CSE 472: Social Media Mining

Project II - Social Media Data Analysis

Zero-shot AI-generated text detection

Prof. Huan Liu
Due at 2023 November 24nd, 11:59PM

Name: Bala Sujith Potineni **Mentor:** Raha Moraffah **ASUID**: 1229564013

1. Abstract

In this project, I aimed to analyze a text-detection system called Detect-GPT that will enable the automatic identification of AI -generated material without the need for pre-existing training data. Detect-GPT uses the property that text sampled from a LLM typically lies in negative curvature regions (near local maxima) of the log probability function compared to its perturbated texts, and constructs a novel curvature-based criterion to determine if a passage is created from a particular LLM by utilizing this insight. According to the Detect-GPT, minor rewrites of human-written text may have a higher or lower log likelihood than the original sample. I intend to contradict this with a hypothesis that introduction of perturbation to a human-written text increases it's probability, thereby making the original text's log probability lying near a local minima. Based on this identification condition of human-written text, classification of AI-generated text from human-written text is achieved.

2. Introduction

2.1 Use of Transformers in Detect-GPT

As the AI-generated text is produced from an LLM, it is rational to choose an LLM model with a transformer architecture to identify generated from a model similar in architecture to it. Because of their ability to capture intricate linguistic patterns and nuances that separate human-written text from AI-generated text, large language models (LLMs) are increasingly being employed for AI-generated text identification. Traditional text recognition methods, such as keyword matching and rule-based systems, sometimes fall behind the increasing capabilities of LLMs, which can produce very sophisticated and human-like language.

LLMs have a number of characteristics that make them ideal for AI-generated text detection:

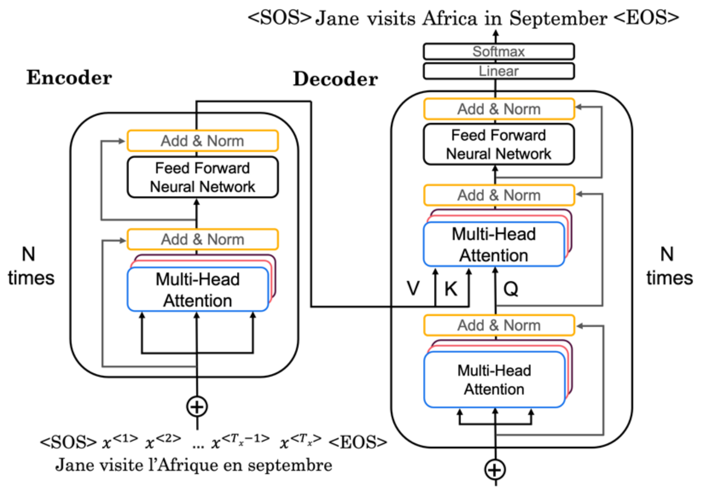
1) LLMs are trained on huge volumes of text data, allowing them to recognize the context and meaning of words inside a sentence or paragraph. With this contextual knowledge, they may discover patterns and abnormalities that may suggest AI-generated writing.

2) LLMs can analyze the semantic similarity of distinct pieces of text, letting them to determine whether a text sample is comparable to known AI-generated text styles or deviates from human-written language patterns.

3) As new AI-generated text develops, LLMs can continuously adjust and increase their detection capabilities. This adaptability is critical in the ever-changing world of AI-generated content.

4) LLMs can extract linguistic elements like as grammatical structures, word choices, and stylistic trends from text. These characteristics can be used to train machine learning models to tell the difference between human-written and AI-generated material.

2.2 Internal structure of LLM/ Transformers Architecture



<SOS> Jane visits Africa in September <EOS>
Softmax
Linear
Encoder
Decoder
Add & Norm
Feed Forward Neural Network
Add & Norm
Multi-Head Attention
N times
V
K
Q
<SOS> $x^{<1>}$ $x^{<2>}$ ... $x^{<T_x-1>}$ $x^{<T_x>}$ <EOS>
Jane visite l'Afrique en septembre

(Source: Coursera- Generative AI with LLMs)

<BeginAI - ChatGPT>

LLMs (Large Language Models) are neural network models that have been trained on enormous volumes of text data to generate prose that is human-quality. They rely on the Transformers architecture, which uses an attention mechanism to detect long-term relationships in text. This enables LLMs to comprehend the connections between words and phrases, resulting in more cohesive and meaningful output.

Here's a more in-depth explanation of how LLMs generate text combining the Transformers architecture with the attention mechanism:

Encoder:

1) Input Embedding: The input text is first converted into a sequence of word embeddings. These embeddings represent each word as a vector of numbers, capturing its semantic meaning.

2) Encoder Layers: The word embeddings are then passed through multiple encoder layers. Each encoder layer consists of two main components: a self-attention mechanism and a feed-forward network.

3) Self-attention Mechanism: The self-attention mechanism allows each word to attend to other words in the input sequence. This helps the model to understand the relationships between words and capture long-range dependencies.

4) Feed-forward Network: The feed-forward network applies a non-linear transformation to each position in the sequence, allowing the model to learn more complex patterns.

5) Hidden States: The output of the encoder layers is a sequence of hidden states. These hidden states represent the context and meaning of the input text, incorporating the relationships between words and phrases.

Decoder:

1. Decoder Layers: The hidden states from the encoder are then passed through multiple decoder layers. Each decoder layer also consists of a self-attention mechanism and a feed-forward network.

2. Decoder Self-attention Mechanism: The decoder self-attention mechanism allows each word in the output sequence to attend to other words in the generated text, ensuring coherence and consistency.

3. Encoder-Decoder Attention: The decoder also attends to the hidden states from the encoder. This allows the model to incorporate the context of the input text into the generated output.

4. Vocabulary Prediction: At each time step, the decoder predicts the next word in the output sequence. This prediction is based on the decoder's hidden state, the previously generated words, and the attention weights from both self-attention and encoder-decoder attention. It constructs a probability distribution for each word in vocabulary, representing how likely the word should be the next generated word.

5. Output Generation: The model repeatedly generates words based on the probability distributions for each word, based on set algorithms and parameters like top_k, top_p, temperature, etc until it reaches a predefined stopping criterion, such as reaching a maximum length or encountering a special end-of-sequence token.

The attention mechanism plays a crucial role in both the encoder and decoder, enabling the model to understand long-range dependencies and generate grammatically correct, semantically meaningful text. The transformer architecture, with its efficient parallelization capabilities, has made it possible to train and run large LLMs, leading to significant advancements in natural language processing.

<EndAI - ChatGPT>

2.3 Detect-GPT

Detect-GPT goal is to categorize whether a passage originated from a specific source model. Transformers log-probability is used in zero-shot AI-text detection in Detect-GPT to distinguish between human-written text and AI-generated text. This model makes use of local structure of the learned probability function around the passage, which contains the useful information of the passage. minor rewrites of human-written text may have a higher or lower log probability than the original sample, whereas minor rewrites of text generated by the model tend to have a lower log probability under the model. Put differently, unlike text created by humans, text generated by models typically appears in regions where the log probability function exhibits a negative curvature, such as in close proximity to local maxima of the log probability. This is because human-written text is typically less coherent and less likely to follow the same patterns as AI-generated text. Thus, when perturbations are introduced to Ai-generated text, it breaks the pattern, thus reducing the log probability. Whereas, for perturbed human-generated text, as it originally doesn't have specific pattern of generation, its log probability is uncertain, according to Detect-GPT.

