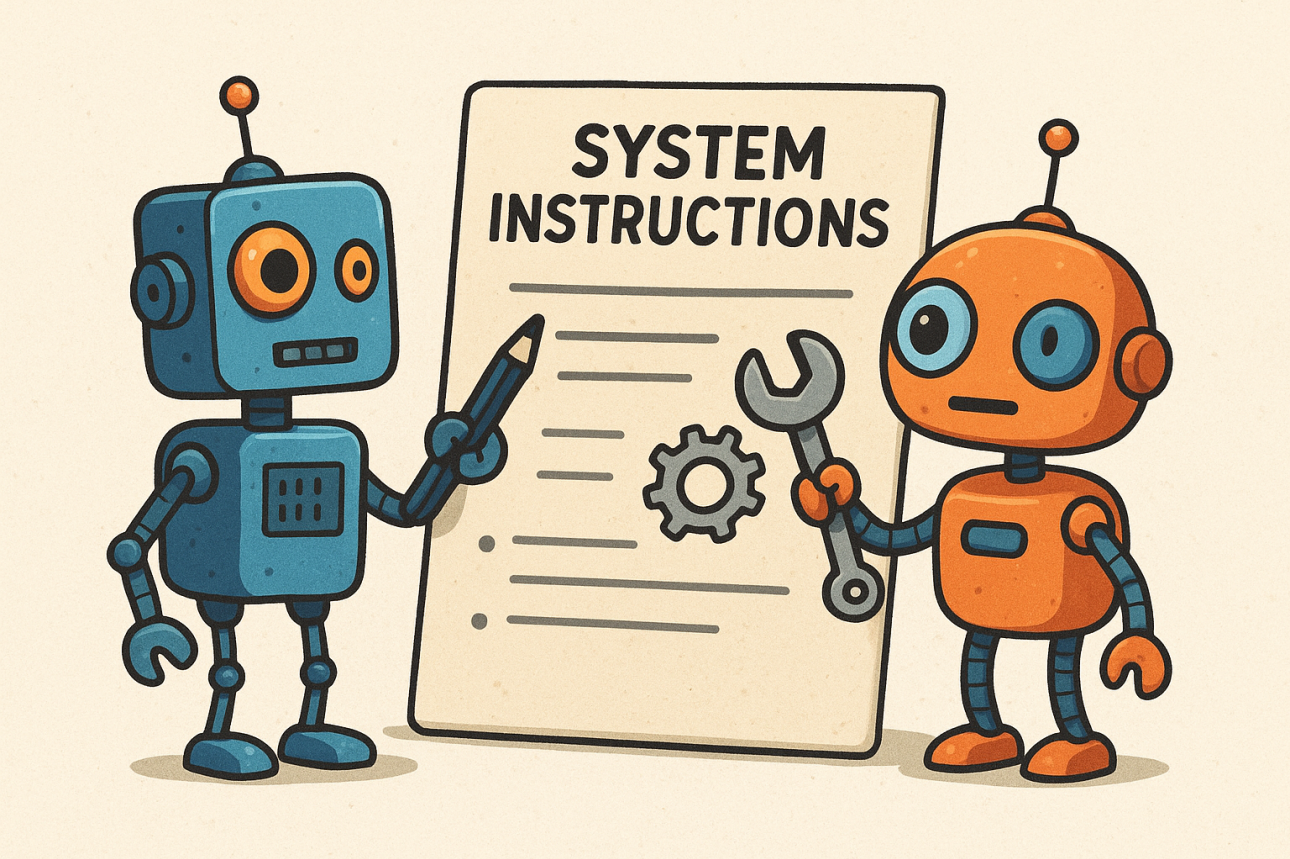
Systematic LLM Prompt Engineering Using DSPy Optimization

1.0 The challenge of prompt iteration

Robert Martin-Short • 27 min read • 2025-08-25

https://towardsdatascience.com/systematic-llm-prompt-engineering-using-dspy-optimization/



A practical methodology for optimizing both generator and LLM judge prompts using automated techniques

ChatGPT's interpretation of "two robots working together to improve their prompt". Image generated by the author.

This article is a journey into the fascinating and rapidly evolving science of LLM prompt iteration, which is a fundamental part of Large Language Model Operations (LLMOPs). We'll use the example of generating customer service responses with a real-world dataset to show how both generator and LLM-judge prompts can be developed in a systematic fashion with DSPy. All the code for this project, including several notebooks and a nice README written by Claude, can be found here.

