



# GenAI: Best Practices

*Release 1.0*

**Wenqiang Feng, Di Zhen and Wenyun Wang**

**December 22, 2024**



# CONTENTS

<b>1 Preface</b>	<b>3</b>
1.1 About . . . . .	3
1.1.1 About this book . . . . .	3
1.1.2 About the authors . . . . .	3
1.2 Feedback and suggestions . . . . .	5
<b>2 Preliminary</b>	<b>7</b>
2.1 Math Preliminary . . . . .	7
2.1.1 Vector . . . . .	7
2.1.2 Norm . . . . .	9
2.1.3 Distances . . . . .	9
2.2 NLP Preliminary . . . . .	11
2.2.1 Vocabulary . . . . .	11
2.2.2 Tagging . . . . .	12
2.2.3 Lemmatization . . . . .	13
2.2.4 Tokenization . . . . .	15
2.2.5 BERT Tokenization . . . . .	16
2.2.6 Modern BERT . . . . .	18
2.3 Platform and Packages . . . . .	21
2.3.1 Google Colab . . . . .	21
2.3.2 HuggingFace . . . . .	22
2.3.3 Ollama . . . . .	23
2.3.4 langchain . . . . .	24
<b>3 Word and Sentence Embedding</b>	<b>27</b>
3.1 Traditional word embeddings . . . . .	27
3.1.1 One Hot Encoder . . . . .	28
3.1.2 CountVectorizer . . . . .	29
3.1.3 TF-IDF . . . . .	30
3.2 Static word embeddings . . . . .	34
3.2.1 Word2Vec . . . . .	35
3.2.2 GloVe . . . . .	37
3.2.3 Fast Text . . . . .	38
3.3 Contextual word embeddings . . . . .	39
3.3.1 BERT . . . . .	39

3.3.2	gte-large-en-v1.5	40
3.3.3	bge-base-en-v1.5	41
<b>4</b>	<b>Prompt Engineering</b>	<b>45</b>
4.1	Prompt	45
4.2	Prompt Engineering	45
4.2.1	What's Prompt Engineering	45
4.2.2	Key Elements of a Prompt	45
4.3	Advanced Prompt Engineering	46
4.3.1	Role Assignment	46
4.3.2	Contextual Setup	47
4.3.3	Explicit Instructions	48
4.3.4	Chain of Thought (CoT) Prompting	49
4.3.5	Few-Shot Prompting	50
4.3.6	Iterative Prompting	51
4.3.7	Instructional Chaining	53
4.3.8	Use Constraints	54
4.3.9	Creative Prompting	54
4.3.10	Feedback Incorporation	54
4.3.11	Scenario-Based Prompts	54
4.3.12	Multimodal Prompting	55
<b>5</b>	<b>Retrieval-Augmented Generation</b>	<b>57</b>
5.1	Overview	57
5.2	Indexing	58
5.2.1	Naive Chunking	58
5.2.2	Late Chunking	62
5.2.3	Types of Indexing	66
5.2.4	Vector Database	66
5.3	Retrieval	69
5.3.1	Common retrieval methods	69
5.3.2	Reciprocal Rank Fusion	72
5.4	Generation	74
5.5	Advanced Topic	75
5.5.1	Self-RAG	75
5.5.2	Corrective RAG	93
5.5.3	Adaptive RAG	110
5.5.4	Agentic RAG	131
<b>6</b>	<b>Fine Tuning</b>	<b>145</b>
6.1	Cutting-Edge Strategies for LLM Fine-Tuning	146
6.1.1	LoRA (Low-Rank Adaptation)	146
6.1.2	QLoRA (Quantized Low-Rank Adaptation)	146
6.1.3	PEFT (Parameter-Efficient Fine-Tuning)	147
6.1.4	SFT (Supervised Fine-Tuning)	147
6.1.5	Summary Table	148
6.2	Key Early Fine-Tuning Methods	148
6.2.1	Full Fine-Tuning	148

6.2.2	Feature-Based Approach . . . . .	149
6.2.3	Layer-Specific Fine-Tuning . . . . .	149
6.2.4	Task-Adaptive Pre-training . . . . .	149
6.3	Embedding Model Fine-Tuning . . . . .	149
6.3.1	Prepare Dataset . . . . .	150
6.3.2	Import and Evaluate Pretrained Baseline Model . . . . .	151
6.3.3	Loss Function with Matryoshka Representation . . . . .	153
6.3.4	Fine-tune Embedding Model . . . . .	155
6.3.5	Evaluate Fine-tuned Model . . . . .	156
6.3.6	Results Comparison . . . . .	156
6.4	LLM Fine-Tuning . . . . .	157
6.4.1	Load Dataset and Pretrained Model . . . . .	157
6.4.2	Fine-tuning Configuration . . . . .	158
6.4.3	Fine-tune model . . . . .	159
<b>7</b>	<b>Pre-training</b>	<b>161</b>
<b>8</b>	<b>LLM Evaluation Metrics</b>	<b>163</b>
8.1	Statistical Scorers (Traditional Metrics) . . . . .	164
8.2	Model-Based Scorers (Learned Metrics) . . . . .	164
8.3	Human-Centric Evaluations (Augmenting Metrics) . . . . .	165
8.4	GEval with DeepEval . . . . .	165
8.4.1	G-Eval Algorithm . . . . .	165
8.4.2	G-Eval with DeepEval . . . . .	165
<b>9</b>	<b>Main Reference</b>	<b>177</b>
<b>Bibliography</b>		<b>179</b>





Welcome to our **GenAI: Best Practices!!!** For each chapter, we provide detailed Colab notebooks [!\[\]\(d84e7ea36f695d92cb39ec32c307ac93\_img.jpg\) Open in Colab](#) that you can open and run directly in Google Colab. The PDF version can be downloaded from [HERE](#).



---

# CHAPTER ONE

---

## PREFACE

### 1.1 About

#### 1.1.1 About this book

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

In this book, we aim to demonstrate best practices for Generative AI through detailed demo code and practical examples. For each chapter, we provide detailed Colab notebooks  [Open in Colab](#) that you can open and run directly in Google Colab.

#### 1.1.2 About the authors

- Authors

- Wenqiang Feng

- \* Sr. Mgr Data Enginner and PhD in Mathematics
    - \* University of Tennessee at Knoxville
    - \* Webpage: <https://github.com/runawayhorse001>
    - \* Email: [von198@gmail.com](mailto:von198@gmail.com)

- Di Zhen

- \* Sr. Analyst - Data Science and M.S. in Computational Biology
    - \* Harvard University
    - \* Email: [dizhen318@gmail.com](mailto:dizhen318@gmail.com)

- Wenyun Wang

- \* Ph.D. candidate in Applied Physics
    - \* Harvard University
    - \* Email: [wenyunw08@gmail.com](mailto:wenyunw08@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).

- **Di Zhen** is a Senior Data Science Analyst at American Express, where she drives impactful business decisions by leveraging advanced analytics and cutting-edge technologies. Her expertise spans causal inference, predictive modeling, natural language processing, and generative AI, with a focus on empowering sales enablement through data-driven insights.

Di earned her Master of Science in Computational Biology and Quantitative Genetics from Harvard University in 2023, where she developed a robust foundation in computation and statistical analysis. Passionate about solving complex, real-world problems, she combines technical precision with innovative thinking to deliver actionable solutions that enhance business performance and customer experiences. Dedicated to continuous learning, Di is committed to staying at the forefront of data science advancements to unlock new possibilities.

- **Wenyun Wang** is currently a Ph.D. candidate in Applied Physics at Harvard University. She also holds a Master’s degree in Computational Science and Engineering from Harvard University. Her research interests lie at the intersection of data science, machine learning, and generative AI, with a focus on solving practical problems in scientific research and real-world applications. She is passionate about leveraging advanced computational techniques to extract insights from complex data and drive innovation across diverse domains.

- **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.

**Warning**

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](mailto:von198@gmail.com), Di Zhen: [dizhen318@gmail.com](mailto:dizhen318@gmail.com) and Wenyun Wang: [wenyunw08@gmail.com](mailto:wenyunw08@gmail.com)) for improvements.



---

## CHAPTER TWO

---

## PRELIMINARY

### Chinese proverb

A journey of a thousand miles begins with a single step. – Lao Tzu

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

### Colab Notebook for This Chapter

- Math Preliminary:  [Open in Colab](#)
- Ollama in Colab:  [Open in Colab](#)
- BERT Tokenization:  [Open in Colab](#)

## 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]
```

(continues on next page)

(continued from previous page)

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

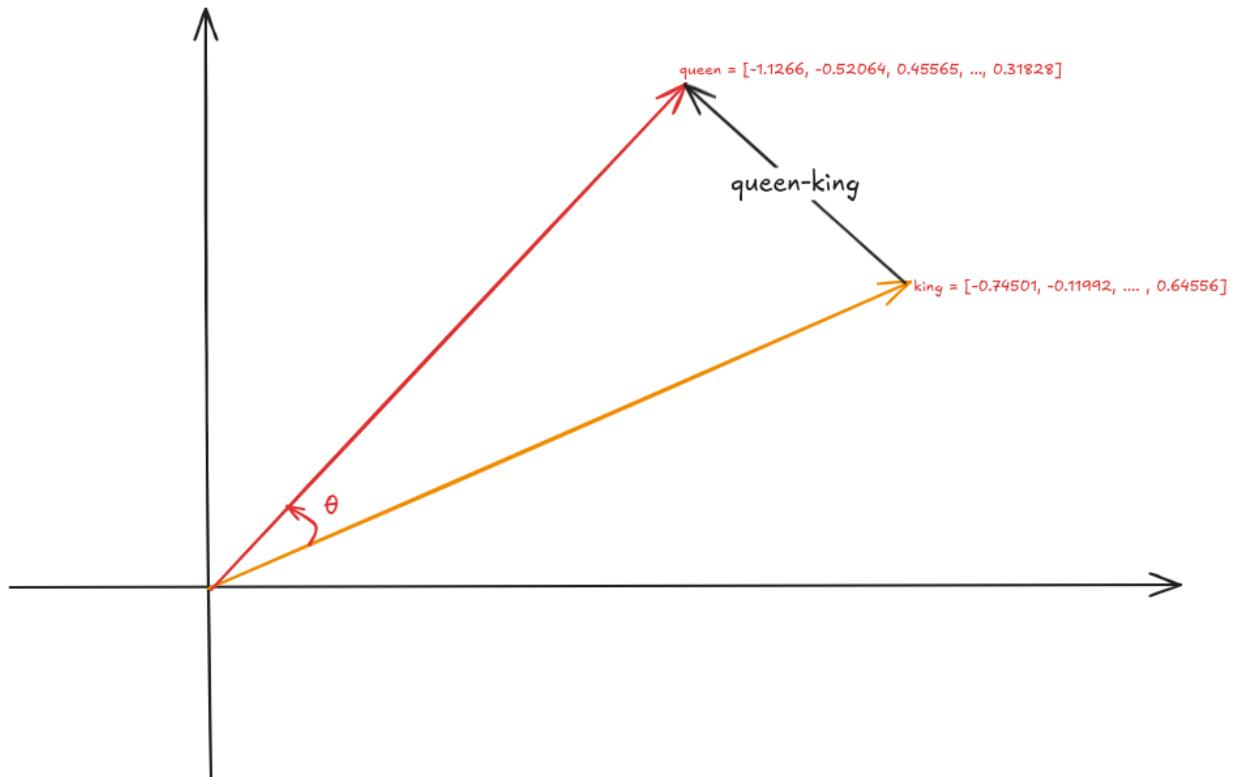


Fig. 1: Vector

## 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.

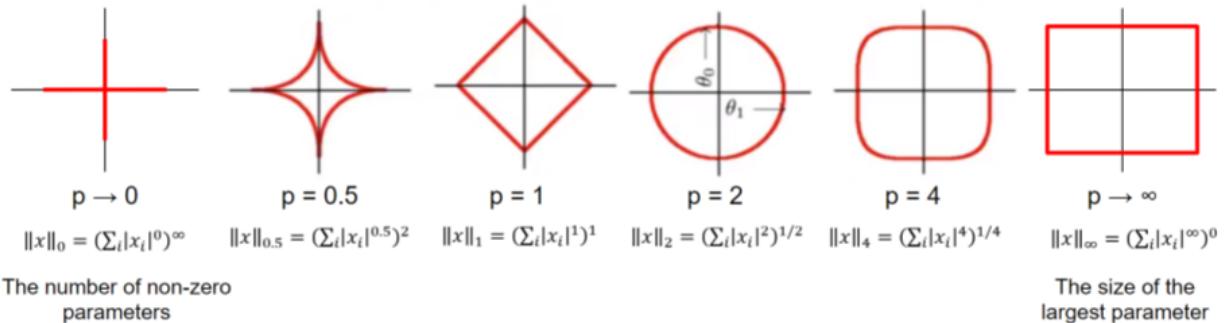


Fig. 2: Geometrical Interpretation of Norm (source\_1)

- Formula:

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

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

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

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

### 13.188952
```

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

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

### 4.3206835
```

- $\ell^\infty$  norm (Maximum norm): The largest absolute value of a vector component.

## 2.1.3 Distances

- Manhattan Distance ( $\ell^1$  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  $\vec{u} = (u_1, u_2, \dots, u_n)$  and  $\vec{v} = (v_1, v_2, \dots, v_n)$ , the Manhattan Distance distance  $d(\vec{u}, \vec{v})$  is

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

- Euclidean Distance ( $\ell^2$  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  $\vec{u} = (u_1, u_2, \dots, u_n)$  and  $\vec{v} = (v_1, v_2, \dots, v_n)$ , the Euclidean Distance distance  $d(\vec{u}, \vec{v})$  is

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

- Minkowski Distance ( $\ell^p$  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  $\vec{u} = (u_1, u_2, \dots, u_n)$  and  $\vec{v} = (v_1, v_2, \dots, v_n)$  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)
```

(continues on next page)

(continued from previous page)

```
# 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]
```

## 2.2.6 Modern BERT

ModernBERT is an encoder-only model. It is based on the architecture of the original BERT (Bidirectional Encoder Representations from Transformers), which is designed to process and understand text by encoding it into dense numerical representations (embedding vectors).

### Note

- **encoder-only** model is a type of transformer architecture designed primarily to understand and process input data by encoding it into a dense numerical representation, often called an embedding vector.
- **decoder-only** model is a transformer architecture designed for generating or predicting sequences, such as text. such as GPT, Llama, and Claude.

The new architecture delivers significant improvements over its predecessors in both speed and accuracy (Fig. *Modern BERT Pareto Curve (Source: Modern BERT)*).

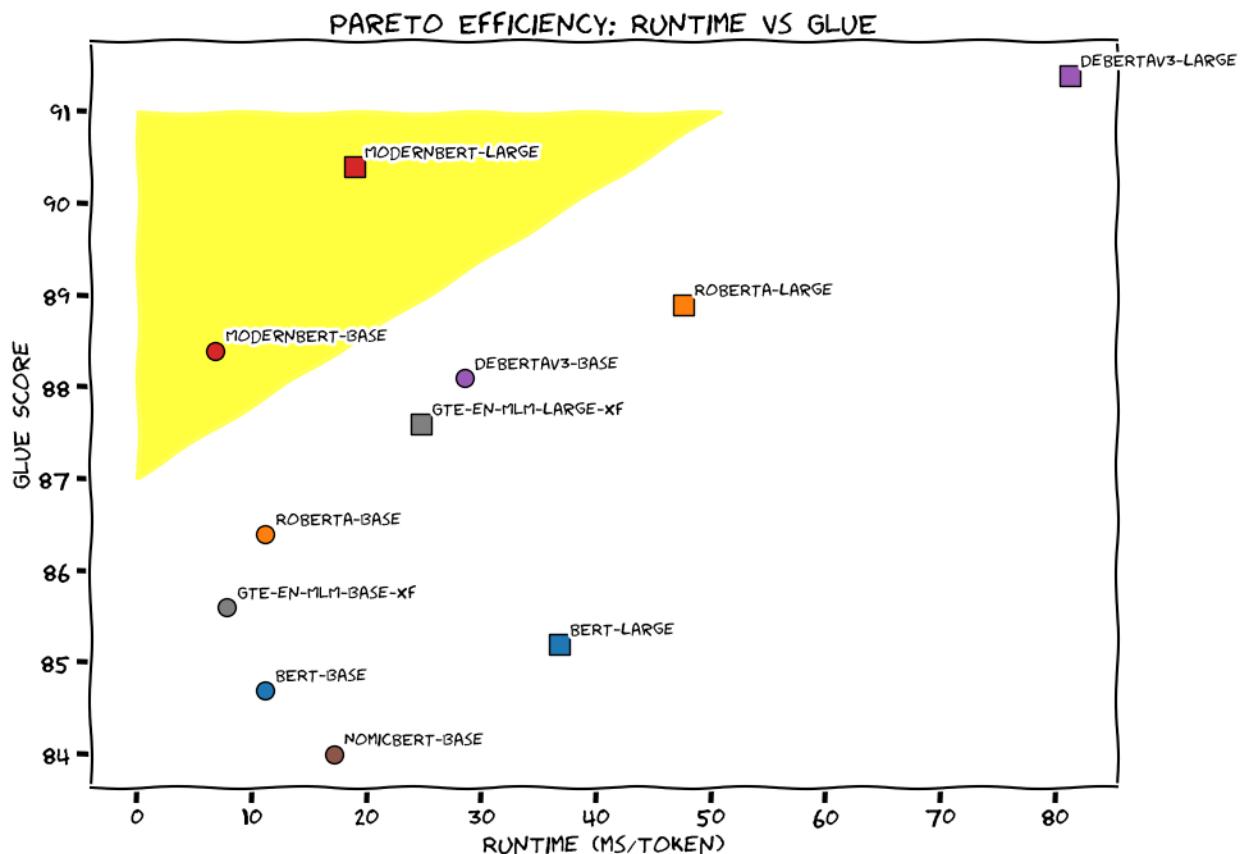


Fig. 3: Modern BERT Pareto Curve (Source: Modern BERT)

- Global and Local Attention

One of ModernBERT's most impactful features is Alternating Attention, rather than full global attention.

- Unpadding and Sequence Packing

Another core mechanism contributing to ModernBERT's efficiency is its use for Unpadding and Sequence packing.

```
from transformers import AutoTokenizer, AutoModelForMaskedLM
```

(continues on next page)

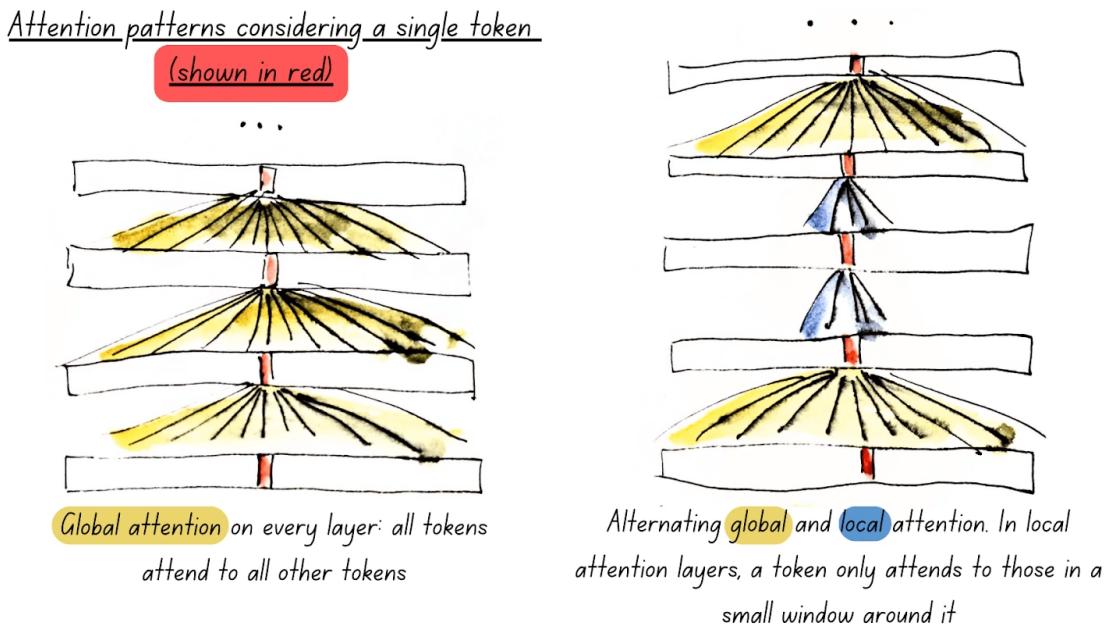


Fig. 4: ModernBERT Alternating Attention (Source: Modern BERT)

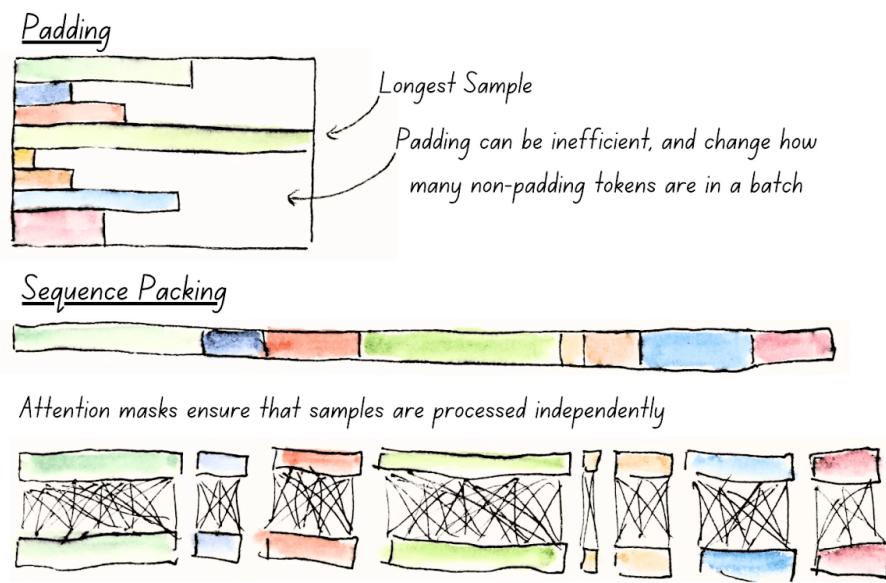


Fig. 5: ModernBERT Unpadding (Source: Modern BERT)

(continued from previous page)

```

model_id = "answerdotai/ModernBERT-base"
tokenizer = AutoTokenizer.from_pretrained(model_id)
model = AutoModelForMaskedLM.from_pretrained(model_id)

text = "The capital of France is [MASK]."
inputs = tokenizer(text, return_tensors="pt")
outputs = model(**inputs)

# To get predictions for the mask:
masked_index = inputs["input_ids"][[0]].tolist().index(tokenizer.mask_token_id)
predicted_token_id = outputs.logits[0, masked_index].argmax(axis=-1)
predicted_token = tokenizer.decode(predicted_token_id)
print("Predicted token:", predicted_token)
# Predicted token: Paris

```

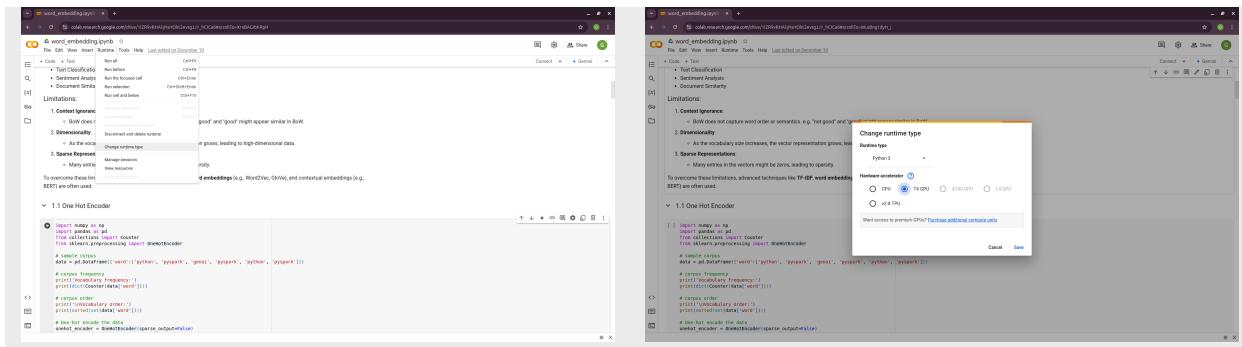
## 2.3 Platform and Packages

### 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



### Tips

You can use the Gemini API for code troubleshooting in a Colab notebook for free.

The screenshot shows a Colab notebook titled "Run-ollama-in-colab.ipynb". The code cell [4] installs the Ollama library and runs a sample prompt. The output shows the installation process and the generated response from the Ollama model. To the right, a "Validation Error" panel from the Gemini API is open, displaying troubleshooting steps for validation errors, such as specifying the "format='json'" parameter when initializing the OllamaLLM class. It also includes an "Explanation of Changes" section with a note about the "format='json'" parameter.

### 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

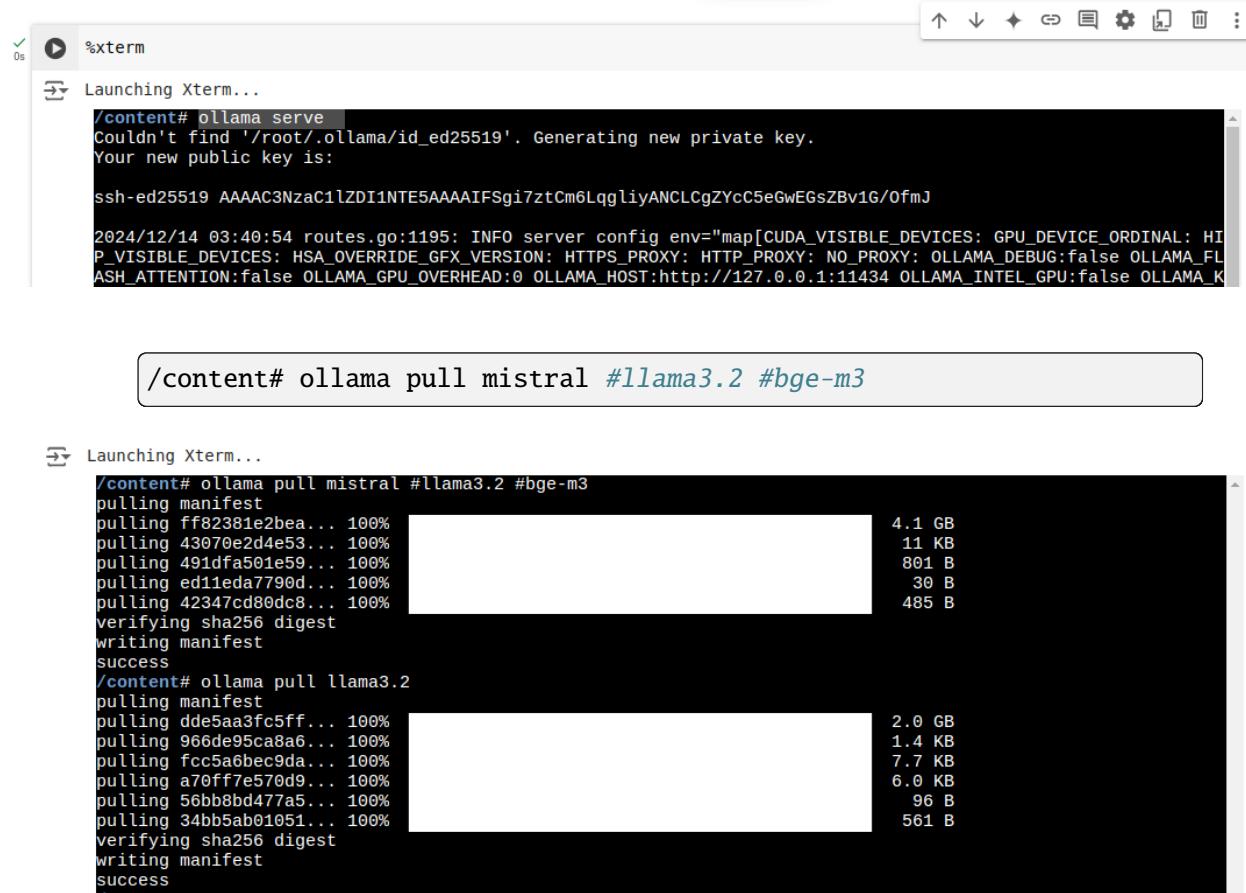
```
/content# curl https://ollama.ai/install.sh | sh
```

```
%xterm # curl https://ollama.ai/install.sh | sh
Launching Xterm...
/content# curl https://ollama.ai/install.sh | sh
% Total    % Received % Xferd  Average Speed   Time     Time      Time  Current
          Dload  Upload Total   Spent    Left Speed
100 13269    0 13269    0      0  47685      0 --:--:-- --:--:-- 47730
>>> Installing ollama to /usr/local
>>> Downloading Linux amd64 bundle
#####
>>> Creating ollama user...
>>> Adding ollama user to video group...
>>> Adding current user to ollama group...
>>> Creating ollama systemd service...
WARNING: systemd is not running
WARNING: Unable to detect NVIDIA/AMD GPU. Install lspci or lshw to automatically detect and install GPU dependencies.
>>> The ollama API is now available at 127.0.0.1:11434.
>>> Install complete. Run "ollama" from the command line.
/content#
```

3. launch Ollama serve

```
/content# ollama serve
```

4. download models



```

%xterm
Launching Xterm...
/content# ollama serve
Couldn't find '/root/.ollama/id_ed25519'. Generating new private key.
Your new public key is:
ssh-ed25519 AAAAC3NzaC1lZDI1NTE5AAAAIFsgj7ztCm6LqglijANCLCgZYCC5eGwEGsZBv1G/OfmJ
2024/12/14 03:40:54 routes.go:1195: INFO server config env="map[CUDA_VISIBLE_DEVICES: GPU_DEVICE_ORDINAL: HI_P_VISIBLE_DEVICES: HSA_OVERRIDE_GFX_VERSION: HTTPS_PROXY: HTTP_PROXY: NO_PROXY: OLLAMA_DEBUG:false OLLAMA_FLUSH_ATTENTION:false OLLAMA_GPU_OVERHEAD:0 OLLAMA_HOST:http://127.0.0.1:11434 OLLAMA_INTEL_GPU:false OLLAMA_K
/content# ollama pull mistral #llama3.2 #bge-m3
Launching Xterm...
/content# ollama pull mistral #llama3.2 #bge-m3
pulling manifest
pulling ff82381e2bea... 100% 4.1 GB
pulling 43070e2d4e53... 100% 11 KB
pulling 491dfa501e59... 100% 801 B
pulling ed11eda7790d... 100% 30 B
pulling 42347cd80dc8... 100% 485 B
verifying sha256 digest
writing manifest
success
/content# ollama pull llama3.2
pulling manifest
pulling dde5aa3fc5ff... 100% 2.0 GB
pulling 966de95ca8a6... 100% 1.4 KB
pulling fcc5aebec9da... 100% 7.7 KB
pulling a70ff7e570d9... 100% 6.0 KB
pulling 56bb8bd477a5... 100% 96 B
pulling 34bb5ab01051... 100% 561 B
verifying sha256 digest
writing manifest
success

