Hallucination detection project

Motivation
Although hallucination detection in large language models has attracted growing attention most existing research is conducted in high-resource languages such as English. These approaches often rely on language-specific tools, datasets, or knowledge bases that are not available for many low-resource languages. As a result, their effectiveness in low-resource settings remains largely unverified. This project is motivated by the gap in current research and aims to explore and improve hallucination detection techniques tailored to low-resource languages, ultimately contributing to more inclusive and reliable multilingual language technologies.

Tasks
Based on Hallushift paper, we have 12 existing methods for hallucination detection, which are already applied on high-resource languages. Now we need to apply them on the 3 low resource languages and with two models(llama&opt) respectively to see if they work and evaluate the results.
Hallushift paper: https://arxiv.org/abs/2504.09482

Model	Method	Single Sampling	TruthfulQA	TriviaQA	CoQA	TydiQA-GP
OPT-6.7B	Perplexity [42]	✓	59.13	69.51	70.21	63.97
	LN-Entropy [35]	✗	54.42	71.42	71.23	52.03
	Semantic Entropy [43]	✗	52.04	70.08	69.82	56.29
	Lexical Similarity [44]	✗	49.74	71.07	66.56	60.32
	EigenScore [31]	✗	41.83	70.07	60.24	56.43
	SelfCKGPT [28]	✗	50.17	71.49	64.26	75.28
	Verbalize [45]	✓	50.45	50.72	55.21	57.43
	Self-evaluation [46]	✓	51.00	53.92	47.29	52.05
	CCS [47]	✓	60.27	51.11	53.09	65.73
	CCS* [47]	✓	63.91	53.89	57.95	64.62
	HaloScope [17]	✓	73.17	72.36	77.64	80.98
	HalluShift (Ours)	✓	**89.91**	**86.95**	**90.61**	**85.11**
LLaMA-2-7B	Perplexity [42]	✓	56.77	72.13	69.45	78.45
	LN-Entropy [35]	✗	61.51	70.91	72.96	76.27
	Semantic Entropy [43]	✗	62.17	73.21	63.21	73.89
	Lexical Similarity [44]	✗	55.69	75.96	74.70	44.41
	EigenScore [31]	✗	51.93	73.98	71.74	46.36
	SelfCKGPT [28]	✗	52.95	73.22	73.38	48.79
	Verbalize [45]	✓	53.04	52.45	48.45	47.97
	Self-evaluation [46]	✓	51.81	55.68	46.03	55.36
	CCS [47]	✓	61.27	60.73	50.22	75.49
	CCS* [47]	✓	67.95	63.61	51.32	80.38
	HaloScope [17]	✓	78.64	77.40	76.42	**94.04**
	HalluShift (Ours)	✓	**89.93**	**89.03**	**87.60**	87.61
LLaMA-3.1-8B	**HalluShift (Ours)**	✓	**92.97**	**99.23**	**90.38**	**87.70**

Models:
- Llama
- Opt

Dataset (for low resource languages)
- Tigrinya QA dataset: https://github.com/hailaykidu/TigQA-Dataset
- Armenian QA dataset: https://huggingface.co/datasets/gayaneghazaryan/SynDARin

— Basque QA dataset: https://huggingface.co/datasets/ixa-hitz/elkarhizketak

Methods:
- perplexity: **no repo** https://openreview.net/forum?id=kJUS5nD0vPB (see demo code in supplementary material in the link, or see file sent in ipynb) slides: https://iclr.cc/media/iclr-2023/Slides/11478.pdf
- LN-Entropy: https://github.com/KaosEngineer/structured-uncertainty blog: https://towardsdatascience.com/baseline-walkthrough-for-the-machine-translation-task-of-the-shifts-challenge-at-neurips-2021-e432c92882de/
- Semantic Entropy: https://github.com/lorenzkuhn/semantic_uncertainty
- Lexical Similarity: https://github.com/zlin7/UQ-NLG
- EigenScore: https://github.com/D2I-ai/eigenscore?utm_source=chatgpt.com
- SelfCKGPT: https://github.com/potsawee/selfcheckgpt
- Verbalize: https://github.com/sylinrl/CalibratedMath
- Self-evaluation: **no repo** https://arxiv.org/abs/2207.05221
- CSS: https://github.com/collin-burns/discovering_latent_knowledge
- HaloScope: https://github.com/deeplearning-wisc/haloscope
- Hallushift: https://github.com/sharanya-dasgupta001/hallushift/tree/main

如果没有repo，所有调用时用的参数都按Hallushift代码里的, for reference:
Self-evaluation：

1. Core idea

- When an LM answers a question, the **log-probability of its own generated tokens** (conditioned on the prompt) correlates strongly with whether it's correct.
- By **aggregating these log-probs** into a single "confidence" score, you can:
  - Predict whether the answer is correct.
  - Selectively abstain from answering low-confidence questions (improving accuracy on the remaining answers).

