Open in app ↗

Search

Tuning for Geometry

Fine-tuning Large Language Model for generating Midcurve

Yogesh Haribhau Kulkarni (PhD)
Published in Analytics Vidhya
11 min read · Just now

Listen Share More

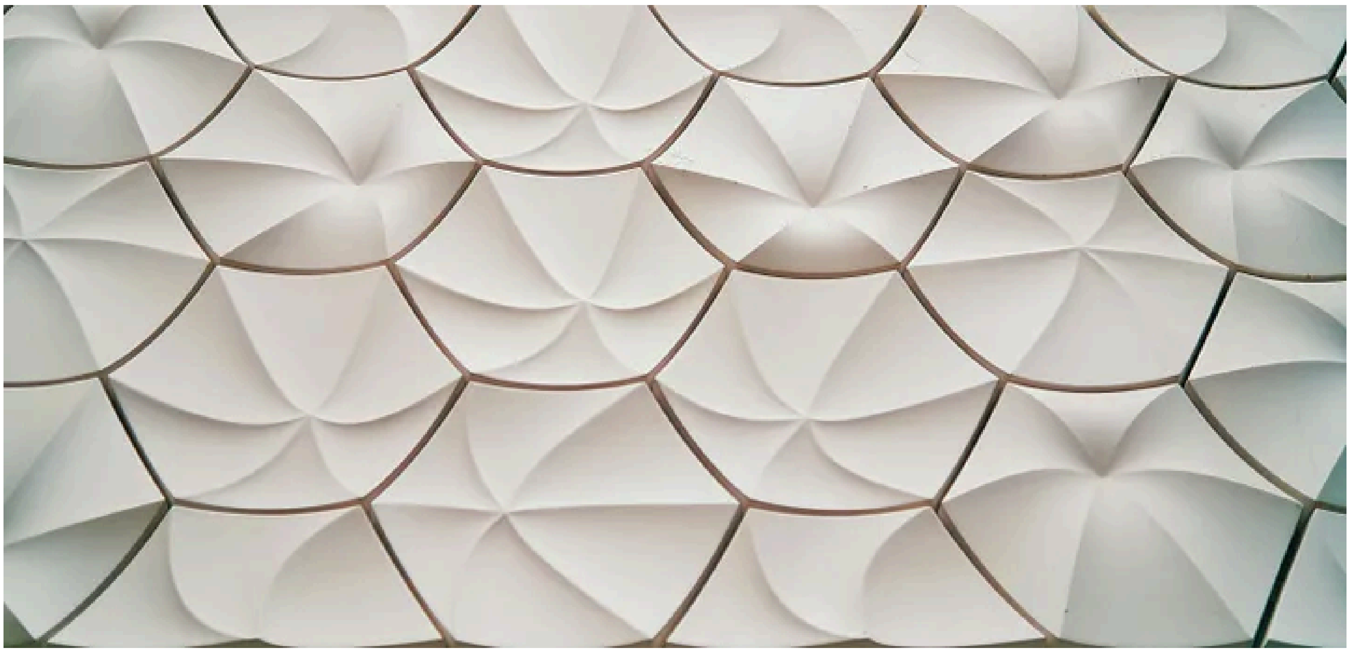
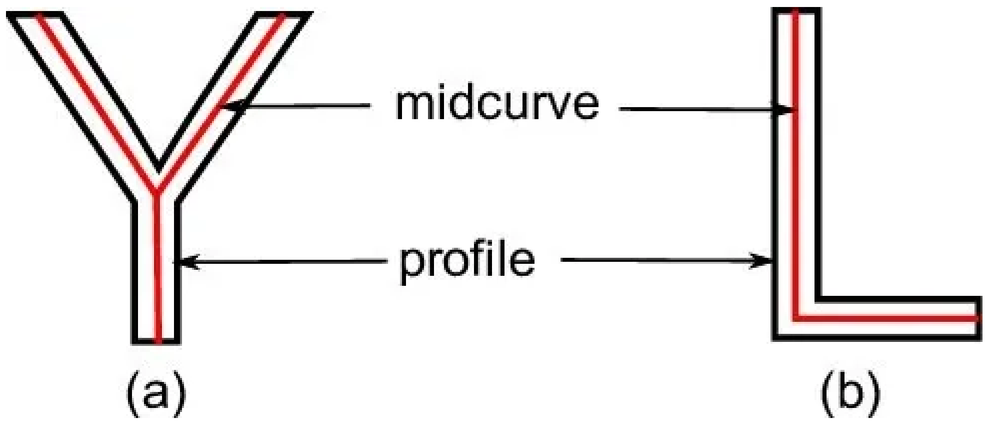


Photo by Leiada Krozjhen on Unsplash

Large Language Models (LLMs) are commonly utilized for a variety of creative endeavors, ranging from crafting stories and poems to handling more mundane tasks such as drafting proposals and emails. Additionally, they are capable of tackling more complex tasks like coding, given their text-based nature. While multi-modal LLMs have shown promise in handling diverse media types including images, sounds, and videos, they have yet to explore multidimensional fields such as geometry, particularly shapes. Neural networks have made strides in geometric modeling, albeit in a limited capacity, as the inherently non-vectorizable nature of geometry presents semantic challenges. One potential avenue for bridging this gap is by translating geometric data into language, thereby rendering it accessible for LLMs to process. This article delves into the process of fine-tuning a Large Language Model (LLM) using geometric data, specifically focusing on 2D shapes, with the aim of generating Midcurve.

Midcurve

The midcurve of a 2D geometric profile refers to a curve positioned equidistantly between the bounding curves of the profile. This curve essentially encapsulates the "center" of the profile, providing a simplified representation of its shape while preserving crucial geometric details. By leveraging the midcurve, one can effectively convey the essence of the profile in a more straightforward manner compared to the original rendition, without compromising on the integrity of its geometric characteristics.



Examples of Midcurves of 2D Geometric profiles (ref)

In the picture about outer shapes (in 'black') are profiles and their corresponding midcurves are shown in 'red'. The problem at hand is that given a profile, how to compute its midcurve. For further insights into Midcurve generation, you can explore additional resources available on Medium. Check out the following articles:

1. "Geometry, Graphs, and GPT" offers valuable information about the intersection of geometry and language models. Find it here.

2. Curious about ChatGPT's comprehension of geometry? Dive into the discussion in "Does ChatGPT Understand Geometry?" Access the article here.

3. Explore "MidcurveNN" for a detailed exploration of Midcurve generation techniques. This article delves into the nuances of leveraging neural networks for this purpose. Read it here.

Fine-tuning

Given the absence of readily available Large Language Models (LLMs) capable of generating midcurves, coupled with the unsuitability of RAG (Retrieval Augmented Generation) due to the lack of vector embeddings for geometry descriptions, fine-tuning emerges as a viable solution. This process involves refining an LLM using a corpus of textual descriptions corresponding to profiles and their respective midcurves.

Fine-tuning offers a strategic advantage by affording greater control, adaptability, and optimization tailored to the specific task and dataset. This approach ultimately leads to improved performance in generating midcurves. For further details on fine-tuning transformers with custom datasets, refer to the article available at this link.

Base LLM chosen for fine-tuning is Code T5.

CodeT5

T5, introduced in the paper "Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer" represents a significant advancement in natural language processing. The authors detailed their approach to pre-training this text-to-text Transformer model on a vast text corpus, yielding state-of-the-art results across various NLP tasks following fine-tuning.