```

### 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:

- Conversational AI** Create chatbots or virtual assistants that maintain context, integrate with APIs, and provide intelligent responses.
- 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)"})
print(response)
```



## WORD AND SENTENCE EMBEDDING

### Chinese proverb

Though the forms may vary, the essence remains unchanged. – old Chinese proverb

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.

### Colab Notebook for This Chapter

- Word Embedding:  [Open in Colab](#)

## 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

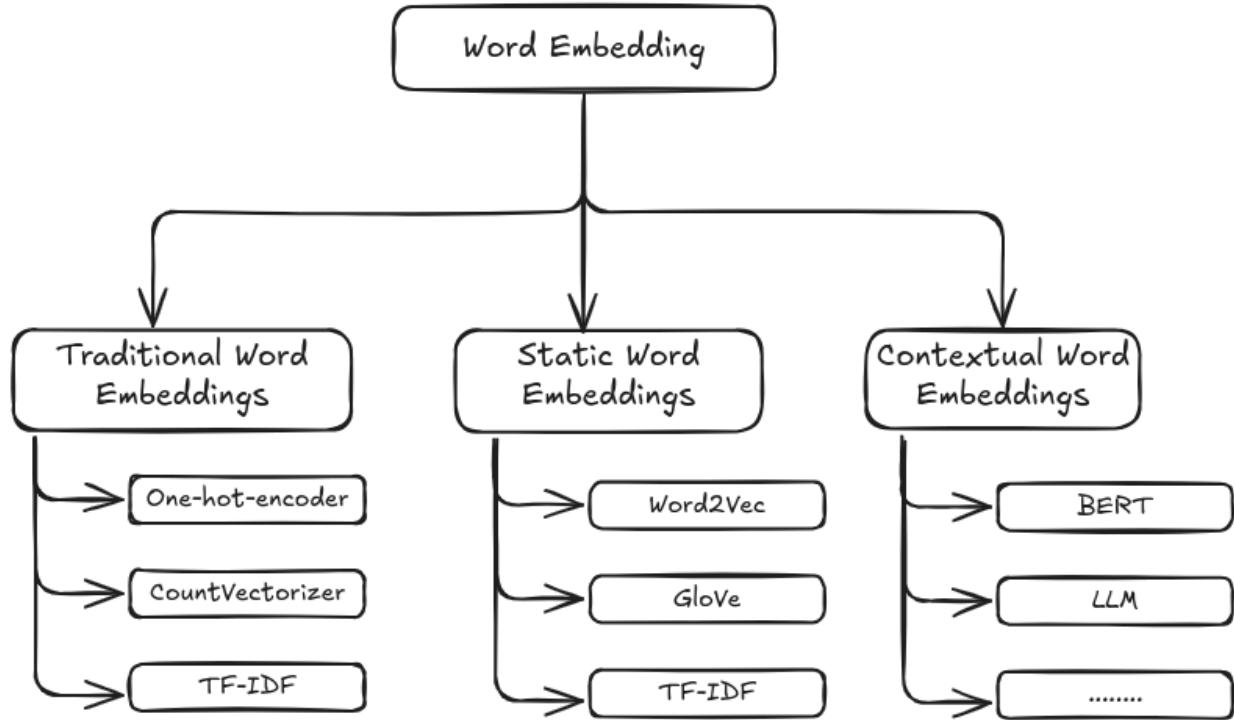


Fig. 1: Embedding Diagram

- 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:')
  
```

(continues on next page)

(continued from previous page)

```

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
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)

```

(continues on next page)

(continued from previous page)

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

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

**Vocabulary:**

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

**Embedded representation:**

```
[[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-IDF
- **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:

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

```
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())
```

(continues on next page)

(continued from previous page)

```
# [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.
```

(continues on next page)

(continued from previous page)

```

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

# Step 7: l2 normalization
tf_idf['TF-IDF(12)'] = tf_idf['TF-IDF']/np.linalg.norm(tf_idf['TF-IDF'])

print(tf_idf)

```

	D	TF	DF	IDF	TF-IDF	TF-IDF(12)
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 = [
'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

```

(continues on next page)

(continued from previous page)

```

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)

# 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]]

```

(continues on next page)

(continued from previous page)

```
-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")
# 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 = [
'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 ]
 [ 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],

```

(continues on next page)

(continued from previous page)

```

→ '[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],
         [-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]

```

(continues on next page)

(continued from previous page)

```
# (Optional) 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
```

(continues on next page)

(continued from previous page)

```

encoded_input = tokenizer(sentences, padding=True, truncation=True, return_
↪tokens='pt')
print(encoded_input)

# Compute token embeddings
with torch.no_grad():
    model_output = model(**encoded_input)
    # Perform pooling. In this case, cls pooling.
    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], 'attention_mask': tensor([[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, 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]])}

```

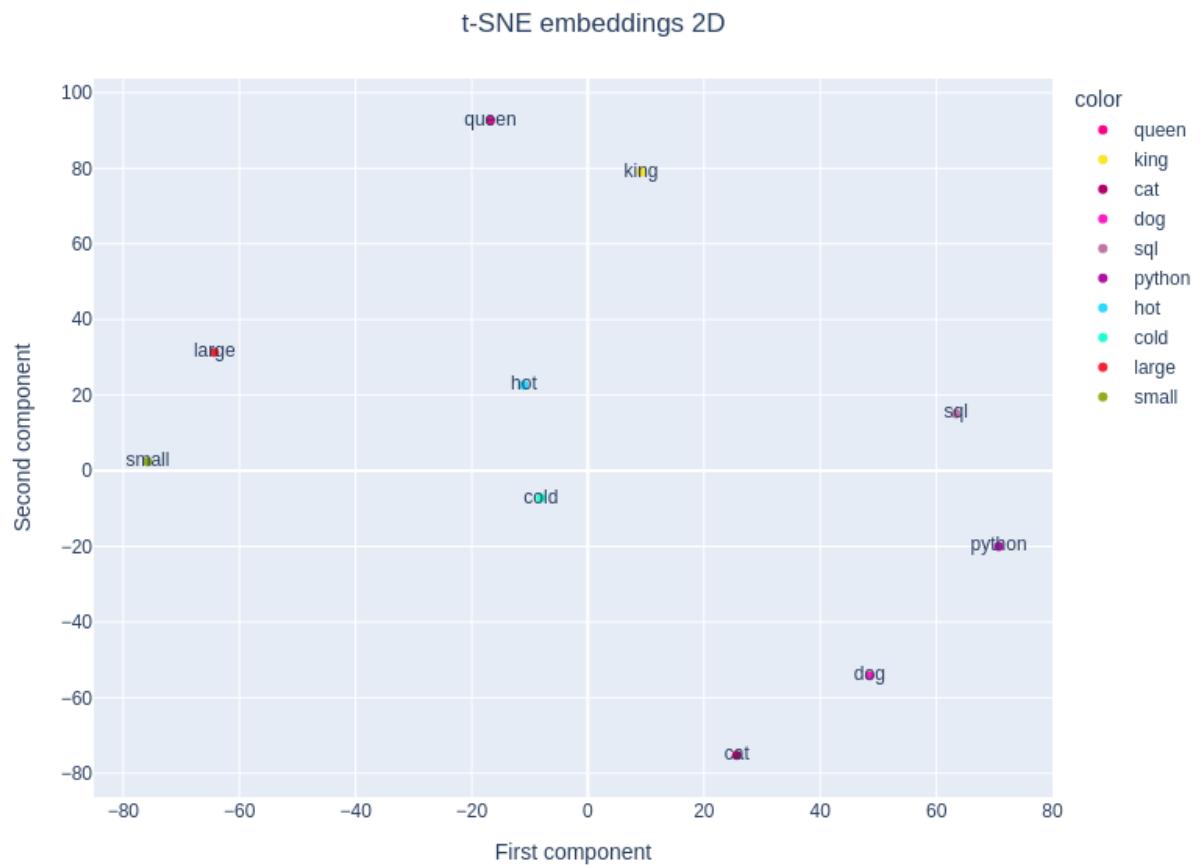


Fig. 2: t-SNE embeddings 2D



---

## CHAPTER FOUR

---

# PROMPT ENGINEERING

### Proverb

Our master knows how to guide people skillfully and methodically. He broadens my mind with culture and restrains me with ritual. – Zi Han

### Colab Notebook for This Chapter

- Prompt Engineering with Local LLM:  [Open in Colab](#)
- Prompt Engineering with OpenAI API:  [Open in Colab](#)

## 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:
"""

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,
```

(continues on next page)

(continued from previous page)

```

→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
→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.",
  "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
```

(continues on next page)

(continued from previous page)

```

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

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

```

(continues on next page)

(continued from previous page)

```
# 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')
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\u2192 60 miles / 1.5 hours = 40 miles/hour.",
    "Step 3": "Now, let's find the average speed of the train per minute. Since\u2192 there are 60 minutes in an hour, the speed per minute would be the speed per\u2192 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 * \u2192 (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.
# '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 \\\nreject the loan application ",\
                        'example': {'input': {'fico':800, 'income':100000,\n                           'loan_amount': 10000},\n                           '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}
# '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}})
```

(continues on next page)

(continued from previous page)

```
        '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, ↵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.”*



## RETRIEVAL-AUGMENTED GENERATION

### Colab Notebook for This Chapter

- Naive Chunking:  [Open in Colab](#)
- Late Chunking:  [Open in Colab](#)
- Reciprocal Rank Fusion:  [Open in Colab](#)
- RAG in colab:  [Open in Colab](#)
- Langchain Google Web Search  [Open in Colab](#)
- Langchain SQL Query  [Open in Colab](#)
- Self-RAG:  [Open in Colab](#)
- Corrective RAG:  [Open in Colab](#)
- Adaptive RAG:  [Open in Colab](#)
- Agentic RAG:  [Open in Colab](#)

### Note

The naive chunking strategy was used in the diagram above. More advanced strategies, such as Late Chunking [[lateChunking](#)] (or Chunked Pooling), are discussed later in this chapter.

## 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 three main components

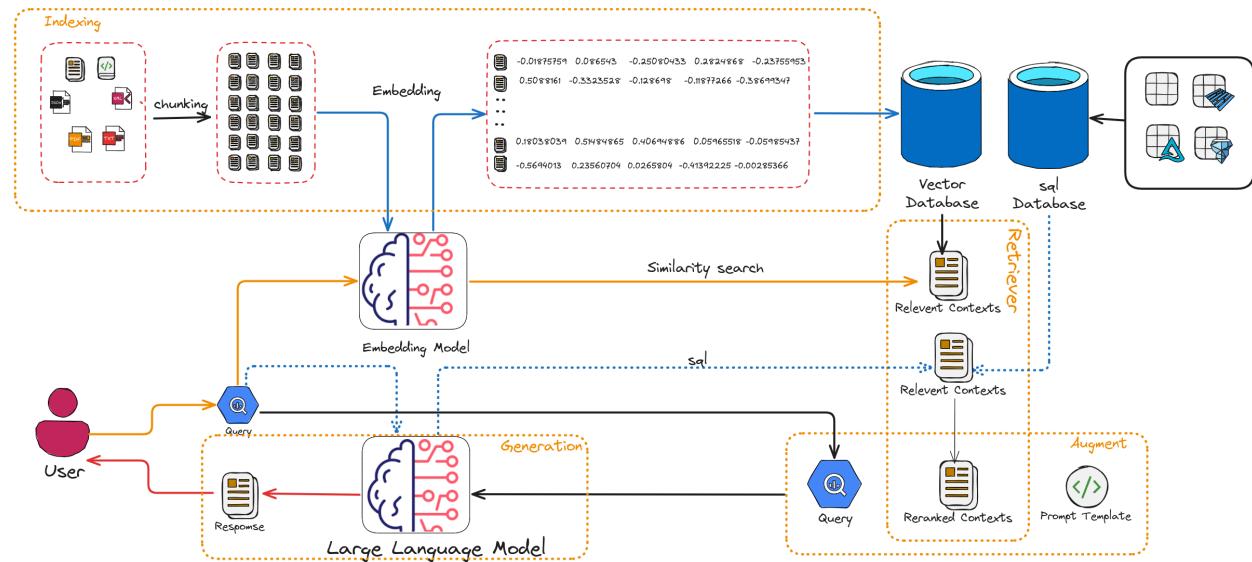
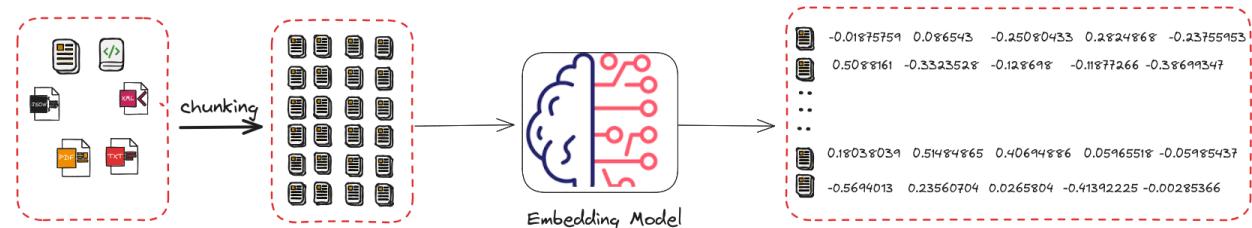


Fig. 1: Retrieval-Augmented Generation Diagram

- 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

The indexing 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.



### 5.2.1 Naive 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).

Below are the common chunking strategies used in RAG workflows:

## 1. Fixed-Length Chunking

- Chunks are created with a predefined, fixed length (e.g., 200 words or 512 tokens).
- Simple and easy to implement but might split content mid-sentence or lose semantic coherence.

**Example:**

```
def fixed_length_chunking(text, chunk_size=200):
    words = text.split()
    return [
        " ".join(words[i:i + chunk_size])
        for i in range(0, len(words), chunk_size)
    ]

# Example Usage
document = "This is a sample document with multiple sentences to demonstrate fixed-length chunking."
chunks = fixed_length_chunking(document, chunk_size=10)
for idx, chunk in enumerate(chunks):
    print(f"Chunk {idx + 1}: {chunk}")
```

**Output:**

- Chunk 1: This is a sample document with multiple sentences to demonstrate
- Chunk 2: fixed-length chunking.

## 2. Sliding Window Chunking

- Creates overlapping chunks to preserve context across splits.
- Ensures important information in overlapping regions is retained.

**Example:**

```
def sliding_window_chunking(text, chunk_size=100, overlap_size=20):
    words = text.split()
    chunks = []
    for i in range(0, len(words), chunk_size - overlap_size):
        chunk = " ".join(words[i:i + chunk_size])
        chunks.append(chunk)
    return chunks

# Example Usage
document = "This is a sample document with multiple sentences to demonstrate sliding window chunking."
chunks = sliding_window_chunking(document, chunk_size=10, overlap_size=3)
for idx, chunk in enumerate(chunks):
    print(f"Chunk {idx + 1}: {chunk}")
```

**Output:**

- Chunk 1: This is a sample document with multiple sentences to demonstrate
- Chunk 2: with multiple sentences to demonstrate sliding window chunking.
- Chunk 3: sliding window chunking.

### 3. Semantic Chunking

- Splits text based on natural language boundaries such as paragraphs, sentences, or specific delimiters (e.g., headings).
- Retains semantic coherence, ideal for better retrieval and generation accuracy.

**Example:**

```
import nltk
nltk.download('punkt_tab')

def semantic_chunking(text, sentence_len=50):
    sentences = nltk.sent_tokenize(text)

    chunks = []
    chunk = ""
    for sentence in sentences:
        if len(chunk.split()) + len(sentence.split()) <= sentence_len:
            chunk += " " + sentence
        else:
            chunks.append(chunk.strip())
            chunk = sentence
    if chunk:
        chunks.append(chunk.strip())
    return chunks

# Example Usage
document = ("This is a sample document. It is split based on semantic boundaries."
             "Each chunk will have coherent meaning for better retrieval.")
chunks = semantic_chunking(document, 10)
for idx, chunk in enumerate(chunks):
    print(f"Chunk {idx + 1}: {chunk}")
```

**Output:**

- Chunk 1: This is a sample document.
- Chunk 2: It is split based on semantic boundaries.
- Chunk 3: Each chunk will have coherent meaning for better retrieval.

### 4. Dynamic Chunking

- Adapts chunk sizes based on content properties such as token count, content density, or specific criteria.

- Useful when handling diverse document types with varying information density.

**Example:**

```
from transformers import AutoTokenizer

def dynamic_chunking(text, max_tokens=200, tokenizer_name="bert-base-uncased"):
    tokenizer = AutoTokenizer.from_pretrained(tokenizer_name)
    tokens = tokenizer.encode(text, add_special_tokens=False)
    chunks = []
    for i in range(0, len(tokens), max_tokens):
        chunk = tokens[i:i + max_tokens]
        chunks.append(tokenizer.decode(chunk))
    return chunks

# Example Usage
document = ("This is a sample document to demonstrate dynamic chunking. "
            "The tokenizer adapts the chunks based on token limits.")
chunks = dynamic_chunking(document, max_tokens=10)
for idx, chunk in enumerate(chunks):
    print(f"Chunk {idx + 1}: {chunk}")
```

**Output:**

- Chunk 1: this is a sample document to demonstrate dynamic chunking
- Chunk 2: . the tokenizer adapts the chunks based on
- Chunk 3: token limits.

Comparison of Strategies

Strategy	Pros	Cons
Fixed-Length Chunking	Simple, fast	May split text mid-sentence or lose coherence.
Sliding Window Chunking	Preserves context	Overlapping increases redundancy.
Semantic Chunking	Coherent chunks	Requires NLP preprocessing.
Dynamic Chunking	Adapts to content	Computationally intensive.

Each strategy has its strengths and weaknesses. Select based on the task requirements, context, and available computational resources.

The optimal chunk length depends on the type of content being processed and the intended use case. Below are recommendations for chunk lengths based on different context types, along with their rationale:

Context Type	Chunk Length (Tokens)	Rationale
FAQs or Short Texts	100-200	Short enough to handle specific queries.
Articles or Blog Posts	300-500	Covers logical sections while fitting multiple chunks in the LLM context.
Research Papers or Reports	500-700	Captures detailed sections like methodology or results.
Legal or Technical Texts	200-300	Maintains precision due to dense information.

The valuating Chunking Strategies for Retrieval can be found at: <https://research.trychroma.com/evaluating-chunking>

### 5.2.2 Late Chunking

**Late Chunking** refers to a strategy in Retrieval-Augmented Generation (RAG) where chunking of data is **deferred until query time**. Unlike pre-chunking, where documents are split into chunks during preprocessing, late chunking dynamically extracts relevant content when a query is made.

- Key Concepts of Late Chunking
  - **Dynamic Chunk Creation:**
    - \* Full documents or large sections are stored in the vector database.
    - \* Relevant chunks are dynamically extracted at query time based on the query and similarity match.
  - **Query-Time Optimization:**
    - \* The system identifies relevant content using similarity search or semantic analysis.
    - \* Only the most relevant content is chunked and passed to the language model.
  - **Reduced Preprocessing Time:**
    - \* Eliminates extensive preprocessing and fixed chunking during data ingestion.
    - \* Higher computational cost occurs during query-time retrieval.

The following implementations are from Jina AI, and the copyright belongs to the original author.

```
def chunk_by_sentences(input_text: str, tokenizer: callable):
    """
    Split the input text into sentences using the tokenizer
    :param input_text: The text snippet to split into sentences
    :param tokenizer: The tokenizer to use
    :return: A tuple containing the list of text chunks and their corresponding
    token spans
    """
    inputs = tokenizer(input_text, return_tensors='pt', return_offsets_
    
```

(continues on next page)

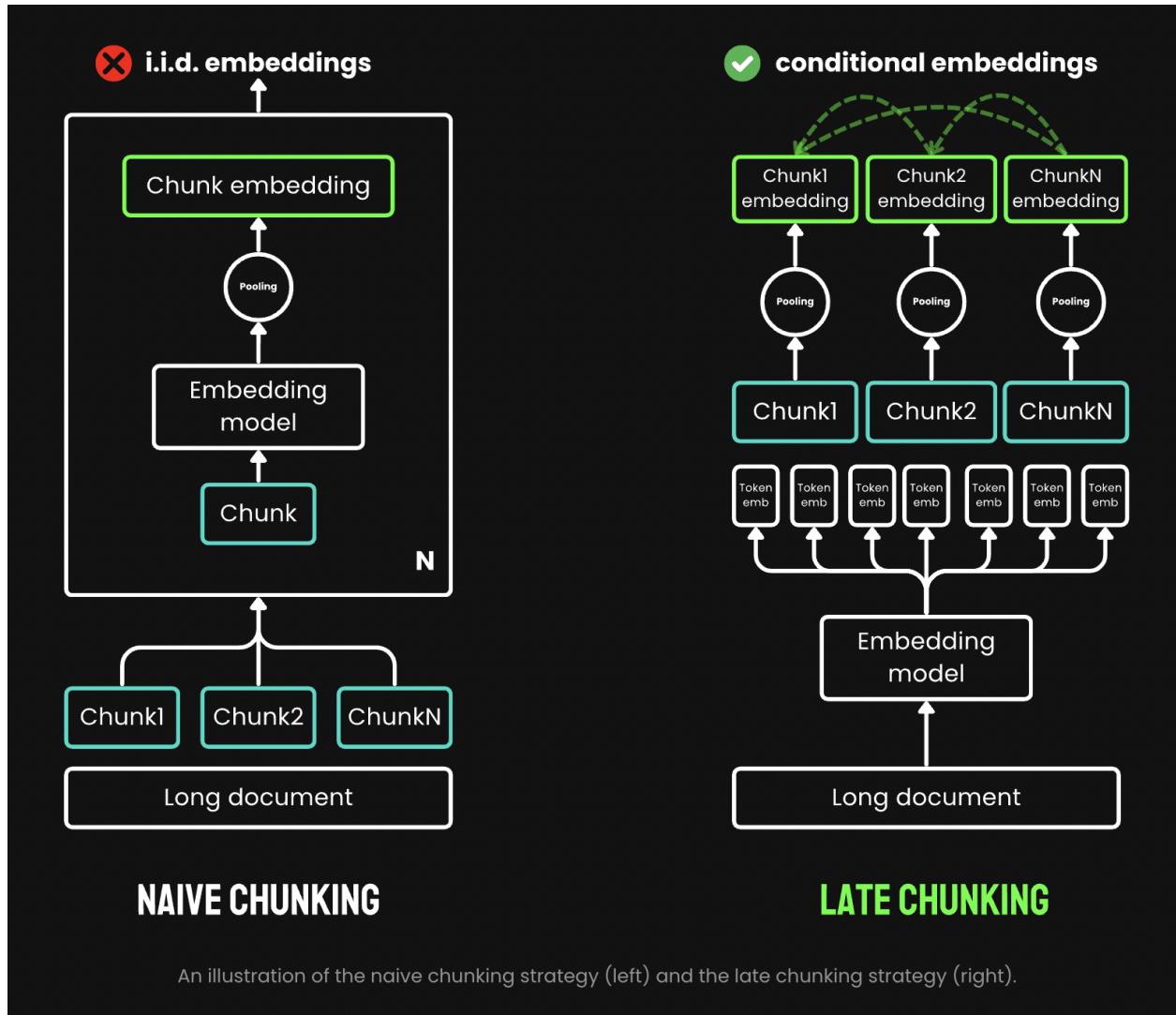


Fig. 2: An illustration of the naive chunking strategy (left) and the late chunking strategy (right). (Source [Jina AI](#))

(continued from previous page)

```

→mapping=True)
    punctuation_mark_id = tokenizer.convert_tokens_to_ids('.')
    sep_id = tokenizer.convert_tokens_to_ids('[SEP]')
    token_offsets = inputs['offset_mapping'][0]
    token_ids = inputs['input_ids'][0]
    chunk_positions = [
        (i, int(start + 1))
        for i, (token_id, (start, end)) in enumerate(zip(token_ids, token_
→offsets))
        if token_id == punctuation_mark_id
        and (
            token_offsets[i + 1][0] - token_offsets[i][1] > 0
            or token_ids[i + 1] == sep_id
        )
    ]
    chunks = [
        input_text[x[1] : y[1]]
        for x, y in zip([(1, 0)] + chunk_positions[:-1], chunk_positions)
    ]
    span_annotations = [
        (x[0], y[0]) for (x, y) in zip([(1, 0)] + chunk_positions[:-1], chunk_
→positions)
    ]
    return chunks, span_annotations

def late_chunking(
    model_output: 'BatchEncoding', span_annotation: list, max_length=None
):
    token_embeddings = model_output[0]
    outputs = []
    for embeddings, annotations in zip(token_embeddings, span_annotation):
        if (
            max_length is not None
        ): # remove annotations which go beyond the max-length of the model
            annotations = [
                (start, min(end, max_length - 1))
                for (start, end) in annotations
                if start < (max_length - 1)
            ]
        pooled_embeddings = [
            embeddings[start:end].sum(dim=0) / (end - start)
            for start, end in annotations
            if (end - start) >= 1
        ]
        pooled_embeddings = [
            embedding.detach().cpu().numpy() for embedding in pooled_embeddings
        ]

```

(continues on next page)

(continued from previous page)

```

    ]
outputs.append(pooled_embeddings)

return outputs

```

```

input_text = "Berlin is the capital and largest city of Germany, both by area and by population. Its more than 3.85 million inhabitants make it the European Union's most populous city, as measured by population within city limits. The city is also one of the states of Germany, and is the third smallest state in the country in terms of area."

# determine chunks
chunks, span_annotations = chunk_by_sentences(input_text, tokenizer)
print('Chunks:\n- ' + '\n- '.join(chunks) + '\n')

# chunk before
embeddings_traditional_chunking = model.encode(chunks)

# chunk afterwards (context-sensitive chunked pooling)
inputs = tokenizer(input_text, return_tensors='pt')
model_output = model(**inputs)
embeddings = late_chunking(model_output, [span_annotations])[0]

import numpy as np

cos_sim = lambda x, y: np.dot(x, y) / (np.linalg.norm(x) * np.linalg.norm(y))

berlin_embedding = model.encode('Berlin')

for chunk, new_embedding, trad_embeddings in zip(chunks, embeddings, embeddings_traditional_chunking):
    print(f'similarity_new("Berlin", "{chunk}"): ', cos_sim(berlin_embedding, new_embedding))
    print(f'similarity_trad("Berlin", "{chunk}"): ', cos_sim(berlin_embedding, trad_embeddings))

```

Quer Chunk	similar- ity_new	similar- ity_trad
Berli Berlin is the capital and largest city of Germany, both by area and by population.	0.849546	0.8486219
Berli Its more than 3.85 million inhabitants make it the European Union's most populous city, as measured by population within city limits."	0.8248902	0.70843387
Berli The city is also one of the states of Germany, and is the third smallest state in the country in terms of area."	0.8498009	0.75345534

### 5.2.3 Types of Indexing

The embedding methods we introduced in Chapter *Word and Sentence Embedding* can be applied here to convert each chunk into embeddings and create indexing. These indexings(embeddings) will be used to retrieve relevant documents or information.

- Sparse Indexing:

Uses traditional keyword-based methods (e.g., TF-IDF, BM25). Index stores the frequency of terms and their associations with documents.

- Advantages: Easy to understand and deploy and works well for exact matches or keyword-heavy queries.
- Disadvantages: Struggles with semantic understanding or paraphrased queries.

- Dense Indexing:

Uses vector embeddings to capture semantic meaning. Documents are represented as vectors in a high-dimensional space, enabling similarity search.

- Advantages: Excellent for semantic search, handling synonyms, and paraphrasing.
- Disadvantages: Requires more computational resources for storage and retrieval.

- Hybrid Indexing:

Combines sparse and dense indexing for more robust search capabilities. For example, Elasticsearch can integrate BM25 with vector search.

### 5.2.4 Vector Database

Vector databases are essential for Retrieval-Augmented Generation (RAG) systems, enabling efficient similarity search on dense vector embeddings. Below is a comprehensive overview of popular vector databases for RAG workflows:

#### 1. FAISS (Facebook AI Similarity Search)

- **Description:** - An open-source library developed by Facebook AI for efficient similarity search and clustering of dense vectors.
- **Features:** - High performance and scalability. - Supports various indexing methods like Flat, IVF, and HNSW. - GPU acceleration for faster searches.
- **Use Cases:** - Research and prototyping. - Scenarios requiring custom implementations.
- **Limitations:** - File-based storage; lacks a built-in distributed or managed cloud solution.
- **Official Website:** [FAISS GitHub](#)

#### 2. Pinecone

- **Description:** - A fully managed vector database designed for production-scale workloads.
- **Features:** - Scalable and serverless architecture. - Automatic scaling and optimization of indexes. - Hybrid search (combining vector and keyword search). - Integrates with popular frameworks like LangChain and OpenAI.

