LLM Matrix

None

None

None

Table of contents

1. LLM Matrix

LLM Matrix is a tool for running, evaluating, and comparing different language models across a matrix of hyperparameters.

1.1 Overview

LLM Matrix enables you to:

- Define test suites with multiple cases in YAML
- Run test cases across different models and parameters
- Evaluate model responses using specialized metrics
- Generate comprehensive reports to analyze performance
- Compare results across models and parameters

1.2 Key Concepts

1.2.1 Test Suites

A test suite is a YAML file that defines:

- Test cases (inputs and expected outputs)
- Templates for system and user prompts
- Models to test
- Hyperparameters to vary
- Metrics for evaluation

1.2.2 Matrix Execution

LLM Matrix runs each test case against a matrix of parameters, such as:

- Different LLM models (e.g., GPT-4, Claude)
- Various temperature settings
- Different prompt templates

1.2.3 Metrics

Custom metrics evaluate model outputs by comparing them to ideal answers:

- QA metrics for question-answering tasks
- Binary classification metrics (YES/NO questions)
- List matching metrics for enumeration tasks

1.2.4 Result Analysis

After running a test suite, you can analyze:

- Model performance by score
- Statistical summaries
- Per-case breakdowns
- Comparisons across models and parameters

1.3 Installation

```
pip install llm-matrix
```

For optional features:

```
# Excel report support
pip install "llm-matrix[excel]"

# MLflow integration
pip install "llm-matrix[mlflow]"

# LinkML mapping support
pip install "llm-matrix[map]"
```

1.4 Quick Links

- Tutorial - Get started with LLM Matrix
- Configuration - Learn how to configure test suites
- Metrics - Available metrics for evaluation
- CLI - Command-line interface reference
- API Reference - Python API documentation

2. Getting Started

2.1 Getting Started with LLM Matrix

This tutorial walks you through the basic usage of LLM Matrix, from installation to running your first evaluation.

2.1.1 Installation

Install LLM Matrix using pip:

```
pip install llm-matrix
```

2.1.2 Creating Your First Test Suite

Create a file named `first-test.yaml` with the following content:

```yaml
name: my-first-test
template: simple_qa
templates:
  simple_qa:
    system: Answer the following question accurately and concisely.
    prompt: "{input}"
    metrics:
      - qa_simple
matrix:
  hyperparameters:
    model: [gpt-3.5-turbo]
    temperature: [0.0]
cases:
  - input: What is the capital of France?
    ideal: Paris
    tags: [geography]
  - input: What is 2+2?
    ideal: "4"
    tags: [math]
```

Understanding the YAML Structure

- `name` : Identifier for your test suite
- `template` : Default template to use
- `templates` : Define different prompt templates
- `system` : System instructions to the model
- `prompt` : Template for user prompt (with variables in curly braces)
- `metrics` : Evaluation metrics to apply
- `matrix.hyperparameters` : Parameters to vary in the test
- `cases` : Individual test cases
- `input` : Query to send to the model
- `ideal` : Expected response (used for evaluation)
- `tags` : Optional categorization

2.1.3 Running the Test Suite

Execute your test suite with:

```
llm-matrix run first-test.yaml
```

This will: 1. Create a database file to cache results 2. Run each test case against the specified model(s) 3. Generate evaluation reports

2.1.4 Viewing Results

After running the test suite, you'll find a `first-test-output` directory with:

- `results.csv` : Raw results for all runs
- `summary.csv` : Statistical summary of model performance
- `by_model.csv` : Performance breakdown by model
- `grouped_by_input.tsv` : Detailed breakdown by test case

2.1.5 Next Steps

- Configuring Test Suites: Learn about advanced configuration options
- Understanding Metrics: Explore different evaluation metrics
- CLI Reference: Discover all command-line options

3. User Guide

3.1 Configuration Guide

This guide covers how to configure LLM Matrix test suites and runners.

3.1.1 Test Suite Configuration

Test suites are defined in YAML files with the following structure:

```yaml
name: suite-name
template: default-template-name
templates:
  template-name:
    system: System prompt text
    prompt: User prompt template with {variables}
    metrics:
      - metric_name1
      - metric_name2
matrix:
  hyperparameters:
    model: [model1, model2]
    temperature: [0.0, 0.7]
    # Other parameters as needed
cases:
  - input: Input text for the first case
    ideal: Expected output text
    tags: [tag1, tag2]
  - input: Input text for the second case
    ideal: Expected output text
    tags: [tag3]
    # Case-specific parameters to override defaults
    template: special-template
```

Core Components

TEMPLATES

Templates define how inputs are formatted for the LLM:

```yaml
templates:
  binary_qa:
    system: Answer the following question with YES or NO only.
    prompt: "Question: {input}"
    metrics:
      - binary_exact

  complex_qa:
    system: Provide a detailed answer to the following question.
    prompt: "Question: {input}\nPlease explain in detail."
    metrics:
      - qa_with_explanation
```

MATRIX CONFIGURATION

The matrix defines parameter combinations to test:

```yaml
matrix:
  hyperparameters:
    model: [gpt-4o, gpt-3.5-turbo, claude-3-opus]
    temperature: [0.0, 0.7]
    max_tokens: [100, 500]
```

LLM Matrix will run each test case with every combination of these parameters.

TEST CASES

Individual test cases define inputs and expected outputs:

```yaml
cases:
  - input: Is water wet?
    ideal: "YES"
    tags: [science, basic]
  - input: |
      What are the three primary colors?
    ideal: "The three primary colors are red, blue, and yellow."
    tags: [art]
    template: list_qa  # Override the default template
```

3.1.2 Runner Configuration

You can customize the runner behavior with a separate config file:

```
concurrency: 5  # Number of concurrent API calls
retries: 3      # Number of retry attempts for failed calls
timeout: 30     # Timeout in seconds for API calls
plugins:
  - citeseek    # Enable plugins
```

Pass this config to the CLI with:

```
llm-matrix run test-suite.yaml --runner-config runner-config.yaml
```

3.1.3 Advanced Configuration

Using Variables

You can use variables in your prompts:

```
templates:
  complex_prompt:
    system: "You are an expert in {domain}."
    prompt: "Question about {domain}: {input}"
```

Then in your cases:

```
cases:
  - input: What is DNA?
    ideal: "DNA is a molecule that carries genetic information..."
    variables:
      domain: genetics
```

Tags for Analysis

Tags help organize and filter your results:

```
cases:
  - input: What is photosynthesis?
    ideal: "Photosynthesis is a process used by plants..."
    tags: [biology, plants, difficulty:easy]
```

You can later filter or group results by these tags.

