Framework for Evaluating LLM Outputs

When comparing the performance of two large language models (LLMs) for a Boolean question-answering task, I would use a systematic approach focusing on correctness, consistency, and reliability. Here's how I would go about it:

---

1. Evaluation Process

a. Data Preparation

First, I would ensure both models are tested on the same inputs to maintain consistency. For this example, the paragraph and Boolean questions are:

Paragraph:
"The company launched its new software with features like data encryption and automatic backups. Pricing information is available, but there's no mention of customer support options."

Boolean Questions:

1. "Does it mention pricing?"
2. "Is customer support discussed?"
3. "Does it talk about data encryption?"

This dataset ensures that any evaluation is fair and unbiased.

---

b. Evaluation Criteria

To assess the models, I would focus on three primary metrics:

1. Accuracy
This is the core metric to determine whether each model's Yes/No answers align with the ground truth. For instance:

- **Ground Truth**: ["Yes", "No", "Yes"]
- **Model A Output**: ["Yes", "No", "Yes"] → Correct answers: 3/3 = 100% accuracy
- **Model B Output**: ["Yes", "No", "No"] → Correct answers: 2/3 = 66.7% accuracy

2. Explanation Quality
If the models provide reasoning for their answers, I would evaluate how well these explanations align with the text:

- Is the reasoning relevant to the paragraph?
- Is it clear and unambiguous?
  For example:
  - **Model A**: "Pricing is mentioned explicitly; customer support is not discussed."

        - Clear and directly tied to the paragraph.
    - **Model B**: "Pricing is in the text, and support is not mentioned. Data encryption isn't highlighted as a feature."
        - Incorrectly claims that data encryption isn't a feature.

3. Ambiguity Handling
When topics are implied or unclear, I'd assess whether the models correctly flagged ambiguity or made logical assumptions. For example, if the paragraph only hinted at customer support, the model should either explicitly state uncertainty or not make false claims.

4. Reliability
I would test the same input multiple times to ensure the models produce consistent outputs. For example:

- If Model A always outputs "Yes" for "Does it mention pricing?" across all runs, it's more reliable than Model B, which might fluctuate.

---

c. Quantitative and Qualitative Comparison

I would use a mix of quantitative and qualitative analyses to compare the models:

Quantitative Analysis
Calculate accuracy and confusion matrices for each model. For instance:

Model A Confusion Matrix

```
[[1 0]
  [0 2]]
```

-
    - Indicates perfect performance for both "Yes" and "No" categories.

Model B Confusion Matrix

```
[[1 0]
   [1 1]]
```

-
    - Shows a misclassification for one "Yes" case.

Qualitative Analysis
I'd compare the clarity of reasoning and the handling of ambiguous topics by manually reviewing the explanations. A detailed classification report for precision, recall, and F1-score also adds depth to the evaluation.

---

d. Human Review

When models disagree or fail to align with the ground truth, human evaluators play a key role. I would propose simple guidelines for reviewers:

- Mark outputs as Correct (1), Incorrect (0), or Ambiguous (flag for further review).
- Base evaluations on explicit mentions in the text while considering logical assumptions when necessary.

---

e. Recommendation

Finally, I would use a weighted scoring formula to combine the evaluation metrics (e.g., Accuracy 60%, Explanation Quality 30%, Ambiguity Handling 10%). Based on the example:

- **Model A**:
  - Accuracy: 100%, Explanation Quality: 90%, Ambiguity Handling: 80%
  - Final Score = (0.6 * 100) + (0.3 * 90) + (0.1 * 80) = **94**
- **Model B**:
  - Accuracy: 67%, Explanation Quality: 60%, Ambiguity Handling: 70%
  - Final Score = (0.6 * 67) + (0.3 * 60) + (0.1 * 70) = **64.2**

Based on this, I would recommend **Model A**, as it demonstrates higher accuracy, better reasoning, and superior handling of ambiguous topics.

