Building Evaluation from Scratch

We show how you can build evaluation modules from scratch. This includes both evaluation of the final generated response (where the output is plain text), as well as the evaluation of retrievers (where the output is a ranked list of items).

We have in-house modules in our Evaluation section.

Setup

We load some data and define a very simple RAG query engine that we'll evaluate (uses top-k retrieval).

```
!mkdir data
!wget --user-agent "Mozilla" "https://arxiv.org/pdf/2307.09288.pdf" -O "data/llama2.pdf"
```

```
mkdir: data: File exists
--2023-09-19 00:05:14--  https://arxiv.org/pdf/2307.09288.pdf
Resolving arxiv.org (arxiv.org)... 128.84.21.199
Connecting to arxiv.org (arxiv.org)|128.84.21.199|:443... connected.
HTTP request sent, awaiting response... 200 OK
Length: 13661300 (13M) [application/pdf]
Saving to: 'data/llama2.pdf'

data/llama2.pdf     100%[===================>]  13.03M  1.56MB/s    in 9.3s

2023-09-19 00:05:25 (1.40 MB/s) - 'data/llama2.pdf' saved [13661300/13661300]
```

```
from pathlib import Path
from llama_hub.file.pymu_pdf.base import PyMuPDFReader
```

```
loader = PyMuPDFReader()
documents = loader.load(file_path="./data/llama2.pdf")
```

```
from llama_index import VectorStoreIndex, ServiceContext
from llama_index.node_parser import SimpleNodeParser
from llama_index.llms import OpenAI
```

```
llm = OpenAI(model="gpt-4")
node_parser = SimpleNodeParser.from_defaults(chunk_size=1024)
service_context = ServiceContext.from_defaults(llm=llm)
```

```
nodes = node_parser.get_nodes_from_documents(documents)
```

↑ Back to top

```
index = VectorStoreIndex(nodes, service_context)
```

```
query_engine = index.as_query_engine()
```

Dataset Generation

We first go through an exercise of generating a synthetic evaluation dataset. We do this by synthetically generating a set of questions from existing context. We then run each question with existing context through a powerful LLM (e.g. GPT-4) to generate a "ground-truth" response.

Define Functions

We define the functions that we will use for dataset generation:

```
from llama_index.schema import BaseNode
from llama_index.llms import OpenAI
from llama_index.prompts import (
    ChatMessage,
    ChatPromptTemplate,
    MessageRole,
    PromptTemplate,
)
from typing import Tuple, List
import re

llm = OpenAI(model="gpt-4")
```

We define `generate_answers_for_questions` to generate answers from questions given context.

```
QA_PROMPT = PromptTemplate(
    "Context information is below.\n"
    "---------------------\n"
    "{context_str}\n"
    "---------------------\n"
    "Given the context information and not prior knowledge, "
    "answer the query.\n"
    "Query: {query_str}\n"
    "Answer: "
)


def generate_answers_for_questions(
    questions: List[str], context: str, llm: OpenAI
) -> str:
    """Generate answers for questions given context."""
    answers = []
    for question in questions:
        fmt_qa_prompt = QA_PROMPT.format(context_str=context, query_str=question)
        response_obj = llm.complete(fmt_qa_prompt)
        answers.append(str(response_obj))
    return answers
```

