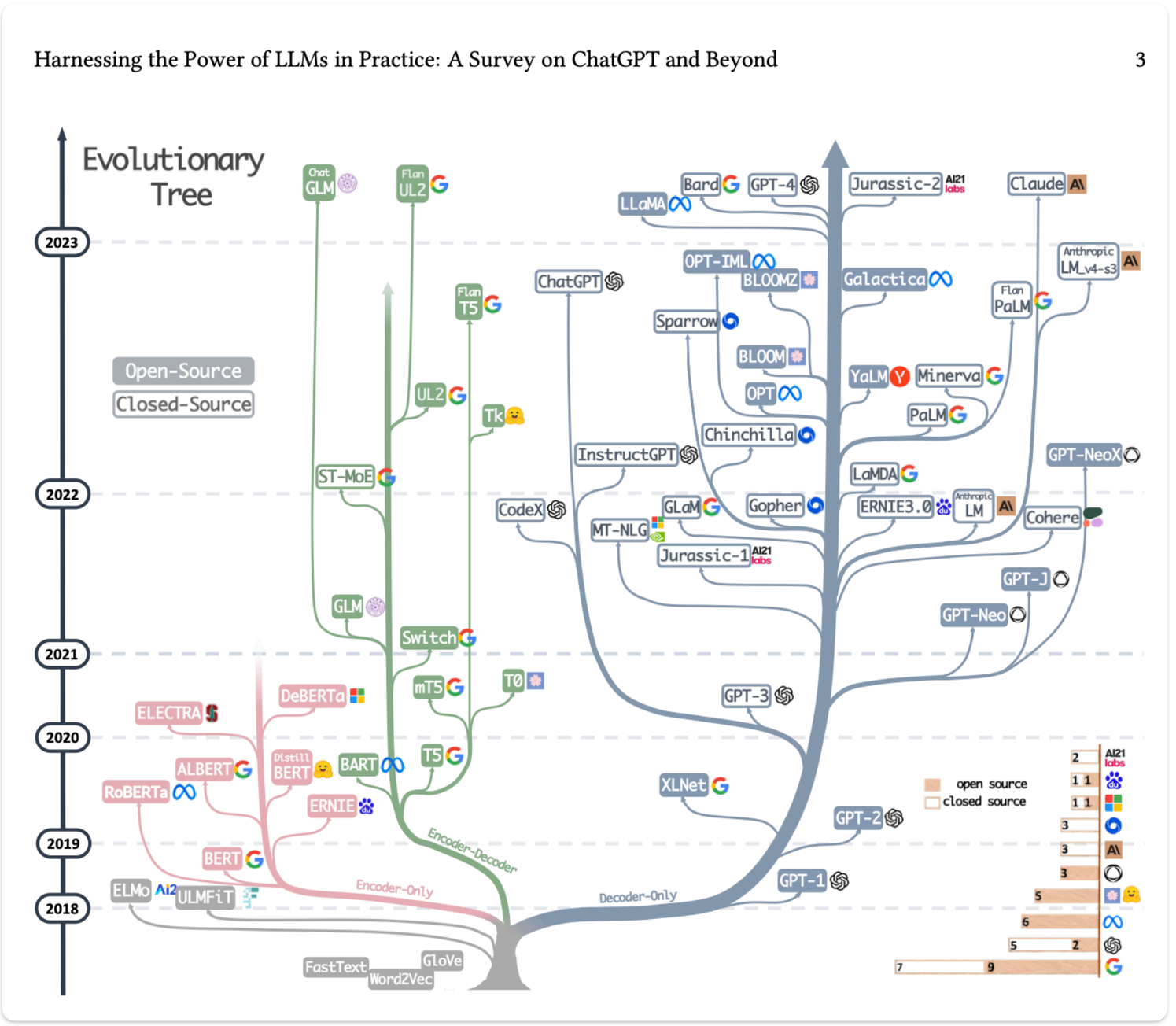
Three Main LLM Architectures

Large Language Models (LLMs) can be broadly categorized into three primary architectures, each with distinct characteristics, use cases, and training methodologies.