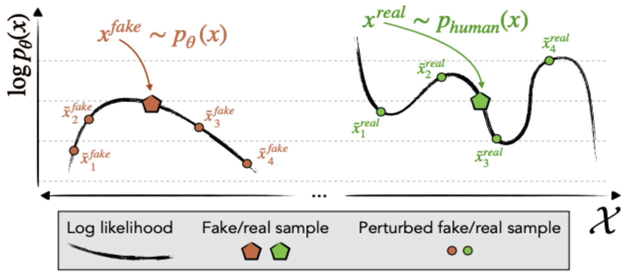


Figure 2. We identify and exploit the tendency of machine-generated passages $x \sim p_\theta(\cdot)$ **(left)** to lie in negative curvature regions of $\log p(x)$, where nearby samples have lower model log probability on average. In contrast, human-written text $x \sim p_{real}(\cdot)$ **(right)** tends not to occupy regions with clear negative log probability curvature; nearby samples may have higher or lower log probability.

(Source: DetectGPT: Zero-Shot Machine-Generated Text Detection using Probability Curvature)

To use transformers log-probability for zero-shot AI-text detection, the following steps are taken:

1. Compute the log probability of the input text under a transformer model.
2. Compute the average log probability of several perturbations of the input text.
3. Compare the log probability of the input text to the average log probability of the perturbations.

If the log probability of the input text is significantly lower than the average log probability of the perturbations, then the input text is likely to be human-written. Otherwise, the input text is likely to be AI-generated.

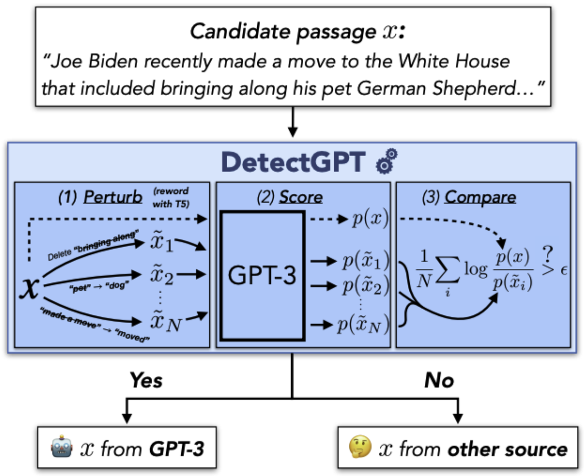


Figure 1. We aim to determine whether a piece of text was generated by a particular LLM p, such as GPT-3. To classify a candidate passage x, DetectGPT first generates minor **perturbations** of the passage $\tilde{x}_i$ using a generic pre-trained model such as T5. Then DetectGPT **compares** the log probability under p of the original sample x with each perturbed sample $\tilde{x}_i$. If the average log ratio is high, the sample is likely from the source model.

(Source: DetectGPT: Zero-Shot Machine-Generated Text Detection using Probability Curvature)

Detect-GPT is a zero-shot AI-text detection method that uses transformers log-probability to identify AI-generated text. Detect-GPT has been shown to be effective at detecting AI-generated text, even when the AI-generated text is of high quality.

2.4 Modification to Detect-GPT Hypothesis

If the perturbations are introduced to human-generated text, as the perturbations are generated by a model, they follow a specific pattern, and can thereby partially be considered an AI-generated text. Thus, these perturbations should have higher log probabilities compared to the original human-generated text. Based on this identification of human-generated passage, we can classify the AI-generated text as the converse.

3. Related Works

1) " Real or fake? Learning to discriminate machine from human generated text, 2019":

   This paper proposes a method for detecting text generated by large language models (LLMs) using a stylometric approach. Stylometry is a technique for analysing the writing style of a text to determine its author or origin. This method is based on the observation that LLMs tend to exhibit certain stylistic features that are different from those of human-written text. These features include "Higher frequency of rare words"," More frequent use of grammatical constructions" and "Less variation in sentence length". This method uses a machine learning algorithm to classify text as either human-written or LLM-generated based on the presence of these stylistic features. The results show that this method is able to achieve high accuracy in detecting LLM-generated text.

2) “Authorship attribution for neural text generation”:

This paper investigates the effectiveness of human collaboration in detecting text generated by LLMs. The authors conducted a study in which participants were asked to individually and collaboratively assess the likelihood that a given text was generated by an LLM. The results showed that human collaboration significantly improved the accuracy of LLM detection. The authors attribute this improvement to several factors like “Sharing of information about the stylistic features of the text”, “Different perspectives to identify wider range of stylistic features” and “reducing confirmation bias”. The findings suggested that human collaboration can be a valuable tool for detecting text generated by LLMs. This has implications for a variety of applications, such as spam filtering, fake news detection, and social media moderation.

3) "Automatic Detection of Machine Generated Text: A Critical Survey":

This paper provides a comprehensive overview of the field of machine-generated text (MGT) detection. The authors discuss the challenges and opportunities of detecting MGT, and they review a variety of state-of-the-art MGT detection approaches.

4) “A watermark for large language models, 2023”:

This proposes a novel method for embedding watermarks into text generated by large language models (LLMs). Watermarks are hidden signals that can be used to identify the source of a piece of text, even if the text has been modified or rephrased. The proposed method works by embedding a small number of "green" tokens into the text before each word is generated. These green tokens are chosen to be semantically similar to the words they are replacing, so that they do not significantly alter the meaning of the text. However, the green tokens are also chosen to be unique and unpredictable, so that they can be easily detected by an algorithm. To detect the presence of a watermark, an algorithm simply needs to scan the text for the presence of green tokens. If enough green tokens are found, then the algorithm can conclude that the text was generated by an LLM that was trained with the watermark.

4. Model Description and Implementation

NOTE: The code for this project is updated after cloning from https://github.com/eric-mitchell/detect-gpt/tree/main

4.1 Implementational Challenges

a) Size of source model and mask-filling models like gpt2-xl, EleutherAI/gpt-j-6B, t5-3b, etc has billions of parameters which are used in processing, leading to RAM crashes below 100 compute units in Google Colab Pro. Hence, “gpt2” (has 117M parameters) and “t5-small” (has 60M parameters) are used as base-model and mask-filling models for this project.

b) ‘CUDA’ supported pytorch library is unavailable for MacOS. Thus, models are loaded to cpu not utilising the distribute computing, increasing the run time.

