GenAI Interview: Complete LLM Quality & Evaluation Study Guide

Introduction

If you're preparing for a GenAI interview at an enterprise company, you need to understand that the conversation will center around one critical question: "How do you prove your LLM actually works, stays safe, and scales in production?" This isn't about memorizing metric names—it's about deeply understanding how to evaluate, monitor, and improve large language models in real-world settings where mistakes can cost money, damage reputation, or harm users.

This guide walks you through everything you need to know, from foundational prerequisites to advanced production systems. Each section includes clear explanations, concrete examples, and hands-on activities to solidify your understanding.

---

0. Foundational Prerequisites

Before diving into LLM-specific evaluation, you need a solid foundation in several areas. These aren't just checkboxes—they're the building blocks that will help you understand why certain evaluation approaches work and others fail.

Math & Statistics

Understanding probability and statistics is essential because LLMs are fundamentally probabilistic systems. When a model generates text, it's actually predicting probability distributions over vocabulary tokens at each step.

Probability & Distributions

You need to understand how probability distributions work, particularly multinomial distributions (which describe the probability of different categorical outcomes) and Gaussian distributions (the bell curve that describes many natural phenomena). When an LLM generates the next token, it's computing a multinomial distribution over its entire vocabulary. Understanding this helps you interpret model confidence and uncertainty.

Concrete Example: Imagine you ask an LLM to complete the sentence "The capital of France is ___". The model doesn't just output "Paris"—internally, it computes probabilities for every word in its vocabulary:

- "Paris": 0.85 (85% probability)
- "Lyon": 0.08 (8% probability)
- "London": 0.03 (3% probability)
- "the": 0.01 (1% probability)

- All other words: remaining 3%

This is a multinomial distribution. The model then samples from this distribution (or picks the highest probability) to generate the next token. When you set temperature higher, you flatten this distribution, making unlikely words more probable (more creative but potentially less accurate). When temperature is low, the distribution becomes sharper, heavily favoring the most likely token (more deterministic but potentially boring).

Hands-On Activity:

1. Use OpenAI's API or HuggingFace with `output_scores=True` to see actual token probabilities
2. Prompt: "The weather today is" and examine top 10 token probabilities
3. Try the same prompt with temperature=0.1, then temperature=1.5
4. Notice how the probability distribution changes—high temperature spreads probability across more tokens

python

```python
from transformers import AutoTokenizer, AutoModelForCausalLM
import torch

model_name = "gpt2"
tokenizer = AutoTokenizer.from_pretrained(model_name)
model = AutoModelForCausalLM.from_pretrained(model_name)

prompt = "The weather today is"
inputs = tokenizer(prompt, return_tensors="pt")

# Generate with different temperatures
for temp in [0.1, 1.0, 2.0]:
    outputs = model.generate(**inputs, max_length=20,
                             temperature=temp,
                             output_scores=True,
                             return_dict_in_generate=True)

    # Look at probabilities for the first generated token
    first_token_scores = outputs.scores[0][0]
    probs = torch.softmax(first_token_scores, dim=-1)

    # Get top 5 most likely tokens
    top_probs, top_indices = torch.topk(probs, 5)
    print(f"\nTemperature {temp}:")
    for prob, idx in zip(top_probs, top_indices):
        print(f"  {tokenizer.decode([idx])}: {prob:.4f}")
```

Divergence Measures

These mathematical tools let you compare probability distributions, which is crucial for evaluating LLMs. Think of them as ways to measure "how different" two probability distributions are from each other.

Cross-Entropy: This measures how well one distribution predicts another. It's the basis for the loss function used to train language models. Lower cross-entropy means better predictions.

Concrete Example: Suppose you're training a model to predict the next word. The true distribution (what actually appears in your training data) after "I love" might be:

- "you": 0.4
- "pizza": 0.3
- "it": 0.2
- "dogs": 0.1

Your model predicts:

- "you": 0.5
- "pizza": 0.2
- "it": 0.2
- "dogs": 0.1

Cross-entropy = $-[0.4\times\log(0.5) + 0.3\times\log(0.2) + 0.2\times\log(0.2) + 0.1\times\log(0.1)] = 1.28$

If your model predicted perfectly (same distribution as truth), cross-entropy would be lower. If your model predicted completely wrong distributions, cross-entropy would be very high. During training, we minimize cross-entropy to make the model's predictions match reality better.

KL Divergence (Kullback-Leibler): This measures how one probability distribution differs from a reference distribution. Unlike cross-entropy, it explicitly compares two distributions and tells you the "information lost" when approximating one distribution with another.

Concrete Example: Let's say you fine-tune GPT-4 for medical advice. You want to check if the fine-tuned model's output distribution has drifted too far from the original model (which had broad safety training). You could:

1. Generate 1000 responses from both models to the same prompts
2. Compute the distribution of topics, toxicity scores, or sentiment
3. Calculate KL divergence between the two distributions

Original model topic distribution:

- Medical facts: 0.6
- Disclaimers: 0.3
- Refusals: 0.1

Fine-tuned model:

- Medical facts: 0.9
- Disclaimers: 0.08
- Refusals: 0.02

KL divergence = $0.6 \times \log(0.6/0.9) + 0.3 \times \log(0.3/0.08) + 0.1 \times \log(0.1/0.02) = 0.29$

A high KL divergence (like above) warns you that your fine-tuned model behaves very differently—it might be giving medical advice too confidently without appropriate disclaimers. This is a red flag for production deployment.

Jensen-Shannon Divergence: This is a symmetric version of KL divergence. It's useful when you want to compare two distributions without treating one as the "reference." It's bounded between 0 and 1, making it easier to interpret.

Concrete Example: You're A/B testing two different LLM versions for customer support. You want to know if they generate meaningfully different types of responses. Collect 500 responses from each model and analyze:

Model A response characteristics:

- Apologetic tone: 0.5
- Solution-focused: 0.3
- Question-asking: 0.2

Model B response characteristics:

- Apologetic tone: 0.3
- Solution-focused: 0.5
- Question-asking: 0.2

Calculate JS divergence to quantify how different these response styles are. If JS divergence is small (<0.1), the models are behaviorally similar despite different training. If it's large (>0.3), you're essentially deploying a completely different personality to customers.

