Invisible Entropy: Towards Safe and Efficient Low-Entropy LLM Watermarking

Anonymous ACL submission

Abstract

Logit-based LLM watermarking traces and verifies AI-generated content by maintaining green and red token lists and increasing the likelihood of green tokens during generation. However, it fails in low-entropy scenarios, where predictable outputs make green token selection difficult without disrupting natural text flow. Existing approaches address this by assuming access to the original LLM to calculate entropy and selectively watermark high-entropy tokens. However, these methods face two major challenges: (1) high computational costs and detection delays due to reliance on the original LLM, and (2) potential risks of model leakage. To address these limitations, we propose Invisible Entropy (IE), a watermarking paradigm designed to enhance both safety and efficiency. Instead of relying on the original LLM, IE introduces a lightweight feature extractor and an entropy tagger to predict whether the entropy of the next token is high or low. Furthermore, based on theoretical analysis, we develop a threshold navigator that adaptively sets entropy thresholds. It identifies a threshold where the watermark ratio decreases as the green token count increases, enhancing the naturalness of the watermarked text and improving detection robustness. Experiments on HumanEval and MBPP datasets demonstrate that IE reduces parameter size by 99% while achieving performance on par with state-of-the-art methods. Our work introduces a safe and efficient paradigm for low-entropy watermarking. IE-official-repo

1 Introduction

Textual watermarking, which aims to embed subtle patterns in the generated text to make it detectable by algorithms but invisible to humans, is an important step towards trustworthy AI. It can be applied at various stages, including logits generation (Kirchenbauer et al., 2023), token sampling (Christ et al., 2024), and training (Sun et al., 2022, 2023; Gu et al., 2024). Logit-based watermarking is cost-efficient, modifying probabilities before token selection without adding training or sampling steps (Liu et al., 2024).

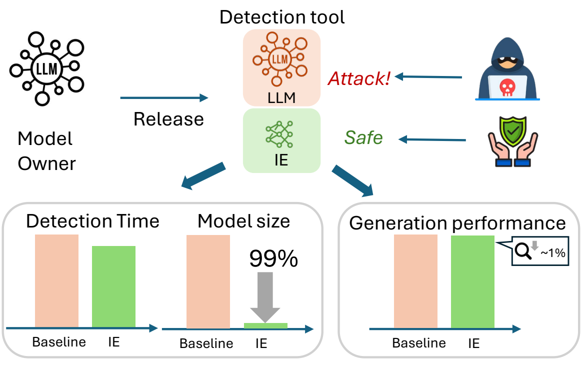


Figure 1: Existing watermarking methods in low-entropy scenarios face safety and cost challenges, while our method addresses them efficiently and securely.

As a pioneering work in logit-based watermarking, Kirchenbauer et al. (2023) introduced KGW, the first logit-based watermarking approach. This method partitions the vocabulary into green and red lists based on the previous token and a hash key, then boosts the logits of the green list to embed the watermark and decreases the probabilities of tokens outside this green list (red list). However, KGW fails in ***low-entropy scenarios*** where the next token is highly predictable, such as the prompt "import numpy as" almost certainly leading to"np" (entropy 0.048). If this expected token is placed in the red list, two issues may arise: (1) If the model still selects it despite the reduced probability, the unexpected inclusion of a red-list token may weaken the watermark's detectability. (2) If the model instead picks a green-list token due to the boosted logits, it may disrupt text fluency. Similarly, if the expected token is directly placed in the green list, it may lead to a *false inflation* of green-list token frequency in generated text, thereby increasing the likelihood of misclassify human-written content as machine-generated. As a result, the watermark detection system becomes less reliable.

To address the low-entropy problem, Lee et al. (2024) proposed SWEET, which applies watermarks only to high-entropy tokens, preserving text quality. Similarly, Lu et al. (2024) introduced EWD, which enhances detection by assigning higher weights to high-entropy tokens. However, these entropy-based watermarking methods face a critical limitation: *they assume the detector has access to the original LLM to calculate entropy.* This reliance on the original model introduces several challenges, as illustrated in Fig. 1. First, providing the original model to third parties poses significant risks of model leakage, potentially leading to unintended exposure or unauthorized access (Song and Raghunathan, 2020; Duc et al., 2014). Second, using the original LLM incurs substantial computational costs, particularly when processing large-scale datasets or running multiple detections.