4.2 Detect-GPT

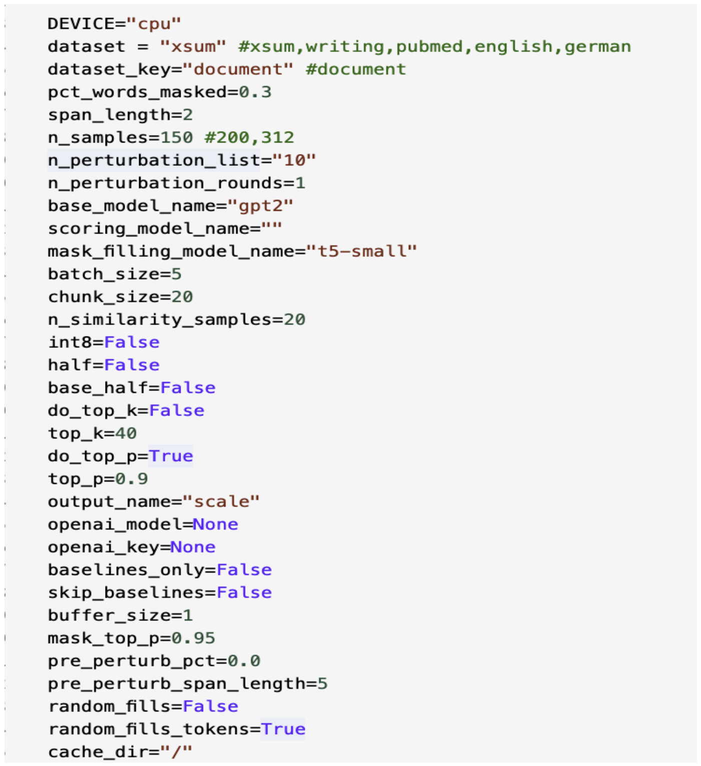
Detect-GPT relies on the idea that instances generated from a source model, denoted as p_θ, tend to exist in regions characterized by negative curvature within the log probability function of p_θ, a trait not commonly observed in human-generated text. When making slight modifications (perturbations) to a passage x drawn from p_θ, resulting in $\tilde{x}$, the difference between log p_θ (x) and log p_θ ($\tilde{x}$) should, on average, be more pronounced for machine-generated samples compared to human-written text. To apply this concept, consider a perturbation function, q(· | x), which provides a distribution over $\tilde{x}$—slightly altered versions of x that retain a similar meaning. For example, q(· | x) could involve a human rewriting one of the sentences in x while preserving its meaning. With this perturbation function in mind, the perturbation discrepancy, denoted as d (x, p_θ, q), is defined as the difference between log p_θ (x) and the expected log probability of the perturbed versions, offering a measure of the deviation from the model's norm for both machine-generated and human-written text.

(Information Source: DetectGPT: Zero-Shot Machine-Generated Text Detection using Probability Curvature**)**

In this project, we consider q(· | x) as the samples produced from "t5-small" mask-filling model to apply perturbations for a given text, x.

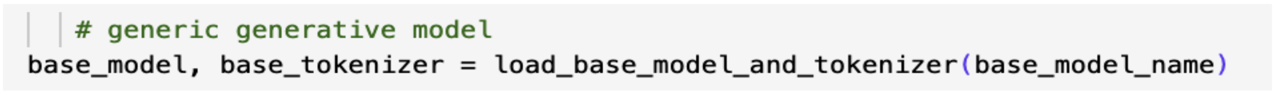
4.3 Detect-GPT Code Analysis

1) Input the hyperparameters like number of percentage of words masked in encoder(pct_words_masked), number of tokens masked (span_length), number of samples (n_samples), list of numbers of perturbations on a single passage (n_perturbation_list), source model (base_model_name), scoring model which is used for calculating log probabilities on base-model sampled text instead of base model (scoring_model_name), model that introduces perturbations in passage (mask-filling model), top_p(selecting completion with in options of probability sum top_p), top_k (selecting completion among top k options with highest probabilities), etc.



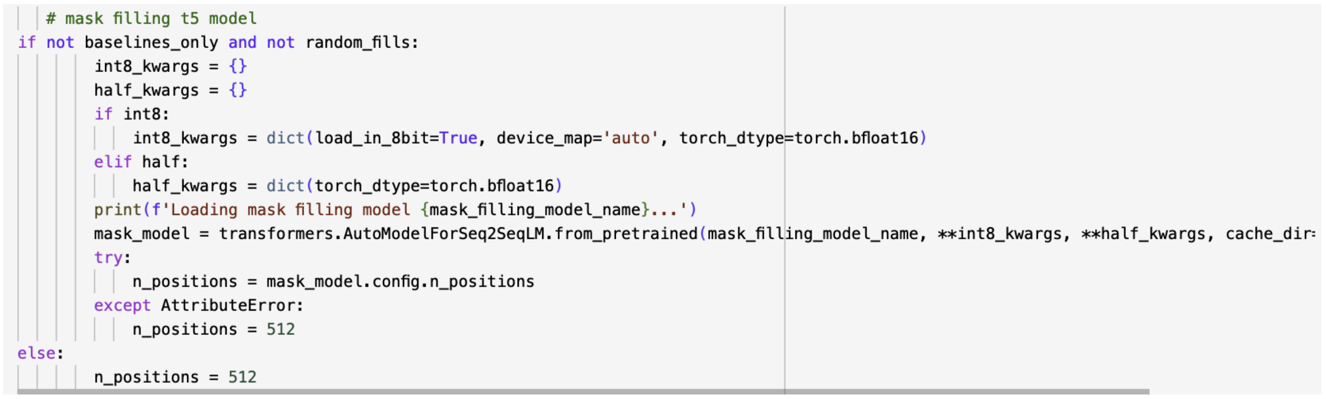
```
DEVICE="cpu"
dataset = "xsum" #xsum,writing,pubmed,english,german
dataset_key="document" #document
pct_words_masked=0.3
span_length=2
n_samples=150 #200,312
n_perturbation_list="10"
n_perturbation_rounds=1
base_model_name="gpt2"
scoring_model_name=""
mask_filling_model_name="t5-small"
batch_size=5
chunk_size=20
n_similarity_samples=20
int8=False
half=False
base_half=False
do_top_k=False
top_k=40
do_top_p=True
top_p=0.9
output_name="scale"
openai_model=None
openai_key=None
baselines_only=False
skip_baselines=False
buffer_size=1
mask_top_p=0.95
pre_perturb_pct=0.0
pre_perturb_span_length=5
random_fills=False
random_fills_tokens=True
cache_dir="/"
```

2) Load source model which generates the text as base model and similarly, its tokenizer. I chose ‘gpt2’ model as base_model and tokenizer.



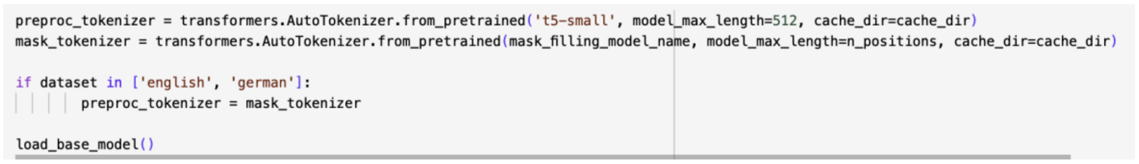
```
    # generic generative model
base_model, base_tokenizer = load_base_model_and_tokenizer(base_model_name)
```

3) Load the mask-filling model, specifically “t5-small”, that perturbates texts in this project.



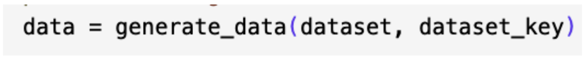
```
    # mask filling t5 model
if not baselines_only and not random_fills:
        int8_kwargs = {}
        half_kwargs = {}
        if int8:
            int8_kwargs = dict(load_in_8bit=True, device_map='auto', torch_dtype=torch.bfloat16)
        elif half:
            half_kwargs = dict(torch_dtype=torch.bfloat16)
        print(f'Loading mask filling model {mask_filling_model_name}...')
        mask_model = transformers.AutoModelForSeq2SeqLM.from_pretrained(mask_filling_model_name, **int8_kwargs, **half_kwargs, cache_dir=
        try:
            n_positions = mask_model.config.n_positions
        except AttributeError:
            n_positions = 512
else:
        n_positions = 512
```

4) Load the mask-filling tokenizer for preprocessing words to tokens, specifically “t5-small” here, and load the base model to cpu.