- **Use Cases:** - Enterprise-grade applications. - Handling large datasets with minimal operational overhead.
- **Official Website:** [Pinecone](#)

### 3. Weaviate

- **Description:** - An open-source vector search engine with a strong focus on modularity and customization.
- **Features:** - Supports hybrid search and symbolic reasoning. - Schema-based data organization. - Plugin support for pre-built and custom vectorization modules. - Cloud-managed and self-hosted options.
- **Use Cases:** - Applications requiring hybrid search capabilities. - Knowledge graphs and semantically rich data.
- **Official Website:** [Weaviate](#)

### 4. Milvus

- **Description:** - An open-source, high-performance vector database designed for similarity search on large datasets.
- **Features:** - Distributed and scalable architecture. - Integration with FAISS, Annoy, and HNSW indexing techniques. - Built-in support for time travel queries (searching historical data).
- **Use Cases:** - Video, audio, and image search applications. - Large-scale datasets requiring real-time indexing and retrieval.
- **Official Website:** [Milvus](#)

### 5. Qdrant

- **Description:** - An open-source, lightweight vector database focused on ease of use and modern developer needs.
- **Features:** - Supports HNSW for efficient vector search. - Advanced filtering capabilities for combining metadata with vector queries. - REST and gRPC APIs for integration. - Docker-ready deployment.
- **Use Cases:** - Scenarios requiring metadata-rich search. - Lightweight deployments with simplicity in mind.
- **Official Website:** [Qdrant](#)

### 6. Redis (with Vector Similarity Search Module)

- **Description:** - A popular in-memory database with a module for vector similarity search.
- **Features:** - Combines vector search with traditional key-value storage. - Supports hybrid search and metadata filtering. - High throughput and low latency due to in-memory architecture.
- **Use Cases:** - Applications requiring real-time, low-latency search. - Integrating vector search with existing Redis-based systems.
- **Official Website:** [Redis Vector Search](#)

### 7. Zilliz

- **Description:** - A cloud-native vector database built on Milvus for scalable and managed vector storage.
- **Features:** - Fully managed service for vector data. - Seamless scaling and distributed indexing. - Integration with machine learning pipelines.
- **Use Cases:** - Large-scale enterprise deployments. - Cloud-native solutions with minimal infrastructure management.
- **Official Website:** [Zilliz](#)

## 8. Vespa

- **Description:** - A real-time serving engine supporting vector and hybrid search.
- **Features:** - Combines vector search with advanced ranking and filtering. - Scales to large datasets with support for distributed clusters. - Powerful query configuration options.
- **Use Cases:** - E-commerce and recommendation systems. - Applications with complex ranking requirements.
- **Official Website:** [Vespa](#)

## 9. Chroma

- **Description:** - An open-source, user-friendly vector database built for LLMs and embedding-based applications.
- **Features:** - Designed specifically for RAG workflows. - Simple Python API for seamless integration with AI models. - Efficient and customizable vector storage for embedding data.
- **Use Cases:** - Prototyping and experimentation for LLM-based applications. - Lightweight deployments for small to medium-scale RAG systems.
- **Official Website:** [Chroma](#)

Comparison of Vector Databases:

Database	Open Source	Managed Service	Key Features	Best For
FAISS	Yes	No	High performance, GPU acceleration	Research, prototyping
Pinecone	No	Yes	Serverless, automatic scaling	Enterprise-scale applications
Weaviate	Yes	Yes	Hybrid search, modularity	Knowledge graphs
Milvus	Yes	No	Distributed, high performance	Large-scale datasets
Qdrant	Yes	No	Lightweight, metadata filtering	Small to medium-scale apps
Redis	No	Yes	In-memory performance, hybrid search	Real-time apps
Zilliz	No	Yes	Fully managed Milvus	Enterprise cloud solutions
Vespa	Yes	No	Hybrid search, real-time ranking	E-commerce, recommendations
Chroma	Yes	No	LLM-focused, simple API	Prototyping, lightweight apps

### Choosing a Vector Database

- **For Research or Small Projects:** FAISS, Qdrant, Milvus, or Chroma.
- **For Enterprise or Cloud-Native Workflows:** Pinecone, Zilliz, or Weaviate.
- **For Real-Time Use Cases:** Redis or Vespa.

Each database has unique strengths and is suited for specific RAG use cases. The choice depends on scalability, integration needs, and budget.

## 5.3 Retrieval

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

### 5.3.1 Common retrieval methods

- **Sparse Vector Search:** Traditional keyword-based retrieval (e.g., TF-IDF, BM25).
- **Dense Vector Search:** Vector-based search using embeddings e.g.
  - **Approximate Nearest Neighbor (ANN) Search:**
    - \* HNSW (Hierarchical Navigable Small World): Graph-based approach
    - \* IVF (Inverted File Index): Clusters embeddings into groups and searches within relevant clusters.
  - **Exact Nearest Neighbor Search:** Computes similarities exhaustively for all vectors in the corpus
- **Hybrid Search** (Fig *Reciprocal Rank Fusion*): the combination of Sparse and Dense vector search.

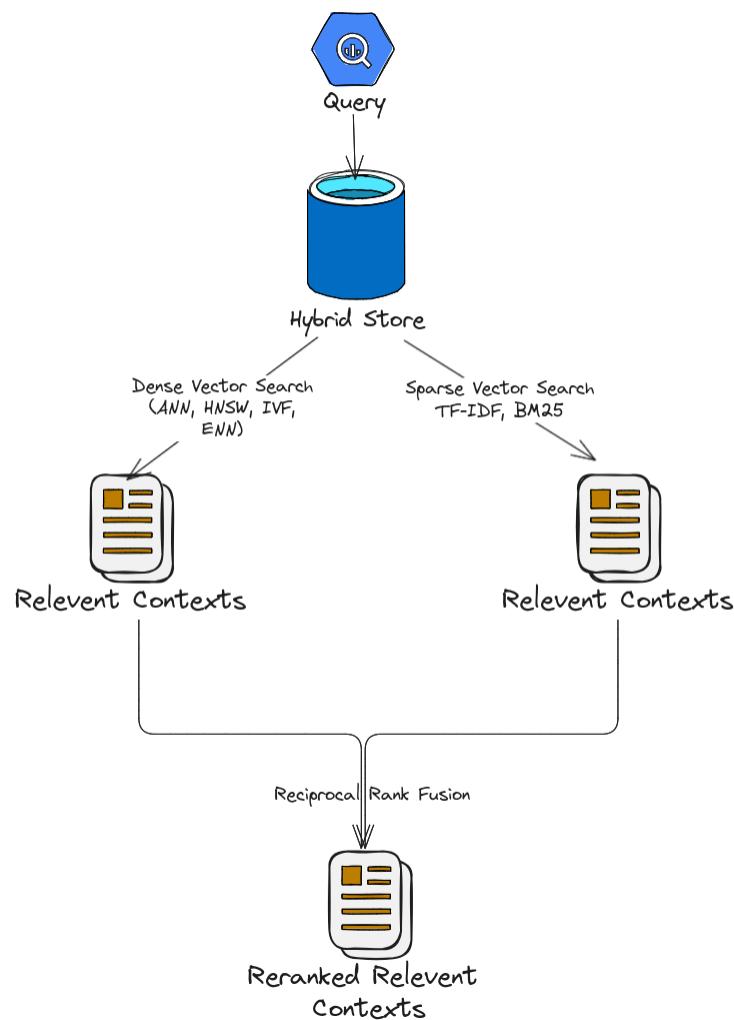


Fig. 3: Reciprocal Rank Fusion

## Note

BM25 (Best Matching 25) is a popular ranking function used by search engines and information retrieval systems to rank documents based on their relevance to a given query. It belongs to the family of **bag-of-words retrieval models** and is an enhancement of the **TF-IDF (Term Frequency-Inverse Document Frequency)** approach.

- **Key Features of BM25**

1. **Relevance Scoring:**

- BM25 scores documents by measuring how well the query terms match the terms in the document.
- It incorporates term frequency, inverse document frequency, and document length normalization.

2. **Formula:**

The BM25 score for a document D given a query Q is calculated as:

$$\text{BM25}(D, Q) = \sum_{t \in Q} \text{IDF}(t) \cdot \frac{f(t, D) \cdot (k_1 + 1)}{f(t, D) + k_1 \cdot (1 - b + b \cdot \frac{|D|}{\text{avgdl}})}$$

Where: - t: Query term. - f(t, D): Frequency of term t in document D. - |D|: Length of document D (number of terms). - avgdl: Average document length in the corpus. - k1: Tuning parameter that controls term frequency saturation (usually set between 1.2 and 2.0). - b: Tuning parameter that controls length normalization (usually set to 0.75). - IDF(t): Inverse Document Frequency of term t, calculated as:

$$\text{IDF}(t) = \log \frac{N - n_t + 0.5}{n_t + 0.5}$$

Where N is the total number of documents in the corpus, and n\_t is the number of documents containing t.

3. **Improvements Over TF-IDF:**

- Document Length Normalization: BM25 adjusts for the length of documents, addressing the bias of TF-IDF toward longer documents.
- Saturation of Term Frequency: BM25 avoids the overemphasis of excessively high term frequencies by using a non-linear saturation function controlled by k1.

4. **Applications:**

- **Information Retrieval:** Ranking search results by relevance.
- **Question Answering:** Identifying relevant documents or passages for a query.
- **Document Matching:** Comparing similarities between textual content.

5. **Limitations:**

- BM25 does not consider semantic meanings or relationships between words, relying solely on exact term matches.

- It may struggle with queries or documents that require contextual understanding.

Summary of Common Algorithms:

Met- ric/Algorith m	Purpose	Common Use
<b>TF-IDF</b>	Keyword matching with term weighting.	Effective for small-scale or structured corpora.
<b>BM25</b>	Advanced keyword matching with term frequency saturation and document length normalization.	Widely used in sparse search; default in tools like Elasticsearch and Solr.
<b>Cosine Similarity</b>	Measures orientation (ignores magnitude).	Widely used; works well with normalized vectors.
<b>Dot Product Similarity</b>	Measures magnitude and direction.	Preferred in embeddings like OpenAI's models.
<b>Euclidean Distance</b>	Measures absolute distance between vectors.	Less common but used in some specific cases.
<b>HNSW (ANN)</b>	Fast and scalable nearest neighbor search.	Default for large-scale systems (e.g., FAISS).
<b>IVF (ANN)</b>	Efficient clustering-based search.	Often combined with product quantization.

### 5.3.2 Reciprocal Rank Fusion

Reciprocal Rank Fusion (RRF) is a ranking technique commonly used in information retrieval and ensemble learning. Although it is not specific to large language models (LLMs), it can be applied to scenarios where multiple ranking systems (or scoring mechanisms) produce different rankings, and you want to combine them into a single, unified ranking.

The reciprocal rank of an item in a ranked list is calculated as  $\frac{1}{k+r}$ , where

- r is the rank of the item (1 for the top rank, 2 for the second rank, etc.).
- k is a small constant (often set to 60 or another fixed value) to control how much weight is given to higher ranks.

Example:

Suppose two retrieval models give ranked lists for query responses:

- Model 1 ranks documents as: [A,B,C,D]
- Model 2 ranks documents as: [B,A,D,C]

RRF combines these rankings by assigning each document a combined score:

- Document A:  $\frac{1}{60+1} + \frac{1}{60+2} = 0.03252247488101534$
- Document B:  $\frac{1}{60+2} + \frac{1}{60+1} = 0.03252247488101534$
- Document C:  $\frac{1}{60+3} + \frac{1}{60+4} = 0.03149801587301587$

- Document D:  $\frac{1}{60+4} + \frac{1}{60+3} = 0.03149801587301587$

```
from collections import defaultdict

def reciprocal_rank_fusion(ranked_results: list[list], k=60):
    """
    Fuse rank from multiple retrieval systems using Reciprocal Rank Fusion.

    Args:
        ranked_results: Ranked results from different retrieval system.
        k (int): A constant used in the RRF formula (default is 60).

    Returns:
        Tuple of list of sorted documents by score and sorted documents
    """

    # Dictionary to store RRF mapping
    rrf_map = defaultdict(float)

    # Calculate RRF score for each result in each list
    for rank_list in ranked_results:
        for rank, item in enumerate(rank_list, 1):
            rrf_map[item] += 1 / (rank + k)

    # Sort items based on their RRF scores in descending order
    sorted_items = sorted(rrf_map.items(), key=lambda x: x[1], reverse=True)

    # Return tuple of list of sorted documents by score and sorted documents
    return sorted_items, [item for item, score in sorted_items]

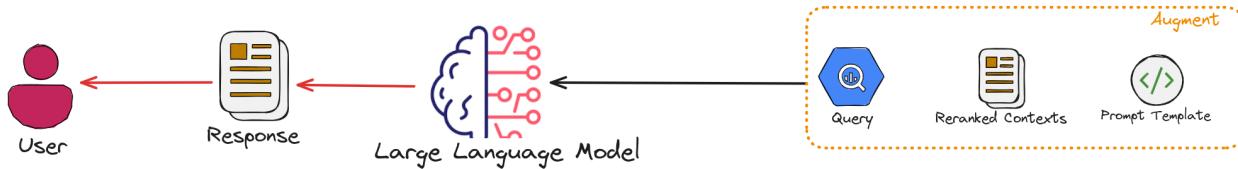
# Example ranked lists from different sources
ranked_a = ['A', 'B', 'C', 'D']
ranked_b = ['B', 'A', 'D', 'C']

# Combine the lists using RRF
combined_list = reciprocal_rank_fusion([ranked_a, ranked_b])
print(combined_list)
```

```
((('A', 0.03252247488101534), ('B', 0.03252247488101534), ('C', 0.
˓→03149801587301587), ('D', 0.03149801587301587)), ['A', 'B', 'C', 'D'])
```

## 5.4 Generation

Finally, the retrieved relevant information will be feed back into the LLMs to generate responses.



### Note

In the remainder of this implementation, we will use the following components:

- Vector database: Chroma
- Embedding model: BAAI/bge-m3
- LLM: mistral
- Web search engine: Google

```

# Load models
from langchain_ollama import OllamaEmbeddings
from langchain_ollama.llms import OllamaLLM

## embedding model
embedding = OllamaEmbeddings(model="bge-m3")

## LLM
llm = OllamaLLM(temperature=0.0, model='mistral', format='json')

# Indexing
from langchain.text_splitter import RecursiveCharacterTextSplitter
from langchain_community.document_loaders import WebBaseLoader
from langchain_community.vectorstores import Chroma
from langchain_ollama import OllamaEmbeddings # Import OllamaEmbeddings instead

urls = [
    "https://python.langchain.com/v0.1/docs/get_started/introduction/",
]

docs = [WebBaseLoader(url).load() for url in urls]
docs_list = [item for sublist in docs for item in sublist]

text_splitter = RecursiveCharacterTextSplitter.from_tiktoken_encoder(
    chunk_size=250, chunk_overlap=0
)

```

(continues on next page)

(continued from previous page)

```

)
doc_splits = text_splitter.split_documents(docs_list)

# Add to vectorDB
vectorstore = Chroma.from_documents(
    documents=doc_splits,
    collection_name="rag-chroma",
    embedding=OllamaEmbeddings(model="bge-m3"),
)

# Retriever
retriever = vectorstore.as_retriever(k=5)

# Generation
questions = [
    "what is LangChain?",
]

for question in questions:
    retrieved_context = retriever.invoke(question)
    formatted_prompt = prompt.format(context=retrieved_context, ↵
                                      question=question)
    response_from_model = model.invoke(formatted_prompt)
    parsed_response = parser.parse(response_from_model)

    print(f"Question: {question}")
    print(f"Answer: {parsed_response}")
    print()

```

```

Answer: {
    "answer": "LangChain refers to chains, agents, and retrieval strategies that ↵
make up an application's cognitive architecture."
}

```

## 5.5 Advanced Topic

### 5.5.1 Self-RAG

In the paper [selfRAG], Four types decisions are made:

1. Should I retrieve from retriever, R
  - **Input:** -  $x$  (question) - OR  $x$  (question),  $y$  (generation)
  - **Description:** Decides when to retrieve  $D$  chunks with  $R$ .
  - **Output:** - yes - no - continue

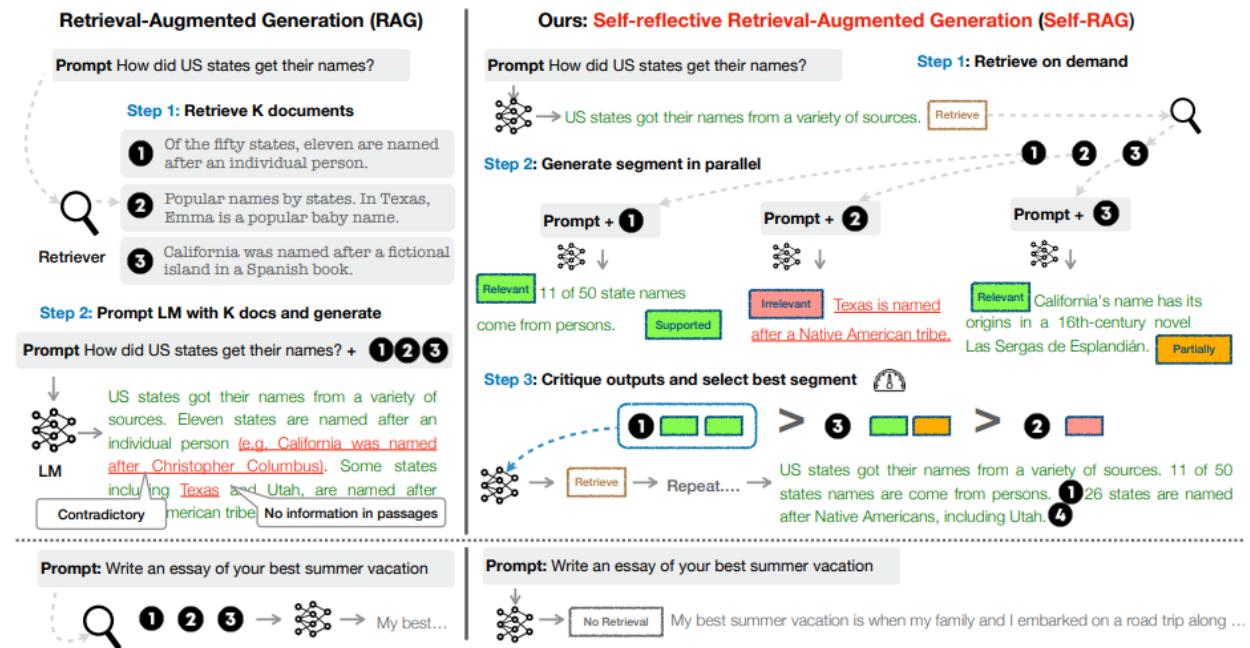


Fig. 4: Overview of SELF-RAG. (Source [selfRAG])

## 2. Are the retrieved passages D relevant to the question x

- Input:** -  $(x \text{ (question)}, d \text{ (chunk)})$  for  $d$  in  $D$
- Description:** Determines if  $d$  provides useful information to solve  $x$ .
- Output:** - *relevant* - *irrelevant*

## 3. Are the LLM generations from each chunk in D relevant to the chunk (hallucinations, etc.)

- Input:** -  $x \text{ (question)}, d \text{ (chunk)}, y \text{ (generation)}$  for  $d$  in  $D$
- Description:** Verifies if all statements in  $y$  (generation) are supported by  $d$ .
- Output:** - *fully supported* - *partially supported* - *no support*

## 4. Is the LLM generation from each chunk in D a useful response to x (question)

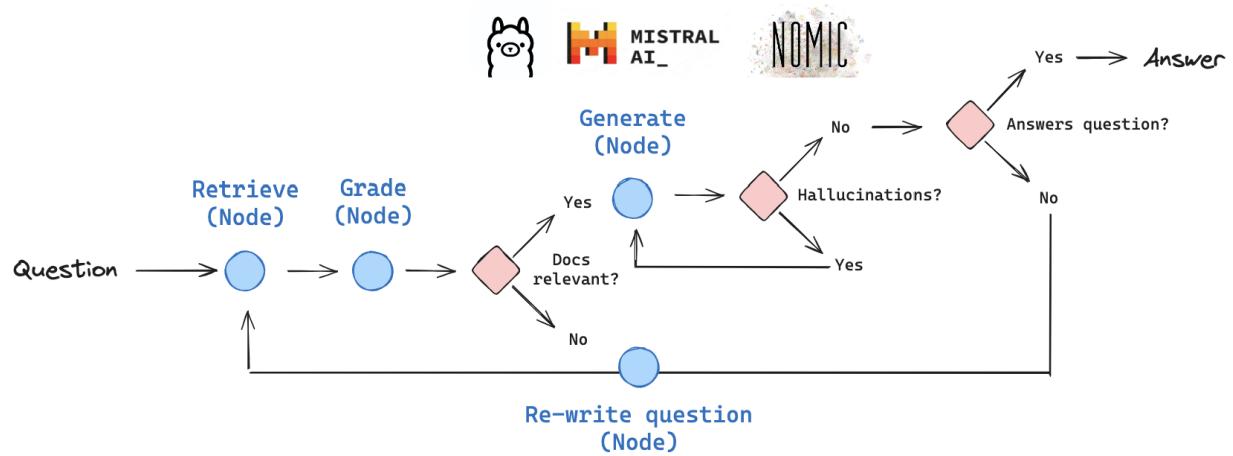
- Input:** -  $x \text{ (question)}, y \text{ (generation)}$  for  $d$  in  $D$
- Description:** Assesses if  $y$  (generation) is a useful response to  $x$  (question).
- Output:** -  $\{5, 4, 3, 2, 1\}$

- Load Models

```
from langchain_llama import OllamaEmbeddings
from langchain_llms import OllamaLLM
```

```
# embedding model
```

(continues on next page)

Fig. 5: Self-RAG langgraph diagram (source [Langgraph self-rag](#))

(continued from previous page)

```
embedding = OllamaEmbeddings(model="bge-m3")

# LLM
llm = OllamaLLM(temperature=0.0, model='mistral', format='json')
```

**Warning**

You need to specify `format='json'` when Initializing `OllamaLLM`. otherwise you will get error:

The screenshot shows a Google Colab notebook titled "Run-ollama-in-colab.ipynb". The code cell [4] contains Python code for setting up an OllamaLLM instance and generating a response. The output shows the download of "httpx-0.27.2-py3-none-any.whl" and the successful installation of "httpx", "ollama", and "langchain-ollama". The generated JSON response is displayed, including the prompt template and the AI's response.

```

from langchain_core.prompts import ChatPromptTemplate
from langchain_ollama.llms import OllamaLLM

template = """Question: {question}
Answer: Let's think step by step.

"""
prompt = ChatPromptTemplate.from_template(template)

llm = OllamaLLM(model="mistral", format='json')

# Generate a response
response = llm.invoke("Explain the concept of artificial intelligence in simple terms.")
print(response)

```

The sidebar shows resources like RAM and Disk usage, and a validation error message: "ValidationError" from "langchain\_ollama.llms". A note says "Use code with caution". An explanation of changes highlights the addition of the "format='json'" parameter to the OllamaLLM class.

- Create Index

```

from langchain.text_splitter import RecursiveCharacterTextSplitter
from langchain_community.document_loaders import WebBaseLoader
from langchain_community.vectorstores import Chroma
from langchain_ollama import OllamaEmbeddings # Import
→ OllamaEmbeddings instead

urls = [
    "https://lilianweng.github.io/posts/2023-06-23-agent/",
    "https://lilianweng.github.io/posts/2023-03-15-prompt-engineering/",
    "https://lilianweng.github.io/posts/2023-10-25-adv-attack-llm/",
]

docs = [WebBaseLoader(url).load() for url in urls]
docs_list = [item for sublist in docs for item in sublist]

text_splitter = RecursiveCharacterTextSplitter.from_tiktoken_encoder(
    chunk_size=250, chunk_overlap=0
)
doc_splits = text_splitter.split_documents(docs_list)

```

(continues on next page)

(continued from previous page)

```
# Add to vectorDB
vectorstore = Chroma.from_documents(
    documents=doc_splits,
    collection_name="rag-chroma",
    embedding=OllamaEmbeddings(model="bge-m3"),
)
retriever = vectorstore.as_retriever()
```

- Retrieval Grader

```
### Retrieval Grader

from langchain_ollama.llms import OllamaLLM
from langchain.prompts import PromptTemplate
from langchain_community.chat_models import ChatOllama
from langchain_core.output_parsers import JsonOutputParser

from langchain_core.pydantic_v1 import BaseModel, Field
from langchain.output_parsers import PydanticOutputParser

# Data model
class GradeDocuments(BaseModel):
    """Binary score for relevance check on retrieved documents."""

    score: str = Field( # Changed field name to 'score'
        description="Documents are relevant to the question, 'yes' or 'no'")

parser = PydanticOutputParser(pydantic_object=GradeDocuments)

prompt = PromptTemplate(
    template="""You are a grader assessing relevance of a retrieved
document to a user question. \n
Here is the retrieved document: \n\n {document} \n\n
Here is the user question: {question} \n
If the document contains keywords related to the user question,
grade it as relevant. \n
It does not need to be a stringent test. The goal is to filter out
erroneous retrievals. \n
Give a binary score 'yes' or 'no' score to indicate whether the document
is relevant to the question. \n
Provide the binary score as a JSON with a single key 'score' and no
preamble or explanation."""",
    input_variables=["question", "document"],
    partial_variables={"format_instructions": parser.get_format_
    instructions()})
)
```

(continues on next page)

(continued from previous page)

```

retrieval_grader = prompt | llm | parser
question = "agent memory"
docs = retriever.invoke(question)
doc_txt = docs[1].page_content
retrieval_grader.invoke({"question": question, "document": doc_txt})

```

Output:

```
GradeDocuments(score='yes')
```

### Warning

OllamaLLM does not have `with_structured_output(GradeDocuments)`. You need to use

- `PydanticOutputParser(pydantic_object=GradeDocuments)`
- `partial_variables={"format_instructions": parser.get_format_instructions()}`

to format the structured output.

- Generate

```

### Generate

from langchain import hub
from langchain_core.output_parsers import StrOutputParser

# Prompt
prompt = hub.pull("rlm/rag-prompt")

# LLM
llm = OllamaLLM(temperature=0.0, model='mistral', format='json')

# Post-processing
def format_docs(docs):
    return "\n\n".join(doc.page_content for doc in docs)

# Chain
rag_chain = prompt | llm | StrOutputParser()

# Run
generation = rag_chain.invoke({"context": docs, "question": question})
print(generation)

```

Output:

```
{
  "Component Two: Memory": [
    {
      "Types of Memory": [
        {
          "Sensory Memory": [
            "This is the earliest stage of memory, providing the ability to retain impressions of sensory information (visual, auditory, etc) after the original stimuli have ended. Sensory memory typically only lasts for up to a few seconds. Subcategories include iconic memory (visual), echoic memory (auditory), and haptic memory (touch)."
          ],
          "Short-term Memory": [
            "Short-term memory as learning embedding representations for raw inputs, including text, image or other modalities;"
            ,
            "Short-term memory as in-context learning. It is short and finite, as it is restricted by the finite context window length of Transformer."
          ],
          "Long-term Memory": [
            "Long-term memory as the external vector store that the agent can attend to at query time, accessible via fast retrieval."
          ]
        },
        {
          "Maximum Inner Product Search (MIPS)": [
            "The external memory can alleviate the restriction of finite attention span. A standard practice is to save the embedding representation of information into a vector store database that can support fast maximum inner-product search (MIPS). To optimize the retrieval speed, the common choice is the approximate nearest neighbors (ANN)\u200b algorithm to return approximately top k nearest neighbors to trade off a little accuracy lost for a huge speedup."
            ,
            "A couple common choices of ANN algorithms for fast MIPS:"
          ]
        }
      ]
    }
  ]
}
```

- Hallucination Grader

```

### Hallucination Grader

# Data model
class GradeHallucinations(BaseModel):
    """Binary score for relevance check on retrieved documents."""

    score: str = Field( # Changed field name to 'score'
        description="Documents are relevant to the question, 'yes' or 'no'")

parser = PydanticOutputParser(pydantic_object=GradeHallucinations)

# Prompt
prompt = PromptTemplate(
    template="""You are a grader assessing whether an answer is grounded in
    ↵/
    supported by a set of facts. \n
    Here are the facts:
    \n ----- \n
    {documents}
    \n ----- \n
    Here is the answer: {generation}
    Give a binary score 'yes' or 'no' score to indicate whether
    the answer is grounded in / supported by a set of facts. \n
    Provide the binary score as a JSON with a single key 'score'
    and no preamble or explanation."""",
    input_variables=["generation", "documents"],
    partial_variables={"format_instructions": parser.get_format_
    ↵instructions()}
)

hallucination_grader = prompt | llm | parser
hallucination_grader.invoke({"documents": docs, "generation": generation})

```

Output:

```
GradeHallucinations(score='yes')
```

- Answer Grader

```

### Answer Grader

# Data model
class GradeAnswer(BaseModel):
    """Binary score for relevance check on retrieved documents."""

    score: str = Field( # Changed field name to 'score'
        description="Documents are relevant to the question, 'yes' or 'no'")


```

(continues on next page)

(continued from previous page)

```

parser = PydanticOutputParser(pydantic_object=GradeAnswer)

# Prompt
prompt = PromptTemplate(
    template="""You are a grader assessing whether an answer is useful to
    resolve a question. \n
    Here is the answer:
    \n ----- \n
    {generation}
    \n ----- \n
    Here is the question: {question}
    Give a binary score 'yes' or 'no' to indicate whether
    the answer is useful to resolve a question. \n
    Provide the binary score as a JSON with a single key
    'score' and no preamble or explanation."""",
    input_variables=["generation", "question"],
    partial_variables={"format_instructions": parser.get_format_
    ↪instructions()}
)

answer_grader = prompt | llm | parser
answer_grader.invoke({"question": question, "generation": generation})

```

Output:

```
GradeAnswer(score='yes')
```

- Question Re-writer

```

### Question Re-writer

# Prompt
re_write_prompt = PromptTemplate(
    template="""You a question re-writer that converts an input question
    to a better version that is optimized \n for vectorstore
    retrieval. Look at the input and try to reason about the
    underlying semantic intent / meaning. \n
    Here is the initial question: \n\n {question}.
    Formulate an improved question.\n """,
    input_variables=["generation", "question"]
)

question_rewriter = re_write_prompt | llm | StrOutputParser()
question_rewriter.invoke({"question": question})

```

Output:

```
{ "question": "What is the function or purpose of an agent's memory in a  
→given context?" }
```

- Create the Graph

```
from typing import List

from typing_extensions import TypedDict

class GraphState(TypedDict):
    """
    Represents the state of our graph.

    Attributes:
        question: question
        generation: LLM generation
        documents: list of documents
    """

    question: str
    generation: str
    documents: List[str]

    ### Nodes

    def retrieve(state):
        """
        Retrieve documents

        Args:
            state (dict): The current graph state

        Returns:
            state (dict): New key added to state, documents, that contains
            →retrieved documents
        """
        print("---RETRIEVE---")
        question = state["question"]

        # Retrieval
        documents = retriever.invoke(question)
        return {"documents": documents, "question": question}
```

(continues on next page)

(continued from previous page)

```

def generate(state):
    """
    Generate answer

    Args:
        state (dict): The current graph state

    Returns:
        state (dict): New key added to state, generation, that contains LLM_
        ↪generation
        """

    print("---GENERATE---")
    question = state["question"]
    documents = state["documents"]

    # RAG generation
    generation = rag_chain.invoke({"context": documents, "question":_
    ↪question})
    return {"documents": documents, "question": question, "generation":_
    ↪generation}

def grade_documents(state):
    """
    Determines whether the retrieved documents are relevant to the question.

    Args:
        state (dict): The current graph state

    Returns:
        state (dict): Updates documents key with only filtered relevant_
        ↪documents
        """

    print("---CHECK DOCUMENT RELEVANCE TO QUESTION---")
    question = state["question"]
    documents = state["documents"]

    # Score each doc
    filtered_docs = []
    for d in documents:
        score = retrieval_grader.invoke(
            {"question": question, "document": d.page_content}
        )
        grade = score.score
        if grade == "yes" or grade==1:

```

(continues on next page)

(continued from previous page)

```

        print("---GRADE: DOCUMENT RELEVANT---")
        filtered_docs.append(d)
    else:
        print("---GRADE: DOCUMENT NOT RELEVANT---")
        continue
    return {"documents": filtered_docs, "question": question}

def transform_query(state):
    """
    Transform the query to produce a better question.

    Args:
        state (dict): The current graph state

    Returns:
        state (dict): Updates question key with a re-phrased question
    """

    print("---TRANSFORM QUERY---")
    question = state["question"]
    documents = state["documents"]

    # Re-write question
    better_question = question_rewriter.invoke({"question": question})
    return {"documents": documents, "question": better_question}

### Edges

def decide_to_generate(state):
    """
    Determines whether to generate an answer, or re-generate a question.

    Args:
        state (dict): The current graph state

    Returns:
        str: Binary decision for next node to call
    """

    print("---ASSESS GRADED DOCUMENTS---")
    state["question"]
    filtered_documents = state["documents"]

```

(continues on next page)

(continued from previous page)

```

if not filtered_documents:
    # All documents have been filtered check_relevance
    # We will re-generate a new query
    print(
        "---DECISION: ALL DOCUMENTS ARE NOT RELEVANT TO QUESTION, ↵
    TRANSFORM QUERY---"
    )
    return "transform_query"
else:
    # We have relevant documents, so generate answer
    print("---DECISION: GENERATE---")
    return "generate"

def grade_generation_v_documents_and_question(state):
    """
    Determines whether the generation is grounded in the document and ↵
    answers question.

    Args:
        state (dict): The current graph state

    Returns:
        str: Decision for next node to call
    """

    print("---CHECK HALLUCINATIONS---")
    question = state["question"]
    documents = state["documents"]
    generation = state["generation"]

    score = hallucination_grader.invoke(
        {"documents": documents, "generation": generation}
    )
    grade = score.score

    # Check hallucination
    if grade == "yes":
        print("---DECISION: GENERATION IS GROUNDED IN DOCUMENTS---")
        # Check question-answering
        print("---GRADE GENERATION vs QUESTION---")
        score = answer_grader.invoke({"question": question, "generation": ↵
    generation})
        grade = score.score
        if grade == "yes":
            print("---DECISION: GENERATION ADDRESSES QUESTION---")

```

(continues on next page)

(continued from previous page)

```

        return "useful"
    else:
        print("---DECISION: GENERATION DOES NOT ADDRESS QUESTION---")
        return "not useful"
    else:
        print("---DECISION: GENERATION IS NOT GROUNDED IN DOCUMENTS, RE-TRY-"
        "RETRY---")
        return "not supported"

```

- Compile Graph

```

from langgraph.graph import END, StateGraph, START

workflow = StateGraph(GraphState)

# Define the nodes
workflow.add_node("retrieve", retrieve) # retrieve
workflow.add_node("grade_documents", grade_documents) # grade documents
workflow.add_node("generate", generate) # generatae
workflow.add_node("transform_query", transform_query) # transform_query

# Build graph
workflow.add_edge(START, "retrieve")
workflow.add_edge("retrieve", "grade_documents")
workflow.add_conditional_edges(
    "grade_documents",
    decide_to_generate,
    {
        "transform_query": "transform_query",
        "generate": "generate",
        "out of context": "generate"
    },
)
workflow.add_edge("transform_query", "retrieve")
workflow.add_conditional_edges(
    "generate",
    grade_generation_v_documents_and_question,
    {
        "not supported": END,
        "useful": END,
        "not useful": "transform_query",
    },
)
# Compile
app = workflow.compile()

```

- Graph visualization

```
from IPython.display import Image, display

try:
    display(Image(app.get_graph(xray=True).draw_mermaid_png()))
except:
    pass
```

Ouput

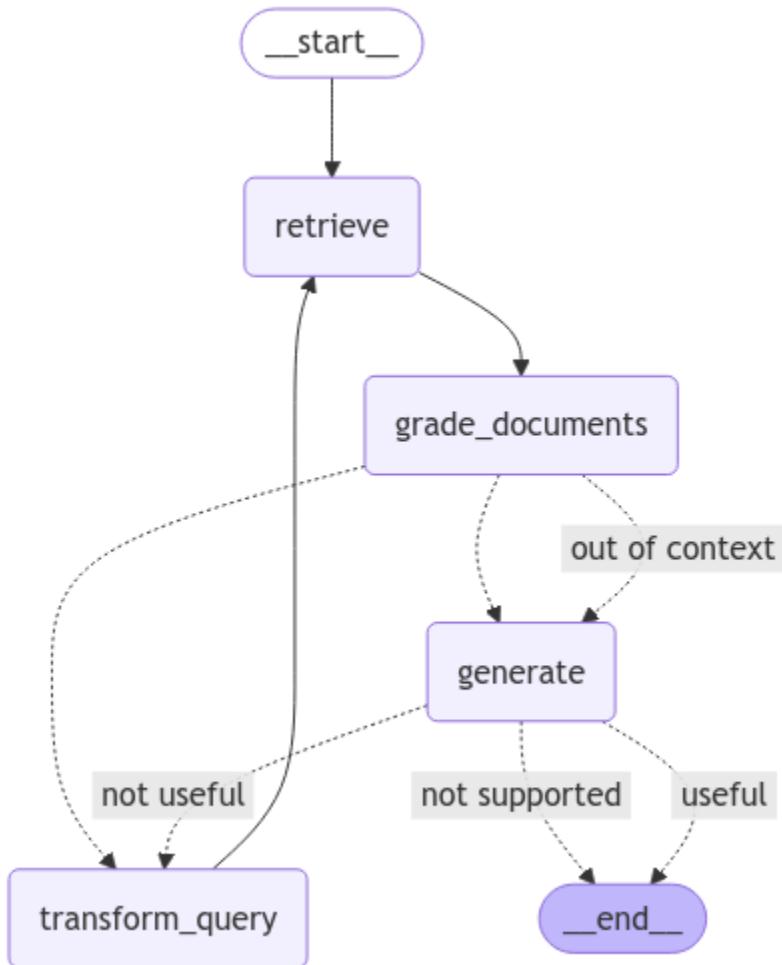


Fig. 6: Self-RAG Graph

- Test
  - Relevant retrieval

```
from pprint import pprint

# Run
```

(continues on next page)

(continued from previous page)

```

inputs = {"question": "What is prompt engineering?"}
for output in app.stream(inputs):
    for key, value in output.items():
        # Node
        pprint(f"Node '{key}':")
        # Optional: print full state at each node
        # pprint.pprint(value["keys"], indent=2, width=80, depth=None)
        pprint("\n---\n")

    # Final generation
    pprint(value["generation"])

```

Output:

```

---RETRIEVE---
"Node 'retrieve':"
'\n---\n'
---CHECK DOCUMENT RELEVANCE TO QUESTION---
---GRADE: DOCUMENT RELEVANT---
---GRADE: DOCUMENT RELEVANT---
---GRADE: DOCUMENT RELEVANT---
---GRADE: DOCUMENT RELEVANT---
---ASSESS GRADED DOCUMENTS---
---DECISION: GENERATE---
"Node 'grade_documents':"
'\n---\n'
---GENERATE---
---CHECK HALLUCINATIONS---
---DECISION: GENERATION IS GROUNDED IN DOCUMENTS---
---GRADE GENERATION vs QUESTION---
---DECISION: GENERATION ADDRESSES QUESTION---
"Node 'generate':"
'\n---\n'
('{\n'
    "Prompt Engineering" : "A method for communicating with language\u202a
    \u202amodels '
    '(LLMs) to steer their behavior towards desired outcomes without\u202a
    \u202aupdating '
    'model weights. It involves alignment and model steerability, and\u202a
    \u202arequires '
    'heavy experimentation and heuristics."\n'
}'')

```

- Irrelevant retrieval

```
from pprint import pprint
```

(continues on next page)

(continued from previous page)

```

# Run
inputs = {"question": "SegRNN?"}
for output in app.stream(inputs):
    for key, value in output.items():
        # Node
        pprint(f"Node '{key}':")
        # Optional: print full state at each node
        # pprint.pprint(value["keys"], indent=2, width=80, depth=None)
        pprint("\n---\n")

# Final generation
pprint(value["generation"])

```

Output:

```

---RETRIEVE---
"Node 'retrieve':"
'\n---\n'
---CHECK DOCUMENT RELEVANCE TO QUESTION---
---GRADE: DOCUMENT NOT RELEVANT---
---GRADE: DOCUMENT NOT RELEVANT---
---GRADE: DOCUMENT NOT RELEVANT---
---GRADE: DOCUMENT NOT RELEVANT---
---ASSESS GRADED DOCUMENTS---
---DECISION: ALL DOCUMENTS ARE NOT RELEVANT TO QUESTION, TRANSFORM_
↳ QUERY---
"Node 'grade_documents':"
'\n---\n'
---TRANSFORM QUERY---
"Node 'transform_query':"
'\n---\n'
---RETRIEVE---
"Node 'retrieve':"
'\n---\n'
---CHECK DOCUMENT RELEVANCE TO QUESTION---
---GRADE: DOCUMENT NOT RELEVANT---
---GRADE: DOCUMENT NOT RELEVANT---
---GRADE: DOCUMENT NOT RELEVANT---
---GRADE: DOCUMENT NOT RELEVANT---
---ASSESS GRADED DOCUMENTS---
---DECISION: ALL DOCUMENTS ARE NOT RELEVANT TO QUESTION, TRANSFORM_
↳ QUERY---
"Node 'grade_documents':"
'\n---\n'
---TRANSFORM QUERY---

```

(continues on next page)

(continued from previous page)

```
"Node 'transform_query':"
'\n---\n'
---RETRIEVE---
"Node 'retrieve':"
'\n---\n'
---CHECK DOCUMENT RELEVANCE TO QUESTION---
---GRADE: DOCUMENT NOT RELEVANT---
---GRADE: DOCUMENT NOT RELEVANT---
---GRADE: DOCUMENT NOT RELEVANT---
---GRADE: DOCUMENT NOT RELEVANT---
---ASSESS GRADED DOCUMENTS---
---DECISION: ALL DOCUMENTS ARE NOT RELEVANT TO QUESTION, TRANSFORM_
    ↵QUERY---
"Node 'grade_documents':"
'\n---\n'
---TRANSFORM QUERY---
"Node 'transform_query':"
'\n---\n'
---RETRIEVE---
"Node 'retrieve':"
'\n---\n'
---CHECK DOCUMENT RELEVANCE TO QUESTION---
---GRADE: DOCUMENT NOT RELEVANT---
---GRADE: DOCUMENT NOT RELEVANT---
---GRADE: DOCUMENT NOT RELEVANT---
---GRADE: DOCUMENT NOT RELEVANT---
---ASSESS GRADED DOCUMENTS---
---DECISION: ALL DOCUMENTS ARE NOT RELEVANT TO QUESTION, TRANSFORM_
    ↵QUERY---
"Node 'grade_documents':"
'\n---\n'
---TRANSFORM QUERY---
"Node 'transform_query':"
'\n---\n'
---RETRIEVE---
"Node 'retrieve':"
'\n---\n'
---CHECK DOCUMENT RELEVANCE TO QUESTION---
---GRADE: DOCUMENT RELEVANT---
---GRADE: DOCUMENT NOT RELEVANT---
---GRADE: DOCUMENT NOT RELEVANT---
---GRADE: DOCUMENT RELEVANT---
---ASSESS GRADED DOCUMENTS---
---DECISION: GENERATE---
"Node 'grade_documents':"
'\n---\n'
```

(continues on next page)

(continued from previous page)

```

---GENERATE---
---CHECK HALLUCINATIONS---
---DECISION: GENERATION IS NOT GROUNDED IN DOCUMENTS, RE-TRY---
"Node 'generate':"
'\n---\n'
('{\n'
'    "Question": "Define and provide an explanation for a Sequential "
'Recurrent Neural Network (SegRNN)",\n'
'    "Answer": "A Sequential Recurrent Neural Network (SegRNN) is a\u
↪type of '
'artificial neural network used in machine learning. It processes input\u
↪data '
'sequentially, allowing it to maintain internal state over time and use\u
↪this '
'context when processing new data points. This makes SegRNNs\u
↪particularly '
'useful for tasks such as speech recognition, language modeling, and\u
↪time '
'series analysis."\n'
'  ')

```

## 5.5.2 Corrective RAG

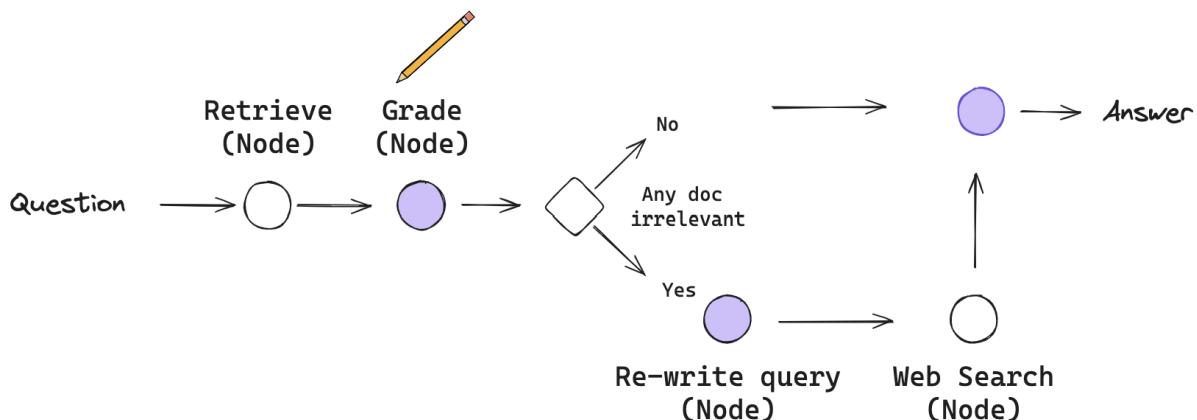


Fig. 7: Corrective-RAG langgraph diagram (source Langgraph c-rag)

- Load Models

```

from langchain_ollama import OllamaEmbeddings
from langchain_ollama.llms import OllamaLLM

```

(continues on next page)

(continued from previous page)

```
# embedding model
embedding = OllamaEmbeddings(model="bge-m3")

# LLM
llm = OllamaLLM(temperature=0.0, model='mistral', format='json')
```

## Warning

You need to specify `format='json'` when Initializing `OllamaLLM`. otherwise you will get error:

```
from langchain.llms import OllamaLLM
template = """Question: {question}
Answer: Let's think step by step.
"""
prompt = ChatPromptTemplate.from_template(template)
llm = OllamaLLM(model="mistral", format='json')
response = llm.invoke("Explain the concept of artificial intelligence in simple terms.")
print(response)

{
    "role": "assistant",
    "model": "text-davinci-003",
    "pid": "123456789abcdef",
    "object": {
        "object": "answer"
    },
    "content": "Artificial Intelligence (AI) is a broad term that refers to the ability of a machine to perform tasks that normally require human intelligence, such as visual perception, language understanding, and decision-making. AI systems are designed to learn from experience and improve their performance over time through machine learning algorithms. They can process large amounts of data and identify patterns to make predictions or take actions based on those patterns. AI has applications in various fields, including robotics, computer vision, natural language processing, and automation.
```

**Validation Error**

```
from langchain.llms import OllamaLLM
template = """Question: {question}
Answer: Let's think step by step.
"""

prompt = ChatPromptTemplate.from_template(template)
llm = OllamaLLM(model="mistral", format='json')

# # Generate a response
# response = llm.invoke("Explain the concept of artificial intelligence in simple terms.")
# print(response)
# chain = prompt | model

# chain.invoke("question": "What is MoE in AI?")

llm.invoke("Come up with 10 names for a song about parrots")
```

Use code with caution

Explanation of Changes:

- Added `format='json'`: The `OllamaLLM` class has a `format` parameter that defaults to `None`. By explicitly setting `format='json'`, you ensure that the response is returned in JSON format.

Enter a prompt here

Responses may display inaccurate or offensive information that doesn't represent Google's views. [Learn more](#)

- Create Index

```
from langchain.text_splitter import RecursiveCharacterTextSplitter
from langchain_community.document_loaders import WebBaseLoader
from langchain_community.vectorstores import Chroma
from langchain_ollama import OllamaEmbeddings # Import
→ OllamaEmbeddings instead

urls = [
    "https://lilianweng.github.io/posts/2023-06-23-agent/",
    "https://lilianweng.github.io/posts/2023-03-15-prompt-engineering/",
```

(continues on next page)

(continued from previous page)

```

        "https://lilianweng.github.io/posts/2023-10-25-adv-attack-llm/",
    ]

docs = [WebBaseLoader(url).load() for url in urls]
docs_list = [item for sublist in docs for item in sublist]

text_splitter = RecursiveCharacterTextSplitter.from_tiktoken_encoder(
    chunk_size=250, chunk_overlap=0
)
doc_splits = text_splitter.split_documents(docs_list)

# Add to vectorDB
vectorstore = Chroma.from_documents(
    documents=doc_splits,
    collection_name="rag-chroma",
    embedding=OllamaEmbeddings(model="bge-m3"),
)
retriever = vectorstore.as_retriever()

```

- Retrieval Grader

```

### Retrieval Grader

from langchain_ollama.llms import OllamaLLM
from langchain.prompts import PromptTemplate
from langchain_community.chat_models import ChatOllama
from langchain_core.output_parsers import JsonOutputParser

from langchain_core.pydantic_v1 import BaseModel, Field
from langchain.output_parsers import PydanticOutputParser

# Data model
class GradeDocuments(BaseModel):
    """Binary score for relevance check on retrieved documents."""

    score: str = Field( # Changed field name to 'score'
        description="Documents are relevant to the question, 'yes' or 'no'")

parser = PydanticOutputParser(pydantic_object=GradeDocuments)

prompt = PromptTemplate(
    template="""You are a grader assessing relevance of a retrieved
document to a user question. \n
Here is the retrieved document: \n\n {document} \n\n
Here is the user question: {question} \n
If the document contains keywords related to the user question,

```

(continues on next page)

(continued from previous page)

```

grade it as relevant. \n
It does not need to be a stringent test. The goal is to filter out
erroneous retrievals. \n
Give a binary score 'yes' or 'no' score to indicate whether the document
is relevant to the question. \n
Provide the binary score as a JSON with a single key 'score' and no
premable or explanation."",
input_variables=["question", "document"],
partial_variables={"format_instructions": parser.get_format_
instructions()}
)

retrieval_grader = prompt | llm | parser
question = "agent memory"
docs = retriever.invoke(question)
doc_txt = docs[1].page_content
retrieval_grader.invoke({"question": question, "document": doc_txt})

```

Output:

```
GradeDocuments(score='yes')
```

### Warning

The output from LangChain Official tutorials ([Langgraph c-rag](#)) is `{'score': 1}`. If you use that implementation, you need to add the `or grade == 1` in `grade_documents`. Otherwise, it will always use web search.

- Generate

```

### Generate

from langchain_core.output_parsers import StrOutputParser

# Prompt
prompt = PromptTemplate(
    template="""You are an assistant for question-answering tasks.

    Use the following documents to answer the question.

    If you don't know the answer, just say that you don't know.

    Use three sentences maximum and keep the answer concise:
    Question: {question}
    Documents: {documents}

```

(continues on next page)

(continued from previous page)

```

Answer:
"""
),
    input_variables=["question", "documents"],
)

# Chain
rag_chain = prompt | llm | StrOutputParser()

# Run
generation = rag_chain.invoke({"documents": docs, "question": question})
print(generation)

```

Output:

```
{
    "short_term_memory": ["They discussed the risks, especially with illicit drugs and bioweapons.", "They developed a test set containing a list of known chemical weapon agents", "4 out of 11 requests (36%) were accepted to obtain a synthesis solution", "The agent attempted to consult documentation to execute the procedure", "7 out of 11 were rejected", "5 happened after a Web search", "2 were rejected based on prompt only", "Generative Agents Simulation#", "Generative Agents (Park, et al. 2023) is super fun experiment where 25 virtual characters"],
    "long_term_memory": ["The design of generative agents combines LLM with memory, planning and reflection mechanisms to enable agents to behave conditioned on past experience", "The memory stream: is a long-term memory module (external database) that records a comprehensive list of agents' experience in natural language"]
}
```

- Router

```

### Router

from langchain.prompts import PromptTemplate
from langchain_community.chat_models import ChatOllama
from langchain_core.output_parsers import JsonOutputParser

prompt = PromptTemplate(
    template="""You are an expert at routing a user question to a vectorstore or web search. Use the vectorstore for questions on LLM agents, prompt engineering, prompting, and adversarial attacks. You can also use words that are similar to those, no need to have exactly those words. Otherwise, use web-search.

```

(continues on next page)

(continued from previous page)

```

Give a binary choice 'web_search' or 'vectorstore' based on the ↵
question.

Return the a JSON with a single key 'datasource' and
no preamble or explanation.

Examples:
Question: When will the Euro of Football take place?
Answer: {"datasource": "web_search"}

Question: What are the types of agent memory?
Answer: {"datasource": "vectorstore"}

Question: What are the basic approaches for prompt engineering?
Answer: {"datasource": "vectorstore"}

Question: What is prompt engineering?
Answer: {"datasource": "vectorstore"}

Question to route:
{question}""",
input_variables=["question"],
)

question_router = prompt | llm | JsonOutputParser()

print(question_router.invoke({"question": "When will the Euro of Football \
take place?"}))
print(question_router.invoke({"question": "What are the types of agent \
memory?"})) ### Index

print(question_router.invoke({"question": "What are the basic approaches \
for \
prompt engineering?"})) ### Index

```

Output:

```

{'datasource': 'web_search'}
{'datasource': 'vectorstore'}
{'datasource': 'vectorstore'}

```

- Hallucination Grader

```

### Hallucination Grader

# Data model

```

(continues on next page)

(continued from previous page)

```

class GradeHallucinations(BaseModel):
    """Binary score for relevance check on retrieved documents."""

    score: str = Field( # Changed field name to 'score'
        description="Documents are relevant to the question, 'yes' or 'no'")

parser = PydanticOutputParser(pydantic_object=GradeHallucinations)

# Prompt
prompt = PromptTemplate(
    template="""You are a grader assessing whether an answer is grounded in
    ↵/
        supported by a set of facts. \n
        Here are the facts:
        \n ----- \n
        {documents}
        \n ----- \n
        Here is the answer: {generation}
        Give a binary score 'yes' or 'no' score to indicate whether
        the answer is grounded in / supported by a set of facts. \n
        Provide the binary score as a JSON with a single key 'score'
        and no preamble or explanation."""",
    input_variables=["generation", "documents"],
    partial_variables={"format_instructions": parser.get_format_
    ↵instructions()}
)

hallucination_grader = prompt | llm | parser
hallucination_grader.invoke({"documents": docs, "generation": generation})

```

Output:

```
GradeHallucinations(score='yes')
```

- Answer Grader

```

### Answer Grader

# Data model
class GradeAnswer(BaseModel):
    """Binary score for relevance check on retrieved documents."""

    score: str = Field( # Changed field name to 'score'
        description="Documents are relevant to the question, 'yes' or 'no'")

parser = PydanticOutputParser(pydantic_object=GradeAnswer)

```

(continues on next page)

(continued from previous page)

```
# Prompt
prompt = PromptTemplate(
    template="""You are a grader assessing whether an answer is useful to
    resolve a question. \n
    Here is the answer:
    \n ----- \n
    {generation}
    \n ----- \n
    Here is the question: {question}
    Give a binary score 'yes' or 'no' to indicate whether
    the answer is useful to resolve a question. \n
    Provide the binary score as a JSON with a single key
    'score' and no preamble or explanation."""",
    input_variables=["generation", "question"],
    partial_variables={"format_instructions": parser.get_format_
    ↪instructions()}
)
answer_grader = prompt | llm | parser
answer_grader.invoke({"question": question, "generation": generation})
```

Output:

```
GradeAnswer(score='yes')
```

- Question Re-writer

```
### Question Re-writer

# Prompt
re_write_prompt = PromptTemplate(
    template="""You a question re-writer that converts an input question
    to a better version that is optimized \n for vectorstore
    retrieval. Look at the input and try to reason about the
    underlying semantic intent / meaning. \n
    Here is the initial question: \n\n {question}.
    Formulate an improved question.\n """,
    input_variables=["generation", "question"],
)
question_rewriter = re_write_prompt | llm | StrOutputParser()
question_rewriter.invoke({"question": question})
```

Output:

```
{ "question": "What is the function or purpose of an agent's memory in a given context?" }
```

- Web Search Tool (Google)

```
from langchain_google_community import GoogleSearchAPIWrapper, GoogleSearchResults

from google.colab import userdata
api_key = userdata.get('GOOGLE_API_KEY')
cx = userdata.get('GOOGLE_CSE_ID')
# Replace with your actual API key and CX ID

# Create an instance of the GoogleSearchAPIWrapper
google_search_wrapper = GoogleSearchAPIWrapper(google_api_key=api_key, google_cse_id=cx)

# Pass the api_wrapper to GoogleSearchResults
web_search_tool = GoogleSearchResults(api_wrapper=google_search_wrapper, k=3)
# web_results = web_search_tool.invoke({"query": question})
```

- Create the Graph

```
from typing import List
from typing_extensions import TypedDict
from IPython.display import Image, display
from langchain.schema import Document
from langgraph.graph import START, END, StateGraph


class GraphState(TypedDict):
    """
    Represents the state of our graph.

    Attributes:
        question: question
        generation: LLM generation
        search: whether to add search
        documents: list of documents
    """

    question: str
    generation: str
    search: str
    documents: List[str]
    steps: List[str]
```

(continues on next page)

(continued from previous page)

```
def retrieve(state):
    """
    Retrieve documents

    Args:
        state (dict): The current graph state

    Returns:
        state (dict): New key added to state, documents, that contains ↵
        ↪retrieved documents
    """

    question = state["question"]
    documents = retriever.invoke(question)
    steps = state["steps"]
    steps.append("retrieve_documents")
    return {"documents": documents, "question": question, "steps": steps}

def generate(state):
    """
    Generate answer

    Args:
        state (dict): The current graph state

    Returns:
        state (dict): New key added to state, generation, that contains LLM ↵
        ↪generation
    """

    question = state["question"]
    documents = state["documents"]
    generation = rag_chain.invoke({"documents": documents, "question": ↵
        ↪question})
    steps = state["steps"]
    steps.append("generate_answer")
    return {
        "documents": documents,
        "question": question,
        "generation": generation,
        "steps": steps,
    }
```

(continues on next page)

(continued from previous page)

```

def grade_documents(state):
    """
    Determines whether the retrieved documents are relevant to the question.

    Args:
        state (dict): The current graph state

    Returns:
        state (dict): Updates documents key with only filtered relevant_
        ↪documents
    """

    question = state["question"]
    documents = state["documents"]
    steps = state["steps"]
    steps.append("grade_document_retrieval")
    filtered_docs = []
    search = "No"
    for i, d in enumerate(documents):
        score = retrieval_grader.invoke(
            {"question": question, "documents": d.page_content}
        )
        grade = score["score"]

        if grade == "yes" or grade == 1:
            print(f"---GRADE: DOCUMENT {i} RELEVANT---")
            filtered_docs.append(d)
        else:
            print(f"---GRADE: DOCUMENT {i} ISN'T RELEVANT---")
            search = "Yes"
            continue
    return {
        "documents": filtered_docs,
        "question": question,
        "search": search,
        "steps": steps,
    }

def web_search(state):
    """
    Web search based on the re-phrased question.

    Args:
        state (dict): The current graph state
    """

```

(continues on next page)

(continued from previous page)

```

Returns:
    state (dict): Updates documents key with appended web results
"""

question = state["question"]
documents = state.get("documents", [])
steps = state["steps"]
steps.append("web_search")
web_results = web_search_tool.invoke({"query": question})
documents.extend(
    [
        Document(page_content=d["snippet"], metadata={"url": d["link"]})
        for d in eval(web_results)
    ]
)
return {"documents": documents, "question": question, "steps": steps}

def decide_to_generate(state):
"""
Determines whether to generate an answer, or re-generate a question.

Args:
    state (dict): The current graph state

Returns:
    str: Binary decision for next node to call
"""

search = state["search"]
if search == "Yes":
    return "search"
else:
    return "generate"

```

- Compile Graph

```

from langgraph.graph import START, END, StateGraph

workflow = StateGraph(GraphState)

# Define the nodes
workflow.add_node("retrieve", retrieve) # retrieve
workflow.add_node("grade_documents", grade_documents) # grade documents
workflow.add_node("generate", generate) # generatae
workflow.add_node("web_search", web_search) # web search

```

(continues on next page)

(continued from previous page)

```
# Build graph
workflow.add_edge(START, "retrieve")
workflow.add_edge("retrieve", "grade_documents")
workflow.add_conditional_edges(
    "grade_documents",
    decide_to_generate,
    {
        "search": "web_search",
        "generate": "generate",
    },
)
workflow.add_edge("web_search", "generate")
workflow.add_edge("generate", END)

# Compile
app = workflow.compile()
```

- Graph visualization

```
from IPython.display import Image, display

try:
    display(Image(app.get_graph(xray=True).draw_mermaid_png()))
except:
    pass
```

Ouput

- Test

- Relevant retrieval

```
import uuid

config = {"configurable": {"thread_id": str(uuid.uuid4())}}
example = {"input": "What are the basic approaches for \
            prompt engineering?"}

state_dict = app.invoke({"question": example["input"], "steps": []}, ↴
                      config)
state_dict
```

Ouput:

```
---GRADE: DOCUMENT 0 RELEVANT---
---GRADE: DOCUMENT 1 RELEVANT---
```

(continues on next page)