In practical applications, T5-based models such as CodeT5 and CodeT5+ offer promising potential as AI-powered coding assistants, fostering increased productivity among software developers. At Salesforce, an AI coding assistant demo was developed utilizing CodeT5 as a Visual Studio Code plugin, offering three key functionalities:

1. Text-to-code generation: Creating code snippets based on natural language descriptions.
2. Code autocompletion: Automatically completing code functions given the target function name.
3. Code summarization: Generating natural language summaries of code functions.

This deployment underscores the versatility and utility of T5-based models in facilitating various coding tasks, demonstrating their efficacy as valuable aids in software development workflows.

Given that the descriptions of both profiles and midcurves are structured in JSON format, akin to code, CodeT5 was chosen as the foundational model for fine-tuning. This decision stems from the model's inherent capacity to comprehend and process structured data, aligning well with the JSON format typically used to represent such information. Leveraging CodeT5 as the base model offers a promising starting point for fine-tuning efforts aimed at generating midcurves from profile descriptions, facilitating more effective and accurate results.

In addition to CodeT5, several other libraries were instrumental in constructing this demo:

1. Transformers library by Hugging Face: This library, documented here, offers a comprehensive suite of tools and pre-trained models for natural language processing tasks. Leveraging its functionalities, including model loading, fine-tuning, and inference, streamlined the implementation of the CodeT5-based coding assistant.

2. PyTorch / PyTorch Lightning: PyTorch, along with the PyTorch Lightning framework, provided the foundational infrastructure for training and deploying machine learning models. The documentation for PyTorch Lightning, available here, offers guidance on various aspects of model training and experimentation, ensuring robustness and scalability in the development process.

Code Walk-through

To install the required libraries, you can execute the following commands:

```
!pip install -q transformers datasets
!pip install -q pytorch-lightning wandb
```

These commands will install the necessary dependencies:

1. HuggingFace Transformers library for utilizing the CodeT5 model.
2. HuggingFace Datasets library for dataset loading and preprocessing.
3. PyTorch Lightning for training the model.
4. Weights and Biases for logging training metrics.

Make sure to run these commands in your Python environment before proceeding with the implementation.

```
!wget https://raw.githubusercontent.com/yogeshhk/MidcurveNN/master/src/codeT5/c
```

After running this command, the CSV file will be downloaded and stored in a folder named "data". Once the download is complete, you can comment out this cell to prevent re-downloading the file in subsequent executions.

For preprocessing the data, you can use the following code:

```
!pip install wget
import wget

# Replace the URL with the raw URL of the file on GitHub
url_midcurve_llm = "https://raw.githubusercontent.com/yogeshhk/MidcurveNN/maste
url_midcurve_llm_test = "https://raw.githubusercontent.com/yogeshhk/MidcurveNN/
url_midcurve_llm_train = "https://raw.githubusercontent.com/yogeshhk/MidcurveNN
url_midcurve_llm_val = "https://raw.githubusercontent.com/yogeshhk/MidcurveNN/m

# Download the files
wget.download(url_midcurve_llm, 'midcurve_llm.csv')
wget.download(url_midcurve_llm_test, 'midcurve_llm_test.csv')
wget.download(url_midcurve_llm_train, 'midcurve_llm_train.csv')
wget.download(url_midcurve_llm_val, 'midcurve_llm_val.csv')
```

These commands will download the necessary CSV files for preprocessing. Once downloaded, you can proceed with loading and processing the dataset.

The code snippet below loads the dataset from CSV files containing profiles and their corresponding midcurves in BREP format. Here's a brief explanation:

```
from datasets import load_dataset, Dataset, DatasetDict

base_url = "./"
dataset = load_dataset("csv", data_files={"train": base_url + "midcurve_llm_tra
                                          "test": base_url + "midcurve_llm_test
```

```
                                        "validation": base_url + "midcurve_ll

print(dataset)
```

This code snippet utilizes the `load_dataset` function from the Hugging Face Datasets library to load the dataset from CSV files. The dataset consists of three splits: training, testing, and validation. Each split contains profiles (`Profile_brep`) and their corresponding midcurves (`Midcurve_brep`) in BREP format.

Each row in the dataset corresponds to a pair of a 2D profile in BREP format (`Profile_brep`) and its corresponding 1D midcurve in BREP format (`Midcurve_brep`). This dataset mirrors real-world profiles and midcurves, making it suitable for training and evaluating models for midcurve generation tasks.

Let’s look at one particular example:

```
example = dataset['train'][0]

print("Profile_brep:", example["Profile_brep"])
print("Midcurve_brep:", example["Midcurve_brep"])

Profile_brep: "{\"Points\": [[-3.21, 6.3], [-1.67, 11.06], [-15.93, 15.69], [-1
Midcurve_brep: "{\"Points\": [[-2.44, 8.68], [-16.7, 13.31]], \"Lines\": [[0, 1
```

Below, we define a preprocessing function, which we can apply on the entire dataset.

```
from transformers import RobertaTokenizer

tokenizer = RobertaTokenizer.from_pretrained("Salesforce/codet5-small")

prefix = "Skeletonize the Profile: "
max_input_length = 256
max_target_length = 128

def preprocess_examples(examples):
  # encode the code-docstring pairs
  profiles = examples['Profile_brep']
```

```
    midcurves = examples['Midcurve_brep']

    inputs = [prefix + profile for profile in profiles]
    model_inputs = tokenizer(inputs, max_length=max_input_length, padding="max_le

    # encode the summaries
    labels = tokenizer(midcurves, max_length=max_target_length, padding="max_leng

    # important: we need to replace the index of the padding tokens by -100
    # such that they are not taken into account by the CrossEntropyLoss
    labels_with_ignore_index = []
    for labels_example in labels:
      labels_example = [label if label != 0 else -100 for label in labels_example
      labels_with_ignore_index.append(labels_example)

    model_inputs["labels"] = labels_with_ignore_index

    return model_inputs
```

Now that we have defined the function, let's call .map() on the HuggingFace Dataset object, which allows us to apply this function in batches — hence super fast.

```
dataset = dataset.map(preprocess_examples, batched=True)

 DatasetDict({
    train: Dataset({
        features: ['ShapeName', 'Profile', 'Midcurve', 'Profile_brep', 'Midcurv
        num_rows: 793
    })
    test: Dataset({
        features: ['ShapeName', 'Profile', 'Midcurve', 'Profile_brep', 'Midcurv
        num_rows: 99
    })
    validation: Dataset({
        features: ['ShapeName', 'Profile', 'Midcurve', 'Profile_brep', 'Midcurv
        num_rows: 100
    })
})
```

Next, let's set the format to "torch" and create PyTorch dataloaders.

```
from torch.utils.data import DataLoader

dataset.set_format(type="torch", columns=['input_ids', 'attention_mask', 'label
train_dataloader = DataLoader(dataset['train'], shuffle=True, batch_size=8)
valid_dataloader = DataLoader(dataset['validation'], batch_size=4)
test_dataloader = DataLoader(dataset['test'], batch_size=4)
```

Let's verify an example, by decoding it back into text:

```
tokenizer.decode(batch['input_ids'][0])
<s>Skeletonize the Profile: "{\"Points\": [[-4.16, 5.72], [-3.37, 10.66], [-18.

labels = batch['labels'][0]
tokenizer.decode([label for label in labels if label != -100])

<s>"{\"Points\": [[-3.77, 8.19], [-18.58, 10.54]], \"Lines\": [[0, 1]], \"Segme
```

To fine-tune the model using PyTorch Lightning, we need to define a LightningModule. This module will include the forward pass, training_step (and optionally validation_step and test_step), and the corresponding dataloaders. PyTorch Lightning automates many aspects of the training process, such as device placement and support for various loggers and callbacks.

