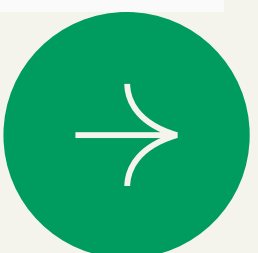
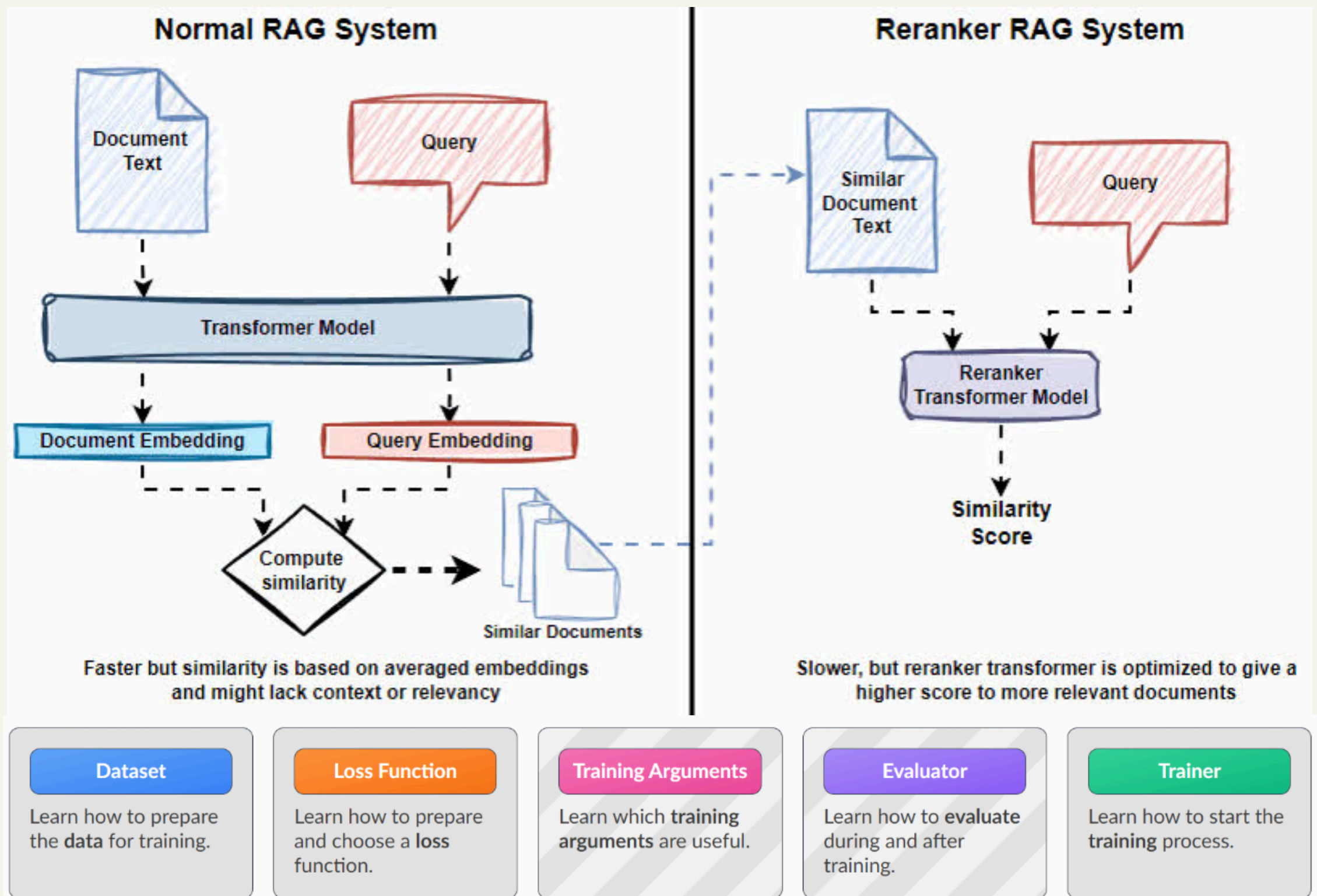
