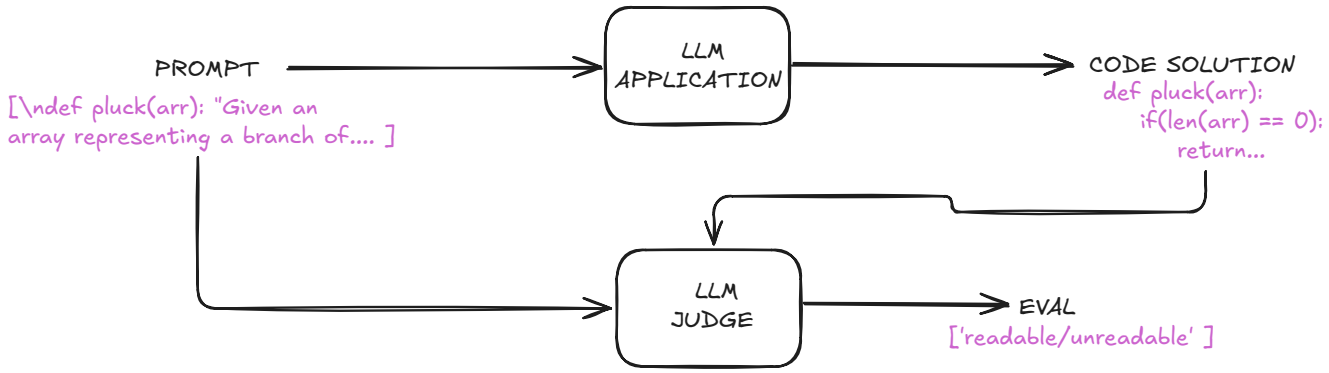
LLM as a Judge for Code Generation

Author : Vineet Sunil Gattani

Traditionally, evaluating generated code relies on **human review**, where experts assess correctness, readability, and efficiency. However, as **large language models (LLMs)** improve, high-quality code generation is becoming cheaper and faster, making human evaluation difficult to scale. In this context, the best way to assess the performance of **LLM-based code generation** is by using a separate **evaluation LLM**.

The **Phoenix LLM Evals** library provides a simple, fast, and accurate framework for **automated code evaluation**, helping developers assess readability, and adherence to best practices without requiring extensive human oversight.



The following project can be implemented using the `python` notebook in GitHub repository [LLM-Evaluator-Code-Generation](https://github.com/vineet0814/LLM-Evaluator-Code-Generation)

Part 1: How Good is LLM as a Judge ?

In this section, we aim to quantify the reliability of an LLM as a judge—that is, given ground truth labels, how accurately the LLM Judge can predict them. The below steps follow Part 1 in the `python` notebook.

Step 1: To do this, we look at a benchmark dataset OpenAI Human Eval Dataset which consists of following columns:

- `prompts` : Description of a task
- `canonical_solution` : Ground truth solution
- `solution` : Code generated by a LLM application for a given `prompt` . Given this is an OpenAI dataset, the solution generated is most likely from an OpenAI model (for instance, `gpt-4` or `gpt-3.5-turbo`)
- `readable` : A boolean field which states if the `solution` generated by LLM application is readable or not. These labels are generated by expert humans.