↑ Back to top

We define `generate_qa_pairs` to generate qa pairs over an entire list of Nodes.

```
QUESTION_GEN_USER_TMPL = (
    "Context information is below.\n"
    "---------------------\n"
    "{context_str}\n"
    "---------------------\n"
    "Given the context information and not prior knowledge, "
    "generate the relevant questions. "
)

QUESTION_GEN_SYS_TMPL = """\
You are a Teacher/ Professor. Your task is to setup \
{num_questions_per_chunk} questions for an upcoming \
quiz/examination. The questions should be diverse in nature \
across the document. Restrict the questions to the \
context information provided.\
"""

question_gen_template = ChatPromptTemplate(
    message_templates=[
        ChatMessage(role=MessageRole.SYSTEM, content=QUESTION_GEN_SYS_TMPL),
        ChatMessage(role=MessageRole.USER, content=QUESTION_GEN_USER_TMPL),
    ]
)


def generate_qa_pairs(
    nodes: List[BaseNode], llm: OpenAI, num_questions_per_chunk: int = 10
) -> List[Tuple[str, str]]:
    """Generate questions."""
    qa_pairs = []
    for idx, node in enumerate(nodes):
        print(f"Node {idx}/{len(nodes)}")
        context_str = node.get_content(metadata_mode="all")
        fmt_messages = question_gen_template.format_messages(
            num_questions_per_chunk=10,
            context_str=context_str,
        )
        chat_response = llm.chat(fmt_messages)
        raw_output = chat_response.message.content
        result_list = str(raw_output).strip().split("\n")
        cleaned_questions = [
            re.sub(r"^\d+[\).\s]", "", question).strip() for question in result_list
        ]
        answers = generate_answers_for_questions(cleaned_questions, context_str, llm)
        cur_qa_pairs = list(zip(cleaned_questions, answers))
        qa_pairs.extend(cur_qa_pairs)
    return qa_pairs
```

```
qa_pairs
```

```
[('What is the main focus of the work described in the document?',
  'The main focus of the work described in the document is the development and release of Llam
 ('What is the range of parameters for t uage models (LLMs) developed in this work
  'The range of parameters for the larg els (LLMs) developed in this work is from
 ('What is the specific name given to the fine-tuned LLMs optimized for dialogue use cases?',
  'The specific name given to the fine-tuned LLMs optimized for dialogue use cases is Llama 2-
 ('How do the models developed in this work compare to open-source chat models based on the be
  'The models developed in this work, specifically the fine-tuned LLMs called Llama 2-Chat, ou
 ('What are the two key areas of human evaluation mentioned in the document for the developed
  'The two key areas of human evaluation mentioned in the document for the developed models ar
 ('What is the purpose of providing a detailed description of the approach to fine-tuning and
  'The purpose of providing a detailed description of the approach to fine-tuning and safety i
 ('What is the intended benefit for the community from this work?',
  'The intended benefit for the community from this work is to enable them to build on the wor
 ('Who are the corresponding authors of this work and how can they be contacted?',
  'The corresponding authors of this work are Thomas Scialom and Hugo Touvron. They can be con
 ('What is the source of the document and how many pages does it contain?',
  'The source of the document is "1" and it contains 77 pages.'),
 ('Where can the contributions of all the authors be found in the document?',
  'The contributions of all the authors can be found in Section A.1 of the document.')]
```


Getting Pairs over Dataset

NOTE: This can take a long time. For the sake of speed try inputting a subset of the nodes.

```
qa_pairs = generate_qa_pairs(
    # nodes[:1],
    nodes,
    llm,
    num_questions_per_chunk=10,
)
```

[Optional] Define save/load

```
# save
import pickle

pickle.dump(qa_pairs, open("eval_dataset.pkl", "wb"))
```

```
# save
import pickle

qa_pairs = pickle.load(open("eval_dataset.pkl", "rb"))
```

Evaluating Generation

In this section we walk through a few methods for evaluating the generated results level we use an "evaluation LLM" to measure the quality of the generated results. W both the **with labels** setting and **without labels** setting.

We go through the following evaluation algorithms:

- **Correctness**: Compares the generated answer against the ground-truth answer.
- **Faithfulness**: Evaluates whether a response is faithful to the contexts (label-free).

Building a Correctness Evaluator

The correctness evaluator compares the generated answer to the reference ground-truth answer, given the query. We output a score between 1 and 5, where 1 is the worst and 5 is the best.