Hands-On Activity:

```python
import numpy as np
from scipy.spatial.distance import jensenshannon
from scipy.special import kl_div

# Compare two hypothetical LLM output distributions
# Distribution of sentiment in generated reviews
model_a_sentiment = np.array([0.6, 0.3, 0.1])  # positive, neutral, negative
model_b_sentiment = np.array([0.4, 0.4, 0.2])

# Calculate KL divergence (asymmetric)
kl_a_to_b = np.sum(kl_div(model_a_sentiment, model_b_sentiment))
kl_b_to_a = np.sum(kl_div(model_b_sentiment, model_a_sentiment))

print(f"KL(A||B): {kl_a_to_b:.4f}")
print(f"KL(B||A): {kl_b_to_a:.4f}")
print(f"Notice they're different - KL is asymmetric!")

# Calculate JS divergence (symmetric)
js_div = jensenshannon(model_a_sentiment, model_b_sentiment)
print(f"\nJS divergence: {js_div:.4f}")
print(f"JS is always symmetric and bounded [0,1]")

# Practical interpretation
if js_div < 0.1:
    print("Models are very similar in behavior")
elif js_div < 0.3:
    print("Models have moderate differences")
else:
    print("Models behave quite differently - investigate before deployment!")
```

Exercise Challenge:

1. Generate 50 completions from GPT-3.5 and GPT-4 for the same prompt
2. Analyze the distribution of response lengths (bucket into: short <50 tokens, medium 50-150, long >150)
3. Calculate JS divergence between the two models' length distributions
4. What does this tell you about their behavior differences?

Statistical Testing & Confidence Intervals

When you're evaluating whether one model is truly better than another, you can't just look at raw scores. You need hypothesis testing to determine if differences are statistically significant or just random noise.

Concrete Example: You're comparing two chatbots. Chatbot A gets 4.2/5 average rating from 100 users. Chatbot B gets 4.4/5 from 100 users. Is B really better, or did B just get luckier with which users it served?

This is where hypothesis testing comes in. Your null hypothesis is "there's no real difference between the models." You calculate a p-value, which tells you the probability of seeing this difference (or larger) if the null hypothesis were true. If p-value < 0.05, you reject the null hypothesis and conclude B is statistically significantly better.

Confidence intervals tell you the range within which you can expect the true performance to fall. If Chatbot B has a 95% confidence interval of [4.2, 4.6], that means you're 95% confident the true average rating is somewhere in that range. Notice it overlaps with Chatbot A's score of 4.2—this suggests the difference might not be meaningful.

Hands-On Activity:

```python
```

```python
import numpy as np
from scipy import stats

# Simulate user ratings for two models
np.random.seed(42)
model_a_ratings = np.random.normal(4.2, 0.8, 100)  # mean=4.2, std=0.8, n=100
model_b_ratings = np.random.normal(4.4, 0.8, 100)  # mean=4.4, std=0.8, n=100

# Perform t-test
t_statistic, p_value = stats.ttest_ind(model_a_ratings, model_b_ratings)

print(f"Model A mean: {np.mean(model_a_ratings):.3f}")
print(f"Model B mean: {np.mean(model_b_ratings):.3f}")
print(f"Difference: {np.mean(model_b_ratings) - np.mean(model_a_ratings):.3f}")
print(f"\nT-statistic: {t_statistic:.3f}")
print(f"P-value: {p_value:.4f}")

if p_value < 0.05:
    print("✓ Statistically significant difference (p < 0.05)")
    print("  Safe to conclude Model B is truly better")
else:
    print("✗ NOT statistically significant (p >= 0.05)")
    print("  Difference might be due to random chance")

# Calculate 95% confidence intervals
ci_a = stats.t.interval(0.95, len(model_a_ratings)-1,
                        loc=np.mean(model_a_ratings),
                        scale=stats.sem(model_a_ratings))
ci_b = stats.t.interval(0.95, len(model_b_ratings)-1,
                        loc=np.mean(model_b_ratings),
                        scale=stats.sem(model_b_ratings))

print(f"\nModel A 95% CI: [{ci_a[0]:.3f}, {ci_a[1]:.3f}]")
print(f"Model B 95% CI: [{ci_b[0]:.3f}, {ci_b[1]:.3f}]")

if ci_a[1] < ci_b[0]:
    print("✓ Confidence intervals don't overlap - strong evidence B is better")
elif ci_a[0] > ci_b[1]:
    print("✓ Confidence intervals don't overlap - strong evidence A is better")
else:
    print("⚠ Confidence intervals overlap - difference might not be meaningful")
```

Real Interview Scenario: You'll be asked: "We A/B tested two models and Model X scored 2% higher on our eval set. Should we deploy it?" Your answer should consider:

- Sample size (100 examples vs. 10,000 examples matters)
- Statistical significance (was this tested properly?)
- Practical significance (is 2% worth the deployment cost?)
- Variance in the metric (high variance means less confidence)

Machine Learning Basics

Even though LLMs are specialized deep learning systems, they still follow fundamental machine learning principles. Understanding these basics helps you avoid common pitfalls in evaluation.

Supervised vs Unsupervised Learning

Supervised Learning uses labeled examples to teach models. You provide input-output pairs, and the model learns to map inputs to outputs. Examples: sentiment classification (text → positive/negative label), named entity recognition (text → tagged entities), question answering (question + context → answer).

Unsupervised Learning finds patterns in data without labels. The model discovers structure on its own. Examples: clustering similar documents, topic modeling, anomaly detection.

Why This Matters for LLMs: Large language models use both paradigms at different stages:

1. **Pre-training** (mostly unsupervised): The model learns to predict the next token from billions of web pages. No human labels needed—the "label" is just the next word that appeared in the text. This is why we can train on massive datasets.
2. **Fine-tuning** (supervised): We take the pre-trained model and train it on specific tasks with human-labeled examples. For instance, we might show it 10,000 examples of good customer service responses to teach it how to handle support queries.
3. **RLHF - Reinforcement Learning from Human Feedback** (supervised + reinforcement learning): Humans rank multiple model outputs, and the model learns to prefer outputs that humans rate highly.

