# About DSPy

**DSPy is a framework for algorithmically optimizing LM prompts and weights**, especially when LMs are used one or more times within a pipeline. To use LMs to build a complex system *without* DSPy, you generally have to: (1) break the problem down into steps, (2) prompt your LM well until each step works well in isolation, (3) tweak the steps to work well together, (4) generate synthetic examples to tune each step, and (5) use these examples to finetune smaller LMs to cut costs. Currently, this is hard and messy: every time you change your pipeline, your LM, or your data, all prompts (or finetuning steps) may need to change.

To make this more systematic and much more powerful, **DSPy** does two things. First, it separates the flow of your program (`modules`) from the parameters (LM prompts and weights) of each step. Second, **DSPy** introduces new `optimizers`, which are LM-driven algorithms that can tune the prompts and/or the weights of your LM calls, given a `metric` you want to maximize.

**DSPy** can routinely teach powerful models like `GPT-3.5` or `GPT-4` and local models like `T5-base` or `Llama2-13b` to be much more reliable at tasks, i.e. having higher quality and/or avoiding specific failure patterns. **DSPy** optimizers will "compile" the *same* program into *different* instructions, few-shot prompts, and/or weight updates (finetunes) for each LM. This is a new paradigm in which LMs and their prompts fade into the background as optimizable pieces of a larger system that can learn from data. **tldr;** less prompting, higher scores, and a more systematic approach to solving hard tasks with LMs.

## Analogy to Neural Networks

When we build neural networks, we don't write manual *for-loops* over lists of *hand-tuned* floats. Instead, you might use a framework like PyTorch to compose layers (e.g., `Convolution` or `Dropout`) and then use optimizers (e.g., SGD or Adam) to learn the parameters of the network.

Ditto! **DSPy** gives you the right general-purpose modules (e.g., `ChainOfThought`, `ReAct`, etc.), which replace string-based prompting tricks. To replace prompt hacking and one-off synthetic data generators, **DSPy** also gives you general optimizers (`BootstrapFewShotWithRandomSearch` or `MIPRO`), which are algorithms that update parameters in your program. Whenever you modify your code, your data, your assertions, or your metric, you can *compile* your program again and **DSPy** will create new effective prompts that fit your changes.

# DSPy

## Programming—not prompting—Language Models

**Get Started with DSPy**

## The Way of DSPy

### Systematic Optimization

Choose from a range of optimizers to enhance your program. Whether it's generating refined instructions, or fine-tuning weights, DSPy's optimizers are engineered to maximize efficiency and effectiveness.

### Modular Approach

With DSPy, you can build your system using predefined modules, replacing intricate prompting techniques with straightforward, effective solutions.

## Cross-LM Compatibility

Whether you're working with powerhouse models like GPT-3.5 or GPT-4, or local models such as T5-base or Llama2-13b, DSPy seamlessly integrates and enhances their performance in your system.

# Tutorials

Step-by-step illustrations of solving a task in DSPy.

### [01] RAG: Retrieval-Augmented Generation

Retrieval-augmented generation (RAG) is an approach that allows LLMs to tap into a large corpus of knowledge from sources and query its knowledge s…

### [02] Multi-Hop Question Answering

A single search query is often not enough for complex QA tasks. For instance, an example within HotPotQA includes a question about the birth city of t…

### Community Examples

The DSPy team believes complexity has to be justified. We take this seriously: we never release a complex tutorial (above) or example (below) unless we…

### Additional Resources

Tutorials

# API References

Welcome to the API References for DSPy! This is where you'll find easy-to-understand information about all the parts of DSPy that you can use in your projects. We've got guides on different tools and helpers that DSPy has, like modules and optimizers. Everything is sorted so you can quickly find what you need. If you're making something and need to quickly get started with DSPy to do certain tasks, this place will show you how to set it up and get it working just right.

# DSPy Cheatsheet

This page will contain snippets for frequent usage patterns.

## DSPy DataLoaders

Import and initializing a DataLoader Object:

```
import dspy
from dspy.datasets import DataLoader

dl = DataLoader()
```

### Loading from HuggingFace Datasets

```
code_alpaca = dl.from_huggingface("HuggingFaceH4/CodeAlpaca_20K")
```

You can access the dataset of the splits by calling key of the corresponding split:

```
train_dataset = code_alpaca['train']
test_dataset = code_alpaca['test']
```

### Loading specific splits from HuggingFace

You can also manually specify splits you want to include as a parameters and it'll return a dictionary where keys are splits that you specified:

```
code_alpaca = dl.from_huggingface(
    "HuggingFaceH4/CodeAlpaca_20K",
    split = ["train", "test"],
)

print(f"Splits in dataset: {code_alpaca.keys()}")
```

If you specify a single split then dataloader will return a List of `dspy.Example` instead of dictionary:

```
code_alpaca = dl.from_huggingface(
    "HuggingFaceH4/CodeAlpaca_20K",
    split = "train",
)

print(f"Number of examples in split: {len(code_alpaca)}")
```

You can slice the split just like you do with HuggingFace Dataset too:

```
code_alpaca_80 = dl.from_huggingface(
    "HuggingFaceH4/CodeAlpaca_20K",
    split = "train[:80%]",
)

print(f"Number of examples in split: {len(code_alpaca_80)}")

code_alpaca_20_80 = dl.from_huggingface(
    "HuggingFaceH4/CodeAlpaca_20K",
    split = "train[20%:80%]"
```

```
    split = 'train[20%:80%]',
)

print(f"Number of examples in split: {len(code_alpaca_20_80)}")
```

### Loading specific subset from HuggingFace

If a dataset has a subset you can pass it as an arg like you do with `load_dataset` in HuggingFace:

```
gms8k = dl.from_huggingface(
    "gsm8k",
    "main",
    input_keys = ("question",),
)

print(f"Keys present in the returned dict: {list(gms8k.keys())}")

print(f"Number of examples in train set: {len(gms8k['train'])}")
print(f"Number of examples in test set: {len(gms8k['test'])}")
```

### Loading from CSV

```
dolly_100_dataset = dl.from_csv("dolly_subset_100_rows.csv",)
```

You can choose only selected columns from the csv by specifying them in the arguments:

```
dolly_100_dataset = dl.from_csv(
    "dolly_subset_100_rows.csv",
    fields=("instruction", "context", "response"),
    input_keys=("instruction", "context")
)
```

### Splitting a List of `dspy.Example`

```
splits = dl.train_test_split(dataset, train_size=0.8) # `dataset` is a List of dspy.Example
train_dataset = splits['train']
test_dataset = splits['test']
```

### Sampling from List of `dspy.Example`

```
sampled_example = dl.sample(dataset, n=5) # `dataset` is a List of dspy.Example
```

## DSPy Programs

### dspy.Signature

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

    question = dspy.InputField()
    answer = dspy.OutputField(desc="often between 1 and 5 words")
```

### dspy.ChainOfThought

```
generate_answer = dspy.ChainOfThought(BasicQA)

# Call the predictor on a particular input alongside a hint.
question='What is the color of the sky?'
```

```
pred = generate_answer(question=question)
```

### dspy.ChainOfThoughtwithHint

```
generate_answer = dspy.ChainOfThoughtWithHint(BasicQA)

# Call the predictor on a particular input alongside a hint.
question='What is the color of the sky?'
hint = "It's what you often see during a sunny day."
pred = generate_answer(question=question, hint=hint)
```

### dspy.ProgramOfThought

```
pot = dspy.ProgramOfThought(BasicQA)

question = 'Sarah has 5 apples. She buys 7 more apples from the store. How many apples does Sarah have now?'
result = pot(question=question)

print(f"Question: {question}")
print(f"Final Predicted Answer (after ProgramOfThought process): {result.answer}")
```

### dspy.ReACT

```
react_module = dspy.ReAct(BasicQA)

question = 'Sarah has 5 apples. She buys 7 more apples from the store. How many apples does Sarah have now?'
result = react_module(question=question)

print(f"Question: {question}")
print(f"Final Predicted Answer (after ReAct process): {result.answer}")
```

### dspy.Retrieve

```
colbertv2_wiki17_abstracts = dspy.ColBERTv2(url='http://20.102.90.50:2017/wiki17_abstracts')
dspy.settings.configure(rm=colbertv2_wiki17_abstracts)

#Define Retrieve Module
retriever = dspy.Retrieve(k=3)

query='When was the first FIFA World Cup held?'

# Call the retriever on a particular query.
topK_passages = retriever(query).passages

for idx, passage in enumerate(topK_passages):
    print(f'{idx+1}]', passage, '\n')
```

## DSPy Metrics

### Function as Metric

To create a custom metric you can create a function that returns either a number or a boolean value:

```
def parse_integer_answer(answer, only_first_line=True):
    try:
        if only_first_line:
            answer = answer.strip().split('\n')[0]
```

```
        # find the last token that has a number in it
        answer = [token for token in answer.split() if any(c.isdigit() for c in token)][-1]
        answer = answer.split('.')[0]
        answer = ''.join([c for c in answer if c.isdigit()])
        answer = int(answer)

    except (ValueError, IndexError):
        # print(answer)
        answer = 0

    return answer

# Metric Function
def gsm8k_metric(gold, pred, trace=None) -> int:
    return int(parse_integer_answer(str(gold.answer))) == int(parse_integer_answer(str(pred.answer)))
```

### LLM as Judge

```
class FactJudge(dspy.Signature):
    """Judge if the answer is factually correct based on the context."""

    context = dspy.InputField(desc="Context for the prediciton")
    question = dspy.InputField(desc="Question to be answered")
    answer = dspy.InputField(desc="Answer for the question")
    factually_correct = dspy.OutputField(desc="Is the answer factually correct based on the context?",
prefix="Factual[Yes/No]:")

judge = dspy.ChainOfThought(FactJudge)

def factuality_metric(example, pred):
    factual = judge(context=example.context, question=example.question, answer=pred.answer)
    return int(factual=="Yes")
```

## DSPy Evaluation

```
from dspy.evaluate import Evaluate

evaluate_program = Evaluate(devset=devset, metric=your_defined_metric, num_threads=NUM_THREADS,
display_progress=True, display_table=num_rows_to_display)

evaluate_program(your_dspy_program)
```

## DSPy Optimizers

### LabeledFewShot

```
from dspy.teleprompt import LabeledFewShot

labeled_fewshot_optimizer = LabeledFewShot(k=8)
your_dspy_program_compiled = labeled_fewshot_optimizer.compile(student = your_dspy_program,
trainset=trainset)
```

### BootstrapFewShot

```
from dspy.teleprompt import BootstrapFewShot

fewshot_optimizer = BootstrapFewShot(metric=your_defined_metric, max_bootstrapped_demos=4,
max_labeled_demos=16, max_rounds=1, max_errors=5)

your_dspy_program_compiled = fewshot_optimizer.compile(student = your_dspy_program, trainset=trainset)
```

**Using another LM for compilation, specifying in teacher_settings**

```
from dspy.teleprompt import BootstrapFewShot

fewshot_optimizer = BootstrapFewShot(metric=your_defined_metric, max_bootstrapped_demos=4,
max_labeled_demos=16, max_rounds=1, max_errors=5, teacher_settings=dict(lm=gpt4))

your_dspy_program_compiled = fewshot_optimizer.compile(student = your_dspy_program, trainset=trainset)
```

**Compiling a compiled program - bootstrapping a bootstrapped program**

```
your_dspy_program_compiledx2 = teleprompter.compile(
    your_dspy_program,
    teacher=your_dspy_program_compiled,
    trainset=trainset,
)
```

**Saving/loading a compiled program**

```
save_path = './v1.json'
your_dspy_program_compiledx2.save(save_path)
```

```
loaded_program = YourProgramClass()
loaded_program.load(path=save_path)
```

## BootstrapFewShotWithRandomSearch

```
from dspy.teleprompt import BootstrapFewShotWithRandomSearch

fewshot_optimizer = BootstrapFewShotWithRandomSearch(metric=your_defined_metric,
max_bootstrapped_demos=2, num_candidate_programs=8, num_threads=NUM_THREADS)

your_dspy_program_compiled = fewshot_optimizer.compile(student = your_dspy_program, trainset=trainset,
valset=devset)
```

Other custom configurations are similar to customizing the `BootstrapFewShot` optimizer.

## Ensemble

```
from dspy.teleprompt import BootstrapFewShotWithRandomSearch
from dspy.teleprompt.ensemble import Ensemble

fewshot_optimizer = BootstrapFewShotWithRandomSearch(metric=your_defined_metric,
max_bootstrapped_demos=2, num_candidate_programs=8, num_threads=NUM_THREADS)
your_dspy_program_compiled = fewshot_optimizer.compile(student = your_dspy_program, trainset=trainset,
valset=devset)

ensemble_optimizer = Ensemble(reduce_fn=dspy.majority)
programs = [x[-1] for x in your_dspy_program_compiled.candidate_programs]
your_dspy_program_compiled_ensemble = ensemble_optimizer.compile(programs[:3])
```

## BootstrapFinetune

```
from dspy.teleprompt import BootstrapFewShotWithRandomSearch, BootstrapFinetune

#Compile program on current dspy.settings.lm
fewshot_optimizer = BootstrapFewShotWithRandomSearch(metric=your_defined_metric,
```

```
max_bootstrapped_demos=2, num_threads=NUM_THREADS)
your_dspy_program_compiled = tp.compile(your_dspy_program, trainset=trainset[:some_num],
valset=trainset[some_num:])

#Configure model to finetune
config = dict(target=model_to_finetune, epochs=2, bf16=True, bsize=6, accumsteps=2, lr=5e-5)

#Compile program on BootstrapFinetune
finetune_optimizer = BootstrapFinetune(metric=your_defined_metric)
finetune_program = finetune_optimizer.compile(your_dspy_program,
trainset=some_new_dataset_for_finetuning_model, **config)

finetune_program = your_dspy_program

#Load program and activate model's parameters in program before evaluation
ckpt_path = "saved_checkpoint_path_from_finetuning"
LM = dspy.HFModel(checkpoint=ckpt_path, model=model_to_finetune)

for p in finetune_program.predictors():
    p.lm = LM
    p.activated = False
```

### COPRO

```
from dspy.teleprompt import COPRO

eval_kwargs = dict(num_threads=16, display_progress=True, display_table=0)

copro_teleprompter = COPRO(prompt_model=model_to_generate_prompts, metric=your_defined_metric,
breadth=num_new_prompts_generated, depth=times_to_generate_prompts,
init_temperature=prompt_generation_temperature, verbose=False)

compiled_program_optimized_signature = copro_teleprompter.compile(your_dspy_program, trainset=trainset,
eval_kwargs=eval_kwargs)
```

### MIPRO

```
from dspy.teleprompt import MIPRO

teleprompter = MIPRO(prompt_model=model_to_generate_prompts, task_model=model_that_solves_task,
metric=your_defined_metric, num_candidates=num_new_prompts_generated,
init_temperature=prompt_generation_temperature)

kwargs = dict(num_threads=NUM_THREADS, display_progress=True, display_table=0)

compiled_program_optimized_bayesian_signature = teleprompter.compile(your_dspy_program,
trainset=trainset, num_trials=100, max_bootstrapped_demos=3, max_labeled_demos=5, eval_kwargs=kwargs)
```

### Signature Optimizer with Types

```
from dspy.teleprompt.signature_opt_typed import optimize_signature
from dspy.evaluate.metrics import answer_exact_match
from dspy.functional import TypedChainOfThought

compiled_program = optimize_signature(
    student=TypedChainOfThought("question -> answer"),
    evaluator=Evaluate(devset=devset, metric=answer_exact_match, num_threads=10,
display_progress=True),
    n_iterations=50,
).program
```

### KNNFewShot

```
from dspy.predict import KNN
from dspy.teleprompt import KNNFewShot

knn_optimizer = KNNFewShot(KNN, k=3, trainset=trainset)

your_dspy_program_compiled = knn_optimizer.compile(student=your_dspy_program, trainset=trainset,
valset=devset)
```

### BootstrapFewShotWithOptuna

```
from dspy.teleprompt import BootstrapFewShotWithOptuna

fewshot_optuna_optimizer = BootstrapFewShotWithOptuna(metric=your_defined_metric,
max_bootstrapped_demos=2, num_candidate_programs=8, num_threads=NUM_THREADS)

your_dspy_program_compiled = fewshot_optuna_optimizer.compile(student=your_dspy_program,
trainset=trainset, valset=devset)
```

Other custom configurations are similar to customizing the `dspy.BootstrapFewShot` optimizer.

## DSPy Assertions

### Including `dspy.Assert` and `dspy.Suggest` statements

```
dspy.Assert(your_validation_fn(model_outputs), "your feedback message",
target_module="YourDSPyModuleSignature")

dspy.Suggest(your_validation_fn(model_outputs), "your feedback message",
target_module="YourDSPyModuleSignature")
```

### Activating DSPy Program with Assertions

**Note**: To use Assertions properly, you must **activate** a DSPy program that includes `dspy.Assert` or `dspy.Suggest` statements from either of the methods above.

```
#1. Using `assert_transform_module:
from dspy.primitives.assertions import assert_transform_module, backtrack_handler

program_with_assertions = assert_transform_module(ProgramWithAssertions(), backtrack_handler)

#2. Using `activate_assertions()`
program_with_assertions = ProgramWithAssertions().activate_assertions()
```

## Compiling with DSPy Programs with Assertions

```
program_with_assertions = assert_transform_module(ProgramWithAssertions(), backtrack_handler)
fewshot_optimizer = BootstrapFewShotWithRandomSearch(metric = your_defined_metric,
max_bootstrapped_demos=2, num_candidate_programs=6)
compiled_dspy_program_with_assertions = fewshot_optimizer.compile(student=program_with_assertions,
teacher = program_with_assertions, trainset=trainset, valset=devset) #student can also be
program_without_assertions
```

# Quick Start

Getting started with DSPy, for building and optimizing LM pipelines.

## Installation

To install DSPy run:

## Minimal Working Example

In this post, we walk you through a minimal working example using the DSPy library.

# DSPy Building Blocks

DSPy introduces signatures (to abstract prompts), modules (to abstract prompting techniques), and optimizers that can tune the prompts (or weights) of modules.

### Using DSPy in 8 Steps

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

### Language Models

The most powerful features in DSPy revolve around algorithmically optimizing the prompts (or weights) of LMs, especially when you're building progra…

### Signatures

When we assign tasks to LMs in DSPy, we specify the behavior we need as a Signature.

### Modules

A DSPy module is a building block for programs that use LMs.

### Data

DSPy is a machine learning framework, so working in it involves training sets, development sets, and test sets.

### Metrics

DSPy is a machine learning framework, so you must think about your automatic metrics for evaluation (to track your progress) and optimization (so DSP…

### Optimizers (formerly Teleprompters)

A DSPy optimizer is an algorithm that can tune the parameters of a DSPy program (i.e., the prompts and/or the LM weights) to maximize the metrics yo…

### DSPy Assertions

Introduction

## Typed Predictors

In DSPy Signatures, we have InputField and OutputField that define the nature of inputs and outputs of the field. However, the inputs and output to th…

# FAQs

## Is DSPy right for me? DSPy vs. other frameworks

The **DSPy** philosophy and abstraction differ significantly from other libraries and frameworks, so it's usually straightforward to decide when **DSPy** is (or isn't) the right framework for your usecase. If you're a NLP/AI researcher (or a practitioner exploring new pipelines or new tasks), the answer is generally an invariable **yes**. If you're a practitioner doing other things, please read on.

**DSPy vs. thin wrappers for prompts (OpenAI API, MiniChain, basic templating)** In other words: *Why can't I just write my prompts directly as string templates?* Well, for extremely simple settings, this *might* work just fine. (If you're familiar with neural networks, this is like expressing a tiny two-layer NN as a Python for-loop. It kinda works.) However, when you need higher quality (or manageable cost), then you need to iteratively explore multi-stage decomposition, improved prompting, data bootstrapping, careful finetuning, retrieval augmentation, and/or using smaller (or cheaper, or local) models. The true expressive power of building with foundation models lies in the interactions between these pieces. But every time you change one piece, you likely break (or weaken) multiple other components. **DSPy** cleanly abstracts away (*and* powerfully optimizes) the parts of these interactions that are external to your actual system design. It lets you focus on designing the module-level interactions: the *same program* expressed in 10 or 20 lines of **DSPy** can easily be compiled into multi-stage instructions for `GPT-4`, detailed prompts for `Llama2-13b`, or finetunes for `T5-base`. Oh, and you wouldn't need to maintain long, brittle, model-specific strings at the core of your project anymore.

**DSPy vs. application development libraries like LangChain, LlamaIndex** LangChain and LlamaIndex target high-level application development; they offer *batteries-included*, pre-built application modules that plug in with your data or configuration. If you'd be happy to use a generic, off-the-shelf prompt for question answering over PDFs or standard text-to-SQL, you will find a rich ecosystem in these libraries. **DSPy** doesn't internally contain hand-crafted prompts that target specific applications. Instead, **DSPy** introduces a small set of much more powerful and general-purpose modules *that can learn to prompt (or finetune) your LM within your pipeline on your data*. when you change your data, make tweaks to your program's control flow, or change your target LM, the **DSPy compiler** can map your program into a new set of prompts (or finetunes) that are optimized specifically for this pipeline. Because of this, you may find that **DSPy** obtains the highest quality for your task, with the least effort, provided you're willing to implement (or extend) your own short program. In short, **DSPy** is for when you need a lightweight but automatically-optimizing programming model — not a library of predefined prompts and integrations. If you're familiar with neural networks: This is like the difference between PyTorch (i.e., representing **DSPy**) and HuggingFace Transformers (i.e., representing the higher-level libraries).

**DSPy vs. generation control libraries like Guidance, LMQL, RELM, Outlines** These are all exciting new libraries for controlling the individual completions of LMs, e.g., if you want to enforce JSON output schema or constrain sampling to a particular regular expression. This is very useful in many settings, but it's generally focused on low-level, structured control of a single LM call. It doesn't help ensure the JSON (or structured output) you get is going to be correct or useful for your task. In contrast, **DSPy** automatically optimizes the prompts in your programs to align them with various task needs, which may also include producing valid structured outputs. That said, we are considering allowing **Signatures** in **DSPy** to express regex-like constraints that are implemented by these libraries.

## Basic Usage

**How should I use DSPy for my task?** We wrote a eight-step guide on this. In short, using DSPy is an iterative process. You first 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.

**How do I convert my complex prompt into a DSPy pipeline?** See the same answer above.

**What do DSPy optimizers tune?** Or, *what does compiling actually do?* Each optimizer is different, but they all seek to maximize a metric on your program by updating prompts or LM weights. Current DSPy `optimizers` can inspect your data, simulate traces through your program to generate good/bad examples of each step, propose or refine instructions for each step based on past results, finetune the weights of your LM on self-generated examples, or combine several of these to improve quality or cut cost. We'd

love to merge new optimizers that explore a richer space: most manual steps you currently go through for prompt engineering, "synthetic data" generation, or self-improvement can probably generalized into a DSPy optimizer that acts on arbitrary LM programs.

Other FAQs. We welcome PRs to add formal answers to each of these here. You will find the answer in existing issues, tutorials, or the papers for all or most of these.

- **How do I get multiple outputs?**

You can specify multiple output fields. For the short-form signature, you can list multiple outputs as comma separated values, following the "->" indicator (e.g. "inputs -> output1, output2"). For the long-form signature, you can include multiple `dspy.OutputField`s.

- **How can I work with long responses?**

You can specify the generation of long responses as a `dspy.OutputField`. To ensure comprehensive checks of the content within the long-form generations, you can indicate the inclusion of citations per referenced context. Such constraints such as response length or citation inclusion can be stated through Signature descriptions, or concretely enforced through DSPy Assertions. Check out the LongFormQA notebook to learn more about **Generating long-form length responses to answer questions**.

- **How can I ensure that DSPy doesn't strip new line characters from my inputs or outputs?**

DSPy uses Signatures to format prompts passed into LMs. In order to ensure that new line characters aren't stripped from longer inputs, you must specify `format=str` when creating a field.

```
class UnstrippedSignature(dspy.Signature):
    """Enter some information for the model here."""

    title = dspy.InputField()
    object = dspy.InputField(format=str)
    result = dspy.OutputField(format=str)
```

`object` can now be a multi-line string without issue.

- **How do I define my own metrics? Can metrics return a float?**

You can define metrics as simply Python functions that process model generations and evaluate them based on user-defined requirements. Metrics can compare existent data (e.g. gold labels) to model predictions or they can be used to assess various components of an output using validation feedback from LMs (e.g. LLMs-as-Judges). Metrics can return `bool`, `int`, and `float` types scores. Check out the official Metrics docs to learn more about defining custom metrics and advanced evaluations using AI feedback and/or DSPy programs.

- **How expensive or slow is compiling??**

To reflect compiling metrics, we highlight an experiment for reference, compiling the `SimplifiedBaleen` using the `dspy.BootstrapFewShotWithRandomSearch` optimizer on the `gpt-3.5-turbo-1106` model over 7 candidate programs and 10 threads. We report that compiling this program takes around 6 minutes with 3200 API calls, 2.7 million input tokens and 156,000 output tokens, reporting a total cost of $3 USD (at the current pricing of the OpenAI model).

Compiling DSPy `optimizers` naturally will incur additional LM calls, but we substantiate this overhead with minimalistic executions with the goal of maximizing performance. This invites avenues to enhance performance of smaller models by compiling DSPy programs with larger models to learn enhanced behavior during compile-time and propagate such behavior to the tested smaller model during inference-time.

## Deployment or Reproducibility Concerns

- **How do I save a checkpoint of my compiled program?**

Here is an example of saving/loading a compiled module:

```
cot_compiled = teleprompter.compile(CoT(), trainset=trainset, valset=devset)
```

```
#Saving
cot_compiled.save('compiled_cot_gsm8k.json')

#Loading:
cot = CoT()
cot.load('compiled_cot_gsm8k.json')
```

- **How do I export for deployment?**

Exporting DSPy programs is simply saving them as highlighted above!

- **How do I search my own data?**

Open source libraries such as RAGautouille enable you to search for your own data through advanced retrieval models like ColBERT with tools to embed and index documents. Feel free to integrate such libraries to create searchable datasets while developing your DSPy programs!

- **How do I turn off the cache? How do I export the cache?**

You can turn off the cache by setting the `DSP_CACHEBOOL` environment variable to `False`, which disables the `cache_turn_on` flag.

Your local cache will be saved to the global env directory `os.environ["DSP_CACHEDIR"]` or for notebooks `os.environ["DSP_NOTEBOOK_CACHEDIR"]`. You can usually set the cachedir to `os.path.join(repo_path, 'cache')` and export this cache from here:

```
os.environ["DSP_NOTEBOOK_CACHEDIR"] = os.path.join(os.getcwd(), 'cache')
```

## Advanced Usage

- **How do I parallelize?** You can parallelize DSPy programs during both compilation and evaluation by specifying multiple thread settings in the respective DSPy `optimizers` or within the `dspy.Evaluate` utility function.

- **How do freeze a module?**

Modules can be frozen by setting their `._compiled` attribute to be True, indicating the module has gone through optimizer compilation and should not have its parameters adjusted. This is handled internally in optimizers such as `dspy.BootstrapFewShot` where the student program is ensured to be frozen before the teacher propagates the gathered few-shot demonstrations in the bootstrapping process.

- **How do I get JSON output?**

You can specify JSON-type descriptions in the `desc` field of the long-form signature `dspy.OutputField` (e.g. `output = dspy.OutputField(desc='key-value pairs')`).

If you notice outputs are still not conforming to JSON formatting, try Asserting this constraint! Check out Assertions (or the next question!)

- **How do I use DSPy assertions?**

  a) **How to Add Assertions to Your Program**:

  - **Define Constraints**: Use `dspy.Assert` and/or `dspy.Suggest` to define constraints within your DSPy program. These are based on boolean validation checks for the outcomes you want to enforce, which can simply be Python functions to validate the model outputs.
  - **Integrating Assertions**: Keep your Assertion statements following a model generations (hint: following a module layer)

  b) **How to Activate the Assertions**:

  i. **Using `assert_transform_module`**:

i. **Using `assert_transform_module`**:
   - Wrap your DSPy module with assertions using the `assert_transform_module` function, along with a `backtrack_handler`. This function transforms your program to include internal assertions backtracking and retry logic, which can be customized as well: `program_with_assertions = assert_transform_module(ProgramWithAssertions(), backtrack_handler)`

ii. **Activate Assertions**:
   - Directly call `activate_assertions` on your DSPy program with assertions: `program_with_assertions = ProgramWithAssertions().activate_assertions()`

**Note**: To use Assertions properly, you must **activate** a DSPy program that includes `dspy.Assert` or `dspy.Suggest` statements from either of the methods above.

# Errors

- **How do I deal with "context too long" errors?**

If you're dealing with "context too long" errors in DSPy, you're likely using DSPy optimizers to include demonstrations within your prompt, and this is exceeding your current context window. Try reducing these parameters (e.g. `max_bootstrapped_demos` and `max_labeled_demos`). Additionally, you can also reduce the number of retrieved passages/docs/embeddings to ensure your prompt is fitting within your model context length.

A more general fix is simply increasing the number of `max_tokens` specified to the LM request (e.g. `lm = dspy.OpenAI(model = ..., max_tokens = ...`).

- **How do I deal with timeouts or backoff errors?**

Firstly, please refer to your LM/RM provider to ensure stable status or sufficient rate limits for your use case!

Additionally, try reducing the number of threads you are testing on as the corresponding servers may get overloaded with requests and trigger a backoff + retry mechanism.

If all variables seem stable, you may be experiencing timeouts or backoff errors due to incorrect payload requests sent to the api providers. Please verify your arguments are compatible with the SDK you are interacting with. At times, DSPy may have hard-coded arguments that are not relevant for your compatible, in which case, please free to open a PR alerting this or comment out these default settings for your usage.

# Contributing

**What if I have a better idea for prompting or synthetic data generation?** Perfect. We encourage you to think if it's best expressed as a module or an optimizer, and we'd love to merge it in DSPy so everyone can use it. DSPy is not a complete project; it's an ongoing effort to create structure (modules and optimizers) in place of hacky prompt and pipeline engineering tricks.

**How can I add my favorite LM or vector store?**

Check out these walkthroughs on setting up a Custom LM client and Custom RM client.

# Deep Dive

DSPy introduces signatures (to abstract prompts), modules (to abstract prompting techniques), and optimizers that can tune the prompts (or weights) of modules.

## Data Handling

3 items

## Signatures

3 items

## Modules

6 items

## Typed Predictors

2 items

## Language Model Clients

3 items

## Retrieval Model Clients

6 items

## Teleprompters

2 items

# Build & Release Workflow Implementation

The build_and_release workflow automates deployments of dspy-ai to pypi. For a guide to triggering a release using the workflow, refer to release checklist.

## Overview

At a high level, the workflow works as follows:

1. Maintainer of the repo pushes a tag following semver versioning for the new release.
2. This triggers the github action which extracts the tag (the version)
3. Builds and publishes a release on test-pypi
4. Uses the test-pypi release to run build_utils/tests/intro.py with the new release as an integration test. Note intro.py is a copy of the intro notebook.
5. Assuming the test runs successfully, it pushes a release to pypi. If not, the user can delete the tag, make the fixes and then push the tag again. Versioning for multiple releases to test-pypi with the same tag version is taken care of by the workflow by appending a pre-release identifier, so the user only needs to consider the version for pypi.
6. (Currently manual) the user creates a release and includes release notes, as described in docs/docs/release-checklist.md

## Implementation Details

The workflow executes a series of jobs in sequence:

- extract-tag
- build-and-publish-test-pypi
- test-intro-script
- build-and-publish-pypi

**extract-tag**
Extracts the tag pushed to the commit. This tag is expected to be the version of the new deployment.

**build-and-publish-test-pypi**
Builds and publishes the package to test-pypi.

1. Determines the version that should be deployed to test-pypi. There may be an existing deployment with the version specified by the tag in the case that a deployment failed and the maintainer made some changes and pushed the same tag again (which is the intended usage). The following logic is implemented test_version.py
    i. Load the releases on test-pypi
    ii. Check if there is a release matching our current tag
        a. If not, create a release with the current tag
        b. If it exists, oad the latest published version (this will either be the version with the tag itself, or the tag + a pre-release version). In either case, increment the pre-release version.
2. Updates the version placeholder in setup.py to the version obtained in step 1.
3. Updates the version placeholder in pyproject.toml to the version obtained in step 1.
4. Updates the package name placeholder in setup.py to `dspy-ai-test`*
5. Updates the package name placeholder in pyproject.toml to `dspy-ai-test`*
6. Builds the binary wheel
7. Publishes the package to test-pypi.

**test-intro-script**
Runs the pytest containing the intro script as an integration test using the package published to test-pypi. This is a validation step before publishing to pypi.

1. Uses a loop to install the version just published to test-pypi as sometimes there is a race condition between the package becoming available for installation and this job executing.

2. Runs the test to ensure the package is working as expected.
3. If this fails, the workflow fails and the maintainer needs to make a fix and delete and then recreate the tag.

**build-and-publish-pypi**
Builds and publishes the package to pypi.

1. Updates the version placeholder in setup.py to the version obtained in step 1.
2. Updates the version placeholder in pyproject.toml to the version obtained in step 1.
3. Updates the package name placeholder in setup.py to `dspy-ai`*
4. Updates the package name placeholder in pyproject.toml to `dspy-ai`*
5. Builds the binary wheel
6. Publishes the package to pypi.

* The package name is updated by the worfklow to allow the same files to be used to build both the pypi and test-pypi packages.

# [01] RAG: Retrieval-Augmented Generation

Retrieval-augmented generation (RAG) is an approach that allows LLMs to tap into a large corpus of knowledge from sources and query its knowledge store to find relevant passages/content and produce a well-refined response.

RAG ensures LLMs can dynamically utilize real-time knowledge even if not originally trained on the subject and give thoughtful answers. However, with this nuance comes greater complexities in setting up refined RAG pipelines. To reduce these intricacies, we turn to **DSPy**, which offers a seamless approach to setting up prompting pipelines!

## Configuring LM and RM

We'll start by setting up the language model (LM) and retrieval model (RM), which **DSPy** supports through multiple LM and RM APIs and local models hosting.

In this notebook, we'll work with GPT-3.5 (`gpt-3.5-turbo`) and the `ColBERTv2` retriever (a free server hosting a Wikipedia 2017 "abstracts" search index containing the first paragraph of each article from this 2017 dump). We configure the LM and RM within DSPy, allowing DSPy to internally call the respective module when needed for generation or retrieval.

```
import dspy

turbo = dspy.OpenAI(model='gpt-3.5-turbo')
colbertv2_wiki17_abstracts = dspy.ColBERTv2(url='http://20.102.90.50:2017/wiki17_abstracts')

dspy.settings.configure(lm=turbo, rm=colbertv2_wiki17_abstracts)
```

## Loading the Dataset

For this tutorial, we make use of the `HotPotQA` dataset, a collection of complex question-answer pairs typically answered in a multi-hop fashion. We can load this dataset provided by DSPy through the `HotPotQA` class:

```
from dspy.datasets import HotPotQA

# Load the dataset.
dataset = HotPotQA(train_seed=1, train_size=20, eval_seed=2023, dev_size=50, test_size=0)

# Tell DSPy that the 'question' field is the input. Any other fields are labels and/or metadata.
trainset = [x.with_inputs('question') for x in dataset.train]
devset = [x.with_inputs('question') for x in dataset.dev]

len(trainset), len(devset)
```

**Output:**

```
(20, 50)
```

## Building Signatures

Now that we have the data loaded, let's start defining the signatures for the sub-tasks of our pipeline.

We can identify our simple input `question` and output `answer`, but since we are building out a RAG pipeline, we wish to utilize some contextual information from our ColBERT corpus. So let's define our signature: `context, question --> answer`.

```
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")
```

We include small descriptions for the `context` and `answer` fields to define more robust guidelines on what the model will receive and should generate.

## Building the Pipeline

We will build our RAG pipeline as a DSPy module which will require two methods:

- The `__init__` method will simply declare the sub-modules it needs: `dspy.Retrieve` and `dspy.ChainOfThought`. The latter is defined to implement our `GenerateAnswer` signature.
- The `forward` method will describe the control flow of answering the question using the modules we have: Given a question, we'll search for the top-3 relevant passages and then feed them as context for answer generation.

```
class RAG(dspy.Module):
    def __init__(self, num_passages=3):
        super().__init__()

        self.retrieve = dspy.Retrieve(k=num_passages)
        self.generate_answer = dspy.ChainOfThought(GenerateAnswer)

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

## Optimizing the Pipeline

**Compiling the RAG program**

Having defined this program, let's now **compile** it. Compiling a program will update the parameters stored in each module. In our setting, this is primarily in the form of collecting and selecting good demonstrations for inclusion within the prompt(s).

Compiling depends on three things:

1. **A training set.** We'll just use our 20 question–answer examples from `trainset` above.
2. **A metric for validation.** We'll define a simple `validate_context_and_answer` that checks that the predicted answer is correct and that the retrieved context actually contains the answer.
3. **A specific teleprompter.** The **DSPy** compiler includes a number of **teleprompters** that can optimize your programs.

```
from dspy.teleprompt import BootstrapFewShot

# Validation logic: check that the predicted answer is correct.
# Also check that the retrieved context does actually contain that answer.
def validate_context_and_answer(example, pred, trace=None):
    answer_EM = dspy.evaluate.answer_exact_match(example, pred)
    answer_PM = dspy.evaluate.answer_passage_match(example, pred)
    return answer_EM and answer_PM

# Set up a basic teleprompter, which will compile our RAG program.
teleprompter = BootstrapFewShot(metric=validate_context_and_answer)

# Compile!
compiled_rag = teleprompter.compile(RAG(), trainset=trainset)
```

> **INFO**
>
> **Teleprompters:** Teleprompters are powerful optimizers that can take any program and learn to bootstrap and select effective prompts for its modules. Hence the name which means "prompting at a distance".

> Different teleprompters offer various tradeoffs in terms of how much they optimize cost versus quality, etc. We will used a simple default `BootstrapFewShot` in the example above.
>
> *If you're into analogies, you could think of this as your training data, your loss function, and your optimizer in a standard DNN supervised learning setup. Whereas SGD is a basic optimizer, there are more sophisticated (and more expensive!) ones like Adam or RMSProp.*

## Executing the Pipeline

Now that we've compiled our RAG program, let's try it out.

```
# Ask any question you like to this simple RAG program.
my_question = "What castle did David Gregory inherit?"

# Get the prediction. This contains `pred.context` and `pred.answer`.
pred = compiled_rag(my_question)

# Print the contexts and the answer.
print(f"Question: {my_question}")
print(f"Predicted Answer: {pred.answer}")
print(f"Retrieved Contexts (truncated): {[c[:200] + '...' for c in pred.context]}")
```

Excellent. How about we inspect the last prompt for the LM?

```
turbo.inspect_history(n=1)
```

**Output:**

```
Answer questions with short factoid answers.

---

Question: At My Window was released by which American singer-songwriter?
Answer: John Townes Van Zandt

Question: "Everything Has Changed" is a song from an album released under which record label ?
Answer: Big Machine Records
...(truncated)
```

Even though we haven't written any of this detailed demonstrations, we see that DSPy was able to bootstrap this 3,000 token prompt for 3-shot retrieval-augmented generation with hard negative passages and uses Chain-of-Thought reasoning within an extremely simply-written program.

This illustrates the power of composition and learning. Of course, this was just generated by a particular teleprompter, which may or may not be perfect in each setting. As you'll see in DSPy, there is a large but systematic space of options you have to optimize and validate with respect to your program's quality and cost.

You can also easily inspect the learned objects themselves.

```
for name, parameter in compiled_rag.named_predictors():
    print(name)
    print(parameter.demos[0])
    print()
```

## Evaluating the Pipeline

We can now evaluate our `compiled_rag` program on the dev set. Of course, this tiny set is *not* meant to be a reliable benchmark, but it'll be instructive to use it for illustration.

Let's evaluate the accuracy (exact match) of the predicted answer.

```
from dspy.evaluate.evaluate import Evaluate

# Set up the `evaluate_on_hotpotqa` function. We'll use this many times below.
evaluate_on_hotpotqa = Evaluate(devset=devset, num_threads=1, display_progress=False, display_table=5)

# Evaluate the `compiled_rag` program with the `answer_exact_match` metric.
metric = dspy.evaluate.answer_exact_match
evaluate_on_hotpotqa(compiled_rag, metric=metric)
```

**Output:**

```
Average Metric: 22 / 50  (44.0): 100%|██████████| 50/50 [00:00<00:00, 116.45it/s]
Average Metric: 22 / 50  (44.0%)

44.0
```

## Evaluating the Retrieval

It may also be instructive to look at the accuracy of retrieval. While there are multiple ways to do this, we can simply check whether the retrieved passages contain the answer.

We can make use of our dev set which includes the gold titles that should be retrieved.

```
def gold_passages_retrieved(example, pred, trace=None):
    gold_titles = set(map(dspy.evaluate.normalize_text, example['gold_titles']))
    found_titles = set(map(dspy.evaluate.normalize_text, [c.split(' | ')[0] for c in pred.context]))

    return gold_titles.issubset(found_titles)

compiled_rag_retrieval_score = evaluate_on_hotpotqa(compiled_rag, metric=gold_passages_retrieved)
```

**Output:**

```
Average Metric: 13 / 50  (26.0): 100%|██████████| 50/50 [00:00<00:00, 671.76it/s]Average Metric: 13 /
50  (26.0%)
```

Although this simple `compiled_rag` program is able to answer a decent fraction of the questions correctly (on this tiny set, over 40%), the quality of retrieval is much lower.

This potentially suggests that the LM is often relying on the knowledge it memorized during training to answer questions. To address this weak retrieval, let's explore a second program that involves more advanced search behavior.

# [02] Multi-Hop Question Answering

A single search query is often not enough for complex QA tasks. For instance, an example within `HotPotQA` includes a question about the birth city of the writer of "Right Back At It Again". A search query often identifies the author correctly as "Jeremy McKinnon", but lacks the capability to compose the intended answer in determining when he was born.

The standard approach for this challenge in retrieval-augmented NLP literature is to build multi-hop search systems, like GoldEn (Qi et al., 2019) and Baleen (Khattab et al., 2021). These systems read the retrieved results and then generate additional queries to gather additional information when necessary before arriving to a final answer. Using DSPy, we can easily simulate such systems in a few lines of code.

## Configuring LM and RM

We'll start by setting up the language model (LM) and retrieval model (RM), which **DSPy** supports through multiple LM and RM APIs and local models hosting.

In this notebook, we'll work with GPT-3.5 (`gpt-3.5-turbo`) and the `ColBERTv2` retriever (a free server hosting a Wikipedia 2017 "abstracts" search index containing the first paragraph of each article from this 2017 dump). We configure the LM and RM within DSPy, allowing DSPy to internally call the respective module when needed for generation or retrieval.

```
import dspy

turbo = dspy.OpenAI(model='gpt-3.5-turbo')
colbertv2_wiki17_abstracts = dspy.ColBERTv2(url='http://20.102.90.50:2017/wiki17_abstracts')

dspy.settings.configure(lm=turbo, rm=colbertv2_wiki17_abstracts)
```

## Loading the Dataset

For this tutorial, we make use of the mentioned `HotPotQA` dataset, a collection of complex question-answer pairs typically answered in a multi-hop fashion. We can load this dataset provided by DSPy through the `HotPotQA` class:

```
from dspy.datasets import HotPotQA

# Load the dataset.
dataset = HotPotQA(train_seed=1, train_size=20, eval_seed=2023, dev_size=50, test_size=0)

# Tell DSPy that the 'question' field is the input. Any other fields are labels and/or metadata.
trainset = [x.with_inputs('question') for x in dataset.train]
devset = [x.with_inputs('question') for x in dataset.dev]

len(trainset), len(devset)
```

**Output:**

```
(20, 50)
```

## Building Signature

Now that we have the data loaded let's start defining the signatures for sub-tasks of out Baleen pipeline.

We'll start by creating the `GenerateAnswer` signature that'll take `context` and `question` as input and give `answer` as output.

```
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")
```

Unlike usual QA pipelines, we have an intermediate question-generation step in Baleen for which we'll need to define a new Signature for the "hop" behavior: inputting some context and a question to generate a search query to find missing information.

```
class GenerateSearchQuery(dspy.Signature):
    """Write a simple search query that will help answer a complex question."""

    context = dspy.InputField(desc="may contain relevant facts")
    question = dspy.InputField()
    query = dspy.OutputField()
```

> **INFO**
>
> We could have written `context = GenerateAnswer.signature.context` to avoid duplicating the description of the context field.

Now that we have the necessary signatures in place, we can start building the Baleen pipeline!

## Building the Pipeline

So, let's define the program itself `SimplifiedBaleen`. There are many possible ways to implement this, but we'll keep this version down to the key elements.

```
from dsp.utils import deduplicate

class SimplifiedBaleen(dspy.Module):
    def __init__(self, passages_per_hop=3, max_hops=2):
        super().__init__()

        self.generate_query = [dspy.ChainOfThought(GenerateSearchQuery) for _ in range(max_hops)]
        self.retrieve = dspy.Retrieve(k=passages_per_hop)
        self.generate_answer = dspy.ChainOfThought(GenerateAnswer)
        self.max_hops = max_hops

    def forward(self, question):
        context = []

        for hop in range(self.max_hops):
            query = self.generate_query[hop](context=context, question=question).query
            passages = self.retrieve(query).passages
            context = deduplicate(context + passages)

        pred = self.generate_answer(context=context, question=question)
        return dspy.Prediction(context=context, answer=pred.answer)
```

As we can see, the `__init__` method defines a few key sub-modules:

- **generate_query**: For each hop, we will have one `dspy.ChainOfThought` predictor with the `GenerateSearchQuery` signature.
- **retrieve**: This module will conduct the search using the generated queries over our defined ColBERT RM search index via the `dspy.Retrieve` module.
- **generate_answer**: This `dspy.Predict` module will be used with the `GenerateAnswer` signature to produce the final answer.

The `forward` method uses these sub-modules in simple control flow.

1. First, we'll loop up to `self.max_hops` times

1. First, we'll loop up to `self.max_hops` times.
2. In each iteration, we'll generate a search query using the predictor at `self.generate_query[hop]`.
3. We'll retrieve the top-k passages using that query.
4. We'll add the (deduplicated) passages to our `context` accumulator.
5. After the loop, we'll use `self.generate_answer` to produce an answer.
6. We'll return a prediction with the retrieved `context` and predicted `answer`.

