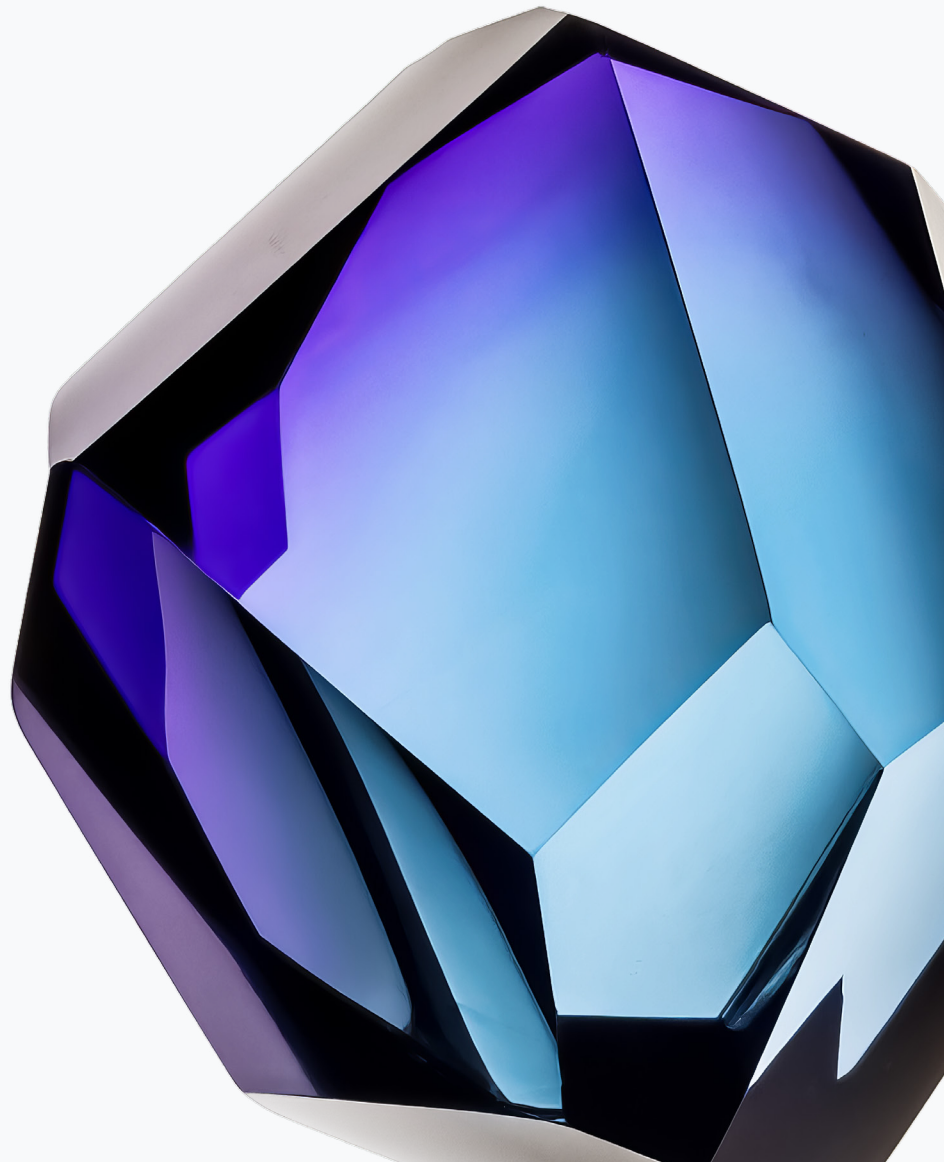


Embeddings & Vector Stores

Authors: Anant Nawalgaria
and Xiaoqi Ren



Acknowledgements

Reviewers and 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

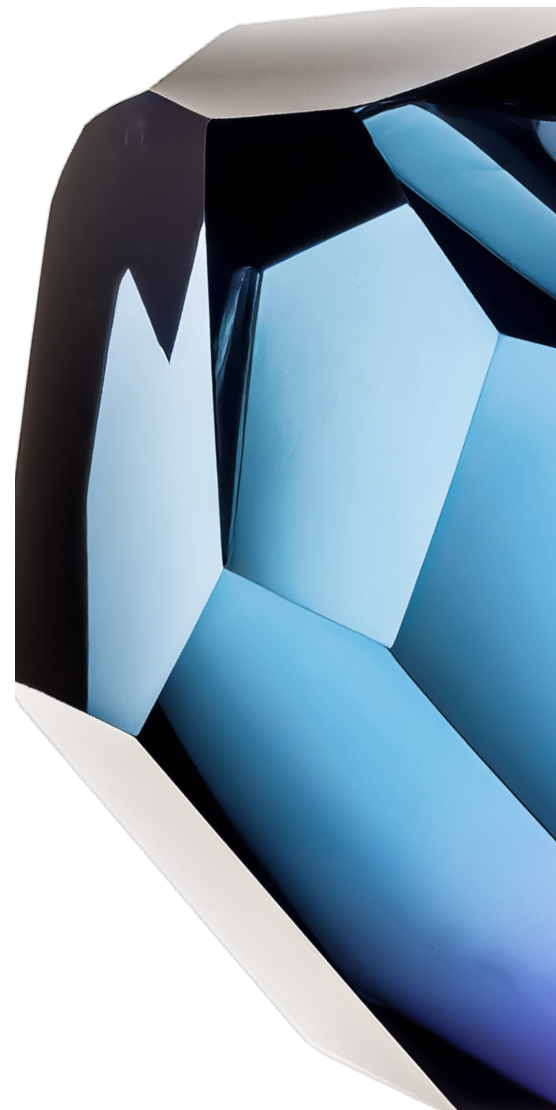
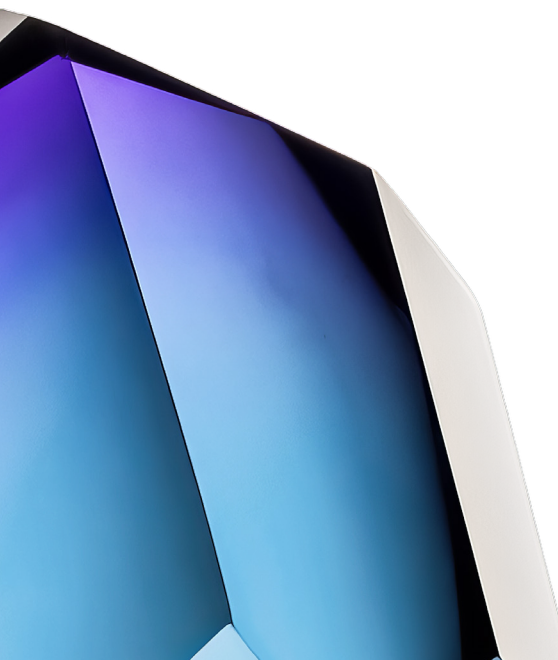



Table of contents

Introduction	5
Why embeddings are important	6
Types of embeddings	9
Text embeddings	9
Word embeddings	11
Document embeddings	15
Shallow BoW models	16
Deeper pretrained large language models	18
Image & multimodal embeddings	22
Structured data embeddings	24
General structured data	24
User/item structured data	25
Graph embeddings	25
Training Embeddings	26



Vector search	28
Important vector search algorithms	29
Locality sensitive hashing & trees	30
Hierarchical navigable small worlds	33
ScaNN	34
Vector databases	37
Operational considerations	39
Applications	40
Q & A with sources (retrieval augmented generation)	42
Summary	47
Endnotes	49



These low-dimensional numerical representations of real-world data significantly helps efficient large-scale data processing and storage by acting as means of lossy compression of the original data.

Introduction

Modern machine learning thrives on diverse data—images, text, audio, and more. This whitepaper explores the power of embeddings, which transform this heterogeneous data into a unified vector representation for seamless use in various applications.

We'll guide you through:

- **Understanding Embeddings:** Why they are essential for handling multimodal data and their diverse applications.
- **Embedding Techniques:** Methods for mapping different data types into a common vector space.

- **Efficient Management:** Techniques for storing, retrieving, and searching vast collections of embeddings.
- **Vector Databases:** Specialized systems for managing and querying embeddings, including practical considerations for production deployment.
- **Real-World Applications:** Concrete examples of how embeddings and vector databases are combined with large language models (LLMs) to solve real-world problems.

Throughout the whitepaper, code snippets provide hands-on illustrations of key concepts.

Why embeddings are important

In essence, embeddings are numerical representations of real-world data such as text, speech, image, or videos. They are expressed as low-dimensional vectors where the geometric distances of two vectors in the vector space is a projection of the relationships between the two real-world objects that the vectors represent. In other words they help you with providing compact representations of data of different types, while simultaneously also allowing you to compare two different data objects and tell how similar or different they are on a numerical scale: for example: The word 'computer' has a similar meaning to the picture of a computer, as well as the word 'laptop' but not to the word 'car'. These low-dimensional numerical representations of real-world data significantly helps efficient large-scale data processing and storage by acting as means of lossy compression of the original data while retaining its important properties.

