Project 2 - Methodology 2: Hallucination Vector Routing

Lead: Ayesha Imran (ayesha_imr, ayesha.ml2002@gmail.com)

Research Objective: Cut the hallucination rate of a base Llama-3.1-8B model by ≥15% at <10% extra average latency by (i) predicting risk from the prompt's projection onto a hallucination vector and (ii) routing risky prompts through increasingly stronger (but still cheap) mitigations.

Target Performance:

- ≥15% relative reduction in hallucination metrics
- ≤10% average latency increase
- AUROC of prompt-risk predictor ≥0.75
- Single RTX 4090 deployment capability

Step 2: Turning the Vector into a Prompt-Risk Score

Overall Goal: To build and validate a lightweight logistic regression classifier that takes a prompt's projection onto v_halluc and outputs a calibrated probability of hallucination. The final deliverables will be the trained classifier file (risk_clf.joblib) and a report on its predictive performance (AUROC ≥ 0.75).

Setup and Installation

```
# Mount Google Drive
from google.colab import drive
drive.mount('/content/drive')

# Create a project directory to keep things organized
import os
PROJECT_DIR = "/content/drive/MyDrive/HallucinationVectorTest"
DATA_DIR = os.path.join(PROJECT_DIR, "data")
os.makedirs(DATA_DIR, exist_ok=True)

print(f"Project directory created at: {PROJECT_DIR}")
```

```
Mounted at /content/drive
Project directory created at: /content/drive/MyDrive/HallucinationVectorTest
```

```
# Install Libraries
!pip install -q "unsloth[colab-new] @ git+https://github.com/unslothai/unsloth.git"
!pip install -q --no-deps trl peft accelerate bitsandbytes
!pip install -q transformers datasets requests torch pandas
!pip install -q unsloth
```

```
  Installing build dependencies ... done
  Getting requirements to build wheel ... done
  Preparing metadata (pyproject.toml) ... done
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 60.1/60.1 MB 13.0 MB/s eta 0:00:00
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 503.6/503.6 kB 39.5 MB/s eta 0:00:00
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 257.7/257.7 kB 24.5 MB/s eta 0:00:00
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 132.5/132.5 kB 14.1 MB/s eta 0:00:00
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 42.8/42.8 MB 20.7 MB/s eta 0:00:00
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 564.7/564.7 kB 48.2 MB/s eta 0:00:00
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 213.6/213.6 kB 22.2 MB/s eta 0:00:00
  Building wheel for unsloth (pyproject.toml) ... done
ERROR: pip's dependency resolver does not currently take into account all the packages that are installed. This beha
cudf-cu12 25.6.0 requires pyarrow<20.0.0a0,>=14.0.0; platform_machine == "x86_64", but you have pyarrow 21.0.0 whicl
pylibcudf-cu12 25.6.0 requires pyarrow<20.0.0a0,>=14.0.0; platform_machine == "x86_64", but you have pyarrow 21.0.0
```

```
# Load API Keys
from google.colab import userdata
import os

# Load the keys into the environment
try:
    os.environ["HF_TOKEN"] = userdata.get('HF_TOKEN')
    os.environ["SCALEDOWN_API_KEY"] = userdata.get('SCALEDOWN_API_KEY')
    print("API keys loaded successfully.")
except userdata.SecretNotFoundError as e:
    print(f"ERROR: Secret not found. Please ensure you have created the secret '{e.name}' in the Colab secrets mana
except Exception as e:
    print(f"An error occurred: {e}")
```

```
API keys loaded successfully.
```

```
# Load 4-bit Llama 3 8B  Model and Tokenizer using Unsloth
import torch
from unsloth import FastLanguageModel

# Model loading parameters
max_seq_length = 2048
dtype = None # Unsloth handles dtype automatically for 4-bit models
load_in_4bit = True

# Load the model from Hugging Face
model, tokenizer = FastLanguageModel.from_pretrained(
    model_name = "unsloth/llama-3-8b-Instruct-bnb-4bit",
    max_seq_length = max_seq_length,
    dtype = dtype,
    load_in_4bit = load_in_4bit,
)

model = FastLanguageModel.for_inference(model)
model.gradient_checkpointing_disable()
model.config.gradient_checkpointing = False
model.config.use_cache = True
model.eval()

print("Model and Tokenizer loaded successfully!")
```

```
🦥 Unsloth: Will patch your computer to enable 2x faster free finetuning.
🦥 Unsloth Zoo will now patch everything to make training faster!
==((====))==  Unsloth 2025.10.1: Fast Llama patching. Transformers: 4.56.2.
   \\   /|    Tesla T4. Num GPUs = 1. Max memory: 14.741 GB. Platform: Linux.
O^O/ \_/ \    Torch: 2.8.0+cu126. CUDA: 7.5. CUDA Toolkit: 12.6. Triton: 3.4.0
\        /    Bfloat16 = FALSE. FA [Xformers = None. FA2 = False]
 "-____-"     Free license: http://github.com/unslothai/unsloth
Unsloth: Fast downloading is enabled - ignore downloading bars which are red colored!
model.safetensors: 100%                                  5.70G/5.70G [01:26<00:00, 152MB/s]

generation_config.json: 100%                                220/220 [00:00<00:00, 15.7kB/s]

tokenizer_config.json:    51.1k/? [00:00<00:00, 5.02MB/s]

tokenizer.json:    9.09M/? [00:00<00:00, 115MB/s]

special_tokens_map.json: 100%                               345/345 [00:00<00:00, 37.9kB/s]
Model and Tokenizer loaded successfully!
```