## Executing the Pipeline

Let's execute this program in its zero-shot (uncompiled) setting.

This doesn't necessarily imply the performance will be bad but rather that we're bottlenecked directly by the reliability of the underlying LM to understand our sub-tasks from minimal instructions. Often, this is perfectly fine when using the most expensive/powerful models (e.g., GPT-4) on the easiest and most standard tasks (e.g., answering simple questions about popular entities).

```
# Ask any question you like to this simple RAG program.
my_question = "How many storeys are in the castle that David Gregory inherited?"

# Get the prediction. This contains `pred.context` and `pred.answer`.
uncompiled_baleen = SimplifiedBaleen()  # uncompiled (i.e., zero-shot) program
pred = uncompiled_baleen(my_question)

# Print the contexts and the answer.
print(f"Question: {my_question}")
print(f"Predicted Answer: {pred.answer}")
print(f"Retrieved Contexts (truncated): {[c[:200] + '...' for c in pred.context]}")
```

**Output:**

```
Question: How many storeys are in the castle that David Gregory inherited?
Predicted Answer: five
Retrieved Contexts (truncated): ['David Gregory (physician) | David Gregory (20 December 1625 – 1720)
was a Scottish physician and inventor. His surname is sometimes spelt as Gregorie, the original
Scottish spelling. He inherited Kinn...', 'The Boleyn Inheritance | The Boleyn Inheritance is a novel
by British author Philippa Gregory which was first published in 2006. It is a direct sequel to her
previous novel "The Other Boleyn Girl," an...', 'Gregory of Gaeta | Gregory was the Duke of Gaeta from
963 until his death. He was the second son of Docibilis II of Gaeta and his wife Orania. He succeeded
his brother John II, who had left only daugh...', 'Kinnairdy Castle | Kinnairdy Castle is a tower
house, having five storeys and a garret, two miles south of Aberchirder, Aberdeenshire, Scotland. The
alternative name is Old Kinnairdy....', 'Kinnaird Head | Kinnaird Head (Scottish Gaelic: "An Ceann Àrd"
, "high headland") is a headland projecting into the North Sea, within the town of Fraserburgh,
Aberdeenshire on the east coast of Scotla...', 'Kinnaird Castle, Brechin | Kinnaird Castle is a 15th-
century castle in Angus, Scotland. The castle has been home to the Carnegie family, the Earl of
Southesk, for more than 600 years....']
```

We can inspect the last **three** calls to the LM (i.e., generating the first hop's query, generating the second hop's query, and generating the answer) using:

```
turbo.inspect_history(n=3)
```

## Optimizing the Pipeline

However, a zero-shot approach quickly falls short for more specialized tasks, novel domains/settings, and more efficient (or open) models.

To address this, **DSPy** offers compilation. Let's compile our multi-hop (`SimplifiedBaleen`) program.

Let's first define our validation logic for compilation:

Let's first define our validation logic for compilation.

- The predicted answer matches the gold answer.
- The retrieved context contains the gold answer.
- None of the generated queries is rambling (i.e., none exceeds 100 characters in length).
- None of the generated queries is roughly repeated (i.e., none is within 0.8 or higher F1 score of earlier queries).

```
def validate_context_and_answer_and_hops(example, pred, trace=None):
    if not dspy.evaluate.answer_exact_match(example, pred): return False
    if not dspy.evaluate.answer_passage_match(example, pred): return False

    hops = [example.question] + [outputs.query for *_, outputs in trace if 'query' in outputs]

    if max([len(h) for h in hops]) > 100: return False
    if any(dspy.evaluate.answer_exact_match_str(hops[idx], hops[:idx], frac=0.8) for idx in range(2,
len(hops))): return False

    return True
```

We'll use one of the most basic teleprompters in **DSPy**, namely, `BootstrapFewShot` to optimize the predictors in pipeline with few-shot examples.

```
from dspy.teleprompt import BootstrapFewShot

teleprompter = BootstrapFewShot(metric=validate_context_and_answer_and_hops)
compiled_baleen = teleprompter.compile(SimplifiedBaleen(),
teacher=SimplifiedBaleen(passages_per_hop=2), trainset=trainset)
```

## Evaluating the Pipeline

Let's now define our evaluation function and compare the performance of the uncompiled and compiled Baleen pipelines. While this devset does not serve as a completely reliable benchmark, it is instructive to use for this tutorial.

```
from dspy.evaluate.evaluate import Evaluate

# Define metric to check if we retrieved the correct documents
def gold_passages_retrieved(example, pred, trace=None):
    gold_titles = set(map(dspy.evaluate.normalize_text, example["gold_titles"]))
    found_titles = set(
        map(dspy.evaluate.normalize_text, [c.split(" | ")[0] for c in pred.context])
    )
    return gold_titles.issubset(found_titles)

# Set up the `evaluate_on_hotpotqa` function. We'll use this many times below.
evaluate_on_hotpotqa = Evaluate(devset=devset, num_threads=1, display_progress=True, display_table=5)

uncompiled_baleen_retrieval_score = evaluate_on_hotpotqa(uncompiled_baleen,
metric=gold_passages_retrieved, display=False)

compiled_baleen_retrieval_score = evaluate_on_hotpotqa(compiled_baleen, metric=gold_passages_retrieved)

print(f"## Retrieval Score for uncompiled Baleen: {uncompiled_baleen_retrieval_score}")
print(f"## Retrieval Score for compiled Baleen: {compiled_baleen_retrieval_score}")
```

**Output:**

```
## Retrieval Score for uncompiled Baleen: 36.0
## Retrieval Score for compiled Baleen: 60.0
```

Excellent! There might be something to this compiled, multi-hop program then.

Earlier, we said simple programs are not very effective at finding all evidence required for answering each question. Is this resolved by the adding some greater prompting techniques in the forward function of `SimplifiedBaleen`? Does compiling programs improve performance?

While in our tutorial we demonstrate our findings, the answer for these questions will not always be obvious. However, DSPy makes it extremely easy to try out the many diverse approaches with minimal effort.

Now that you've seen a example of how to build a simple yet powerful pipeline, it's time for you to build one yourself!

# Community Examples

The DSPy team believes complexity has to be justified. We take this seriously: we never release a complex tutorial (above) or example (below) *unless we can demonstrate empirically that this complexity has generally led to improved quality or cost.* This kind of rule is rarely enforced by other frameworks or docs, but you can count on it in DSPy examples.

There's a bunch of examples in the `examples/` directory and in the top-level directory. We welcome contributions!

You can find other examples tweeted by @lateinteraction on Twitter/X.

**Some other examples (not exhaustive, feel free to add more via PR):**

- Applying DSPy Assertions
  - Long-form Answer Generation with Citations, by Arnav Singhvi
  - Generating Answer Choices for Quiz Questions, by Arnav Singhvi
  - Generating Tweets for QA, by Arnav Singhvi
- Compiling LCEL runnables from LangChain in DSPy
- AI feedback, or writing LM-based metrics in DSPy
- DSPy Optimizers Benchmark on a bunch of different tasks, by Michael Ryan
- Indian Languages NLI with gains due to compiling by Saiful Haq
- Sophisticated Extreme Multi-Class Classification, IReRa, by Karel D'Oosterlinck
- DSPy on BIG-Bench Hard Example, by Chris Levy
- Using Ollama with DSPy for Mistral (quantized) by @jrknox1977
- Using DSPy, "The Unreasonable Effectiveness of Eccentric Automatic Prompts" (paper) by VMware's Rick Battle & Teja Gollapudi, and interview at TheRegister

There are also recent cool examples at Weaviate's DSPy cookbook by Connor Shorten. See tutorial on YouTube.

# Additional Resources

## Tutorials

| Level | Tutorial | Run in Colab | Description |
|---|---|---|---|
| Beginner | **Getting Started** |  | Introduces the basic building blocks in DSPy. Tackles the task of complex question answering with HotPotQA. |
| Beginner | **Minimal Working Example** | N/A | Builds and optimizes a very simple chain-of-thought program in DSPy for math question answering. Very short. |
| Beginner | **Compiling for Tricky Tasks** | N/A | Teaches LMs to reason about logical statements and negation. Uses GPT-4 to bootstrap few-shot CoT demonstrations for GPT-3.5. Establishes a state-of-the-art result on ScoNe. Contributed by Chris Potts. |
| Beginner | **Local Models & Custom Datasets** |  | Illustrates two different things together: how to use local models (Llama-2-13B in particular) and how to use your own data examples for training and development. |
| Intermediate | **The DSPy Paper** | N/A | Sections 3, 5, 6, and 7 of the DSPy paper can be consumed as a tutorial. They include explained code snippets, results, and discussions of the abstractions and API. |
| Intermediate | **DSPy Assertions** |  | Introduces example of applying DSPy Assertions while generating long-form responses to questions with citations. Presents comparative evaluation in both zero-shot and compiled settings. |
| Intermediate | **Finetuning for Complex Programs** |  | Teaches a local T5 model (770M) to do exceptionally well on HotPotQA. Uses only 200 labeled answers. Uses no hand-written prompts, no calls to OpenAI, and no labels for retrieval or reasoning. |
| Advanced | **Information Extraction** |  | Tackles extracting information from long articles (biomedical research papers). Combines in-context learning and retrieval to set SOTA on BioDEX. Contributed by Karel D'Oosterlinck. |

## Resources

- DSPy talk at ScaleByTheBay Nov 2023.
- DSPy webinar with MLOps Learners, a bit longer with Q&A.
- Hands-on Overviews of DSPy by the community: DSPy Explained! by Connor Shorten, DSPy explained by code_your_own_ai, DSPy Crash Course by AI Bites
- Interviews: Weaviate Podcast in-person, and you can find 6-7 other remote podcasts on YouTube from a few different perspectives/audiences.
- **Tracing in DSPy** with Arize Phoenix: Tutorial for tracing your prompts and the steps of your DSPy programs
- **Tracing & Optimization Tracking in DSPy** with Parea AI: Tutorial on tracing & evaluating a DSPy RAG program

# Typed Predictors

In DSPy Signatures, we have `InputField` and `OutputField` that define the nature of inputs and outputs of the field. However, the inputs and output to these fields are always `str`-typed, which requires input and output string processing.

Pydantic `BaseModel` is a great way to enforce type constraints on the fields, but it is not directly compatible with the `dspy.Signature`. Typed Predictors resolves this as a way to enforce the type constraints on the inputs and outputs of the fields in a `dspy.Signature`.

## Executing Typed Predictors

Using Typed Predictors is not too different than any other module with the minor additions of type hints to signature attributes and using a special Predictor module instead of `dspy.Predict`. Let's take a look at a simple example to understand this.

### Defining Input and Output Models

Let's take a simple task as an example i.e. given the `context` and `query`, the LLM should return an `answer` and `confidence_score`. Let's define our `Input` and `Output` models via pydantic.

```
from pydantic import BaseModel, Field

class Input(BaseModel):
    context: str = Field(description="The context for the question")
    query: str = Field(description="The question to be answered")

class Output(BaseModel):
    answer: str = Field(description="The answer for the question")
    confidence: float = Field(ge=0, le=1, description="The confidence score for the answer")
```

As you can see, we can describe the attributes by defining a simple Signature that takes in the input and returns the output.

### Creating Typed Predictor

A Typed Predictor needs a Typed Signature, which extends a `dspy.Signature` with the addition of specifying "field type".

```
class QASignature(dspy.Signature):
    """Answer the question based on the context and query provided, and on the scale of 10 tell how confident you are about the answer."""

    input: Input = dspy.InputField()
    output: Output = dspy.OutputField()
```

Now that we have the `QASignature`, let's define a Typed Predictor that executes this Signature while conforming to the type constraints.

```
predictor = dspy.TypedPredictor(QASignature)
```

Similar to other modules, we pass the `QASignature` to `dspy.TypedPredictor` which enforces the typed constraints.

And similarly to `dspy.Predict`, we can also use a "string signature", which we type as:

```
predictor = dspy.TypedPredictor("input:Input -> output:Output")
```

### I/O in Typed Predictors

Now let's test out the Typed Predictor by providing some sample input to the predictor and verifying the output type. We can create a `Input` instance and pass it to the predictor to get a prediction of the output.

an `Input` instance and pass it to the predictor to get a dictionary of the output.

```
doc_query_pair = Input(
    context="The quick brown fox jumps over the lazy dog",
    query="What does the fox jumps over?",
)

prediction = predictor(input=doc_query_pair)
```

Let's see the output and its type.

```
answer = prediction.output.answer
confidence_score = prediction.output.confidence

print(f"Prediction: {prediction}\n\n")
print(f"Answer: {answer}, Answer Type: {type(answer)}")
print(f"Confidence Score: {confidence_score}, Confidence Score Type: {type(confidence_score)}")
```

## Typed Chain of Thoughts with `dspy.TypedChainOfThought`

Extending the analogous comparison of `TypedPredictor` to `dspy.Predict`, we create `TypedChainOfThought`, the typed counterpart of `dspy.ChainOfThought`:

```
cot_predictor = dspy.TypedChainOfThought(QASignature)

doc_query_pair = Input(
    context="The quick brown fox jumps over the lazy dog",
    query="What does the fox jumps over?",
)

prediction = cot_predictor(input=doc_query_pair)
```

## Typed Predictors as Decorators

While the `dspy.TypedPredictor` and `dspy.TypedChainOfThought` provide a convenient way to use typed predictors, you can also use them as decorators to enforce type constraints on the inputs and outputs of the function. This relies on the internal definitions of the Signature class and its function arguments, outputs, and docstrings.

```
@dspy.predictor
def answer(doc_query_pair: Input) -> Output:
    """Answer the question based on the context and query provided, and on the scale of 0-1 tell how confident you are about the answer."""
    pass

@dspy.cot
def answer(doc_query_pair: Input) -> Output:
    """Answer the question based on the context and query provided, and on the scale of 0-1 tell how confident you are about the answer."""
    pass

prediction = answer(doc_query_pair=doc_query_pair)
```

## Composing Functional Typed Predictors in `dspy.Module`

If you're creating DSPy pipelines via `dspy.Module`, then you can simply use Functional Typed Predictors by creating these class methods and using them as decorators. Here is an example of using functional typed predictors to create a `SimplifiedBaleen` pipeline:

```
class SimplifiedBaleen(FunctionalModule):
    def __init__(self, passages_per_hop=3, max_hops=1):
        super().__init__()
        self.retrieve = dspy.Retrieve(k=passages_per_hop)
        self.max_hops = max_hops

    @cot
    def generate_query(self, context: list[str], question) -> str:
        """Write a simple search query that will help answer a complex question."""
        pass

    @cot
    def generate_answer(self, context: list[str], question) -> str:
        """Answer questions with short factoid answers."""
        pass

    def forward(self, question):
        context = []

        for _ in range(self.max_hops):
            query = self.generate_query(context=context, question=question)
            passages = self.retrieve(query).passages
            context = deduplicate(context + passages)

        answer = self.generate_answer(context=context, question=question)
        return dspy.Prediction(context=context, answer=answer)
```

## Optimizing Typed Predictors

Typed predictors can be optimized on the Signature instructions through the `optimize_signature` optimizer. Here is an example of this optimization on the `QASignature`:

```
import dspy
from dspy.evaluate import Evaluate
from dspy.evaluate.metrics import answer_exact_match
from dspy.teleprompt.signature_opt_typed import optimize_signature

turbo = dspy.OpenAI(model='gpt-3.5-turbo', max_tokens=4000)
gpt4 = dspy.OpenAI(model='gpt-4', max_tokens=4000)
dspy.settings.configure(lm=turbo)

evaluator = Evaluate(devset=devset, metric=answer_exact_match, num_threads=10, display_progress=True)

result = optimize_signature(
    student=dspy.TypedPredictor(QASignature),
    evaluator=evaluator,
    initial_prompts=6,
    n_iterations=100,
    max_examples=30,
    verbose=True,
    prompt_model=gpt4,
)
```

# Modules

Modules in DSPy

### dspy.ChainOfThought

Constructor

### dspy.ChainOfThoughtWithHint

Constructor

### dspy.MultiChainComparison

Constructor

### dspy.Predict

Constructor

### dspy.ProgramOfThought

Constructor

### dspy.ReAct

Constructor

### dspy.Retrieve

Constructor

# Functional

This documentation provides an overview of the Typed Predictors.

### dspy.TypedPredictor

The TypedPredictor class is a sophisticated module designed for making predictions with strict type validations. It leverages a signature to enforce type…

### dspy.TypedChainOfThought

Overview

### dspy.predictor

Overview

### dspy.cot

Overview

# Optimizers

Teleprompters are powerful optimizers (included in DSPy) that can learn to bootstrap and select effective prompts for the modules of any program. (The "tele-" in the name means "at a distance", i.e., automatic prompting at a distance.) This documentation provides an overview of the DSPy Teleprompters.

### teleprompt.LabeledFewShot

Constructor

### teleprompt.BootstrapFewShot

Constructor

### teleprompt.Ensemble

Constructor

### teleprompt.BootstrapFewShotWithRandomSearch

Constructor

### teleprompt.BootstrapFinetune

Constructor

# Retrieval Model Clients

This documentation provides an overview of the DSPy Retrieval Model Clients.

### dspy.ColBERTv2

Constructor

### retrieve.AzureCognitiveSearch

Constructor

### retrieve.ChromadbRM

Constructor

### retrieve.FaissRM

Constructor

### retrieve.MilvusRM

Constructor

### retrieve.MyScaleRM

Constructor

### retrieve.neo4j_rm

Constructor

### retrieve.RAGatouilleRM

Constructor

### retrieve.SnowflakeRM

Constructor

### retrieve.WatsonDiscoveryRM

Constructor

### retrieve.YouRM

Constructor

# Language Model API Clients

This documentation provides an overview of the DSPy Language Model Clients.

### dspy.OpenAI

Usage

### dspy.AzureOpenAI

Usage

### dspy.Cohere

Usage

### dspy.HFClientTGI

Usage

### dspy.HFClientVLLM

Usage

### dspy.PremAI

PremAI is an all-in-one platform that simplifies the process of creating robust, production-ready applications powered by Generative AI. By streamlining…

### dspy.Anyscale

Usage

### dspy.Together

Usage

### dspy.Databricks

Usage

### dspy.GROQ

Usage

### dspy.Mistral

Usage

### dspy.AWSMistral, dspy.AWSAnthropic, dspy.AWSMeta

Usage

### dspy.Bedrock, dspy.Sagemaker

Usage

### dspy.CloudflareAI

Usage

### dspy.GoogleVertexAI

This guide provides instructions on how to use the GoogleVertexAI class to interact with Google Vertex AI's API for text and code generation.

### dspy.Snowflake

Usage

### dspy.Watsonx

This guide provides instructions on how to use the Watsonx class to interact with IBM Watsonx.ai API for text and code generation.

# dspy.You

Wrapper around You.com's conversational Smart and Research APIs.

# Local Language Model Clients

DSPy supports various methods including `built-in wrappers`, `server integration`, and `external package integration` for model loading. This documentation provides a concise introduction on how to load in models within DSPy extending these capabilities for your specific needs.

### dspy.HFModel

Initialize HFModel within your program with the desired model to load in. Here's an example call:

### dspy.ChatModuleClient

Prerequisites

### dspy.OllamaLocal

Adapted from documentation provided by https://github.com/insop

### dspy.HFClientTGI

Prerequisites

### dspy.TensorRTModel

TensorRT LLM by Nvidia happens to be one of the most optimized inference engines to run open-source Large Language Models locally or in production.

### dspy.HFClientVLLM

Setting up the vLLM Server

# DSPy Assertions

Language models (LMs) have transformed how we interact with machine learning, offering vast capabilities in natural language understanding and generation. However, ensuring these models adhere to domain-specific constraints remains a challenge. Despite the growth of techniques like fine-tuning or "prompt engineering", these approaches are extremely tedious and rely on heavy, manual hand-waving to guide the LMs in adhering to specific constraints. Even DSPy's modularity of programming prompting pipelines lacks mechanisms to effectively and automatically enforce these constraints.

To address this, we introduce DSPy Assertions, a feature within the DSPy framework designed to automate the enforcement of computational constraints on LMs. DSPy Assertions empower developers to guide LMs towards desired outcomes with minimal manual intervention, enhancing the reliability, predictability, and correctness of LM outputs.

## dspy.Assert and dspy.Suggest API

We introduce two primary constructs within DSPy Assertions:

- **`dspy.Assert`**:
  - **Parameters**:
    - `constraint (bool)`: Outcome of Python-defined boolean validation check.
    - `msg (Optional[str])`: User-defined error message providing feedback or correction guidance.
    - `backtrack (Optional[module])`: Specifies target module for retry attempts upon constraint failure. The default backtracking module is the last module before the assertion.
  - **Behavior**: Initiates retry upon failure, dynamically adjusting the pipeline's execution. If failures persist, it halts execution and raises a `dspy.AssertionError`.
- **`dspy.Suggest`**:
  - **Parameters**: Similar to `dspy.Assert`.
  - **Behavior**: Encourages self-refinement through retries without enforcing hard stops. Logs failures after maximum backtracking attempts and continues execution.
- **dspy.Assert vs. Python Assertions**: Unlike conventional Python `assert` statements that terminate the program upon failure, `dspy.Assert` conducts a sophisticated retry mechanism, allowing the pipeline to adjust.

Specifically, when a constraint is not met:

- Backtracking Mechanism: An under-the-hood backtracking is initiated, offering the model a chance to self-refine and proceed, which is done through
- Dynamic Signature Modification: internally modifying your DSPy program's Signature by adding the following fields:
  - Past Output: your model's past output that did not pass the validation_fn
  - Instruction: your user-defined feedback message on what went wrong and what possibly to fix

If the error continues past the `max_backtracking_attempts`, then `dspy.Assert` will halt the pipeline execution, altering you with an `dspy.AssertionError`. This ensures your program doesn't continue executing with "bad" LM behavior and immediately highlights sample failure outputs for user assessment.

- **dspy.Suggest vs. dspy.Assert**: `dspy.Suggest` on the other hand offers a softer approach. It maintains the same retry backtracking as `dspy.Assert` but instead serves as a gentle nudger. If the model outputs cannot pass the model constraints after the `max_backtracking_attempts`, `dspy.Suggest` will log the persistent failure and continue execution of the program on the rest of the data. This ensures the LM pipeline works in a "best-effort" manner without halting execution.
- **`dspy.Suggest`** are best utilized as "helpers" during the evaluation phase, offering guidance and potential corrections without halting the pipeline.
- **`dspy.Assert`** are recommended during the development stage as "checkers" to ensure the LM behaves as expected, providing a

- `dspy.Assert` are recommended during the development stage as "checkers" to ensure the LM behaves as expected, providing a robust mechanism for identifying and addressing errors early in the development cycle.

## Use Case: Including Assertions in DSPy Programs

We start with using an example of a multi-hop QA SimplifiedBaleen pipeline as defined in the intro walkthrough.

```
class SimplifiedBaleen(dspy.Module):
    def __init__(self, passages_per_hop=2, max_hops=2):
        super().__init__()

        self.generate_query = [dspy.ChainOfThought(GenerateSearchQuery) for _ in range(max_hops)]
        self.retrieve = dspy.Retrieve(k=passages_per_hop)
        self.generate_answer = dspy.ChainOfThought(GenerateAnswer)
        self.max_hops = max_hops

    def forward(self, question):
        context = []
        prev_queries = [question]

        for hop in range(self.max_hops):
            query = self.generate_query[hop](context=context, question=question).query
            prev_queries.append(query)
            passages = self.retrieve(query).passages
            context = deduplicate(context + passages)

        pred = self.generate_answer(context=context, question=question)
        pred = dspy.Prediction(context=context, answer=pred.answer)
        return pred

baleen = SimplifiedBaleen()

baleen(question = "Which award did Gary Zukav's first book receive?")
```

To include DSPy Assertions, we simply define our validation functions and declare our assertions following the respective model generation.

For this use case, suppose we want to impose the following constraints:

1. Length - each query should be less than 100 characters
2. Uniqueness - each generated query should differ from previously-generated queries.

We can define these validation checks as boolean functions:

```
#simplistic boolean check for query length
len(query) <= 100

#Python function for validating distinct queries
def validate_query_distinction_local(previous_queries, query):
    """check if query is distinct from previous queries"""
    if previous_queries == []:
        return True
    if dspy.evaluate.answer_exact_match_str(query, previous_queries, frac=0.8):
        return False
    return True
```

We can declare these validation checks through `dspy.Suggest` statements (as we want to test the program in a best-effort demonstration). We want to keep these after the query generation `query = self.generate_query[hop](context=context, question=question).query`.

```
dspy.Suggest(
    len(query) <= 100,
```

```
    "Query should be short and less than 100 characters",
)

dspy.Suggest(
    validate_query_distinction_local(prev_queries, query),
    "Query should be distinct from: "
    + "; ".join(f"{i+1}) {q}" for i, q in enumerate(prev_queries)),
)
```

It is recommended to define a program with assertions separately than your original program if you are doing comparative evaluation for the effect of assertions. If not, feel free to set Assertions away!

Let's take a look at how the SimplifiedBaleen program will look with Assertions included:

```
class SimplifiedBaleenAssertions(dspy.Module):
    def __init__(self, passages_per_hop=2, max_hops=2):
        super().__init__()
        self.generate_query = [dspy.ChainOfThought(GenerateSearchQuery) for _ in range(max_hops)]
        self.retrieve = dspy.Retrieve(k=passages_per_hop)
        self.generate_answer = dspy.ChainOfThought(GenerateAnswer)
        self.max_hops = max_hops

    def forward(self, question):
        context = []
        prev_queries = [question]

        for hop in range(self.max_hops):
            query = self.generate_query[hop](context=context, question=question).query

            dspy.Suggest(
                len(query) <= 100,
                "Query should be short and less than 100 characters",
            )

            dspy.Suggest(
                validate_query_distinction_local(prev_queries, query),
                "Query should be distinct from: "
                + "; ".join(f"{i+1}) {q}" for i, q in enumerate(prev_queries)),
            )

            prev_queries.append(query)
            passages = self.retrieve(query).passages
            context = deduplicate(context + passages)

        if all_queries_distinct(prev_queries):
            self.passed_suggestions += 1

        pred = self.generate_answer(context=context, question=question)
        pred = dspy.Prediction(context=context, answer=pred.answer)
        return pred
```

Now calling programs with DSPy Assertions requires one last step, and that is transforming the program to wrap it with internal assertions backtracking and Retry logic.

```
from dspy.primitives.assertions import assert_transform_module, backtrack_handler

baleen_with_assertions = assert_transform_module(SimplifiedBaleenAssertions(), backtrack_handler)

# backtrack_handler is parameterized over a few settings for the backtracking mechanism
# To change the number of max retry attempts, you can do
baleen_with_assertions_retry_once = assert_transform_module(SimplifiedBaleenAssertions(),
    functools.partial(backtrack_handler, max_backtracks=1))
```

Alternatively, you can also directly call `activate_assertions` on the program with `dspy.Assert/Suggest` statements using the default backtracking mechanism (`max_backtracks=2`):

```
baleen_with_assertions = SimplifiedBaleenAssertions().activate_assertions()
```

Now let's take a look at the internal LM backtracking by inspecting the history of the LM query generations. Here we see that when a query fails to pass the validation check of being less than 100 characters, its internal `GenerateSearchQuery` signature is dynamically modified during the backtracking+Retry process to include the past query and the corresponding user-defined instruction: `"Query should be short and less than 100 characters"`.

```
Write a simple search query that will help answer a complex question.

---

Follow the following format.

Context: may contain relevant facts

Question: ${question}

Reasoning: Let's think step by step in order to ${produce the query}. We ...

Query: ${query}

---

Context:
[1] «Kerry Condon | Kerry Condon (born 4 January 1983) is [...]»
[2] «Corona Riccardo | Corona Riccardo (c. 1878October 15, 1917) was [...]»

Question: Who acted in the shot film The Shore and is also the youngest actress ever to play Ophelia in
a Royal Shakespeare Company production of "Hamlet." ?

Reasoning: Let's think step by step in order to find the answer to this question. First, we need to
identify the actress who played Ophelia in a Royal Shakespeare Company production of "Hamlet." Then, we
need to find out if this actress also acted in the short film "The Shore."

Query: "actress who played Ophelia in Royal Shakespeare Company production of Hamlet" + "actress in
short film The Shore"



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

---

Follow the following format.

Context: may contain relevant facts

Question: ${question}

Past Query: past output with errors

Instructions: Some instructions you must satisfy

Query: ${query}

---

Context:
```

```
[1] «Kerry Condon | Kerry Condon (born 4 January 1983) is an Irish television and film actress, best known for her role as Octavia of the Julii in the HBO/BBC series "Rome," as Stacey Ehrmantraut in AMC's "Better Call Saul" and as the voice of F.R.I.D.A.Y. in various films in the Marvel Cinematic Universe. She is also the youngest actress ever to play Ophelia in a Royal Shakespeare Company production of "Hamlet."»
[2] «Corona Riccardo | Corona Riccardo (c. 1878October 15, 1917) was an Italian born American actress who had a brief Broadway stage career before leaving to become a wife and mother. Born in Naples she came to acting in 1894 playing a Mexican girl in a play at the Empire Theatre. Wilson Barrett engaged her for a role in his play "The Sign of the Cross" which he took on tour of the United States. Riccardo played the role of Ancaria and later played Berenice in the same play. Robert B. Mantell in 1898 who struck by her beauty also cast her in two Shakespeare plays, "Romeo and Juliet" and "Othello". Author Lewis Strang writing in 1899 said Riccardo was the most promising actress in America at the time. Towards the end of 1898 Mantell chose her for another Shakespeare part, Ophelia im Hamlet. Afterwards she was due to join Augustin Daly's Theatre Company but Daly died in 1899. In 1899 she gained her biggest fame by playing Iras in the first stage production of Ben-Hur.»

Question: Who acted in the shot film The Shore and is also the youngest actress ever to play Ophelia in a Royal Shakespeare Company production of "Hamlet." ?

Past Query: "actress who played Ophelia in Royal Shakespeare Company production of Hamlet" + "actress in short film The Shore"

Instructions: Query should be short and less than 100 characters

Query: "actress Ophelia RSC Hamlet" + "actress The Shore"
```

## Assertion-Driven Optimizations

DSPy Assertions work with optimizations that DSPy offers, particularly with `BootstrapFewShotWithRandomSearch`, including the following settings:

- Compilation with Assertions This includes assertion-driven example bootstrapping and counterexample bootstrapping during compilation. The teacher model for bootstrapping few-shot demonstrations can make use of DSPy Assertions to offer robust bootstrapped examples for the student model to learn from during inference. In this setting, the student model does not perform assertion aware optimizations (backtracking and retry) during inference.
- Compilation + Inference with Assertions -This includes assertion-driven optimizations in both compilation and inference. Now the teacher model offers assertion-driven examples but the student can further optimize with assertions of its own during inference time.

```
teleprompter = BootstrapFewShotWithRandomSearch(
    metric=validate_context_and_answer_and_hops,
    max_bootstrapped_demos=max_bootstrapped_demos,
    num_candidate_programs=6,
)

#Compilation with Assertions
compiled_with_assertions_baleen = teleprompter.compile(student = baleen, teacher =
baleen_with_assertions, trainset = trainset, valset = devset)

#Compilation + Inference with Assertions
compiled_baleen_with_assertions = teleprompter.compile(student=baleen_with_assertions, teacher =
baleen_with_assertions, trainset=trainset, valset=devset)
```

# Installation

To install DSPy run:

```
pip install dspy-ai
```

Or open our intro notebook in Google Colab: 


By default, DSPy depends on `openai==0.28`. However, if you install `openai>=1.0`, the library will use that just fine. Both are supported.

For the optional Pinecone, Qdrant, ChromaDB, Marqo, or Milvus retrieval integration(s), include the extra(s) below:

**INSTALLATION COMMAND**

**No Extras** | **Pinecone** | **Qdrant** | **ChromaDB** | **Marqo** | **MongoDB** | **Weaviate** | **Milvus**

```
pip install dspy-ai
```

# Minimal Working Example

In this post, we walk you through a minimal working example using the DSPy library.

We make use of the GSM8K dataset and the OpenAI GPT-3.5-turbo model to simulate prompting tasks within DSPy.

## Setup

Before we jump into the example, let's ensure our environment is properly configured. We'll start by importing the necessary modules and configuring our language model:

```
import dspy
from dspy.datasets.gsm8k import GSM8K, gsm8k_metric

# Set up the LM.
turbo = dspy.OpenAI(model='gpt-3.5-turbo-instruct', max_tokens=250)
dspy.settings.configure(lm=turbo)

# Load math questions from the GSM8K dataset.
gsm8k = GSM8K()
gsm8k_trainset, gsm8k_devset = gsm8k.train[:10], gsm8k.dev[:10]
```

Let's take a look at what `gsm8k_trainset` and `gsm8k_devset` are:

```
print(gsm8k_trainset)
```

The `gsm8k_trainset` and `gsm8k_devset` datasets contain lists of `dspy.Examples`, with each example having `question` and `answer` fields.

## Define the Module

With our environment set up, let's define a custom program that utilizes the `ChainOfThought` module to perform step-by-step reasoning to generate answers:

```
class CoT(dspy.Module):
    def __init__(self):
        super().__init__()
        self.prog = dspy.ChainOfThought("question -> answer")

    def forward(self, question):
        return self.prog(question=question)
```

## Compile and Evaluate the Model

With our simple program in place, let's move on to compiling it with the `BootstrapFewShot` teleprompter:

```
from dspy.teleprompt import BootstrapFewShot

# Set up the optimizer: we want to "bootstrap" (i.e., self-generate) 4-shot examples of our CoT
program.
config = dict(max_bootstrapped_demos=4, max_labeled_demos=4)

# Optimize! Use the `gsm8k_metric` here. In general, the metric is going to tell the optimizer how well
it's doing.
teleprompter = BootstrapFewShot(metric=gsm8k_metric, **config)
optimized_cot = teleprompter.compile(CoT(), trainset=gsm8k_trainset)
```

Note that `BootstrapFewShot` is not an optimizing teleprompter, i.e. it simply creates and validates examples for steps of the pipeline (in this case, chain-of-thought reasoning) but does not optimize the metric. Other teleprompters like `BootstrapFewShotWithRandomSearch` and `MIPRO` will apply direct optimization.

## Evaluate

Now that we have a compiled (optimized) DSPy program, let's move to evaluating its performance on the dev dataset.

```
from dspy.evaluate import Evaluate

# Set up the evaluator, which can be used multiple times.
evaluate = Evaluate(devset=gsm8k_devset, metric=gsm8k_metric, num_threads=4, display_progress=True,
display_table=0)

# Evaluate our `optimized_cot` program.
evaluate(optimized_cot)
```

## Inspect the Model's History

For a deeper understanding of the model's interactions, we can review the most recent generations through inspecting the model's history:

```
turbo.inspect_history(n=1)
```

And there you have it! You've successfully created a working example using the DSPy library.

This example showcases how to set up your environment, define a custom module, compile a model, and rigorously evaluate its performance using the provided dataset and teleprompter configurations.

Feel free to adapt and expand upon this example to suit your specific use case while exploring the extensive capabilities of DSPy.

If you want to try what you just built, run `optimized_cot(question='Your Question Here')`.

---

**Written By:** Herumb Shandilya

# 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 define a baseline 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 with DSPy) 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.

# Language Models

The most powerful features in DSPy revolve around algorithmically optimizing the prompts (or weights) of LMs, especially when you're building programs that use the LMs within a pipeline.

Let's first make sure you can set up your language model. DSPy support clients for many remote and local LMs.

## Setting up the LM client.

You can just call the constructor that connects to the LM. Then, use `dspy.configure` to declare this as the default LM.

For example, to use OpenAI language models, you can do it as follows.

```
gpt3_turbo = dspy.OpenAI(model='gpt-3.5-turbo-1106', max_tokens=300)
dspy.configure(lm=gpt3_turbo)
```

## Directly calling the LM.

You can simply call the LM with a string to give it a raw prompt, i.e. a string.

```
gpt3_turbo("hello! this is a raw prompt to GPT-3.5")
```

**Output:**

```
['Hello! How can I assist you today?']
```

This is almost never the recommended way to interact with LMs in DSPy, but it is allowed.

## Using the LM with DSPy signatures.

You can also use the LM via DSPy `signature` (input/output spec) and `modules`, which we discuss in more depth in the remaining guides.

```
# Define a module (ChainOfThought) and assign it a signature (return an answer, given a question).
qa = dspy.ChainOfThought('question -> answer')

# Run with the default LM configured with `dspy.configure` above.
response = qa(question="How many floors are in the castle David Gregory inherited?")
print(response.answer)
```

**Output:**

```
The castle David Gregory inherited has 7 floors.
```

## Using multiple LMs at once.

The default LM above is GPT-3.5, `gpt3_turbo`. What if I want to run a piece of code with, say, GPT-4 or LLama-2?

Instead of changing the default LM, you can just change it inside a block of code.

**Tip:** Using `dspy.configure` and `dspy.context` is thread-safe!

```
# Run with the default LM configured above, i.e. GPT-3.5
response = qa(question="How many floors are in the castle David Gregory inherited?")
```

```
print('GPT-3.5:', response.answer)

gpt4_turbo = dspy.OpenAI(model='gpt-4-1106-preview', max_tokens=300)

# Run with GPT-4 instead
with dspy.context(lm=gpt4_turbo):
    response = qa(question="How many floors are in the castle David Gregory inherited?")
    print('GPT-4-turbo:', response.answer)
```

**Output:**

```
GPT-3.5: The castle David Gregory inherited has 7 floors.
GPT-4-turbo: The number of floors in the castle David Gregory inherited cannot be determined with the
information provided.
```

## Tips and Tricks.

In DSPy, all LM calls are cached. If you repeat the same call, you will get the same outputs. (If you change the inputs or configurations, you will get new outputs.)

To generate 5 outputs, you can use `n=5` in the module constructor, or pass `config=dict(n=5)` when invoking the module.

```
qa = dspy.ChainOfThought('question -> answer', n=5)

response = qa(question="How many floors are in the castle David Gregory inherited?")
response.completions.answer
```

**Output:**

```
["The specific number of floors in David Gregory's inherited castle is not provided here, so further
research would be needed to determine the answer.",
    'The castle David Gregory inherited has 4 floors.',
    'The castle David Gregory inherited has 5 floors.',
    'David Gregory inherited 10 floors in the castle.',
    'The castle David Gregory inherited has 5 floors.']
```

If you just call `qa(...)` in a loop with the same input, it will always return the same value! That's by design.

To loop and generate one output at a time with the same input, bypass the cache by making sure each request is (slightly) unique, as below.

```
for idx in range(5):
    response = qa(question="How many floors are in the castle David Gregory inherited?",
config=dict(temperature=0.7+0.0001*idx))
    print(f'{idx+1}.', response.answer)
```

**Output:**

```
1. The specific number of floors in David Gregory's inherited castle is not provided here, so further
research would be needed to determine the answer.
2. It is not possible to determine the exact number of floors in the castle David Gregory inherited
without specific information about the castle's layout and history.
3. The castle David Gregory inherited has 5 floors.
4. We need more information to determine the number of floors in the castle David Gregory inherited.
5. The castle David Gregory inherited has a total of 6 floors.
```

## Remote LMs.

These models are managed services. You just need to sign up and obtain an API key. Calling any of the remote LMs below assumes authentication and mirrors the following format for setting up the LM:

```
lm = dspy.{provider_listed_below}(model="your model", model_request_kwargs="...")
```

1. `dspy.OpenAI` for GPT-3.5 and GPT-4.
2. `dspy.Cohere`
3. `dspy.Anyscale` for hosted Llama2 models.
4. `dspy.Together` for hosted various open source models.
5. `dspy.PremAI` for hosted best open source and closed source models.

## Local LMs.

You need to host these models on your own GPU(s). Below, we include pointers for how to do that.

1. `dspy.HFClientTGI`: for HuggingFace models through the Text Generation Inference (TGI) system. Tutorial: How do I install and launch the TGI server?

```
tgi_mistral = dspy.HFClientTGI(model="mistralai/Mistral-7B-Instruct-v0.2", port=8080,
url="http://localhost")
```

2. `dspy.HFClientVLLM`: for HuggingFace models through vLLM. Tutorial: How do I install and launch the vLLM server?

```
vllm_mistral = dspy.HFClientVLLM(model="mistralai/Mistral-7B-Instruct-v0.2", port=8080,
url="http://localhost")
```

3. `dspy.HFModel` (experimental) Tutorial: How do I initialize models using HFModel

```
mistral = dspy.HFModel(model = 'mistralai/Mistral-7B-Instruct-v0.2')
```

4. `dspy.Ollama` (experimental) for open source models through Ollama. Tutorial: How do I install and use Ollama on a local computer?\n",

```
ollama_mistral = dspy.OllamaLocal(model='mistral')
```

5. `dspy.ChatModuleClient` (experimental): How do I install and use MLC?

```
model = 'dist/prebuilt/mlc-chat-Llama-2-7b-chat-hf-q4f16_1'
model_path = 'dist/prebuilt/lib/Llama-2-7b-chat-hf-q4f16_1-cuda.so'

llama = dspy.ChatModuleClient(model=model, model_path=model_path)
```

# Signatures

When we assign tasks to LMs in DSPy, we specify the behavior we need as a Signature.

**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.

You're probably familiar with function signatures, which specify the input and output arguments and their types. DSPy signatures are similar, but the differences are that:

- While typical function signatures just *describe* things, DSPy Signatures *define and control the behavior* of modules.
- The field names matter in DSPy Signatures. You express semantic roles in plain English: a `question` is different from an `answer`, a `sql_query` is different from `python_code`.

## Why should I use a DSPy Signature?

**tl;dr** For modular and clean code, in which LM calls can be optimized into high-quality prompts (or automatic finetunes).

**Long Answer:** Most people coerce LMs to do tasks by hacking long, brittle prompts. Or by collecting/generating data for fine-tuning.

Writing signatures is far more modular, adaptive, and reproducible than hacking at prompts or finetunes. The DSPy compiler will figure out how to build a highly-optimized prompt for your LM (or finetune your small LM) for your signature, on your data, and within your pipeline. In many cases, we found that compiling leads to better prompts than humans write. Not because DSPy optimizers are more creative than humans, but simply because they can try more things and tune the metrics directly.

## Inline DSPy Signatures

Signatures can be defined as a short string, with argument names that define semantic roles for inputs/outputs.

1. Question Answering: `"question -> answer"`
2. Sentiment Classification: `"sentence -> sentiment"`
3. Summarization: `"document -> summary"`

Your signatures can also have multiple input/output fields.

4. Retrieval-Augmented Question Answering: `"context, question -> answer"`
5. Multiple-Choice Question Answering with Reasoning: `"question, choices -> reasoning, selection"`

**Tip:** For fields, any valid variable names work! Field names should be semantically meaningful, but start simple and don't prematurely optimize keywords! Leave that kind of hacking to the DSPy compiler. For example, for summarization, it's probably fine to say `"document -> summary"`, `"text -> gist"`, or `"long_context -> tldr"`.

### Example A: Sentiment Classification

```
sentence = "it's a charming and often affecting journey."  # example from the SST-2 dataset.

classify = dspy.Predict('sentence -> sentiment')
classify(sentence=sentence).sentiment
```

**Output:**

```
'Positive'
```

### Example B: Summarization

### Example B: Summarization

```
# Example from the XSum dataset.
document = """The 21-year-old made seven appearances for the Hammers and netted his only goal for them
in a Europa League qualification round match against Andorran side FC Lustrains last season. Lee had
two loan spells in League One last term, with Blackpool and then Colchester United. He scored twice for
the U's but was unable to save them from relegation. The length of Lee's contract with the promoted
Tykes has not been revealed. Find all the latest football transfers on our dedicated page."""

summarize = dspy.ChainOfThought('document -> summary')
response = summarize(document=document)

print(response.summary)
```

**Output:**

```
The 21-year-old Lee made seven appearances and scored one goal for West Ham last season. He had loan
spells in League One with Blackpool and Colchester United, scoring twice for the latter. He has now
signed a contract with Barnsley, but the length of the contract has not been revealed.
```

Many DSPy modules (except `dspy.Predict`) return auxiliary information by expanding your signature under the hood.

For example, `dspy.ChainOfThought` also adds a `rationale` field that includes the LM's reasoning before it generates the output `summary`.

```
print("Rationale:", response.rationale)
```

**Output:**

```
Rationale: produce the summary. We need to highlight the key points about Lee's performance for West
Ham, his loan spells in League One, and his new contract with Barnsley. We also need to mention that
his contract length has not been disclosed.
```

## Class-based DSPy Signatures

For some advanced tasks, you need more verbose signatures. This is typically to:

1. Clarify something about the nature of the task (expressed below as a `docstring`).
2. Supply hints on the nature of an input field, expressed as a `desc` keyword argument for `dspy.InputField`.
3. Supply constraints on an output field, expressed as a `desc` keyword argument for `dspy.OutputField`.

### Example C: Classification

Notice how the docstring contains (minimal) instructions, which in this case are necessary to have a fully-defined task.

Some optimizers in DSPy, like `COPRO`, can take this simple docstring and then generate more effective variants if needed.

```
class Emotion(dspy.Signature):
    """Classify emotion among sadness, joy, love, anger, fear, surprise."""

    sentence = dspy.InputField()
    sentiment = dspy.OutputField()

sentence = "i started feeling a little vulnerable when the giant spotlight started blinding me"  # from
dair-ai/emotion

classify = dspy.Predict(Emotion)
```

```
classify(sentence=sentence)
```

**Output:**

```
Prediction(
    sentiment='Fear'
)
```

**Tip:** There's nothing wrong with specifying your requests to the LM more clearly. Class-based Signatures help you with that. However, don't prematurely tune the keywords of the your signature by hand. The DSPy optimizers will likely do a better job (and will transfer better across LMs).