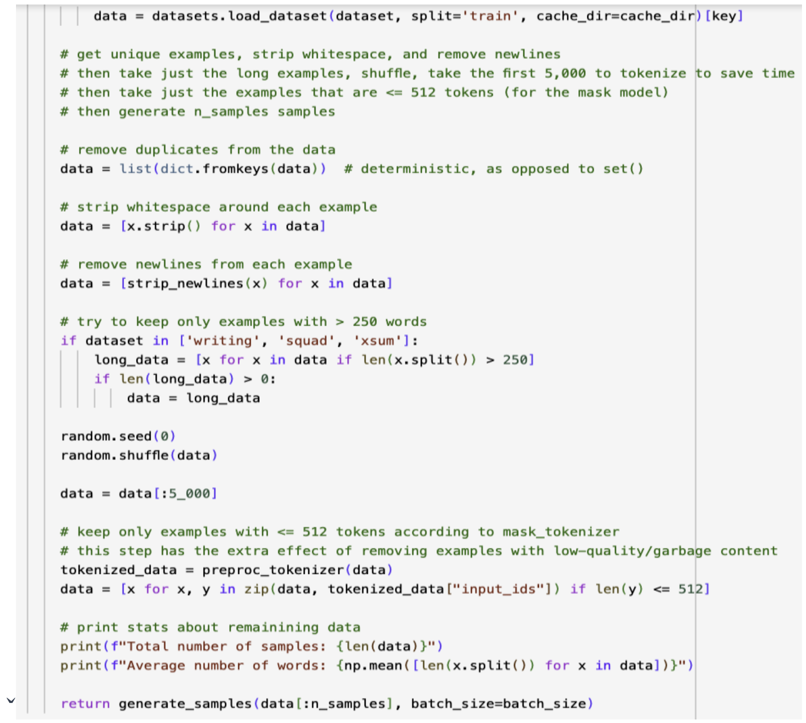


```
preproc_tokenizer = transformers.AutoTokenizer.from_pretrained('t5-small', model_max_length=512, cache_dir=cache_dir)
mask_tokenizer = transformers.AutoTokenizer.from_pretrained(mask_filling_model_name, model_max_length=n_positions, cache_dir=cache_dir)

if dataset in ['english', 'german']:
        preproc_tokenizer = mask_tokenizer

load_base_model()
```

5) Pre-process the data (loaded from datasets library), i.e, limiting the number of words of text, sampling the original text by feeding the text the text to the base-model to generate the samples, replacing or stripping unnecessary tokens, etc..



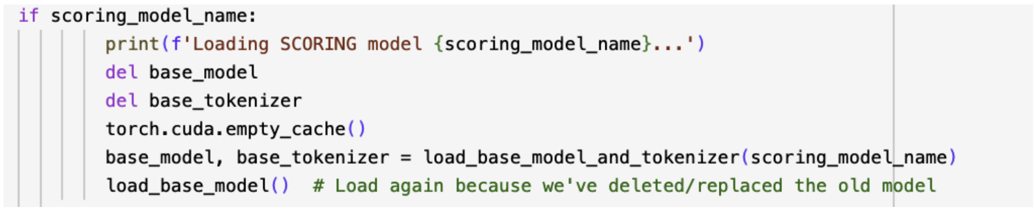
```
data = generate_data(dataset, dataset_key)
```



```
        data = datasets.load_dataset(dataset, split='train', cache_dir=cache_dir)[key]

    # get unique examples, strip whitespace, and remove newlines
    # then take just the long examples, shuffle, take the first 5,000 to tokenize to save time
    # then take just the examples that are <= 512 tokens (for the mask model)
    # then generate n_samples samples

    # remove duplicates from the data
    data = list(dict.fromkeys(data))  # deterministic, as opposed to set()

    # strip whitespace around each example
    data = [x.strip() for x in data]

    # remove newlines from each example
    data = [strip_newlines(x) for x in data]

    # try to keep only examples with > 250 words
    if dataset in ['writing', 'squad', 'xsum']:
        long_data = [x for x in data if len(x.split()) > 250]
        if len(long_data) > 0:
            data = long_data

    random.seed(0)
    random.shuffle(data)

    data = data[:5_000]

    # keep only examples with <= 512 tokens according to mask_tokenizer
    # this step has the extra effect of removing examples with low-quality/garbage content
    tokenized_data = preproc_tokenizer(data)
    data = [x for x, y in zip(data, tokenized_data["input_ids"]) if len(y) <= 512]

    # print stats about remainining data
    print(f"Total number of samples: {len(data)}")
    print(f"Average number of words: {np.mean([len(x.split()) for x in data])}")

    return generate_samples(data[:n_samples], batch_size=batch_size)
```

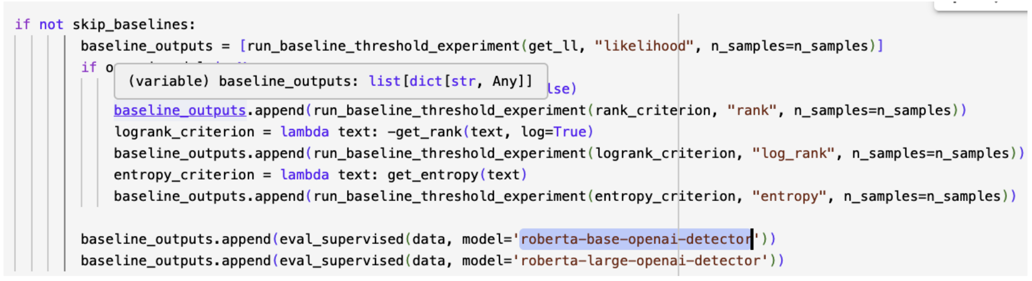
-

6) If a model (scoring model) other than base model is used to evaluate the log probabilities, ,load it as base model and corresponding tokenizer.

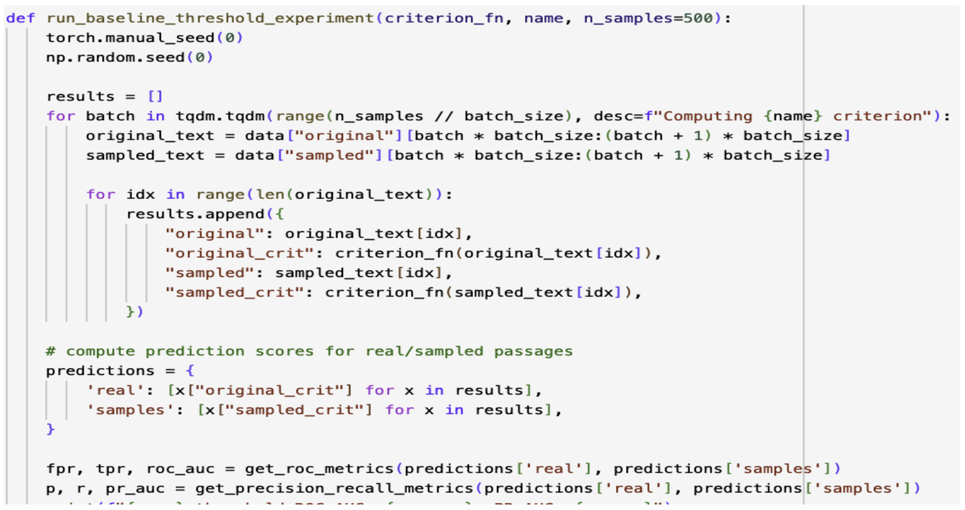


```
if scoring_model_name:
        print(f'Loading SCORING model {scoring_model_name}...')
        del base_model
        del base_tokenizer
        torch.cuda.empty_cache()
        base_model, base_tokenizer = load_base_model_and_tokenizer(scoring_model_name)
        load_base_model()  # Load again because we've deleted/replaced the old model
```

7) Run the baseline threshold experiments on data, which calculate the original and sampled text's log probabilities, entropy, rank and log rank. These results are used to calculate their evaluation metrics like Area under curve(AUC) and Precision-Recall values. The data is also used to classify texts based on pretrained supervised models like "roberta-base-openai-detector" and "roberta-large-openai-detector" to compare with DetectGPT later.

