Using DSPy in 8 Steps

Using DSPy well for solving a new task is just doing good machine learning with LMs.

What this means is that it's an iterative process. You make some initial choices, which will be sub-optimal, and then you refine them incrementally.

As we discuss below, you will define your task and the metrics you want to maximize, and prepare a few example inputs — typically without labels (or only with labels for the final outputs, if your metric requires them). Then, you build your pipeline by selecting built-in layers (`modules`) to use, giving each layer a `signature` (input/output spec), and then calling your modules freely in your Python code. Lastly, you use a DSPy `optimizer` to compile your code into high-quality instructions, automatic few-shot examples, or updated LM weights for your LM.

1) Define your task.

You cannot use DSPy well if you haven't defined the problem you're trying to solve.

Expected Input/Output Behavior: Are you trying to build a chatbot over your data? A code assistant? A system for extracting information from papers? Or perhaps a translation system? Or a system for highlighting snippets from search results? Or a system to summarize information on a topic, with citations?

It's often useful to come up with just 3-4 examples of the inputs and outputs of your program (e.g., questions and their answers, or topics and their summaries).

If you need help thinking about your task, we recently created a Discord server for the community.

Quality and Cost Specs: You probably don't have infinite budget. Your final system can't be too expensive to run, and it should probably respond to users quickly enough.

Take this as an opportunity to guess what kind of language model you'd like to use. Maybe GPT-3.5? Or a small open model, like Mistral-7B or Llama2-13B-chat? Or Mixtral? Or maybe you really need GPT-4-turbo? Or perhaps your resources are very constrained, and you want your final LM to be T5-base.

2) Define your pipeline.

What should your DSPy program do? Can it just be a simple chain-of-thought step? Or do you need the LM to use retrieval? Or maybe other tools, like a calculator or a calendar API?

Is there a typical workflow for solving your problem in multiple well-defined steps? Or do you want a fully open-ended LM (or open-ended tool use with agents) for your task?

Think about this space but always start simple. Almost every task should probably start with just a single dspy.ChainofThought module, and then add complexity incrementally as you go.

Then write your (initial) DSPy program. Again: start simple, and let the next few steps guide any complexity you will add.

3) Explore a few examples.

By this point, you probably have a few examples of the task you're trying to solve.

Run them through your pipeline. Consider using a large and powerful LM at this point, or a couple of different LMs, just to understand what's possible. (DSPy will make swapping these LMs pretty easy - LM Guide.)

At this point, you're still using your pipeline zero-shot, so it will be far from perfect. DSPy will help you optimize the instructions, few-shot examples, and even weights of your LM calls below, but understanding where things go wrong in zero-shot usage will go a long way.

Record the interesting (both easy and hard) examples you try: even if you don't have labels, simply tracking the inputs you tried will be useful for DSPy optimizers below.

4) Define your data.

Now it's time to more formally declare your training and validation data for DSPy evaluation and optimization - Data Guide.

You can use DSPy optimizers usefully with as few as 10 examples, but having 50-100 examples (or even better, 300-500 examples) goes a long way.

How can you get examples like these? If your task is extremely unusual, please invest in preparing ~10 examples by hand. Often times, depending on your metric below, you just need inputs and not labels, so it's not that hard.

However, chances are that your task is not actually that unique. You can almost always find somewhat adjacent datasets on, say, HuggingFace datasets or other forms of data that you can leverage here.

If there's data whose licenses are permissive enough, we suggest you use them. Otherwise, you can also start using/deploying/demoing your system and collect some initial data that way.

5) Define your metric.

What makes outputs from your system good or bad? Invest in defining metrics and improving them over time incrementally. It's really hard to consistently improve what you aren't able to define.

A metric is just a function that will take examples from your data and take the output of your system, and return a score that quantifies how good the output is - Metric Guide.

For simple tasks, this could be just "accuracy" or "exact match" or "F1 score". This may be the case for simple classification or short-form QA tasks.

However, for most applications, your system will output long-form outputs. There, your metric should probably be a smaller DSPy program that checks multiple properties of the output (quite possibly using AI feedback from LMs).

Getting this right on the first try is unlikely, but you should start with something simple and iterate. (If your metric is itself a DSPy program, notice that one of the most powerful ways to iterate is to compile (optimize) your metric itself. That's usually easy because the output of the metric is usually a simple value (e.g., a score out of 5) so the metric's metric is easy to define and optimize by collecting a few examples.)

6) Collect preliminary "zero-shot" evaluations.

Now that you have some data and a metric, run evaluation on your pipeline before any optimizer runs.

Look at the outputs and the metric scores. This will probably allow you to spot any major issues, and it will defi eline for your next step.

7) Compile with a DSPy optimizer.

Given some data and a metric, we can now optimize the program you built - Optimizer Guide.

DSPy includes many optimizers that do different things. Remember: DSPy optimizers will create examples of each step, craft instructions, and/or update LM weights. In general, you don't need to have labels for your pipeline steps, but your data examples need to have input values and whatever labels your metric requires (e.g., no labels if your metric is reference-free, but final output labels otherwise in most cases).

Here's the general guidance on getting started:

- If you have very little data, e.g. 10 examples of your task, use `BootstrapFewShot`
- If you have slightly more data, e.g. 50 examples of your task, use `BootstrapFewShotWithRandomSearch`.
- If you have more data than that, e.g. 300 examples or more, use `MIPRO`.
- If you have been able to use one of these with a large LM (e.g., 7B parameters or above) and need a very efficient program, compile that down to a small LM with `BootstrapFinetune`.

8) Iterate.

At this point, you are either very happy with everything (we've seen quite a few people get it right on first try w[illegible]) or, more likely, you've made a lot of progress but you don't like something about the final program or the metric.

At this point, go back to step 1 and revisit the major questions. Did you define your task well? Do you need to collect (or find online) more data for your problem? Do you want to update your metric? And do you want to use a more sophisticated

optimizer? Do you need to consider advanced features like DSPy Assertions? Or, perhaps most importantly, do you want to add some more complexity or steps in your DSPy program itself? Do you want to use multiple optimizers in a sequence?

Iterative development is key. DSPy gives you the pieces to do that incrementally: iterating on your data, your program structure, your assertions, your metric, and your optimization steps.

Optimizing complex LM programs is an entirely new paradigm that only exists in DSPy at the time of writing, so naturally the norms around what to do are still emerging. If you need help, we recently created a Discord server for the community.