Fine-tune ModernBERT for RAG with Synthetic Data

Retrieval Augmented Generation (RAG) is a widely adopted framework for building question-answering systems. By retrieving relevant information from a knowledge base—whether it's the web or your documents—RAG enhances users' trustworthiness and reliability. It provides up-to-date and verifiable domain-specific data while being more efficient and cost-effective, eliminating the need for post-training LLMs.

To improve the quality of the generated responses provided by the RAG system, it is essential to have good-performing retrieval and reranking models. For that, we can fine-tune them with our own data so that they can accurately identify the relevant information and order it. However, additional data related to your task is required to fine-tune them, which is not always available.

This blog post will showcase **how to fine-tune retrieval and reranking models using your documents**. Using these documents, we can create synthetic training data representing your domain. This enables you to improve performance even when real-world data is scarce. In our use case, we will improve a RAG system that responds to legal documentation on human and civil rights.

The first step is to generate synthetic data for RAG. We will use the Synthetic Data Generator, a user-friendly application that uses no code to create custom datasets with LLMs.

> *For more information about the details and usage of the Synthetic Data Generator, check Introducing the Synthetic Data Generator - Build Datasets with Natural Language and the original GitHub Repository.*

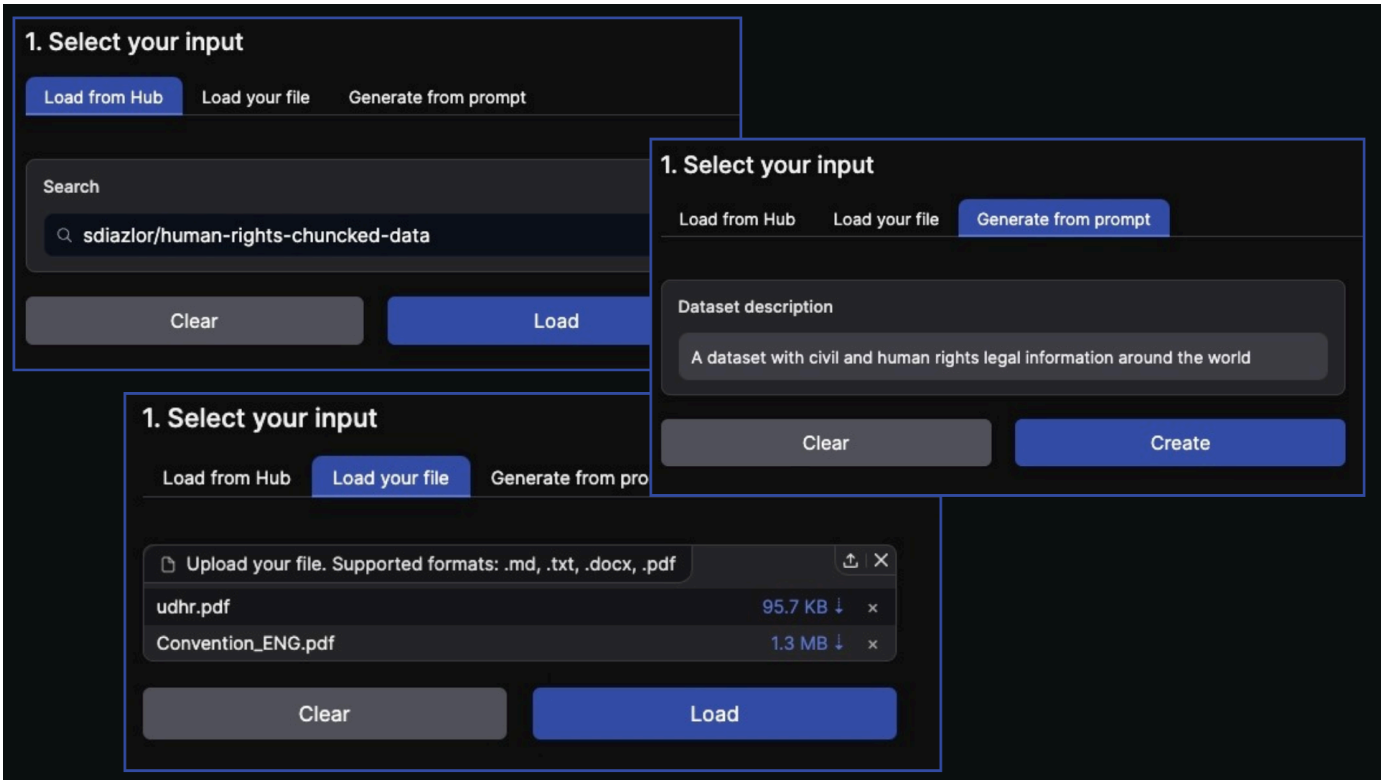
Generating data with the Synthetic Data Generator is a straightforward process that involves just three key steps:

1. **Selecting the Input Data**: Choose a representative sample dataset, document that reflects the structure and characteristics of your target knowledge base, or even generate from scratch specifying the type of dataset.
2. **Configuring the Generator**: Set up the generator parameters and iterate over a sample dataset to refine and validate the generation process.
3. **Generating the Dataset**: Once the configuration is optimized, use the generator to create the complete synthetic dataset.

In the following sections, we will explore these steps to guide you.

To generate relevant information for your use case, you can provide a source of information in the form of a dataset from the Hub or directly upload raw documents in formats such as .pdf, .md, .txt, or .docx. Moreover, you can instead write a dataset description that should outline the topic, scope, and specific requirements of your retrieval task, ensuring the generated data is both relevant and fit for purpose.

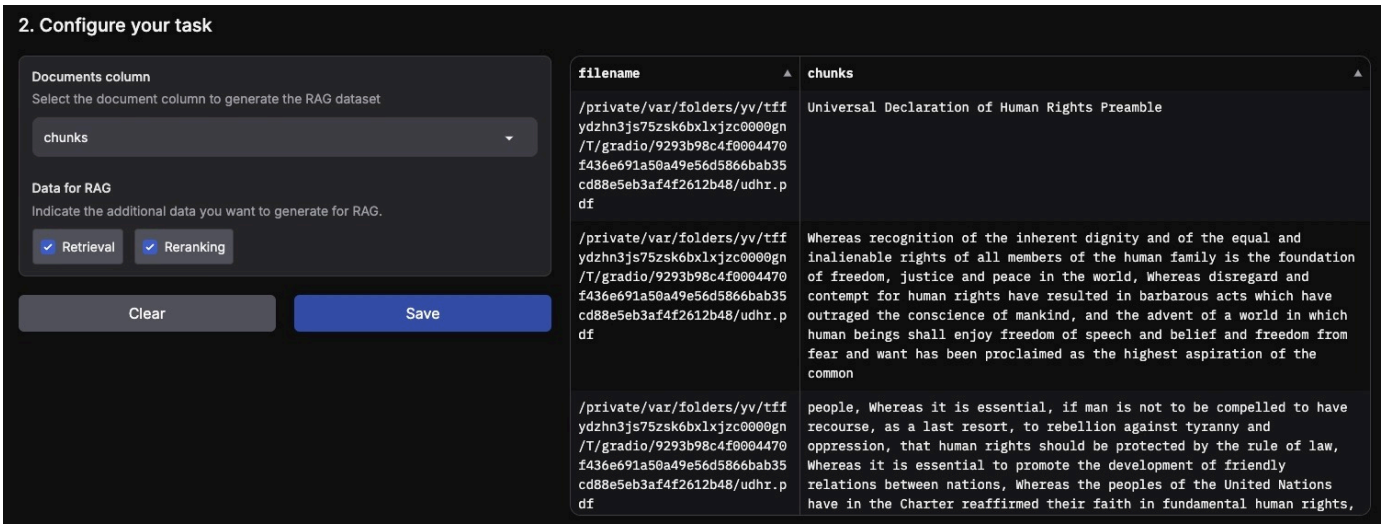
In our examples, we could not find a suitable dataset containing information about human rights. Instead, we created two datasets. We provided two PDF files for the first one: The European Convention on Human Rights and the Universal Declaration of Human Rights. These documents are the foundation for generating synthetic data, ensuring it aligns closely with the topic. For the second one, we opted to broaden the scope by writing a detailed description of our dataset, offering a more general perspective on human rights. By combining these approaches, we ensured our datasets captured both specific and general aspects of the topic, improving the versatility of the generated data.



