GenAI: Best Practices

Release 1.0

Wenqiang Feng and Di Zhen

December 16, 2024

CONTENTS

Welcome to our **GenAI: Best Practices**!!! The PDF version can be downloaded from HERE.

CHAPTER

ONE

PREFACE

Chinese proverb

Good tools are prerequisite to the successful execution of a job. – old Chinese proverb

1.1 About

1.1.1 About this book

This is the book for our Generative AI: Best practics [GenAI]. The PDF version can be downloaded from HERE. **You may download and distribute it. Please beaware, however, that the note contains typos as well as inaccurate or incorrect description.**

In this book, I aim to demonstrate best practices for Generative AI through detailed demo code and practical examples. If you notice that your work has not been properly cited, please do not hesitate to reach out and let me know.

1.1.2 About the authors

- **Wenqiang Feng**
  - Sr. Mgr Data Enginner and PhD in Mathematics
  - University of Tennessee at Knoxville
  - Webpage: http://web.utk.edu/~wfeng1/
  - Email: von198@gmail.com

- **Biography**

  Wenqiang Feng is the Senior Manager of Data Engineering and former Director of AI Engineering/Data Science at American Express (AMEX). Before his tenure at AMEX, Dr. Feng served as a Senior Data Scientist in the Machine Learning Lab at H&R Block and as a Data Scientist at Applied Analytics Group, DST (now SS&C). Throughout his career, Dr. Feng has focused on equipping clients with cutting-edge skills and technologies, including Big Data analytics, advanced modeling techniques, and data enhancement strategies.

  Dr. Feng brings extensive expertise in data mining, analytic systems, machine learning algorithms, business intelligence, and the application of Big Data tools to solve complex, cross-functional industry challenges. Prior to his role at DST, Dr. Feng was an IMA Data Science Fellow at the Institute for Mathematics and its Applications (IMA) at the University of Minnesota. In this capacity, he collaborated with startups to develop predictive analytics solutions that informed strategic marketing decisions.

Dr. Feng holds a Ph.D. in Computational Mathematics and a Master's degree in Statistics from the University of Tennessee, Knoxville. He also earned a Master's degree in Computational Mathematics from Missouri University of Science and Technology (MST) and a Master's degree in Applied Mathematics from the University of Science and Technology of China (USTC).

- **Declaration**

  The work of Wenqiang Feng was supported by the IMA, while working at IMA. However, any opinion, finding, and conclusions or recommendations expressed in this material are those of the author and do not necessarily reflect the views of the IMA, UTK and DST.

> **Caution**
>
> ChatGPT has been extensively used in the creation of this book. If you notice that your work has not been cited or has been cited incorrectly, please notify us.

1.2 Feedback and suggestions

Your comments and suggestions are highly appreciated. I am more than happy to receive corrections, suggestions or feedback through email (Wenqiang Feng: von198@gmail.com and Di Zhen: dizhen318@gmail.com) for improvements.

CHAPTER TWO

PRELIMINARY

In this chapter, we will introduce some math and NLP preliminaries which is highly used in Generative AI.

2.1 Math Preliminary

2.1.1 Vector

A vector is a mathematical representation of data characterized by both magnitude and direction. In this context, each data point is represented as a feature vector, with each component corresponding to a specific feature or attribute of the data.

```
import numpy as np
import gensim.downloader as api
# Download pre-trained GloVe model
glove_vectors = api.load("glove-twitter-25")

# Get word vectors (embeddings)
word1 = "king"
word2 = "queen"

# embedding
king = glove_vectors[word1]
queen = glove_vectors[word2]

print('king:\n', king)
print('queen:\n', queen)
```

```
king:
[-0.74501  -0.11992   0.37329   0.36847  -0.4472   -0.2288    0.70118
0.82872   0.39486  -0.58347   0.41488   0.37074  -3.6906   -0.20101
0.11472  -0.34661   0.36208   0.095679 -0.01765   0.68498  -0.049013
0.54049  -0.21005  -0.65397   0.64556 ]
queen:
[-1.1266   -0.52064   0.45565   0.21079  -0.05081  -0.65158   1.1395
0.69897  -0.20612  -0.71803  -0.02811   0.10977  -3.3089   -0.49299
```

(continues on next page)

(continued from previous page)

```
-0.51375   0.10363  -0.11764  -0.084972  0.02558   0.6859   -0.29196
0.4594   -0.39955  -0.40371   0.31828 ]
```

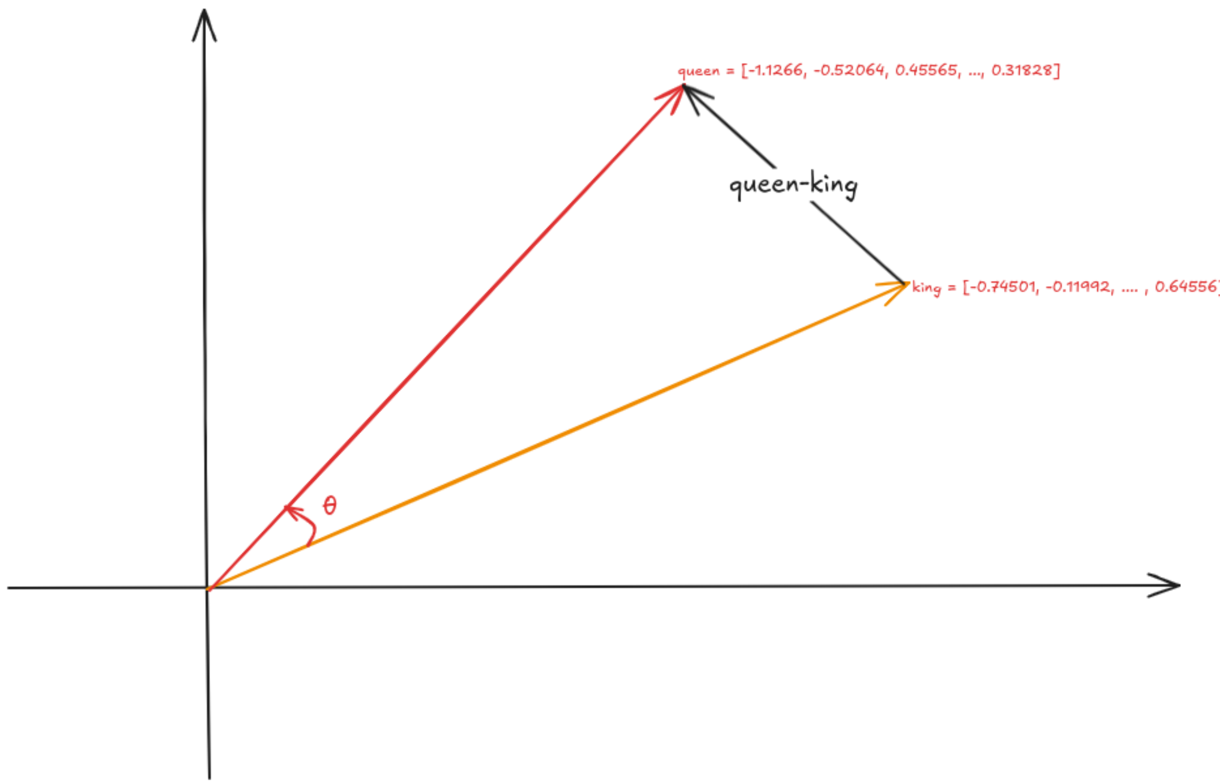


Fig. 1: Vector

2.1.2 Norm

A norm is a function that maps a vector to a single positive value, representing its magnitude. Norms are essential for calculating distances between vectors, which play a crucial role in measuring prediction errors, performing feature selection, and applying regularization techniques in models.

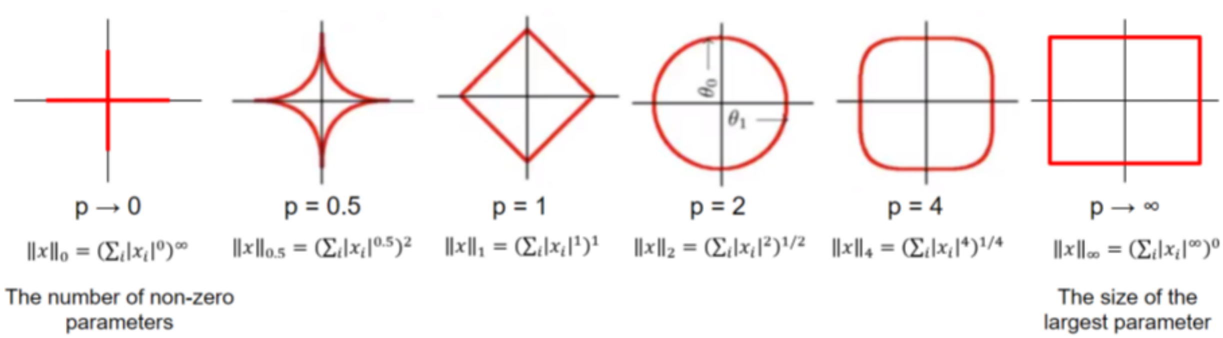


Fig. 2: Geometrical Interpretation of Norm (source_1)

- Formula:

The ℓ^p norm for $\vec{v} = (v_1, v_2, \cdots, v_n)$ is

$$||\vec{v}||_p = \sqrt[p]{|v_1|^p + |v_2|^p + \cdots + |v_n|^p}$$

- ℓ^1 norm: Sum of absolute values of vector components, often used for feature selection due to its tendency to produce sparse solutions.

```
# l1 norm
np.linalg.norm(king,ord=1) #    max(sum(abs(x), axis=0))

### 13.188952
```

- ℓ^2 norm: Square root of the sum of squared vector components, the most common norm used in many machine learning algorithms.

```
# l2 norm
np.linalg.norm(king,ord=2)

### 4.3206835
```

- ℓ^∞ norm (Maximum norm): The largest absolute value of a vector component.

2.1.3 Distances

- Manhattan Distance (ℓ^1 Distance)

  Also known as taxicab or city block distance, Manhattan distance measures the absolute differences between the components of two vectors. It represents the distance a point would travel along grid lines in a Cartesian plane, similar to navigating through city streets.

  For two vector $\vec{u} = (u_1, u_2, \cdots, u_n)$ and $\vec{v} = (v_1, v_2, \cdots, v_n)$, the Manhattan Distance distance $d(\vec{u}, \vec{v})$ is

$$d(\vec{u}, \vec{v}) = ||\vec{u} - \vec{v}||_1 = |u_1 - v_1| + |u_2 - v_2| + \cdots + |u_n - v_n|$$

- Euclidean Distance (ℓ^2 Distance)

  Euclidean distance is the most common way to measure the distance between two points (vectors) in space. It is essentially the straight-line distance between them, calculated using the Pythagorean theorem.