Concrete Example:

Pre-training (unsupervised):
Text: "The cat sat on the ___"
Model learns to predict: "mat" (high probability)

Fine-tuning for medical Q&A (supervised):
Input: "What causes fever?"
Labeled good output: "Fever is caused by..."
Labeled bad output: "I don't know"
Model learns to prefer the informative response

RLHF (learning from preferences):
Generate 4 responses to "Explain quantum computing"
Human ranks them: Response C > Response A > Response D > Response B
Model learns to generate responses more like C and A

Hands-On Activity:

python

```python
# Simulate the difference between pre-training and fine-tuning

from transformers import GPT2LMHeadModel, GPT2Tokenizer

tokenizer = GPT2Tokenizer.from_pretrained('gpt2')
model = GPT2LMHeadModel.from_pretrained('gpt2')

# Pre-training objective: next token prediction
text = "The cat sat on the"
inputs = tokenizer(text, return_tensors='pt')

# Model predicts next token probabilities
outputs = model(**inputs)
next_token_logits = outputs.logits[0, -1, :]
next_token_probs = torch.softmax(next_token_logits, dim=-1)

# Get top 5 predictions
top_probs, top_indices = torch.topk(next_token_probs, 5)
print("Pre-training prediction (next token):")
for prob, idx in zip(top_probs, top_indices):
    print(f"  '{tokenizer.decode([idx])}': {prob:.3f}")

# Fine-tuning would adjust these probabilities based on your specific task
# For example, if fine-tuning for formal writing, it might increase
# probability of "sofa" and decrease "mat"
```

Precision, Recall, and F1

These metrics are fundamental for understanding model performance, especially in information retrieval and extraction tasks common in LLM applications.

Precision: Of all the things the model said were positive, how many actually were? Formula: True Positives / (True Positives + False Positives)

Recall: Of all the actual positives, how many did the model find? Formula: True Positives / (True Positives + False Negatives)

F1: The harmonic mean of precision and recall. Formula: 2 × (Precision × Recall) / (Precision + Recall)

Concrete Example: You're building an LLM to extract company names from news articles.

Article: "Apple and Microsoft announced a partnership. Google declined to comment."
Ground truth companies: {Apple, Microsoft, Google}

Scenario 1 - High Precision, Low Recall: Model extracts: {Apple, Microsoft}

- True Positives: 2 (Apple, Microsoft)
- False Positives: 0 (nothing incorrect)
- False Negatives: 1 (missed Google)
- Precision: 2/(2+0) = 100% (everything it found was correct)
- Recall: 2/(2+1) = 67% (it only found 2 out of 3 companies)
- F1: 2×(1.0×0.67)/(1.0+0.67) = 80%

Scenario 2 - Low Precision, High Recall: Model extracts: {Apple, Microsoft, Google, Amazon, partnership}

- True Positives: 3 (Apple, Microsoft, Google)
- False Positives: 2 (Amazon wasn't mentioned, partnership isn't a company)
- False Negatives: 0 (found all companies)
- Precision: 3/(3+2) = 60% (some errors)
- Recall: 3/(3+0) = 100% (found everything)
- F1: 2×(0.6×1.0)/(0.6+1.0) = 75%

When to Optimize for What:

- **Optimize for Precision**: When false positives are expensive (e.g., flagging content for legal review—you don't want to waste lawyers' time on false alarms)
- **Optimize for Recall**: When missing things is expensive (e.g., detecting fraud, finding all relevant documents for litigation)
- **Balance with F1**: When both matter equally (most general-purpose tasks)

Hands-On Activity:

```
python
```

```
from sklearn.metrics import precision_score, recall_score, f1_score

# Simulate an NER task: extracting person names from text
# Ground truth: which tokens are person names?
y_true = [1, 1, 0, 0, 1, 0, 0, 1, 1, 0]  # 1 = person name, 0 = not
# Person names were at positions: 0,1,4,7,8

# Model predictions (conservative model - high precision)
y_pred_conservative = [1, 1, 0, 0, 0, 0, 0, 1, 0, 0]
# Found: 0,1,7  Missed: 4,8

precision_cons = precision_score(y_true, y_pred_conservative)
recall_cons = recall_score(y_true, y_pred_conservative)
f1_cons = f1_score(y_true, y_pred_conservative)

print("Conservative Model (careful, might miss things):")
print(f"  Precision: {precision_cons:.2f} (high - rarely wrong)")
print(f"  Recall: {recall_cons:.2f} (low - missed some names)")
print(f"  F1: {f1_cons:.2f}\n")

# Aggressive model (tries to find everything)
y_pred_aggressive = [1, 1, 1, 0, 1, 1, 0, 1, 1, 0]
# Found: 0,1,2,4,5,7,8  (includes 2 false positives: 2,5)

precision_agg = precision_score(y_true, y_pred_aggressive)
recall_agg = recall_score(y_true, y_pred_aggressive)
f1_agg = f1_score(y_true, y_pred_aggressive)

print("Aggressive Model (tries to find everything):")
print(f"  Precision: {precision_agg:.2f} (lower - some mistakes)")
print(f"  Recall: {recall_agg:.2f} (high - found everything!)")
print(f"  F1: {f1_agg:.2f}\n")

# Real interview question simulation
print("Interview Question: Which model would you deploy for:")
print("1. Medical diagnosis assistant? → Conservative (precision matters)")
print("2. Resume screening tool? → Aggressive (recall matters - don't miss candidates)")
print("3. General entity extraction? → Balanced (F1 optimization)")
```

Overfitting and Generalization

Overfitting is when a model memorizes training data but fails on new, unseen examples. This is particularly dangerous with LLMs because they can memorize vast amounts of training data.

Concrete Example: You fine-tune GPT-4 on 1,000 customer support conversations from your company. The model gets 95% accuracy on those exact conversations. But when you deploy it:

- On new customer questions, it only gets 70% accuracy
- It sometimes quotes exact phrases from the training conversations even when they don't fit
- It struggles with questions that are similar but not identical to training examples

This is overfitting. The model memorized specific conversations instead of learning general principles of good customer support.

