The Modern Scienti... · Follow publication

An Exploratory Tour of DSPy: A Framework for Programing Language Models, not Prompting

Is declarative approach appropriate to programming LLMs? A peek at DSPy programming model?

15 min read · Jun 3, 2024

 Jules S. Damji Follow

Introduction

Some contend that in the realm of software development, less than ideal innovations, with less than ideal developer adoption and community, fade into irrelevance. In today's frenzy and tizzy world of Generative AI, better "ways of doing stuff" emerge daily, each with their proponents advocating with infectious enthusiasm their merits over other innovations.

Courtesy of DALL-E 3

Is DSPy, pronounced as *dee-s-pie*, the new framework from Stanford NLP, for programming language models less than an ideal innovation? Is it a replacement, as some of its proponents aver, to prompt engineering techniques? Lastly, is the goal of framework attainable by replacing the artful and skillful yet tedious and fragile prompt construction in favor of systematic, modular, and composable programs?

In this article, I examine what's about DSPy that is promising and confusing, what's lacking and needs improving; and how it can construct modular pipelines to interact with LLMs. Through a few end-to-end prompting examples, I'll convert established prompting techniques to equivalent DSPy modular versions and asses its merit along the way. You can peruse these IPython notebooks on Google Colab and Python apps on GitHub.

Python Notebooks

Notebook Description	Open with Colab
Common NLP Tasks with DSPy Signature Modules	Open in Colab
Chain of Thought (CoT) Tasks with DSPy Modules	Open in Colab
Program of Thought (PoT) Tasks with DSPy Modules	Open in Colab
Naive RAG with DSPy Modules	Open in Colab
ReAct Tasks with DSPy Modules	Open in Colab
Zero Short Learning with DSPy Modules	Open in Colab

Figure 1a. Python DSPy notebooks showcasing how to use DSPy modules

Python Apps

Python file Description	View it on Github
Basic inline DSPy signatures for LLM tasks	Python App
Basic Class Signature modules for LLM Tasks	Python App
Common NLP Tasks with DSPy Modules	Python App
Chain Of Thought Tasks with DSPy Modules	Python App
Program Of Thought: Python code generation tasks with DSPy Modules	Python App
Naive RAG with DSPy Module	Python App
ReAct tasks with DSPy Module	Python App
Zero Short Learning with DSPy Module	Python App

Figure 1b. Python DSPy apps showcasing how to use DSPy modules

DSPy Programming Model

The ML community is quickly advancing in techniques for prompting language models (LMs) and integrating them into pipelines to tackle complex tasks. However, current LM pipelines often rely on hard-coded "prompt templates," which are lengthy, fragile, brittle and hand-crafted prompts developed through trial and error [1].

Researchers Omar Khatab and Arnav Singhvi et al., of Stanford NLP group, argue that this approach, though common, can be brittle, fragile, and unscalable, akin to hand-tuning classifier weights. Also, a specific or an elaborate string prompt may not generalize well to different pipelines, language models, data domains, or inputs [2].

Hence, they propose a more declarative, systematic and programmatic approach to interface with language models–something that PyTorch and Python developers are accustomed to in developing machine learning (ML) programs and ML related concepts.

The DSPy programming model comprises three high-level abstractions: signatures, modules, and teleprompters (aka optimizers). Signatures abstract and dictate the input/output behavior of a module; modules replace existing hand-prompting techniques and can be composed as arbitrary pipelines; and teleprompters, through compilation, optimize all modules in the pipeline to maximize a metric [3].

Let's start with Signatures first.

Signatures abstract away prompting

A DSPy signature is a natural-language typed function declaration: a concise specification describing *what* a text transformation should achieve (e.g., "consume questions and return answers"), rather than detailing *how* a specific LM should be prompted to perform that task.

As such they have two advantages over prompts. First, they can be compiled into self-improving, pipeline-adaptive prompts or fine-tuned by bootstrapping useful examples for each signature. Second, they manage structured formatting and

parsing logic, reducing or ideally eliminating brittle string manipulation in user programs [4].

> *"A signature is a declarative specification of input/output behavior of a DSPy module. Signatures allow you to tell the LM* what *it needs to do, rather than specify* how *we should ask the LM to do it," states the docs. [5]*

For instance, you can declaratively define a DSPy Signature object using a shorthand string notation as an argument. Effectively, this Signature now is declaring a task as a concise prompt: given a question, return an answer. In short, this shorthand notation is your declarative replacement of a prompt for a simple task. Here are a few examples of shorthand notations:

```
import dspy
sig_1 = dspy.Signature("question -> answer")
sig_2 = dspy.Signature("document -> summary")
sig_3 = dspy.Signature("question, context -> answer")
```

Aside from in-line shorthand notation to declare tasks, you can define a class-based Signature, giving you more control over the input/output fields format, style, and descriptive task description. The task description is a Python doc string in the class definition. And the format, style or behavior of the output can be a descriptive and declarative argument to the *dspy.OutputField,* making it easier to tweak it, rather than part of a larger prompt.

```
class BasicQA(dspy.Signature):
  """Answer questions with short factoid answers"""

  question = dspy.InputField()
  answer = dspy.OutputField(desc="often between 1 and 5 words",
                            prefix="Question's Answer:")
```

Internally, DSPy, converts both above declarative formats into a prompt for the underlying LLM as shown in Figure 2. Optionally, using DSPy teleprompters (optimizers), these prompts can be compiled to iteratively generate an optimized LLM prompt (see the section below on Optimizers), akin to how you would optimize

an ML model with learning optimizers, such as SGD, in an ML framework like PyTorch.

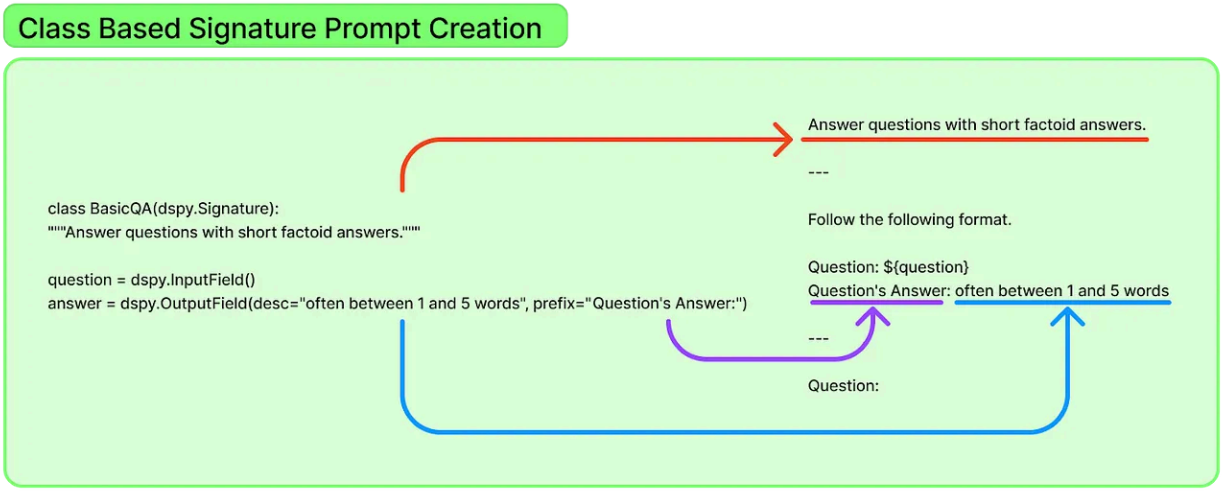


Figure 2. Class-based declarative signature translation into an LLM prompt

Using the above class-based Signature is simple and intuitive.

```
generate_response  = dspy.Predict(BasicQA)
pred = generate_resonse(question="When was the last Solar Eclipse in the United
print(f"Answer: {pred.answer}")
```