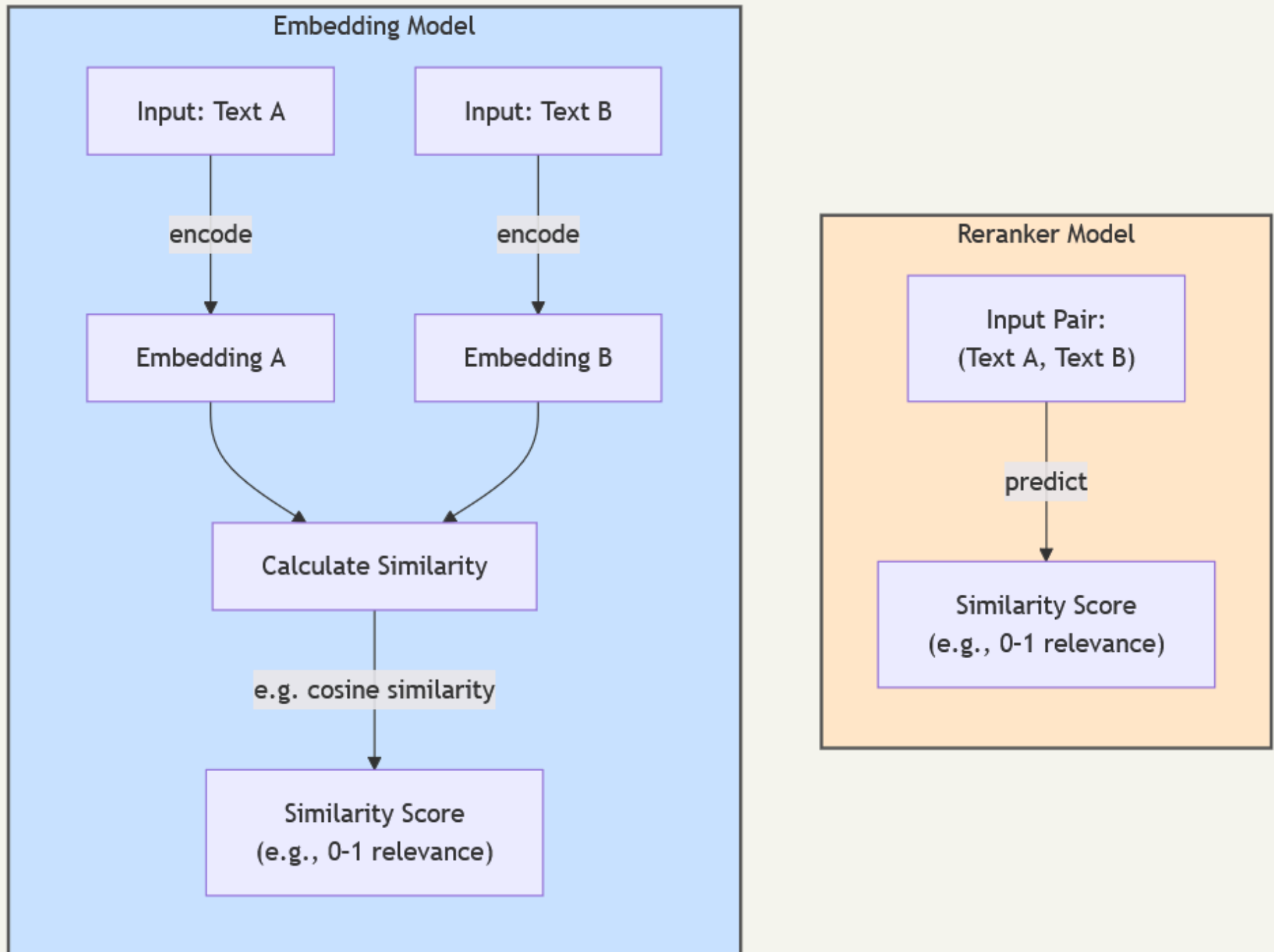


