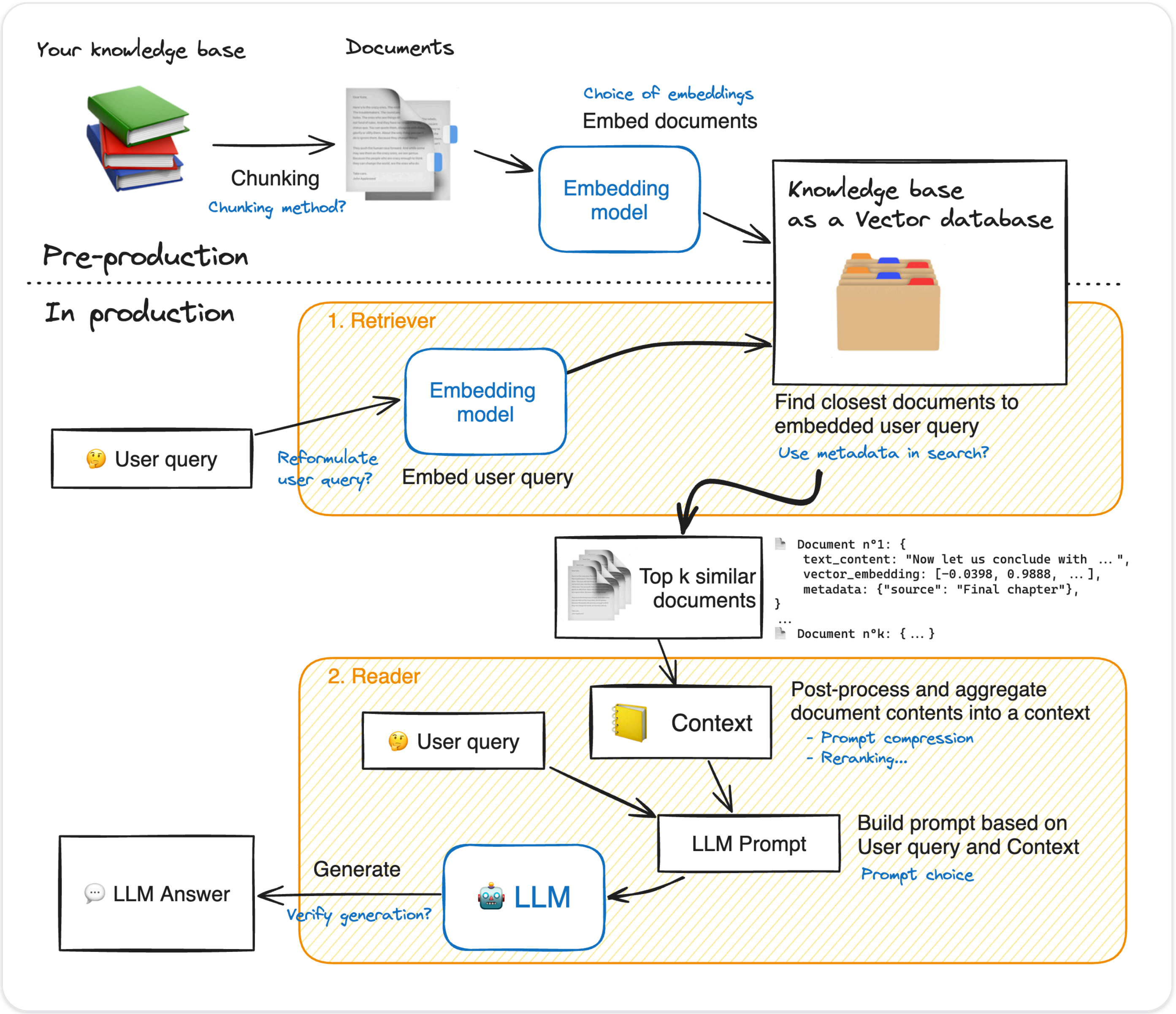
RAG Evaluation

Authored by: Aymeric Roucher

This notebook demonstrates how you can evaluate your RAG (Retrieval Augmented Generation), by building a synthetic evaluation dataset and using LLM-as-a-judge to compute the accuracy of your system.