One of the key applications for embeddings is retrieval and recommendations, where the result is usually from a massive search space. For example, Google Search is a retrieval with the search space of the whole internet. Today's retrieval and recommendation systems' success depends on the following:

1. Precomputing the embeddings for billions items of the search space.
2. Mapping query embeddings to the same embedding space.
3. Efficient computing and retrieving of the nearest neighbors of the query embeddings in the search space.

Embeddings also shine in the world of multimodality. Most applications work with large amounts of data of various modalities: text, speech, image, and videos to name a few. Because every entity or object is represented in its own unique format, it's very difficult to project these objects into the same vector space that is both compact and informative. Ideally, such a representation would capture as much of the original object's characteristics as possible. An *embedding* refers to the projected vector of an object from an input space to a relatively low-dimensional vector space. Each vector is a list of floating point numbers.

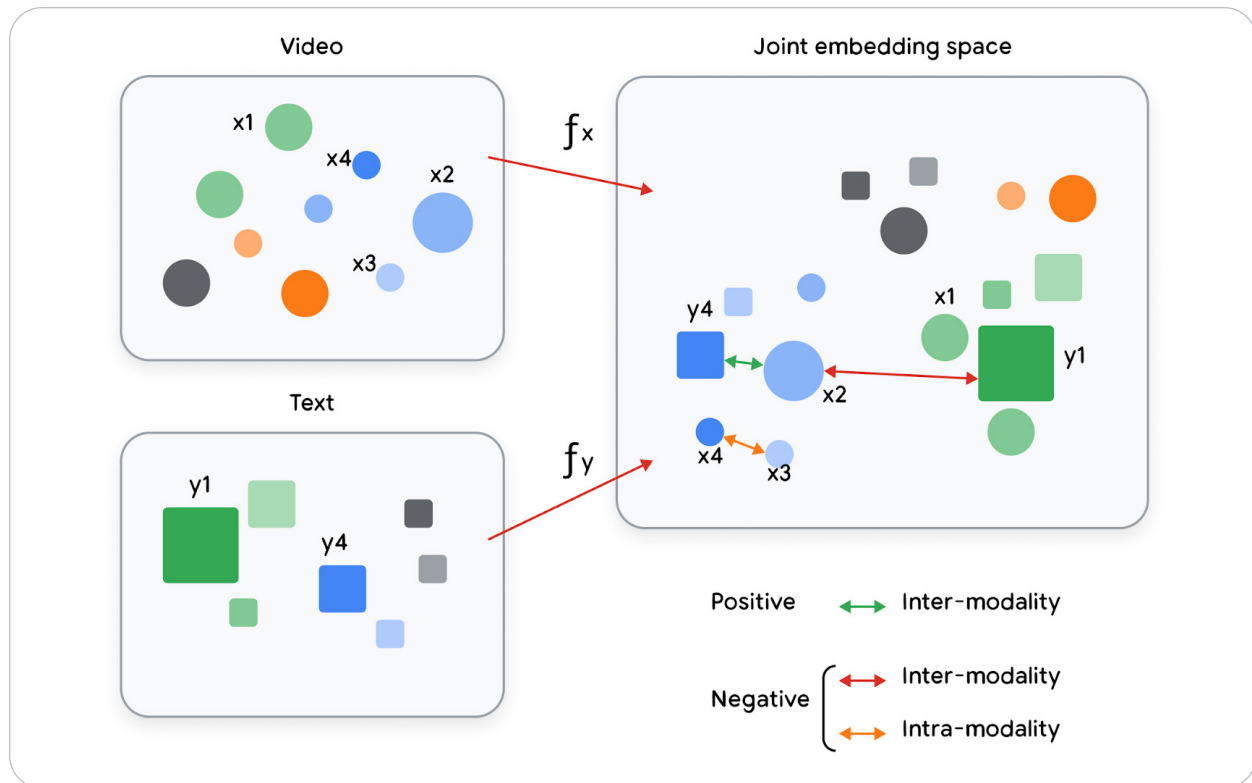


Figure 1. Projecting objects/content into a joint vector space with semantic meaning

Ideally the embeddings are created so they place objects with similar **semantic properties** **closer in the embedding space** (a low-dimensional vector space where items can be projected). The embeddings can then be used as a condensed, meaningful input in downstream applications. For example, you can use them as features for ML models, recommender systems, search engines, and many more. So your data not only gets a compact numerical representation, but this representation also preserves the semantic meanings for a specific task or across a variety of tasks. The fact that these representations are task-specific means you can generate different embeddings for the same object, optimized for the task at hand.

Types of embeddings

Embeddings aim to obtain a low dimensional representation of the original data while preserving most of the ‘essential information’. The types of data an embedding represents can be of various different forms. Below you’ll see some standard techniques used for different types of data, including text and image.

Text embeddings

Text embeddings are used extensively as part of natural language processing (NLP). They are often used to embed the meaning of natural language in machine learning for processing in various downstream applications such as text generation, classification, sentiment analysis, and more. These embeddings broadly fall into two categories: token/word and document embeddings.

Before diving deeper into these categories, it’s important to understand the entire lifecycle of text: from its input by the user to its conversion to embeddings.

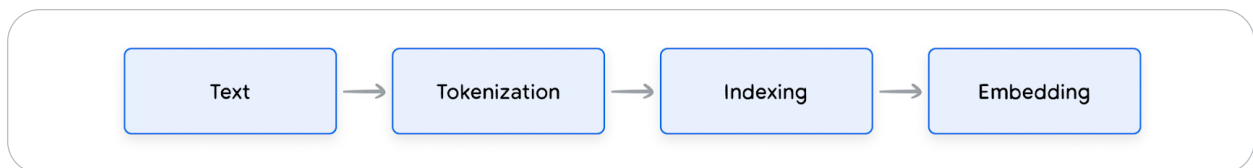


Figure 2. The process of turning text into embeddings

It all starts with the input string which is split into smaller meaningful pieces called tokens. This process is called *tokenization*. Commonly, these tokens are wordpieces, characters, words, numbers, and punctuations using one of the many existing tokenization techniques.¹ After the string is tokenized, each of these tokens is then assigned a unique integer value

usually in the the range: [0, cardinality of the total number of tokens in the corpus]. For example, for a 16 word vocabulary the IDs would range between 0-15. This value is also referred to as token ID. These tokens can be used to represent each string as a sparse numerical vector representation of documents used for downstream tasks directly, or after one-hot encoding. One-hot encoding is a binary representation of categorical values where the presence of a word is represented by 1, and its absence by 0. This ensures that the token IDs are treated as categorical values as they are, but often results in a dense vector the size of the vocabulary of the corpus. Snippet 1 and Figure 3 show an example of how this can be done using 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 punctuations, 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 )
```

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. 1. 1. 1. 1. 0. 1. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 1. 0. 0. 1. 1. 1. 1. 1. 1. 0.]]
```

Figure 3. Output of Snippet 1

However, since these Integer IDs (or their corresponding one-hot encoded vectors) are assigned randomly to words, they lack any inherent semantic meaning. This is where embeddings are much more useful. Although it's possible to embed character and sub-word level tokens as well, let us look at word and document embeddings to understand some of the methods behind them.

Word embeddings

In this section, you'll see a few word embedding techniques and algorithms to both train and use word embeddings. While there are many ML driven algorithms developed over time optimized for different objectives, the most common ones are GloVe,² SWIVEL,³ and Word2Vec.⁴ Word embeddings or sub-word embeddings can also be directly obtained from hidden layers of language models. **However, the embeddings will be different for the same word in different contexts of the text.** This section focuses on lightweight, context-free word embedding and leaves the context-aware document embeddings for the document embeddings section. Word embedding can be directly applied to downstream tasks like named entity extraction and topic modeling.

Word2Vec is a family of model architectures that operates on the principle of "the semantic meaning of a word is defined by its neighbors", or words that frequently appear close to each other in the training corpus. This method can be both used to train your own embeddings

from large datasets or be quickly integrated through one of the readily available pre-trained embeddings available online.⁵ The embeddings for each word - which are essentially fixed length vectors - are randomly initialized to kick off the process, resulting in a matrix of shape (size_of_vocabulary, size_of_each_embedding). This matrix can be used as a lookup table after the training process is completed using one of the following methods (see Figure 4).

- The Continuous bag of words (CBOW) approach: Tries to predict the middle word, using the embeddings of the surrounding words as input. This method is agnostic to the order of the surrounding words in the context. This approach is fast to train and is slightly more accurate for frequent words.
- The skip-gram approach: The setup is inverse of that of CBOW, with the middle word being used to predict the surrounding words within a certain range. This approach is slower to train but works well with small data and is more accurate for rare words.