How to Detect Overfitting in LLMs:

1. **Train/Test Split**: Performance is much better on training data than held-out test data
2. **Exact Memorization**: Model reproduces training examples verbatim
3. **Poor Transfer**: Model fails on slightly different domains or phrasings
4. **Overconfidence**: Model is very confident even when wrong

How to Prevent Overfitting:

1. **More diverse training data**: Don't fine-tune on too narrow a dataset
2. **Regularization**: Techniques like dropout, weight decay
3. **Early stopping**: Stop training when validation performance plateaus
4. **Parameter-efficient fine-tuning (PEFT)**: Use LoRA or adapters to update fewer parameters

Hands-On Activity:

```python
```

```
# Simulate overfitting detection

# Training accuracy over epochs
train_acc = [0.65, 0.78, 0.85, 0.91, 0.95, 0.97, 0.98, 0.99]
val_acc =   [0.64, 0.76, 0.83, 0.85, 0.85, 0.84, 0.83, 0.82]

import matplotlib.pyplot as plt

plt.figure(figsize=(10, 6))
plt.plot(range(1, 9), train_acc, marker='o', label='Training Accuracy', linewidth=2)
plt.plot(range(1, 9), val_acc, marker='s', label='Validation Accuracy', linewidth=2)
plt.xlabel('Epoch', fontsize=12)
plt.ylabel('Accuracy', fontsize=12)
plt.title('Detecting Overfitting: Training vs Validation Performance', fontsize=14)
plt.legend(fontsize=11)
plt.grid(True, alpha=0.3)
plt.axvline(x=4, color='red', linestyle='--', label='Optimal stopping point')

# Annotate the overfitting region
plt.annotate('Overfitting starts here', xy=(5, 0.84), xytext=(6, 0.75),
             arrowprops=dict(arrowstyle='->', color='red', lw=2),
             fontsize=11, color='red')

plt.tight_layout()
plt.show()

print("Analysis:")
print("- Epochs 1-4: Both metrics improve together (healthy learning)")
print("- Epoch 4: Peak validation performance (85%)")
print("- Epochs 5-8: Training keeps improving, validation declines")
print("- Conclusion: Stop training at epoch 4 to prevent overfitting")
print("\nIn production: Set up early stopping to automatically halt training")
```

Real Interview Question: "You fine-tuned an LLM and it performs perfectly on your test set but poorly in production. What could be wrong?"

Potential answers:

1. **Test set is too similar to training set** (data leakage)
2. **Production data has different distribution** (domain shift)
3. **Test set is too small** (lucky performance)
4. **Overfitting**: Model memorized test patterns

5. **Prompt format changed** between testing and production

Deep Learning Fundamentals

Transformers Architecture

Transformers are the architecture behind all modern LLMs (GPT, Claude, Llama, etc.). Understanding how they work is essential for understanding their evaluation challenges.

Self-Attention Mechanism: This is the core innovation. Instead of processing text sequentially (like older RNNs), transformers look at all words simultaneously and learn which words should "pay attention" to which other words.

Concrete Example: Consider the sentence: "The animal didn't cross the street because it was too tired."

What does "it" refer to? The animal or the street?

A transformer uses self-attention to figure this out:

1. The word "it" creates queries to look for relevant context
2. Other words ("animal", "street", "tired") provide keys and values
3. Attention scores determine relevance: "animal" and "tired" get high attention, "street" gets low attention
4. The model combines information: "it" = animal (because animals get tired, streets don't)

The attention mechanism learns these associations from data. In simpler terms: attention is how the model decides which words are relevant to understanding each other word.

Why This Matters for Evaluation:

- Transformers can capture long-range dependencies (understanding "it" refers to "animal" even with words in between)
- But they have context length limits (typically 4K-128K tokens)
- Attention can be visualized to debug why a model made a decision
- Some evaluation metrics test whether models properly track references across long contexts

Positional Embeddings: Transformers don't inherently know word order. "Dog bites man" vs. "Man bites dog" would look the same without positional information. Positional embeddings add information about each word's position in the sequence.

Hands-On Activity:

python

```python
from transformers import AutoTokenizer, AutoModel
import torch

tokenizer = AutoTokenizer.from_pretrained('bert-base-uncased')
model = AutoModel.from_pretrained('bert-base-uncased', output_attentions=True)

# Example with pronoun resolution
text = "The animal didn't cross the street because it was too tired"
inputs = tokenizer(text, return_tensors='pt')

# Get attention weights
outputs = model(**inputs)
attention = outputs.attentions  # Tuple of attention matrices, one per layer

# Look at attention in the last layer
last_layer_attention = attention[-1]  # Shape: [batch, heads, seq_len, seq_len]

# Find the token position of "it"
tokens = tokenizer.convert_ids_to_tokens(inputs['input_ids'][0])
it_position = tokens.index('it')

# See what "it" attends to (average across all attention heads)
it_attention = last_layer_attention[0, :, it_position, :].mean(dim=0)

print(f"What does 'it' pay attention to?\n")
for token, attention_score in zip(tokens, it_attention):
    if attention_score > 0.05:  # Only show significant attention
        print(f"  {token}: {attention_score:.3f}")

print("\nExpected: 'animal' and 'tired' should have high attention")
print("This shows the model correctly resolves the pronoun!")
```

Pre-training vs Fine-tuning vs PEFT

Understanding these training approaches is crucial because they affect how you evaluate models.

Pre-training: Training a model from scratch on massive amounts of text to learn general language understanding. This is expensive (millions of dollars, thousands of GPUs) and only done by large organizations.

Example: GPT-4 was pre-trained on trillions of tokens from the internet, books, code, etc. It learned grammar, facts, reasoning patterns, and even some biases present in the training data.

Fine-tuning: Taking a pre-trained model and training it further on a specific task or domain. This is much cheaper and is what most practitioners do.

Example: You take GPT-3.5 and fine-tune it on 10,000 examples of customer support conversations from your company. The model adapts its general language abilities to your specific use case.

PEFT (Parameter-Efficient Fine-Tuning): Instead of updating all billions of parameters during fine-tuning (expensive and risks overfitting), PEFT methods only update a small subset.

LoRA (Low-Rank Adaptation): Adds small "adapter" matrices alongside frozen model weights. You only train the small adapters (~1% of parameters) while keeping the original model frozen.