### Example D: A metric that evaluates faithfulness to citations

```
class CheckCitationFaithfulness(dspy.Signature):
    """Verify that the text is based on the provided context."""

    context = dspy.InputField(desc="facts here are assumed to be true")
    text = dspy.InputField()
    faithfulness = dspy.OutputField(desc="True/False indicating if text is faithful to context")

context = "The 21-year-old made seven appearances for the Hammers and netted his only goal for them in
a Europa League qualification round match against Andorran side FC Lustrains last season. Lee had two
loan spells in League One last term, with Blackpool and then Colchester United. He scored twice for the
U's but was unable to save them from relegation. The length of Lee's contract with the promoted Tykes
has not been revealed. Find all the latest football transfers on our dedicated page."

text = "Lee scored 3 goals for Colchester United."

faithfulness = dspy.ChainOfThought(CheckCitationFaithfulness)
faithfulness(context=context, text=text)
```

**Output:**

```
Prediction(
    rationale="produce the faithfulness. We know that Lee had two loan spells in League One last term,
with Blackpool and then Colchester United. He scored twice for the U's but was unable to save them from
relegation. However, there is no mention of him scoring three goals for Colchester United.",
    faithfulness='False'
)
```

## Using signatures to build modules & compiling them

While signatures are convenient for prototyping with structured inputs/outputs, that's not the main reason to use them!

You should compose multiple signatures into bigger DSPy modules and compile these modules into optimized prompts and finetunes.

# Modules

A **DSPy module** is a building block for programs that use LMs.

- Each built-in module abstracts a **prompting technique** (like chain of thought or ReAct). Crucially, they are generalized to handle any DSPy Signature.
- A DSPy module has **learnable parameters** (i.e., the little pieces comprising the prompt and the LM weights) and can be invoked (called) to process inputs and return outputs.
- Multiple modules can be composed into bigger modules (programs). DSPy modules are inspired directly by NN modules in PyTorch, but applied to LM programs.

## How do I use a built-in module, like `dspy.Predict` or `dspy.ChainOfThought`?

Let's start with the most fundamental module, `dspy.Predict`. Internally, all other DSPy modules are just built using `dspy.Predict`.

We'll assume you are already at least a little familiar with DSPy signatures, which are declarative specs for defining the behavior of any module we use in DSPy.

To use a module, we first **declare** it by giving it a signature. Then we **call** the module with the input arguments, and extract the output fields!

```
sentence = "it's a charming and often affecting journey."  # example from the SST-2 dataset.

# 1) Declare with a signature.
classify = dspy.Predict('sentence -> sentiment')

# 2) Call with input argument(s).
response = classify(sentence=sentence)

# 3) Access the output.
print(response.sentiment)
```

**Output:**

```
Positive
```

When we declare a module, we can pass configuration keys to it.

Below, we'll pass `n=5` to request five completions. We can also pass `temperature` or `max_len`, etc.

Let's use `dspy.ChainOfThought`. In many cases, simply swapping `dspy.ChainOfThought` in place of `dspy.Predict` improves quality.

```
question = "What's something great about the ColBERT retrieval model?"

# 1) Declare with a signature, and pass some config.
classify = dspy.ChainOfThought('question -> answer', n=5)

# 2) Call with input argument.
response = classify(question=question)

# 3) Access the outputs.
response.completions.answer
```

**Output:**

```
['One great thing about the ColBERT retrieval model is its superior efficiency and effectiveness
compared to other models.',
 'Its ability to efficiently retrieve relevant information from large document collections.',
 'One great thing about the ColBERT retrieval model is its superior performance compared to other
models and its efficient use of pre-trained language models.',
 'One great thing about the ColBERT retrieval model is its superior efficiency and accuracy compared to
other models.',
 'One great thing about the ColBERT retrieval model is its ability to incorporate user feedback and
support complex queries.']
```

Let's discuss the output object here.

The `dspy.ChainOfThought` module will generally inject a `rationale` before the output field(s) of your signature.

Let's inspect the (first) rationale and answer!

```
print(f"Rationale: {response.rationale}")
print(f"Answer: {response.answer}")
```

**Output:**

```
Rationale: produce the answer. We can consider the fact that ColBERT has shown to outperform other
state-of-the-art retrieval models in terms of efficiency and effectiveness. It uses contextualized
embeddings and performs document retrieval in a way that is both accurate and scalable.
Answer: One great thing about the ColBERT retrieval model is its superior efficiency and effectiveness
compared to other models.
```

This is accessible whether we request one or many completions.

We can also access the different completions as a list of `Prediction`s or as several lists, one for each field.

```
response.completions[3].rationale == response.completions.rationale[3]
```

**Output:**

```
True
```

## What other DSPy modules are there? How can I use them?

The others are very similar. They mainly change the internal behavior with which your signature is implemented!

1. **`dspy.Predict`**: Basic predictor. Does not modify the signature. Handles the key forms of learning (i.e., storing the instructions and demonstrations and updates to the LM).
2. **`dspy.ChainOfThought`**: Teaches the LM to think step-by-step before committing to the signature's response.
3. **`dspy.ProgramOfThought`**: Teaches the LM to output code, whose execution results will dictate the response.
4. **`dspy.ReAct`**: An agent that can use tools to implement the given signature.
5. **`dspy.MultiChainComparison`**: Can compare multiple outputs from `ChainOfThought` to produce a final prediction.

We also have some function-style modules:

We also have some function-style modules:

6. `dspy.majority`: Can do basic voting to return the most popular response from a set of predictions.

Check out further examples in each module's respective guide.

## How do I compose multiple modules into a bigger program?

DSPy is just Python code that uses modules in any control flow you like. (There's some magic internally at `compile` time to trace your LM calls.)

What this means is that, you can just call the modules freely. No weird abstractions for chaining calls.

This is basically PyTorch's design approach for define-by-run / dynamic computation graphs. Refer to the intro tutorials for examples.

# Data

DSPy is a machine learning framework, so working in it involves training sets, development sets, and test sets.

For each example in your data, we distinguish typically between three types of values: the inputs, the intermediate labels, and the final label. You can use DSPy effectively without any intermediate or final labels, but you will need at least a few example inputs.

## How much data do I need and how do I collect data for my task?

Concretely, you can use DSPy optimizers usefully with as few as 10 example inputs, 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.

## DSPy `Example` objects

The core data type for data in DSPy is `Example`. You will use **Examples** to represent items in your training set and test set.

DSPy **Examples** are similar to Python `dict`s but have a few useful utilities. Your DSPy modules will return values of the type `Prediction`, which is a special sub-class of `Example`.

When you use DSPy, you will do a lot of evaluation and optimization runs. Your individual datapoints will be of type `Example`:

```
qa_pair = dspy.Example(question="This is a question?", answer="This is an answer.")

print(qa_pair)
print(qa_pair.question)
print(qa_pair.answer)
```

**Output:**

```
Example({'question': 'This is a question?', 'answer': 'This is an answer.'}) (input_keys=None)
This is a question?
This is an answer.
```

Examples can have any field keys and any value types, though usually values are strings.

```
object = Example(field1=value1, field2=value2, field3=value3, ...)
```

You can now express your training set for example as:

```
trainset = [dspy.Example(report="LONG REPORT 1", summary="short summary 1"), ...]
```

### Specifying Input Keys

In traditional ML, there are separated "inputs" and "labels".

In DSPy, the `Example` objects have a `with_inputs()` method, which can mark specific fields as inputs. (The rest are just metadata or labels.)

```
# Single Input.
print(qa_pair.with_inputs("question"))

# Multiple Inputs; be careful about marking your labels as inputs unless you mean it.
print(qa_pair.with_inputs("question", "answer"))
```

Values can be accessed using the `.`(dot) operator. You can access the value of key `name` in defined object `Example(name="John Doe", job="sleep")` through `object.name`.

To access or exclude certain keys, use `inputs()` and `labels()` methods to return new Example objects containing only input or non-input keys, respectively.

```
article_summary = dspy.Example(article= "This is an article.", summary= "This is a
summary.").with_inputs("article")

input_key_only = article_summary.inputs()
non_input_key_only = article_summary.labels()

print("Example object with Input fields only:", input_key_only)
print("Example object with Non-Input fields only:", non_input_key_only)
```

**Output**

```
Example object with Input fields only: Example({'article': 'This is an article.'}) (input_keys=None)
Example object with Non-Input fields only: Example({'summary': 'This is a summary.'}) (input_keys=None)
```

# Metrics

DSPy is a machine learning framework, so you must think about your **automatic metrics** for evaluation (to track your progress) and optimization (so DSPy can make your programs more effective).

## What is a metric and how do I define a metric for my task?

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. What makes outputs from your system good or bad?

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.

## Simple metrics

A DSPy metric is just a function in Python that takes `example` (e.g., from your training or dev set) and the output `pred` from your DSPy program, and outputs a `float` (or `int` or `bool`) score.

Your metric should also accept an optional third argument called `trace`. You can ignore this for a moment, but it will enable some powerful tricks if you want to use your metric for optimization.

Here's a simple example of a metric that's comparing `example.answer` and `pred.answer`. This particular metric will return a `bool`.

```
def validate_answer(example, pred, trace=None):
    return example.answer.lower() == pred.answer.lower()
```

Some people find these utilities (built-in) convenient:

- `dspy.evaluate.metrics.answer_exact_match`
- `dspy.evaluate.metrics.answer_passage_match`

Your metrics could be more complex, e.g. check for multiple properties. The metric below will return a `float` if `trace is None` (i.e., if it's used for evaluation or optimization), and will return a `bool` otherwise (i.e., if it's used to bootstrap demonstrations).

```
def validate_context_and_answer(example, pred, trace=None):
    # check the gold label and the predicted answer are the same
    answer_match = example.answer.lower() == pred.answer.lower()

    # check the predicted answer comes from one of the retrieved contexts
    context_match = any((pred.answer.lower() in c) for c in pred.context)

    if trace is None: # if we're doing evaluation or optimization
        return (answer_match + context_match) / 2.0
    else: # if we're doing bootstrapping, i.e. self-generating good demonstrations of each step
        return answer_match and context_match
```

Defining a good metric is an iterative process, so doing some initial evaluations and looking at your data and your outputs are key.

## Evaluation

Once you have a metric, you can run evaluations in a simple Python loop.

```
scores = []
for x in devset:
    pred = program(**x.inputs())
    score = metric(x, pred)
    scores.append(score)
```

If you need some utilities, you can also use the built-in `Evaluate` utility. It can help with things like parallel evaluation (multiple threads) or showing you a sample of inputs/outputs and the metric scores.

```
from dspy.evaluate import Evaluate

# Set up the evaluator, which can be re-used in your code.
evaluator = Evaluate(devset=YOUR_DEVSET, num_threads=1, display_progress=True, display_table=5)

# Launch evaluation.
evaluator(YOUR_PROGRAM, metric=YOUR_METRIC)
```

## Intermediate: Using AI feedback for your metric

For most applications, your system will output long-form outputs, so your metric should check multiple dimensions of the output using AI feedback from LMs.

This simple signature could come in handy.

```
# Define the signature for automatic assessments.
class Assess(dspy.Signature):
    """Assess the quality of a tweet along the specified dimension."""

    assessed_text = dspy.InputField()
    assessment_question = dspy.InputField()
    assessment_answer = dspy.OutputField(desc="Yes or No")
```

For example, below is a simple metric that uses GPT-4-turbo to check if a generated tweet (1) answers a given question correctly and (2) whether it's also engaging. We also check that (3) `len(tweet) <= 280` characters.

```
gpt4T = dspy.OpenAI(model='gpt-4-1106-preview', max_tokens=1000, model_type='chat')

def metric(gold, pred, trace=None):
    question, answer, tweet = gold.question, gold.answer, pred.output

    engaging = "Does the assessed text make for a self-contained, engaging tweet?"
    correct = f"The text should answer `{question}` with `{answer}`. Does the assessed text contain
this answer?"

    with dspy.context(lm=gpt4T):
        correct =  dspy.Predict(Assess)(assessed_text=tweet, assessment_question=correct)
        engaging = dspy.Predict(Assess)(assessed_text=tweet, assessment_question=engaging)

    correct, engaging = [m.assessment_answer.lower() == 'yes' for m in [correct, engaging]]
    score = (correct + engaging) if correct and (len(tweet) <= 280) else 0

    if trace is not None: return score >= 2
    return score / 2.0
```

When compiling, `trace is not None`, and we want to be strict about judging things, so we will only return `True` if `score >= 2`.

Otherwise, we return a score out of 1.0 (i.e., `score / 2.0`).

## Advanced: Using a DSPy program as your metric

## Advanced: Using a DSPy program as your metric

If your metric is itself a DSPy program, 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.

## Advanced: Accessing the `trace`

When your metric is used during evaluation runs, DSPy will not try to track the steps of your program.

But during compiling (optimization), DSPy will trace your LM calls. The trace will contain inputs/outputs to each DSPy predictor and you can leverage that to validate intermediate steps for optimization.

```
def validate_hops(example, pred, trace=None):
    hops = [example.question] + [outputs.query for *_, outputs in trace if 'query' in outputs]

    if max([len(h) for h in hops]) > 100: return False
    if any(dspy.evaluate.answer_exact_match_str(hops[idx], hops[:idx], frac=0.8) for idx in range(2,
len(hops))): return False

    return True
```

# Optimizers (formerly Teleprompters)

A **DSPy optimizer** is an algorithm that can tune the parameters of a DSPy program (i.e., the prompts and/or the LM weights) to maximize the metrics you specify, like accuracy.

There are many built-in optimizers in DSPy, which apply vastly different strategies. A typical DSPy optimizer takes three things:

- Your **DSPy program**. This may be a single module (e.g., `dspy.Predict`) or a complex multi-module program.
- Your **metric**. This is a function that evaluates the output of your program, and assigns it a score (higher is better).
- A few **training inputs**. This may be very small (i.e., only 5 or 10 examples) and incomplete (only inputs to your program, without any labels).

If you happen to have a lot of data, DSPy can leverage that. But you can start small and get strong results.

**Note:** Formerly called **DSPy Teleprompters**. We are making an official name update, which will be reflected throughout the library and documentation.

## What does a DSPy Optimizer tune? How does it tune them?

Traditional deep neural networks (DNNs) can be optimized with gradient descent, given a loss function and some training data.

DSPy programs consist of multiple calls to LMs, stacked together as [DSPy modules]. Each DSPy module has internal parameters of three kinds: (1) the LM weights, (2) the instructions, and (3) demonstrations of the input/output behavior.

Given a metric, DSPy can optimize all of these three with multi-stage optimization algorithms. These can combine gradient descent (for LM weights) and discrete LM-driven optimization, i.e. for crafting/updating instructions and for creating/validating demonstrations. DSPy Demonstrations are like few-shot examples, but they're far more powerful. They can be created from scratch, given your program, and their creation and selection can be optimized in many effective ways.

In many cases, we found that compiling leads to better prompts than human writing. Not because DSPy optimizers are more creative than humans, but simply because they can try more things, much more systematically, and tune the metrics directly.

## What DSPy Optimizers are currently available?

Subclasses of Teleprompter

All of these can be accessed via `from dspy.teleprompt import *`.

**Automatic Few-Shot Learning**

These optimizers extend the signature by automatically generating and including **optimized** examples within the prompt sent to the model, implementing few-shot learning.

1. **`LabeledFewShot`**: Simply constructs few-shot examples (demos) from provided labeled input and output data points. Requires `k` (number of examples for the prompt) and `trainset` to randomly select `k` examples from.

2. **`BootstrapFewShot`**: Uses a `teacher` module (which defaults to your program) to generate complete demonstrations for every stage of your program, along with labeled examples in `trainset`. Parameters include `max_labeled_demos` (the number of demonstrations randomly selected from the `trainset`) and `max_bootstrapped_demos` (the number of additional examples generated by the `teacher`). The bootstrapping process employs the metric to validate demonstrations, including only those that pass the metric in the "compiled" prompt. Advanced: Supports using a `teacher` program that is a *different* DSPy program that has compatible structure, for harder tasks.

3. **`BootstrapFewShotWithRandomSearch`**: Applies `BootstrapFewShot` several times with random search over generated demonstrations, and selects the best program over the optimization. Parameters mirror those of `BootstrapFewShot`, with the addition of `num_candidate_programs`, which specifies the number of random programs evaluated over the optimization, including candidates of the uncompiled program, `LabeledFewShot` optimized program, `BootstrapFewShot` compiled program

with unshuffled examples and `num_candidate_programs` or `BootstrapFewShot` compiled programs with randomized example sets.

4. **`BootstrapFewShotWithOptuna`**: Applies `BootstrapFewShot` with Optuna optimization across demonstration sets, running trials to maximize evaluation metrics and selecting the best demonstrations.

5. **`KNNFewShot`**. Selects demonstrations through k-Nearest Neighbors algorithm to pick a diverse set of examples from different clusters. Vectorizes the examples, and then clusters them, using cluster centers with `BootstrapFewShot` for bootstrapping/selection process. This will be useful when there's a lot of data over random spaces: using KNN helps optimize the `trainset` for `BootstrapFewShot`. See this notebook for an example.

**Automatic Instruction Optimization**
These optimizers produce optimal instructions for the prompt and, in the case of MIPRO also optimize the set of few-shot demonstrations.

6. **`COPRO`**: Generates and refines new instructions for each step, and optimizes them with coordinate ascent (hill-climbing using the metric function and the `trainset`). Parameters include `depth` which is the number of iterations of prompt improvement the optimizer runs over.

7. **`MIPRO`**: Generates instructions *and* few-shot examples in each step. The instruction generation is data-aware and demonstration-aware. Uses Bayesian Optimization to effectively search over the space of generation instructions/demonstrations across your modules.

**Automatic Finetuning**
This optimizer is used to fine-tune the underlying LLM(s).

6. **`BootstrapFinetune`**: Distills a prompt-based DSPy program into weight updates (for smaller LMs). The output is a DSPy program that has the same steps, but where each step is conducted by a finetuned model instead of a prompted LM.

**Program Transformations**
8. **`Ensemble`**: Ensembles a set of DSPy programs and either uses the full set or randomly samples a subset into a single program.

## Which optimizer should I use?
As a rule of thumb, if you don't know where to start, use `BootstrapFewShotWithRandomSearch`.

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`.

## How do I use an optimizer?
They all share this general interface, with some differences in the keyword arguments (hyperparameters).

Let's see this with the most common one, `BootstrapFewShotWithRandomSearch`.

```
from dspy.teleprompt import BootstrapFewShotWithRandomSearch

# Set up the optimizer: we want to "bootstrap" (i.e., self-generate) 8-shot examples of your program's steps.
# The optimizer will repeat this 10 times (plus some initial attempts) before selecting its best attempt on the devset.
config = dict(max_bootstrapped_demos=4, max_labeled_demos=4, num_candidate_programs=10, num_threads=4)

teleprompter = BootstrapFewShotWithRandomSearch(metric=YOUR_METRIC_HERE, **config)
optimized_program = teleprompter.compile(YOUR_PROGRAM_HERE, trainset=YOUR_TRAINSET_HERE)
```

## Saving and loading optimizer output

After running a program through an optimizer, it's useful to also save it. At a later point, a program can be loaded from a file and used for inference. For this, the `load` and `save` methods can be used.

### Saving a program

```
optimized_program.save(YOUR_SAVE_PATH)
```

The resulting file is in plain-text JSON format. It contains all the parameters and steps in the source program. You can always read it and see what the optimizer generated.

### Loading a program

To load a program from a file, you can instantiate an object from that class and then call the load method on it.

```
loaded_program = YOUR_PROGRAM_CLASS()
loaded_program.load(path=YOUR_SAVE_PATH)
```

# DSPy Assertions

## Introduction

Language models (LMs) have transformed how we interact with machine learning, offering vast capabilities in natural language understanding and generation. However, ensuring these models adhere to domain-specific constraints remains a challenge. Despite the growth of techniques like fine-tuning or “prompt engineering”, these approaches are extremely tedious and rely on heavy, manual hand-waving to guide the LMs in adhering to specific constraints. Even DSPy's modularity of programming prompting pipelines lacks mechanisms to effectively and automatically enforce these constraints.

To address this, we introduce DSPy Assertions, a feature within the DSPy framework designed to automate the enforcement of computational constraints on LMs. DSPy Assertions empower developers to guide LMs towards desired outcomes with minimal manual intervention, enhancing the reliability, predictability, and correctness of LM outputs.

### dspy.Assert and dspy.Suggest API

We introduce two primary constructs within DSPy Assertions:

- **`dspy.Assert`**:
  - **Parameters**:
    - `constraint (bool)`: Outcome of Python-defined boolean validation check.
    - `msg (Optional[str])`: User-defined error message providing feedback or correction guidance.
    - `backtrack (Optional[module])`: Specifies target module for retry attempts upon constraint failure. The default backtracking module is the last module before the assertion.
  - **Behavior**: Initiates retry upon failure, dynamically adjusting the pipeline's execution. If failures persist, it halts execution and raises a `dspy.AssertionError`.
- **`dspy.Suggest`**:
  - **Parameters**: Similar to `dspy.Assert`.
  - **Behavior**: Encourages self-refinement through retries without enforcing hard stops. Logs failures after maximum backtracking attempts and continues execution.
- **dspy.Assert vs. Python Assertions**: Unlike conventional Python `assert` statements that terminate the program upon failure, `dspy.Assert` conducts a sophisticated retry mechanism, allowing the pipeline to adjust.

Specifically, when a constraint is not met:

- Backtracking Mechanism: An under-the-hood backtracking is initiated, offering the model a chance to self-refine and proceed, which is done through
- Dynamic Signature Modification: internally modifying your DSPy program’s Signature by adding the following fields:
  - Past Output: your model's past output that did not pass the validation_fn
  - Instruction: your user-defined feedback message on what went wrong and what possibly to fix

If the error continues past the `max_backtracking_attempts`, then `dspy.Assert` will halt the pipeline execution, alerting you with an `dspy.AssertionError`. This ensures your program doesn't continue executing with “bad” LM behavior and immediately highlights sample failure outputs for user assessment.

- **dspy.Suggest vs. dspy.Assert**: `dspy.Suggest` on the other hand offers a softer approach. It maintains the same retry backtracking as `dspy.Assert` but instead serves as a gentle nudger. If the model outputs cannot pass the model constraints after the `max_backtracking_attempts`, `dspy.Suggest` will log the persistent failure and continue execution of the program on the rest of the data. This ensures the LM pipeline works in a "best-effort" manner without halting execution.

- **`dspy.Suggest`** are best utilized as "helpers" during the evaluation phase, offering guidance and potential corrections without halting the pipeline.

- `dspy.Assert` are recommended during the development stage as "checkers" to ensure the LM behaves as expected, providing a robust mechanism for identifying and addressing errors early in the development cycle.

## Use Case: Including Assertions in DSPy Programs

We start with using an example of a multi-hop QA SimplifiedBaleen pipeline as defined in the intro walkthrough.

```
class SimplifiedBaleen(dspy.Module):
    def __init__(self, passages_per_hop=2, max_hops=2):
        super().__init__()

        self.generate_query = [dspy.ChainOfThought(GenerateSearchQuery) for _ in range(max_hops)]
        self.retrieve = dspy.Retrieve(k=passages_per_hop)
        self.generate_answer = dspy.ChainOfThought(GenerateAnswer)
        self.max_hops = max_hops

    def forward(self, question):
        context = []
        prev_queries = [question]

        for hop in range(self.max_hops):
            query = self.generate_query[hop](context=context, question=question).query
            prev_queries.append(query)
            passages = self.retrieve(query).passages
            context = deduplicate(context + passages)

        pred = self.generate_answer(context=context, question=question)
        pred = dspy.Prediction(context=context, answer=pred.answer)
        return pred

baleen = SimplifiedBaleen()

baleen(question = "Which award did Gary Zukav's first book receive?")
```

To include DSPy Assertions, we simply define our validation functions and declare our assertions following the respective model generation.

For this use case, suppose we want to impose the following constraints:

1. Length - each query should be less than 100 characters
2. Uniqueness - each generated query should differ from previously-generated queries.

We can define these validation checks as boolean functions:

```
#simplistic boolean check for query length
len(query) <= 100

#Python function for validating distinct queries
def validate_query_distinction_local(previous_queries, query):
    """check if query is distinct from previous queries"""
    if previous_queries == []:
        return True
    if dspy.evaluate.answer_exact_match_str(query, previous_queries, frac=0.8):
        return False
    return True
```

We can declare these validation checks through `dspy.Suggest` statements (as we want to test the program in a best-effort demonstration). We want to keep these after the query generation `query = self.generate_query[hop](context=context, question=question).query`.

```
dspy.Suggest(
    len(query) <= 100,
    "Query should be short and less than 100 characters",
)

dspy.Suggest(
    validate_query_distinction_local(prev_queries, query),
    "Query should be distinct from: "
    + "; ".join(f"{i+1}) {q}" for i, q in enumerate(prev_queries)),
)
```

It is recommended to define a program with assertions separately than your original program if you are doing comparative evaluation for the effect of assertions. If not, feel free to set Assertions away!

Let's take a look at how the SimplifiedBaleen program will look with Assertions included:

```
class SimplifiedBaleenAssertions(dspy.Module):
    def __init__(self, passages_per_hop=2, max_hops=2):
        super().__init__()
        self.generate_query = [dspy.ChainOfThought(GenerateSearchQuery) for _ in range(max_hops)]
        self.retrieve = dspy.Retrieve(k=passages_per_hop)
        self.generate_answer = dspy.ChainOfThought(GenerateAnswer)
        self.max_hops = max_hops

    def forward(self, question):
        context = []
        prev_queries = [question]

        for hop in range(self.max_hops):
            query = self.generate_query[hop](context=context, question=question).query

            dspy.Suggest(
                len(query) <= 100,
                "Query should be short and less than 100 characters",
            )

            dspy.Suggest(
                validate_query_distinction_local(prev_queries, query),
                "Query should be distinct from: "
                + "; ".join(f"{i+1}) {q}" for i, q in enumerate(prev_queries)),
            )

            prev_queries.append(query)
            passages = self.retrieve(query).passages
            context = deduplicate(context + passages)

        if all_queries_distinct(prev_queries):
            self.passed_suggestions += 1

        pred = self.generate_answer(context=context, question=question)
        pred = dspy.Prediction(context=context, answer=pred.answer)
        return pred
```

Now calling programs with DSPy Assertions requires one last step, and that is transforming the program to wrap it with internal assertions backtracking and Retry logic.

```
from dspy.primitives.assertions import assert_transform_module, backtrack_handler

baleen_with_assertions = assert_transform_module(SimplifiedBaleenAssertions(), backtrack_handler)

# backtrack_handler is parameterized over a few settings for the backtracking mechanism
# To change the number of max retry attempts, you can do
baleen_with_assertions_retry_once = assert_transform_module(SimplifiedBaleenAssertions()
```

```
baleen_with_assertions_retry_once = assert_transform_module(SimplifiedBaleenAssertions(),
    functools.partial(backtrack_handler, max_backtracks=1))
```

Alternatively, you can also directly call `activate_assertions` on the program with `dspy.Assert/Suggest` statements using the default backtracking mechanism (`max_backtracks=2`):

```
baleen_with_assertions = SimplifiedBaleenAssertions().activate_assertions()
```

Now let's take a look at the internal LM backtracking by inspecting the history of the LM query generations. Here we see that when a query fails to pass the validation check of being less than 100 characters, its internal `GenerateSearchQuery` signature is dynamically modified during the backtracking+Retry process to include the past query and the corresponding user-defined instruction: `"Query should be short and less than 100 characters"`.

```
Write a simple search query that will help answer a complex question.

---

Follow the following format.

Context: may contain relevant facts

Question: ${question}

Reasoning: Let's think step by step in order to ${produce the query}. We ...

Query: ${query}

---

Context:
[1] «Kerry Condon | Kerry Condon (born 4 January 1983) is [...]»
[2] «Corona Riccardo | Corona Riccardo (c. 1878October 15, 1917) was [...]»

Question: Who acted in the shot film The Shore and is also the youngest actress ever to play Ophelia in
a Royal Shakespeare Company production of "Hamlet." ?

Reasoning: Let's think step by step in order to find the answer to this question. First, we need to
identify the actress who played Ophelia in a Royal Shakespeare Company production of "Hamlet." Then, we
need to find out if this actress also acted in the short film "The Shore."

Query: "actress who played Ophelia in Royal Shakespeare Company production of Hamlet" + "actress in
short film The Shore"



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

---

Follow the following format.

Context: may contain relevant facts

Question: ${question}

Past Query: past output with errors

Instructions: Some instructions you must satisfy

Query: ${query}

---
```

```
Context:
[1] «Kerry Condon | Kerry Condon (born 4 January 1983) is an Irish television and film actress, best known for her role as Octavia of the Julii in the HBO/BBC series "Rome," as Stacey Ehrmantraut in AMC's "Better Call Saul" and as the voice of F.R.I.D.A.Y. in various films in the Marvel Cinematic Universe. She is also the youngest actress ever to play Ophelia in a Royal Shakespeare Company production of "Hamlet."»
[2] «Corona Riccardo | Corona Riccardo (c. 1878October 15, 1917) was an Italian born American actress who had a brief Broadway stage career before leaving to become a wife and mother. Born in Naples she came to acting in 1894 playing a Mexican girl in a play at the Empire Theatre. Wilson Barrett engaged her for a role in his play "The Sign of the Cross" which he took on tour of the United States. Riccardo played the role of Ancaria and later played Berenice in the same play. Robert B. Mantell in 1898 who struck by her beauty also cast her in two Shakespeare plays, "Romeo and Juliet" and "Othello". Author Lewis Strang writing in 1899 said Riccardo was the most promising actress in America at the time. Towards the end of 1898 Mantell chose her for another Shakespeare part, Ophelia im Hamlet. Afterwards she was due to join Augustin Daly's Theatre Company but Daly died in 1899. In 1899 she gained her biggest fame by playing Iras in the first stage production of Ben-Hur.»

Question: Who acted in the shot film The Shore and is also the youngest actress ever to play Ophelia in a Royal Shakespeare Company production of "Hamlet." ?

Past Query: "actress who played Ophelia in Royal Shakespeare Company production of Hamlet" + "actress in short film The Shore"

Instructions: Query should be short and less than 100 characters

Query: "actress Ophelia RSC Hamlet" + "actress The Shore"
```

## Assertion-Driven Optimizations

DSPy Assertions work with optimizations that DSPy offers, particularly with `BootstrapFewShotWithRandomSearch`, including the following settings:

- Compilation with Assertions This includes assertion-driven example bootstrapping and counterexample bootstrapping during compilation. The teacher model for bootstrapping few-shot demonstrations can make use of DSPy Assertions to offer robust bootstrapped examples for the student model to learn from during inference. In this setting, the student model does not perform assertion aware optimizations (backtracking and retry) during inference.
- Compilation + Inference with Assertions -This includes assertion-driven optimizations in both compilation and inference. Now the teacher model offers assertion-driven examples but the student can further optimize with assertions of its own during inference time.

```
teleprompter = BootstrapFewShotWithRandomSearch(
    metric=validate_context_and_answer_and_hops,
    max_bootstrapped_demos=max_bootstrapped_demos,
    num_candidate_programs=6,
)

#Compilation with Assertions
compiled_with_assertions_baleen = teleprompter.compile(student = baleen, teacher =
baleen_with_assertions, trainset = trainset, valset = devset)

#Compilation + Inference with Assertions
compiled_baleen_with_assertions = teleprompter.compile(student=baleen_with_assertions, teacher =
baleen_with_assertions, trainset=trainset, valset=devset)
```

# Understanding Signatures

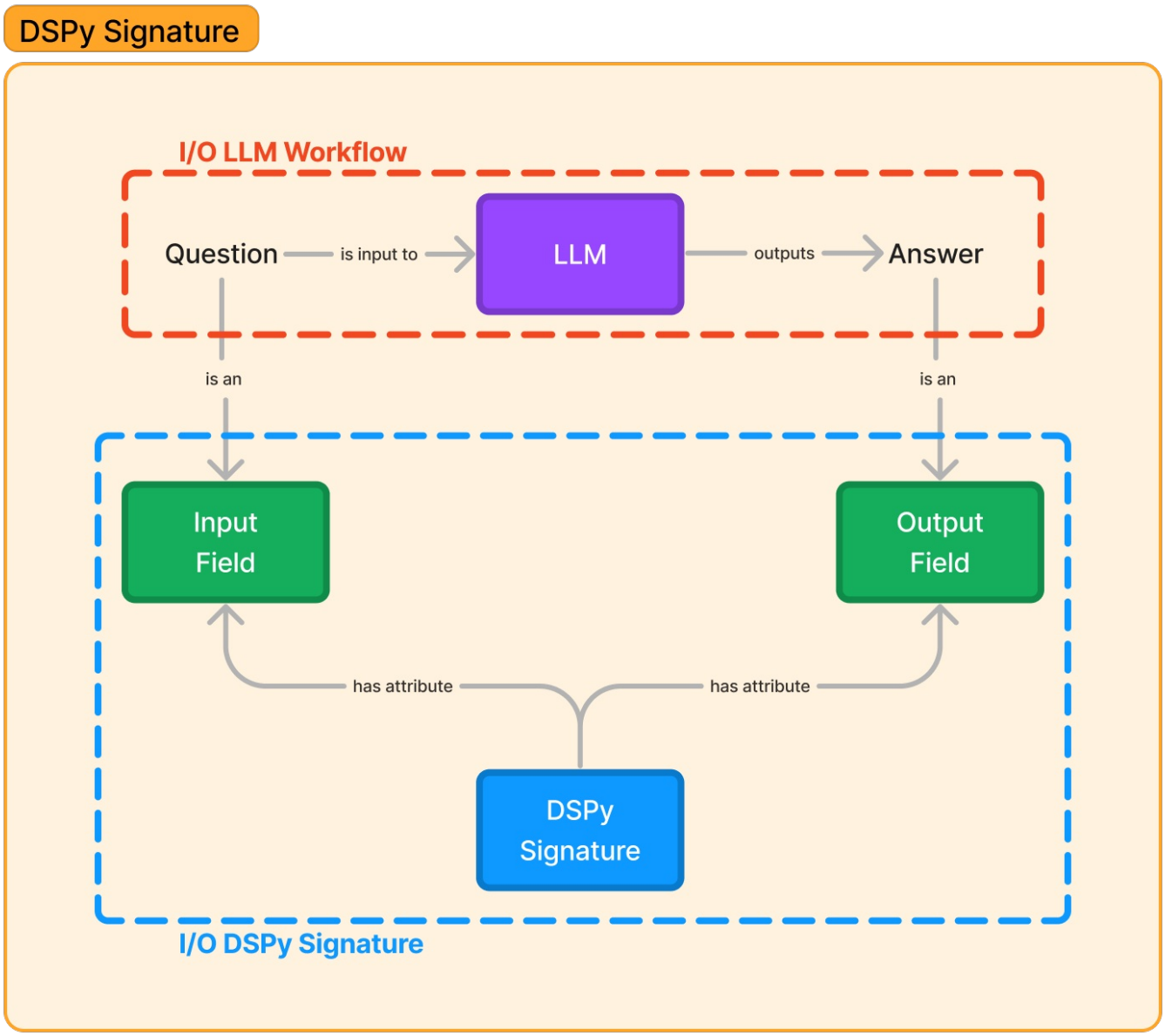
A DSPy Signature is the most basic form of task description which simply requires inputs and outputs and optionally, a small description about them and the task too.

There are 2 ways to define a Signature: **Inline** and **Class-Based**. But before diving into creating signatures, let's understand what a signature is and why we need it.

## What is a Signature?

In the typical LLM pipeline, you'll have two key components at work i.e. an LLM and a prompt. In DSPy, we have an LLM configured at the beginning of any DSPy script via the LM(Language Model - which is shown in the next blog) and a prompt defined via **Signatures**.

A Signature is usually composed of 2 essential components: **Input Fields** and **Output Fields**. You can optionally pass an instruction defining more robust requirements of your task. An **Input Field** is an attribute of Signature that defines an input to the prompt and an **Output Field** is an attribute of Signature that defines an output of the prompt received from an LLM call. Let's understand this by an example.




Let's think of a basic Question-Answer task where the question serves as an input to the LLM from which you receive an answer

response. We directly map this in DSPy as the question serves as the Signature's **Input Field** and the answer as the Signature's **Output Field** .

Now that we understand the components of a Signature, let's see how we can declare a signature and what a prompt for that signature looks like.

## Inline Method

DSPy offers an intuitive, simple approach for defining tasks: simply state the inputs and outputs to convey the task in its simplest form. For example, if your input is **question** and output is **answer**, it should be clear that the task is a Question-Answer task. If your inputs are **context** and **question** and outputs are **answer** and **reason**, this should imply some form of Chain-Of-Thought prompting, potentially within a RAG pipeline.

Inspired by this simplicity, DSPy Signatures mirrors an Einops-like abstract manner:

```
input_field_1,input_field_2,input_field_3...->output_field_1,output_field_2,output_field_3...
```

**Input Fields** of the Signature are declared on the left side of `->` with the **Output Fields** on the right side. So let's go ahead and define DSPy signatures for the QA and RAG tasks:

```
QA Task: question->answer
RAG Task: context,question->answer,rationale
```

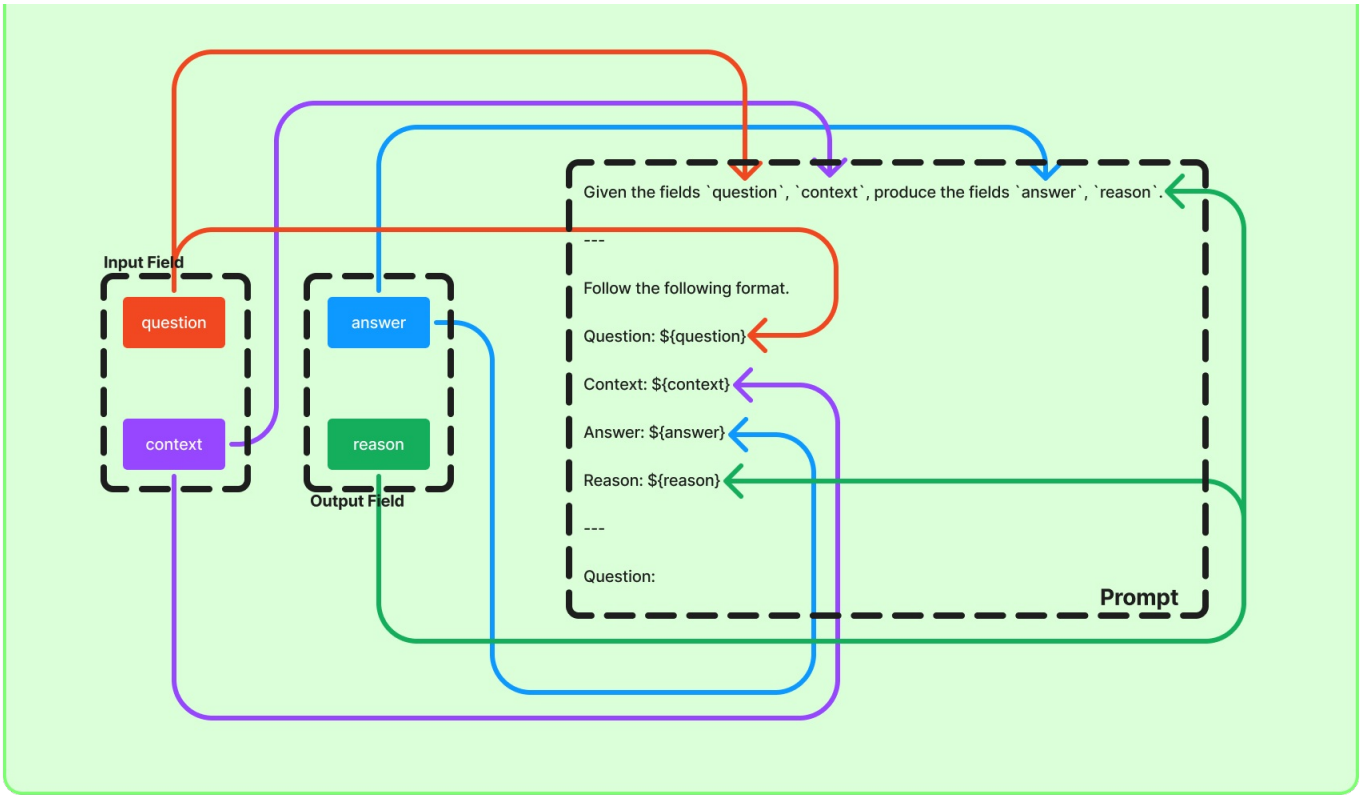
This simplistic naming of the fields is essential for the LLM to understand the nature of inputs and outputs, reducing sensitivity and ensuring clarity for expected inputs and generations.

However, this barebones signature may not provide a clear picture for how the model should approach the task, and to meet these needs, DSPy modules offer simplistic yet robust instructional templates that integrate the Signatures. Let's take a deeper look at the prompt constructed by DSPy to understand it better when used within a `dspy.Predict` module as `dspy.Predict(question->answer)`:

```
Given the fields `question`, produce the fields `answer`.

---

Follow the following format.

Question: ${question}
Answer: ${answer}

---

Question:
```

As you can see, DSPy populates the instruction `Given the fields ``question``, produce the fields ``answer``.` to define the task and provides instructions for the prompt format. And this format is pretty standard for any Signature you create as we can see in this prompting setup for RAG:

Prompt Creation




Now these instructional templates are well defined for their respective prompting techniques (CoT, ProgramOfThought, ReAct), leaving the user only having to define their task's Signature input and outputs with the rest handled by the DSPy modules library!

However, it would be nice to give more instructions beyond the simplistic in-line signature and for this, we turn to class-based signatures.

## Class Based Method

A Signature class comprises of three things:

- **Task Description/Instruction:** We define in the signature class docstring.
- **Inputs Field:** We define these as `dspy.InputField()`.
- **Outputs Field:** We define these as `dspy.OutputField()`.

```
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:")
```

The I/O Fields take 3 inputs: `desc`, `prefix` and `format`. `desc` is the description to the input, `prefix` is the placeholder text of the field in the prompt(one that has been `${field_name}` until now) and `format` which is a method that'll define how to handle non-string inputs. If the input to field is a list rather than a string, we can specify this through `format`.

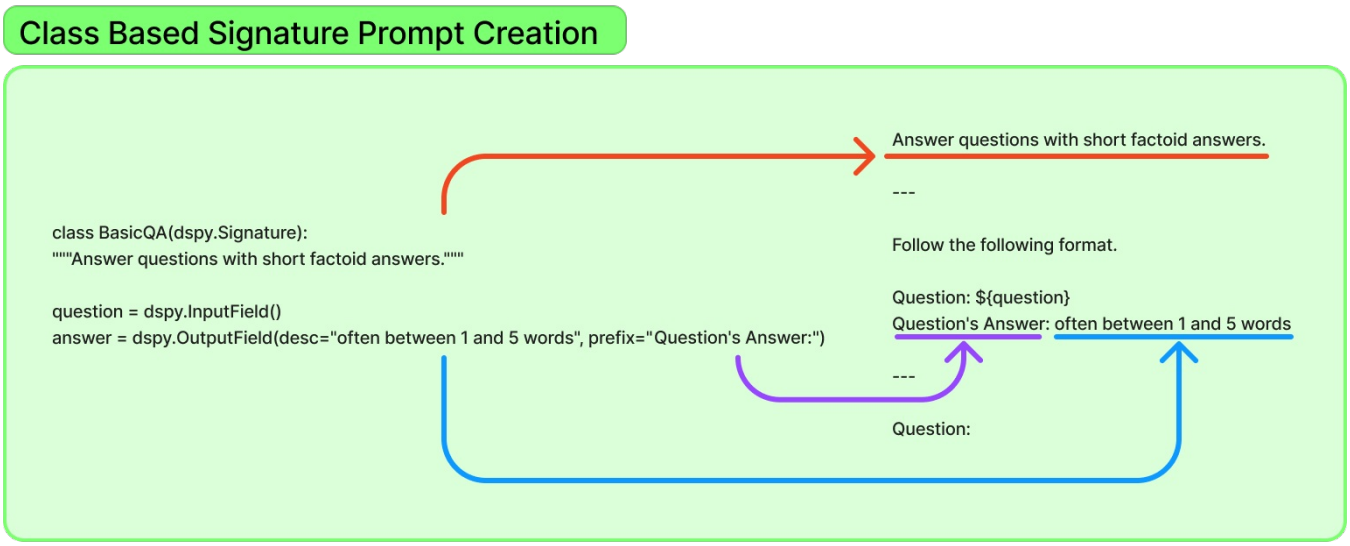
Both `InputField` and `OutputField` are similar in implementation as well:

```
class InputField(Field):
    def __init__(self, *, prefix=None, desc=None, format=None):
        super().__init__(prefix=prefix, desc=desc, input=True, format=format)

class OutputField(Field):
    def __init__(self, *, prefix=None, desc=None, format=None):
        super().__init__(prefix=prefix, desc=desc, input=False, format=format)
```

Let's take a look at how a prompt for the class based signature looks like:

```
Answer questions with short factoid answers.

---

Follow the following format.

Question: ${question}
Question's Answer: often between 1 and 5 words

---

Question:
```

As you can see, the instruction is more well-defined by our task's instruction in the docstring. The prefix and description for the `answer` field reflects our definitions. This ensures a more refined prompt structure, giving the user more control on defining its contents per task requirements.




---

**Written By:** Herumb Shandilya

# BootstrapFewShot

When compiling a DSPy program, we generally invoke a teleprompter, which is an optimizer that takes the program, a training set, and a metric—and returns a new optimized program. Different teleprompters apply different strategies for optimization. This family of teleprompters is focused on optimizing the few shot examples. Let's take an example of a Sample pipeline and see how we can use teleprompter to optimizes it.

## Setting up a Sample Pipeline

We'll be making a basic answer generation pipeline over GSM8K dataset that we saw in the Minimal Example, we won't be changing anything in it! So let's start by configuring the LM which will be OpenAI LM client with `gpt-3.5-turbo` as the LLM in use.

```
import dspy

turbo = dspy.OpenAI(model='gpt-3.5-turbo', max_tokens=250)
dspy.settings.configure(lm=turbo)
```

Now that we have the LM client setup it's time to import the train-dev split in `GSM8k` class that DSPy provides us:

```
from dspy.datasets.gsm8k import GSM8K, gsm8k_metric

gms8k = GSM8K()

trainset, devset = gms8k.train, gms8k.dev
```

We'll now define a basic QA inline signature i.e. `question->answer` and pass it to `ChainOfThought` module, that applies necessary addition for CoT style prompting to the Signature.

```
class CoT(dspy.Module):
    def __init__(self):
        super().__init__()
        self.prog = dspy.ChainOfThought("question -> answer")

    def forward(self, question):
        return self.prog(question=question)
```

Now we need to evaluate this pipeline too!! So we'll use the `Evaluate` class that DSPy provides us, as for the metric we'll use the `gsm8k_metric` that we imported above.

```
from dspy.evaluate import Evaluate

evaluate = Evaluate(devset=devset[:], metric=gsm8k_metric, num_threads=NUM_THREADS,
display_progress=True, display_table=False)
```

To evaluate the `CoT` pipeline we'll need to create an object of it and pass it as an arg to the `evaluator` call.

```
cot_baseline = CoT()

evaluate(cot_baseline, devset=devset[:])
```

Now we have the baseline pipeline ready to use, so let's try using the `BootstrapFewShot` teleprompter and optimizing our pipeline to make it even better!