👍👍👍: Using a task description as a Python class docstring or shorthand notation to generate an LLM prompt, without hand-crafting a prompt, aligns with a key DSPy framework assertion. It feels like Python programming, not manual hand-crafting delicate prompts. Converting some prompting technique examples above in Figures 1(a) and 1(b) instilled and inspired that sense in me.

While the *dspy.Signature* class is the core building block, DSPy also includes built-in modules that effectively translate well to prompting techniques like chain of thought, ReAct, RAG, program of thought, and complex reasoning.

At the heart of all these modules is the *dspy.Predict* module, which all modules, including Signature, invoke through their *forward()* function call. Internally, Predict stores the Signature and uses it to construct a prompt.

Let's explore these modules next.

Modules Build Complex Pipelines

According to the DSPy documentation, a DSPy module is a fundamental building block for DSPy pipelines or programs that use language models. Each module abstracts a prompting technique, such as chain of thought or ReAct, and is generalized to handle any DSPy Signature.

Modules can have learnable parameters, including prompt components and LM weights. As callable classes, they can be invoked with inputs and return outputs. Because they are building blocks, multiple modules can be combined into larger composable programs as pipelines. Inspired by NN modules in PyTorch, DSPy modules are designed for LLM programs [6].

As callable classes, modules can be invoked with inputs and return outputs. Because they are building blocks, multiple modules can be combined into larger programs as pipelines. Inspired by NN modules in PyTorch, DSPy modules are designed for LLM programs.

Think of modules as a smart shortcut to simplify complex prompting techniques. They're like prefabricated blocks you can snap together to build your program. Creating your own modules is encouraged and is the core way to construct complex data pipeline programs in DSPy. These modules can function as standalone or combined as pipelines for more complex tasks, and they can be used in various applications [7].

For instance, I can define a standalone *ChainOfThought* module using shorthand Signature notation.

```
class ChainOfThought(dspy.Module):
 def __init__( self, signature):
  super().__init__()
  self.predict = dspy.Signature(signature)

 # overwrite the forward function
 def forward(self, **kwargs):
  return self.predict(**kwargs)

# create an instance of class with shorthand Signature notation
# as argument
cot_generate = ChainOfThought("context, question → answer")

# call the instance with input parameters specified in the
```

```
# signature
response = cot_generate("context=....",
   "question=How to compute area of a triangle with height 5 feet and width 3 f
print(f"Area of triangle: {response.answer}")
```

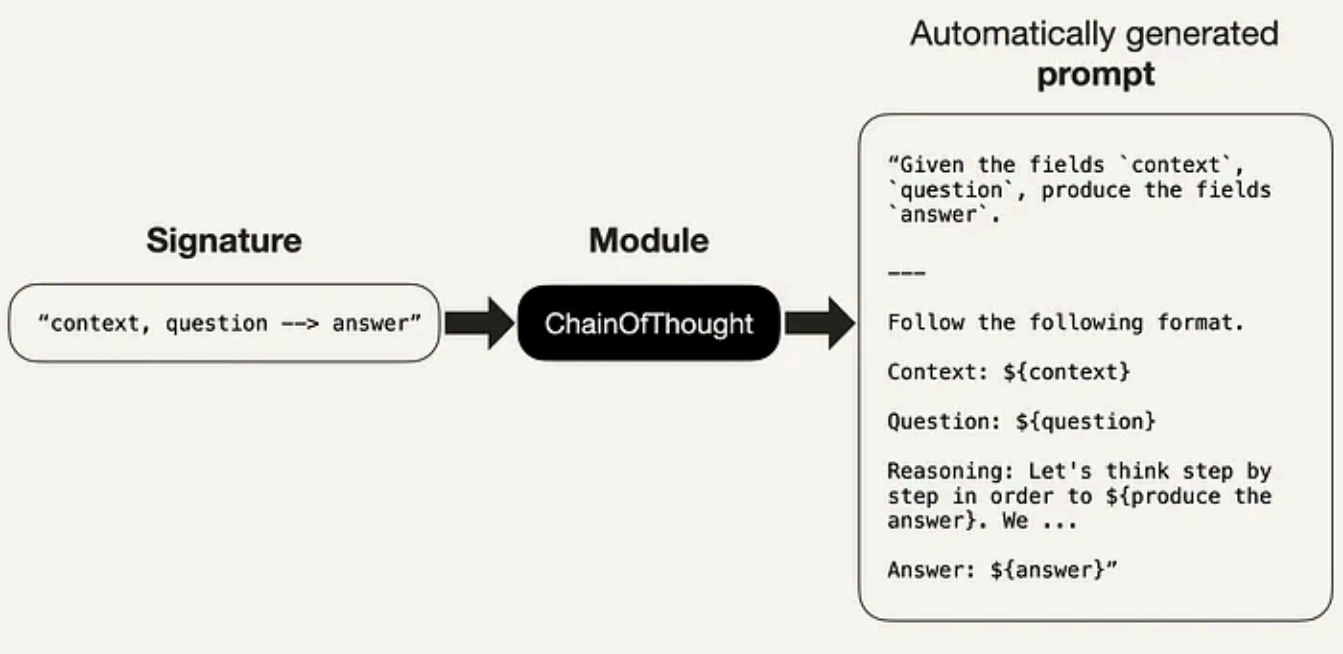


Figure 3. Module with a shorthand signature generated as an LLM prompt [8]

Let's take this notion one step further and build a composable pipeline using the building block we defined above, along with a built-in *dspy.Retriever* module to illustrate how you can create a composable pipeline as a RAG DSPy program.

```
class RAGSignature(dspy.Signature):
      """
      Given a context and question, answer the question.
      """
      context = dspy.InputField()
      question = dspy.InputField()
      answer = dspy.OutputField()

class RAG(dspy.Module) :
      def __init__ ( self , num_passages=3) :
          super().__init__()
          # Retrieve will use the user's default retrieval settings
          # unless overridden .
          self.retrieve = dspy.Retrieve(k=num_passages)

          # ChainOfThought with signature that generates
          # answers given retrieval context & question .
          self.generate_answer = dspy.ChainOfThought(RAGSignature)
```

```
        def forward (self, question) :
            context = self.retrieve (question).passages
            return self.generate_answer(context=context, question=question)
```

