colbert-kit Package Documentation

Thuong Dang, Qiqi Chen

April 2025

Contents

1 Introduction

This is documentation for our work on ColBERT. In this document, we provide a detailed introduction to ColBERT and the functionalities of our ColBERT package. The documentation is structured as follows:

In the first section, we briefly review ColBERT and its MaxSim scoring function. In the second section, we discuss our indexing and retrieval methods for ColBERT, including the inputs and outputs used in our package. The third section covers the re-ranking functionality. In the fourth section, we explain how to train a ColBERT model and evaluate the training using standard information retrieval (IR) datasets. Our package includes training and evaluation scripts to compare the performance of ColBERT with BM25.

Finally, we address the general difficulty of selecting effective negatives for training or fine-tuning ranking models. To this end, we have developed several methods for negative sampling, which are also included in our package.

2 ColBERT and MaxSim score

ColBERT is a dense retrieval model that uses multiple embeddings per query and document to enable fine-grained similarity matching between their tokens.

Let q, d be a query, and document, respectively. Let $[q_1 \cdots q_m]$ be the embeddings of tokens of q, and $[d_1 \cdots d_n]$ be the embeddings of tokens of d. The *MaxSim score* is defined as follows

$$\text{MaxSim}(q, d) = \sum_{i=1}^{m} \max_j S(q_i, d_j),$$

where $S(q_i, d_j)$ is the cosine similarity between the token q_i and d_j. As an intuition, this can be seen as a soft extension of keyword matching: if we define $S(q_i, d_j) = 1$ if $q_i = d_j$ and 0 otherwise, then the MaxSim score becomes a ranking function based on keyword matching. It means the more keyword matching, the higher ranking a document will have. In ColBERT, this hard match is replaced by semantic similarity, allowing tokens with similar meaning to contribute to the relevance score.

3 Indexing

To index with our ColBERT package, we need to define

```
# Place and name where we want to save the token to doc map
token_to_doc_map_name = 'token_to_doc_map.npy'
token_to_doc_map_path = './indices_test/'
# Place and name for saving the document dictionary
# This information is for computing ColBERT scores between a
    query and a document after we retrieve the nearest
    tokens
doc_dict_name = 'documents.json'
doc_dict_path = './indices_test/'
# We can load the model from huggingface or local
embedding_model_path = './model/colbert'
```

We then initialize the method for embedding and create index

```
from colbert_kit.embedding.colbert_embedding import
    colBERTEmbedding

    colbertEmbed = colBERTEmbedding(
    model_name_or_path=embedding_model_path,
    token_to_doc_map_name=token_to_doc_map_name,
    doc_dict_name=doc_dict_name,
    token_to_doc_map_path=token_to_doc_map_path,
    doc_dict_path=doc_dict_path,
    device="cpu"
)

```

```
# texts is a list of string (or a list of our documents for retrieving)
embeddings = colbertEmbed.encode(texts) #output: numpy.ndarray

# We then to create FAISS index to search for nearest tokens
# Place and name to save the FAISS index
index_output_path = './indices_test/'
index_name = 'colbert_test_index.index'
dimension = embeddings.shape[1]

# Create an HNSW index
# There are three options: flat, IVF and HNSW
hnsw_index = HNSWFaissIndex(index_name=index_name, dimension=dimension, use_gpu=False, index_output_path=index_output_path)
hnsw_index.create_index(embeddings)
```

4 Retrieving

To retrieve, the main steps are

- Loading token to doc map, FAISS index, the document dictionary
- For an input query, use ColBERT model to have token embeddings of the query
- For each query's token embedding, retrieve top k' nearest tokens
- Map them back to original documents and compute ColBERT MaxSim scores
- Sort the results, and return the candidate documents

```
from colbert_kit.indexing.faiss.flat_faiss_index import FlatFaissIndex
from colbert_kit.indexing.faiss.HNSW_faiss_index import HNSWFaissIndex
from colbert_kit.indexing.faiss.IVF_faiss_index import IVFFaissIndex

    colbertSearchFaiss = colBERTSearchFaiss(
    model_name_or_path=embedding_model_path,
    index_path=index_output_path,
    index_name=index_name,
    index_type='HNSW',
    use_gpu=False,
    token_to_doc_map_name=token_to_doc_map_name,
    doc_dict_name=doc_dict_name,
```

```
    token_to_doc_map_path=token_to_doc_map_path,
    doc_dict_path=doc_dict_path,
    device="cpu"
)

colbertSearchFaiss.load_index_map_and_dict()

# Search
scores, colbert_indices  = colbertSearchFaiss.
    colbert_search_results(query, top_k_results=5,
    top_k_search_tokens=10)
```