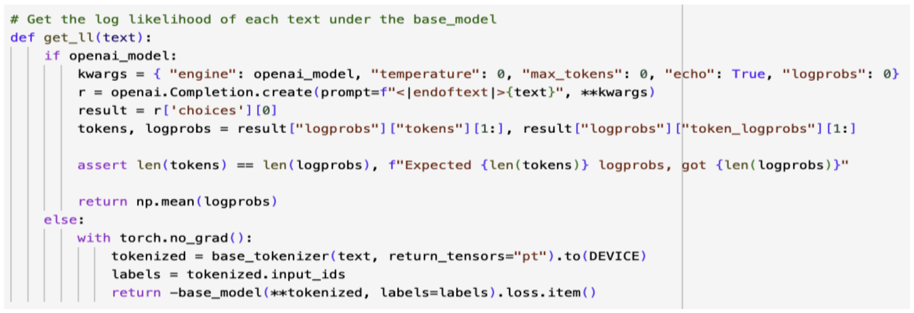


```
if not skip_baselines:
        baseline_outputs = [run_baseline_threshold_experiment(get_ll, "likelihood", n_samples=n_samples)]
        if o (variable) baseline_outputs: list[dict[str, Any]]
                                                                lse)
            baseline_outputs.append(run_baseline_threshold_experiment(rank_criterion, "rank", n_samples=n_samples))
            logrank_criterion = lambda text: -get_rank(text, log=True)
            baseline_outputs.append(run_baseline_threshold_experiment(logrank_criterion, "log_rank", n_samples=n_samples))
            entropy_criterion = lambda text: get_entropy(text)
            baseline_outputs.append(run_baseline_threshold_experiment(entropy_criterion, "entropy", n_samples=n_samples))

        baseline_outputs.append(eval_supervised(data, model='roberta-base-openai-detector'))
        baseline_outputs.append(eval_supervised(data, model='roberta-large-openai-detector'))
```



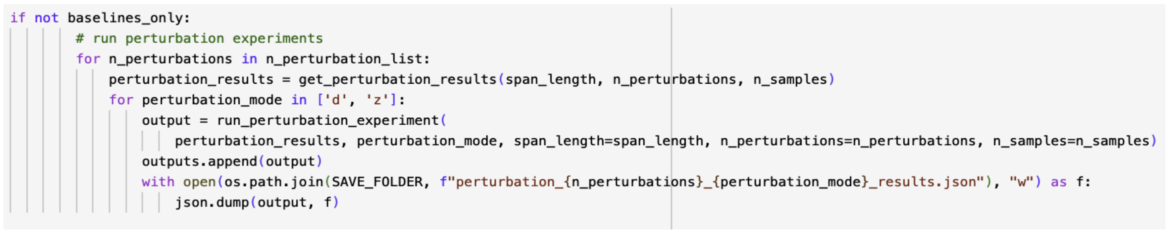
```python
def run_baseline_threshold_experiment(criterion_fn, name, n_samples=500):
    torch.manual_seed(0)
    np.random.seed(0)

    results = []
    for batch in tqdm.tqdm(range(n_samples // batch_size), desc=f"Computing {name} criterion"):
        original_text = data["original"][batch * batch_size:(batch + 1) * batch_size]
        sampled_text = data["sampled"][batch * batch_size:(batch + 1) * batch_size]

        for idx in range(len(original_text)):
            results.append({
                "original": original_text[idx],
                "original_crit": criterion_fn(original_text[idx]),
                "sampled": sampled_text[idx],
                "sampled_crit": criterion_fn(sampled_text[idx]),
            })

    # compute prediction scores for real/sampled passages
    predictions = {
        'real': [x["original_crit"] for x in results],
        'samples': [x["sampled_crit"] for x in results],
    }

    fpr, tpr, roc_auc = get_roc_metrics(predictions['real'], predictions['samples'])
    p, r, pr_auc = get_precision_recall_metrics(predictions['real'], predictions['samples'])
```

‘get_ll’ function is defined to calculate the log probability of text from the scoring model.

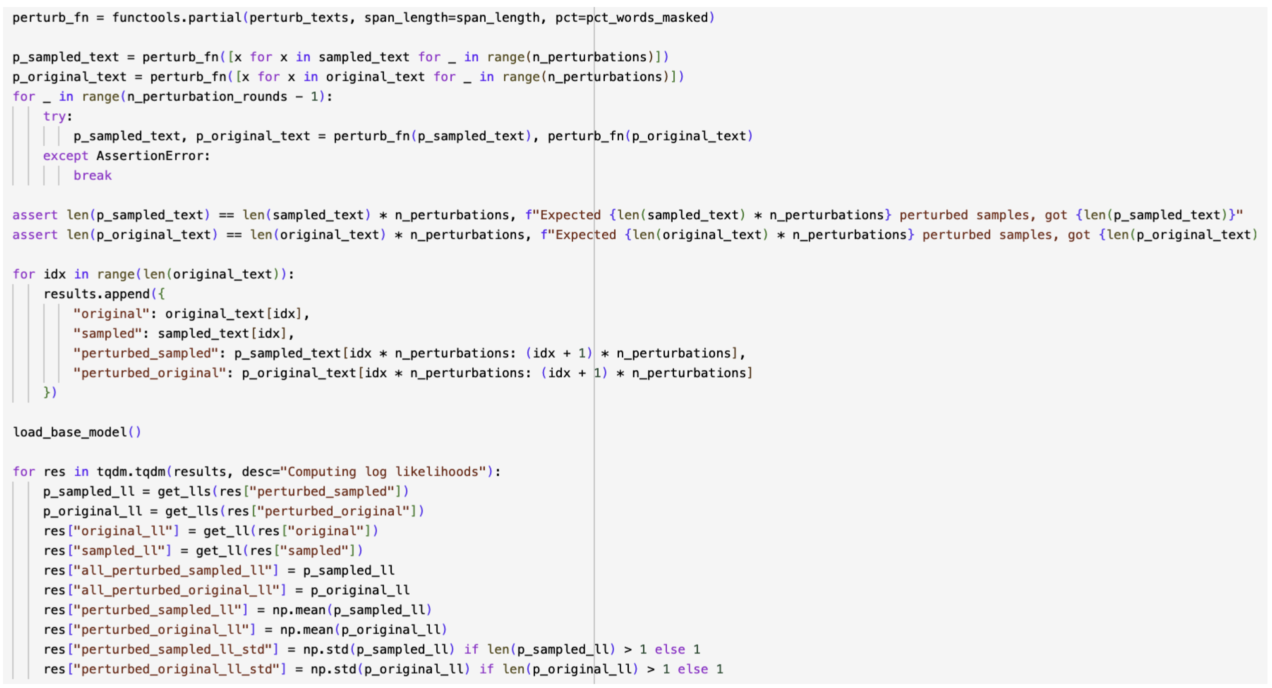


```python
# Get the log likelihood of each text under the base_model
def get_ll(text):
    if openai_model:
        kwargs = { "engine": openai_model, "temperature": 0, "max_tokens": 0, "echo": True, "logprobs": 0}
        r = openai.Completion.create(prompt=f"<|endoftext|>{text}", **kwargs)
        result = r['choices'][0]
        tokens, logprobs = result["logprobs"]["tokens"][1:], result["logprobs"]["token_logprobs"][1:]

        assert len(tokens) == len(logprobs), f"Expected {len(tokens)} logprobs, got {len(logprobs)}"

        return np.mean(logprobs)
    else:
        with torch.no_grad():
            tokenized = base_tokenizer(text, return_tensors="pt").to(DEVICE)
            labels = tokenized.input_ids
            return -base_model(**tokenized, labels=labels).loss.item()
```