3.1.4 Examples

See the examples directory for complete examples of test suite configurations.

3.2 Evaluation Metrics

LLM Matrix provides various metrics to evaluate model responses against expected outputs. These metrics are specified in the template configuration.

3.2.1 Available Metrics

Basic Metrics

`binary_exact`

Evaluates binary (YES/NO) responses with exact matching.

```
templates:
  binary_qa:
    system: Answer the following question with YES or NO only.
    prompt: "Question: {input}"
    metrics:
      - binary_exact
```

- Score: 1.0 for exact match, 0.0 otherwise
- Best for: Simple YES/NO questions where only the answer matters, not explanation

`qa_simple`

Simple string matching for question answering.

```
metrics:
  - qa_simple
```

- Score: 1.0 for exact match, partial scores for close matches
- Best for: Factual questions with specific answers

Explanation Metrics

`qa_with_explanation`

Evaluates both the answer and explanation.

```
metrics:
  - qa_with_explanation
```

- Score: Combined score for answer correctness and explanation quality
- Best for: Questions requiring both correct answers and explanations

`binary_with_explanation`

For YES/NO questions with explanations.

```
metrics:
  - binary_with_explanation
```

- Score: 1.0 for correct binary answer with good explanation, lower for partial matches
- Best for: Binary questions where explanation is important

List Metrics

`list_comparison`

Compares lists of items.

```
metrics:
  - list_comparison
```

- Score: Based on overlap between expected and actual lists
- Best for: Enumeration tasks (e.g., "List the planets in the solar system")

3.2.2 Creating Custom Metrics

You can create custom metrics by implementing the `Metric` class:

```
from llm_matrix.metrics import Metric
from llm_matrix.schema import EvalResult, TestCase, LLMResponse

class MyCustomMetric(Metric):
    def evaluate(self, case: TestCase, response: LLMResponse) -> EvalResult:
        # Implement your evaluation logic
        score = calculate_score(case.ideal, response.text)

        return EvalResult(
            score=score,
            explanation="Reason for this score"
        )
```

Register your custom metric:

```
from llm_matrix.metrics import register_metric

register_metric("my_custom_metric", MyCustomMetric())
```

Then use it in your templates:

```
templates:
  my_template:
    system: Custom prompt
    prompt: "{input}"
    metrics:
      - my_custom_metric
```

3.2.3 Best Practices

- Choose metrics appropriate for your task type
- Consider using multiple metrics for complex tasks
- For binary tasks, prefer specialized binary metrics over general ones
- When using list-based metrics, ensure your ideal answer is formatted as expected
- For most accurate scoring, define clear evaluation criteria in your prompts

3.2.4 Metrics API Reference

For detailed API documentation of all metrics, see the Metrics API Reference.

3.3 Command Line Interface

LLM Matrix provides a command-line interface for running evaluations and managing results.

3.3.1 Basic Usage

```
llm-matrix run my-suite.yaml
```

3.3.2 Global Options

```
--verbose, -v      Increase verbosity (can be used multiple times)
--help             Show help message and exit
```

3.3.3 Commands

`run`

Run a test suite against specified models.

```
llm-matrix run <suite-path> [options]
```

ARGUMENTS

- `suite-path` : Path to the evaluation suite YAML file

OPTIONS

- `--store-path, -s <path>` : Path to the cache store (defaults to same directory as suite with .db extension)
- `--runner-config, -C <path>` : Path to the runner config file
- `--output-file, -o <path>` : Path to save output file
- `--output-dir, -D <path>` : Directory to save output files (defaults to suite-name-output)
- `--output-format, -F <format>` : Output format (csv, tsv, excel, jsonl, json, yaml)

EXAMPLES

```
# Basic usage
llm-matrix run my-suite.yaml

# Specify custom output location
llm-matrix run my-suite.yaml -D ./results

# Use custom runner configuration
llm-matrix run my-suite.yaml -C runner-config.yaml

# Increase verbosity for debugging
llm-matrix run my-suite.yaml -vv

# Output as Excel file
llm-matrix run my-suite.yaml -o results.xlsx -F excel
```

`convert`

Convert between different file formats.

```
llm-matrix convert <input-files> [options]
```

ARGUMENTS

- `input-files` : Paths to files to be converted

OPTIONS

- `--source, -s <field>` : Source field

- `--target, -t <field>` : Target field

3.3.4 Output Formats

LLM Matrix supports the following output formats:

- `csv` : Comma-separated values
- `tsv` : Tab-separated values
- `excel` : Microsoft Excel format (requires the `excel` extra)
- `jsonl` : JSON Lines format (one JSON object per line)
- `json` : Single JSON array with all results
- `yaml` : YAML format

3.3.5 Output Directory Structure

When using the default output directory, LLM Matrix creates:

```
suite-name-output/
├── by_model.csv            # Performance summary by model
├── by_model_ideal.csv      # Performance by model and ideal answer
├── grouped_by_input.tsv    # Detailed view of each test case
├── grouped_by_input.xlsx   # Excel version of the above
├── results.csv             # Raw results
├── results.html            # HTML view of raw results
├── summary.csv             # Statistical summary
└── summary.html            # HTML view of summary
```

4. API Reference

This section provides detailed documentation of the LLM Matrix Python API.