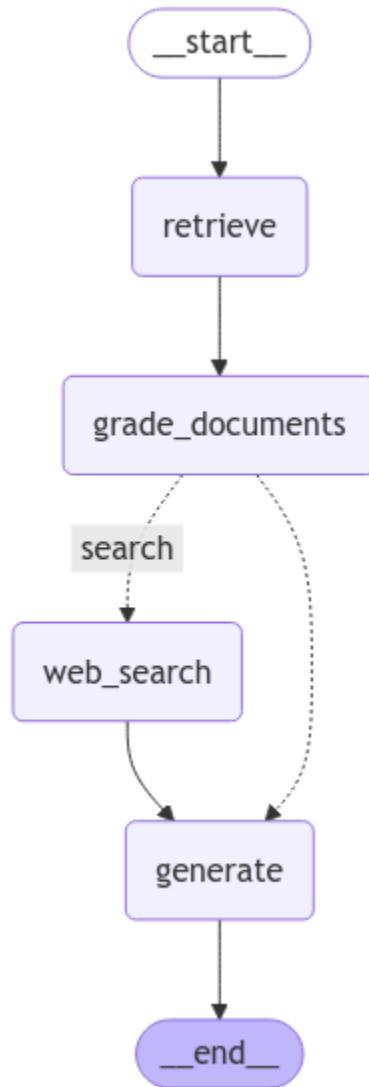


Fig. 8: Corrective-RAG Graph

(continued from previous page)

```

---GRADE: DOCUMENT 2 RELEVANT---
---GRADE: DOCUMENT 3 RELEVANT---
{'question': 'What are the basic approaches for
→prompt engineering?',
'generation': '{\n      "Basic Prompting" : "A basic approach for
→prompt engineering is to provide clear and concise instructions to
→the language model, guiding it towards the desired output."\n    },
'search': 'No',
'documents': [Document(metadata={'description': 'Prompt Engineering,
→also known as In-Context Prompting, refers to methods for how to
→communicate with LLM to steer its behavior for desired outcomes,
→without updating the model weights. It is an empirical science and
→the effect of prompt engineering methods can vary a lot among models,
→thus requiring heavy experimentation and heuristics.\nThis post only
→focuses on prompt engineering for autoregressive language models, so
→nothing with Cloze tests, image generation or multimodality models.
→At its core, the goal of prompt engineering is about alignment and
→model steerability. Check my previous post on controllable text
→generation.', 'language': 'en', 'source': 'https://lilianweng.github.
→io/posts/2023-03-15-prompt-engineering/', 'title': "Prompt
→Engineering | Lil'Log"}, page_content='Prompt Engineering, also known
→as In-Context Prompting, refers to methods for how to communicate
→with LLM to steer its behavior for desired outcomes without updating
→the model weights. It is an empirical science and the effect of
→prompt engineering methods can vary a lot among models, thus
→requiring heavy experimentation and heuristics.\nThis post only
→focuses on prompt engineering for autoregressive language models, so
→nothing with Cloze tests, image generation or multimodality models.
→At its core, the goal of prompt engineering is about alignment and
→model steerability. Check my previous post on controllable text
→generation.\n[My personal spicy take] In my opinion, some prompt
→engineering papers are not worthy 8 pages long, since those tricks
→can be explained in one or a few sentences and the rest is all about
→benchmarking. An easy-to-use and shared benchmark infrastructure
→should be more beneficial to the community. Iterative prompting or
→external tool use would not be trivial to set up. Also non-trivial to
→align the whole research community to adopt it.\nBasic Prompting#'),
  Document(metadata={'description': 'Prompt Engineering, also known as
→In-Context Prompting, refers to methods for how to communicate with
→LLM to steer its behavior for desired outcomes without updating the
→model weights. It is an empirical science and the effect of prompt
→engineering methods can vary a lot among models, thus requiring heavy
→experimentation and heuristics.\nThis post only focuses on prompt
→engineering for autoregressive language models, so nothing with Cloze
→tests, image generation or multimodality models. At its core, the
→goal of prompt engineering is about alignment and model steerability.'}
}

```

(continues on next page)

(continued from previous page)

```
→Check my previous post on controllable text generation.', 'language':  
→'en', 'source': 'https://lilianweng.github.io/posts/2023-03-15-prompt-  
→engineering/', 'title': "Prompt Engineering | Lil'Log"}, page_content=  
→'Prompt Engineering, also known as In-Context Prompting, refers to  
→methods for how to communicate with LLM to steer its behavior for  
→desired outcomes without updating the model weights. It is an  
→empirical science and the effect of prompt engineering methods can  
→vary a lot among models, thus requiring heavy experimentation and  
→heuristics.\nThis post only focuses on prompt engineering for  
→autoregressive language models, so nothing with Cloze tests, image  
→generation or multimodality models. At its core, the goal of prompt  
→engineering is about alignment and model steerability. Check my  
→previous post on controllable text generation.\n[My personal spicy  
→take] In my opinion, some prompt engineering papers are not worthy 8  
→pages long, since those tricks can be explained in one or a few  
→sentences and the rest is all about benchmarking. An easy-to-use and  
→shared benchmark infrastructure should be more beneficial to the  
→community. Iterative prompting or external tool use would not be  
→trivial to set up. Also non-trivial to align the whole research  
→community to adopt it.\nBasic Prompting#'),  
Document(metadata={'description': 'Prompt Engineering, also known as  
→In-Context Prompting, refers to methods for how to communicate with  
→LLM to steer its behavior for desired outcomes without updating the  
→model weights. It is an empirical science and the effect of prompt  
→engineering methods can vary a lot among models, thus requiring heavy  
→experimentation and heuristics.\nThis post only focuses on prompt  
→engineering for autoregressive language models, so nothing with Cloze  
→tests, image generation or multimodality models. At its core, the  
→goal of prompt engineering is about alignment and model steerability.  
→Check my previous post on controllable text generation.', 'language':  
→'en', 'source': 'https://lilianweng.github.io/posts/2023-03-15-prompt-  
→engineering/', 'title': "Prompt Engineering | Lil'Log"}, page_content=  
→'Prompt Engineering, also known as In-Context Prompting, refers to  
→methods for how to communicate with LLM to steer its behavior for  
→desired outcomes without updating the model weights. It is an  
→empirical science and the effect of prompt engineering methods can  
→vary a lot among models, thus requiring heavy experimentation and  
→heuristics.\nThis post only focuses on prompt engineering for  
→autoregressive language models, so nothing with Cloze tests, image  
→generation or multimodality models. At its core, the goal of prompt  
→engineering is about alignment and model steerability. Check my  
→previous post on controllable text generation.\n[My personal spicy  
→take] In my opinion, some prompt engineering papers are not worthy 8  
→pages long, since those tricks can be explained in one or a few  
→sentences and the rest is all about benchmarking. An easy-to-use and  
→shared benchmark infrastructure should be more beneficial to the
```

(continues on next page)

(continued from previous page)

- Irrelevant retrieval

```
example = {"input": "What is the capital of China?"}
config = {"configurable": {"thread_id": str(uuid.uuid4())}}
state_dict = app.invoke({"question": example["input"], "steps": [], "config": config})
state_dict
```

## Output:

---GRADE: DOCUMENT 0 ISN'T RELEVANT---

---GRADE: DOCUMENT 1 ISN'T RELEVANT---

---GRADE: DOCUMENT 2 ISN'T RELEVANT---

---GRADE: DOCUMENT 3 ISN'T RELEVANT---

(continues on next page)

(continued from previous page)

```
{
  'question': 'What is the capital of China?',
  'generation': '{\n      "answer": "Beijing is the capital of China."\\n    }',
  'search': 'Yes',
  'documents': [Document(metadata={'url': 'https://clintonwhitehouse3.archives.gov/WH/New/China/beijing.html'}, page_content='The modern day capital of China is Beijing (literally "Northern Capital"), which first served as China\'s capital city in 1261, when the Mongol ruler Kublai\xA0...'),
    Document(metadata={'url': 'https://en.wikipedia.org/wiki/Beijing'}, page_content="Beijing, previously romanized as Peking, is the capital city of China. With more than 22 million residents, it is the world's most populous national capital\xA0..."),
    Document(metadata={'url': 'https://pubmed.ncbi.nlm.nih.gov/38294063/'}, page_content='Supercritical and homogenous transmission of monkeypox in the capital of China. J Med Virol. 2024 Feb;96(2):e29442. doi: 10.1002/jmv.29442. Authors. Yunjun\xA0...'),
    Document(metadata={'url': 'https://www.sciencedirect.com/science/article/pii/S0304387820301358'}, page_content='This paper investigates the impacts of fires on cognitive performance. We find that a one-standard-deviation increase in the difference between upwind and\xA0...')],
  'steps': ['retrieve_documents',
    'grade_document_retrieval',
    'web_search',
    'generate_answer']
}
```

### 5.5.3 Adaptive RAG

- Load Models

```
from langchain_ollama import OllamaEmbeddings
from langchain_ollama.llms import OllamaLLM

# embedding model
embedding = OllamaEmbeddings(model="bge-m3")

# LLM
llm = OllamaLLM(temperature=0.0, model='mistral', format='json')
```

#### Warning

You need to specify `format='json'` when Initializing `OllamaLLM`. otherwise you will get error:

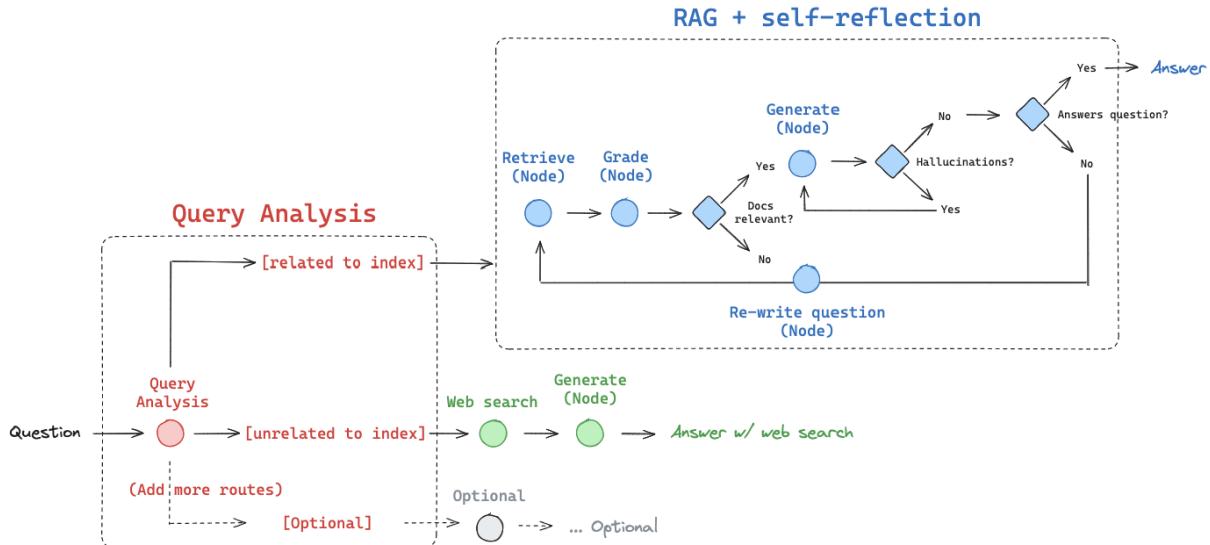


Fig. 9: Adaptive-RAG langgraph diagram (source Langgraph A-rag)

A screenshot of a Jupyter Notebook cell titled "Run-ollama-in-colab.ipynb". The cell contains Python code for interacting with an OllamaLLM. The code includes importing modules, downloading packages, and defining a ChatPromptTemplate. It then creates an OllamaLLM instance and uses it to invoke a response. The output pane shows the execution progress and a validation error message:

```

[4] Downloading https://github.com/lm-sys/FastChat/releases/download/v0.27.2/https-0.27.2-py3-none-any.whl (76 kB)
  76.4/76.4 kB 8.5 MB/s eta 0:00:00
[4] Installing collected packages: httpsx, ollama, langchain-ollama
  Attempting uninstall: httpsx
    Found existing installation: httpsx 0.28.0
    Uninstalling httpsx-0.28.0:
      Successfully uninstalled httpsx-0.28.0
  Successfully installed httpsx-0.27.2 langchain-ollama-0.2.1 ollama-0.4.3
[4] 
from langchain_core.prompts import ChatPromptTemplate
from langchain_ollama.llms import OllamaLLM

template = """Question: {question}

Answer: Let's think step by step.

"""

prompt = ChatPromptTemplate.from_template(template)

llm = OllamaLLM(model="mistral", format='json')

# Generate a response
response = llm.invoke("Explain the concept of artificial intelligence in simple terms.")

print(response)
  
```

Resources      Validation Error

```

from langchain_ollama.llms import OllamaLLM

template = """Question: {question}

Answer: Let's think step by step.

"""

prompt = ChatPromptTemplate.from_template(template)

# Specify format='json' when initializing OllamaLLM
llm = OllamaLLM(model="mistral", format='json')

# # Generate a response
# response = llm.invoke("Explain the concept of artificial intelligence in simple terms.")
# print(response)
# chain = prompt | model

# chain.invoke({"question": "What is MoE in AI?"})

llm.invoke("Come up with 10 names for a song about parrots")
  
```

Use code with caution

Explanation of Changes:

- Added `format='json'`: The OllamaLLM class has a `format` parameter that defaults to `None`. By explicitly setting `format='json'`, you ensure compatibility with other parts of the system.

Enter a prompt here

0 / 2000

Responses may display inaccurate or offensive information that doesn't represent Google's views. [Learn more](#)

- Create Index

```

from langchain.text_splitter import RecursiveCharacterTextSplitter
from langchain_community.document_loaders import WebBaseLoader
from langchain_community.vectorstores import Chroma
from langchain_ollama import OllamaEmbeddings # Import
    ↪OllamaEmbeddings instead

urls = [
    "https://lilianweng.github.io/posts/2023-06-23-agent/",
    "https://lilianweng.github.io/posts/2023-03-15-prompt-engineering/",
    "https://lilianweng.github.io/posts/2023-10-25-adv-attack-lm/",
]

docs = [WebBaseLoader(url).load() for url in urls]
docs_list = [item for sublist in docs for item in sublist]

text_splitter = RecursiveCharacterTextSplitter.from_tiktoken_encoder(
    chunk_size=250, chunk_overlap=0
)
doc_splits = text_splitter.split_documents(docs_list)

# Add to vectorDB
vectorstore = Chroma.from_documents(
    documents=doc_splits,
    collection_name="rag-chroma",
    embedding=OllamaEmbeddings(model="bge-m3"),
)
retriever = vectorstore.as_retriever(k=4)

```

- Router

```

### Router

from typing import Literal

from langchain_ollama.llms import OllamaLLM
from langchain.prompts import PromptTemplate
from langchain_community.chat_models import ChatOllama
from langchain_core.output_parsers import JsonOutputParser

from langchain.output_parsers import PydanticOutputParser
from pydantic import BaseModel, Field

# Data model
class RouteQuery(BaseModel):

```

(continues on next page)

(continued from previous page)

```

"""Route a user query to the most relevant datasource."""

datasource: Literal["vectorstore", "web_search"] = Field(
    ...,
    description="Given a user question choose to route it \
        to web search or a vectorstore.",
)

# LLM with function call
structured_llm_router = PydanticOutputParser(pydantic_object=RouteQuery)

# Prompt
route_prompt = PromptTemplate(
    template="""You are an expert at routing a user question to a
        vectorstore or web search. The vectorstore contains
        documents related to agents, prompt engineering,
        and adversarial attacks.
        Use the vectorstore for questions on these topics.
        Otherwise, use web-search. \n
        Here is the user question: {question}. \n
        Respond with a JSON object containing only the key
    ↵'datasource'
    ↵        and its value, which should be either 'vectorstore' or
    ↵'web_search'.

    Example:
    {"datasource": "vectorstore"}
    """,
    input_variables=["question"],
    partial_variables={"format_instructions": \
        structured_llm_router.get_format_instructions()}
)

question_router = route_prompt | llm | structured_llm_router

print(question_router.invoke({"question": \
    "Who will the Bears draft first in the NFL\
    ↵draft?"}))

print(question_router.invoke({"question": \
    "What are the types of agent memory?"}))

```

Output:

```
datasource='web_search'
```

(continues on next page)

(continued from previous page)

```
datasource='vectorstore'
```

### Note

We introduced above new implementation with `pydantic` Data Model for the output parser. But, you can still use the similar one we implemented in *Corrective RAG*.

```
### OR
### Router
```

```
from langchain.prompts import PromptTemplate
from langchain_community.chat_models import ChatOllama
from langchain_core.output_parsers import JsonOutputParser

prompt = PromptTemplate(
    template="""You are an expert at routing a
    user question to a vectorstore or web search. Use the
    ↵vectorstore for
    questions on LLM agents, prompt engineering, prompting, and
    ↵adversarial
    attacks. You can also use words that are similar to those,
    no need to have exactly those words. Otherwise, use web-search.

    Give a binary choice 'web_search' or 'vectorstore' based on the
    ↵question.
    Return the a JSON with a single key 'datasource' and
    no preamble or explanation.

    Examples:
    Question: When will the Euro of Football take place?
    Answer: {"datasource": "web_search"}

    Question: What are the types of agent memory?
    Answer: {"datasource": "vectorstore"}

    Question: What are the basic approaches for prompt engineering?
    Answer: {"datasource": "vectorstore"}

    Question: What is prompt engineering?
    Answer: {"datasource": "vectorstore"}

    Question to route:
    {question}""",
    input_variables=["question"],
)

```

```

question_router = prompt | llm | JsonOutputParser()

print(question_router.invoke({"question": "When will the Euro of Football \
                                take place?"}))
print(question_router.invoke({"question": "What are the types of agent \
                                memory?"})) ### Index

print(question_router.invoke({"question": "What are the basic approaches for \
                                prompt engineering?"})) ##
# Index

```

- Retrieval Grader

```

### Retrieval Grader

from langchain_llms import OllamaLLM
from langchain_prompts import PromptTemplate
from langchain_community.chat_models import ChatOllama
from langchain_core.output_parsers import JsonOutputParser

# Import BaseModel and Field from langchain_core.pydantic_v1
# from langchain_core.pydantic_v1 import BaseModel, Field
from pydantic import BaseModel, Field
from langchain.output_parsers import PydanticOutputParser

# Data model
class GradeDocuments(BaseModel):
    """Binary score for relevance check on retrieved documents."""

    score: str = Field( # Changed field name to 'score'
        description="Documents are relevant to the question, 'yes' or 'no'")

parser = PydanticOutputParser(pydantic_object=GradeDocuments)

prompt = PromptTemplate(
    template="""You are a grader assessing relevance of a retrieved
document to a user question. \n
Here is the retrieved document: \n\n {document} \n\n
Here is the user question: {question} \n
If the document contains keywords related to the user question,
grade it as relevant. \n
"""
)

```

(continues on next page)

(continued from previous page)

```

It does not need to be a stringent test. The goal is to filter out
erroneous retrievals. \n
Give a binary score 'yes' or 'no' score to indicate whether the document
is relevant to the question. \n
Provide the binary score as a JSON with a single key 'score' and no
preamble or explanation."",
input_variables=["question", "document"],
partial_variables={"format_instructions": parser.get_format_
instructions()}
)

retrieval_grader = prompt | llm | parser
question = "agent memory"
docs = retriever.invoke(question)
doc_txt = docs[1].page_content
retrieval_grader.invoke({"question": question, "document": doc_txt})

```

- Generate

```

### Generate

from langchain_core.output_parsers import StrOutputParser

# Prompt
prompt = PromptTemplate(
    template="""You are an assistant for question-answering tasks.

    Use the following documents to answer the question.

    If you don't know the answer, just say that you don't know.

    Use three sentences maximum and keep the answer concise:
    Question: {question}
    Documents: {documents}
    Answer:
    """,
    input_variables=["question", "documents"],
)

# Chain
rag_chain = prompt | llm | StrOutputParser()

# Run
generation = rag_chain.invoke({"documents": docs, "question": question})
print(generation)

```

Ouput:

```
{
  "answer": [
    {
      "role": "assistant",
      "content": "In the context of an LLM (Large Language Model),\n→powered autonomous agent, memory can be divided into three types: Sensory\n→Memory, Short-term Memory, and Long-term Memory. \\n\\nSensory Memory is\n→learning embedding representations for raw inputs, including text, image\n→or other modalities. It's the earliest stage of memory, providing the\n→ability to retain impressions of sensory information after the original\n→stimuli have ended. Sensory memory typically only lasts for up to a few\n→seconds. Subcategories include iconic memory (visual), echoic memory\n→(auditory), and haptic memory (touch). \\n\\nShort-term memory, also known\n→as working memory, is short and finite, as it is restricted by the finite\n→context window length of Transformer. It stores and manipulates the\n→information that the agent currently needs to solve a task. \\n\\nLong-term\n→memory is the external vector store that the agent can attend to at query\n→time, accessible via fast retrieval. This provides the agent with the\n→capability to retain and recall (infinite) information over extended\n→periods, often by leveraging an external vector store and fast retrieval\n→methods such as Maximum Inner Product Search (MIPS). To optimize the\n→retrieval speed, the common choice is the approximate nearest neighbors\n→(ANN) algorithm to return approximately top k nearest neighbors, trading\n→off a little accuracy lost for a huge speedup. A couple common choices of\n→ANN algorithms for fast MIPS are HNSW (Hierarchical Navigable Small\n→World) and Annoy (Approximate Nearest Neighbors Oh Yeah)."
    }
  ]
}
```

- Hallucination Grader

```
### Hallucination Grader

# Data model
class GradeHallucinations(BaseModel):
    """Binary score for hallucination present in generation answer."""

    score: str = Field(
        description="Answer is grounded in the facts, 'yes' or 'no'"
    )

parser = PydanticOutputParser(pydantic_object=GradeHallucinations)

# Prompt
prompt = PromptTemplate(
```

(continues on next page)

(continued from previous page)

```

template="""You are a grader assessing whether an answer is grounded
in / supported by a set of facts. \n
Here are the facts:
\n ----- \n
{documents}
\n ----- \n
Here is the answer: {generation}
Give a binary score 'yes' or 'no' score to indicate whether the answer
is grounded in / supported by a set of facts. \n
Provide the binary score as a JSON with a single key 'score' and no
preamble or explanation.""",
input_variables=["generation", "documents"],
)

hallucination_grader = prompt | llm | parser
hallucination_grader.invoke({"documents": docs, "generation": generation})

```

- Answer Grader

```

### Answer Grader

# Data model
class GradeAnswer(BaseModel):
    """Binary score to assess answer addresses question."""

    score: str = Field(
        description="Answer addresses the question, 'yes' or 'no'"
    )

# parser
parser = PydanticOutputParser(pydantic_object=GradeAnswer)

# Prompt
prompt = PromptTemplate(
    template="""You are a grader assessing whether an answer is useful
    to
    resolve a question. \n
    Here is the answer:
\n ----- \n
{generation}
\n ----- \n
    Here is the question: {question}
    Give a binary score 'yes' or 'no' to indicate whether the answer is
    useful to resolve a question. \n
    Provide the binary score as a JSON with a single key 'score' and no
    preamble or explanation.""",

```

(continues on next page)

(continued from previous page)

```

        input_variables=["generation", "question"],
    )

answer_grader = prompt | llm | parser
answer_grader.invoke({"question": question, "generation": generation})

```

- Question Re-writer

```

### Question Re-writer

# Prompt
re_write_prompt = PromptTemplate(
    template="""You a question re-writer that converts an input question to
    a better version that is optimized \n
    for vectorstore retrieval. Look at the initial and formulate an improved
    question. \n
    Here is the initial question: \n\n {question}. Improved question
    with no preamble: \n """,
    input_variables=["generation", "question"],
)

question_rewriter = re_write_prompt | llm | StrOutputParser()
question_rewriter.invoke({"question": question})

```

- Google Web Search

```

## Google Web Search
from langchain_google_community import GoogleSearchAPIWrapper, ↴
    GoogleSearchResults

from google.colab import userdata
api_key = userdata.get('GOOGLE_API_KEY')
cx = userdata.get('GOOGLE_CSE_ID')
# Replace with your actual API key and CX ID

# Create an instance of the GoogleSearchAPIWrapper
google_search_wrapper = GoogleSearchAPIWrapper(google_api_key=api_key, \
                                                google_cse_id=cx)

# Pass the api_wrapper to GoogleSearchResults
web_search_tool = GoogleSearchResults(api_wrapper=google_search_wrapper, ↴
                                         k=3)

```

- Create the Graph

```
from typing import List
```

(continues on next page)

(continued from previous page)

```
from typing_extensions import TypedDict
from IPython.display import Image, display
from langchain.schema import Document
from langgraph.graph import START, END, StateGraph

class GraphState(TypedDict):
    """
    Represents the state of our graph.

    Attributes:
        question: question
        generation: LLM generation
        documents: list of documents
    """

    question: str
    generation: str
    documents: List[str]

def retrieve(state):
    """
    Retrieve documents

    Args:
        state (dict): The current graph state

    Returns:
        state (dict): New key added to state,
                      documents, that contains retrieved documents
    """
    print("---RETRIEVE---")
    question = state["question"]

    # Retrieval
    documents = retriever.invoke(question)
    return {"documents": documents, "question": question}

def generate(state):
    """
    Generate answer

    Args:
        state (dict): The current graph state
    
```

(continues on next page)

(continued from previous page)

```

Returns:
    state (dict): New key added to state, generation,
    that contains LLM generation
    """
    print("---GENERATE---")
    question = state["question"]
    documents = state["documents"]

    # RAG generation
    generation = rag_chain.invoke({"documents": documents, \
                                    "question": question})
    return {"documents": documents, \
            "question": question, \
            "generation": generation}

def grade_documents(state):
    """
    Determines whether the retrieved documents are relevant to the question.

    Args:
        state (dict): The current graph state

    Returns:
        state (dict): Updates documents key with only filtered relevant
        ↪documents
        """
        """

    print("---CHECK DOCUMENT RELEVANCE TO QUESTION---")
    question = state["question"]
    documents = state["documents"]

    # Score each doc
    filtered_docs = []
    for d in documents:
        score = retrieval_grader.invoke(
            {"question": question, "document": d.page_content}
        )
        grade = score.score
        if grade == "yes":
            print("---GRADE: DOCUMENT RELEVANT---")
            filtered_docs.append(d)
        else:
            print("---GRADE: DOCUMENT NOT RELEVANT---")
            continue
    return {"documents": filtered_docs, "question": question}

```

(continues on next page)

(continued from previous page)

```

def transform_query(state):
    """
    Transform the query to produce a better question.

    Args:
        state (dict): The current graph state

    Returns:
        state (dict): Updates question key with a re-phrased question
    """

    print("---TRANSFORM QUERY---")
    question = state["question"]
    documents = state["documents"]

    # Re-write question
    better_question = question_rewriter.invoke({"question": question})
    return {"documents": documents, "question": better_question}

def web_search(state):
    """
    Web search based on the re-phrased question.

    Args:
        state (dict): The current graph state

    Returns:
        state (dict): Updates documents key with appended web results
    """

    question = state["question"]
    documents = state.get("documents", [])

    web_results = web_search_tool.invoke({"query": question})
    documents.extend(
        [
            Document(page_content=d["snippet"], metadata={"url": d["link"]})
            for d in eval(web_results)
        ]
    )
    return {"documents": documents, \
             "question": grade_generation_v_documents_and_question}

### Edges ###

```

(continues on next page)

(continued from previous page)

```

def route_question(state):
    """
    Route question to web search or RAG.

    Args:
        state (dict): The current graph state

    Returns:
        str: Next node to call
    """

    print("---ROUTE QUESTION---")
    question = state["question"]
    source = question_router.invoke({"question": question})
    if source.datasource == "web_search":
        print("---ROUTE QUESTION TO WEB SEARCH---")
        return "web_search"
    elif source.datasource == "vectorstore":
        print("---ROUTE QUESTION TO RAG---")
        return "vectorstore"

def decide_to_generate(state):
    """
    Determines whether to generate an answer, or re-generate a question.

    Args:
        state (dict): The current graph state

    Returns:
        str: Binary decision for next node to call
    """

    print("---ASSESS GRADED DOCUMENTS---")
    state["question"]
    filtered_documents = state["documents"]

    if not filtered_documents:
        # All documents have been filtered check_relevance
        # We will re-generate a new query
        print("---DECISION: ALL DOCUMENTS ARE NOT RELEVANT TO QUESTION, \
              TRANSFORM QUERY---")
        return "transform_query"
    else:

```

(continues on next page)

(continued from previous page)

```

# We have relevant documents, so generate answer
print("---DECISION: GENERATE---")
return "generate"

def grade_generation_v_documents_and_question(state):
    """
    Determines whether the generation is grounded in the document
    and answers question.

    Args:
        state (dict): The current graph state

    Returns:
        str: Decision for next node to call
    """

    print("---CHECK HALLUCINATIONS---")
    question = state["question"]
    documents = state["documents"]
    generation = state["generation"]

    score = hallucination_grader.invoke(
        {"documents": documents, "generation": generation}
    )
    grade = score.score

    # Check hallucination
    if grade == "yes":
        print("---DECISION: GENERATION IS GROUNDED IN DOCUMENTS---")
        # Check question-answering
        print("---GRADE GENERATION vs QUESTION---")
        score = answer_grader.invoke({"question": question, \
                                      "generation": generation})
        grade = score.score
        if grade == "yes":
            print("---DECISION: GENERATION ADDRESSES QUESTION---")
            return "useful"
        else:
            print("---DECISION: GENERATION DOES NOT ADDRESS QUESTION---")
            return "not useful"
    else:
        print("---DECISION: GENERATION IS NOT GROUNDED IN DOCUMENTS, RE-TRY-")
        return "not supported"

```

- Compile Graph

```

from langgraph.graph import END, StateGraph, START

workflow = StateGraph(GraphState)

# Define the nodes
workflow.add_node("web_search", web_search) # web search
workflow.add_node("retrieve", retrieve) # retrieve
workflow.add_node("grade_documents", grade_documents) # grade documents
workflow.add_node("generate", generate) # generate
workflow.add_node("transform_query", transform_query) # transform_query

# Build graph
workflow.add_conditional_edges(
    START,
    route_question,
    {
        "web_search": "web_search",
        "vectorstore": "retrieve",
    },
)
workflow.add_edge("web_search", "generate")
workflow.add_edge("retrieve", "grade_documents")
workflow.add_conditional_edges(
    "grade_documents",
    decide_to_generate,
    {
        "transform_query": "transform_query",
        "generate": "generate",
    },
)
workflow.add_edge("transform_query", "retrieve")
workflow.add_conditional_edges(
    "generate",
    grade_generation_v_documents_and_question,
    {
        "not supported": "generate",
        "useful": END,
        "not useful": "transform_query",
    },
)
# Compile
app = workflow.compile()

```

- Graph visualization

```
from IPython.display import Image, display

try:
    display(Image(app.get_graph(xray=True).draw_mermaid_png()))
except:
    pass
```