## Using `BootstrapFewShot`

Let's start by importing and initializing our teleprompter, for the metric we'll be using the same `gsm8k_metric` imported and used above:

```
from dspy.teleprompt import BootstrapFewShotWithRandomSearch

teleprompter = BootstrapFewShotWithRandomSearch(
    metric=gsm8k_metric,
    max_bootstrapped_demos=8,
    max_labeled_demos=8,
)
```

`metric` is a an obvious parameter but what are `max_bootstrapped_demos` and `max_labeled_demos` parameters? Let's see the difference via a table:

| Feature | `max_labeled_demos` | `max_bootstrapped_demos` |
|---|---|---|
| **Purpose** | Refers to the maximum number of labeled demonstrations (examples) that will be used for training the student module directly. Labeled demonstrations are typically pre-existing, manually labeled examples that the module can learn from. | Refers to the maximum number of demonstrations that will be bootstrapped. Bootstrapping in this context likely means generating new training examples based on the predictions of a teacher module or some other process. These bootstrapped demonstrations are then used alongside or instead of the manually labeled examples. |
| **Training Usage** | Directly used in training; typically more reliable due to manual labeling. | Augment training data; potentially less accurate as they are generated examples. |
| **Data Source** | Pre-existing dataset of manually labeled examples. | Generated during the training process, often using outputs from a teacher module. |
| **Influence on Training** | Higher quality and reliability, assuming labels are accurate. | Provides more data but may introduce noise or inaccuracies. |

This teleprompter augments any necessary field even if you data doesn't have it, for example we don't have rationale for labelling however you'll see the rationale for each few shot example in the prompt that this teleprompter curated, how? By generating them all via a `teacher` module which is an optional parameter, since we didn't pass it the teleprompted creates a teacher from the module we are training or the `student` module.

In the next section, we'll seen this process step by step but for now let's start optimizing our `CoT` module by calling the `compile` method in the teleprompter:

```
cot_compiled = teleprompter.compile(CoT(), trainset=trainset, valset=devset)
```

Once the training is done you'll have a more optimized module that you can save or load again for use anytime:

```
cot_compiled.save('turbo_gsm8k.json')

# Loading:
# cot = CoT()
# cot.load('turbo_gsm8k.json')
```

## How `BootstrapFewShot` works?

`LabeledFewShot` is the most vanilla teleprompter that takes a training set as input and it assigns subsets of the trainset in each student's predictor's demos attribute. You can think of it as basically adding few shot examples to the prompt.

`BootStrapFewShot` starts by doing this only, it starts by:

1. Initializing a student program which which is the one we are optimizing and a teacher program which unless specified otherwise is a clone of the student.
2. Then to the teacher it add the demos by using `LabeledFewShot` teleprompter.
3. Mappings are created between the names of predictors and their corresponding instances in both student and teacher models.
4. The maximum number of bootstrap demonstrations (max_bootstraps) is determined. This limits the amount of initial training data generated.
5. The process iterates over each example in the training set. For each example, the method checks if the maximum number of bootstraps has been reached. If so, the process stops.
6. For each training example, the teacher model attempts to generate a prediction.
7. If the teacher model successfully generates a prediction, the trace of this prediction process is captured. This trace includes details about which predictors were called, the inputs they received, and the outputs they produced.
8. If the prediction is successful, a demonstration (demo) is created for each step in the trace. This demo includes the inputs to the predictor and the outputs it generated.

This is how it works, aside from that. `BootstrapFewShotWithOptuna`, `BootstrapFewShotWithRandomSearch` etc. which work on the same principle with slight changes in the example discovery process.

---

**Written By:** Herumb Shandilya

# Creating a Custom Local Model (LM) Client

DSPy provides you with multiple LM clients that you can use to execute any of your pipelines. However, if you have an API or LM that is unable to be executed by any of the existing clients hosted in DSPy, you can create one yourself!! It's not too difficult so let's see how!!

## Format of LM Client

An LM client needs to implement 3 methods at minimum: `__init__`, `basic_request` and `__call__`. So your custom LM client should follow the template below:

```
from dsp import LM


class CustomLMClient(LM):
    def __init__(self):
        self.provider = "default"

        self.history = []

    def basic_request(self, prompt, **kwargs):
        pass

    def __call__(self, prompt, only_completed=True, return_sorted=False, **kwargs):
        pass
```

While you can mostly add to and customize the client per your requirements, the client should be configurable with some key components to utilize every feature of DSPy without interruption. One of these key features is viewing the history of calls made to the LM via `inspect_history`. To elaborate:

- `__init__`: Should contain the `self.provider="default"` and `self.history=[]`. `self.history` will contain the prompt-completion pair created via LM call since the object was initialized. `self.provider` is used in `inspect_history` method and for most part you can leave it as **"default"**.
- `basic_request`: This function makes the call to the LM and retrieves the completion for the given prompt over the given `kwargs` which usually have parameters like `temperature`, `max_tokens`, etc. After you receive the completion from the LM, you must update the `self.history` list by appending the dictionary `{"prompt": prompt, "response": response}` to it. These fields are mandatory but you can add any other parameters as you see fit.
- `__call__`: This function should return the list of completions returned by the model. This can be a list of string completions in the basic case. Or it can be a tuple pair with the completion and corresponding likelihood as returned by the `Cohere` LM client. Additionally, the completion should be received by the `request` call which unless modified just calls `basic_request` as is, ensuring the history is updated as well.

By now you must've realized the reason we have these rules is mainly for making the history inspection and modules work without breaking.

> **INFO**
>
> You can mitigate issues with updating the history in the `__call__` itself. So if you can take care of history updates in `__call__` itself, you just need to implement `__init__` and `__call__`.
>
> Or if you are up for it, feel free to rewrite `inspect_history` method as per your requirements!

## Implementing our Custom LM

Based on whatever we have learned so far, let's implement our custom LM that calls the Claude API. In Claude, we need to initialize 2 components to make a successful call: `API_KEY` and `BASE_URL`. The base URL from on the **Anthropic docs** is

`https://api.anthropic.com/v1/messages`. Let's write our `__init__` method:

```
def __init__(model, api_key):
    self.model = model
    self.api_key = api_key
    self.provider = "default"
    self.history = []

    self.base_url = "https://api.anthropic.com/v1/messages"
```

Based on the implementation above, we want to now pass in our `model` like `claude-2` and the `api_key` which you'll see in Claude's API Console. Now it's time to define the `basic_request` method where we'll make the call to `self.base_url` and get the completion:

```
def basic_request(self, prompt: str, **kwargs):
    headers = {
        "x-api-key": self.api_key,
        "anthropic-version": "2023-06-01",
        "anthropic-beta": "messages-2023-12-15",
        "content-type": "application/json"
    }

    data = {
        **kwargs,
        "model": self.model,
        "messages": [
            {"role": "user", "content": prompt}
        ]
    }

    response = requests.post(self.base_url, headers=headers, json=data)
    response = response.json()

    self.history.append({
        "prompt": prompt,
        "response": response,
        "kwargs": kwargs,
    })
    return response
```

Now it's time to define `__call__` method that'll bring it all together:

```
def __call__(self, prompt, only_completed=True, return_sorted=False, **kwargs):
    response = self.request(prompt, **kwargs)

    completions = [result["text"] for result in response["content"]]

    return completions
```

To write it all in a single class, we'll get:

```
from dsp import LM

class Claude(LM):
    def __init__(model, api_key):
        self.model = model
        self.api_key = api_key
        self.provider = "default"

        self.base_url = "https://api.anthropic.com/v1/messages"
```

```
    def basic_request(self, prompt: str, **kwargs):
        headers = {
            "x-api-key": self.api_key,
            "anthropic-version": "2023-06-01",
            "anthropic-beta": "messages-2023-12-15",
            "content-type": "application/json"
        }

        data = {
            **kwargs,
            "model": self.model,
            "messages": [
                {"role": "user", "content": prompt}
            ]
        }

        response = requests.post(self.base_url, headers=headers, json=data)
        response = response.json()

        self.history.append({
            "prompt": prompt,
            "response": response,
            "kwargs": kwargs,
        })
        return response

    def __call__(self, prompt, only_completed=True, return_sorted=False, **kwargs):
        response = self.request(prompt, **kwargs)

        completions = [result["text"] for result in response["content"]]

        return completions
```

That's it! Now we can configure this as an `lm` in DSPy and use it in the pipeline like any other LM Client:

```
import dspy

claude = Claude(model='claude-2')

dspy.settings.configure(lm=claude)
```

---

**Written By:** Arnav Singhvi

# Creating Custom RM Client

DSPy provides support for various retrieval modules out of the box like `ColBERTv2`, `AzureCognitiveSearch`, `Pinecone`, `Weaviate`, etc. Unlike Language Model (LM) modules, creating a custom RM module is much more simple and flexible.

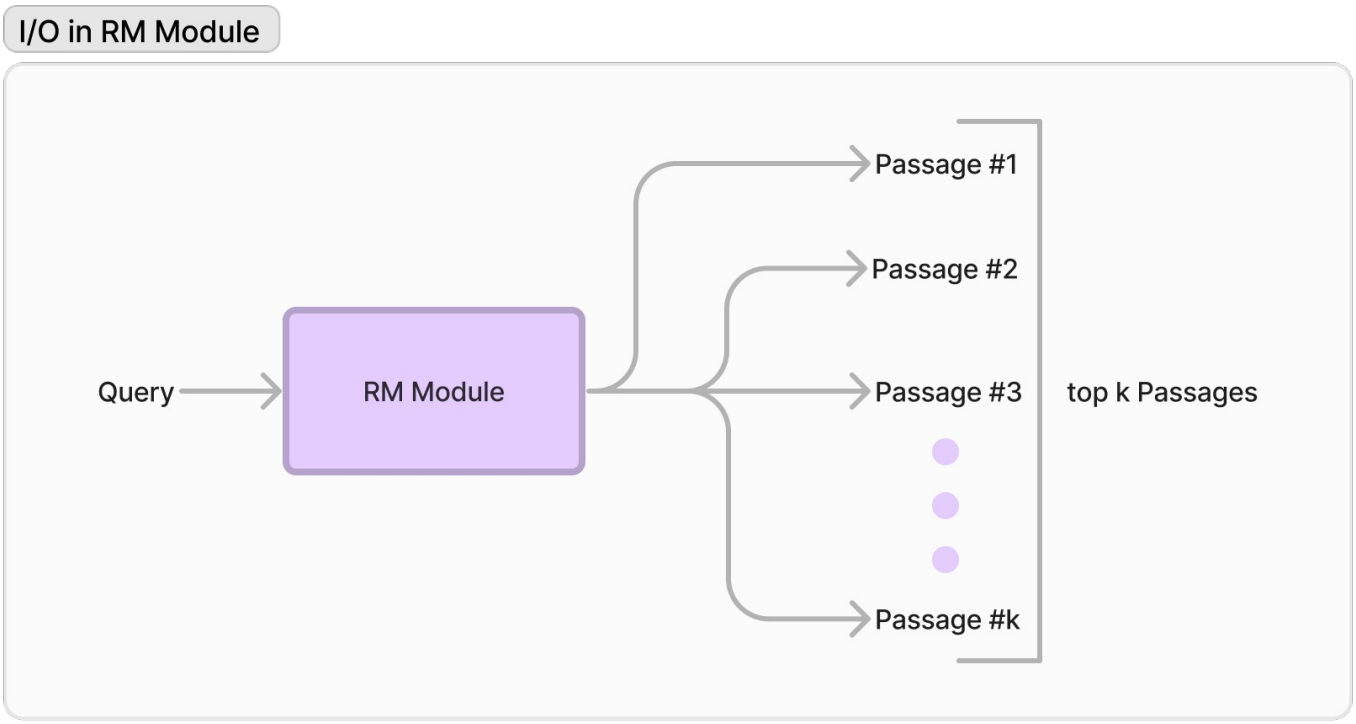
As of now, DSPy offers 2 ways to create a custom RM: the Pythonic way and the DSPythonic way. We'll take a look at both, understand why both are performing the same behavior, and how you can implement each!

## I/O of RM Client

Before understanding the implementation, let's understand the idea and I/O within RM modules.

The **input** to an RM module is either 1) a single query or 2) a list of queries.

The **output** is the `top-k` passages per query retrieved from a retrieval model, vector store, or search client.




Conventionally, we simply call the RM module object through the `__call__` method, inputting the query/queries as argument(s) of the call with the corresponding output returned as a list of strings.

We'll see how this I/O is essentially same in both methods of implementation but differs in their delivery.

## The Pythonic Way

To account for our RM I/O, we create a class that conducts the retrieval logic, which we implement in the `__init__` and `__call__` methods:

```
from typing import List, Union

class PythonicRMClient:
    def __init__(self):
        pass

    def __call__(self, query: Union[str, List[str]], k:int) -> Union[List[str], List[List[str]]]:
        pass
```

> **INFO**
>
> Don't worry about the extensive type-hinting above. `typing` is a package that provides type-definitions for function inputs and outputs.
>
> `Union` covers all possible types of the argument/output. So:
>
> - `Union[str, List[str]]`: Assigned to `query` to work with a single query string or a list of queries strings.
> - `Union[List[str], List[List[str]]]`: Assigned to the output of `__call__` to work with a single query string as a list or a list of multiple query string lists.

So let's start by implementing `PythonicRMClient` for a local retrieval model hosted on a API with endpoint being `/`. We'll start by implementing the `__init__` method, which simply initializes the class attributes, `url` and `port`, and attaches the port to the url if present.

```
def __init__(self, url: str, port:int = None):
    self.url = f`{url}:{port}` if port else url
```

Now it's time to write the retrieval logic in `__call__` method:

```
def __call__(self, query:str, k:int) -> List[str]:
    params = {"query": query, "k": k}
    response = requests.get(self.url, params=params)

    response = response.json()["retrieved_passages"]    # List of top-k passages
    return response
```

This serves to represent our API request call to retrieve our list of **top-k passages** which we return as the response. Let's bring it all together and see how our RM class looks like:

```
from typing import List

class PythonicRMClient:
    def __init__(self, url: str, port:int = None):
        self.url = f`{url}:{port}` if port else url

    def __call__(self, query:str, k:int) -> List[str]:
        # Only accept single query input, feel free to modify it to support

        params = {"query": query, "k": k}
        response = requests.get(self.url, params=params)

        response = response.json()["retrieved_passages"]    # List of top k passages
        return response
```

That's all!! This is the most basic way to implement a RM model and mirrors DSP-v1-hosted RM models like `ColBERTv2` and `AzureCognitiveSearch`.

Now let's take a look at how we streamline this process in DSPy!

## The DSPythonic Way

The DSPythonic way mirrors the Pythonic way in maintaining the same input but now returning an object of `dspy.Prediction` class, the standard output format for any DSPy module as we've seen in previous docs. Additionally, this class would now inherit the `dspy.Retrieve` class to maintain state management within the DSPy library.
So let's implement `__init__` and `forward` method where our class's `__call__` is calling the `forward` method as is=:

```
import dspy
from typing import List, Union, Optional

class DSPythonicRMClient(dspy.Retrieve):
    def __init__(self, k:int):
        pass

    def forward(self, query: Union[str, List[str]], k:Optional[str]) -> dspy.Prediction:
        pass
```

Unlike `PythonicRMClient`, we initialize `k` as part of the initialization call and the `forward` method will take query/queries as arguments and the `k` number of retrieved passages as an optional argument. `k` is used within the inherited `dspy.Retrieve` initialization when we call `super().__init__()`.

We'll be implementing `DSPythonicRMClient` for the same local retrieval model API we used above. We'll start by implementing the `__init__` method, which mirrors the `PythonicRMClient`.

```
def __init__(self, url: str, port:int = None, k:int = 3):
    super().__init__(k=k)

    self.url = f`{url}:{port}` if port else url
```

We'll now implement the `forward` method, returning the output as a `dspy.Prediction` object under the `passage` attribute which is standard among all the RM modules. The call will default to the defined `self.k` argument unless overridden in this call.

```
def forward(self, query:str, k:Optional[int]) -> dspy.Prediction:
    params = {"query": query, "k": k if k else self.k}
    response = requests.get(self.url, params=params)

    response = response.json()["retrieved_passages"]    # List of top k passages
    return dspy.Prediction(
        passages=response
    )
```

Let's bring it all together and see how our RM class looks like:

```
import dspy
from typing import List, Union, Optional

class DSPythonicRMClient(dspy.Retrieve):
    def __init__(self, url: str, port:int = None, k:int = 3):
        super().__init__(k=k)

        self.url = f`{url}:{port}` if port else url

    def forward(self, query_or_queries:str, k:Optional[int]) -> dspy.Prediction:
        params = {"query": query_or_queries, "k": k if k else self.k}
        response = requests.get(self.url, params=params)

        response = response.json()["retrieved_passages"]    # List of top k passages
        return dspy.Prediction(
            passages=response
        )
```

That's all!! This is the way to implement a custom RM model client within DSPy and how more recently-supported RM models like `QdrantRM`, `WeaviateRM`, etc. are implemented in DSPy.

Let's take a look at how we use these retrievers

Let's take a look at how we use these retrievers.

## Using Custom RM Models

DSPy offers two ways of using custom RM clients: Direct Method and using `dspy.Retrieve`.

### Direct Method

The most straightforward way to use your custom RM is by directly using its object within the DSPy Pipeline.

Let's take a look at the following pseudocode of a DSPy Pipeline as an example:

```
class DSPyPipeline(dspy.Module):
    def __init__(self):
        super().__init__()

        url = "http://0.0.0.0"
        port = 3000

        self.pythonic_rm = PythonicRMClient(url=url, port=port)
        self.dspythonic_rm = DSPythonicRMClient(url=url, port=port, k=3)

        ...

    def forward(self, *args):
        ...

        passages_from_pythonic_rm = self.pythonic_rm(query)
        passages_from_dspythonic_rm = self.dspythonic_rm(query).passages

        ...
```

This ensures you retrieve a list of passages from your RM client and can interact with the results within your forward pass in whichever way needed for your pipeline's purpose!

### Using `dspy.Retrieve`

This way is more experimental in essence, allowing you to maintain the same pipeline and experiment with different RMs. How? By configuring it!

```
import dspy

lm = ...
url = "http://0.0.0.0"
port = 3000

# pythonic_rm = PythonicRMClient(url=url, port=port)
dspythonic_rm = DSPythonicRMClient(url=url, port=port, k=3)

dspy.settings.configure(lm=lm, rm=dspythonic_rm)
```

Now, in the pipeline, you just need to use `dspy.Retrieve` which will use this `rm` client to get the top-k passage for a given query!

```
class DSPyPipeline(dspy.Module):
    def __init__(self):
        super().__init__()

        url = "http://0.0.0.0"
        port = 3000

        self.rm = dspy.Retrieve(k=3)
        ...

    def forward(self, *args):
```

```
        ...

        passages = self.rm(query)

        ...
```

Now if you'd like to use a different RM, you can just update the `rm` parameter via `dspy.settings.configure`.

> **HOW `dspy.Retrieve` USES `rm`**
>
> When we call `dspy.Retrieve` the `__call__` method will execute the `forward` method as is. In `forward`, the top-k passages are received by the `dsp.retrieveEnsemble` method in search.py.
>
> If an `rm` is not initialized in `dsp.settings`, this would raise an error.

---

**Written By:** Arnav Singhvi

# Data Handling

Data Handling in DSPy

## Examples in DSPy

Working in DSPy involves training sets, development sets, and test sets. This is like traditional ML, but you usually need far fewer labels (or zero labels) …

## Utilizing Built-in Datasets

It's easy to use your own data in DSPy: a dataset is just a list of Example objects. Using DSPy well involves being able to find and re-purpose existing dat…

## Creating a Custom Dataset

We've seen how to work with with Example objects and use the HotPotQA class to load the HuggingFace HotPotQA dataset as a list of Example objects…

# Signatures

Signatures in DSPy

## Understanding Signatures

A DSPy Signature is the most basic form of task description which simply requires inputs and outputs and optionally, a small description about them an…

## Executing Signatures

So far we've understood what signatures are and how we can use them to craft our prompt, but now let's take a look at how to execute them.

## internal-signatures

DSPy's Internal Signatures

# Modules

Modules in DSPy

### DSPy Assertions

Introduction

### ChainOfThoughtWithHint

This class builds upon the ChainOfThought class by introducing an additional input field to provide hints for reasoning. The inclusion of a hint allows for…

### Program of Thought

Background

### ReAct

Background

### Guide: DSPy Modules

Quick Recap

### Retrieve

Background

# Typed Predictors

Typed Predictors in DSPy

## Functional Typed Predictors

Typed Predictors as Decorators

## Understanding Typed Predictors

Why use a Typed Predictor?

# Language Model Clients

Language Model Clients in DSPy

### Remote Language Model Clients

5 items

### Creating a Custom Local Model (LM) Client

DSPy provides you with multiple LM clients that you can use to execute any of your pipelines. However, if you have an API or LM that is unable to be exe…

### local_models

2 items

# Retrieval Model Clients

Retrieval Models in DSPy

### ChromadbRM

Adapted from documentation provided by https://github.com/animtel

### ColBERTv2

Setting up the ColBERTv2 Client

### MilvusRM

MilvusRM uses OpenAI's text-embedding-3-small embedding by default or any customized embedding function.

### Weaviate Retrieval Model

Weaviate is an open-source vector database that can be used to retrieve relevant passages before passing it to the language model. Weaviate supports…

### AzureAISearch

A retrieval module that utilizes Azure AI Search to retrieve top passages for a given query.

### Creating Custom RM Client

DSPy provides support for various retrieval modules out of the box like ColBERTv2, AzureCognitiveSearch, Pinecone, Weaviate, etc. Unlike Language M…

# Teleprompters

Teleprompters are powerful optimizers (included in DSPy) that can learn to bootstrap and select effective prompts for the modules of any program. (The "tele-" in the name means "at a distance", i.e., automatic prompting at a distance.)

### BootstrapFewShot

When compiling a DSPy program, we generally invoke a teleprompter, which is an optimizer that takes the program, a training set, and a metric—and re…

### Signature Optimizer

COPRO which aims to improve the output prefixes and instruction of the signatures in a module in a zero/few shot setting. This teleprompter is especia…

# Release Checklist

☐ On `main` Create a git tag with pattern X.Y.Z where X, Y, and Z follow the semver pattern. Then push the tag to the origin git repo (github).

- 

```
git tag X.Y.Z
git push origin --tags
```

- This will trigger the github action to build and release the package.

☐ Confirm the tests pass and the package has been published to pypi.

- If the tests fail, you can remove the tag from your local and github repo using:

```
git push origin --delete X.Y.Z # Delete on Github
git tag -d X.Y.Z # Delete locally
```

- Fix the errors and then repeat the steps above to recreate the tag locally and push to Github to restart the process.
- Note that the github action takes care of incrementing the release version on test-pypi automatically by adding a pre-release identifier in the scenario where the tests fail and you need to delete and push the same tag again.

☐ Create a release

☐ Add release notes. You can make use of automatically generated release notes

If creating a new release for major or minor version:

- ☐ Create a new release branch with the last commit and name it 'release/X.Y`
- ☐ Update the default branch on the github rep to the new release branch.

## Prerequisites

The automation requires a trusted publisher to be set up on both the pypi and test-pypi packages. If the package is migrated to a new project, please follow the steps to create a trusted publisher. If you have no releases on the new project, you may have to create a pending trusted publisher to allow the first automated deployment.

# Signature Optimizer

`COPRO` which aims to improve the output prefixes and instruction of the signatures in a module in a zero/few shot setting. This teleprompter is especially beneficial for fine-tuning the prompt for language models and ensure they perform tasks more effectively, all from a vague and unrefined prompt.

## Setting up a Sample Pipeline

We'll be creating our CoT pipeline from scratch including the metric itself! So let's start by configuring the LM which will be OpenAI LM client with `gpt-3.5-turbo` as the LLM in use.

```
import dspy

turbo = dspy.OpenAI(model='gpt-3.5-turbo')
dspy.settings.configure(lm=turbo)
```

Now that we have the LM client setup it's time to import the train-dev split in `HotPotQA` class that DSPy provides us:

```
from dspy.datasets import HotPotQA

dataset = HotPotQA(train_seed=1, train_size=20, eval_seed=2023, dev_size=50, test_size=0)

trainset, devset = dataset.train, dataset.dev
```

We'll now define a class based signature for QA task similar to `question->answer` and pass it to `ChainOfThought` module, that will give us the result via Chain Of Thought from the LM client for this signature.

```
class CoTSignature(dspy.Signature):
    """Answer the question and give the reasoning for the same."""

    question = dspy.InputField(desc="question about something")
    answer = dspy.OutputField(desc="often between 1 and 5 words")

class CoTPipeline(dspy.Module):
    def __init__(self):
        super().__init__()

        self.signature = CoTSignature
        self.predictor = dspy.ChainOfThought(self.signature)

    def forward(self, question):
        result = self.predictor(question=question)
        return dspy.Prediction(
            answer=result.answer,
            reasoning=result.rationale,
        )
```

Now we need to evaluate this pipeline too!! So we'll use the `Evaluate` class that DSPy provides us, as for the metric we'll use the `validate_context_and_answer` that we'll define. `validate_context_and_answer` uses `dspy.evaluate.answer_exact_match` metric in DSPy which in essence sees if pred and example are same or not.

```
from dspy.evaluate import Evaluate

def validate_context_and_answer(example, pred, trace=None):
    answer_EM = dspy.evaluate.answer_exact_match(example, pred)
    return answer_EM
```

```
NUM_THREADS = 5
evaluate = Evaluate(devset=devset, metric=validate_context_and_answer, num_threads=NUM_THREADS,
display_progress=True, display_table=False)
```

To evaluate the `CoTPipeline` we'll need to create an object of it and pass it as an arg to the `evaluator` call.

```
cot_baseline = CoTPipeline()

devset_with_input = [dspy.Example({"question": r["question"], "answer":
r["answer"]}).with_inputs("question") for r in devset]
evaluate(cot_baseline, devset=devset_with_input)
```

Now we have the baseline pipeline ready to use, so let's try using the `COPRO` teleprompter and optimizing our pipeline to make it even better!

## Using `COPRO`

Let's start by importing and initializing our teleprompter, for the metric we'll be using the same `validate_context_and_answer` imported and used above:

```
from dspy.teleprompt import COPRO

teleprompter = COPRO(
    metric=validate_context_and_answer,
    verbose=True,
)
```

In this teleprompter there is a breadth and depth argument that defines the number of instruction/prefix candidate and number of iterations in the optimization step. We'll understand this in depth in the next section. This teleprompter comes up with better instruction candidates for the signature and better prefix candidates for the output fields of the signature. Let's start optimizing our `CoT` module by calling the `compile` method in the teleprompter:

```
kwargs = dict(num_threads=64, display_progress=True, display_table=0) # Used in Evaluate class in the
optimization process

compiled_prompt_opt = teleprompter.compile(cot, trainset=devset, eval_kwargs=kwargs)
```

Once the training is done you'll have better instructions and prefixes that you'll need to edit in signature manually. So let's say the output during optimization is like:

```
i: "Please answer the question and provide your reasoning for the answer. Your response should be clear
and detailed, explaining the rationale behind your decision. Please ensure that your answer is well-
reasoned and supported by relevant explanations and examples."
p: "[Rationale]"
Average Metric (78.9) ...
```

Then you'll copy this and edit the original instruction class to:

```
class CoTSignature(dspy.Signature):
    """Please answer the question and provide your reasoning for the answer. Your response should be
clear and detailed, explaining the rationale behind your decision. Please ensure that your answer is
well-reasoned and supported by relevant explanations and examples."""

    question = dspy.InputField(desc="question about something")
```

```
    reasoning = dspy.OutputField(desc="reasoning for the answer", prefix="[Rationale]")
    answer = dspy.OutputField(desc="often between 1 and 5 words")
```

> **INFO**
>
> The prefix would be proposed only for the output field that is defined first i.e. reasoning in `CoTSignature`.

Reinitialize the Pipeline object and reevaluate the pipeline! And now you have a more powerful predictor with more optimized Signature!

## How `COPRO` works?

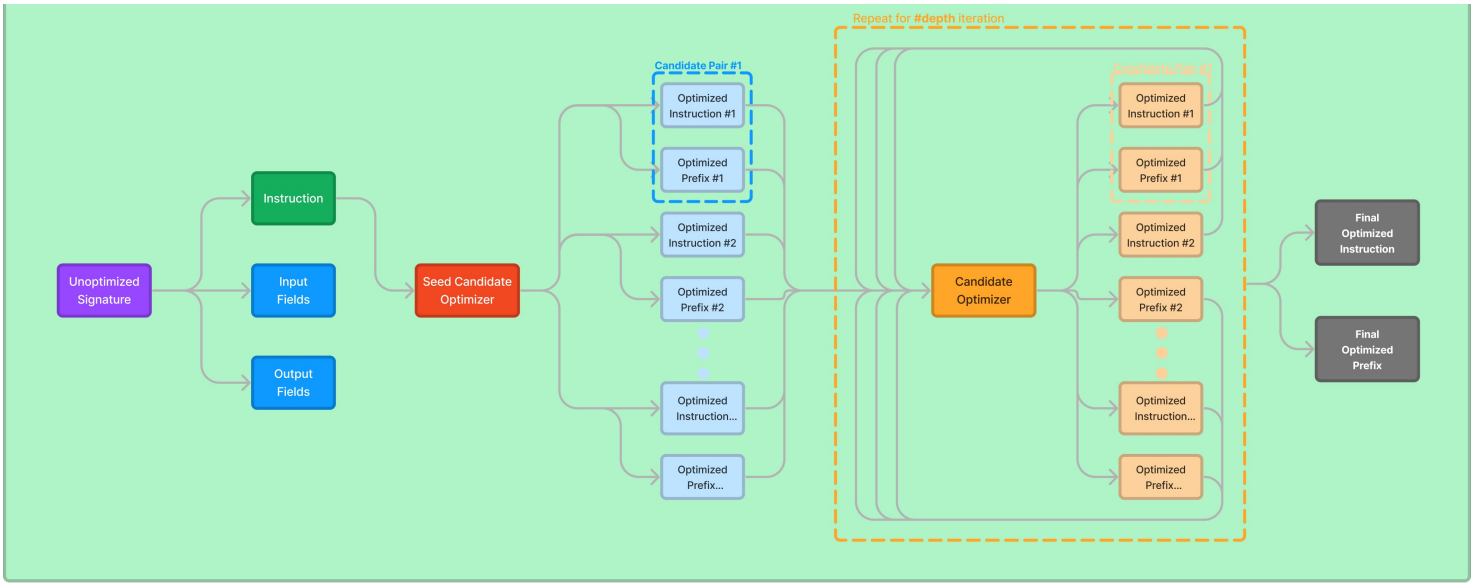
It is interesting that to get optimal prefixes and instruction, `COPRO` uses Signatures. Basically `COPRO` uses Signature to optimize Signature!! Let's look at the codebase a bit more closely:

```
class BasicGenerateInstruction(Signature):
    """You are an instruction optimizer for large language models. I will give you a ``signature`` of fields (inputs and outputs) in English. Your task is to propose an instruction that will lead a good language model to perform the task well. Don't be afraid to be creative."""

    basic_instruction = dspy.InputField(desc="The initial instructions before optimization")
    proposed_instruction = dspy.OutputField(desc="The improved instructions for the language model")
    proposed_prefix_for_output_field = dspy.OutputField(desc="The string at the end of the prompt, which will help the model start solving the task")

class GenerateInstructionGivenAttempts(dspy.Signature):
    """You are an instruction optimizer for large language models. I will give some task instructions I've tried, along with their corresponding validation scores. The instructions are arranged in increasing order based on their scores, where higher scores indicate better quality.

Your task is to propose a new instruction that will lead a good language model to perform the task even better. Don't be afraid to be creative."""

    attempted_instructions = dspy.InputField(format=dsp.passages2text)
    proposed_instruction = dspy.OutputField(desc="The improved instructions for the language model")
    proposed_prefix_for_output_field = dspy.OutputField(desc="The string at the end of the prompt, which will help the model start solving the task")
```

These two signatures are what give use the optimal instruction and prefixes. Now, the `BasicGenerateInstruction` will generate `n` instruction and prefixes based on the `breadth` parameter, basically `n=breadth`. This happens only one time in the start to seed the instruction attempts.

It uses these instructions and pass them to `GenerateInstructionGivenAttempts` which outputs hopefully a more optimal instruction. This then happens for `m` iterations which is the `depth` parameter in DSPy.

Signature Optimizer




Let's break down the process stepwise:

1. **Starting Point:** Use BasicGenerateInstruction to create initial optimized instructions and prefixes. This is based on a basic instruction input.
2. **Iterative Improvement:** Pass these initial instructions to GenerateInstructionGivenAttempts.
3. **Repeat Optimization:** In each iteration (up to m times):
    - Evaluate the current instructions and their effectiveness.
    - Propose new, more optimized instructions and prefixes based on the evaluation.
4. **Outcome:** After m iterations, the system ideally converges to a set of highly optimized instructions and corresponding prefixes that lead to better performance of the language model on the given task.

This iterative approach allows for continuous refinement of instructions and prefixes, leveraging the strengths of the teleprompter and improving task performance over time.

---

**Written By:** Herumb Shandilya

# HFClientTGI

## HFClient TGI

### Prerequisites - Launching TGI Server locally

Refer to the Text Generation-Inference Server API for setting up the TGI server locally.

```
#Example TGI Server Launch

model=meta-llama/Llama-2-7b-hf # set to the specific Hugging Face model ID you wish to use.
num_shard=1 # set to the number of shards you wish to use.
volume=$PWD/data # share a volume with the Docker container to avoid downloading weights every run

docker run --gpus all --shm-size 1g -p 8080:80 -v $volume:/data -e HUGGING_FACE_HUB_TOKEN={your_token}
ghcr.io/huggingface/text-generation-inference:latest --model-id $model --num-shard $num_shard
```

This command will start the server and make it accessible at `http://localhost:8080`.

### Setting up the TGI Client

The constructor initializes the `HFModel` base class to support the handling of prompting HuggingFace models. It configures the client for communicating with the hosted TGI server to generate requests. This requires the following parameters:

- `model` (*str*): ID of Hugging Face model connected to the TGI server.
- `port` (*int* or *list*): Port for communicating to the TGI server. This can be a single port number (`8080`) or a list of TGI ports (`[8080, 8081, 8082]`) to route the requests to.
- `url` (*str*): Base URL of hosted TGI server. This will often be `"http://localhost"`.
- `http_request_kwargs` (*dict*): Dictionary of additional keyword agruments to pass to the HTTP request function to the TGI server. This is `None` by default.
- `**kwargs`: Additional keyword arguments to configure the TGI client.

Example of the TGI constructor:

```
class HFClientTGI(HFModel):
    def __init__(self, model, port, url="http://future-hgx-1", http_request_kwargs=None, **kwargs):
```

### Under the Hood

**`_generate(self, prompt, **kwargs) -> dict`**

**Parameters:**

- `prompt` (*str*): Prompt to send to model hosted on TGI server.
- `**kwargs`: Additional keyword arguments for completion request.

**Returns:**

- `dict`: dictionary with `prompt` and list of response `choices`.

Internally, the method handles the specifics of preparing the request prompt and corresponding payload to obtain the response.

After generation, the method parses the JSON response received from the server and retrieves the output through `json_response["generated_text"]`. This is then stored in the `completions` list.

If the JSON response includes the additional `details` argument and correspondingly, the `best_of_sequences` within `details`,

this indicates multiple sequences were generated. This is also usually the case when `best_of > 1` in the initialized kwargs. Each of these sequences is accessed through `x["generated_text"]` and added to the `completions` list.

Lastly, the method constructs the response dictionary with two keys: the original request `prompt` and `choices`, a list of dictionaries representing generated completions with the key `text` holding the response's generated text.

### Using the TGI Client

```
tgi_llama2 = dspy.HFClientTGI(model="meta-llama/Llama-2-7b-hf", port=8080, url="http://localhost")
```

### Sending Requests via TGI Client

1. ***Recommended*** Configure default LM using `dspy.configure`.

This allows you to define programs in DSPy and simply call modules on your input fields, having DSPy internally call the prompt on the configured LM.

```
dspy.configure(lm=tgi_llama2)

#Example DSPy CoT QA program
qa = dspy.ChainOfThought('question -> answer')

response = qa(question="What is the capital of Paris?") #Prompted to tgi_llama2
print(response.answer)
```

2. Generate responses using the client directly.

```
response = tgi_llama2._generate(prompt='What is the capital of Paris?')
print(response)
```

---

**Written By:** Arnav Singhvi

# Page Not Found

We could not find what you were looking for.

Please contact the owner of the site that linked you to the original URL and let them know their link is broken.

# dspy.ChainOfThought

## Constructor

The constructor initializes the `ChainOfThought` class and sets up its attributes. It inherits from the `Predict` class and adds specific functionality for chain of thought processing.

Internally, the class initializes the `activated` attribute to indicate if chain of thought processing has been selected. It extends the `signature` to include additional reasoning steps and an updated `rationale_type` when chain of thought processing is activated.

```
class ChainOfThought(Predict):
    def __init__(self, signature, rationale_type=None, activated=True, **config):
        super().__init__(signature, **config)

        self.activated = activated

        signature = ensure_signature(self.signature)
        *_keys, last_key = signature.output_fields.keys()

        rationale_type = rationale_type or dspy.OutputField(
            prefix="Reasoning: Let's think step by step in order to",
            desc="${produce the " + last_key + "}. We ...",
        )

        self.extended_signature = signature.prepend("rationale", rationale_type, type_=str)
```

**Parameters:**

- `signature` (*Any*): Signature of predictive model.
- `rationale_type` (*dspy.OutputField*, *optional*): Rationale type for reasoning steps. Defaults to `None`.
- `activated` (*bool*, *optional*): Flag for activated chain of thought processing. Defaults to `True`.
- `**config` (*dict*): Additional configuration parameters for model.

## Method

**`forward(self, **kwargs)`**

This method extends the parent `Predict` class' forward pass while updating the signature when chain of thought reasoning is activated or if the language model is a GPT3 model.

**Parameters:**

- `**kwargs`: Keyword arguments required for prediction.

**Returns:**

- The result of the `forward` method.

## Examples

```
#Define a simple signature for basic question answering
class BasicQA(dspy.Signature):
    """Answer questions with short factoid answers."""
    question = dspy.InputField()
    answer = dspy.OutputField(desc="often between 1 and 5 words")

#Pass signature to ChainOfThought module
generate_answer = dspy.ChainOfThought(BasicQA)

# Call the predictor on a particular input.
question='What is the color of the sky?'
pred = generate_answer(question=question)
```

```
print(f"Question: {question}")
print(f"Predicted Answer: {pred.answer}")
```

The following example shows how to specify your custom rationale. Here `answer` corresponds to the last key to produce, it may be different in your case.

```
#define a custom rationale
rationale_type = dspy.OutputField(
            prefix="Reasoning: Let's think step by step in order to",
            desc="${produce the answer}. We ...",
        )
#Pass signature to ChainOfThought module
generate_answer = dspy.ChainOfThought(BasicQA, rationale_type=rationale_type)
```

# dspy.ChainOfThoughtWithHint

## Constructor

The constructor initializes the `ChainOfThoughtWithHint` class and sets up its attributes, inheriting from the `Predict` class. This class enhances the `ChainOfThought` class by offering an additional option to provide hints for reasoning. Two distinct signature templates are created internally depending on the presence of the hint.

```
class ChainOfThoughtWithHint(Predict):
    def __init__(self, signature, rationale_type=None, activated=True, **config):
        super().__init__(signature, **config)
        self.activated = activated
        signature = self.signature

        *keys, last_key = signature.fields.keys()
        rationale_type = rationale_type or dspy.OutputField(
            prefix="Reasoning: Let's think step by step in order to",
            desc="${produce the " + last_key + "}. We ...",
        )
        self.extended_signature1 = self.signature.insert(-2, "rationale", rationale_type, type_=str)

        DEFAULT_HINT_TYPE = dspy.OutputField()
        self.extended_signature2 = self.extended_signature1.insert(-2, "hint", DEFAULT_HINT_TYPE,
type_=str)
```

**Parameters:**

- `signature` (*Any*): Signature of predictive model.
- `rationale_type` (*dspy.OutputField*, *optional*): Rationale type for reasoning steps. Defaults to `None`.
- `activated` (*bool*, *optional*): Flag for activated chain of thought processing. Defaults to `True`.
- `**config` (*dict*): Additional configuration parameters for model.

## Method

**`forward(self, **kwargs)`**

This method extends the parent `Predict` class's forward pass, updating the signature dynamically based on the presence of `hint` in the keyword arguments and the `activated` attribute.

**Parameters:**

- `**kwargs`: Keyword arguments required for prediction.

**Returns:**

- The result of the `forward` method in the parent `Predict` class.

## Examples

```
#Define a simple signature for basic question answering
class BasicQA(dspy.Signature):
    """Answer questions with short factoid answers."""
    question = dspy.InputField()
    answer = dspy.OutputField(desc="often between 1 and 5 words")

#Pass signature to ChainOfThought module
generate_answer = dspy.ChainOfThoughtWithHint(BasicQA)

# Call the predictor on a particular input alongside a hint.
question='What is the color of the sky?'
hint = "It's what you often see during a sunny day."
pred = generate_answer(question=question, hint=hint)
```

```
print(f"Question: {question}")
print(f"Predicted Answer: {pred.answer}")
```

# dspy.MultiChainComparison

## Constructor

The constructor initializes the `MultiChainComparison` class and sets up its attributes. It inherits from the `Predict` class and adds specific functionality for multiple chain comparisons.

The class incorporates multiple student attempt reasonings and concludes with the selected best reasoning path out of the available attempts.

```
from .predict import Predict
from ..primitives.program import Module

import dsp

class MultiChainComparison(Module):
    def __init__(self, signature, M=3, temperature=0.7, **config):
        super().__init__()

        self.M = M
        signature = Predict(signature).signature
        *keys, last_key = signature.kwargs.keys()

        extended_kwargs = {key: signature.kwargs[key] for key in keys}

        for idx in range(M):
            candidate_type = dsp.Type(prefix=f"Student Attempt #{idx+1}:", desc="${reasoning attempt}")
            extended_kwargs.update({f'reasoning_attempt_{idx+1}': candidate_type})

        rationale_type = dsp.Type(prefix="Accurate Reasoning: Thank you everyone. Let's now
holistically", desc="${corrected reasoning}")
        extended_kwargs.update({'rationale': rationale_type, last_key: signature.kwargs[last_key]})

        signature = dsp.Template(signature.instructions, **extended_kwargs)
        self.predict = Predict(signature, temperature=temperature, **config)
        self.last_key = last_key
```

**Parameters:**

- `signature` (*Any*): Signature of predictive model.
- `M` (*int*, *optional*): Number of student reasoning attempts. Defaults to `3`.
- `temperature` (*float*, *optional*): Temperature parameter for prediction. Defaults to `0.7`.
- `**config` (*dict*): Additional configuration parameters for model.

## Method

**`forward(self, completions, **kwargs)`**

This method aggregates all the student reasoning attempts and calls the predict method with extended signatures to get the best reasoning.

**Parameters:**

- `completions`: List of completion objects which include student reasoning attempts.
- `**kwargs`: Additional keyword arguments.

**Returns:**

- The result of the `predict` method for the best reasoning.

## Examples

**Examples**

```
class BasicQA(dspy.Signature):
    """Answer questions with short factoid answers."""
    question = dspy.InputField()
    answer = dspy.OutputField(desc="often between 1 and 5 words")

# Example completions generated by a model for reference
completions = [
    dspy.Prediction(rationale="I recall that during clear days, the sky often appears this color.",
answer="blue"),
    dspy.Prediction(rationale="Based on common knowledge, I believe the sky is typically seen as this
color.", answer="green"),
    dspy.Prediction(rationale="From images and depictions in media, the sky is frequently represented
with this hue.", answer="blue"),
]

# Pass signature to MultiChainComparison module
compare_answers = dspy.MultiChainComparison(BasicQA)

# Call the MultiChainComparison on the completions
question = 'What is the color of the sky?'
final_pred = compare_answers(completions, question=question)

print(f"Question: {question}")
print(f"Final Predicted Answer (after comparison): {final_pred.answer}")
print(f"Final Rationale: {final_pred.rationale}")
```

# dspy.Predict

## Constructor

The constructor initializes the `Predict` class and sets up its attributes, taking in the `signature` and additional config options. If the `signature` is a string, it processes the input and output fields, generates instructions, and creates a template for the specified `signature` type.

```
class Predict(Parameter):
    def __init__(self, signature, **config):
        self.stage = random.randbytes(8).hex()
        self.signature = signature
        self.config = config
        self.reset()

        if isinstance(signature, str):
            inputs, outputs = signature.split("->")
            inputs, outputs = inputs.split(","), outputs.split(",")
            inputs, outputs = [field.strip() for field in inputs], [field.strip() for field in outputs]

            assert all(len(field.split()) == 1 for field in (inputs + outputs))

            inputs_ = ', '.join([f"`{field}`" for field in inputs])
            outputs_ = ', '.join([f"`{field}`" for field in outputs])

            instructions = f"""Given the fields {inputs_}, produce the fields {outputs_}."""

            inputs = {k: InputField() for k in inputs}
            outputs = {k: OutputField() for k in outputs}

            for k, v in inputs.items():
                v.finalize(k, infer_prefix(k))

            for k, v in outputs.items():
                v.finalize(k, infer_prefix(k))

            self.signature = dsp.Template(instructions, **inputs, **outputs)
```

**Parameters:**

- `signature` (*Any*): Signature of predictive model.
- `**config` (*dict*): Additional configuration parameters for model.

## Method

`__call__(self, **kwargs)`

This method serves as a wrapper for the `forward` method. It allows making predictions using the `Predict` class by providing keyword arguments.