The field of applied AI, which typically involves building pipelines that connect data to Large Language Models (LLMs) in a way that generates business value, is evolving rapidly. There is a large and growing array of open and closed source models for developers to choose from, and

many of the latest models match or surpass expert human level performance in generative tasks such as coding and technical writing. Several flagship models such as Gemini 2.5 are also natively multimodal, with video and audio capabilities that facilitate conversation with users in an uncannily human-like fashion. Indeed, AI assistance is already quickly creeping into our daily lives - for example I've been using it more and more over the last few months for coding, brainstorming ideas and general advice. It's a new world that we are all learning to navigate and with such powerful technology at our fingertips, it's easy to get started and build a POC. But LLM-powered projects still face significant hurdles on their way from research to production.

Prompt iteration - central to building trustworthy generative AI products - is hard because there are so many ways to write a prompt, models can be very sensitive to small changes and with generative tasks judgement of the results is often subjective. Over time, prompts can grow through iteration and edge case fixes into complicated text files that are highly optimized against a particular model version. This presents a challenge when teams want to upgrade to the latest version or switch model provider, likely requiring significant refactoring. If a regression testing process is established, then iteration is possible by incrementally adjusting the generator prompt, re-running the test suite and evaluating the result. However this process is also tedious and evaluation can be highly subjective - even teams fortunate enough to employ subject matter experts will struggle when those experts disagree.

2.0 Who evaluates the output?

LLMs are non-deterministic, generative models. These attributes are core to their functionality, but make them difficult to evaluate. Traditional natural language processing metrics are rarely applicable, and often there is no ground truth against which to compare the LLM outputs. The concept of LLM judges can help here but adds complexity. An LLM judge is typically a powerful model that tries to do the job of an expert annotator by determining the quality of the generator model output. State of the art foundation models are typically very good at classifying whether or not a generated output meets certain pre-defined criteria. Therefore, in the ideal situation we have a judge whose prompt distills the thought process of a human SME and which produces outputs that are reliably aligned with SME consensus. We can then apply this automatically to a representative development set and compare results across versions of the generator prompt to make sure we're iterating in the right direction.

3.0 Adding complexity

How do we know that the LLM judge is reliable? Human effort is usually still needed to label a training set for the judge, whose prompt can then be aligned to generate results that match the human labels as far as possible. All the complexities and model version dependencies that we discussed above with reference to the generator prompt also apply to LLM judges and in a multi-stage project with several LLM calls there may also need to be several judges, each with their own training set.

Even for a single-prompt workflow, our setup has now become quite complex. We have a generator prompt that we want to iterate on, and a development dataset of inputs. We then have a separate dataset of inputs and outputs that is used to develop the judge. That dataset should then be labelled by an SME and split into a training portion - to build the judge prompt - and a test

portion, to test the judge prompt and prevent overfitting. Once the judge has been trained, it is used to optimize the generator prompt against the development dataset. We then ideally need another holdout set which we can use to check whether or not the generator prompt has been overfit to the development set. Prompt versions and datasets should be logged so that experiments are reproducible. The diagram below illustrates this situation.

Figure 1: High level workflow for generator and LLM judge prompt optimization. What started as a seemingly simple process of updating a POC generator prompt can become a complex pipeline that would benefit from versioning, experiment tracking and other LLMOPs concepts. Image generated by the author.

With so many components for the optimization of a single generator prompt, we ideally need an LLMOps framework with built in logging and tracking. Description of such frameworks is beyond the scope of this article, but I would recommend the mlflow documentation or the later modules of this excellent class for more information.

4.0 Purpose of this article

In this article, we will build the essential components of the system shown in figure 1 for a simple toy problem that involves generating helpful customer service responses. We'll make use of a modified version of this airline customer support messages dataset, which is available from Kaggle with a CC0 license. A sample of this dataset was used to seed the generation of synthetic conversation set, which is then used as the starting point for this project. Details of the data generation process are shared in the next section.