1. Encoder-only Models
- **Composition:** Utilizes only the **Encoder component** of the Transformer architecture.
- **Key Characteristics:**
- Specializes in **bidirectional contextual understanding**.
- Analyzes the input text deeply, understanding context from both left and right.
- Pre-trained using tasks like Masked Language Modeling (MLM) where masked tokens in a sentence are predicted.
-
- **Representative Models:** BERT, RoBERTa, ELECTRA
- **Primary Use Cases:**
- **Natural Language Understanding (NLU) tasks:** Text classification, sentiment analysis, named entity recognition, question answering (where the answer is extracted from the text).

2. Decoder-only Models
- **Composition:** Utilizes only the **Decoder component** of the Transformer architecture.
- **Key Characteristics:**
- Specializes in **unidirectional (auto-regressive) text generation**.
- Predicts the next token based on the preceding tokens.
- Forms the backbone of most recent and prominent LLMs due to their strong generative capabilities.
-
- **Representative Models:** GPT series (GPT-2, GPT-3, GPT-4), LLaMA, PaLM, Claude
- **Primary Use Cases:**
- **Natural Language Generation (NLG) tasks:** Text writing, conversational AI, story generation, code generation, summarization.
- **Prompt-based tasks:** Instruction following, few-shot/zero-shot learning.

3. Encoder-Decoder Models
- **Composition:** Utilizes **both the Encoder and Decoder components** of the Transformer architecture.
- **Key Characteristics:**
- Optimized for **sequence-to-sequence (Seq2Seq) tasks**, transforming an input sequence into a different output sequence.
- The encoder processes the input, and the decoder generates the output based on the encoder's representation.
-
- **Representative Models:** T5, BART, NLLB
- **Primary Use Cases:**
- **Seq2Seq tasks:** Machine translation, text summarization, conditional text generation (generating text based on a specific input), data-to-text generation.



LLM Training and Utilization Process

The development and utilization of LLMs can be broadly divided into two main stages: **Pre-training** and **Post-training/Fine-tuning**.

1. Pre-training
- **Definition:** The process where model developers (e.g., Google, OpenAI, Meta) train models on **massive amounts of unstructured text data (e.g., web crawl data, books, Wikipedia)** to develop the model's **fundamental language understanding and generation capabilities**.
- **Objective:** To enable the model to **autonomously** acquire statistical patterns of language, grammar, semantics, and world knowledge.
- **Training Methods:**
- Conducted through **Unsupervised Learning** or **Self-supervised Learning**.
- **Encoder-only (BERT):** Primarily learns bidirectional context through Masked Language Modeling (MLM - predicting masked words in sentences) and Next Sentence Prediction (NSP).
- **Decoder-only (GPT):** Primarily learns unidirectional text generation capabilities through Next Token Prediction.
- **Encoder-Decoder (T5, BART):** Learns by corrupting input text and reconstructing the original, or by transforming input text into output text.
- **Result:** After this stage, the model becomes a "pre-trained model" with broad language capabilities, though not optimized for specific tasks.