  For two vector $\vec{u} = (u_1, u_2, \cdots, u_n)$ and $\vec{v} = (v_1, v_2, \cdots, v_n)$, the Euclidean Distance distance $d(\vec{u}, \vec{v})$ is

$$d(\vec{u}, \vec{v}) = ||\vec{u} - \vec{v}||_2 = \sqrt{(u_1 - v_1)^2 + (u_2 - v_2)^2 + \cdots + (u_n - v_n)^2}$$

- Minkowski Distance (ℓ^p Distance)

  Minkowski distance is a generalization of both Euclidean and Manhattan distances. It incorporates a parameter, p, which allows for adjusting the sensitivity of the distance metric.

- Cos Similarity

  Cosine similarity measures the angle between two vectors rather than their straight-line distance. It evaluates the similarity of two vectors by focusing on their orientation rather than their magnitude. This makes it particularly useful for high-dimensional data, such as text, where the direction of the vectors is often more significant than their magnitude.

  The Cos similarity for two vector $\vec{u} = (u_1, u_2, \cdots, u_n)$ and $\vec{v} = (v_1, v_2, \cdots, v_n)$ is

  $$cos(\theta) = \frac{\vec{u} \cdot \vec{v}}{||\vec{u}||||\vec{v}||}$$

  - 1 means the vectors point in exactly the same direction (perfect similarity).
  - 0 means they are orthogonal (no similarity).
  - -1 means they point in opposite directions (complete dissimilarity).

```
# Compute cosine similarity between the two word vectors
np.dot(king,queen)/(np.linalg.norm(king)*np.linalg.norm(queen))

### 0.92024213
```

```
# Compute cosine similarity between the two word vectors
similarity = glove_vectors.similarity(word1, word2)
print(f"Word vectors for '{word1}': {king}")
print(f"Word vectors for '{word2}': {queen}")
print(f"Cosine similarity between '{word1}' and '{word2}':
↪{similarity}")
```

```
Word vectors for 'king': [-0.74501  -0.11992   0.37329   0.36847  -
↪0.4472   -0.2288    0.70118
0.82872   0.39486  -0.58347   0.41488   0.37074  -3.6906   -0.20101
0.11472  -0.34661   0.36208   0.095679 -0.01765   0.68498  -0.049013
0.54049  -0.21005  -0.65397   0.64556 ]
Word vectors for 'queen': [-1.1266   -0.52064   0.45565   0.21079  -
↪0.05081  -0.65158   1.1395
0.69897  -0.20612  -0.71803  -0.02811   0.10977  -3.3089   -0.49299
-0.51375   0.10363  -0.11764  -0.084972  0.02558   0.6859   -0.29196
0.4594   -0.39955  -0.40371   0.31828 ]
Cosine similarity between 'king' and 'queen': 0.920242190361023
```

2.2 NLP Preliminary

2.2.1 Vocabulary

In Natural Language Processing (NLP), **vocabulary** refers to the complete set of unique words or tokens that a model recognizes or works with during training and inference. Vocabulary plays a critical role in text processing and understanding, as it defines the scope of linguistic units a model can handle.

- Types of Vocabulary in NLP

  1. **Word-level Vocabulary**: - Each word in the text is treated as a unique token. - For example, the sentence "I love NLP" would generate the vocabulary: `{I, love, NLP}`.

  2. **Subword-level Vocabulary**: - Text is broken down into smaller units like prefixes, suffixes, or character sequences. - For example, the word "loving" might be split into `{lov, ing}` using techniques like Byte Pair Encoding (BPE) or SentencePiece. - Subword vocabularies handle rare or unseen words more effectively.

  3. **Character-level Vocabulary**: - Each character is treated as a token. - For example, the word "love" would generate the vocabulary: `{l, o, v, e}`.

- Importance of Vocabulary

  1. **Text Representation**: - Vocabulary is the basis for converting text into numerical representations like one-hot vectors, embeddings, or input IDs for machine learning models.

  2. **Model Efficiency**: - A larger vocabulary increases the model's memory and computational requirements. - A smaller vocabulary may lack the capacity to represent all words effectively, leading to a loss of meaning.

  3. **Handling Out-of-Vocabulary (OOV) Words**: - Words not present in the vocabulary are either replaced with a special token like <UNK> or processed using subword/character-based techniques.

- Building a Vocabulary

  Common practices include:

  1. Tokenizing the text into words, subwords, or characters.
  2. Counting the frequency of tokens.
  3. Keeping only the most frequent tokens up to a predefined size (e.g., top 50,000 tokens).
  4. Adding special tokens like <PAD>, <UNK>, <BOS> (beginning of sentence), and <EOS> (end of sentence).

- Challenges
- **Balancing Vocabulary Size**: A larger vocabulary increases the richness of representation but requires more computational resources.
- **Domain-specific Vocabularies**: In specialized fields like medicine or law, standard vocabularies may not be sufficient, requiring domain-specific tokenization strategies.

2.2.2 Tagging

Tagging in NLP refers to the process of assigning labels or annotations to words, phrases, or other linguistic units in a text. These labels provide additional information about the syntactic, semantic, or structural role of the elements in the text.

- Types of Tagging

  1. **Part-of-Speech (POS) Tagging**: - Assigns grammatical tags (e.g., noun, verb, adjective) to each word in a sentence. - Example: For the sentence "The dog barks," the tags might be:

- `The/DET` (Determiner)
- `dog/NOUN` (Noun)
- `barks/VERB` (Verb).

2. **Named Entity Recognition (NER) Tagging**: - Identifies and classifies named entities in a text, such as names of people, organizations, locations, dates, or monetary values. - Example: In the sentence "John works at Google in California," the tags might be:
   - `John/PERSON`
   - `Google/ORGANIZATION`
   - `California/LOCATION`.
3. **Chunking (Syntactic Tagging)**: - Groups words into syntactic chunks like noun phrases (NP) or verb phrases (VP). - Example: For the sentence "The quick brown fox jumps," a chunking result might be:
   - `[NP The quick brown fox] [VP jumps]`.
4. **Sentiment Tagging**: - Assigns sentiment labels (e.g., positive, negative, neutral) to words, phrases, or entire documents. - Example: The word "happy" might be tagged as `positive`, while "sad" might be tagged as `negative`.
5. **Dependency Parsing Tags**: - Identifies the grammatical relationships between words in a sentence, such as subject, object, or modifier. - Example: In "She enjoys cooking," the tags might show:
   - `She/nsubj` (nominal subject)
   - `enjoys/ROOT` (root of the sentence)
   - `cooking/dobj` (direct object).

- Importance of Tagging
  - **Understanding Language Structure**: Tags help NLP models understand the grammatical and syntactic structure of text.
  - **Improving Downstream Tasks**: Tagging is foundational for tasks like machine translation, sentiment analysis, question answering, and summarization.
  - **Feature Engineering**: Tags serve as features for training machine learning models in text classification or sequence labeling tasks.
- Tagging Techniques
  1. **Rule-based Tagging**: Relies on predefined linguistic rules to assign tags. Example: Using dictionaries or regular expressions to match specific patterns.
  2. **Statistical Tagging**: Uses probabilistic models like Hidden Markov Models (HMMs) to predict tags based on word sequences.
  3. **Neural Network-based Tagging**: Employs deep learning models like LSTMs, GRUs, or Transformers to tag text with high accuracy.
- Challenges

  - **Ambiguity**:Words with multiple meanings can lead to incorrect tagging. Example: The word "bank" could mean a financial institution or a riverbank.
  - **Domain-Specific Language**: General tagging models may fail to perform well on specialized text like medical or legal documents.
  - **Data Sparsity**: Rare words or phrases may lack sufficient training data for accurate tagging.

2.2.3 Lemmatization

Lemmatization in NLP is the process of reducing a word to its base or dictionary form, known as the **lemma**. Unlike stemming, which simply removes word suffixes, lemmatization considers the context and grammatical role of the word to produce a linguistically accurate root form.

- How Lemmatization Works
  1. **Contextual Analysis**: - Lemmatization relies on a vocabulary (lexicon) and morphological analysis to identify a word's base form. - For example:
     - `running` → `run`
     - `better` → `good`
  2. **Part-of-Speech (POS) Tagging**: - The process uses POS tags to determine the correct lemma for a word. - Example:
     - `barking` (verb) → `bark`
     - `barking` (adjective, as in "barking dog") → `barking`.
- Importance of Lemmatization
  1. **Improves Text Normalization**: - Lemmatization helps normalize text by grouping different forms of a word into a single representation. - Example:
     - `run`, `running`, and `ran` → `run`.
  2. **Enhances NLP Applications**: - Lemmatized text improves the performance of tasks like information retrieval, text classification, and sentiment analysis.
  3. **Reduces Vocabulary Size**: - By mapping inflected forms to their base form, lemmatization reduces redundancy in text, resulting in a smaller vocabulary.
- Lemmatization vs. Stemming
  - **Lemmatization**: - Produces linguistically accurate root forms. - Considers the word's context and POS. - Example:
    * `studies` → `study`.
  - **Stemming**: - Applies heuristic rules to strip word suffixes without considering context. - May produce non-dictionary forms. - Example:
    * `studies` → `studi`.
- Techniques for Lemmatization

1. **Rule-Based Lemmatization**: - Relies on predefined linguistic rules and dictionaries. - Example: WordNet-based lemmatizers.
2. **Statistical Lemmatization**: - Uses probabilistic models to predict lemmas based on the context.
3. **Deep Learning-Based Lemmatization**: - Employs neural networks and sequence-to-sequence models for highly accurate lemmatization in complex contexts.

- Challenges
  - **Ambiguity**: Words with multiple meanings may result in incorrect lemmatization without proper context. - Example:
    - `left` (verb) → `leave`
    - `left` (noun/adjective) → `left`.
  - **Language-Specific Complexity**: Lemmatization rules vary widely across languages, requiring language-specific tools and resources.
  - **Resource Dependency**: Lemmatizers require extensive lexicons and morphological rules, which can be resource-intensive to develop.

2.2.4 Tokenization

Tokenization in NLP refers to the process of splitting a text into smaller units, called **tokens**, which can be words, subwords, sentences, or characters. These tokens serve as the basic building blocks for further analysis in NLP tasks.

- Types of Tokenization

1. **Word Tokenization**:

- Splits the text into individual words or terms.
- **Example:**
  - Sentence: "I love NLP."
  - Tokens: `["I", "love", "NLP"]`.

2. **Sentence Tokenization**:

- Divides a text into sentences.
- **Example:**
  - Text: "I love NLP. It's amazing."
  - Tokens: `["I love NLP.", "It's amazing."]`.

3. **Subword Tokenization**:

- Breaks words into smaller units, often using methods like Byte Pair Encoding (BPE) or SentencePiece.
- **Example:**
  - Word: `unhappiness`.

        * Tokens: `["un", "happiness"]` (or subword units like `["un", "happi", "ness"]`).