We do this through a system and user prompt with a chat interface.

```
from llama_index.prompts import (
    ChatMessage,
    ChatPromptTemplate,
    MessageRole,
    PromptTemplate,
)
from typing import Dict
```

```
CORRECTNESS_SYS_TMPL = """
You are an expert evaluation system for a question answering chatbot.

You are given the following information
- a user query,
- a reference answer, and
- a generated answer.

Your job is to judge the relevance and correctness of the generated answer.
Output a single score that represents a holistic evaluation.
You must return your response in a line with only the score.
Do not return answers in any other format.
On a separate line provide your reasoning for the score as well.

Follow these guidelines for scoring:
- Your score has to be between 1 and 5, where 1 is the worst and 5 is the best.
- If the generated answer is not relevant to the user query, \
you should give a score of 1.
- If the generated answer is relevant but contains mistakes, \
you should give a score between 2 and 3.
- If the generated answer is relevant and fully correct, \
you should give a score between 4 and 5.
"""

CORRECTNESS_USER_TMPL = """
## User Query
{query}

## Reference Answer
{reference_answer}

## Generated Answer
{generated_answer}
"""
```

↑ Back to top

```
eval_chat_template = ChatPromptTemplate(
    message_templates=[
        ChatMessage(role=MessageRole.SYSTEM, content=CORRECTNESS_SYS_TMPL),
        ChatMessage(role=MessageRole.USER, content=CORRECTNESS_USER_TMPL),
    ]
)
```

Now that we've defined the prompts template, let's define an evaluation function that feeds the prompt to the LLM and parses the output into a dict of results.

CTRL + K

```
from llama_index.llms import OpenAI


def run_correctness_eval(
    query_str: str,
    reference_answer: str,
    generated_answer: str,
    llm: OpenAI,
    threshold: float = 4.0,
) -> Dict:
    """Run correctness eval."""
    fmt_messages = eval_chat_template.format_messages(
        llm=llm,
        query=query_str,
        reference_answer=reference_answer,
        generated_answer=generated_answer,
    )
    chat_response = llm.chat(fmt_messages)
    raw_output = chat_response.message.content

    # Extract from response
    score_str, reasoning_str = raw_output.split("\n", 1)
    score = float(score_str)
    reasoning = reasoning_str.lstrip("\n")

    return {"passing": score >= threshold, "score": score, "reason": reasoning}
```

↑ Back to top

Now let's try running this on some sample inputs with a chat model (GPT-4).

```
llm = OpenAI(model="gpt-4")
```

```
# query_str = "What is the range of parameters for the large language models (LLMs) developed
# reference_answer = "The range of parameters for the large language models (LLMs) developed i

query_str = "What is the specific name given to the fine-tuned LLMs optimized for dialogue use
reference_answer = "The specific name given to the fine-tuned LLMs optimized for dialogue use
```

```
generated_answer = str(query_engine.query(query_str))
```

```
print(str(generated_answer))
```

```
The fine-tuned Large Language Models (LLMs) optimized for dialogue use cases are specifically
```

CTRL + K

```
eval_results = run_correctness_eval(
    query_str, reference_answer, generated_answer, llm=llm, threshold=4.0
)
display(eval_results)
```

↑ Back to top

```
{'passing': True,
 'score': 5.0,
 'reason': 'The generated answer is completely relevant to the user query and matches the refe
```

Building a Faithfulness Evaluator

The faithfulness evaluator evaluates whether the response is faithful to any of the retrieved contexts.

This is a step up in complexity from the correctness evaluator. Since the set of contexts can be quite long, they might overflow the context window. We would need to figure out how to implement a form of **response synthesis** strategy to iterate over contexts in sequence.