2. How they compute confidence

1. **Generate the answer** normally (greedy or sampling).
2. **While generating**, record the **log-probability of each chosen token** from the model's output distribution.
3. **Aggregate** these log-probs into a single number:
   - Often the **mean log-probability** (or average per-token probability) over the generated answer tokens.
   - You can also use the **minimum per-token probability** or more complex aggregations; the paper explores variants.
4. Use this aggregated score as the **model's confidence estimate**.

3. Using it for hallucination detection

- **Assumption:** If the model's answer is wrong/hallucinated, it tends to assign lower probabilities to its own output tokens.
- **Pipeline:**
  1. Get answer + mean token log-prob → **confidence score**.
  2. Calibrate a **threshold** using a validation set with correctness/hallucination labels.
  3. At inference:
     - If `confidence < threshold` → flag as **high hallucination risk** or abstain.
     - Else → accept the answer.

4. Applying it with LLaMA on your dataset

You can do this with Hugging Face pretty easily:

python 复制 编辑

```python
import torch
from transformers import AutoTokenizer, AutoModelForCausalLM
import numpy as np

model_name = "meta-llama/Llama-2-7b-hf"
device = "cuda" if torch.cuda.is_available() else "cpu"

tokenizer = AutoTokenizer.from_pretrained(model_name)
model = AutoModelForCausalLM.from_pretrained(model_name)
model.to(device)
model.eval()

def generate_with_confidence(prompt, max_new_tokens=50):
    inputs = tokenizer(prompt, return_tensors="pt").to(device)
    # Generate while keeping scores
    outputs = model.generate(
        **inputs,
        max_new_tokens=max_new_tokens,
        return_dict_in_generate=True,
        output_scores=True
    )

    # Decode answer (excluding the prompt)
    answer_tokens = outputs.sequences[0][inputs.input_ids.shape[1]:]
    answer_text = tokenizer.decode(answer_tokens, skip_special_tokens=True)

    # Compute per-token probabilities
    # output_scores is a list of logits for each generated token
    probs = []
    for score, token_id in zip(outputs.scores, answer_tokens):
        log_probs = torch.log_softmax(score, dim=-1)
        token_prob = torch.exp(log_probs[0, token_id])
        probs.append(token_prob.item())

    # Mean probability as confidence
    confidence = np.mean(probs)
    return answer_text, confidence

# Example
answer, conf = generate_with_confidence("Who is the president of France?")
print(answer)
print("Confidence:", conf)
```

5. How to evaluate

- **If you have hallucination labels** for your dataset:
  - Run this on all QA pairs.
  - Compare `confidence` scores to labels → compute AUROC, AUPRC.
  - Find a threshold that maximizes F1 or achieves desired TPR/FPR.
- **If you don't have labels**:
  - Use this score relatively: sort answers by confidence and only trust the top X%.

– Perplexity:

Great, you've got the core demo for **Relative Mahalanobis Distance (RMD)**. Here's exactly what you need to prepare—and how—to run it for hallucination/OOD detection on your datasets.

What inputs you must prepare

The demo expects **four numpy arrays of embeddings** (all with the same dimensionality **D**) plus a few trivial label arrays:

1. **Training (in-domain) embeddings**
- `embs_train_ind` : shape **[N_ind_train, D]**
  Feature vectors from your **in-domain training** inputs (the domain you care about, e.g., your QA/summarization task).
2. **Training (general-domain) embeddings**
- `embs_train_ood` : shape **[N_ood_train, D]**
  Feature vectors from a **broad/general corpus** (e.g., C4 for summarization, ParaCrawl for translation, or any large generic text set).
  These represent "background"/general distribution.
3. **Test (in-domain) embeddings**
- `embs_ind` : shape **[N_ind_test, D]**
  Feature vectors for your **in-domain test** inputs.
4. **Test (OOD/suspect) embeddings**
- `embs_ood` : shape **[N_ood_test, D]**
  Feature vectors for **OOD or suspected hallucination-inducing** inputs (e.g., domain-shifted, adversarial, or intentionally out-of-knowledge questions).
5. **Labels for evaluation**
- For AUROC/PR in the demo, you need binary labels for the **test** split:
  - `labels_ind = 0` for in-domain test
  - `labels_ood = 1` for OOD/suspect test
  (These are not class labels for classification; they're just OOD vs IND flags.)

