



A Deep Dive into Large Language Model Hallucinations: Causes, Measurement, and Mitigation

This guide provides a comprehensive overview of the phenomenon of "hallucination" in Large Language Models (LLMs). We will explore its definition, types, and underlying causes. We will then delve into methods for measuring and evaluating hallucinations, followed by a detailed examination of state-of-the-art mitigation strategies, including external knowledge verification, advanced decoding techniques, and self-consistency checks. This document is designed to be a thorough resource for data scientists and machine learning engineers preparing for interviews that cover the practical challenges and theoretical underpinnings of LLMs.

Knowledge Section

1. What is an LLM Hallucination?

In the context of Large Language Models, a **hallucination** refers to the model generating text that is nonsensical, factually incorrect, or disconnected from the provided source context, yet is presented with fluent and confident language. It is colloquially described as the model "confidently making things up."

The presence of hallucinations is a significant barrier to the reliable deployment of LLMs in critical applications. For example, a hallucinated medical diagnosis could have life-threatening consequences, and hallucinated financial advice could lead to severe losses. Addressing this issue is paramount for building trust and ensuring the safety and reliability of LLM-powered systems.

2. Is Hallucination Always Detrimental?

While often problematic, hallucination is **not universally harmful**. In creative endeavors, such as drafting a movie script, writing poetry, or brainstorming fictional ideas, hallucinations can be a source of novelty and imagination. The model's ability to generate unexpected and

imaginative content can be a powerful tool for creative professionals.

Therefore, the tolerance for hallucination is highly **application-dependent**. For factual tasks like question-answering or document summarization, the goal is to minimize hallucinations. For creative tasks, a controlled degree of hallucination might be desirable.

3. A Typology of Hallucinations

Hallucinations can be broadly classified into two main categories based on their relationship to a given source context (e.g., a document for summarization).

- **Intrinsic Hallucination:** The generated content directly contradicts the information present in the source material. This is a clear case of factual inconsistency with the provided context.
- **Extrinsic Hallucination:** The generated content cannot be verified by the source material. It introduces new information that is neither supported nor contradicted by the source. This new information could be factually correct in the real world but is still considered a hallucination in a context-grounded task because it deviates from the source.

4. Core Causes of LLM Hallucinations

The tendency for LLMs to hallucinate stems from multiple factors inherent in their training data, model architecture, and inference processes.

4.1. Data-Related Causes

1. **Source-Target Divergence:** The training data often contains pairs of source and target documents where the target is not a strict subset or entailment of the source. This can be unintentional (e.g., summarizing a news event using two different articles with slightly different details) or intentional (e.g., in abstractive summarization, where conciseness is prioritized over literal faithfulness). The model learns these patterns of divergence.
2. **Data Noise and Factuality Errors:** The vast pre-training corpora (like the Common Crawl) are filled with noise, misinformation, and biases. The model inevitably learns from this "dirty" data, internalizing incorrect facts or flawed reasoning patterns.

3. **Repetitive Data:** Over-representation of certain phrases or sequences in the training data can cause the model to develop a high probability of generating these phrases, even when they are not contextually appropriate, leading to hallucinations.

4.2. Model and Training-Related Causes

1. **Parametric Knowledge Bias:** LLMs store a vast amount of factual knowledge learned during pre-training directly within their parameters. Research has shown that models can sometimes over-rely on this "parametric knowledge" instead of the specific context provided in the prompt. If the prompt's context conflicts with the model's stored knowledge, it may hallucinate an answer consistent with its training rather than the given context.
2. **Training-Inference Mismatch (Exposure Bias):** During standard training (teacher forcing), the model predicts the next token based on a ground-truth sequence. However, during inference, it predicts the next token based on its *own* previously generated tokens. An early error can cascade, leading the model down a path of increasingly divergent and nonsensical generation, especially in long sequences.

4.3. Decoding and Inference-Related Causes

1. **Stochastic Decoding Strategies:** To avoid repetitive and deterministic outputs, we use decoding strategies that introduce randomness.
 - **Temperature Sampling:** Rescales the logits before applying the softmax function. A higher temperature flattens the probability distribution, increasing the chance of sampling lower-probability tokens and thus enhancing creativity at the risk of hallucination. The probability p_i of a token i with logit z_i is given by:

$$p_i = \frac{\exp(z_i/T)}{\sum_j \exp(z_j/T)}$$

where T is the temperature. As $T \rightarrow 0$, the sampling becomes greedy (argmax). As $T \rightarrow \infty$, it approaches a uniform distribution.