Fine-tuning Reranker Models for RAG Systems

A Practical Guide



Why Fine-Tune Reranker Models?



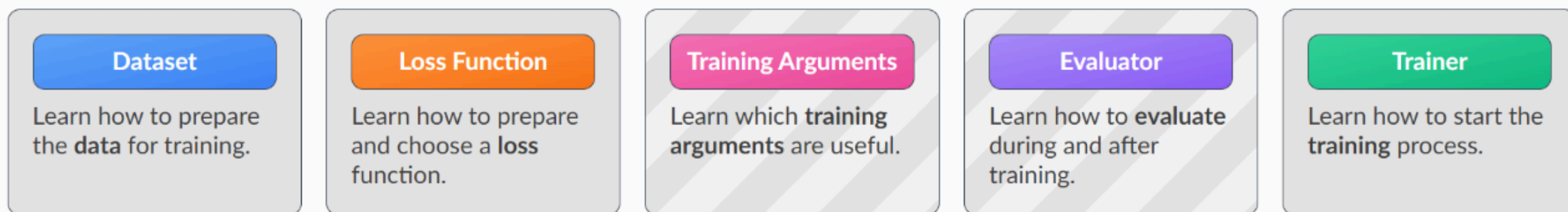
- **Reranker models are cross-encoders unlike bi-encoder transformer models which are used for embedding models**
- **These models are usually already fine-tuned with pairs of text and outputs a relevance score**
- **Most off-the-shelf rerankers are trained on a wide variety of domains but lack the speciality or exclusivity of your domain**
- **Fine-tuning rerankers can help infuse and adapt these models to your enterprise data, context, lingo and scenarios**

How to Fine-tune Reranker Models

We will be using the very useful Sentence Transformers library to fine-tune our embedder model. We will also use an open embedder model so you can host it anywhere.

The key components and steps in the fine-tuning process are depicted in the following figure:

Training Sentence Transformer models involves between 3 to 5 components:



- **Step 1: Prepare your dataset for training the model (with augmentation)**
- **Step 2: Load Pre-trained Reranker Model and Choose your Loss Function**
- **Step 3: Setup Training Arguments**
- **Step 4: Build an evaluator for evaluating the model on validation data (optional)**
- **Step 5: Train the Reranker Model**

Step 1 – Dataset Preparation

Data on the Hugging Face Hub

You can use the `load_dataset` function to load data from datasets in the [Hugging Face Hub](#)

```
from datasets import load_dataset

train_dataset = load_dataset("sentence-transformers/natural-questions", split="train")

print(train_dataset)
"""
Dataset({
  features: ['query', 'answer'],
  num_rows: 100231
})
"""
```

Local Data (CSV, JSON, Parquet, Arrow, SQL)

You can also use `load_dataset` for loading local data in certain file formats:

```
from datasets import load_dataset

dataset = load_dataset("csv", data_files="my_file.csv")
# or
dataset = load_dataset("json", data_files="my_file.json")
```

The `CrossEncoderTrainer` uses `datasets.Dataset` OR `datasets.DatasetDict` instances for training and evaluation. You can load data from the [Hugging Face Datasets Hub](#) or use your local data in whatever format you prefer (e.g. CSV, JSON, Parquet, Arrow, or SQL).

Step 1 – Dataset Preparation

Dataset Format

It is important that your dataset format matches your loss function (or that you choose a loss function that matches your dataset format and model). Verifying whether a dataset format and model work with a loss function involves three steps:

1. All columns not named "label", "labels", "score", or "scores" are considered *Inputs* according to the [Loss Overview](#) table. The number of remaining columns must match the number of valid inputs for your chosen loss.
2. If your loss function requires a *Label* according to the [Loss Overview](#) table, then your dataset must have a **column named "label", "labels", "score", or "scores"**. This column is automatically taken as the label.
3. The number of model output labels matches what is required for the loss according to [Loss Overview](#) table.

For example, given a dataset with columns `["text1", "text2", "label"]` where the "label" column has float similarity score ranging from 0 to 1 and a model outputting 1 label, we can use it with

[BinaryCrossEntropyLoss](#) because:

1. the dataset has a "label" column as is required for this loss function.
2. the dataset has 2 non-label columns, exactly the amount required by this loss functions.
3. the model has 1 output label, exactly as required by this loss function.