**Paramters:**

- `**kwargs`: Keyword arguments required for prediction.

**Returns:**

- The result of `forward` method.

## Examples

```
#Define a simple signature for basic question answering
class BasicQA(dspy.Signature):
    """Answer questions with short factoid answers."""
```

```
    question = dspy.InputField()
    answer = dspy.OutputField(desc="often between 1 and 5 words")

#Pass signature to Predict module
generate_answer = dspy.Predict(BasicQA)

# Call the predictor on a particular input.
question='What is the color of the sky?'
pred = generate_answer(question=question)

print(f"Question: {question}")
print(f"Predicted Answer: {pred.answer}")
```

# dspy.ProgramOfThought

## Constructor

The constructor initializes the `ProgramOfThought` class and sets up its attributes. It is designed to generate and execute Python code based on input fields, producing a single output field through iterative refinement. It supports multiple iterations to refine the code in case of errors.

```
import dsp
import dspy
from ..primitives.program import Module
from ..primitives.python_interpreter import CodePrompt, PythonInterpreter
import re

class ProgramOfThought(Module):
    def __init__(self, signature, max_iters=3):
        ...
```

**Parameters:**

- `signature` (*dspy.Signature*): Signature defining the input and output fields for the program.
- `max_iters` (*int*, *optional*): Maximum number of iterations for refining the generated code. Defaults to `3`.

## Methods

**`_generate_signature(self, mode)`**

Generates a signature dict for different modes: `generate`, `regenerate`, and `answer`.

The `generate` mode serves as an initial generation of Python code with the signature (`question -> generated_code`). The `regenerate` mode serves as a refining generation of Python code, accounting for the past generated code and existing error with the signature (`question, previous_code, error -> generated_code`). The `answer` mode serves to execute the last stored generated code and output the final answer to the question with the signature (`question, final_generated_code, code_output -> answer`).

**Parameters:**

- `mode` (*str*): Mode of operation of Program of Thought.

**Returns:**

- A dictionary representing the signature for specified mode.

**`_generate_instruction(self, mode)`**

Generates instructions for code generation based on the mode. This ensures the signature accounts for relevant instructions in generating an answer to the inputted question by producing executable Python code.

The instructional modes mirror the signature modes: `generate`, `regenerate`, `answer`

**Parameters:**

- `mode` (*str*): Mode of operation.

**Returns:**

- A string representing instructions for specified mode.

**`parse_code(self, code_data)`**

Parses the generated code and checks for formatting errors.

**Parameters:**

- `code_data` (*dict*): Data containing the generated code. `key - generated_code`, `val - {Python code string}`

**Returns:**

- Tuple containing parsed code and any error message.

`execute_code(self, code)`
Executes the parsed code and captures the output or error.

**Parameters:**

- `code` (*str*): Code to be executed.

**Returns:**

- Tuple containing the code, its output, and any error message.

`forward(self, **kwargs)`
Main method to execute the code generation and refinement process.

**Parameters:**

- `**kwargs`: Keyword arguments corresponding to input fields.

**Returns:**

- The final answer generated by the program or `None` in case of persistent errors.

## Examples

```
#Define a simple signature for basic question answering
class GenerateAnswer(dspy.Signature):
    """Answer questions with short factoid answers."""
    question = dspy.InputField()
    answer = dspy.OutputField(desc="often between 1 and 5 words")

# Pass signature to ProgramOfThought Module
pot = dspy.ProgramOfThought(GenerateAnswer)

#Call the ProgramOfThought module on a particular input
question = 'Sarah has 5 apples. She buys 7 more apples from the store. How many apples does Sarah have
now?'
result = pot(question=question)

print(f"Question: {question}")
print(f"Final Predicted Answer (after ProgramOfThought process): {result.answer}")
```

# dspy.ReAct

## Constructor

The constructor initializes the `ReAct` class and sets up its attributes. It is specifically designed to compose the interleaved steps of Thought, Action, and Observation.

```
import dsp
import dspy
from ..primitives.program import Module
from .predict import Predict

class ReAct(Module):
    def __init__(self, signature, max_iters=5, num_results=3, tools=None):
        ...
```

**Parameters:**

- `signature` (*Any*): Signature of the predictive model.
- `max_iters` (*int*, *optional*): Maximum number of iterations for the Thought-Action-Observation cycle. Defaults to `5`.
- `num_results` (*int*, *optional*): Number of results to retrieve in the action step. Defaults to `3`.
- `tools` (*List[dspy.Tool]*, *optional*): List of tools available for actions. If none is provided, a default `Retrieve` tool with `num_results` is used.

## Methods

**`_generate_signature(self, iters)`**

Generates a signature for the Thought-Action-Observation cycle based on the number of iterations.

**Parameters:**

- `iters` (*int*): Number of iterations.

**Returns:**

- A dictionary representation of the signature.

---

**`act(self, output, hop)`**

Processes an action and returns the observation or final answer.

**Parameters:**

- `output` (*dict*): Current output from the Thought.
- `hop` (*int*): Current iteration number.

**Returns:**

- A string representing the final answer or `None`.

---

**`forward(self, **kwargs)`**

Main method to execute the Thought-Action-Observation cycle for a given set of input fields.

**Parameters:**

- `**kwargs`: Keyword arguments corresponding to input fields.

**Returns:**

**Returns:**

- A `dspy.Prediction` object containing the result of the ReAct process.

## Examples

```
# Define a simple signature for basic question answering
class BasicQA(dspy.Signature):
    """Answer questions with short factoid answers."""
    question = dspy.InputField()
    answer = dspy.OutputField(desc="often between 1 and 5 words")

# Pass signature to ReAct module
react_module = dspy.ReAct(BasicQA)

# Call the ReAct module on a particular input
question = 'What is the color of the sky?'
result = react_module(question=question)

print(f"Question: {question}")
print(f"Final Predicted Answer (after ReAct process): {result.answer}")
```

# dspy.Retrieve

## Constructor

The constructor initializes the `Retrieve` class and sets up its attributes, taking in `k` number of retrieval passages to return for a query.

```
class Retrieve(Parameter):
    def __init__(self, k=3):
        self.stage = random.randbytes(8).hex()
        self.k = k
```

**Parameters:**

- `k` (*Any*): Number of retrieval responses

## Method

`__call__(self, *args, **kwargs):`

This method serves as a wrapper for the `forward` method. It allows making retrievals on an input query using the `Retrieve` class.

**Parameters:**

- `**args`: Arguments required for retrieval.
- `**kwargs`: Keyword arguments required for retrieval.

**Returns:**

- The result of the `forward` method.

## Examples

```
query='When was the first FIFA World Cup held?'

# Call the retriever on a particular query.
retrieve = dspy.Retrieve(k=3)
topK_passages = retrieve(query).passages

print(f"Top {retrieve.k} passages for question: {query} \n", '-' * 30, '\n')

for idx, passage in enumerate(topK_passages):
    print(f'{idx+1}]', passage, '\n')
```

# dspy.TypedPredictor

The `TypedPredictor` class is a sophisticated module designed for making predictions with strict type validations. It leverages a signature to enforce type constraints on inputs and outputs, ensuring that the data follows to the expected schema.

## Constructor

```
TypedPredictor(
    CodeSignature
    max_retries=3
)
```

Parameters:

- `signature` (dspy.Signature): The signature that defines the input and output fields along with their types.
- `max_retries` (int, optional): The maximum number of retries for generating a valid prediction output. Defaults to 3.

## Methods

**`copy() -> "TypedPredictor"`**
Creates and returns a deep copy of the current TypedPredictor instance.

**Returns:** A new instance of TypedPredictor that is a deep copy of the original instance.

**`_make_example(type_: Type) -> str`**
A static method that generates a JSON object example based pn the schema of the specified Pydantic model type. This JSON object serves as an example for the expected input or output format.

**Parameters:**

- `type_`: A Pydantic model class for which an example JSON object is to be generated.

**Returns:** A string that represents a JSON object example, which validates against the provided Pydantic model's JSON schema. If the method is unable to generate a valid example, it returns an empty string.

**`_prepare_signature() -> dspy.Signature`**
Prepares and returns a modified version of the signature associated with the TypedPredictor instance. This method iterates over the signature's fields to add format and parser functions based on their type annotations.

**Returns:** A dspy.Signature object that has been enhanced with formatting and parsing specifications for its fields.

**`forward(**kwargs) -> dspy.Prediction`**
Executes the prediction logic, making use of the `dspy.Predict` component to generate predictions based on the input arguments. This method handles type validation, parsing of output data, and implements retry logic in case the output does not initially follow to the specified output schema.

**Parameters:**

- `**kwargs`: Keyword arguments corresponding to the input fields defined in the signature.

**Returns:** A dspy.Prediction object containing the prediction results. Each key in this object corresponds to an output field defined in the signature, and its value is the parsed result of the prediction.

## Example

```
from dspy import InputField, OutputField, Signature
from dspy.functional import TypedPredictor
from pydantic import BaseModel

# We define a pydantic type that automatically checks if it's argument is valid python code.
```

```
class CodeOutput(BaseModel):
    code: str
    api_reference: str

class CodeSignature(Signature):
    function_description: str = InputField()
    solution: CodeOutput = OutputField()

cot_predictor = TypedPredictor(CodeSignature)
prediction = cot_predictor(
    function_description="Write a function that adds two numbers."
)

print(prediction["code"])
print(prediction["api_reference"])
```

# dspy.TypedChainOfThought

## Overview

`def TypedChainOfThought(signature, max_retries=3) -> dspy.Module`

Adds a Chain of Thoughts `dspy.OutputField` to the `dspy.TypedPredictor` module by prepending it to the Signature. Similar to `dspy.TypedPredictor` but automatically adds a "reasoning" output field.

- **Inputs**:
  - `signature`: The `dspy.Signature` specifying the input/output fields
  - `max_retries`: Maximum number of retries if outputs fail validation
- **Output**: A dspy.Module instance capable of making predictions.

## Example

```
from dspy import InputField, OutputField, Signature
from dspy.functional import TypedChainOfThought
from pydantic import BaseModel

# We define a pydantic type that automatically checks if it's argument is valid python code.
class CodeOutput(BaseModel):
    code: str
    api_reference: str

class CodeSignature(Signature):
    function_description: str = InputField()
    solution: CodeOutput = OutputField()

cot_predictor = TypedChainOfThought(CodeSignature)
prediction = cot_predictor(
    function_description="Write a function that adds two numbers."
)

print(prediction["code"])
print(prediction["api_reference"])
```

# dspy.predictor

## Overview

`def predictor(func) -> dspy.Module`

The `@predictor` decorator is used to create a predictor module based on the provided function. It automatically generates a `dspy.TypedPredictor` and from the function's type annotations and docstring.

- **Input**: Function with input parameters and return type annotation.
- **Output**: A dspy.Module instance capable of making predictions.

## Example

```
import dspy

context = ["Roses are red.", "Violets are blue"]
question = "What color are roses?"

@dspy.predictor
def generate_answer(self, context: list[str], question) -> str:
    """Answer questions with short factoid answers."""
    pass

generate_answer(context=context, question=question)
```

# dspy.cot

## Overview

`def cot(func) -> dspy.Module`

The `@cot` decorator is used to create a Chain of Thoughts module based on the provided function. It automatically generates a `dspy.TypedPredictor` from the function's type annotations and docstring. Similar to predictor, but adds a "Reasoning" output field to capture the model's step-by-step thinking.

- **Input**: Function with input parameters and return type annotation.
- **Output**: A dspy.Module instance capable of making predictions.

## Example

```
import dspy

context = ["Roses are red.", "Violets are blue"]
question = "What color are roses?"

@dspy.cot
def generate_answer(self, context: list[str], question) -> str:
    """Answer questions with short factoid answers."""
    pass

generate_answer(context=context, question=question)
```

# teleprompt.LabeledFewShot

## Constructor

The constructor initializes the `LabeledFewShot` class and sets up its attributes, particularly defining `k` number of samples to be used by the predictor.

```
class LabeledFewShot(Teleprompter):
    def __init__(self, k=16):
        self.k = k
```

**Parameters:**

- `k` (*int*): Number of samples to be used for each predictor. Defaults to 16.

## Method

**`compile(self, student, *, trainset)`**

This method compiles the `LabeledFewShot` instance by configuring the `student` predictor. It assigns subsets of the `trainset` in each student's predictor's `demos` attribute. If the `trainset` is empty, the method returns the original `student`.

**Parameters:**

- `student` (*Teleprompter*): Student predictor to be compiled.
- `trainset` (*list*): Training dataset for compiling with student predictor.

**Returns:**

- The compiled `student` predictor with assigned training samples for each predictor or the original `student` if the `trainset` is empty.

## Example

```
import dspy

#Assume defined trainset
class RAG(dspy.Module):
    def __init__(self, num_passages=3):
        super().__init__()

        #declare retrieval and predictor modules
        self.retrieve = dspy.Retrieve(k=num_passages)
        self.generate_answer = dspy.ChainOfThought(GenerateAnswer)

    #flow for answering questions using predictor and retrieval modules
    def forward(self, question):
        context = self.retrieve(question).passages
        prediction = self.generate_answer(context=context, question=question)
        return dspy.Prediction(context=context, answer=prediction.answer)

#Define teleprompter
teleprompter = LabeledFewShot()

# Compile!
compiled_rag = teleprompter.compile(student=RAG(), trainset=trainset)
```

# teleprompt.BootstrapFewShot

## Constructor

The constructor initializes the `BootstrapFewShot` class and sets up parameters for bootstrapping.

```
class BootstrapFewShot(Teleprompter):
    def __init__(self, metric=None, metric_threshold=None, teacher_settings={},
max_bootstrapped_demos=4, max_labeled_demos=16, max_rounds=1, max_errors=5):
        self.metric = metric
        self.teacher_settings = teacher_settings

        self.max_bootstrapped_demos = max_bootstrapped_demos
        self.max_labeled_demos = max_labeled_demos
        self.max_rounds = max_rounds
```

**Parameters:**

- `metric` (*callable*, *optional*): Metric function to evaluate examples during bootstrapping. Defaults to `None`.
- `metric_threshold` (*float*, *optional*): Score threshold for metric to determine successful example. Defaults to `None`.
- `teacher_settings` (*dict*, *optional*): Settings for teacher predictor. Defaults to empty dictionary.
- `max_bootstrapped_demos` (*int*, *optional*): Maximum number of bootstrapped demonstrations per predictor. Defaults to 4.
- `max_labeled_demos` (*int*, *optional*): Maximum number of labeled demonstrations per predictor. Defaults to 16.
- `max_rounds` (*int*, *optional*): Maximum number of bootstrapping rounds. Defaults to 1.
- `max_errors` (*int*): Maximum errors permitted during evaluation. Halts run with the latest error message. Defaults to 5. Configure to 1 if no evaluation run error is desired.

## Method

**`compile(self, student, *, teacher=None, trainset, valset=None)`**

This method compiles the BootstrapFewShot instance by performing bootstrapping to refine the student predictor.

This process includes preparing the student and teacher predictors, which involves creating predictor copies, verifying the student predictor is uncompiled, and compiling the teacher predictor with labeled demonstrations via LabeledFewShot if the teacher predictor hasn't been compiled.

The next stage involves preparing predictor mappings by validating that both the student and teacher predictors have the same program structure and the same signatures but are different objects.

The final stage is performing the bootstrapping iterations.

**Parameters:**

- `student` (*Teleprompter*): Student predictor to be compiled.
- `teacher` (*Teleprompter*, *optional*): Teacher predictor used for bootstrapping. Defaults to `None`.
- `trainset` (*list*): Training dataset used in bootstrapping.
- `valset` (*list*, *optional*): Validation dataset used in compilation. Defaults to `None`.

**Returns:**

- The compiled `student` predictor after bootstrapping with refined demonstrations.

## Example

```
#Assume defined trainset
#Assume defined RAG class
...
```

```
#Define teleprompter and include teacher
teacher = dspy.OpenAI(model='gpt-3.5-turbo', api_key = openai.api_key, api_provider = "openai",
model_type = "chat")
teleprompter = BootstrapFewShot(teacher_settings=dict({'lm': teacher}))

# Compile!
compiled_rag = teleprompter.compile(student=RAG(), trainset=trainset)
```

# teleprompt.Ensemble

## Constructor

The constructor initializes the `Ensemble` class and sets up its attributes. This teleprompter is designed to create ensembled versions of multiple programs, reducing various outputs from different programs into a single output.

```
class Ensemble(Teleprompter):
    def __init__(self, *, reduce_fn=None, size=None, deterministic=False):
```

**Parameters:**

- `reduce_fn` (*callable*, *optional*): Function used to reduce multiple outputs from different programs into a single output. A common choice is `dspy.majority`. Defaults to `None`.
- `size` (*int*, *optional*): Number of programs to randomly select for ensembling. If not specified, all programs will be used. Defaults to `None`.
- `deterministic` (*bool*, *optional*): Specifies whether ensemble should operate deterministically. Currently, setting this to `True` will raise an error as this feature is pending implementation. Defaults to `False`.

## Method

**`compile(self, programs)`**

This method compiles an ensemble of programs into a single program that when run, can either randomly sample a subset of the given programs to produce outputs or use all of them. The multiple outputs can then be reduced into a single output using the `reduce_fn`.

**Parameters:**

- `programs` (*list*): List of programs to be ensembled.

**Returns:**

- `EnsembledProgram` (*Module*): An ensembled version of the input programs.

## Example

```
import dspy
from dspy.teleprompt import Ensemble

# Assume a list of programs
programs = [program1, program2, program3, ...]

# Define Ensemble teleprompter
teleprompter = Ensemble(reduce_fn=dspy.majority, size=2)

# Compile to get the EnsembledProgram
ensembled_program = teleprompter.compile(programs)
```

# teleprompt.BootstrapFewShotWithRandomSearch

## Constructor

The constructor initializes the `BootstrapFewShotWithRandomSearch` class and sets up its attributes. It inherits from the `BootstrapFewShot` class and introduces additional attributes for the random search process.

```
class BootstrapFewShotWithRandomSearch(BootstrapFewShot):
    def __init__(self, metric, teacher_settings={}, max_bootstrapped_demos=4, max_labeled_demos=16,
max_rounds=1, num_candidate_programs=16, num_threads=6, max_errors=10, stop_at_score=None,
metric_threshold=None):
        self.metric = metric
        self.teacher_settings = teacher_settings
        self.max_rounds = max_rounds
        self.num_threads = num_threads
        self.stop_at_score = stop_at_score
        self.metric_threshold = metric_threshold
        self.min_num_samples = 1
        self.max_num_samples = max_bootstrapped_demos
        self.max_errors = max_errors
        self.num_candidate_sets = num_candidate_programs
        self.max_num_traces = 1 + int(max_bootstrapped_demos / 2.0 * self.num_candidate_sets)

        self.max_bootstrapped_demos = self.max_num_traces
        self.max_labeled_demos = max_labeled_demos

        print("Going to sample between", self.min_num_samples, "and", self.max_num_samples, "traces per
predictor.")
        print("Going to sample", self.max_num_traces, "traces in total.")
        print("Will attempt to train", self.num_candidate_sets, "candidate sets.")
```

**Parameters:**

- `metric` (*callable*, *optional*): Metric function to evaluate examples during bootstrapping. Defaults to `None`.
- `teacher_settings` (*dict*, *optional*): Settings for teacher predictor. Defaults to an empty dictionary.
- `max_bootstrapped_demos` (*int*, *optional*): Maximum number of bootstrapped demonstrations per predictor. Defaults to 4.
- `max_labeled_demos` (*int*, *optional*): Maximum number of labeled demonstrations per predictor. Defaults to 16.
- `max_rounds` (*int*, *optional*): Maximum number of bootstrapping rounds. Defaults to 1.
- `num_candidate_programs` (*int*): Number of candidate programs to generate during random search. Defaults to 16.
- `num_threads` (*int*): Number of threads used for evaluation during random search. Defaults to 6.
- `max_errors` (*int*): Maximum errors permitted during evaluation. Halts run with the latest error message. Defaults to 10. Configure to 1 if no evaluation run error is desired.
- `stop_at_score` (*float*, *optional*): Score threshold for random search to stop early. Defaults to `None`.
- `metric_threshold` (*float*, *optional*): Score threshold for metric to determine successful example. Defaults to `None`.

## Method

Refer to teleprompt.BootstrapFewShot documentation.

## Example

```
#Assume defined trainset
#Assume defined RAG class
...

#Define teleprompter and include teacher
teacher = dspy.OpenAI(model='gpt-3.5-turbo', api_key = openai.api_key, api_provider = "openai",
model_type = "chat")
teleprompter = BootstrapFewShotWithRandomSearch(teacher_settings=dict({'lm': teacher}))
```

```
# Compile!
compiled_rag = teleprompter.compile(student=RAG(), trainset=trainset)
```

# teleprompt.BootstrapFinetune

## Constructor

`__init__(self, metric=None, teacher_settings={}, multitask=True)`

The constructor initializes a `BootstrapFinetune` instance and sets up its attributes. It defines the teleprompter as a `BootstrapFewShot` instance for the finetuning compilation.

```
class BootstrapFinetune(Teleprompter):
    def __init__(self, metric=None, teacher_settings={}, multitask=True):
```

**Parameters:**

- `metric` (*callable*, *optional*): Metric function to evaluate examples during bootstrapping. Defaults to `None`.
- `teacher_settings` (*dict*, *optional*): Settings for teacher predictor. Defaults to empty dictionary.
- `multitask` (*bool*, *optional*): Enable multitask fine-tuning. Defaults to `True`.

## Method

`compile(self, student, *, teacher=None, trainset, valset=None, target='t5-large', bsize=12, accumsteps=1, lr=5e-5, epochs=1, bf16=False)`

This method first compiles for bootstrapping with the `BootstrapFewShot` teleprompter. It then prepares fine-tuning data by generating prompt-completion pairs for training and performs finetuning. After compilation, the LMs are set to the finetuned models and the method returns a compiled and fine-tuned predictor.

**Parameters:**

- `student` (*Predict*): Student predictor to be fine-tuned.
- `teacher` (*Predict*, *optional*): Teacher predictor to help with fine-tuning. Defaults to `None`.
- `trainset` (*list*): Training dataset for fine-tuning.
- `valset` (*list*, *optional*): Validation dataset for fine-tuning. Defaults to `None`.
- `target` (*str*, *optional*): Target model for fine-tuning. Defaults to `'t5-large'`.
- `bsize` (*int*, *optional*): Batch size for training. Defaults to `12`.
- `accumsteps` (*int*, *optional*): Gradient accumulation steps. Defaults to `1`.
- `lr` (*float*, *optional*): Learning rate for fine-tuning. Defaults to `5e-5`.
- `epochs` (*int*, *optional*): Number of training epochs. Defaults to `1`.
- `bf16` (*bool*, *optional*): Enable mixed-precision training with BF16. Defaults to `False`.

**Returns:**

- `compiled2` (*Predict*): A compiled and fine-tuned `Predict` instance.

## Example

```
#Assume defined trainset
#Assume defined RAG class
...

#Define teleprompter
teleprompter = BootstrapFinetune(teacher_settings=dict({'lm': teacher}))

# Compile!
compiled_rag = teleprompter.compile(student=RAG(), trainset=trainset, target='google/flan-t5-base')
```

# dspy.ColBERTv2

## Constructor

The constructor initializes the `ColBERTv2` class instance and sets up the request parameters for interacting with the ColBERTv2 server.

```
class ColBERTv2:
    def __init__(
        self,
        url: str = "http://0.0.0.0",
        port: Optional[Union[str, int]] = None,
        post_requests: bool = False,
    ):
```

**Parameters:**

- `url` (*str*): URL for ColBERTv2 server.
- `port` (*Union[str, int]*, *Optional*): Port endpoint for ColBERTv2 server. Defaults to `None`.
- `post_requests` (*bool*, *Optional*): Flag for using HTTP POST requests. Defaults to `False`.

## Methods

**`__call__(self, query: str, k: int = 10, simplify: bool = False) -> Union[list[str], list[dotdict]]`**

Enables making queries to the ColBERTv2 server for retrieval. Internally, the method handles the specifics of preparing the request prompt and corresponding payload to obtain the response. The function handles the retrieval of the top-k passages based on the provided query.

**Parameters:**

- `query` (*str*): Query string used for retrieval.
- `k` (*int*, *optional*): Number of passages to retrieve. Defaults to 10.
- `simplify` (*bool*, *optional*): Flag for simplifying output to a list of strings. Defaults to False.

**Returns:**

- `Union[list[str], list[dotdict]]`: Depending on `simplify` flag, either a list of strings representing the passage content (`True`) or a list of `dotdict` instances containing passage details (`False`).

## Quickstart

```
import dspy

colbertv2_wiki17_abstracts = dspy.ColBERTv2(url='http://20.102.90.50:2017/wiki17_abstracts')

retrieval_response = colbertv2_wiki17_abstracts('When was the first FIFA World Cup held?', k=5)

for result in retrieval_response:
    print("Text:", result['text'], "\n")
```

# dspy.ColBERTv2RetrieverLocal

This is taken from the official documentation of Colbertv2 following the paper.

You can install Colbertv2 by the following instructions from here

## Constructor

The constructor initializes the ColBERTv2 as a local retriever object. You can initialize a server instance from your ColBERTv2 local

instance using the code snippet from here

```
class ColBERTv2RetrieverLocal:
    def __init__(
        self,
        passages:List[str],
        colbert_config=None,
        load_only:bool=False):
```

**Parameters**

- `passages` (*List[str]*): List of passages to be indexed
- `colbert_config` (*ColBERTConfig*, *Optional*): colbert config for building and searching. Defaults to None.
- `load_only` (*Boolean*): whether to load the index or build and then load. Defaults to False.

The `colbert_config` object is required for ColBERTv2, and it can be imported from `from colbert.infra.config import ColBERTConfig`. You can find the descriptions of config attributes from here

## Methods

**`forward(self, query:str, k:int, **kwargs) -> Union[list[str], list[dotdict]]`**
It retrieves relevant passages from the index based on the query. If you already have a local index, then you can pass the `load_only` flag as `True` and change the `index` attribute of ColBERTConfig to the local path. Also, make sure to change the `checkpoint` attribute of ColBERTConfig to the embedding model that you used to build the index.

**Parameters:**

- `query` (*str*): Query string used for retrieval.
- `k` (*int*, *optional*): Number of passages to retrieve. Defaults to 7

It returns a `Prediction` object for each query

```
Prediction(
    pid=[33, 6, 47, 74, 48],
    passages=['No pain, no gain.', 'The best things in life are free.', 'Out of sight, out of mind.',
'To be or not to be, that is the question.', 'Patience is a virtue.']
)
```

# dspy.ColBERTv2RerankerLocal

You can also use ColBERTv2 as a reranker in DSPy.

## Constructor

```
class ColBERTv2RerankerLocal:

    def __init__(
        self,
        colbert_config=None,
        checkpoint:str='bert-base-uncased'):
```

**Parameters**

- `colbert_config` (*ColBERTConfig*, *Optional*): colbert config for building and searching. Defaults to None.
- `checkpoint` (*str*): Embedding model for embeddings the documents and query

## Methods

**`forward(self,query:str,passages:List[str])`**
Based on a query and list of passages, it reranks the passages and returns the scores along with the passages ordered in descending

Based on a query and list of passages, it reranks the passages and returns the scores along with the passages ordered in descending order based on the similarity scores.

**Parameters:**

- `query` (*str*): Query string used for reranking.
- `passages` (*List[str]*): List of passages to be reranked

It returns the similarity scores array and you can link it to the passages by

```
for idx in np.argsort(scores_arr)[::-1]:
    print(f"Passage = {passages[idx]} --> Score = {scores_arr[idx]}")
```

# retrieve.AzureCognitiveSearch

## Constructor

The constructor initializes an instance of the `AzureCognitiveSearch` class and sets up parameters for sending queries and retreiving results with the Azure Cognitive Search server.

```
class AzureCognitiveSearch:
    def __init__(
        self,
        search_service_name: str,
        search_api_key: str,
        search_index_name: str,
        field_text: str,
        field_score: str, # required field to map with "score" field in dsp framework
    ):
```

**Parameters:**

- `search_service_name` (*str*): Name of Azure Cognitive Search server.
- `search_api_key` (*str*): API Authentication token for accessing Azure Cognitive Search server.
- `search_index_name` (*str*): Name of search index in the Azure Cognitive Search server.
- `field_text` (*str*): Field name that maps to DSP "content" field.
- `field_score` (*str*): Field name that maps to DSP "score" field.

## Methods

Refer to ColBERTv2 documentation. Keep in mind there is no `simplify` flag for AzureCognitiveSearch.

AzureCognitiveSearch supports sending queries and processing the received results, mapping content and scores to a correct format for the Azure Cognitive Search server.

## Deprecation Notice

This module is scheduled for removal in future releases. Please use the AzureAISearchRM class from dspy.retrieve.azureaisearch_rm instead.For more information, refer to the updated documentation(docs/docs/deep-dive/retrieval_models_clients/Azure.mdx).

# retrieve.ChromadbRM

## Constructor

Initialize an instance of the `ChromadbRM` class, with the option to use OpenAI's embeddings or any alternative supported by chromadb, as detailed in the official chromadb embeddings documentation.

```
ChromadbRM(
    collection_name: str,
    persist_directory: str,
    embedding_function: Optional[EmbeddingFunction[Embeddable]] = OpenAIEmbeddingFunction(),
    k: int = 7,
)
```

**Parameters:**

- `collection_name` (*str*): The name of the chromadb collection.
- `persist_directory` (*str*): Path to the directory where chromadb data is persisted.
- `embedding_function` (*Optional[EmbeddingFunction[Embeddable]]*, *optional*): The function used for embedding documents and queries. Defaults to `DefaultEmbeddingFunction()` if not specified.
- `k` (*int*, *optional*): The number of top passages to retrieve. Defaults to 7.

## Methods

**`forward(self, query_or_queries: Union[str, List[str]], k: Optional[int] = None) -> dspy.Prediction`**

Search the chromadb collection for the top `k` passages matching the given query or queries, using embeddings generated via the specified `embedding_function`.

**Parameters:**

- `query_or_queries` (*Union[str, List[str]]*): The query or list of queries to search for.
- `k` (*Optional[int]*, *optional*): The number of results to retrieve. If not specified, defaults to the value set during initialization.

**Returns:**

- `dspy.Prediction`: Contains the retrieved passages, each represented as a `dotdict` with schema `[{"id": str, "score": float, "long_text": str, "metadatas": dict }]`

## Quickstart with OpenAI Embeddings

ChromadbRM have the flexibility from a variety of embedding functions as outlined in the chromadb embeddings documentation. While different options are available, this example demonstrates how to utilize OpenAI embeddings specifically.

```
from dspy.retrieve.chromadb_rm import ChromadbRM
import os
import openai
from chromadb.utils.embedding_functions import OpenAIEmbeddingFunction

embedding_function = OpenAIEmbeddingFunction(
    api_key=os.environ.get('OPENAI_API_KEY'),
    model_name="text-embedding-ada-002"
)

retriever_model = ChromadbRM(
    'your_collection_name',
    '/path/to/your/db',
    embedding_function=embedding_function,
    k=5
)
```

```
results = retriever_model("Explore the significance of quantum computing", k=5)

for result in results:
    print("Document:", result.long_text, "\n")
```

# retrieve.FaissRM

## Constructor

Initialize an instance of FaissRM by providing it with a vectorizer and a list of strings

```
FaissRM(
    document_chunks: List[str],
    vectorizer: dsp.modules.sentence_vectorizer.BaseSentenceVectorizer,
    k: int = 3
)
```

**Parameters:**

- `document_chunks` (*List[str]*): a list of strings that comprises the corpus to search. You cannot add/insert/upsert to this list after creating this FaissRM object.
- `vectorizer` (*dsp.modules.sentence_vectorizer.BaseSentenceVectorizer*, *optional*): If not provided, a dsp.modules.sentence_vectorizer.SentenceTransformersVectorizer object is created and used.
- `k` (*int*, *optional*): The number of top passages to retrieve. Defaults to 3.

## Methods

`forward(self, query_or_queries: Union[str, List[str]]) -> dspy.Prediction`

Search the FaissRM vector database for the top `k` passages matching the given query or queries, using embeddings generated via the vectorizer specified at FaissRM construction time

**Parameters:**

- `query_or_queries` (*Union[str, List[str]]*): The query or list of queries to search for.

**Returns:**

- `dspy.Prediction`: Contains the retrieved passages, each represented as a `dotdict` with a `long_text` attribute and an `index` attribute. The `index` attribute is the index in the document_chunks array provided to this FaissRM object at construction time.

## Quickstart with the default vectorizer

The **FaissRM** module provides a retriever that uses an in-memory Faiss vector database. This module does not include a vectorizer; instead it supports any subclass of **dsp.modules.sentence_vectorizer.BaseSentenceVectorizer**. If a vectorizer is not provided, an instance of **dsp.modules.sentence_vectorizer.SentenceTransformersVectorizer** is created and used by **FaissRM**. Note that the default embedding model for **SentenceTransformersVectorizer** is **all-MiniLM-L6-v2**

```
import dspy
from dspy.retrieve.faiss_rm import FaissRM

document_chunks = [
    "The superbowl this year was played between the San Francisco 49ers and the Kanasas City Chiefs",
    "Pop corn is often served in a bowl",
    "The Rice Bowl is a Chinese Restaurant located in the city of Tucson, Arizona",
    "Mars is the fourth planet in the Solar System",
    "An aquarium is a place where children can learn about marine life",
    "The capital of the United States is Washington, D.C",
    "Rock and Roll musicians are honored by being inducted in the Rock and Roll Hall of Fame",
    "Music albums were published on Long Play Records in the 70s and 80s",
    "Sichuan cuisine is a spicy cuisine from central China",
    "The interest rates for mortgages is considered to be very high in 2024",
]

frm = FaissRM(document_chunks)
turbo = dspy.OpenAI(model="gpt-3.5-turbo")
dspy.settings.configure(lm=turbo, rm=frm)
```

```
dspy.settings.configure(lm=turbo, rm=rm)
print(frm(["I am in the mood for Chinese food"]))
```

# retrieve.MilvusRM

## Constructor

Initialize an instance of the `MilvusRM` class, with the option to use OpenAI's `text-embedding-3-small` embeddings or any customized embedding function.

```
MilvusRM(
    collection_name: str,
    uri: Optional[str] = "http://localhost:19530",
    token: Optional[str] = None,
    db_name: Optional[str] = "default",
    embedding_function: Optional[Callable] = None,
    k: int = 3,
)
```

**Parameters:**

- `collection_name (str)`: The name of the Milvus collection to query against.
- `uri (str, optional)`: The Milvus connection uri. Defaults to "http://localhost:19530".
- `token (str, optional)`: The Milvus connection token. Defaults to None.
- `db_name (str, optional)`: The Milvus database name. Defaults to "default".
- `embedding_function (callable, optional)`: The function to convert a list of text to embeddings. The embedding function should take a list of text strings as input and output a list of embeddings. Defaults to None. By default, it will get OpenAI client by the environment variable OPENAI_API_KEY and use OpenAI's embedding model "text-embedding-3-small" with the default dimension.
- `k (int, optional)`: The number of top passages to retrieve. Defaults to 3.

## Methods

**`forward(self, query_or_queries: Union[str, List[str]], k: Optional[int] = None) -> dspy.Prediction`**

Search the Milvus collection for the top `k` passages matching the given query or queries, using embeddings generated via the default OpenAI embedding or the specified `embedding_function`.

**Parameters:**

- `query_or_queries` (*Union[str, List[str]]*): The query or list of queries to search for.
- `k` (*Optional[int]*, *optional*): The number of results to retrieve. If not specified, defaults to the value set during initialization.

**Returns:**

- `dspy.Prediction`: Contains the retrieved passages, each represented as a `dotdict` with schema `[{"id": str, "score": float, "long_text": str, "metadatas": dict }]`

## Quickstart

To support passage retrieval, it assumes that a Milvus collection has been created and populated with the following field:

- `text`: The text of the passage

MilvusRM uses OpenAI's `text-embedding-3-small` embedding by default or any customized embedding function. While different options are available, the examples below demonstrate how to utilize the default OpenAI embeddings and a customized embedding function using the BGE model.

**Default OpenAI Embeddings**

```
from dspy.retrieve.milvus_rm import MilvusRM
import os
```

```
os.envrion["OPENAI_API_KEY"] = "<YOUR_OPENAI_API_KEY>"

retriever_model = MilvusRM(
    collection_name="<YOUR_COLLECTION_NAME>",
    uri="<YOUR_MILVUS_URI>",
    token="<YOUR_MILVUS_TOKEN>"  # ignore this if no token is required for Milvus connection
    )

results = retriever_model("Explore the significance of quantum computing", k=5)

for result in results:
    print("Document:", result.long_text, "\n")
```

**Customized Embedding Function**

```
from dspy.retrieve.milvus_rm import MilvusRM
from sentence_transformers import SentenceTransformer

model = SentenceTransformer('BAAI/bge-base-en-v1.5')

def bge_embedding_function(texts: List[str]):
    embeddings = model.encode(texts, normalize_embeddings=True)
    return embeddings

retriever_model = MilvusRM(
    collection_name="<YOUR_COLLECTION_NAME>",
    uri="<YOUR_MILVUS_URI>",
    token="<YOUR_MILVUS_TOKEN>",  # ignore this if no token is required for Milvus connection
    embedding_function=bge_embedding_function
    )

results = retriever_model("Explore the significance of quantum computing", k=5)

for result in results:
    print("Document:", result.long_text, "\n")
```

# retrieve.MyScaleRM

## Constructor

Initializes an instance of the `MyScaleRM` class, which is designed to use MyScaleDB (a ClickHouse fork optimized for vector similarity and full-text search) to retrieve documents based on query embeddings. This class supports embedding generation using either local models or OpenAI's API and manages database interactions efficiently.

### Syntax

```
MyScaleRM(
    client: clickhouse_connect.driver.client.Client,
    table: str,
    database: str = 'default',
    metadata_columns: List[str] = ['text'],
    vector_column: str = 'vector',
    k: int = 3,
    openai_api_key: Optional[str] = None,
    openai_model: Optional[str] = None,
    local_embed_model: Optional[str] = None
)
```

## Parameters for `MyScaleRM` Constructor

- `client` (*clickhouse_connect.driver.client.Client*): A client connection to the MyScaleDB database, used to execute queries and manage interactions with the database.
- `table` (*str*): Specifies the table within MyScaleDB from which data will be retrieved. This table should be equipped with a vector column for conducting similarity searches.
- `database` (*str*, optional): The name of the database where the table is located, defaulting to `"default"`.
- `metadata_columns` (*List[str], optional*): Columns to include as metadata in the output, defaulting to `["text"]`.
- `vector_column` (*str, optional*): The column that contains vector data, used for similarity searches, defaulting to `"vector"`.
- `k` (*int, optional*): The number of closest matches to return for each query, defaulting to 3.
- `openai_api_key` (*str, optional*): API key for accessing OpenAI services, necessary if using OpenAI for embedding generation.
- `openai_model` (*str, optional*): The specific OpenAI model to use for embeddings, required if an OpenAI API key is provided.
- `local_embed_model` (*str, optional*): Specifies a local model for embedding generation, chosen if local computation is preferred.

## Methods

### `forward`

Executes a retrieval operation based on a user's query and returns the top `k` relevant results using the embeddings generated by the specified method.

### Syntax

```
def forward(self, user_query: str, k: Optional[int] = None) -> dspy.Prediction
```

## Parameters

- `user_query` (*str*): The query or list of queries for which to retrieve matching passages.
- `k` (*Optional[int], optional*): The number of top matches to retrieve. If not provided, it defaults to the `k` value set during class initialization.

## Returns

- `dspy.Prediction`: Contains the retrieved passages, formatted as a list of `dotdict` objects. Each entry includes:
    - **long_text (str)**: The text content of the retrieved passage.

## Description

## Description

The `forward` method leverages the MyScaleDB's vector search capabilities to find the top `k` passages that best match the provided query. This method is integral for utilizing the MyScaleRM class to access and retrieve data efficiently based on semantic similarity, facilitated by the chosen embedding generation technique (either via a local model or the OpenAI API).

## Quickstart

This section provides practical examples of how to instantiate and use the `MyScaleRM` class to retrieve data from MyScaleDB efficiently using text embeddings.

```
from dspy.retrieve.myscaledb_rm import MyScaleRM

MyScale_model = MyScaleRM(client=client,
                table="table_name",
                openai_api_key="sk-***",
                openai_model="embeddings_model",
                vector_column="vector_column_name",
                metadata_columns=["add_your_columns_here"],
                k=6)

MyScale_model("Please suggest me some funny movies")

passages = results.passages

# Loop through each passage and print the 'long_text'
for passage in passages:
    print(passage['long_text'], "\n")
```

# retrieve.neo4j_rm

## Constructor

Initialize an instance of the `Neo4jRM` class.

```
Neo4jRM(
    index_name: str,
    text_node_property: str,
    k: int = 5,
    retrieval_query: str = None,
    embedding_provider: str = "openai",
    embedding_model: str = "text-embedding-ada-002",
)
```

**Environment Variables:**

You need to define the credentials as environment variables:

- `NEO4J_USERNAME` (*str*): Specifies the username required for authenticating with the Neo4j database. This is a crucial security measure to ensure that only authorized users can access the database.
- `NEO4J_PASSWORD` (*str*): Defines the password associated with the `NEO4J_USERNAME` for authentication purposes. This password should be kept secure to prevent unauthorized access to the database.
- `NEO4J_URI` (*str*): Indicates the Uniform Resource Identifier (URI) used to connect to the Neo4j database. This URI typically includes the protocol, hostname, and port, providing the necessary information to establish a connection to the database.
- `NEO4J_DATABASE` (*str*, optional): Specifies the name of the database to connect to within the Neo4j instance. If not set, the system defaults to using `"neo4j"` as the database name. This allows for flexibility in connecting to different databases within a single Neo4j server.
- `OPENAI_API_KEY` (*str*): Specifies the API key required for authenticiating with OpenAI's services.

**Parameters:**

- `index_name` (*str*): Specifies the name of the vector index to be used within Neo4j for organizing and querying data.
- `text_node_property` (*str*, *optional*): Defines the specific property of nodes that will be returned.
- `k` (*int*, *optional*): The number of top results to return from the retrieval operation. It defaults to 5 if not explicitly specified.
- `retrieval_query` (*str*, *optional*): A custom query string provided for retrieving data. If not provided, a default query tailored to the `text_node_property` will be used.
- `embedding_provider` (*str*, *optional*): The name of the service provider for generating embeddings. Defaults to "openai" if not specified.
- `embedding_model` (*str*, *optional*): The specific embedding model to use from the provider. By default, it uses the "text-embedding-ada-002" model from OpenAI.

## Methods

**`forward(self, query: [str], k: Optional[int] = None) -> dspy.Prediction`**

Search the neo4j vector index for the top `k` passages matching the given query or queries, using embeddings generated via the specified `embedding_model`.

**Parameters:**

- `query` (str_): The query.
- `k` (*Optional[int]*, *optional*): The number of results to retrieve. If not specified, defaults to the value set during initialization.

**Returns:**

**Returns:**

- `dspy.Prediction`: Contains the retrieved passages as a list of string with the prediction signature.

ex:

```
Prediction(
    passages=['Passage 1 Lorem Ipsum awesome', 'Passage 2 Lorem Ipsum Youppidoo', 'Passage 3 Lorem Ipsum Yasssss']
)
```

## Quick Example how to use Neo4j in a local environment.

```
from dspy.retrieve.neo4j_rm import Neo4jRM
import os

os.environ["NEO4J_URI"] = 'bolt://localhost:7687'
os.environ["NEO4J_USERNAME"] = 'neo4j'
os.environ["NEO4J_PASSWORD"] = 'password'
os.environ["OPENAI_API_KEY"] = 'sk-'

retriever_model = Neo4jRM(
    index_name="vector",
    text_node_property="text"
)

results = retriever_model("Explore the significance of quantum computing", k=3)

for passage in results:
    print("Document:", passage, "\n")
```

# retrieve.RAGatouilleRM

## Constructor

The constructor initializes the `RAGatouille` class instance and sets up the required parameters for interacting with the index created using RAGatouille library.

```
class RAGatouilleRM(dspy.Retrieve):
    def __init__(
        self,
        index_root: str,
        index_name: str,
        k: int = 3,
    ):
```

**Parameters:**

- `index_root` (*str*): Folder path where your index is stored.
- `index_name` (*str*): Name of the index you want to retrieve from.
- `k` (*int*): The default number of passages to retrieve. Defaults to `3`.

## Methods

**`forward(self, query_or_queries: Union[str, List[str]], k:Optional[int]) -> dspy.Prediction`**

Enables making queries to the RAGatouille-made index for retrieval. Internally, the method handles the specifics of preparing the query to obtain the response. The function handles the retrieval of the top-k passages based on the provided query.

**Parameters:**

- `query_or_queries` (Union[str, List[str]]): Query string used for retrieval.
- `k` (*int*, *optional*): Number of passages to retrieve. Defaults to 3.

**Returns:**

- `dspy.Prediction`: List of k passages

# retrieve.SnowflakeRM

## Constructor

Initialize an instance of the `SnowflakeRM` class, with the option to use `e5-base-v2` or `snowflake-arctic-embed-m` embeddings or any other Snowflake Cortex supported embeddings model.

```
SnowflakeRM(
     snowflake_table_name: str,
     snowflake_credentials: dict,
     k: int = 3,
     embeddings_field: str,
     embeddings_text_field:str,
     embeddings_model: str = "e5-base-v2",
)
```

**Parameters:**

- `snowflake_table_name (str)`: The name of the Snowflake table containing embeddings.
- `snowflake_credentials (dict)`: The connection parameters needed to initialize a Snowflake Snowpark Session.
- `k (int, optional)`: The number of top passages to retrieve. Defaults to 3.
- `embeddings_field (str)`: The name of the column in the Snowflake table containing the embeddings.
- `embeddings_text_field (str)`: The name of the column in the Snowflake table containing the passages.
- `embeddings_model (str)`: The model to be used to convert text to embeddings

## Methods

**`forward(self, query_or_queries: Union[str, List[str]], k: Optional[int] = None) -> dspy.Prediction`**

Search the Snowflake table for the top `k` passages matching the given query or queries, using embeddings generated via the default `e5-base-v2` model or the specified `embedding_model`.

