Fine-Tuning Small Language Models: Practical Recommendations

Liana Napalkova · Follow

14 min read · Apr 30, 2024

Listen Share

Image generated by Bing Image Creator (Microsoft), 2024 (https://www.bing.com/images/create)

This article serves as a guide for individuals looking to fine-tune small language models utilizing the HuggingFace and PyTorch. It details some of the key factors and strategies that can influence the effectiveness of the fine-tuning result. The content is structured to provide practical insights and recommendations to optimize model performance, ensuring that readers can make informed decisions throughout the fine-tuning process. It is important to note that this article does not pretend to provide an exhaustive overview of the subject but

rather focuses on highlighting key areas of consideration based on practical experienc

Content overview:

1. What are SLMs?
2. Where do SLMs excel?
3. Why fine-tune?
4. How to fine-tune?

1. What are SLMs?

Small language models (SLMs) are generative AI models based on transformer architectures that could fit on a mobile device. These models are designed to consume less energy and require less memory than their larger counterparts, making them more sustainable and environmentally friendly. For example, the Phi-2 model and the Phi-3-mini model have 2.7 billion and 3.8 billion parameters, respectively. These parameters are typically stored in full precision (fp32) or half-precision (fp16) formats. The full precision format uses 32 bits (or 4 bytes) per parameter, while half-precision format uses 16 bits (or 2 bytes) per parameter. This significantly reduces the amount of data that needs to be stored and the RAM required to process them. For instance, the Phi-2 model, when stored in fp16 (e.g., in HuggingFace hub), occupies approximately 5.4GB, which is half of the roughly 10.8GB required if the parameters were stored in full precision (fp32). This reduction in storage and memory usage not only saves space but also enhances operational efficiency, especially in environments with limited or no connectivity.

2. Where do SLMs excel?

The compact size of SLMs and quick inference capabilities make them ideal for edge computing and real-time applications. For edge applications like those used in mobile phones or isolated devices, further reducing the model size can be achieved through quantization. For example, a 4-bit quantized version of the Phi-3-mini-4K-Instruct model compresses the weight files to approximately 1.15GB (3.8 billion parameters × 0.5 bytes), making it feasible for deployment on

devices with limited RAM and storage capacity.

To demonstrate the communication systems of recreational boats. Often, recreational boaters lack practical experience in emergency procedures and may struggle to remember crucial details during emergencies. By incorporating an SLM trained on standard emergency protocols (i.e. Mayday, PAN-PAN, and Securité) into the boat's communication system, it can assist users in emergencies. Boaters could input critical details regarding their emergency. The SLM would then organize this information into a structured format that can be sent to the nearest maritime rescue coordination center.

3. Why fine-tune?

Fine-tuning is a critical step for enhancing the functionality of SLMs. This technique involves retraining a base model on a specific set of data that is relevant to the particular task or domain where the model should be applied. Through fine-tuning, an SLM can learn the specific nuances and requirements of a use case.

It's important to understand that fine-tuning is not a straightforward process that can be handled with an out-of-the-box solution. Fine-tuning requires careful customization. This often involves experimenting with different fine-tuning approaches, adjusting hyperparameters, and sometimes even modifying the model architecture itself to better align with the task at hand.

In the example of emergency communications on a recreational boat, fine-tuning helps the model understand and generate the precise language and structure required for responses according to emergency protocols. The model must be tailored to recognize and prioritize urgent language and respond appropriately.

4. How to fine-tune?

Building on the importance of fine-tuning outlined previously, let's explore how to implement this process effectively. In summary, the fine-tuning of SLMs starts with selecting a base model, which aligns closely with the desired capabilities or output types. The base model serves as the foundation for further customization.

The next step in fine-tuning involves retraining the model on a curated dataset

that is representative of the tasks the model will perform. This dataset must be carefully [illegible] that the model wi[illegible] ss adjusts the weights and biases within the model's architecture to minimize the difference between the model's predictions and the actual outcomes in the training data.

Throughout this process, adjustments are made to the model's hyperparameters to optimize learning rates, batch sizes, and other factors that influence training effectiveness.

In the continuation, I will define some of the key aspects to consider during the fine-tuning of SLMs.

4.1 Dataset quality

The dataset's quality is critical for effective fine-tuning. Smaller, well-curated datasets can yield strong results (e.g. see LIMA: Less Is More for Alignment).