4.1 Core Modules

4.1.1 LLMRunner

The main class for running evaluations.

```python
from llm_matrix import LLMRunner

runner = LLMRunner(store_path="results.db")
results = runner.run(suite)
```

Source code in `src/llm_matrix/runner.py`

```python
@dataclass
class LLMRunner:

    store_path: Optional[Path] = None
    _aimodels: Optional[Dict[tuple, AIModel]] = None
    _store: Optional[Store] = None
    config: Optional[LLMRunnerConfig] = None

    def run(self, suite: Suite) -> List[TestCaseResult]:
        """
        Run the suite of cases

        :param suite:
        :return:
        """
        return list(self.run_iter(suite))

    def run_iter(self, suite: Suite) -> Iterator[TestCaseResult]:
        """
        Run the suite of cases iterating over the results.

        :param suite:
        :return:
        """
        logger.info(f"Running suite {suite.name}")
        for params in iter_hyperparameters(suite.matrix):
            logger.info(f"Running {len(suite.cases)} with {params}")
            for n, case in enumerate(suite.cases):
                logger.debug(f"Running case {n+1}/{len(suite.cases)}")
                result = self.run_case(case, params, suite)
                yield result

    def run_case(self, case: TestCase, params: Dict[str, Any], suite: Suite) -> TestCaseResult:
        logger.info(f"Running case {case.input} with {params}")
        store = self._get_store()
        cached = store.get_result(suite, case, params)
        if cached:
            return cached
        actual_params = copy(params)
        if self.config and "model" in params:
            model_logical_name = params["model"]
            actual_params["model"] = self.config.model_name_map.get(model_logical_name, model_logical_name)
            logger.info(f"Mapping model {model_logical_name} to {actual_params['model']}")
        model = self.get_aimodel(actual_params, suite=suite)
        template = self.get_template(case, suite)
        response = model.prompt(case.input, template=template, case=case)
        result = TestCaseResult(
            case=case,
            response=response,
            hyperparameters=params,
            metrics=template.metrics if template else None,
        )
        from llm_matrix.metrics import evaluate_result
        evaluate_result(result, runner=self)
        store.add_result(suite, result)
        return result

    def get_template(self, case, suite: Suite) -> Optional[Template]:
        tn = case.template
        if not tn:
            tn = suite.template
        if not tn:
            return None
        return suite.templates[tn]

    def get_aimodel(self, params: Dict[str, Any], suite: Suite=None) -> AIModel:
        if not self._aimodels:
            self._aimodels = {}
        key = tuple(sorted(params.items()))
        if key not in self._aimodels:
            if suite and suite.models:
                bespoke_models = suite.models
                model_name = params.get("model")
                if model_name and model_name in bespoke_models:
                    model_info = bespoke_models[model_name]
                    params = {**params, **model_info.parameters}
                    plugins = model_info.plugins or []
                    # TODO: proper mechanism for plugins
                    if "citeseek" in plugins:
                        from llm_matrix.plugins.citeseek_plugin import CiteseekPlugin
                        self._aimodels[key] = CiteseekPlugin(parameters=params)
            if key not in self._aimodels:
                self._aimodels[key] = AIModel(parameters=params)
        m = self._aimodels[key]
        return self._aimodels[key]

    def _get_store(self) -> Store:
        if not self._store:
            if not self.store_path:
                self.store_path = Path("results") / "cache.db"
            self.store_path.parent.mkdir(parents=True, exist_ok=True)
            self._store = Store(self.store_path)
        return self._store
```

4.2 `run(suite)`

Run the suite of cases

Parameters:

Name	Type	Description	Default
`suite`	`Suite`		*required*

Returns:

Type	Description
`List[TestCaseResult]`	

Source code in `src/llm_matrix/runner.py`

```python
def run(self, suite: Suite) -> List[TestCaseResult]:
    """
    Run the suite of cases

    :param suite:
    :return:
    """
    return list(self.run_iter(suite))
```

4.3 `run_iter(suite)`

Run the suite of cases iterating over the results.

Parameters:

Name	Type	Description	Default
`suite`	`Suite`		*required*

Returns:

Type	Description
`Iterator[TestCaseResult]`	

Source code in `src/llm_matrix/runner.py`

```python
def run_iter(self, suite: Suite) -> Iterator[TestCaseResult]:
    """
    Run the suite of cases iterating over the results.

    :param suite:
    :return:
    """
    logger.info(f"Running suite {suite.name}")
    for params in iter_hyperparameters(suite.matrix):
        logger.info(f"Running {len(suite.cases)} with {params}")
        for n, case in enumerate(suite.cases):
            logger.debug(f"Running case {n+1}/{len(suite.cases)}")
            result = self.run_case(case, params, suite)
            yield result
```

4.3.1 Schema

Data models for test cases, templates, responses, and results.

4.4 `StrictBaseModel`

Bases: `BaseModel`

Base class for Pydantic models that forbids extra fields.

Source code in `src/llm_matrix/schema.py`

```python
class StrictBaseModel(BaseModel):
    """
    Base class for Pydantic models that forbids extra fields.
    """
    model_config = ConfigDict(extra="forbid")

    def as_flat_dict(self, simple=True, prefix = None) -> Dict[str, Any]:
        """
        Convert the model to a flat dictionary.

        Suitable for conversion to a DataFrame.

        :param simple:
        :param prefix:
        :return:
        """
        def is_complex(v):
            if isinstance(v, dict):
                return True
            if isinstance(v, list) and any(isinstance(i, dict) for i in v):
                return True
            return False
        def mk_key(k):
            if prefix:
                return f"{prefix}_{k}"
            return k
        d = {mk_key(k): v for k, v in self.model_dump().items() if not is_complex(v)}
        return d
```

4.4.1 `as_flat_dict(simple=True, prefix=None)`

Convert the model to a flat dictionary.

Suitable for conversion to a DataFrame.

Parameters:

Name	Type	Description	Default
`simple`			`True`
`prefix`			`None`

Returns:

Type	Description
`Dict[str, Any]`	

Source code in `src/llm_matrix/schema.py`

```python
def as_flat_dict(self, simple=True, prefix = None) -> Dict[str, Any]:
    """
    Convert the model to a flat dictionary.

    Suitable for conversion to a DataFrame.

    :param simple:
    :param prefix:
    :return:
    """
    def is_complex(v):
        if isinstance(v, dict):
            return True
        if isinstance(v, list) and any(isinstance(i, dict) for i in v):
            return True
        return False
    def mk_key(k):
        if prefix:
            return f"{prefix}_{k}"
        return k
    d = {mk_key(k): v for k, v in self.model_dump().items() if not is_complex(v)}
    return d
```

4.5 `MetricEnum`

Bases: `Enum`

Enum for metrics to be evaluated.

Designed to be extensible.

