Author - Aniket Phutane

RAG Evaluation: From Theory to Implementation

Introduction to RAG Evaluation

Retrieval-Augmented Generation (RAG) systems combine information retrieval with generative AI to produce accurate, contextually relevant responses. Evaluating RAG systems is complex because it involves assessing multiple components and their interactions. This guide provides both theoretical foundations and practical implementations for RAG evaluation metrics.

Theoretical Foundations

1. Retrieval Quality Assessment

1.1 Theoretical Basis

Retrieval quality measures how effectively the system identifies and retrieves relevant context from the knowledge base. Key theoretical considerations include:

- **Relevance Theory**: Based on information retrieval principles where relevance is measured through topical similarity, contextual appropriateness, and information utility.
- **Ranking Quality**: Evaluates the system's ability to prioritize the most relevant documents, considering both precision and ranking position.
- **Coverage Analysis**: Assesses whether retrieved documents contain sufficient information to answer the query comprehensively.

1.2 Key Metrics and Their Significance

- **Hit Rate @ K**
    - Theoretical Basis: Measures the presence of relevant documents in top-K retrievals
    - Significance: Indicates retriever's ability to surface relevant information within a practical limit
    - Limitations: Binary metric that doesn't consider ranking quality

```
def compute_hit_rate_at_k(retrieved_docs, relevant_docs, k=3):
    """
    Compute Hit Rate@K for retrieval evaluation
    Args:
        retrieved_docs: List of retrieved document IDs
        relevant_docs: List of relevant document IDs
        k: Number of top documents to consider

    Returns:
        hit_rate: Binary score indicating if any relevant doc is in top-k
    """
    retrieved_set = set(retrieved_docs[:k])
    relevant_set = set(relevant_docs)

    return float(len(retrieved_set.intersection(relevant_set)) > 0)
```

- **Mean Reciprocal Rank (MRR)**
  - Theoretical Basis: Evaluates ranking quality by focusing on the position of the first relevant document
  - Significance: Important for applications where finding one correct document quickly is crucial
  - Limitations: Doesn't consider multiple relevant documents

```python
def compute_mrr(retrieved_docs, relevant_docs):
    """
    Compute Mean Reciprocal Rank for retrieval evaluation

    Args:
        retrieved_docs: List of retrieved document IDs
        relevant_docs: List of relevant document IDs

    Returns:
        mrr: Mean Reciprocal Rank score
    """
    relevant_set = set(relevant_docs)

    for rank, doc_id in enumerate(retrieved_docs, 1):
        if doc_id in relevant_set:
            return 1.0 / rank
    return 0.0
```

- **Normalized Discounted Cumulative Gain (NDCG)**
  - Theoretical Basis: Combines relevance grading with position-based discounting
  - Significance: Provides nuanced evaluation of ranking quality
  - Limitations: Requires relevance grades for documents

```python
import numpy as np

def compute_ndcg(retrieved_docs, relevance_scores, k=None):
    """
    Compute NDCG@K for retrieval evaluation
    Args:
        retrieved_docs: List of retrieved document IDs
        relevance_scores: Dict mapping doc_id to relevance score
        k: Number of documents to consider (optional)

    Returns:
        ndcg: NDCG@K score
    """
    if k is None:
        k = len(retrieved_docs)

    # Calculate DCG
    dcg = 0
    for i, doc_id in enumerate(retrieved_docs[:k]):
        rel = relevance_scores.get(doc_id, 0)
        dcg += (2 ** rel - 1) / np.log2(i + 2)

    # Calculate ideal DCG
    ideal_ordering = sorted(relevance_scores.values(), reverse=True)
    idcg = 0
    for i, rel in enumerate(ideal_ordering[:k]):
        idcg += (2 ** rel - 1) / np.log2(i + 2)

    return dcg / idcg if idcg > 0 else 0.0
```

2. Answer Quality Evaluation

2.1 Theoretical Basis

Answer quality evaluation focuses on the generated response's correctness, relevance, and usefulness. Key theoretical aspects include:

- **Information Accuracy**: Measures factual correctness and alignment with source documents
- **Response Completeness**: Evaluates whether all relevant aspects of the query are addressed
- **Answer Coherence**: Assesses logical flow and structural integrity of the response

2.2 Key Metrics and Their Significance

- **Context Relevance Score**
  - Theoretical Basis: Semantic similarity between query and retrieved contexts
  - Significance: Indicates retriever's understanding of query intent
  - Limitations: May not capture nuanced relevance aspects

```
from sentence_transformers import SentenceTransformer
```

```
import torch

class ContextRelevanceScorer:
    def __init__(self):
        self.model = SentenceTransformer('all-MiniLM-L6-v2')

    def compute_relevance(self, query, retrieved_contexts):
        """
        Compute semantic similarity between query and retrieved contexts

        Args:
            query: Input query string
            retrieved_contexts: List of retrieved context strings

        Returns:
            relevance_scores: List of similarity scores
        """
        # Encode query and contexts
        query_embedding = self.model.encode(query, convert_to_tensor=True)
        context_embeddings = self.model.encode(retrieved_contexts,
convert_to_tensor=True)

        # Compute cosine similarities
        similarities = torch.nn.functional.cosine_similarity(
            query_embedding.unsqueeze(0),
            context_embeddings
        )

        return similarities.cpu().numpy()
```