named: 0	task_id	prompt	canonical_solution	test	entry_point	readable
0	HumanEval/0	from typing import List\n\n\ndef has_close_elements(numbers: List[float], threshold: float) -> bool:\n """ Check if in given list of numbers, are any two numbers closer to each other than\n given threshold.\n >>> has_close_elements([1.0, 2.0, 3.0], 0.5)\n False\n >>> has_close_elements([1.0, 2.8, 3.0, 4.0, 5.0, 2.0], 0.3)\n True\n """\n	for idx, elem in enumerate(numbers):\n for idx2, elem2 in enumerate(numbers):\n if idx != idx2:\n distance = abs(elem - elem2)\n if distance < threshold:\n return True\n\n return False\n	\n\nMETADATA = {\n 'author': 'jt',\n 'dataset': 'test'\n}\n\n\ndef check(candidate):\n assert candidate([1.0, 2.0, 3.9, 4.0, 5.0, 2.2], 0.3) == True\n assert candidate([1.0, 2.0, 3.9, 4.0, 5.0, 2.2], 0.05) == False\n assert candidate([1.0, 2.0, 5.9, 4.0, 5.0], 0.95) == True\n assert candidate([1.0, 2.0, 5.9, 4.0, 5.0], 0.8) == False\n assert candidate([1.0, 2.0, 3.0, 4.0, 5.0, 2.0], 0.1) == True\n assert candidate([1.1, 2.2, 3.1, 4.1, 5.1], 1.0) == True\n assert candidate([1.1, 2.2, 3.1, 4.1, 5.1], 0.5) == False\n\n	has_close_elements	True

Step 2: To evaluate the `solution` generated by LLM application, we can use the following evaluation template which can used readily as a part of **Phoenix LLM Evals** library.

```
You are a stern but practical senior software engineer who cares a lot about simplicity and
readability of code. Can you review the following code that was written by another engineer?
Focus on readability of the code. Respond with "readable" if you think the code is readable,
or "unreadable" if the code is unreadable or needlessly complex for what it's trying
to accomplish.

ONLY respond with "readable" or "unreadable"

Task Assignment:

{query}

Implementation to Evaluate:

{code}
```

Step 3: After loading the dataset, finalizing the evaluation template, the next step is to choose a LLM that will act as Judge. For this project I choose, MISTRAL AI's `codestral-mamba-2407`. The reason I picked this is because:

- *Optimized for Code*: Trained specifically on programming languages (`Python`, `Java` etc.), making it more accurate for code-related tasks.
- *Uses Mamba Instead of Transformers*: Unlike models like `gpt-4`, `codestral` is built on **Mamba**, which is more efficient for long sequences.
- *Better Handling of Long Code Contexts*: Mamba's structured state-space layers allow it to handle long-range dependencies efficiently.

Configure the LLM Judge / Evaluator

Configure your MISTRALAI API key.

```
[ ]  1  if not (mistralai_api_key := os.getenv("MISTRALAI_API_KEY")):
     2      mistralai_api_key = getpass("🔑 Enter your MISTRALAI API key: ")
     3  os.environ["MISTRALAI_API_KEY"] = mistralai_api_key
```

```
🔑 Enter your MISTRALAI API key: ··········
```

Instantiate the LLM Judge / Evaluator and set parameters.

```
[ ]  1  model = MistralAIModel(
     2      model="codestral-mamba-2407",
     3      temperature=0.0,
     4      api_key=mistralai_api_key,
     5  )
```

Simple test to show that we are able call the model

```
[ ]  1  model("Hello, are you ready to classify code as readable or not?")
```

```
'Hello! I'm here to help you with your question. However, I need more information to provide a meaningful response. Could you please pr
o classify as readable or not? Additionally, it would be helpful if you could specify the programming language of the code.'
```

Step 4: Now we can run the code generation evaluation by using the `llm_classify` method from the **Phoenix LLM Evals** library.

LLM Code Readability Classifications

Run readability classifications against a subset of the data.

```
[ ]  1  # The rails is used to hold the output to specific values based on the template
     2  # It will remove text such as ",,," or "..."
     3  # Will ensure the binary value expected from the template is returned
     4  rails = list(CODE_READABILITY_PROMPT_RAILS_MAP.values())
     5  readability_classifications = llm_classify(
     6      dataframe=df,
     7      template=CODE_READABILITY_PROMPT_TEMPLATE,
     8      model=model,
     9      rails=rails,
    10      concurrency=1,
    11      verbose=True,
    12      provide_explanation=True,
    13      max_retries=20,
    14      run_sync=False,
    15  )["label"].tolist()
```

dataset (HumanEval)