## Ouput

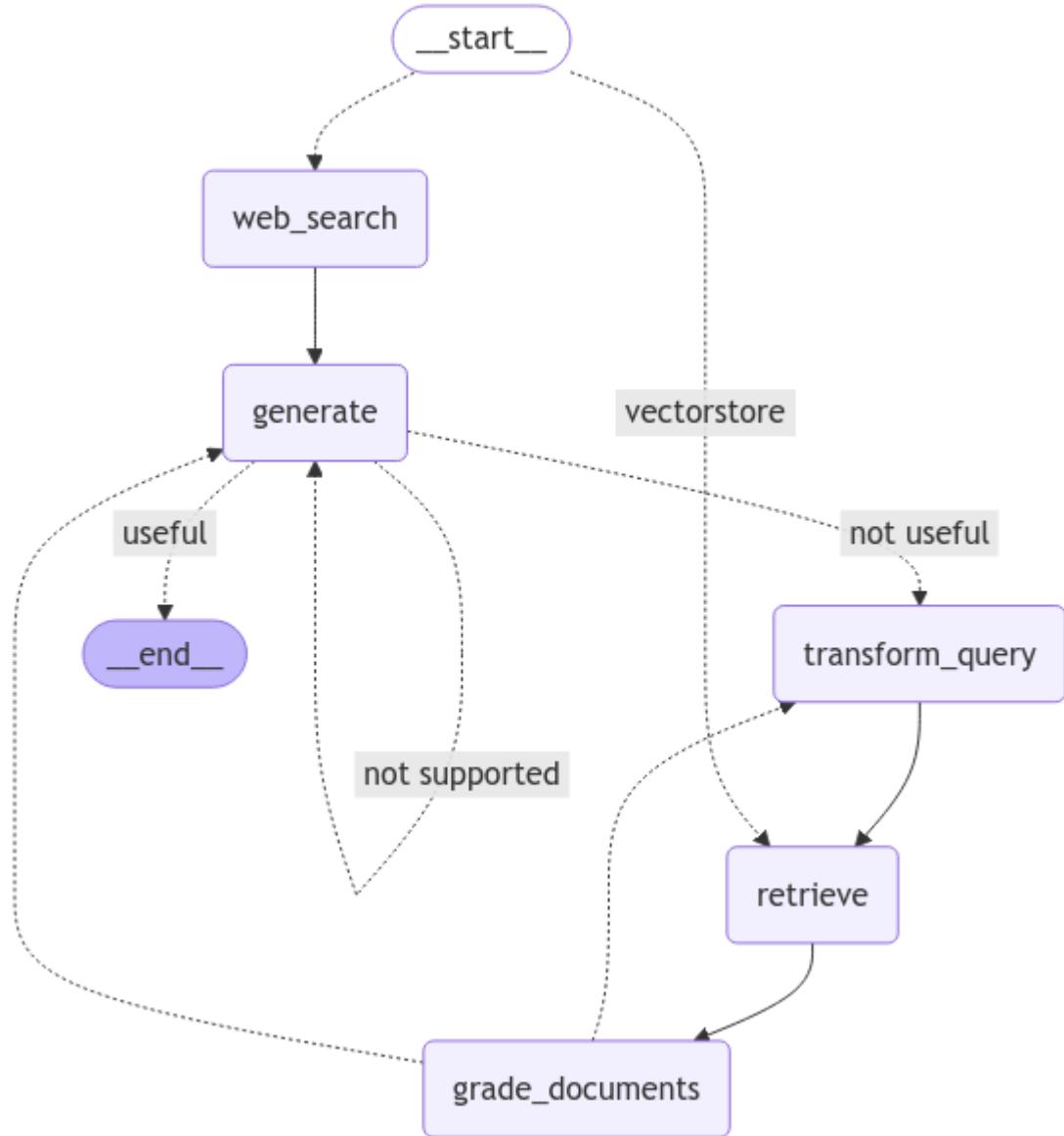


Fig. 10: Adaptive-RAG Graph

- Test
    - Relevant retrieval

```
import uuid

example = {"input": "What is the capital of China?"}
config = {"configurable": {"thread_id": str(uuid.uuid4())}}
state_dict = app.invoke({"question": example["input"], "steps": [], "config": config})
state_dict
```

Ouput:

```
---ROUTE QUESTION---
---ROUTE QUESTION TO WEB SEARCH---
---GENERATE---
---CHECK HALLUCINATIONS---
---DECISION: GENERATION IS GROUNDED IN DOCUMENTS---
---GRADE GENERATION vs QUESTION---
---DECISION: GENERATION ADDRESSES QUESTION---

{'question': <function __main__.grade_generation_v_documents_and_question(state)>,
 'generation': '{\n    "question": "<function grade_generation_v_documents_and_question at 0x7c7eb21f8670>",\n    "answer": "The capital city of China is Beijing, as mentioned in three documents.\n    The first document states that Beijing served as China\\'s capital\n    city in 1261, the second document confirms it as the current capital\n    with over 22 million residents, and the third document does not\n    directly mention the capital but is related to a study conducted in\n    China."\n}',\n'documents': [Document(metadata={'url': 'https://clintonwhitehouse3.archives.gov/WH/New/China/beijing.html'}, page_content='The modern day capital of China is Beijing (literally "Northern Capital"), which first served as China\\'s capital city in 1261, when the Mongol ruler Kublai\xA0...'),\n Document(metadata={'url': 'https://en.wikipedia.org/wiki/Beijing'}, page_content='Beijing, previously romanized as Peking, is the capital city of China. With more than 22 million residents, it is the world\\\'s most populous national capital\xA0...'),\n Document(metadata={'url': 'https://pubmed.ncbi.nlm.nih.gov/38294063/'}, page_content='Supercritical and homogenous transmission of monkeypox in the capital of China. J Med Virol. 2024 Feb;96(2):e29442. doi: 10.1002/jmv.29442. Authors. Yunjun\xA0...'),\n Document(metadata={'url': 'https://www.sciencedirect.com/science/article/pii/S0304387820301358'}, page_content='This paper investigates the impacts of fires on cognitive performance. We find that a one-standard-deviation increase in the difference between upwind and\xA0...')]}}
```

- Irrelevant retrieval

```

input = "What are the types of agent memory?"

example = {"input": input}
config = {"configurable": {"thread_id": str(uuid.uuid4())}}
state_dict = app.invoke({"question": example["input"], "steps": [], ↴
    ↪ config})
state_dict

```

Ouput:

```

---ROUTE QUESTION---
---ROUTE QUESTION TO RAG---
---RETRIEVE---
---CHECK DOCUMENT RELEVANCE TO QUESTION---
---GRADE: DOCUMENT RELEVANT---
---GRADE: DOCUMENT RELEVANT---
---GRADE: DOCUMENT NOT RELEVANT---
---GRADE: DOCUMENT RELEVANT---
---ASSESS GRADED DOCUMENTS---
---DECISION: GENERATE---
---GENERATE---
---CHECK HALLUCINATIONS---
---DECISION: GENERATION IS GROUNDED IN DOCUMENTS---
---GRADE GENERATION vs QUESTION---
---DECISION: GENERATION ADDRESSES QUESTION---
{'question': 'What are the types of agent memory?',
'generation': '{\n      "Short-term memory": "In-context learning,\n      ↪ restricted by the finite context window length of Transformer",\n      ↪ "Long-term memory": "External vector store that the agent can\n      ↪ attend to at query time, accessible via fast retrieval"\n    }',
'documents': [Document(metadata={'description': 'Building agents with\n      ↪ LLM (large language model) as its core controller is a cool concept.\n      ↪ Several proof-of-concepts demos, such as AutoGPT, GPT-Engineer and\n      ↪ BabyAGI, serve as inspiring examples. The potentiality of LLM extends\n      ↪ beyond generating well-written copies, stories, essays and programs;\n      ↪ it can be framed as a powerful general problem solver.\n      ↪ Agent System\n      ↪ Overview\n      ↪ In a LLM-powered autonomous agent system, LLM functions as\n      ↪ the agent\'s brain, complemented by several key components:\n      ↪ Planning\n      ↪ Subgoal and decomposition: The agent breaks down large\n      ↪ tasks into smaller, manageable subgoals, enabling efficient handling\n      ↪ of complex tasks.\n      ↪ Reflection and refinement: The agent can do self-\n      ↪ criticism and self-reflection over past actions, learn from mistakes\n      ↪ and refine them for future steps, thereby improving the quality of\n      ↪ final results.\n      ↪ Memory\n      ↪ Short-term memory: I would consider all\n      ↪ the in-context learning (See Prompt Engineering) as utilizing short-\n      ↪ term memory of the model to learn.\n      ↪ Long-term memory: This provides\n      ↪ '}}'
}

```

(continues on next page)

(continued from previous page)

→ the agent with the capability to retain and recall (infinite)  
 → information over extended periods, often by leveraging an external  
 → vector store and fast retrieval.  
 → Tool use  
 → The agent learns to  
 → call external APIs for extra information that is missing from the  
 → model weights (often hard to change after pre-training), including  
 → current information, code execution capability, access to proprietary  
 → information sources and more.  
 → Fig. 1. Overview of a LLM-powered autonomous agent system.  
 → Component One: Planning  
 → complicated task usually involves many steps. An agent needs to know  
 → what they are and plan ahead.', 'language': 'en', 'source': 'https://lilianweng.github.io/posts/2023-06-23-agent/', 'title': "LLM Powered Autonomous Agents | Lil'Log"}, page\_content='Fig. 7. Comparison of AD, ED, source policy and RL^2 on environments that require memory and exploration. Only binary reward is assigned. The source policies are trained with A3C for "dark" environments and DQN for watermaze.(Image source: Laskin et al. 2023)  
 → Component Two: Memory#  
 → (Big thank you to ChatGPT for helping me draft this section. I've learned a lot about the human brain and data structure for fast MIPS in my conversations with ChatGPT.)  
 → Types of Memory#  
 → Memory can be defined as the processes used to acquire, store, retain, and later retrieve information. There are several types of memory in human brains.  
 → nSensory Memory: This is the earliest stage of memory, providing the ability to retain impressions of sensory information (visual, auditory, etc) after the original stimuli have ended. Sensory memory typically only lasts for up to a few seconds. Subcategories include iconic memory (visual), echoic memory (auditory), and haptic memory (touch).'),

Document(metadata={'description': 'Building agents with LLM (large language model) as its core controller is a cool concept. Several proof-of-concepts demos, such as AutoGPT, GPT-Engineer and BabyAGI, serve as inspiring examples. The potentiality of LLM extends beyond generating well-written copies, stories, essays and programs; it can be framed as a powerful general problem solver.'})  
 → Agent System  
 → Overview  
 → In a LLM-powered autonomous agent system, LLM functions as the agent's brain, complemented by several key components:  
 → nPlanning  
 → nSubgoal and decomposition: The agent breaks down large tasks into smaller, manageable subgoals, enabling efficient handling of complex tasks.  
 → nReflection and refinement: The agent can do self-criticism and self-reflection over past actions, learn from mistakes and refine them for future steps, thereby improving the quality of final results.  
 → nMemory  
 → nShort-term memory: I would consider all the in-context learning (See Prompt Engineering) as utilizing short-term memory of the model to learn.  
 → nLong-term memory: This provides the agent with the capability to retain and recall (infinite)  
 → information over extended periods, often by leveraging an external  
 → vector store and fast retrieval.  
 → Tool use  
 → The agent learns to

(continues on next page)

(continued from previous page)

```

→ call external APIs for extra information that is missing from the
→ model weights (often hard to change after pre-training), including
→ current information, code execution capability, access to proprietary
→ information sources and more.\n\n\nFig. 1. Overview of a LLM-
→ powered autonomous agent system.\nComponent One: Planning\nA
→ complicated task usually involves many steps. An agent needs to know
→ what they are and plan ahead.', 'language': 'en', 'source': 'https://
→ lilianweng.github.io/posts/2023-06-23-agent/', 'title': "LLM Powered
→ Autonomous Agents | Lil'Log"}, page_content='Short-term memory: I
→ would consider all the in-context learning (See Prompt Engineering)
→ as utilizing short-term memory of the model to learn.\nLong-term
→ memory: This provides the agent with the capability to retain and
→ recall (infinite) information over extended periods, often by
→ leveraging an external vector store and fast retrieval.\n\n\nTool use
→ \n\nThe agent learns to call external APIs for extra information that
→ is missing from the model weights (often hard to change after pre-
→ training), including current information, code execution capability,
→ access to proprietary information sources and more.'),
Document(metadata={'description': 'Building agents with LLM (large
→ language model) as its core controller is a cool concept. Several
→ proof-of-concepts demos, such as AutoGPT, GPT-Engineer and BabyAGI,
→ serve as inspiring examples. The potentiality of LLM extends beyond
→ generating well-written copies, stories, essays and programs; it can
→ be framed as a powerful general problem solver.\nAgent System
→ Overview\nIn a LLM-powered autonomous agent system, LLM functions as
→ the agent's brain, complemented by several key components:\n\
→ nPlanning\n\nSubgoal and decomposition: The agent breaks down large
→ tasks into smaller, manageable subgoals, enabling efficient handling
→ of complex tasks.\nReflection and refinement: The agent can do self-
→ criticism and self-reflection over past actions, learn from mistakes
→ and refine them for future steps, thereby improving the quality of
→ final results.\n\n\nMemory\n\nShort-term memory: I would consider all
→ the in-context learning (See Prompt Engineering) as utilizing short-
→ term memory of the model to learn.\nLong-term memory: This provides
→ the agent with the capability to retain and recall (infinite)
→ information over extended periods, often by leveraging an external
→ vector store and fast retrieval.\n\n\nTool use\n\nThe agent learns to
→ call external APIs for extra information that is missing from the
→ model weights (often hard to change after pre-training), including
→ current information, code execution capability, access to proprietary
→ information sources and more.\n\n\nFig. 1. Overview of a LLM-
→ powered autonomous agent system.\nComponent One: Planning\nA
→ complicated task usually involves many steps. An agent needs to know
→ what they are and plan ahead.', 'language': 'en', 'source': 'https://
→ lilianweng.github.io/posts/2023-06-23-agent/', 'title': "LLM Powered
→ Autonomous Agents | Lil'Log"}, page_content='Sensory memory as

```

(continues on next page)

(continued from previous page)

↵learning embedding representations for raw inputs, including text, ↵  
 ↵image or other modalities; \nShort-term memory as in-context learning. ↵  
 ↵It is short and finite, as it is restricted by the finite context ↵  
 ↵window length of Transformer.\nLong-term memory as the external ↵  
 ↵vector store that the agent can attend to at query time, accessible ↵  
 ↵via fast retrieval.\n\nMaximum Inner Product Search (MIPS) #\nThe ↵  
 ↵external memory can alleviate the restriction of finite attention ↵  
 ↵span. A standard practice is to save the embedding representation of ↵  
 ↵information into a vector store database that can support fast ↵  
 ↵maximum inner-product search (MIPS). To optimize the retrieval speed, ↵  
 ↵the common choice is the approximate nearest neighbors (ANN)\u200b ↵  
 ↵algorithm to return approximately top k nearest neighbors to trade ↵  
 ↵off a little accuracy lost for a huge speedup.\nA couple common ↵  
 ↵choices of ANN algorithms for fast MIPS: ')}]}

## 5.5.4 Agentic RAG

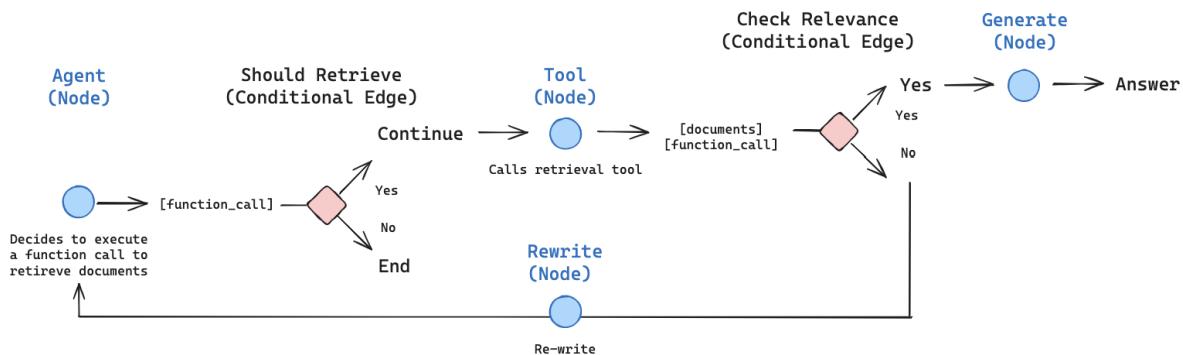


Fig. 11: Agentic-RAG langgraph diagram (source [Langgraph Agentic-rag](#))

- Load Models

```

from langchain_ollama import OllamaEmbeddings
from langchain_ollama.llms import OllamaLLM

# embedding model
embedding = OllamaEmbeddings(model="bge-m3")

# LLM
llm = OllamaLLM(temperature=0.0, model='mistral', format='json')

```

### Warning

You need to specify `format='json'` when Initializing `OllamaLLM`. otherwise you will get error:

The screenshot shows a Google Colab notebook titled "Run-ollama-in-colab.ipynb". The code cell contains Python code for initializing an OllamaLLM and generating a response. A validation error is displayed in the sidebar, stating: "Specifying format='json' when initializing OllamaLLM". The error message also includes a note: "Use code with caution".

```

from langchain_llms import OllamaLLM
template = """Question: {question}
Answer: Let's think step by step.
"""
prompt = ChatPromptTemplate.from_template(template)
llm = OllamaLLM(model="mistral", format='json')
response = llm.invoke("Explain the concept of artificial intelligence in simple terms.")
print(response)

{
    "role": "assistant",
    "model": "text-davinci-003",
    "pid": "123456789abcdef",
    "object": {
        "object": "answer"
    },
    "content": "Artificial Intelligence (AI) is a broad term that refers to the ability of a machine
}

```

- Create Index

```

from langchain.text_splitter import RecursiveCharacterTextSplitter
from langchain_community.document_loaders import WebBaseLoader
from langchain_community.vectorstores import Chroma
from langchain_ollama import OllamaEmbeddings # Import
→ OllamaEmbeddings instead

urls = [
    "https://lilianweng.github.io/posts/2023-06-23-agent/",
    "https://lilianweng.github.io/posts/2023-03-15-prompt-engineering/",
    "https://lilianweng.github.io/posts/2023-10-25-adv-attack-l1m/",
]

docs = [WebBaseLoader(url).load() for url in urls]
docs_list = [item for sublist in docs for item in sublist]

text_splitter = RecursiveCharacterTextSplitter.from_tiktoken_encoder(
    chunk_size=250, chunk_overlap=0
)

```

(continues on next page)

(continued from previous page)

```
doc_splits = text_splitter.split_documents(docs_list)

# Add to vectorDB
vectorstore = Chroma.from_documents(
    documents=doc_splits,
    collection_name="rag-chroma",
    embedding=OllamaEmbeddings(model="bge-m3"),
)
retriever = vectorstore.as_retriever(k=4)
```

- Define Tool

```
from langchain.tools.retriever import create_retriever_tool

retriever_tool = create_retriever_tool(
    retriever,
    "retrieve_blog_posts",
    "Search and return information about Lilian Weng blog posts on \
    LLM agents, prompt engineering, and adversarial attacks on LLMs.",
)

tools = [retriever_tool]
```

### Note

If you need web search, you can add web search tool we implement in *Corrective RAG* and *Adaptive RAG*. More details related to Langchain Agent can be found at [Langchain Agents](#) and [Langchain build-in tools](#).

```
# define tools

from langchain_google_community import GoogleSearchAPIWrapper, ↴
    GoogleSearchResults

from google.colab import userdata
api_key = userdata.get('GOOGLE_API_KEY')
cx = userdata.get('GOOGLE_CSE_ID')
# Replace with your actual API key and CX ID

# Create an instance of the GoogleSearchAPIWrapper
google_search_wrapper = GoogleSearchAPIWrapper(google_api_key=api_key, ↴
    google_cse_id=cx)

# Pass the api_wrapper to GoogleSearchResults
web_search_tool = GoogleSearchResults(api_wrapper=google_search_wrapper, ↴
    k=3)
```

```
tools = [retriever_tool, web_search_tool]

# define prompt and agent

from langchain import hub

## Get the prompt to use - you can modify this!
prompt = hub.pull("hwchase17/openai-functions-agent")
prompt.messages

## agent
from langchain.agents import create_tool_calling_agent

agent = create_tool_calling_agent(llm, tools, prompt)

# Agent executor

from langchain.agents import AgentExecutor

agent_executor = AgentExecutor(agent=agent, tools=tools, verbose=True)
agent_executor.invoke({"input": "whats the weather in sf?"})
```

- Retrieval Grader

```
### Retrieval Grader

from langchain_ollama.llms import OllamaLLM
from langchain.prompts import PromptTemplate

# Import from pydantic directly instead of langchain_core.pydantic_v1
from pydantic import BaseModel, Field
from langchain.output_parsers import PydanticOutputParser

# Data model
class GradeDocuments(BaseModel):
    """Binary score for relevance check on retrieved documents."""

    score: str = Field( # Changed field name to 'score'
        description="Documents are relevant to the question, 'yes' or 'no'")

parser = PydanticOutputParser(pydantic_object=GradeDocuments)

prompt = PromptTemplate(
    template="You are a grader assessing relevance of a retrieved
    document to a user question. \n
    Here is the retrieved document: \n\n {context} \n\n")
```

(continues on next page)

(continued from previous page)

```

Here is the user question: {question} \n
If the document contains keywords related to the user question,
grade it as relevant. \n
It does not need to be a stringent test. The goal is to filter out
erroneous retrievals. \n
Give a binary score 'yes' or 'no' score to indicate whether the document
is relevant to the question. \n
Provide the binary score as a JSON with a single key 'score' and no
preamble or explanation."",",
input_variables=["context", "question"],
partial_variables={"format_instructions": parser.get_format_
instructions()}
)

retrieval_grader = prompt | llm | parser
question = "agent memory"
docs = retriever.invoke(question)
doc_txt = docs[1].page_content
retrieval_grader.invoke({"question": question, "context": doc_txt})

```

Output:

```
GradeDocuments(score='yes')
```

- Agent State

```

from typing import Annotated, Sequence
from typing_extensions import TypedDict

from langchain_core.messages import BaseMessage

from langgraph.graph.message import add_messages

class AgentState(TypedDict):
    # The add_messages function defines how an update should be processed
    # Default is to replace. add_messages says "append"
    messages: Annotated[Sequence[BaseMessage], add_messages]

```

- Create the Graph

```

from typing import Annotated, Literal, Sequence
from typing_extensions import TypedDict

from langchain import hub
from langchain_core.messages import BaseMessage, HumanMessage

```

(continues on next page)

(continued from previous page)

```
from langchain_core.output_parsers import StrOutputParser
from langchain_core.prompts import PromptTemplate

from pydantic import BaseModel, Field
from langchain_experimental.llms.ollama_functions import OllamaFunctions

from langgraph.prebuilt import tools_condition

### Edges

def grade_documents(state) -> Literal["generate", "rewrite"]:
    """
    Determines whether the retrieved documents are relevant to the question.

    Args:
        state (messages): The current state

    Returns:
        str: A decision for whether the documents are relevant or not
    """

    print("---CHECK RELEVANCE---")

    messages = state["messages"]
    last_message = messages[-1]

    question = messages[0].content
    docs = last_message.content

    scored_result = retrieval_grader.invoke({"question": question, \
                                              "context": docs})

    score = scored_result.score

    if score == "yes":
        print("---DECISION: DOCS RELEVANT---")
        return "generate"

    else:
        print("---DECISION: DOCS NOT RELEVANT---")
        print(score)
        return "rewrite"
```

(continues on next page)

(continued from previous page)

```

### Nodes

def agent(state):
    """
    Invokes the agent model to generate a response based on the current_
    ↪state.
    Given the question, it will decide to retrieve using the retriever tool,
    or simply end.

    Args:
        state (messages): The current state

    Returns:
        dict: The updated state with the agent response appended to messages
    """
    print("---CALL AGENT---")
    messages = state["messages"]

    model = OllamaFunctions(model="mistral", format='json')
    model = model.bind_tools(tools)
    response = model.invoke(messages)
    # We return a list, because this will get added to the existing list
    return {"messages": [response]}

def rewrite(state):
    """
    Transform the query to produce a better question.

    Args:
        state (messages): The current state

    Returns:
        dict: The updated state with re-phrased question
    """
    print("---TRANSFORM QUERY---")
    messages = state["messages"]
    question = messages[0].content

    msg = [
        HumanMessage(
            content=f"""\n

```

(continues on next page)

(continued from previous page)

```

Look at the input and try to reason about the underlying semantic_
↳intent /
meaning. \n
Here is the initial question:
\n ----- \n
{question}
\n ----- \n
Formulate an improved question: "",
)
]

# Grader
response = llm.invoke(msg)
return {"messages": [response]}

def generate(state):
"""
Generate answer

Args:
    state (messages): The current state

Returns:
    dict: The updated state with re-phrased question
"""

print("---GENERATE---")
messages = state["messages"]
question = messages[0].content
last_message = messages[-1]

docs = last_message.content

# Prompt
prompt = hub.pull("rlm/rag-prompt")

# Post-processing
def format_docs(docs):
    return "\n\n".join(doc.page_content for doc in docs)

# Chain
rag_chain = prompt | llm | StrOutputParser()

# Run
response = rag_chain.invoke({"context": docs, "question": question})

```

(continues on next page)

(continued from previous page)

```

return {"messages": [response]}

# print("*" * 20 + "Prompt[rlm/rag-prompt]" + "*" * 20)
# # Show what the prompt looks like
# prompt = hub.pull("rlm/rag-prompt").pretty_print()

```

- Compile Graph

```

from langgraph.graph import END, StateGraph, START
from langgraph.prebuilt import ToolNode

# Define a new graph
workflow = StateGraph(AgentState)

# Define the nodes we will cycle between
workflow.add_node("agent", agent) # agent
retrieve = ToolNode([retriever_tool])
workflow.add_node("retrieve", retrieve) # retrieval
workflow.add_node("rewrite", rewrite) # Re-writing the question
workflow.add_node(
    "generate", generate
) # Generating a response after we know the documents are relevant
# Call agent node to decide to retrieve or not
workflow.add_edge(START, "agent")

# Decide whether to retrieve
workflow.add_conditional_edges(
    "agent",
    # Assess agent decision
    tools_condition,
    {
        # Translate the condition outputs to nodes in our graph
        "tools": "retrieve",
        END: END,
    },
)

# Edges taken after the `action` node is called.
workflow.add_conditional_edges(
    "retrieve",
    # Assess agent decision
    grade_documents,
)
workflow.add_edge("generate", END)
workflow.add_edge("rewrite", "agent")

```

(continues on next page)

(continued from previous page)

```
# Compile
graph = workflow.compile()
```

### Warning

OllamaLLM object has no attribute bind\_tools. You need to Install langchain-experimental: OllamaFunctions is initialized with the desired model name:

```
model = OllamaFunctions(model="mistral", format='json')
model = model.bind_tools(tools)
response = model.invoke(messages)
```

If you use OpenAI model, the code should be like:

```
model = ChatOpenAI(temperature=0, streaming=True, model="gpt-4-turbo")
model = model.bind_tools(tools)
response = model.invoke(messages)
```

- Graph visualization

```
from IPython.display import Image, display

try:
    display(Image(app.get_graph(xray=True).draw_mermaid_png()))
except:
    pass
```

Ouput

- Test

```
import pprint

inputs = {
    "messages": [
        ("user", "What does Lilian Weng say about the types of agent memory?
        ↪"),
    ]
}
for output in graph.stream(inputs):
    for key, value in output.items():
        pprint.pprint(f"Output from node '{key}':")
        pprint.pprint("---")
        pprint.pprint(value, indent=2, width=80, depth=None)
        pprint.pprint("\n---\n")
```

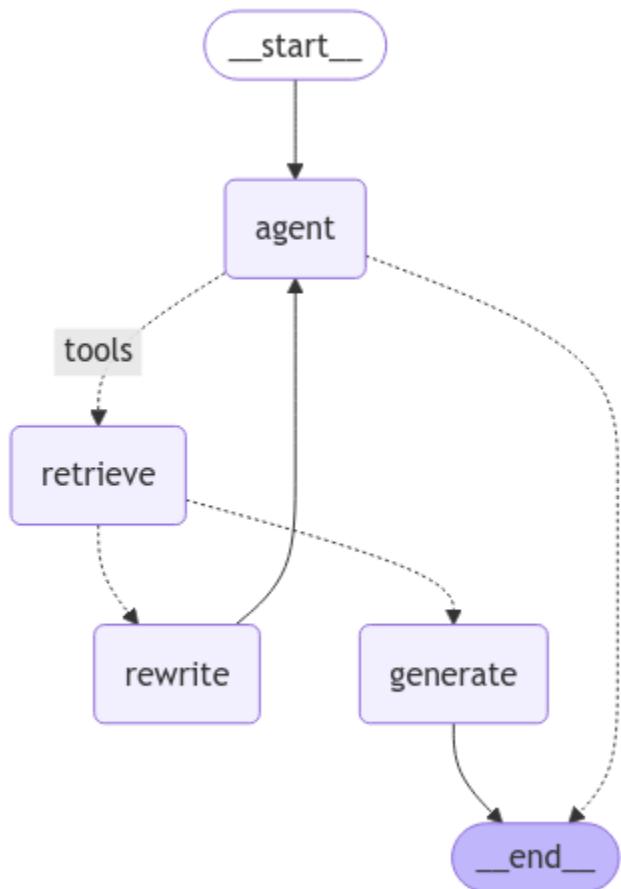


Fig. 12: Agentic-RAG Graph

Ouput:

```

---CALL AGENT---
"Output from node 'agent':"
'---'

{ 'messages': [ AIMessage(content='', additional_kwargs={}, response_
→metadata={}, id='run-ba7e9f54-7b32-44be-a39b-b083a6db462d-0', tool_calls=[_
→{'name': 'retrieve_blog_posts', 'args': {'query': 'types of agent memory'}_
→, 'id': 'call_4b1ac43a51c545cb942498b35321693a', 'type': 'tool_call'}])]}
'\n---\n'

---CHECK RELEVANCE---
---DECISION: DOCS RELEVANT---
"Output from node 'retrieve':"
'---'

{ 'messages': [ ToolMessage(content='Fig. 7. Comparison of AD, ED, source_
→policy and RL^2 on environments that require memory and exploration. Only_
→binary reward is assigned. The source policies are trained with A3C for_
→"dark" environments and DQN for watermaze.(Image source: Laskin et al.-
→2023)\nComponent Two: Memory#\n(Big thank you to ChatGPT for helping me_
→draft this section. I've learned a lot about the human brain and data_
→structure for fast MIPS in my conversations with ChatGPT.)\nTypes of_
→Memory#\nMemory can be defined as the processes used to acquire, store,_
→retain, and later retrieve information. There are several types of memory_
→in human brains.\n\n\nSensory Memory: This is the earliest stage of_
→memory, providing the ability to retain impressions of sensory_
→information (visual, auditory, etc) after the original stimuli have ended.-
→ Sensory memory typically only lasts for up to a few seconds.-
→Subcategories include iconic memory (visual), echoic memory (auditory),_
→and haptic memory (touch).\n\nShort-term memory: I would consider all the_
→in-context learning (See Prompt Engineering) as utilizing short-term_
→memory of the model to learn.\nLong-term memory: This provides the agent_
→with the capability to retain and recall (infinite) information over_
→extended periods, often by leveraging an external vector store and fast_
→retrieval.\n\n\nTool use\n\nThe agent learns to call external APIs for_
→extra information that is missing from the model weights (often hard to_
→change after pre-training), including current information, code execution_
→capability, access to proprietary information sources and more.\n\n
→Sensory memory as learning embedding representations for raw inputs,_
→including text, image or other modalities;\nShort-term memory as in-
→context learning. It is short and finite, as it is restricted by the_
→finite context window length of Transformer.\nLong-term memory as the_
→external vector store that the agent can attend to at query time,_
→accessible via fast retrieval.\n\nMaximum Inner Product Search (MIPS)#
→The external memory can alleviate the restriction of finite attention_
→span. A standard practice is to save the embedding representation of_
→information into a vector store database that can support fast maximum_
→inner-product search (MIPS). To optimize the retrieval speed, the common_

```

(continues on next page)

(continued from previous page)

```

→choice is the approximate nearest neighbors (ANN)\u200b algorithm to_
→return approximately top k nearest neighbors to trade off a little_
→accuracy lost for a huge speedup.\nA couple common choices of ANN_
→algorithms for fast MIPS:\n\nThey also discussed the risks, especially_
→with illicit drugs and bioweapons. They developed a test set containing a_
→list of known chemical weapon agents and asked the agent to synthesize_
→them. 4 out of 11 requests (36%) were accepted to obtain a synthesis_
→solution and the agent attempted to consult documentation to execute the_
→procedure. 7 out of 11 were rejected and among these 7 rejected cases, 5_
→happened after a Web search while 2 were rejected based on prompt only.\\
→nGenerative Agents Simulation#\nGenerative Agents (Park, et al. 2023) is_
→super fun experiment where 25 virtual characters, each controlled by a_
→LLM-powered agent, are living and interacting in a sandbox environment,_
→inspired by The Sims. Generative agents create believable simulacra of_
→human behavior for interactive applications.\nThe design of generative_
→agents combines LLM with memory, planning and reflection mechanisms to_
→enable agents to behave conditioned on past experience, as well as to_
→interact with other agents.\n\nMemory stream: is a long-term memory_
→module (external database) that records a comprehensive list of agents'_
→experience in natural language.', name='retrieve_blog_posts', id=
→'2ca2d54b-214b-4463-a491-20c8d08e79cc', tool_call_id='call_
→4b1ac43a51c545cb942498b35321693a')]}
'\n---\n'
---GENERATE---
/usr/local/lib/python3.10/dist-packages/langsmith/client.py:261:_
→LangSmithMissingAPIKeyWarning: API key must be provided when using hosted_
→LangSmith API
    warnings.warn(
"Output from node 'generate':"
'---'
{ 'messages': [ '{\n'
    '    "Lilian Weng describes three types of memory: Sensory '
    'Memory, Short-Term Memory, and Long-Term Memory. Sensory '
    'Memory is the earliest stage, lasting for up to a few '
    'seconds, and includes iconic (visual), echoic (auditory),'
→and '
    'haptic memory. Short-Term Memory is used for in-context '
    'learning and is finite due to the limited context window '
    'length of Transformer. Long-Term Memory provides agents'
→with '
    'the capability to retain and recall information over'
→extended '
    'periods by leveraging an external vector store and fast '
    '"retrieval.}'{: .language-json }. In the given context, it '
    'does not explicitly mention any specific agent memory'
→types '

```

(continues on next page)

(continued from previous page)

---

## CHAPTER SIX

---

# FINE TUNING

### Chinese proverb

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

### Colab Notebook for This Chapter

- Embedding Model Fine-tuning: [!\[\]\(12ece6c01938cae9bb46a06bef424ab2\_img.jpg\) Open in Colab](#)
- LLM (Llama 2 7B) Model Fine-tuning: [!\[\]\(24b4331a61988038dbce71cc4f00ea9f\_img.jpg\) Open in Colab](#)

Fine-tuning is a machine learning technique where a pre-trained model (like a large language model or neural network) is further trained on a smaller, specific dataset to adapt it to a particular task or domain. Instead of training a model from scratch, fine-tuning leverages the knowledge already embedded in the pre-trained model, saving time, computational resources, and data requirements.