The generator model's aim will be to take a conversation and produce the next support agent message that is as helpful as possible. To keep track of the prompts and perform optimization, we'll use the DSPy orchestration library. DSPy is unique in that it abstracts away raw, text based prompts into modular Python code (using what it calls Signatures and Modules, which we'll discuss later), and provides tools to define success metrics and automatically optimize prompts towards them. The promise of DSPy is that with a good metric and some ability to calculate it (either ground truth data or an LLM judge), one can automatically optimize the prompt without needing to manually edit a text file. Through this project, we'll see some of the pros and cons of this approach. It should be noted that we only really scratch the surface of DSPy uses here, and the library has excellent documentation for further reading.

The sections are arranged as follows:

- First we discuss the data and objectives. We load the data, do some preprocessing, and sample a few hundred conversations to become the basis for our prompt development sets
- We explore the baseline generator and use it to create outputs for the judge development set. We see how this can be done via direct API calls (i.e. no LLM orchestration framework) vs what it looks like with DSPy.
- Next we discuss creating the gold standard dataset for the LLM judge. To save time, we choose to generate this with a powerful closed-source LLM, but this is the step that in real projects will need human subject matter expert input
- With our gold standard judge dataset in place, we can use it to tune a prompt for the LLM

judge that we'll eventually be using to help us iterate. Along the way we'll touch on how DSPy's optimizers work

- We'll then attempt to use DSPy with our optimized judge to tune our baseline generator prompt and get the best scores possible on the development dataset

The purpose of all this is to take steps towards a robust methodology for prompt development that is reproducible and fairly automated, meaning that it could be run again to re-optimize the generator prompt if we chose to switch model provider, for example. The methodology is far from perfect and in terms of result quality I suspect it is no substitute for careful manual prompt engineering in collaboration with SMEs. However, do I think it encapsulates many of the principles of evaluation driven development and highlights the power of DSPy to speed up iteration in LLM projects.

5.0 Dataset and Objective

The Kaggle customer support dataset that we're using provides us with about 30k customer questions related to air travel. In order to convert this into a more realistic "customer support conversation" table, I used `gemini-2.5-flash` to create a set of 5000 synthetic conversations using samples from the Kaggle dataset to seed the topics. Each conversation contains both customer and support messages, and is associated with a unique id and company name. This generation process is designed to create a reasonably sized example dataset that is similar to what could be constructed from real customer support logs.

Here's an example of a synthetic sample conversation related to American Airlines

```
Customer: Trying to sort out a friend's return flight from Heathrow but no luck
with the usual telephone number. I thought somebody posted another number some
time ago but I've searched and can't find anything.
Support:  The main number is 800-433-7300.  What information do you need
regarding your friend's flight?  A confirmation number would be helpful.
Customer:  I don't have a confirmation number.  It's for a friend; I only have
her name and the approximate travel dates – sometime in October.  Is there
another way to track this down?
Support:  Unfortunately, without a confirmation number or more precise dates,
tracking her flight is impossible.  Have her check her email inbox for a
confirmation.  If she can't find it, she should contact us directly.
```

The goal of our agent will be to assume the role of customer support, responding to the user's query in a way that is as helpful as possible given the situation. This is a challenging problem since customer questions may require highly specific context in order to answer well, or may be completely outside the support agent's control. In such situations the model must do the best it can, showing empathy and understanding just like a trained human agent would. It must also not hallucinate facts, which is a very important check that is beyond the scope of this article but should certainly be made in production, ideally using an LLM judge that has access to suitable knowledge banks.

To get the data ready, we can apply some basic preprocessing, covered in the code here. We take advantage of the parallelism offered by Huggingface's datasets library, which allows us to

apply basic models like a fasttext language detector on this dataset. We also need to randomly truncate the conversations so that the final utterance is always a “Customer” message, therefore setting up the model to generate the next “Support” response. For this purpose we have this simple truncator class. Finally, it seems helpful to provide the model with the company name for additional context (this is something whose accuracy lift we can test with the framework proposed here!, so we append that to the conversations too.

```
from dspy_judge.data_loader.dataset_loader import CustomerSupportDatasetLoader
from dspy_judge.processor.conversation_truncator import ConversationTruncator
from dspy_judge.processor.utils import concat_company_and_conversation

data_loader = CustomerSupportDatasetLoader()
# load and preprocess
dataset = data_loader.load_dataset(split="train")
processed_dataset = data_loader.preprocess_dataset(dataset)

truncator = ConversationTruncator(seed=101)
# truncate the conversations
truncated_dataset = truncator.process_dataset(
    processed_dataset,
    min_turns=1,
    ensure_customer_last=True
)
# apply function that concatenates company name to conversation
truncated_dataset = truncated_dataset.map(concat_company_and_conversation)

# sample just a subset of the full dataset
truncated_loaded_sampled = data_loader.get_sample(
truncated_dataset,n_samples=400,seed=10
)
# split that sample into test and train segments
split_dataset = truncated_loaded_sampled.train_test_split(
test_size=0.4, seed=10
)
```