Why This Matters for Evaluation:

- Pre-trained models need general benchmarks (MMLU, HellaSwag)
- Fine-tuned models need task-specific evaluation
- PEFT models need to be tested for both: did they learn the new task AND retain general capabilities?

Concrete Example:

```
Base Model (GPT-3.5): General purpose, can do many tasks okay
↓ Fine-tune on medical Q&A
Specialized Medical Model: Great at medical questions, but might forget some general knowledge
↓ Use LoRA instead
LoRA Medical Model: Great at medical questions AND retains general capabilities

Evaluation must test:
1. Medical accuracy (did fine-tuning work?)
2. General knowledge (did we catastrophically forget?)
3. Safety (did fine-tuning break guardrails?)
```

Hands-On Activity:

```python
```

```python
# Conceptual comparison of fine-tuning approaches

class ModelEvaluation:
    def __init__(self, model_name):
        self.model_name = model_name

    def evaluate(self, task):
        # Simulate evaluation on different tasks
        scores = {
            'base_model': {
                'general_qa': 75,
                'medical_qa': 45,
                'coding': 70,
                'safety': 90
            },
            'full_finetune': {
                'general_qa': 65,  # Dropped!
                'medical_qa': 90,  # Improved!
                'coding': 60,      # Dropped!
                'safety': 85       # Slightly dropped
            },
            'lora_finetune': {
                'general_qa': 73,  # Mostly retained
                'medical_qa': 88,  # Improved!
                'coding': 68,      # Mostly retained
                'safety': 89       # Retained
            }
        }
        return scores.get(self.model_name, {}).get(task, 0)

# Compare approaches
models = ['base_model', 'full_finetune', 'lora_finetune']
tasks = ['general_qa', 'medical_qa', 'coding', 'safety']

print("Performance Comparison:\n")
print(f"{'Task':<15} {'Base Model':<15} {'Full Fine-tune':<15} {'LoRA Fine-tune':<15}")
print("-" * 65)

for task in tasks:
    scores = [ModelEvaluation(model).evaluate(task) for model in models]
    print(f"{task:<15} {scores[0]:<15} {scores[1]:<15} {scores[2]:<15}")

print("\nKey Insight:")
```

```
print("✓ LoRA achieves most of the task-specific gains")
print("✓ LoRA retains general capabilities better than full fine-tuning")
print("✓ Full fine-tuning risks catastrophic forgetting")
print("→ In production: Use LoRA for most fine-tuning tasks")
```

Tokenization

Tokenization is how text gets converted into numbers that models can process. This seemingly simple step has major implications for evaluation.

Subword Tokenization: Modern LLMs don't work with whole words. They break text into subword units. Common algorithms: BPE (Byte-Pair Encoding), WordPiece, SentencePiece.

Concrete Example:

```
Text: "The unhappiness is unbearable"

Word-level tokenization (old approach):
["The", "unhappiness", "is", "unbearable"]
Problem: Vocabulary explodes with all word variants

Subword tokenization (modern approach):
["The", "un", "happiness", "is", "un", "bear", "able"]
Benefit: Model learns "un-" is a negation prefix, "able" is a suffix
```

Why This Matters for Evaluation:

1. **Token counts affect costs**: GPT-4 charges per token. "Internationalization" is 6 tokens, "I18n" is 3 tokens. Same meaning, different cost.
2. **Different tokenizers handle languages differently**: English text might be 1.3 tokens per word, while Chinese might be 2.5 tokens per character. Evaluation metrics based on token counts can be unfair across languages.
3. **Rare words get split more**: "Antidisestablishmentarianism" becomes many tokens. The model sees it as separate pieces, not as a whole word. This affects performance on specialized vocabulary.
4. **Token limits are hard boundaries**: If your context limit is 4,096 tokens, that might be 3,000 English words or 1,500 Chinese characters.

Hands-On Activity:

```
python
```

```python
from transformers import AutoTokenizer

# Compare different tokenizers
tokenizers = {
    'GPT-2': AutoTokenizer.from_pretrained('gpt2'),
    'GPT-4 (cl100k)': AutoTokenizer.from_pretrained('Xenova/gpt-4'),  # approximation
    'Llama-2': AutoTokenizer.from_pretrained('meta-llama/Llama-2-7b-hf')
}

test_texts = [
    "Hello world",
    "Internationalization",
    "你好世界",  # Chinese: Hello world
    "The unhappiness is unbearable",
    "AI/ML is transforming industries"
]

print("Token Count Comparison:\n")
print(f"{'Text':<40} {'GPT-2':<10} {'GPT-4':<10} {'Llama-2':<10}")
print("-" * 75)

for text in test_texts:
    counts = {}
    for name, tokenizer in tokenizers.items():
        try:
            tokens = tokenizer.encode(text)
            counts[name] = len(tokens)
        except:
            counts[name] = "N/A"

    print(f"{text:<40} {counts.get('GPT-2', 'N/A'):<10} {counts.get('GPT-4 (cl100k)', 'N/A'):<10} {counts.get('Llama-2', 'N/A')

# Show actual tokens for one example
print("\n\nDetailed Tokenization Example:")
text = "The unhappiness is unbearable"
print(f"Text: '{text}'\n")

for name, tokenizer in tokenizers.items():
    try:
        tokens = tokenizer.tokenize(text)
        print(f"{name}:")
        print(f"  Tokens: {tokens}")
        print(f"  Count: {len(tokens)}\n")
```

```
    except:
        print(f"{name}: Not available\n")

print("Key Observations:")
print("1. Different tokenizers split text differently")
print("2. This affects context length calculations")
print("3. Multilingual text often uses more tokens")
print("4. When evaluating, normalize by tokens, not words!")
```

Interview Question: "Why does GPT-4 seem to perform better on English than other languages?"

Potential answer: "Multiple factors, but tokenization is significant. English text is tokenized more efficiently—each token captures more semantic information. For languages like Thai or Japanese, a single concept might require 3-4x more tokens, effectively giving the model less 'thinking space' within the same context window. When evaluating multilingual models, we need to account for tokenization efficiency, not just raw performance metrics."