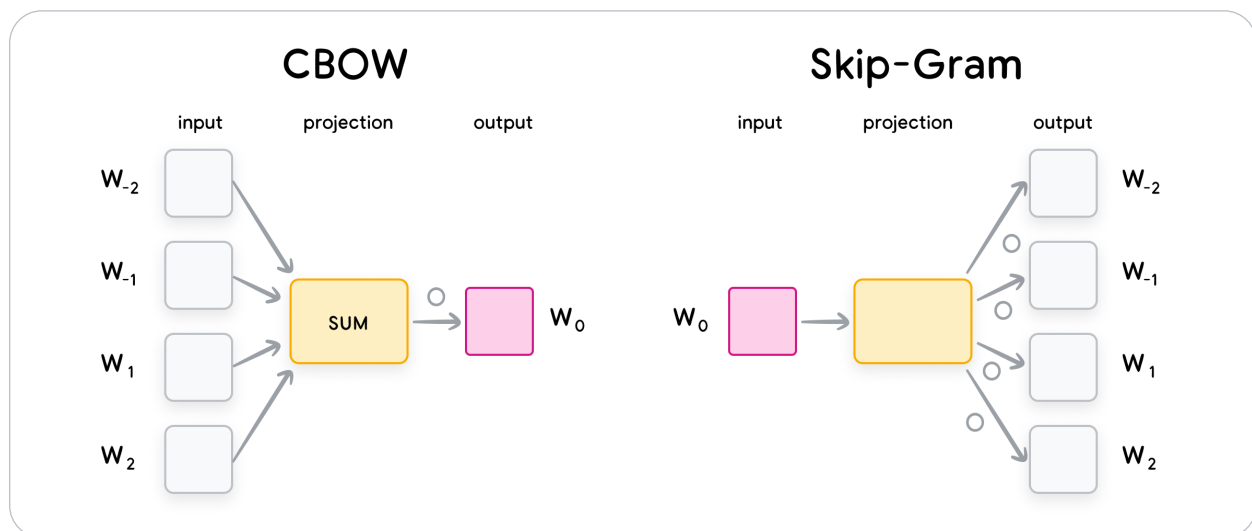


Figure 4. Diagram explaining how CBOW and Skip-Gram methods work

The Word2Vec algorithms can also be extended to the sub-word level, which has been the inspiration for algorithms such as FastText.⁶ However, one of the major caveats of Word2Vec is that although it **accounts well for local statistics of words within a certain sliding window**, it does not capture the global statistics (words in the whole corpus). This shortcoming is what methods like the GloVe algorithm address.

GloVe is a word embedding technique that leverages both global and local statistics of words. It does this by first creating a **co-occurrence matrix**, which represents the relationships between words. GloVe then uses a factorization technique to learn word representations from the co-occurrence matrix. The resulting word representations are able to capture both global and local information about words, and they are useful for a variety of NLP tasks.

In addition to GloVe, SWIVEL is another approach which leverages the co-occurrence matrix to learn word embeddings. SWIVEL stands for Skip-Window Vectors with Negative Sampling. Unlike GloVe, it uses local windows to learn the word vectors by taking into account the co-occurrence of words within a fixed window of its neighboring words. Furthermore, SWIVEL also considers unobserved co-occurrences and handles it using a special piecewise loss, boosting its performance with rare words. It is generally considered only slightly less accurate than GloVe on average, but is considerably faster to train. This is because it leverages distributed training by subdividing the Embedding vectors into smaller sub-matrices and executing matrix factorization in parallel on multiple machines. Snippet 2 below demonstrates loading pre-trained word embeddings for both Word2Vec and GloVe and visualizing them in a 2D space, and computing nearest neighbors.

Word embeddings can be directly used in some downstream tasks like 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 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)

```

Snippet 2. Loading and plotting GloVe and Word2Vec embeddings in 2D

Figure 5 Shows semantically similar words are clustered differently for the two algorithms

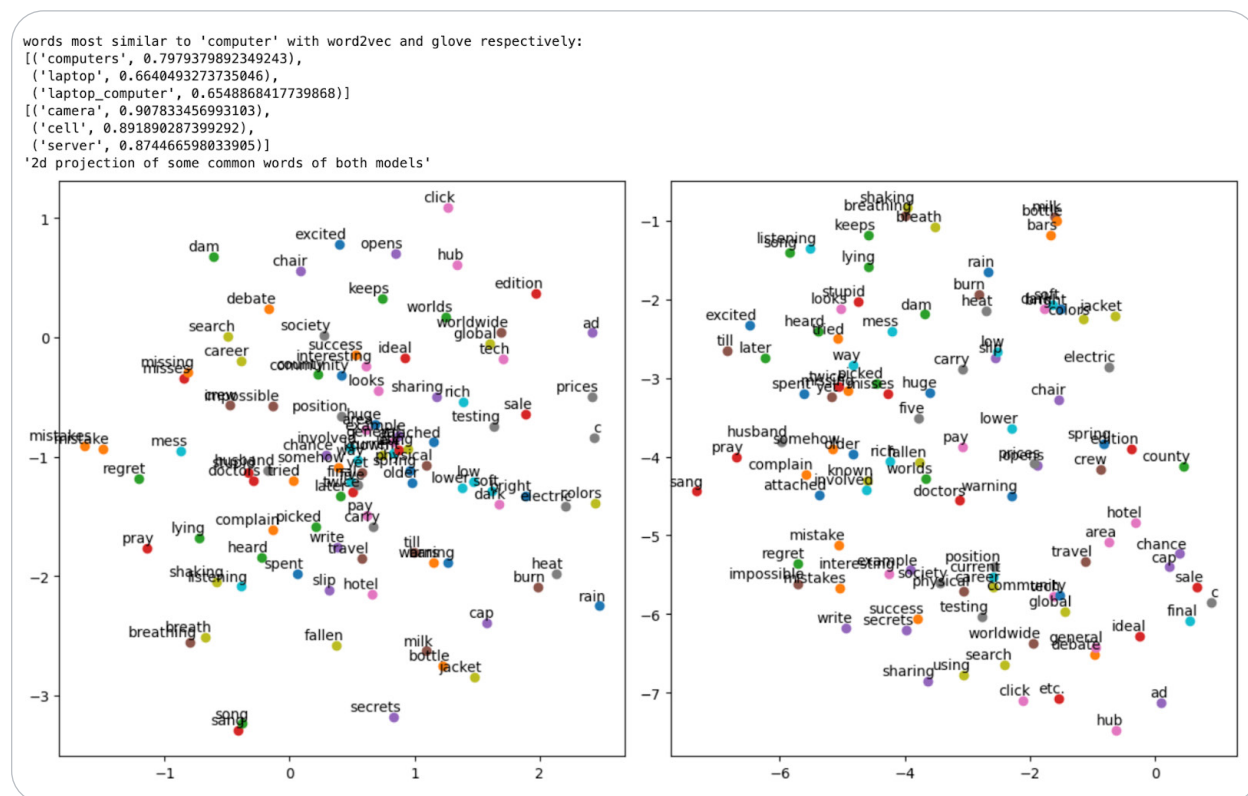


Figure 5. 2D visualization of pre-trained GloVe and Word2Vec word embeddings

Document embeddings

Embedding documents to low-dimensional dense embedding has attracted long-lasting interests since the 1980s. Document embeddings can be used in various applications, including semantic search, topic discovery, classification, and clustering to embed the meaning of a series of words in paragraphs and documents and use it for various

downstream applications. The evolution of the embeddings models can mainly be categorized into two stages: shallow Bag-of-words (BoW) models and deeper pretrained large language models.

Shallow BoW models

Early document embedding works follow the bag-of-words (BoW) paradigm, assuming a document is an unordered collection of words. These early works include latent semantic analysis (LSA)⁷ and latent dirichlet allocation (LDA).⁸ Latent semantic analysis (LSA) uses a co-occurrence matrix of words in documents and latent dirichlet allocation (LDA) uses a bayesian network to model the document embeddings. Another famous bag-of-words family of document embeddings is TF-IDF (term frequency-inverse document frequency) based models, which are statistical models that use the word frequency to represent the document embedding. TF-IDF-based models can either be a sparse embedding, which represents the term-level importance, or can be combined with word embeddings as a weighting factor to generate a dense embedding for the documents. For example, BM25, a TF-IDF-based bag-of-words model, is still a strong baseline in today's retrieval benchmarks.⁹

However, the bag-of-words paradigm also has two major weaknesses: both the word ordering and the semantic meanings are ignored. BoW models fail to capture the sequential relationships between words, which are crucial for understanding meaning and context. Inspired by Word2Vec, Doc2Vec¹⁰ was proposed in 2014 for generating document embeddings using (shallow) neural networks. The Doc2Vec model adds an additional 'paragraph' embedding or, in other words, document embedding in the model of Word2Vec as illustrated in Figure 6. **The paragraph embedding is concatenated or averaged with other word embeddings to predict a random word in the paragraph.** After training, for existing paragraphs or documents, the learned embeddings can be directly used in downstream tasks. For a new paragraph or document, extra inference steps need to be performed to generate the paragraph or document embedding.

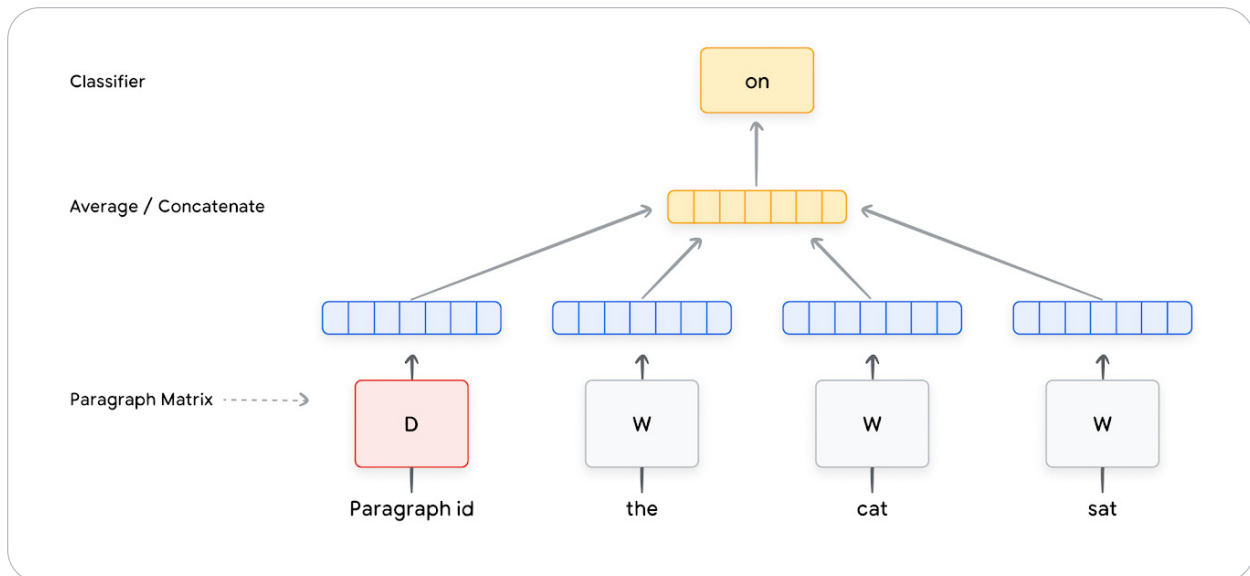


Figure 6. Doc2vec CBOW model

Snippet 3 below shows how you can train your own doc2Vec models on a custom corpus:

```
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"])
```

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

The success of applying neural networks in the embedding world inspired an increasing interest in using deep neural networks to generate embeddings.

Deeper pretrained large language models

Motivated by the development of deep neural networks, different embedding models and techniques were proposed, and the state-of-the-art models are refreshed frequently. Main changes of the models include:

1. Using more complex learning models, especially bi-directional deep neural network models.
2. The use of massive pre-training on unlabeled text.
3. The use of a subword tokenizer.
4. Using fine-tuning for various downstream NLP tasks.

In 2018, BERT¹¹ - which stands for bidirectional encoder representations from transformers - was proposed with groundbreaking results on 11 NLP tasks. Transformer, the model paradigm BERT based on, has become the mainstream model paradigm until today. Besides using a transformer as the model backbone, another key of BERT's success is from pre-training with a massive unlabeled corpus. In pretraining, BERT utilized masked language model (MLM) as the pre-training objective. It did this by randomly masking some tokens of the input and using the masked token id as the prediction objective. This allows the model to utilize both the right and left context to pretrain a deep bidirectional transformer. BERT also utilizes the next sentence prediction task in pretraining. BERT outputs a contextualized embedding for every token in the input. Typically, the embedding of the first token (a special token named [CLS]) is used as the embedding for the whole input.

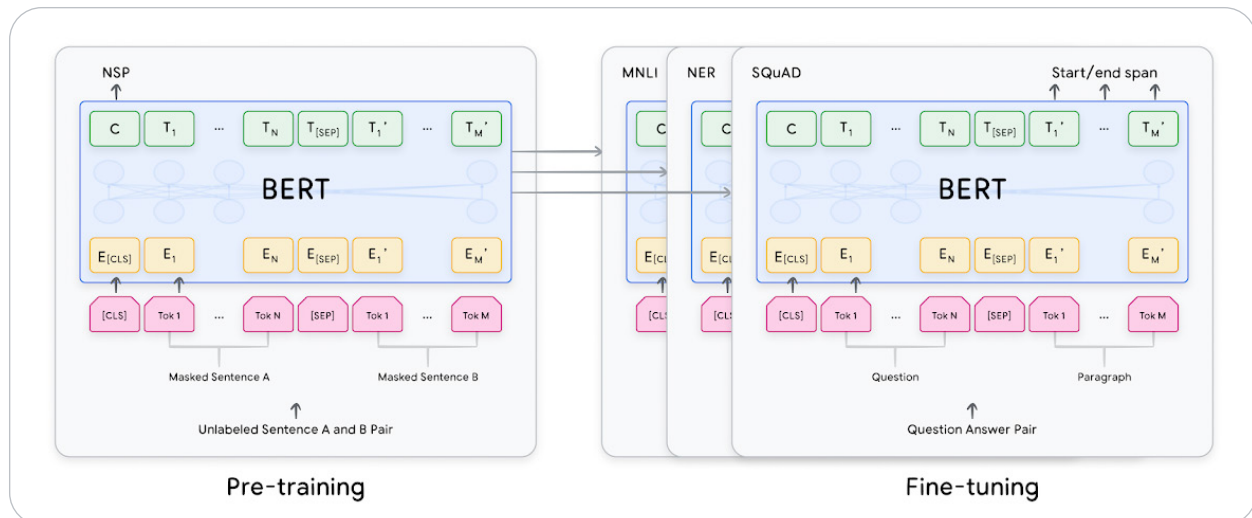


Figure 7. The BERT architecture

BERT became the base model for multiple embedding models, including Sentence-BERT,¹² SimCSE,¹³ and E5.¹⁴ Meanwhile, the evolution of language models - especially large language models - never stops. T5 was proposed in 2019 with up to 11B parameters. PaLM was proposed in 2022 to push the large language model to a surprising 540B parameters. Models like Gemini from Google, GPT models from OpenAI and Llama models from Meta are also evolving to newer generations at astonishing speed. Please refer to the whitepaper on Foundational models for more information about some common LLMs.

New embedding models based on large language models have been proposed. For example, GTR and Sentence-T5 show better performance on retrieval and sentence similarity (respectively) than BERT family models.

Another approach to new embeddings models development is generating multi-vector embeddings instead of a single vector to enhance the representational power of the models. Embedding models in this family include ColBERT¹⁵ and XTR.¹⁶

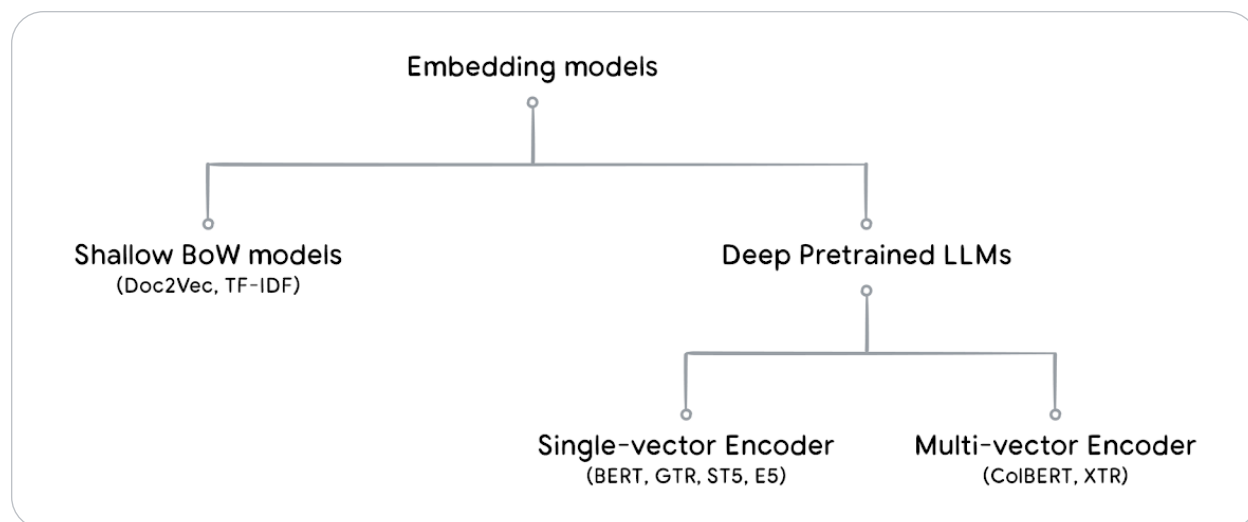


Figure 8. An illustration of the taxonomy diagram of the embedding models

Although the deep neural network models require a lot more data and compute time to train, they have much better performance compared to models using bag-of-words paradigms. For example, for the same word the embeddings would be different with different contexts. Snippet 4 demonstrates how pre-trained document embedding models from Tensorflow-hub¹⁷ (for example, Sentence t5)^A and Vertex AI^B can be used for training models with Keras and TF datasets. Vertex Generative AI text embeddings can be used with the Vertex AI SDK, Langchain, and Google’s BigQuery (Snippet 5) for embedding and advanced workflows.¹⁸

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, 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))
```

Snippet 4. Creating & integrating 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

Image & multimodal embeddings

Much like text, it's also possible to create both image and multimodal embeddings.

Unimodal image embeddings can be derived in many ways: one of which is by training a CNN or Vision Transformer model on a large scale image classification task (for example, Imagenet), and then using the penultimate layer as the image embedding. This layer has learnt some important discriminative feature maps for the training task. It contains a set of feature maps that are discriminative for the task at hand and can be extended to other tasks as well.

To obtain *multimodal embeddings*¹⁹ you take the individual unimodal text and image embeddings and their semantic relationships learnt via another training process. This gives you a fixed size semantic representation in the same latent space. The below snippet (Snippet 6) can be used to compute image and multimodal embeddings for images and text and be used with a keras model directly (much like the text embedding example).

```
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 model
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 = encoded_content

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

Continues next page...