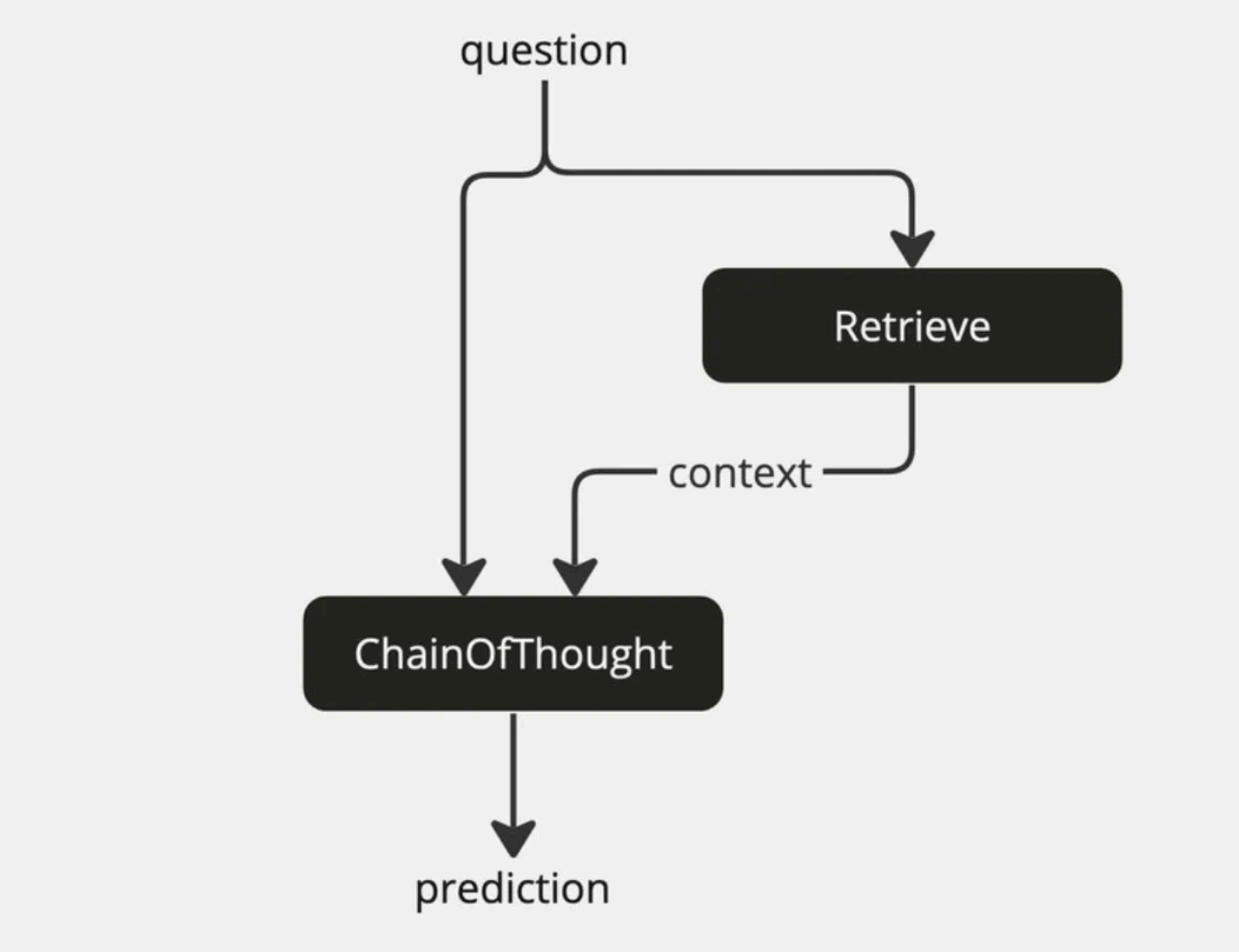


Figure 4: Composable DSPy RAG program module pipeline with Retrieve and ChainOfThought

To see full implementation of a Naive RAG with DSPy modules peruse the links in Figures 1(a) and 1(b).

👍👍👍: DSPy modules are Python declarative code, encapsulating your task logic (of *what-to-do* not *how-to-do*), behavior, input/output format, style and any custom code. No need to write a novel as an elaborate prompt, no need to belabor trial and error with the prompts. Rather, construct your flow as a pipeline of composable blocks. I rather enjoyed writing Python code than English, albeit I do love writing. And let DSPy do the work in generating the prompt and interfacing with the language model.

Mapping well to the common prompting techniques, the built-in modules, listed in the table below, are extensible and customizable. At the time of this writing, the following DSPy modules were available:

Module	Description
Predict	As the basic predictor, it's used by all modules. It takes in a Signature, handles key forms of learning, such as storing instructions, demonstrations, and sends updates to the LM, along with the generated prompt.
ChainOfThought	Trains the LM to think step-by-step before committing to the signature's response.
ProgramOfThought	Instructs the LM to output code, in languages supported by the targeted LLM. The generated code will be executed and the results, successful run or failure, will dictate the response
ReAct	Allows to develop agent-like capabilities with the tools supported by the targeted LM.
MultiChainComparison	Combined with ChainOfThought, this module compares multiple outputs from ChainOfThought to produce a final prediction.

Table 1: List of supported DSPy modules mapping the common prompting techniques [9]

What's even better is that, like ML models, you can optimize these modules for efficient prompt generation and response evaluation with DSPy optimizers through compilation. Let's examine how to achieve that next.

Optimizing and Compiling Modules

Just like PyTorch optimizers, like SGD, for minimizing loss and maximizing accuracy in ML, the DSPy optimizers API allows you to provide training examples and an evaluation metric to measure accuracy. If you're familiar with PyTorch, this concept will resonate.

Optimizers take in a training set (to bootstrap a few selective examples to learn how to generate the prompt) and a metric (to measure proximity to or match correct response); they generate an instance of an optimized program that can be used to compile a DSPy program module. Currently, DSPy supports a number of built-in optimizers, each with a degree of rigor to maximize your metric.

Best to illustrate with some example code. Consider a small training set of examples that you want to use to train a DSPy module to conduct sentiment analysis. In prompt engineering, this is akin to a few-shot learning technique as part of the larger in-context prompt.

The response from the module, after interfacing with the target language model, is either positive, negative, or neutral. Your metric can, then, check if the answer returned is one of those class sentiment categories–or something bogus.