For an introduction to RAG, you can check this other cookbook!

RAG systems are complex: here a RAG diagram, where we noted in blue all possibilities for system enhancement:



Implementing any of these improvements can bring a huge performance boost; but changing anything is useless if you cannot monitor the impact of your changes on the system's performance! So let's see how to evaluate our RAG system.

Evaluating RAG performance

Since there are so many moving parts to tune with a big impact on performance, benchmarking the RAG system is crucial.

For our evaluation pipeline, we will need:

1. An evaluation dataset with question - answer couples (QA couples)

2. An evaluator to compute the accuracy of our system on the above evaluation dataset.

➡️ It turns out, we can use LLMs to help us all along the way!

1. The evaluation dataset will be synthetically generated by an LLM 🤖, and questions will be filtered out by other LLMs 🤖
2. An LLM-as-a-judge agent 🤖 will then perform the evaluation on this synthetic dataset.

Let's dig into it and start building our evaluation pipeline! First, we install the required model dependancies.

```
!pip install -q torch transformers langchain sentence-transformers tqdm openpyxl openai p
```

```
%reload_ext autoreload
%autoreload 2
```

```
from tqdm.auto import tqdm
import pandas as pd
from typing import Optional, List, Tuple
import json
import datasets

pd.set_option("display.max_colwidth", None)
```

```
from huggingface_hub import notebook_login

notebook_login()
```

Load your knowledge base

```
ds = datasets.load_dataset("m-ric/huggingface_doc", split="train")
```

1. Build a synthetic dataset for evaluation

We first build a synthetic dataset of questions and associated contexts. The method is to get elements from our knowledge base, and ask an LLM to generate questions based on these documents.

Then we setup other LLM agents to act as quality filters for the generated QA couples: each of them will act as the filter for a specific flaw.

1.1. Prepare source documents

```
from langchain.text_splitter import RecursiveCharacterTextSplitter
from langchain.docstore.document import Document as LangchainDocument

langchain_docs = [LangchainDocument(page_content=doc["text"], metadata={"source": doc["so


text_splitter = RecursiveCharacterTextSplitter(
    chunk_size=2000,
    chunk_overlap=200,
    add_start_index=True,
    separators=["\n\n", "\n", ".", " ", ""],
)

docs_processed = []
for doc in langchain_docs:
    docs_processed += text_splitter.split_documents([doc])
```

1.2. Setup agents for question generation

We use Mixtral for QA couple generation because it it has excellent performance in leaderboards such as Chatbot Arena.

```
from huggingface_hub import InferenceClient


repo_id = "mistralai/Mixtral-8x7B-Instruct-v0.1"

llm_client = InferenceClient(
    model=repo_id,
    timeout=120,
)


def call_llm(inference_client: InferenceClient, prompt: str):
    response = inference_client.post(
        json={
            "inputs": prompt,
            "parameters": {"max_new_tokens": 1000},
            "task": "text-generation",
        },
    )
    return json.loads(response.decode())[0]["generated_text"]


call_llm(llm_client, "This is a test context")
```

```
QA_generation_prompt = """
Your task is to write a factoid question and an answer given a context.
Your factoid question should be answerable with a specific, concise piece of factual info
Your factoid question should be formulated in the same style as questions users could ask
This means that your factoid question MUST NOT mention something like "according to the p

Provide your answer as follows:

Output:::
Factoid question: (your factoid question)
Answer: (your answer to the factoid question)

Now here is the context.

Context: {context}\n
Output:::"""
```

Now let's generate our QA couples. For this example, we generate only 10 QA couples and will load the rest from the Hub.