The development dataset that we use to tune our generator and judge needs to be small enough that tuning is fast and efficient, but still representative of what the model will see in production. Here we just choose a random sample of 400 truncated conversations, which will be split further into testing and training datasets in the coming sections. Smarter methods for choosing a representative sample for prompt optimization are a topic of active research, an interesting example of which is here.

6.0 Baseline generator and judge development set

Let’s split our dataset of representative inputs further: 160 examples for judge development and 240 for generator prompt development. These numbers are arbitrary but reflect a practical compromise between representativeness and time/cost.

To proceed with LLM judge development we need some outputs. Let’s generate some via a

baseline version of our generator, which we'll refine later.

LLMs are typically very good at the customer service task we're currency focused on so in order to see meaningful performance gains in this toy project let's use `gpt-3.5-turbo` as our generator model. One of the most powerful features of DSPy is the ease of switching between models without the need to manually re-optimize prompts, so it would be easy and interesting to swap this out for other models once the system is built.

6.1 A basic version without DSPy

The first version of a baseline generator that I built for this project actually does not use DSPy. It consists of the following basic manually-typed prompt

```
baseline_customer_response_support_system_prompt = "You are a customer service
agent whose job is to provide a single, concise response to a customer query.
You will receive a transcript of the interaction so far, and your job is to
respond to the latest customer message. You'll also be given the name of the
company you work for, which should help you understand the context of the
messages."
```

To call an LLM, the code contains a number of modules inheriting from `LLMCallerBase`, which can also optionally enforce structured output using the instructor library (more about how this works here). There is also a module called ParallelProcessor, which allows us to make API calls in parallel and minimizes errors in the process by using backoff to automatically throttle the calls. There are likely many ways to make these parallel calls - in a previous article I made use of Huggingface datasets `.map()` functionality, whereas here inside ParallelProcessor we directly use python's multiprocessing library and then re-form the dataset from the list of outputs.

This is likely not an efficient approach if you're dealing with really large datasets, but it works well for the few thousand examples that I have tested it with. It's also very important to be aware of API costs when testing and running parallel LLM calls!

Putting these things together, we can generate the baseline results from a sample of the truncated conversation dataset like this

```
from dspy_judge.llm_caller import OpenAITextOutputCaller
from dspy_judge.processor.parallel_processor import ParallelProcessor

baseline_model_name = "gpt-3.5-turbo"
baseline_model = OpenAITextOutputCaller(api_key=secrets["OPENAI_API_KEY"])
baseline_processor = ParallelProcessor(baseline_model, max_workers=4)

#split dataset is divided into the generator and judge development segments
dev_dataset = split_dataset["train"]
judge_dataset = split_dataset["test"]

# company_and_transcript is the name of the field generated by the
concat_company_and_conversation
# function
baseline_results_for_judge = baseline_processor.process_dataset(
```

```
        judge_dataset,
        system_prompt=baseline_customer_response_support_system_prompt,
        model_name=baseline_model_name,
        input_field="company_and_transcript",
        temperature=1.0
    )

# create a full transcript by adding the latest generated response to the end
# of the input truncated conversation
baseline_results = baseline_results.map(concat_latest_response)
```

6.2 How does this look with DSPy?

Using dspy feels different from other LLM orchestration libraries because the prompting component is abstracted away. However our codebase allows us to follow a similar pattern to the “direct API call” approach above.