2. Post-training / Fine-tuning
- **Definition:** The process of taking a pre-trained model and further adjusting its **weights (internal parameters)** to suit **specific tasks (task-specific)** or **purposes (purpose-specific)**.
- **Objectives:**
- To achieve **higher performance** on specific NLP tasks (e.g., sentiment analysis, text summarization, question answering).
- To make models **better understand and follow user instructions (Instruction Following)** or **align** better with human preferences.
- **Training Methods:**
- **Data:** Uses much smaller, **labeled datasets** specialized for the specific task, compared to pre-training data.
- **Developer's Role:** Developers prepare task-specific datasets, load pre-trained models, and configure **optimizers, loss functions, learning rates, batch sizes, and number of epochs** to train the model.
- **Training Loop:** Repeats the typical deep learning training loop (forward pass → loss calculation → backpropagation → weight update).
- **Epochs:** An epoch is one complete pass through the entire training dataset. Fine-tuning often requires only a few epochs.
- **Special Considerations for GPT-series:**
- **Instruction Tuning:** Training models to respond well to various forms of instructions.
- **RLHF (Reinforcement Learning from Human Feedback):** Aligns models through reinforcement learning using human feedback to ensure responses are helpful, honest, and harmless.
- **Prompt Engineering:** Modern large decoder-only LLMs (GPT-3/4) can perform various tasks **without additional fine-tuning** through appropriate prompts alone (Few-shot/Zero-shot learning), thanks to powerful pre-training. Users often access these models via APIs and control them through prompts rather than directly fine-tuning them.
-
-

Table 1. Summary of Large Language Models.

	Characteristic		LLMs
Encoder-Decoder or Encoder-only (BERT-style)	Training: Model type: Pretrain task:	Masked Language Models Discriminative Predict masked words	ELMo [80], BERT [28], RoBERTa [65], DistilBERT [90], BioBERT [57], XLM [54], Xlnet [119], ALBERT [55], ELECTRA [24], T5 [84], GLM [123], XLM-E [20], ST-MoE [133], AlexaTM [95]
Decoder-only (GPT-style)	Training Model type: Pretrain task:	Autoregressive Language Models Generative Predict next word	GPT-3 [16], OPT [126]. PaLM [22], BLOOM [92], MT-NLG [93], GLaM [32],Gopher [83], chinchilla [41], LaMDA [102], GPT-J [107], LLaMA [103], GPT-4 [76], BloombergGPT [117]

Encoder-Decoder (T5, BART)
├── Advantages: Suitable for tasks with significantly different input/output lengths
├── Disadvantages: Minimal performance gains relative to complexity
└── Current: Used only for specialized tasks (translation, summarization)

Decoder-Only (GPT)
├── Advantages: Simple, scalable, versatile
├── Disadvantages: Limited bidirectional context understanding
└── Current: Mainstream (ChatGPT, Claude, etc.)

Encoder-Only (BERT)
├── Advantages: Excellent contextual understanding
├── Disadvantages: Cannot generate text
└── Current: Specialized for classification/extraction tasks

1. Encoder-Decoder Architecture (T5, BART)

Main Use Cases: Translation, Summarization, Question Answering, Text-to-Text tasks

```
from transformers import T5ForConditionalGeneration, T5Tokenizer

# Load pre-trained T5 model
model = T5ForConditionalGeneration.from_pretrained("t5-small")
tokenizer = T5Tokenizer.from_pretrained("t5-small")

# Example 1: Translation
input_text = "translate English to French: The house is wonderful."
inputs = tokenizer(input_text, return_tensors="pt")
outputs = model.generate(**inputs, max_length=50)
result = tokenizer.decode(outputs[0], skip_special_tokens=True)
print(f"Translation: {result}")  # "La maison est merveilleuse."

# Example 2: Summarization
text = """summarize: The tower is 324 metres tall, about the same height as
an 81-storey building. It is the tallest structure in Paris."""
inputs = tokenizer(text, return_tensors="pt", max_length=512, truncation=True)
outputs = model.generate(**inputs, max_length=50, min_length=10)
summary = tokenizer.decode(outputs[0], skip_special_tokens=True)
print(f"Summary: {summary}")

# Example 3: Question Answering
qa_input = """question: What is the capital of France?
context: Paris is the capital and most populous city of France."""
inputs = tokenizer(qa_input, return_tensors="pt")
outputs = model.generate(**inputs)
answer = tokenizer.decode(outputs[0], skip_special_tokens=True)
print(f"Answer: {answer}")  # "Paris"
```