We have a corresponding tutorial showing you how to build response synthesis from scratch. We also have out-of-the-box response synthesis modules. In this guide we'll use the out of the box modules.

```
EVAL_TEMPLATE = PromptTemplate(
    "Please tell if a given piece of information "
    "is supported by the context.\n"
    "You need to answer with either YES
    "Answer YES if any of the context supports the information, even "
    "if most of the context is unrelated. "
    "Some examples are provided below. \n\n"
    "Information: Apple pie is generally double-crusted.\n"
    "Context: An apple pie is a fruit pie in which the principal filling "
    "ingredient is apples. \n"
    "Apple pie is often served with whipped cream, ice cream "
    "('apple pie à la mode'), custard or cheddar cheese.\n"
    "It is generally double-crusted, with pastry both above "
    "and below the filling; the upper crust may be solid or "
    "latticed (woven of crosswise strips).\n"
    "Answer: YES\n"
    "Information: Apple pies tastes bad.\n"
    "Context: An apple pie is a fruit pie in which the principal filling "
    "ingredient is apples. \n"
    "Apple pie is often served with whipped cream, ice cream "
    "('apple pie à la mode'), custard or cheddar cheese.\n"
    "It is generally double-crusted, with pastry both above "
    "and below the filling; the upper crust may be solid or "
    "latticed (woven of crosswise strips).\n"
    "Answer: NO\n"
    "Information: {query_str}\n"
    "Context: {context_str}\n"
    "Answer: "
)

EVAL_REFINE_TEMPLATE = PromptTemplate(
    "We want to understand if the following information is present "
    "in the context information: {query_str}\n"
    "We have provided an existing YES/NO answer: {existing_answer}\n"
    "We have the opportunity to refine the existing answer "
    "(only if needed) with some more context below.\n"
    "------------\n"
    "{context_msg}\n"
    "------------\n"
    "If the existing answer was already YES, still answer YES. "
    "If the information is present in the new context, answer YES. "
    "Otherwise answer NO.\n"
)
```

↑ Back to top

NOTE: In the current response synthesizer setup we don't separate out a system and user message for chat endpoints, so we just use our standard `llm.complete` for text completion.

We now define our function below. Since we defined both a standard eval template piece of context but also a refine template for subsequent contexts, we implement and-refine" response synthesis strategy to obtain the answer.

```
from llama_index.response_synthesizers import Refine
from llama_index import ServiceContext
from typing import List, Dict
```

↑ Back to top

```
def run_faithfulness_eval(
    generated_answer: str,
    contexts: List[str],
    llm: OpenAI,
) -> Dict:
    """Run faithfulness eval."""

    service_context = ServiceContext.from_defaults(llm=llm)
    refine = Refine(
        text_qa_template=EVAL_TEMPLATE,
        refine_template=EVAL_REFINE_TEMPLATE,
    )

    response_obj = refine.get_response(generated_answer, contexts)
    response_txt = str(response_obj)

    if "yes" in response_txt.lower():
        passing = True
    else:
        passing = False

    return {"passing": passing, "reason": str(response_txt)}
```

Let's try it out on some data

```
# use the same query_str, and reference_answer as above
# query_str = "What is the specific name given to the fine-tuned LLMs optimized for dialogue u
# reference_answer = "The specific name given to the fine-tuned LLMs optimized for dialogue us

response = query_engine.query(query_str)
generated_answer = str(response)
```

```
context_list = [n.get_content() for n in response.source_nodes]
eval_results = run_faithfulness_eval(
    generated_answer,
    contexts=context_list,
    llm=llm,
)
display(eval_results)
```

```
{'passing': True, 'reason': 'YES'}
```

CTRL + K

Running Evaluation over our Eval Dataset

Now let's tie the two above sections toget ur eval modules over our eval dataset!

↑ Back to top

NOTE: For the sake of speed/cost we extract a very limited sample.

```
import random

sample_size = 5
qa_pairs_sample = random.sample(qa_pairs, sample_size)
```

```
import pandas as pd


def run_evals(qa_pairs: List[Tuple[str, str]], llm: OpenAI, query_engine):
    results_list = []
    for question, reference_answer in qa_pairs:
        response = query_engine.query(question)
        generated_answer = str(response)
        correctness_results = run_correctness_eval(
            query_str, reference_answer, generated_answer, llm=llm, threshold=4.0
        )
        faithfulness_results = run_faithfulness_eval(
            generated_answer,
            contexts=context_list,
            llm=llm,
        )
        cur_result_dict = {
            "correctness": correctness_results["passing"],
            "faithfulness": faithfulness_results["passing"],
        }
        results_list.append(cur_result_dict)
    return pd.DataFrame(results_list)
```

```
evals_df = run_evals(qa_pairs_sample, llm, query_engine)
```

```
evals_df["correctness"].mean()
```

```
0.4
```

```
evals_df["faithfulness"].mean()
```

```
0.6
```

↑ Back to top