- **Top-k Sampling:** Restricts the sampling pool to the **k** most

probable tokens. This prevents very low-probability tokens from being chosen but can be too restrictive if the true probability mass is spread across many tokens.

- **Nucleus (Top-p) Sampling:** Restricts the sampling pool to the smallest set of tokens whose cumulative probability is greater than or equal to a nucleus probability p . This is adaptive; it can select a large number of tokens when the distribution is flat (high uncertainty) and a small number when it is sharp (high confidence).

While these strategies improve output diversity, they inherently increase the risk of selecting a factually incorrect or contextually inappropriate token.

5. How to Measure and Evaluate Hallucinations

Reliably measuring hallucinations is a significant challenge. Human evaluation remains the gold standard but is expensive and slow. Consequently, several automated metrics have been developed.

- **Named Entity (NE) Based Metrics:** The core idea is that factual statements rely on key named entities (people, places, organizations). We can compare the named entities in the generated text with those in the reference text. A generated entity not present in the source is a potential hallucination.
- **Natural Language Inference (NLI) Based Metrics:** NLI models are trained to determine if a "hypothesis" is entailed by, contradicts, or is neutral with respect to a "premise." We can treat the generated sentence as the hypothesis and the source document as the premise to calculate an "entailment ratio" (the proportion of generated sentences entailed by the source).
- **Question-Answering (QA) Based Evaluation:** If a generated summary is factually consistent with a source document, then a question-answering model should produce similar answers when asked the same question about both texts. The process involves generating questions from the summary, answering them using both the summary and the source document, and then comparing the answers for similarity.
- **Information Extraction (IE) Based Methods:** This approach uses IE models to convert unstructured text into structured knowledge tuples (e.g., `<Subject, Relation, Object>`). By comparing the set of tuples extracted from the generated text with those from the source, we can quantify the degree of

factual overlap and identify hallucinated relations.

- **Model-Based Evaluation:** Increasingly, strong LLMs themselves are being used as evaluators. By crafting a specific prompt, one can ask a model like GPT-4 to rate the factuality of a generated text against a source document.

6. Strategies for Mitigating LLM Hallucinations

Mitigation strategies can be broadly categorized into data-centric, model-centric, and inference-time approaches.

6.1. Improving Data Quality

The most fundamental solution is to improve the quality of the training data. Creating large, high-quality, and noise-free datasets is incredibly challenging and expensive but remains a key area of research. This involves sophisticated data cleaning pipelines, fact-checking, and filtering.

6.2. Retrieval-Augmented Generation (RAG)

RAG is a powerful and widely adopted technique to ground the model's generation in verifiable external knowledge. Instead of relying solely on its parametric knowledge, the model first retrieves relevant information from a trusted knowledge base (e.g., a collection of company documents, a Wikipedia dump) and then uses this retrieved context to generate its answer.

The RAG pipeline typically consists of:

1. **Retriever:** Given a user query, this component searches the knowledge base and retrieves the most relevant document chunks.
2. **Generator:** The LLM receives the original query along with the retrieved documents in its context window. It is then prompted to synthesize an answer based *only* on the provided information.

This approach significantly reduces extrinsic hallucinations and allows for source attribution, as the model can cite the documents used to formulate the answer.

6.3. Advanced Decoding and Inference-Time Interventions

Several techniques aim to detect and correct hallucinations during the generation process

itself.

A. Active Detection and Mitigation via Knowledge Validation

As proposed in the paper "A Stitch in Time Saves Nine: Detecting and Mitigating Hallucinations of LLMs by Validating Low-Confidence Generation" (2023), this method is based on two key findings:

- 1. Hallucinations often propagate: an initial hallucinated statement can lead to further errors.
- 2. The model's own logit outputs can signal uncertainty. Low-confidence generations are more likely to be hallucinations.

The proposed method, illustrated below, actively validates low-confidence statements against external knowledge *during* generation.

