- Read
- Write
- Glob
- Grep

DSPy Evaluation Suite

Goal

Systematically evaluate DSPy programs using built-in and custom metrics with parallel execution.

When to Use

- Measuring program performance before/after optimization
- Comparing different program variants
- Establishing baselines
- Validating production readiness

Inputs

Input	Type	Description
`program`	`dspy.Module`	Program to evaluate
`devset`	`list[dspy.Example]`	Evaluation examples
`metric`	`callable`	Scoring function
`num_threads`	`int`	Parallel threads

Outputs

Output	Type	Description
`score`	`float`	Average metric score
`results`	`list`	Per-example results

Workflow

Phase 1: Setup Evaluator

```python
from dspy.evaluate import Evaluate

evaluator = Evaluate(
    devset=devset,
    metric=my_metric,
    num_threads=8,
    display_progress=True
)
```

Phase 2: Run Evaluation

```python
score = evaluator(my_program)
print(f"Score: {score:.2%}")
```

Built-in Metrics

answer_exact_match

```python
import dspy

# Normalized, case-insensitive comparison
metric = dspy.evaluate.answer_exact_match
```

SemanticF1

LLM-based semantic evaluation:

```python
from dspy.evaluate import SemanticF1

semantic = SemanticF1()
score = semantic(example, prediction)
```

Custom Metrics

Basic Metric

```python
def exact_match(example, pred, trace=None):
    """Returns bool, int, or float."""
    return example.answer.lower().strip() == pred.answer.lower().strip()
```

Multi-Factor Metric

```python
def quality_metric(example, pred, trace=None):
    """Score based on multiple factors."""
    score = 0.0

    # Correctness (50%)
    if example.answer.lower() in pred.answer.lower():
        score += 0.5

    # Conciseness (25%)
    if len(pred.answer.split()) <= 20:
        score += 0.25

    # Has reasoning (25%)
    if hasattr(pred, 'reasoning') and pred.reasoning:
        score += 0.25

    return score
```

GEPA-Compatible Metric

```python
def feedback_metric(example, pred, trace=None):
    """Returns (score, feedback) for GEPA optimizer."""
    correct = example.answer.lower() in pred.answer.lower()

    if correct:
        return 1.0, "Correct answer provided."
    else:
        return 0.0, f"Expected '{example.answer}', got '{pred.answer}'"
```

Production Example

```python
import dspy
from dspy.evaluate import Evaluate, SemanticF1
import json
import logging
from typing import Optional
from dataclasses import dataclass

logger = logging.getLogger(__name__)

@dataclass
class EvaluationResult:
    score: float
    num_examples: int
    correct: int
    incorrect: int
    errors: int

def comprehensive_metric(example, pred, trace=None) -> float:
    """Multi-dimensional evaluation metric."""
    scores = []

    # 1. Correctness
    if hasattr(example, 'answer') and hasattr(pred, 'answer'):
        correct = example.answer.lower().strip() in pred.answer.lower().strip()
        scores.append(1.0 if correct else 0.0)

    # 2. Completeness (answer not empty or error)
    if hasattr(pred, 'answer'):
        complete = len(pred.answer.strip()) > 0 and "error" not in pred.answer.lower()
        scores.append(1.0 if complete else 0.0)

    # 3. Reasoning quality (if available)
    if hasattr(pred, 'reasoning'):
        has_reasoning = len(str(pred.reasoning)) > 20
        scores.append(1.0 if has_reasoning else 0.5)

    return sum(scores) / len(scores) if scores else 0.0

class EvaluationSuite:
    def __init__(self, devset, num_threads=8):
        self.devset = devset
        self.num_threads = num_threads

    def evaluate(self, program, metric=None) -> EvaluationResult:
        """Run full evaluation with detailed results."""
        metric = metric or comprehensive_metric

        evaluator = Evaluate(
            devset=self.devset,
            metric=metric,
            num_threads=self.num_threads,
            display_progress=True,
            return_all_scores=True
        )

        score, results = evaluator(program)

        correct = sum(1 for r in results if r >= 0.5)
        errors = sum(1 for r in results if r == 0)

        return EvaluationResult(
            score=score,
            num_examples=len(self.devset),
            correct=correct,
            incorrect=len(self.devset) - correct - errors,
            errors=errors
        )

    def compare(self, programs: dict, metric=None) -> dict:
        """Compare multiple programs."""
        results = {}

        for name, program in programs.items():
            logger.info(f"Evaluating: {name}")
            results[name] = self.evaluate(program, metric)

        # Rank by score
        ranked = sorted(results.items(), key=lambda x: x[1].score, reverse=True)

        print("\n=== Comparison Results ===")
        for rank, (name, result) in enumerate(ranked, 1):
            print(f"{rank}. {name}: {result.score:.2%}")

        return results

    def export_report(self, program, output_path: str, metric=None):
        """Export detailed evaluation report."""
        result = self.evaluate(program, metric)

        report = {
            "summary": {
                "score": result.score,
                "total": result.num_examples,
                "correct": result.correct,
                "accuracy": result.correct / result.num_examples
            },
            "config": {
                "num_threads": self.num_threads,
                "num_examples": len(self.devset)
            }
        }

        with open(output_path, 'w') as f:
            json.dump(report, f, indent=2)

        logger.info(f"Report saved to {output_path}")
        return report

# Usage
suite = EvaluationSuite(devset, num_threads=8)

# Single evaluation
result = suite.evaluate(my_program)
print(f"Score: {result.score:.2%}")

# Compare variants
results = suite.compare({
    "baseline": baseline_program,
    "optimized": optimized_program,
    "finetuned": finetuned_program
})
```

Best Practices

1. **Hold out test data** - Never optimize on evaluation set
2. **Multiple metrics** - Combine correctness, quality, efficiency
3. **Statistical significance** - Use enough examples (100+)
4. **Track over time** - Version control evaluation results

Limitations

- Metrics are task-specific; no universal measure
- SemanticF1 requires LLM calls (cost)
- Parallel evaluation can hit rate limits
- Edge cases may not be captured