5 Re-ranking

Re-ranking with ColBERT in our package are more straight-forward. The inputs for this is quite simple, we just need to define document candidates, document ids, batch size and number of top results.

```
from colbert_kit.reranking.colbert_batch_reranker import
    colBERTReRankerBatch

embedding_model_path = "./model/colbert"
batchreranker = colBERTReRankerBatch(model_name_or_path=
    embedding_model_path, device='cpu')
```

We can then define our query and candidate documents and candidate ids:

```
    query = "Was ist die Hauptstadt von Frankreich?"

doc_candidates = [
    "Paris ist die Hauptstadt von Frankreich.",
    "Berlin ist die Hauptstadt von Deutschland.",
    "Madrid ist die Hauptstadt von Spanien.",
    "Rom ist die Hauptstadt von Italien.",
    "Der Eiffelturm befindet sich in Paris."
    ]

    candidate_idx = list(range(len(doc_candidates)))
```

And then, finally, re-rank with ColBERT for top results

```
top_scores, top_indices = batchreranker.reranker(query,
    doc_candidates, candidate_idx, batch_size=8, top_n=10)
print("Top Scores:", top_scores)
print("Top Indices:", top_indices)
for id in top_indices:
    print(doc_candidates[id])
```

6 Training

To train a ColBERT model, we follow the training loss proposed in the original paper. It is the softmax-cross-entropy loss on pairwise triplets. Given a triplet $[q, d^+, d^-]$, where q is a query, d^+ is a positive document, and d^- is a negative document. The loss function is defined as follows

$$L = -\log \frac{\exp \text{MaxSim}(q, d^+)}{\exp \text{MaxSim}(q, d^+) + \exp \text{MaxSim}(q, d^-)}$$

The training script is simple, for really scaled dataset like MSMARCO, saving checkpoints and evaluating after each epoch is important. Therefore, we also included script to save and train from checkpoints.

```python
import pandas as pd
import torch
from colbert_kit.training import ColBERTTrainer

# Create simple test dataset
test_data = [
    ("A man is playing a guitar.", "A person is playing a musical instrument.", "A bird is flying in the sky."),
    ("A woman is cooking food.", "A person is preparing a meal.", "A car is driving down the road."),
    ("A child is reading a book.", "A young person is engaged with a novel.", "The sun is setting behind the mountains.")
]
# Or we can train with triplet dataset with three columns: sentence, positive_sentence and negative_sentence

# test_df = pd.DataFrame(test_data, columns=["sentence", "positive_sentence", "negative_sentence"])
# test_df.to_csv("data_test/test_triplets.csv", sep='\t', index=False)

# Load BERT model for training
bert_model_name = "model/bert"
device = 'cuda' if torch.cuda.is_available() else 'cpu'

trainer = ColBERTTrainer(
    model_name_or_path=bert_model_name,
    device=device,
    triplets=test_data,
    # Or define triplet data from a csv file as above
    # triplets_path="data_test/test_triplets.csv",
    batch_size=2,
    lr=2e-5,
    num_epochs=4,
```

```
    checkpoint_dir="./test_checkpoints/"
)
trainer.train()
trainer.save_model("./model/saved_model_test/")
```

We can also reload models from checkpoints and continue the trianing process.

```
trainer = ColBERTTrainer(
    model_name_or_path=bert_model_name,
    triplets=test_data,
    # triplets_path="data_test/test_triplets.csv",
    device = device,
    batch_size=256,
    lr=2e-5,
    num_epochs=3,
    checkpoint_dir="./checkpoints/",
    checkpoint_path="./checkpoints/checkpoint_epoch_2.pth"
)
trainer.train()
trainer.save_model("./model/saved_model_test_3/")
```

7 Evaluating

To evaluate information retrieval (IR) metrics, we need standard IR datasets. They consist of three components

- queries component: a dataset with two columns query_id and text
- documents component: a dataset with two columns doc_id and text
- qrel component: a dataset with three columns query_id, doc_id and relevant_score

To compute IR metric for our retrieving method, we need to

- create an index from documents component for retrieving
- for each query_id in qrel component, map it back to the original query, and then retrieve from document index, collect all the results and compute metrics.