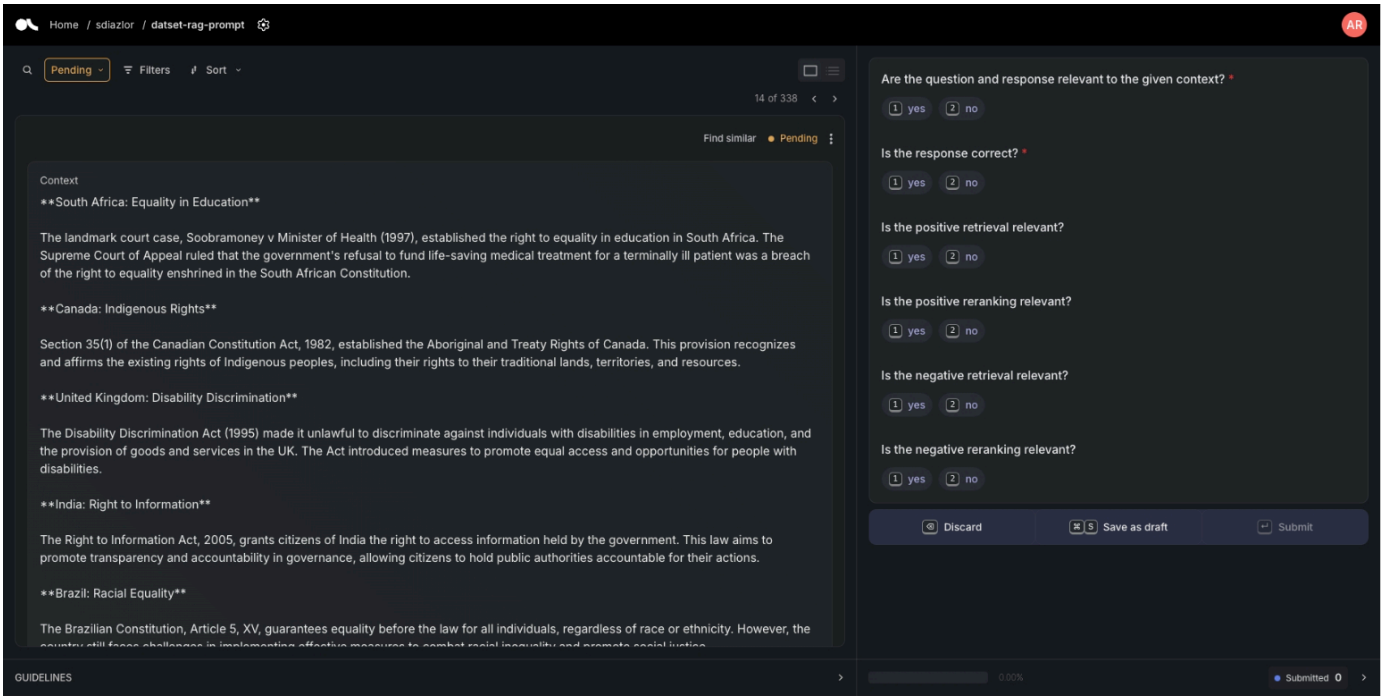
Next, we will iterate over a sample dataset to configure generation parameters. This configuration slightly differs depending on the selected input.