Here's a basic outline of how to define a LightningModule for fine-tuning:

```
from transformers import T5ForConditionalGeneration, AdamW, get_linear_schedule
import pytorch_lightning as pl

class CodeT5(pl.LightningModule):
    def __init__(self, lr=5e-5, num_train_epochs=100, warmup_steps=1000):
        super().__init__()
        self.model_name = "Salesforce/codet5-small"
        self.model = T5ForConditionalGeneration.from_pretrained(self.model_name
        self.save_hyperparameters()

    def forward(self, input_ids, attention_mask, labels=None):
        outputs = self.model(input_ids=input_ids, attention_mask=attention_mask
        return outputs
    def common_step(self, batch, batch_idx):
        outputs = self(**batch)
        loss = outputs.loss

        return loss

    def training_step(self, batch, batch_idx):
        loss = self.common_step(batch, batch_idx)
        # logs metrics for each training_step,
        # and the average across the epoch
        self.log("training_loss", loss)

        return loss

    def validation_step(self, batch, batch_idx):
        loss = self.common_step(batch, batch_idx)
        self.log("validation_loss", loss, on_epoch=True)

        return loss

    def test_step(self, batch, batch_idx):
        loss = self.common_step(batch, batch_idx)

        return loss

    def configure_optimizers(self):
        # create optimizer
        optimizer = AdamW(self.parameters(), lr=self.hparams.lr)
        # create learning rate scheduler
        num_train_optimization_steps = self.hparams.num_train_epochs * len(trai
        lr_scheduler = {'scheduler': get_linear_schedule_with_warmup(optimizer,
                                                    num_warmup_steps=self.hpara
                                                    num_training_steps=num_trai
                        'name': 'learning_rate',
                        'interval':'step',
                        'frequency': 1}

        return {"optimizer": optimizer, "lr_scheduler": lr_scheduler}

    def train_dataloader(self):
        return train_dataloader

    def val_dataloader(self):
        return valid_dataloader

    def test_dataloader(self):
        return test_dataloader
```

In this LightningModule:

- We define the model architecture (T5ForConditionalGeneration) and tokenizer (T5Tokenizer) in the constructor.
- We define the forward method to perform the forward pass.
- We define training_step, validation_step, and test_step to compute the loss during training, validation, and testing.
- We configure the optimizer using configure_optimizers method.

Once the LightningModule is defined, we can initialize it and train the model using PyTorch Lightning's Trainer class.

Let's start up Weights and Biases! This will initialize W&B and allow you to log various metrics and visualizations during the training process. You can then use W&B to track the training progress, monitor model performance, and visualize results.

```
import wandb

wandb.login()
```

Next, we initialize the model and can now simply start training on GPUs.

```
from pytorch_lightning import Trainer
from pytorch_lightning.loggers import WandbLogger
from pytorch_lightning.callbacks import EarlyStopping, LearningRateMonitor

wandb_logger = WandbLogger(name='codet5-finetune-midcurve', project='CodeT5')
# for early stopping, see https://pytorch-lightning.readthedocs.io/en/1.0.0/ear
early_stop_callback = EarlyStopping(
    monitor='validation_loss',
    patience=3,
    strict=False,
    verbose=False,
    mode='min'
)
lr_monitor = LearningRateMonitor(logging_interval='step')

trainer = Trainer(max_epochs=100,accelerator="auto", #gpus=1,
                  default_root_dir="./Checkpoints",
                  logger=wandb_logger,
                        callbacks=[early_stop_callback, lr_monitor])
trainer.fit(model)
```

Once we're done training, we can also save the HuggingFace model as follows:

```
save_directory = "./Checkpoints" # save in the current working directory, you c
model.model.save_pretrained(save_directory)
```

To perform inference on the trained model, you can follow these steps:

1. Load the Test Dataset: Load the test dataset using the `load_dataset` function provided by the Hugging Face Datasets library.
2. Load the Trained Model: Load your trained model using `T5ForConditionalGeneration.from_pretrained`.
3. Prepare the Example: Prepare the example by tokenizing the input using the tokenizer and converting it to input_ids.
4. Generate Midcurve: Generate the midcurve using the model's `generate` method.

Here's the code to achieve this:

```
from transformers import T5ForConditionalGeneration, T5Tokenizer

# Load the test dataset
dataset_infr = load_dataset("csv", data_files={"train": base_url + "midcurve_ll
                                          "test": base_url + "midcurve_llm_test
                                          "validation": base_url + "midcurve_ll

# Load the trained model
model = T5ForConditionalGeneration.from_pretrained(save_directory)

# Get an example from the test set
test_example = dataset_infr['test'][2]
print("Profile_brep:", test_example['Profile_brep'])

Profile_brep: "{\"Points\": [[-6.96, 1.23], [-9.83, 5.32], [-22.12, -3.28], [-1
```

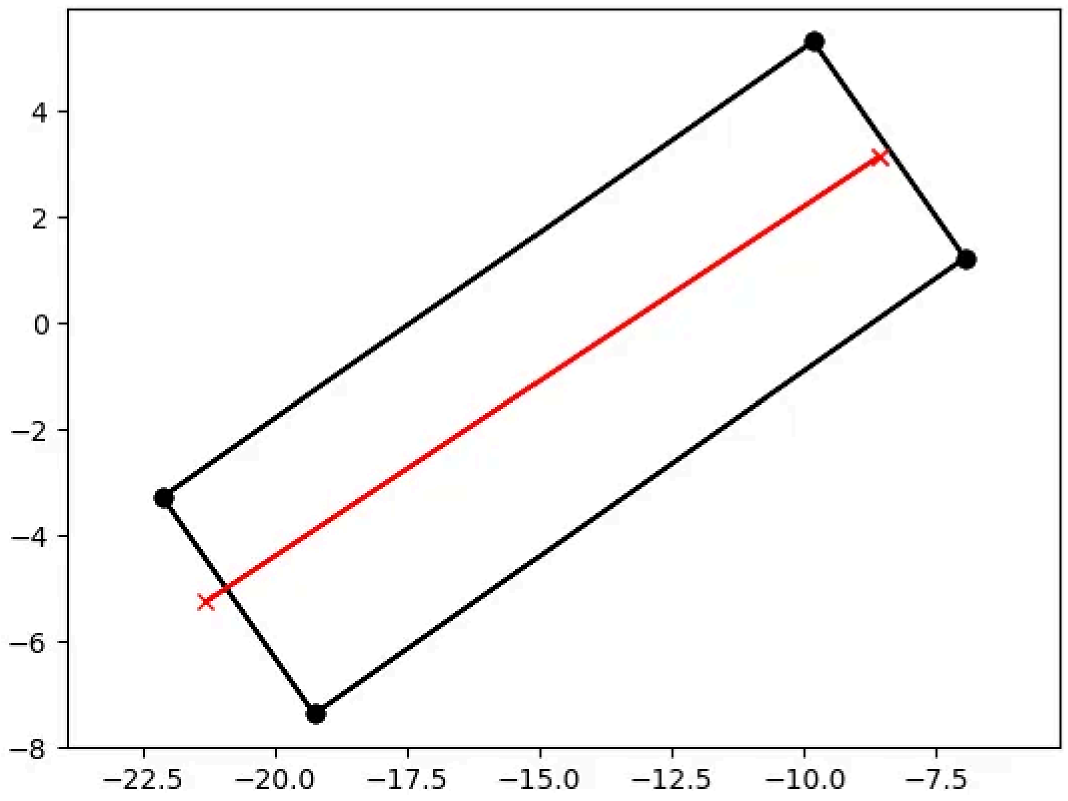
```
# Prepare the example for the model
tokenizer = T5Tokenizer.from_pretrained(save_directory)
input_ids = tokenizer(test_example['Profile_brep'], return_tensors='pt').input_

# Generate Midcurve
outputs = model.generate(input_ids, max_length=200)
Midcurve_brep_str_Predicted = tokenizer.decode(outputs[0], skip_special_tokens=
print("Generated Midcurve:", Midcurve_brep_str_Predicted)

Generated Midcurve: [[-8.59, 3.14], [-21.3, -5.26]], \"Lines\": [[0, 1]], \"Seg
```