Be sure to re-order your dataset columns with [Dataset.select_columns](#) if your columns are not ordered correctly. For example, if your dataset has `["good_answer", "bad_answer", "question"]` as columns, then this dataset can technically be used with a loss that requires (anchor, positive, negative) triplets, but the `good_answer` column will be taken as the anchor, `bad_answer` as the positive, and `question` as the negative.

Additionally, if your dataset has extraneous columns (e.g. `sample_id`, `metadata`, `source`, `type`), you should remove these with [Dataset.remove_columns](#) as they will be used as inputs otherwise. You can also use [Dataset.select_columns](#) to keep only the desired columns.

Step 1 – Dataset Augmentation

The success of training reranker models often depends on the quality of the *negatives*, i.e. the passages for which the query-negative score should be low. Negatives can be divided into two types:

- **Soft negatives:** passages that are completely unrelated. Also called easy negatives.
- **Hard negatives:** passages that seem like they might be relevant for the query, but are not.

A concise example is:

- **Query:** Where was Apple founded?
- **Soft Negative:** The Cache River Bridge is a Parker pony truss that spans the Cache River between Walnut Ridge and Paragould, Arkansas.
- **Hard Negative:** The Fuji apple is an apple cultivar developed in the late 1930s, and brought to market in 1962.

The strongest CrossEncoder models are generally trained to recognize hard negatives, and so it's valuable to be able to "mine" hard negatives to train with. Sentence Transformers supports a strong `mine_hard_negatives` function that can assist, given a dataset of query-answer pairs:

Step 1 – Dataset Augmentation


```
from datasets import load_dataset
from sentence_transformers import SentenceTransformer
from sentence_transformers.util import mine_hard_negatives

# Load the GooAQ dataset: https://huggingface.co/datasets/sentence-transformers/gooaq
train_dataset = load_dataset("sentence-transformers/gooaq",
split="train").select(range(100_000))
print(train_dataset)

# Mine hard negatives using a very efficient embedding model
embedding_model = SentenceTransformer("sentence-transformers/static-retrieval-mrl-en-
v1", device="cpu")
hard_train_dataset = mine_hard_negatives(
    train_dataset,
    embedding_model,
    num_negatives=5, # How many negatives per question-answer pair
    range_min=10, # Skip the x most similar samples
    range_max=100, # Consider only the x most similar samples
    max_score=0.8, # Only consider samples with a similarity score of at most x
    margin=0.1, # Similarity between query and negative samples should be x lower than
query-positive similarity
    sampling_strategy="top", # Randomly sample negatives from the range
    batch_size=4096, # Use a batch size of 4096 for the embedding model
    output_format="labeled-pair", # The output format is (query, passage, label), as
required by BinaryCrossEntropyLoss
    use_faiss=True, # Using FAISS is recommended to keep memory usage low (pip install
faiss-gpu or pip install faiss-cpu)
)

print(hard_train_dataset)
print(hard_train_dataset[1])
```


Step 2 - Load Pre-Trained Reranker Model



```
from sentence_transformers import CrossEncoder

# Load a model to train/finetune
model = CrossEncoder("mixedbread-ai/mxbai-rerank-large-v2",
                    num_labels=1) # num_labels=1 is for rerankers
```

- We use a popular open-source model **mxbai-rerank-large-v2**
- You can choose any reranker model for this
- All credits to Sentence Transformers for creating useful interfaces to load, train and use these models

Step 2 – Choose Loss Function

Loss Function

Loss functions help evaluate a model's performance on a set of data and direct the training process. The right loss function for your task depends on the data you have and what you're trying to achieve. You can find a full list of available loss functions in the [Loss Overview](#).