- If you have selected a dataset or raw file as input, which is automatically chunked to facilitate processing, you first need to select the column containing the pieces of information.
- A system prompt is automatically generated based on your description when using a dataset description. This prompt outlines the task retrieval and can be regenerated or modified as needed to fit your needs better.

You can add data during this step regardless of the input type. If no additional settings are applied, the generated dataset will include three columns: Context, Question, and Answer. If retrieval is indicated, the output will include positive and negative queries. If reranking is indicated, the output will include positive and negative examples based on the context. In our case, we selected retrieval and reranking to fine-tune the models for these tasks.



Once we have completed the previous steps, we are ready to generate the full dataset! The generated datasets will be automatically available in the Hub and Argilla, ready for review and use.



We generated 500 rows for each using the Serverless Inference API, which took around 40 minutes from the source files and 1 hour from the dataset description.

Great! You've mastered the use of the Synthetic Data Generator. Let's go to the next step: training the models with our data!

> *The next sections present simplified code snippets for clarity and easy understanding. You can access the full notebook here if you'd like to explore the complete implementation.*

To optimize the retrieval, we will fine-tune a sentence similarity model with a bi-encoder (faster but less accurate) and, for reranking, a cross-encoder (slower but more accurate). For this, we will use the Sentence Transformers library and nomic-ai/modernbert-embed-base, an embedding model trained from ModernBERT-base.

▶ What's the difference between a bi-encoder and a cross-encoder?

Before training our models, we will combine our datasets, filter them, clean them, and prepare them for retrieval and reranking. For retrieval, we will use triplets (anchor, positive, and negative). In the case of reranking, where triplets are not recommended, we will use a sentence pair (anchor and positive) and a similarity score, so we will compute it using Snowflake/snowflake-arctic-embed-m-v1.5 based on the MTEB leaderboard.

```
# Load the datasets and combine them
dataset_rag_from_file = load_dataset(f"{REPO_NAME}/rag-human-rights-from-files", split="train")
dataset_rag_from_prompt = load_dataset(f"{REPO_NAME}/rag-human-rights-from-prompt", split="train")

combined_rag_dataset = concatenate_datasets(
    [dataset_rag_from_file, dataset_rag_from_prompt]
)
```

```
# Filter the empty and NaN values
filtered_rag_dataset = combined_rag_dataset.filter(filter_empty_or_nan).shuffle(seed=42)

# Format the data for retrieval and reranking
clean_rag_dataset_biencoder = rename_and_reorder_columns(
    filtered_rag_dataset,
    rename_map={"context": "anchor", "positive_retrieval": "positive", "negative_retrieval": "negative"},
    selected_columns=["anchor", "positive", "negative"],
)

clean_rag_dataset_crossencoder = rename_and_reorder_columns(
    filtered_rag_dataset,
    rename_map={"context": "anchor", "positive_retrieval": "positive"}, #TODO
    selected_columns=["anchor", "positive"],
)

# Add scores for reranking
clean_rag_dataset_crossencoder = clean_rag_dataset_crossencoder.map(
    add_reranking_scores, batched=True, batch_size=250
)

# Split the datasets
dataset_rag_biencoder = split_dataset(clean_rag_dataset_biencoder)
dataset_rag_crossencoder = split_dataset(clean_rag_dataset_crossencoder)
```

Now, we can initialize our model and start training. Configure your training arguments according to your resource requirements to improve performance and accuracy. This will push our sdiazlor/modernbert-embed-base-biencoder-human-rights model.

```
# Initialize the SentenceTransformer model
model_biencoder = SentenceTransformer(
    MODEL,
    model_card_data=SentenceTransformerModelCardData(
        language="en",
        license="apache-2.0",
        model_name=MODEL_NAME_BIENCODER,
    ),

# Train the model
trainer = SentenceTransformerTrainer(
    model=model_biencoder,
    args=training_args,
    train_dataset=dataset_rag_biencoder["train"],
    eval_dataset=dataset_rag_biencoder["eval"],
    loss=loss_biencoder,
    evaluator=triplet_evaluator,
)
trainer.train()

# Save the model to the local directory and push it to the Hub
model_biencoder.save_pretrained(f"models/{MODEL_NAME_BIENCODER}")
model_biencoder.push_to_hub(f"{REPO_NAME}/{MODEL_NAME_BIENCODER}")
```