Let's create a short training set, metric, and module. Note that metrics can be as simple as returning a numeric score (like 0 or 1), an exact match (EM) or F1, as well as an entire DSPy program that balances and measures multiple concerns in the prediction.

```
# Evaluate a metric for the right response category
def evaluate_sentiment(example, pred, trace=None)->bool:
  return pred in ["positive", "negative", "neutral"]

def get_examples() -> List[dspy.Example]:
  trainset = [dspy.Example(sentence="""This movie is a true cinematic gem,
                                    blending an engaging plot with superb perf
                           sentiment="positive").with_inputs("sentence"),
              dspy.Example(sentence="""Regrettably, the film failed to live
                                    up to expectations, with a convoluted
                                    storyline, lackluster acting, and
                                    uninspiring cinematography.
                                    disappointment overall."""
                           sentiment="negative").with_inputs("sentence")
              dspy.Example(sentence="""The movie had its moments, offering
                                    a decent storyline and average
                                    performances. While not groundbreaking,
                                    it provided an enjoyable viewing
                                    experience.""",
                           sentiment="neutral").with_inputs("sentence")

              ...
            ]
      return trainset

# define our DSPy module that you want to optimize and compile
class ClassifyEmotion(dspy.Signature):
  """ classify emotion based on the input sentence and provide the sentiment as
   sentence = dspy.InputField()
   sentiment = dspy.OutputField(desc="generate sentiment as
   positive, negative or neutral")
```

```
from dspy.teleprompt import BootstrapFewShot

# Create an optimizer
optimizer = BootstrapFewShot(metric=evaluate_sentiment,
trainset=get_examples())
compiled_classifier = optimizer.compile(ClassifyEmotion(),
trainset=get_examples()

# Use our compiled classifier that has learned through bootstrapping
# a few examples how to generate the response
response = compiled_classifier(sentence="I can't believe how beautiful the suns
print(response.sentiment).
```

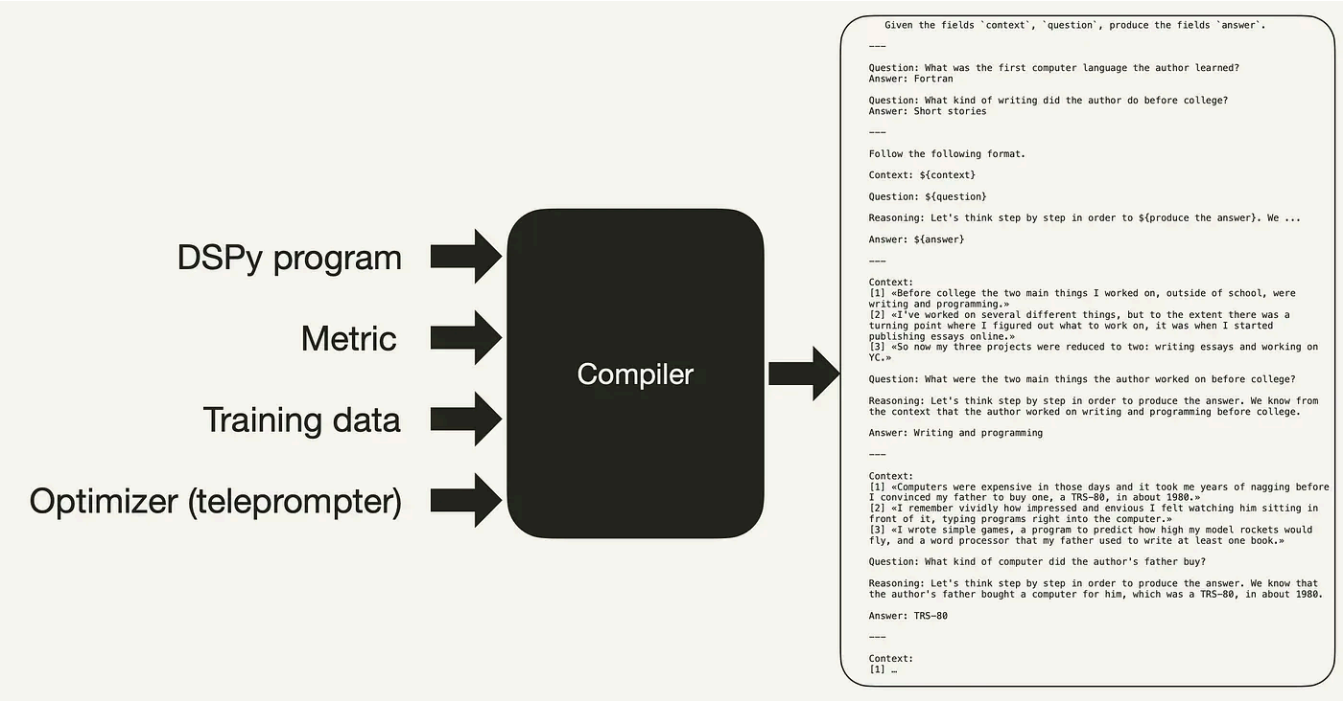


Figure 5: Set of artifacts for the compiler to generate optimized prompts [12]

Internally, all the above achieves the following:

1. Bootstrap our trainset to learn using few-shot in-context prompting
2. Use the metric to evaluate if the output predicts one of the three sentiment categories
3. Compile the DSPy module
4. Generate prompts
5. Use the compiled classifier to classify our sentences with the best prompt

Omar Khattab and Arnav Singhvi et al. describe the above optimizing and compiling process undergoing three stages [13]:

1. **Candidate generation**: select the candidate predictor module, if more than one.
2. **Parameter optimization:** select candidates' instructions or demonstrations in the prompts and then optimize with different LM weights for the best response.
3. **Higher-Order program optimization:** think of these as a language compiler code optimization where code is rearranged for better execution. In DSPy, complicated pipelines are simplified as ensembles and rearranged to alter the control flow.

To see how the DSPy framework optimized and tuned your prompts, simply print the history of all generated prompts with this command, where n > 0.

```
your_model.inpspect_history(n=3)
```

This will print out three different optimized prompts generated for the LLM. To see a full example of a few-shot example with optimizer and compilation, peruse the notebook or Python app on ReAct tasks in Figures 1(a) and 1(b).

Another discussion by Frederick Ros examines tuning and optimizing DSPy modules [14]. Lastly, Omar Khattab et al, offer a couple of case studies with empirical data to suggest that optimized and compiled modules offer tangible efficiency, performance, and accuracy over unoptimized ones with respect to quantifiable measurements for complex reasoning tasks [15].

👎: The concept of optimizers and compilers in the DSPy framework can be difficult to understand, seem non-intuitive, and mysterious as a black box. Although they deliver their objectives, they fall short in clarity and simplicity: why not conflate optimize and compile as a single API call instead of two separate phases. Also, documentation is scanty with no explicit examples or illustrations that shed light into darkness and bring it to sharper focus.