4. **Character Tokenization**:

   - Treats each character in a word as a separate token.
   - **Example:**

        * Word: `hello`.
        * Tokens: `["h", "e", "l", "l", "o"]`.

- Importance of Tokenization

  1. **Text Preprocessing**:

  - Tokenization is the first step in many NLP tasks like text classification, translation, and summarization, as it converts text into manageable pieces.

  2. **Text Representation**:

  - Tokens are converted into numerical representations (e.g., word embeddings) for model input in tasks like sentiment analysis, named entity recognition (NER), or language modeling.

  3. **Improving Accuracy**:

  - Proper tokenization ensures that a model processes text at the correct granularity (e.g., words or subwords), improving accuracy for tasks like machine translation or text generation.

- Challenges of Tokenization

  1. **Ambiguity**:

  - Certain words or phrases can be tokenized differently based on context.
  - Example: "New York" can be treated as one token (location) or two separate tokens (`["New", "York"]`).

  2. **Handling Punctuation**:

  - Deciding how to treat punctuation marks can be challenging. For example, should commas, periods, or quotes be treated as separate tokens or grouped with adjacent words?

  3. **Multi-word Expressions (MWEs)**:

  - Some expressions consist of multiple words that should be treated as a single token, such as "New York" or "machine learning."

- Techniques for Tokenization

  1. **Rule-Based Tokenization**: Uses predefined rules to split text based on spaces, punctuation, and other delimiters.
  2. **Statistical and Machine Learning-Based Tokenization**: Uses trained models to predict token boundaries based on patterns learned from large corpora.
  3. **Deep Learning-Based Tokenization**: Modern tokenization models, such as those used in transformers (e.g., BERT, GPT), may rely on subword tokenization and neural networks to handle complex tokenization tasks.

2.2.5 BERT Tokenization

- Vocabulary: The BERT Tokenizer's vocabulary contains 30,522 unique tokens.

```
from transformers import BertTokenizer, BertModel
tokenizer = BertTokenizer.from_pretrained('bert-base-uncased')
# model = BertModel.from_pretrained("bert-base-uncased")

# vocabulary size
print(tokenizer.vocab_size)

# vocabulary
print(tokenizer.vocab)
```

```
# vocabulary size
30522

# vocabulary
OrderedDict([('[PAD]', 0), ('[unused0]', 1)
              ...........,
              ('writing', 3015), ('bay', 3016),
              ...........,
              ('##', 30520), ('##', 30521)])
```

- Tokens and IDs
  - Tokens to IDs

```
text = "Gen AI is awesome"
encoded_input = tokenizer(text, return_tensors='pt')

# tokens to ids
print(encoded_input)

# output
{'input_ids': tensor([[  101,  8991,  9932,  2003, 12476,   102]]), \
'token_type_ids': tensor([[0, 0, 0, 0, 0, 0]]), \
'attention_mask': tensor([[1, 1, 1, 1, 1, 1]])}
```

You might notice that there are only four words, yet we have six token IDs. This is due to the inclusion of two additional special tokens [CLS] and [SEP].

```
print({x : tokenizer.encode(x, add_special_tokens=False) for x in [
↪'[CLS]']+ text.split()+ ['[SEP]']})

### output
{'[CLS]': [101], 'Gen': [8991], 'AI': [9932], 'is': [2003], 'awesome':␣
↪[12476], '[SEP]': [102]}
```

– Special Tokens

```
# Special tokens
print({x : tokenizer.encode(x, add_special_tokens=False) for x in [
↪'[CLS]', '[SEP]', '[MASK]', '[EOS]']})

# tokens to ids
{'[CLS]': [101], '[SEP]': [102], '[MASK]': [103], '[EOS]': [1031, 1041,␣
↪2891, 1033]}
```

– IDs to tokens

```
# ids to tokens
token_id = encoded_input['input_ids'].tolist()[0]
print({tokenizer.convert_ids_to_tokens(id, skip_special_
↪tokens=False):id \
      for id in token_id})

### output
{'[CLS]': 101, 'gen': 8991, 'ai': 9932, 'is': 2003, 'awesome': 12476,
↪'[SEP]': 102}
```

– Out-of-vocabulary tokens

```
text = "Gen AI is awesome "
encoded_input = tokenizer(text, return_tensors='pt')

print({x : tokenizer.encode(x, add_special_tokens=False) for x in [
↪'[CLS]']+ text.split()+ ['[SEP]']})
print(tokenizer.convert_ids_to_tokens(100, skip_special_tokens=False))

### output
{'[CLS]': [101], 'Gen': [8991], 'AI': [9932], 'is': [2003], 'awesome':␣
↪[12476], '': [100], '[SEP]': [102]}
[UNK]
```

– Subword Tokenization

```
# Subword Tokenization
text = "GenAI is awesome "
print({x : tokenizer.encode(x, add_special_tokens=False) for x in [
↪'[CLS]']+ text.split()+ ['[SEP]']})
print(tokenizer.convert_ids_to_tokens(100, skip_special_tokens=False))

# output
{'[CLS]': [101], 'GenAI': [8991, 4886], 'is': [2003], 'awesome':␣
↪[12476], '': [100], '[SEP]': [102]}
[UNK]
```

2.3 Platform and Packages

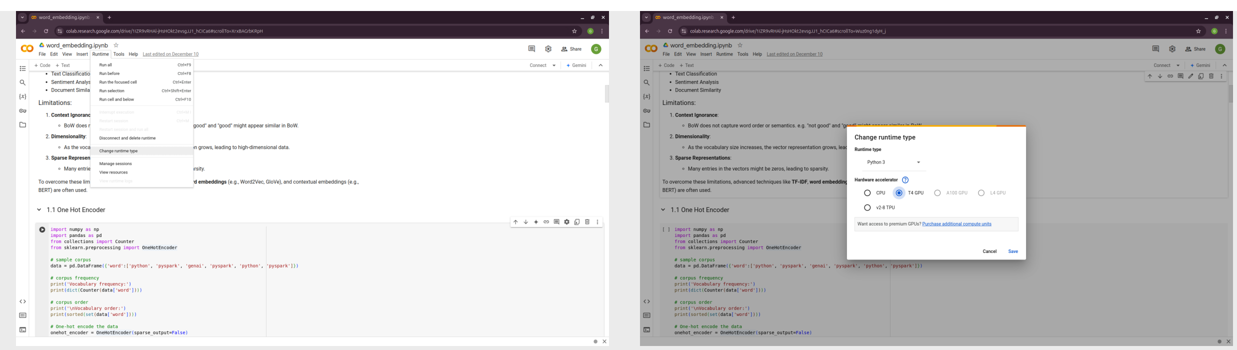
2.3.1 Google Colab

Google Colab (short for Colaboratory) is a free, cloud-based platform that provides users with the ability to write and execute Python code in an interactive notebook environment. It is based on Jupyter notebooks and is powered by Google Cloud services, allowing for seamless integration with Google Drive and other Google services. We will primarily use Google Colab with free T4 GPU runtime throughout this book.

- Key Features

1. **Free Access to GPUs and TPUs** Colab offers free access to Graphics Processing Units (GPUs) and Tensor Processing Units (TPUs), making it an ideal environment for machine learning, deep learning, and other computationally intensive tasks.
2. **Integration with Google Drive** You can store and access notebooks directly from your Google Drive, making it easy to collaborate with others and keep your projects organized.
3. **No Setup Required** Since Colab is entirely cloud-based, you don't need to worry about setting up an environment or managing dependencies. Everything is ready to go out of the box.
4. **Support for Python Libraries** Colab comes pre-installed with many popular Python libraries, including TensorFlow, PyTorch, Keras, and OpenCV, among others. You can also install any additional libraries using *pip*.
5. **Collaborative Features** Multiple users can work on the same notebook simultaneously, making it ideal for collaboration. Changes are synchronized in real-time.
6. **Rich Media Support** Colab supports the inclusion of rich media, such as images, videos, and LaTeX equations, directly within the notebook. This makes it a great tool for data analysis, visualization, and educational purposes.
7. **Easy Sharing** Notebooks can be easily shared with others via a shareable link, just like Google Docs. Permissions can be set for viewing or editing the document.

- GPU Activation `Runtime --> change runtime type --> T4/A100 GPU`



2.3.2 HuggingFace

Hugging Face is a company and open-source community focused on providing tools and resources for NLP and machine learning. It is best known for its popular **Transformers** library, which allows easy access to pre-trained models for a wide variety of NLP tasks. MOreover, Hugging Face's libraries provide simple

Python APIs that make it easy to load models, preprocess data, and run inference. This simplicity allows both beginners and advanced users to leverage cutting-edge NLP models. We will mainly use the embedding models and Large Language Models (LLMs) from **Hugging Face Model Hub** central repository.

2.3.3 Ollama

Ollama is a package designed to run LLMs locally on your personal device or server, rather than relying on external cloud services. It provides a simple interface to download and use AI models tailored for various tasks, ensuring privacy and control over data while still leveraging the power of LLMs.

- Key features of Ollama:
  - Local Execution: Models run entirely on your hardware, making it ideal for users who prioritize data privacy.
  - Pre-trained Models: Offers a curated set of LLMs optimized for local usage.
  - Cross-Platform: Compatible with macOS, Linux, and other operating systems, depending on hardware specifications.
  - Ease of Use: Designed to make setting up and using local AI models simple for non-technical users.
  - Efficiency: Focused on lightweight models optimized for local performance without needing extensive computational resources.

To simplify the management of access tokens for various LLMs, we will use Ollama in Google Colab.