Using a proxy model to approximate entropy calculation is potentially feasible. SWEET replaces the original model, e.g., LLaMA2-13B (Touvron et al., 2023), with a smaller model from the same family, e.g., LLaMA2-7B, for entropy estimation. Although this practice outperforms KGW, it still suffers from significant performance degradation. While the original EWD work does not explicitly explore the use of proxy models, our experimental results in Tab. 1 show a similar trend. It is also important to note that the effectiveness of a proxy model heavily depends on its architectural similarity to the original model.

Motivated by this, we attempt to train a lightweight proxy model to eliminate the dependency on the original LLM during entropy-based watermark detection. Our experiments in App. C show that regressing continuous entropy using an MLP is challenging, but reframing the task as a classification problem – determining whether the entropy of next token exceeds a given threshold – is more feasible. Considering the aforementioned issue that proxy models rely on architectural similarity to the original model, we introduce a Unified Feature Extractor that converts prefix tokens into a unified feature representation using a token translator and an embedding model, thereby ensuring compatibility across different LLMs and tokenizers. When using a fixed threshold to distinguish high- and low-entropy tokens, we observe that applying the same threshold across all samples ignores inter-sample variability. Moreover, in practical scenarios, the generator and the detector cannot share threshold, which limits the applicability of watermarking methods. To balance the naturalness of generated text and watermark detectability, we propose a sample-level entropy threshold optimization method. We evaluate our method in a representative low-entropy setting, namely the code generation task, with two widely used benchmarks: HumanEval (Chen et al., 2021) and MBPP (Austin et al., 2021).

Our main contributions are as follows: (1) We propose IE, a novel watermarking framework that relies on a small MLP instead of the original LLM to enable safe, efficient and accurate watermark detection. (2) We present *Threshold Navigator*, a low-high entropy threshold auto-optimization method that enhances detection performance not only for our framework but also for various watermarking approaches. (3) Our proposed watermarking framework, IE, which integrates the three components, achieves a 99% reduction in parameter usage while delivering state-of-the-art detection performance, showcasing its efficiency and scalability.

2 Related Work

Traditional Text Watermarking typically modifies generated text to embed watermarks. Based on the granularity of these modifications, existing approaches can be categorized as format-based, lexical-based, syntactic-based, and generation-based methods. *Format-based* watermarking (Rizzo et al., 2016; Brassil et al., 1995; Por et al., 2012; Sato et al., 2023) originates from image watermarking and focuses on altering the text format rather than its content, such as by adjusting text layout or using Unicode-based substitutions. *Lexical-based* watermarking (Munyer et al., 2024; Ni et al., 2023; Yang et al., 2023; Yoo et al., 2023; Yang et al., 2022) replaces selected words with their synonyms while preserving the original sentence's syntactic structure. However, this approach is susceptible to attacks involving random synonym replacements. To address this vulnerability, syntactic-based methods (Atallah et al., 2001; Topkara et al., 2006; Meral et al., 2009) embed watermarks by modifying the text's syntactic structure, which enhances resistance to removal attacks. Nevertheless, these methods often produce unnatural transformations, degrading the quality of the generated text and increasing its susceptibility to detection and targeted attacks.

LLM-Based Watermarking embeds watermarks in LLMs by intervening at different generation stages, including logits generation, token sampling, and training. Watermarking during *logits generation* adjusts the probability distribution over tokens to embed identifiable patterns, while *token sampling* (Christ et al., 2024; Kuditipudi et al., 2024; Hou et al., 2024a,b) modifies the token selection process to incorporate watermarks. Watermarks can also be embedded into model weights *during training* (Sun et al., 2022, 2023; Gu et al., 2024; Xu et al., 2024b,a), encoding watermarks into the model itself to ensure traceability and resilience against removal or tampering.