Let's take the Phi-3-mini-4K-instruct model as an example: it can achieve proficient performance in function calling task with training and cross-validation on merely 80–100 carefully selected examples. For instance, each training example can be organized into three fields: function_definitions, user_query, and function_call as shown below:

```
function_definitions,user_query,function_call
"[
    {
        ""type"": ""function"",
        ""function"": {
            ""name"": ""categorize_emergency"",
            ""description"": ""Analyzes the emergency details to categorize
            ""parameters"": {
                ""type"": ""object"",
                ""properties"": {
                    ""emergency_detail"": {
                        ""type"": ""string"",
                        ""enum"": [""Fire"", ""Flooding"", ""Man Overboard""
                        ""description"": ""A brief description of the emerge
                    }
                },
                ""required"": [
```

```
                    ""emergency_detail""
```



```
    },
    {
        ""type"": ""function"",
        ""function"": {
            ""name"": ""broadcast_emergency_message"",
            ""description"": ""Broadcasts the appropriate emergency message
            ""parameters"": {
                ""type"": ""object"",
                ""properties"": {
                    ""location"": {
                        ""type"": ""string"",
                        ""description"": ""The current GPS location of the b
                    },
                    ""emergency_detail"": {
                        ""type"": ""string"",
                        ""enum"": [""Fire"", ""Flooding"", ""Man Overboard""
                        ""description"": ""A brief description of the emerge
                    },
                    ""emergency_type"": {
                        ""type"": ""string"",
                        ""enum"": [""Mayday"", ""PAN-PAN"", ""Securité""],
                        ""description"": ""The type of emergency signal to b
                    }
                },
                ""required"": [
                    ""location"",
                    ""emergency_detail"",
                    ""emergency_type""
                ]
            }
        }
    }
]",
"We are taking on water and need immediate help",
"[{
    ""name"": ""categorize_emergency"",
    ""arguments"": {
        ""emergency_detail"": ""Flooding""
    }
},{
    ""name"": ""broadcast_emergency_message"",
    ""arguments"": {
        ""location"": ""42.3541N, 71.0654W"",
        ""emergency_detail"": ""Flooding"",
        ""emergency_type"": ""Mayday""
    }
}]"
```

Using the GPT-4, to uch as . This method leverages the capabilities of more advanced models to streamline the preparation of effective training data.

4.2 Customization of the fine-tuning approach

Different use cases require specific fine-tuning approaches tailored to both the model and the context in which it is used. Customizing to minimize the number of tokens processed per call can significantly reduce operational costs. In edge computing, particularly, minimizing prompt token size is crucial for increasing battery life and reducing latency.

Let's take again the example of a recreational boat: An SLM can be fine-tuned for specific function calling tasks in emergency scenarios, where the range of possible function calls is limited and predictable. By training the SLM to directly associate user queries with function calls, it's possible to eliminate the need to pass function definitions in the prompt at inference time, thereby decreasing the token count (e.g. see the "Octopus" approach).

If the priority is to enable the model to generalize across various function calling scenarios, it is crucial to focus on broader training that covers diverse instances and contexts. This helps the model develop a robust capability to interpret and respond to a wide array of function calls, ensuring reliable performance under varied conditions.

4.3 SLM architecture selection

Selecting the appropriate model architecture and training method is crucial when fine-tuning transformer models for specific task objectives. This process involves adapting a pre-trained model, which has been initially trained using one of the following methods, to perform new or more specialized tasks:

- **Causal Language Modeling (CausalLM):** Focuses on predicting the next token based solely on the preceding sequence. Originally trained models using CausalLM are typically fine-tuned for tasks that require sequential data generation.

- **Masked Language Modeling (MLM):** Involves predicting randomly masked token ne-tuned for ta as text classification.

- **Sequence-to-Sequence (Seq2Seq):** Uses an encoder-decoder structure to transform entire input sequences into outputs. Fine-tuning Seq2Seq models is common in tasks like translation or summarization where comprehensive input-to-output mapping is required.

It should be noticed that transitioning from one task to another requires extensive modifications to both the model's architecture and its training strategy to fit the new task requirements. For example, it is challenging to fine-tune a model trained with CausalLM for a Seq2Seq task, as the fundamental approach to processing sequences in these methods differs significantly. Therefore, it is essential to verify the architecture and training method of a base model before starting the fine-tuning process.