But for your specific knowledge base, given that you want to get at least ~100 test samples, and accounting for the fact that we will filter out around half of these with our critique agents later on, you should generate much more, in the >200 samples.

```python
import random

N_GENERATIONS = 10  # We intentionally generate only 10 QA couples here for cost and time

print(f"Generating {N_GENERATIONS} QA couples...")

outputs = []
for sampled_context in tqdm(random.sample(docs_processed, N_GENERATIONS)):
    # Generate QA couple
    output_QA_couple = call_llm(llm_client, QA_generation_prompt.format(context=sampled_c
    try:
        question = output_QA_couple.split("Factoid question: ")[-1].split("Answer: ")[0]
        answer = output_QA_couple.split("Answer: ")[-1]
        assert len(answer) < 300, "Answer is too long"
        outputs.append(
            {
                "context": sampled_context.page_content,
                "question": question,
                "answer": answer,
                "source_doc": sampled_context.metadata["source"],
            }
        )
    except:
        continue
```

```python
display(pd.DataFrame(outputs).head(1))
```

1.3. Setup critique agents

The questions generated by the previous agent can have many flaws: we should do a quality check before validating these questions.

We thus build critique agents that will rate each question on several criteria, given in this paper:

- **Groundedness:** can the question be answered from the given context?
- **Relevance:** is the question relevant to users? For instance, `"What is the date when transformers 4.29.1 was released?"` is not relevant for ML practitioners.

One last failure case we've noticed is when a function is tailored for the particular setting where the question was generated, but undecipherable by itself, like `"What is the name of the function used in this guide?"`. We also build a critique agent for this criteria:

- **Stand-alone**: is the question understandable free of any context, for someone with domain knowledge/Internet access? The opposite of this would be `What is the function used in this article?` for a question generated from a specific blog article.

We systematically score functions with all these agents, and whenever the score is too low for any one of the agents, we eliminate the question from our eval dataset.

💡 ***When asking the agents to output a score, we first ask them to produce its rationale. This will help us verify scores, but most importantly, asking it to first output rationale gives the model more tokens to think and elaborate an answer before summarizing it into a single score token.***

We now build and run these critique agents.

```
question_groundedness_critique_prompt = """
You will be given a context and a question.
Your task is to provide a 'total rating' scoring how well one can answer the given questi
Give your answer on a scale of 1 to 5, where 1 means that the question is not answerable

Provide your answer as follows:

Answer:::
Evaluation: (your rationale for the rating, as a text)
Total rating: (your rating, as a number between 1 and 5)

You MUST provide values for 'Evaluation:' and 'Total rating:' in your answer.

Now here are the question and context.

Question: {question}\n
Context: {context}\n
Answer::: """

question_relevance_critique_prompt = """
You will be given a question.
Your task is to provide a 'total rating' representing how useful this question can be to
```

```
Give your answer on a scale of 1 to 5, where 1 means that the question is not useful at a

Provide your answer as follows:

Answer:::
Evaluation: (your rationale for the rating, as a text)
Total rating: (your rating, as a number between 1 and 5)

You MUST provide values for 'Evaluation:' and 'Total rating:' in your answer.

Now here is the question.

Question: {question}\n
Answer::: """

question_standalone_critique_prompt = """
You will be given a question.
Your task is to provide a 'total rating' representing how context-independent this questi
Give your answer on a scale of 1 to 5, where 1 means that the question depends on additio
For instance, if the question refers to a particular setting, like 'in the context' or 'i
The questions can contain obscure technical nouns or acronyms like Gradio, Hub, Hugging F

For instance, "What is the name of the checkpoint from which the ViT model is imported?"

Provide your answer as follows:

Answer:::
Evaluation: (your rationale for the rating, as a text)
Total rating: (your rating, as a number between 1 and 5)

You MUST provide values for 'Evaluation:' and 'Total rating:' in your answer.

Now here is the question.

Question: {question}\n
Answer::: """
```

```
print("Generating critique for each QA couple...")
for output in tqdm(outputs):
    evaluations = {
        "groundedness": call_llm(
            llm_client,
            question_groundedness_critique_prompt.format(context=output["context"], quest
        ),
        "relevance": call_llm(
            llm_client,
            question_relevance_critique_prompt.format(question=output["question"]),
        ),
        "standalone": call_llm(
            llm_client,
            question_standalone_critique_prompt.format(question=output["question"]),
        ),
    }
    try:
        for criterion, evaluation in evaluations.items():
            score, eval = (
                int(evaluation.split("Total rating: ")[-1].strip()),
                evaluation.split("Total rating: ")[-2].split("Evaluation: ")[1],
            )
            output.update(
                {
                    f"{criterion}_score": score,
                    f"{criterion}_eval": eval,
                }
            )
    except Exception as e:
        continue
```