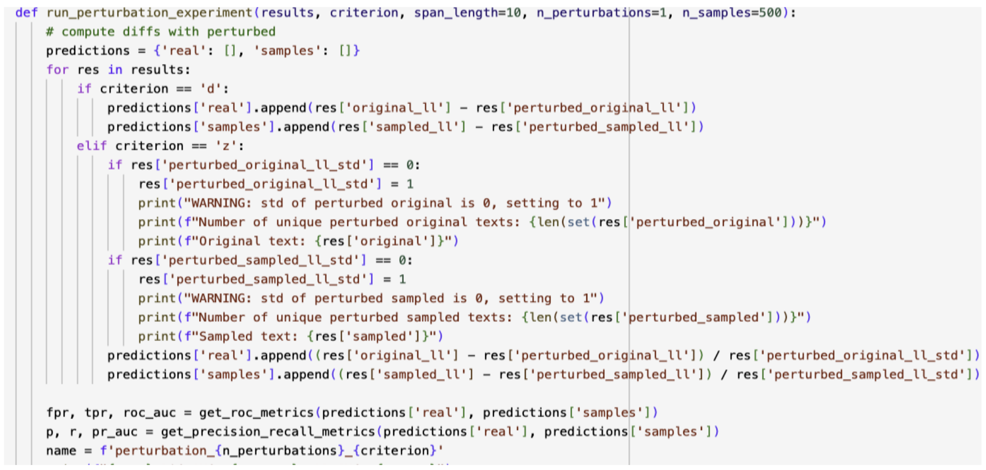
8) Run the perturbation experiments after getting perturbation results, i.e calculating the log probabilities of generated perturbed texts and evaluating the perturbation discrepancies based on their mean. Concurrently normalized discrepancies using their standard deviation are also measured to analyse its effect on the model later.



```python
if not baselines_only:
        # run perturbation experiments
        for n_perturbations in n_perturbation_list:
            perturbation_results = get_perturbation_results(span_length, n_perturbations, n_samples)
            for perturbation_mode in ['d', 'z']:
                output = run_perturbation_experiment(
                    perturbation_results, perturbation_mode, span_length=span_length, n_perturbations=n_perturbations, n_samples=n_samples)
                outputs.append(output)
                with open(os.path.join(SAVE_FOLDER, f"perturbation_{n_perturbations}_{perturbation_mode}_results.json"), "w") as f:
                    json.dump(output, f)
```



```python
perturb_fn = functools.partial(perturb_texts, span_length=span_length, pct=pct_words_masked)

p_sampled_text = perturb_fn([x for x in sampled_text for _ in range(n_perturbations)])
p_original_text = perturb_fn([x for x in original_text for _ in range(n_perturbations)])
for _ in range(n_perturbation_rounds - 1):
    try:
        p_sampled_text, p_original_text = perturb_fn(p_sampled_text), perturb_fn(p_original_text)
    except AssertionError:
        break

assert len(p_sampled_text) == len(sampled_text) * n_perturbations, f"Expected {len(sampled_text) * n_perturbations} perturbed samples, got {len(p_sampled_text)}"
assert len(p_original_text) == len(original_text) * n_perturbations, f"Expected {len(original_text) * n_perturbations} perturbed samples, got {len(p_original_text)

for idx in range(len(original_text)):
    results.append({
        "original": original_text[idx],
        "sampled": sampled_text[idx],
        "perturbed_sampled": p_sampled_text[idx * n_perturbations: (idx + 1) * n_perturbations],
        "perturbed_original": p_original_text[idx * n_perturbations: (idx + 1) * n_perturbations]
    })

load_base_model()

for res in tqdm.tqdm(results, desc="Computing log likelihoods"):
    p_sampled_ll = get_lls(res["perturbed_sampled"])
    p_original_ll = get_lls(res["perturbed_original"])
    res["original_ll"] = get_ll(res["original"])
    res["sampled_ll"] = get_ll(res["sampled"])
    res["all_perturbed_sampled_ll"] = p_sampled_ll
    res["all_perturbed_original_ll"] = p_original_ll
    res["perturbed_sampled_ll"] = np.mean(p_sampled_ll)
    res["perturbed_original_ll"] = np.mean(p_original_ll)
    res["perturbed_sampled_ll_std"] = np.std(p_sampled_ll) if len(p_sampled_ll) > 1 else 1
    res["perturbed_original_ll_std"] = np.std(p_original_ll) if len(p_original_ll) > 1 else 1
```



```python
def run_perturbation_experiment(results, criterion, span_length=10, n_perturbations=1, n_samples=500):
    # compute diffs with perturbed
    predictions = {'real': [], 'samples': []}
    for res in results:
        if criterion == 'd':
            predictions['real'].append(res['original_ll'] - res['perturbed_original_ll'])
            predictions['samples'].append(res['sampled_ll'] - res['perturbed_sampled_ll'])
        elif criterion == 'z':
            if res['perturbed_original_ll_std'] == 0:
                res['perturbed_original_ll_std'] = 1
                print("WARNING: std of perturbed original is 0, setting to 1")
                print(f"Number of unique perturbed original texts: {len(set(res['perturbed_original']))}")
                print(f"Original text: {res['original']}")
            if res['perturbed_sampled_ll_std'] == 0:
                res['perturbed_sampled_ll_std'] = 1
                print("WARNING: std of perturbed sampled is 0, setting to 1")
                print(f"Number of unique perturbed sampled texts: {len(set(res['perturbed_sampled']))}")
                print(f"Sampled text: {res['sampled']}")
            predictions['real'].append((res['original_ll'] - res['perturbed_original_ll']) / res['perturbed_original_ll_std'])
            predictions['samples'].append((res['sampled_ll'] - res['perturbed_sampled_ll']) / res['perturbed_sampled_ll_std'])

    fpr, tpr, roc_auc = get_roc_metrics(predictions['real'], predictions['samples'])
    p, r, pr_auc = get_precision_recall_metrics(predictions['real'], predictions['samples'])
    name = f'perturbation_{n_perturbations}_{criterion}'
```