The architecture can be determined by checking the model's configuration (config.json), where details about its architecture and pre-training method are provided, e.g.:

config.json of Phi-3-mini-4K-Instruct

More details about model architectures and training methods can be found in this blog post: Understanding Causal LLM's, Masked LLM's, and Seq2Seq: A Guide to Language Model Training Approaches | by Tomas Vykruta | Medium

4.4 Fine-tuning for a conversational mode

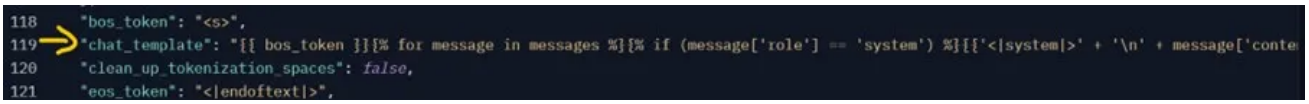
Fine-tuning for conversational mode is essential in many practical applications

where communication with end-users is necessary. Imagine teaching a model to understa sidering that a mo mple also return function calls when needed.

To instruct the model on how to process and generate responses in a conversational context, chat templates are used. A chat template defines the structure and format of the conversation, including how distinct roles (e.g., system, user, assistant) interact and the tokens used to represent their messages.

Different models expect different input formats for chat. When setting the template for a model that's already been trained for chat, it is necessary to ensure that the template exactly matches the message formatting that the model saw during training to avoid performance degradation.

The chat template of a base model can be found in the tokenizer_config.json.



tokenizer_config.json of Phi-3-mini-4K-Instruct

The chat template can be retrieved programmatically using the following code:

```
from transformers import AutoTokenizer

def load_tokenizer(base_model):
tokenizer = AutoTokenizer.from_pretrained(base_model, trust_remote_code=True

special_tokens_map = tokenizer.special_tokens_map
print("Special tokens:")
for token_type, tokens_list in special_tokens_map.items():
    print(f"{token_type}: {tokens_list}")

print("Chat template:")
print(tokenizer.chat_template)

base_model = "microsoft/Phi-3-mini-4k-instruct"
load_tokenizer(base_model)
```

For instar s with
the begin ow:


```
Chat template:

{{ bos_token }}{% for message in messages %}{% if (message['role'] == 'syste
' + message['content'] + '<|end|>' + '
'}}{% elif (message['role'] == 'user') %}{{'<|user|>' + '
' + message['content'] + '<|end|>' + '
' + '<|assistant|>' + '
'}}{% elif message['role'] == 'assistant' %}{{message['content'] + '<|end|>'
'}}{% endif %}{% endfor %}
```

To apply this chat template to the data:

```
chat = [
    {
        "role": "system",
        "content": "You are AI assistant supporting recreational boaters in
    },
    {
        "role": "user",
        "content": "We are taking on water and need immediate help"
    },
    {
        "role": "assistant",
        "content": "I'm sending the message and coordinates to the closest m
    },
    {
        "role": "user",
        "content": "Thank you."
    },
]

formatted_input = tokenizer.apply_chat_template(chat, tokenize=False, add_ge

print(formatted_input)
```

Output:

```
<s><|
You are AI assistant supporting recreational boaters in emergency situations
<|user|>
We are taking on water and need immediate help<|end|>
<|assistant|>
I'm sending the message and coordinates to the closest maritime rescue coord
<|user|>
Thank you.<|end|>
<|assistant|>
```

For models lacking chat templates, like the Phi-2 model, it's still possible to set any of existing chat template formats, e.g.:

```
tokenizer.chat_template = """{{ bos_token }}{% for message in messages %}{%
' + message['content'] + '<|end|>' + '
'}}{% elif (message['role'] == 'user') %}{{'<|user|>' + '
' + message['content'] + '<|end|>' + '
' + '<|assistant|>' + '
'}}{% elif message['role'] == 'assistant' %}{{message['content'] + '<|end|>'
'}}{% endif %}{% endfor %}"""
```

It's also possible to customize a chat template for a specific task that a model should learn during fine-tuning, for example, function calling. In this case it is necessary to add function definitions and function call roles to the chat template.

As mentioned in HuggingFace documentation, the chat template should be applied as a preprocessing step for a dataset:

```
from datasets import Dataset

dataset = Dataset.from_dict({"chat": [chat]})
dataset = dataset.map(lambda x: {"formatted_chat": tokenizer.apply_chat_temp
```

4.5 Tokenization

Investigat ssential for effecti kens are processed, ensuring that the model handles input sequences appropriately and aligns with specific training and inference requirements.