- **Answer Faithfulness**
  - Theoretical Basis: Measures alignment between generated answer and source context
  - Significance: Critical for preventing hallucinations
  - Limitations: Complex to measure due to linguistic variations

```
from transformers import AutoModelForSequenceClassification, AutoTokenizer

class FaithfulnessChecker:
    def __init__(self):
        self.model =
AutoModelForSequenceClassification.from_pretrained("microsoft/deberta-
base-mnli")
        self.tokenizer = AutoTokenizer.from_pretrained("microsoft/deberta-
base-mnli")

    def check_faithfulness(self, context, generated_answer):
        """
        Check if generated answer is supported by the context
        Args:
            context: Retrieved context string
            generated_answer: Generated answer string

        Returns:
            faithfulness_score: Entailment probability score
        """
        inputs = self.tokenizer(
            premise=context,
            hypothesis=generated_answer,
            return_tensors="pt",
            truncation=True,
            max_length=512
        )

        with torch.no_grad():
            outputs = self.model(**inputs)
            probs = torch.nn.functional.softmax(outputs.logits, dim=-1)

        # Return entailment probability (class index 2)
        return probs[0][2].item()
```

- **Answer Relevance**
  - Theoretical Basis: Semantic and structural similarity to reference answers
  - Significance: Indicates response quality when references are available
  - Limitations: Reference answers may not be exhaustive

```
def compute_answer_relevance(generated_answer, reference_answer,
metric="rouge"):
    """
    Compute relevance between generated and reference answers

    Args:
        generated_answer: Generated answer string
        reference_answer: Reference answer string
        metric: Metric to use (rouge, bleu, or bert_score)

    Returns:
        relevance_score: Similarity score between answers
    """
    if metric == "rouge":
        from rouge_score import rouge_scorer
        scorer = rouge_scorer.RougeScorer(['rouge1', 'rouge2', 'rougeL'])
        scores = scorer.score(reference_answer, generated_answer)
        return {
            'rouge1': scores['rouge1'].fmeasure,
            'rouge2': scores['rouge2'].fmeasure,
            'rougeL': scores['rougeL'].fmeasure
        }
    elif metric == "bleu":
        from nltk.translate.bleu_score import sentence_bleu
        from nltk.tokenize import word_tokenize

        reference_tokens = [word_tokenize(reference_answer)]
        generated_tokens = word_tokenize(generated_answer)

        return sentence_bleu(reference_tokens, generated_tokens)

    elif metric == "bert_score":
        from bert_score import score

        P, R, F1 = score([generated_answer], [reference_answer],
lang="en")
        return {
            'precision': P.item(),
            'recall': R.item(),
            'f1': F1.item()
        }
```

3. Context Utilization Analysis

3.1 Theoretical Basis

Context utilization examines how effectively the generator uses retrieved information. Key theoretical aspects include:

- **Information Integration**: How well the system combines multiple context pieces
- **Source Attribution**: Accuracy in using and attributing information from sources
- **Hallucination Prevention**: Ability to stay grounded in provided context

3.2 Key Metrics and Their Significance

- **Context Usage Rate**
  - Theoretical Basis: Measures proportion of context information used in generation
  - Significance: Indicates effective use of retrieved information
  - Limitations: Quantity doesn't guarantee quality of usage

```
def compute_context_usage(context, generated_answer):
    """
    Compute how much of the context is used in the generated answer

    Args:
        context: Retrieved context string
        generated_answer: Generated answer string

    Returns:
        usage_metrics: Dict containing various usage statistics
    """
    # Tokenize context and answer
    context_tokens = set(context.lower().split())
    answer_tokens = set(generated_answer.lower().split())

    # Compute overlap statistics
    overlap_tokens = context_tokens.intersection(answer_tokens)

    metrics = {
        'token_overlap_rate': len(overlap_tokens) / len(context_tokens),
        'novel_token_rate': len(answer_tokens - context_tokens) /
len(answer_tokens),
        'total_tokens_used': len(overlap_tokens)
    }

    return metrics
```

