa, số khác phù hợp với dữ liệu dày đặc), nhu cầu kinh doanh, tính chất của khối lượng công việc, ngân sách, đảm bảo bảo mật và quyền riêng tư, nhu cầu tìm kiếm ngữ nghĩa và cú pháp, cũng như các hệ thống cơ sở dữ liệu đang được sử dụng.

Trong phần này, chúng ta đã tìm hiểu các phương pháp tìm kiếm ANN khác nhau, cũng như nhu cầu và lợi ích của cơ sở dữ liệu vector. Phần tiếp theo sẽ minh họa một ví dụ về việc sử dụng **Vector AI Vector Search** để tìm kiếm ngữ nghĩa.

E. Applications

Các mô hình embedding là một trong những mô hình học máy cơ bản, hỗ trợ nhiều ứng dụng khác nhau. Dưới đây, chúng tôi tóm tắt một số ứng dụng phổ biến trong bảng sau.

Task	Mô tả
Retrieval	Cho một truy vấn và một tập hợp các đối tượng (ví dụ: tài liệu, hình ảnh, video), truy xuất các đối tượng liên quan nhất. Dựa trên định nghĩa về các đối tượng liên quan, các nhiệm vụ con bao gồm: hỏi-đáp và hệ thống gợi ý.
Semantic text similarity	Xác định xem hai câu có cùng ý nghĩa ngữ nghĩa hay không. Các nhiệm vụ con bao gồm: diễn đạt lại (paraphrasing), phát hiện trùng lặp (duplicate detection), và khai thác bitext (bitext mining).
Classification	Phân loại các đối tượng vào các danh mục có thể có. Dựa trên số lượng nhãn, các nhiệm vụ con bao gồm: phân loại nhị phân (binary classification), phân loại đa lớp (multi-class classification), và phân loại nhiều nhãn (multilabel classification).
Clustering	Nhóm các đối tượng lại với nhau.

Reranking	Sắp xếp lại thứ hạng một tập hợp các đối tượng dựa trên một truy vấn nhất định.

Embedding kết hợp với các kho vector hỗ trợ ANN có thể là những công cụ mạnh mẽ được sử dụng cho nhiều ứng dụng khác nhau. Các ứng dụng này bao gồm **Retrieval Augmented Generation (RAG)** cho các mô hình ngôn ngữ lớn (LLMs), tìm kiếm, hệ thống gợi ý, phát hiện bất thường, phân loại với ít ví dụ (few-shot classification), và nhiều ứng dụng khác.

Đối với các bài toán xếp hạng như tìm kiếm và hệ thống gợi ý, embedding thường được sử dụng ở giai đoạn đầu tiên của quy trình. Chúng truy xuất các ứng viên tiềm năng có sự tương đồng ngữ nghĩa, từ đó cải thiện mức độ liên quan của kết quả tìm kiếm. Vì lượng thông tin cần xử lý có thể rất lớn (trong một số trường hợp có thể lên đến hàng triệu hoặc hàng tỷ), các kỹ thuật ANN như **ScaNN** hỗ trợ đáng kể trong việc thu hẹp không gian tìm kiếm một cách có thể mở rộng.

Hãy cùng xem một ứng dụng kết hợp cả LLMs và RAG để hỗ trợ trả lời câu hỏi.

Q & A with sources (retrieval augmented generation)

Retrieval Augmented Generation (RAG) cho hỏi-đáp (Q&A) là một kỹ thuật kết hợp những ưu điểm tốt nhất của truy xuất thông tin (retrieval) và sinh nội dung (generation). Đầu tiên, nó truy xuất các tài liệu liên quan từ một cơ sở tri thức, sau đó sử dụng mở rộng gợi ý (*prompt expansion*) để tạo ra câu trả lời từ những tài liệu đó.

Mở rộng gợi ý là một kỹ thuật rất mạnh mẽ khi được kết hợp với tìm kiếm trong cơ sở dữ liệu. Với mở rộng gợi ý, mô hình truy xuất thông tin liên quan từ cơ sở dữ liệu (thường sử dụng kết hợp tìm kiếm ngữ nghĩa và các quy tắc nghiệp vụ), sau đó bổ sung thông tin này vào gợi ý ban đầu. Mô hình sử dụng gợi ý đã được bổ sung này để tạo ra nội dung thú vị, chính xác và giàu thông tin hơn nhiều so với việc chỉ sử dụng truy xuất hoặc sinh nội dung riêng lẻ.