The core dspy objects are signatures and modules. Signatures are classes that allow us to define the inputs and outputs of each LLM call. So for example our baseline generator signature looks like this

```
import dspy

class SupportTranscriptNextResponse(dspy.Signature):

    transcript: str = dspy.InputField(desc="Input transcript to judge")
    llm_response: str = dspy.OutputField(desc="The support agent's next
utterance")
```

Signatures can also have docstrings, which are essentially the system instructions that are sent to the model. We can therefore make use of the baseline prompt we’ve already written simply by adding the line

```
  SupportTranscriptNextResponse.__doc__ =
baseline_customer_response_support_system_prompt.strip()
```

A DSPy module is also a core building block that implements a prompting strategy for any given signature. Like signatures they are fully customizable, though for this project we will mainly use a signature called `ChainOfThought`, which implements the chain of thought prompting strategy and thus forces the model to generate a reasoning field along with the response specified in the signature.

```
import dspy

support_transcript_generator_module = dspy.ChainOfThought(
    SupportTranscriptNextResponse
)
```

```
ParallelProcessor has been written to support working with dspy too, so the full
code to generate our baseline results with dspy looks like this

# Create DSPy configuration for multiprocessing
dspy_config = {
  "model_name": "openai/gpt-3.5-turbo",
  "api_key": secrets["OPENAI_API_KEY"],
  "temperature": 1
}

generation_processor = ParallelProcessor()

# note judge_dataset is the judge split from Section 5
baseline_results_for_judge = generation_processor.process_dataset_with_dspy(
  judge_dataset,
  input_field="company_and_transcript",
  dspy_module=support_transcript_generator_module,
  dspy_config=dspy_config,
)
```

You can take a look at the `process_dataset_with_dspy` method to see the details of the setup here. To avoid pickling errors, we extract the DSPy signature from the supplied module, deconstruct and then re-assemble it on each of the workers. Each worker then calls

```
dspy_module.predict(transcript=input_text)
```

on each row in the batch it receives and then the result is reassabled. The final result should be similar to that generated with the “native API” method in the previous sections, with the only differences arising from the high temperature setting.

The big advantage of DSPy at this stage is that the module `support_transcript_generator_module` can be easily saved, reloaded and fed directly into other DSPy tools like Evaluate and Optimize, which we’ll see later.

7.0 The judge training dataset

With our baseline generator set up, we can move on to LLM judge development. In our problem, we want the LLM judge to act like a human customer support expert who is able to read an interaction between another agent and a customer, judge the agent’s success at resolving the customer’s issue and also give critiques to explain the reasoning . To do this, it is very helpful to have some gold standard judgments and critiques. In a real project this can be done by running the judge development dataset through the baseline generator and then having a subject matter expert review the inputs and outputs, generating a labeled dataset of results. To keep things simple a binary yes/no judgement is often preferable, enabling us to directly calculate metrics like accuracy, precision and Cohen’s kappa. To speed up this step in this toy project, I used Claude Opus 4.0 along with a “gold standard” judge prompt carefully designed with aid of GPT5. This is powerful, but is no substitute for a human SME and only used because this is a demo project.

Once again we can use DSPy's `ChainOfThought` module with a signature like this

```
import dspy

class SupportTranscriptJudge(dspy.Signature):
    transcript: str = dspy.InputField(desc="Input transcript to judge")
    satisfied: bool = dspy.OutputField(
        desc="Whether the agent satisfied the customer query"
    )
```

Because we're requesting chain of thought, the reasoning field will automatically get generated and doesn't need to specified in the signature

Running our "gold standard judge" to simulate the SME labelling phase looks like this

```
import dspy
from dspy_judge.processor.utils import extract_llm_response_fields_dspy

dspy_config = {
      "model_name": "anthropic/claude-sonnet-4-20250514",
      "api_key": secrets["ANTHROPIC_API_KEY"],
      "temperature": 0
}

gold_standard_judge_generator_module =
dspy.ChainOfThought(SupportTranscriptJudge)

gold_standard_dspy_judge_processor = ParallelProcessor()

dspy_judge_results_optimized =
gold_standard_dspy_judge_processor.process_dataset_with_dspy(
  judge_dataset.select_columns(
    ["conversation_id","output_transcript"]
  ),
  input_field="output_transcript",
  dspy_module=gold_standard_judge_generator_module,
  dspy_config=dspy_config
)
gold_standard_dspy_judge_results = gold_standard_dspy_judge_results.map(
    extract_llm_response_fields_dspy
)
```

The judge training dataset contains a fix of "positive" and "negative" results and their associated explanations. This is desirable because we need to make sure that our LLM judge is tuned to know how to distinguish the two. This also has the advantage of giving us our first indication of the performance of the baseline generator. For our sample dataset, the performance is not great, with an almost 50% failure rate as seen in figure 2. In a more serious project, we would want to pause at this SME labelling stage and conduct a careful error analysis to understand the main types of failure mode.