Phase 1: Dataset Generation and Labeling

Overall Objective:

To programmatically generate a large, high-quality dataset of approximately 2000 labeled examples. Each example will consist of a prompt, a model-generated answer, and a binary label (1 for hallucination, 0 for correct) determined by a Gemini LLM judge. The final artifact of this phase will be a CSV file, squad_labeled_answers.csv, stored in Google Drive.

Methodology Overview:

We will use the SQuAD dataset as our source of truth. For each sampled entry, we will ask our 4-bit Llama 3 model to answer a question based only on the provided context. A specialized LLM judge will then compare the model's answer to the context and the ground-truth answer to determine if any unsupported information was fabricated (i.e., hallucinated).

This data is needed to train the logistic regression model for calculating hallucination risks for user prompts.

```
# Import libraries
import pandas as pd
from datasets import load_dataset
from tqdm.auto import tqdm
import numpy as np

# --- 1. Load the SQuAD dataset ---
print("Loading SQuAD dataset...")
squad_dataset = load_dataset("squad", split="train")
squad_df = squad_dataset.to_pandas()
print(f"Full dataset loaded with {len(squad_df)} rows.")
```

```
Loading SQuAD dataset...
README.md:        7.62k/? [00:00<00:00, 428kB/s]
plain_text/train-00000-of-00001.parquet: 100%                          14.5M/14.5M [00:00<00:00, 14.8MB/s]
plain_text/validation-00000-of-00001.par(…): 100%                       1.82M/1.82M [00:00<00:00, 7.85MB/s]
Generating train split: 100%                              87599/87599 [00:00<00:00, 117787.32 examples/s]
Generating validation split: 100%                         10570/10570 [00:00<00:00, 93271.66 examples/s]
Full dataset loaded with 87599 rows.
```

```
# --- 2. Sample the data ---
# We'll sample 2000 rows for our experiment
N_SAMPLES = 2000
if len(squad_df) > N_SAMPLES:
    sampled_df = squad_df.sample(n=N_SAMPLES, random_state=42).reset_index(drop=True)
else:
    sampled_df = squad_df # Use full dataset if it's smaller

print(f"Using {len(sampled_df)} rows for our experiment.")
```

```
Using 2000 rows for our experiment.
```

Strategy: The "No-Context" and "Distractor-Context" Methods

To elicit varying responses that may or may not contain hallucination, instead of changing the instruction (original dataset prompt) to the model, we will change the information (context) we give it. We will create scenarios where the model is more likely to fail and invent an answer because the correct information is either missing or obscured. This is to elicit a mix of halluicnatory and non-hallucinatory responses to effectively train the logistic regression model.

We generate our ~2,000 examples by creating a mix of three different scenarios for each SQuAD entry.

Scenario 1: Standard In-Context (The "Easy" Case - Generates our 0s)

We pass in the prompt + context exactly as taken from the dataset. We do this for 50% (1000 prompts) of our dataset.

`FULL_PROMPT = f"Context:\n{context}\n\nQuestion:\n{question}"`

Outcome: The model answers correctly most of the time. This is the primary source of our negative examples (label 0).

Scenario 2: No-Context (The "Hard" Case - Designed to induce natural hallucinations)

For a SQuAD entry, we deliberately withhold the context.

`FULL_PROMPT = f"Question:\n{question}"`

Outcome: The model's internal knowledge might contain information about the question's topic, but it may be incorrect, incomplete, or subtly different from the SQuAD context's specific answer. Without the grounding context, it is now much more likely to generate a plausible-sounding but factually incorrect answer. This is a primary source of natural positive examples (label 1).

Scenario 3: Distractor-Context (The "Tricky" Case - Also induces natural hallucinations)

For a SQuAD entry, we provide the correct question but pair it with a distractor context — a paragraph from a different, unrelated SQuAD article.