Eval Template

LLM Judge (Codestral)

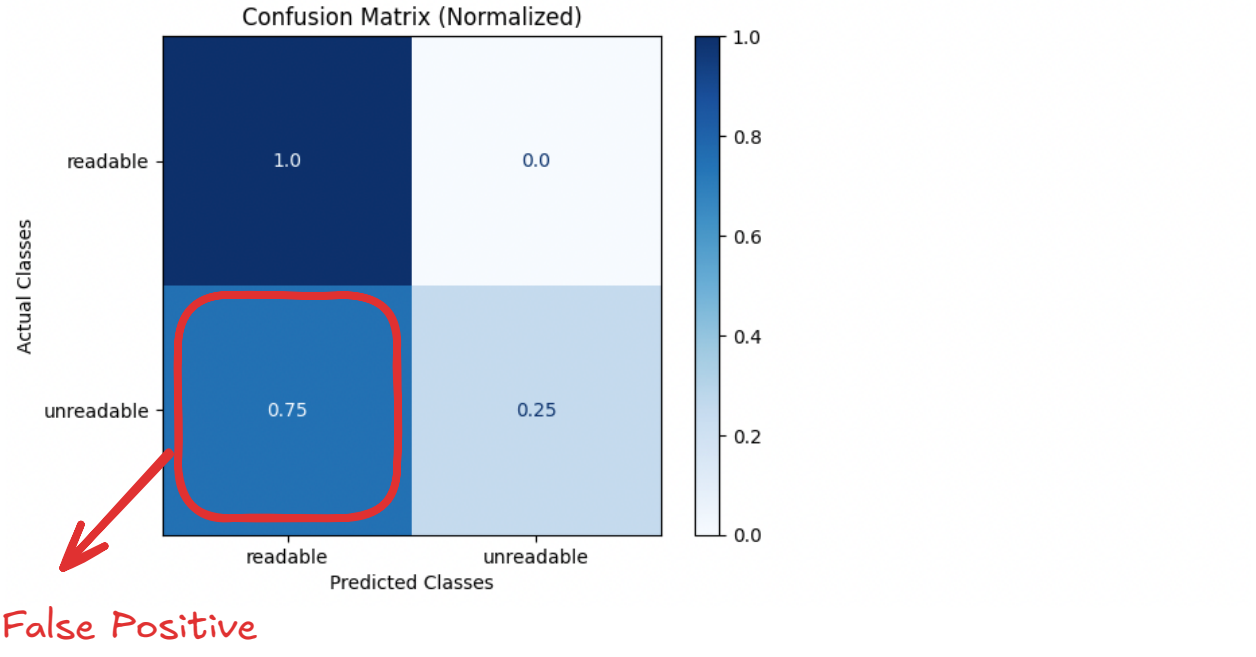
```
Using prompt:

[PromptPartTemplate(content_type=<PromptPartContentType.TEXT: 'text'>, template='\nYou are
<ipython-input-11-56a93cf1f06c>:5: DeprecationWarning: `dataframe` argument is deprecated;
  readability_classifications = llm_classify(
llm_classify: 20/20 (100.0%) | ⌛ 02:34<00:00 | 3.82s/it
- Snapped 'readable"' to rail: readable
- Snapped 'unreadable"' to rail: unreadable
- Snapped 'unreadable"' to rail: unreadable
- Snapped 'readable"\n\nExplanation: The code is simple, straightforward, and easy to unde
- Snapped 'readable"\n\nExplanation: The code is now more readable because it uses a clear
```

Classification of code readable or not w/ explanation

Step 5: Now we are set to interpret results and compare it with our ground-truth readability labels.

