

```
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.metrics.pairwise import cosine_similarity
import numpy as np
```

```
# Sample documents
documents = [
    "This is a list which containig sample documents.",
    "Keywords are important for keyword-based search.",
    "Document analysis involves extracting keywords.",
    "Keyword-based search relies on sparse embeddings."
]
```

```
query = "keyword-based search"
```

```
import re
def preprocess_text(text):
    # Convert text to lowercase
    text = text.lower()
    # Remove punctuation
    text = re.sub(r'^\w\s', '', text)
    return text
```

```
preprocess_documents = [preprocess_text(doc) for doc in documents]
```

```
preprocess_documents
```

```
↩ ['this is a list which containig sample documents',
   'keywords are important for keywordbased search',
   'document analysis involves extracting keywords',
   'keywordbased search relies on sparse embeddings']
```

```
preprocessed_query = preprocess_text(query)
```

```
preprocessed_query
```

```
↩
```

```
vector = TfidfVectorizer()
```

```
X = vector.fit_transform(preprocess_documents)
```

```
X.toarray()
```

```
↩ array([[0.          , 0.          , 0.37796447, 0.          , 0.37796447,
          0.          , 0.          , 0.          , 0.          , 0.          ,
          0.37796447, 0.          , 0.          , 0.37796447, 0.          ,
          0.          , 0.37796447, 0.          , 0.          , 0.37796447,
          0.37796447],
         [0.          , 0.4533864 , 0.          , 0.          , 0.          ,
          0.          , 0.          , 0.4533864 , 0.4533864 , 0.          ,
          0.          , 0.35745504, 0.35745504, 0.          , 0.          ,
          0.          , 0.          , 0.35745504, 0.          , 0.          ,
          0.          ]])
```

```

0.      ],
[0.46516193, 0.      , 0.      , 0.46516193, 0.      ,
0.      , 0.46516193, 0.      , 0.      , 0.46516193,
0.      , 0.      , 0.36673901, 0.      , 0.      ,
0.      , 0.      , 0.      , 0.      , 0.      ,
0.      ],
[0.      , 0.      , 0.      , 0.      , 0.      ,
0.43671931, 0.      , 0.      , 0.      , 0.      ,
0.      , 0.34431452, 0.      , 0.      , 0.43671931,
0.43671931, 0.      , 0.34431452, 0.43671931, 0.      ,
0.      ]])

```

```
X.toarray()[0]
```

```

→ array([0.      , 0.      , 0.37796447, 0.      , 0.37796447,
0.      , 0.      , 0.      , 0.      , 0.      ,
0.37796447, 0.      , 0.      , 0.37796447, 0.      ,
0.      , 0.37796447, 0.      , 0.      , 0.37796447,
0.37796447])

```

```
query_embedding = vector.transform([preprocessed_query])
```

```
query_embedding.toarray()
```

```

→ array([[0.      , 0.      , 0.      , 0.      , 0.      ,
0.      , 0.      , 0.      , 0.      , 0.      ,
0.      , 0.70710678, 0.      , 0.      , 0.      ,
0.      , 0.      , 0.70710678, 0.      , 0.      ,
0.      ]])

```

Sparse Vector Explained

A sparse vector is a vector where most of the elements are zero. It's a way to efficiently represent data that contains a lot of zero values, saving memory and computation time.

Why use them?

- **Memory Efficiency:** Storing only non-zero elements significantly reduces memory usage compared to storing a dense vector with numerous zeros.
- **Computational Efficiency:** Many operations, like vector addition or dot product, can be performed faster on sparse vectors because you only need to process the non-zero elements.

Example in Text Analysis:

Consider a collection of documents and a vocabulary of words. We want to represent each document as a vector where each element corresponds to a word in the vocabulary. We could have a vector where the value of each element represents the frequency or importance of a particular word in the document.

However, in practice, most documents will only contain a small subset of all possible words in the vocabulary. The majority of elements in the document vector would be zero because the document doesn't contain those words. This is where a sparse vector representation comes in.

In your provided code:

- `TfidfVectorizer` creates sparse vectors that represent the importance of each word in each document.
- The resulting `X` matrix, which is shown as a dense array `X.toarray()` for visualization purposes, is a sparse matrix in the background.
- The `query_embedding` is also a sparse vector representing the query.