Now let us filter out bad questions based on our critique agent scores:

```
>>> import pandas as pd

>>> pd.set_option("display.max_colwidth", None)

>>> generated_questions = pd.DataFrame.from_dict(outputs)

>>> print("Evaluation dataset before filtering:")
>>> display(
...     generated_questions[
...         [
...             "question",
...             "answer",
...             "groundedness_score",
...             "relevance_score",
...             "standalone_score",
...         ]
...     ]
... )
>>> generated_questions = generated_questions.loc[
...     (generated_questions["groundedness_score"] >= 4)
...     & (generated_questions["relevance_score"] >= 4)
...     & (generated_questions["standalone_score"] >= 4)
... ]
>>> print("============================================")
>>> print("Final evaluation dataset:")
>>> display(
...     generated_questions[
...         [
...             "question",
...             "answer",
...             "groundedness_score",
...             "relevance_score",
...             "standalone_score",
...         ]
...     ]
... )

>>> eval_dataset = datasets.Dataset.from_pandas(generated_questions, split="train", prese
```

```
Evaluation dataset before filtering:
```

Now our synthetic evaluation dataset is complete! We can evaluate different RAG systems on this evaluation dataset.

We have generated only a few QA couples here to reduce time and cost. But let's kickstart the next part by loading a pre-generated dataset:

```
eval_dataset = datasets.load_dataset("m-ric/huggingface_doc_qa_eval", split="train")
```

2. Build our RAG System

2.1. Preprocessing documents to build our vector database

- In this part, **we split the documents from our knowledge base into smaller chunks**: these will be the snippets that are picked by the Retriever, to then be ingested by the Reader LLM as supporting elements for its answer.
- The goal is to build semantically relevant snippets: not too small to be sufficient for supporting an answer, and not too large too avoid diluting individual ideas.

Many options exist for text splitting:

- split every `n` words / characters, but this has the risk of cutting in half paragraphs or even sentences
- split after `n` words / character, but only on sentence boundaries
- **recursive split** tries to preserve even more of the document structure, by processing it tree-like way, splitting first on the largest units (chapters) then recursively splitting on smaller units (paragraphs, sentences).

To learn more about chunking, I recommend you read this great notebook by Greg Kamradt.

This space lets you visualize how different splitting options affect the chunks you get.

> *“In the following, we use Langchain’s `RecursiveCharacterTextSplitter`.”*

💡 *To measure chunk length in our Text Splitter, our length function will not be the count of characters, but the count of tokens in the tokenized text: indeed, for subsequent embedder that processes token, measuring length in tokens is more relevant and empirically performs better.*

```
from langchain.docstore.document import Document as LangchainDocument

RAW_KNOWLEDGE_BASE = [
    LangchainDocument(page_content=doc["text"], metadata={"source": doc["source"]}) for d
]
```

```
from langchain.text_splitter import RecursiveCharacterTextSplitter
from transformers import AutoTokenizer
```

```
def split_documents(
    chunk_size: int,
    knowledge_base: List[LangchainDocument],
    tokenizer_name: str,
) -> List[LangchainDocument]:
    """
    Split documents into chunks of size `chunk_size` characters and return a list of docu
    """
    text_splitter = RecursiveCharacterTextSplitter.from_huggingface_tokenizer(
        AutoTokenizer.from_pretrained(tokenizer_name),
        chunk_size=chunk_size,
        chunk_overlap=int(chunk_size / 10),
        add_start_index=True,
        strip_whitespace=True,
        separators=["\n\n", "\n", ".", " ", ""],
    )

    docs_processed = []
    for doc in knowledge_base:
        docs_processed += text_splitter.split_documents([doc])

    # Remove duplicates
    unique_texts = {}
    docs_processed_unique = []
    for doc in docs_processed:
        if doc.page_content not in unique_texts:
            unique_texts[doc.page_content] = True
            docs_processed_unique.append(doc)

    return docs_processed_unique
```

2.2. Retriever - embeddings 🗂️

The **retriever acts like an internal search engine**: given the user query, it returns the most relevant documents from your knowledge base.