RAG có thể giúp giải quyết một vấn đề phổ biến với các mô hình ngôn ngữ lớn (LLMs): xu hướng "ảo giác" (*hallucinate*) và tạo ra các phản hồi nghe có vẻ hợp lý nhưng lại không chính xác về mặt thực tế. Mặc dù RAG có thể giảm thiểu hiện tượng ảo giác này, nhưng không thể loại bỏ hoàn toàn. Để giảm thiểu vấn đề này hơn nữa, một giải pháp là trả về cả các nguồn từ quá trình truy xuất và thực hiện

kiểm tra tính nhất quán nhanh chóng, hoặc bởi con người, hoặc bởi một LLM khác. Điều này đảm bảo rằng phản hồi của LLM phù hợp với các nguồn ngữ nghĩa liên quan.

Hãy cùng xem một ví dụ (**Snippet 11**) về RAG với các nguồn, có thể được triển khai mở rộng bằng cách sử dụng **Vertex AI LLM Text Embeddings** và **Vertex AI Vector Search**, kết hợp với các thư viện như **LangChain**.

```
# Before you start run this command:
# pip install --upgrade --user --quiet google-cloud-aiplatform langchain_google_
# 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 textembedding-gecko@004 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

# Initialize AI Platform
aiplatform.init(project=PROJECT_ID, location=REGION, staging_bucket=BUCKET_

# Initialize embedding model
embedding_model = VertexAIEmbeddings(model_name="textembedding-gecko(

# NOTE: This operation can take up to 30 seconds
my_index = aiplatform.MatchingEngineIndex.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_UPDAT
)

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

# NOTE: This operation can take up to 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 ind
index_id = my_index_endpoint.deployed_indexes[-1].index.split("/")[-1]
endpoint_id = my_index_endpoint.name
```

```
# TODO: Replace 1234567890123456789 with your actual index ID
my_index = aiplatform.MatchingEngineIndex(index_id)

# TODO: Replace 1234567890123456789 with your actual 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 vector store
vector_store.add_texts(texts=texts)

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

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 conte
```

```
the user's question.
If you don't know the answer, just say that "I don't know", don't try to make up ar
{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 Markdown 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

# Ask the question and get the result
query = "What shape is the planet where humans live?"
```

```
result = chain(query)

# Display the result
display(Markdown(print_result(result)))
```

Snippet 11. Build/deploy ANN Index for Vertex AI Vector Search and use RAG with LLM prompts to generate grounded results/sources.

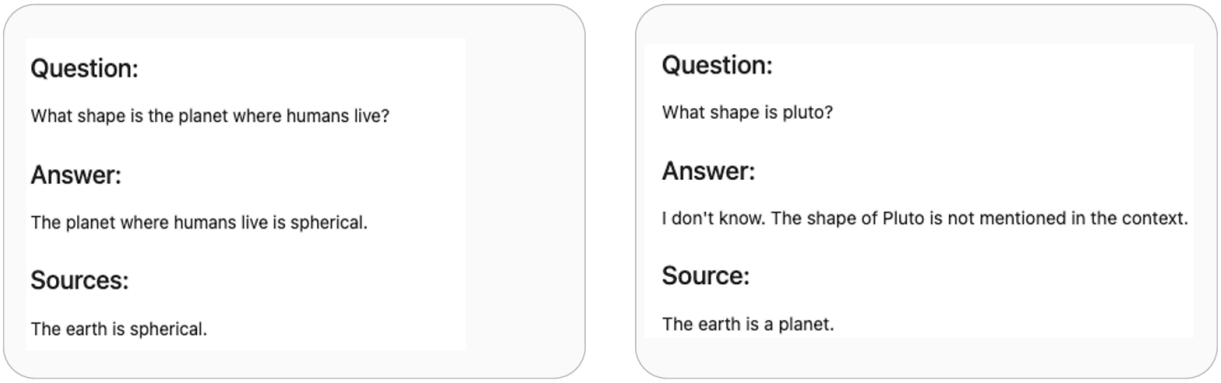


Figure 16. Model responses along with sources demonstrating the LLM being grounded in the database

Như chúng ta có thể suy ra từ Hình 16, đầu ra không chỉ dựa trên các kết quả có ngữ nghĩa tương tự được truy xuất từ cơ sở dữ liệu (do đó từ chối trả lời khi không tìm thấy ngữ cảnh trong cơ sở dữ liệu). Điều này không chỉ giảm đáng kể hiện tượng "ảo giác" mà còn cung cấp các nguồn để xác minh, thông qua con người hoặc bằng cách sử dụng một LLM khác.