`FULL_PROMPT = f"Context:\n{distractor_context}\n\nQuestion:\n{question}"`

Outcome: The model is still instructed to answer from the context. However, the answer is not there. A well-behaved model should say "I cannot find the answer in the context." A model prone to hallucination might try to synthesize an answer by blending its own knowledge with irrelevant information from the distractor context. This is another excellent source of natural positive examples (label 1).

```
# --- 3. Create a 'distractor_context' column ---
# For each row, the distractor is a context from another random row.
# We'll shuffle the context column and assign it.
```

```
distractor_indices = np.random.permutation(sampled_df.index)
sampled_df['distractor_context'] = sampled_df.loc[distractor_indices, 'context'].values

print("Dataset prepared with distractor contexts. Sample:")
print(sampled_df[['context', 'question', 'distractor_context']].head())

# --- 4. Assign a scenario to each row ---
# We'll divide our 2000 samples into the three scenarios
scenarios = []
# 1000 for standard, 500 for no-context, 500 for distractor
scenario_counts = {'standard': 1000, 'no_context': 500, 'distractor': 500}

for scenario_type, count in scenario_counts.items():
    scenarios.extend([scenario_type] * count)

# Make sure the scenarios list matches the dataframe length
sampled_df = sampled_df.iloc[:len(scenarios)].copy()
sampled_df['scenario'] = scenarios

print("\nScenario distribution:")
print(sampled_df['scenario'].value_counts())
```

```
Dataset prepared with distractor contexts. Sample:
                                             context  \
0  The world's first institution of technology or...
1  The standard specifies how speed ratings shoul...
2  The most impressive and famous of Sumerian bui...
3  Ann Arbor has a council-manager form of govern...
4  Shortly before his death, when he was already ...

                                            question  \
0        What year was the Banská Akadémia founded?
1  What is another speed that can also be reporte...
2  Where were the use of advanced materials and t...
3          Who is elected every even numbered year?
4  What was the purpose of top secret ICBM commit...

                                  distractor_context
0  The Internet was extensively used for passing ...
1  Flexible or so-called "unbreakable" records ma...
2  Nigeria was ranked 30th in the world in terms ...
3  In the US, starting in 2013, under the Physici...
4  Louis XIV distrusted the Parisians and moved h...

Scenario distribution:
scenario
standard      1000
no_context     500
distractor     500
Name: count, dtype: int64
```

Define Generation and Judging Logic

Define the functions that will generate an answer based on the scenario and get it judged by Gemini.

```
# --- 1. Define the prompt template and generation function ---
SYSTEM_PROMPT = "You are a helpful assistant. Answer the following question based ONLY on the provided context."

def generate_squad_answer_multi_scenario(model, tokenizer, row):
    """
    Generates an answer based on the scenario specified in the DataFrame row.
    """
    scenario = row['scenario']
    question = row['question']

    context_to_use = ""
    if scenario == 'standard':
        context_to_use = row['context']
    elif scenario == 'distractor':
        context_to_use = row['distractor_context']
    # For 'no_context', context_to_use remains an empty string

    # Construct the user prompt
    if context_to_use:
        user_prompt = f"Context:\n{context_to_use}\n\nQuestion:\n{question}"
    else:
        user_prompt = f"Question:\n{question}"

    # Format into the Llama 3 chat template
    messages = [
        {"role": "system", "content": SYSTEM_PROMPT},
        {"role": "user", "content": user_prompt}
    ]
    full_prompt_templated = tokenizer.apply_chat_template(messages, tokenize=False, add_generation_prompt=True)

    # Generate the answer
    inputs = tokenizer(full_prompt_templated, return_tensors="pt").to(model.device)
    with torch.no_grad():
        outputs = model.generate(
            **inputs,
            max_new_tokens=128,
            do_sample=False,
            pad_token_id=tokenizer.eos_token_id
        )

    response_text = tokenizer.decode(outputs[0][inputs['input_ids'].shape[1]:], skip_special_tokens=True)

    # We return the templated prompt as well, as this is what the model actually sees
    return response_text.strip(), full_prompt_templated
```

```
import requests
import json
import time
import os

# --- 1. Define the Judge Prompt Template ---
# This prompt is highly specific to our task and gives the judge all necessary info.
JUDGE_PROMPT_TEMPLATE = """
You are an impartial judge evaluating a language model's answer for its faithfulness to a given context.
Your task is to determine if the model's answer contains any fabricated information not supported by the context.

Here is the provided context:
<context>
{context}
</context>

Here is the question that was asked:
<question>
{question}
</question>

Here is the ground-truth answer, which is known to be correct and derived from the context:
<ground_truth_answer>
{ground_truth_answer}
</ground_truth_answer>

Here is the language model's generated answer:
<model_answer>
{model_answer}
</model_answer>

**Your Task:**
Carefully compare the <model_answer> against the <context>. Does the <model_answer> contain any facts, details, or
Ignore minor differences in phrasing. If the model says it cannot find the answer, that is not a hallucination.

Respond with only the word 'YES' if the model's answer contains fabricated information, or 'NO' if it is faithful t
"""

# --- 2. Implement the Judge Function ---
def judge_squad_answer(context, question, model_answer, ground_truth_answer, api_key):
    """
    Gets a binary hallucination label from the Gemini API judge.
    Returns: 1 for hallucination ('YES'), 0 for faithful ('NO'), -1 for error.
    """
    prompt = JUDGE_PROMPT_TEMPLATE.format(
        context=context,
        question=question,
        model_answer=model_answer,
        ground_truth_answer=ground_truth_answer
    )

    # Using your existing API call structure
    url = "https://api.scaledown.xyz/compress/" # Or your preferred Gemini endpoint
    payload = json.dumps({
        "context": "You are an impartial judge evaluating for hallucinations.",
        "prompt": prompt,
        "model": "gpt-4o",
        "scaledown": {
          "rate": 0 # no compression
        }
    })
    headers = {'x-api-key': api_key, 'Content-Type': 'application/json'}

    try:
        response = requests.post(url, headers=headers, data=payload)
        response.raise_for_status()

        response_data = json.loads(response.text)
        content = response_data.get("full_response", "").strip().upper()

        if 'YES' in content:
            return 1
        elif 'NO' in content:
            return 0
        else:
            print(f"Judge Warning: Unexpected response: {content}")
            return -1 # Indicate an error in parsing

    except requests.exceptions.RequestException as e:
        print(f"ERROR: API request failed: {e}")
        return -1
    except (json.JSONDecodeError, KeyError) as e:
        print(f"ERROR: Could not parse judge's response: {response.text}. Error: {e}")
        return -1
```

