Comparing Chain Outputs

Suppose you have two different prompts (or LLMs). How do you know which will generate "better" results?

One automated way to predict the preferred configuration is to use a `PairwiseStringEvaluator` like the `PairwiseStringEvalChain`[1]. This chain prompts an LLM to select which output is preferred, given a specific input.

For this evaluation, we will need 3 things:

1. An evaluator
2. A dataset of inputs
3. 2 (or more) LLMs, Chains, or Agents to compare

Then we will aggregate the restults to determine the preferred model.

Step 1. Create the Evaluator

In this example, you will use gpt-4 to select which output is preferred.

```
from langchain.evaluation import load_evaluator

eval_chain = load_evaluator("pairwise_string")
```

API Reference:

- load_evaluator from `langchain.evaluation`

Step 2. Select Dataset

If you already have real usage data for your LLM, you can use a representative sample. More examples provide more reliable results. We will use some example queries someone might have about how to use langchain here.

```
from langchain.evaluation.loading import load_dataset

dataset = load_dataset("langchain-howto-queries")
```

API Reference:

- load_dataset from `langchain.evaluation.loading`

```
    Found cached dataset parquet
(/Users/wfh/.cache/huggingface/datasets/LangChainDatasets___parquet/LangCh
langchain-howto-queries-
bbb748bbee7e77aa/0.0.0/14a00e99c0d15a23649d0db8944380ac81082d4b021f398733c



      0%|          | 0/1 [00:00<?, ?it/s]
```

Step 3. Define Models to Compare

We will be comparing two agents in this case.

```
from langchain import SerpAPIWrapper
from langchain.agents import initialize_agent, Tool
from langchain.agents import AgentType
from langchain.chat_models import ChatOpenAI


# Initialize the language model
# You can add your own OpenAI API key by adding openai_api_key="
<your_api_key>"
llm = ChatOpenAI(temperature=0, model="gpt-3.5-turbo-0613")

# Initialize the SerpAPIWrapper for search functionality
# Replace <your_api_key> in openai_api_key="<your_api_key>" with your
actual SerpAPI key.
search = SerpAPIWrapper()

# Define a list of tools offered by the agent
tools = [
    Tool(
        name="Search",
        func=search.run,
        coroutine=search.arun,
        description="Useful when you need to answer questions about
current events. You should ask targeted questions.",
    ),
]
```

API Reference:

- initialize_agent from `langchain.agents`
- Tool from `langchain.agents`
- AgentType from `langchain.agents`
- ChatOpenAI from `langchain.chat_models`

```
functions_agent = initialize_agent(
    tools, llm, agent=AgentType.OPENAI_MULTI_FUNCTIONS, verbose=False
)
conversations_agent = initialize_agent(
    tools, llm, agent=AgentType.CHAT_ZERO_SHOT_REACT_DESCRIPTION,
verbose=False
)
```

Step 4. Generate Responses

We will generate outputs for each of the models before evaluating them.

```
from tqdm.notebook import tqdm
import asyncio

results = []
agents = [functions_agent, conversations_agent]
concurrency_level = 6  # How many concurrent agents to run. May need to
decrease if OpenAI is rate limiting.

# We will only run the first 20 examples of this dataset to speed
things up
# This will lead to larger confidence intervals downstream.
batch = []
for example in tqdm(dataset[:20]):
    batch.extend([agent.acall(example["inputs"]) for agent in agents])
    if len(batch) >= concurrency_level:
        batch_results = await asyncio.gather(*batch,
return_exceptions=True)
        results.extend(list(zip(*[iter(batch_results)] * 2)))
        batch = []
if batch:
    batch_results = await asyncio.gather(*batch,
return_exceptions=True)
    results.extend(list(zip(*[iter(batch_results)] * 2)))
```

```
  0%|          | 0/20 [00:00<?, ?it/s]
```

Step 5. Evaluate Pairs

Now it's time to evaluate the results. For each agent response, run the evaluation chain to select which output is preferred (or return a tie).