All assets I examined so far–documentation and published blogs–come short to lucidly explain this powerful notion. For this concept is central to self-improvement and optimization aspect of the framework. With absence of clear cut examples and practical use cases, this concept eluded me to elicit the WOW 🤯 appeal in me.

DSPy End-to-end Example Programs

Programming frameworks, write Omar Khattab and Arnav Singhvi et al., can be evaluated along many dimensions, including computational efficiency, developer efficiency, intuitiveness of the code and concepts, etc. The authors evaluate the DSPy programming framework along three hypotheses [15]:

- **H1**: With DSPy, we can replace hand-crafted prompt strings with concise and well-defined modules, without reducing quality or expressive power.
- **H2:** Parameterizing the modules and treating prompting as an optimization problem makes DSPy better at adapting to different LMs, and it may outperform expert-written prompts.
- **H3:** The resulting modularity makes it possible to more thoroughly explore complex pipelines that have useful performance characteristics or that fit nuanced metrics.

I used the DSPy framework to convert all my examples from a previously published Prompt Engineering blog with explicit and elaborate prompts techniques. Using the local OLama language model and DSPy's built-in tools like Retrievers, I was able to modularize and construct complex-pipelines for complex reasoning tasks.

NLP Tasks

I used DSPy declarative signatures to express *how-to* code examples for common natural language understanding capabilities of a generalized LLM, such as ChatGPT, OLlama, Mistral, and Llama 3 series:

- Text generation or completion

- Text summarization
- Text extraction
- Text classification or sentiment analysis
- Text categorization
- Text transformation and translation
- Simple and complex reasoning

To that end, the DSPy modules were up to the task. Code is modular, declarative, and no storytelling with prompts; no need for the CO-STAR prompt framework to hand-craft an elaborate prompt. See notebooks and Python apps in Figures 1(a) and 1(b) in GenAI Cookbook GitHub Repository.

Program of Thought Task

Program of thought prompting, like chain of thought, for LLMs involves providing a sequence of reasoning steps in the prompt to guide the model toward a solution. This technique helps the model to process complex problems by breaking them down into intermediate steps, much like a human would. By mimicking human-like reasoning, chain of thought prompting improves the model's ability to handle tasks that require logic, deduction, and programming.

Using DSPy Program of Thought (PoT) Module *dspy.ProgramOfThought*, most of these examples generate Python code to solve the problem. Hardly, any need to specify elaborate prompts, just a concise task description.

See notebooks and Python apps in Figures 1(a) and 1(b) in GenAI Cookbook GitHub Repository.

Naive RAG

Amazingly easy and modular, DSPy Modules can be chained or stacked to create a pipeline. In our case, building a Naive RAG comprises using *dspy.Signature* and *dspy.ChainOfThought*, and module class RAG (see implementation in dspy_utils.)

Out of the box, DSPy supports a set of Retrievers clients. For this example, I used the supported *dspy.ColBERTv2* tool.

See notebooks and Python apps in Figures 1(a) and 1(b) in GenAI Cookbook GitHub Repository.

ReAct Tasks

First introduced in a paper by Yao et al., 2022, ReAct is a reasoning and acting paradigm that guides LLM to respond in a structured manner to complex queries. Reasoning and actions are interleaved and progressive, in the manner of chain of thought, so that LLM progresses from one result to another, using the previous answer.

Results suggest that ReAct outperforms other leading methods in language and decision-making tasks, enhancing human understanding and trusting in large language models (LLMs). It is best when combined with chain-of-thought (CoT) steps as individual tasks with results being used for the next step, utilizing both internal knowledge and external information during reasoning.

This task was a DSPy conversion of LLM ReAct prompting notebook.

See notebooks and Python apps in Figures 1(a) and 1(b) in GenAI Cookbook GitHub Repository.

👍👍👍**:** In the all above prompt engineering tasks, the DSPy's creators' hypotheses **H1** and **H3** seem to hold up well and meet my expectations. However, **H2** is a bit unclear and unintuitive, and I was unable to get my head around it, as I lament above in the optimizing and compiling section. To some extent the mapping of DSPy modules to common prompting tasks worked, underpinning **H1** and **H2**.

Conclusion

In this article, we covered DSPy framework and programming model, an innovative, declarative, systematic, and modular way to program and interface with language models instead of using explicit and elaborate prompts. Through elaborate examples of prompt engineering techniques explored in my previous blog, I converted prompting techniques into their equivalent DSPy programs.

Along the way, I praised many aspects of DSPy that appealed to me and may also appeal to Python developers accustomed to declarative methods of programming. I pointed out some of its shortcomings with respect to the concepts, documented examples, and lack of use cases. Two out of three hypotheses explored in the original paper seem to hold up well during my programmatic endeavor: converting

elaborate and explicit prompt engineering techniques and constructing modular and declarative DSPy programs.

Is the new framework from Stanford NLP, for programming language models less than an ideal innovation? Will it fade into irrelevance?

I don't think so. While it has not exploded in popularity like other LLM frameworks such as LangChain and LLamaIndex, it has an expanding community, growing presence on its GitHub (with 924 forks, 150 contributors, and over 12K stars), and gathering lively discussions on Reddit and discord forum, so not likely to fade into irrelevance either.

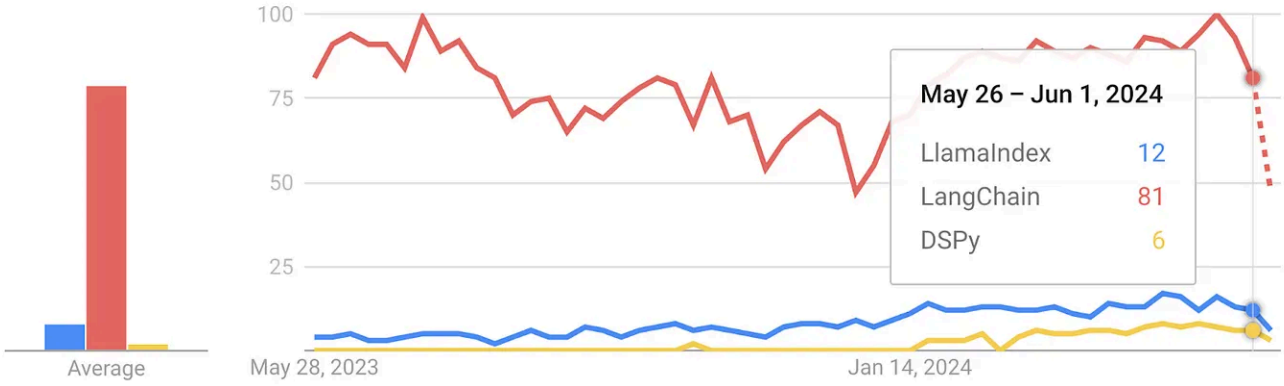


Figure 6 Comparison searches of DSPy with LlamaIndex and LangChain

Is it a replacement, as some of its proponents aver, to prompt engineering techniques? It's a likely optional preference where applicable with tangible use cases that demonstrate its efficacy and usage. If more GenAI, Data, AI pioneer companies like Databricks showcase its usage with their ecosystem, or a well-funded startup company behind DSPy emerges, we are likely to see wider usage.