Figure 2: Performance of the baseline generator according to the gold standard LLM judge, which is run against the judge training dataset and generates a binary classification for each output pair and a short critique. Image generated by the author.

8.0 Optimizing the judge prompt

With our gold standard judge dataset in place, we can now proceed to develop and optimize a judge prompt. One way to do this is start with a baseline judge by trying to encode some of the SME reasoning into a prompt, run it on the judge training set and make incremental edits until the alignment between the SME scores and judges scores starts to level off. It's useful to log each version of the prompt and its associated alignment score so that progress can be tracked and any leveling off detected. Another approach is to start by using dspy's prompt optimizers to do some of this work for us.

Dspy optimizers take our module, a metric function and small training set and attempt to optimize the prompt to score and maximize the metric. In our case, the metric will be the match accuracy between our judge's binary classification and the ground truth from the SME labelled dataset. There are multiple algorithms for this, and here we focus on MIPROv2 because it can adapt both the system instructions and create or edit few-shot examples. In summary, MIPROv2 is an automatic, iterative process with the following steps

- Step 1: It runs the supplied module against a subset of the training dataset and filters high scoring trajectories to generate few-shot examples.
- Step 2: It uses LLM calls to generate multiple candidate system prompts based on observations from step 1
- Step 3: It searches for combinations of candidate system prompts and candidate few shot examples that maximize the metric value when run against mini-batches of the training data

The promise of algorithms like this is that they can help us design prompts in a data-driven fashion similar to how traditional ML models are trained. The downside is that they separate developers from the data in such a way that it becomes more difficult to explain why the selected prompt is optimal, except for "the model said so". They also have numerous hyperparameters whose settings can be quite influential on the result.

In my experience so far, optimizers make sense to use in cases where we already have ground truth, and their output can then be used as a starting point for manual iteration and further collaboration with the subject matter expert.

Let's see how MIPROv2 can be applied in our project to optimize the LLM judge. We'll choose our judge model to be Gemini 1.5 flash, which is cheap and fast.

```
import dspy
from dspy_judge.prompts.dspy_signatures import SupportTranscriptJudge

judge_model = dspy.LM(
    "gemini/gemini-1.5-flash",
    api_key=secrets["GEMINI_API_KEY"],
    cache=False,
```

```
    temperature=0
)
dspy.configure(lm=judge_model,track_usage=True,adapter=dspy.JSONAdapter())
baseline_judge = dspy.ChainOfThought(SupportTranscriptJudge)
```

Note that baseline judge here represents our “best attempt” at a judge based on reviewing the SME labelled dataset.

Out of curiosity, let’s first see what the initial alignment between the baseline and gold standard judge is like. We can do that using `dspy.Evaluate` running on a set of examples, a metric function and a module.

```
import dspy
from dspy_judge.processor.utils import convert_dataset_to_dspy_examples

#this is our simple metric function to determine where the judge score matches
the gold #standard judge label

def match_judge_metric(example, pred, trace=None):
    example_str = str(example.satisfied).lower().strip()
    # this is going to be True or False
    pred_str = str(pred.satisfied).lower().strip()

    if example_str == pred_str:
        return 1
    else:
        return 0

# Load the SME labelled dataset
dspy_gold_standard_judge_results =
data_loader.load_local_dataset("datasets/gold_standard_judge_result")

# Convert HF dataset to list of dspy examples
judge_dataset_examples = convert_dataset_to_dspy_examples(
    dspy_gold_standard_judge_results,
    field_mapping = {"transcript":"output_transcript","satisfied":"satisfied"},
    input_field="transcript"
)

evaluator = dspy.Evaluate(
    metric=match_judge_metric,
    devset=judge_dataset_examples,
    display_table=True,
    display_progress=True,
    num_threads=24,
)
original_score = evaluator(baseline_judge)
```

For me, running this gives a baseline judge score of ~60%. The question is, can we use MIPROv2 to improve this?