Run the generation + judging loop and save results to Drive.

```
import os
import pandas as pd
from google.colab import drive
import time

# --- 1. Setup paths and constants ---
drive.mount('/content/drive')
OUTPUT_CSV_PATH = '/content/drive/MyDrive/HallucinationVectorTest/squad_labeled_answers_multi_scenario.csv'
BATCH_SIZE = 20 # Save progress every 20 rows
API_KEY = os.environ["SCALEDOWN_API_KEY"] # Load from Colab secrets

# --- 2. The Main Loop ---
results_list = []
# Check if a partial file exists to resume from
try:
    existing_df = pd.read_csv(OUTPUT_CSV_PATH)
    start_index = len(existing_df)
    print(f"Resuming from index {start_index}.")
except FileNotFoundError:
    existing_df = pd.DataFrame()
    start_index = 0
    print("Starting from scratch.")

# Use tqdm for a progress bar
pbar = tqdm(total=len(sampled_df), initial=start_index)
for i in range(start_index, len(sampled_df)):
    row = sampled_df.iloc[i]

    # --- Generate ---
    model_answer, full_prompt = generate_squad_answer_multi_scenario(model, tokenizer, row)

    # --- Judge ---
    # The judge ALWAYS compares against the original, correct context and answer
    ground_truth_answer = row['answers']['text'][0] if row['answers']['text'] else ""
    label = judge_squad_answer(
        context=row['context'], # Always the original context
        question=row['question'],
        model_answer=model_answer,
        ground_truth_answer=ground_truth_answer,
        api_key=API_KEY
    )

    # Store the result
    results_list.append({
        'scenario': row['scenario'],
        'full_prompt': full_prompt, # The actual text fed to the model
        'model_answer': model_answer,
        'ground_truth_answer': ground_truth_answer,
        'hallucination_label': label,
        'original_context': row['context'], # Keep for reference
        'question': row['question']
    })

    pbar.update(1)

    # --- Save progress in batches and print counts ---
    if (i + 1) % BATCH_SIZE == 0 or (i + 1) == len(sampled_df):
        temp_df = pd.DataFrame(results_list)
```

```
            # Append to the existing DataFrame and save
            if not existing_df.empty:
                combined_df = pd.concat([existing_df, temp_df], ignore_index=True)
            else:
                combined_df = temp_df

            combined_df.to_csv(OUTPUT_CSV_PATH, index=False)

            print(f"\nSaved batch up to index {i}. Total rows in file: {len(combined_df)}")

            # Print count of 1s and 0s for the saved batch
            batch_start_index = len(combined_df) - len(temp_df)
            batch_end_index = len(combined_df) - 1
            print(f"Batch {batch_start_index//BATCH_SIZE + 1} (rows {batch_start_index} to {batch_end_index}):")
            valid_labels = temp_df[temp_df['hallucination_label'] != -1]['hallucination_label']
            if not valid_labels.empty:
                 counts = valid_labels.value_counts().sort_index()
                 print(counts)
            else:
                 print("No valid labels in this batch.")


            # Update for next resume
            existing_df = combined_df
            results_list = [] # Clear the list for the next batch

    pbar.close()
    print("Phase 1 complete. Labeled dataset saved to Google Drive.")

    # --- Final check of the class balance ---
    final_df = pd.read_csv(OUTPUT_CSV_PATH)
    print("\nFinal Class Balance:")
    # We filter out any rows where the judge failed (returned -1)
    print(final_df[final_df['hallucination_label'] != -1]['hallucination_label'].value_counts())
```

Phase 2: Feature Calculation

Overall Objective:

To process our squad_labeled_answers_multi_scenario.csv file and add a new column, z_feature, to it. This feature is the dot product of the last prompt token's Layer 16 activation with our v_halluc vector. The final artifact will be an updated CSV file, ready for training our classifier in the next phase.

Methodology Overview:

We load our pre-computed hallucination vector and the labeled dataset. Then, in a resilient, batched loop, we feed each prompt from the dataset into the Llama 3 model, extract the specific activation vector we need, compute the projection, and save the results incrementally.