> *"For the knowledge base, we use Langchain vector databases since **it offers a convenient FAISS index and allows us to keep document metadata throughout the processing**."*

🛠️ Options included:

- Tune the chunking method:
  - Size of the chunks
  - Method: split on different separators, use semantic chunking…
- Change the embedding model

```python
from langchain.vectorstores import FAISS
from langchain_community.embeddings import HuggingFaceEmbeddings
from langchain_community.vectorstores.utils import DistanceStrategy
import os


def load_embeddings(
    langchain_docs: List[LangchainDocument],
    chunk_size: int,
    embedding_model_name: Optional[str] = "thenlper/gte-small",
) -> FAISS:
    """
    Creates a FAISS index from the given embedding model and documents. Loads the index d

    Args:
        langchain_docs: list of documents
        chunk_size: size of the chunks to split the documents into
        embedding_model_name: name of the embedding model to use

    Returns:
        FAISS index
    """
    # load embedding_model
    embedding_model = HuggingFaceEmbeddings(
        model_name=embedding_model_name,
        multi_process=True,
        model_kwargs={"device": "cuda"},
        encode_kwargs={"normalize_embeddings": True},  # set True to compute cosine simil
    )

    # Check if embeddings already exist on disk
    index_name = f"index_chunk:{chunk_size}_embeddings:{embedding_model_name.replace('/',
    index_folder_path = f"./data/indexes/{index_name}/"
    if os.path.isdir(index_folder_path):
        return FAISS.load_local(
            index_folder_path,
            embedding_model,
            distance_strategy=DistanceStrategy.COSINE,
        )

    else:
        print("Index not found, generating it...")
        docs_processed = split_documents(
            chunk_size,
            langchain_docs,
            embedding_model_name,
        )
        knowledge_index = FAISS.from_documents(
            docs_processed, embedding_model, distance_strategy=DistanceStrategy.COSINE
        )
        knowledge_index.save_local(index_folder_path)
        return knowledge_index
```

2.3. Reader - LLM 💬

In this part, the **LLM Reader reads the retrieved documents to formulate its answer.**

🛠️ Here we tried the following options to improve results:

- Switch reranking on/off
- Change the reader model

```
RAG_PROMPT_TEMPLATE = """
<|system|>
Using the information contained in the context,
give a comprehensive answer to the question.
Respond only to the question asked, response should be concise and relevant to the questi
Provide the number of the source document when relevant.
If the answer cannot be deduced from the context, do not give an answer.</s>
<|user|>
Context:
{context}
---
```

```
Now here is the question you need to answer.

Question: {question}
</s>
<|assistant|>
"""
```

```
from langchain_community.llms import HuggingFaceHub

repo_id = "HuggingFaceH4/zephyr-7b-beta"
READER_MODEL_NAME = "zephyr-7b-beta"
HF_API_TOKEN = ""

READER_LLM = HuggingFaceHub(
    repo_id=repo_id,
    task="text-generation",
    huggingfacehub_api_token=HF_API_TOKEN,
    model_kwargs={
        "max_new_tokens": 512,
        "top_k": 30,
        "temperature": 0.1,
        "repetition_penalty": 1.03,
    },
)
```

```
from ragatouille import RAGPretrainedModel
from langchain_core.vectorstores import VectorStore
from langchain_core.language_models.llms import LLM


def answer_with_rag(
    question: str,
    llm: LLM,
    knowledge_index: VectorStore,
    reranker: Optional[RAGPretrainedModel] = None,
    num_retrieved_docs: int = 30,
    num_docs_final: int = 7,
) -> Tuple[str, List[LangchainDocument]]:
    """Answer a question using RAG with the given knowledge index."""
    # Gather documents with retriever
    relevant_docs = knowledge_index.similarity_search(query=question, k=num_retrieved_doc
    relevant_docs = [doc.page_content for doc in relevant_docs]  # keep only the text

    # Optionally rerank results
    if reranker:
        relevant_docs = reranker.rerank(question, relevant_docs, k=num_docs_final)
        relevant_docs = [doc["content"] for doc in relevant_docs]

    relevant_docs = relevant_docs[:num_docs_final]

    # Build the final prompt
    context = "\nExtracted documents:\n"
    context += "".join([f"Document {str(i)}:::\n" + doc for i, doc in enumerate(relevant_

    final_prompt = RAG_PROMPT_TEMPLATE.format(question=question, context=context)

    # Redact an answer
    answer = llm(final_prompt)

    return answer, relevant_docs
```

3. Benchmarking the RAG system

The RAG system and the evaluation datasets are now ready. The last step is to judge the RAG system's output on this evaluation dataset.