One issue seen here is that the first value has no key (not very accurate prediction) so added it manually. But later as the model improves or training improves, this post processing may not be needed. To rectify this, you manually add the key to the generated midcurve. While this workaround may be necessary for current predictions, it might become unnecessary as the model improves through further training.

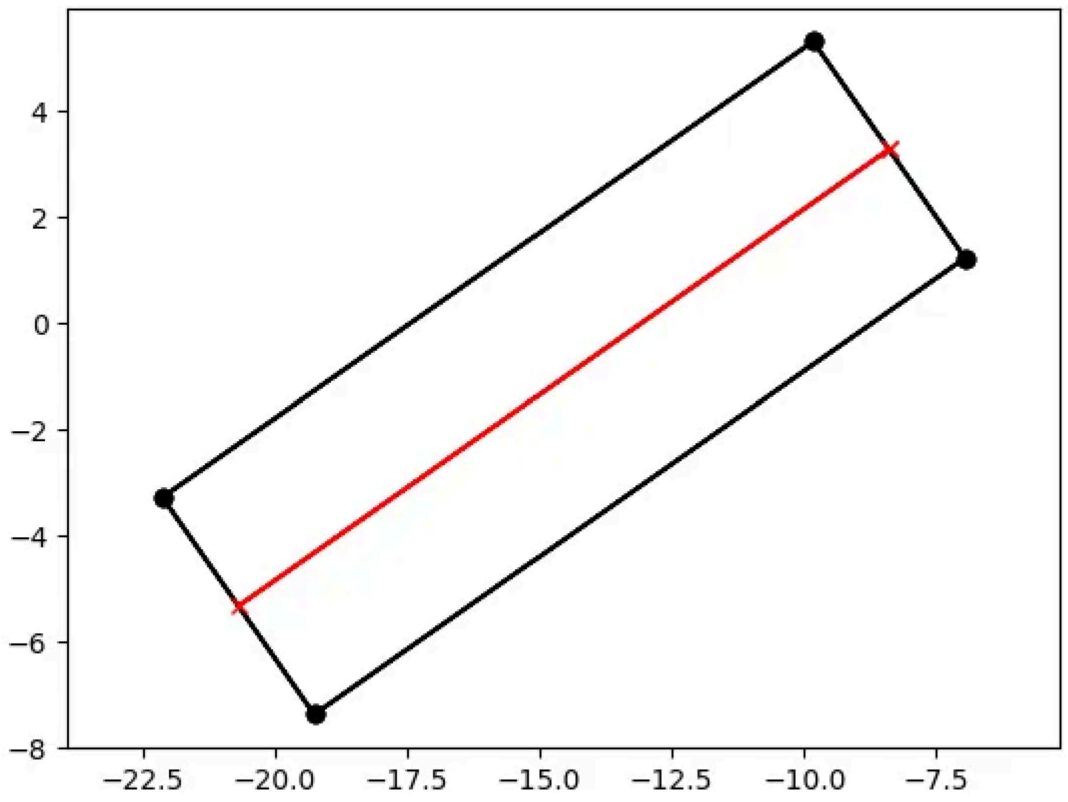
Here’s how you can incorporate this post-processing step into your code:

```
import copy

# Deep copy the original example
test_example_predicted = copy.deepcopy(test_example)

# Add the missing key to the generated midcurve
Midcurve_brep_str_Predicted_with_key = '"{\\"Points\\": ' + Midcurve_brep_str_P

# Replace the midcurve in the predicted example
test_example_predicted['Midcurve_brep'] = Midcurve_brep_str_Predicted_with_key

# Print the predicted example
print("Predicted Example:", test_example_predicted)

# Compare with the ground truth
print("Ground Truth:", test_example['Midcurve_brep'])
```

Visualizing both the midcurves



Predicted Profile and Midcurve



Ground truth Profile and Midcurve

It can be clearly seen that the prediction and ground truth are close, but not perfect. To improve the model's performance:

1. More Training Data: Increasing the size and diversity of the training dataset can help the model learn better representations and improve its generalization capabilities.

2. More Epochs: Training the model for more epochs allows it to see the data multiple times, which can lead to better convergence and improved performance.

3. Better LLM (Large Language Model): Utilizing more advanced and sophisticated pre-trained LLMs, or fine-tuning existing ones with more specialized data, can enhance the model's ability to understand and generate accurate midcurves.

By implementing these strategies and fine-tuning the training process, you can expect to see improvements in the model's accuracy and its ability to generate midcurves that closely resemble the ground truth. Additionally, monitoring the model's performance during training and experimenting with different hyperparameters and architectures can further aid in enhancing its effectiveness.

Conclusion

The process of generating midcurves from 2D profiles using large language models presents a promising avenue for automating geometric modeling tasks. While our initial exploration demonstrates encouraging results, there is clear room for improvement through strategies such as leveraging larger datasets, increasing training epochs, and employing more advanced LLM architectures. By addressing these areas, we can expect to enhance the accuracy and reliability of midcurve generation, paving the way for more efficient and precise geometric modeling workflows in various domains.