Most loss functions are easy to set up - you just need to provide the `CrossEncoder` model you're training:

```
from datasets import load_dataset
from sentence_transformers import CrossEncoder
from sentence_transformers.cross_encoder.losses import CachedMultipleNegativesRankingLoss

# Load a model to train/finetune
model = CrossEncoder("xlm-roberta-base", num_labels=1) # num_labels=1 is for rerankers

# Initialize the CachedMultipleNegativesRankingLoss, which requires pairs of
# related texts or triplets
loss = CachedMultipleNegativesRankingLoss(model)

# Load an example training dataset that works with our loss function:
train_dataset = load_dataset("sentence-transformers/gooaq", split="train")

...
```

- **Loss functions quantify how well a model performs for a given batch of data, allowing an optimizer to update the model weights to produce more favourable (i.e., lower) loss values. This is the core of the training process.**
- **MultipleNegativesRankingLoss is a great loss function if you only have positive pairs, for example, only pairs of related texts like pairs of (query, response), or (query, context)**

Step 3 – Setup Training Arguments

You can customize the training process using the [CrossEncoderTrainingArguments](#) class. This class lets you adjust parameters that can impact training speed and help you understand what's happening during training.

For more information on the most useful training arguments, check out the [Cross Encoder > Training Overview > Training Arguments](#). It's worth reading to get the most out of your training.

Here's an example of how to set up [CrossEncoderTrainingArguments](#):

```
from sentence_transformers.cross_encoder import CrossEncoderTrainingArguments

args = CrossEncoderTrainingArguments(
    # Required parameter:
    output_dir="models/reranker-MiniLM-msmarco-v1",
    # Optional training parameters:
    num_train_epochs=1,
    per_device_train_batch_size=16,
    per_device_eval_batch_size=16,
    learning_rate=2e-5,
    warmup_ratio=0.1,
    fp16=True, # Set to False if you get an error that your GPU can't run on
               # FP16
    bf16=False, # Set to True if you have a GPU that supports BF16
    batch_sampler=BatchSamplers.NO_DUPLICATES, # losses that use "in-batch
negatives" benefit from no duplicates
    # Optional tracking/debugging parameters:
    eval_strategy="steps",
    eval_steps=100,
    save_strategy="steps",
    save_steps=100,
    save_total_limit=2,
    logging_steps=100,
)
```

- **learning_rate** is useful when gradient updates are happening, do not set this to a very high number as it can destroy what the model has previously learnt
- ***_batch_size** is the number of records which will pass through the model before a full gradient update happens through backpropagation and model weights are updated

Step 4 – Setup Trainer & Fine-tune Reranker Model

The `CrossEncoderTrainer` is where all previous components come together. We only have to specify the trainer with the model, training arguments (optional), training dataset, evaluation dataset (optional), loss function, evaluator (optional) and we can start training. Let's have a look at a script where all of these components come together:

```
trainer = CrossEncoderTrainer(  
    model=model,  
    args=args,  
    train_dataset=hard_train_dataset,  
    loss=loss,  
    evaluator=evaluator,  
)  
trainer.train()
```

- **trainer.train()** will start the training (fine-tuning) process for your pre-trained reranker model
- In case of GPU OOM errors consider reducing the batch size
- To track your model's performance during training, you can pass an `eval_dataset` to the `CrossEncoderTrainer`.
- You might want more detailed metrics and evaluators can help you assess your model's performance using specific metrics
- You can use an evaluation dataset, an evaluator, both, or neither, depending on your needs.

Bonus: Training Tips

Cross Encoder models have their own unique quirks, so here's some tips to help you out:

1. CrossEncoder models overfit rather quickly, so it's recommended to use an evaluator like `CrossEncoderNanoBEIREvaluator` OR `CrossEncoderRerankingEvaluator` together with the `load_best_model_at_end` and `metric_for_best_model` training arguments to load the model with the best evaluation performance after training.
2. CrossEncoders are particularly receptive to strong hard negatives (`mine hard negatives`). They teach the model to be very strict, useful e.g. when distinguishing between passages that answer a question or passages that relate to a question.
 1. Note that if you only use hard negatives, your model may unexpectedly perform worse for easier tasks. This can mean that reranking the top 200 results from a first-stage retrieval system (e.g. with a `SentenceTransformer` model) can actually give worse top-10 results than reranking the top 100. Training using random negatives alongside hard negatives can mitigate this.
3. Don't underestimate `BinaryCrossEntropyLoss`, it remains a very strong option despite being simpler than learning-to-rank (`LambdaLoss`, `ListNetLoss`) or in-batch negatives (`CachedMultipleNegativesRankingLoss`, `MultipleNegativesRankingLoss`) losses, and its data is easy to prepare, especially using `mine hard negatives`.