- **Hallucination Detection**
  - Theoretical Basis: Identifies generated content not supported by context
  - Significance: Critical for trustworthiness and reliability
  - Limitations: Complex to detect subtle hallucinations

```
class HallucinationDetector:
    def __init__(self):
        self.tokenizer = AutoTokenizer.from_pretrained("microsoft/deberta-
large-mnli")
        self.model =
AutoModelForSequenceClassification.from_pretrained("microsoft/deberta-
large-mnli")

    def detect_hallucinations(self, context, generated_answer):
        """
        Detect potential hallucinations in generated answer

        Args:
            context: Retrieved context string
            generated_answer: Generated answer string

        Returns:
            hallucination_score: Probability of hallucination
        """
        # Split answer into sentences
        from nltk.tokenize import sent_tokenize
        answer_sentences = sent_tokenize(generated_answer)

        hallucination_scores = []
        for sentence in answer_sentences:
            inputs = self.tokenizer(
                premise=context,
                hypothesis=sentence,
                return_tensors="pt",
                truncation=True
            )

            with torch.no_grad():
                outputs = self.model(**inputs)
                probs = torch.nn.functional.softmax(outputs.logits,
dim=-1)
                # Contradiction probability (class index 0)
                hallucination_scores.append(probs[0][0].item())

        return {
            'max_hallucination_score': max(hallucination_scores),
            'avg_hallucination_score': sum(hallucination_scores) /
len(hallucination_scores),
            'sentence_scores': hallucination_scores
        }
```

4. End-to-End System Evaluation

4.1 Comprehensive RAG Evaluator Implementation

```
class RAGEvaluator:
    def __init__(self):
        self.context_scorer = ContextRelevanceScorer()
        self.faithfulness_checker = FaithfulnessChecker()
        self.hallucination_detector = HallucinationDetector()

    def evaluate(self, query, retrieved_contexts, generated_answer,
reference_answer=None):
        """
        Comprehensive RAG evaluation

        Args:
            query: Input query string
            retrieved_contexts: List of retrieved context strings
            generated_answer: Generated answer string
            reference_answer: Optional reference answer string

        Returns:
            evaluation_results: Dict containing all evaluation metrics
        """
        results = {}

        # Evaluate retrieval quality
        results['context_relevance'] =
self.context_scorer.compute_relevance(
            query, retrieved_contexts
        ).tolist()

        # Evaluate answer quality
        results['faithfulness'] =
self.faithfulness_checker.check_faithfulness(
            " ".join(retrieved_contexts),
            generated_answer
        )

        if reference_answer:
            results['answer_relevance'] = compute_answer_relevance(
                generated_answer,
                reference_answer,
                metric="rouge"
            )

        # Evaluate context utilization
        results['context_usage'] = compute_context_usage(
            " ".join(retrieved_contexts),
            generated_answer
        )

        results['hallucination'] =
self.hallucination_detector.detect_hallucinations(
            " ".join(retrieved_contexts),
            generated_answer
        )

        return results
```

4.2 Aggregated Metrics Implementation

```
def compute_aggregated_metrics(evaluation_results_list):
    """
    Compute aggregated metrics across multiple evaluations

    Args:
        evaluation_results_list: List of evaluation results from
RAGEvaluator

    Returns:
        aggregated_metrics: Dict containing aggregated statistics
    """
    aggregated = {
        'avg_context_relevance': [],
        'avg_faithfulness': [],
        'avg_rouge_scores': {'rouge1': [], 'rouge2': [], 'rougeL': []},
        'avg_context_usage': [],
        'avg_hallucination': []
    }

    for results in evaluation_results_list:

aggregated['avg_context_relevance'].append(np.mean(results['context_relevance
        aggregated['avg_faithfulness'].append(results['faithfulness'])

        if 'answer_relevance' in results:
            for rouge_type in aggregated['avg_rouge_scores']:
                aggregated['avg_rouge_scores'][rouge_type].append(
                    results['answer_relevance'][rouge_type]
                )

        aggregated['avg_context_usage'].append(
            results['context_usage']['token_overlap_rate']
        )
        aggregated['avg_hallucination'].append(
            results['hallucination']['avg_hallucination_score']
        )

    # Compute final averages
    final_metrics = {
        'context_relevance': np.mean(aggregated['avg_context_relevance']),
        'faithfulness': np.mean(aggregated['avg_faithfulness']),
        'rouge_scores': {
            k: np.mean(v) for k, v in
aggregated['avg_rouge_scores'].items()
        },
        'context_usage': np.mean(aggregated['avg_context_usage']),
        'hallucination_rate': np.mean(aggregated['avg_hallucination'])
    }

    return final_metrics
```

5. Usage Example

```
# Initialize evaluator
evaluator = RAGEvaluator()

# Example evaluation
query = "What is the capital of France?"
contexts = ["Paris is the capital and largest city of France."]
generated_answer = "Paris is the capital of France."
reference_answer = "The capital of France is Paris."

# Run evaluation
results = evaluator.evaluate(
    query=query,
    retrieved_contexts=contexts,
    generated_answer=generated_answer,
    reference_answer=reference_answer
)

# Print results
print(json.dumps(results, indent=2))
```

```
In [28]:
```

```
In [ ]:
```