References

Transformers-Tutorials/T5/Fine_tune_CodeT5_for_generating_docstrings_fro…

This repository contains demos I made with the Transformers library by HuggingFace. …

github.com

GitHub - salesforce/CodeT5: Home of CodeT5: Open Code LLMs for Code Understanding and Generation

Home of CodeT5: Open Code LLMs for Code Understanding and Generation - salesforce/CodeT5

github.com

Click pic below or visit LinkedIn to know more about the author

Generative Ai Tools Artificial Intelligence Geometry Finetune Llm Future

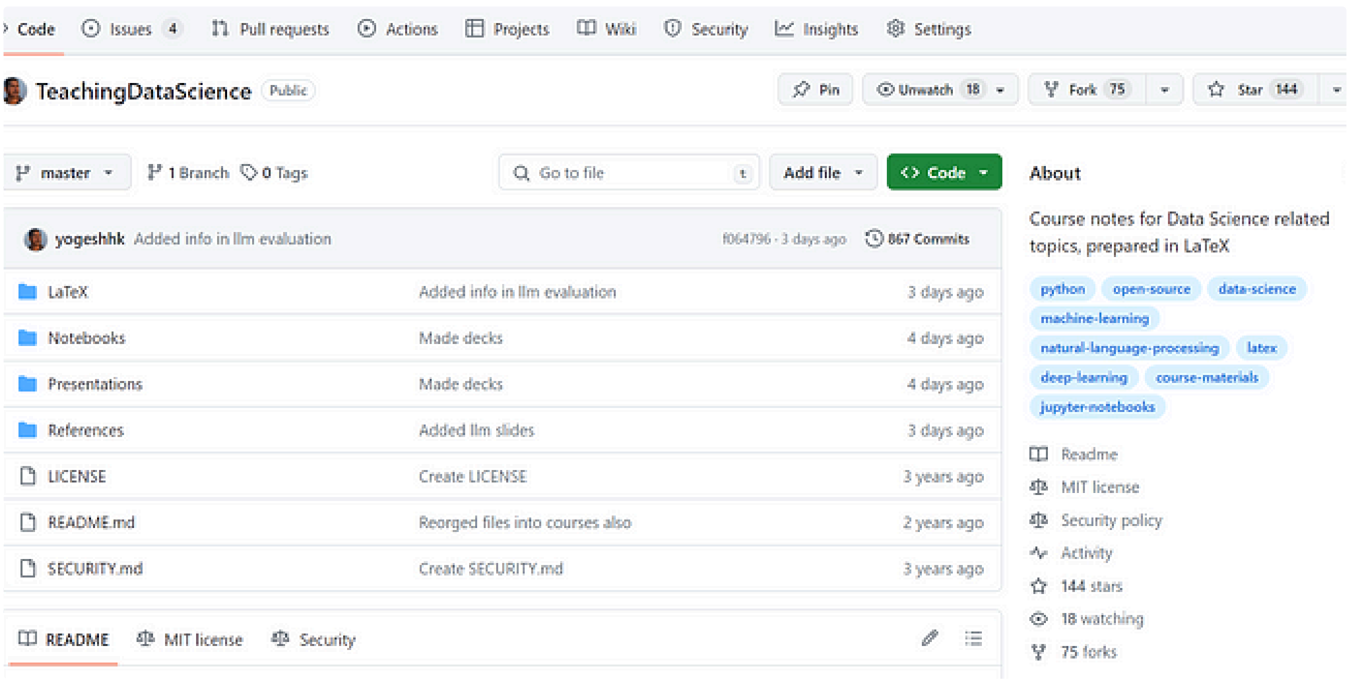
Edit profile

Written by Yogesh Haribhau Kulkarni (PhD)

1.3K Followers · Writer for Analytics Vidhya

PhD in Geometric Modeling | Google Developer Expert (Machine Learning) | Top Writer 3x (Medium) | More at https://www.linkedin.com/in/yogeshkulkarni/

More from Yogesh Haribhau Kulkarni (PhD) and Analytics Vidhya



yogeshhk Added info in llm evaluation f064796 · 3 days ago 867 Commits

LaTeX	Added info in llm evaluation	3 days ago
Notebooks	Made decks	4 days ago
Presentations	Made decks	4 days ago
References	Added llm slides	3 days ago
LICENSE	Create LICENSE	3 years ago
README.md	Reorged files into courses also	2 years ago
SECURITY.md	Create SECURITY.md	3 years ago

About

Course notes for Data Science related topics, prepared in LaTeX

python open-source data-science machine-learning natural-language-processing latex deep-learning course-materials jupyter-notebooks

Readme
MIT license
Security policy
Activity
144 stars
18 watching
75 forks

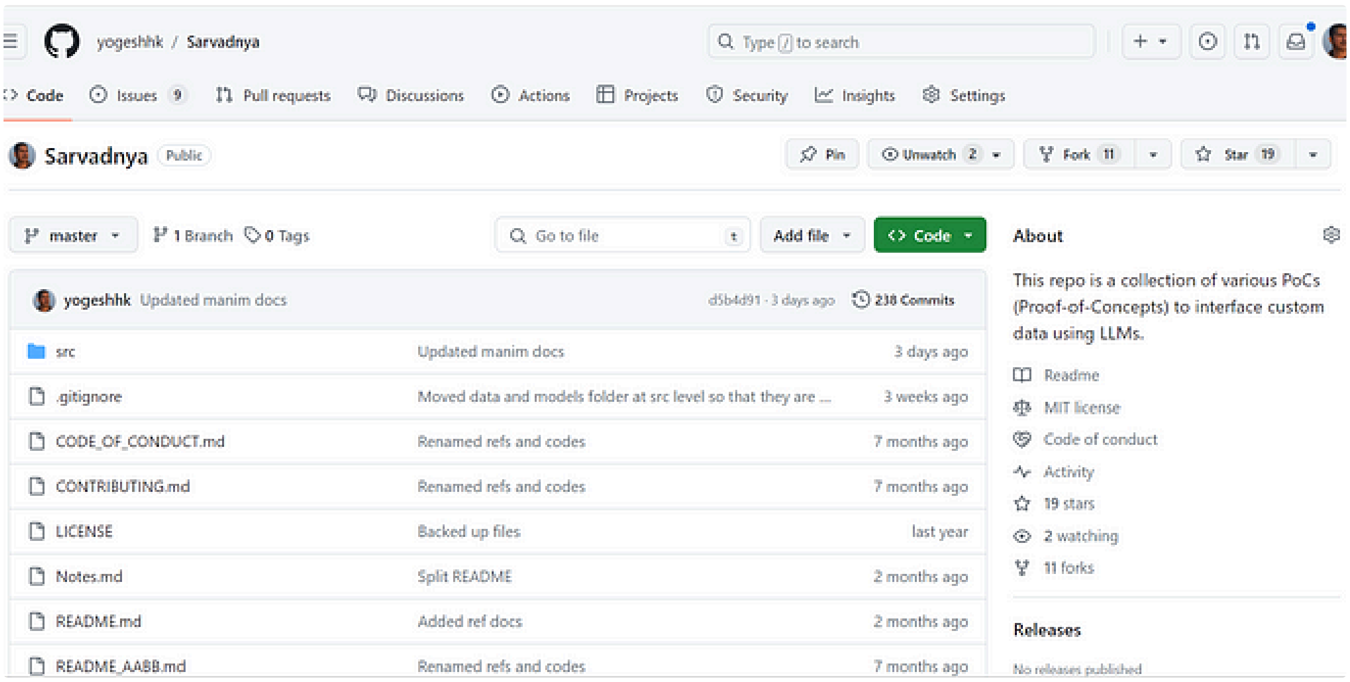
Yogesh Haribhau Kulkarni (PhD) in Technology Hits

Teaching Data Science

An open-source repository for teaching material, open-free to all

4 min read · Mar 4, 2024

166


yogeshhk / Sarvadnya

Code · Issues 9 · Pull requests · Discussions · Actions · Projects · Security · Insights · Settings

Sarvadnya Public

Pin · Unwatch 2 · Fork 11 · Star 19

master · 1 Branch · 0 Tags · Go to file · Add file · Code

yogeshhk Updated manim docs d5b4d91 · 3 days ago 238 Commits

src	Updated manim docs	3 days ago
.gitignore	Moved data and models folder at src level so that they are ...	3 weeks ago
CODE_OF_CONDUCT.md	Renamed refs and codes	7 months ago
CONTRIBUTING.md	Renamed refs and codes	7 months ago
LICENSE	Backed up files	last year
Notes.md	Split README	2 months ago
README.md	Added ref docs	2 months ago
README_AABB.md	Renamed refs and codes	7 months ago

About

This repo is a collection of various PoCs (Proof-of-Concepts) to interface custom data using LLMs.