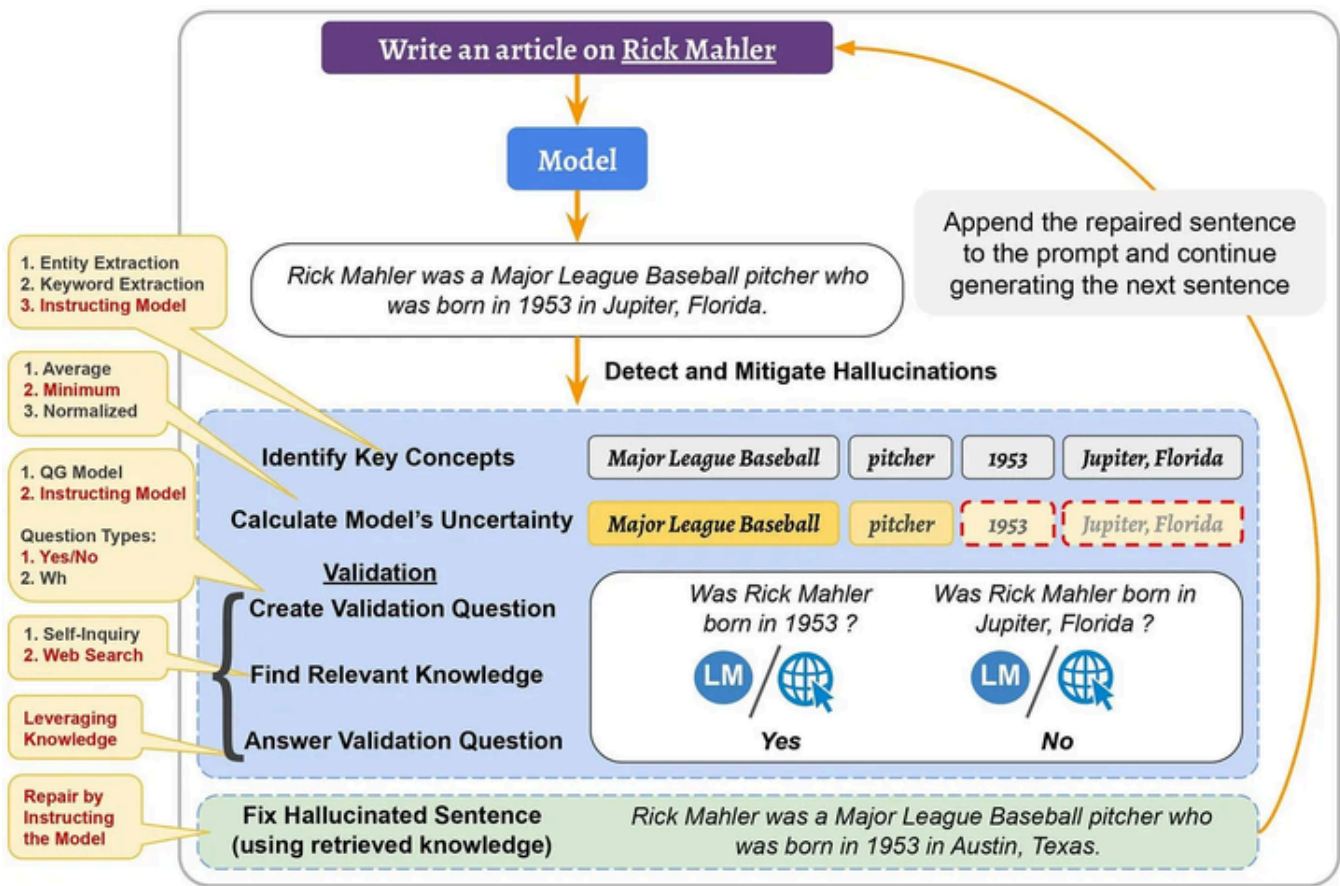


Figure 1: The process of actively detecting and mitigating hallucinations. The system identifies low-confidence concepts, retrieves external knowledge to validate them, and repairs the sentence before continuing generation.

Process:

1. **Detect:** Identify key concepts in the sentence being generated. Use the model's logit outputs to calculate an uncertainty score for these concepts.
2. **Validate:** If the uncertainty is high, query an external knowledge base (like a search engine) to find evidence related to the generated statement.
3. **Mitigate:** If the statement is contradicted by the retrieved knowledge, "repair" the generated sentence to align with the evidence. This corrected sentence is then used as context for generating the subsequent text, preventing error propagation.

B. Factual-Nucleus Sampling

The paper "*Factuality Enhanced Language Models for Open-Ended Text Generation*" (2022) observes that randomness is more harmful to factuality later in a sentence. The start of a sentence has many valid possibilities, but as it progresses, the constraints increase. Factual-Nucleus Sampling dynamically adjusts the nucleus probability p during generation.