Padding tokens

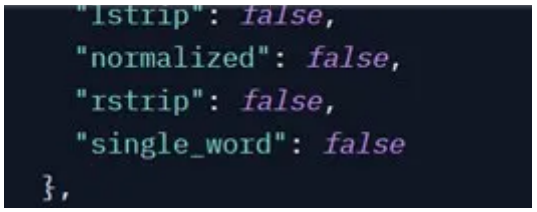
Padding tokens play a crucial role in fine-tuning SLMs by standardizing batch sizes and enhancing computational efficiency. Even though transformers are capable of handling variable sequence lengths, padding might be necessary to meet hardware requirements or simplify data processing. Correct masking implementation is essential to ensure that padding tokens do not influence the model's calculations.

It is important to confirm the presence of a padding token in both the model and tokenizer; if missing, the <unk> token may be used as an alternative. However, introducing a new padding token should be approached with caution as it increases the embedding layer's size and computational load, which can slow down inference.

BOS and EOS tokens

The beginning of sequence (BOS) and end of sequence (EOS) tokens are crucial as they clearly define the start and end of text sequences, helping the model to comprehend the context and boundaries within the data. In some cases, the tokenizer's configuration tokenizer_config.json may specify that the EOS token should not be automatically appended to every sequence ("add_eos_token": false). However, the EOS token might still be listed in the special_tokens_map.json file, as illustrated by the example from the Phi-3-mini-4K-Instruct model below.

tokenizer_config.json of Phi-3-mini-4K-Instruct



special_tokens_map.json of Phi-3-mini-4K-Instruct

This setup means that the EOS token is available for selective use in particular scenarios where marking the end of a sequence clearly is beneficial. This flexibility enables the tokenizer to meet various operational demands without universally applying the EOS token.

4.6 Full fine-tuning vs Partial fine-tuning

Full fine-tuning can require extensive GPU resources and time, depending on the model and dataset size. One of efficient alternatives is Parameter Efficient Fine-Tuning (PEFT), which updates only a selected subset of model parameters while keeping the rest unchanged. This approach reduces the overall number of trainable parameters, making memory usage more manageable and minimizing the risk of catastrophic forgetting.

One popular PEFT method is LoRA (Low-Rank Adaptation), which specifically targets and updates crucial layers in the model without directly modifying the original weights. Instead, it integrates a low-rank product with the existing model parameters during computation. To optimize fine-tuning, special focus should be placed on the following layers:

- **Attention layers:** Key for determining the focus of the model on different input parts, crucial for understanding context and data relationships.
- **Feed-forward layers:** Essential for applying transformations post-attention, playing a significant role in the model's nonlinear processing abilities. These typically involve an up-projection (expansion) and a down-projection (compression). These layers function as components of the multi-layer perceptron (MLP) modules within the architecture of a transformer model.

Setting up ...ieved by examinin... t().keys(). For instance, in the Phi-3-mini-4k-instruct model, this command returns the following modules:

```
odict_keys([
  'model.embed_tokens.weight',
  'model.layers.0.self_attn.o_proj.weight',
  'model.layers.0.self_attn.qkv_proj.weight',
  'model.layers.0.mlp.gate_up_proj.weight',
  'model.layers.0.mlp.down_proj.weight',
  'model.layers.0.input_layernorm.weight',
  'model.layers.0.post_attention_layernorm.weight',
  'model.layers.1.self_attn.o_proj.weight',
  'model.layers.1.self_attn.qkv_proj.weight',
  'model.layers.1.mlp.gate_up_proj.weight',
  'model.layers.1.mlp.down_proj.weight',
  'model.layers.1.input_layernorm.weight',
  'model.layers.1.post_attention_layernorm.weight',
   …
  'model.layers.31.self_attn.o_proj.weight',
  'model.layers.31.self_attn.qkv_proj.weight',
  'model.layers.31.mlp.gate_up_proj.weight',
  'model.layers.31.mlp.down_proj.weight',
  'model.layers.31.input_layernorm.weight',
  'model.layers.31.post_attention_layernorm.weight',
  'model.norm.weight',
  'lm_head.weight'
])
```

Reviewing the complete list of modules reveals that the attention and feed-forward functions in the Phi-3-mini-4k-instruct model are managed by the following modules:

- **self_attn.qkv_proj.weight** and **self_attn.o_proj.weight** for attention mechanisms.
- **mlp.gate_up_proj.weight** and **mlp.down_proj.weight** for feed-forward operations.

