Agora: A Comprehensive Framework for Evaluating and Generating Synthetic Data with Language Models

OpenHands AI Assistant
Technical Report
December 2023

Abstract—**This technical report presents an in-depth analysis of Agora, an innovative open-source framework designed for generating and evaluating synthetic data using Large Language Models (LLMs). We explore the architecture, key components, and methodological approaches implemented in the framework. Agora provides a standardized environment for assessing LLMs' capabilities in data generation across multiple domains including mathematics, general instruction-following, and code generation. The framework introduces AgoraBench, a comprehensive benchmark that enables systematic comparison of different LLMs as data generators. We discuss the framework's modular design, evaluation metrics, and practical applications in enhancing LLM training and evaluation. Our analysis reveals that Agora represents a significant advancement in synthetic data generation and evaluation, offering valuable insights into the capabilities and limitations of modern language models.**

I. Introduction

The increasing importance of high-quality training data in machine learning has led to growing interest in synthetic data generation using Large Language Models (LLMs). The Agora framework, inspired by the ancient Athenian marketplace where knowledge was freely exchanged, provides a systematic approach to generating and evaluating synthetic data. This framework addresses a critical gap in the field by offering standardized methods for comparing different LLMs' capabilities as data generators.

A. Background and Motivation

The development of large language models has revolutionized natural language processing, but their effectiveness is heavily dependent on the quality and quantity of training data. Traditional data collection methods often face challenges such as:

- Limited availability of domain-specific data
- High costs associated with manual data annotation
- Privacy concerns and data protection regulations
- Bias and quality control issues in collected datasets

Synthetic data generation using LLMs offers a promising solution to these challenges, but it introduces new questions about the quality and reliability of generated data. The Agora framework aims to address these concerns by providing:

- Standardized evaluation metrics for synthetic data quality
- Reproducible benchmarking methodology
- Domain-specific validation techniques
- Comprehensive analysis tools for generated datasets

B. Related Work

Previous research in synthetic data generation has primarily focused on specific domains or applications. Notable approaches include:

- Template-based generation methods
- Rule-based synthetic data creation
- GAN-based data synthesis
- Few-shot learning approaches

Agora builds upon these foundations while introducing novel approaches to evaluation and quality control. The framework's design incorporates lessons learned from previous work while addressing their limitations.

II. Framework Architecture

Agora is built with a modular architecture that facilitates customization and extension. The framework's design emphasizes flexibility, scalability, and ease of use, making it suitable for both research and production environments.

A. Core Components

The framework consists of several key components, each designed to handle specific aspects of the data generation and evaluation process:

1) Prompt Loader: The Prompt Loader component is responsible for:

- Template management and customization
- Context window optimization
- Few-shot example selection
- Dynamic prompt construction

It implements sophisticated algorithms for selecting and arranging in-context examples, ensuring optimal prompt construction for different tasks and domains. The component supports various placeholder formats:

```
placeholder_formats = {
    "demonstration_input_placeholder": "<
        input@>",
    "demonstration_output_placeholder": "<
        output@>",
    "test_input_placeholder": "<input>",
    "test_output_placeholder": "<o>",
    "test_input_trigger": "INPUT:",
    "test_output_trigger": "OUTPUT:",
    "stop_phrase": "[END]"
}
```

2) Parser: The Parser component handles:

- Output format standardization
- Structured data extraction
- Error handling and recovery
- Format validation and normalization

The parser is designed to be robust against variations in model outputs while maintaining strict format compliance. Here's an example of a custom parser implementation:

```
class InstanceGenerationParser(Parser):
    def parse(self, prompt,
        teacher_model_output,
            placeholder_formats):
        instruction = (
            teacher_model_output.split(
                placeholder_formats["
                    test_input_trigger"]
            )[-1]
            .split(
                placeholder_formats["
                    test_output_trigger"]
            )[0]
            .strip()
        )
        response = (
            teacher_model_output.split(
                placeholder_formats["
                    test_output_trigger"]
            )[-1]
            .split(
                placeholder_formats["
                    stop_phrase"]
            )[0]
            .strip()
        )
        return {
            "instruction": instruction,
            "response": response
        }
```

3) Validator: The Validator ensures data quality through:

- Semantic consistency checking
- Domain-specific rule validation
- Format compliance verification
- Quality metrics computation

Example validator implementation:

```
class CustomValidator(Validator):
    def validate(self, instruction, response):
        # Check minimum length requirements
        if len(instruction) < 10 or len(
            response) < 10:
            return False

        # Verify format compliance
        if not self._check_format(instruction,
             response):
            return False

        # Domain-specific validation
        if not self._validate_domain_rules(
            instruction, response):
            return False

        return True
```

4) LLM Interface: The LLM Interface provides:

- Unified API for multiple model providers
- Batch processing capabilities
- Error handling and retry logic
- Resource optimization