Setting up an optimization run is straightforward, though be aware that multiple LLM calls are made during this process and so running multiple times or on large training datasets can be costly. It's also recommended to check the documentation for the hyperparameter explanations and be prepared for the optimization to not work as expected.

```
import dspy

#split the SME labelled dataset into training and testing
training_set = judge_dataset_examples[:110]
validation_set = judge_dataset_examples[110:]

optimizer = dspy.MIPROv2(
    metric=match_judge_metric,
    auto="medium",
    init_temperature=1.0,
    seed=101
)

judge_optimized = optimizer.compile(
    baseline_judge,
    trainset=training_set,
    requires_permission_to_run=False,
)
```

At this stage, we have a new dspy module called judge_optimized, and we can evaluate it with dspy.Evaluator against the training and validation sets. I get an accuracy score of ~70% when I do this, suggesting that the optimization has indeed made the judge prompt more aligned with the gold standard labels.

What specifically has changed? To find out, we can run the following

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

Which will show the latest version of the system prompt, any added few shot examples and the last call that was made. Running the optimization several times will likely produce quite different results with system prompts that range from minor modifications to the baseline to complete rewrites, all of which achieve somewhat similar final scores. Few-shot examples almost always get added from the training set, which suggests that these are the main drivers of any metric improvements. Given that the few-shot examples are sourced from the training set, it's very important to run the final evaluation against both this and the judge validation set to protect against overfitting, though I suspect this is less of a problem than in traditional machine learning.

Finally, we should save the judge and baseline modules for future use and reproduction of any experiments.

```
judge_optimized.save("dspy_modules/optimized_llm_judge",save_program=True)
baseline_judge.save("dspy_modules/baseline_llm_judge",save_program=True)
```

9.0 Using the optimized judge to optimize the generator

Let's say we have an optimized LLM judge that we're confident is suitably aligned with the SME labelled dataset to be useful and we now want to use it to improve the baseline generator prompt. The process is similar to that which we used to build the optimized judge prompt, only this time we're using the judge to provide us with ground truth labels. As mentioned in section 6.0, we have a generator development set of 240 examples. As part of manual prompt iteration, we would each generate a generator prompt version on these input examples, then run the judge on the results and calculate the accuracy. We'd then review the judge critiques, iterate on the prompt, save a new version and then try again. Many iterations will likely be needed, which explains why the development set should be fairly small. DSPy can help get us started down this path with automatic optimization, and the code is very similar to section 8.0, with one notable exception being the optimization metric.

During this phase, we'll be using two LLMs in the optimization: The generator model, which is `gpt-3.5-turbo` and the judge mode, which is `gemini-1.5-flash`. To facilitate this, we can make a simple custom module called `ModuleWithLM` that uses a specific LLM, otherwise it will just use whatever model is defined in the last `dspy.configure()` call

```
import dspy

optimized_judge = dspy.load(
    "dspy_modules/optimized_llm_judge"
)

# our judge LLM will be gemini 1.5
judge_lm = dspy.LM(
    "gemini/gemini-1.5-flash",
    api_key=secrets["GEMINI_API_KEY"],
    cache=False,
    temperature=0
)

# A helper that runs a module with a specific LM context
class ModuleWithLM(dspy.Module):
    def __init__(self, lm, module):
        super().__init__()
        self.lm = lm
        self.module = module

    def forward(self, **kwargs):
        with dspy.context(lm=self.lm):
            return self.module(**kwargs)

# our new module, which allows us the package a separate llm with a previously
# loaded module
optimized_judge_program = ModuleWithLM(judge_lm, optimized_judge)
```

With this established, we can call `optimized_judge_program` inside a metric function, which

serves the same purpose as `match_judge_metric()` did in the judge optimization process.

```
def LLM_judge_metric(example, prediction, trace=None):

    # the input transcript
    transcript_text = str(example.transcript)

    # the output llm response
    output_text = str(prediction.llm_response)

    transcript_text = f"{transcript_text}\nSupport: {output_text}"

    if not transcript_text:
        # Fallback or raise; metric must be deterministic
        return False

    judged = optimized_judge_program(transcript=transcript_text)
    return bool(judged.satisfied)
```