To this end, **we setup a judge agent**. ⚖️ 🤖

Out of the different RAG evaluation metrics, we choose to focus only on faithfulness since it the best end-to-end metric of our system's performance.

> *"We use GPT4 as a judge for its empirically good performance, but you could try with other models such as kaist-ai/prometheus-13b-v1.0 or BAAI/JudgeLM-33B-v1.0."*

💡 *In the evaluation prompt, we give a detailed description each metric on the scale 1-5, as is done in Prometheus's prompt template: this helps the model ground its metric precisely. If instead you give the judge LLM a vague scale to work with, the outputs will not be consistent enough between different examples.*

💡 *Again, prompting the LLM to output rationale before giving its final score gives it more tokens to help it formalize and elaborate a judgement.*

```
from langchain_core.language_models import BaseChatModel


def run_rag_tests(
    eval_dataset: datasets.Dataset,
    llm,
    knowledge_index: VectorStore,
    output_file: str,
    reranker: Optional[RAGPretrainedModel] = None,
    verbose: Optional[bool] = True,
    test_settings: Optional[str] = None,  # To document the test settings used
):
    """Runs RAG tests on the given dataset and saves the results to the given output file
    try:  # load previous generations if they exist
        with open(output_file, "r") as f:
            outputs = json.load(f)
    except:
        outputs = []

    for example in tqdm(eval_dataset):
        question = example["question"]
        if question in [output["question"] for output in outputs]:
            continue

        answer, relevant_docs = answer_with_rag(question, llm, knowledge_index, reranker=
        if verbose:
            print("=======================================================")
            print(f"Question: {question}")
            print(f"Answer: {answer}")
            print(f'True answer: {example["answer"]}')
        result = {
            "question": question,
            "true_answer": example["answer"],
            "source_doc": example["source_doc"],
            "generated_answer": answer,
            "retrieved_docs": [doc for doc in relevant_docs],
        }
        if test_settings:
            result["test_settings"] = test_settings
        outputs.append(result)

        with open(output_file, "w") as f:
            json.dump(outputs, f)
```

```
EVALUATION_PROMPT = """###Task Description:
An instruction (might include an Input inside it), a response to evaluate, a reference an
1. Write a detailed feedback that assess the quality of the response strictly based on th
2. After writing a feedback, write a score that is an integer between 1 and 5. You should
3. The output format should look as follows: \"Feedback: {{write a feedback for criteria}
4. Please do not generate any other opening, closing, and explanations. Be sure to includ

###The instruction to evaluate:
{instruction}

###Response to evaluate:
{response}

###Reference Answer (Score 5):
{reference_answer}

###Score Rubrics:
[Is the response correct, accurate, and factual based on the reference answer?]
Score 1: The response is completely incorrect, inaccurate, and/or not factual.
Score 2: The response is mostly incorrect, inaccurate, and/or not factual.
Score 3: The response is somewhat correct, accurate, and/or factual.
Score 4: The response is mostly correct, accurate, and factual.
Score 5: The response is completely correct, accurate, and factual.

###Feedback:"""

from langchain.prompts.chat import (
    ChatPromptTemplate,
    HumanMessagePromptTemplate,
)
from langchain.schema import SystemMessage


evaluation_prompt_template = ChatPromptTemplate.from_messages(
    [
        SystemMessage(content="You are a fair evaluator language model."),
        HumanMessagePromptTemplate.from_template(EVALUATION_PROMPT),
    ]
)
```

```
from langchain.chat_models import ChatOpenAI

OPENAI_API_KEY = ""

eval_chat_model = ChatOpenAI(model="gpt-4-1106-preview", temperature=0, openai_api_key=OP
evaluator_name = "GPT4"


def evaluate_answers(
    answer_path: str,
    eval_chat_model,
    evaluator_name: str,
    evaluation_prompt_template: ChatPromptTemplate,
) -> None:
    """Evaluates generated answers. Modifies the given answer file in place for better ch
    answers = []
    if os.path.isfile(answer_path):  # load previous generations if they exist
        answers = json.load(open(answer_path, "r"))

    for experiment in tqdm(answers):
        if f"eval_score_{evaluator_name}" in experiment:
            continue

        eval_prompt = evaluation_prompt_template.format_messages(
            instruction=experiment["question"],
            response=experiment["generated_answer"],
            reference_answer=experiment["true_answer"],
        )
        eval_result = eval_chat_model.invoke(eval_prompt)
        feedback, score = [item.strip() for item in eval_result.content.split("[RESULT]")
        experiment[f"eval_score_{evaluator_name}"] = score
        experiment[f"eval_feedback_{evaluator_name}"] = feedback

        with open(answer_path, "w") as f:
            json.dump(answers, f)
```