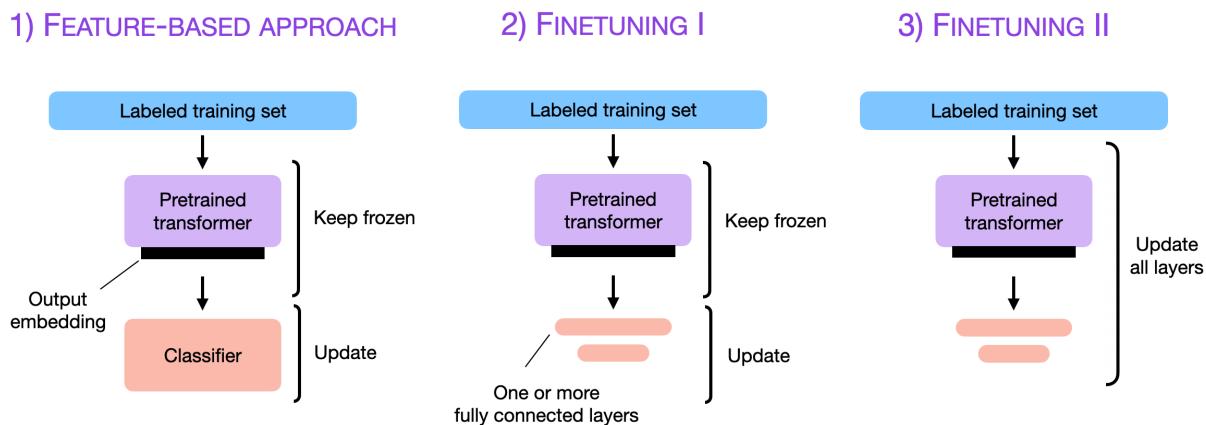


Fig. 1: The three conventional feature-based and finetuning approaches (Source Finetuning Sebastian).

## 6.1 Cutting-Edge Strategies for LLM Fine-Tuning

Over the past year, fine-tuning methods have made remarkable strides. Modern methods for fine-tuning LLMs focus on efficiency, scalability, and resource optimization. The following strategies are at the forefront:

### 6.1.1 LoRA (Low-Rank Adaptation)

**LoRA** reduces the number of trainable parameters by introducing **low-rank decomposition** into the fine-tuning process.

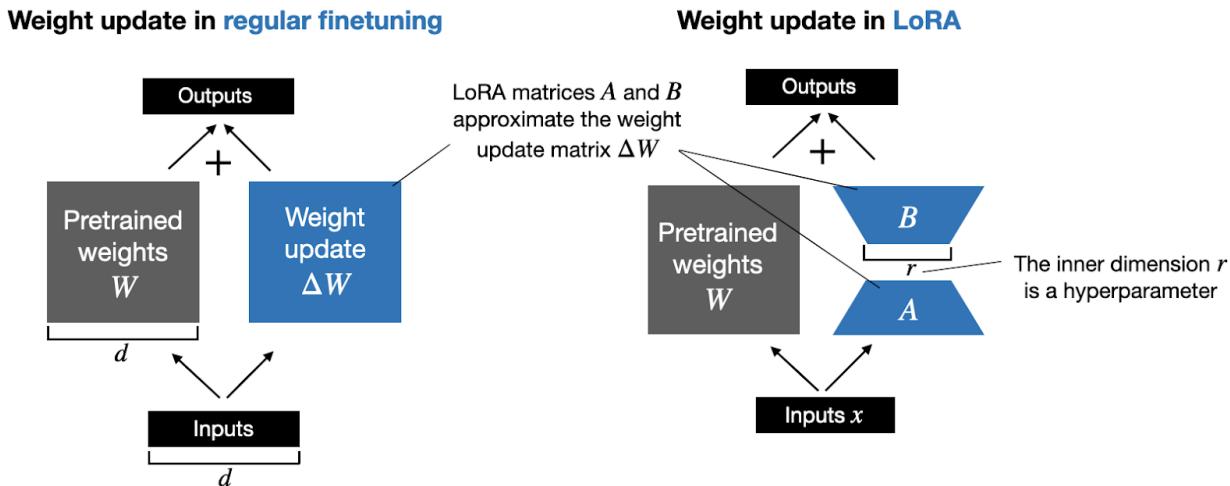


Fig. 2: Weight update matrix (Source [LORA Sebastian](#)).

#### How It Works:

- Instead of updating all model weights, LoRA injects **low-rank adapters** into the model's layers.
- The original pre-trained weights remain frozen; only the low-rank parameters are optimized.

#### Benefits:

- Reduces memory and computational requirements.
- Enables fine-tuning on resource-constrained hardware.

### 6.1.2 QLoRA (Quantized Low-Rank Adaptation)

**QLoRA** combines **low-rank adaptation** with **4-bit quantization** of the pre-trained model.

#### How It Works:

- The LLM is quantized to **4-bit precision** to reduce memory usage.
- LoRA adapters are applied to the quantized model for fine-tuning.
- Precision is maintained using methods like **NF4 (Normalized Float 4)** and double backpropagation.

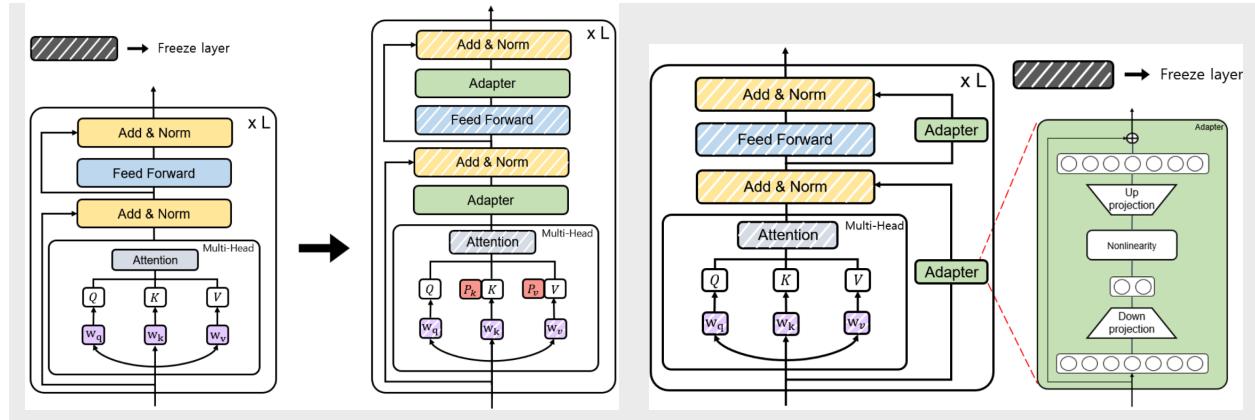
#### Benefits:

- Further reduces memory usage compared to LoRA.

- Enables fine-tuning of massive models on consumer-grade GPUs.

### 6.1.3 PEFT (Parameter-Efficient Fine-Tuning)

PEFT is a general framework for fine-tuning LLMs with minimal trainable parameters.



Source: [PEFT]

#### Techniques Under PEFT:

- **LoRA:** Low-rank adaptation of weights.
- **Adapters:** Small trainable layers inserted into the model.
- **Prefix Tuning:** Fine-tuning input prefixes instead of weights.
- **Prompt Tuning:** Optimizing soft prompts in the input space.

#### Benefits:

- Reduces the number of trainable parameters.
- Faster training and lower hardware requirements.

### 6.1.4 SFT (Supervised Fine-Tuning)

SFT adapts an LLM using a labeled dataset in a fully supervised manner.

#### How It Works:

- The model is initialized with pre-trained weights.
- It is fine-tuned on a task-specific dataset with a supervised loss function (e.g., cross-entropy).

#### Benefits:

- Achieves high performance on specific tasks.
- Essential for aligning models with labeled datasets.

### 6.1.5 Summary Table

Method	Description	Key Benefit
LoRA	Low-rank adapters for parameter-efficient tuning.	Reduces trainable parameters significantly.
QLoRA	LoRA with 4-bit quantization of the model.	Fine-tunes massive models on smaller hardware.
PEFT	General framework for efficient fine-tuning.	Includes LoRA, Adapters, Prefix Tuning, etc.
SFT	Supervised fine-tuning with labeled data.	High performance on task-specific datasets

These strategies represent the forefront of **LLM fine-tuning**, offering efficient and scalable solutions for real-world applications. To choose the most suitable strategy, consider the following factors:

- **Resource-Constrained Environments:** Use LoRA or QLoRA.
- **Large-Scale Models:** QLoRA for low-memory fine-tuning.
- **High Performance with Labeled Data:** SFT.
- **Minimal Setup:** Zero-shot or Few-shot learning.
- **General Efficiency:** Use PEFT frameworks.

## 6.2 Key Early Fine-Tuning Methods

Early fine-tuning methods laid the foundation for current approaches. These methods primarily focused on updating the entire model or selected components.

### 6.2.1 Full Fine-Tuning

All the parameters of a pre-trained model are updated using task-specific data *The three conventional feature-based and finetuning approaches (Souce Finetuning Sebastian)*. (right).

#### How It Works:

- The pre-trained model serves as the starting point.
- Fine-tuning is conducted on a smaller, labeled dataset using a supervised loss function.
- A low learning rate is used to prevent **catastrophic forgetting**.

#### Benefits:

- Effective at adapting models to specific tasks.

#### Challenges:

- Computationally expensive.
- Risk of overfitting on small datasets.

## 6.2.2 Feature-Based Approach

The pre-trained model is used as a **feature extractor**, while only a task-specific head is trained *The three conventional feature-based and finetuning approaches (Souce Finetuning Sebastian)*. (left).

### How It Works:

- The model processes inputs and extracts features (embeddings).
- A separate classifier (e.g., linear or MLP) is trained on top of these features.
- The pre-trained model weights remain **frozen**.

### Benefits:

- Computationally efficient since only the task-specific head is trained.

## 6.2.3 Layer-Specific Fine-Tuning

Only certain layers of the pre-trained model are fine-tuned while the rest remain frozen *The three conventional feature-based and finetuning approaches (Souce Finetuning Sebastian)*. (middle).

### How It Works:

- Earlier layers (which capture general features) are frozen.
- Later layers (closer to the output) are fine-tuned on task-specific data.

### Benefits:

- Balances computational efficiency and task adaptation.

## 6.2.4 Task-Adaptive Pre-training

Before fine-tuning on a specific task, the model undergoes additional **pre-training** on a domain-specific corpus.

### How It Works:

- A general pre-trained model is further pre-trained (unsupervised) on domain-specific data.
- Fine-tuning is then performed on the downstream task.

### Benefits:

- Provides a better starting point for domain-specific tasks.

## 6.3 Embedding Model Fine-Tuning

In the chapter *Retrieval-Augmented Generation*, we discussed how embedding models are crucial for the success of RAG applications. However, their general-purpose training often limits their effectiveness for company- or domain-specific use cases. Customizing embeddings with domain-specific data can significantly improve the retrieval performance of your RAG application.

In this chapter, we will demonstrate how to fine-tune embedding models using the `SentenceTransformersTrainer`, building on insights shared in the blog [fineTuneEmbedding] and

Sentence Transformer Training Overview. Our main contribution was introducing LoRA to enable functionality on NVIDIA T4 GPUs, while the rest of the pipeline and code remained almost unchanged.

### Note

Please ensure that the package versions are set as follows:

```
pip install "torch==2.1.2" tensorboard

pip install --upgrade \
    sentence-transformers>=3 \
    datasets==2.19.1 \
    transformers==4.41.2 \
    peft==0.10.0
```

Otherwise, you may encounter the error.

### 6.3.1 Prepare Dataset

We are going to directly use the synthetic dataset `philschmid/finanical-rag-embedding-dataset`, which includes 7,000 positive text pairs of questions and corresponding context from the [2023\\_10 NVIDIA SEC Filing](#).

```
from datasets import load_dataset

# Load dataset from the hub
dataset = load_dataset("philschmid/finanical-rag-embedding-dataset", split="train")
dataset = dataset.rename_column("question", "anchor")
dataset = dataset.rename_column("context", "positive")

# Add an id column to the dataset
dataset = dataset.add_column("id", range(len(dataset)))

# split dataset into a 10% test set
dataset = dataset.train_test_split(test_size=0.1)

# save datasets to disk
dataset["train"].to_json("train_dataset.json", orient="records")
dataset["test"].to_json("test_dataset.json", orient="records")
```

### Note

In practice, most dataset configurations will take one of four forms:

- **Positive Pair:** A pair of related sentences. This can be used both for symmetric tasks (semantic

textual similarity) or asymmetric tasks (semantic search), with examples including pairs of paraphrases, pairs of full texts and their summaries, pairs of duplicate questions, pairs of (query, response), or pairs of (source\_language, target\_language). Natural Language Inference datasets can also be formatted this way by pairing entailing sentences.

- **Triplets:** (anchor, positive, negative) text triplets. These datasets don't need labels.
- **Pair with Similarity Score:** A pair of sentences with a score indicating their similarity. Common examples are "Semantic Textual Similarity" datasets.
- **Texts with Classes:** A text with its corresponding class. This data format is easily converted by loss functions into three sentences (triplets) where the first is an "anchor", the second a "positive" of the same class as the anchor, and the third a "negative" of a different class.

Note that it is often simple to transform a dataset from one format to another, such that it works with your loss function of choice.

### 6.3.2 Import and Evaluate Pretrained Baseline Model

```
import torch
from sentence_transformers import SentenceTransformer
from sentence_transformers.evaluation import (
    InformationRetrievalEvaluator,
    SequentialEvaluator,
)
from sentence_transformers.util import cos_sim
from datasets import load_dataset, concatenate_datasets
from peft import LoraConfig, TaskType

model_id = "BAAI/bge-base-en-v1.5"
matryoshka_dimensions = [768, 512, 256, 128, 64] # Important: large to small

# Load a model
model = SentenceTransformer(
    model_id,
    trust_remote_code=True,
    device="cuda" if torch.cuda.is_available() else "cpu"
)

# Load test dataset
test_dataset = load_dataset("json", data_files="test_dataset.json", split="train")
train_dataset = load_dataset("json", data_files="train_dataset.json", split="train")
corpus_dataset = concatenate_datasets([train_dataset, test_dataset])

# Convert the datasets to dictionaries
corpus = dict(
```

(continues on next page)

(continued from previous page)

```

    zip(corpus_dataset["id"], corpus_dataset["positive"])
) # Our corpus (cid => document)
queries = dict(
    zip(test_dataset["id"], test_dataset["anchor"])
) # Our queries (qid => question)

# Create a mapping of relevant document (1 in our case) for each query
relevant_docs = {} # Query ID to relevant documents (qid => set([relevant_cids]))
for q_id in queries:
    relevant_docs[q_id] = [q_id]

matryoshka_evaluators = []
# Iterate over the different dimensions
for dim in matryoshka_dimensions:
    ir_evaluator = InformationRetrievalEvaluator(
        queries=queries,
        corpus=corpus,
        relevant_docs=relevant_docs,
        name=f"dim_{dim}",
        truncate_dim=dim, # Truncate the embeddings to a certain dimension
        score_functions={"cosine": cos_sim},
    )
    matryoshka_evaluators.append(ir_evaluator)

# Create a sequential evaluator
evaluator = SequentialEvaluator(matryoshka_evaluators)

```

**Note**

If you encounter the error `Cannot import name 'EncoderDecoderCache' from 'transformers'`, ensure that the package versions are set to `peft==0.10.0` and `transformers==4.37.2`.

```

# Evaluate the model
results = evaluator(model)

# Print the main score
for dim in matryoshka_dimensions:
    key = f"dim_{dim}_cosine_ndcg@10"
    print
    print(f"{key}: {results[key]}")

```

`dim_768_cosine_ndcg@10: 0.754897248109794`

(continues on next page)

(continued from previous page)

```
dim_512_cosine_ndcg@10: 0.7549275773474213
dim_256_cosine_ndcg@10: 0.7454714780163237
dim_128_cosine_ndcg@10: 0.7116728650043451
dim_64_cosine_ndcg@10: 0.6477174937632066
```

### 6.3.3 Loss Function with Matryoshka Representation

```
from sentence_transformers import SentenceTransformerModelCardData,
    SentenceTransformer

# Hugging Face model ID: https://huggingface.co/BAAI/bge-base-en-v1.5
model_id = "BAAI/bge-base-en-v1.5"

# load model with SDPA for using Flash Attention 2
model = SentenceTransformer(
    model_id,
    model_kwarg={"attn_implementation": "sdpa"},
    model_card_data=SentenceTransformerModelCardData(
        language="en",
        license="apache-2.0",
        model_name="BGE base Financial Matryoshka",
    ),
)

# Apply PEFT with PromptTuningConfig
peft_config = LoraConfig(
    task_type=TaskType.FEATURE_EXTRACTION,
    inference_mode=False,
    r=8,
    lora_alpha=32,
    lora_dropout=0.1,
)
model.add_adapter(peft_config, "dense")

# train loss
from sentence_transformers.losses import MatryoshkaLoss,
    MultipleNegativesRankingLoss

matryoshka_dimensions = [768, 512, 256, 128, 64] # Important: large to small
inner_train_loss = MultipleNegativesRankingLoss(model)
train_loss = MatryoshkaLoss(model,
                           inner_train_loss,
                           matryoshka_dims=matryoshka_dimensions)
```

**Note**

Loss functions play a critical role in the performance of your fine-tuned model. Sadly, there is no “one size fits all” loss function. Ideally, this table should help narrow down your choice of loss function(s) by matching them to your data formats.

You can often convert one training data format into another, allowing more loss functions to be viable for your scenario. For example,

Inputs	La-bels	Appropriate Loss Functions
single sentences	class	BatchAllTripletLoss, BatchHardSoftMarginTripletLoss, BatchHardTripletLoss, BatchSemiHardTripletLoss
single sentences	none	ContrastiveTensionLoss, DenoisingAutoEncoderLoss
(anchor, anchor) pairs	none	ContrastiveTensionLossInBatchNegatives
(damaged_sent	none	DenoisingAutoEncoderLoss
original_sent pairs		
(sentence_A, sentence_B)	class	SoftmaxLoss
pairs		
(anchor, positive)	none	MultipleNegativesRankingLoss, CachedMultipleNegativesRankingLoss, MultipleNegativesSymmetricRankingLoss, CachedMultipleNegativesSymmetricRankingLoss, MegaBatchMarginLoss, GISTEmbedLoss, CachedGISTEmbedLoss
pairs		
(anchor, positive/ negative)	1 if positive, 0 if negative	ContrastiveLoss, OnlineContrastiveLoss
pairs		
(sentence_A, sentence_B)	float similarity score	CoSENTLoss, AngleLoss, CosineSimilarityLoss
pairs		
(anchor, positive, negative)	none	MultipleNegativesRankingLoss, CachedMultipleNegativesRankingLoss, TripletLoss, CachedGISTEmbedLoss, GISTEmbedLoss
triplets		
(anchor, positive, negative_1, ..., negative_n)	none	MultipleNegativesRankingLoss, CachedMultipleNegativesRankingLoss, CachedGISTEmbedLoss

### 6.3.4 Fine-tune Embedding Model

```

from sentence_transformers import SentenceTransformerTrainingArguments
from sentence_transformers.training_args import BatchSamplers

# load train dataset again
train_dataset = load_dataset("json", data_files="train_dataset.json", split=
    "train")

# define training arguments
args = SentenceTransformerTrainingArguments(
    output_dir=output_dir, # output directory and hugging face model ID
    num_train_epochs=4, # number of epochs
    per_device_train_batch_size=32, # train batch size
    gradient_accumulation_steps=16, # for a global batch size of 512
    per_device_eval_batch_size=16, # evaluation batch size
    warmup_ratio=0.1, # warmup ratio
    learning_rate=2e-5, # learning rate, 2e-5 is a good
    value
        lr_scheduler_type="cosine", # use constant learning rate
    scheduler
        optim="adamw_torch_fused", # use fused adamw optimizer
        tf32=False, # use tf32 precision
        bf16=False, # use bf16 precision
        batch_sampler=BatchSamplers.NO_DUPLICATES, # MultipleNegativesRankingLoss
    benefits from no duplicate samples in a batch
        eval_strategy="epoch", # evaluate after each epoch
        save_strategy="epoch", # save after each epoch
        logging_steps=10, # log every 10 steps
        save_total_limit=3, # save only the last 3 models
        load_best_model_at_end=True, # load the best model when
    training ends
        metric_for_best_model="eval_dim_128_cosine_ndcg@10", # Optimizing for the
    best ndcg@10 score for the 128 dimension
        greater_is_better=True, # maximize the ndcg@10 score
)

from sentence_transformers import SentenceTransformerTrainer

trainer = SentenceTransformerTrainer(
    model=model, # bg-base-en-v1
    args=args, # training arguments
    train_dataset=train_dataset.select_columns(
        ["anchor", "positive"]
    ), # training dataset
    loss=train_loss,
    evaluator=evaluator,
)

```

(continues on next page)

(continued from previous page)

```
)
# start training
trainer.train()

# save the best model
#trainer.save_model()
trainer.model.save_pretrained("bge-base-finetuning")
```

### 6.3.5 Evaluate Fine-tuned Model

```
from sentence_transformers import SentenceTransformer

fine_tuned_model = SentenceTransformer(
    'bge-base-finetuning', device="cuda" if torch.cuda.is_available() else "cpu"
)
# Evaluate the model
results = evaluator(fine_tuned_model)

# # COMMENT IN for full results
# print(results)

# Print the main score
for dim in matryoshka_dimensions:
    key = f"dim_{dim}_cosine_ndcg@10"
    print(f"{key}: {results[key]}")
```

```
dim_768_cosine_ndcg@10: 0.7650276801072632
dim_512_cosine_ndcg@10: 0.7603951540556889
dim_256_cosine_ndcg@10: 0.754743133407988
dim_128_cosine_ndcg@10: 0.7205317098443929
dim_64_cosine_ndcg@10: 0.6609117856061502
```

### 6.3.6 Results Comparison

Although we did not observe the significant performance boost reported in the original blog, the fine-tuned model outperformed the baseline model across all dimensions using only 6.3k samples and partial parameter fine-tuning. More details can be found as follows:

Dimension	Baseline	Fine-tuned	Improvement
768	0.75490	0.76503	1.34%
512	0.75492	0.76040	0.73%
256	0.74547	0.75474	1.24%
128	0.71167	0.72053	1.24%
64	0.64772	0.66091	2.04%

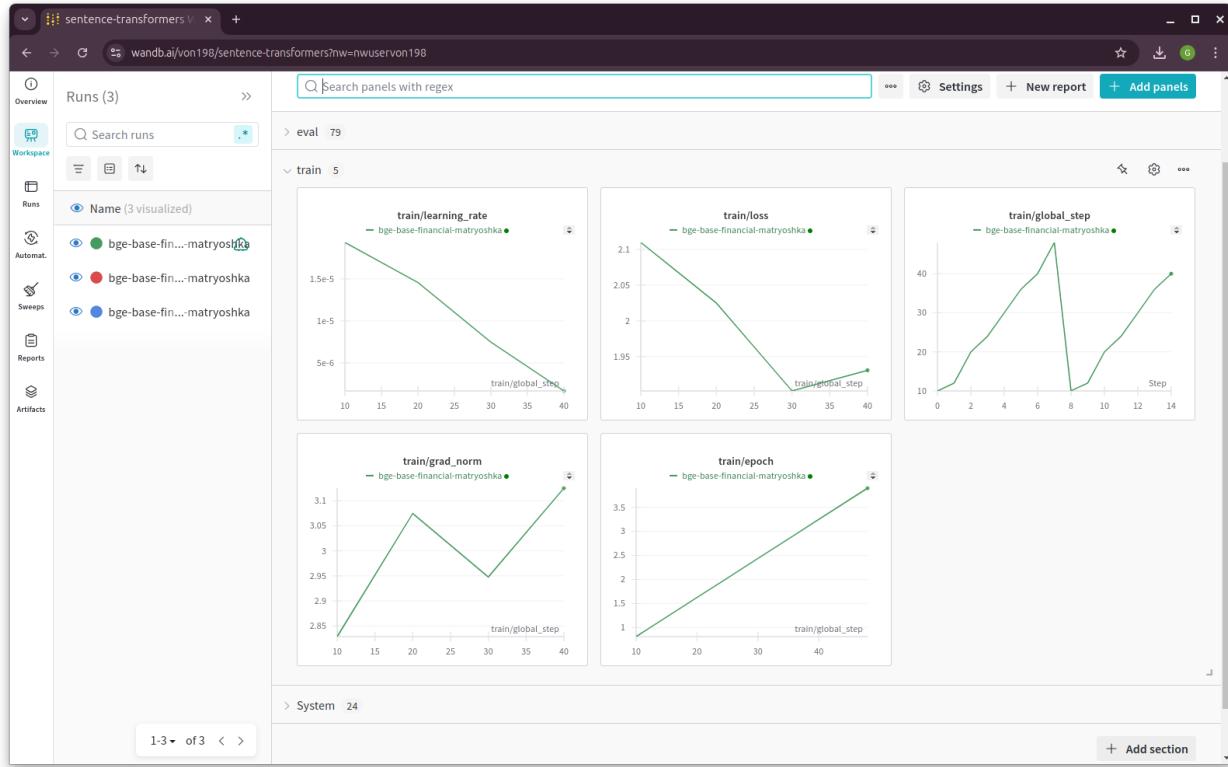


Fig. 3: Epoch, Training Loss/steps in Wandb

## 6.4 LLM Fine-Tuning

In this chapter, we will demonstrate how to fine-tune a Llama 2 model with 7 billion parameters using a T4 GPU with 16 GB of VRAM. Due to VRAM limitations, traditional fine-tuning is not feasible, making parameter-efficient fine-tuning (PEFT) techniques like LoRA or QLoRA essential. For this demonstration, we use QLoRA, which leverages 4-bit precision to significantly reduce VRAM consumption.

The following code is from notebook [fineTuneLLM], and the copyright belongs to the original author.