Watermarking during logits generation is the most cost-effective approach, avoiding the overhead of retraining or complex dynamic sampling while remaining flexible for post hoc application. Kirchenbauer et al. (2023) proposed the classic vocabulary partitioning method, dividing tokens into "green" and "red" sets, biasing generation toward "green" tokens. Building on this, studies (Fernandez et al., 2023; Lu et al., 2024; Kirchenbauer et al., 2024) improved detectability, while others (Hu et al., 2024; Wu et al., 2023; Fu et al., 2024; Guan et al., 2024; Lee et al., 2024; Chen et al., 2024; Liu and Bu, 2024; Wang et al., 2024; Wouters, 2024; Wang et al., 2025) focused on preserving text quality.

To handle low-entropy scenarios, Lee et al. (2024) focused on watermarking only high-entropy tokens, while Lu et al. (2024) applied entropy-weighted adjustments to detection statistics. However, both approaches rely on re-querying original LLM during detection. Our proposed method, IE, eliminates the need for the original LLM during detection, enhancing safety and efficiency.

3 Preliminaries

Our method builds upon the KGW watermarking strategy (Kirchenbauer et al., 2023) for logits generation. KGW operates in two phases: the generation phase and the detection phase.

During the generation phase, when generating the t-th token s_t, a hash key is derived from the previous token s_{t-1}. Using this hash key, the vocabulary is divided into a green list and a red list, with the proportion of green tokens determined by γ. A bias δ is then added to the logits of tokens in the green list, increasing their likelihood of being selected during sampling.

In the detection phase, for a generated sequence $\{s_1, s_2, \ldots, s_{|T|}\}$, where $|T|$ is the number of tokens, the count of green tokens is denoted as $|S|_G$. A watermark detection statistic z is calculated as:

$$z = \frac{|S|_G - \gamma|T|}{\sqrt{|T|\gamma(1-\gamma)}}. \qquad (1)$$

A detection threshold $\hat{z}$ is predefined. If $z > \hat{z}$, the text is classified as watermarked; otherwise, it is considered human-generated.

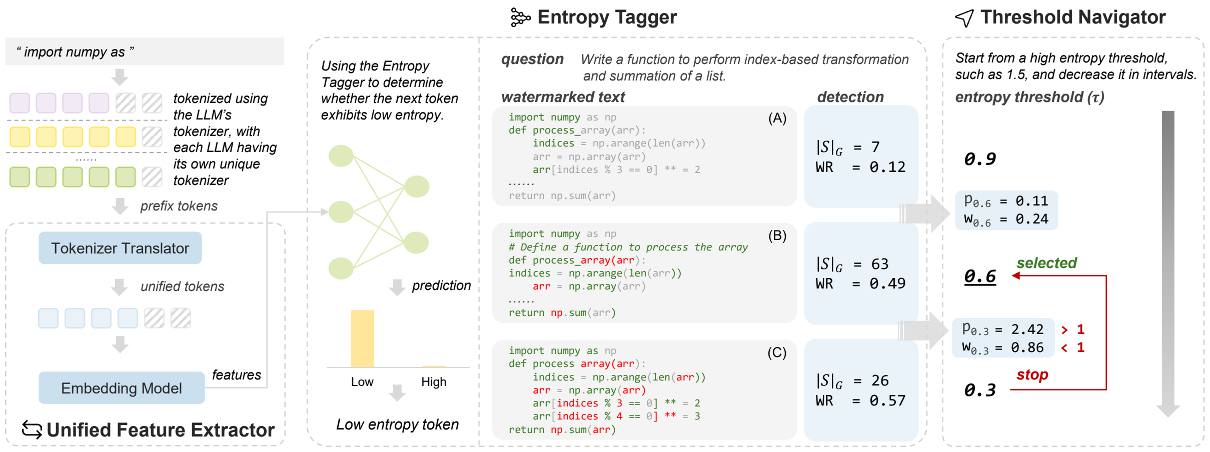
4 Methodology

In this section, we introduce our IE model in detail. The model consists of three modules: the *Unified Feature Extractor*, *Entropy Tagger*, and *Threshold Navigator*, as illustrated in Fig. 2.