**Parameters:**

- `query_or_queries` (*Union[str, List[str]]*): The query or list of queries to search for.
- `k` (*Optional[int]*, *optional*): The number of results to retrieve. If not specified, defaults to the value set during initialization.

**Returns:**

- `dspy.Prediction`: Contains the retrieved passages, each represented as a `dotdict` with schema `[{"id": str, "score": float, "long_text": str, "metadatas": dict }]`

## Quickstart

To support passage retrieval, it assumes that a Snowflake table has been created and populated with the passages in a column `embeddings_text_field` and the embeddings in another column `embeddings_field`

SnowflakeRM uses `e5-base-v2` embeddings model by default or any Snowflake Cortex supported embeddings model.

**Default OpenAI Embeddings**

```
from dspy.retrieve.snowflake_rm import SnowflakeRM
import os

connection_parameters = {

    "account": os.getenv('SNOWFLAKE_ACCOUNT'),
    "user": os.getenv('SNOWFLAKE_USER'),
    "password": os.getenv('SNOWFLAKE_PASSWORD'),
    "role": os.getenv('SNOWFLAKE_ROLE'),
    "warehouse": os.getenv('SNOWFLAKE_WAREHOUSE'),
    "database": os.getenv('SNOWFLAKE_DATABASE')
```

```
    "database": os.getenv('SNOWFLAKE_DATABASE'),
    "schema": os.getenv('SNOWFLAKE_SCHEMA')}

retriever_model = SnowflakeRM(
    snowflake_table_name="<YOUR_SNOWFLAKE_TABLE_NAME>",
    snowflake_credentials=connection_parameters,
    embeddings_field="<YOUR_EMBEDDINGS_COLUMN_NAME>",
    embeddings_text_field= "<YOUR_PASSAGE_COLUMN_NAME>"
    )

results = retriever_model("Explore the meaning of life", k=5)

for result in results:
    print("Document:", result.long_text, "\n")
```

# retrieve.WatsonDiscoveryRM

## Constructor

The constructor initializes the `WatsonDiscoveryRM` class instance and sets up the request parameters for interacting with Watson Discovery service at IBM Cloud.

```
class WatsonDiscoveryRM:
    def __init__(
        self,
        apikey: str,
        url:str,
        version:str,
        project_id:str,
        collection_ids:list=[],
        k: int = 7,
    ):
```

**Parameters:**

- `apikey` (str): apikey for authentication purposes,
- `url` (str): endpoint URL that includes the service instance ID
- `version` (str): Release date of the version of the API you want to use. Specify dates in YYYY-MM-DD format.
- `project_id` (str): The Universally Unique Identifier (UUID) of the project.
- `collection_ids` (list): An array containing the collections on which the search will be executed.
- `k` (int, optional): The number of top passages to retrieve. Defaults to 7.

## Methods

**`forward(self, query_or_queries: Union[str, list[str]], k: Optional[int]= None) -> dspy.Prediction:`**
Search the Watson Discovery collection for the top `k` passages matching the given query or queries.

**Parameters:**

- `query_or_queries` (*Union[str, list[str]]*): The query or list of queries to search for.
- `k` (*Optional[int], optional*): The number of results to retrieve. If not specified, defaults to the value set during initialization.

**Returns:**

- `dspy.Prediction`: Contains the retrieved passages, each represented as a `dotdict` with schema `[{"title":str, "long_text": str, "passage_score": float, "document_id": str, "collection_id": str, "start_offset": int, "end_offset": int, "field": str}]`

## Quickstart

```
import dspy

retriever_model = WatsonDiscoveryRM(
    apikey = "Your API Key",
    url = "URL of the Watson Discovery Service",
    version = "2023-03-31",
    project_id = "Project Id",
    collection_ids = ["Collection ID"],
    k = 5
)

retrieval_response = retriever_model("Explore the significance of quantum computing",k=5)

for result in retrieval_response:
    print("Document:", result.long_text, "\n")
```

# retrieve.YouRM

## Constructor

Initialize an instance of the `YouRM` class that calls the You.com APIs for web-based document retrieval. Options available are the "Search" and "News" APIs.

```
YouRM(
    ydc_api_key: Optional[str] = None,
    k: int = 3,
    endpoint: Literal["search", "news"] = "search",
    num_web_results: Optional[int] = None,
    safesearch: Optional[Literal["off", "moderate", "strict"]] = None,
    country: Optional[str] = None,
    search_lang: Optional[str] = None,
    ui_lang: Optional[str] = None,
    spellcheck: Optional[bool] = None,
)
```

**Parameters:**

- `ydc_api_key` (Optional[str]): you.com API key, if `YDC_API_KEY` is not set in the environment
- `k` (int): If `endpoint="search"`, the max snippets to return per search hit. If `endpoint="news"`, the max articles to return.
- `endpoint` (Literal["search", "news"]): you.com endpoints
- `num_web_results` (Optional[int]): The max number of web results to return, must be under 20
- `safesearch` (Optional[Literal["off", "moderate", "strict"]]): Safesearch settings, one of "off", "moderate", "strict", defaults to moderate
- `country` (Optional[str]): Country code, ex: 'US' for United States, see API reference for more info
- `search_lang` (Optional[str]): (News API) Language codes, ex: 'en' for English, see API reference for more info
- `ui_lang` (Optional[str]): (News API) User interface language for the response, ex: 'en' for English. See API reference for more info
- `spellcheck` (Optional[bool]): (News API) Whether to spell check query or not, defaults to True

## Methods

**`forward(self, query_or_queries: Union[str, List[str]], k: Optional[int] = None) -> dspy.Prediction`**

If `endpoint="search"`, search the web for the top `k` snippets matching the given query or queries.

If `endpoint="news"`, search the web for the top `k` articles matching the given query or queries.

**Parameters:**

- `query_or_queries` (*Union[str, List[str]]*): The query or list of queries to search for.
- `k` (*Optional[int]*, *optional*): The number of results to retrieve. If not specified, defaults to the value set during initialization.

**Returns:**

- `dspy.Prediction`: Contains the retrieved passages, each represented as a `dotdict` with schema `[{"long_text": str}]`

## Quickstart

Obtain a You.com API key from https://api.you.com/.

Export this key to an environment variable `YDC_API_KEY`.

```
from dspy.retrieve.you_rm import YouRM
import os
```

```python
# The retriever obtains the API key from the `YDC_API_KEY` env var
retriever_model = YouRM(endpoint="search")

results = retriever_model("Tell me about national parks in the US", k=5)

for result in results:
    print("Document:", result.long_text, "\n")
```

# dspy.OpenAI

## Usage

```
lm = dspy.OpenAI(model='gpt-3.5-turbo')
```

## Constructor

The constructor initializes the base class `LM` and verifies the provided arguments like the `api_provider`, `api_key`, and `api_base` to set up OpenAI request retrieval. The `kwargs` attribute is initialized with default values for relevant text generation parameters needed for communicating with the GPT API, such as `temperature`, `max_tokens`, `top_p`, `frequency_penalty`, `presence_penalty`, and `n`.

```
class OpenAI(LM):
    def __init__(
        self,
        model: str = "text-davinci-002",
        api_key: Optional[str] = None,
        api_provider: Literal["openai"] = "openai",
        model_type: Literal["chat", "text"] = None,
        **kwargs,
    ):
```

**Parameters:**

- `api_key` (*Optional[str]*, *optional*): API provider authentication token. Defaults to None.
- `api_provider` (*Literal["openai"]*, *optional*): API provider to use. Defaults to "openai".
- `model_type` (*Literal["chat", "text"]*): Specified model type to use.
- `**kwargs`: Additional language model arguments to pass to the API provider.

## Methods

**`__call__(self, prompt: str, only_completed: bool = True, return_sorted: bool = False, **kwargs) -> List[Dict[str, Any]]`**

Retrieves completions from OpenAI by calling `request`.

Internally, the method handles the specifics of preparing the request prompt and corresponding payload to obtain the response.

After generation, the completions are post-processed based on the `model_type` parameter. If the parameter is set to 'chat', the generated content look like `choice["message"]["content"]`. Otherwise, the generated text will be `choice["text"]`.

**Parameters:**

- `prompt` (*str*): Prompt to send to OpenAI.
- `only_completed` (*bool*, *optional*): Flag to return only completed responses and ignore completion due to length. Defaults to True.
- `return_sorted` (*bool*, *optional*): Flag to sort the completion choices using the returned averaged log-probabilities. Defaults to False.
- `**kwargs`: Additional keyword arguments for completion request.

**Returns:**

- `List[Dict[str, Any]]`: List of completion choices.

# dspy.AzureOpenAI

## Usage

```
lm = dspy.AzureOpenAI(api_base='...', api_version='2023-12-01-preview', model='gpt-3.5-turbo')
```

## Constructor

The constructor initializes the base class `LM` and verifies the provided arguments like the `api_provider`, `api_key`, and `api_base` to set up OpenAI request retrieval through Azure. The `kwargs` attribute is initialized with default values for relevant text generation parameters needed for communicating with the GPT API, such as `temperature`, `max_tokens`, `top_p`, `frequency_penalty`, `presence_penalty`, and `n`.

Azure requires that the deployment id of the Azure deployment to be also provided using the argument `deployment_id`.

```
class AzureOpenAI(LM):
    def __init__(
        self,
        api_base: str,
        api_version: str,
        model: str = "gpt-3.5-turbo-instruct",
        api_key: Optional[str] = None,
        model_type: Literal["chat", "text"] = None,
        **kwargs,
    ):
```

**Parameters:**

- `api_base` (str): Azure Base URL.
- `api_version` (str): Version identifier for Azure OpenAI API.
- `api_key` (*Optional[str]*, *optional*): API provider authentication token. Retrieves from `AZURE_OPENAI_KEY` environment variable if None.
- `model_type` (*Literal["chat", "text"]*): Specified model type to use, defaults to 'chat'.
- `**kwargs`: Additional language model arguments to pass to the API provider.

## Methods

**`__call__(self, prompt: str, only_completed: bool = True, return_sorted: bool = False, **kwargs) -> List[Dict[str, Any]]`**

Retrieves completions from Azure OpenAI Endpoints by calling `request`.

Internally, the method handles the specifics of preparing the request prompt and corresponding payload to obtain the response.

After generation, the completions are post-processed based on the `model_type` parameter. If the parameter is set to 'chat', the generated content look like `choice["message"]["content"]`. Otherwise, the generated text will be `choice["text"]`.

**Parameters:**

- `prompt` (*str*): Prompt to send to Azure OpenAI.
- `only_completed` (*bool*, *optional*): Flag to return only completed responses and ignore completion due to length. Defaults to True.
- `return_sorted` (*bool*, *optional*): Flag to sort the completion choices using the returned averaged log-probabilities. Defaults to False.
- `**kwargs`: Additional keyword arguments for completion request.

**Returns:**

- `List[Dict[str, Any]]`: List of completion choices.

- `List[Dict[str, Any]]`: List of completion choices.

# dspy.Cohere

## Usage

```
lm = dspy.Cohere(model='command-nightly')
```

## Constructor

The constructor initializes the base class `LM` and verifies the `api_key` to set up Cohere request retrieval.

```
class Cohere(LM):
    def __init__(
        self,
        model: str = "command-nightly",
        api_key: Optional[str] = None,
        stop_sequences: List[str] = [],
    ):
```

**Parameters:**

- `model` (*str*): Cohere pretrained models. Defaults to `command-nightly`.
- `api_key` (*Optional[str]*, *optional*): API provider from Cohere. Defaults to None.
- `stop_sequences` (*List[str]*, *optional*): List of stopping tokens to end generation.

## Methods

Refer to `dspy.OpenAI` documentation.

# dspy.HFClientTGI

## Usage

```
lm = dspy.HFClientTGI(model="meta-llama/Llama-2-7b-hf", port=8080, url="http://localhost")
```

## Prerequisites

Refer to the Text Generation-Inference Server section of the `Using Local Models` documentation.

## Constructor

The constructor initializes the `HFModel` base class and configures the client for communicating with the TGI server. It requires a `model` instance, communication `port` for the server, and the `url` for the server to host generate requests. Additional configuration can be provided via keyword arguments in `**kwargs`.

```
class HFClientTGI(HFModel):
    def __init__(self, model, port, url="http://future-hgx-1", **kwargs):
```

**Parameters:**

- `model` (*HFModel*): Instance of Hugging Face model connected to the TGI server.
- `port` (*int*): Port for TGI server.
- `url` (*str*): Base URL where the TGI server is hosted.
- `**kwargs`: Additional keyword arguments to configure the client.

## Methods

Refer to `dspy.OpenAI` documentation.

# dspy.HFClientVLLM

## Usage

```
lm = dspy.HFClientVLLM(model="meta-llama/Llama-2-7b-hf", port=8080, url="http://localhost")
```

## Prerequisites

Refer to the vLLM Server section of the `Using Local Models` documentation.

## Constructor

Refer to `dspy.TGI` documentation. Replace with `HFClientVLLM`.

## Methods

Refer to `dspy.OpenAI` documentation.

# dspy.PremAI

PremAI is an all-in-one platform that simplifies the process of creating robust, production-ready applications powered by Generative AI. By streamlining the development process, PremAI allows you to concentrate on enhancing user experience and driving overall growth for your application.

## Prerequisites

Refer to the quick start guide to getting started with the PremAI platform, create your first project and grab your API key.

## Usage

Please make sure you have premai python sdk installed. Otherwise you can do it using this command:

```
pip install -U premai
```

Here is a quick example on how to use premai python sdk with DSPy

```
from dspy import PremAI

llm = PremAI(model='mistral-tiny', project_id=123, api_key="your-premai-api-key")
print(llm("what is a large language model"))
```

> Please note: Project ID 123 is just an example. You can find your project ID inside our platform under which you created your project.

## Constructor

The constructor initializes the base class `LM` and verifies the `api_key` provided or defined through the `PREMAI_API_KEY` environment variable.

```
class PremAI(LM):
    def __init__(
        self,
        model: str,
        project_id: int,
        api_key: str,
        base_url: Optional[str] = None,
        session_id: Optional[int] = None,
        **kwargs,
    ) -> None:
```

**Parameters:**

- `model` (*str*): Models supported by PremAI. Example: `mistral-tiny`. We recommend using the model selected in project launchpad.
- `project_id` (*int*): The project id which contains the model of choice.
- `api_key` (*Optional[str]*, *optional*): API provider from PremAI. Defaults to None.
- `**kwargs`: Additional language model arguments will be passed to the API provider.

## Methods

**`__call__(self, prompt: str, **kwargs) -> List[Dict[str, Any]]`**

Retrieves completions from PremAI by calling `request`.

Internally, the method handles the specifics of preparing the request prompt and corresponding payload to obtain the response.

**Parameters:**

**Parameters:**

- `prompt` (*str*): Prompt to send to PremAI.
- `**kwargs`: Additional keyword arguments for completion request. Example: parameters like `temperature`, `max_tokens` etc. You can find all the additional kwargs here.

## Prem Templates

Writing Prompt Templates can be super messy. Prompt templates are long, hard to manage, and must be continuously tweaked to improve and keep the same throughout the application.

With **Prem**, writing and managing prompts can be super easy. The ***Templates*** tab inside the launchpad helps you write as many prompts as you need and use it inside the SDK to make your application run using those prompts. You can read more about Prompt Templates here.

Using templates in DSPy is quite easy. First, here is an example of a prompt template that you store / re-iterate inside Prem.

```
template = """
Summarize the following content by creating a bullet point list and return it in json
${content}
"""
```

**Please note:** Prompt templates are not defined in Python code. Here is an example of a Template, which has an associated template id.

Assuming this prompt template is stored under a template with id: `78069ce8-xxxxx-xxxxx-xxxx-xxx`:

```
from dspy import PremAI

client = PremAI(project_id=1234)
template_id = "78069ce8-xxxxx-xxxxx-xxxx-xxx"

msg = """
I am playing in a garden on a fine sunday
evening. I then went to my friend Tara's place
and then went for movies in the city center mall.
"""

response = client(
    prompt="some-random-dummy-text",
    template_id=template_id,
    params={"content": msg}
)

print(response)
```

When you use templates, there is no need to place `msg` / prompt inside the `prompt` argument. DSPy only accepts string prompts to conduct LLM requests, but we might require multiple string inputs when using templates. Templates allow you to provide additional inputs (key:value pairs) within the `params` argument, which should be mapped with the template variable. Our example template had one variable `content`, so our params become: `{"content": msg}`.

## Native RAG Support

PremAI Repositories allow users to upload documents (.txt, .pdf, etc.) and connect those repositories to the LLMs to serve as vector databases and support native RAG. You can learn more about PremAI repositories here.

Repositories are also supported through the dspy-premai integration. Here is how you can use this workflow:

```
query = "what is the diameter of individual Galaxy"
repository_ids = [1991, ]
repositories = dict(
```

```
    ids=repository_ids,
    similarity_threshold=0.3,
    limit=3
)
```

First, we start by defining our repository with some valid repository ids. You can learn more about how to get the repository id here.

> Note: This is similar to LM integrations where now you are overriding the repositories connected in the launchpad when you invoke the argument' repositories'.

Now, we connect the repository with our chat object to invoke RAG-based generations.

```
response = llm(query, max_tokens=100, repositories=repositories)

print(response)
print("---")
print(json.dumps(llm.history, indent=4))
```

Here is what an example generation would look like with PremAI Repositories.

```
'The diameters of individual galaxies range from 80,000-150,000 light-years.'
---
[
    {
        "prompt": "what is the diameter of individual Galaxy",
        "response": "The diameters of individual galaxies range from 80,000-150,000 light-years.",
        "document_chunks": [
            {
                "repository_id": 1991,
                "document_id": 1307,
                "chunk_id": 173926,
                "document_name": "Kegy 202 Chapter 2",
                "similarity_score": 0.586126983165741,
                "content": "n thousands\n
of           light-years. The diameters of individual\n
galaxies range from 80,000-150,000 light\n
"
            },`
        ],
        "kwargs": {
            "max_tokens": 100,
            "repositories": {
                "ids": [
                    1991
                ],
                "similarity_threshold": 0.3,
                "limit": 3
            }
        },
        "raw_kwargs": {
            "max_tokens": 100,
            "repositories": {
                "ids": [
                    1991
                ],
                "similarity_threshold": 0.3,
                "limit": 3
            }
        }
    }
]
```

So this also means that you do not need to create your own RAG pipeline when using the PremAI Platform and can instead take advantage of its local RAG technology to deliver best-in-class performance for Retrieval Augmented Generations.

> Ideally, you do not need to connect Repository IDs here to get Retrieval Augmented Generations. You can still get the same result if you have connected the repositories in PremAI platform.

# dspy.Anyscale

## Usage

```
lm = dspy.Anyscale(model="mistralai/Mistral-7B-Instruct-v0.1")
```

## Constructor

The constructor initializes the base class `LM` and verifies the `api_key` for using Anyscale API. We expect the following environment variables to be set:

- `ANYSCALE_API_KEY`: API key for Together.
- `ANYSCALE_API_BASE`: API base URL for Together.

```
class Anyscale(HFModel):
    def __init__(self, model, **kwargs):
```

**Parameters:**

- `model` (*str*): models hosted on Together.

## Methods

Refer to `dspy.OpenAI` documentation.

# dspy.Together

## Usage

```
lm = dspy.Together(model="mistralai/Mistral-7B-v0.1")
```

## Constructor

The constructor initializes the base class `LM` and verifies the `api_key` for using Together API. We expect the following environment variables to be set:

- `TOGETHER_API_KEY`: API key for Together.
- `TOGETHER_API_BASE`: API base URL for Together.

```
class Together(HFModel):
    def __init__(self, model, **kwargs):
```

**Parameters:**

- `model` (*str*): models hosted on Together.
- `stop` (*List[str]*, *optional*): List of stopping tokens to end generation.

## Methods

Refer to `dspy.OpenAI` documentation.

# dspy.Databricks

## Usage

```
lm = dspy.Databricks(model="databricks-mpt-30b-instruct")
```

## Constructor

The constructor inherits from the `GPT3` class and verifies the Databricks authentication credentials for using Databricks Model Serving API through the OpenAI SDK. We expect the following environment variables to be set:

- `openai.api_key`: Databricks API key.
- `openai.base_url`: Databricks Model Endpoint url

The `kwargs` attribute is initialized with default values for relevant text generation parameters needed for communicating with the Databricks OpenAI SDK, such as `temperature`, `max_tokens`, `top_p`, and `n`. However, it removes the `frequency_penalty` and `presence_penalty` arguments as these are not currently supported by the Databricks API.

```
class Databricks(GPT3):
    def __init__(
        self,
        model: str,
        api_key: Optional[str] = None,
        api_base: Optional[str] = None,
        model_type: Literal["chat", "text"] = None,
        **kwargs,
    ):
```

**Parameters:**

- `model` (*str*): models hosted on Databricks.
- `stop` (*List[str], optional*): List of stopping tokens to end generation.
- `api_key` (*Optional[str]*): Databricks API key. Defaults to None
- `api_base` (*Optional[str]*): Databricks Model Endpoint url Defaults to None.
- `model_type` (*Literal["chat", "text", "embeddings"]*): Specified model type to use.
- `**kwargs`: Additional language model arguments to pass to the API provider.

## Methods

Refer to `dspy.OpenAI` documentation.

# dspy.GROQ

## Usage

```
lm = dspy.GROQ(model='mixtral-8x7b-32768', api_key ="gsk_***" )
```

## Constructor

The constructor initializes the base class `LM` and verifies the provided arguments like the `api_key` for GROQ api retriver. The `kwargs` attribute is initialized with default values for relevant text generation parameters needed for communicating with the GROQ API, such as `temperature`, `max_tokens`, `top_p`, `frequency_penalty`, `presence_penalty`, and `n`.

```
class GroqLM(LM):
    def __init__(
        self,
        api_key: str,
        model: str = "mixtral-8x7b-32768",
        **kwargs,
    ):
```

**Parameters:**

- `api_key` str: API provider authentication token. Defaults to None.
- `model` str: model name. Defaults to "mixtral-8x7b-32768' options: ['llama2-70b-4096', 'gemma-7b-it']
- `**kwargs`: Additional language model arguments to pass to the API provider.

## Methods

**`def __call__(self, prompt: str, only_completed: bool = True, return_sorted: bool = False, **kwargs, ) -> list[dict[str, Any]]:`**

Retrieves completions from GROQ by calling `request`.

Internally, the method handles the specifics of preparing the request prompt and corresponding payload to obtain the response.

After generation, the generated content look like `choice["message"]["content"]`.

**Parameters:**

- `prompt` (*str*): Prompt to send to GROQ.
- `only_completed` (*bool*, *optional*): Flag to return only completed responses and ignore completion due to length. Defaults to True.
- `return_sorted` (*bool*, *optional*): Flag to sort the completion choices using the returned averaged log-probabilities. Defaults to False.
- `**kwargs`: Additional keyword arguments for completion request.

**Returns:**

- `List[Dict[str, Any]]`: List of completion choices.

# dspy.Mistral

## Usage

```
lm = dspy.Mistral(model='mistral-medium-latest', api_key="your-mistralai-api-key")
```

## Constructor

The constructor initializes the base class `LM` and verifies the `api_key` provided or defined through the `MISTRAL_API_KEY` environment variable.

```
class Mistral(LM):
    def __init__(
        self,
        model: str = "mistral-medium-latest",
        api_key: Optional[str] = None,
        **kwargs,
    ):
```

**Parameters:**

- `model` (*str*): Mistral AI pretrained models. Defaults to `mistral-medium-latest`.
- `api_key` (*Optional[str]*, *optional*): API provider from Mistral AI. Defaults to None.
- `**kwargs`: Additional language model arguments to pass to the API provider.

## Methods

Refer to `dspy.OpenAI` documentation.

# dspy.AWSMistral, dspy.AWSAnthropic, dspy.AWSMeta

## Usage

```
# Notes:
# 1. Install boto3 to use AWS models.
# 2. Configure your AWS credentials with the AWS CLI before using these models

# initialize the bedrock aws provider
bedrock = dspy.Bedrock(region_name="us-west-2")
# For mixtral on Bedrock
lm = dspy.AWSMistral(bedrock, "mistral.mixtral-8x7b-instruct-v0:1", **kwargs)
# For haiku on Bedrock
lm = dspy.AWSAnthropic(bedrock, "anthropic.claude-3-haiku-20240307-v1:0", **kwargs)
# For llama2 on Bedrock
lm = dspy.AWSMeta(bedrock, "meta.llama2-13b-chat-v1", **kwargs)

# initialize the sagemaker aws provider
sagemaker = dspy.Sagemaker(region_name="us-west-2")
# For mistral on Sagemaker
# Note: you need to create a Sagemaker endpoint for the mistral model first
lm = dspy.AWSMistral(sagemaker, "<YOUR_MISTRAL_ENDPOINT_NAME>", **kwargs)
```

## Constructor

The `AWSMistral` constructor initializes the base class `AWSModel` which itself inherits from the `LM` class.

```
class AWSMistral(AWSModel):
    """Mistral family of models."""

    def __init__(
        self,
        aws_provider: AWSProvider,
        model: str,
        max_context_size: int = 32768,
        max_new_tokens: int = 1500,
        **kwargs
    ) -> None:
```

**Parameters:**

- `aws_provider` (AWSProvider): The aws provider to use. One of `dspy.Bedrock` or `dspy.Sagemaker`.
- `model` (*str*): Mistral AI pretrained models. For Bedrock, this is the Model ID in https://docs.aws.amazon.com/bedrock/latest/userguide/model-ids.html#model-ids-arns. For Sagemaker, this is the endpoint name.
- `max_context_size` (*Optional[int]*, *optional*): Max context size for this model. Defaults to 32768.
- `max_new_tokens` (*Optional[int]*, *optional*): Max new tokens possible for this model. Defaults to 1500.
- `**kwargs`: Additional language model arguments to pass to the API provider.

## Methods

```
def _format_prompt(self, raw_prompt: str) -> str:
```

This function formats the prompt for the model. Refer to the model card for the specific formatting required.

```
def _create_body(self, prompt: str, **kwargs) -> tuple[int, dict[str, str | float]]:
```

This function creates the body of the request to the model. It takes the prompt and any additional keyword arguments and returns a tuple of the number of tokens to generate and a dictionary of keys including the prompt used to create the body of the request.

```
def _call_model(self, body: str) -> str:
```

This function calls the model using the provider `call_model()` function and extracts the generated text (completion) from the provider-specific response.

The above model-specific methods are called by the `AWSModel::basic_request()` method, which is the main method for querying the model. This method takes the prompt and any additional keyword arguments and calls the `AWSModel::_simple_api_call()` which then delegates to the model-specific `_create_body()` and `_call_model()` methods to create the body of the request, call the model and extract the generated text.

Refer to `dspy.OpenAI` documentation for information on the `LM` base class functionality.

`AWSAnthropic` and `AWSMeta` work exactly the same as `AWSMistral`.

# dspy.Bedrock, dspy.Sagemaker

## Usage

The `AWSProvider` class is the base class for the AWS providers - `dspy.Bedrock` and `dspy.Sagemaker`. An instance of one of these providers is passed to the constructor when creating an instance of an AWS model class (e.g., `dspy.AWSMistral`) that is ultimately used to query the model.

```
# Notes:
# 1. Install boto3 to use AWS models.
# 2. Configure your AWS credentials with the AWS CLI before using these models

# initialize the bedrock aws provider
bedrock = dspy.Bedrock(region_name="us-west-2")

# initialize the sagemaker aws provider
sagemaker = dspy.Sagemaker(region_name="us-west-2")
```

## Constructor

The `Bedrock` constructor initializes the base class `AWSProvider`.

```
class Bedrock(AWSProvider):
    """This class adds support for Bedrock models."""

    def __init__(
        self,
        region_name: str,
        profile_name: Optional[str] = None,
        batch_n_enabled: bool = False,    # This has to be setup manually on Bedrock.
    ) -> None:
```

**Parameters:**

- `region_name` (str): The AWS region where this LM is hosted.
- `profile_name` (str, optional): boto3 credentials profile.
- `batch_n_enabled` (bool): If False, call the LM N times rather than batching.

## Methods

```
def call_model(self, model_id: str, body: str) -> str:
```

This function implements the actual invocation of the model on AWS using the boto3 provider.

`Sagemaker` works exactly the same as `Bedrock`.

# dspy.CloudflareAI

## Usage

```
lm = dspy.CloudflareAI(model="@hf/meta-llama/meta-llama-3-8b-instruct")
```

## Constructor

The constructor initializes the base class `LM` and verifies the `api_key` and `account_id` for using Cloudflare AI API. The following environment variables are expected to be set or passed as arguments:

- `CLOUDFLARE_ACCOUNT_ID`: Account ID for Cloudflare.
- `CLOUDFLARE_API_KEY`: API key for Cloudflare.

```
class CloudflareAI(LM):
  def __init__(
          self,
          model: str = "@hf/meta-llama/meta-llama-3-8b-instruct",
          account_id: Optional[str] = None,
          api_key: Optional[str] = None,
          system_prompt: Optional[str] = None,
          **kwargs,
      ):
```

**Parameters:**

- `model` (*str*): Model hosted on Cloudflare. Defaults to `@hf/meta-llama/meta-llama-3-8b-instruct`.
- `account_id` (*Optional[str]*, *optional*): Account ID for Cloudflare. Defaults to None. Reads from environment variable `CLOUDFLARE_ACCOUNT_ID`.
- `api_key` (*Optional[str]*, *optional*): API key for Cloudflare. Defaults to None. Reads from environment variable `CLOUDFLARE_API_KEY`.
- `system_prompt` (*Optional[str]*, *optional*): System prompt to use for generation.

## Methods

Refer to `dspy.OpenAI` documentation.

# dspy.GoogleVertexAI

This guide provides instructions on how to use the `GoogleVertexAI` class to interact with Google Vertex AI's API for text and code generation.

## Requirements

- Python 3.10 or higher.
- The `vertexai` package installed, which can be installed via pip.
- A Google Cloud account and a configured project with access to Vertex AI.

## Installation

Ensure you have installed the `vertexai` package along with other necessary dependencies:

```
pip install dspy-ai[google-vertex-ai]
```

## Configuration

Before using the `GoogleVertexAI` class, you need to set up access to Google Cloud:

1. Create a project in Google Cloud Platform (GCP).
2. Enable the Vertex AI API for your project.
3. Create authentication credentials and save them in a JSON file.

## Usage

Here's an example of how to instantiate the `GoogleVertexAI` class and send a text generation request:

```
from dsp.modules import GoogleVertexAI  # Import the GoogleVertexAI class

# Initialize the class with the model name and parameters for Vertex AI
vertex_ai = GoogleVertexAI(
    model_name="text-bison@002",
    project="your-google-cloud-project-id",
    location="us-central1",
    credentials="path-to-your-service-account-file.json"
)
```

## Customizing Requests

You can customize requests by passing additional parameters such as `temperature`, `max_output_tokens`, and others supported by the Vertex AI API. This allows you to control the behavior of the text generation.

## Important Notes

- Make sure you have correctly set up access to Google Cloud to avoid authentication issues.
- Be aware of the quotas and limits of the Vertex AI API to prevent unexpected interruptions in service.

With this guide, you're ready to use `GoogleVertexAI` for interacting with Google Vertex AI's text and code generation services.

# dspy.Snowflake

## Usage

```
import dspy
import os

connection_parameters = {

    "account": os.getenv('SNOWFLAKE_ACCOUNT'),
    "user": os.getenv('SNOWFLAKE_USER'),
    "password": os.getenv('SNOWFLAKE_PASSWORD'),
    "role": os.getenv('SNOWFLAKE_ROLE'),
    "warehouse": os.getenv('SNOWFLAKE_WAREHOUSE'),
    "database": os.getenv('SNOWFLAKE_DATABASE'),
    "schema": os.getenv('SNOWFLAKE_SCHEMA')}

lm = dspy.Snowflake(model="mixtral-8x7b",credentials=connection_parameters)
```

## Constructor

The constructor inherits from the base class `LM` and verifies the `credentials` for using Snowflake API.

```
class Snowflake(LM):
    def __init__(
        self,
        model,
        credentials,
        **kwargs):
```

**Parameters:**

- `model` (*str*): model hosted by Snowflake Cortex.
- `credentials` (*dict*): connection parameters required to initialize a snowflake snowpark session

## Methods

Refer to `dspy.Snowflake` documentation.

# dspy.Watsonx

This guide provides instructions on how to use the `Watsonx` class to interact with IBM Watsonx.ai API for text and code generation.

## Requirements

- Python 3.10 or higher.
- The `ibm-watsonx-ai` package installed, which can be installed via pip.
- An IBM Cloud account and a Watsonx configured project.

## Installation

Ensure you have installed the `ibm-watsonx-ai` package along with other necessary dependencies:

## Configuration

Before using the `Watsonx` class, you need to set up access to IBM Cloud:

1. Create an IBM Cloud account
2. Enable a Watsonx service from the catalog
3. Create a new project and associate a Watson Machine Learning service instance.
4. Create an IAM authentication credentials and save them in a JSON file.

## Usage

Here's an example of how to instantiate the `Watsonx` class and send a generation request:

```
import dspy

''' Initialize the class with the model name and parameters for Watsonx.ai
    You can choose between many different models:
    * (Mistral) mistralai/mixtral-8x7b-instruct-v01
    * (Meta) meta-llama/llama-3-70b-instruct
    * (IBM) ibm/granite-13b-instruct-v2
    * and many others.
'''
watsonx=dspy.Watsonx(
    model='mistralai/mixtral-8x7b-instruct-v01',
    credentials={
        "apikey": "your-api-key",
        "url": "https://us-south.ml.cloud.ibm.com"
    },
    project_id="your-watsonx-project-id",
    max_new_tokens=500,
    max_tokens=1000
    )

dspy.settings.configure(lm=watsonx)
```

## Customizing Requests

You can customize requests by passing additional parameters such as `decoding_method`, `max_new_tokens`, `stop_sequences`, `repetition_penalty`, and others supported by the Watsonx.ai API. This allows you to control the behavior of the generation. Refer to `ibm-watsonx-ai library` documentation.

# dspy.You

Wrapper around You.com's conversational Smart and Research APIs.

Each API endpoint is designed to generate conversational responses to a variety of query types, including inline citations and web results when relevant.

Smart Mode:

- Quick, reliable answers for a variety of questions
- Cites the entire web page URL

Research Mode:

- In-depth answers with extensive citations for a variety of questions
- Cites the specific web page snippet relevant to the claim

For more information, check out the documentations at https://documentation.you.com/api-reference/.

## Constructor

```
You(
    endpoint: Literal["smart", "research"] = "smart",
    ydc_api_key: Optional[str] = None,
)
```

**Parameters:**

- `endpoint`: You.com conversational endpoints. Choose from "smart" or "research"
- `ydc_api_key`: You.com API key, if `YDC_API_KEY` is not set in the environment

## Usage

Obtain a You.com API key from https://api.you.com/.

Export this key to an environment variable `YDC_API_KEY`.

```
import dspy

# The API key is inferred from the `YDC_API_KEY` environment variable
lm = dspy.You(endpoint="smart")
```

# dspy.HFModel

Initialize `HFModel` within your program with the desired model to load in. Here's an example call:

```
llama = dspy.HFModel(model = 'meta-llama/Llama-2-7b-hf')
```

# dspy.ChatModuleClient

## Prerequisites

1. Install the required packages using the following commands:

```
pip install --no-deps --pre --force-reinstall mlc-ai-nightly-cu118 mlc-chat-nightly-cu118 -f
https://mlc.ai/wheels
pip install transformers
git lfs install
```

   Adjust the pip wheels according to your OS/platform by referring to the provided commands in MLC packages.

## Running MLC Llama-2 models

1. Create a directory for prebuilt models:

```
mkdir -p dist/prebuilt
```

2. Clone the necessary libraries from the repository:

```
git clone https://github.com/mlc-ai/binary-mlc-llm-libs.git dist/prebuilt/lib
cd dist/prebuilt
```

3. Choose a Llama-2 model from MLC LLMs and clone the model repository:

```
git clone https://huggingface.co/mlc-ai/mlc-chat-Llama-2-7b-chat-hf-q4f16_1
```

4. Initialize the `ChatModuleClient` within your program with the desired parameters. Here's an example call:

```
llama = dspy.ChatModuleClient(model='dist/prebuilt/mlc-chat-Llama-2-7b-chat-hf-q4f16_1',
model_path='dist/prebuilt/lib/Llama-2-7b-chat-hf-q4f16_1-cuda.so')
```

Please refer to the official MLC repository for more detailed information and documentation.

# dspy.OllamaLocal

> **NOTE**
>
> Adapted from documentation provided by https://github.com/insop

Ollama is a good software tool that allows you to run LLMs locally, such as Mistral, Llama2, and Phi. The following are the instructions to install and run Ollama.

## Prerequisites

Install Ollama by following the instructions from this page:

- https://ollama.ai

Download model: `ollama pull`

Download a model by running the `ollama pull` command. You can download Mistral, Llama2, and Phi.

```
# download mistral
ollama pull mistral
```

Here is the list of other models you can download:

- https://ollama.ai/library

## Running Ollama model

Run model: `ollama run`

You need to start the model server with the `ollama run` command.

```
# run mistral
ollama run mistral
```

## Sending requests to the server

Here is the code to load a model through Ollama:

```
lm = dspy.OllamaLocal(model='mistral')
```

# dspy.HFClientTGI

## Prerequisites

- Docker must be installed on your system. If you don't have Docker installed, you can get it from here.

## Setting up the Text-Generation-Inference Server

1. Clone the Text-Generation-Inference repository from GitHub by executing the following command:

```
git clone https://github.com/huggingface/text-generation-inference.git
```

2. Change into the cloned repository directory:

```
cd text-generation-inference
```

3. Execute the Docker command under the "Get Started" section to run the server:

```
model=meta-llama/Llama-2-7b-hf # set to the specific Hugging Face model ID you wish to use.
num_shard=2 # set to the number of shards you wish to use.
volume=$PWD/data # share a volume with the Docker container to avoid downloading weights every run

docker run --gpus all --shm-size 1g -p 8080:80 -v $volume:/data ghcr.io/huggingface/text-
generation-inference:0.9 --model-id $model --num-shard $num_shard
```

   This command will start the server and make it accessible at `http://localhost:8080`.

If you want to connect to Meta Llama 2 models, make sure to use version 9.3 (or higher) of the docker image (ghcr.io/huggingface/text-generation-inference:0.9.3) and pass in your huggingface token as an environment variable.

```
   docker run --gpus all --shm-size 1g -p 8080:80 -v $volume:/data -e HUGGING_FACE_HUB_TOKEN=
{your_token} ghcr.io/huggingface/text-generation-inference:0.9.3 --model-id $model --num-shard
$num_shard
```

## Sending requests to the server

After setting up the text-generation-inference server and ensuring that it displays "Connected" when it's running, you can interact with it using the `HFClientTGI`.

Initialize the `HFClientTGI` within your program with the desired parameters. Here is an example call:

```
lm = dspy.HFClientTGI(model="meta-llama/Llama-2-7b-hf", port=8080, url="http://localhost")
```

Customize the `model`, `port`, and `url` according to your requirements. The `model` parameter should be set to the specific Hugging Face model ID you wish to use.

### FAQs

1. If your model doesn't require any shards, you still need to set a value for `num_shard`, but you don't need to include the parameter `--num-shard` on the command line.

2. If your model runs into any "token exceeded" issues, you can set the following parameters on the command line to adjust the

2. If your model runs into any "token exceeded" issues, you can set the following parameters on the command line to adjust the input length and token limit:

    - `--max-input-length`: Set the maximum allowed input length for the text.
    - `--max-total-tokens`: Set the maximum total tokens allowed for text generation.

Please refer to the official Text-Generation-Inference repository for more detailed information and documentation.

# dspy.TensorRTModel

TensorRT LLM by Nvidia happens to be one of the most optimized inference engines to run open-source Large Language Models locally or in production.

## Prerequisites

Install TensorRT LLM by the following instructions here. You need to install `dspy` inside the same Docker environment in which `tensorrt` is installed.

In order to use this module, you should have the model weights file in engine format. To understand how we convert weights in torch (from HuggingFace models) to TensorRT engine format, you can check out this documentation.

## Running TensorRT model inside dspy

```
from dspy import TensorRTModel

engine_dir = "<your-path-to-engine-dir>"
model_name_or_path = "<hf-model-id-or-path-to-tokenizer>"

model = TensorRTModel(engine_dir=engine_dir, model_name_or_path=model_name_or_path)
```

You can perform more customization on model loading based on the following example. Below is a list of optional parameters that are supported while initializing the `dspy` TensorRT model.

- **use_py_session** (`bool`, optional): Whether to use a Python session or not. Defaults to `False`.
- **lora_dir** (`str`): The directory of LoRA adapter weights.
- **lora_task_uids** (`List[str]`): List of LoRA task UIDs; use `-1` to disable the LoRA module.
- **lora_ckpt_source** (`str`): The source of the LoRA checkpoint.

If `use_py_session` is set to `False`, the following kwargs are supported (This runs in C++ runtime):

- **max_batch_size** (`int`, optional): The maximum batch size. Defaults to `1`.
- **max_input_len** (`int`, optional): The maximum input context length. Defaults to `1024`.
- **max_output_len** (`int`, optional): The maximum output context length. Defaults to `1024`.
- **max_beam_width** (`int`, optional): The maximum beam width, similar to `n` in OpenAI API. Defaults to `1`.
- **max_attention_window_size** (`int`, optional): The attention window size that controls the sliding window attention / cyclic KV cache behavior. Defaults to `None`.
- **sink_token_length** (`int`, optional): The sink token length. Defaults to `1`.

> Please note that you need to complete the build processes properly before applying these customizations, because a lot of customization depends on how the model engine was built. You can learn more here.

Now to run the model, we need to add the following code:

```
response = model("hello")
```

This gives this result:

```
["nobody is perfect, and we all have our own unique struggles and challenges. But what sets us apart is
how we respond to those challenges. Do we let them define us, or do we use them as opportunities to
grow and learn?\nI know that I have my own personal struggles, and I'm sure you do too. But I also know
that we are capable of overcoming them, and becoming the best versions of ourselves. So let's embrace
our imperfections, and use them to fuel our growth and success.\nRemember, nobody is perfect, but
everybody has the potential to be amazing. So let's go out there and make it happen!"]
```

You can also invoke chat mode by just changing the prompt to chat format like this:

```
prompt = [{"role":"user", "content":"hello"}]
response = model(prompt)

print(response)
```

Output:

```
[" Hello! It's nice to meet you. Is there something I can help you with or would you like to chat?"]
```

Here are some optional parameters that are supported while doing generation:

- **max_new_tokens** (`int`): The maximum number of tokens to output. Defaults to `1024`.
- **max_attention_window_size** (`int`): Defaults to `None`.
- **sink_token_length** (`int`): Defaults to `None`.
- **end_id** (`int`): The end of sequence ID of the tokenizer, defaults to the tokenizer's default end ID.
- **pad_id** (`int`): The pad sequence ID of the tokenizer, defaults to the tokenizer's default end ID.
- **temperature** (`float`): The temperature to control probabilistic behavior in generation. Defaults to `1.0`.
- **top_k** (`int`): Defaults to `1`.
- **top_p** (`float`): Defaults to `1`.
- **num_beams** (`int`): The number of responses to generate. Defaults to `1`.
- **length_penalty** (`float`): Defaults to `1.0`.
- **repetition_penalty** (`float`): Defaults to `1.0`.
- **presence_penalty** (`float`): Defaults to `0.0`.
- **frequency_penalty** (`float`): Defaults to `0.0`.
- **early_stopping** (`int`): Use this only when `num_beams` > 1. Defaults to `1`.

# dspy.HFClientVLLM

## Setting up the vLLM Server

Follow these steps to set up the vLLM Server:

1. Build the server from source by following the instructions provided in the Build from Source guide.

2. Start the server by running the following command, and specify your desired model, host, and port using the appropriate arguments. The default server address is http://localhost:8000.

Example command:

```
python -m vllm.entrypoints.openai.api_server --model mosaicml/mpt-7b --port 8000
```

This will launch the vLLM server.

## Sending requests to the server

After setting up the vLLM server and ensuring that it displays "Connected" when it's running, you can interact with it using the `HFClientVLLM`.

Initialize the `HFClientVLLM` within your program with the desired parameters. Here is an example call:

```
lm = dspy.HFClientVLLM(model="mosaicml/mpt-7b", port=8000, url="http://localhost")
```

Customize the `model`, `port`, `url`, and `max_tokens` according to your requirements. The `model` parameter should be set to the specific Hugging Face model ID you wish to use.

Please refer to the official vLLM repository for more detailed information and documentation.

## Sending requests to vLLM server using dspy.OpenAI

Query the vLLM server using OpenAI SDK through `dspy.OpenAI` with your desired parameters. Here is an example call:

```
lm = dspy.OpenAI(model="mosaicml/mpt-7b", api_base="http://localhost:8000/v1/", api_key="EMPTY")
```

Similarly, customize the `model`, `port`, `url` (vLLM arguments), along with the remaining OpenAI client arguments according to your requirements. The `model` parameter should be set to the specific Hugging Face model ID you wish to use.

# HFClientVLLM

## HFClient vLLM

### Prerequisites - Launching vLLM Server locally

Refer to the vLLM Server API for setting up the vLLM server locally.

```
#Example vLLM Server Launch

 python -m vllm.entrypoints.api_server --model meta-llama/Llama-2-7b-hf --port 8080
```

This command will start the server and make it accessible at `http://localhost:8080`.

### Setting up the vLLM Client

The constructor initializes the `HFModel` base class to support the handling of prompting models, configuring the client for communicating with the hosted vLLM server to generate requests. This requires the following parameters:

- `model` (*str*): ID of model connected to the vLLM server.
- `port` (*int*): Port for communicating to the vLLM server.
- `url` (*str*): Base URL of hosted vLLM server. This will often be `"http://localhost"`.
- `**kwargs`: Additional keyword arguments to configure the vLLM client.

Example of the vLLM constructor:

```
class HFClientVLLM(HFModel):
    def __init__(self, model, port, url="http://localhost", **kwargs):
```

### Under the Hood

**`_generate(self, prompt, **kwargs) -> dict`**

**Parameters:**

- `prompt` (*str*): Prompt to send to model hosted on vLLM server.
- `**kwargs`: Additional keyword arguments for completion request.

**Returns:**

- `dict`: dictionary with `prompt` and list of response `choices`.