A total replacement of any technological innovation is a hyperbolic claim; total replacement is only attained over time, not overnight. Not likely that DSPy will immediately dislodge or displace skillful and hand-crafted prompts and prompt template engineering yet.

What's Next?

Meanwhile, if you missed my sequence of blog series on GenAI Cookbook on LLMs, take a read:

- Best Prompt Techniques for Best LLM Responses
- LLM Beyond its Core Capabilities as AI Assistants or Agents
- Crafting Intelligent User Experiences: A Deep Dive into OpenAI Assistants API
- An Intuitive 101 Guide to Vector Embeddings
- An Exploratory Tour of Retrieval Augmented Generation (RAG) Paradigm

To stay abreast with updates or upcoming blogs, follow me on X @2twitme or LinkedIn. Stay tuned for next blog on Assertions, Datasets, Examples, Evaluate, and more Compilers and Optimizers, as I attempt another go at it and get my head around it.

References and Resources

[1, 2, 3, 4, 11, 13, 14, 15] https://arxiv.org/pdf/2310.03714

[5] https://dspy-docs.vercel.app/docs/building-blocks/signatures

[6] https://dspy-docs.vercel.app/docs/building-blocks/modules

[7, 8] https://www.theaidream.com/post/dspy-a-revolutionary-framework-for-programming-llms

[9] https://dspy-docs.vercel.app/docs/building-blocks/modules#what-other-dspy-modules-are-there-how-can-i-use-them

[10] https://medium.com/@frederick.ros/dspy-essentials-mastering-model-tuning-with-optimizers-32b1db951915

[12] https://towardsdatascience.com/intro-to-dspy-goodbye-prompting-hello-programming-4ca1c6ce3eb9

[16] https://www.databricks.com/blog/optimizing-databricks-llm-pipelines-dspy

[17] https://dspy-docs.vercel.app/docs/tutorials/examples

Llm | Dspy | Prompt Engineering | Genai | Python Programming

Follow

Published in The Modern Scientist

1.2K followers · Last published Sep 22, 2025

The Modern Scientist aspires to connect builders & the curious to forward-thinking ideas. Either you are novice or expert, TMS will share contents that fulfils your ambition and interest. Write with us: shorturl.at/hjO39

Follow

Written by Jules S. Damji

606 followers · 207 following

Developer at heart; Advocate by nature | Communicator by choice | Avid Arsenal Fan| Love Reading, Writing, APIs, Coding | All Tweets are Mine

Responses (6)

Write a response

What are your thoughts?

Jules S. Damji Author

Jun 21, 2024

Ivan, as far as I know there is no direct way API support but I suppose in your Signature you can examine the input. For exampls,

class WordMathProblem(dspy.Signature):

"""Given a word math problem, solve the problem and provide step by step... more

3 Reply

Maryk
May 26

Thank you. that's very useful

1 Reply

Krishna Mishra
Oct 16, 2024

Thank your for all your efforts on this one.

2 Reply

See all responses

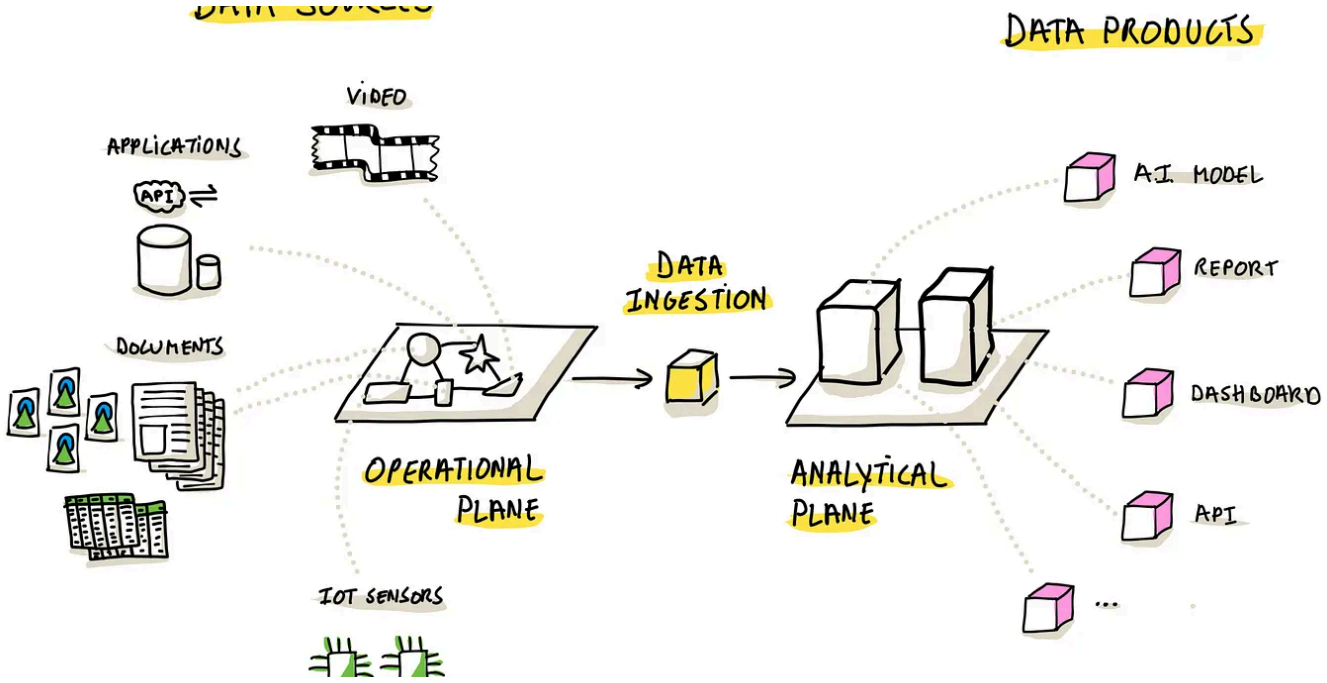
More from Jules S. Damji and The Modern Scientist