Source code in `src/llm_matrix/schema.py`

```python
class MetricEnum(Enum):
    """
    Enum for metrics to be evaluated.

    Designed to be extensible.
    """
    QA_WITH_EXPLANATION = "qa_with_explanation"
    SIMPLE_QUESTION = "simple_question"
    LIST_MEMBERSHIP = "list_membership"
    RANKED_LIST = "ranked_list"
    NARRATIVE_SIMILARITY = "narrative_similarity"
    REVIEW = "review"
    BLEU = "bleu"
    ROUGE = "rouge"
    METEOR = "meteor"
```

4.6 `Response`

Bases: `StrictBaseModel`

Response from the AI model.

Source code in `src/llm_matrix/schema.py`

```python
class Response(StrictBaseModel):
    """
    Response from the AI model.
    """
    text : str = Field(..., description="The text of the response from the AI model")
    prompt: Optional[str] = Field(None, description="The prompt used to generate the response")
    system: Optional[str] = Field(None, description="The system prompt used to generate the response")
```

4.7 `Template`

Bases: `StrictBaseModel`

Template for generating prompts to the AI model.

Source code in `src/llm_matrix/schema.py`

```python
class Template(StrictBaseModel):
    """
    Template for generating prompts to the AI model.
    """
    system: Optional[FormatString] = Field(None, description="System prompt")
    prompt: Optional[FormatString] = Field(None, description="Prompt format string to the model")
    metrics: Optional[List[Metric]] = Field(None, description="Metrics to be evaluated")
```

4.8 `Matrix`

Bases: `StrictBaseModel`

Specifies a combination of hyperparameters to be evaluated.

Source code in `src/llm_matrix/schema.py`

```python
class Matrix(StrictBaseModel):
    """
    Specifies a combination of hyperparameters to be evaluated.
    """
    hyperparameters: Dict[Hyperparameter, List[Any]] = Field(..., description="Hyperparameters to be evaluated")
    exclude: Optional["Matrix"] = Field(None, description="Hyperparameters to be excluded")
```

4.9 `TestCase`

Bases: `StrictBaseModel`

Test case for the AI model.

Source code in `src/llm_matrix/schema.py`

```python
class TestCase(StrictBaseModel):
    """
    Test case for the AI model.
    """
    input: str = Field(..., description="Input to the model")
    original_input: Optional[Dict[str, Any]] = Field(
        None,
        description="""Original input to the model, prior to transformation into text.
        An example is a structured data record.
        """
    )
    #output: Optional[str] = Field(None, description="Provided output from the model")
    ideal: Optional[str] = Field(None, description="Ideal output from the model")
    template: Optional[TemplateName] = Field(None, description="Template for the test case")
    tags: Optional[List[str]] = Field(None, description="Tags for the test case")
    comments: Optional[List[str]] = Field(None, description="Comments for the test case")

    def as_flat_dict(self, **kwargs) -> Dict[str, Any]:
        top_dict = super().as_flat_dict(**kwargs)
        orig = self.original_input or {}
        return {**top_dict, **orig}
```

4.10 `TestCaseResult`

Bases: `StrictBaseModel`

Result of running an individual case on a model with hyperparameters.

Source code in `src/llm_matrix/schema.py`

```python
class TestCaseResult(StrictBaseModel):
    """
    Result of running an individual case on a model with hyperparameters.
    """
    case: TestCase = Field(..., description="Test case")
    response: Response = Field(..., description="Response from the model")
    hyperparameters: Dict[Hyperparameter, Any] = Field(..., description="Hyperparameters used for the test case")
    metrics: Optional[List[Metric]] = Field(None, description="Metrics to be evaluated")
    score: Optional[float] = Field(None, description="Score for the test case", ge=0, le=1)
    evaluation_message: Optional[str] = Field(None, description="Message from the evaluation")

    def as_flat_dict(self, **kwargs) -> Dict[str, Any]:
        top_dict = super().as_flat_dict(**kwargs)
        case_dict = self.case.as_flat_dict(prefix="case")
        response_dict = self.response.as_flat_dict(prefix="response")
        hyperparameters_dict = {k: str(v) for k, v in self.hyperparameters.items()}
        hyperparameters_str = "_".join(f"{k}={v}" for k, v in hyperparameters_dict.items())
        return {"hyperparameters": hyperparameters_str, **top_dict, **case_dict, **response_dict, **hyperparameters_dict}
```

4.10.1 Metrics

Evaluation metrics for comparing model outputs to expected answers.

4.11 `MetricEvaluator`

Bases: `ABC`

Base class for metric evaluators.

Source code in `src/llm_matrix/metrics.py`

```python
class MetricEvaluator(ABC):
    """Base class for metric evaluators."""

    @abstractmethod
    def evaluate(
        self,
        actual_output: str,
        expected_output: str,
        runner: Optional[LLMRunner] = None,
        result: Optional[TestCaseResult] = None
    ) -> float:
        """
        Evaluate the result and return a score between 0 and 1.

        :param actual_output: The actual output from the model
        :param expected_output: The expected output
        :param runner: The LLMRunner instance
        :param result: The TestCaseResult instance, for storing evaluation messages
        :return: A score between 0 and 1
        """
        pass
```

4.11.1 `evaluate(actual_output, expected_output, runner=None, result=None)` `abstractmethod`

Evaluate the result and return a score between 0 and 1.

Parameters:

Name	Type	Description	Default
`actual_output`	`str`	The actual output from the model	*required*
`expected_output`	`str`	The expected output	*required*
`runner`	`Optional[LLMRunner]`	The LLMRunner instance	`None`
`result`	`Optional[TestCaseResult]`	The TestCaseResult instance, for storing evaluation messages	`None`

Returns:

Type	Description
`float`	A score between 0 and 1

Source code in `src/llm_matrix/metrics.py`

```python
@abstractmethod
def evaluate(
    self,
    actual_output: str,
    expected_output: str,
    runner: Optional[LLMRunner] = None,
    result: Optional[TestCaseResult] = None
) -> float:
    """
    Evaluate the result and return a score between 0 and 1.

    :param actual_output: The actual output from the model
    :param expected_output: The expected output
    :param runner: The LLMRunner instance
    :param result: The TestCaseResult instance, for storing evaluation messages
    :return: A score between 0 and 1
    """
    pass
```

4.12 `QAWithExplanationEvaluator`

Bases: `MetricEvaluator`

Evaluator for QA with explanation metrics.