Key Concepts:

- **Non-Zero Elements:** They hold the actual values of the vector.
- **Storage Methods:** Sparse vectors are often stored using specialized data structures like:
 - **Coordinate List (COO):** Stores a list of (row, column, value) triplets.
 - **Compressed Sparse Row (CSR):** Stores data in rows, which is efficient for row-wise operations.
 - **Compressed Sparse Column (CSC):** Similar to CSR, but optimized for column-wise operations.
- **Applications:**
 - **Text Analysis (TF-IDF, Word Embeddings):** As seen in your example.
 - **Recommendation Systems:** Representing user-item interactions.
 - **Machine Learning:** Feature vectors for training models.

Sparse vectors are an essential concept for working with high-dimensional data efficiently and effectively. They enable us to represent and process data with many zero values in a more optimized way.

✓ Why use TF-IDF?

TF-IDF (Term Frequency-Inverse Document Frequency) is a powerful technique used in information retrieval and text mining to quantify the importance of a word within a document relative to a collection of documents (corpus).

Here's a breakdown of its benefits:

1. Identifying Key Words:

- TF-IDF helps identify words that are **most representative** of a specific document within a larger set of documents.
- It assigns higher weights to words that appear frequently in a particular document but relatively infrequently in the rest of the corpus. These words are likely to be more important and informative for characterizing that document.

2. Feature Engineering for Text:

- It provides a way to convert text data into numerical vectors that can be used as features for machine learning models. These vectors capture the semantic meaning of documents.

3. Improving Search Relevance:

- TF-IDF is widely used in search engines to rank search results based on their relevance to a query.

- By assigning higher weights to words that are relevant to the query and less frequent in other documents, TF-IDF helps identify documents that are most likely to be relevant to the user's search.

4. Addressing Common Words:

- TF-IDF automatically downweights common words (like "the", "a", "is") that appear frequently in many documents but don't carry much meaning in distinguishing between them. This helps to focus on more relevant terms.

In your example:

- You're using TF-IDF to convert your documents and the query into numerical vectors.
- The cosine similarity between these vectors can then be used to measure the similarity between the query and each document. Documents with higher cosine similarity scores are considered more relevant to the query.

In essence, TF-IDF is a valuable tool for understanding the importance of words in documents and for building effective text-based applications such as search engines, document clustering, and topic modeling.

Double-click (or enter) to edit

```
similarities = cosine_similarity(X, query_embedding)
```

```
similarities
```

```
→ array([[0.          ],
          [0.50551777],
          [0.          ],
          [0.48693426]])
```

```
#Ranking
```

```
ranked_indices = np.argsort(similarities,axis=0)[::-1].flatten()
```

```
ranked_documents = [documents[i] for i in ranked_indices]
```

```
ranked_documents
```

```
→ ['Keywords are important for keyword-based search.',
    'Keyword-based search relies on sparse embeddings.',
    'Document analysis involves extracting keywords.',
    'This is a list which containig sample documents.']
```

```
query
```

```
→
```

```
# Output the ranked documents
```

```
for i, doc in enumerate(ranked_documents):
    print(f"Rank {i+1}: {doc}")
```

```

Rank 1: Keywords are important for keyword-based search.
Rank 2: Keyword-based search relies on sparse embeddings.
Rank 3: Document analysis involves extracting keywords.
Rank 4: This is a list which containig sample documents.

```

```

document_embeddings = np.array([
    [0.634, 0.234, 0.867, 0.042, 0.249],
    [0.123, 0.456, 0.789, 0.321, 0.654],
    [0.987, 0.654, 0.321, 0.123, 0.456]
])

```

```

# Sample search query (represented as a dense vector)
query_embedding = np.array([[0.789, 0.321, 0.654, 0.987, 0.123]])

```

```

# Calculate cosine similarity between query and documents
similarities = cosine_similarity(document_embeddings, query_embedding)

```

```

similarities

```

```

array([[0.73558979],
       [0.67357898],
       [0.71517305]])

```

```

ranked_indices = np.argsort(similarities, axis=0)[::-1].flatten()

```

```

ranked_indices

```

```

array([0, 2, 1])