Readme
MIT license
Code of conduct
Activity
19 stars
2 watching
11 forks

Releases

No releases published

Yogesh Haribhau Kulkarni (PhD) in Technology Hits

Sarvadnya

An open-source repository for All-Knowing Custom Chatbot

3 min read · Mar 4, 2024

9

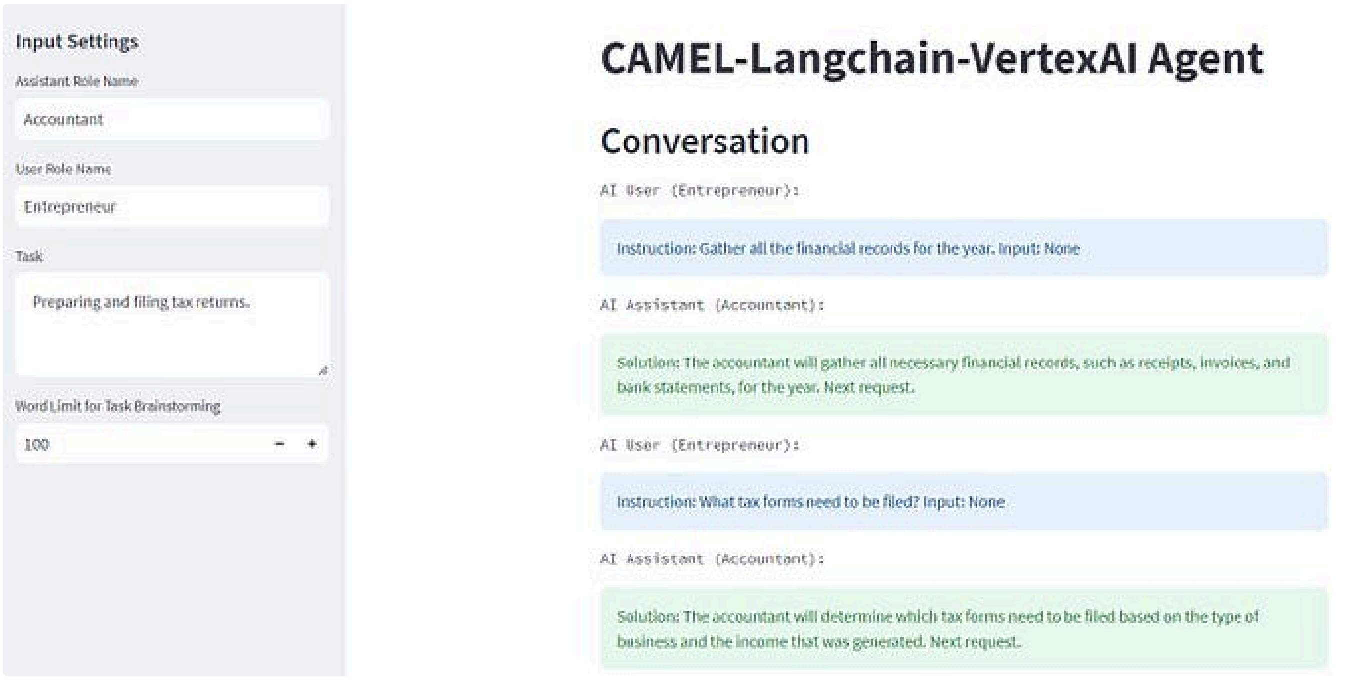
Yogesh Haribhau Kulkarni (PhD) in Technology Hits

Unveiling Manifold learning

What a neural network is really doing?

5 min read · Feb 23, 2024

275 1


Input Settings

Assistant Role Name: Accountant

User Role Name: Entrepreneur

Task: Preparing and filing tax returns.

Word Limit for Task Brainstorming: 100

CAMEL-Langchain-VertexAI Agent

Conversation

AI User (Entrepreneur):

Instruction: Gather all the financial records for the year. Input: None

AI Assistant (Accountant):

Solution: The accountant will gather all necessary financial records, such as receipts, invoices, and bank statements, for the year. Next request.

AI User (Entrepreneur):

Instruction: What tax forms need to be filed? Input: None

AI Assistant (Accountant):

Solution: The accountant will determine which tax forms need to be filed based on the type of business and the income that was generated. Next request.

Yogesh Haribhau Kulkarni (PhD) in Google Developer Experts

A CAMEL ride

A Story of AI Role-Playing using CAMEL, Langchain and VertexAI

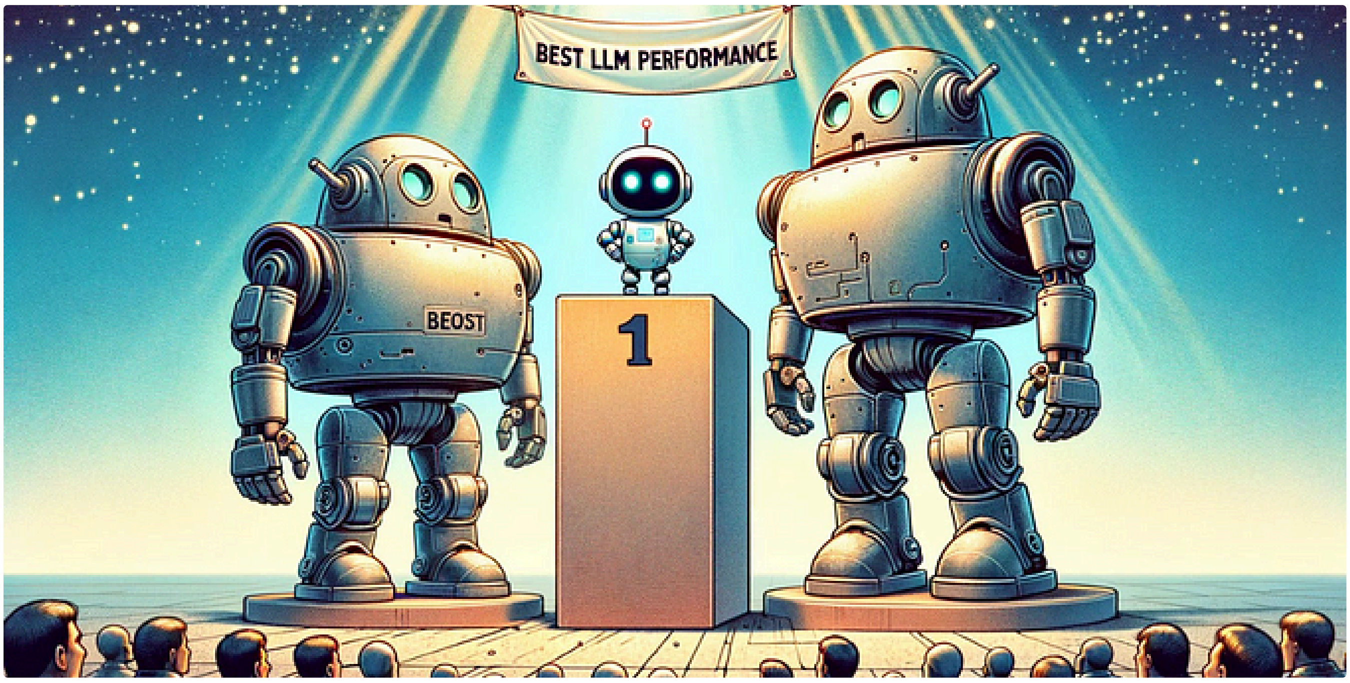
8 min read · Oct 2, 2023

54

See all from Yogesh Haribhau Kulkarni (PhD)