```
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..")
```

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

Structured data embeddings

There are two common ways to generate embeddings for structured data, one is more general while the other is more tailored for recommendation applications.

Unlike unstructured data, where a pre-trained embedding model is typically available, we have to **create the embedding model for the structured data** since it would be specific to a particular application.

General structured data

Given a general structured data table, we can create embedding for each row. This can be done by the ML models in the dimensionality reduction category, such as the PCA model.

One use case for these embeddings are for anomaly detection. For example, we can create embeddings for anomaly detection using large data sets of labeled sensor information that identify anomalous occurrences.²⁰ Another case use is to feed these embeddings to downstream ML tasks such as classification. Compared to using the original high-dimensional data, using embeddings to train a supervised model requires less data. This is particularly important in cases where training data is not sufficient.

User/item structured data

The input is no longer a general structured data table as above. Instead, the input includes the user data, item/product data plus the data describing the interaction between user and item/product, such as rating score.

This category is for recommendation purposes, as it maps two sets of data (user dataset, item/product/etc dataset) into the same embedding space. For recommender systems, we can create embeddings out of structured data that correlate to different entities such as products, articles, etc. Again, we have to create our own embedding model. Sometimes this can be combined with unstructured embedding methods when images or text descriptions are found.

Graph embeddings

Graph embeddings are another embedding technique that lets you represent not only information about a specific object but also its neighbors (namely, their graph representation). Take an example of a social network where each person is a node, and the connections between people are defined as edges. Using graph embedding you can model each node as an embedding, such that the embedding captures not only the semantic information about the person itself, but also its relations and associations hence enriching the embedding. For example, if two nodes are connected by an edge, the vectors for those nodes would be similar. You might then be able to predict who the person is most similar to and recommend new connections. Graph embeddings can also be used for a variety of tasks, including node classification, graph classification, link prediction, clustering, search, recommendation systems, and more. Popular algorithms^{21,22} for graph embedding include DeepWalk, Node2vec, LINE, and GraphSAGE.²³

Training Embeddings

Current embedding models usually use dual encoder (two tower) architecture. For example, for the text embedding model used in question-answering, one tower is used to encode the queries and the other tower is used to encode the documents. For the image and text embedding model, one tower is used to encode the images and the other tower is used to encode the text. The model can have various sub architectures, depending on how the model components are shared between the two towers. The following figure shows some architectures of the dual encoders.²⁴

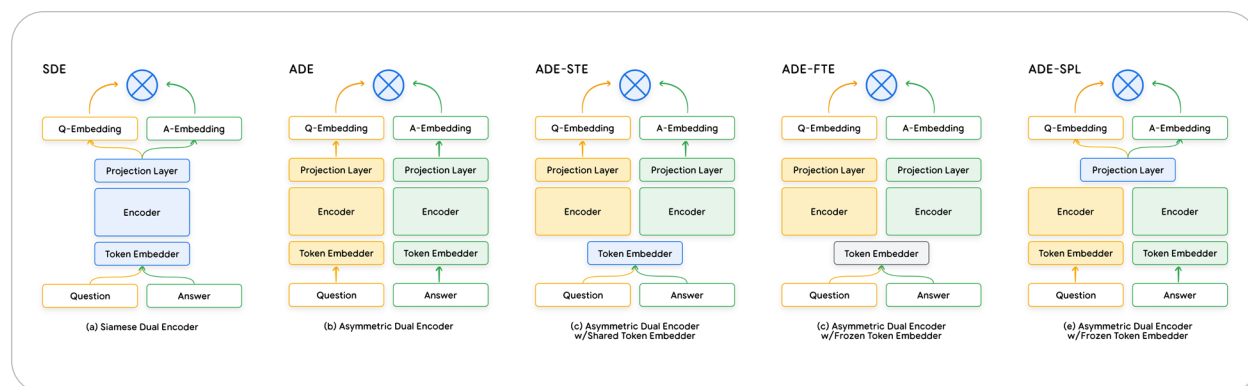


Figure 9. Some architectures of dual encoders

The loss used in embedding models training is usually a variation of **contrastive loss**, which takes a tuple of <inputs, positive targets, [optional] negative targets> as the inputs. Training with contrastive loss brings positive examples closer and negative examples far apart.

Similar to foundation model training, training of an embedding model from scratch usually includes two stages: **pretraining (unsupervised learning)** and **fine tuning (supervised learning)**. Nowadays, the embedding models are usually directly initialized from foundation models such as BERT, T5, GPT, Gemini, CoCa. You can use these base models to leverage the massive knowledge that has been learned from the large-scale pretraining of the foundation

models. The fine-tuning of the embedding models can have one or more phases. The fine-tuning datasets can be created in various methods, including **human labeling, synthetic dataset generation, model distillation, and hard negative mining**.

To use embeddings for downstream tasks like classification or named entity recognition, extra layers (for example, softmax classification layer) can be added on top of the embedding models. The embedding model can either be frozen (especially when the training dataset is small), trained from scratch, or fine-tuned together with the downstream tasks.

Vertex AI provides the ability to customize the Vertex AI text embedding models.²⁵ Users can also choose to fine-tune the models directly. See²⁶ for an example of fine tuning the BERT model using tensorflow model garden. You can also directly load the embedding models from tfhub and fine-tune on top of the model. Snippet 7 shows an example how to build a classifier based on tfhub models.

```
# 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
```

Snippet 7. Creating a Keras model using trainable tfhub layer

So far you've seen the various types of embeddings, techniques and best practices to train them for various data modalities, and some of their applications. The next section discusses how to persist and search the embeddings that have been created in a fast and scalable way for production workloads.

Vector search

Full-text keyword search has been the lynchpin of modern IT systems for years. Full-text search engines and databases (relational and non-relational) often rely on explicit keyword matching. For example, if you search for 'cappuccino' the search engine or database returns all documents that mention the exact query in the tags or text description. However, if the key word is misspelled or described with a differently worded text, a traditional keyword search returns incorrect or no results. There are traditional approaches which are tolerant of misspellings and other typographical errors. However, they are still unable to find the results having the closest underlying semantic meanings to the query. This is where vector search is very powerful: it uses the vector or embedded semantic representation of documents.

Vector search lets you to go beyond searching for exact query literals and allows you to search for the meaning across various data modalities. This provides you more nuanced results. After you have a function that can compute embeddings of various items, you compute the embedding of the items of interest and store this embedding in a database. You then embed the incoming query in the same vector space as the items. Next, you have to find the best matches to the query. This process is analogous to finding the most 'similar' matches across the entire collection of searchable vectors: similarity between vectors can be computed using a metric such as euclidean distance, cosine similarity, or dot product.

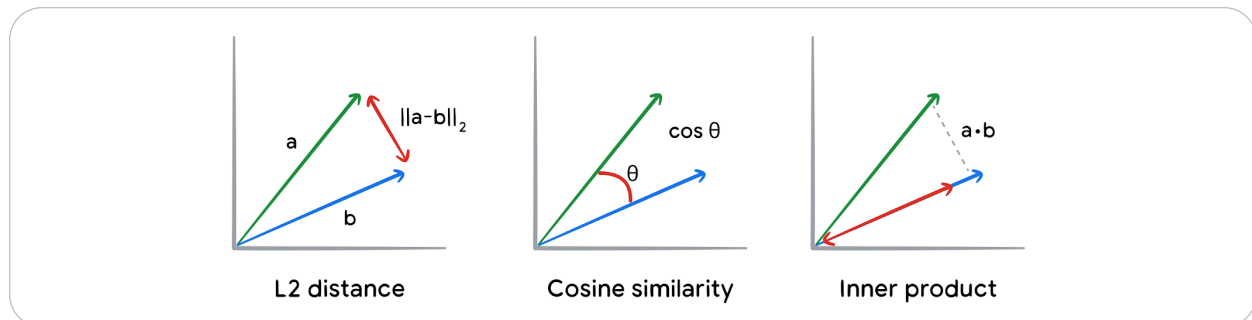


Figure 10. Visualization of how different metrics compute vector similarity

Euclidean distance (i.e., L2 distance) is a geometric measure of the distance between two points in a vector space. This works well for lower dimensions. Cosine similarity is a measure of the angle between two vectors. And inner/dot product, is the projection of one vector onto another. They are equivalent when the vector norms are 1. This seems to work better for higher dimensional data. Vector databases store and help manage and operationalize the complexity of vector search at scale, while also addressing the common database needs.

Important vector search algorithms

The most straightforward way to find the most similar match is to run a traditional linear search by comparing the query vector with each document vector and return the one with the highest similarity. However, the runtime of this approach scales linearly ($O(N)$) with the amount of documents or items to search. This approach is unacceptably slow for most use cases involving several millions of documents or more. Using **approximate nearest neighbour**

(ANN) search for that purpose is more practical. ANN is a technique for finding the closest points to a given point in a dataset with a small margin of error - but with a tremendous boost in performance. There are many approaches with varying trade-offs across scale, indexing time, performance, simplicity and more.²⁷ They use one or more implementations of the following techniques: quantization, hashing, clustering and trees, among others. Some of the most popular approaches are discussed below.

Locality sensitive hashing & trees

Locality sensitive hashing (LSH)²⁸ is a technique for finding similar items in a large dataset. It does this by creating one or more hash functions that map similar items to the same hash bucket with high probability. This means that you can quickly find all of the similar items to a given item by only looking at the candidate items in the same hash bucket (or adjacent buckets) and do a linear search amongst those candidate pairs. This allows for significantly faster lookups within a specific radius. The number of hash functions/tables and buckets determine the search recall/speed tradeoff, as well as the false positive / true positive one. Having too many hash functions might cause similar items to different buckets, while too few might result in too many items falsely being hashed to the same bucket and the number of linear searches to increase.

Another intuitive way to think about LSH is grouping residences by their postal code or neighborhood name. Then based on where someone chooses to move you look at the residences for only that neighborhood and find the closest match.

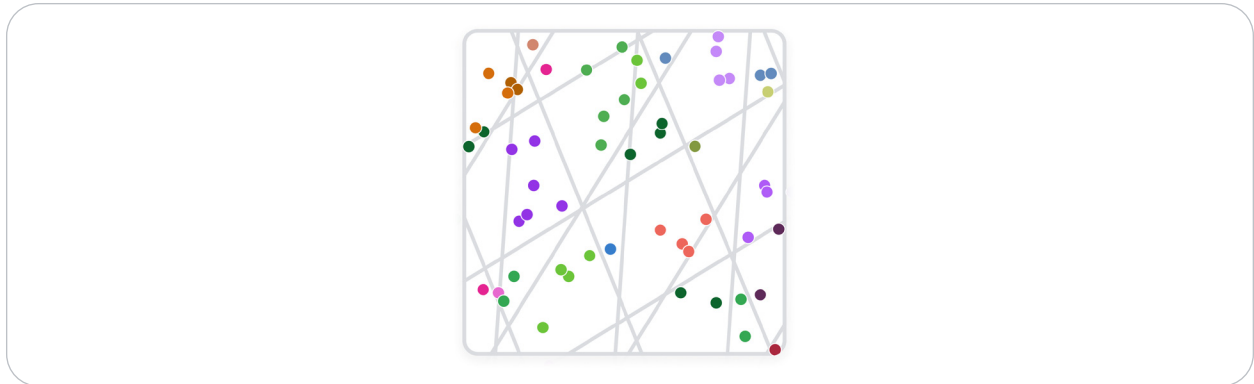


Figure 11. Visualization of how LSH uses random hyperplanes to partition the vector space

Tree-based algorithms work similarly. For example, **the Kd-tree approach** works by creating the decision boundaries by computing the median of the values of the first dimension, then that of the second dimension and so on. This approach is very much like a decision tree. Naturally this can be ineffective if searchable vectors are high dimensional. In that case, the **Ball-tree algorithm is better suited.** It is similar in functionality, except instead of going by dimension-wise medians it creates buckets based on the radial distance of the data points from the center. Here is an example of the implementation of these three approaches:

```

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

```

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

Hashing and tree-based approaches can also be combined and extended upon to obtain the optimal tradeoff between recall and latency for search algorithms. FAISS with HNSW and ScaNN are good examples.

Hierarchical navigable small worlds

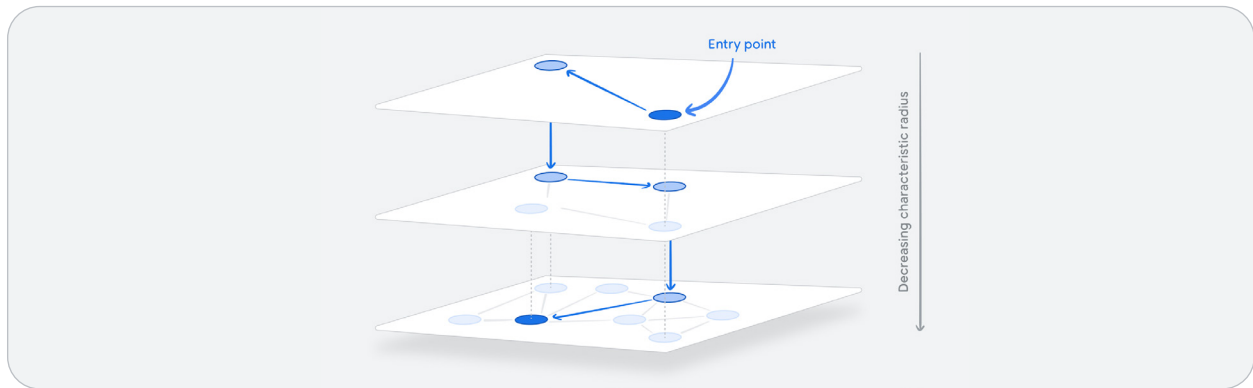


Figure 12. Diagram showing how HNSW ‘zooms in’ to perform ANN

One of the FAISS (Facebook AI similarity search) implementations leverages the concept of hierarchical navigable small world (HNSW) ³¹ to perform vector similarity search in sub-linear ($O(\text{Log}n)$) runtime with a good degree of accuracy. A HNSW is a proximity graph with a hierarchical structure where the graph links are spread across different layers. The top layer has the longest links and the bottom layer has the shortest ones. As shown in Figure 9, the search starts at the topmost layer where the algorithm greedily traverses the graph to find the vertex most semantically similar to the query. Once the local minimum for that layer is found, it then switches to the graph for the closest vertex on the layer below. This process continues iteratively until the local minimum for the lowest layer is found, with the algorithm keeping track of all the vertices traversed to return the K-nearest neighbors. This algorithm can be optionally augmented with quantization and vector indexing to boost speed and memory efficiency.

```
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)
```

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

ScaNN

Google developed the scalable approximate nearest neighbor (ScaNN)^{32,33} approach which is used across a lot of its products and services. This includes being externally available to all customers of Google Cloud through the Vertex AI Vector Search. Below is how ScaNN uses a variety of steps to perform efficient vector search, with each one of them having their own subset of parameters.

The first step is the optional partitioning step during training: it uses one of the multiple algorithms available to partition the vector store into logical partitions/clusters where the semantically related are grouped together. The partitioning step is optional for small datasets. However, for larger datasets with >100k embedding vectors, the partitioning step is crucial since by pruning the search space it cuts down the search space by magnitudes therefore significantly speeds up the query. The space pruning is configured through the number of partitions and the number of partitions to search. A larger number leads to better recall but larger partition creation time. A good heuristic is to set the number of partitions to be the square root of the number of vectors.

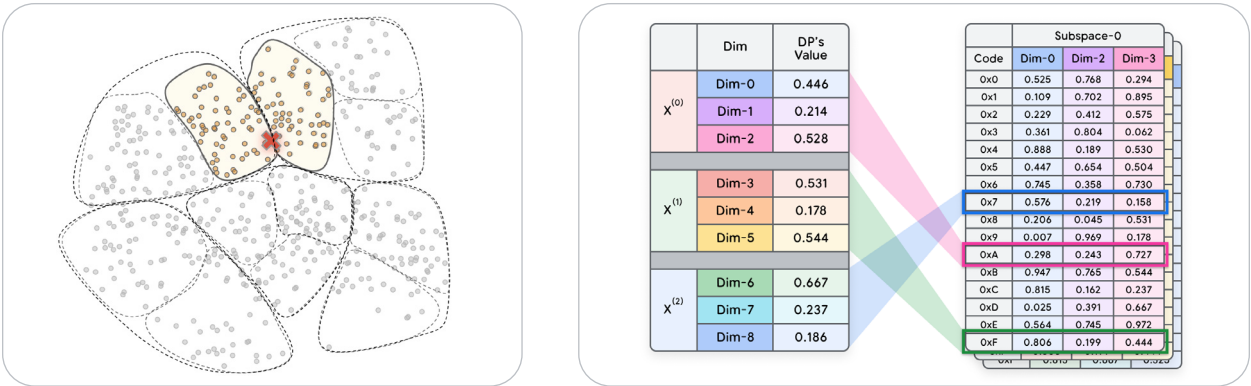


Figure 13. Search space partitioning & pruning(left) & Approximate scoring (right)

At query time ScaNN uses the user-specified distance measure to select the specified number of top partitions (a value specified by the user), and then executes the scoring step next. In this step ScaNN compares the query with all the points in the top partitions and selects the top K'. This distance computation can be configured as exact distance or approximate distance. The approximate distance computation leverages either standard product quantization or anisotropic quantization techniques, the latter of which is a specific method employed by ScaNN which gives the better speed and accuracy tradeoffs.

Finally, as a last step the user can optionally choose to rescore the user specified top K number of results more accurately. This results in an industry leading speed/accuracy tradeoff ScaNN is known for as can be inferred from Figure 14. Snippet 10 shows a code example.

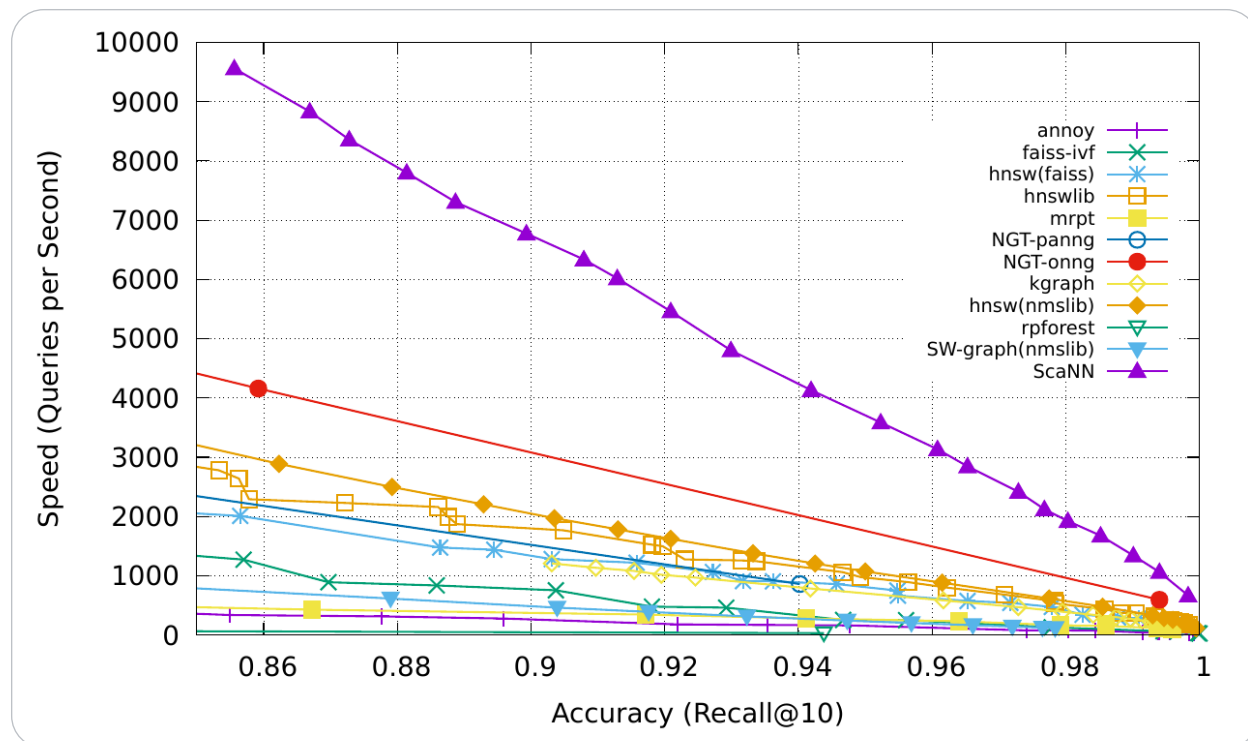


Figure 14. Accuracy/speed tradeoffs for various SOTA ANN search algorithms