Source code in `src/llm_matrix/metrics.py`

```python
class QAWithExplanationEvaluator(MetricEvaluator):
    """Evaluator for QA with explanation metrics."""

    def evaluate(
        self,
        actual_output: str,
        expected_output: str,
        runner: Optional[LLMRunner] = None,
        result: Optional[TestCaseResult] = None
    ) -> float:
        """Evaluate QA with explanation result."""
        # First token regex
        pattern = re.compile(r"^(\w+)")
        match = pattern.match(actual_output)
        actual_answer = match.group(1).upper() if match else "OTHER"

        expected_match = pattern.match(expected_output)
        if not expected_match:
            logger.warning(f"Could not extract first token from expected output: {expected_output}")
            return 0.0

        expected_answer = expected_match.group(1).upper()

        if actual_answer == expected_answer:
            score = 1.0
        elif actual_answer == "OTHER" or expected_answer == "OTHER":
            score = 0.5
        else:
            score = 0.0

        return score
```

4.12.1 `evaluate(actual_output, expected_output, runner=None, result=None)`

Evaluate QA with explanation result.

Source code in `src/llm_matrix/metrics.py`

```python
def evaluate(
    self,
    actual_output: str,
    expected_output: str,
    runner: Optional[LLMRunner] = None,
    result: Optional[TestCaseResult] = None
) -> float:
    """Evaluate QA with explanation result."""
    # First token regex
    pattern = re.compile(r"^(\w+)")
    match = pattern.match(actual_output)
    actual_answer = match.group(1).upper() if match else "OTHER"

    expected_match = pattern.match(expected_output)
    if not expected_match:
        logger.warning(f"Could not extract first token from expected output: {expected_output}")
        return 0.0

    expected_answer = expected_match.group(1).upper()

    if actual_answer == expected_answer:
        score = 1.0
    elif actual_answer == "OTHER" or expected_answer == "OTHER":
        score = 0.5
    else:
        score = 0.0

    return score
```

4.13 `LLMBasedEvaluator`

Bases: `MetricEvaluator`

Base class for evaluators that use LLMs for scoring.

Source code in `src/llm_matrix/metrics.py`

```python
class LLMBasedEvaluator(MetricEvaluator):
    """Base class for evaluators that use LLMs for scoring."""

    def __init__(self, system_prompt: str, user_input_template: str):
        self.system_prompt = system_prompt
        self.user_input_template = user_input_template

    def evaluate(
        self,
        actual_output: str,
        expected_output: str,
        runner: Optional[LLMRunner] = None,
        result: Optional[TestCaseResult] = None
    ) -> float:
        """Evaluate using an LLM."""
        if not runner:
            logger.error("Runner is required for LLM-based evaluation")
            return 0.0

        eval_model = self._get_eval_model(runner)
        if not eval_model:
            logger.error("Could not get evaluation model")
            return 0.0

        eval_response = self._prompt_model(eval_model, actual_output, expected_output)
        eval_response_text = eval_response.text.strip()

        if result:
            result.evaluation_message = eval_response_text

        score = self._extract_score(eval_response_text)
        return score

    def _get_eval_model(self, runner: LLMRunner):
        """Get the evaluation model."""
        if runner.config and runner.config.evaluation_model_name:
            eval_model_name = runner.config.evaluation_model_name
        else:
            eval_model_name = DEFAULT_EVALUATION_MODEL_NAME

        return runner.get_aimodel({"model": eval_model_name})

    def _prompt_model(self, model, actual_output: str, expected_output: str):
        """Prompt the evaluation model."""
        user_input = self._format_user_input(actual_output, expected_output)

        return model.prompt(
            system_prompt=self.system_prompt,
            user_input=user_input
        )

    def _format_user_input(self, actual_output: str, expected_output: str) -> str:
        """Format the user input for the evaluation model."""
        return self.user_input_template.format(
            actual_output=actual_output,
            expected_output=expected_output
        )

    def _extract_score(self, response_text: str) -> float:
        """Extract a score from the LLM response."""
        pattern = re.compile(r"(\d+(\.\d+)?)")
        matches = pattern.match(response_text)

        if matches:
            return float(matches.group(1))
        else:
            logger.error(f"Could not parse score from {response_text}")
            raise ValueError(f"Could not parse score from {response_text}")
```

4.13.1 `evaluate(actual_output, expected_output, runner=None, result=None)`

Evaluate using an LLM.

Source code in `src/llm_matrix/metrics.py`

```python
def evaluate(
    self,
    actual_output: str,
    expected_output: str,
    runner: Optional[LLMRunner] = None,
    result: Optional[TestCaseResult] = None
) -> float:
    """Evaluate using an LLM."""
    if not runner:
        logger.error("Runner is required for LLM-based evaluation")
        return 0.0

    eval_model = self._get_eval_model(runner)
    if not eval_model:
        logger.error("Could not get evaluation model")
        return 0.0

    eval_response = self._prompt_model(eval_model, actual_output, expected_output)
    eval_response_text = eval_response.text.strip()

    if result:
        result.evaluation_message = eval_response_text

    score = self._extract_score(eval_response_text)
    return score
```

4.13.2 `_get_eval_model(runner)`

Get the evaluation model.

Source code in `src/llm_matrix/metrics.py`

```python
def _get_eval_model(self, runner: LLMRunner):
    """Get the evaluation model."""
    if runner.config and runner.config.evaluation_model_name:
        eval_model_name = runner.config.evaluation_model_name
    else:
        eval_model_name = DEFAULT_EVALUATION_MODEL_NAME

    return runner.get_aimodel({"model": eval_model_name})
```

4.13.3 `_prompt_model(model, actual_output, expected_output)`

Prompt the evaluation model.

Source code in `src/llm_matrix/metrics.py`

```python
def _prompt_model(self, model, actual_output: str, expected_output: str):
    """Prompt the evaluation model."""
    user_input = self._format_user_input(actual_output, expected_output)

    return model.prompt(
        system_prompt=self.system_prompt,
        user_input=user_input
    )
```

4.13.4 `_format_user_input(actual_output, expected_output)`

Format the user input for the evaluation model.