After that, we can start training the cross-encoder. We will set the number of labels as 1 as it's a regression task. This will push our sdiazlor/modernbert-embed-base-crossencoder-human-rights model.

```
# Initialize the CrossEncoder model
model_crossencoder = CrossEncoder(model_name=MODEL, num_labels=1)

# Train the model
model_crossencoder.fit(
    train_dataloader=train_dataloader,
    evaluator=evaluator,
    epochs=3,
    warmup_steps=500,
    output_path=f"models/{MODEL_NAME_CROSSENCODER}",
    save_best_model=True,
)

# Save the model to the local directory and push it to the Hub
model_crossencoder.save_pretrained(f"models/{MODEL_NAME_CROSSENCODER}")
model_crossencoder.push_to_hub(f"{REPO_NAME}/{MODEL_NAME_CROSSENCODER}")
```

Voilà! We've successfully trained our models for retrieval and reranking. In our case, the training process took approximately 1 hour for each model. However, keep in mind that the training duration can vary significantly depending on the training arguments and the number of samples used.
Feel free to experiment with these configurations to optimize performance or simply to explore how different settings impact the results.

Ready to use your models? We will use Haystack, an open-source framework for building production-ready LLM applications, retrieval-augmented generative pipelines and state-of-the-art search systems. So, we will build a RAG pipeline with a retriever (the bi-encoder model), the ranker (the cross-encoder model), and meta-llama/Llama-3.1-8B-Instruct as the LLM.

```
# Initialize the pipeline with the components
rag_pipeline = Pipeline()
rag_pipeline.add_component("text_embedder", text_embedder)
rag_pipeline.add_component("retriever", retriever)
rag_pipeline.add_component("ranker", ranker)
rag_pipeline.add_component("prompt_builder", prompt_builder)
rag_pipeline.add_component("llm", chat_generator)

# Connect the components to each other
rag_pipeline.connect("text_embedder.embedding", "retriever.query_embedding")
rag_pipeline.connect("retriever.documents", "ranker.documents")
rag_pipeline.connect("ranker", "prompt_builder")
rag_pipeline.connect("prompt_builder.prompt", "llm.messages")
```

Once we have our pipeline, we can start asking our system.

```
response = rag_pipeline.run(
    {
        "text_embedder": {"text": question},
```

```
            "prompt_builder": {"question": question},
            "ranker": {"query": question},
        }
)
```

Depending on the provided documentation and your fine-tuned models, it will get the information or indicate if it lacks some data. For instance:

```
# A response lacking information with the base model
Unfortunately, the text doesn't provide a specific answer to the question of how many human rights there are. It discusses various human

# A response lacking information with the fine-tuned model
It seems that there is not enough information given in the human rights protocols provided to accurately answer the question. However, we

Not possible to answer your question due to lack of information, however we can tell you the most widely respected declared world documen

# A response with the base model
The question is incomplete. However, based on the information provided, I can infer that the correct information might be related to the

The Right to a Fair Trial is not explicitly mentioned in the provided text. However, equality before the law and freedom from arbitrary d

# A response with the fine-tuned model
The information you provided does not directly list the "Right of Fair Trial" but looking under articles of the Convention for the Protec

Article 6. Right to a fair Trial [...]
```

In this blog, we have seen the full workflow for building your RAG system, from generating synthetic data for RAG on our custom use case to fine-tuning the models for retrieval and reranking, and finally creating the full pipeline.

You can also check the blog-posts for the rest of the available tasks in the Synthetic Data Generator:

What are you waiting for? First step: Start synthesizing!