In The Modern Scientist by Jules S. Damji

Best Prompt Techniques for Best LLM Responses

Better prompts is all you need for better responses

Feb 12, 2024 959 8

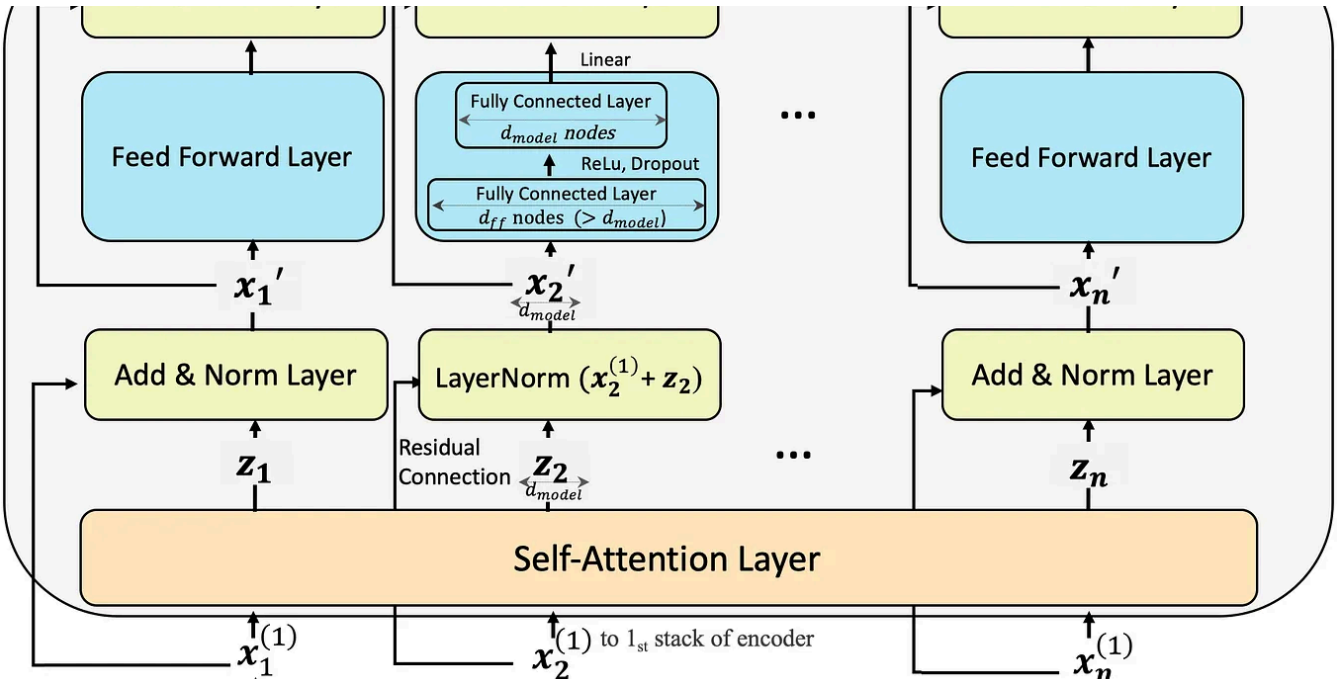


In The Modern Scientist by janmeskens

Data Ingestion — Part 1: Architectural Patterns

Over the course of two articles, I will thoroughly explore data ingestion, a fundamental process that bridges the operational and...

Nov 27, 2023 2.8K 25

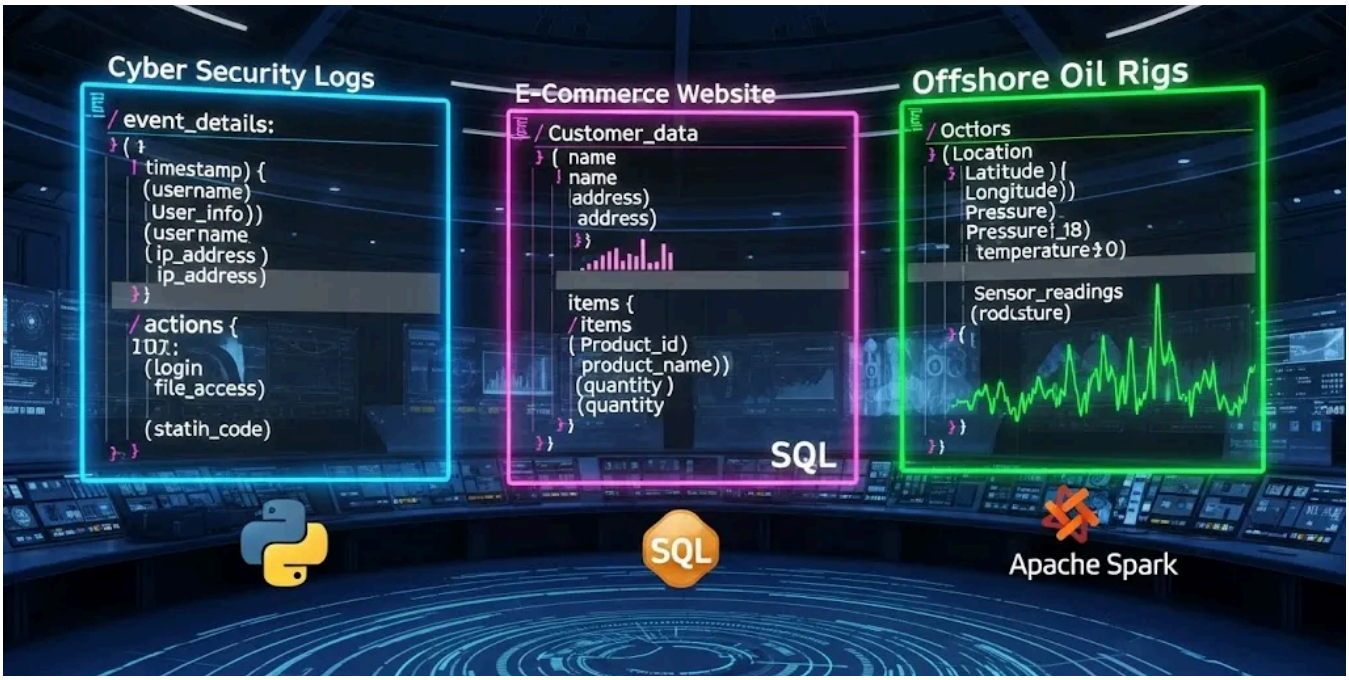


In The Modern Scientist by Yule Wang, PhD

Step-by-Step Illustrated Explanations of Transformer

with detailed explained Attention Mechanism

Feb 27, 2023 416 4



Jules S. Damji

Apache Spark™ 4.0's Variant Data Type Eliminates Complex JSON String Processing

Three synthetic real-world use cases demonstrate gains and usage through binary encoding and direct field access.