Adding Automation with Python

To streamline this process, I would use Python to automate accuracy scoring, classification reports, and confusion matrix generation. Here's a script I'd use:

```python
from sklearn.metrics import accuracy_score, classification_report,
confusion_matrix
import seaborn as sns
import matplotlib.pyplot as plt

def evaluate_models(model_a_outputs, model_b_outputs, ground_truth):
    acc_model_a = accuracy_score(ground_truth, model_a_outputs)
    acc_model_b = accuracy_score(ground_truth, model_b_outputs)

    report_model_a = classification_report(ground_truth,
model_a_outputs, target_names=["No", "Yes"])
    report_model_b = classification_report(ground_truth,
model_b_outputs, target_names=["No", "Yes"])
```

```
    cm_model_a = confusion_matrix(ground_truth, model_a_outputs)
    cm_model_b = confusion_matrix(ground_truth, model_b_outputs)

    print("Model A Evaluation:")
    print(f"Accuracy: {acc_model_a:.2f}")
    print("Classification Report:")
    print(report_model_a)

    print("\nModel B Evaluation:")
    print(f"Accuracy: {acc_model_b:.2f}")
    print("Classification Report:")
    print(report_model_b)

    fig, axes = plt.subplots(1, 2, figsize=(12, 5))
    sns.heatmap(cm_model_a, annot=True, fmt="d", cmap="Blues",
xticklabels=["No", "Yes"], yticklabels=["No", "Yes"], ax=axes[0])
    axes[0].set_title("Model A Confusion Matrix")
    sns.heatmap(cm_model_b, annot=True, fmt="d", cmap="Blues",
xticklabels=["No", "Yes"], yticklabels=["No", "Yes"], ax=axes[1])
    axes[1].set_title("Model B Confusion Matrix")
    plt.tight_layout()
    plt.show()

ground_truth = ["Yes", "No", "Yes"]
model_a_outputs = ["Yes", "No", "Yes"]
model_b_outputs = ["Yes", "No", "No"]

evaluate_models(model_a_outputs, model_b_outputs, ground_truth)
```

With this, I can evaluate the models quantitatively and visualize the results to support my recommendations.

Reinforcement Learning:
Reinforcement Learning (RL) can be an effective approach to train LLMs for Boolean question-answering tasks, particularly in optimizing their ability to handle ambiguity, improve reasoning, and provide correct responses. Using **Reinforcement Learning with Human Feedback (RLHF)** or simulated feedback, we can iteratively fine-tune models based on reward signals that evaluate their performance on specific criteria, such as accuracy, explanation quality, and ambiguity resolution.

For instance, we can design a reward system where:

- Correct Yes/No answers aligned with the ground truth yield high rewards.
- Clear, relevant, and logical explanations increase the reward further.

- Ambiguous or incorrect answers, or explanations that lack relevance, result in penalties.

By framing the Boolean QA task as a Markov Decision Process (MDP), where the model's output (Yes/No and explanation) represents the actions, the paragraph and question context act as the state, and the reward is based on predefined evaluation metrics, we can systematically train the model. This method helps improve the model's decision-making, making it better at handling edge cases, unclear phrasing, and reasoning tasks.

```python
import numpy as np
import torch
from transformers import AutoModelForCausalLM, AutoTokenizer

# Load a pre-trained model and tokenizer
model_name = "gpt2"
model = AutoModelForCausalLM.from_pretrained(model_name)
tokenizer = AutoTokenizer.from_pretrained(model_name)

# Sample paragraph and questions
paragraph = "The company launched its new software with features like data encryption and automatic backups. Pricing information is available, but there's no mention of customer support options."
questions = [
    "Does it mention pricing?",
    "Is customer support discussed?",
    "Does it talk about data encryption?",
]
ground_truth = ["Yes", "No", "Yes"]

# Reward function (simplified for demonstration)
def reward_function(answer, explanation, ground_truth, question):
    reward = 0
    # Reward for correct answer
    if answer == ground_truth:
        reward += 10
    # Explanation quality (example logic, can be improved)
    if explanation and ground_truth.lower() in explanation.lower():
        reward += 5
    # Penalty for incorrect answers or vague explanations
    if answer != ground_truth or not explanation:
        reward -= 5
    return reward

# Training Loop (simplified)
```

```
optimizer = torch.optim.AdamW(model.parameters(), lr=1e-5)
num_epochs = 5

for epoch in range(num_epochs):
    total_reward = 0
    for i, question in enumerate(questions):
        # Encode input
        input_text = f"Paragraph: {paragraph}\nQuestion: {question}\nAnswer
and Explanation:"
        inputs = tokenizer(input_text, return_tensors="pt")

        # Generate response
        outputs = model.generate(inputs["input_ids"], max_length=100)
        response = tokenizer.decode(outputs[0], skip_special_tokens=True)
        print(respnse)
        # Parse response
        try:
            answer, explanation = response.split("Explanation: ")
        except ValueError:
            answer, explanation = response, ""

        # Calculate reward
        reward = reward_function(answer.strip(), explanation.strip(),
ground_truth[i], question)
        total_reward += reward

        # Update model using the reward
        loss = -torch.tensor(reward, dtype=torch.float, requires_grad=True)
# Ensure dtype is float
        loss.backward()
        optimizer.step()
        optimizer.zero_grad()

    print(f"Epoch {epoch + 1}, Total Reward: {total_reward}")

print("Training complete!")
```