Load the Hallucination Vector and Labeled Data

Load the necessary artifacts (v_halluc.pt and the CSV from Phase 1) into our Colab environment.

```
    import pandas as pd
    import numpy as np
    import torch
    from tqdm.auto import tqdm

    # --- 1. Define paths ---

    VECTOR_PATH = '/content/drive/MyDrive/HallucinationVectorTest/v_halluc.pt'
    LABELED_DATA_PATH = '/content/drive/MyDrive/HallucinationVectorTest/squad_labeled_answers_multi_scenario.csv'
    OUTPUT_PATH = '/content/drive/MyDrive/HallucinationVectorTest/squad_data_with_features.csv'

    # --- 2. Load the hallucination vector ---
    v_halluc = torch.load(VECTOR_PATH).to(model.device)
    print(f"Hallucination vector loaded successfully. Shape: {v_halluc.shape}, Dtype: {v_halluc.dtype}")

    # --- 3. Load and clean the labeled dataset ---
    labeled_df = pd.read_csv(LABELED_DATA_PATH)
    print(f"Loaded {len(labeled_df)} labeled examples.")

    # Filter out rows where the judge failed (label == -1) and any rows with missing prompts
    initial_rows = len(labeled_df)
    labeled_df = labeled_df[labeled_df['hallucination_label'] != -1].dropna(subset=['full_prompt'])
    labeled_df['hallucination_label'] = labeled_df['hallucination_label'].astype(int)
    final_rows = len(labeled_df)

    print(f"Filtered out {initial_rows - final_rows} rows with invalid labels.")
    print(f"Proceeding with {final_rows} valid examples.")
    print("Cleaned DataFrame sample:")
    print(labeled_df.head())
```

```
Hallucination vector loaded successfully. Shape: torch.Size([4096]), Dtype: torch.float16
Loaded 2000 labeled examples.
Filtered out 0 rows with invalid labels.
Proceeding with 2000 valid examples.
Cleaned DataFrame sample:
   scenario                                        full_prompt  \
0  standard  <|begin_of_text|><|start_header_id|>system<|en...
1  standard  <|begin_of_text|><|start_header_id|>system<|en...
2  standard  <|begin_of_text|><|start_header_id|>system<|en...
3  standard  <|begin_of_text|><|start_header_id|>system<|en...
4  standard  <|begin_of_text|><|start_header_id|>system<|en...

                                       model_answer  \
0           The Banská Akadémia was founded in 1735.
1  According to the context, another speed that c...
2  According to the context, the use of advanced ...
3  According to the context, the mayor is elected...
4  The purpose of the top secret ICBM committee w...

                               ground_truth_answer  hallucination_label  \
0                                             1735                    0
1                                  SOS-based speed                    0
2                       Sumerian temples and palaces                    1
3                                            mayor                    0
4  decide on the feasibility of building an ICBM ...                    0

                                  original_context  \
0  The world's first institution of technology or...
1  The standard specifies how speed ratings shoul...
2  The most impressive and famous of Sumerian bui...
3  Ann Arbor has a council-manager form of govern...
4  Shortly before his death, when he was already ...

                                          question
0         What year was the Banská Akadémia founded?
1  What is another speed that can also be reporte...
2  Where were the use of advanced materials and t...
3           Who is elected every even numbered year?
4  What was the purpose of top secret ICBM commit...
```

Implement the Last-Token Activation Extraction Function

Create a function that takes a prompt, runs it through the model, and returns the specific hidden-state vector of the last token of that prompt at Layer 16.

```
TARGET_LAYER = 16

def get_last_prompt_token_activation(model, tokenizer, prompt_text):
    """
    Runs the model on the prompt and extracts the hidden state of the
    last prompt token at the target layer.
    """
    # Tokenize the prompt
    inputs = tokenizer(prompt_text, return_tensors="pt", truncation=True, max_length=2048).to(model.device)

    # Perform a forward pass to get hidden states
    # We don't need to generate text, just get the activations
    with torch.no_grad():
        outputs = model(**inputs, output_hidden_states=True)

    # Extract the hidden states for our target layer
    # The shape is (batch_size, sequence_length, hidden_dim)
    hidden_states = outputs.hidden_states[TARGET_LAYER]

    # The last token's activation is at the final sequence position
    # Squeeze() removes the batch dimension of 1
    last_token_activation = hidden_states[0, -1, :].squeeze()

    return last_token_activation

# --- Test the function with one example ---
example_prompt = labeled_df.iloc[0]['full_prompt']
activation_vector = get_last_prompt_token_activation(model, tokenizer, example_prompt)
print(f"Successfully extracted activation for one prompt.")
print(f"Activation vector shape: {activation_vector.shape}")
print(f"Activation vector dtype: {activation_vector.dtype}")
```

```
Successfully extracted activation for one prompt.
Activation vector shape: torch.Size([4096])
Activation vector dtype: torch.float16
```