Internally, the method handles the specifics of preparing the request prompt and corresponding payload to obtain the response.

After generation, the method parses the JSON response received from the server and retrieves the output through `json_response["choices"]` and stored as the `completions` list.

Lastly, the method constructs the response dictionary with two keys: the original request `prompt` and `choices`, a list of dictionaries representing generated completions with the key `text` holding the response's generated text.

### Using the vLLM Client

```
vllm_llama2 = dspy.HFClientVLLM(model="meta-llama/Llama-2-7b-hf", port=8080, url="http://localhost")
```

### Sending Requests via vLLM Client

1. ***Recommended*** Configure default LM using `dspy.configure`.

This allows you to define programs in DSPy and simply call modules on your input fields, having DSPy internally call the prompt on the

configured LM.

```
dspy.configure(lm=vllm_llama2)

#Example DSPy CoT QA program
qa = dspy.ChainOfThought('question -> answer')

response = qa(question="What is the capital of Paris?") #Prompted to vllm_llama2
print(response.answer)
```

2. Generate responses using the client directly.

```
response = vllm_llama2._generate(prompt='What is the capital of Paris?')
print(response)
```

---

**Written By:** Arnav Singhvi

# Optimizers (formerly Teleprompters)

A **DSPy optimizer** is an algorithm that can tune the parameters of a DSPy program (i.e., the prompts and/or the LM weights) to maximize the metrics you specify, like accuracy.

There are many built-in optimizers in DSPy, which apply vastly different strategies. A typical DSPy optimizer takes three things:

- Your **DSPy program**. This may be a single module (e.g., `dspy.Predict`) or a complex multi-module program.
- Your **metric**. This is a function that evaluates the output of your program, and assigns it a score (higher is better).
- A few **training inputs**. This may be very small (i.e., only 5 or 10 examples) and incomplete (only inputs to your program, without any labels).

If you happen to have a lot of data, DSPy can leverage that. But you can start small and get strong results.

**Note:** Formerly called **DSPy Teleprompters**. We are making an official name update, which will be reflected throughout the library and documentation.

## What does a DSPy Optimizer tune? How does it tune them?

Traditional deep neural networks (DNNs) can be optimized with gradient descent, given a loss function and some training data.

DSPy programs consist of multiple calls to LMs, stacked together as [DSPy modules]. Each DSPy module has internal parameters of three kinds: (1) the LM weights, (2) the instructions, and (3) demonstrations of the input/output behavior.

Given a metric, DSPy can optimize all of these three with multi-stage optimization algorithms. These can combine gradient descent (for LM weights) and discrete LM-driven optimization, i.e. for crafting/updating instructions and for creating/validating demonstrations. DSPy Demonstrations are like few-shot examples, but they're far more powerful. They can be created from scratch, given your program, and their creation and selection can be optimized in many effective ways.

In many cases, we found that compiling leads to better prompts than human writing. Not because DSPy optimizers are more creative than humans, but simply because they can try more things, much more systematically, and tune the metrics directly.

## What DSPy Optimizers are currently available?

Subclasses of Teleprompter

All of these can be accessed via `from dspy.teleprompt import *`.

**Automatic Few-Shot Learning**

These optimizers extend the signature by automatically generating and including **optimized** examples within the prompt sent to the model, implementing few-shot learning.

1. **`LabeledFewShot`**: Simply constructs few-shot examples (demos) from provided labeled input and output data points. Requires `k` (number of examples for the prompt) and `trainset` to randomly select `k` examples from.

2. **`BootstrapFewShot`**: Uses a `teacher` module (which defaults to your program) to generate complete demonstrations for every stage of your program, along with labeled examples in `trainset`. Parameters include `max_labeled_demos` (the number of demonstrations randomly selected from the `trainset`) and `max_bootstrapped_demos` (the number of additional examples generated by the `teacher`). The bootstrapping process employs the metric to validate demonstrations, including only those that pass the metric in the "compiled" prompt. Advanced: Supports using a `teacher` program that is a *different* DSPy program that has compatible structure, for harder tasks.

3. **`BootstrapFewShotWithRandomSearch`**: Applies `BootstrapFewShot` several times with random search over generated demonstrations, and selects the best program over the optimization. Parameters mirror those of `BootstrapFewShot`, with the addition of `num_candidate_programs`, which specifies the number of random programs evaluated over the optimization, including candidates of the uncompiled program, `LabeledFewShot` optimized program, `BootstrapFewShot` compiled program

with unshuffled examples and `num_candidate_programs` or `BootstrapFewShot` compiled programs with randomized example sets.

4. **`BootstrapFewShotWithOptuna`**: Applies `BootstrapFewShot` with Optuna optimization across demonstration sets, running trials to maximize evaluation metrics and selecting the best demonstrations.

5. **`KNNFewShot`**. Selects demonstrations through k-Nearest Neighbors algorithm to pick a diverse set of examples from different clusters. Vectorizes the examples, and then clusters them, using cluster centers with `BootstrapFewShot` for bootstrapping/selection process. This will be useful when there's a lot of data over random spaces: using KNN helps optimize the `trainset` for `BootstrapFewShot`. See this notebook for an example.

**Automatic Instruction Optimization**
These optimizers produce optimal instructions for the prompt and, in the case of MIPRO also optimize the set of few-shot demonstrations.

6. **`COPRO`**: Generates and refines new instructions for each step, and optimizes them with coordinate ascent (hill-climbing using the metric function and the `trainset`). Parameters include `depth` which is the number of iterations of prompt improvement the optimizer runs over.

7. **`MIPRO`**: Generates instructions *and* few-shot examples in each step. The instruction generation is data-aware and demonstration-aware. Uses Bayesian Optimization to effectively search over the space of generation instructions/demonstrations across your modules.

**Automatic Finetuning**
This optimizer is used to fine-tune the underlying LLM(s).

6. **`BootstrapFinetune`**: Distills a prompt-based DSPy program into weight updates (for smaller LMs). The output is a DSPy program that has the same steps, but where each step is conducted by a finetuned model instead of a prompted LM.

**Program Transformations**
8. **`Ensemble`**: Ensembles a set of DSPy programs and either uses the full set or randomly samples a subset into a single program.

## Which optimizer should I use?
As a rule of thumb, if you don't know where to start, use `BootstrapFewShotWithRandomSearch`.

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`.

## How do I use an optimizer?
They all share this general interface, with some differences in the keyword arguments (hyperparameters).

Let's see this with the most common one, `BootstrapFewShotWithRandomSearch`.

```
from dspy.teleprompt import BootstrapFewShotWithRandomSearch

# Set up the optimizer: we want to "bootstrap" (i.e., self-generate) 8-shot examples of your program's steps.
# The optimizer will repeat this 10 times (plus some initial attempts) before selecting its best attempt on the devset.
config = dict(max_bootstrapped_demos=4, max_labeled_demos=4, num_candidate_programs=10, num_threads=4)

teleprompter = BootstrapFewShotWithRandomSearch(metric=YOUR_METRIC_HERE, **config)
optimized_program = teleprompter.compile(YOUR_PROGRAM_HERE, trainset=YOUR_TRAINSET_HERE)
```

## Saving and loading optimizer output

After running a program through an optimizer, it's useful to also save it. At a later point, a program can be loaded from a file and used for inference. For this, the `load` and `save` methods can be used.

### Saving a program

```
optimized_program.save(YOUR_SAVE_PATH)
```

The resulting file is in plain-text JSON format. It contains all the parameters and steps in the source program. You can always read it and see what the optimizer generated.

### Loading a program

To load a program from a file, you can instantiate an object from that class and then call the load method on it.

```
loaded_program = YOUR_PROGRAM_CLASS()
loaded_program.load(path=YOUR_SAVE_PATH)
```

# Executing Signatures

So far we've understood what signatures are and how we can use them to craft our prompt, but now let's take a look at how to execute them.

## Configuring LM

To execute signatures, we require DSPy modules which are themselves dependent on a client connection to a language model (LM) client. DSPy supports LM APIs and local model hosting. Within this example, we will make use of the OpenAI client and configure the GPT-3.5 (`gpt-3.5-turbo`) model.

```
turbo = dspy.OpenAI(model='gpt-3.5-turbo')
dspy.settings.configure(lm=turbo)
```

## Executing Signatures

Let's make use of the simplest module in DSPy - the `Predict` module that takes this signature as input to construct the prompt sent to the LM and generates a response for it.

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

    question = dspy.InputField()
    answer = dspy.OutputField(desc="often between 1 and 5 words")

# Define the predictor.
predictor = dspy.Predict(BasicQA)

# Call the predictor on a particular input.
pred = predictor(question=devset[0].question)

# Print the input and the prediction.
print(f"Question: {devset[0].question}")
print(f"Predicted Answer: {pred.answer}")
print(f"Actual Answer: {devset[0].answer}")
```

**Output:**

```
Question: Are both Cangzhou and Qionghai in the Hebei province of China?
Predicted Answer: No.
Actual Answer: no
```

The `Predict` module generates a response via the LM we configured above and executes the prompt crafted by the signature. This returns the output i.e. `answer` which is present in the object returned by the `predictor` and can be accessed via `.` operator.

## Inspecting Output

Let's dive deeper into DSPy uses our signature to build up the prompt, which we can do through the `inspect_history` method on the configured LM following the program's execution. This method returns the last `n` prompts executed by LM.

```
turbo.inspect_history(n=1)
```

**Output:**

```
Answer questions with short factoid answers.

---

Follow the following format.

Question: ${question}
Answer: often between 1 and 5 words

---

Question: Are both Cangzhou and Qionghai in the Hebei province of China?
Answer: No.
```

Additionally, if you want to store or use this prompt, you can access the `history` attribute of the LM object, which stores a list of dictionaries containing respective `prompt:response` entries for each LM generation.

```
turbo.history[0]
```

**Output:**


```
{'prompt': "Answer questions with short factoid answers.\n\n---\n\nFollow the following
format.\n\nQuestion: ${question}\nQuestion's Answer: often between 1 and 5 words\n\n---\n\nQuestion:
Are both Cangzhou and Qionghai in the Hebei province of China?\nQuestion's Answer:",
 'response': <OpenAIObject chat.completion id=chatcmpl-8kCPsxikpVpmSaxdGLUIqubFZS05p at 0x7c3ba41fa840>
JSON: {
   "id": "chatcmpl-8kCPsxikpVpmSaxdGLUIqubFZS05p",
   "object": "chat.completion",
   "created": 1706021508,
   "model": "gpt-3.5-turbo-0613",
   "choices": [
     {
       "index": 0,
       "message": {
         "role": "assistant",
         "content": "No."
       },
       "logprobs": null,
       "finish_reason": "stop"
     }
   ],
   "usage": {
     "prompt_tokens": 64,
     "completion_tokens": 2,
     "total_tokens": 66
   },
   "system_fingerprint": null
 },
 'kwargs': {'stringify_request': '{"temperature": 0.0, "max_tokens": 150, "top_p": 1,
"frequency_penalty": 0, "presence_penalty": 0, "n": 1, "model": "gpt-3.5-turbo", "messages": [{"role":
"user", "content": "Answer questions with short factoid answers.\\n\\n---\\n\\nFollow the following
format.\\n\\nQuestion: ${question}\\nQuestion\'s Answer: often between 1 and 5 words\\n\\n---
\\n\\nQuestion: Are both Cangzhou and Qionghai in the Hebei province of China?\\nQuestion\'s
Answer:"}]}'},
 'raw_kwargs': {}}
```


## How `Predict` works?

The output of predictor is a `Prediction` class object which mirrors the `Example` class with additional functionalities for LM completion interactivity.

How does `Predict` module actually 'predict' though? Here is a step-by-step breakdown:

1. A call to the predictor will get executed in `__call__` method of `Predict` Module which executes the `forward` method of the class.

2. In `forward` method, DSPy initializes the signature, LM call parameters and few-shot examples, if any.

3. The `_generate` method formats the few shots example to mirror the signature and uses the LM object we configured to generate the output as a `Prediction` object.

In case you are wondering how the prompt is constructed, the DSPy Signature framework internally handles the prompt structure, utilizing the DSP Template primitive to craft the prompt.

Predict gives you a predefined pipeline to execute signature which is nice but you can build much more complicated pipelines with this by creating custom Modules.

---

**Written By:** Herumb Shandilya

# internal-signatures

## DSPy's Internal Signatures

DSPy makes extensive use of signatures internally to define tasks and their requirements. Signatures are used to define the inputs and outputs of a task, and optionally, a small description about them and the task too.

Sometimes, you may want to update the signature of a task to include instructions or additional fields. You can access and replace any signature in DSPy using the `dspy.Signature.replace` context manager.

## Updating Signatures

To update a signature, you can use the `dspy.Signature.replace` context manager. This context manager allows you to replace the signature of a task with a new one. Here's an example of updating one of the signatures used in the Copro optimizer to add additional instructions:

```
from dspy.teleprompt import copro_optimizer
import dspy

class MyBasicGenerateInstruction(mipro_optimizer.BasicGenerateInstruction):
    """
    <PERSONA>
    You are an instruction optimizer for large language models.
    </PERSONA>
    <TASK>
    I will give you a ``signature`` of fields (inputs and outputs) in English. Your task is to propose
an instruction that will lead a good language model to perform the task well. Don't be afraid to be
creative, but the new instruction you propose should be clear and concise.
    </TASK>
    """
    basic_instruction = dspy.InputField(desc="The unoptimized instruction. New instructions should
achieve the same goals.")

with mipro_optimizer.BasicGenerateInstruction.replace(MyBasicGenerateInstruction):
    teleprompter = COPRO(prompt_model=prompt_model, metric=metric, breadth=BREADTH, depth=DEPTH,
init_temperature=INIT_TEMPERATURE)
    kwargs = dict(num_threads=NUM_THREADS, display_progress=True, display_table=0)
    compiled_prompt_opt = teleprompter.compile(program.deepcopy(), trainset=trainset[:DEV_NUM],
eval_kwargs=kwargs)
    eval_score = evaluate(compiled_prompt_opt, devset=evalset[:EVAL_NUM], **kwargs)
```

This will now update the signature of the `BasicGenerateInstruction` task in the Copro optimizer to include the new `basic_instruction` field.

You can replace any signature in DSPy using this method, which will be used as long as the context manager is active.

There is also a helper function `dspy.update_signatures` that can update multiple signatures at once. This function takes a dictionary of signatures to update, with the key being the signature to update and the value being the new signature class.

```
from dspy.teleprompt import copro_optimizer
import dspy

class MyBasicGenerateInstruction(mipro_optimizer.BasicGenerateInstruction):
    """
    <PERSONA>
    You are an instruction optimizer for large language models.
    </PERSONA>
    <TASK>
    I will give you a ``signature`` of fields (inputs and outputs) in English. Your task is to propose
```

```
an instruction that will lead a good language model to perform the task well. Don't be afraid to be
creative, but the new instruction you propose should be clear and concise.
    </TASK>
    """
    basic_instruction = dspy.InputField(desc="The unoptimized instruction. New instructions should
achieve the same goals.")

class GenerateInstructionGivenAttempts(mipro_optimizer.GenerateInstructionGivenAttempts):
    """
    <PERSONA>
    You are an instruction optimizer for large language models.
    </PERSONA>
    <TASK>
    I will give some task instructions I've tried, along with their corresponding validation scores.
The instructions are arranged in increasing order based on their scores, where higher scores indicate
better quality.

    Your task is to propose a new instruction that will lead a good language model to perform the task
even better. Don't be afraid to be creative
    </TASK>
    """
    pass

with dspy.update_signatures({
    copro_optimizer.BasicGenerateInstruction: MyBasicGenerateInstruction,
    copro_optimizer.GenerateInstructionGivenAttempts: GenerateInstructionGivenAttempts
    }):
    ...
```

---

**Written By:** Michael Jones

# Remote Language Model Clients

Remote Language Model Clients in DSPy

### Anyscale

Anyscale

### Cohere

Cohere

### OpenAI

OpenAI

### PremAI

PremAI

### Together

Togetherß

# Together

## Togetherß

**Adapted from documentation provided by https://github.com/insop**

## Prerequisites

- Together `api_key` and `api_base` (***for non-cached examples***). Set these within your developer environment `.env` as follows:

```
TOGETHER_API_BASE = ...
TOGETHER_API_KEY = ...
```

which will be retrieved within the Together Client as:

```
self.api_base = os.getenv("TOGETHER_API_BASE")
self.token = os.getenv("TOGETHER_API_KEY")
```

### Setting up the Together Client

The constructor initializes the `HFModel` base class to support the handling of prompting models. This requires the following parameters:

**Parameters:**

- `model` (*str*): ID of model hosted on Together endpoint.
- `**kwargs`: Additional keyword arguments to configure the Together client.

Example of the Together constructor:

```
class Together(HFModel):
    def __init__(self, model, **kwargs):
```

### Under the Hood

**`_generate(self, prompt, use_chat_api=False, **kwargs):`**

**Parameters:**

- `prompt` (*str*): Prompt to send to Together.
- `use_chat_api` (*bool*): Flag to use the Together Chat models endpoint. Defaults to False.
- `**kwargs`: Additional keyword arguments for completion request.

**Returns:**

- `dict`: dictionary with `prompt` and list of response `choices`.

Internally, the method handles the specifics of preparing the request prompt and corresponding payload to obtain the response.

The Together token is set within the request headers to ensure authorization to send requests to the endpoint.

If `use_chat_api` is set, the method sets up Together url chat endpoint and prompt template for chat models. It then retrieves the generated JSON response and sets up the `completions` list by retrieving the response's `message` : `content`.

If `use_chat_api` is not set, the method uses the default Together url endpoint. It similarly retrieves the generated JSON response and but sets up the `completions` list by retrieving the response's `text` as the completion.

Finally, after processing the requests and responses, the method constructs the response dictionary with two keys: the original request `prompt` and `choices`, a list of dictionaries representing generated `completions` with the key `text` holding the response's

generated text.

### Using the Together client

```
together = dspy.Together(model="mistralai/Mistral-7B-v0.1")
```

### Sending Requests via Together Client

1. ***Recommended*** Configure default LM using `dspy.configure`.

This allows you to define programs in DSPy and simply call modules on your input fields, having DSPy internally call the prompt on the configured LM.

```
dspy.configure(lm=together)

#Example DSPy CoT QA program
qa = dspy.ChainOfThought('question -> answer')

response = qa(question="What is the capital of Paris?") #Prompted to together
print(response.answer)
```

2. Generate responses using the client directly.

```
response = together(prompt='What is the capital of Paris?')
print(response)
```

---

**Written By:** Arnav Singhvi

# ChromadbRM

**Adapted from documentation provided by https://github.com/animtel**

# ChromadbRM

ChromadbRM have the flexibility from a variety of embedding functions as outlined in the chromadb embeddings documentation. While different options are available, this example demonstrates how to utilize OpenAI embeddings specifically.

## Setting up the ChromadbRM Client

The constructor initializes an instance of the `ChromadbRM` class, with the option to use OpenAI's embeddings or any alternative supported by chromadb, as detailed in the official chromadb embeddings documentation.

- `collection_name` (*str*): The name of the chromadb collection.
- `persist_directory` (*str*): Path to the directory where chromadb data is persisted.
- `embedding_function` (*Optional[EmbeddingFunction[Embeddable]]*, *optional*): The function used for embedding documents and queries. Defaults to `DefaultEmbeddingFunction()` if not specified.
- `k` (*int*, *optional*): The number of top passages to retrieve. Defaults to 7.

Example of the ChromadbRM constructor:

```
ChromadbRM(
    collection_name: str,
    persist_directory: str,
    embedding_function: Optional[EmbeddingFunction[Embeddable]] = OpenAIEmbeddingFunction(),
    k: int = 7,
)
```

## Under the Hood

**`forward(self, query_or_queries: Union[str, List[str]], k: Optional[int] = None) -> dspy.Prediction`**

**Parameters:**

- `query_or_queries` (*Union[str, List[str]]*): The query or list of queries to search for.
- `k` (*Optional[int]*, *optional*): The number of results to retrieve. If not specified, defaults to the value set during initialization.

**Returns:**

- `dspy.Prediction`: Contains the retrieved passages, each represented as a `dotdict` with a `long_text` attribute.

Search the chromadb collection for the top `k` passages matching the given query or queries, using embeddings generated via the specified `embedding_function`.

## Sending Retrieval Requests via ChromadbRM Client

```
from dspy.retrieve.chromadb_rm import ChromadbRM
import os
import openai
from chromadb.utils.embedding_functions import OpenAIEmbeddingFunction

embedding_function = OpenAIEmbeddingFunction(
    api_key=os.environ.get('OPENAI_API_KEY'),
    model_name="text-embedding-ada-002"
)

retriever_model = ChromadbRM(
    'your_collection_name',
    '/path/to/your/db'
```

```
    "/path/to/your/db",
    embedding_function=embedding_function,
    k=5
)

results = retriever_model("Explore the significance of quantum computing", k=5)

for result in results:
    print("Document:", result.long_text, "\n")
```

# ColBERTv2

## Setting up the ColBERTv2 Client

The constructor initializes the `ColBERTv2` class instance and sets up the request parameters for interacting with the ColBERTv2 retrieval server. This server is hosted remotely at `'http://20.102.90.50:2017/wiki17_abstracts`.

- `url` (*str*): URL for ColBERTv2 server. Defaults to `"http://0.0.0.0"`
- `port` (*Union[str, int]*, *Optional*): Port endpoint for ColBERTv2 server. Defaults to `None`.
- `post_requests` (*bool*, *Optional*): Flag for using HTTP POST requests. Defaults to `False`.

Example of the ColBERTv2 constructor:

```
class ColBERTv2:
    def __init__(
        self,
        url: str = "http://0.0.0.0",
        port: Optional[Union[str, int]] = None,
        post_requests: bool = False,
    ):
```

## Under the Hood

### `__call__(self, query: str, k: int = 10, simplify: bool = False) -> Union[list[str], list[dotdict]]`

**Parameters:**

- `query` (*str*): Search query string used for retrieval sent to ColBERTv2 server.
- `k` (*int*, *optional*): Number of passages to retrieve. Defaults to 10.
- `simplify` (*bool*, *optional*): Flag for simplifying output to a list of strings. Defaults to False.

**Returns:**

- `Union[list[str], list[dotdict]]`: Depending on `simplify` flag, either a list of strings representing the passage content (`True`) or a list of `dotdict` instances containing passage details (`False`).

Internally, the method handles the specifics of preparing the request query to the ColBERTv2 server and corresponding payload to obtain the response.

The function handles the retrieval of the top-k passages based on the provided query.

If `post_requests` is set, the method sends a query to the server via a POST request else via a GET request.

It then processes and returns the top-k passages from the response with the list of retrieved passages dependent on the `simplify` flag return condition above.

## Sending Retrieval Requests via ColBERTv2 Client

1. ***Recommended*** Configure default RM using `dspy.configure`.

This allows you to define programs in DSPy and have DSPy internally conduct retrieval using `dsp.retrieve` on the query on the configured RM.

```
import dspy
import dsp

dspy.settings.configure(rm=colbertv2_wiki17_abstracts)
retrieval_response = dsp.retrieve("When was the first FIFA World Cup held?", k=5)
```

```
for result in retrieval_response:
    print("Text:", result, "\n")
```

2. Generate responses using the client directly.

```
import dspy

retrieval_response = colbertv2_wiki17_abstracts('When was the first FIFA World Cup held?', k=5)

for result in retrieval_response:
    print("Text:", result['text'], "\n")
```

---

**Written By:** Arnav Singhvi

# MilvusRM

MilvusRM uses OpenAI's `text-embedding-3-small` embedding by default or any customized embedding function. To support passage retrieval, it assumes that a Milvus collection has been created and populated with the following field:

- `text`: The text of the passage

## Set up the MilvusRM Client

The constructor initializes an instance of the `MilvusRM` class, with the option to use OpenAI's `text-embedding-3-small` embeddings or any customized embedding function .

- `collection_name (str)`: The name of the Milvus collection to query against.
- `uri (str, optional)`: The Milvus connection uri. Defaults to "http://localhost:19530".
- `token (str, optional)`: The Milvus connection token. Defaults to None.
- `db_name (str, optional)`: The Milvus database name. Defaults to "default".
- `embedding_function (callable, optional)`: The function to convert a list of text to embeddings. The embedding function should take a list of text strings as input and output a list of embeddings. Defaults to None. By default, it will get OpenAI client by the environment variable OPENAI_API_KEY and use OpenAI's embedding model "text-embedding-3-small" with the default dimension.
- `k (int, optional)`: The number of top passages to retrieve. Defaults to 3.

Example of the MilvusRM constructor:

```
MilvusRM(
    collection_name: str,
    uri: Optional[str] = "http://localhost:19530",
    token: Optional[str] = None,
    db_name: Optional[str] = "default",
    embedding_function: Optional[Callable] = None,
    k: int = 3,
)
```

## Under the Hood

### `forward(self, query_or_queries: Union[str, List[str]], k: Optional[int] = None) -> dspy.Prediction`

**Parameters:**

- `query_or_queries` (*Union[str, List[str]]*): The query or list of queries to search for.
- `k` (*Optional[int]*, *optional*): The number of results to retrieve. If not specified, defaults to the value set during initialization.

**Returns:**

- `dspy.Prediction`: Contains the retrieved passages, each represented as a `dotdict` with a `long_text` attribute.

Search the Milvus collection for the top `k` passages matching the given query or queries, using embeddings generated via the default OpenAI embedding or the specified `embedding_function`.

## Sending Retrieval Requests via MilvusRM Client

```
from dspy.retrieve.milvus_rm import MilvusRM
import os

os.envrion["OPENAI_API_KEY"] = "<YOUR_OPENAI_API_KEY>"

retriever_model = MilvusRM(
    collection_name="<YOUR_COLLECTION_NAME>"
```

```
    collection_name="<YOUR_COLLECTION_NAME>",
    uri="<YOUR_MILVUS_URI>",
    token="<YOUR_MILVUS_TOKEN>"  # ignore this if no token is required for Milvus connection
    )

results = retriever_model("Explore the significance of quantum computing", k=5)

for result in results:
    print("Document:", result.long_text, "\n")
```

# Weaviate Retrieval Model

Weaviate is an open-source vector database that can be used to retrieve relevant passages before passing it to the language model. Weaviate supports a variety of embedding models from OpenAI, Cohere, Google and more! Before building your DSPy program, you will need a Weaviate cluster running with data. You can follow this notebook as an example.

## Configuring the Weaviate Client

Weaviate is available via a hosted service (WCD) or as a self managed instance. You can learn about the different installation methods here.

- `weaviate_collection_name` (str): The name of the Weaviate collection
- `weaviate_client` (WeaviateClient): An instance of the Weaviate client
- `k` (int, optional): The number of top passages to retrieve. The default is set to `3`

An example of the WeaviateRM constructor:

```
WeaviateRM(
    collection_name: str
    weaviate_client: str,
    k: int = 5
)
```

## Under the Hood

`forward(self, query_or_queries: Union[str, List[str]], k: Optional[int] = None, **kwargs) -> dspy.Prediction`

### Parameters

- `query_or_queries` (Union[str, List[str]]): The query or queries to search for
- `k (Optional[int]): The number of top passages to retrieve. It defaults to` self.k`
- `**kwargs`: Additional keyword arguments like `rerank` for example

### Returns

- `dspy.Prediction`: An object containing the retrieved passages

## Sending Retrieval Requests via the WeaviateRM Client

Here is an example of the Weaviate constructor using embedded:

```
import weaviate
import dspy
from dspy.retrieve.weaviate_rm import WeaviateRM

weaviate_client = weaviate.connect_to_embedded() # you can also use local or WCD

retriever_model = WeaviateRM(
    collection_name="<WEAVIATE_COLLECTION>",
    weaviate_client=weaviate_client
)

results = retriever_model("Explore the significance of quantum computing", k=5)

for result in results:
    print("Document:", result.long_text, "\n")
```

You can follow along with more DSPy and Weaviate examples here!

# AzureAISearch

A retrieval module that utilizes Azure AI Search to retrieve top passages for a given query.

## Prerequisites

```
pip install azure-search-documents
```

## Setting up the AzureAISearchRM Client

The constructor initializes an instance of the `AzureAISearchRM` class and sets up parameters for sending queries and retrieving results with the Azure AI Search server.

- `search_service_name` (str): The name of the Azure AI Search service.
- `search_api_key` (str): The API key for accessing the Azure AI Search service.
- `search_index_name` (str): The name of the search index in the Azure AI Search service.
- `field_text` (str): The name of the field containing text content in the search index. This field will be mapped to the "content" field in the dsp framework.
- `field_vector` (Optional[str]): The name of the field containing vector content in the search index.
- `k` (int, optional): The default number of top passages to retrieve. Defaults to 3.
- `azure_openai_client` (Optional[openai.AzureOpenAI]): An instance of the AzureOpenAI client. Either openai_client or embedding_func must be provided. Defaults to None.
- `openai_embed_model` (Optional[str]): The name of the OpenAI embedding model. Defaults to "text-embedding-ada-002".
- `embedding_func` (Optional[Callable]): A function for generating embeddings. Either openai_client or embedding_func must be provided. Defaults to None.
- `semantic_ranker` (bool, optional): Whether to use semantic ranking. Defaults to False.
- `filter` (str, optional): Additional filter query. Defaults to None.
- `query_language` (str, optional): The language of the query. Defaults to "en-Us".
- `query_speller` (str, optional): The speller mode. Defaults to "lexicon".
- `use_semantic_captions` (bool, optional): Whether to use semantic captions. Defaults to False.
- `query_type` (Optional[QueryType], optional): The type of query. Defaults to QueryType.FULL.
- `semantic_configuration_name` (str, optional): The name of the semantic configuration. Defaults to None.
- `is_vector_search` (Optional[bool]): Whether to enable vector search. Defaults to False.
- `is_hybrid_search` (Optional[bool]): Whether to enable hybrid search. Defaults to False.
- `is_fulltext_search` (Optional[bool]): Whether to enable fulltext search. Defaults to True.
- `vector_filter_mode` (Optional[VectorFilterMode]): The vector filter mode. Defaults to None.

**Available Query Types:**

- SIMPLE """Uses the simple query syntax for searches. Search text is interpreted using a simple query #: language that allows for symbols such as +, * and "". Queries are evaluated across all #: searchable fields by default, unless the searchFields parameter is specified."""

- FULL """Uses the full Lucene query syntax for searches. Search text is interpreted using the Lucene #: query language which allows field-specific and weighted searches, as well as other advanced #: features."""

- SEMANTIC """Best suited for queries expressed in natural language as opposed to keywords. Improves #: precision of search results by re-ranking the top search results using a ranking model trained #: on the Web corpus.""

  More Details: https://learn.microsoft.com/en-us/azure/search/search-query-overview

**Available Vector Filter Mode:**

**Available vector filter mode:**

- POST_FILTER = "postFilter" """The filter will be applied after the candidate set of vector results is returned. Depending on #: the filter selectivity, this can result in fewer results than requested by the parameter 'k'."""
- PRE_FILTER = "preFilter" """The filter will be applied before the search query."""

  More Details: https://learn.microsoft.com/en-us/azure/search/vector-search-filters

**Note**

- The `AzureAISearchRM` client allows you to perform Vector search, Hybrid search, or Full text search.
- By default, the `AzureAISearchRM` client uses the Azure OpenAI Client for generating embeddings. If you want to use something else, you can provide your custom embedding_func, but either the openai_client or embedding_func must be provided.
- If you need to enable semantic search, either with vector, hybrid, or full text search, then set the `semantic_ranker` flag to True.
- If `semantic_ranker` is True, always set the `query_type` to QueryType.SEMANTIC and always provide the `semantic_configuration_name`.

Example of the AzureAISearchRM constructor:

```
AzureAISearchRM(
    search_service_name: str,
    search_api_key: str,
    search_index_name: str,
    field_text: str,
    field_vector: Optional[str] = None,
    k: int = 3,
    azure_openai_client: Optional[openai.AzureOpenAI] = None,
    openai_embed_model: Optional[str] = "text-embedding-ada-002",
    embedding_func: Optional[Callable] = None,
    semantic_ranker: bool = False,
    filter: str = None,
    query_language: str = "en-Us",
    query_speller: str = "lexicon",
    use_semantic_captions: bool = False,
    query_type: Optional[QueryType] = QueryType.FULL,
    semantic_configuration_name: str = None,
    is_vector_search: Optional[bool] = False,
    is_hybrid_search: Optional[bool] = False,
    is_fulltext_search: Optional[bool] = True,
    vector_filter_mode: Optional[VectorFilterMode.PRE_FILTER] = None
)
```

## Under the Hood

### `forward(self, query_or_queries: Union[str, List[str]], k: Optional[int] = None) -> dspy.Prediction`

**Parameters:**

- `query_or_queries` (Union[str, List[str]]): The query or queries to search for.
- `k` (*Optional[int]*, *optional*): The number of results to retrieve. If not specified, defaults to the value set during initialization.

**Returns:**

- `dspy.Prediction`: Contains the retrieved passages, each represented as a `dotdict` with a `long_text` attribute.

Internally, the method handles the specifics of preparing the request query to the Azure AI Search service and corresponding payload to obtain the response.

The function handles the retrieval of the top-k passages based on the provided query.

## Sending Retrieval Requests via AzureAISearchRM Client

## Sending Retrieval Requests via AzureAISearchRM Client

1. ***Recommended*** Configure default RM using `dspy.configure`.

This allows you to define programs in DSPy and have DSPy internally conduct retrieval using `dsp.retrieve` on the query on the configured RM.

```
import dspy
from dspy.retrieve.azureaisearch_rm import AzureAISearchRM

azure_search = AzureAISearchRM(
    "search_service_name",
    "search_api_key",
    "search_index_name",
    "field_text",
    "k"=3
)

dspy.settings.configure(rm=azure_search)
retrieve = dspy.Retrieve(k=3)
retrieval_response = retrieve("What is Thermodynamics").passages

for result in retrieval_response:
    print("Text:", result, "\n")
```

2. Generate responses using the client directly.

```
from dspy.retrieve.azureaisearch_rm import AzureAISearchRM

azure_search = AzureAISearchRM(
    "search_service_name",
    "search_api_key",
    "search_index_name",
    "field_text",
    "k"=3
)

retrieval_response = azure_search("What is Thermodynamics", k=3)
for result in retrieval_response:
    print("Text:", result.long_text, "\n")
```

3. Example of Semantic Hybrid Search.

```
from dspy.retrieve.azureaisearch_rm import AzureAISearchRM

azure_search = AzureAISearchRM(
    search_service_name="search_service_name",
    search_api_key="search_api_key",
    search_index_name="search_index_name",
    field_text="field_text",
    field_vector="field_vector",
    k=3,
    azure_openai_client="azure_openai_client",
    openai_embed_model="text-embedding-ada-002"
    semantic_ranker=True,
    query_type=QueryType.SEMANTIC,
    semantic_configuration_name="semantic_configuration_name",
    is_hybrid_search=True,
)

retrieval_response = azure_search("What is Thermodynamics", k=3)
for result in retrieval_response:
    print("Text:", result.long_text, "\n")
```

**Written By:** Prajapati Harishkumar Kishorkumar

# Examples in DSPy

Working in DSPy involves training sets, development sets, and test sets. This is like traditional ML, but you usually need far fewer labels (or zero labels) to use DSPy effectively.

The core data type for data in DSPy is `Example`. You will use **Examples** to represent items in your training set and test set.

DSPy **Examples** are similar to Python `dict`s but have a few useful utilities. Your DSPy modules will return values of the type `Prediction`, which is a special sub-class of `Example`.

## Creating an `Example`

When you use DSPy, you will do a lot of evaluation and optimization runs. Your individual datapoints will be of type `Example`:

```
qa_pair = dspy.Example(question="This is a question?", answer="This is an answer.")

print(qa_pair)
print(qa_pair.question)
print(qa_pair.answer)
```

**Output:**

```
Example({'question': 'This is a question?', 'answer': 'This is an answer.'}) (input_keys=None)
This is a question?
This is an answer.
```

Examples can have any field keys and any value types, though usually values are strings.

```
object = Example(field1=value1, field2=value2, field3=value3, ...)
```

## Specifying Input Keys

In traditional ML, there are separated "inputs" and "labels".

In DSPy, the `Example` objects have a `with_inputs()` method, which can mark specific fields as inputs. (The rest are just metadata or labels.)

```
# Single Input.
print(qa_pair.with_inputs("question"))

# Multiple Inputs; be careful about marking your labels as inputs unless you mean it.
print(qa_pair.with_inputs("question", "answer"))
```

This flexibility allows for customized tailoring of the `Example` object for different DSPy scenarios.

When you call `with_inputs()`, you get a new copy of the example. The original object is kept unchanged.

## Element Access and Updation

Values can be accessed using the `.`(dot) operator. You can access the value of key `name` in defined object `Example(name="John Doe", job="sleep")` through `object.name`.

To access or exclude certain keys, use `inputs()` and `labels()` methods to return new Example objects containing only input or non-input keys, respectively.

```
article_summary = dspy.Example(article= "This is an article.", summary= "This is a
summary.").with_inputs("article")

input_key_only = article_summary.inputs()
non_input_key_only = article_summary.labels()

print("Example object with Input fields only:", input_key_only)
print("Example object with Non-Input fields only:", non_input_key_only)
```

**Output**

```
Example object with Input fields only: Example({'article': 'This is an article.'}) (input_keys=None)
Example object with Non-Input fields only: Example({'summary': 'This is a summary.'}) (input_keys=None)
```

To exclude keys, use `without()`:

```
article_summary = dspy.Example(context="This is an article.", question="This is a question?",
answer="This is an answer.", rationale= "This is a rationale.").with_inputs("context", "question")

print("Example object without answer & rationale keys:", article_summary.without("answer",
"rationale"))
```

**Output**

```
Example object without answer & rationale keys: Example({'context': 'This is an article.', 'question':
'This is a question?'}) (input_keys=None)
```

Updating values is simply assigning a new value using the `.` operator.

```
article_summary.context = "new context"
```

## Iterating over Example

Iteration in the `Example` class also functions like a dictionary, supporting methods like `keys()`, `values()`, etc:

```
for k, v in article_summary.items():
    print(f"{k} = {v}")
```

**Output**

```
context = This is an article.
question = This is a question?
answer = This is an answer.
rationale = This is a rationale.
```

---

**Written By:** Herumb Shandilya

# Utilizing Built-in Datasets

It's easy to use your own data in DSPy: a dataset is just a list of `Example` objects. Using DSPy well involves being able to find and re-purpose existing datasets for your own pipelines in new ways; DSPy makes this a particularly powerful strategy.

For convenience, DSPy currently also provides support for the following dataset out of the box:

- **HotPotQA** (multi-hop question answering)
- **GSM8k** (math questions)
- **Color** (basic dataset of colors)

## Loading HotPotQA

HotPotQA is which is a collection of question-answer pairs.

```
from dspy.datasets import HotPotQA

dataset = HotPotQA(train_seed=1, train_size=5, eval_seed=2023, dev_size=50, test_size=0)

print(dataset.train)
```

**Output:**

```
[Example({'question': 'At My Window was released by which American singer-songwriter?', 'answer': 'John
Townes Van Zandt'}) (input_keys=None),
 Example({'question': 'which  American actor was Candace Kita  guest starred with ', 'answer': 'Bill
Murray'}) (input_keys=None),
 Example({'question': 'Which of these publications was most recently published, Who Put the Bomp or
Self?', 'answer': 'Self'}) (input_keys=None),
 Example({'question': 'The Victorians - Their Story In Pictures is a documentary series written by an
author born in what year?', 'answer': '1950'}) (input_keys=None),
 Example({'question': 'Which magazine has published articles by Scott Shaw, Tae Kwon Do Times or
Southwest Art?', 'answer': 'Tae Kwon Do Times'}) (input_keys=None)]
```

We just loaded trainset (5 examples) and devset (50 examples). Each example in our training set contains just a question and its (human-annotated) answer. As you can see, it is loaded as a list of `Example` objects. However, one thing to note is that it doesn't set the input keys implicitly, so that is something that we'll need to do!!

```
trainset = [x.with_inputs('question') for x in dataset.train]
devset = [x.with_inputs('question') for x in dataset.dev]

print(trainset)
```

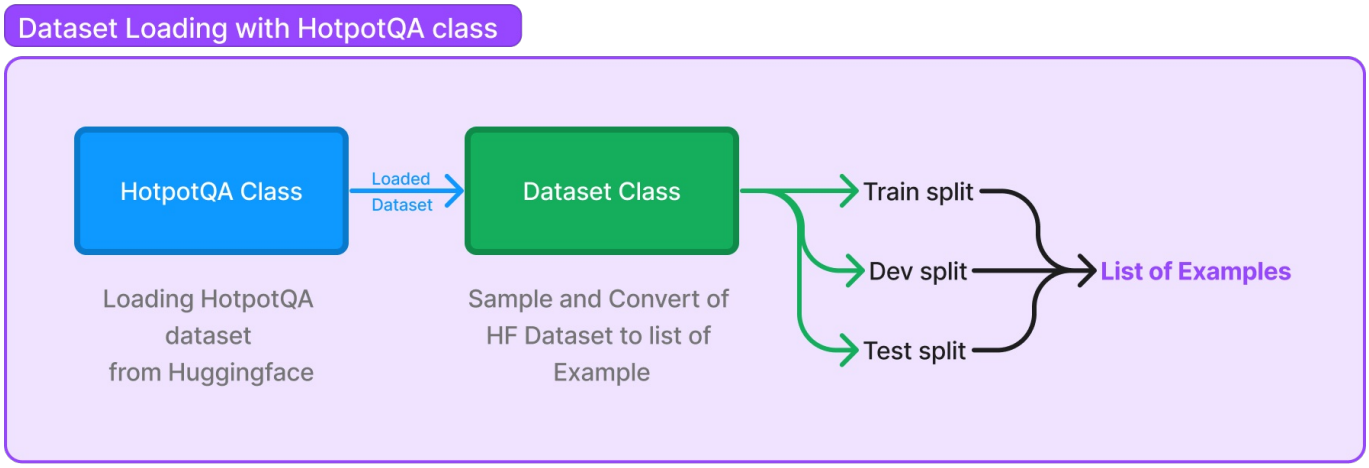
**Output:**

```
[Example({'question': 'At My Window was released by which American singer-songwriter?', 'answer': 'John
Townes Van Zandt'}) (input_keys={'question'}),
 Example({'question': 'which  American actor was Candace Kita  guest starred with ', 'answer': 'Bill
Murray'}) (input_keys={'question'}),
 Example({'question': 'Which of these publications was most recently published, Who Put the Bomp or
Self?', 'answer': 'Self'}) (input_keys={'question'}),
 Example({'question': 'The Victorians - Their Story In Pictures is a documentary series written by an
author born in what year?', 'answer': '1950'}) (input_keys={'question'}),
 Example({'question': 'Which magazine has published articles by Scott Shaw, Tae Kwon Do Times or
Southwest Art?', 'answer': 'Tae Kwon Do Times'}) (input_keys={'question'})]
```

DSPy typically requires very minimal labeling. Whereas your pipeline may involve six or seven complex steps, you only need labels for the initial question and the final answer. DSPy will bootstrap any intermediate labels needed to support your pipeline. If you change your pipeline in any way, the data bootstrapped will change accordingly!

## Advanced: Inside DSPy's `Dataset` class (Optional)

We've seen how you can use `HotPotQA` dataset class and load the HotPotQA dataset, but how does it actually work? The `HotPotQA` class inherits from the `Dataset` class, which takes care of the conversion of the data loaded from a source into train-test-dev split, all of which are *list of examples*. In the `HotPotQA` class, you only implement the `__init__` method, where you populate the splits from HuggingFace into the variables `_train`, `_test` and `_dev`. The rest of the process is handled by methods in the `Dataset` class.




But how do the methods of the `Dataset` class convert the data from HuggingFace? Let's take a deep breath and think step by step...pun intended. In example above, we can see the splits accessed by `.train`, `.dev` and `.test` methods, so let's take a look at the implementation of the `train()` method:

```
@property
def train(self):
    if not hasattr(self, '_train_'):
        self._train_ = self._shuffle_and_sample('train', self._train, self.train_size, self.train_seed)

    return self._train_
```

As you can see, the `train()` method serves as a property, not a regular method. Within this property, it first checks if the `_train_` attribute exists. If not, it calls the `_shuffle_and_sample()` method to process the `self._train` where the HuggingFace dataset is loaded. Let's see the `_shuffle_and_sample()` method:

```
def _shuffle_and_sample(self, split, data, size, seed=0):
    data = list(data)
    base_rng = random.Random(seed)

    if self.do_shuffle:
        base_rng.shuffle(data)

    data = data[:size]
    output = []

    for example in data:
        output.append(Example(**example, dspy_uuid=str(uuid.uuid4()), dspy_split=split))

        return output
```

The `_shuffle_and_sample()` method does two things:

- It shuffles the data if `self.do_shuffle` is True.
- It then takes a sample of size `size` from the shuffled data.
- It then loops through the sampled data and converts each element in `data` into an `Example` object. The `Example` along with example data also contains a unique ID, and the split name.

Converting the raw examples into `Example` objects allows the Dataset class to process them in a standardized way later. For example, the collate method, which is used by the PyTorch DataLoader, expects each item to be an `Example`.

To summarize, the `Dataset` class handles all the necessary data processing and provides a simple API to access the different splits. This differentiates from the dataset classes like HotpotQA which require only definitions on how to load the raw data.

---

**Written By:** Herumb Shandilya

# Creating a Custom Dataset

We've seen how to work with with `Example` objects and use the `HotPotQA` class to load the HuggingFace HotPotQA dataset as a list of `Example` objects. But in production, such structured datasets are rare. Instead, you'll find yourself working on a custom dataset and might question: how do I create my own dataset or what format should it be?

In DSPy, your dataset is a list of `Examples`, which we can accomplish in two ways:

- **Recommended: The Pythonic Way:** Using native python utility and logic.
- **Advanced: Using DSPy's `Dataset` class**

## Recommended: The Pythonic Way

To create a list of `Example` objects, we can simply load data from the source and formulate it into a Python list. Let's load an example CSV `sample.csv` that contains 3 fields: (**context**, **question** and **summary**) via Pandas. From there, we can construct our data list.