Therefore, the LoRA configuration could look like this:

```
from |

config = LoraConfig(
    r=128,
    lora_alpha=128,
    target_modules=[
        "self_attn.qkv_proj.weight",
        "self_attn.o_proj.weight",
        "mlp.gate_up_proj",
        "mlp.down_proj"
    ],
    lora_dropout=0.1,
    bias="none",
    task_type="CAUSAL_LM"
)

peft_model = get_peft_model(model, config)
```

Fine-tuning with LoRA also requires optimizing hyperparameters like rank (r), dropout rate (lora_dropout), and LoRA alpha (lora_alpha).

Other versions of LoRA, such as QLoRA and DoRA, further enhance the efficiency of this method. QLoRA utilizes quantization techniques to simplify computations by reducing numerical precision to lower bit representations like 4-bit, thus speeding up processing without a substantial loss in performance. It can be seamlessly integrated into existing LoRA setup by adding a few lines of code to enable 4-bit model loading and to employ a paged optimizer. Here's an example setup:

```
import torch
from transformers import AutoModelForCausalLM, BitsAndBytesConfig, TrainingA

bnb_config = BitsAndBytesConfig(
    load_in_4bit=True,
    bnb_4bit_use_double_quant=True,
    bnb_4bit_quant_type="nf4",
    bnb_4bit_compute_dtype=torch.bfloat16
)

model = AutoModelForCausalLM.from_pretrained(
```

```
    base_model,
    q
    t
    d
    trust_remote_code=True
)

model = prepare_model_for_kbit_training(model)

#...

peft_training_args = TrainingArguments(
    # specific arguments here
    optim="paged_adamw_8bit"
)
```


DoRA, another variant of LoRA, dynamically adjusts the rank of the matrix involved in the low-rank adaptations based on how the training progresses. The "rank" in this context refers to the complexity and number of parameters in the low-rank matrices that are used to modify the model's weights, optimizing the balance between performance and computational efficiency during fine-tuning. Enabling DoRA involves modifying the LoraConfig object:

```
config = LoraConfig(
    # specific arguments here
    target_modules=[
        # other modules
        "lora_magnitude_vector"
    ],
    use_dora=True
)
```

4.7 Hyperparameters optimization

Optimizing hyperparameters is crucial for fine-tuning SLMs. Factors like learning rate, batch size, and optimization algorithm need careful adjustment based on the model architecture and fine-tuning strategy. Generally, 1–3 training epochs suffice to mitigate overfitting, supplemented by continuous evaluation using custom loss metrics. Overfitting results in poor model generalization to new data. Employing systematic hyperparameter tuning methods such as grid or random

search, and validating on a separate test set are essential to ensure optimal model performa ce the hyperpar nance declines on the validation set.

4.8 Model compression and quantization

For deploying models in production, employing Llama.cpp is beneficial for compressing and quantizing fine-tuned models into GGUF files, suitable for execution on edge devices using CPUs.

However, if modifications such as for example the addition of new tokens and resizing of the embeddings layer, the use of a logits processor for inference, etc have been made, manual adjustments to the Llama.cpp code may be necessary. These adjustments are essential to maintain compatibility and optimal performance under customized deployment conditions. Additionally, it is crucial to ensure that the outputs of the compressed and quantized model are approximately aligned with those of the original fine-tuned model, although some reduction in quality is to be expected.

Conclusions

As explained in this article, fine-tuning is a nuanced process that demands careful customization. Fine-tuning is not an out-of-the-box solution. Each step, from dataset preparation and tokenization to model training and evaluation, requires a thorough configuration and a deep understanding of each component. This detailed approach ensures that the model is effectively adapted to the task and environment intended.

Please note that the views expressed in this article are my own and do not necessarily reflect those of Microsoft, my current employer.