Software & Infrastructure

Understanding the technical stack is essential because evaluation doesn't happen in isolation—it's part of a production system.

Python Ecosystem for LLMs

NumPy & Pandas: Core libraries for numerical computing and data manipulation. You'll use these constantly for analyzing evaluation results, computing metrics, and processing datasets.

Concrete Example: After running evaluations, you need to analyze results:

```python
```

```python
import pandas as pd
import numpy as np

# Evaluation results from 1000 test cases
results = pd.DataFrame({
    'prompt_id': range(1000),
    'model_a_score': np.random.normal(0.75, 0.15, 1000),
    'model_b_score': np.random.normal(0.78, 0.15, 1000),
    'latency_ms': np.random.gamma(2, 50, 1000),
    'category': np.random.choice(['factual', 'creative', 'reasoning'], 1000)
})

# Analysis tasks you'll do regularly
print("Overall Performance:")
print(results[['model_a_score', 'model_b_score']].describe())

print("\n\nPerformance by Category:")
category_perf = results.groupby('category')[['model_a_score', 'model_b_score']].mean()
print(category_perf)

# Identify where Model B significantly outperforms Model A
results['improvement'] = results['model_b_score'] - results['model_a_score']
significant_improvements = results[results['improvement'] > 0.2]
print(f"\n\nCases with >0.2 improvement: {len(significant_improvements)}")

# Check if high latency correlates with better scores
correlation = results['latency_ms'].corr(results['model_b_score'])
print(f"\nCorrelation between latency and performance: {correlation:.3f}")
```

PyTorch & TensorFlow: Deep learning frameworks. You'll use these to load models, run inference, and extract internal states for evaluation.

Hands-On Activity:

```python
```

```python
import torch
from transformers import AutoModelForCausalLM, AutoTokenizer

# Load a model for evaluation
model_name = "gpt2"
tokenizer = AutoTokenizer.from_pretrained(model_name)
model = AutoModelForCausalLM.from_pretrained(model_name)

# Evaluation task: measure perplexity on test sentences
test_sentences = [
    "The capital of France is Paris.",
    "The capital of France is London.",  # Factually incorrect
    "Colorless green ideas sleep furiously."  # Grammatically correct but nonsensical
]

def calculate_perplexity(text, model, tokenizer):
    """
    Lower perplexity = model is more confident/less surprised
    Higher perplexity = model is confused/uncertain
    """
    encodings = tokenizer(text, return_tensors='pt')

    with torch.no_grad():
        outputs = model(**encodings, labels=encodings['input_ids'])
        loss = outputs.loss
        perplexity = torch.exp(loss)

    return perplexity.item()

print("Perplexity Analysis (lower = more confident):\n")
for sentence in test_sentences:
    ppl = calculate_perplexity(sentence, model, tokenizer)
    print(f"Sentence: {sentence}")
    print(f"Perplexity: {ppl:.2f}\n")

print("Interpretation:")
print("✓ Factually correct sentence: Low perplexity")
print("✗ Factually incorrect sentence: Higher perplexity (model is 'surprised')")
print("? Nonsensical but grammatical: Moderate perplexity")
```

FastAPI, Docker, Kubernetes

These are the standard tools for deploying LLMs in production. Understanding them is crucial for production evaluation.

FastAPI: Python web framework for building APIs. You'll wrap your model in a FastAPI endpoint so applications can query it.

Docker: Containerization tool. Packages your model, code, and dependencies into a single container that runs consistently everywhere.

Kubernetes (K8s): Orchestration platform. Manages multiple containers, handles scaling, load balancing, and rollbacks.

Why This Matters for Evaluation:

- Evaluation must happen at each stage: development, staging, production
- You need automated evaluation pipelines that run on every deployment
- Production evaluation metrics (latency, throughput) differ from research metrics (accuracy)

Concrete Example - Deployment Pipeline with Evaluation Gates:

```
Development:
├── Train/fine-tune model
├── Run offline evaluation suite
│   ├── Accuracy benchmarks
│   ├── Safety tests
│   └── Hallucination detection
└── If pass → Package in Docker

Staging:
├── Deploy to staging K8s cluster
├── Run integration tests
│   ├── API response times
│   ├── Concurrent request handling
│   └── Error rate monitoring
├── Shadow production traffic (10%)
└── If pass → Deploy to production

Production:
├── Blue-green deployment (gradual rollout)
├── Continuous monitoring
│   ├── Real-time performance metrics
│   ├── User feedback collection
│   └── Drift detection
└── Automated rollback if metrics degrade
```

Hands-On Activity:

python

```python
# Simple FastAPI wrapper for a model (evaluation endpoint)
from fastapi import FastAPI
from pydantic import BaseModel
import time

app = FastAPI()

class EvaluationRequest(BaseModel):
    prompt: str
    ground_truth: str

class EvaluationResponse(BaseModel):
    prediction: str
    latency_ms: float
    confidence: float

@app.post("/evaluate")
async def evaluate_model(request: EvaluationRequest):
    start_time = time.time()

    # Simulate model inference
    prediction = generate_response(request.prompt)

    latency = (time.time() - start_time) * 1000

    # Calculate metrics
    confidence = calculate_confidence(prediction)

    return EvaluationResponse(
        prediction=prediction,
        latency_ms=latency,
        confidence=confidence
    )

@app.post("/batch_evaluate")
async def batch_evaluate(requests: list[EvaluationRequest]):
    """
    Batch evaluation for efficiency
    Critical for large-scale testing
    """
    results = []
    for req in requests:
        result = await evaluate_model(req)
```

```
        results.append(result)
    return results

# In production, you'd add:
# - Authentication
# - Rate limiting
# - Logging/monitoring
# - Error handling
# - Evaluation result storage
```

Vector Databases

Vector databases store embeddings and enable semantic search. Essential for RAG (Retrieval-Augmented Generation) systems.