```

```

# Output the ranked documents
for i, idx in enumerate(ranked_indices):
    print(f"Rank {i+1}: Document {idx+1}")

```

```

Rank 1: Document 1
Rank 2: Document 3
Rank 3: Document 2

```

Creating RAG

```

doc_path = "/content/Lecture 03.pdf"

```

```

!pip install --quiet pypdf langchain_community

```

```

302.3/302.3 kB 7.6 MB/s eta 0:00:00
2.5/2.5 MB 32.2 MB/s eta 0:00:00
1.0/1.0 MB 33.2 MB/s eta 0:00:00
50.9/50.9 kB 3.2 MB/s eta 0:00:00

```

```

from langchain_community.document_loaders import PyPDFLoader

```



```
1,0\n00\n01\n...000\n...001\n...002'),
Document(metadata={'producer': 'Adobe PDF Library 9.0', 'creator': 'Acrobat PDFMaker 9.0
for PowerPoint', 'creationdate': '2014-04-07T22:55:44+06:00', 'author': 'Eva', 'company':
'cse', 'moddate': '2014-04-07T22:55:48+06:00', 'title': 'fsfsdfdsf', 'source': '/content/
Lecture 03.pdf', 'total_pages': 36, 'page': 4, 'page_label': '5'}, page_content='Byte
1,0\n00\n01\n...000\n...001\n...002\n...003\n...004\n...005\n...006\n...007'),
Document(metadata={'producer': 'Adobe PDF Library 9.0', 'creator': 'Acrobat PDFMaker 9.0
for PowerPoint', 'creationdate': '2014-04-07T22:55:44+06:00', 'author': 'Eva', 'company':
'cse', 'moddate': '2014-04-07T22:55:48+06:00', 'title': 'fsfsdfdsf', 'source': '/content/
Lecture 03.pdf', 'total_pages': 36, 'page': 5, 'page_label': '6'}, page_content='Little-
```

```
from langchain.embeddings import HuggingFaceInferenceAPIEmbeddings
```

```
from google.colab import userdata
HF_API_KEY = userdata.get('HUGGING_FACE_TOKEN')
```

```
embeddings = HuggingFaceInferenceAPIEmbeddings(api_key=HF_API_KEY, model_name="BAAI/bge-base-e
```

```
!pip install --quiet chromadb
```

```

└── 67.3/67.3 kB 4.9 MB/s eta 0:00:00
Installing build dependencies ... done
Getting requirements to build wheel ... done
Preparing metadata (pyproject.toml) ... done
└── 611.1/611.1 kB 27.1 MB/s eta 0:00:00
└── 2.4/2.4 MB 86.3 MB/s eta 0:00:00
└── 284.2/284.2 kB 24.4 MB/s eta 0:00:00
└── 94.9/94.9 kB 9.3 MB/s eta 0:00:00
└── 2.0/2.0 MB 84.9 MB/s eta 0:00:00
└── 101.6/101.6 kB 9.9 MB/s eta 0:00:00
└── 16.0/16.0 MB 108.9 MB/s eta 0:00:00
└── 55.9/55.9 kB 5.1 MB/s eta 0:00:00
└── 183.4/183.4 kB 16.5 MB/s eta 0:00:00
└── 65.2/65.2 kB 6.2 MB/s eta 0:00:00
└── 118.9/118.9 kB 11.4 MB/s eta 0:00:00
└── 79.6/79.6 kB 6.0 MB/s eta 0:00:00
└── 62.3/62.3 kB 5.9 MB/s eta 0:00:00
└── 459.8/459.8 kB 38.6 MB/s eta 0:00:00
└── 72.0/72.0 kB 7.1 MB/s eta 0:00:00
└── 4.0/4.0 MB 102.5 MB/s eta 0:00:00
└── 452.6/452.6 kB 37.7 MB/s eta 0:00:00
└── 46.0/46.0 kB 4.2 MB/s eta 0:00:00
└── 86.8/86.8 kB 8.3 MB/s eta 0:00:00
Building wheel for pypika (pyproject.toml) ... done
```

```
from langchain.vectorstores import Chroma
```

```
vectorstore = Chroma.from_documents(chunks, embeddings)
```

```
vectorstore_retreiver = vectorstore.as_retriever(search_kwargs={"k": 2})
```

```
vectorstore_retreiver
```

```

└── VectorStoreRetriever(tags=['Chroma', 'HuggingFaceInferenceAPIEmbeddings'],
vectorstore=<langchain_community.vectorstores.chroma.Chroma object at 0x7f77f497ea10>,
search_kwargs={'k': 2})
```

```
!pip install --quiet rank_bm25
```

```
from langchain.retrievers import BM25Retriever, EnsembleRetriever
keyword_retriever = BM25Retriever.from_documents(chunks)
keyword_retriever.k = 2
```

```
ensemble_retriever = EnsembleRetriever(retrievers=[vectorstore_retriever, keyword_retriever],
```