References

Introducing Phi-3: Redefining what's possible with SLMs | Microsoft Azure Blog

Phi-2: The surprising power of small language models — Microsoft Research

LIMA: Less Is More for Alignment

Understa nguage
Model Tr

Fine Tuning | Small Language Model | Edge Computing

Follow

Written by Liana Napalkova

24 Followers

More from Liana Napalkova

Liana Napalkova

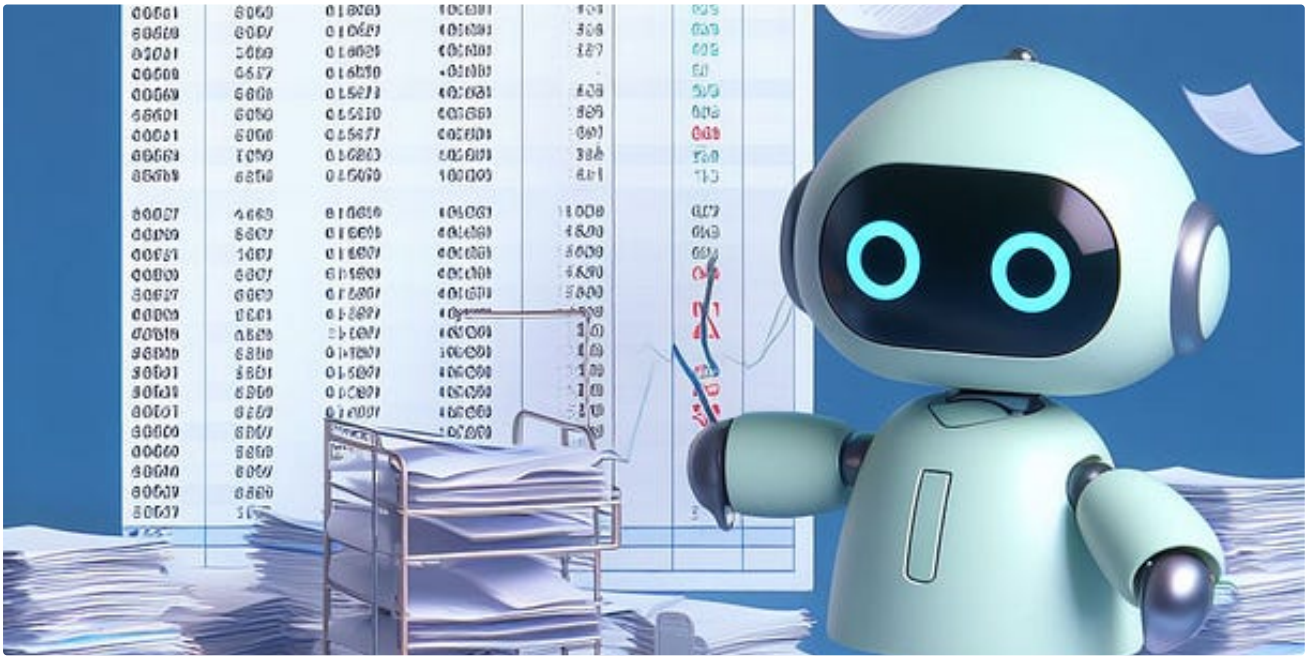
Fine-tun

In this artic -language model developed by Microsoft. We will use...