	precision	recall	f1-score	support
readable	0.44	1.00	0.61	7
unreadable	1.00	0.23	0.38	13
micro avg	0.53	0.50	0.51	20
macro avg	0.72	0.62	0.49	20
weighted avg	0.80	0.50	0.46	20

```
<Axes: title={'center': 'Confusion Matrix (Normalized)'}, xlabel='Predicted Classes', ylabel='Actual Classes'>
```



A few remarks are in order:

- We observe that the LLM Judge correctly identifies all the readable examples.
- However, the LLM Judge has a high Type 1 error (false positive rate). A few actionable insights to improve this can be using different LLM Judge which has more reasoning capabilities (`code-llama` , `codegemma` or `gpt-4`) or provide more information in the eval template which might help the existing LLM Judge (in our case, `codestral`).

Part 2: LLM as a Judge in a Real-World Scenario

I refer the reader to Part 2 section of the `python` notebook.

In this section, I consider a real-world scenario where we are presented with a set of `prompts` and we generate `code` based on a LLM which is part of our application. As human evaluation of the generated `code` is expensive or beyond the means of our organization, I wish to use to **Phoenix LLM Evals** library to classify the generated `code` as readable or not.

I assume that I ONLY have access to `prompts` from OpenAI Human Eval Dataset .

The LLM model that I will be using to generate code is `codet5-large-ntp-py` by Salesforce. The motivation to use this model comes from recent success of the paper CodeRL.

Step 1: Load `codet5-large-ntp-py` model.

```python
from huggingface_hub import login

# Replace 'your_hf_token' with your actual token
login(token=HF_TOKEN)
```

```python
from transformers import RobertaTokenizer, T5ForConditionalGeneration, AutoTokenizer, AutoModelForCausalLM
tokenizer = AutoTokenizer.from_pretrained("Salesforce/codet5-large-ntp-py")
gen_model = T5ForConditionalGeneration.from_pretrained("Salesforce/codet5-large-ntp-py")
```

```
tokenizer_config.json: 100% 1.30k/1.30k [00:00<00:00, 68.2kB/s]
vocab.json: 100% 511k/511k [00:00<00:00, 8.50MB/s]
merges.txt: 100% 294k/294k [00:00<00:00, 14.9MB/s]
```

Step 2: Given `prompts` from our dataset OpenAI Human Eval Dataset we generate `code` solution for each `prompt`.

```python
for index in range(N_EVAL_SAMPLE_SIZE):
  # Generate input prompt
  prompt = df["input"].iloc[index]
  entry_point = df["entry_point"].iloc[index]
  input_text = f"""
  You are a Python expert. Here is a task you need to complete:
  {prompt}.

  Please implement the function named '{entry_point}' and only return the code inside the function named {entry_point}. Remove any other text or explanation.

  Make sure to use correct Python syntax.
  """


  inputs = tokenizer(input_text, return_tensors="pt", padding=True, truncation=True)

  # Generate code
  output = gen_model.generate(**inputs, max_length=256)
  generated_code = tokenizer.decode(output[0], skip_special_tokens=True)
  codet5_df.loc[index, "output"] = generated_code
```

Input prompt for each sample

Generated code for each sample

Output a dataframe containing prompt and generated code

Step 3: Use `codestral-mamba` as LLM Judge to evaluate the code generated.

```python
rails = list(CODE_READABILITY_PROMPT_RAILS_MAP.values())
readability_classifications_codet5 = llm_classify(
    dataframe=codet5_df,
    template=CODE_READABILITY_PROMPT_TEMPLATE,
    model=model,
    rails=rails,
    concurrency=1,
    verbose=True,
    provide_explanation=True,
    max_retries=20,
    run_sync=False,
)["label"].tolist()
```

```
Using prompt:

[PromptPartTemplate(content_type=<PromptPartContentType.TEXT: 'text'>, template='\nYou are a
<ipython-input-21-4f84ba3ff771>:2: DeprecationWarning: `dataframe` argument is deprecated; us
  readability_classifications_codet5 = llm_classify(
llm_classify 20/20 (100.0%) | ⏳ 01:17<00:00 | 4.43s/it
- Snapped 'readable"' to rail: readable
- Snapped 'readable"' to rail: readable
- Snapped 'readable"\n\n**********\nEXPLANATION: The code is readable because it has a clea
- Snapped 'readable"\n\nExplanation: The code is readable because it is simple, straightforwa
- Snapped 'for this code would be "readable".\n\nLABEL: "readable"' to rail: readable
- Snapped 'for this code is "readable".\n\nEXPLANATION: The code is readable because it is si
- Snapped 'for this code is "readable".\n\nEXPLANATION: The code is well-documented, has clea
- Snapped 'readable"' to rail: readable
```