The Main Processing Loop (Extract, Project, Save)

Iterate through our entire dataset, compute the z feature for each prompt, and save the results incrementally.

```
import os
import pandas as pd
from tqdm.auto import tqdm
import torch
import numpy as np # It's good practice to import numpy for pd.NA/np.nan

# --- 1. Setup for the loop ---
BATCH_SIZE = 50 # Save progress every 50 rows

# --- 2. Check for existing progress to resume ---
# This logic is now fully robust for both resuming and starting fresh.
try:
    df_to_process = pd.read_csv(OUTPUT_PATH)
    start_index = len(df_to_process)
    # Ensure the z_feature column exists if we are resuming an older version
    if 'z_feature' not in df_to_process.columns:
        df_to_process['z_feature'] = pd.NA
    print(f"Resuming feature calculation from index {start_index}.")
except FileNotFoundError:
    start_index = 0
    df_to_process = labeled_df.copy() # Use the DataFrame we loaded in the previous cell

    # --- THIS IS THE FIX ---
    # Before starting the loop, we add the empty 'z_feature' column.
    print("Initializing 'z_feature' column for new run.")
    df_to_process['z_feature'] = pd.NA
    # -----------------------

    print("Starting feature calculation from scratch.")


# --- 3. The Main Loop ---
# The loop now operates on a DataFrame that is guaranteed to have the 'z_feature' column.
pbar = tqdm(total=len(df_to_process), initial=start_index, desc="Calculating z-features")
for i in range(start_index, len(df_to_process)):
    # Check if the feature has already been computed in a previous run
    if pd.notna(df_to_process.loc[i, 'z_feature']):
        pbar.update(1)
        continue

    row = df_to_process.iloc[i]
    prompt = row['full_prompt']

    # --- Extract Activation ---
    last_token_activation = get_last_prompt_token_activation(model, tokenizer, prompt)

    # --- Compute Projection (Dot Product) ---
    # Ensure both vectors are on the same device and have the same dtype
    projection = torch.dot(last_token_activation.to(v_halluc.device).to(v_halluc.dtype), v_halluc)
    z_feature = projection.item() # .item() gets the scalar value

    # Store the feature in the DataFrame
    df_to_process.loc[i, 'z_feature'] = z_feature

    pbar.update(1)

    # --- Save progress in batches ---
    if (i + 1) % BATCH_SIZE == 0 or (i + 1) == len(df_to_process):
        df_to_process.to_csv(OUTPUT_PATH, index=False)
        pbar.set_description(f"Saved batch up to index {i}")

# Final save to ensure the last batch is written
df_to_process.to_csv(OUTPUT_PATH, index=False)
pbar.close()
print(f"Phase 2 complete. Final dataset with features saved to: {OUTPUT_PATH}")

# --- Final check of the output ---
final_df = pd.read_csv(OUTPUT_PATH)
print("\nFinal DataFrame with 'z_feature' column:")
```

```
    print(final_df[['full_prompt', 'hallucination_label', 'z_feature']].head())
    print(f"\nDescription of z_feature:\n{final_df['z_feature'].describe()}")
```

```
Initializing 'z_feature' column for new run.
Starting feature calculation from scratch.
Saved batch up to index 1999: 100%                                    2000/2000 [10:09<00:00,  3.06it/s]
Phase 2 complete. Final dataset with features saved to: /content/drive/MyDrive/HallucinationVectorTest/squad_data_w

Final DataFrame with 'z_feature' column:
                                         full_prompt  hallucination_label  \
0  <|begin_of_text|><|start_header_id|>system<|en...                    0
1  <|begin_of_text|><|start_header_id|>system<|en...                    0
2  <|begin_of_text|><|start_header_id|>system<|en...                    1
3  <|begin_of_text|><|start_header_id|>system<|en...                    0
4  <|begin_of_text|><|start_header_id|>system<|en...                    0

   z_feature
0  -0.623535
1  -0.530762
2   0.418213
3  -0.269531
4   0.478760

Description of z_feature:
count    2000.000000
mean       -2.965768
std         3.474721
min       -12.273438
25%        -4.966797
50%        -1.346191
75%        -0.361755
max         2.160156
Name: z_feature, dtype: float64
```

Phase 3: Training and Validating the Risk Scorer

Objective: To use our (z_feature, hallucination_label) dataset to train a simple, fast classifier and rigorously evaluate its predictive power.