```
import tensorflow as tf
import tensorflow_recommenders as tfrs
from vertexai.language_models import TextEmbeddingModel, 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, "RETRIEVAL_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)
```

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

In this whitepaper we have seen both State-of-the-Art SOTA and traditional ANN search algorithms: ScaNN, FAISS, LSH, KD-Tree, and Ball-tree, and examined the great speed/accuracy tradeoffs that they provide. However, to use these algorithms they need to be deployed in a scalable, secure and production-ready manner. For that we need vector databases.

Vector databases

Vector embeddings embody semantic meanings of data, while vector search algorithms provide a means for efficiently querying them. Historically traditional databases lacked the means to combine semantic meaning and efficient querying in a way that the most relevant embeddings can be both stored, queried, and retrieved in a secure, scalable, and flexible manner for complex analysis and real-time enterprise grade applications. This is what gave rise to vector databases, which are built ground-up to manage these embeddings for production scenarios. Due to the recent popularity of Generative AI, an increasing number of traditional databases are starting to incorporate supporting vector search functionality as well in addition to traditional search ('hybrid search') functionalities. Let's look at the workflow for a simple Vector Database, with hybrid search capabilities.

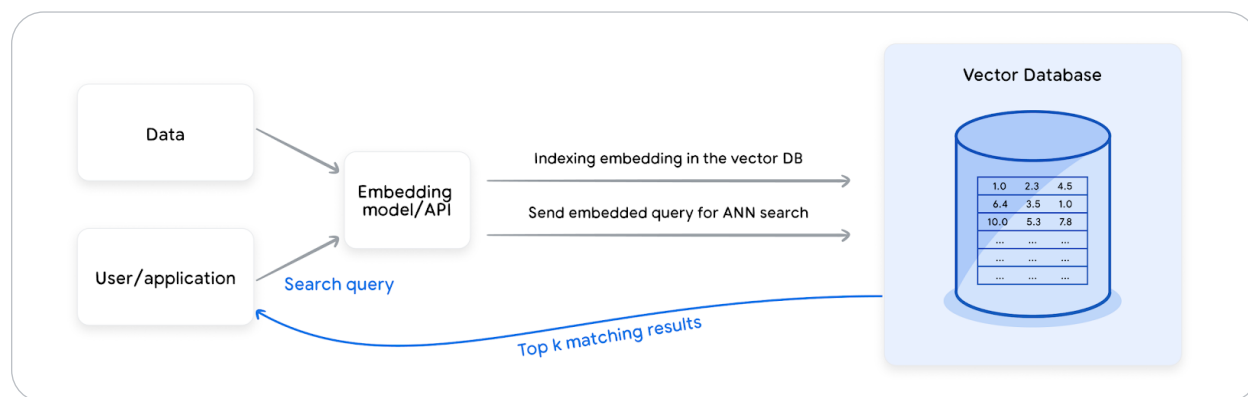


Figure 15. Populating and querying vector databases

Each vector database differs in its implementation, but the general flow is shown in Figure 15:

1. An appropriate trained embedding model is used to embed the relevant data points as vectors with fixed dimensions.
2. The vectors are then augmented with appropriate metadata and complementary information (such as tags) and indexed using the specified algorithm for efficient search.
3. An incoming query gets embedded with the same model, and used to query and return specific amounts of the most semantically similar items and their associated unembedded content/metadata. Some databases might provide caching and pre-filtering (based on tags) and post-filtering capabilities (reranking using another more accurate model) to further enhance the query speed and performance.

There are quite a few vector databases available today, each tailored to different business needs and considerations. A few good examples of commercially managed vector databases include Google Cloud's Vertex Vector Search,³⁵ Google Cloud's AlloyDB & Cloud SQL Postgres ElasticSearch,³⁶ and Pinecone³⁷ to name a few. Vertex AI Vector Search is a vector database built by Google that uses the ScaNN algorithm for fast vector search, while still maintaining all the security and access guarantees of Google Cloud. AlloyDB & Cloud SQL Postgres supports vector search through the OSS pgvector³⁸ extension, which allows for SQL queries to combine ANN search with traditional predicates and the usual transactional semantics for ANN search index. AlloyDB also has a ScaNN index extension that is a native implementation of ScaNN and is pgvector-compatible. Similarly, many of the other traditional databases have also started to add plugins to enable vector search. Pinecone and Weaviate leverage HNSW for their fast vector search in addition to the ability to filter data using

traditional search. Amongst their open source peers: Weaviate³⁹ and ChromaDB⁴⁰ provide a full suite of functionality upon deployment and can be tested in memory as well during the prototyping phase.

Operational considerations

Vector Databases are critical to managing the majority of technical challenges that arise with storing and querying embeddings at scale. Some of these challenges are specific to the nature of vector stores, while others overlap with that of traditional databases. These include horizontal and vertical scalability, availability, data consistency, real time updates, backups, access control, compliance, and much more. However, there are also many more challenges and considerations you need to take into account while using embedding and vector stores.

Firstly, embeddings, unlike traditional content, can mutate over time. This means that the same text, image, video or other content could and should be embedded using different embedding models to optimize for the performance of the downstream applications. This is especially true for embeddings of supervised models after the model is retrained to account for various drifts or changing objectives. Similarly, the same applies to unsupervised models when they are updated to a newer model. However, frequently updating the embeddings - especially those trained on large amounts of data - can be prohibitively expensive. Consequently, a balance needs to be struck. This necessitates a well-defined automated process to store, manage, and possibly purge embeddings from the vector databases taking the budget into consideration.

Secondly, while embeddings are great at representing semantic information, sometimes they can be suboptimal at representing literal or syntactic information. This is especially true for domain-specific words or IDs. These values are potentially missing or underrepresented in the data the embeddings models were trained on. For example, if a user enters a query that contains the ID of a specific number along with a lot of text, the model might find semantically similar neighbors which match the meaning of the text closely, but not the ID, which is the most important component in this context. You can overcome this challenge by using a combination of full-text search to pre-filter or post-filter the search space before passing it onto the semantic search module.

Another important point to consider is that depending on the nature of the workload in which the semantic query occurs, it might be worth relying on different vector databases. For example, for OLTP workloads that require frequent reads/write operations, an operational database like Postgres or CloudSQL is the best choice. For large-scale OLAP analytical workloads and batch use cases, using Bigquery's vector search is preferable.

In conclusion, a variety of factors need to be considered when choosing a vector database. These factors include size and type of your dataset (some are good at sparse and others dense), business needs, the nature of the workload, budget, security, privacy guarantees, the needs for semantic and syntactic search as well as the database systems that are already in use. In this section we have seen the various ANN search approaches as well the need and benefits of vector databases. The next section demonstrates an example of using a Vector AI Vector Search for semantic search.

Applications

Embeddings models are one of the fundamental machine learning models that power a variety of applications. We summarize some popular applications in the following table.

Task	Description
Retrieval	Given a query and a set of objects (for example, documents, images, and videos), retrieve the most relevant objects. Based on the definition of relevant objects, the subtasks include question answering and recommendations.
Semantic text similarity	Determine whether two sentences have the same semantic meaning. The subtasks include: paraphrasing, duplicate detection, and bitext mining.
Classification	Classify objects into possible categories. Based on the number of labels, the subtasks include binary classification, multi-class classification, and multilabel classifications.
Clustering	Cluster objects together.
Reranking	Rerank a set of objects based on a certain query.

Embeddings together with vector stores providing ANN can be powerful tools which can be used for a variety of applications. These include Retrieval augmented Generation for LLMs, Search, Recommendation Systems, Anomaly detection, few shot- classification and much more.

For ranking problems like search and recommendations, embeddings are normally used at the first stage of the process. They retrieve the potentially good candidates that are semantically similar and consequently improve the relevance of search results. Since the amount of information to sort through can be quite large (in some cases even millions or billions) ANN techniques like ScaNN greatly aids in scalably narrowing the search space.

Let's look at an application which combines both LLMs and RAG to help answer questions.

Q & A with sources (retrieval augmented generation)

Retrieval augmented generation (RAG) for Q&A is a technique that combines the best of both worlds from retrieval and generation. It first retrieves relevant documents from a knowledge base and then uses prompt expansion to generate an answer from those documents. Prompt expansion is a technique that when combined with database search can be very powerful. With prompt expansion the model retrieves relevant information from the database (mostly using a combination of semantic search and business rules), and augments the original prompt with it. The model uses this augmented prompt to generate much more interesting, factual, and informative content than with retrieval or generation alone.

RAGs can help with a common problem with LLMs: their tendency to ‘hallucinate’ and generate factually incorrect but plausible sounding responses. Although RAG can reduce hallucinations, it does not completely eliminate them. What can help mitigate this problem further is to also return the sources from the retrieval and do a quick coherence check either by a human or an LLM. This ensures the LLM response is consistent with the semantically relevant sources. Let’s look at an example (Snippet 11 and 12) of RAG with sources, which can be scalably implemented using Vertex AI LLM text embeddings and Vertex AI Vector Search in conjunction with libraries like langchain.⁴¹ We start with the initial setup in Snippet 11.

