Notes on the DSPy Model

compiled by D.Gueorguiev, 6/24/25

DSPy framework treats LMs as abstract devices for text generation, and optimizes their usage in arbitrary computation graphs. DSPy programs are written in Python – each program takes a task input (e.g. a question to answer or a paper to summarize) and returns the output (e.g. an answer or a summary) after series of steps. DSPy presents three abstractions toward automatic optimization – *signatures*, *modules,* and *teleprompters*.

*Signatures* abstract the input/output behavior of a module. *Modules* replace existing hand-prompting techniques and can be composed in arbitrary pipelines. *Teleprompters* optimize all modules present in a pipeline to maximize a metric.

Details on *Signatures*

Instead of free form string prompts, DSPy programs use natural language *signatures* to assign work to the LM.

A DSPy signature is a *natural-language typed* declaration of a function: a short declarative spec that tells DSPy **what** a text transformation needs to do (e.g. *“consume questions and return answers”*), rather than **how** a specific LM should be prompted to implement that behavior. More formally, a DSPy signature is a tuple of *input fields* and *output fields* (and an optional *instruction*). A field consists of *field name* and optional *field metadata*.

In typical usage, the roles of the fields are inferred by DSPy as a function of field names. For instance, the DSPy compiler will use in-context learning to interpret question differently from answer and will iteratively refine its usage of these fields.

Signatures offer two benefits over prompts – they can be compiled into self-improving and pipeline-adaptive prompts or finetunes. This is primarily done by bootstrapping useful demonstrating examples for each signature. Additionally, signatures absorb the structured formatting and parsing logic to reduce brittle string manipulation in user programs.

In practice, DSPy signatures are expressed with a shorthand notation like question -> answer, as shown in the example below:

1 qa = dspy.Predict(“question -> answer”)

2 qa(question=”Where is Guarani spoken?”)

# out: Prediction(answer=’Guarani is spoken mainly in South America.’)

Each field’s name should indicate the semantic role that the input (or output) field plays in the transformation. DSPy will parse the notation and expand the field names into meaningful instructions for the LM, so that english\_document -> french\_translation would prompt for English-to-French translation.

When needed DSPy offers more advanced programming interfaces for expressing more explicit constraints on signatures.

Note on Advanced Signatures

When more control is desired, one can express signatures as Python classes to provide explicit instructions of the transformation and describe the format or role of each field more directly. For instance, the following signature generates search queries using context and optional question:

1 class GenerateSearchQuery(dspy.Signature):

2 “””Write a simple search query that will help answer complex question.”””

3

4 context = dspy.InputField(desc=”may contain relevant facts”)

5 question = dspy.InputField()

6 query = dspy.OutputField(dtype=dspy.SearchQuery)

Using the above, we can specify a complete system for the generation of a synthetic IR dataset where the queries are mediated by a question generated by the LM:

1 query\_gen = dspy.Predict(GenerateSearchQuery)

2 query\_gen(context=”Language typology”)

3 # Out: Prediction(question=’What are the main types of language classification?’,

4 # query=’”language classification” OR “language typology” -Wikipedia’)

If questions are available, they can be supplied as shown:

1 query\_gen(context=”Language typology”, question=”What are the primary language families of South America?”)

As a work in progress feature, users can optionally specify the type of output fields as bool, int, float, list or dict instead of the default free-form string type as in contexts, question -> answer\_found: bool.

Similarly to type signatures in programming languages, DSPy signatures simply define an interface and provide type-like hints on the expected behavior. To use a signature, we must declare a *module* with that signature like we instantiated Predict module in the earlier examples.

Notes on the Predict Module

The core module for working with signatures is Predict.

class Predict(dspy.Module):

def \_\_init\_\_(self, signature, \*\*config):

self.signature = dspy.Signature(signature)

self.config = config

# Module parameters

self.lm = dspy.ParameterLM(None) # use the default LM

self.demonstrations = dspy.ParameterDemonstrations([])