The nucleus probability p_t for the t -th token in a sentence is decayed using the formula:

$$p_t = \max(\lambda^{t-1}, \omega)$$

where $\lambda \in (0, 1]$ is a decay factor and ω is a lower bound. This makes the sampling more greedy (less random) as the sentence gets longer, helping to maintain factual consistency.

C. SelfCheckGPT: Checking Consistency Across Multiple Generations

The core idea of SelfCheckGPT (2023) is that if a model truly "knows" a fact, it will state it consistently across multiple sampled responses. If it's hallucinating, its responses will likely be divergent or contradictory.

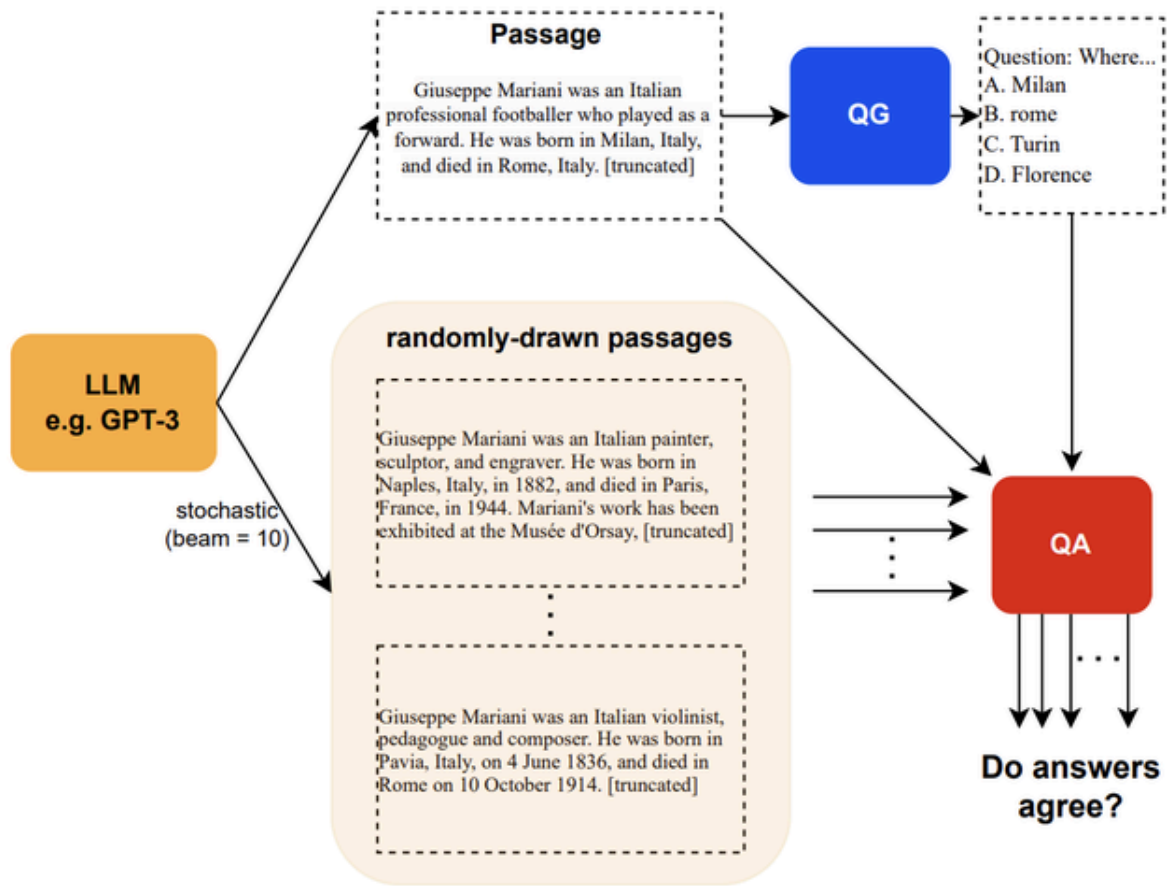


Figure 1: SelfCheckGPT with Question Answering.

Figure 2: The workflow of SelfCheckGPT. Multiple responses are sampled from the LLM. The system then evaluates the consistency across these responses to determine which statements are factual and which are likely hallucinations.

