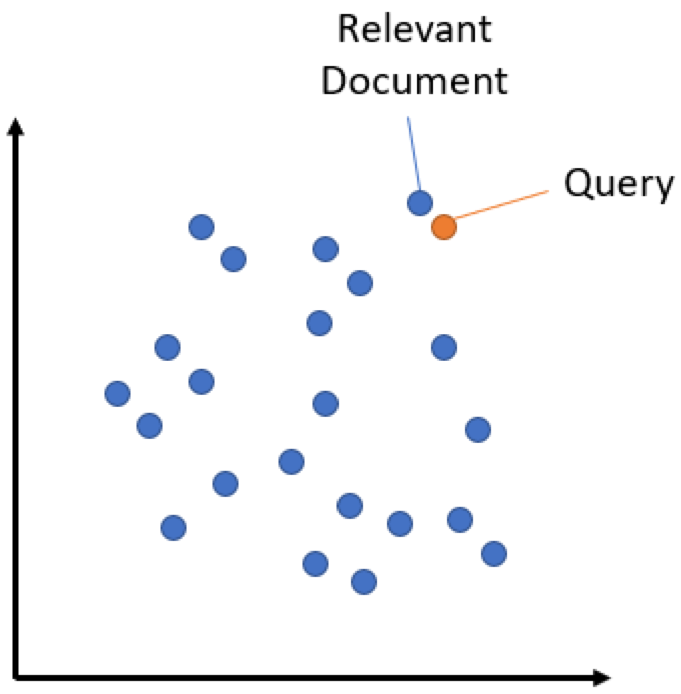
Semantic search seeks to improve search accuracy by understanding the semantic meaning of the search query and the corpus to search over. Semantic search can also perform well given synonyms, abbreviations, and misspellings, unlike keyword search engines that can only find documents based on lexical matches.

Background

The idea behind semantic search is to embed all entries in your corpus, whether they be sentences, paragraphs, or documents, into a vector space. At search time, the query is embedded into the same vector space and the closest embeddings from your corpus are found. These entries should have a high semantic similarity with the query.



Symmetric vs. Asymmetric Semantic Search

A **critical distinction** for your setup is *symmetric* vs. *asymmetric semantic search*:

- For **symmetric semantic search** your query and the entries in your corpus are of about the same length and have the same amount of content. An example would be searching for similar questions: Your query could for example be *"How to learn Python online?"* and you want to find an entry like *"How to learn Python on the web?"*. For symmetric tasks, you could potentially flip the query and the entries in your corpus.
    - Related training example: Quora Duplicate Questions.

not make sense.

- Related training example: MS MARCO
- Suitable models: Pre-Trained MS MARCO Models

It is critical **that you choose the right model** for your type of task.

Manual Implementation

For small corpora (up to about 1 million entries), we can perform semantic search with a manual implementation by computing the embeddings for the corpus as well as for our query, and then calculating the semantic textual similarity using `SentenceTransformer.similarity`. For a simple example, see semantic_search.py:

Output

```
This script outputs for various queries the top 5
most similar sentences in the corpus.
"""

import torch

from sentence_transformers import
SentenceTransformer

embedder = SentenceTransformer("all-MiniLM-L6-v2")

# Corpus with example sentences
corpus = [
    "A man is eating food.",
    "A man is eating a piece of bread.",
    "The girl is carrying a baby.",
    "A man is riding a horse.",
    "A woman is playing violin.",
    "Two men pushed carts through the woods.",
    "A man is riding a white horse on an enclosed
ground.",
    "A monkey is playing drums.",
    "A cheetah is running behind its prey.",
]
# Use "convert_to_tensor=True" to keep the tensors
on GPU (if available)
corpus_embeddings = embedder.encode(corpus,
convert_to_tensor=True)

# Query sentences:
queries = [
    "A man is eating pasta.",
    "Someone in a gorilla costume is playing a set
of drums.",
    "A cheetah chases prey on across a field.",
]

# Find the closest 5 sentences of the corpus for
each query sentence based on cosine similarity
top_k = min(5, len(corpus))
for query in queries:
    query_embedding = embedder.encode(query,
convert_to_tensor=True)

    # We use cosine-similarity and torch.topk to
find the highest 5 scores
    similarity_scores =
embedder.similarity(query_embedding,
corpus_embeddings)[0]
    scores, indices = torch.topk(similarity_scores,
k=top_k)

    print("\nQuery:", query)
    print("Top 5 most similar sentences in
```

```
A man is riding a horse.
(Score: 0.1889)
A man is riding a white
horse on an enclosed
ground. (Score: 0.1047)
A cheetah is running behind
its prey. (Score: 0.0980)

Query: Someone in a gorilla
costume is playing a set of
drums.
Top 5 most similar
sentences in corpus:
A monkey is playing drums.
(Score: 0.6433)
A woman is playing violin.
(Score: 0.2564)
A man is riding a horse.
(Score: 0.1389)
A man is riding a white
horse on an enclosed
ground. (Score: 0.1191)
A cheetah is running behind
its prey. (Score: 0.1080)

Query: A cheetah chases
prey on across a field.
Top 5 most similar
sentences in corpus:
A cheetah is running behind
its prey. (Score: 0.8253)
A man is eating food.
(Score: 0.1399)
A monkey is playing drums.
(Score: 0.1292)
A man is riding a white
horse on an enclosed
ground. (Score: 0.1097)
A man is riding a horse.
(Score: 0.0650)
```

```
    hits = hits[0]       #Get the hits for the first qu
    for hit in hits:
        print(corpus[hit['corpus_id']], "(Score: {:.4
    """
```

Optimized Implementation

Instead of implementing semantic search by yourself, you can use the `util.semantic_search` function.

The function accepts the following parameters:

`sentence_transformers.util.semantic_search`(*query_embeddings: torch.Tensor, corpus_embeddings: torch.Tensor, query_chunk_size: int = 100, corpus_chunk_size: int = 500000, top_k: int = 10, score_function: Callable[[torch.Tensor, torch.Tensor], torch.Tensor] = <function cos_sim>*) → List[List[Dict[str, Union[int, float]]]] [source]

This function performs a cosine similarity search between a list of query embeddings and a list of corpus embeddings. It can be used for Information Retrieval / Semantic Search for corpora up to about 1 Million entries.

Parameters:

- **query_embeddings** (*Tensor*) – A 2 dimensional tensor with the query embeddings.
- **corpus_embeddings** (*Tensor*) – A 2 dimensional tensor with the corpus embeddings.
- **query_chunk_size** (*int*, *optional*) – Process 100 queries simultaneously. Increasing that value increases the speed, but requires more memory. Defaults to 100.
- **corpus_chunk_size** (*int*, *optional*) – Scans the corpus 100k entries at a time. Increasing that value increases the speed, but requires more memory. Defaults to 500000.
- **top_k** (*int*, *optional*) – Retrieve top k matching entries. Defaults to 10.
- **score_function** (*Callable[[Tensor, Tensor], Tensor]*, *optional*) – Function for computing scores. By default, cosine similarity.

Returns: A list with one entry for each query. Each entry is a list of dictionaries with the keys 'corpus_id' and 'score', sorted by decreasing cosine similarity scores.

Return type: List[List[Dict[str, Union[int, float]]]] `query_embeddings` as well as the `corpus_embeddings` on the same GPU-device. This significantly boost the performance. Further, we can normalize the corpus embeddings so that each corpus embeddings is of length 1. In that case, we can use dot-product for computing scores.

```
corpus_embeddings = corpus_embeddings.to("cuda")
corpus_embeddings = util.normalize_embeddings(corpus_embeddings)

query_embeddings = query_embeddings.to("cuda")
query_embeddings = util.normalize_embeddings(query_embeddings)
hits = util.semantic_search(query_embeddings, corpus_embeddings,
score_function=util.dot_score)
```

Elasticsearch

Elasticsearch has the possibility to index dense vectors and to use them for document scoring. We can easily index embedding vectors, store other data alongside our vectors and, most importantly, efficiently retrieve relevant entries using approximate nearest neighbor search (HNSW, see also below) on the embeddings.

For further details, see semantic_search_quora_elasticsearch.py.