2. Decoder-Only Architecture (GPT)

Main Use Cases: Text Generation, Completion, Chat, Creative Writing

```
from transformers import GPT2LMHeadModel, GPT2Tokenizer

# Load pre-trained GPT-2 model
model = GPT2LMHeadModel.from_pretrained("gpt2")
tokenizer = GPT2Tokenizer.from_pretrained("gpt2")
tokenizer.pad_token = tokenizer.eos_token

# Example 1: Text Completion
prompt = "The future of artificial intelligence is"
inputs = tokenizer(prompt, return_tensors="pt")
outputs = model.generate(
    **inputs,
    max_length=50,
    num_return_sequences=1,
    temperature=0.8,
    do_sample=True
)
generated = tokenizer.decode(outputs[0], skip_special_tokens=True)
print(f"Generated: {generated}")

# Example 2: Story Generation
story_prompt = "Once upon a time in a magical forest,"
inputs = tokenizer(story_prompt, return_tensors="pt")
outputs = model.generate(
    **inputs,
    max_length=100,
    temperature=0.9,
    top_p=0.95,
    do_sample=True
)
story = tokenizer.decode(outputs[0], skip_special_tokens=True)
print(f"Story: {story}")

# Example 3: Code Generation (with CodeGen or similar)
code_prompt = "def fibonacci(n):"
inputs = tokenizer(code_prompt, return_tensors="pt")
outputs = model.generate(**inputs, max_length=100, temperature=0.2)
code = tokenizer.decode(outputs[0], skip_special_tokens=True)
print(f"Code: {code}")
```

3. Encoder-Only Architecture (BERT)

Main Use Cases: Classification, Named Entity Recognition, Sentiment Analysis, Token Classification

```
from transformers import (
    BertForSequenceClassification,
    BertForTokenClassification,
    BertTokenizer,
    pipeline
)
import torch

# Example 1: Sentiment Classification
model = BertForSequenceClassification.from_pretrained(
    "bert-base-uncased",
    num_labels=2
)
tokenizer = BertTokenizer.from_pretrained("bert-base-uncased")

text = "I love this movie! It's absolutely fantastic."
inputs = tokenizer(text, return_tensors="pt", padding=True, truncation=True)
with torch.no_grad():
    outputs = model(**inputs)
    predictions = torch.nn.functional.softmax(outputs.logits, dim=-1)
    predicted_class = torch.argmax(predictions, dim=-1)
print(f"Sentiment: {'Positive' if predicted_class == 1 else 'Negative'}")

# Example 2: Named Entity Recognition (NER)
ner_pipeline = pipeline(
    "ner",
    model="bert-base-cased",
    aggregation_strategy="simple"
)
text = "Apple Inc. was founded by Steve Jobs in Cupertino."
entities = ner_pipeline(text)
for entity in entities:
    print(f"Entity: {entity['word']} - Type: {entity['entity_group']}")

# Example 3: Question Answering (Extractive)
from transformers import BertForQuestionAnswering

qa_model = BertForQuestionAnswering.from_pretrained(
    "bert-large-uncased-whole-word-masking-finetuned-squad"
)
qa_tokenizer = BertTokenizer.from_pretrained(
    "bert-large-uncased-whole-word-masking-finetuned-squad"
)

question = "What is the capital of France?"
context = "Paris is the capital and largest city of France."

inputs = qa_tokenizer(question, context, return_tensors="pt")
with torch.no_grad():
    outputs = qa_model(**inputs)
    start_pos = torch.argmax(outputs.start_logits)
    end_pos = torch.argmax(outputs.end_logits)

answer_tokens = inputs["input_ids"][0][start_pos:end_pos+1]
answer = qa_tokenizer.decode(answer_tokens)
print(f"Answer: {answer}")  # "paris"
```

Summary of GPT-style Language Models: Decoder-only

GPT-style language models are characterized by their decoder-only architecture and an autoregressive training paradigm. While language models are generally task-agnostic in their architecture, traditional methods often require fine-tuning on specific downstream task datasets.