See all from Analytics Vidhya

Recommended from Medium



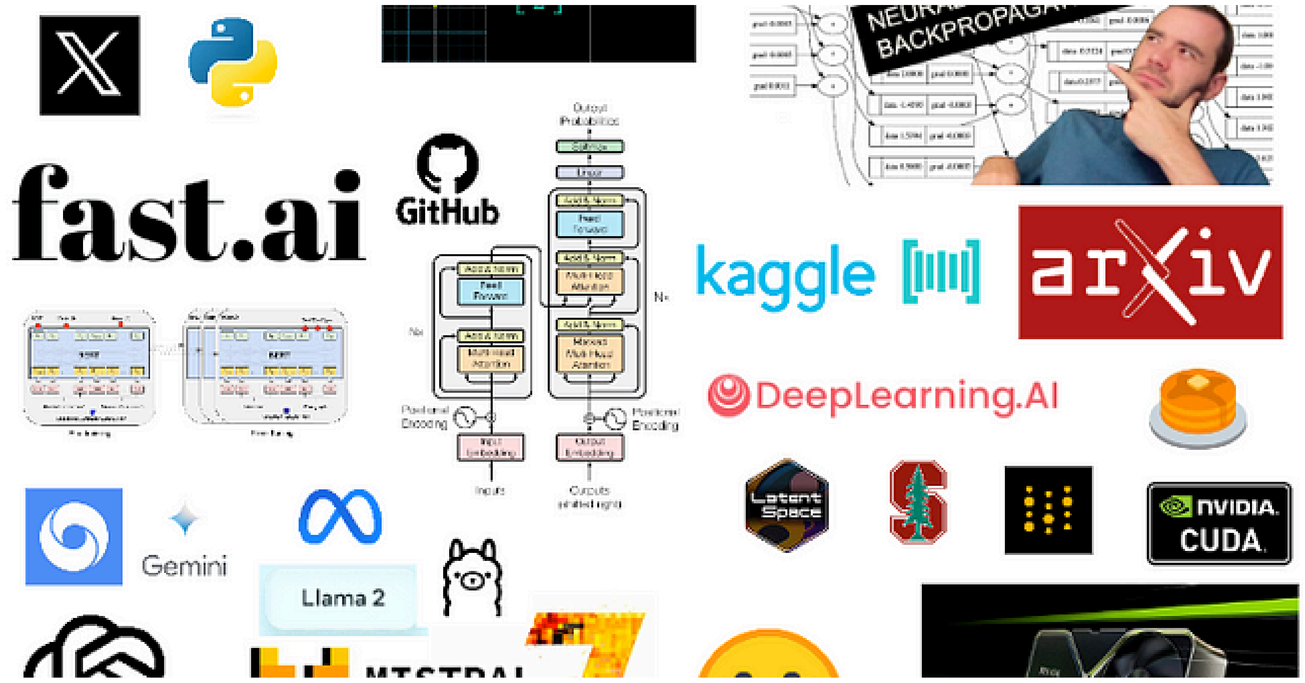
Ignacio de Gregorio

Microsoft Opens The Era of 1-Bit LLMs

16 Times Smaller, But Shocking Performance

· 9 min read · 6 days ago

1.3K 13



Benedict Neo in bitgrit Data Science Publication

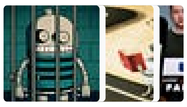
Roadmap to Learn AI in 2024

A free curriculum for hackers and programmers to learn AI

11 min read · Mar 11, 2024

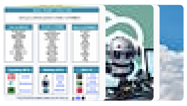
6.9K 79

Lists



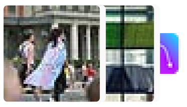
AI Regulation

6 stories · 372 saves



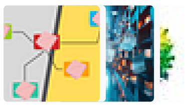
ChatGPT

21 stories · 525 saves



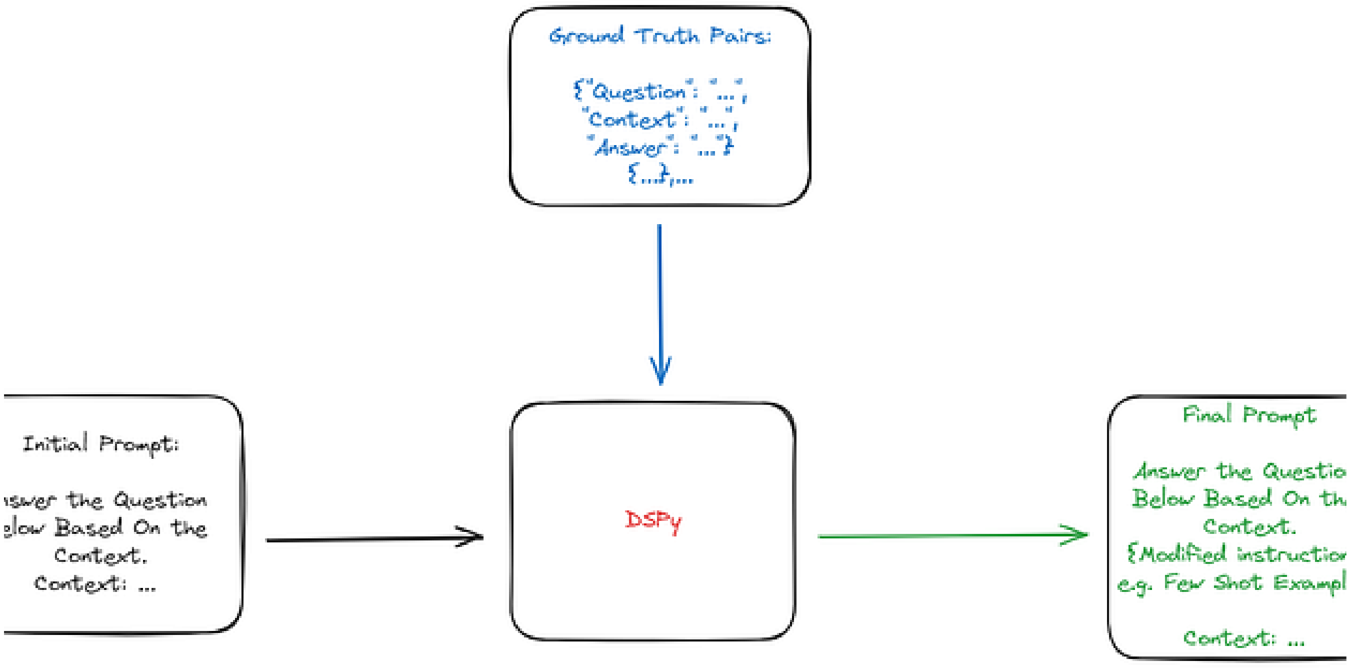
Generative AI Recommended Reading

52 stories · 841 saves



Natural Language Processing

1296 stories · 789 saves



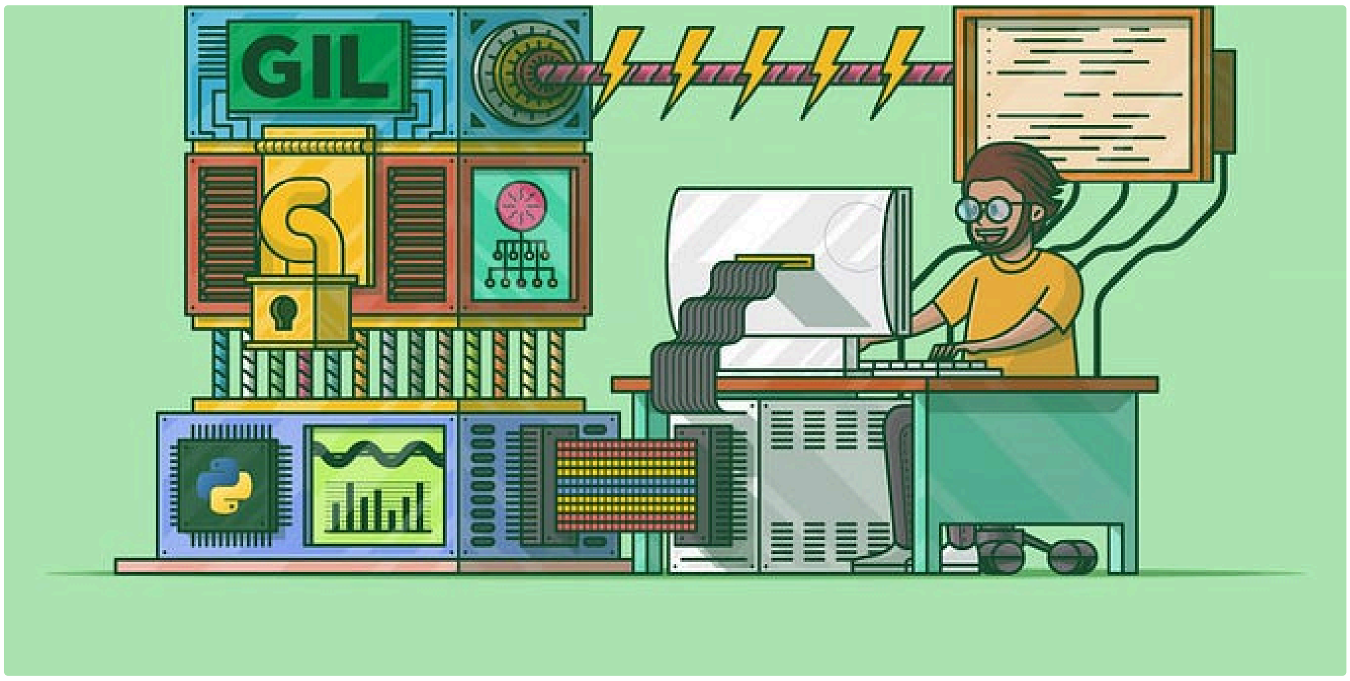
Skanda Vivek in EMAlpha

DSPy — Does It Live Up To The Hype?

