Custom Pairwise Evaluator

You can make your own pairwise string evaluators by inheriting from `PairwiseStringEvaluator` class and overwriting the `_evaluate_string_pairs` method (and the `_aevaluate_string_pairs` method if you want to use the evaluator asynchronously).

In this example, you will make a simple custom evaluator that just returns whether the first prediction has more whitespace tokenized 'words' than the second.

You can check out the reference docs for the PairwiseStringEvaluator interface for more info.

```
from typing import Optional, Any
from langchain.evaluation import PairwiseStringEvaluator


class LengthComparisonPairwiseEvalutor(PairwiseStringEvaluator):
    """
    Custom evaluator to compare two strings.
    """

    def _evaluate_string_pairs(
        self,
        *,
        prediction: str,
        prediction_b: str,
        reference: Optional[str] = None,
        input: Optional[str] = None,
        **kwargs: Any,
    ) -> dict:
        score = int(len(prediction.split()) >
len(prediction_b.split()))
        return {"score": score}
```

API Reference:

- PairwiseStringEvaluator from `langchain.evaluation`

```
evaluator = LengthComparisonPairwiseEvalutor()
```

```
evaluator.evaluate_string_pairs(
    prediction="The quick brown fox jumped over the lazy dog.",
    prediction_b="The quick brown fox jumped over the dog.",
)
```

```
    {'score': 1}
```

LLM-Based Example

That example was simple to illustrate the API, but it wasn't very useful in practice. Below, use an LLM with some custom instructions to form a simple preference scorer similar to the built-in PairwiseStringEvalChain. We will use `ChatAnthropic` for the evaluator chain.

```
# %pip install anthropic
# %env ANTHROPIC_API_KEY=YOUR_API_KEY
```

```
from typing import Optional, Any
from langchain.evaluation import PairwiseStringEvaluator
from langchain.chat_models import ChatAnthropic
from langchain.chains import LLMChain


class CustomPreferenceEvaluator(PairwiseStringEvaluator):
    """
    Custom evaluator to compare two strings using a custom LLMChain.
    """

    def __init__(self) -> None:
        llm = ChatAnthropic(model="claude-2", temperature=0)
        self.eval_chain = LLMChain.from_string(
            llm,
            """Which option is preferred? Do not take order into
account. Evaluate based on accuracy and helpfulness. If neither is
preferred, respond with C. Provide your reasoning, then finish with
Preference: A/B/C

Input: How do I get the path of the parent directory in python 3.8?
Option A: You can use the following code:
```python
import os
```
```

```
os.path.dirname(os.path.dirname(os.path.abspath(__file__)))
```

API Reference:

- PairwiseStringEvaluator from `langchain.evaluation`
- ChatAnthropic from `langchain.chat_models`
- LLMChain from `langchain.chains`

Option B: You can use the following code:

```
from pathlib import Path
Path(__file__).absolute().parent
```

Reasoning: Both options return the same result. However, since option B is more concise and easily understand, it is preferred. Preference: B

Which option is preferred? Do not take order into account. Evaluate based on accuracy and helpfulness. If neither is preferred, respond with C. Provide your reasoning, then finish with Preference: A/B/C Input: {input} Option A: {prediction} Option B: {prediction_b} Reasoning:""",)

```
@property
def requires_input(self) -> bool:
    return True

@property
def requires_reference(self) -> bool:
    return False

def _evaluate_string_pairs(
    self,
    *,
    prediction: str,
    prediction_b: str,
    reference: Optional[str] = None,
    input: Optional[str] = None,
    **kwargs: Any,
) -> dict:
    result = self.eval_chain(
        {
            "input": input,
            "prediction": prediction,
            "prediction_b": prediction_b,
```

```
            "stop": ["Which option is preferred?"],
        },
        **kwargs,
    )

    response_text = result["text"]
    reasoning, preference = response_text.split("Preference:",
maxsplit=1)
    preference = preference.strip()
    score = 1.0 if preference == "A" else (0.0 if preference == "B"
else None)
    return {"reasoning": reasoning.strip(), "value": preference,
"score": score}
```

```python
evaluator = CustomPreferenceEvaluator()
```

```
evaluator.evaluate_string_pairs(
    input="How do I import from a relative directory?",
    prediction="use importlib! importlib.import_module('.my_package',
'.')",
    prediction_b="from .sibling import foo",
)
```

```
    {'reasoning': 'Option B is preferred over option A for importing
from a relative directory, because it is more straightforward and
concise.\n\nOption A uses the importlib module, which allows importing
a module by specifying the full name as a string. While this works, it
is less clear compared to option B.\n\nOption B directly imports from
the relative path using dot notation, which clearly shows that it is a
relative import. This is the recommended way to do relative imports in
Python.\n\nIn summary, option B is more accurate and helpful as it uses
the standard Python relative import syntax.',
     'value': 'B',
     'score': 0.0}
```

```
# Setting requires_input to return True adds additional validation to
avoid returning a grade when insufficient data is provided to the
chain.
```

```
try:
    evaluator.evaluate_string_pairs(
        prediction="use importlib!
importlib.import_module('.my_package', '.')",
        prediction_b="from .sibling import foo",
    )
except ValueError as e:
    print(e)
```

```
    CustomPreferenceEvaluator requires an input string.
```