F. Summary

Trong tài liệu này, chúng ta đã thảo luận về nhiều phương pháp để tạo, quản lý, lưu trữ và truy xuất embedding của các kiểu dữ liệu khác nhau một cách hiệu quả trong bối cảnh các ứng dụng đạt tiêu chuẩn sản xuất. Việc tạo, duy trì và sử dụng embedding cho các ứng dụng phía sau có thể là một nhiệm vụ phức tạp, đòi hỏi sự tham gia của nhiều vai trò trong tổ chức. Tuy nhiên, bằng cách vận hành và tự động hóa quy trình này một cách cẩn thận, bạn có thể tận dụng an toàn những lợi ích tuyệt vời mà embedding mang lại cho các ứng dụng quan trọng nhất.

Một số điểm chính từ tài liệu này bao gồm:

1. **Lựa chọn mô hình embedding phù hợp với dữ liệu và trường hợp sử dụng của bạn**:

   Đảm bảo dữ liệu được sử dụng trong suy luận (inference) nhất quán với dữ liệu được sử dụng trong huấn luyện. Sự thay đổi phân phối từ giai đoạn huấn luyện sang suy luận có thể xuất phát từ nhiều yếu tố, bao gồm sự thay đổi phân phối theo lĩnh vực hoặc theo nhiệm vụ phía sau. Nếu không có mô hình embedding hiện có nào phù hợp với phân phối dữ liệu suy luận hiện tại, việc tinh chỉnh (fine-tuning) mô hình hiện có có thể cải thiện đáng kể hiệu suất.

   Một sự đánh đổi khác đến từ kích thước mô hình. Các mô hình dựa trên mạng nơ-ron sâu lớn (như mô hình đa phương thức lớn) thường có hiệu suất tốt hơn nhưng có thể dẫn đến thời gian phản hồi dài hơn. Sử dụng các dịch vụ embedding dựa trên đám mây có thể giải quyết vấn đề này bằng cách cung cấp cả dịch vụ embedding chất lượng cao và độ trễ thấp. Đối với hầu hết các ứng dụng doanh nghiệp, sử dụng mô hình embedding đã được huấn luyện trước là một điểm khởi đầu tốt, có thể tiếp tục tinh chỉnh hoặc tích hợp vào các mô hình phía sau. Nếu dữ liệu có cấu trúc đồ thị vốn có, embedding đồ thị có thể mang lại hiệu suất vượt trội.

2. **Chọn cơ sở dữ liệu vector phù hợp với ngân sách và nhu cầu kinh doanh của bạn**:

   Mặc dù có vẻ nhanh hơn khi tạo nguyên mẫu bằng các lựa chọn mã nguồn mở có sẵn, việc chọn một cơ sở dữ liệu vector được quản lý an toàn, có khả năng mở rộng và đã được kiểm nghiệm thực tế sẽ tốt hơn về lâu dài. Có nhiều lựa chọn mã nguồn mở sử dụng một trong các thuật toán tìm kiếm vector ANN mạnh mẽ, nhưng **ScaNN** và **HNSW** đã chứng minh mang lại sự cân bằng tốt nhất giữa độ chính xác và hiệu suất, theo thứ tự đó.

3. **Embedding kết hợp với cơ sở dữ liệu vector được hỗ trợ bởi ANN là một công cụ cực kỳ mạnh mẽ**:

   Nó có thể được áp dụng cho nhiều ứng dụng, bao gồm **tìm kiếm (Search)**, **hệ thống gợi ý (Recommendation Systems)** và **Retrieval Augmented Generation (RAG)** cho các mô hình ngôn ngữ lớn (LLMs). Cách tiếp cận này không chỉ giảm thiểu vấn đề "ảo giác" (*hallucination*) mà còn tăng cường khả năng xác minh và sự tin cậy của các hệ thống dựa trên LLM.