### 6.4.1 Load Dataset and Pretrained Model

```
# Step 1 : Load dataset (you can process it here)
dataset = load_dataset(dataset_name, split="train")

# Step 2 : Load tokenizer and model with QLoRA configuration
compute_dtype = getattr(torch, bnb_4bit_compute_dtype)

bnb_config = BitsAndBytesConfig(
    load_in_4bit=use_4bit,
    bnb_4bit_quant_type=bnb_4bit_quant_type,
    bnb_4bit_compute_dtype=compute_dtype,
    bnb_4bit_use_double_quant=use_nested_quant,
```

(continues on next page)

(continued from previous page)

```

)
# Step 3 :Check GPU compatibility with bfloat16
if compute_dtype == torch.float16 and use_4bit:
    major, _ = torch.cuda.get_device_capability()
    if major >= 8:
        print("=" * 80)
        print("Your GPU supports bfloat16: accelerate training with bf16=True")
        print("=" * 80)

# Step 4 :Load base model
model = AutoModelForCausalLM.from_pretrained(
    model_name,
    quantization_config=bnb_config,
    device_map=device_map
)
model.config.use_cache = False
model.config.pretraining_tp = 1

# Step 5 :Load LLaMA tokenizer
tokenizer = AutoTokenizer.from_pretrained(model_name, trust_remote_code=True)
tokenizer.add_special_tokens({'pad_token': '[PAD]'})
tokenizer.pad_token = tokenizer.eos_token
tokenizer.padding_side = "right"

```

#### 6.4.2 Fine-tuning Configuration

```

# Step 6 :Load LoRA configuration
peft_config = LoraConfig(
    lora_alpha=lora_alpha,
    lora_dropout=lora_dropout,
    r=lora_r,
    bias="none",
    task_type="CAUSAL_LM",
)

# Step 7 :Set training parameters
training_arguments = TrainingArguments(
    output_dir=output_dir,
    num_train_epochs=num_train_epochs,
    per_device_train_batch_size=per_device_train_batch_size,
    gradient_accumulation_steps=gradient_accumulation_steps,
    optim=optim,
    save_steps=save_steps,
    logging_steps=logging_steps,

```

(continues on next page)

(continued from previous page)

```
learning_rate=learning_rate,
weight_decay=weight_decay,
fp16=fp16,
bf16=bf16,
max_grad_norm=max_grad_norm,
max_steps=max_steps,
warmup_ratio=warmup_ratio,
group_by_length=group_by_length,
lr_scheduler_type=lr_scheduler_type,
report_to="tensorboard"
)
```

#### 6.4.3 Fine-tune model

```
# Step 8 :Set supervised fine-tuning parameters
trainer = SFTTrainer(
    model=model,
    train_dataset=dataset,
    peft_config=peft_config,
    dataset_text_field="text",
    max_seq_length=max_seq_length,
    tokenizer=tokenizer,
    args=training_arguments,
    packing=packing,
)

# Step 9 :Train model
trainer.train()

# Step 10 :Save trained model
trainer.model.save_pretrained(new_model)
```

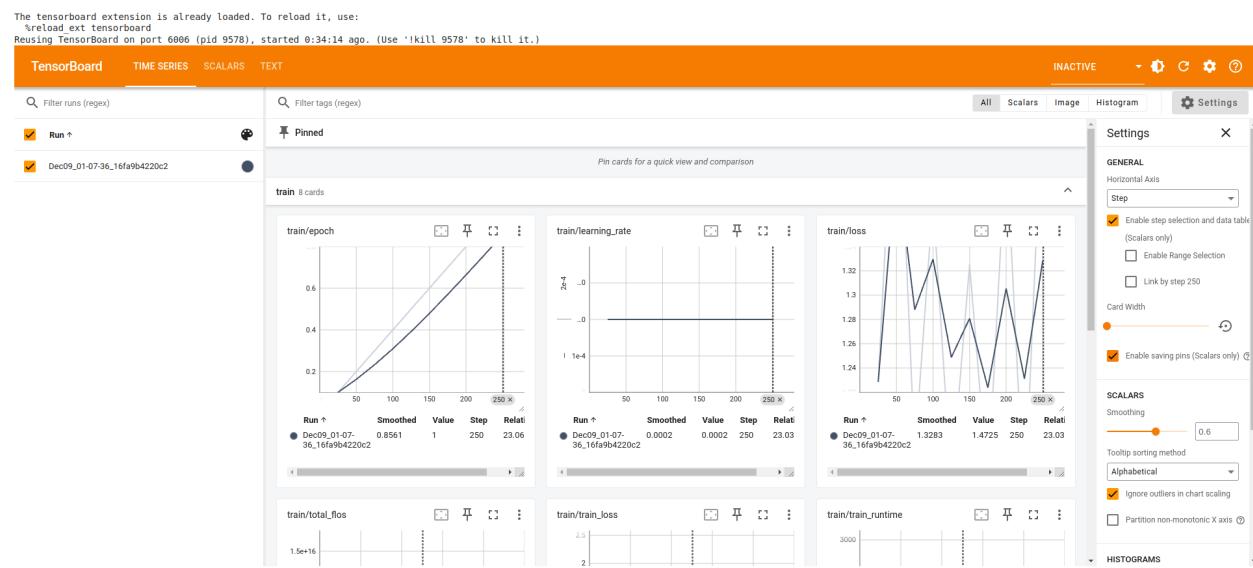


Fig. 4: Llama 2 Model Fine-Tuning TensorBoard

## PRE-TRAINING

### Proverb

Pre-training as we know it will end. – Ilya Sutskever at neurips 2024

### Pre-training as we know it will end

Compute is growing:

- Better hardware
- Better algorithms
- Larger clusters

Data is not growing:

- We have but one internet
- **The fossil fuel of AI**

Fig. 1: Ilya Sutskever at neurips 2024

In industry, most companies focus primarily on prompt engineering, RAG, and fine-tuning, while advanced techniques like pre-training from scratch or deep model customization remain less common due to the significant resources and expertise required.

LLMs, like GPT (Generative Pre-trained Transformer), BERT (Bidirectional Encoder Representations from Transformers), and others, are large-scale models built using the transformer architecture. These models are trained on vast amounts of text data to learn patterns in language, enabling them to generate human-like text,

answer questions, summarize information, and perform other natural language processing tasks.

This chapter delves into transformer models, drawing on insights from [The Annotated Transformer](#) and [Tracing the Transformer in Diagrams](#), to explore their underlying architecture and practical applications.

---

## CHAPTER EIGHT

---

# LLM EVALUATION METRICS

### Colab Notebook for This Chapter

GEval with DeepEval: [Open in Colab](#)

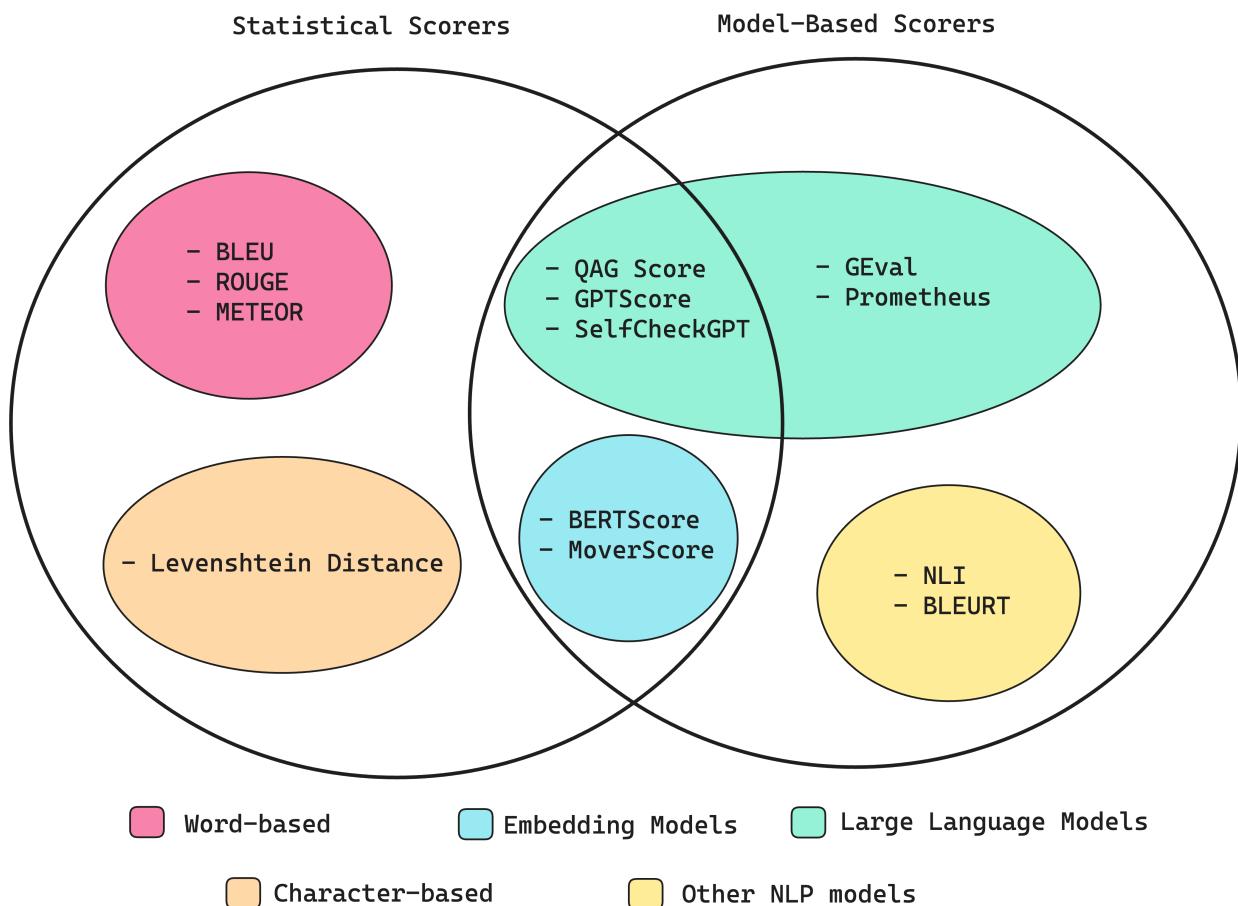


Fig. 1: Types of metric scorers (Source: LLM Evaluation Metrics: The Ultimate LLM Evaluation Guide)

## 8.1 Statistical Scorers (Traditional Metrics)

These metrics evaluate text outputs based on statistical comparisons to references or expected outputs.

### Note

I completely agree with the author of [LLM Evaluation Metrics: The Ultimate LLM Evaluation Guide](#) that statistical scoring methods are, in my opinion, non-essential to focus on. These methods tend to perform poorly whenever reasoning is required, making them too inaccurate as scorers for most LLM evaluation criteria. Additionally, more advanced metrics, such as GEval [\*GEval with DeepEval\*](#), provide significantly better alternatives.

- **BLEU (Bilingual Evaluation Understudy):** Measures overlap of n-grams between generated and reference texts. Common for translation tasks.
- **ROUGE (Recall-Oriented Understudy for Gisting Evaluation):** Focuses on recall of n-grams (ROUGE-N), longest common subsequences (ROUGE-L), and skip bigrams (ROUGE-S). Popular in summarization tasks.
- **METEOR (Metric for Evaluation of Translation with Explicit ORdering):** Considers synonymy and paraphrasing via stemming and synonym matching.
- **TER (Translation Edit Rate):** Measures the number of edits required to turn the generated output into the reference text.
- **CIDEr (Consensus-based Image Description Evaluation):** Designed for image captioning, using TF-IDF weighting of n-grams.
- **BERTScore:** Leverages contextual embeddings (e.g., BERT) to compute similarity between generated and reference texts.
- **GLEU (Google BLEU):** A variation of BLEU designed for grammatical error correction tasks.

## 8.2 Model-Based Scorers (Learned Metrics)

These metrics employ models trained to assess the quality of generated text, often based on human annotations.

- **BLEURT:** Combines pre-trained models (e.g., BERT) with fine-tuning on human judgment data.
- **COMET (Cross-lingual Optimized Metric for Evaluation of Translation):** A neural network model trained on translation quality data.
- **PRISM:** Measures semantic similarity by paraphrasing both the hypothesis and reference into a shared space.
- **UniEval:** A unified framework for evaluation across multiple tasks, focusing on both factual accuracy and linguistic quality.
- **Perplexity:** Estimates the likelihood of generated text under the original model's probability distribution (lower is better).
- **GPTScore:** Uses a large pre-trained LLM (e.g., GPT-4) to rate the quality of outputs.

- **MAUVE:** Measures the divergence between the distribution of generated text and that of human-written text.
- **DRIFT:** Focuses on domain-specific evaluation, checking how well outputs align with domain-specific data distributions.

## 8.3 Human-Centric Evaluations (Augmenting Metrics)

While not automated, human evaluations are crucial for assessing subjective qualities such as:

- **Fluency**
- **Coherence**
- **Relevance**
- **Factuality**
- **Style Appropriateness**

Both statistical and model-based scorers are often used in tandem with human evaluation to ensure a holistic assessment of LLM outputs.

## 8.4 G-Eval with DeepEval

G-Eval is a recently developed evaluation framework developed from paper [[G-Eval](#)] to assess large language models (LLMs) using GPT-based evaluators. It leverages the capabilities of advanced LLMs (like GPT-4 or beyond) to rate and critique the outputs of other models, including themselves, across various tasks. This approach shifts the evaluation paradigm by relying on the intrinsic understanding and reasoning power of the models, rather than traditional metrics.

### 8.4.1 G-Eval Algorithm

### 8.4.2 G-Eval with DeepEval

In DeepEval, a metric serves as a standard for measuring the performance of an LLM's output based on specific criteria of interest. Essentially, while the metric functions as the “ruler”, a test case represents the subject being measured. [DeepEval](#), provides a variety of default metrics to help you get started quickly, including:

- G-Eval
- Summarization
- Faithfulness
- Answer Relevancy
- Contextual Relevancy
- Contextual Precision
- Contextual Recall
- Ragas

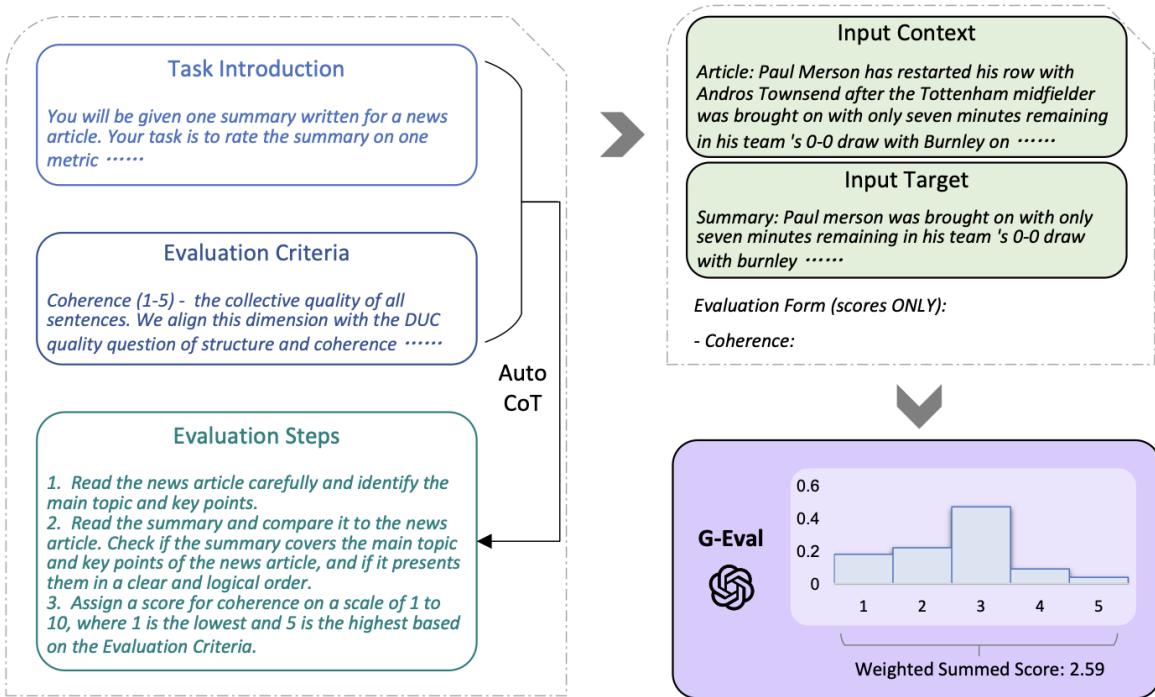


Figure 1: The overall framework of G-EVAL. We first input Task Introduction and Evaluation Criteria to the LLM, and ask it to generate a CoT of detailed Evaluation Steps. Then we use the prompt along with the generated CoT to evaluate the NLG outputs in a form-filling paradigm. Finally, we use the probability-weighted summation of the output scores as the final score.

Fig. 2: G-Eval Algorithm (Source: [GEval])

- Hallucination
- Toxicity
- Bias

DeepEval also provides conversational metrics, designed to evaluate entire conversations rather than individual, granular LLM interactions. These include:

- Conversation Completeness
- Conversation Relevancy
- Knowledge Retention
- **Set Up Local Model**

```
deepeval set-local-model --model-name='mistral' \
--base-url="http://localhost:11434/v1/" \
--api-key="ollama"
```

- **Default Metrics**

- AnswerRelevancyMetric

```
from deepeval import evaluate
from deepeval.metrics import AnswerRelevancyMetric
from deepeval.test_case import LLMTTestCase

answer_relevancy_metric = AnswerRelevancyMetric(threshold=0.7)
test_case = LLMTTestCase(
    input="What if these shoes don't fit?",
    # Replace this with the actual output from your LLM
    application="We offer a 30-day full refund at no extra costs.",
    retrieval_context=["All customers are eligible for a 30 day full refund at no extra costs."]
)
evaluate([test_case], [answer_relevancy_metric])
```

- \* Metrics Summary

- Answer Relevancy (score: 1.0, threshold: 0.7, strict: False, evaluation model: local model, reason: The score is 1.00 because it directly and accurately answered the question about shoe fitting, making it highly relevant., error: None)

- \* For test case:

- input: What if these shoes don't fit?
- actual output: We offer a 30-day full refund at no extra costs.
- expected output: None

- context: None
- retrieval context: ['All customers are eligible for a 30 day full refund at no extra costs.']}
- \* Overall Metric Pass Rates

Answer Relevancy: 100.00% pass rate

```
EvaluationResult(test_results=[TestResult(name='test_case_0',  
    success=True, metrics_data=[MetricData(name='Answer Relevancy',  
        threshold=0.7, success=True, score=1.0, reason='The score is  
        1.00 because it directly and accurately answered the question  
        about shoe fitting, making it highly relevant.', strict_  
        mode=False, evaluation_model='local model', error=None,  
        evaluation_cost=0.0, verbose_logs='Statements:\n\n    "We  
        offer a 30-day full refund",\n    "The refund does not incur  
        any additional costs"\n    \n    Verdicts:\n    {\n        "verdict": "yes",\n        "reason": "The statements about the  
        refund policy are relevant to addressing the input, which asks  
        about what to do if the shoes don\'t fit."\n    },\n    {\n        "verdict": "yes",\n        "reason": "The statement that  
        the refund does not incur any additional costs is also  
        relevant as it provides further information about the refund  
        process.\n    }\n) ], conversational=False, multimodal=False,  
    input="What if these shoes don't fit?", actual_output='We  
offer a 30-day full refund at no extra costs.', expected_  
output=None, context=None, retrieval_context=['All customers  
are eligible for a 30 day full refund at no extra costs.']),  
    confident_link=None)
```

- FaithfulnessMetric

```
from deepeval import evaluate
from deepeval.metrics import FaithfulnessMetric
from deepeval.test_case import LLMTTestCase

# input
input = "What if these shoes don't fit?"

# Replace this with the actual output from your LLM application
actual_output = "We offer a 30-day full refund at no extra cost."

# Replace this with the actual retrieved context from your RAG
# pipeline
retrieval_context = ["All customers are eligible for a 30 day  
full refund at no extra cost."]
```

(continues on next page)

(continued from previous page)

```

metric = FaithfulnessMetric(
    threshold=0.7,
    #model="gpt-4",
    include_reason=True
)
test_case = LLMTestCase(
    input=input,
    actual_output=actual_output,
    retrieval_context=retrieval_context
)

metric.measure(test_case)
print(metric.score)
print(metric.reason)

# or evaluate test cases in bulk
evaluate([test_case], [metric])

```

\* Metrics Summary

- Faithfulness (score: 1.0, threshold: 0.7, strict: False, evaluation model: local model, reason: The faithfulness score is 1.00 because there are no contradictions found between the actual output and the retrieval context., error: None)

\* For test case:

- input: What if these shoes don't fit?
- actual output: We offer a 30-day full refund at no extra cost.
- expected output: None
- context: None
- retrieval context: ['All customers are eligible for a 30 day full refund at no extra cost.']}

\* Overall Metric Pass Rates

Faithfulness: 100.00% pass rate

```

EvaluationResult(test_results=[TestResult(name='test_case_0',
    success=True, metrics_data=[MetricData(name='Faithfulness',
        threshold=0.7, success=True, score=1.0, reason='The
        faithfulness score is 1.00 because there are no contradictions
        found between the actual output and the retrieval context.',

        strict_mode=False, evaluation_model='local model', error=None,
        evaluation_cost=0.0, verbose_logs='Truths (limit=None):\n\n
        "Customers are eligible for a 30 day full refund.",\n        "The
        refund is at no extra cost."\n\n\nClaims:\n\n        "The
        '

```

(continues on next page)

(continued from previous page)

```

→refund is offered for a period of 30 days.",\n      "The refund\n
→does not incur any additional costs."\n] \n \nVerdicts:\n[\n  ↳ {\n    "verdict": "yes",\n      "reason": null\n  },\n  ↳ {\n    "verdict": "yes",\n      "reason": null\n  }\n]\n)], conversational=False, multimodal=False, input="What\n→if these shoes don't fit?", actual_output='We offer a 30-day\n→full refund at no extra cost.', expected_output=None,\n→context=None, retrieval_context=['All customers are eligible\n→for a 30 day full refund at no extra cost.']), confident_\n→link=None)

```

- ContextualPrecisionMetric

```

from deepeval import evaluate
from deepeval.metrics import ContextualPrecisionMetric
from deepeval.test_case import LLMTTestCase

# input
input = "What if these shoes don't fit?"

# Replace this with the actual output from your LLM application
actual_output = "We offer a 30-day full refund at no extra cost."

# Replace this with the expected output from your RAG generator
expected_output = "You are eligible for a 30 day full refund at\n→no extra cost."

# Replace this with the actual retrieved context from your RAG
→pipeline
retrieval_context = ["All customers are eligible for a 30 day\n→full refund at no extra cost."]

metric = ContextualPrecisionMetric(
    threshold=0.7,
    #model="gpt-4",
    include_reason=True
)
test_case = LLMTTestCase(
    input=input,
    actual_output=actual_output,
    expected_output=expected_output,
    retrieval_context=retrieval_context
)

metric.measure(test_case)
print(metric.score)

```

(continues on next page)

(continued from previous page)

```

print(metric.reason)

# or evaluate test cases in bulk
evaluate([test_case], [metric])

```

\* Metrics Summary

- Contextual Precision (score: 1.0, threshold: 0.7, strict: False, evaluation model: local model, reason: The contextual precision score is 1.00 because the node ranked first (with reason: ‘The text verifies that customers are indeed eligible for a 30 day full refund at no extra cost.’) is relevant and correctly placed as the highest-ranked response to the input ‘What if these shoes don’t fit?’. All other nodes, if present, should be ranked lower due to their irrelevance to the question., error: None)

\* For test case:

- input: What if these shoes don’t fit?
- actual output: We offer a 30-day full refund at no extra cost.
- expected output: You are eligible for a 30 day full refund at no extra cost.
- context: None
- retrieval context: [‘All customers are eligible for a 30 day full refund at no extra cost.’]

\* Overall Metric Pass Rates

- Contextual Precision: 100.00% pass rate

- ContextualRecallMetric

```

from deepeval import evaluate
from deepeval.metrics import ContextualRecallMetric
from deepeval.test_case import LLMTTestCase

metric = ContextualRecallMetric(
    threshold=0.7,
    model="gpt-4",
    include_reason=True
)
test_case = LLMTTestCase(
    input=input,
    actual_output=actual_output,
    expected_output=expected_output,
    retrieval_context=retrieval_context
)

metric.measure(test_case)

```

(continues on next page)

(continued from previous page)

```
print(metric.score)
print(metric.reason)

# or evaluate test cases in bulk
evaluate([test_case], [metric])
```

\* Metrics Summary

- Contextual Recall (score: 1.0, threshold: 0.7, strict: False, evaluation model: local model, reason: The score is 1.00 because the expected output is exactly as stated in the retrieval context., error: None)

\* For test case:

- input: What if these shoes don't fit?
- actual output: We offer a 30-day full refund at no extra cost.
- expected output: You are eligible for a 30 day full refund at no extra cost.
- context: None
- retrieval context: ['All customers are eligible for a 30 day full refund at no extra cost.]

\* Overall Metric Pass Rates

- Contextual Recall: 100.00% pass rate

- HallucinationMetric

```
from deepeval import evaluate
from deepeval.metrics import HallucinationMetric
from deepeval.test_case import LLMTTestCase

# input

input = "What was the blond doing?"

# Replace this with the actual documents that you are passing as
→input to your LLM.
context=["A man with blond-hair, and a brown shirt drinking out"
→of a public water fountain."]

# Replace this with the actual output from your LLM application
actual_output="A blond drinking water in public."

test_case = LLMTTestCase(
    input= input,
    actual_output=actual_output,
    context=context
```

(continues on next page)

(continued from previous page)

```

)
metric = HallucinationMetric(threshold=0.5)

metric.measure(test_case)
print(metric.score)
print(metric.reason)

# or evaluate test cases in bulk
evaluate([test_case], [metric])

```

\* Metrics Summary

- Hallucination (score: 0.0, threshold: 0.5, strict: False, evaluation model: local model, reason: The score is 0.00 because the actual output correctly aligns with the provided context., error: None)

\* For test case:

- input: What was the blond doing?
- actual output: A blond drinking water in public.
- expected output: None
- context: ['A man with blond-hair, and a brown shirt drinking out of a public water fountain.']}
- retrieval context: None

\* Overall Metric Pass Rates

Hallucination: 100.00% pass rate

• Custom Metrics

```

from deepeval.metrics import GEval
from deepeval.test_case import LLMTTestCaseParams

correctness_metric = GEval(
    name="Correctness",
    criteria="Determine whether the actual output is factually correct based on the expected output.",
    # NOTE: you can only provide either criteria or evaluation_steps, and not both
    evaluation_steps=[
        "Check whether the facts in 'actual output' contradicts any facts in 'expected output'",
        "You should also heavily penalize omission of detail",
        "Vague language, or contradicting OPINIONS, are OK"
    ],
    evaluation_params=[LLMTTestCaseParams.INPUT, LLMTTestCaseParams.

```

(continues on next page)

(continued from previous page)

```

    ↵ACTUAL_OUTPUT, LLMTestCaseParams.EXPECTED_OUTPUT],
)

test_case = LLMTestCase(
    input="The dog chased the cat up the tree, who ran up the tree?
",
    actual_output="It depends, some might consider the cat, while
others might argue the dog.",
    expected_output="The cat."
)

correctness_metric.measure(test_case)
print(correctness_metric.score)
print(correctness_metric.reason)

```

Event loop **is** already running. Applying nest\_asyncio patch to allow  
**async** execution...

0.1

The actual output does **not** match the expected output **and** omits  
specific details about which animal climbed the tree.

Evaluation Framework:

```

from deepeval import evaluate
from deepeval.metrics import GEval
from deepeval.test_case import LLMTestCase
from deepeval.test_case import LLMTestCaseParams

correctness_metric = GEval(
    name="Correctness",
    criteria="Determine whether the actual output is factually correct,
based on the expected output.",
    # NOTE: you can only provide either criteria or evaluation_steps,
and not both
    evaluation_steps=[
        "Check whether the facts in 'actual output' contradicts any
facts in 'expected output'",
        "You should also heavily penalize omission of detail",
        "Vague language, or contradicting OPINIONS, are OK"
    ],
    evaluation_params=[LLMTestCaseParams.INPUT, LLMTestCaseParams.
ACTUAL_OUTPUT, LLMTestCaseParams.EXPECTED_OUTPUT],
)

test_case = LLMTestCase(
    input="The dog chased the cat up the tree, who ran up the tree?",
```

(continues on next page)

(continued from previous page)

```

    actual_output="It depends, some might consider the cat, while others might argue the dog.",
    expected_output="The cat."
)

evaluate([test_case], [correctness_metric])

```

- Metrics Summary
  - Correctness (GEval) (score: 0.2, threshold: 0.5, strict: False, evaluation model: local model, reason: Actual output omits the expected detail (the cat) and contradicts the expected output., error: None)
- For test case:
  - input: The dog chased the cat up the tree, who ran up the tree?
  - actual output: It depends, some might consider the cat, while others might argue the dog.
  - expected output: The cat.
  - context: None
  - retrieval context: None
- Overall Metric Pass Rates

Correctness (GEval): 0.00% pass rate

```

EvaluationResult(test_results=[TestResult(name='test_case_0',
success=False, metrics_data=[MetricData(name='Correctness (GEval)',
threshold=0.5, success=False, score=0.2, reason='Actual output omits the expected detail (the cat) and contradicts the expected output.', strict_mode=False, evaluation_model='local model', error=None,
evaluation_cost=0.0, verbose_logs='Criteria:\nDetermine whether the actual output is factually correct based on the expected output.\n\nEvaluation Steps:\n[\n    "Check whether the facts in \'actual output\' contradicts any facts in \'expected output\'",\n    "You should also heavily penalize omission of detail",\n    "Vague language, or contradicting OPINIONS, are OK"\n]\n'), conversational=False, multimodal=False, input='The dog chased the cat up the tree, who ran up the tree?', actual_output='It depends, some might consider the cat, while others might argue the dog.', expected_output='The cat.', context=None, retrieval_context=None)], confident_link=None)

```



---

**CHAPTER  
NINE**

---

**MAIN REFERENCE**



## BIBLIOGRAPHY

- [GenAI] Wenqiang Feng, Di Zhen. [GenAI: Best Practices](#), 2024.
- [PySpark] Wenqiang Feng. [Learning Apache Spark with Python](#), 2017.
- [lateChunking] Michael Gunther etc. [Late Chunking: Contextual Chunk Embeddings Using Long-Context Embedding Models](#), 2024.
- [selfRAG] Akari Asai etc. [Self-RAG: Learning to Retrieve, Generate, and Critique through Self-Reflection](#), 2023.
- [PEFT] Yunho Mo etc. [Parameter-Efficient Fine-Tuning Method for Task-Oriented Dialogue Systems](#), 2023.
- [fineTuneEmbedding] Philipp Schmid. [Fine-tune Embedding models for Retrieval Augmented Generation \(RAG\)](#), 2024.
- [fineTuneLLM] Maxime Labonne. [Fine-Tune Your Own Llama 2 Model in a Colab Notebook](#), 2024.
- [GEval] Yang Liu. [G-EVAL: NLG Evaluation using GPT-4 with Better Human Alignment](#), 2023.