Step 4: Let us now look at the readability classifications

```
print(readability_classifications_codet5)
```

```
readable', 'readable', 'readable', 'readable', 'readable', 'NOT_PARSABLE', 'NOT_PARSABLE', 'readable', 'readable', 'readable', '
```

- One thing to note is that sometimes due to pre-defined `max_length` of the response the functions/methods are not complete in the generated `code`, therefore the evaluator / judge has returned labels like `NOT_PARSABLE`. This can be fixed by choosing a larger value for `max_length`. Due to limited compute power I chose a smaller value for `max_length`.

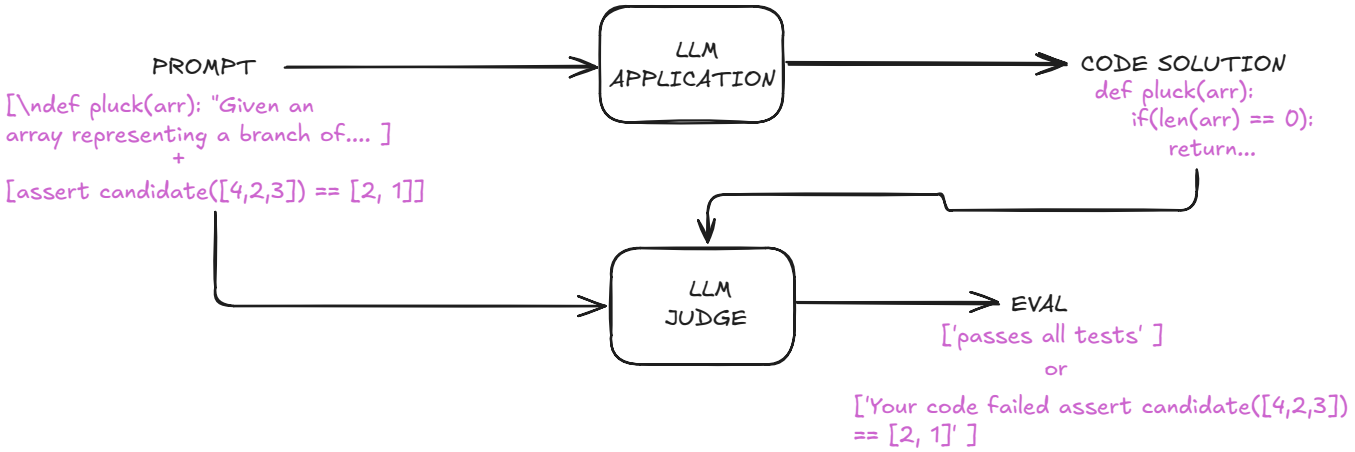
Improvements

In the following section I list a series of ideas or methodologies for improvement and other applications where LLM as a Judge can be useful.