For more detail, the reader can take a look on our sample evaluation codes [...]. Here are parts of the script. For inputs of our IR evaluation, we need no more than a data frame of results and qrel component.

```
from colbert_kit.evaluating.ir_eval import IREvaluation
# Computing IR metrics for BM25

results = []
```

```
query_ids = df_qrels['query_id'].unique()
top_k = 50

for query_id in tqdm(query_ids, desc="Processing queries"):
    query_row = df_queries.loc[df_queries['query_id'] ==
        query_id, text_column]
    if query_row.empty:
        raise ValueError(f"Query ID {query_id} not found in
            df_queries.")
    query = query_row.values[0]

    # Get top document indices from BM25
    bm25_scores, bm25_top_idx = BM25search.search(query,
        top_k)

    top_ranked_ids = []
    for idx in bm25_top_idx:
        idx = int(idx)
        top_ranked_ids.append(df_docs.iloc[idx]['doc_id'])

    for rank, doc_id in enumerate(top_ranked_ids):
        results.append({'query_id': query_id, 'doc_id':
            doc_id, 'rank': rank})

df_results = pd.DataFrame(results)

evaluation = IREvaluation()

k_set = [1, 5, 10, 20, 50]
for k in k_set:
    evaluation.compute_evaluation_metrics(df_qrels,
        df_results, k)

# Computing metrics for ColBERT
results = []
query_ids = df_qrels['query_id'].unique()
top_k = 50

# ColBERT retrieving metrics

for query_id in tqdm(query_ids, desc="Processing queries"):
    query_row = df_queries.loc[df_queries['query_id'] ==
        query_id, text_column]
    if query_row.empty:
        raise ValueError(f"Query ID {query_id} not found in
            df_queries.")
    query = query_row.values[0]

    # Semantic search with colBERT
    colbert_top_idx, scores = colbertSearchFaiss.
```

```
        colbert_search_results(query, top_k_results=top_k,
        top_k_search_tokens=10)

    top_ranked_ids = []
    for idx in colbert_top_idx:
        idx = int(idx)
        top_ranked_ids.append(df_docs.iloc[idx]['doc_id'])

    for rank, doc_id in enumerate(top_ranked_ids):
        results.append({'query_id': query_id, 'doc_id':
            doc_id, 'rank': rank})

df_results = pd.DataFrame(results)

evaluation = IREvaluation()

k_set = [1, 5, 10, 20, 50]
for k in k_set:
    evaluation.compute_evaluation_metrics(df_qrels,
        df_results, k)
```

We can also compute metrics for ColBERT as a re-ranker. For more detail, the reader can look at [...].

```
from colbert_kit.evaluating.ir_eval import IREvaluation
from colbert_kit.reranking.colbert_batch_reranker import
    colBERTReRankerBatch
from colbert_kit.evaluating.ir_eval import IREvaluation

results = []
query_ids = df_qrels['query_id'].unique()
top_k = 100

# Load colBERT
model_name_or_path = "./model/colbert"

colbert_reranker_batch = colBERTReRankerBatch(
    model_name_or_path=model_name_or_path, device="cpu")

for query_id in tqdm(query_ids, desc="Processing queries"):
    query_row = df_queries.loc[df_queries['query_id'] ==
        query_id, text_column]
    if query_row.empty:
        raise ValueError(f"Query ID {query_id} not found in
            df_queries.")
    query = query_row.values[0]

    # Get top document indices from BM25
    bm25_scores, bm25_top_idx = BM25search.search(query,
        top_k)
```

```
22
23      doc_candidates = [texts[idx] for idx in bm25_top_idx]
24      candidate_idx = list(bm25_top_idx)
25
26      top_n_scores, top_n_idx = colbert_reranker_batch.
            reranker(query, doc_candidates, candidate_idx,
            batch_size = 1, top_n=50)
27
28      top_ranked_ids = []
29      for idx in top_n_idx:
30          idx = int(idx)
31          top_ranked_ids.append(df_docs.iloc[idx]['doc_id'])
32
33      for rank, doc_id in enumerate(top_ranked_ids):
34          results.append({'query_id': query_id, 'doc_id':
                doc_id, 'rank': rank})
35
36  df_results = pd.DataFrame(results)
37
38  evaluation = IREvaluation()
39
40  k_set = [1, 5, 10, 20, 50]
41  for k in k_set:
42      evaluation.compute_evaluation_metrics(df_qrels,
            df_results, k)
```