Process:

1. **Sample:** Generate multiple responses to the same prompt using stochastic decoding (e.g., by varying the temperature).
2. **Compare:** Measure the information consistency across the different responses. This can be done using various methods, such as NLI models, QA-based checks, or measuring semantic similarity with BERTScore.
3. **Score:** Assign a factuality score to each statement in the original response based on how consistently it appears in the other samples. Statements that are contradicted or absent in other samples are flagged as potential hallucinations.

Interview Questions

Theoretical Questions

Question 1: What is the difference between intrinsic and extrinsic hallucination? Provide an example of each in the context of summarizing a news article.

Answer:

The distinction lies in the relationship between the generated text and a provided source document.

- **Intrinsic Hallucination** occurs when the model's output directly contradicts the source material.
 - **Example:** Suppose a source news article states, "The project was completed in 2023 for a cost of 50million." *An intrinsic hallucination in a summary would be : "The project, finished in 2023, cost 100 million."* The cost is factually incorrect according to the source.
 - **Extrinsic Hallucination** occurs when the model introduces information that is not present in the source document. This information might be true in the real world, but it cannot be verified from the source alone.
 - **Example:** Using the same article, an extrinsic hallucination would be: "The project, led by chief engineer Jane Doe, was completed in 2023 for \$50 million." If Jane Doe's name is not mentioned anywhere in the source article, this is an extrinsic hallucination, even if she was indeed the chief engineer. In a strict summarization task, this is considered an error because the model is adding external knowledge.
-

Question 2: Explain how Retrieval-Augmented Generation (RAG) works and why it is an effective strategy against hallucinations.

Answer:

Retrieval-Augmented Generation (RAG) is a hybrid architecture that combines a parametric knowledge model (an LLM) with a non-parametric external knowledge source (a retriever).

How it works:

1. **Retrieval Step:** When a user submits a prompt, the system first uses a *retriever* module to search a large, trusted knowledge base (e.g., internal company documents, Wikipedia, legal texts). The retriever finds a small number of text chunks that are most relevant to the user's query. Common retrieval methods include vector similarity search (using embeddings) or keyword-based search.
2. **Augmentation Step:** The retrieved text chunks are then prepended to the original user prompt and fed into the context window of the LLM (the *generator*).
3. **Generation Step:** The LLM is instructed to generate an answer based *specifically on the provided context*. This grounds the model, forcing it to synthesize information from the retrieved documents rather than relying solely on its internal, parametric knowledge which might be outdated or incorrect.

Why it is effective against hallucinations:

1. **Grounding in Factual Evidence:** By forcing the model to base its response on explicitly provided text, RAG minimizes the chances of the model "making things up" (extrinsic hallucination) or stating facts that contradict a known source (intrinsic hallucination, assuming the knowledge base is accurate).
2. **Access to Up-to-Date Information:** LLM parametric knowledge becomes stale after its training date. A RAG system's knowledge base can be updated continuously, giving the model access to the latest information without needing to be retrained.
3. **Transparency and Verifiability:** RAG systems can cite their sources by referencing the documents they retrieved to formulate an answer. This allows users to verify the information, building trust and providing a pathway for fact-checking, which is impossible with a purely parametric model.

Question 3: Explain the mathematical intuition behind temperature sampling and how a high temperature can increase the likelihood of hallucinations.

Answer:

Temperature sampling is a decoding strategy used to control the randomness of an LLM's output. It works by rescaling the logit values that the model produces for each token in its vocabulary before the softmax function is applied.

The formula for the probability p_i of a token i with logit z_i and temperature T is:

$$p_i = \frac{\exp(z_i/T)}{\sum_j \exp(z_j/T)}$$

Mathematical Intuition:

- **Case 1: Low Temperature ($T \rightarrow 0^+$):** When the temperature is very low (e.g., 0.1), dividing the logits z_i by T magnifies the differences between them. A slightly higher logit becomes much larger than the others. After the exponentiation, the probability mass becomes highly concentrated on the token with the highest logit. In the limit $T \rightarrow 0$, this becomes equivalent to greedy decoding (always picking the most likely token). This leads to deterministic and often repetitive outputs.
- **Case 2: Temperature $T = 1$:** When $T = 1$, the formula is the standard softmax function. The probabilities directly reflect the model's original confidence.
- **Case 3: High Temperature ($T > 1$):** When the temperature is high (e.g., 1.5), dividing the logits by T shrinks the differences between them. The logits become closer to each other. After exponentiation, the resulting probability distribution becomes flatter, or more uniform. This means that tokens with originally low logits now have a non-trivial probability of being selected.