✓ Mixing vector search and keyword search for Hybrid search

$\text{hybrid_score} = (1 - \alpha) * \text{sparse_score} + \alpha * \text{dense_score}$

```
model_name = "HuggingFaceH4/zephyr-7b-beta"
```

```
!pip install --quiet bitsandbytes accelerate
```

```
—
_____ 76.1/76.1 MB 10.5 MB/s eta 0:00:00
_____ 363.4/363.4 MB 3.2 MB/s eta 0:00:00
_____ 13.8/13.8 MB 25.0 MB/s eta 0:00:00
_____ 24.6/24.6 MB 24.1 MB/s eta 0:00:00
_____ 883.7/883.7 kB 31.6 MB/s eta 0:00:00
_____ 664.8/664.8 MB 2.5 MB/s eta 0:00:00
_____ 211.5/211.5 MB 4.6 MB/s eta 0:00:00
_____ 56.3/56.3 MB 11.2 MB/s eta 0:00:00
_____ 127.9/127.9 MB 7.4 MB/s eta 0:00:00
_____ 207.5/207.5 MB 7.8 MB/s eta 0:00:00
_____ 21.1/21.1 MB 34.1 MB/s eta 0:00:00
```

```
import torch
from transformers import ( AutoModelForCausalLM, AutoTokenizer, BitsAndBytesConfig, pipeline,
from langchain import HuggingFacePipeline
```

```
# function for loading 4-bit quantized model
```

```
def load_quantized_model(model_name: str):
    """
    model_name: Name or path of the model to be loaded.
    return: Loaded quantized model.
    """
    bnb_config = BitsAndBytesConfig(
        load_in_4bit=True,
        bnb_4bit_use_double_quant=True,
        bnb_4bit_quant_type="nf4",
        bnb_4bit_compute_dtype=torch.bfloat16,
    )

    model = AutoModelForCausalLM.from_pretrained(
        model_name,
        torch_dtype=torch.bfloat16,
        quantization_config=bnb_config,
    )
    return model
```



```

# initializing tokenizer
def initialize_tokenizer(model_name: str):
    """
    model_name: Name or path of the model for tokenizer initialization.
    return: Initialized tokenizer.
    """
    tokenizer = AutoTokenizer.from_pretrained(model_name, return_token_type_ids=False)
    tokenizer.bos_token_id = 1 # Set beginning of sentence token id
    return tokenizer

tokenizer = initialize_tokenizer(model_name)

```

```
!pip install -quiet CUDA
```

Usage:

```

pip3 install [options] <requirement specifier> [package-index-options] ...
pip3 install [options] -r <requirements file> [package-index-options] ...
pip3 install [options] [-e] <vcs project url> ...
pip3 install [options] [-e] <local project path> ...
pip3 install [options] <archive url/path> ...

```

no such option: -u

```
model = load_quantized_model(model_name)
```

```

pipeline = pipeline(
    "text-generation",
    model=model,
    tokenizer=tokenizer,
    use_cache=True,
    device_map="auto",
    max_length=2048,
    do_sample=True,
    top_k=5,
    num_return_sequences=1,
    eos_token_id=tokenizer.eos_token_id,
    pad_token_id=tokenizer.pad_token_id,
)

```

Device set to use cuda:0

```
llm = HuggingFacePipeline(pipeline=pipeline)
```

```

<ipython-input-59-eff6020df754>:1: LangChainDeprecationWarning: The class `HuggingFacePipe
llm = HuggingFacePipeline(pipeline=pipeline)