8 Choosing negatives

We employed two techniques for choosing negatives. In real applications, it is difficult in general to choose negatives for training or fine-tuning ranking models. The first technique is choosing random negative. The inputs include a data frame with two columns (EXACTLY) "sentence" and "positive_sentence", number of negatives, and triplet output path. .

```
1  import pandas as pd
2  from colbert_kit.choosing_negatives.
       choosing_random_negatives import build_random_triplets
3
4  # Step 1: Sample DataFrame
5  data = {
6      "sentence": [
7          "A man is playing guitar.",
8          "A woman is cooking.",
9          "Children are playing soccer.",
10         "A cat is sleeping on the couch.",
11         "He is reading a book."
12     ],
13     "positive_sentence": [
```

```
        "Someone plays a musical instrument.",
        "She is preparing food.",
        "Kids are having fun outdoors.",
        "An animal rests on furniture.",
        "A person is reading."
    ]
}

df = pd.DataFrame(data)

# Step 3: Run it and save to CSV
output_path = "triplets_example.csv"
build_random_triplets(
    df=df,
    n_negatives=2,
    triplet_output_path=output_path
)

triplet_df = pd.read_csv("triplets_example.csv")

# Step 4: Load and show results
print("Sample triplets:")
print(triplet_df.head())
```

The second technique we used is to choose hard negatives. This is a little more complicated, as we use a sentence-transformer model to retrieve negatives that are very similar to the positive sentence. Our indexing method is HNSW. The inputs include

- a data frame with EXACTLY two columns named "sentence" and "positive_sentence"
- a sentence-transformer model name or path
- a start index: an index from which to start selecting hard negatives among the top retrieved results
- number of negatives : number of hard negatives to retrieve per anchor sentence
- triplet_output_path: File path to save the generated triplets as a CSV
- index_name and index_output_path: a place to save the index with sentence transformer
- use_gpu: Whether to use GPU acceleration for sentence transformer encoding and FAISS. Default is False.

```
import pandas as pd
from colbert_kit.choosing_negatives.choosing_hard_negatives
    import build_hard_triplets
```

```
3
4  # Step 1: Sample DataFrame
5  data = {
6      "sentence": [
7          "A man is playing guitar.",
8          "A woman is cooking.",
9          "Children are playing soccer.",
10         "A cat is sleeping on the couch.",
11         "He is reading a book."
12     ],
13     "positive_sentence": [
14         "Someone plays a musical instrument.",
15         "She is preparing food.",
16         "Kids are having fun outdoors.",
17         "An animal rests on furniture.",
18         "A person is reading."
19     ]
20 }
21
22 df = pd.DataFrame(data)
23
24 # Step 3: Run it and save to CSV
25 build_hard_triplets(
26     df=df,
27     model_name_or_path='all-MiniLM-L6-v2',
28     start_index=2,
29     n_negatives=2,
30     triplet_output_path='hard_triplets.csv',
31     index_name='hard.index',
32     index_output_path='./indices_test',
33     use_gpu=False
34 )
35
36 triplet_df = pd.read_csv("hard_triplets.csv")
37
38 # Step 4: Load and show results
39 print("Sample triplets:")
40 print(triplet_df)
```