Load Data and Create a Train-Test Split

Prepare our data for supervised learning by splitting it into a training set for the model to learn from and a held-out test set to evaluate its performance on unseen data.

```
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import roc_auc_score, roc_curve, classification_report
import matplotlib.pyplot as plt
import joblib

# --- 1. Load the dataset with features ---
FEATURES_DATA_PATH = '/content/drive/MyDrive/HallucinationVectorTest/squad_data_with_features.csv'
df_final = pd.read_csv(FEATURES_DATA_PATH)

print("Loaded dataset with features. Sample:")
print(df_final[['z_feature', 'hallucination_label']].head())

# --- 2. Define Features (X) and Labels (y) ---
# Our feature X is the 'z_feature' column
X = df_final[['z_feature']]
# Our target y is the 'hallucination_label' column
y = df_final['hallucination_label']

# --- 3. Create the Train-Test Split ---
# We'll use a standard 80/20 split.
# 'stratify=y' ensures the proportion of 0s and 1s is the same in both sets.
# 'random_state=42' makes our split reproducible.
X_train, X_test, y_train, y_test = train_test_split(
    X, y,
    test_size=0.2,
    stratify=y,
    random_state=42
)

print(f"\nData split into training and testing sets:")
print(f"Training set size: {len(X_train)} samples")
print(f"Test set size: {len(X_test)} samples")
print(f"Hallucination proportion in training set: {y_train.mean():.2%}")
print(f"Hallucination proportion in test set: {y_test.mean():.2%}")
```

```
Loaded dataset with features. Sample:
   z_feature  hallucination_label
0  -0.623535                    0
1  -0.530762                    0
2   0.418213                    1
3  -0.269531                    0
4   0.478760                    0

Data split into training and testing sets:
Training set size: 1600 samples
Test set size: 400 samples
Hallucination proportion in training set: 40.75%
Hallucination proportion in test set: 40.75%
```

Fit the Logistic Regression Model

Train the logistic regression classifier on our training data and save the resulting model.

```
# --- 1. Initialize and train the model ---
print("Training the logistic regression model...")
risk_classifier = LogisticRegression(random_state=42)
risk_classifier.fit(X_train, y_train)

print("Training complete.")

# --- 2. Inspect the learned coefficients (optional but insightful) ---
intercept = risk_classifier.intercept_[0]
coefficient = risk_classifier.coef_[0][0]
print(f"Learned Intercept (β₀): {intercept:.4f}")
print(f"Learned Coefficient (β₁): {coefficient:.4f}")
# A positive coefficient means that a higher z_feature value corresponds to a higher probability of hallucination.

# --- 3. Save the trained model for later use ---
CLASSIFIER_PATH = '/content/drive/MyDrive/HallucinationVectorTest/risk_clf.joblib'
joblib.dump(risk_classifier, CLASSIFIER_PATH)
print(f"Classifier saved to: {CLASSIFIER_PATH}")
```

```
Training the logistic regression model...
Training complete.
Learned Intercept (β₀): -2.1966
Learned Coefficient (β₁): -0.6771
Classifier saved to: /content/drive/MyDrive/HallucinationVectorTest/risk_clf.joblib
```