🚀 Let’s run the tests and evaluate answers! 👇

```
if not os.path.exists("./output"):
    os.mkdir("./output")

for chunk_size in [200]:  # Add other chunk sizes (in tokens) as needed
    for embeddings in ["thenlper/gte-small"]:  # Add other embeddings as needed
        for rerank in [True, False]:
            settings_name = f"chunk:{chunk_size}_embeddings:{embeddings.replace('/', '~')
            output_file_name = f"./output/rag_{settings_name}.json"

            print(f"Running evaluation for {settings_name}:")

            print("Loading knowledge base embeddings...")
            knowledge_index = load_embeddings(
                RAW_KNOWLEDGE_BASE,
                chunk_size=chunk_size,
                embedding_model_name=embeddings,
            )

            print("Running RAG...")
            reranker = RAGPretrainedModel.from_pretrained("colbert-ir/colbertv2.0") if re
            run_rag_tests(
                eval_dataset=eval_dataset,
                llm=READER_LLM,
                knowledge_index=knowledge_index,
                output_file=output_file_name,
                reranker=reranker,
                verbose=False,
                test_settings=settings_name,
            )

            print("Running evaluation...")
            evaluate_answers(
                output_file_name,
                eval_chat_model,
                evaluator_name,
                evaluation_prompt_template,
            )
```

Inspect results

```
import glob

outputs = []
for file in glob.glob("./output/*.json"):
    output = pd.DataFrame(json.load(open(file, "r")))
    output["settings"] = file
    outputs.append(output)
result = pd.concat(outputs)
```

```
result["eval_score_GPT4"] = result["eval_score_GPT4"].apply(lambda x: int(x) if isinstanc
result["eval_score_GPT4"] = (result["eval_score_GPT4"] - 1) / 4
```

```
average_scores = result.groupby("settings")["eval_score_GPT4"].mean()
average_scores.sort_values()
```

Example results

Let us load the results that I obtained by tweaking the different options available in this notebook. For more detail on why these options could work or not, see the notebook on advanced_RAG.