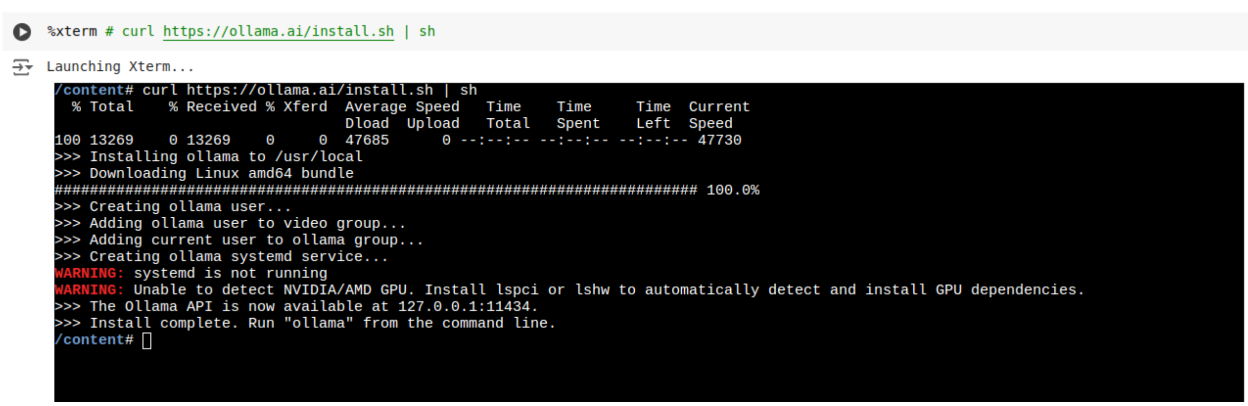
- Ollama installation in Google Colab

  1. colab-xterm

     ```
     !pip install colab-xterm
     %load_ext colabxterm
     ```

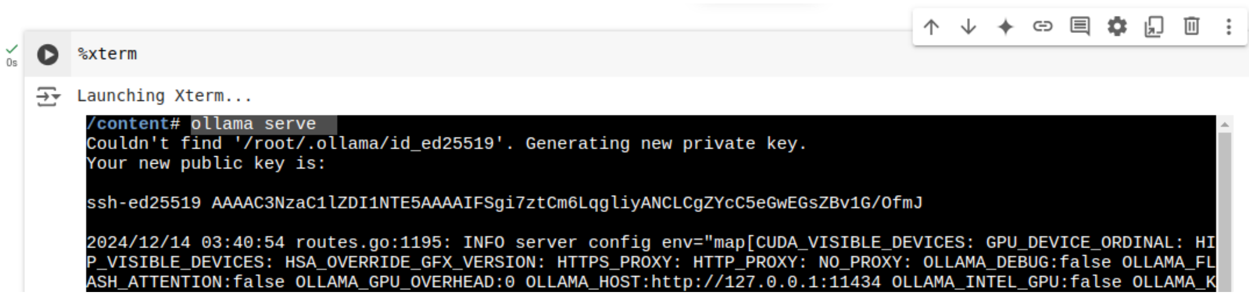
  2. download ollama

     ```
     /content# curl https://ollama.ai/install.sh | sh
     ```



  3. launch Ollama serve

```
/content# ollama serve
```



4. download models

```
/content# ollama pull mistral #llama3.2 #bge-m3
```

5. check

```
!ollama list

####
NAME              ID              SIZE      MODIFIED
llama3.2:latest   a80c4f17acd5    2.0 GB    14 seconds ago
mistral:latest    f974a74358d6    4.1 GB    About a minute ago
```

2.3.4 langchain

LangChain is a powerful framework for building AI applications that combine the capabilities of large language models with external tools, memory, and custom workflows. It enables developers to create intelligent, context-aware, and dynamic applications with ease.

It has widely applied in:

1. **Conversational AI** Create chatbots or virtual assistants that maintain context, integrate with APIs, and provide intelligent responses.

2. **Knowledge Management** Combine LLMs with external knowledge bases or databases to answer complex questions or summarize documents.
3. **Automation** Automate workflows by chaining LLMs with tools for decision-making, data extraction, or content generation.
4. **Creative Applications** Use LangChain for generating stories, crafting marketing copy, or producing artistic content.

We will primarily use LangChain in this book. For instance, to work with downloaded Ollama LLMs, the `langchain_ollama` package is required.

```
# chain of thought prompting
from langchain_ollama.llms import OllamaLLM
from langchain_core.prompts import ChatPromptTemplate
from langchain.output_parsers import CommaSeparatedListOutputParser


template = """Question: {question}

Answer: Let's think step by step.
"""

prompt = ChatPromptTemplate.from_template(template)
model = OllamaLLM(temperature=0.0, model='mistral', format='json')
output_parser = CommaSeparatedListOutputParser()

chain = prompt | model | output_parser

response = chain.invoke({"question": "What is Mixture of Experts(MoE) in AI?"})
print(response)
```

```
['{"answer":"MoE', 'or Mixture of Experts', "is a neural network architecture␣
↪that allows for \
   efficient computation and model parallelism. It consists of multiple 'experts
↪'", 'each of \
   which is a smaller neural network that specializes in handling different␣
↪parts of the input \
   data. The final output is obtained by combining the outputs of these experts␣
↪based on their \
   expertise relevance to the input. This architecture is particularly useful␣
↪in tasks where \
   the data exhibits complex and diverse patterns."}',
   '\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t']
```

CHAPTER

THREE

WORD AND SENTENCE EMBEDDING

Word embedding is a method in natural language processing (NLP) to represent words as dense vectors of real numbers, capturing semantic relationships between them. Instead of treating words as discrete symbols (like one-hot encoding), word embeddings map words into a continuous vector space where similar words are located closer together.

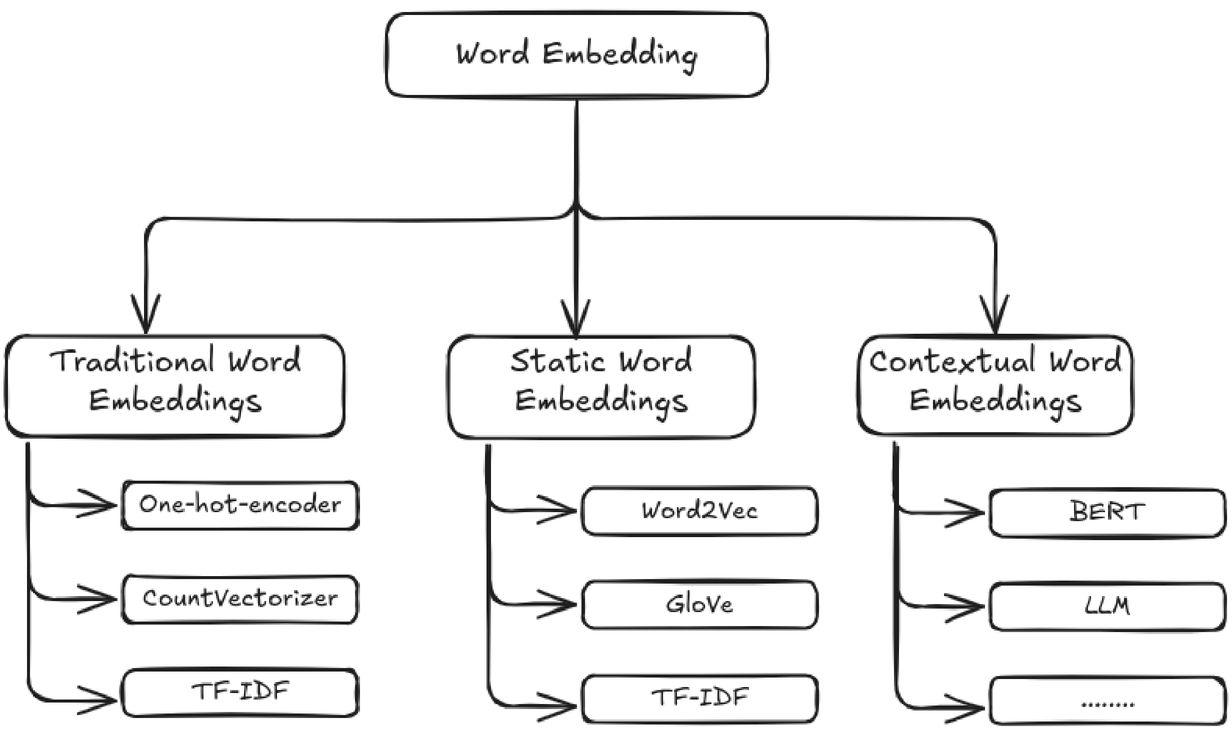


Fig. 1: Embedding Diagram

3.1 Traditional word embeddings

Bag of Words (BoW) is a simple and widely used text representation technique in natural language processing (NLP). It represents a text (e.g., a document or a sentence) as a collection of words, ignoring grammar, order, and context but keeping their frequency.

Key Features of Bag of Words:

1. **Vocabulary Creation**: - A list of all unique words in the dataset (the "vocabulary") is created. - Each word becomes a feature.
2. **Representation**: - Each document is represented as a vector or a frequency count of words from the vocabulary. - If a word from the vocabulary is present in the document, its count is included in the vector. - Words not present in the document are assigned a count of zero.
3. **Simplicity**: - The method is computationally efficient and straightforward. - However, it ignores the sequence and semantic meaning of the words.

Applications:

- Text Classification
- Sentiment Analysis
- Document Similarity

Limitations:

1. **Context Ignorance**: - BoW does not capture word order or semantics. - For example, "not good" and "good" might appear similar in BoW.
2. **Dimensionality**: - As the vocabulary size increases, the vector representation grows, leading to high-dimensional data.
3. **Sparse Representations**: - Many entries in the vectors might be zeros, leading to sparsity.

3.1.1 One Hot Encoder

```
import numpy as np
import pandas as pd
from collections import Counter
from sklearn.preprocessing import OneHotEncoder

# sample corpus
data = pd.DataFrame({'word':['python', 'pyspark', 'genai', 'pyspark', 'python',
↪'pyspark']})

# corpus frequency
print('Vocabulary frequency:')
print(dict(Counter(data['word'])))

# corpus order
print('\nVocabulary order:')
print(sorted(set(data['word'])))

# One-hot encode the data
onehot_encoder = OneHotEncoder(sparse_output=False)
onehot_encoded = onehot_encoder.fit_transform(data[['word']])

# the encoded order base on the order of the copus
```

(continues on next page)

(continued from previous page)

```
print('\nEncoded representation:')
print(onehot_encoded)
```

```
Vocabulary frequency:
{'python': 2, 'pyspark': 3, 'genai': 1}

Vocabulary order:
['genai', 'pyspark', 'python']

Encoded representation:
[[0. 0. 1.]
[0. 1. 0.]
[1. 0. 0.]
[0. 1. 0.]
[0. 0. 1.]
[0. 1. 0.]]
```

3.1.2 CountVectorizer

```
from sklearn.feature_extraction.text import CountVectorizer

# sample corpus
corpus = [
'Gen AI is awesome',
'Gen AI is fun',
'Gen AI is hot'
]

# Initialize the CountVectorizer
vectorizer = CountVectorizer()

# Fit and transform
X = vectorizer.fit_transform(corpus)


print('Vocabulary:')
print(vectorizer.get_feature_names_out())

print('\nEmbedded representation:')
print(X.toarray())
```

```
Vocabulary:
['ai' 'awesome' 'fun' 'gen' 'hot' 'is']

Embedded representation:
```

(continues on next page)

(continued from previous page)

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

To overcome these limitations, advanced techniques like **TF-IDF**, **word embeddings** (e.g., Word2Vec, GloVe), and contextual embeddings (e.g., BERT) are often used.

3.1.3 TF-IDF

TF-IDF (Term Frequency-Inverse Document Frequency) is a statistical measure used in text analysis to evaluate the importance of a word in a document relative to a collection (or corpus) of documents. It builds upon the **Bag of Words (BoW)** model by not only considering the frequency of a word in a document but also taking into account how common or rare the word is across the corpus. The pyspark implementation can be found at [PySpark].