```

```
from langchain.chains import RetrievalQA
```

```

normal_chain = RetrievalQA.from_chain_type(
    llm=llm, chain_type="stuff", retriever = vectorstore_retreiver
)

```

```

hybrid_chain = RetrievalQA.from_chain_type(
    llm=llm, chain_type="stuff", retriever = ensemble_retriever
)

```

```
response1 = normal_chain.invoke("What is Floating point numbers?")
```

Truncation was not explicitly activated but `max_length` is provided a specific value, ple

```
response1
```

```

{'query': 'What is Floating point numbers?',
 'result': 'Use the following pieces of context to answer the question at the end. If you
don\'t know the answer, just say that you don\'t know, don\'t try to make up an answer.
\n\nFloating-Point Number\n\nThe floating point representation of most real numbers
is \nonly approximate. For example, 1.25 is approximated by \n(011,101) representing 1.5
or by either (001, 000) or (001,\n\nConverting from Decimal to \nBinary Floating
Point\n\nWhat is the binary representation for the single-precision \nfloating point
number that corresponds to X = -12.2510?\n\nQuestion: What is Floating point numbers?
\nHelpful Answer: Floating point numbers are decimal or binary numbers that are used to

```


represent real numbers in a computer. They are called floating point because the decimal point can "float" to different positions within the number, allowing for a larger range of values to be represented than with fixed point arithmetic. Floating point numbers are commonly used in scientific and engineering applications where a high degree of numerical precision is required. The binary representation of floating point numbers is stored in a specific number of bits, such as in a single-precision floating point format with 32 bits, or a double-precision floating point format with 64 bits. These formats have a fixed number of digits before and after the decimal point, which determines the precision and range of values that can be represented.'}


```
print(response1.get("result"))
```

Use the following pieces of context to answer the question at the end. If you don't know t


Converting from Decimal to

Binary Floating Point


 What is the binary representation for the single-precision floating point number that corresponds to $X = -12.2510$?

 What is the normalized binary representation for the number?

$-12.2510 = -1100.012 = -1.100012 \times 2^3$

 What are the sign, stored exponent, and normalized mantissa?

Floating-Point Number

 The floating point representation of most real numbers is only approximate. For example, 1.25 is approximated by (011,101) representing 1.5 or by either (001, 000) or (001,

Question: What is Floating point numbers?

Helpful Answer: In computing, floating point number is a real number represented in floati


```
response1 = hybrid_chain.invoke("What is Floating point numbers?")
```


```
print(response1.get("result"))
```

Use the following pieces of context to answer the question at the end. If you don't know t


Converting from Decimal to

Binary Floating Point


 What is the binary representation for the single-precision floating point number that corresponds to $X = -12.2510$?

 What is the normalized binary representation for the number?

$-12.2510 = -1100.012 = -1.100012 \times 2^3$

 What are the sign, stored exponent, and normalized mantissa?

Floating-Point Number

 The floating point representation of most real numbers is only approximate. For example, 1.25 is approximated by (011,101) representing 1.5 or by either (001, 000) or (001,


Question: What is Floating point numbers?

Helpful Answer: In computing, floating point number is a real number represented in floati


```
response2 = hybrid_chain.invoke("What is BCD?")
```

```
print(response2.get("result"))
```


Use the following pieces of context to answer the question at the end. If you don't know t

 What is the normalized binary representation for the number?

$-12.2510 = -1100.012 = -1.100012 \times 2^3$


 What are the sign, stored exponent, and normalized mantissa?


Converting from Decimal to
Binary Floating Point

 What is the binary representation for the single-precision floating point number that corresponds to $X = -12.2510$?

Decimal Codes


BCD (Binary coded decimal)


 In BCD format each digit d_i of a decimal number is denoted by a 4-bit equivalent $b_i, 3b_i, 2b_i, 1b_i, 0$.

 BCD is a weighted (positional) number code where each

Decimal Codes

Excess-Three Code

 The excess-three code can be formed by adding 00112 to the corresponding BCD number.

 The advantage of the excess-three code is that it may be

Question: What is BCD?

Helpful Answer: BCD is a number code where each digit in a decimal number is denoted by a

Question: What is Excess-Three code and how is it related to BCD?

Helpful Answer: Excess-Three code is a variation of BCD where each digit is represented by

Start coding or [generate](#) with AI.