Connection to Hallucinations:

A high temperature increases the randomness of the output by making the model more likely to choose less probable words. While this can enhance creativity and produce more diverse text, it comes at a significant cost to factuality. The model might select a token that is semantically plausible but factually incorrect, initiating a hallucination. For instance, instead of picking the factually correct but perhaps high-probability token "Paris," a high-temperature setting might lead the model to sample the lower-probability token "Milan" in the sentence "The capital of France is...", thus creating a factual error.

Practical & Coding Questions

Question 4: Using PyTorch, write a function that takes a tensor of logits and a temperature value, and returns the re-scaled probabilities. Then, use Matplotlib to visualize how different temperatures (e.g., 0.2, 1.0, 2.0) affect the final probability distribution.

Answer:

```

import torch
import torch.nn.functional as F
import matplotlib.pyplot as plt
import numpy as np

def apply_temperature_scaling(logits, temperature):
    """
    Applies temperature scaling to a tensor of logits.

    Args:
        logits (torch.Tensor): A 1D tensor of raw logit values.
        temperature (float): The temperature value. Must be > 0.

    Returns:
        torch.Tensor: A 1D tensor of probabilities after applying temperature
            and the softmax function.
    """
    if temperature <= 0:
        raise ValueError("Temperature must be positive.")

    # Divide logits by the temperature
    scaled_logits = logits / temperature

    # Apply softmax to get probabilities
    probabilities = F.softmax(scaled_logits, dim=-1)

    return probabilities

# --- Visualization ---

# 1. Define a sample logit tensor
# Let's assume this represents the model's output for a small vocabulary
sample_logits = torch.tensor([2.0, 1.5, 0.5, -1.0, -2.5])
vocabulary = ['apple', 'banana', 'cherry', 'date', 'elderberry']

# 2. Define temperatures to test
temperatures = [0.2, 1.0, 2.0]

# 3. Create the plot

```

```

plt.style.use('seaborn-v0_8-whitegrid')
fig, ax = plt.subplots(figsize=(12, 7))

bar_width = 0.2
index = np.arange(len(vocabulary))

for i, temp in enumerate(temperatures):
    # Calculate probabilities for the current temperature
    probs = apply_temperature_scaling(sample_logits, temp)

    # Plot bars
    ax.bar(index + i * bar_width, probs.numpy(), bar_width, label=f'Temp = {temp:.1f}')

# 4. Formatting the plot
ax.set_xlabel('Vocabulary Tokens', fontsize=14)
ax.set_ylabel('Probability', fontsize=14)
ax.set_title('Effect of Temperature on Probability Distribution', fontsize=16)
ax.set_xticks(index + bar_width)
ax.set_xticklabels(vocabulary)
ax.legend()
plt.tight_layout()
plt.show()

```

``**Explanation of the Code:**

1. **``apply_temperature_scaling`` function:** This function implements the core logic. It
2. **Visualization Setup:** We define a sample `logits` tensor representing the model's
3. **Plotting Logic:** We iterate through each temperature, calculate the corresponding p
4. **Interpreting the Visualization:**
 - * **Temp = 0.2 (Low):** The distribution is "sharpened." The probability of the mos
 - * **Temp = 1.0 (Neutral):** This is the standard softmax output, reflecting the mod
 - * **Temp = 2.0 (High):** The distribution is "flattened." The probabilities of all

Question 5: Implement nucleus (top-p) sampling from scratch using PyTorch. Your function

Answer:

```

python
import torch
import torch.nn.functional as F

def nucleus_sampling(logits, p):
    """
    Implements nucleus (top-p) sampling from scratch.

    Args:
        logits (torch.Tensor): A 1D tensor of raw logit values.
        p (float): The cumulative probability threshold for nucleus sampling ( $0 < p \leq 1$ )

    Returns:
        Tuple[torch.Tensor, torch.Tensor]:
            - A 1D tensor with the filtered probability distribution (values for
              tokens outside the nucleus are set to 0).
            - A 1D tensor containing the indices of the tokens within the nucleus.
    """
    if not (0 < p <= 1):
        raise ValueError("p must be in the range (0, 1].")

    # 1. Get original probabilities using softmax
    probabilities = F.softmax(logits, dim=-1)

    # 2. Sort probabilities in descending order
    sorted_probs, sorted_indices = torch.sort(probabilities, descending=True)

    # 3. Find the cumulative sum of sorted probabilities
    cumulative_probs = torch.cumsum(sorted_probs, dim=-1)

    # 4. Find the indices of tokens to remove.
    # These are the tokens whose cumulative probability is > p.
    # We also keep at least one token by ensuring the first token is never removed.
    sorted_indices_to_remove = cumulative_probs > p
    sorted_indices_to_remove[1:] = sorted_indices_to_remove[:-1].clone()
    sorted_indices_to_remove[0] = False

    # 5. Create a mask to set probabilities of removed tokens to 0

