DSPy Framework Documentation

Comprehensive reference for the DSPy framework - a declarative system for programming and optimizing LLM applications.

Table of Contents

Core Concepts

DSPy abstracts LLM interaction into three layers:

1. **Signatures** - Declarative I/O specifications
2. **Modules** - Composable program building blocks
3. **Adapters** - Bridge between DSPy and LLM APIs

```
import dspy

# Configure the LM globally
dspy.configure(lm=dspy.LM("openai/gpt-4o-mini"))
```

Signatures

A **Signature** defines what a module does, not how. DSPy compiles it into an effective prompt.

Inline Signatures

```
# Simple
"question -> answer"

# With types
"sentence -> sentiment: bool"

# Multiple fields
"context: list[str], question: str -> answer: str"
```

Class-based Signatures

```
from typing import Literal
import dspy

class Emotion(dspy.Signature):
    """Classify the emotion of the sentence."""
    sentence: str = dspy.InputField(desc="The text to classify.")
    sentiment: Literal['sadness','joy','love','anger','fear','surprise'] = dspy.Out-
putField()
```

Type Hints

Supported types:
- Basic: `str`, `int`, `float`, `bool`
- Collections: `list[str]`, `dict[str, int]`
- Optional: `Optional[str]`
- Literals: `Literal['a', 'b', 'c']`
- Pydantic models for complex structures

Modules

Modules are composable building blocks with learnable parameters.

dspy.Predict

Basic prediction without reasoning:

```
classify = dspy.Predict('sentence -> sentiment: bool')
response = classify(sentence="It's a charming journey.")
print(response.sentiment)  # True
```

dspy.ChainOfThought

Adds step-by-step reasoning:

```
qa = dspy.ChainOfThought('question -> answer')
response = qa(question="What's special about ColBERT?")
print(response.reasoning)  # Step-by-step explanation
print(response.answer)
```

dspy.ReAct

Agent with tool use:

```
def search_wiki(query: str) -> str:
    """Search Wikipedia for information."""
    results = dspy.ColBERTv2(url='http://20.102.90.50:2017/wiki17_abstracts')(query,
k=1)
    return results[0]['text']

def calculate(expression: str) -> float:
    """Evaluate a math expression."""
    return dspy.PythonInterpreter({}).execute(expression)

agent = dspy.ReAct("question -> answer: float", tools=[search_wiki, calculate])
result = agent(question="What is 100 divided by the year Columbus discovered Amer-
ica?")
```

Adapters

Adapters translate between DSPy structures and LLM APIs.

dspy.ChatAdapter (Default)

- Human-readable marker format: `[[ ## field_name ## ]]`
- Auto-fallback to JSONAdapter on failure
- Good for debugging

dspy.JSONAdapter

- Native JSON output mode
- Lower latency, more reliable parsing
- Best for structured data

```
# Explicitly set adapter
dspy.configure(
    lm=dspy.LM("openai/gpt-4o-mini"),
    adapter=dspy.JSONAdapter()
)
```

Optimizers

Few-Shot Learning Optimizers

LabeledFewShot

Simplest optimizer - random selection from training set.

```
from dspy.teleprompt import LabeledFewShot

optimizer = LabeledFewShot(k=3)  # Use 3 examples
compiled = optimizer.compile(program, trainset=trainset)
```

Parameter	Description
`k`	Number of examples to include

BootstrapFewShot

Generates demonstrations using a teacher model.

```
from dspy.teleprompt import BootstrapFewShot

optimizer = BootstrapFewShot(
    metric=dspy.evaluate.answer_exact_match,
    max_bootstrapped_demos=4,
    max_labeled_demos=4,
    teacher_settings={'lm': dspy.LM("openai/gpt-4o")}
)
compiled = optimizer.compile(program, trainset=trainset)
```

Parameter	Description	Default
`metric`	Evaluation function	Required
`max_bootstrapped_demos`	Max generated demos	4
`max_labeled_demos`	Max labeled demos	16
`teacher_settings`	Teacher LM config	None

BootstrapFewShotWithRandomSearch

Extends Bootstrap with random search over demo sets.

```
from dspy.teleprompt import BootstrapFewShotWithRandomSearch

optimizer = BootstrapFewShotWithRandomSearch(
    metric=metric,
    max_bootstrapped_demos=4,
    max_labeled_demos=4,
    num_candidate_programs=10,
    num_threads=4
)
```

Parameter	Description	Default
`num_candidate_programs`	Candidates to evaluate	16
`num_threads`	Parallel threads	6

KNNFewShot

Dynamic example selection via k-NN.

```python
from dspy.teleprompt import KNNFewShot

optimizer = KNNFewShot(k=3, trainset=trainset)
```

Instruction Optimizers

COPRO

Iterative instruction refinement.

```python
from dspy.teleprompt import COPRO

optimizer = COPRO(
    metric=metric,
    breadth=10,  # Initial candidates
    depth=3,     # Refinement iterations
    prompt_model=dspy.LM("openai/gpt-4o")
)
```