Source code in `src/llm_matrix/metrics.py`

```python
def _format_user_input(self, actual_output: str, expected_output: str) -> str:
    """Format the user input for the evaluation model."""
    return self.user_input_template.format(
        actual_output=actual_output,
        expected_output=expected_output
    )
```

4.13.5 `_extract_score(response_text)`

Extract a score from the LLM response.

Source code in `src/llm_matrix/metrics.py`

```python
def _extract_score(self, response_text: str) -> float:
    """Extract a score from the LLM response."""
    pattern = re.compile(r"(\d+(\.\d+)?)")
    matches = pattern.match(response_text)

    if matches:
        return float(matches.group(1))
    else:
        logger.error(f"Could not parse score from {response_text}")
        raise ValueError(f"Could not parse score from {response_text}")
```

4.14 `ListMembershipEvaluator`

Bases: `LLMBasedEvaluator`

Evaluator for list membership metrics.

Source code in `src/llm_matrix/metrics.py`

```python
class ListMembershipEvaluator(LLMBasedEvaluator):
    """Evaluator for list membership metrics."""

    def __init__(self):
        super().__init__(
            system_prompt=(
                "Check if all the expected list items are present in the text. "
                "Your response should be an overlap score between 0 and 1, where 1 is a perfect "
                "match (all members match) and 0 is the worst possible match (no members match). "
                "Your response should be the score followed by any explanatory text. "
                "For example, '0.5 Only half of the items matched'. "
                "Do NOT put ANY text before the score. ALWAYS start with the score. "
                "Note the text you are evaluating may have additional verbiage, do not "
                "penalize this. Your task is just to determine if the list is presented clearly "
                "and if the items match"
            ),
            user_input_template=(
                "The expected list is: {expected_output}. "
                "The text: {actual_output}. "
            )
        )
```

4.15 `ReviewEvaluator`

Bases: `LLMBasedEvaluator`

Evaluator for review metrics.

Source code in `src/llm_matrix/metrics.py`

```python
class ReviewEvaluator(LLMBasedEvaluator):
    """Evaluator for review metrics."""

    def __init__(self):
        super().__init__(
            system_prompt=(
                "Review the output for correctness, completeness, and clarity. "
                "The response should be a score between 0 (worst) and 1 (best). "
                "Your response should be the score followed by any explanatory text. "
                "For example, '0.3 The response has many inaccuracies'. "
            ),
            user_input_template="The output to score is: {actual_output}. "
        )
```

4.16 `RankedListEvaluator`

Bases: `LLMBasedEvaluator`

Evaluator for ranked list metrics.

Source code in `src/llm_matrix/metrics.py`

```python
class RankedListEvaluator(LLMBasedEvaluator):
    """Evaluator for ranked list metrics."""

    def __init__(self):
        super().__init__(
            system_prompt=(
                "Compare the ranked list to the expected output. "
                "The response should be a score between 0 and 1. "
                "If the item ranked first is equal to the expected item, score is 1. "
                "If there is no overlap between the ranked list and the expected list, score is 0. "
                "Otherwise score according to rank, with 0.5 for 2nd, 0.25 for 3rd, and so on."
            ),
            user_input_template=(
                "The expected answer is: {expected_output}. "
                "The output to score is: {actual_output}. "
            )
        )
```

4.17 `SimpleQuestionEvaluator`

Bases: `LLMBasedEvaluator`

Evaluator for simple question metrics.

Source code in `src/llm_matrix/metrics.py`

```python
class SimpleQuestionEvaluator(LLMBasedEvaluator):
    """Evaluator for simple question metrics."""

    def __init__(self):
        super().__init__(
            system_prompt=(
                "Compare the answer given to the expected output. "
                "The response should be a score between 0 and 1. "
                "The answer should be provided first, explanations may follow "
                "A precise correct answer is 1, a wrong answer is 0."
                "You can use values in between for imprecise answers"
            ),
            user_input_template=(
                "The expected answer is: {expected_output}. "
                "The output to score is: {actual_output}. "
            )
        )
```

4.18 `register_metric_evaluator(metric_name, evaluator)`

Register a custom metric evaluator.

Parameters:

Name	Type	Description	Default
`metric_name`	`str`	The name of the metric	*required*
`evaluator`	`MetricEvaluator`	The evaluator instance	*required*

Source code in `src/llm_matrix/metrics.py`

```python
def register_metric_evaluator(metric_name: str, evaluator: MetricEvaluator) -> None:
    """
    Register a custom metric evaluator.

    :param metric_name: The name of the metric
    :param evaluator: The evaluator instance
    """
    METRIC_REGISTRY[metric_name] = evaluator
```

4.19 `evaluate_result(result, runner=None)`

Evaluate the result of a test case.

Example:

```
>>> result = TestCaseResult(
...     case=TestCase(input="What is II+IV?", ideal="VI. Blah"),
...     response=Response(text="VI"),
...     hyperparameters={"model": "gpt-4o"},
...     metrics=["qa_with_explanation"],
... )
>>> evaluate_result(result)
>>> result.score
1.0
>>> result.response.text = "VII"
>>> evaluate_result(result)
>>> result.score
0.0
>>> result.response.text = "Other"
>>> evaluate_result(result)
>>> result.score
0.5
```

Parameters:

Name	Type	Description	Default
`result`	`TestCaseResult`	The test case result to evaluate	*required*
`runner`	`Optional[LLMRunner]`	The LLMRunner instance	`None`

Source code in `src/llm_matrix/metrics.py`

```python
def evaluate_result(result: TestCaseResult, runner: Optional[LLMRunner] = None):
    """
    Evaluate the result of a test case.

    Example:

        >>> result = TestCaseResult(
        ...     case=TestCase(input="What is II+IV?", ideal="VI. Blah"),
        ...     response=Response(text="VI"),
        ...     hyperparameters={"model": "gpt-4o"},
        ...     metrics=["qa_with_explanation"],
        ... )
        >>> evaluate_result(result)
        >>> result.score
        1.0
        >>> result.response.text = "VII"
        >>> evaluate_result(result)
        >>> result.score
        0.0
        >>> result.response.text = "Other"
        >>> evaluate_result(result)
        >>> result.score
        0.5

    :param result: The test case result to evaluate
    :param runner: The LLMRunner instance
    """
    actual_output = result.response.text
    expected_output = result.case.ideal

    scores = []

    for metric_name in result.metrics or []:
        if metric_name not in METRIC_REGISTRY:
            raise NotImplementedError(f"Metric {metric_name} not implemented")

        evaluator = METRIC_REGISTRY[metric_name]
        try:
            score = evaluator.evaluate(
                actual_output=actual_output,
                expected_output=expected_output,
                runner=runner,
                result=result
            )
            scores.append(score)
        except Exception as e:
            logger.error(f"Error evaluating metric {metric_name}: {e}")
            # Optionally fall back to a default score
            # scores.append(0.0)
            # Or re-raise the exception
            raise

    if scores:
        result.score = sum(scores) / len(scores)
```

4.19.1 Store

Cache for storing results and avoiding redundant API calls.