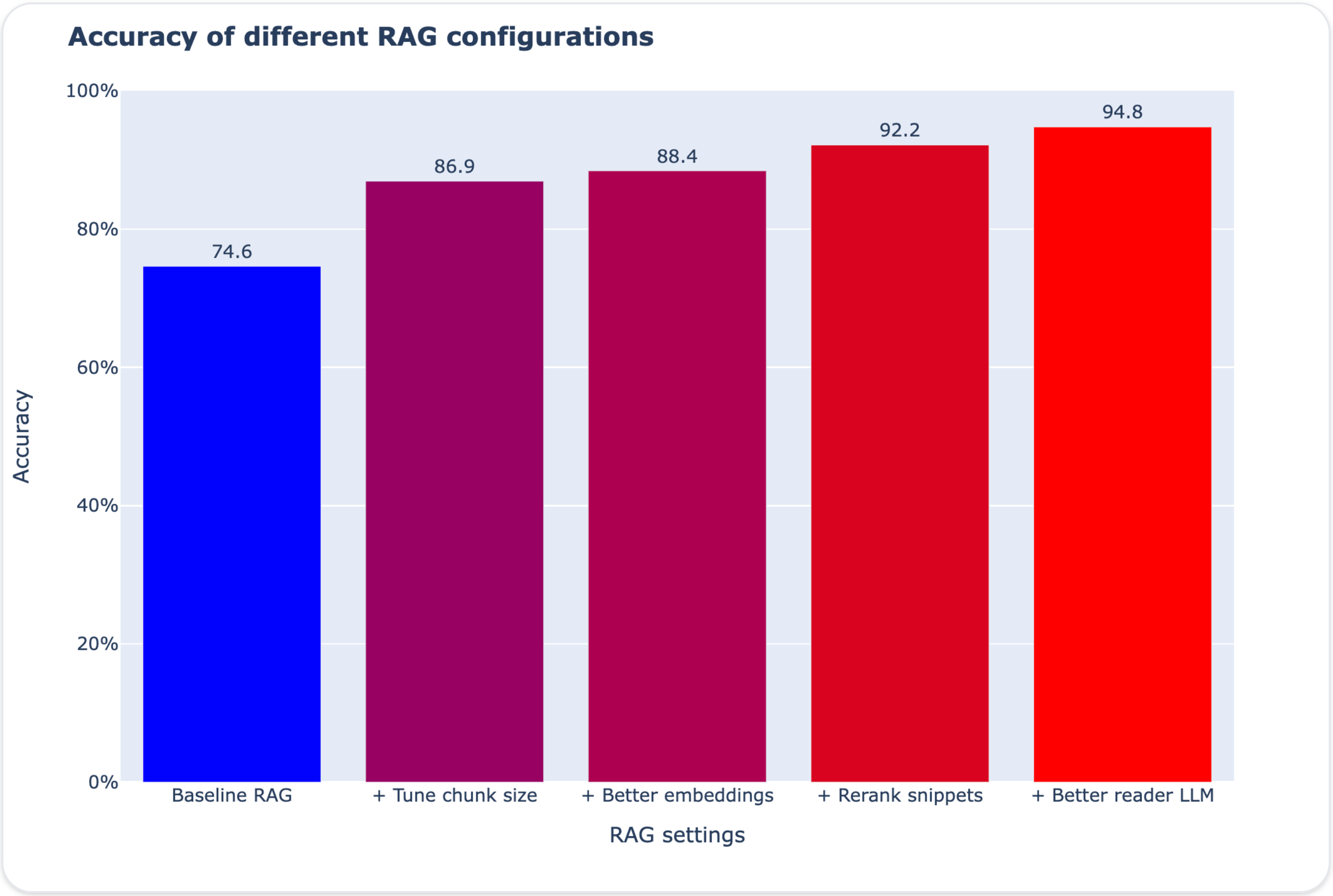
As you can see in the graph below, some tweaks do not bring any improvement, some give huge performance boosts.

➡️ ***There is no single good recipe: you should try several different directions when tuning your RAG systems.***

```
import plotly.express as px

scores = datasets.load_dataset("m-ric/rag_scores_cookbook", split="train")
scores = pd.Series(scores["score"], index=scores["settings"])
```

```
fig = px.bar(
    scores,
    color=scores,
    labels={
        "value": "Accuracy",
        "settings": "Configuration",
    },
    color_continuous_scale="bluered",
)
fig.update_layout(
    width=1000,
    height=600,
    barmode="group",
    yaxis_range=[0, 100],
    title="<b>Accuracy of different RAG configurations</b>",
    xaxis_title="RAG settings",
    font=dict(size=15),
)
fig.layout.yaxis.ticksuffix = "%"
fig.update_coloraxes(showscale=False)
fig.update_traces(texttemplate="%{y:.1f}", textposition="outside")
fig.show()
```



As you can see, these had varying impact on performance. In particular, tuning the chunk size is both easy and very impactful.

But this is our case: your results could be very different: now that you have a robust evaluation pipeline, you can set on to explore other options! 🗺️

<> Update on GitHub

← RAG with Hugging Face and Milvus

Using LLM-as-a-judge for an automated and versatile evaluation →