1. Key Training Paradigm: Autoregressive Language Models (Including Paper Content):
- **Concept:** These models are trained by generating the next word in a sequence given the preceding words. This "next-token prediction" approach is fundamental to their operation.
- **Mechanism:** Unlike BERT-style models that mask tokens, GPT models predict tokens sequentially, learning to model the probability distribution of natural language.
- **Examples:** Notable examples include GPT-3 [16], OPT [126], PaLM [22], and BLOOM [92].

2. Scaling and Performance Breakthroughs (Including Paper Content):
- **Scaling Effect:** Researchers found that significantly scaling up language models (i.e., increasing their size and the amount of training data) dramatically improves their performance, particularly in few-shot and even zero-shot scenarios [16].
- **Few-shot / Zero-shot Learning:** This means the models can perform tasks with very few examples, or sometimes no examples at all, without explicit fine-tuning for that specific task.
- **GPT-3's Impact:** GPT-3 was a "game changer," being the first to demonstrate reasonable few-shot and zero-shot performance through **prompting** and **in-context learning**. This showcased the superior capability of autoregressive language models in this regard.
- **Prompting:** Providing a natural language instruction or context to guide the model's generation.
- **In-context Learning:** The model learns from examples given directly in the prompt, without updating its weights.
-

3. Specialized Models and Recent Developments (Including Paper Content):
- **Task-Specific Optimizations:** Some autoregressive models are optimized for specific tasks, such as CodeX [2] for code generation or BloombergGPT [117] for the financial domain.
- **ChatGPT's Breakthrough:** ChatGPT represents a recent significant advancement. It refines GPT-3 specifically for conversational tasks, leading to more interactive, coherent, and context-aware conversations for various real-world applications.

```
# 1. Import necessary libraries
from datasets import load_dataset, load_metric
from transformers import AutoTokenizer, AutoModelForSequenceClassification, TrainingArguments, Trainer
import numpy as np

# 2. Load the dataset
print("1. Loading dataset...")
# IMDB movie review dataset (positive/negative classification)
dataset = load_dataset("imdb")

# 3. Load the tokenizer (tokenizer matching the pre-trained model)
# We will be using the 'bert-base-uncased' model here, so we load its corresponding tokenizer.
print("2. Loading tokenizer...")
tokenizer = AutoTokenizer.from_pretrained("bert-base-uncased")

# 4. Preprocess data (tokenization)
# Convert text into a format the model can understand and process.
def tokenize_function(examples):
    # Tokenize the 'text' column, and pad or truncate to the maximum length.
    return tokenizer(examples["text"], padding="max_length", truncation=True)

print("3. Preprocessing data (tokenization)...")
# Apply the tokenization function to the 'train' and 'test' splits of the dataset.
tokenized_datasets = dataset.map(tokenize_function, batched=True)

# 5. Load the model (pre-trained model)
# Load a pre-trained BERT model for sequence classification.
# num_labels is the number of classes in the dataset (IMDB has 2: positive/negative).
print("4. Loading model...")
model = AutoModelForSequenceClassification.from_pretrained("bert-base-uncased", num_labels=2)

# 6. Training configuration (TrainingArguments)
# Define various settings required for training (number of epochs, batch size, learning rate, etc.).
print("5. Configuring training settings...")
training_args = TrainingArguments(
    output_dir="./results",          # Directory to save outputs
    num_train_epochs=3,              # Number of training epochs
    per_device_train_batch_size=16,  # Training batch size per device
    per_device_eval_batch_size=16,   # Evaluation batch size per device
    warmup_steps=500,                # Number of warmup steps for learning rate scheduler
    weight_decay=0.01,               # Weight decay (regularization)
    logging_dir="./logs",            # Directory to save logs
    logging_steps=100,               # Log every X update steps
    evaluation_strategy="epoch",     # Evaluate at the end of each epoch
    save_strategy="epoch",           # Save model at the end of each epoch
    load_best_model_at_end=True,     # Load the best model at the end of training
    metric_for_best_model="accuracy",# Metric to use for selecting the best model
)

# 7. Define evaluation metric
# Define the metric (accuracy in this case) to evaluate model performance.
print("6. Defining evaluation metric...")
metric = load_metric("accuracy")

def compute_metrics(eval_pred):
    logits, labels = eval_pred
    predictions = np.argmax(logits, axis=-1)
    return metric.compute(predictions=predictions, references=labels)

# 8. Create Trainer object and start training
# Trainer abstracts the training loop, simplifying the fine-tuning process.
print("7. Creating Trainer object and starting training...")
trainer = Trainer(
    model=model,                         # The model to train
    args=training_args,                  # Training arguments
    train_dataset=tokenized_datasets["train"].shuffle(seed=42).select(range(5000)), # Training dataset (using a subset)
    eval_dataset=tokenized_datasets["test"].select(range(1000)),   # Evaluation dataset (using a subset)
    tokenizer=tokenizer,                 # Tokenizer (used for data collation)
    compute_metrics=compute_metrics,     # Function to compute evaluation metrics
)

# Start training!
trainer.train()

print("\nFine-tuning complete!")

# Optional: Save the fine-tuned model
# trainer.save_model("./my_finetuned_bert_model")

# Example of loading and using the saved model
# from transformers import pipeline
# classifier = pipeline("sentiment-analysis", model="./my_finetuned_bert_model", tokenizer="bert-base-uncased")
# print(classifier("This movie was absolutely fantastic!"))
# print(classifier("I hated every minute of it."))
```