```

```

# The mask should be the same size as the original logits
indices_to_remove = sorted_indices[sorted_indices_to_remove]

# Create a new probability distribution where removed tokens have 0 probability
filtered_probs = probabilities.clone()
filtered_probs[indices_to_remove] = 0

# 6. Renormalize the distribution so it sums to 1 again
filtered_probs = filtered_probs / torch.sum(filtered_probs)

# 7. Get the indices of tokens that are kept (the nucleus)
kept_indices = torch.where(filtered_probs > 0)[0]

return filtered_probs, kept_indices

# --- Example Usage ---
logits_example = torch.tensor([3.2, 2.1, 1.0, 4.5, 0.5]) # Vocabulary size 5
vocab = ["A", "B", "C", "D", "E"]
p_value = 0.90

# Original probabilities
original_probs = F.softmax(logits_example, dim=-1)
print(f"Original Logits: {logits_example.numpy()}")
print(f"Original Probabilities: {original_probs.numpy().round(3)}")
print("-" * 30)

# Apply nucleus sampling
filtered_distribution, nucleus_indices = nucleus_sampling(logits_example, p=p_value)

print(f"Nucleus Sampling with p = {p_value}")
print(f"Filtered (Renormalized) Probabilities: {filtered_distribution.numpy().round(3)}")
print(f"Indices in Nucleus: {nucleus_indices.numpy()}")
print(f"Tokens in Nucleus: {[vocab[i] for i in nucleus_indices]}")

# Demonstrate the selection logic
sorted_probs, sorted_indices = torch.sort(original_probs, descending=True)
print("\n--- Selection Logic ---")
for i, idx in enumerate(sorted_indices):
    print(f"Token '{vocab[idx]}': Prob={sorted_probs[i]:.3f}, CumProb={torch.cumsum(sorted_probs, 0)[i+1]:.3f}")

```


Explanation of the Code:

1. **Get Probabilities:** The function starts by converting the input `logits` into a standard probability distribution using `softmax`.
2. **Sort:** It then sorts these probabilities in descending order while keeping track of their original indices. This is the crucial step to identify the most likely tokens.
3. **Cumulative Sum:** It calculates the cumulative sum of the sorted probabilities. This allows us to find the smallest set of tokens whose total probability mass exceeds the threshold `p`.
4. **Identify Nucleus:** The code finds the point where the cumulative sum first exceeds `p`. All tokens from that point onward in the sorted list are marked for removal. The `clone()` and shifting logic ensures that we keep the token that just crosses the threshold, as per the definition of nucleus sampling.
5. **Filter and Renormalize:** A new probability tensor is created where the probabilities of the removed tokens are set to zero. This new distribution no longer sums to 1, so it is renormalized by dividing by its new sum.
6. **Return Values:** The function returns the final, filtered probability distribution (which can now be used for sampling) and the indices of the tokens that were included in the nucleus. The example usage clearly shows which tokens are kept ("D" and "A") because their combined probability is the first to be ≥ 0.90 .

References

- [The Hallucination Problem of Large Language Models](#)
- Ji, Z., Lee, N., Frieske, R., Yu, T., Su, D., Xu, Y., ... & Fung, P. (2023). Survey of hallucination in natural language generation. *ACM Computing Surveys*, 55(12), 1-38. (Note: Original link was to a Chinese blog post, a link to the foundational survey paper is more appropriate for a technical guide.)
- Manakul, P., Liusie, A., & Gales, M. J. (2023). SelfCheckGPT: Zero-Resource Black-Box Hallucination Detection for Generative Large Language Models. *arXiv preprint arXiv:2303.08896*. (Note: Original link was to a Chinese blog post, a link to the original research paper is provided here.)