4.20 `Store` `dataclass`

A persistent store using DuckDB to cache test results with JSON support for Pydantic models.

Example:

```
>>> store = Store("test-cache.db")
>>> case = TestCase(input="1+1", ideal="2")
>>> suite = Suite(name="test", cases=[case], matrix={"hyperparameters": {}})
>>> response = Response(text="2")
>>> result = TestCaseResult(case=case, response=response, hyperparameters={"model": "gpt-4"})
>>> store.add_result(suite, result)
>>> cached = store.get_result(suite, case, {"model": "gpt-4"})
>>> assert cached.response == response
```

To use an in-memory database, pass `None` as the `db_path`:

```
>>> store = Store(None)
```

Source code in `src/llm_matrix/store.py`

```python
@dataclass
class Store:
    """A persistent store using DuckDB to cache test results with JSON support for Pydantic models.

    Example:

        >>> store = Store("test-cache.db")
        >>> case = TestCase(input="1+1", ideal="2")
        >>> suite = Suite(name="test", cases=[case], matrix={"hyperparameters": {}})
        >>> response = Response(text="2")
        >>> result = TestCaseResult(case=case, response=response, hyperparameters={"model": "gpt-4"})
        >>> store.add_result(suite, result)
        >>> cached = store.get_result(suite, case, {"model": "gpt-4"})
        >>> assert cached.response == response

    To use an in-memory database, pass `None` as the `db_path`:

        >>> store = Store(None)

    """
    db_path: Optional[str] = None
    _conn: Optional[duckdb.DuckDBPyConnection] = None

    def __post_init__(self):
        """Initialize the database connection and create the table if it doesn't exist."""
        self._conn = duckdb.connect(str(self.db_path) if self.db_path else ":memory:")
        # Using JSON type for storing Pydantic models and hyperparameters
        self._conn.execute("""
            CREATE TABLE IF NOT EXISTS results (
                suite_name VARCHAR,
                test_case VARCHAR,
                ideal VARCHAR,
                hyperparameters JSON,
                result JSON,
                PRIMARY KEY (suite_name, test_case, ideal, hyperparameters)
            )
        """)

    def add_result(self, suite: Suite, result: TestCaseResult):
        """Add a result to the store."""
        self._conn.execute("""
            INSERT OR REPLACE INTO results
            (suite_name, test_case, ideal, hyperparameters, result)
            VALUES (?, ?, ?, ?, ?)
        """, (
            *unique_key(suite, result.case, result.hyperparameters),
            result.model_dump_json(exclude_unset=True),
        ))
        logger.debug(f"Added result for {suite.name} {result.case} {result.hyperparameters}")
        self._conn.commit()

    def get_result(self, suite: Suite, case: TestCase, hyperparameters: dict) -> Optional[TestCaseResult]:
        """Get a result from the store."""
        result = self._conn.execute("""
            SELECT result
            FROM results
            WHERE suite_name = ?
            AND test_case = ?
            AND ideal = ?
            AND hyperparameters = ?
        """, (
            *unique_key(suite, case, hyperparameters),
        )).fetchone()

        logger.debug(f"Present: {result is not None} when looking up {suite.name} {case} {hyperparameters}")

        if result:
            return TestCaseResult.model_validate_json(result[0])
        return None

    @property
    def size(self) -> int:
        """Get the number of results in the store."""
        return self._conn.execute("SELECT COUNT(*) FROM results").fetchone()[0]

    def __del__(self):
        """Close the database connection when the object is destroyed."""
        if self._conn:
            self._conn.close()
```

4.20.1 `size` `property`

Get the number of results in the store.

4.20.2 `__post_init__()`

Initialize the database connection and create the table if it doesn't exist.

Source code in `src/llm_matrix/store.py`

```python
def __post_init__(self):
    """Initialize the database connection and create the table if it doesn't exist."""
    self._conn = duckdb.connect(str(self.db_path) if self.db_path else ":memory:")
    # Using JSON type for storing Pydantic models and hyperparameters
    self._conn.execute("""
        CREATE TABLE IF NOT EXISTS results (
            suite_name VARCHAR,
            test_case VARCHAR,
            ideal VARCHAR,
            hyperparameters JSON,
            result JSON,
            PRIMARY KEY (suite_name, test_case, ideal, hyperparameters)
        )
    """)
```

4.20.3 `add_result(suite, result)`

Add a result to the store.

Source code in `src/llm_matrix/store.py`

```python
def add_result(self, suite: Suite, result: TestCaseResult):
    """Add a result to the store."""
    self._conn.execute("""
        INSERT OR REPLACE INTO results
        (suite_name, test_case, ideal, hyperparameters, result)
        VALUES (?, ?, ?, ?, ?)
    """, (
        *unique_key(suite, result.case, result.hyperparameters),
        result.model_dump_json(exclude_unset=True),
    ))
    logger.debug(f"Added result for {suite.name} {result.case} {result.hyperparameters}")
    self._conn.commit()
```

4.20.4 `get_result(suite, case, hyperparameters)`

Get a result from the store.

Source code in `src/llm_matrix/store.py`

```python
def get_result(self, suite: Suite, case: TestCase, hyperparameters: dict) -> Optional[TestCaseResult]:
    """Get a result from the store."""
    result = self._conn.execute("""
        SELECT result
        FROM results
        WHERE suite_name = ?
        AND test_case = ?
        AND ideal = ?
        AND hyperparameters = ?
    """, (
        *unique_key(suite, case, hyperparameters),
    )).fetchone()

    logger.debug(f"Present: {result is not None} when looking up {suite.name} {case} {hyperparameters}")

    if result:
        return TestCaseResult.model_validate_json(result[0])
    return None
```

4.20.5 `__del__()`

Close the database connection when the object is destroyed.