```
import pandas as pd

df = pd.read_csv("sample.csv")
print(df.shape)
```

**Output:**

```
(1000, 3)
```

```
dataset = []

for context, question, answer in df.values:
    dataset.append(dspy.Example(context=context, question=question,
answer=answer).with_inputs("context", "question"))

print(dataset[:3])
```

**Output:**

```
[Example({'context': nan, 'question': 'Which is a species of fish? Tope or Rope', 'answer': 'Tope'})
(input_keys={'question', 'context'}),
 Example({'context': nan, 'question': 'Why can camels survive for long without water?', 'answer':
'Camels use the fat in their humps to keep them filled with energy and hydration for long periods of
time.'}) (input_keys={'question', 'context'}),
 Example({'context': nan, 'question': "Alice's parents have three daughters: Amy, Jessy, and what's the
name of the third daughter?", 'answer': 'The name of the third daughter is Alice'}) (input_keys=
{'question', 'context'})]
```

While this is fairly simple, let's take a look at how loading datasets would look in DSPy - via the DSPythonic way!

## Advanced: Using DSPy's `Dataset` class (Optional)

Let's take advantage of the `Dataset` class we defined in the previous article to accomplish the preprocessing:

- Load data from CSV to a dataframe.
- Split the data to train, dev and test splits.
- Populate `_train`, `_dev` and `_test` class attributes. Note that these attributes should be a list of dictionary, or an iterator over

mapping like HuggingFace Dataset, to make it work.

This is all done through the `__init__` method, which is the only method we have to implement.

```
import pandas as pd
from dspy.datasets.dataset import Dataset

class CSVDataset(Dataset):
    def __init__(self, file_path, *args, **kwargs) -> None:
        super().__init__(*args, **kwargs)

        df = pd.read_csv(file_path)
        self._train = df.iloc[0:700].to_dict(orient='records')

        self._dev = df.iloc[700:].to_dict(orient='records')

dataset = CSVDataset("sample.csv")
print(dataset.train[:3])
```

**Output:**

```
[Example({'context': nan, 'question': 'Which is a species of fish? Tope or Rope', 'answer': 'Tope'})
(input_keys={'question', 'context'}),
 Example({'context': nan, 'question': 'Why can camels survive for long without water?', 'answer':
'Camels use the fat in their humps to keep them filled with energy and hydration for long periods of
time.'}) (input_keys={'question', 'context'}),
 Example({'context': nan, 'question': "Alice's parents have three daughters: Amy, Jessy, and what's the
name of the third daughter?", 'answer': 'The name of the third daughter is Alice'}) (input_keys=
{'question', 'context'})]
```

Let's understand the code step by step:

- It inherits the base `Dataset` class from DSPy. This inherits all the useful data loading/processing functionality.
- We load the data in CSV into a DataFrame.
- We get the **train** split i.e first 700 rows in the DataFrame and convert it to lists of dicts using `to_dict(orient='records')` method and is then assigned to `self._train`.
- We get the **dev** split i.e first 300 rows in the DataFrame and convert it to lists of dicts using `to_dict(orient='records')` method and is then assigned to `self._dev`.

Using the Dataset base class now makes loading custom datasets incredibly easy and avoids having to write all that boilerplate code ourselves for every new dataset.

> **CAUTION**
>
> We did not populate `_test` attribute in the above code, which is fine and won't cause any unneccesary error as such. However it'll give you an error if you try to access the test split.

```
dataset.test[:5]
```

---

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-59-5202f6de3c7b> in <cell line: 1>()
----> 1 dataset.test[:5]

/usr/local/lib/python3.10/dist-packages/dspy/datasets/dataset.py in test(self)
     51     def test(self):
```

```
     52         if not hasattr(self, '_test_'):
---> 53             self._test_ = self._shuffle_and_sample('test', self._test, self.test_size,
self.test_seed)
     54
     55         return self._test_

AttributeError: 'CSVDataset' object has no attribute '_test'
```

To prevent that you'll just need to make sure `_test` is not `None` and populated with the appropriate data.

You can overide the methods in `Dataset` class to customize your class even more.

In summary, the Dataset base class provides a simplistic way to load and preprocess custom datasets with minimal code!

---

**Written By:** Herumb Shandilya

# DSPy Assertions

## Introduction

Language models (LMs) have transformed how we interact with machine learning, offering vast capabilities in natural language understanding and generation. However, ensuring these models adhere to domain-specific constraints remains a challenge. Despite the growth of techniques like fine-tuning or "prompt engineering", these approaches are extremely tedious and rely on heavy, manual hand-waving to guide the LMs in adhering to specific constraints. Even DSPy's modularity of programming prompting pipelines lacks mechanisms to effectively and automatically enforce these constraints.

To address this, we introduce DSPy Assertions, a feature within the DSPy framework designed to automate the enforcement of computational constraints on LMs. DSPy Assertions empower developers to guide LMs towards desired outcomes with minimal manual intervention, enhancing the reliability, predictability, and correctness of LM outputs.

### dspy.Assert and dspy.Suggest API

We introduce two primary constructs within DSPy Assertions:

- **`dspy.Assert`**:
  - **Parameters**:
    - `constraint (bool)`: Outcome of Python-defined boolean validation check.
    - `msg (Optional[str])`: User-defined error message providing feedback or correction guidance.
    - `backtrack (Optional[module])`: Specifies target module for retry attempts upon constraint failure. The default backtracking module is the last module before the assertion.
  - **Behavior**: Initiates retry upon failure, dynamically adjusting the pipeline's execution. If failures persist, it halts execution and raises a `dspy.AssertionError`.
- **`dspy.Suggest`**:
  - **Parameters**: Similar to `dspy.Assert`.
  - **Behavior**: Encourages self-refinement through retries without enforcing hard stops. Logs failures after maximum backtracking attempts and continues execution.
- **dspy.Assert vs. Python Assertions**: Unlike conventional Python `assert` statements that terminate the program upon failure, `dspy.Assert` conducts a sophisticated retry mechanism, allowing the pipeline to adjust.

Specifically, when a constraint is not met:

- Backtracking Mechanism: An under-the-hood backtracking is initiated, offering the model a chance to self-refine and proceed, which is done through
- Dynamic Signature Modification: internally modifying your DSPy program's Signature by adding the following fields:
  - Past Output: your model's past output that did not pass the validation_fn
  - Instruction: your user-defined feedback message on what went wrong and what possibly to fix

If the error continues past the `max_backtracking_attempts`, then `dspy.Assert` will halt the pipeline execution, altering you with an `dspy.AssertionError`. This ensures your program doesn't continue executing with "bad" LM behavior and immediately highlights sample failure outputs for user assessment.

- **dspy.Suggest vs. dspy.Assert**: `dspy.Suggest` on the other hand offers a softer approach. It maintains the same retry backtracking as `dspy.Assert` but instead serves as a gentle nudger. If the model outputs cannot pass the model constraints after the `max_backtracking_attempts`, `dspy.Suggest` will log the persistent failure and continue execution of the program on the rest of the data. This ensures the LM pipeline works in a "best-effort" manner without halting execution.

- **`dspy.Suggest`** are best utilized as "helpers" during the evaluation phase, offering guidance and potential corrections without halting the pipeline.

- `dspy.Assert` are recommended during the development stage as "checkers" to ensure the LM behaves as expected, providing a robust mechanism for identifying and addressing errors early in the development cycle.

## Use Case: Including Assertions in DSPy Programs

We start with using an example of a multi-hop QA SimplifiedBaleen pipeline as defined in the intro walkthrough.

```
class SimplifiedBaleen(dspy.Module):
    def __init__(self, passages_per_hop=2, max_hops=2):
        super().__init__()

        self.generate_query = [dspy.ChainOfThought(GenerateSearchQuery) for _ in range(max_hops)]
        self.retrieve = dspy.Retrieve(k=passages_per_hop)
        self.generate_answer = dspy.ChainOfThought(GenerateAnswer)
        self.max_hops = max_hops

    def forward(self, question):
        context = []
        prev_queries = [question]

        for hop in range(self.max_hops):
            query = self.generate_query[hop](context=context, question=question).query
            prev_queries.append(query)
            passages = self.retrieve(query).passages
            context = deduplicate(context + passages)

        pred = self.generate_answer(context=context, question=question)
        pred = dspy.Prediction(context=context, answer=pred.answer)
        return pred

baleen = SimplifiedBaleen()

baleen(question = "Which award did Gary Zukav's first book receive?")
```

To include DSPy Assertions, we simply define our validation functions and declare our assertions following the respective model generation.

For this use case, suppose we want to impose the following constraints:

1. Length - each query should be less than 100 characters
2. Uniqueness - each generated query should differ from previously-generated queries.

We can define these validation checks as boolean functions:

```
#simplistic boolean check for query length
len(query) <= 100

#Python function for validating distinct queries
def validate_query_distinction_local(previous_queries, query):
    """check if query is distinct from previous queries"""
    if previous_queries == []:
        return True
    if dspy.evaluate.answer_exact_match_str(query, previous_queries, frac=0.8):
        return False
    return True
```

We can declare these validation checks through `dspy.Suggest` statements (as we want to test the program in a best-effort demonstration). We want to keep these after the query generation `query = self.generate_query[hop](context=context, question=question).query`.

```
dspy.Suggest(
    len(query) <= 100,
    "Query should be short and less than 100 characters",
)

dspy.Suggest(
    validate_query_distinction_local(prev_queries, query),
    "Query should be distinct from: "
    + "; ".join(f"{i+1}) {q}" for i, q in enumerate(prev_queries)),
)
```

It is recommended to define a program with assertions separately than your original program if you are doing comparative evaluation for the effect of assertions. If not, feel free to set Assertions away!

Let's take a look at how the SimplifiedBaleen program will look with Assertions included:

```
class SimplifiedBaleenAssertions(dspy.Module):
    def __init__(self, passages_per_hop=2, max_hops=2):
        super().__init__()
        self.generate_query = [dspy.ChainOfThought(GenerateSearchQuery) for _ in range(max_hops)]
        self.retrieve = dspy.Retrieve(k=passages_per_hop)
        self.generate_answer = dspy.ChainOfThought(GenerateAnswer)
        self.max_hops = max_hops

    def forward(self, question):
        context = []
        prev_queries = [question]

        for hop in range(self.max_hops):
            query = self.generate_query[hop](context=context, question=question).query

            dspy.Suggest(
                len(query) <= 100,
                "Query should be short and less than 100 characters",
            )

            dspy.Suggest(
                validate_query_distinction_local(prev_queries, query),
                "Query should be distinct from: "
                + "; ".join(f"{i+1}) {q}" for i, q in enumerate(prev_queries)),
            )

            prev_queries.append(query)
            passages = self.retrieve(query).passages
            context = deduplicate(context + passages)

        if all_queries_distinct(prev_queries):
            self.passed_suggestions += 1

        pred = self.generate_answer(context=context, question=question)
        pred = dspy.Prediction(context=context, answer=pred.answer)
        return pred
```

Now calling programs with DSPy Assertions requires one last step, and that is transforming the program to wrap it with internal assertions backtracking and Retry logic.

```
from dspy.primitives.assertions import assert_transform_module, backtrack_handler

baleen_with_assertions = assert_transform_module(SimplifiedBaleenAssertions(), backtrack_handler)

# backtrack_handler is parameterized over a few settings for the backtracking mechanism
# To change the number of max retry attempts, you can do
baleen_with_assertions_retry_once = assert_transform_module(SimplifiedBaleenAssertions()
```

```
baleen_with_assertions_retry_once = assert_transform_module(SimplifiedBaleenAssertions(),
    functools.partial(backtrack_handler, max_backtracks=1))
```

Alternatively, you can also directly call `activate_assertions` on the program with `dspy.Assert/Suggest` statements using the default backtracking mechanism (`max_backtracks=2`):

```
baleen_with_assertions = SimplifiedBaleenAssertions().activate_assertions()
```

Now let's take a look at the internal LM backtracking by inspecting the history of the LM query generations. Here we see that when a query fails to pass the validation check of being less than 100 characters, its internal `GenerateSearchQuery` signature is dynamically modified during the backtracking+Retry process to include the past query and the corresponding user-defined instruction: `"Query should be short and less than 100 characters"`.

```
Write a simple search query that will help answer a complex question.

---

Follow the following format.

Context: may contain relevant facts

Question: ${question}

Reasoning: Let's think step by step in order to ${produce the query}. We ...

Query: ${query}

---

Context:
[1] «Kerry Condon | Kerry Condon (born 4 January 1983) is [...]»
[2] «Corona Riccardo | Corona Riccardo (c. 1878October 15, 1917) was [...]»

Question: Who acted in the shot film The Shore and is also the youngest actress ever to play Ophelia in
a Royal Shakespeare Company production of "Hamlet." ?

Reasoning: Let's think step by step in order to find the answer to this question. First, we need to
identify the actress who played Ophelia in a Royal Shakespeare Company production of "Hamlet." Then, we
need to find out if this actress also acted in the short film "The Shore."

Query: "actress who played Ophelia in Royal Shakespeare Company production of Hamlet" + "actress in
short film The Shore"



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

---

Follow the following format.

Context: may contain relevant facts

Question: ${question}

Past Query: past output with errors

Instructions: Some instructions you must satisfy

Query: ${query}

---
```

```
Context:
[1] «Kerry Condon | Kerry Condon (born 4 January 1983) is an Irish television and film actress, best
known for her role as Octavia of the Julii in the HBO/BBC series "Rome," as Stacey Ehrmantraut in AMC's
"Better Call Saul" and as the voice of F.R.I.D.A.Y. in various films in the Marvel Cinematic Universe.
She is also the youngest actress ever to play Ophelia in a Royal Shakespeare Company production of
"Hamlet."»
[2] «Corona Riccardo | Corona Riccardo (c. 1878October 15, 1917) was an Italian born American actress
who had a brief Broadway stage career before leaving to become a wife and mother. Born in Naples she
came to acting in 1894 playing a Mexican girl in a play at the Empire Theatre. Wilson Barrett engaged
her for a role in his play "The Sign of the Cross" which he took on tour of the United States. Riccardo
played the role of Ancaria and later played Berenice in the same play. Robert B. Mantell in 1898 who
struck by her beauty also cast her in two Shakespeare plays, "Romeo and Juliet" and "Othello". Author
Lewis Strang writing in 1899 said Riccardo was the most promising actress in America at the time.
Towards the end of 1898 Mantell chose her for another Shakespeare part, Ophelia im Hamlet. Afterwards
she was due to join Augustin Daly's Theatre Company but Daly died in 1899. In 1899 she gained her
biggest fame by playing Iras in the first stage production of Ben-Hur.»

Question: Who acted in the shot film The Shore and is also the youngest actress ever to play Ophelia in
a Royal Shakespeare Company production of "Hamlet." ?

Past Query: "actress who played Ophelia in Royal Shakespeare Company production of Hamlet" + "actress
in short film The Shore"

Instructions: Query should be short and less than 100 characters

Query: "actress Ophelia RSC Hamlet" + "actress The Shore"
```

## Assertion-Driven Optimizations

DSPy Assertions work with optimizations that DSPy offers, particularly with `BootstrapFewShotWithRandomSearch`, including the following settings:

- Compilation with Assertions This includes assertion-driven example bootstrapping and counterexample bootstrapping during compilation. The teacher model for bootstrapping few-shot demonstrations can make use of DSPy Assertions to offer robust bootstrapped examples for the student model to learn from during inference. In this setting, the student model does not perform assertion aware optimizations (backtracking and retry) during inference.
- Compilation + Inference with Assertions -This includes assertion-driven optimizations in both compilation and inference. Now the teacher model offers assertion-driven examples but the student can further optimize with assertions of its own during inference time.

```
teleprompter = BootstrapFewShotWithRandomSearch(
    metric=validate_context_and_answer_and_hops,
    max_bootstrapped_demos=max_bootstrapped_demos,
    num_candidate_programs=6,
)

#Compilation with Assertions
compiled_with_assertions_baleen = teleprompter.compile(student = baleen, teacher =
baleen_with_assertions, trainset = trainset, valset = devset)

#Compilation + Inference with Assertions
compiled_baleen_with_assertions = teleprompter.compile(student=baleen_with_assertions, teacher =
baleen_with_assertions, trainset=trainset, valset=devset)
```

---

**Written By:** Arnav Singhvi

# ChainOfThoughtWithHint

This class builds upon the `ChainOfThought` class by introducing an additional input field to provide hints for reasoning. The inclusion of a hint allows for a more directed problem-solving process, which can be especially useful in complex scenarios.

`ChainOfThoughtWithHint` is instantiated with a user-defined DSPy Signature, and the inclusion of a `hint` argument that takes in a string-form phrase to provide a hint within the prompt template.

Let's take a look at an example:

```
class BasicQA(dspy.Signature):
    """Answer questions with short factoid answers."""
    question = dspy.InputField()
    answer = dspy.OutputField(desc="often between 1 and 5 words")

#Pass signature to ChainOfThought module
generate_answer = dspy.ChainOfThoughtWithHint(BasicQA)

# Call the predictor on a particular input alongside a hint.
question='What is the color of the sky?'
hint = "It's what you often see during a sunny day."
pred = generate_answer(question=question, hint=hint)

print(f"Question: {question}")
print(f"Predicted Answer: {pred.answer}")
```

---

**Written By:** Arnav Singhvi

# Program of Thought

## Background

DSPy supports the Program of Thought (PoT) prompting technique, integrating an advanced module capable of problem-solving with program execution capabilities. PoT builds upon Chain of Thought by translating reasoning steps into executable programming language statements through iterative refinement. This enhances the accuracy of the output and self-corrects errors within the generated code. Upon completing these iterations, PoT delegates execution to a program interpreter. Currently, this class supports the generation and execution of Python code.

## Instantiating Program of Thought

Program of Thought is instantiated based on a user-defined DSPy Signature, which can take a simple form such as `input_fields -> output_fields`. Users can also specify a `max_iters` to set the maximum number of iterations for the self-refinement process of the generated code. The default value is 3 iterations. Once the maximum iterations are reached, PoT will produce the final output.

```
import dsp
import dspy
from ..primitives.program import Module
from ..primitives.python_interpreter import CodePrompt, PythonInterpreter
import re

class ProgramOfThought(Module):
    def __init__(self, signature, max_iters=3):
        ...
```

```
#Example Usage

#Define a simple signature for basic question answering
generate_answer_signature = dspy.Signature("question -> answer")
generate_answer_signature.attach(question=("Question:", "")).attach(answer=("Answer:", "often between 1 and 5 words"))

# Pass signature to ProgramOfThought Module
pot = dspy.ProgramOfThought(generate_answer_signature)
```

## Under the Hood

Program of Thought operates in 3 key modes: generate, regenerate, and answer. Each mode is a Chain of Thought module encapsulating signatures and instructions for each mode's individual purpose. These are dynamically created as the PoT iterations complete, accounting for the user's input and output fields internally.

**Generate mode:**

Initiates the generation of Python code with the signature `(question -> generated_code)` for the initial code generation.

**Regenerate mode:**

Used for refining the generation of Python code, considering the previously generated code and existing errors, with the signature `(question, previous_code, error -> generated_code)`. If errors persist past the maximum iterations, the user is alerted and the output is returned as `None`.

**Answer mode:**

Executes the last stored generated code and outputs the final answer, with the signature `(question, final_generated_code, code_output -> answer)`.

**Key internals:**

**Key Internals:**

- **Code Parsing:** Program of Thought internally processes each code generation as a string and filters out extraneous bits to ensure the code block conforms to executable Python syntax. If the code is empty or does not match these guidelines, the parser returns an error string, signaling the PoT process for regeneration.
- **Code Execution:** Program of Thought relies on a Python interpreter adapted by CAMEL-AI to execute code generated by LLMs. The final code generation is formatted as a CodePrompt instance and executed by the PythonInterpreter. This adaptation is present in DSPy primitives.

## Tying It All Together

Using ProgramOfThought mirrors the simplicity of the base `Predict` and `ChainOfThought` modules. Here is an example call:

```
#Call the ProgramOfThought module on a particular input
question = 'Sarah has 5 apples. She buys 7 more apples from the store. How many apples does Sarah have now?'
result = pot(question=question)

print(f"Question: {question}")
print(f"Final Predicted Answer (after ProgramOfThought process): {result.answer}")
```

```
Question: Sarah has 5 apples. She buys 7 more apples from the store. How many apples does Sarah have now?
Final Predicted Answer (after ProgramOfThought process): 12
```

Let's take a peek at how ProgramOfThought functioned internally by inspecting its history, up to maximum iterations (+ 1 to view the final generation). (This assumes the initial DSPy setup and configurations of LMs and RMs).

`lm.inspect_history(n=4)`

```
You will be given `question` and you will respond with `generated_code`.
Generating executable Python code that programmatically computes the correct `generated_code`.
After you're done with the computation, make sure the last line in your code evaluates to the correct value for `generated_code`.

---

Follow the following format.

Question:
Reasoning: Let's think step by step in order to ${produce the generated_code}. We ...
Code: python code that answers the question

---

Question: Sarah has 5 apples. She buys 7 more apples from the store. How many apples does Sarah have now?
Reasoning: Let's think step by step in order to produce the generated_code. We start with the initial number of apples that Sarah has, which is 5. Then, we add the number of apples she buys from the store, which is 7. Finally, we compute the total number of apples Sarah has by adding these two quantities together.
Code:

apples_initial = 5
apples_bought = 7

total_apples = apples_initial + apples_bought
total_apples

Given the final code `question`, `final_generated_code`, `code_output`, provide the final `answer`.

---
```

```
---

Follow the following format.

Question:

Code: python code that answers the question

Code Output: output of previously-generated python code

Reasoning: Let's think step by step in order to ${produce the answer}. We ...

Answer: often between 1 and 5 words

---

Question: Sarah has 5 apples. She buys 7 more apples from the store. How many apples does Sarah have
now?

Code:
apples_initial = 5
apples_bought = 7

total_apples = apples_initial + apples_bought
total_apples

Code Output: 12

Reasoning: Let's think step by step in order to produce the answer. We start with the initial number of
apples Sarah has, which is 5. Then, we add the number of apples she bought, which is 7. The total
number of apples Sarah has now is 12.

Answer: 12
```

---

**Written By:** Arnav Singhvi

# ReAct

## Background

DSPy supports ReAct, an LLM agent designed to tackle complex tasks in an interactive fashion. ReAct is composed of an iterative loop of interpretation, decision and action-based activities ("Thought, Action, and Observation") based on an evolving set of input and output fields. Through this real-time iterative approach, the ReAct agent can both analyze and adapt to its responses over time as new information becomes available.

## Instantiating ReAct

To instantiate the ReAct module, define and pass in a DSPy Signature.

```
# Define a simple signature for basic question answering
class BasicQA(dspy.Signature):
    """Answer questions with short factoid answers."""
    question = dspy.InputField()
    answer = dspy.OutputField(desc="often between 1 and 5 words")

# Pass signature to ReAct module
react_module = dspy.ReAct(BasicQA)
```

## Under the Hood

### ReAct Cycle

ReAct operates under a dynamic signature integration process, accounting for the signature inputs and performing the Thoughts, Action, Observation cycle to respond with the signature outputs. Thoughts (or reasoning) lead to Actions (such as queries or activities). These Actions then result in Observations (like results or responses), which subsequently feedback into the next Thought.

This cycle is maintained for a predefined number of iterations, specified by `max_iters`. The default value for the Thought-Action-Observation cycle is 5 iterations. Once the maximum iterations are reached, React will return the final output if the Action has finished `(Finish[answer])` or an empty string to indicate the agent could not determine a final output.

> **CAUTION**
>
> Currently, ReAct supports only one output field in its signature. We plan to expand this in future developments.

### ReAct Tools

Tools in ReAct can shape the agent's interaction and response mechanisms, and DSPy ensures this customizability by allowing users to pass in their toolsets tailored for their task scenarios. The default tool is the `dspy.Retrieve` module (serving to retrieve information from Retrieval Models during the Action step) with default `num_results=3`, and these can be passed as arguments to the initialization of the ReAct module.

## Tying It All Together

Using ReAct mirrors the simplicity of the base `Predict` and `ChainOfThought` modules. Here is an example call:

```
# Call the ReAct module on a particular input
question = 'Aside from the Apple Remote, what other devices can control the program Apple Remote was
originally designed to interact with?'
result = react_module(question=question)

print(f"Question: {question}")
print(f"Final Predicted Answer (after ReAct process): {result.answer}")
```

```
Question: Aside from the Apple Remote, what other devices can control the program Apple Remote was
originally designed to interact with?
Final Predicted Answer (after ReAct process): The Apple Remote and the Siri Remote can control the
```

```
Front Row media program.
```

Let's take a peek at how ReAct functioned internally by inspecting its history, up to maximum iterations. (This assumes the initial DSPy setup and configurations of LMs and RMs).

`lm.inspect_history(n=3)`

```
------------------------Step 1---------------------------------
You will be given `question` and you will respond with `answer`.

To do this, you will interleave Thought, Action, and Observation steps.

Thought can reason about the current situation, and Action can be the following types:

(1) Search[query], which takes a search query and returns one or more potentially relevant passages
from a corpus
(2) Finish[answer], which returns the final `answer` and finishes the task

---

Follow the following format.

Question: ${question}
Thought 1: next steps to take based on last observation
Action 1: always either Search[query] or, when done, Finish[answer]

---

Question: Aside from the Apple Remote, what other devices can control the program Apple Remote was
originally designed to interact with?
Thought 1: I need to find out what program the Apple Remote was originally designed to interact with.
Action 1: Search["program Apple Remote was originally designed to interact with"]

------------------------Step 2---------------------------------
You will be given `question` and you will respond with `answer`.

To do this, you will interleave Thought, Action, and Observation steps.

Thought can reason about the current situation, and Action can be the following types:

(1) Search[query], which takes a search query and returns one or more potentially relevant passages
from a corpus
(2) Finish[answer], which returns the final `answer` and finishes the task

---

Follow the following format.

Question: ${question}

Thought 1: next steps to take based on last observation

Action 1: always either Search[query] or, when done, Finish[answer]

Observation 1: observations based on action

Thought 2: next steps to take based on last observation

Action 2: always either Search[query] or, when done, Finish[answer]

---

Question: Aside from the Apple Remote, what other devices can control the program Apple Remote was
originally designed to interact with?
```

```
[Thought 1, Action 1]

Observation 1:
[1] «Apple Remote | The Apple Remote is a remote control device released [...].»
[2] «ITunes Remote | iTunes Remote (also known simply as Remote) is a software [...].»
[3] «Siri Remote | The Siri Remote is a remote control device released in tandem [...]»

Thought 2: I have found that the Apple Remote was originally designed to interact with the Front Row
media program on the iSight iMac G5. I should search for other devices that can control this program.

Action 2: Search["devices that can control Front Row media program"]

Observation 2:
[1] «Apple Remote» - The Apple Remote is the primary control mechanism for the Front Row media program
[...].
[2] «Siri Remote» - The Siri Remote is the successor of the Apple Remote and is designed [...].

-------------------------Step 3--------------------------------

You will be given `question` and you will respond with `answer`.

To do this, you will interleave Thought, Action, and Observation steps.

Thought can reason about the current situation, and Action can be the following types:

(1) Search[query], which takes a search query and returns one or more potentially relevant passages
from a corpus
(2) Finish[answer], which returns the final `answer` and finishes the task

---

Follow the following format.

Question: ${question}

Thought 1: next steps to take based on last observation

Action 1: always either Search[query] or, when done, Finish[answer]

Observation 1: observations based on action

Thought 2: next steps to take based on last observation

Action 2: always either Search[query] or, when done, Finish[answer]

Observation 2: observations based on action

Thought 3: next steps to take based on last observation

Action 3: always either Search[query] or, when done, Finish[answer]

---

Question: Aside from the Apple Remote, what other devices can control the program Apple Remote was
originally designed to interact with?

[Thought 1, Action 1, Observation 1, Thought 2, Action 2]

Observation 2:
[1] «Front Row (software) | Front Row is a discontinued media center [...].»
[2] «Apple Remote | The Apple Remote is a remote control device [...].»
[3] «Media player (software) | A media player is a computer program for [...].»

Thought 3: I have found that the Apple Remote and the Siri Remote can control the Front Row media
program. I should provide this information as the answer.

Action 3: Finish[The Apple Remote and the Siri Remote can control the Front Row media program.]
```

```
Answer: The Apple Remote and the Siri Remote can control the Front Row media program.
```

---

**Written By:** Arnav Singhvi

# Guide: DSPy Modules

Open in Colab

## Quick Recap

This guide assumes you followed the intro tutorial to build your first few DSPy programs.

Remember that **DSPy program** is just Python code that calls one or more **DSPy modules**, like `dspy.Predict` or `dspy.ChainOfThought`, to use LMs.

## 1) What is a DSPy Module?

A **DSPy module** is a building block for programs that use LMs.

- Each built-in module abstracts a **prompting technique** (like chain of thought or ReAct). Crucially, they are generalized to handle any DSPy Signature.
- A DSPy module has **learnable parameters** (i.e., the little pieces comprising the prompt and the LM weights) and can be invoked (called) to process inputs and return outputs.
- Multiple modules can be composed into bigger modules (programs). DSPy modules are inspired directly by NN modules in PyTorch, but applied to LM programs.

## 2) What DSPy Modules are currently built-in?

1. `dspy.Predict`:
2. `dspy.ChainOfThought`:
3. `dspy.ProgramOfThought`:
4. `dspy.ReAct`:
5. `dspy.MultiChainComparison`:

We also have some function-style modules:

6. `dspy.majority`:

## 3) How do I use a built-in module, like `dspy.Predict` or `dspy.ChainOfThought`?

Let's start with the most fundamental one, `dspy.Predict`. Internally, all of the others are just built using it!

We'll assume you are already at least a little familiar with DSPy signatures, which are declarative specs for defining the behavior of any module we use in DSPy. To use a module, we first **declare** it by giving it a signature. Then we **call** the module with the input arguments, and extract the output fields!

```
sentence = "it's a charming and often affecting journey."  # example from the SST-2 dataset.

# 1) Declare with a signature.
classify = dspy.Predict('sentence -> sentiment')

# 2) Call with input argument(s).
response = classify(sentence=sentence)

# 3) Access the output.
print(response.sentiment)
```

```
Positive
```

When we declare a module, we can pass configuration keys to it.

Below, we'll pass `n=5` to request five completions. We can also pass `temperature` or `max_len`, etc.

Let's use `dspy.ChainOfThought`. In many cases, simply swapping `dspy.ChainOfThought` in place of `dspy.Predict` improves quality.

```
question = "What's something great about the ColBERT retrieval model?"

# 1) Declare with a signature, and pass some config.
classify = dspy.ChainOfThought('question -> answer', n=5)

# 2) Call with input argument.
response = classify(question=question)

# 3) Access the outputs.
response.completions.answer
```

```
['One great thing about the ColBERT retrieval model is its superior efficiency and effectiveness
compared to other models.',
 'Its ability to efficiently retrieve relevant information from large document collections.',
 'One great thing about the ColBERT retrieval model is its superior performance compared to other
models and its efficient use of pre-trained language models.',
 'One great thing about the ColBERT retrieval model is its superior efficiency and accuracy compared to
other models.',
 'One great thing about the ColBERT retrieval model is its ability to incorporate user feedback and
support complex queries.']
```

Let's discuss the output object here.

The `dspy.ChainOfThought` module will generally inject a `rationale` before the output field(s) of your signature.

Let's inspect the (first) rationale and answer!

```
print(f"Rationale: {response.rationale}")
print(f"Answer: {response.answer}")
```

Rationale: produce the answer. We can consider the fact that ColBERT has shown to outperform other state-of-the-art retrieval models in terms of efficiency and effectiveness. It uses contextualized embeddings and performs document retrieval in a way that is both accurate and scalable. Answer: One great thing about the ColBERT retrieval model is its superior efficiency and effectiveness compared to other models.

This is accessible whether we request one or many completions.

We can also access the different completions as a list of `Prediction`s or as several lists, one for each field.

```
response.completions[3].rationale == response.completions.rationale[3]
```

```
True
```

## 4) How do I use more complex built-in modules?

The others are very similar, `dspy.ReAct` and `dspy.ProgramOfThought` etc. They mainly change the internal behavior with which

your signature is implemented!

Check out further examples in each module's respective guide.

### 5) How do I compose multiple modules into a bigger program?

DSPy is just Python code that uses modules in any control flow you like. (There's some magic internally at `compile` time to trace your LM calls.)

What this means is that, you can just call the modules freely. No weird abstractions for chaining calls.

This is basically PyTorch's design approach for define-by-run / dynamic computation graphs. Refer to the intro tutorials for examples.

---

**Written By:** Arnav Singhvi

# Retrieve

## Background

DSPy supports retrieval through the Retrieve module that serves to process user queries and output relevant passages from retrieval corpuses. This module ties in with the DSPy-supported Retrieval Model Clients which are retrieval servers that users can utilize to retrieve information for information retrieval tasks.

## Instantiating Retrieve

Retrieve is simply instantiate by a user-defined `k` number of retrieval passages to return for a query.

```
class Retrieve(Parameter):
    def __init__(self, k=3):
        self.stage = random.randbytes(8).hex()
        self.k = k
```

Additionally, configuring a Retrieval model client or server through `dspy.configure` allows for user retrieval in DSPy programs through internal calls from Retrieve.

```
#Example Usage

#Define a retrieval model server to send retrieval requests to
colbertv2_wiki17_abstracts = dspy.ColBERTv2(url='http://20.102.90.50:2017/wiki17_abstracts')

#Configure retrieval server internally
dspy.settings.configure(rm=colbertv2_wiki17_abstracts)

#Define Retrieve Module
retriever = dspy.Retrieve(k=3)
```

### Under the Hood

Retrieve makes use of the internally configured retriever to send a single query or list of multiple queries to determine the top-k relevant passages. The module queries the retriever for each provided query, accumulating scores (or probabilities if the `by_prob` arg is set) for each passage and returns the passages sorted by their cumulative scores or probabilities.

The Retrieve module can also integrate a reranker if this is configured, in which case, the reranker re-scores the retrieved passages based on their relevance to the quer, improving accuracy of the results.

### Tying it All Together

We can now call the Retrieve module on a sample query or list of queries and observe the top-K relevant passages.

```
query='When was the first FIFA World Cup held?'

# Call the retriever on a particular query.
topK_passages = retriever(query).passages

print(f"Top {retriever.k} passages for question: {query} \n", '-' * 30, '\n')

for idx, passage in enumerate(topK_passages):
    print(f'{idx+1}]', passage, '\n')
```

```
Top 3 passages for question: When was the first FIFA World Cup held?
 ------------------------------

1] History of the FIFA World Cup | The FIFA World Cup was first held in 1930, when FIFA president Jules
```

```
Rimet [...].

2] 1950 FIFA World Cup | The 1950 FIFA World Cup, held in Brazil from 24 June to 16 July 1950, [...].

3] 1970 FIFA World Cup | The 1970 FIFA World Cup was the ninth FIFA World Cup, the quadrennial [...].
```

---

**Written By:** Arnav Singhvi

# Functional Typed Predictors

## Typed Predictors as Decorators

## Functional Typed Predictors in `dspy.Module`

## How Functional Typed Predictors work?

# Understanding Typed Predictors

## Why use a Typed Predictor?

## How to use a Typed Predictor?

## Prompt of Typed Predictors

## How Typed Predictors work?

# Anyscale

## Anyscale

### Prerequisites

- Anyscale `api_key` and `api_base` (***for non-cached examples***). Set these within your developer environment `.env` as follows:

```
ANYSCALE_API_BASE = ...
ANYSCALE_API_KEY = ...
```

which will be retrieved within the Anyscale Client as:

```
self.api_base = os.getenv("ANYSCALE_API_BASE")
self.token = os.getenv("ANYSCALE_API_KEY")
```

### Setting up the Anyscale Client

The constructor initializes the `HFModel` base class to support the handling of prompting models. This requires the following parameters:

**Parameters:**

- `model` (*str*): ID of model hosted on Anyscale endpoint.
- `**kwargs`: Additional keyword arguments to configure the Anyscale client.

Example of the Anyscale constructor:

```
class Anyscale(HFModel):
    def __init__(self, model, **kwargs):
```

### Under the Hood

**`_generate(self, prompt, use_chat_api=False, **kwargs):`**
**Parameters:**

- `prompt` (*str*): Prompt to send to Anyscale.
- `use_chat_api` (*bool*): Flag to use the Anyscale Chat models endpoint. Defaults to False.
- `**kwargs`: Additional keyword arguments for completion request.

**Returns:**

- `dict`: dictionary with `prompt` and list of response `choices`.

Internally, the method handles the specifics of preparing the request prompt and corresponding payload to obtain the response.

The Anyscale token is set within the request headers to ensure authorization to send requests to the endpoint.

If `use_chat_api` is set, the method sets up Anyscale url chat endpoint and prompt template for chat models. It then retrieves the generated JSON response and sets up the `completions` list by retrieving the response's `message` : `content`.

If `use_chat_api` is not set, the method uses the default Anyscale url endpoint. It similarly retrieves the generated JSON response and but sets up the `completions` list by retrieving the response's `text` as the completion.

Finally, after processing the requests and responses, the method constructs the response dictionary with two keys: the original request `prompt` and `choices`, a list of dictionaries representing generated `completions` with the key `text` holding the response's

generated text.

### Using the Anyscale client

```
anyscale = dspy.Anyscale(model='meta-llama/Llama-2-70b-chat-hf')
```

### Sending Requests via Anyscale Client

1. ***Recommended*** Configure default LM using `dspy.configure`.

This allows you to define programs in DSPy and simply call modules on your input fields, having DSPy internally call the prompt on the configured LM.

```
dspy.configure(lm=anyscale)

#Example DSPy CoT QA program
qa = dspy.ChainOfThought('question -> answer')

response = qa(question="What is the capital of Paris?") #Prompted to anyscale
print(response.answer)
```

2. Generate responses using the client directly.

```
response = anyscale(prompt='What is the capital of Paris?')
print(response)
```

---

**Written By:** Arnav Singhvi

# Cohere

## Cohere

### Prerequisites

```
pip install cohere
```

- Cohere `api_key` (***for non-cached examples***)

### Setting up the Cohere Client

The constructor initializes the base class `LM` to support prompting requests to Cohere models. This requires the following parameters:

**Parameters:**

- `model` (*str*): Cohere pretrained models. Defaults to `command-xlarge-nightly`.
- `api_key` (*Optional[str]*, *optional*): API provider provider authentication token. Defaults to `None`. This then internally initializes the `cohere.Client`.
- `stop_sequences` (*List[str]*, *optional*): List of stopping tokens to end generation.
- `max_num_generations` *internally set*: Maximum number of completions generations by Cohere client. Defaults to 5.

Example of the Cohere constructor:

```
class Cohere(LM):
    def __init__(
        self,
        model: str = "command-xlarge-nightly",
        api_key: Optional[str] = None,
        stop_sequences: List[str] = [],
    ):
```

### Under the Hood

**`__call__(self, prompt: str, only_completed: bool = True, return_sorted: bool = False, **kwargs) -> List[Dict[str, Any]]`**

**Parameters:**

- `prompt` (*str*): Prompt to send to Cohere.
- `only_completed` (*bool*, *optional*): Flag to return only completed responses and ignore completion due to length. Defaults to True.
- `return_sorted` (*bool*, *optional*): Flag to sort the completion choices using the returned averaged log-probabilities. Defaults to False.
- `**kwargs`: Additional keyword arguments for completion request.

**Returns:**

- `List[str]`: List of generated completions.

Internally, the method handles the specifics of preparing the request prompt and corresponding payload to obtain the response.

The method calculates the number of iterations required to generate the specified number of completions `n` based on the `self.max_num_generations` that the Cohere model can produce in a single request. As it completes the iterations, it updates the official `num_generations` argument passed to the request payload and calls request with the updated arguments.

This process iteratively constructs a `choices` list from which the generated completions are retrieved.

This process iteratively constructs a `choices` list from which the generated completions are retrieved.

If `return_sorted` is set and more than one generation is requested, the completions are sorted by their likelihood scores in descending order and returned as a list with the most likely completion appearing first.

### Using the Cohere client

```
cohere = dsp.Cohere(model='command-xlarge-nightly')
```

### Sending Requests via Cohere Client

1. ***Recommended*** Configure default LM using `dspy.configure`.

This allows you to define programs in DSPy and simply call modules on your input fields, having DSPy internally call the prompt on the configured LM.

```
dspy.configure(lm=cohere)

#Example DSPy CoT QA program
qa = dspy.ChainOfThought('question -> answer')

response = qa(question="What is the capital of Paris?") #Prompted to cohere
print(response.answer)
```

2. Generate responses using the client directly.

```
response = cohere(prompt='What is the capital of Paris?')
print(response)
```

---

**Written By:** Arnav Singhvi

# OpenAI

## OpenAI

### Prerequisites

- OpenAI `api_key` (***for non-cached examples***)

### Setting up the OpenAI Client

The constructor initializes the base class `LM` to support prompting requests to OpenAI models. This requires the following parameters:

- `api_key` (*Optional[str]*, *optional*): OpenAI API provider authentication token. Defaults to `None`.
- `api_provider` (*Literal["openai", "azure"]*, *optional*): OpenAI API provider to use. Defaults to `"openai"`.
- `api_base` (Optional[str], optional): Base URL for the OpenAI API endpoint. Defaults to `None`.
- `model_type` (*Literal["chat", "text"]*): Specified model type to use. Defaults to `"gpt-3.5-turbo-instruct"`.
- `**kwargs`: Additional language model arguments to pass to OpenAI request. This is initialized with default values for relevant text generation parameters needed for communicating with the GPT API, such as `temperature`, `max_tokens`, `top_p`, `frequency_penalty`, `presence_penalty`, and `n`.

Example of the OpenAI constructor:

```
class GPT3(LM): #This is a wrapper for the OpenAI class - dspy.OpenAI = dsp.GPT3
    def __init__(
        self,
        model: str = "gpt-3.5-turbo-instruct",
        api_key: Optional[str] = None,
        api_provider: Literal["openai", "azure"] = "openai",
        api_base: Optional[str] = None,
        model_type: Literal["chat", "text"] = None,
        **kwargs,
    ):
```

### Under the Hood

**`__call__(self, prompt: str, only_completed: bool = True, return_sorted: bool = False, **kwargs) -> List[Dict[str, Any]]`**

**Parameters:**

- `prompt` (*str*): Prompt to send to OpenAI.
- `only_completed` (*bool*, *optional*): Flag to return only completed responses and ignore completion due to length. Defaults to True.
- `return_sorted` (*bool*, *optional*): Flag to sort the completion choices using the returned averaged log-probabilities. Defaults to False.
- `**kwargs`: Additional keyword arguments for completion request.

**Returns:**

- `List[Dict[str, Any]]`: List of completion choices.

Internally, the method handles the specifics of preparing the request prompt and corresponding payload to obtain the response.

After generation, the completions are post-processed based on the `model_type` parameter. If the parameter is set to 'chat', the generated content look like `choice["message"]["content"]`. Otherwise, the generated text will be `choice["text"]`.

### Using the OpenAI client

```
turbo = dspy.OpenAI(model='gpt-3.5-turbo')
```

## Sending Requests via OpenAI Client

1. ***Recommended*** Configure default LM using `dspy.configure`.

This allows you to define programs in DSPy and simply call modules on your input fields, having DSPy internally call the prompt on the configured LM.

```
dspy.configure(lm=turbo)

#Example DSPy CoT QA program
qa = dspy.ChainOfThought('question -> answer')

response = qa(question="What is the capital of Paris?") #Prompted to turbo
print(response.answer)
```

2. Generate responses using the client directly.

```
response = turbo(prompt='What is the capital of Paris?')
print(response)
```

---

**Written By:** Arnav Singhvi

# PremAI

## PremAI

PremAI is an all-in-one platform that simplifies the process of creating robust, production-ready applications powered by Generative AI. By streamlining the development process, PremAI allows you to concentrate on enhancing user experience and driving overall growth for your application.

### Prerequisites

Refer to the quick start guide to getting started with the PremAI platform, create your first project and grab your API key.

### Setting up the PremAI Client

The constructor initializes the base class `LM` to support prompting requests to supported PremAI hosted models. This requires the following parameters:

- `model` (*str*): Models supported by PremAI. Example: `mistral-tiny`. We recommend using the model selected in project launchpad.
- `project_id` (*int*): The project id which contains the model of choice.
- `api_key` (*Optional[str]*, *optional*): API provider from PremAI. Defaults to None.
- `**kwargs`: Additional language model arguments will be passed to the API provider.

Example of PremAI constructor:

```
class PremAI(LM):
    def __init__(
        self,
        model: str,
        project_id: int,
        api_key: str,
        base_url: Optional[str] = None,
        session_id: Optional[int] = None,
        **kwargs,
    ) -> None:
```

### Under the Hood

**`__call__(self, prompt: str, **kwargs) -> str`**

**Parameters:**

- `prompt` (*str*): Prompt to send to PremAI.
- `**kwargs`: Additional keyword arguments for completion request. You can find all the additional kwargs here.

**Returns:**

- `str`: Completions string from the chosen LLM provider

Internally, the method handles the specifics of preparing the request prompt and corresponding payload to obtain the response.

### Using the PremAI client

```
premai_client = dspy.PremAI(project_id=1111)
```

Please note that, this is a dummy `project_id`. You need to change this to the project_id you are interested to use with dspy.

```
dspy.configure(lm=premai_client)

#Example DSPy CoT QA program
```

```
qa = dspy.ChainOfThought('question -> answer')

response = qa(question="What is the capital of Paris?")
print(response.answer)
```

2. Generate responses using the client directly.

```
response = premai_client(prompt='What is the capital of Paris?')
print(response)
```

## Native RAG Support

PremAI Repositories allow users to upload documents (.txt, .pdf, etc.) and connect those repositories to the LLMs to serve as vector databases and support native RAG. You can learn more about PremAI repositories here.

Repositories are also supported through the dspy-premai integration. Here is how you can use this workflow:

```
query = "what is the diameter of individual Galaxy"
repository_ids = [1991, ]
repositories = dict(
    ids=repository_ids,
    similarity_threshold=0.3,
    limit=3
)
```

First, we start by defining our repository with some valid repository ids. You can learn more about how to get the repository id here.

> Note: This is similar to LM integrations where now you are overriding the repositories connected in the launchpad when you invoke the argument' repositories'.

Now, we connect the repository with our chat object to invoke RAG-based generations.

```
response = premai_client(prompt=query, max_tokens=100, repositories=repositories)

print(response)
print("---")
print(json.dumps(premai_client.history, indent=4))
```