Common Options:

- **FAISS**: Facebook's library, fast but in-memory
- **Milvus**: Distributed vector database, production-grade
- **Weaviate**: Vector database with built-in ML models
- **Pinecone**: Managed vector database service

Why This Matters for Evaluation:

- RAG system quality depends heavily on retrieval quality
- You need to evaluate: retrieval accuracy, latency, and relevance
- Vector database performance affects overall system performance

Concrete Example - RAG Evaluation:

```
python
```

```python
from sentence_transformers import SentenceTransformer
import faiss
import numpy as np

# Load embedding model
embedding_model = SentenceTransformer('all-MiniLM-L6-v2')

# Sample knowledge base
knowledge_base = [
    "Paris is the capital of France.",
    "The Eiffel Tower is located in Paris.",
    "France is a country in Europe.",
    "London is the capital of the United Kingdom.",
    "The Seine river flows through Paris."
]

# Create embeddings
embeddings = embedding_model.encode(knowledge_base)
dimension = embeddings.shape[1]

# Build FAISS index
index = faiss.IndexFlatL2(dimension)
index.add(embeddings.astype('float32'))

# Evaluation: Retrieval accuracy
test_queries = [
    {
        'query': "What is the capital of France?",
        'relevant_docs': [0, 1, 4]  # Indices of relevant documents
    },
    {
        'query': "Tell me about rivers in France",
        'relevant_docs': [4]
    }
]

def evaluate_retrieval(query, relevant_docs, k=3):
    """
    Evaluate retrieval quality using Recall@k
    """
    # Embed query
    query_embedding = embedding_model.encode([query])
```

```
    # Search
    distances, indices = index.search(query_embedding.astype('float32'), k)
    retrieved_docs = indices[0].tolist()

    # Calculate Recall@k
    relevant_retrieved = len(set(retrieved_docs) & set(relevant_docs))
    recall = relevant_retrieved / len(relevant_docs)

    return recall, retrieved_docs

print("RAG Retrieval Evaluation:\n")
for test in test_queries:
    recall, retrieved = evaluate_retrieval(test['query'], test['relevant_docs'])
    print(f"Query: {test['query']}")
    print(f"Retrieved docs: {[knowledge_base[i] for i in retrieved]}")
    print(f"Recall@3: {recall:.2f}")
    print(f"Expected relevant docs: {[knowledge_base[i] for i in test['relevant_docs']]}\n")
```

Experiment Tracking

Weights & Biases (wandb) and **MLflow**: Tools for tracking experiments, comparing models, and managing evaluation results.

Why This Matters: When you run hundreds of experiments, you need systematic tracking to know what works.

Hands-On Activity:

python

```python
import mlflow

# Start an experiment
mlflow.set_experiment("llm_evaluation")

with mlflow.start_run():
    # Log model parameters
    mlflow.log_param("model_name", "gpt-3.5-turbo")
    mlflow.log_param("temperature", 0.7)
    mlflow.log_param("max_tokens", 150)

    # Simulate evaluation
    accuracy = 0.85
    f1_score = 0.82
    hallucination_rate = 0.12
    avg_latency = 450  # ms

    # Log metrics
    mlflow.log_metric("accuracy", accuracy)
    mlflow.log_metric("f1_score", f1_score)
    mlflow.log_metric("hallucination_rate", hallucination_rate)
    mlflow.log_metric("avg_latency_ms", avg_latency)

    # Log artifacts (eval results, sample outputs)
    with open("eval_results.txt", "w") as f:
        f.write(f"Accuracy: {accuracy}\nF1: {f1_score}")
    mlflow.log_artifact("eval_results.txt")

    print("Experiment logged successfully!")
    print("Use 'mlflow ui' to view results in web interface")

# Compare multiple runs
print("\nBest practice: Log every evaluation run")
print("Then compare across:")
print("- Different models")
print("- Different hyperparameters")
print("- Different evaluation datasets")
print("- Different time periods (detect drift)")
```

1. Understanding LLM Behavior

Before you can evaluate an LLM, you need to understand how it works internally. This section covers the core concepts that drive LLM behavior.

Autoregressive Language Modeling

LLMs generate text one token at a time, using previously generated tokens to predict the next one. This is called "autoregressive" because the model uses its own outputs as inputs for future predictions.

Concrete Example:

```
Prompt: "The capital of France is"

Step 1: Model sees "The capital of France is"
    Predicts next token: "Paris" (0.85 prob)
    Generates: "Paris"

Step 2: Model sees "The capital of France is Paris"
    Predicts next token: "." (0.75 prob)
    Generates: "."

Step 3: Model sees "The capital of France is Paris."
    Predicts next token: <END> or continues
```

Why This Matters for Evaluation:

- Errors compound: One wrong token can derail entire generation
- Early tokens heavily influence later ones
- Evaluation at token-level vs. sequence-level gives different insights

Log-Likelihood: The model assigns a probability to each token. Log-likelihood is the log of that probability. Higher (less negative) log-likelihood means the model is more confident.

Hands-On Activity:

```
python
```

```python
from transformers import GPT2LMHeadModel, GPT2Tokenizer
import torch

model = GPT2LMHeadModel.from_pretrained('gpt2')
tokenizer = GPT2Tokenizer.from_pretrained('gpt2')

def analyze_generation_step_by_step(prompt, max_tokens=5):
    """
    Show exactly how autoregressive generation works
    """
    input_ids = tokenizer.encode(prompt, return_tensors='pt')

    print(f"Starting prompt: '{prompt}'\n")
    print("Generation process:")
    print("-" * 60)

    for step in range(max_tokens):
        # Get model predictions
        with torch.no_grad():
            outputs = model(input_ids)
            next_token_logits = outputs.logits[0, -1, :]

        # Get probabilities
        probs = torch.softmax(next_token_logits, dim=-1)

        # Get top 5 most likely next tokens
        top_probs, top_indices = torch.topk(probs, 5)

        print(f"\nStep {step + 1}:")
        print(f"Current text: '{tokenizer.decode(input_ids[0])}'")
        print(f"Top 5 next token predictions:")
        for prob, idx in zip(top_probs, top_indices):
            token = tokenizer.decode([idx])
            print(f"  '{token}': {prob:.4f}")

        # Sample next token (greedy: pick most likely)
        next_token = top_indices[0].unsqueeze(0).unsqueeze(0)
        input_ids = torch.cat([input_ids, next_token], dim=1)

        # Show what was chosen
        chosen_token = tokenizer.decode([top_indices[0]])
        print(f"  → Chose: '{chosen_token}'")
```

```
    final_text = tokenizer.decode(input_ids[0])
    print(f"\n{'='*60}")
    print(f'Final generated text: "{final_text}"')
    return final_text

# Try it
analyze_generation_step_by_step("The weather today is", max_tokens=3)
```

Sampling Strategies

How the model chooses the next token dramatically affects output quality. Understanding sampling is crucial for evaluation because different strategies produce very different outputs.