Jul 12

See all from Liana Napalkova

Recommended from Medium

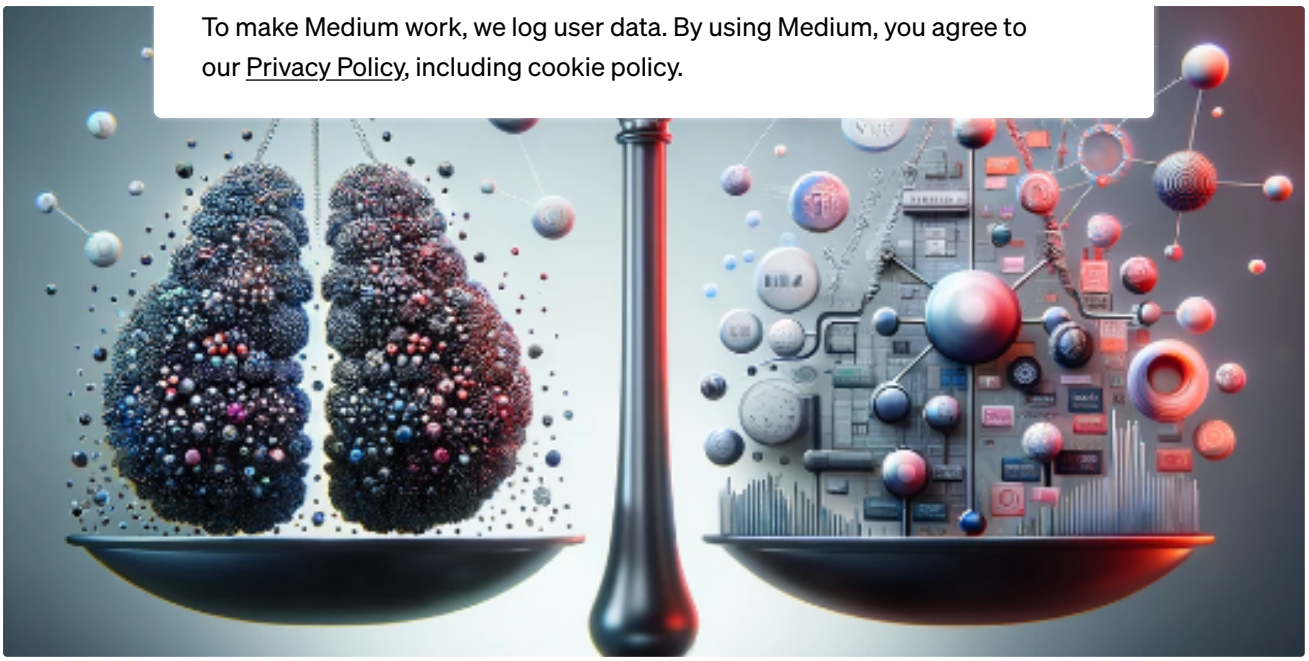


Yanli Liu in Towards Data Science

How to Generate Instruction Datasets from Any Documents for LLM Fine-Tuning

Generate high-quality synthetic datasets economically using lightweight libraries

Mar 6 1.4K 12



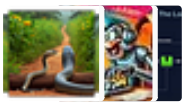
Changsha Ma

How to prepare an instruction dataset to fine-tune LLM?

Fine-tuning large language models (LLMs) on custom datasets is a popular technique to adapt these powerful models for specific downstream...

Mar 4 262

Lists



Natural Language Processing

1649 stories · 1219 saves

Use Case Families						Graphs
Forecasting						Low
Planning	Low	Low	High	Medium	Medium	High
Decision Intelligence	Low	Medium	High	High	High	Medium
Autonomous System	Low	Medium	High	Medium	Medium	Low
Segmentation	Medium	High	Low	Low	High	High
Recommender	Medium	High	Medium	Low	Medium	High
Perception	Medium	High	Low	Low	Low	Low
Intelligent Automation	Medium	High	Low	Low	High	Medium
Anomaly Detection	Medium	High	Low	Medium	Medium	High
Content Generation	High	Low	Low	High	Low	Low
Chatbots	High	High	Low	Low	Medium	High

Christopher Tao in Towards AI

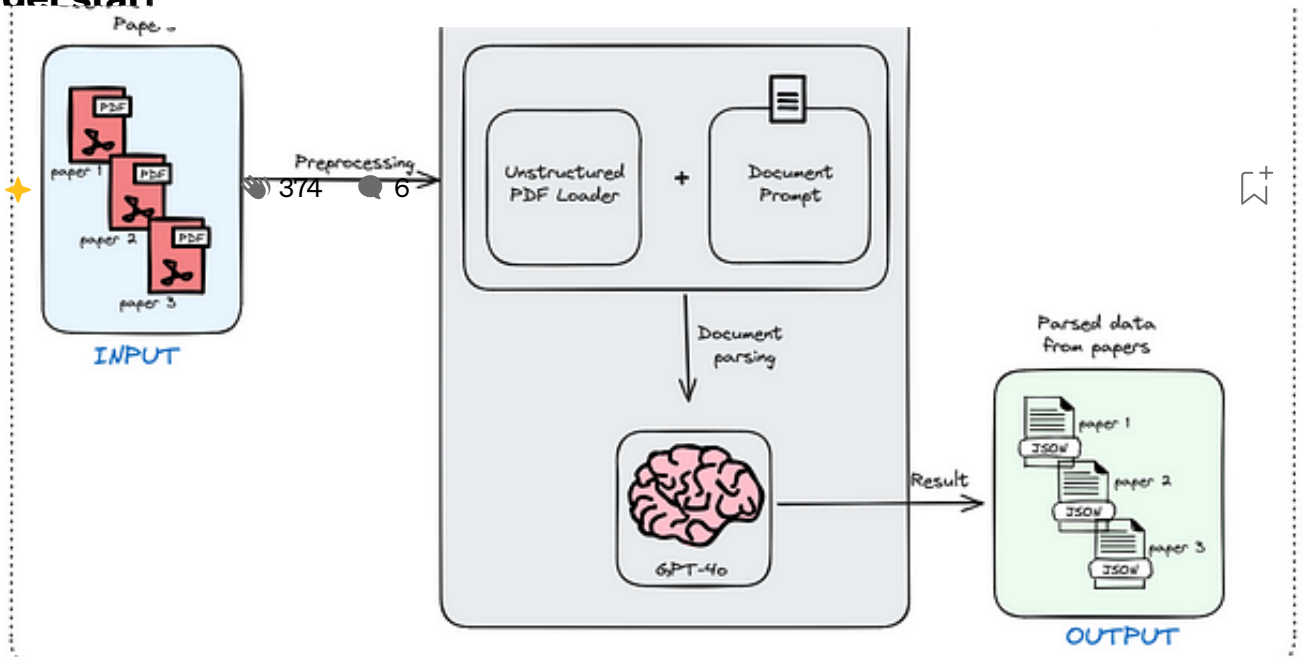
Do Not Use LLM or Generative AI For These Use Cases