- Components of TF-ID
- **t**: the term in corpus.
- **d**: the document.
- **D**: the corpus.
- **|D|**: the length of the corpus or total number of documents.
  - **Document Frequency (DF)**:
  - $DF(t, D)$: the number of documents that contains term t.
  - **Term Frequency (TF):**
    * Measures how frequently a term appears in a document. The higher the frequency, the more important the term is assumed to be to that document.
    * Formula:

$$TF(t, d) = \frac{\text{Number of occurrences of term } t \text{ in document } d}{\text{Total number of terms in document } d}$$

  - **Inverse Document Frequency (IDF):**
    * Measures how important a term is by reducing the weight of common terms (like "the" or "and") that appear in many documents.
    * Formula:

$$IDF(t, D) = \log\left(\frac{|D| + 1}{DF(t, D) + 1}\right) + 1$$

    * Adding 1 to the denominator avoids division by zero when a term is present in all documents.
    * Note that the IDF formula above differs from the standard textbook notation that defines the IDF

Note

The IDF formula above differs from the standard textbook notation that defines the IDF as

$$IDF(t) = \log[|D|/(DF(t, D) + 1)]).$$

- **TF-IDF Score:**
  - The final score is the product of TF and IDF.
  - Formula:

$$TF\text{-}IDF(t, d, D) = TF(t, d) \cdot IDF(t, D)$$

```
import pandas as pd
import numpy as np
from collections import Counter
from sklearn.feature_extraction.text import TfidfVectorizer

# sample corpus
corpus = [
'Gen AI is awesome',
'Gen AI is fun',
'Gen AI is hot'
]

# Initialize the TfidfVectorizer
vectorizer = TfidfVectorizer() # norm default norm='l2'

# Fit and transform
X = vectorizer.fit_transform(corpus)

print('Vocabulary:')
print(vectorizer.get_feature_names_out())

# [item for row in matrix for item in row]
corpus_flatted = [item for sub_list in [s.split(' ') for s in␣
↪corpus]
                     for item in sub_list]

print('\nVocabulary frequency:')
print(dict(Counter(corpus_flatted)))

print('\nEmbedded representation:')
print(X.toarray())
```

```
Vocabulary:
['ai' 'awesome' 'fun' 'gen' 'hot' 'is']

Vocabulary frequency:
{'Gen': 3, 'AI': 3, 'is': 3, 'awesome': 1, 'fun': 1, 'hot': 1}

Embedded representation:
[[0.41285857 0.69903033 0.         0.41285857 0.         0.41285857]
[0.41285857 0.         0.69903033 0.41285857 0.         0.41285857]
[0.41285857 0.         0.         0.41285857 0.69903033 0.41285857]]
```

The above results can be validated by the following steps (IDF in document 1):

```
# Step 1: Vocabulary  `['ai' 'awesome' 'fun' 'gen' 'hot' 'is']`

tf_idf = pd.DataFrame({'term':vectorizer.get_feature_names_out()})\
         .set_index('term')

# Step 2: |D|
tf_idf['|D|'] = [len(corpus)]*len(vectorizer.get_feature_names_
→out())

# Step 3: Compute TF for doc 1:  Gen AI is awesome
# - TF for "ai" in Document 1 = 1 (appears once doc 1)
# - TF for "awesome" in Document 1 = 1 (appears once in doc 1)
# - TF for "fun" in Document 1 = 0 (does not appear in doc 1)
# - TF for "gen" in Document 1 = 1 (appear oncein doc 1 )
# - TF for "hot" in Document 1 = 0 (does not appear doc 1 )
# - TF for "is" in Document 1 = 1 (appear once in doc 1 )

tf_idf['TF'] = [1, 1, 0, 1, 0, 1]

# Step 4:  Compute DF for doc 1
# - DF For "ai": Appears in all 3 documents.
# - DF For "awesome": Appears in 1 document.
# - DF For "fun": Appears in 1 document.
# - DF For "Gen": Appears in all 3 documents.
# - DF For "hot": Appears in 1 document.
# - DF For "is": Appears in all 3 documents.

tf_idf['DF'] = [3, 1, 1, 3, 1, 3]

# Step 5: Compute IDF
tf_idf['IDF'] = np.log((tf_idf['|D|']+1)/(tf_idf['DF']+1))+1

# Step 6: Compute TF-IDF
```

(continues on next page)

(continued from previous page)

```
tf_idf['TF-IDF'] = tf_idf['TF']*tf_idf['IDF']

# Step 7: l2 normlization
tf_idf['TF-IDF(l2)'] = tf_idf['TF-IDF']/np.linalg.norm(tf_idf['TF-
↪IDF'])

print(tf_idf)
```

```
          |D|  TF  DF       IDF    TF-IDF  TF-IDF(l2)
term
ai          3   1   3  1.000000  1.000000    0.412859
awesome     3   1   1  1.693147  1.693147    0.699030
fun         3   0   1  1.693147  0.000000    0.000000
gen         3   1   3  1.000000  1.000000    0.412859
hot         3   0   1  1.693147  0.000000    0.000000
is          3   1   3  1.000000  1.000000    0.412859
```

Fun Fact

TfidfVectorizer is equivalent to CountVectorizer followed by TfidfTransformer.

```
import pandas as pd
import numpy as np
from sklearn.feature_extraction.text import CountVectorizer
from sklearn.feature_extraction.text import TfidfTransformer
from sklearn.pipeline import Pipeline

# sample corpus
corpus = [
'Gen AI is awesome',
'Gen AI is fun',
'Gen AI is hot'
]

# pipeline
pipe = Pipeline([('count', CountVectorizer(lowercase=True)),
               ('tfid', TfidfTransformer())]).fit(corpus)
print(pipe)

# TF
print(pipe['count'].transform(corpus).toarray())

# IDF
print(pipe['tfid'].idf_)
```

```
Pipeline(steps=[('count', CountVectorizer()), ('tfid',␣
↪TfidfTransformer())])
[[1 1 0 1 0 1]
[1 0 1 1 0 1]
[1 0 0 1 1 1]]
[1.         1.69314718 1.69314718 1.         1.69314718 1.        ␣
↪ ]
```

- Applications of TF-IDF
  1. **Information Retrieval**: Ranking documents based on relevance to a query.
  2. **Text Classification**: Feature extraction for machine learning models.
  3. **Document Similarity**: Comparing documents by their weighted term vectors.
- Advantages
  - Highlights important terms while reducing the weight of common terms.
  - Simple to implement and effective for many tasks.
- Limitations
  - Does not capture semantic relationships or word order.
  - Less effective for very large corpora or when working with very short documents.
  - Sparse representation due to high-dimensional feature vectors.

For more advanced representations, embeddings like **Word2Vec** or **BERT** are often used.

3.2 Static word embeddings

Static word embeddings are word representations that assign a fixed vector to each word, regardless of its context in a sentence or paragraph. These embeddings are pre-trained on large corpora and remain unchanged during usage, making them "static." These embeddings are usually pre-trained on large text corpora using algorithms like Word2Vec, GloVe, or FastText.

3.2.1 Word2Vec

- The Context Window
- CBOW and Skip-Gram Model

```
import gensim
from gensim.models import Word2Vec
from nltk.tokenize import sent_tokenize, word_tokenize

# sample corpus
corpus = [
```

(continues on next page)

(continued from previous page)

```
'Gen AI is awesome',
'Gen AI is fun',
'Gen AI is hot'
]


def tokenize_gensim(corpus):

   tokens = []
   # iterate through each sentence in the corpus
   for s in corpus:

      # tokenize the sentence into words
      temp = gensim.utils.tokenize(s, lowercase=True, deacc=False, \
                                   errors='strict', to_lower=False, \
                                   lower=False)

      tokens.append(list(temp))

   return tokens


tokens = tokenize_gensim(corpus)

# Create Word2Vec model
# sg ({0, 1}, optional) - Training algorithm: 1 for skip-gram; otherwise CBOW.
# CBOW
model1 = gensim.models.Word2Vec(tokens, sg=0, min_count=1,
                              vector_size=10, window=5)

# Vocabulary
print(model1.wv.key_to_index)

print(model1.wv.get_normed_vectors())

# Print results
print("Cosine similarity between 'gen' " +
      "and 'ai' - Word2Vec(CBOW) : ",
      model1.wv.similarity('gen', 'ai'))



# Create Word2Vec model
# sg ({0, 1}, optional) - Training algorithm: 1 for skip-gram; otherwise CBOW.
# skip-gram
model2 = gensim.models.Word2Vec(tokens, sg=1, min_count=1,
                              vector_size=10, window=5)
```

(continues on next page)

(continued from previous page)

```
# Vocabulary
print(model2.wv.key_to_index)

print(model2.wv.get_normed_vectors())

# Print results
print("Cosine similarity between 'gen' " +
      "and 'ai' - Word2Vec(skip-gram) : ",
      model2.wv.similarity('gen', 'ai'))
```

```
{'is': 0, 'ai': 1, 'gen': 2, 'hot': 3, 'fun': 4, 'awesome': 5}
[[-0.02660277  0.0117296   0.25318226  0.44695902 -0.4615286  -0.35307196
   0.3204311   0.4451589  -0.24882038 -0.18670462]
[ 0.41619968 -0.08647515 -0.2558276   0.3695945  -0.274073   -0.10240843
   0.1622154   0.05593351 -0.46721786 -0.5328355 ]
[ 0.43418837  0.30108306  0.40128633  0.0453006   0.37712952 -0.20221795
-0.05619935  0.34255028 -0.44665098 -0.2337343 ]
[-0.41098067 -0.05088534  0.5218584  -0.40045303 -0.12768732 -0.10601949
   0.44194022 -0.32449666  0.00247097 -0.2600907 ]
[-0.44081825  0.22984274 -0.40207896 -0.20159177 -0.00161115 -0.0135952
-0.3516631   0.44133204  0.2286844   0.423816  ]
[-0.42753762  0.23561442 -0.21681462  0.04321203  0.44539306 -0.23385239
   0.23675178 -0.35568893 -0.18596812  0.49255413]]
Cosine similarity between 'gen' and 'ai' - Word2Vec(CBOW) :  0.32937223
{'is': 0, 'ai': 1, 'gen': 2, 'hot': 3, 'fun': 4, 'awesome': 5}
[[-0.02660277  0.0117296   0.25318226  0.44695902 -0.4615286  -0.35307196
   0.3204311   0.4451589  -0.24882038 -0.18670462]
[ 0.41619968 -0.08647515 -0.2558276   0.3695945  -0.274073   -0.10240843
   0.1622154   0.05593351 -0.46721786 -0.5328355 ]
[ 0.43418837  0.30108306  0.40128633  0.0453006   0.37712952 -0.20221795
-0.05619935  0.34255028 -0.44665098 -0.2337343 ]
[-0.41098067 -0.05088534  0.5218584  -0.40045303 -0.12768732 -0.10601949
   0.44194022 -0.32449666  0.00247097 -0.2600907 ]
[-0.44081825  0.22984274 -0.40207896 -0.20159177 -0.00161115 -0.0135952
-0.3516631   0.44133204  0.2286844   0.423816  ]
[-0.42753762  0.23561442 -0.21681462  0.04321203  0.44539306 -0.23385239
   0.23675178 -0.35568893 -0.18596812  0.49255413]]
Cosine similarity between 'gen' and 'ai' - Word2Vec(skip-gram) :  0.32937223
```