5. Experiments

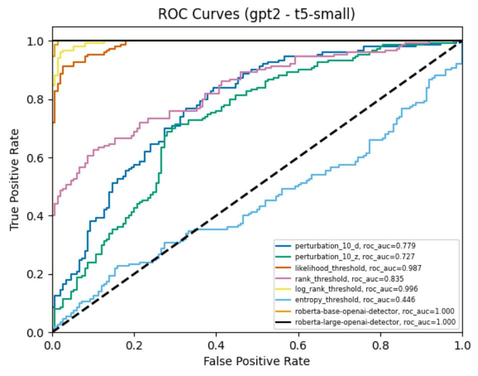
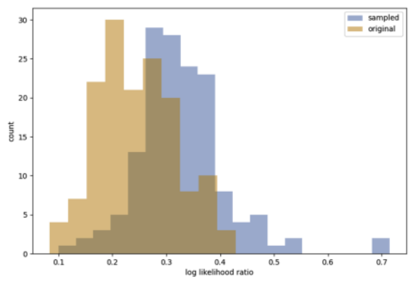
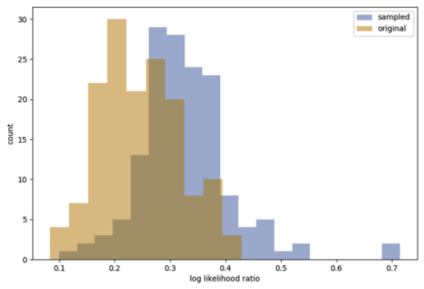
The experiments are conducted to understand the distribution of log probabilities for perturbations compared to that of original text. If it is a human-generated text, its perturbated text's discrepancies should be negative. Along with this, we also study Detect-GPT's performance in comparison to the supervised models and other zero-shot scoring criteria.

5.1 Dataset – "multi_news" (Human-written text)

- "multi_news", consists of news articles and human-written summaries of these articles from the site newser.com. Each summary is professionally written by editors and includes links to the original articles cited.

- Results:

Number of perturbations = 10

ROC Curve	
Receiver Operating Characteristic curve's Area_under_curve (roc_auc)	0.7791555555555555
Precison Recall's AUC	0.7601506109169298
Loss	0.23984938908307019
Histogram of mean discrepancies('d') of samples	
Histogram of mean normalized discrepancies('z') of samples	

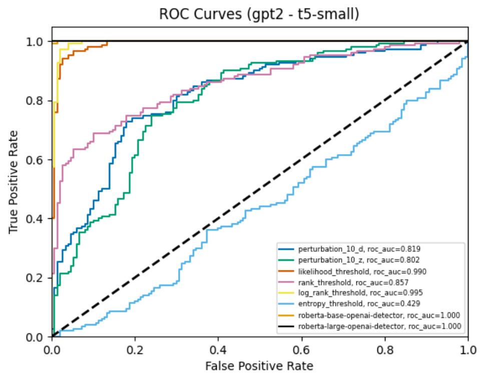
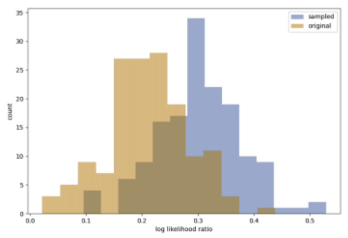
Based on the above results, we can conclude that the average discrepancy for the human-written text isn't negative, contradicting our hypothesis.

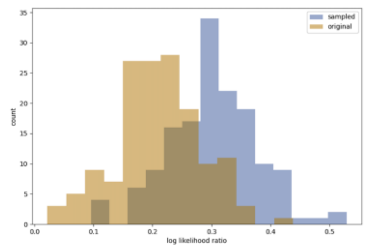
5.2 Dataset – “XSum” (Human-written text)

- “xsum”, consists of 226,711 news articles paired with one-sentence summaries. The articles were collected from BBC News articles published between 2010 and 2017 and cover a wide range of topics, including news, politics, sports, weather, business, technology, science, health, family, education, and entertainment.

- Results:

  Number of perturbations = 10

ROC Curve	
Receiver Operating Characteristic curve’s Area_under_curve (roc_auc)	0.8187999999999999
Precison Recall’s AUC	0.8102192576055209
Loss	0.18978074239447906
Histogram of mean discrepancies(‘d’) of samples	

Histogram of mean normalized discrepancies('z') of samples	

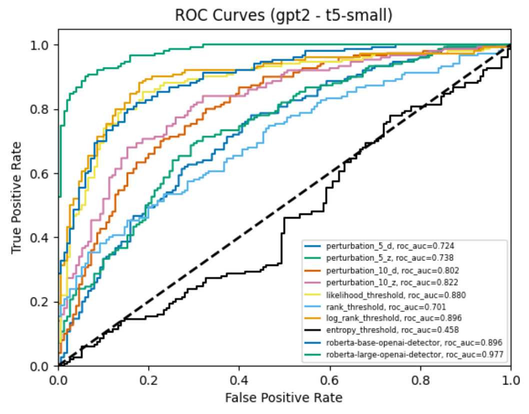
Based on the above results, we can conclude that the average discrepancy for the human-written text isn't negative, contradicting our hypothesis.

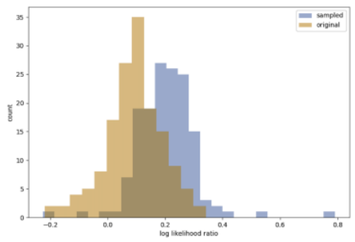
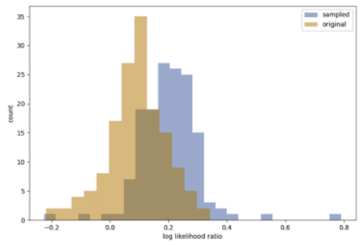
5.3 Dataset – "PubMedQA" (Human-written text ← questions)

- "PubMedQA", is a question answering (QA) dataset for biomedical research consisting of 1,000 expert-annotated QA instances, 61,200 unlabeled QA instances, and 211,300 artificially generated QA instances.

- Results:

  Number of perturbations = 10; span length = 2

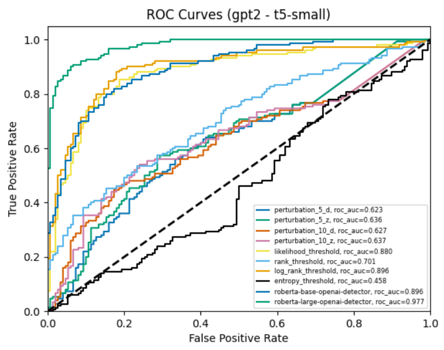
ROC Curve	
Receiver Operating Characteristic curve's Area_under_curve (roc_auc)	0.8019333333333334
Precison Recall's AUC	0.775930803508969

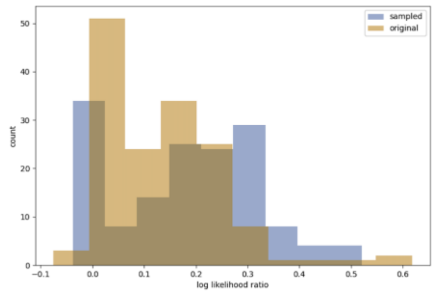
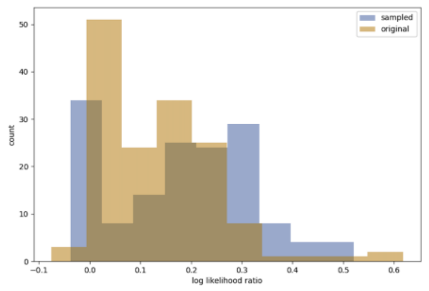
Loss	0.22406919649103096
Histogram of mean discrepancies('d') of samples	
Histogram of mean normalized discrepancies('z') of samples	

Based on the above results, we can conclude that the average discrepancy for the human-written text isn't negative, but the distribution is spread to the negative region. So, let's try varying span length of perturbation on a hypothesis that it might better identify AI-text patterns in perturbation, thereby increasing perturbations log probability.