Parameter	Description	Default
`breadth`	Initial candidates	10
`depth`	Refinement iterations	3
`prompt_model`	LM for generation	Default LM

MIPROv2

State-of-the-art: Bayesian optimization of instructions + demos.

```python
import dspy

optimizer = dspy.MIPROv2(
    metric=dspy.evaluate.answer_exact_match,
    auto="medium",  # "light", "medium", "heavy"
    num_threads=24
)
compiled = optimizer.compile(program, trainset=trainset)
```

Parameter	Description	Default
`metric`	Evaluation metric	Required
`auto`	Preset config	None
`num_trials`	Optimization trials	Varies
`num_candidates`	Candidates per trial	10
`max_bootstrapped_demos`	Set 0 for instruction-only	4
`max_labeled_demos`	Labeled examples	16
`prompt_model`	LM for proposals	Default
`task_model`	LM for execution	Default

Auto presets:
- `"light"`: ~10 trials, quick optimization
- `"medium"`: ~40 trials, balanced
- `"heavy"`: ~100+ trials, thorough

Best for: 200+ training examples, comprehensive tuning.

SIMBA

Identifies hard examples and generates improvement rules.

```
from dspy.teleprompt import SIMBA

optimizer = SIMBA(metric=metric)
```

GEPA (Genetic-Pareto)

Newest optimizer: LLM reflection on full execution traces.

```
import dspy

# GEPA requires a feedback metric
def gepa_metric(example, pred, trace=None):
    is_correct = example.answer.lower() in pred.answer.lower()
    feedback = "Correct answer" if is_correct else f"Expected {example.answer}, got {p
red.answer}"
    return is_correct, feedback

optimizer = dspy.GEPA(
    metric=gepa_metric,
    reflection_lm=dspy.LM("openai/gpt-4o"),
    auto="medium"
)
```

Parameter	Description	Default
`metric`	Must return (score, feedback)	Required
`reflection_lm`	Strong LM for reflection	Default LM
`auto`	“light”, “medium”, “heavy”	None
`enable_tool_optimization`	Optimize tool descriptions	False

Best for: Complex agentic systems, when you have rich textual feedback.

Fine-tuning Optimizer

BootstrapFinetune

Distill a DSPy program into fine-tuned weights.

```
import dspy

optimizer = dspy.BootstrapFinetune(
    metric=lambda gold, pred, trace=None: gold.label == pred.label
)
finetuned = optimizer.compile(
    program,
    trainset=trainset,
    train_kwargs={'learning_rate': 5e-5, 'num_train_epochs': 3}
)
```

Parameter	Description
`metric`	Validation metric (optional)
`train_kwargs`	Training hyperparameters

Ensemble Optimizer

Combine multiple programs.

```
from dspy.teleprompt import Ensemble

ensembled = Ensemble(reduce_fn=dspy.majority).compile([prog1, prog2, prog3])
```

Retrieval & RAG

ColBERTv2 Setup

```
import dspy

colbert = dspy.ColBERTv2(url='http://20.102.90.50:2017/wiki17_abstracts')
dspy.configure(lm=dspy.LM("openai/gpt-4o-mini"), rm=colbert)
```

RAG Module Pattern

```
class GenerateAnswer(dspy.Signature):
    """Answer questions with short factoid answers."""
    context = dspy.InputField(desc="May contain relevant facts")
    question = dspy.InputField()
    answer = dspy.OutputField(desc="Often between 1 and 5 words")

class RAG(dspy.Module):
    def __init__(self, num_passages=3):
        super().__init__()
        self.retrieve = dspy.Retrieve(k=num_passages)
        self.generate = dspy.ChainOfThought(GenerateAnswer)

    def forward(self, question):
        context = self.retrieve(question).passages
        pred = self.generate(context=context, question=question)
        return dspy.Prediction(context=context, answer=pred.answer)
```

Evaluation

Evaluate Class

```
from dspy.evaluate import Evaluate

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

Built-in Metrics

```
# Exact match (normalized, case-insensitive)
dspy.evaluate.answer_exact_match

# F1 token overlap
dspy.evaluate.answer_passage_match
```

SemanticF1

LLM-based semantic evaluation:

```python
from dspy.evaluate import SemanticF1

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

Custom Metrics

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

Best Practices

1. **Start simple** - Use `dspy.Predict` before `ChainOfThought`
2. **Measure first** - Establish baseline metrics before optimizing
3. **Small data works** - BootstrapFewShot works with ~10 examples
4. **Match optimizer to data** - MIPROv2 for 200+, Bootstrap for fewer
5. **Use strong teachers** - GPT-4 as teacher, GPT-3.5 as student
6. **Save compiled programs** - `program.save('optimized.json')`