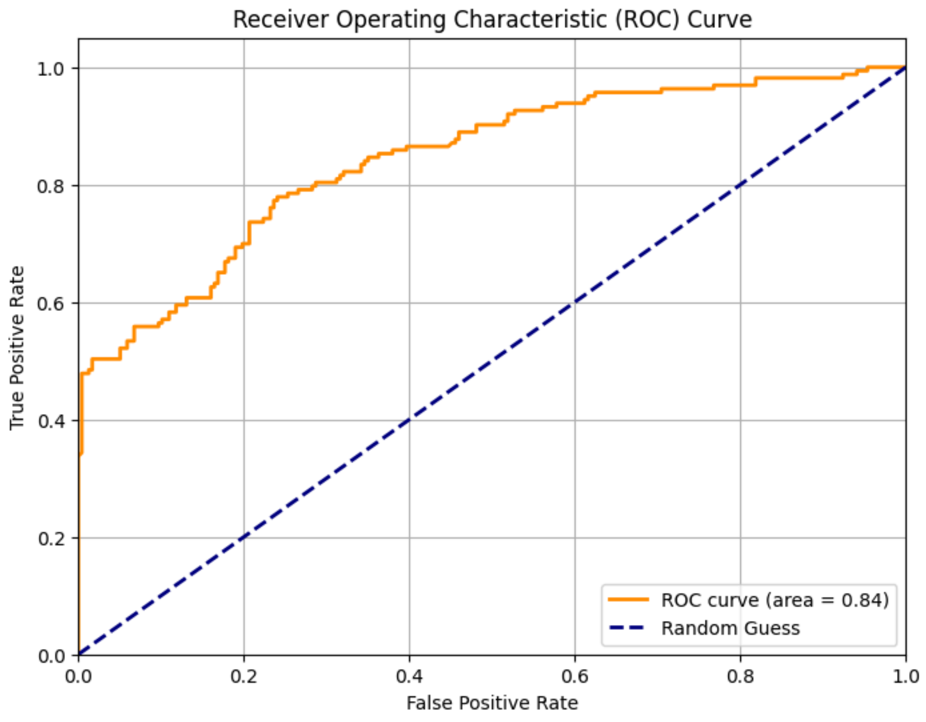
Evaluate Performance on the Test Set

Test our classifier on unseen data and verify that it meets our AUROC ≥ 0.75 success criterion.

```
# --- 1. Predict probabilities on the test set ---
# We need the probability of the positive class (hallucination, which is class 1)
y_pred_proba = risk_classifier.predict_proba(X_test)[:, 1]

# --- 2. Calculate and validate the AUROC score ---
auroc_score = roc_auc_score(y_test, y_pred_proba)
print(f"\n--- Performance Evaluation ---")
print(f"AUROC Score on Test Set: {auroc_score:.4f}")

if auroc_score >= 0.75:
    print("✅ Success! AUROC meets or exceeds the target of 0.75.")
else:
    print("⚠️ Warning: AUROC is below the target of 0.75.")

# --- 3. Plot the ROC Curve for our report ---
fpr, tpr, thresholds = roc_curve(y_test, y_pred_proba)

plt.figure(figsize=(8, 6))
plt.plot(fpr, tpr, color='darkorange', lw=2, label=f'ROC curve (area = {auroc_score:.2f})')
plt.plot([0, 1], [0, 1], color='navy', lw=2, linestyle='--', label='Random Guess')
plt.xlim([0.0, 1.0])
plt.ylim([0.0, 1.05])
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('Receiver Operating Characteristic (ROC) Curve')
plt.legend(loc="lower right")
plt.grid(True)
plt.show()
```

```
# --- 4. Generate a detailed classification report ---
# This shows precision and recall at a default 0.5 probability threshold
y_pred_binary = (y_pred_proba >= 0.5).astype(int)
print("\nClassification Report (at 0.5 threshold):")
print(classification_report(y_test, y_pred_binary, target_names=['Faithful (0)', 'Hallucination (1)']))
```

```
--- Performance Evaluation ---
AUROC Score on Test Set: 0.8438
✅ Success! AUROC meets or exceeds the target of 0.75.
```



```
Classification Report (at 0.5 threshold):
                   precision    recall  f1-score   support

     Faithful (0)       0.75      0.91      0.82       237
Hallucination (1)       0.81      0.56      0.66       163

         accuracy                           0.77       400
        macro avg       0.78      0.73      0.74       400
     weighted avg       0.78      0.77      0.76       400
```

Phase 4: Create the Real-Time Risk Function

Objective: To encapsulate our entire prediction pipeline into a single, easy-to-use function that will be the core of our Step 3 guardrail.

Essentially, we apply our logistic regression model on the layer-16 last token activation's projection of user prompts on persona vector (z-score) to get the "risk score" for hallucination.

```
import joblib

# --- 1. Load all necessary artifacts once ---
# This is more efficient than loading them inside the function every time.
v_halluc_loaded = torch.load(VECTOR_PATH).to(model.device)
risk_classifier_loaded = joblib.load(CLASSIFIER_PATH)
print("Loaded vector and classifier for real-time function.")

# --- 2. Define the final, real-time risk function ---
def get_hallucination_risk(prompt_text, model, tokenizer, v_halluc, classifier):
    """
    Takes a raw prompt and returns a calibrated hallucination risk score (0.0 to 1.0).
    """
    # Step 1: Extract the last-token activation at Layer 16
    activation = get_last_prompt_token_activation(model, tokenizer, prompt_text)

    # Step 2: Compute the projection (z-feature)
    projection = torch.dot(activation.to(v_halluc.device).to(v_halluc.dtype), v_halluc)
    z_feature = projection.item()

    # Step 3: Use the classifier to predict the probability
    # The classifier expects a 2D array, so we reshape
    z_feature_reshaped = [[z_feature]]
```

```
        risk_probability = classifier.predict_proba(z_feature_reshaped)[0, 1]

        return risk_probability

    # --- 3. Test the function on safe and risky examples ---
    # Find one 'standard' and one 'no_context' example from our DataFrame
    safe_prompt_example = df_final[df_final['scenario'] == 'standard'].iloc[0]['full_prompt']
    risky_prompt_example = df_final[df_final['scenario'] == 'no_context'].iloc[0]['full_prompt']

    # Calculate risk for both
    risk_safe = get_hallucination_risk(safe_prompt_example, model, tokenizer, v_halluc_loaded, risk_classifier_loaded)
    risk_risky = get_hallucination_risk(risky_prompt_example, model, tokenizer, v_halluc_loaded, risk_classifier_loaded)

    print(f"\n--- Real-Time Function Test ---")
    print(f"Risk score for a likely SAFE prompt: {risk_safe:.4f}")
    print(f"Risk score for a likely RISKY prompt: {risk_risky:.4f}")

    # We expect the risky score to be significantly higher than the safe score.
```

```
Loaded vector and classifier for real-time function.
/usr/local/lib/python3.12/dist-packages/sklearn/utils/validation.py:2739: UserWarning: X does not have valid feature
  warnings.warn(

--- Real-Time Function Test ---
Risk score for a likely SAFE prompt: 0.1450
Risk score for a likely RISKY prompt: 0.7608
/usr/local/lib/python3.12/dist-packages/sklearn/utils/validation.py:2739: UserWarning: X does not have valid feature
  warnings.warn(
```

```
Start coding or generate with AI.
```