Summary of BERT-style Language Model Pre-training Process

BERT-style language models (e.g., BERT, RoBERTa) are **pre-trained** using large-scale unsupervised learning data. The core of this process is to enable the model to deeply understand the relationships between words and their contextual meanings.

1. Data Sources (Including Paper Content):
- Raw Natural Language Data:These models are trained using vast amounts of text data, known as **corpora**, collected from the internet.
- Common Crawl (Mentioned Example):** Large-scale web crawling data like that from `commoncrawl.org` serves as a prime example of the massive data sources used for model pre-training. This data includes various forms of text such as web pages, articles, and books, forming the foundation for training large language models. (e.g., RoBERTa [65] utilized large datasets including Common Crawl).

2. Core Training Paradigm: **Masked Language Model (MLM)** (Including Paper Content):
Concept: The "Masked Language Model" is the core unsupervised learning method for BERT-style models.
Training Method: During training, a random portion of words (tokens) in an input sentence are masked with a `[MASK]` token. The model is then trained to predict the original masked word based on the surrounding context.
Example: In a phrase like "I drink [MASK]," the model must predict words like 'water', 'coffee', etc.
Learning Objective:Through this training, the model develops capabilities beyond simple memorization:
Word Meaning and Contextual Usage: It learns how words are used and what they mean in specific contexts.
Relationships Between Words:It understands the semantic and syntactic relationships between words.
Grammatical Structure and Dependencies: It learns the sentence structure and grammatical dependencies between words.
Result:This training process allows the model to "develop a deeper understanding of the relationships between words and the context in which they are used."

3. Technical Implementation and Resources:**
Tools:Hugging Face's Transformers library provides BERT-like model architectures, and the Datasets library is useful for handling large-scale datasets.
Preprocessing: Raw data like Common Crawl requires significant preprocessing, including deduplication, filtering, and cleaning, before it can be used for model training.
Resources:Pre-training large language models is a resource-intensive task, requiring substantial computational resources (high-performance GPUs) and time.

In conclusion, the pre-training of BERT-style language models does not involve humans directly "injecting" data. Instead, it's a process where the model autonomously learns the contextual meaning and relationships of words from massive amounts of raw text data (e.g., Common Crawl) through an unsupervised learning method called Masked Language Model (MLM). This pre-training endows the model with powerful general language understanding capabilities, enabling it to perform a wide range of NLP tasks.