Example usage of the LLM interface:

```
llm = OpenAILLM(
    model_name="gpt-4",
    api_key="YOUR_API_KEY"
)

sampling_params = {
    "max_tokens": 4096,
    "temperature": 1.0,
    "top_p": 0.9,
    "stop": "[END]"
}

result = llm.generate(
    prompt=prompt,
    **sampling_params
)
```

B. System Integration

The components are integrated through a well-defined pipeline that ensures:

- Efficient data flow
- Error isolation and handling
- Scalability and parallelization
- Monitoring and logging

Example pipeline configuration:

```
agora = Agora(
    llm=llm,
    placeholder_formats=placeholder_formats,
    prompt_loader=prompt_loader,
    parser=parser,
    validator=validator,
    sampling_params=sampling_params
)

result = agora.run(
    num_instances=10000,
    num_threads=16,
    output_file="./results/final_result.json"
)

\section{AgoraBench}
AgoraBench serves as a standardized evaluation
     framework for assessing LLMs' data
    generation capabilities. It provides
    comprehensive metrics and evaluation
    methodologies across multiple domains.

\subsection{Evaluation Domains}
The benchmark covers three primary domains:

\subsubsection{Mathematics}
Mathematics evaluation includes:
\begin{itemize}
    \item Problem complexity assessment
    \item Solution correctness verification
    \item Step-by-step explanation quality
    \item Mathematical notation accuracy
\end{itemize}

Specific datasets used include:
\begin{itemize}
    \item GSM8K for grade school mathematics
    \item MATH for advanced mathematical
        reasoning
\end{itemize}

Example GSM8K problem generation:
\begin{lstlisting}
# Prompt template for math problem generation
template = """Generate a grade school math
problem that tests {concept} understanding.
The problem should be clear and solvable.

Format:
Problem: [Your problem here]
Solution: [Step-by-step solution]

Examples:
{examples}

Now generate a new problem:"""

# Example usage
problem = generate_math_problem(
    template,
    concept="percentage calculation",
    examples=gsm8k_examples[:3]
)
```

1) General Instruction Following: This domain evaluates:

- Task comprehension accuracy
- Response relevance and completeness
- Instruction adherence
- Output format compliance

Evaluation tools include:

- AlpacaEval 2.0 for instruction following
- Arena-Hard for complex task handling

Example instruction generation:

```
# Generate complex instruction
instruction = generate_instruction(
    domain="task_planning",
    complexity_level="high",
    required_steps=5,
    context_length="medium"
)

# Validate instruction quality
quality_score = evaluate_instruction(
    instruction,
    metrics=["clarity", "complexity", "
        feasibility"]
)
```

2) Code Generation: Code generation assessment covers:

- Functional correctness
- Code quality metrics
- Documentation quality
- Error handling implementation

Benchmark suites include:

- MBPP for basic programming tasks
- HumanEval for realistic coding scenarios

Example code generation evaluation:

```
def evaluate_code_solution(
    problem_spec,
    generated_code,
    test_cases
):
    # Compile and run tests
    results = run_test_suite(
        generated_code,
        test_cases
    )

    # Evaluate code quality
    quality_metrics = analyze_code_quality(
        generated_code,
        metrics=[
            "complexity",
            "maintainability",
            "documentation"
        ]
    )

    return {
        "test_results": results,
        "quality_score": quality_metrics
    }
```

C. Generation Methods

AgoraBench evaluates three distinct approaches to data generation:

1) Instance Generation: This method focuses on:

- Creating new problem-solution pairs
- Ensuring diversity in generated instances
- Maintaining task difficulty distribution
- Validating semantic correctness

Example implementation:

```
class InstanceGenerator:
    def generate_instance(self, domain, params
        ):
        # Generate problem-solution pair
        problem = self.create_problem(domain,
            params)
        solution = self.solve_problem(problem)

        # Validate instance
        if self.validate_instance(problem,
            solution):
            return {
                "problem": problem,
                "solution": solution,
                "metadata": self.get_metadata
                    ()
            }
        return None
```

2) Response Generation: Response generation evaluation includes:

- Answer quality assessment
- Explanation clarity
- Response completeness
- Format adherence

Example response generation:

```
def generate_response(problem, params):
    # Generate detailed response
    response = llm.generate(
        prompt=format_prompt(problem),
        max_tokens=params["max_tokens"],
        temperature=params["temperature"]
    )

    # Validate response quality
    quality_score = evaluate_response(
        problem,
        response,
        criteria=[
            "completeness",
            "clarity",
            "accuracy"
        ]
    )

    return response, quality_score
```

3) Quality Enhancement: This approach examines:

- Improvement of existing instances
- Clarity enhancement
- Error correction
- Content enrichment

Example enhancement process:

```
def enhance_instance(instance):
    # Analyze current quality
    issues = analyze_quality(instance)

    # Generate improvements
    improvements = generate_improvements(
        instance,
        issues
    )

    # Apply and validate changes
    enhanced = apply_improvements(
        instance,
        improvements
    )

    return enhanced
```

III. Implementation Details

The framework implementation incorporates modern software engineering practices and provides extensive customization options.