Greedy Sampling: Always pick the most likely token. Deterministic but can be repetitive.

Temperature Sampling: Adjust the probability distribution before sampling.

- Low temperature (0.1-0.5): Sharper distribution, more confident/focused outputs
- High temperature (1.5-2.0): Flatter distribution, more random/creative outputs

Top-k Sampling: Only consider the k most likely tokens, redistribute probability among them.

Top-p (Nucleus) Sampling: Consider the smallest set of tokens whose cumulative probability exceeds p. More dynamic than top-k.

Concrete Example:

```
Next token probabilities:
"sunny": 0.40
"cloudy": 0.30
"rainy": 0.20
"snowy": 0.05
"foggy": 0.03
...other tokens: 0.02

Greedy: Always picks "sunny"

Temperature = 0.5 (confident):
"sunny": 0.55
"cloudy": 0.28
"rainy": 0.15
... (distribution sharpens toward top choice)

Temperature = 2.0 (creative):
"sunny": 0.28
"cloudy": 0.26
"rainy": 0.24
"snowy": 0.12
... (distribution flattens, more variety)

Top-k = 2:
Only consider "sunny" (0.57) and "cloudy" (0.43)
Ignore all others

Top-p = 0.8:
"sunny" (0.40) + "cloudy" (0.30) + "rainy" (0.20) = 0.90 > 0.80
So consider these 3 tokens only
```

Why This Matters for Evaluation:

- Factual tasks need low temperature (deterministic, accurate)
- Creative tasks need higher temperature (varied, interesting)
- Evaluation should match deployment sampling strategy
- Same model can behave very differently with different sampling

Hands-On Activity:

```
python
```

```
def compare_sampling_strategies(prompt, strategies):
    """
    Generate with different sampling strategies and compare
    """
    model = GPT2LMHeadModel.from_pretrained('gpt2')
    tokenizer = GPT2Tokenizer.from_pretrained('gpt2')

    inputs = tokenizer(prompt, return_tensors='pt')

    print(f"Prompt: '{prompt}'\n")
    print("Outputs with different sampling strategies:")
    print("="*70)

    for name, params in strategies.items():
        output = model.generate(
            **inputs,
            max_length=50,
            num_return_sequences=1,
            pad_token_id=tokenizer.eos_token_id,
            **params
        )

        text = tokenizer.decode(output[0], skip_special_tokens=True)
        print(f"\n{name}:")
        print(f"  {text}")

    return

# Define strategies to compare
strategies = {
    "Greedy (deterministic)": {
        "do_sample": False
    },
    "Low temperature (focused)": {
        "do_sample": True,
        "temperature": 0.3
    },
    "High temperature (creative)": {
        "do_sample": True,
        "temperature": 1.5
    },
    "Top-k=10": {
        "do_sample": True,
```

```
        "top_k": 10
    },
    "Top-p=0.9 (nucleus)": {
        "do_sample": True,
        "top_p": 0.9
    }
}

compare_sampling_strategies(
    "Once upon a time in a distant galaxy",
    strategies
)

print("\n\nKey Insight for Evaluation:")
print("✓ Factual Q&A → Use low temperature or greedy")
print("✓ Creative writing → Use high temperature or top-p")
print("✓ Always report sampling params with evaluation results!")
```

Context Length and Attention

Models have limited context windows (how much text they can "see" at once). This fundamentally affects performance.

Context Windows:

- GPT-3.5: 4K tokens (~3,000 words)
- GPT-4: 8K-32K tokens
- Claude: 100K tokens
- GPT-4 Turbo: 128K tokens

Why This Matters for Evaluation:

- Performance often degrades with longer contexts
- Models might "forget" information from early in the context
- Evaluation must test at realistic context lengths

Attention Bottlenecks: Transformers use self-attention, which has $O(n^2)$ complexity. As context grows, attention becomes the bottleneck.

Concrete Example:

Short context (fits in window):
Query: "What did Mary say about the party?"
Context: "Mary said the party was fun. John agreed."
Result: ✓ Perfect recall

Long context (tests limits):
Query: "What did Mary say about the party?"
Context: [20,000 words of story, Mary's comment buried in middle]
Result: ✗ Model might miss it or hallucinate

This is "lost in the middle" problem - information in the middle of long contexts is often forgotten.

Hands-On Activity:

```
python
```

```python
def test_context_length_performance():
    """
    Test how model performance degrades with context length
    """
    from transformers import pipeline

    qa_pipeline = pipeline("question-answering", model="distilbert-base-uncased-distilled-squad")

    # The key information
    key_info = "The secret password is BLUE ELEPHANT."

    # Test with different amounts of distractor text
    contexts = {
        "Short context (50 words)": key_info + " " + "The weather is nice today. " * 10,
        "Medium context (200 words)": "The weather is nice today. " * 40 + key_info + " The sky is clear. " * 40,
        "Long context (500 words)": "The weather is nice today. " * 100 + key_info + " The sky is clear. " * 100
    }

    question = "What is the secret password?"

    print("Testing Context Length Effects:")
    print("="*60)

    for name, context in contexts.items():
        result = qa_pipeline(question=question, context=context)
        print(f"\n{name}:")
        print(f"  Context length: {len(context.split())} words")
        print(f"  Answer: {result['answer']}")
        print(f"  Confidence: {result['score']:.3f}")

    print("\n\nKey Insight:")
    print("As context grows, model confidence often drops")
    print("Information buried in middle is hardest to retrieve")

test_context_length_performance()
```

This covers the foundational prerequisites and LLM behavior section with detailed explanations, concrete examples, and hands-on activities. Would you like me to continue with the remaining sections (Classical NLP Evaluation, Generation Quality, RAG Systems, etc.)?