Aug 26 11 2

See all from Jules S. Damji

See all from The Modern Scientist

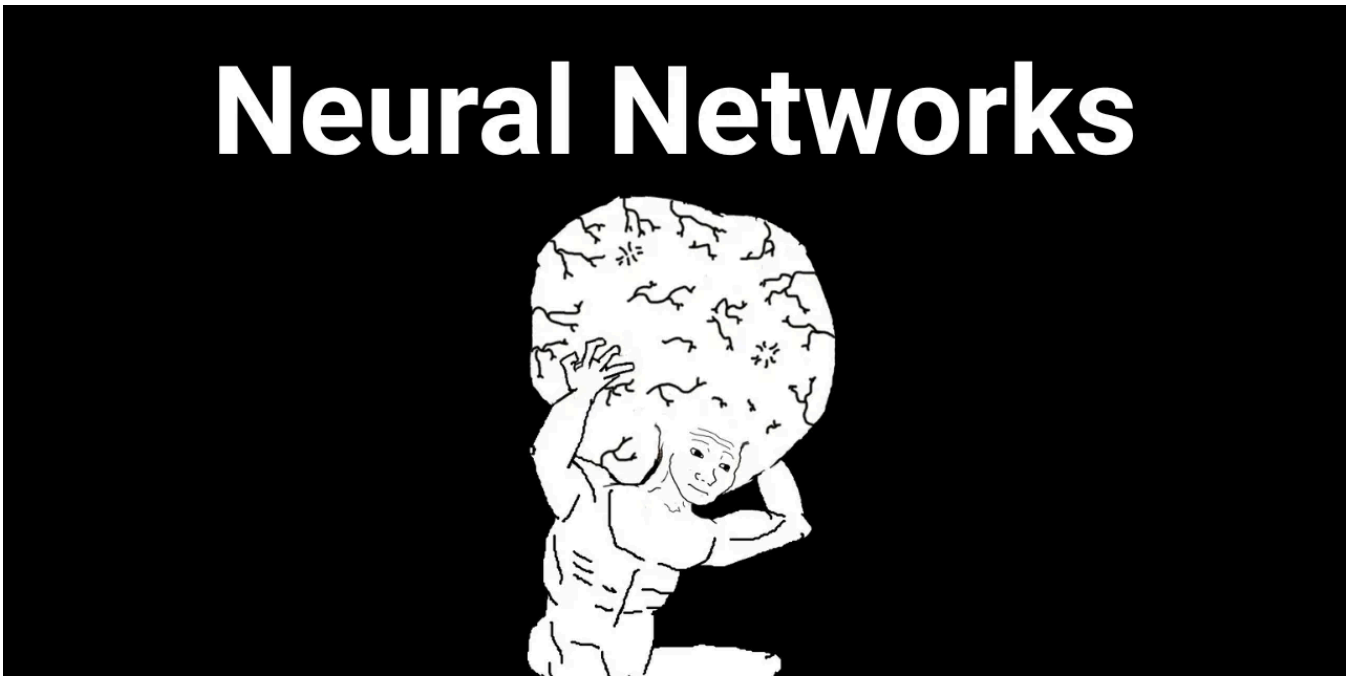
Recommended from Medium

In Artificial Intelligence in Plain English by Keng Lim

Learn DSPy: From Basics to Production-Ready AI

Tired of toy examples in DSPy tutorials?

Jun 2 1



Yashwanth Sai

How I am learning Deep Learning again

Here's a plan if you want to get into this field

Aug 22 85 1

In Stackademic by Dileep Sreepathi

Machine Learning in .Net (ML.Net)

As a .NET developer, I am eager to explore the integration of artificial intelligence (AI) into my applications. Given the growing hype...

Apr 22 6 1

DSPy

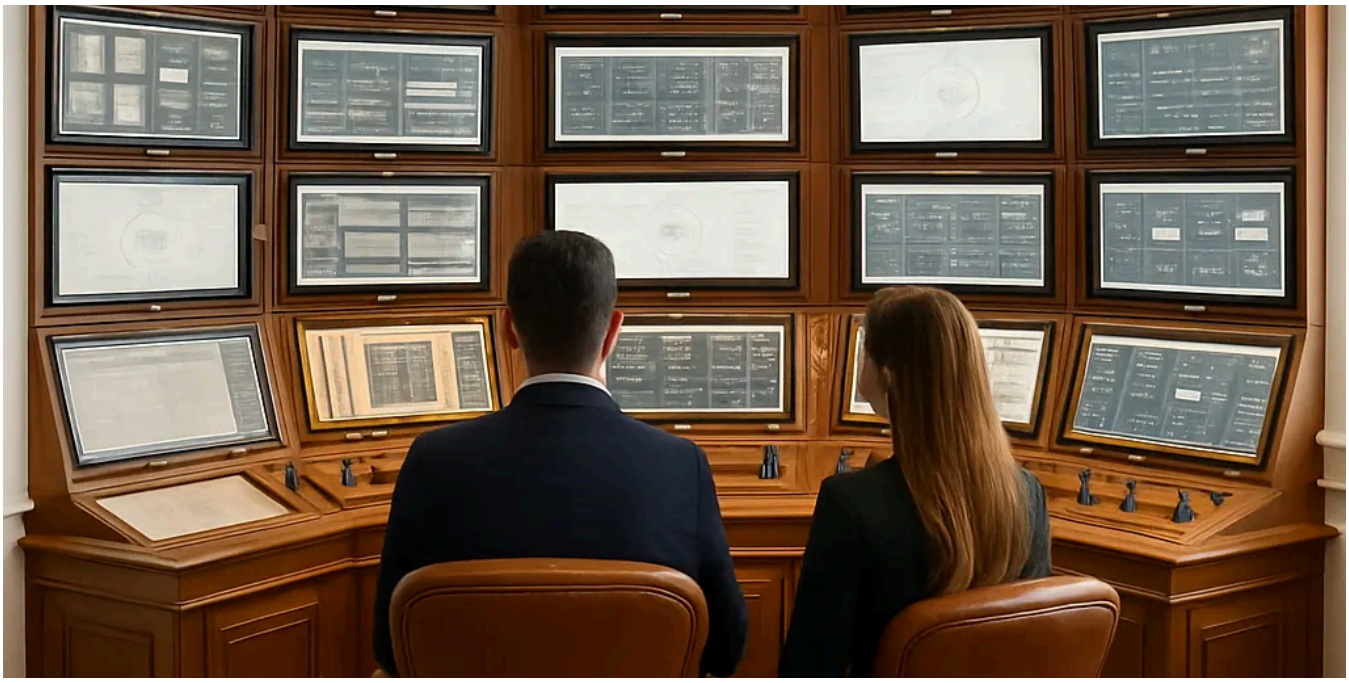
Get Started

The framework for programming—rather than prompting—language models.

AI Engineering

DSPy and GEPA: Underrated Power Tools for AI Engineering

Sep 4 11



Alex Fuentes

DSPy: The Game-Changer for Prompt Engineering 🚀

The Problem We All Face

Sep 5 2

In Python in Plain English by KitFu Coda

Leveling Up LLM Game Development with DSPy

After completing the series on asynchronous programming, I needed some time off to deal with the stress in life. During the break, I...

Apr 27 18

See more recommendations