Source code in `src/llm_matrix/store.py`

```python
def __del__(self):
    """Close the database connection when the object is destroyed."""
    if self._conn:
        self._conn.close()
```

4.21 `unique_key(suite, case, hyperparameters)`

Generate a unique key for a test result.

Source code in `src/llm_matrix/store.py`

```python
def unique_key(suite: Suite, case: TestCase, hyperparameters: dict) -> tuple:
    """Generate a unique key for a test result."""
    suite_name = suite.name
    if suite.version:
        suite_name += f"--{suite.version}"

    def empty(v):
        return v is None or (isinstance(v, (str, list, dict)) and not v)
    return suite_name, case.input, case.ideal or "", {k: v for k, v in hyperparameters.items() if not empty(v)}
```

4.22 Using the API

4.22.1 Basic Example

```python
from llm_matrix import LLMRunner
from llm_matrix.schema import load_suite

# Load a test suite from YAML
suite = load_suite("my-suite.yaml")

# Create a runner
runner = LLMRunner(store_path="results.db")

# Run the suite
results = runner.run(suite)

# Process results
for result in results:
    print(f"Case: {result.case.input}")
    print(f"Score: {result.score}")
    print(f"Response: {result.response.text}")
```

4.22.2 Custom Configuration

```python
from llm_matrix import LLMRunner
from llm_matrix.runner import LLMRunnerConfig

# Create custom configuration
config = LLMRunnerConfig(
    concurrency=5,
    retries=3,
    timeout=30
)

# Initialize runner with configuration
runner = LLMRunner(
    store_path="results.db",
    config=config
)
```

4.22.3 Working with Results

```python
from llm_matrix.schema import results_to_dataframe

# Convert results to pandas DataFrame
df = results_to_dataframe(results)
```

```python
# Calculate statistics
print(df.describe())

# Group by model
model_performance = df.groupby("model").agg({
    "score": ["mean", "std", "count"]
})
```

4.22.4 Creating Test Suites Programmatically

```python
from llm_matrix.schema import TestSuite, TestCase, Template

# Create templates
templates = {
    "qa_template": Template(
        system="Answer the following question accurately.",
        prompt="{input}",
        metrics=["qa_simple"]
    )
}

# Create test cases
cases = [
    TestCase(
        input="What is the capital of France?",
        ideal="Paris",
        tags=["geography"]
    ),
    TestCase(
        input="What is 2+2?",
        ideal="4",
        tags=["math"]
    )
]

# Create test suite
suite = TestSuite(
    name="programmatic-suite",
    template="qa_template",
    templates=templates,
    cases=cases,
    matrix={
        "hyperparameters": {
            "model": ["gpt-3.5-turbo", "gpt-4o"],
            "temperature": [0.0]
        }
    }
)

# Run the suite
runner = LLMRunner(store_path="results.db")
results = runner.run(suite)
```

5. Development

5.1 Contributing to LLM Matrix

We welcome contributions to LLM Matrix! This document provides guidelines and instructions for contributing.

5.1.1 Development Environment

Prerequisites

- Python 3.11 or higher
- Poetry for dependency management

Setup

1. Clone the repository:

```
git clone https://github.com/monarch-initiative/llm-matrix.git
cd llm-matrix
```

1. Install dependencies with Poetry:

```
poetry install
```

1. Activate the virtual environment:

```
poetry shell
```

5.1.2 Development Workflow

Code Style

We use the following tools to maintain code quality:

- **Black**: Code formatter with 120 character line length
- **Ruff**: Linter with various rules (flake8, isort, etc.)
- **MyPy**: Static type checking

To check code quality:

```
# Run type checking
make mypy

# Run linters
tox -e lint

# Auto-fix linting issues where possible
tox -e lint-fix

# Check spelling
tox -e codespell
```

Running Tests

```
# Run all tests, doctest, mypy, and codespell
make test

# Run only pytest tests
make pytest

# Run a specific test
poetry run pytest tests/test_file.py::TestClass::test_function
```

5.1.3 Making Changes

1. Create a new branch for your changes:

```
git checkout -b feature/your-feature-name
```

1. Make your changes following our code style guidelines.
2. Add tests for new functionality.
3. Ensure all tests pass:

```
make test
```

1. Update documentation as needed.

5.1.4 Submitting Changes

1. Push your changes to your fork:

```
git push origin feature/your-feature-name
```

1. Create a pull request on GitHub.
2. Ensure the PR description clearly describes the problem and solution.

5.1.5 Adding New Metrics

To add a new evaluation metric:

1. Create a new class in `src/llm_matrix/metrics.py` that inherits from `Metric`:

```
class MyNewMetric(Metric):
    def evaluate(self, case: TestCase, response: LLMResponse) -> EvalResult:
        # Implement your evaluation logic
        score = calculate_score(case.ideal, response.text)

        return EvalResult(
            score=score,
            explanation="Reason for this score"
        )
```

1. Register your metric:

```
register_metric("my_new_metric", MyNewMetric())
```

1. Add tests for your metric in `tests/test_metrics.py`.
2. Update documentation in `docs/metrics/index.md`.

5.1.6 Creating Plugins

To create a new plugin:

1. Create a new file in `src/llm_matrix/plugins/` (e.g., `my_plugin.py`).
2. Implement the plugin interface:

```
from llm_matrix.plugins.plugin import Plugin

class MyPlugin(Plugin):
    def name(self) -> str:
        return "my_plugin"
    # Implement required methods
```

1. Register your plugin in `src/llm_matrix/plugins/__init__.py`.
2. Add tests for your plugin in `tests/test_plugins/`.
3. Update documentation as needed.

5.1.7 Documentation

We use MkDocs with the Material theme for documentation:

1. Update documentation as needed in the `docs/` directory.
2. Preview documentation locally:

```
mkdocs serve
```

1. Ensure all links and references are correct.

5.1.8 Release Process

1. Update version number in `pyproject.toml`.
2. Update CHANGELOG.md with changes in the release.
3. Create a tag with the new version number.
4. A GitHub Action will automatically build and publish to PyPI.

5.1.9 Questions?

If you have any questions, please open an issue on GitHub or reach out to the maintainers.