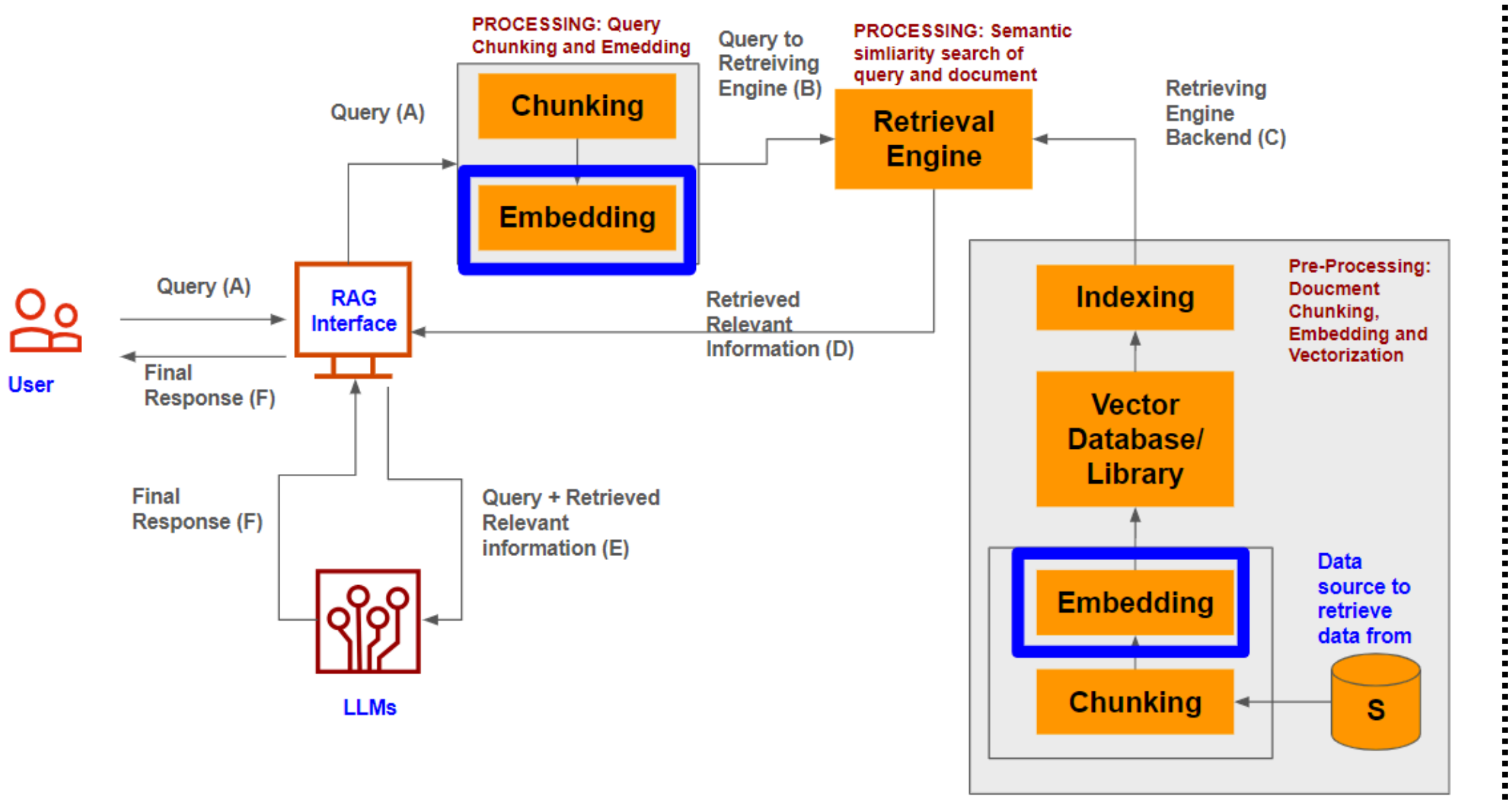