The optimization itself looks similar to the process we used for the LLM judge

```
import dspy
from dspy_judge.processor.utils import convert_dataset_to_dspy_examples

generate_response = dspy.load(
"dspy_modules/baseline_generation",save_program=True
)

#see section 1.3 for the description of dev_dataset
 dev_dataset_examples = convert_dataset_to_dspy_examples(
    dev_dataset,
    field_mapping = {"transcript":"company_and_transcript"},
    input_field="transcript"
)

# A somewhat arbitrary split into test and train
# The split allows us to check for overfitting on the training examples

optimize_training_data =  dev_dataset_examples[:200]
optimize_validation_data = dev_dataset_examples[200:]

optimizer = dspy.MIPROv2(
    metric=LLM_judge_metric,
    auto="medium",
    init_temperature=1.0,
    seed=101
)

generate_response_optimized = optimizer.compile(
    generate_response,
```

```
    trainset=optimize_training_data,
    requires_permission_to_run=False,
)
```

With the optimization complete, we can use dspy.Evaluate to generate an overall accuracy assessment

```
evaluator = dspy.Evaluate(
    metric=LLM_judge_metric,
    devset=optimize_training_data,
    display_table=True,
    display_progress=True,
    num_threads=24,
)
overall_score_baseline = evaluator(generate_response)
latest_score_optimized = evaluator(generate_response_optimized)
```

During testing, I was able to get a fair improvement from ~58% to ~75% accuracy using this method. On our small dataset some of this gain is attributable to just a handful to judge results switching from “unsatisfied” to “satisfied”, and for changes like this it is worth checking if they are within the natural variability of the judge when run multiple times on the same dataset. In my view, the generator prompt optimization’s greatest strength is that it provides us with a new baseline prompt which is grounded in best practice and more defensible than the original baseline, which is typically just the developer’s best guess at a good prompt. The stage is then set for more manual iteration, use of the optimized LLM judge for feedback and further consultation with SMEs for error analysis and edge case handling.

Figure 3: Performance comparison between the baseline and optimized generator prompts, as measured using the optimized judge. The performance lift appears to come mainly from the DSPy optimizer adding few-shot examples. Image generated by the author.

At this stage it is sensible to look in detail at the data to learn about where these modest quality gains are coming from. We can join the output datasets from the baseline and optimized generator runs on conversation id, and look for instances where the judge’s classification changed. In cases where the classification switched from `satisfied=False` to `satisfied=True` my overall sense is that the generator’s responses got longer and more polite, but didn’t really convey any more information. This is not surprising, since for most customer support questions the generator doesn’t have enough context to add meaningful detail. Also, since LLM judges are known to show bias towards longer outputs, the optimization process appears to have pushed the generator in that direction.

10.0 Important learnings

This article has explored prompt optimization with DSPy for both LLM judge and generator refinement. This process is somewhat long-winded, but it enforces good practice in terms of development dataset curation, prompt logging and checking for overfitting. Since the LLM judge alignment process typically requires a human in the loop to generate labelled data, the DSPy

optimization process can be especially powerful here because it reduces expensive iterations with subject matter experts. Although not really discussed in this article, it's important to note that small performance gains from the optimization might not be statistically significant due to the natural variability of LLMs.

With the judge set up, it can also be used by DSPy to optimize the generator prompt (therefore bypassing the need for more human labelling). My sense is that this may also be worth pursuing in some projects, if for no other reason but the automatic curation of good few-shot examples. But it should not be a substitute for manual evaluation and error analysis. Optimization is can also be quite costly in terms of tokens, so care should be taken to avoid high API bills!

Thank you for making it to the end. As always feedback is appreciated and if you have tried a similar framework for prompt engineering I would be very interested to learn about your experience! I'm also curious about the pros and cons of extending this framework to more complex systems that use ReACT or prompt-chaining to accomplish larger tasks.

Written By

Share This Article

- Share on Facebook
- Share on LinkedIn
- Share on X

Towards Data Science is a community publication. Submit your insights to reach our global audience and earn through the TDS Author Payment Program.

Other mentions by Author

- towardsdatascience.com | Related Articles
- towardsdatascience.com | Making Sense of the Promise (and Risks) of Large Language Models
- towardsdatascience.com | Deep Dive into Sora's Diffusion Transformer (DiT) by Hand ✍️
- towardsdatascience.com | Beware of Unreliable Data in Model Evaluation: A LLM Prompt Selection case study with Flan-T5
- towardsdatascience.com | Beating ChatGPT 4 in Chess with a Hybrid AI model
- towardsdatascience.com | Semantic Search Engine for Emojis in 50+ Languages Using AI 😊🌍🚀
- towardsdatascience.com | Deep Dive into Anthropic's Sparse Autoencoders by Hand ✍️