Randomly select the input order to reduce the likelihood that one model will be preferred just because it is presented first.

```
import random


def predict_preferences(dataset, results) -> list:
    preferences = []

    for example, (res_a, res_b) in zip(dataset, results):
        input_ = example["inputs"]
        # Flip a coin to reduce persistent position bias
        if random.random() < 0.5:
            pred_a, pred_b = res_a, res_b
            a, b = "a", "b"
        else:
            pred_a, pred_b = res_b, res_a
            a, b = "b", "a"
        eval_res = eval_chain.evaluate_string_pairs(
            prediction=pred_a["output"] if isinstance(pred_a, dict)
else str(pred_a),
            prediction_b=pred_b["output"] if isinstance(pred_b, dict)
else str(pred_b),
            input=input_,
        )
        if eval_res["value"] == "A":
            preferences.append(a)
        elif eval_res["value"] == "B":
            preferences.append(b)
        else:
            preferences.append(None)  # No preference
    return preferences
```

```
preferences = predict_preferences(dataset, results)
```

Print out the ratio of preferences.

```
from collections import Counter

name_map = {
    "a": "OpenAI Functions Agent",
    "b": "Structured Chat Agent",
}
counts = Counter(preferences)
pref_ratios = {k: v / len(preferences) for k, v in counts.items()}
for k, v in pref_ratios.items():
    print(f"{name_map.get(k)}: {v:.2%}")
```

```
    OpenAI Functions Agent: 95.00%
    None: 5.00%
```

Estimate Confidence Intervals

The results seem pretty clear, but if you want to have a better sense of how confident we are, that model "A" (the OpenAI Functions Agent) is the preferred model, we can calculate confidence intervals.

Below, use the Wilson score to estimate the confidence interval.

```
from math import sqrt


def wilson_score_interval(
    preferences: list, which: str = "a", z: float = 1.96
) -> tuple:
    """Estimate the confidence interval using the Wilson score.

    See:
https://en.wikipedia.org/wiki/Binomial_proportion_confidence_interval#Wils
    for more details, including when to use it and when it should not be u
    """
    total_preferences = preferences.count("a") + preferences.count("b")
    n_s = preferences.count(which)

    if total_preferences == 0:
        return (0, 0)

    p_hat = n_s / total_preferences
```

```
    denominator = 1 + (z**2) / total_preferences
    adjustment = (z / denominator) * sqrt(
        p_hat * (1 - p_hat) / total_preferences
        + (z**2) / (4 * total_preferences * total_preferences)
    )
    center = (p_hat + (z**2) / (2 * total_preferences)) / denominator
    lower_bound = min(max(center - adjustment, 0.0), 1.0)
    upper_bound = min(max(center + adjustment, 0.0), 1.0)

    return (lower_bound, upper_bound)
```

```
for which_, name in name_map.items():
    low, high = wilson_score_interval(preferences, which=which_)
    print(
        f'The "{name}" would be preferred between {low:.2%} and
{high:.2%} percent of the time (with 95% confidence).'
    )
```

```
    The "OpenAI Functions Agent" would be preferred between 83.18% and
100.00% percent of the time (with 95% confidence).
    The "Structured Chat Agent" would be preferred between 0.00% and
16.82% percent of the time (with 95% confidence).
```

Print out the p-value.

```
from scipy import stats

preferred_model = max(pref_ratios, key=pref_ratios.get)
successes = preferences.count(preferred_model)
n = len(preferences) - preferences.count(None)
p_value = stats.binom_test(successes, n, p=0.5, alternative="two-
sided")
print(
    f"""The p-value is {p_value:.5f}. If the null hypothesis is true
(i.e., if the selected eval chain actually has no preference between
the models),
then there is a {p_value:.5%} chance of observing the
{name_map.get(preferred_model)} be preferred at least {successes}
times out of {n} trials."""
)
```

```
    The p-value is 0.00000. If the null hypothesis is true (i.e., if the
selected eval chain actually has no preference between the models),
    then there is a 0.00038% chance of observing the OpenAI Functions Ager
preferred at least 19
    times out of 19 trials.



/var/folders/gf/6rnp_mbx5914kx7qmmh7xzmw0000gn/T/ipykernel_15978/38490768
DeprecationWarning: 'binom_test' is deprecated in favour of 'binomtest' fr
version 1.7.0 and will be removed in Scipy 1.12.0.
      p_value = stats.binom_test(successes, n, p=0.5, alternative="two-sic
```

1. Note: Automated evals are still an open research topic and are best used alongside other evaluation approaches. LLM preferences exhibit biases, including banal ones like the order of outputs. In choosing preferences, "ground truth" may not be taken into account, which may lead to scores that aren't grounded in utility.