Scaling LLM Evaluation with DSPy

Abstract

As LLM adoption moves from "chatbot" prototypes to production pipelines, the brittleness of manual prompt engineering becomes a critical bottleneck. This case study details our transition from "vibe-based" evaluation to a rigorous, programmatic optimization loop using **DSPy**. By implementing **Assertion-Driven Feedback** and leveraging the `MIPROv2` optimizer, we achieved a 99% accuracy rate in structured data extraction tasks, reduced token usage by 40%, and eliminated the need for manual prompt tuning.

1. The Problem: The "Prompt Engineering" Treadmill

In our legacy architecture, extracting structured medical data from unstructured clinical notes was a manual, fragile process.

- **The Workflow:** An engineer would write a prompt, test it on 5 examples, see a failure, tweak the prompt (e.g., "IMPORTANT: output valid JSON only!"), and re-test.
- **The Result:** A 2,000-token "mega-prompt" that worked 85% of the time but failed unpredictably on edge cases.
- **The Metric:** "Vibes." If the output looked good to the engineer, it was shipped.

We hit a wall when we needed to scale to 100,000 documents. The error rate of 15% was unacceptable for clinical data, and the latency of long prompts was cost-prohibitive.

2. The Solution: Programmatic Optimization with DSPy

We re-architected the pipeline using **DSPy (Declarative Self-Improving Language Programs)**. The core shift was treating the LLM interaction not as a string manipulation task, but as a modular *program* with defined signatures, metrics, and optimizers.

2.1 Defining the Signature

Instead of a long prompt, we defined a typed signature:

```
class ClinicalExtraction(dspy.Signature):
    """Extract patient vitals and medication history from clinical notes."""

    clinical_note = dspy.InputField(desc="The doctor's unstructured notes.")
    patient_vitals = dspy.OutputField(desc="JSON object containing BP, HR, Temp.")
    medications = dspy.OutputField(desc="List of medications with dosage and
frequency.")
```

2.2 Implementing Assertions for Reliability

To reach 99% accuracy, we couldn't rely on the model "trying its best." We implemented **DSPy Assertions** (`dspy.Assert` and `dspy.Suggest`) to enforce constraints at runtime.

- **Logic:**
    1. **Format Check:** `dspy.Assert(is_valid_json(output), "Output must be valid JSON")`
    2. **Schema Validation:** `dspy.Assert(validate_schema(output), "Missing required fields: BP, HR")`
    3. **Hallucination Check:** `dspy.Suggest(extract in clinical_note, "Extracted value must appear in source text")`

When an assertion fails, the DSPy pipeline automatically *backtracks*, providing the error message (e.g., "Missing required fields") back to the model as a "correction prompt." This improved zero-shot pass rates from 68% to 92% immediately.

3. The Optimization Loop: Replacing Human Tuning

The final leap to 99% came from the **MIPROv2 (Many-Instruction PRompt Optimizer)**.

We built a "Golden Dataset" of 100 verified examples. Instead of manually tweaking the prompt, we ran the optimizer:

```
optimizer = dspy.MIPROv2(metric=extraction_accuracy_metric)
optimized_program = optimizer.compile(
    student=extraction_module,
    trainset=golden_dataset,
    max_bootstrapped_demos=5,
    max_labeled_demos=5
)
```

What the Optimizer Did:

1. **Instruction Search:** It generated 50 variations of instructions (e.g., "Act as a scribe...", "You are a data parser...").
2. **Few-Shot Selection:** It selected the *mathematically optimal* set of few-shot examples that differentiated edge cases.
3. **Ensemble Scoring:** It evaluated the combinations against our metric and selected the highest-performing prompt.

4. Results & Impact

Metric	Manual Prompting	DSPy + Assertions	DSPy + MIPROv2
Accuracy	85% (Unstable)	92%	**99.1%**
Development Time	3 Days / Iteration	2 Hours	**15 Minutes**
Token Cost	$4.50 / 1k docs	$4.80 (due to retries)	**$2.10 (Optimized)**

Key Takeaways

1. **Assertions are Unit Tests for AI:** They catch errors before they leave the pipeline.
2. **Optimization > Engineering:** Let the algorithm find the best prompt. It finds patterns humans miss.
3. **Metrics Drive Quality:** You cannot optimize what you do not measure. Defining a rigorous `accuracy_metric` was the hardest but most valuable part of the transition.

5. Conclusion

Scaling LLM evaluation requires abandoning the art of "prompt whispering" for the science of **AI Systems Engineering**. By adopting DSPy, we turned a fragile, manual workflow into a robust, self-improving pipeline that gets smarter with every data point we add to our validation set.

Resources

- DSPy GitHub Repository
- LM Assertions Paper