The DSPy framework promises to replace manual prompt engineering with a programming framework for auto-tunded prompts. Let’s see whether…

10 min read · Mar 12, 2024

324



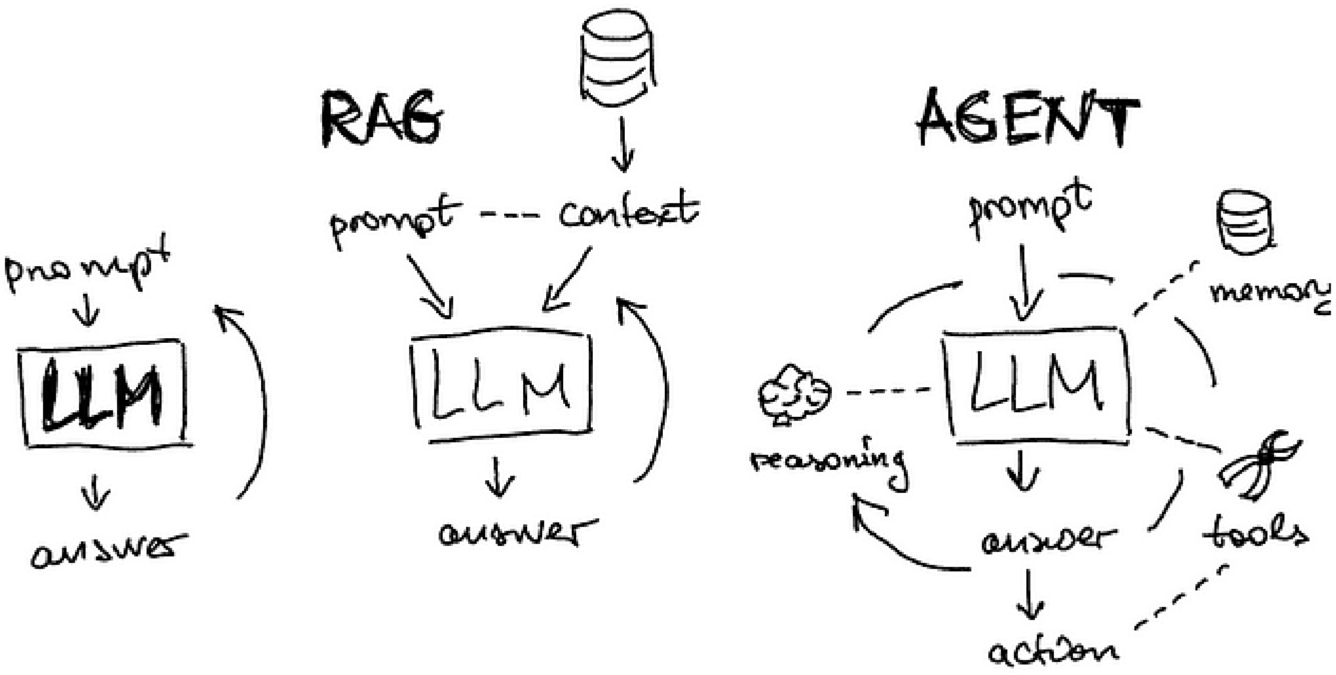
Dylan Cooper in Python in Plain English

Python is Ushering in Real Multi-Threading

The Global Interpreter Lock(GIL) can be removed. From then on, Python will no longer be what people call pseudo-multithreading.

· 5 min read · Mar 11, 2024

1.2K 9



Alex Honchar in Towards Data Science

Intro to LLM Agents with Langchain: When RAG is Not Enough

First-order principles of brain structure for AI assistants

7 min read · 3 days ago

796 5

Andrea D'Agostino in Towards AI

How to use LLMs locally with ollama and Python

This article will walk you through using ollama, a command line tool that allows you to download, explore and use Large Language Models...

· 5 min read · 5 days ago

93

See more recommendations