Dataset n Process

Pre-training Data:

Collection and Curation:Diverse text sources such as books, articles, and websites were collected and carefully curated to comprehensively represent human knowledge, linguistic nuances, and cultural perspectives.
Purpose:To enable LLMs to learn word knowledge, grammar, syntax, semantics, and the ability to recognize context and generate coherent responses.
Characteristics:Includes high-quality, sufficient quantity, and diverse data from various fields to enhance model performance (e.g., multilingual data, social media conversations, book corpora, code data).

Fine-tuning Data:

Usage by Scenario:
Zero-shot:When no annotated data is available, LLMs are used directly to perform tasks (no parameter changes).
Few-shot: A small number of examples are directly included in the LLM's input prompt to guide the model to generalize to the task through in-context learning.
Abundant data:
Fine-tuned models:Models are fine-tuned to fit the data well for maximized performance.
LLMs: Can be used when certain constraints, such as privacy, are present.
*Purpose:To adapt LLMs for specific downstream tasks or to enhance the performance of existing models.

Test/User Datasets

main challenges encountered when deploying LLMs in real-world

Distributional Differences between Training and Real-World Data:
- **Out-of-Distribution (OOD) Variations**
  degrades when the model's training data domain (e.g., news articles from the domain of real-world usage (e.g., medical records).
- **Domain Shifts**
  - The model may fail to perform correctly when encountering new data or patterns not seen in the training data.
- **Adversarial Examples**
  •The model can malfunction or expose security vulnerabilities given intentionally manipulated inputs.
-
- **Hallucinations**
- **Bias**

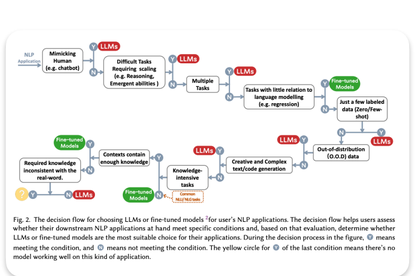


Fig. 2. The decision flow for choosing LLMs or fine-tuned models [2] for user's NLP applications. The decision flow helps users assess whether their downstream NLP applications at hand meet specific conditions and, based on that evaluation, determine whether LLMs or fine-tuned models are the most suitable choice for their applications. During the decision process in the figure, ◎ means meeting the condition, and ◎ means not meeting the condition. The yellow circle for ◎ of the last condition means there's no model working well on this kind of application.

NLP TASKS

1. NLU
2. NLG
3. Reasoning
4. Long-Text Understanding
5. Multimodality

NLU(Natural Language Understanding)

Text Classification :
Sentiment Analysis :IMDB, SST
Toxicity Detection :CivilComments
NER(Name entity Recognition) : CoNLL03
Entailment Prediction / Natural Language Inference, NLI :RTE, SNLI, CB
Question Answering, QA : SQuADv2, QuAC, CoQA
Information Retrieval, IR. :MS MARCO (regular/TREC)
Dependency Parsing

NLG(Natural Language Generation)
- Text Summarization
- Machine Translation
- Dialogue Generation
- Code Generation (implicitly included as a generation task)
- Creative Writing / Open-ended Text Generation

Reasoning Tasks
- **Common Sense Reasoning:** HellaSwag, PIQA, ARC-C, ARC-E, OpenBookQA)
- **Mathematical Reasoning :**GSM8K, MATH
- **Symbolic Reasoning :** Date Understanding, Sport Understanding
- **Code Reasoning** : CodeXGLUE-CC, CodeXGLUE-CS
- **Multihop Reasoning:**HotpotQA
- **Complex Instruction Following**

Long Text

- **Common Sense Reasoning:** HellaSwag, PIQA, ARC-C, ARC-E, OpenBookQA)
- **Mathematical Reasoning :**GSM8K, MATH
- **Symbolic Reasoning :** Date Understanding, Sport Understanding
- **Code Reasoning** : CodeXGLUE-CC, CodeXGLUE-CS
- **Multihop Reasoning:**HotpotQA
- **Complex Instruction Following**