Approximate Nearest Neighbor

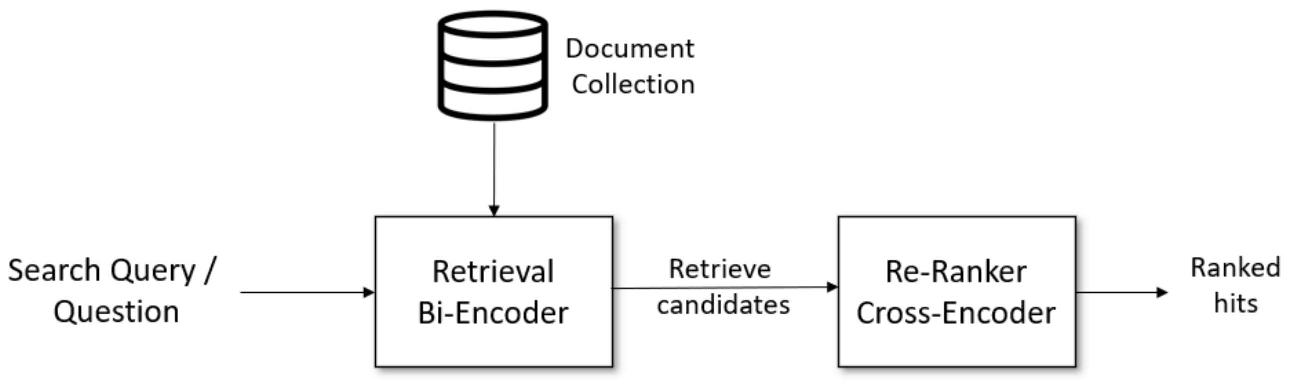
Searching a large corpus with millions of embeddings can be time-consuming if exact nearest neighbor search is used (like it is used by `util.semantic_search`).

In that case, Approximate Nearest Neighbor (ANN) can be helpful. Here, the data is partitioned into smaller fractions of similar embeddings. This index can be searched efficiently and the embeddings with the highest similarity (the nearest neighbors) can be retrieved within milliseconds, even if you have millions of vectors. However, the results are not necessarily exact. It is possible that some vectors with high similarity will be missed.

For all ANN methods, there are usually one or more parameters to tune that determine the recall-speed trade-off. If you want the highest speed, you have a high chance of missing hits. If you want high recall, the search speed decreases.

Three popular libraries for approximate nearest neighbor are Annoy, FAISS, and hnswlib.

For complex semantic search scenarios, a two-stage retrieve & re-rank pipeline is advisable:



For further details, see Retrieve & Re-rank.

Examples

We list a handful of common use cases:

Similar Questions Retrieval

semantic_search_quora_pytorch.py [Colab version] shows an example based on the Quora duplicate questions dataset. The user can enter a question, and the code retrieves the most similar questions from the dataset using the `util.semantic_search` method. As model, we use distilbert-multilingual-nli-stsb-quora-ranking, which was trained to identify similar questions and supports 50+ languages. Hence, the user can input the question in any of the 50+ languages. This is a **symmetric search task**, as the search queries have the same length and content as the questions in the corpus.

Similar Publication Retrieval

semantic_search_publications.py [Colab version] shows an example how to find similar scientific publications. As corpus, we use all publications that have been presented at the EMNLP 2016 - 2018 conferences. As search query, we input the title and abstract of more recent publications and find related publications from our copurs. We use the SPECTER model. This is a **symmetric search task**, as the paper in the corpus consists of title & abstract and we search for title & abstract.

Wikipedia so that it fits easily into memory.

retrieve_rerank_simple_wikipedia.ipynb [Colab Version]: This script uses the Retrieve & Re-rank strategy and is an example for an **asymmetric search task**. We split all Wikipedia articles into paragraphs and encode them with a bi-encoder. If a new query / question is entered, it is encoded by the same bi-encoder and the paragraphs with the highest cosine-similarity are retrieved. Next, the retrieved candidates are scored by a Cross-Encoder re-ranker and the 5 passages with the highest score from the Cross-Encoder are presented to the user. We use models that were trained on the MS Marco Passage Reranking dataset, a dataset with about 500k real queries from Bing search.