A. Code Structure

The codebase is organized into logical components:

1) Core Library:

- Modular architecture design
- Extensible component interfaces
- Comprehensive utility functions
- Robust error handling

Example core module structure:

```
libs/data-agora/
    data_agora/
        core/
            llm.py
            parser.py
            validator.py
            utils.py
        generators/
            instance_gen.py
            response_gen.py
            quality_enhance.py
        evaluation/
            metrics.py
            benchmarks.py
```

2) Scripts and Tools:

- Data processing utilities
- Format conversion tools
- Evaluation scripts
- Benchmark runners

3) Training Infrastructure:

- Integration with llama-recipes
- Distributed training support
- Checkpoint management
- Performance optimization

Example training configuration:

```
training_config = {
    "model_name": "meta-llama/Llama-2-7b",
    "train_batch_size": 4,
    "gradient_accumulation_steps": 8,
    "learning_rate": 1e-5,
    "num_epochs": 3,
    "warmup_steps": 100,
    "evaluation_strategy": "steps",
    "save_steps": 500,
    "eval_steps": 500
}
```

B. Customization Points

The framework supports extensive customization:

1) Prompt Templates:

- Custom template definition
- Variable placeholder system
- Context window management
- Dynamic content insertion

2) Validation Rules:

- Domain-specific validators
- Custom quality metrics
- Validation pipeline configuration
- Error handling policies

3) Output Formats:

- Custom parser definitions
- Format transformation rules
- Schema validation
- Output normalization

IV. Practical Applications

Agora enables several practical applications in machine learning research and development.

A. Data Generation Pipeline

The framework provides a complete pipeline for:

1) Training Data Creation:

- Automated instance generation
- Quality control workflows
- Format standardization
- Dataset augmentation

Example pipeline usage:

```
# Configure data generation pipeline
pipeline = DataGenerationPipeline(
    generator=InstanceGenerator(),
    validator=QualityValidator(),
    formatter=OutputFormatter()
)

# Generate dataset
dataset = pipeline.generate(
    num_instances=10000,
    domain="mathematics",
    quality_threshold=0.8
)

# Export results
dataset.save("generated_dataset.jsonl")
```

2) Evaluation Dataset Generation:

- Benchmark dataset creation
- Test case generation
- Edge case identification
- Difficulty level calibration

B. Model Evaluation

Comprehensive evaluation capabilities include:

1) Performance Metrics:

- Task-specific metrics
- Quality indicators
- Efficiency measures
- Error analysis

Example evaluation workflow:

```
def evaluate_model(model, test_set):
    metrics = {
        "accuracy": [],
        "quality_scores": [],
        "generation_time": [],
        "memory_usage": []
    }

    for test_case in test_set:
        # Generate response
        start_time = time.time()
        response = model.generate(test_case)
        metrics["generation_time"].append(
            time.time() - start_time
        )

        # Evaluate response
        metrics["accuracy"].append(
            evaluate_accuracy(
                test_case,
                response
            )
        )
        metrics["quality_scores"].append(
            assess_quality(response)
        )

    return compute_aggregate_metrics(metrics)
```

2) Comparative Analysis:

- Model comparison frameworks
- Performance benchmarking
- Capability assessment
- Resource utilization analysis

V. Future Work

Several directions for future development have been identified:

A. Technical Enhancements

- Advanced validation techniques
- Improved quality metrics
- Enhanced parallelization
- Automated optimization

Example future enhancement:

```
# Proposed automated optimization system
class AutoOptimizer:
    def optimize_generation_params(
        self,
        initial_params,
        target_metrics
    ):
        # Implement Bayesian optimization
        optimizer = BayesianOptimizer(
            parameter_space=self.define_space
                (),
            objective=self.objective_function
        )

        # Run optimization loops
        best_params = optimizer.optimize(
            n_trials=100,
            target_metrics=target_metrics
        )

        return best_params
```

B. Domain Extensions

- Additional task domains
- Specialized benchmarks
- Custom evaluation metrics
- Domain-specific tools

C. Integration Capabilities

- Additional model providers
- Extended API support
- Third-party tool integration
- Cloud platform deployment

VI. Conclusion

The Agora framework represents a significant advancement in synthetic data generation and evaluation. Its comprehensive approach to assessing LLM capabilities, combined with practical tools for data generation, provides valuable resources for researchers and practitioners. The framework's modular design and extensive customization options ensure its utility across a wide range of applications and domains.

The standardized evaluation methodology introduced by AgoraBench enables meaningful comparisons between different LLMs' data generation capabilities, while the framework's practical tools facilitate the creation and validation of synthetic datasets. As the field continues to evolve, Agora's extensible architecture positions it well to incorporate new developments and address emerging challenges in synthetic data generation.