3.2.2 GloVE

```
import gensim.downloader as api
# Download pre-trained GloVe model
glove_vectors = api.load("glove-wiki-gigaword-50")
```

(continues on next page)

(continued from previous page)

```
# Get word vectors (embeddings)
word1 = "king"
word2 = "queen"
vector1 = glove_vectors[word1]
vector2 = glove_vectors[word2]
# Compute cosine similarity between the two word vectors
similarity = glove_vectors.similarity(word1, word2)
print(f"Word vectors for '{word1}': {vector1}")
print(f"Word vectors for '{word2}': {vector2}")
print(f"Cosine similarity between '{word1}' and '{word2}': {similarity}")
```

```
[==================================================] 100.0% 66.0/66.0MB␣
↪downloaded
Word vectors for 'king': [ 0.50451   0.68607  -0.59517  -0.022801  0.60046  -0.
↪13498  -0.08813
0.47377  -0.61798  -0.31012  -0.076666  1.493    -0.034189 -0.98173
0.68229   0.81722  -0.51874  -0.31503  -0.55809   0.66421   0.1961
-0.13495  -0.11476  -0.30344   0.41177  -2.223    -1.0756   -1.0783
-0.34354   0.33505   1.9927   -0.04234  -0.64319   0.71125   0.49159
0.16754   0.34344  -0.25663  -0.8523    0.1661    0.40102   1.1685
-1.0137   -0.21585  -0.15155   0.78321  -0.91241  -1.6106   -0.64426
-0.51042 ]
Word vectors for 'queen': [ 0.37854    1.8233    -1.2648    -0.1043     0.35829  ␣
↪   0.60029
-0.17538    0.83767   -0.056798  -0.75795    0.22681    0.98587
0.60587   -0.31419    0.28877    0.56013   -0.77456    0.071421
-0.5741     0.21342    0.57674    0.3868    -0.12574    0.28012
0.28135   -1.8053    -1.0421    -0.19255   -0.55375   -0.054526
1.5574     0.39296   -0.2475     0.34251    0.45365    0.16237
0.52464   -0.070272  -0.83744   -1.0326     0.45946    0.25302
-0.17837   -0.73398   -0.20025    0.2347    -0.56095   -2.2839
0.0092753 -0.60284   ]
Cosine similarity between 'king' and 'queen': 0.7839043140411377
```

3.2.3 Fast Text

Fast Text incorporates subword information (useful for handling rare or unseen words)

```
from gensim.models import FastText

import gensim
from gensim.models import Word2Vec

# sample corpus
corpus = [
```

(continues on next page)

(continued from previous page)

```
'Gen AI is awesome',
'Gen AI is fun',
'Gen AI is hot'
]


def tokenize_gensim(corpus):

   tokens = []
   # iterate through each sentence in the corpus
   for s in corpus:

      # tokenize the sentence into words
      temp = gensim.utils.tokenize(s, lowercase=True, deacc=False, \
                                    errors='strict', to_lower=False, \
                                    lower=False)

      tokens.append(list(temp))

   return tokens

tokens = tokenize_gensim(corpus)

# create FastText model
model = FastText(tokens, vector_size=10, window=5, min_count=1, workers=4)
# Train the model
model.train(tokens, total_examples=len(tokens), epochs=10)

# Vocabulary
print(model.wv.key_to_index)

print(model.wv.get_normed_vectors())

# Print results
print("Cosine similarity between 'gen' " +
      "and 'ai' - Word2Vec : ",
      model.wv.similarity('gen', 'ai'))
```

```
WARNING:gensim.models.word2vec:Effective 'alpha' higher than previous training␣
↪cycles
{'is': 0, 'ai': 1, 'gen': 2, 'hot': 3, 'fun': 4, 'awesome': 5}
[[-0.01875759  0.086543   -0.25080433  0.2824868  -0.23755953 -0.11316587
   0.473383    0.39204055 -0.30422893 -0.5566626 ]
[ 0.5088161  -0.3323528  -0.128698   -0.11877266 -0.38699347  0.20977001
   0.05947014 -0.05622245 -0.36257952 -0.5177341 ]
```

(continues on next page)

(continued from previous page)

```
[ 0.18038039  0.51484865  0.40694886  0.05965518 -0.05985437 -0.10832689
   0.37992737  0.5992712   0.01503773  0.1192203 ]
[-0.5694013   0.23560704  0.0265804  -0.41392225 -0.00285366 -0.3076269
   0.2076883  -0.425648    0.29903153  0.19965051]
[-0.23892775  0.10744874 -0.03730153 -0.23521401  0.32083488  0.21598674
-0.29570717 -0.03044808  0.75250715  0.26538488]
[-0.31881964 -0.06544963 -0.44274488  0.15485793  0.39120612 -0.05415314
   0.15772066 -0.05987714 -0.6986104   0.03967094]]
Cosine similarity between 'gen' and 'ai' - Word2Vec :  -0.21662527
```

3.3 Contextual word embeddings

Contextual word embeddings are word representations where the embedding of a word changes depending on its context in a sentence or document. These embeddings capture the meaning of a word as influenced by its surrounding words, addressing the limitations of static embeddings by incorporating contextual nuances.

3.3.1 BERT

```
from transformers import BertTokenizer, BertModel
tokenizer = BertTokenizer.from_pretrained('bert-base-uncased')
model = BertModel.from_pretrained("bert-base-uncased")


text = "Gen AI is awesome"
encoded_input = tokenizer(text, return_tensors='pt')
embeddings = model(**encoded_input).last_hidden_state

print(encoded_input)
print({x : tokenizer.encode(x, add_special_tokens=False) for x in ['[CLS]']+
↪text.split()+ ['[SEP]', '[EOS]']})

print(embeddings.shape)
print(embeddings)
```

```
t{'input_ids': tensor([[  101,  8991,  9932,  2003, 12476,   102]]), 'token_type_
↪ids': tensor([[0, 0, 0, 0, 0, 0]]), 'attention_mask': tensor([[1, 1, 1, 1, 1,
↪1]])}
{'[CLS]': [101], 'Gen': [8991], 'AI': [9932], 'is': [2003], 'awesome': [12476],
↪'[SEP]': [102], '[EOS]': [1031, 1041, 2891, 1033]}


torch.Size([1, 6, 768])
tensor([[[-0.1129, -0.1477, -0.0056,  ..., -0.1335,  0.2605,  0.2113],
         [-0.6841, -1.1196,  0.3349,  ..., -0.5958,  0.1657,  0.6988],
         [-0.5385, -0.2649,  0.2639,  ..., -0.1544,  0.2532, -0.1363],
```

(continues on next page)

(continued from previous page)

```
        [-0.1794, -0.6086,  0.1292,  ..., -0.1620,  0.1721,  0.4356],
        [-0.0187, -0.7320, -0.3420,  ...,  0.4028,  0.1425, -0.2014],
        [ 0.5493, -0.1029, -0.1571,  ...,  0.3503, -0.7601, -0.1398]]],
    grad_fn=<NativeLayerNormBackward0>)
```

3.3.2 gte-large-en-v1.5

The `gte-large-en-v1.5` is a state-of-the-art text embedding model developed by Alibaba's Institute for Intelligent Computing. It's designed for natural language processing tasks and excels in generating dense vector representations (embeddings) of text for applications such as text retrieval, classification, clustering, and reranking.

It can handle up to 8192 tokens, making it suitable for long-context tasks. More details can be found at: https://huggingface.co/Alibaba-NLP/gte-large-en-v1.5 .

```
# Requires transformers>=4.36.0

import torch.nn.functional as F
from transformers import AutoModel, AutoTokenizer

input_texts = [
'Gen AI is awesome',
'Gen AI is fun',
'Gen AI is hot'
]

model_path = 'Alibaba-NLP/gte-large-en-v1.5'
tokenizer = AutoTokenizer.from_pretrained(model_path)
model = AutoModel.from_pretrained(model_path, trust_remote_code=True)

# Tokenize the input texts
batch_dict = tokenizer(input_texts, max_length=8192, padding=True, \
                     truncation=True, return_tensors='pt')

print(batch_dict)


outputs = model(**batch_dict)
embeddings = outputs.last_hidden_state[:, 0]

# (Optionally) normalize embeddings
embeddings = F.normalize(embeddings, p=2, dim=1)
scores = (embeddings[:1] @ embeddings[1:].T) * 100
print(embeddings)
print(scores.tolist())
```

```
{'input_ids': tensor([[  101,  8991,  9932,  2003, 12476,   102],
      [  101,  8991,  9932,  2003,  4569,   102],
      [  101,  8991,  9932,  2003,  2980,   102]]), 'token_type_ids': tensor([[0,
↪ 0, 0, 0, 0, 0],
      [0, 0, 0, 0, 0, 0],
      [0, 0, 0, 0, 0, 0]]), 'attention_mask': tensor([[1, 1, 1, 1, 1, 1],
      [1, 1, 1, 1, 1, 1],
      [1, 1, 1, 1, 1, 1]])}

tensor([[ 0.0079,  0.0008, -0.0001,  ...,  0.0418, -0.0138, -0.0236],
      [ 0.0079,  0.0218, -0.0171,  ...,  0.0412, -0.0230, -0.0237],
      [ 0.0073, -0.0106, -0.0194,  ...,  0.0711, -0.0204, -0.0036]],
      grad_fn=<DivBackward0>)
[[92.85284423828125, 92.81655883789062]]
```

3.3.3 bge-base-en-v1.5

The bge-base-en-v1.5 model is a general-purpose text embedding model developed by the Beijing Academy of Artificial Intelligence (BAAI). It transforms input text into 768-dimensional vector embeddings, making it useful for tasks like semantic search, text similarity, and clustering. This model is fine-tuned using contrastive learning, which helps improve its ability to distinguish between similar and dissimilar sentences effectively. More details can be found at: https://huggingface.co/BAAI/bge-base-en-v1.5 .

```
from transformers import AutoTokenizer, AutoModel
import torch

# Sentences we want sentence embeddings for
sentences = [
'Gen AI is awesome',
'Gen AI is fun',
'Gen AI is hot'
]
# Load model from HuggingFace Hub
tokenizer = AutoTokenizer.from_pretrained('BAAI/bge-large-zh-v1.5')
model = AutoModel.from_pretrained('BAAI/bge-large-zh-v1.5')
model.eval()

# Tokenize sentences
encoded_input = tokenizer(sentences, padding=True, truncation=True, return_
↪tensors='pt')
print(encoded_input)

# Compute token embeddings
with torch.no_grad():
   model_output = model(**encoded_input)
   # Perform pooling. In this case, cls pooling.
```