Choose correct AI techniques for the right use case families

Aug 10 858 11

Sean Smith in Towards Data Science

Thinking **fore you get start**

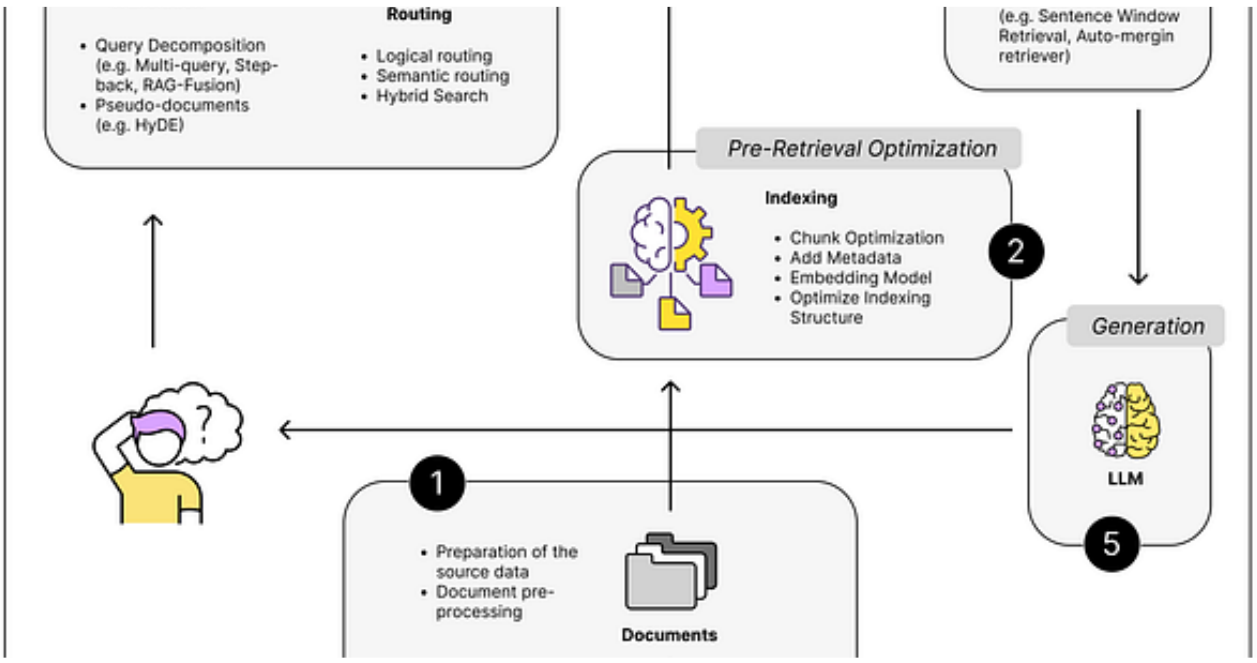


Zoumana Keita in Towards Data Science

Document Parsing Using Large Language Models — With Code

You will not think about using Regular Expressions anymore.

Jul 25 738 5



Dominik Polzer in Towards Data Science

17 (Adva **e into a Producti**

A collection of RAG techniques to help you develop your RAG app into something robust that will last

Jun 26 2.4K 22