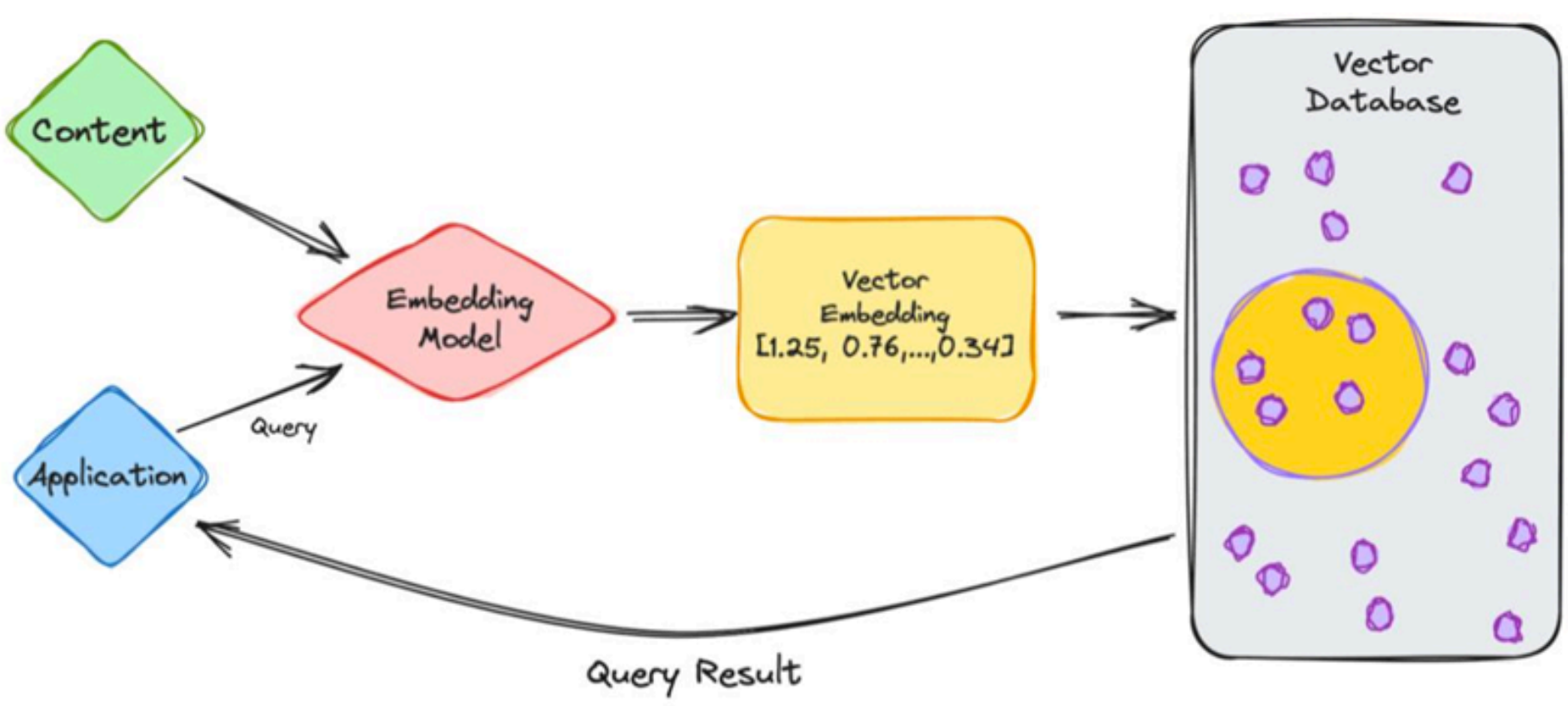
Analytics Vidhya