> In the provided code they pass `np.zeros(len(embs_*))` to `compute_mean_and_cov` because they model **one Gaussian per split** (in-domain vs general-domain), not per class. That's why labels are all zeros there.

How RMD turns those inputs into a score

1. Fit one Gaussian on **in-domain training** (`embs_train_ind`) → `(mean_list, cov)`
2. Fit one Gaussian on **general/background** (`embs_train_ood`) → `(mean_list0, cov0)`
3. Compute **Mahalanobis distance** of each test example to each Gaussian:
   - `mdist_ind = M(x; μ_ind, Σ_ind)`
   - `mdist0_ind = M(x; μ_ood, Σ_ood)`
4. The **RMD score** is the **difference**:
   - `score = mdist_ind − mdist0_ind`
5. Larger scores ⇒ farther from in-domain **relative** to general domain ⇒ **more OOD / higher risk**.
6. Evaluate with `compute_ood_metrics(...)` (AUROC, AUPRC, FPR@TPR=0.95).

Mapping OOD → hallucination detection

- Empirically, **OOD inputs** are more likely to trigger **hallucinations**.
- So you can use RMD as a **risk score**: before generation, compute RMD for the input; if above a threshold, **abstain**, **route to a stronger model**, or **ask for clarification / retrieval**.
- For **hallucination-labeled test sets** (0 = faithful, 1 = hallucinated), you can directly correlate the **RMD score** with hallucination labels to get AUROC/AUPRC.
- Stronger variant: **combine RMD (input OOD) with output perplexity** (or contrastive likelihood) after generation:
  - `final_score = α * RMD(input) + (1−α) * Perplexity(output | input)`
  - Calibrate α on a dev set.

How to produce those embeddings (the most important part)

You need a **consistent way to embed each input example** (same model, layer, pooling). The original method uses **final-layer hidden states** and simple pooling (e.g., mean over tokens).

Choose your base model & layer

- **Encoder–decoder (T5/BART)**: Use the **encoder final hidden states** for the **input text** (e.g., the question or document). Pool them (mean across tokens) → 1 vector per example.
- **Decoder-only (GPT-style)**: Feed the **prompt/input** only (no target); take **last-layer hidden states** for the prompt tokens; mean pool across tokens → 1 vector.
- **Multilingual/low-resource**: Use a multilingual variant of your base model if your datasets are low-resource languages.

Keep it consistent

- Same tokenizer, same max length/truncation strategy, same layer, same pooling for **all four splits**.
- Normalize if you like (e.g., L2-normalize vectors), but do it **everywhere**.

Hugging Face sketch (encoder example)

python 复制 编辑

```python
import torch, numpy as np
from transformers import AutoTokenizer, AutoModel

model_name = "google/mt5-base"  # or your task model
tok = AutoTokenizer.from_pretrained(model_name)
model = AutoModel.from_pretrained(model_name)
model.eval()

def embed_texts(texts, max_len=512, batch_size=16, device="cuda" if torch.cu
    model.to(device)
    outs = []
    with torch.no_grad():
        for i in range(0, len(texts), batch_size):
            batch = texts[i:i+batch_size]
            enc = tok(batch, padding=True, truncation=True, max_length=max_l
            out = model(**enc, output_hidden_states=True, return_dict=True)
            # final hidden states: out.last_hidden_state -> [B, T, D]
            h = out.last_hidden_state.mean(dim=1)  # mean pool over tokens –
            outs.append(h.cpu().numpy())
    return np.concatenate(outs, axis=0)  # [N, D]

# Example:
# embs_train_ind  = embed_texts(ind_train_texts)
# embs_train_ood  = embed_texts(ood_train_texts)
# embs_ind        = embed_texts(ind_test_texts)
# embs_ood        = embed_texts(ood_test_texts)
```

> If you're using an encoder–decoder model but want to include **conditioning on the target** (e.g., question+answer), you can (a) still use encoder embeddings of the input only (pure input OOD), or (b) concatenate input+generated summary/answer and embed that (post-hoc scoring). The original RMD is **input-conditioned OOD detection**, so (a) is the usual default.

Summary: Specific Steps:

1.load dataset

2.load pre–trained model (llama/opt)

3.model will generate response in text

4.use specific method to calculate score

4.1 adapt input: some methods need preprocess text to feature embeddings

4.2 adapt dataset into correct format

5. Evaluation: use auc–roc as metric, as described in Hallushift