```
# 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 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

aiplatform.init(project=PROJECT_ID, location=REGION, staging_bucket=BUCKET_URI)
embedding_model = VertexAIEmbeddings(model_name="textembedding-gecko@003")

# NOTE : This operation can take upto 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_UPDATE , STREAM_UPDATE
)

# 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
)
```

Continues next page...

```
my_index_endpoint.deployed_indexes

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

# TODO : replace 1234567890123456789 with your actual 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.",
]

# 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()

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

Snippet 11. Setting up the network and environment

Then we setup and initialize the Vector AI Vector Search engine ANN index using Vertex text embeddings and then use the Vertex LLMs to do prompt expansion using semantic search. This both grounds the LLMs in factuality and provides sources as well (Snippet 13).

```
# Create dummy embeddings to initialize the vector store
embeddings_vx = VertexAIEmbeddings()
initial_config = {
    "id": str(uuid.uuid4()),
    "embedding": [float(x) for x in list(embeddings_vx.embed_documents(test_items)[0])],
}

with open("data.json", "w") as f:
    json.dump(initial_config, f)
#magic command to be run on terminal or jupyter notebooks
!gsutil cp data.json {EMBEDDING_DIR}/file.json
# Create dummy embeddings to initialize the vector store
aiplatform.init(project=PROJECT_ID, location=REGION, staging_bucket=BUCKET_URI)
my_index = aiplatform.MatchingEngineIndex.create_tree_ah_index(
    display_name=DISPLAY_NAME,
    contents_delta_uri=EMBEDDING_DIR,
    dimensions=DIMENSIONS,
    leafNodeEmbeddingCount=1000,
    fractionLeafNodesToSearch=0.1,
    approximate_neighbors_count=2,
    distance_measure_type="DOT_PRODUCT_DISTANCE")

my_index_endpoint = aiplatform.MatchingEngineIndexEndpoint.create(
    display_name=f"{DISPLAY_NAME}-endpoint",
    network=VPC_NETWORK_FULL)
my_index_endpoint = my_index_endpoint.deploy_index(
    index=my_index, deployed_index_id=DEPLOYED_INDEX_ID
)

#initialize Langchain retriever and add text embeddings to index
texts = [
    "The earth is spherical.",
    "The earth is a planet.",
    "I like to eat at a restaurant.",
]
vector_store = MatchingEngine.from_components(
    project_id=PROJECT_ID,
    region=REGION,
```

Continues next page...

```

gcs_bucket_name=BUCKET_URI,
index_id=my_index.name,
endpoint_id=my_index_endpoint.name,
embedding=embeddings_vx
)
vector_store.add_texts(texts=texts)
retriever=vector_store.as_retriever(search_kwargs={'k':1 })

#Create Retrieval augmented few-shot prompts to provide context to ground LLMs
prompt_template="""You are David, an AI knowledge bot.
Answer the questions using the facts provided. Use the following 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 = {"prompt": prompt}
llm = VertexAI()
#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)
query = "What shape is the planet where humans live?"
result = chain(query)
display(Markdown(print_result(result)))

```

Snippet 12. Build/deploy ANN Index for Vertex Matching engine and use RAG with LLM prompts to generate grounded results/sources

<p>Question:</p> <p>What shape is the planet where humans live?</p> <p>Answer:</p> <p>The planet where humans live is spherical.</p> <p>Sources:</p> <p>The earth is spherical.</p>	<p>Question:</p> <p>What shape is pluto?</p> <p>Answer:</p> <p>I don't know. The shape of Pluto is not mentioned in the context.</p> <p>Source:</p> <p>The earth is a planet.</p>
--	--

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

As we can infer from Figure 16, the output not only grounds LLM in the semantically similar results retrieved from the database (hence refusing to answer when context cannot be found in the database). This not only significantly reduces hallucination, but also provides sources for verification, either human or using another LLM.

Summary

In this whitepaper we have discussed various methods to create, manage, store, and retrieve embeddings of various data modalities effectively in the context of production-grade applications. Creating, maintaining and using embeddings for downstream applications can be a complex task that involves several roles in the organization. However, by thoroughly operationalizing and automating its usage, you can safely leverage the incredible benefits they offer across some of the most important applications. Some key takeaways from this whitepaper include:

1. Choose your embedding model wisely for your data and use case. Ensure the data used in inference is consistent with the data used in training. The distribution shift from training to inference can come from various areas, including domain distribution shift or downstream

task distribution shift. If no existing embedding models fit the current inference data distribution, fine-tuning the existing model can significantly help on the performance. Another tradeoff comes from the model size. The large deep neural network (large multimodal models) based models usually have better performance but can come with a cost of longer serving latency. Using Cloud-based embedding services can conquer the above issue by providing both high-quality and low-latency embedding service. For most business applications using a pre-trained embedding model provides a good baseline, which can be further fine-tuned or integrated in downstream models. In case the data has an inherent graph structure, graph embeddings can provide superior performance.

2. Once your embedding strategy is defined, it's important to make the choice of the appropriate vector database that suits your budget and business needs. It might seem quicker to prototype with available open source alternatives, but opting for a more secure, scalable, and battle-tested managed vector database is certain to be better off in the long term. There are various open source alternatives using one of the many powerful ANN vector search algorithms, but ScaNN and HNSW have proven to provide some of the best accuracy and performance trade offs in that order.
3. Embeddings combined with an appropriate ANN powered vector database is an incredibly powerful tool and can be leveraged for various applications, including Search, Recommendation systems, and Retrieval augment generation for LLMs. This approach can mitigate the hallucination problem and bolster verifiability and trust of LLM-based systems.

Endnotes

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. TensorFlow, 2021, TensorFlow Models: NLP, Fine-tune BERT. Available at: https://www.tensorflow.org/tfmodels/nlp/fine_tune_bert.
27. 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>.
28. 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>.
29. Scikit-learn, 2021, Scikit-learn, a library for unsupervised and supervised neighbors-based learning methods. Available at: <https://scikit-learn.org/>.
30. Lshashing, 2021, An open source python library to perform locality sensitive hashing. Available at: <https://pypi.org/project/lshashing/>.
31. 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>.
32. 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>.
33. 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>.
34. TensorFlow, 2021, TensorFlow Recommenders, an open source library for building ranking & recommender system models. Available at: <https://www.tensorflow.org/recommenders>.
35. 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>.
36. Elasticsearch, 2021, Elasticsearch: a RESTful search and analytics engine. Available at: <https://www.elastic.co/elasticsearch/>.
37. Pinecone, 2021, Pinecone, a commercial fully managed vector database. Available at: <https://www.pinecone.io>.
38. pgvector, 2021, Open Source vector similarity search for Postgres. Available at: <https://github.com/pgvector/pgvector>.
39. Weaviate, 2021, Weaviate, an open source vector database. Available at: <https://weaviate.io/>.

40. ChromaDB, 2021, ChromaDB, an open source vector database. Available at: <https://www.trychroma.com/>.

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