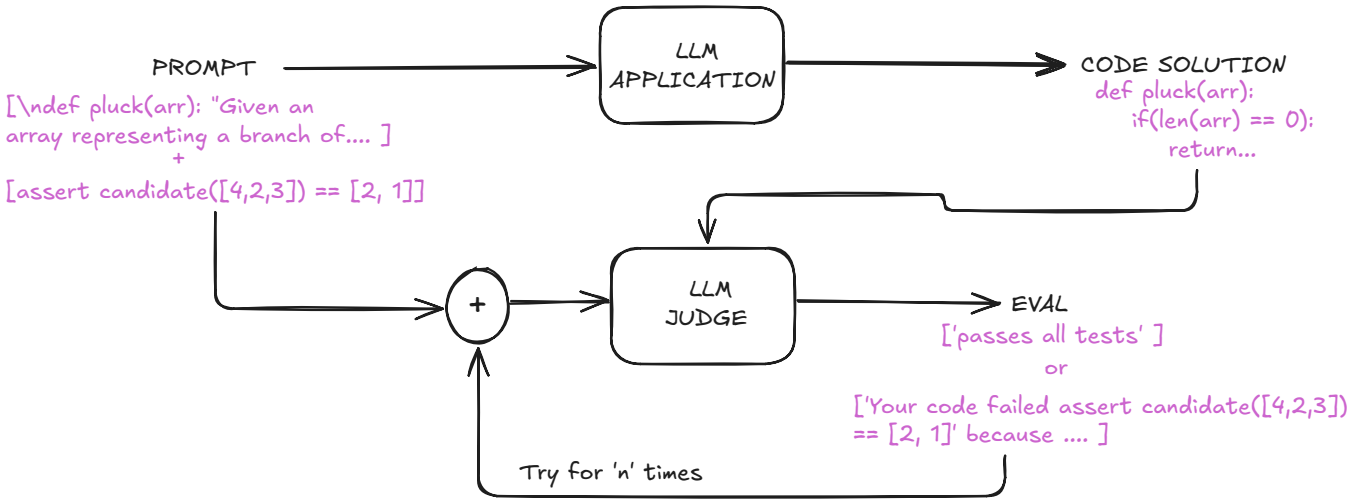
- **Informative explanation can be provided:**
  For instance, readability of a `code` does not necessarily imply functional correctness and guarantee that it will `PASS` all unit tests. Most of the benchmark code datasets (`MBPP`, `APPS`, `BigCodeBench`, `HumanEval`) consists of unit test list (like `assert` or other tests). It can be beneficial for an organization to evaluate their LLM application code to check if it passes all unit tests or if it does not then provide compiler feedback and potential solutions to fix the `code`.

A simple approach can be to design a custom template which can provide insights into which unit tests failed. An example of custom template is illustrated below:

```
You are a strong programming expert who cares a lot about passing all unit test cases. Can
you review the following code that was written by another engineer?
Focus on the code passing all provided unit tests. Respond with "pass" if you think the code
passes all unit tests,
or "fail" if the code is fails any unit test.

If any of the unit test FAILS provide an explanation on why the unit test failed and provide
a potential solution to fix the implementation.

Task Assignment:

{query}

Implementation to Evaluate:

{code}

Unit Tests to Evaluate:

{test}
```



A more nuanced approach can be running the `code` and using the compiler feedback to generate actionable insights into fixing code.



- **Ensemble Evaluation:** Using one LLM as a Judge might not provide enough reasoning capability. It can be beneficial to use an ensemble of them and use a variety of pooling strategies like (Voting, Weighted Average etc.) to generate a final result. This can readily be done using `run_evals` method from **Phoenix LLM Evals** library.
- **Data efficient accuracy estimation for high stakes applications:**
  In mission-critical applications such as defense (object recognition tasks) and medicine (disease classification tasks), large datasets are often available, but labeling them is costly and time-intensive, sometimes taking weeks. When a model is deployed, it is crucial to assess its performance without waiting for extended labeling efforts to determine if its accuracy has degraded. In such cases, using an LLM as a judge to generate ground truth labels can be highly beneficial. These LLM-generated labels can serve as a proxy for ground truth, enabling real-time evaluation of the model's accuracy.