4.1 Unified Feature Extractor

Existing works (Lee et al., 2024; Lu et al., 2024) rely on the original LLM to compute exact entropy for determining whether a token has low entropy. However, this approach significantly increases computational costs and the risk of model leakage. In practical applications, knowing the exact entropy value is often unnecessary—binary classification (low or high entropy) is sufficient. Thus, we propose using a smaller model to perform binary entropy prediction. In this subsection, we introduce a Unified Feature Extractor that learns vector representations of the generated text so far. In the next subsection, we present the binary entropy tagger.

Concretely, assume that a sequence of tokens $\{s_0, s_1, \ldots, s_{t-1}\}$ has already been generated, and the model is currently generating token s_t. These tokens may originate from different tokenizers associated with various LLMs. To handle this, our approach employs a *tokenizer translator* that converts prefix tokens into a unified format. The tokenizer translator first converts the prefix tokens back into raw text and then re-encodes them using the tokenizer of an embedding model. The *embedding model* processes the translated tokens and generates unified token embeddings. While LLMs typically support long input sequences, embedding models—often smaller, encoder-only architectures—are limited by a maximum input length. To address this, the embedding model focuses on processing only the last segment of tokens, up to its maximum allowable length, ensuring that critical information is retained. The representation of the last token, v_t, is used to represent the entire generated sequence. This token encapsulates


" import numpy as "
tokenized using the LLM's tokenizer, with each LLM having its own unique tokenizer
prefix tokens
Tokenizer Translator
unified tokens
Embedding Model
features
Unified Feature Extractor
Using the Entropy Tagger to determine whether the next token exhibits low entropy.
prediction
Low
High
Low entropy token
Entropy Tagger
question
Write a function to perform index-based transformation and summation of a list.
watermarked text
detection
(A)
(B)
(C)
Threshold Navigator
Start from a high entropy threshold, such as 1.5, and decrease it in intervals.
entropy threshold (τ)
0.9
0.6
0.3
selected
stop

```
import numpy as np
def process_array(arr):
    indices = np.arange(len(arr))
    arr = np.array(arr)
    arr[indices % 3 == 0] ** = 2
......
return np.sum(arr)
```

$|S|_G$ = 7
WR = 0.12

```
import numpy as np
# Define a function to process the array
def process_array(arr):
indices = np.arange(len(arr))
    arr = np.array(arr)
......
return np.sum(arr)
```

$|S|_G$ = 63
WR = 0.49

```
import numpy as np
def process array(arr):
    indices = np.arange(len(arr))
    arr = np.array(arr)
    arr[indices % 3 == 0] ** = 2
    arr[indices % 4 == 0] ** = 3
return np.sum(arr)
```

$|S|_G$ = 26
WR = 0.57

$p_{0.6}$ = 0.11
$w_{0.6}$ = 0.24

$p_{0.3}$ = 2.42 > 1
$w_{0.3}$ = 0.86 < 1

Figure 2: **Overview of IE (Invisible Entropy).** The model includes three components: the *Unified Feature Extractor* for tokenizer compatibility and feature extraction, the *Entropy Tagger* to predict if the next token's entropy exceeds threshold τ, and the *Threshold Navigator* to optimize τ for effective watermarking, naturalness, and robustness. Tokens are color-coded as red (red list), green (green list), and gray (unwatermarked). This example shows the search stopping at $\tau = 0.6$. At $\tau = 0.9$, insufficient watermarking occurs, while at $\tau = 0.3$, excessive low-entropy classification causes token generation issues (e.g., the underscore "_").

step-by-step contextual dependencies, providing an effective summary of the preceding text for the binary prediction task.

4.2 Entropy Tagger

Following the motivation outlined in the previous section, we propose an Entropy Tagger that predicts whether a token s_t is low-entropy by leveraging the feature vector v_t obtained from the feature extractor. The tagger outputs the probability p_t that the token's entropy is below a threshold τ, optimized by binary cross-entropy loss:

$$\mathcal{L} = -\frac{1}{N}\sum_{t=1}^{N}\left[y_t \log(p_t) + (1-y_t)\log(1-p_t)\right],$$

where y_t denotes the truth label for the t-th sample, computed by the original LLM (0 for high-entropy tokens and 1 for low-entropy tokens), and N is the total number of samples.