Mastering RAG

Different Types Embedding in RAG





1. Word-Level Embeddings

Overview

Word embeddings represent individual words as fixed-length dense vectors, capturing semantic relationships based on their co-occurrence in a large corpus.

Pros

- Efficient for basic similarity tasks.
- Works well for traditional NLP tasks (e.g., word analogy, clustering).

Cons

- Context-independent (same vector for a word in all contexts).
- Cannot handle out-of-vocabulary (OOV) words effectively.

Analytics Vidhya

Implementation Using Word2Vec

```
from gensim.models import Word2Vec


# Sample text corpus
data = [["RAG", "uses", "retrieval", "and",
"generation"],
        ["Embeddings", "help", "in",
"retrieving", "documents"]]


# Train Word2Vec Model
model = Word2Vec(sentences=data,
vector_size=100, window=5, min_count=1,
workers=4)


# Get embedding for a word
print(model.wv["retrieval"])  # Output: 100-
dimensional vector
```

2. Sentence-Level Embeddings

Overview

Sentence embeddings represent full sentences as vectors, capturing their semantic meaning.

Pros

- Useful for document retrieval and ranking.
- Captures sentence context better than word embeddings.

Cons

- Computationally heavier than word embeddings.
- Requires larger models for better accuracy.

Analytics Vidhya

Implementation Using Sentence-BERT (SBERT)

```
from sentence_transformers import
SentenceTransformer


# Load pre-trained Sentence-BERT model
model = SentenceTransformer("all-MiniLM-L6-
v2")


# Example sentences
sentences = ["RAG improves text generation
by retrieving information.",
             "Sentence embeddings help in
semantic search."]


# Compute sentence embeddings
embeddings = model.encode(sentences)
print(embeddings.shape)  # Output: (2, 384)
(2 sentences, 384-dim vectors)
```

3. Document-Level Embeddings

Overview

Document embeddings represent entire documents, considering overall topic structure and relationships between words and sentences.

Pros

- Useful for large-scale document retrieval.
- Works well for topic modeling and knowledge representation.

Cons

- Requires specialized models.
- Longer documents may need hierarchical chunking.

Implementation Using Doc2Vec

```
from gensim.models.doc2vec import Doc2Vec,
TaggedDocument


# Sample document corpus
documents = ["Retrieval-Augmented Generation
uses embeddings for retrieval.",
             "Document embeddings help in
search and ranking tasks."]


# Convert to TaggedDocument format
tagged_data =
[TaggedDocument(words=doc.split(), tags=
[str(i)]) for i, doc in
enumerate(documents)]


# Train Doc2Vec model
doc_model = Doc2Vec(tagged_data,
vector_size=100, window=5, min_count=1,
workers=4, epochs=10)


# Get document embedding
print(doc_model.dv["0"])  # Output: 100-
dimensional vector for document 0
```

4. Contextual Embeddings

Overview

Contextual embeddings (from models like BERT, RoBERTa) generate different representations for the same word depending on its context, improving retrieval accuracy.

Pros

- Captures word meaning based on surrounding context.
- More effective for complex NLP tasks.

Cons

- Requires substantial computational resources.
- Slower inference compared to static embeddings.

Implementation Using BERT

```
from transformers import AutoTokenizer,
AutoModel
import torch


# Load pre-trained BERT model
tokenizer =
AutoTokenizer.from_pretrained("bert-base-
uncased")
model = AutoModel.from_pretrained("bert-
base-uncased")


# Example text
text = "RAG models use embeddings for better
retrieval."


# Tokenize input
inputs = tokenizer(text,
return_tensors="pt")


# Generate contextual embeddings
outputs = model(**inputs)
embeddings = outputs.last_hidden_state  #
Shape: (1, sequence_length, 768)
print(embeddings.shape)
```

5. Sparse Embeddings

Overview

Sparse embeddings use traditional NLP techniques to represent text as frequency-based vectors.

Pros

- Fast and interpretable.
- Good for keyword-based search and retrieval.

Cons

- Does not capture deep semantic meaning.
- Struggles with synonyms and paraphrased text.

Implementation Using TF-IDF

```
from sklearn.feature_extraction.text import
TfidfVectorizer


# Sample documents
documents = ["RAG models enhance AI-
generated responses.",
             "TF-IDF helps in information
retrieval."]


# Compute TF-IDF vectors
vectorizer = TfidfVectorizer()
tfidf_matrix =
vectorizer.fit_transform(documents)
print(tfidf_matrix.shape)  # Output: (2,
vocab_size)
```

Different embedding techniques serve different purposes in RAG. Choosing the right type of embedding depends on the retrieval model, dataset, and computational constraints.

Comparison of Embedding Types

Embedding Type	Use Case	Pros	Cons
Word Embeddings	Basic NLP tasks	Fast, interpretable	Context-independent
Sentence Embeddings	Semantic search, retrieval	Captures sentence meaning	Computationally expensive
Document Embeddings	Large document retrieval	Retains overall topic structure	Needs specialized models
Contextual Embeddings	Complex NLP, RAG models	Captures deep meaning	High computational cost
Sparse Embeddings	Keyword-based search	Fast, interpretable	Lacks semantic understanding