(continues on next page)

(continued from previous page)

```
    sentence_embeddings = model_output[0][:, 0]
# normalize embeddings
sentence_embeddings = torch.nn.functional.normalize(sentence_embeddings, p=2,␣
↪dim=1)
print("Sentence embeddings:", sentence_embeddings)
```

```
{'input_ids': tensor([[  101, 10234,  8171,  8578,  8310,   143, 11722,  9974, ␣
↪8505,   102],
     [  101, 10234,  8171,  8578,  8310,  9575,   102,     0,     0,     0],
     [  101, 10234,  8171,  8578,  8310,  9286,   102,     0,     0,     0]]),
↪'token_type_ids': tensor([[0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
     [0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
     [0, 0, 0, 0, 0, 0, 0, 0, 0, 0]]), 'attention_mask': tensor([[1, 1, 1, 1, 1,␣
↪1, 1, 1, 1, 1],
     [1, 1, 1, 1, 1, 1, 1, 0, 0, 0],
     [1, 1, 1, 1, 1, 1, 1, 0, 0, 0]])}

Sentence embeddings: tensor([[ 0.0700,  0.0119,  0.0049,  ...,  0.0428, -0.0475,␣
↪ 0.0242],
      [ 0.0800, -0.0065, -0.0519,  ...,  0.0057, -0.0770,  0.0119],
      [ 0.0740, -0.0185, -0.0369,  ...,  0.0083, -0.0026,  0.0016]])
```

CHAPTER

FOUR

PROMPT ENGINEERING

4.1 Prompt

A prompt is the input or query given to an LLM to elicit a specific response. It acts as the user's way of "programming" the model without code, simply by phrasing questions or tasks appropriately.

4.2 Prompt Engineering

4.2.1 What's Prompt Engineering

Prompt engineering is the practice of designing and refining input prompts to guide LLMs to produce desired outputs effectively and consistently. It involves crafting queries, commands, or instructions that align with the model's capabilities and the task's requirements.

4.2.2 Key Elements of a Prompt

- Clarity: A clear and unambiguous prompt ensures the model understands the task.
- Specificity: Including details like tone, format, length, or audience helps tailor the response.
- Context:Providing background information ensures the model generates relevant outputs.

4.3 Advanced Prompt Engineering

4.3.1 Role Assignment

- Assign a specific role or persona to the AI to shape its style and expertise.
- **Example:**

  "You are a professional data scientist. Explain how to build a machine learning model to a beginner."

```
# You are a professional data scientist. Explain how to build a machine
# learning model to a beginner.
template = """Role: you are a {role}
task: {task}
Answer:
```

(continues on next page)

(continued from previous page)

```
"""

prompt = ChatPromptTemplate.from_template(template)
model = OllamaLLM(temperature=0.0, model=MODEL, format='json')
output_parser = CommaSeparatedListOutputParser()

chain = prompt | model | output_parser

response = chain.invoke({'role': 'data scientist', \
                         "task": "Explain how to build a machine \
                         learning model to a beginner"})
print(response)
```

```
 {
"role": "assistant",
"content": "To build a machine learning model, let's follow these steps as a␣
↪beginner: \n\n1. **Define the Problem**: Understand what problem you are␣
↪trying to solve. This could be anything from predicting house prices,␣
↪recognizing images, or even recommending products. \n\n2. **Collect and␣
↪Prepare Data**: Gather relevant data for your problem. This might involve web␣
↪scraping, APIs, or using existing datasets. Once you have the data, clean it␣
↪by handling missing values, outliers, and errors. \n\n3. **Explore and␣
↪Visualize Data**: Understand the structure of your data, its distribution, and␣
↪relationships between variables. This can help in identifying patterns and␣
↪making informed decisions about the next steps. \n\n4. **Feature␣
↪Engineering**: Create new features that might be useful for the model to make␣
↪accurate predictions. This could involve creating interactions between␣
↪existing features or using techniques like one-hot encoding. \n\n5. **Split␣
↪Data**: Split your data into training, validation, and testing sets. The␣
↪training set is used to train the model, the validation set is used to tune␣
↪hyperparameters, and the testing set is used to evaluate the final performance␣
↪of the model. \n\n6. **Choose a Model**: Select a machine learning algorithm␣
↪that suits your problem. Some common algorithms include linear regression for␣
↪regression problems, logistic regression for binary classification problems,␣
↪decision trees, random forests, support vector machines (SVM), and neural␣
↪networks for more complex tasks. \n\n7. **Train the Model**: Use your training␣
↪data to train the chosen model. This involves feeding the data into the model␣
↪and adjusting its parameters based on the error it makes. \n\n8. **Tune␣
↪Hyperparameters**: Adjust the hyperparameters of the model to improve its␣
↪performance. This could involve changing learning rates, number of layers in a␣
↪neural network, or the complexity of a decision tree. \n\n9. **Evaluate the␣
↪Model**: Use your testing data to evaluate the performance of the model.␣
↪Common metrics include accuracy for classification problems, mean squared␣
↪error for regression problems, and precision, recall, and F1 score for␣
↪imbalanced datasets. \n\n10. **Deploy the Model**: Once you are satisfied with␣
```

(continues on next page)

(continued from previous page)

```
↪the performance of your model, deploy it to a production environment where it␣
↪can make predictions on new data."
}
```

4.3.2 Contextual Setup

- Provide sufficient background or context for the AI to understand the task.
- **Example:**

  "I am planing to write a book about GenAI best practice, help me draft the contents for the book."

```
# Contextual Setup

# I am planing to write a book about GenAI best practice, help me draft the
# contents for the book.
template = """Role: you are a {role}
task: {task}
Answer:
"""

prompt = ChatPromptTemplate.from_template(template)
model = OllamaLLM(temperature=0.0, model=MODEL, format='json')
output_parser = CommaSeparatedListOutputParser()

chain = prompt | model | output_parser

response = chain.invoke({'role': 'book writer', \
                         "task": "I am planing to write a book about \
                         GenAI best practice, help me draft the \
                         contents for the book."})
print(response)
```

```
{"1. Introduction": "Introduction to General Artificial Intelligence (GenAI) and␣
↪its significance in today's world.",
"2. Chapter 1 - Understanding AI": "Exploring the basics of Artificial␣
↪Intelligence, its history, and evolution.",
"3. Chapter 2 - Types of AI": "Detailed discussion on various types of AI such␣
↪as Narrow AI, General AI, and Superintelligent AI.",
"4. Chapter 3 - GenAI Architecture": "Exploring the architecture of General AI␣
↪systems, including neural networks, deep learning, and reinforcement learning.
↪",
"5. Chapter 4 - Ethics in AI Development": "Discussing the ethical␣
↪considerations involved in developing GenAI, such as privacy, bias, and␣
↪accountability.",
```

(continues on next page)

(continued from previous page)

```
"6. Chapter 5 - Data Collection and Management": "Understanding the importance␣
↪of data in AI development, best practices for data collection, and responsible␣
↪data management.",
"7. Chapter 6 - Model Training and Optimization": "Exploring techniques for␣
↪training AI models effectively, including hyperparameter tuning,␣
↪regularization, and optimization strategies.",
"8. Chapter 7 - Testing and Validation": "Discussing the importance of testing␣
↪and validation in ensuring the reliability and accuracy of GenAI systems.",
"9. Chapter 8 - Deployment and Maintenance": "Exploring best practices for␣
↪deploying AI models into production environments, as well as ongoing␣
↪maintenance and updates.",
"10. Case Studies": "Real-world examples of successful GenAI implementations␣
↪across various industries, highlighting key takeaways and lessons learned.",
"11. Future Trends in GenAI": "Exploring emerging trends in the field of General␣
↪AI, such as quantum computing, explainable AI, and human-AI collaboration.",
"12. Conclusion": "Summarizing the key points discussed in the book and looking␣
↪forward to the future of General AI."}
```

4.3.3 Explicit Instructions

- Clearly specify the format, tone, style, or structure you want in the response.
- **Example:**

  "Explain the concept of word embeddings in 100 words, using simple language suitable for a high school student."

```
# Explicit Instructions
from langchain_ollama.llms import OllamaLLM
from langchain_core.prompts import ChatPromptTemplate
from langchain.output_parsers import CommaSeparatedListOutputParser


# Explain the concept of word embeddings in 100 words, using simple
# language suitable for a high school student

template = """you are a {role}
task: {task}
instruction: {instruction}
Answer: Let's think step by step.
"""

prompt = ChatPromptTemplate.from_template(template)
model = OllamaLLM(temperature=0.0, model=MODEL, format='json')
output_parser = CommaSeparatedListOutputParser()

chain = prompt | model
```

(continues on next page)

(continued from previous page)

```
response = chain.invoke({'role': 'AI engineer', \
                        'task': "Explain the concept of word embeddings in \
                                100 words",\
                        'instruction': "using simple \
                                language suitable for a high school student"})

print(response)
```

```
{
"assistant": {
    "message": "Word Embeddings are like giving words a special address in a big␣
↪library. Each word gets its own unique location, and words that are used in␣
↪similar ways get placed close together. This helps the computer understand the␣
↪meaning of words better when it's reading text. For example, 'king' might be␣
↪near 'queen', because they are both types of royalty. And 'apple' might be␣
↪near 'fruit', because they are related concepts."
}
}
```

4.3.4 Chain of Thought (CoT) Prompting

- Encourage step-by-step reasoning for complex problems.
- **Example:**

  "Solve this math problem step by step: A train travels 60 miles in 1.5 hours. What is its average speed?"

```
# CoT
from langchain_ollama.llms import OllamaLLM
from langchain_core.prompts import ChatPromptTemplate
from langchain.output_parsers import CommaSeparatedListOutputParser


# Solve this math problem step by step: A train travels 60 miles in 1.5 hours.
# What is its average speed?

template = """you are a {role}
task: {task}
question: {question}
Answer: Let's think step by step.
"""

prompt = ChatPromptTemplate.from_template(template)
model = OllamaLLM(temperature=0.0, model=MODEL, format='json')
```

(continues on next page)

(continued from previous page)

```
output_parser = CommaSeparatedListOutputParser()

chain = prompt | model

response = chain.invoke({'role': 'math student', \
                        'task': "Solve this math problem step by step: \
                                A train travels 60 miles in 1.5 hours.",\
                        'question': "What is its average speed per minute?"})

print(response)
```

```
 {
"Solution": {
   "Step 1": "First, let's find the average speed of the train per hour.",
   "Step 2": "The train travels 60 miles in 1.5 hours. So, its speed per hour is␣
↪60 miles / 1.5 hours = 40 miles/hour.",
   "Step 3": "Now, let's find the average speed of the train per minute. Since␣
↪there are 60 minutes in an hour, the speed per minute would be the speed per␣
↪hour multiplied by the number of minutes in an hour divided by 60.",
   "Step 4": "So, the average speed of the train per minute is (40 miles/hour *␣
↪(1 hour / 60)) = (40/60) miles/minute = 2/3 miles/minute."
             }
 }
```