- Results:

Number of perturbations = 10; span length = 10

ROC Curve	

Receiver Operating Characteristic curve's Area_under_curve (roc_auc)	0.6274888888888889
Precison Recall's AUC	0.6307585845946369
Loss	0.36924141540536315
Histogram of mean discrepancies('d') of samples	
Histogram of mean normalized discrepancies('z') of samples	

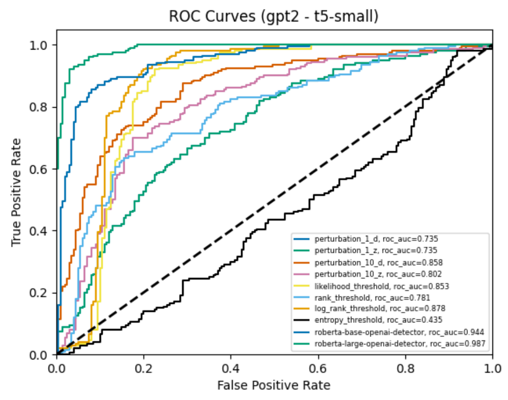
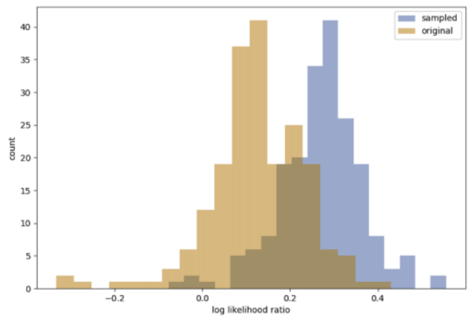
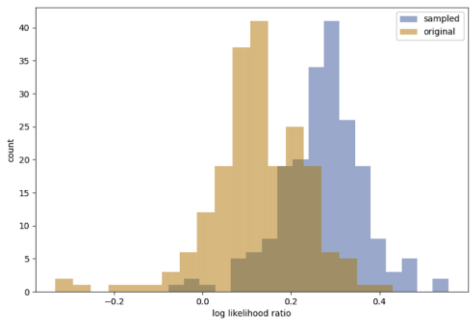
We can observe that increasing the span length had the opposite effect. The reason for this might be that mask-filling model (t5-small) is different from source model and thus not correctly identifying perturbation text AI-text similarities of different model?

5.4 Dataset – "WMT16" (AI-generated text)

- "wmt16", dataset is a large collection of parallel text data used for machine translation research consists of approximately 4.8 million sentence pairs in 7 language pairs, covering a variety of genres, including news, IT, and biomedical texts. For this project, I used translated English sentences as inputs.

- Results:

  Number of perturbations = 10

ROC Curve	
Receiver Operating Characteristic curve's Area_under_curve (roc_auc)	0.8577750000000001
Precison Recall's AUC	0.8483220404199259
Loss	0.15167795958007413
Histogram of mean discrepancies('d') of samples	
Histogram of mean normalized discrepancies('z') of samples	

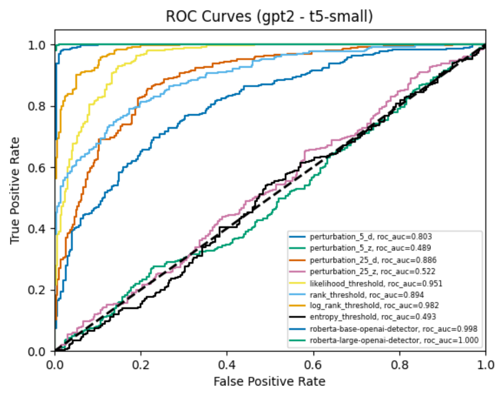
Based on the above results, we can conclude that the average discrepancy for the generated text is positive as expected by Detect-GPT.

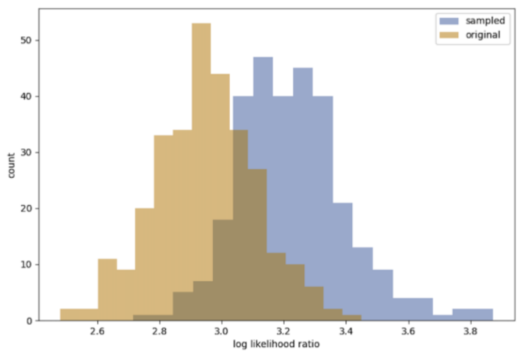
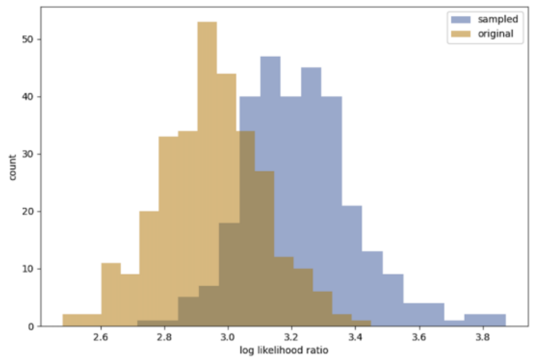
5.5 Dataset – "SQuAD" (AI-generated text)

- The Stanford Question Answering Dataset (SQuAD) is a collection of question-answer pairs (QnA) derived from Wikipedia articles, which are machine generated passages.

- Results:

  Number of perturbations = 25

ROC Curve	
Receiver Operating Characteristic curve's Area_under_curve (roc_auc)	0.8858111111111112
Precison Recall's AUC	0.878975900256652
Loss	0.12102409974334805

Histogram of mean discrepancies('d') of samples	
Histogram of mean normalized discrepancies('z') of samples	

Based on the above results, we can conclude that the average discrepancy for the generated text is highly positive as expected by Detect-GPT.

5.6 General Conclusions

- Based on the observations so far, the hypothesis that log probability of human-written text lying in positive curvature with respect to its perturbated texts doesn't hold true. This might be due insufficient scope of experiments or the affect of hyperparameters chosen.
- Normalization of log probabilities didn't significantly increase the performance of the model.(based on roc_auc and histograms)
- The ROC_AUC isn't matching up to supervised models as justified by the result noted in Detect-GPT paper on scope of the source model and mask-filling model being less.

6. Future Work

- Conduct more and thorough experiment varying hyperparameters and other parametera and analyze their affect on the model.
- Work on prompts fed to the decoder of scoring model to increase the performance.

7. References

a) “DetectGPT: Zero-Shot Machine-Generated Text Detection using Probability Curvature” by Eric Mitchell. Link: https://arxiv.org/abs/2301.11305

b) LLM for learning reference - ChatGPT : https://chat.openai.com/

c) Python packages:

Transformers - library for working with NLP tasks using transformer-based models providing a wide range of pre-trained models for tasks such as text classification, named entity recognition, machine translation, and more.

Pytorch – library that provides a flexible and dynamic computational graph, making it well-suited for research and experimentation in deep learning.

Datasets - provide a collection of high-quality datasets for natural language processing tasks.

d) Generative AI course in coursera: https://www.coursera.org/learn/generative-ai-with-llms/

e) "Automatic Detection of Machine Generated Text: A Critical Survey" by Jawahar. Link: https://aclanthology.org/2020.coling-main.208.pdf