4.3.5 Few-Shot Prompting

- Provide examples to guide the AI on how to respond.
- **Example:**

  ***"Here are examples of loan application decision:**
  'example': {'input': {'fico':800, 'income':100000,'loan_amount': 10000} 'decision': "accept" Now Help me to make a decision to accpet or reject the loan application and give the reason. 'input': "{'fico':820, 'income':100000, 'loan_amount': 1,000}""*

```
# Few-Shot Prompting
from langchain_ollama.llms import OllamaLLM
from langchain_core.prompts import ChatPromptTemplate
from langchain.output_parsers import CommaSeparatedListOutputParser


# Here are examples of loan application decision:
# 'example': {'input': {'fico':800, 'income':100000,'loan_amount': 10000}
# 'decision': "accept"
# Now Help me to make a decision to accpet or reject the loan application and
# give the reason.
```

(continues on next page)

(continued from previous page)

```
# 'input': "{'fico':820, 'income':100000, 'loan_amount': 1,000}"

template = """you are a {role}
task: {task}
examples: {example}
input: {input}
decision:
"""

prompt = ChatPromptTemplate.from_template(template)
model = OllamaLLM(temperature=0.0, model=MODEL, format='json')
output_parser = CommaSeparatedListOutputParser()

chain = prompt | model

response = chain.invoke({'role': 'banker', \
                        'task': "Help me to make a decision to accpet or \
                                reject the loan application ",\
                        'example': {'input': {'fico':800, 'income':100000,\
                                              'loan_amount': 10000},\
                                    'decision': "accept"}, \
                        'input': {'fico':820, 'income':100000, \
                                    'loan_amount': 1000}
                        })

print(response)
```

```
{"decision": "accept"}
```

4.3.6 Iterative Prompting

- Build on the AI's response by asking follow-up questions or refining the output.
- **Example:**
  - *Initial Prompt:* " Help me to make a decision to accpet or reject the loan application."
  - *Follow-Up:* "give me the reason"

```
# Few-Shot Prompting
from langchain_ollama.llms import OllamaLLM
from langchain_core.prompts import ChatPromptTemplate
from langchain.output_parsers import CommaSeparatedListOutputParser


# Here are examples of loan application decision:
# 'example': {'input': {'fico':800, 'income':100000,'loan_amount': 10000}
```

(continues on next page)

(continued from previous page)

```
# 'decision': "accept"
# Now Help me to make a decision to accpet or reject the loan application and
# give the reason.
# 'input': "{'fico':820, 'income':100000, 'loan_amount': 1,000}"

template = """you are a {role}
task: {task}
examples: {example}
input: {input}
decision:
reason:
"""

prompt = ChatPromptTemplate.from_template(template)
model = OllamaLLM(temperature=0.0, model=MODEL, format='json')
output_parser = CommaSeparatedListOutputParser()

chain = prompt | model

response = chain.invoke({'role': 'banker', \
                        'task': "Help me to make a decision to accpet or \
                                reject the loan application and \
                                give the reason.",\
                        'example': {'input': {'fico':800, 'income':100000,\
                                              'loan_amount': 10000},\
                                    'decision': "accept"}, \
                        'input': {'fico':820, 'income':100000, \
                                    'loan_amount': 1000}
                        })

print(response)
```

```
{"decision": "accept", "reason": "The applicant has a high credit score (FICO␣
↪820), a stable income of $100,000, and is requesting a relatively small loan␣
↪amount ($1000). These factors indicate a low risk for the bank."}
```

4.3.7 Instructional Chaining

- Break down a task into a sequence of smaller prompts.
- **Example:**
  - step 1: check the fico score
  - step 2: check the income,
  - step 3: check the loan amount,
  - step 4: make a decision,

– step 5: give the reason.

```
# Instructional Chaining
from langchain_ollama.llms import OllamaLLM
from langchain_core.prompts import ChatPromptTemplate
from langchain.output_parsers import CommaSeparatedListOutputParser


# Now Help me to make a decision to accpet or reject the loan application and
# give the reason.
# "input": {'fico':320, 'income':10000, 'loan_amount': 100000}

template = """you are a {role}
task: {task}
instruction: {instruction}
input: {input}
decision:
reason:
"""

prompt = ChatPromptTemplate.from_template(template)
model = OllamaLLM(temperature=0.0, model=MODEL, format='json')
output_parser = CommaSeparatedListOutputParser()

chain = prompt | model

response = chain.invoke({'role': 'banker', \
                        'task': "Help me to make a decision to accpet or \
                                reject the loan application and \
                                give the reason.",\
                        'instruction': {'step 1': "check the fico score",\
                                        'step 2': "check the income",\
                                        'step 3': "check the loan amount",\
                                        'step 4': "make a decision",\
                                        'step 5': "give the reason"
                                        },
                        'input': {'fico':320, 'income':10000, \
                                    'loan_amount': 100000}
                        })

print(response)
```

```
{
   "decision": "reject",
   "reason": "Based on the provided information, the applicant's FICO score is␣
↪320 which falls below our minimum acceptable credit score. Additionally, the␣
↪proposed loan amount of $100,000 exceeds the income level of $10,000 per year,␣
```

(continues on next page)

(continued from previous page)

```
↪making it difficult for the borrower to repay the loan."
}
```

4.3.8 Use Constraints

- Impose constraints to keep responses concise and on-topic.
- **Example:**

  "List 5 key trends in AI in bullet points, each under 15 words."

4.3.9 Creative Prompting

- Encourage unique or unconventional ideas by framing the task creatively.
- **Example:**

  "Pretend you are a time traveler from the year 2124. How would you describe AI advancements to someone today?"

4.3.10 Feedback Incorporation

- If the response isn't perfect, guide the AI to refine or retry.
- **Example:**

  "This is too general. Could you provide more specific examples for the education industry?"

4.3.11 Scenario-Based Prompts

- Frame the query within a scenario for a contextual response.
- **Example:**

  "Imagine you're a teacher explaining ChatGPT to students. How would you introduce its uses and limitations?"

4.3.12 Multimodal Prompting

- Use prompts designed for mixed text/image inputs (or outputs if using models like DALL·E).
- **Example:**

  "Generate an image prompt for a futuristic cityscape, vibrant, with flying cars and greenery."

CHAPTER

FIVE

RETRIEVAL-AUGMENTED GENERATION

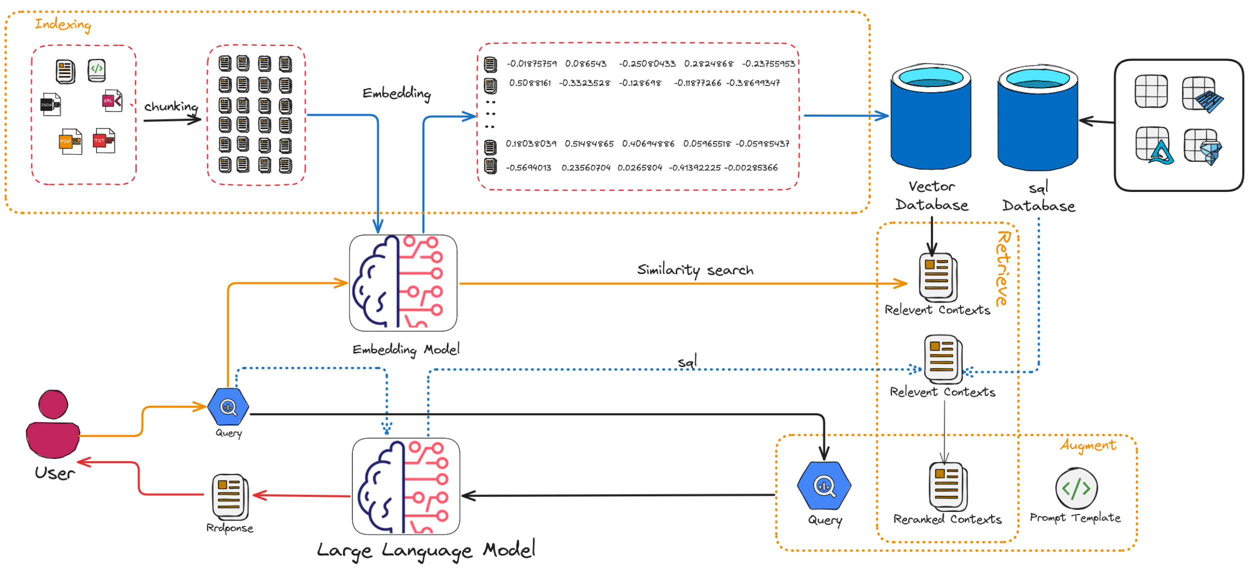


Fig. 1: Retrieval-Augmented Generation Diagram

5.1 Overview

Retrieval-Augmented Generation (RAG) is a framework that enhances large language models (LLMs) by combining their generative capabilities with external knowledge retrieval. The goal of RAG is to improve accuracy, relevance, and factuality by providing the LLM with specific, up-to-date, or domain-specific context from a knowledge base or database during the generation process.

As you can see in *Retrieval-Augmented Generation Diagram*, the RAG has there main components

- Indexer: The indexer processes raw text or other forms of unstructured data and creates an efficient structure (called an index) that allows for fast and accurate retrieval by the retriever when a query is made.
- Retriever: Responsible for finding relevant information from an external knowledge source, such as a document database, a vector database, or the web.
- Generator: An LLM (like GPT-4, T5, or similar) that uses the retrieved context to generate a response. The model is "augmented" with the retrieved information, which reduces hallucination and enhances

factual accuracy.

5.2 Indexing

5.2.1 Chunking

Chunking in Retrieval-Augmented Generation (RAG) involves splitting documents or knowledge bases into smaller, manageable pieces (chunks) that can be efficiently retrieved and used by a language model (LLM).

5.2.2 Embedding

5.2.3 Vector Database

5.3 Retrieval

Common retrieval methods:

- Sparse Search: Traditional keyword-based retrieval (e.g., TF-IDF, BM25).
- Dense Retrieval: Vector-based search using embeddings (e.g., FAISS, Pinecone).

The retriever selects "chunks" of text (e.g., paragraphs or sections) relevant to the user's query.

5.4 Generation

CHAPTER

SIX

FINE TUNING

CHAPTER

SEVEN

PRE-TRAINING

CHAPTER

EIGHT

LLM EVALUATION METRICS

8.1 Statistical Scorers

8.2 Model-Based Scorers

CHAPTER

NINE

MAIN REFERENCE

BIBLIOGRAPHY

[GenAI] Wenqiang Feng, Di Zhen. GenAI: Best Practices, 2024.

[PySpark] Wenqiang Feng. Learning Apache Spark with Python, 2017.