For the tagger implementation, we find that a small learnable multi-layer perceptron is sufficient to make accurate predictions. For entropy calculation, we employ Shannon entropy (Lee et al., 2024) over the dense Spike Entropy (Kirchenbauer et al., 2023), as its dispersed distribution offers clearer boundaries.

4.3 Threshold Navigator

The entropy threshold τ is crucial in balancing watermarked and non-watermarked tokens, directly impacting watermark effectiveness. When τ is too high, more tokens are classified as low-entropy, reducing the number of tokens eligible for watermarking, as seen in Block A of Fig. 2, where gray (unwatermarked) tokens dominate. Conversely, if τ is too low, fewer tokens are treated as low-entropy, leading to excessive watermarking (e.g., colored tokens (watermarked) dominate in Block C of Fig. 2). Existing entropy-based watermarking methods rely on manually predefined or empirically determined entropy thresholds (Lee et al., 2024), making them less robust since they overlook sample variations and depend heavily on the chosen parameter.

To address these limitations, we propose our Threshold Navigator. The Threshold Navigator automatically searches for an appropriate entropy threshold for each sentence. Here, we define an optimistic threshold τ as the point *where the watermark ratio* (WR, *defined as the ratio of watermarked tokens to the total number of generated tokens) drops while the count of green tokens rises*. Intuitively, a lower watermark ratio indicates lighter modifications to the original text, thereby reducing interference from the watermarking mechanism. Meanwhile, an increased count of green tokens signifies better alignment with machine-generated text, making it easier for the watermark to be detected. We also provide a theoretical proof on this in §6.2.

Based on the above sensitivity analysis, we introduce two metrics. Watermark Ratio Change (w) measures the change in watermark ratios between entropy thresholds τ_{i-1} and τ_i: $w_{\tau_i} = \mathrm{WR}_{\tau_{i-1}}/\mathrm{WR}_{\tau_i}$. Green Token Counts Change (p) quantifies the variation in green token counts: $p_{\tau_i} = |S|_{G_{\tau_{i-1}}}/|S|_{G_{\tau_i}}$. During the dynamic adjustment process, the Threshold Navigator lowers the

Method	Params ↓	HumanEval PPR ↑	HumanEval UES ↑	HumanEval Pass@1 ↑	HumanEval AUROC ↑	HumanEval TPR ↑	MBPP PPR ↑	MBPP UES ↑	MBPP Pass@1 ↑	MBPP AUROC ↑	MBPP TPR ↑
Post-hoc											
Log P(X)	120M	5.513	0.662	0.334	0.533	0.113	5.373	0.645	0.378	0.525	0.054
LogRank	120M	5.583	0.670	0.334	0.553	0.127	5.373	0.645	0.378	0.527	0.052
DetectGPT	1.1B	0.613	0.675	0.334	0.533	0.165	0.619	0.681	0.378	0.565	0.158
DetectGPT(T5-3B)	3B	0.220	0.660	0.334	0.549	0.092	0.214	0.643	0.378	0.531	0.040
GPTZero	-	-	0.661	0.334	0.521	0.122	-	0.619	0.378	0.449	0.026
OpenAI Classifier	-	-	0.643	0.334	0.518	0.053	-	0.634	0.378	0.500	0.036
Watermark-based											
KGW	-	-	0.768	0.253	0.904	0.652	-	0.732	0.242	0.930	0.718
EWD	15.5B	0.056	0.872	0.295	0.943	0.780	0.051	0.790	0.293	0.930	0.678
EWD	3B	0.290	0.871	0.295	0.941	0.778	0.256	0.767	0.293	0.916	0.602
EWD	1B	0.861	0.861	0.295	0.931	0.745	0.757	0.757	0.293	0.910	0.567
SWEET	15.5B	0.057	0.884	0.301	0.944	0.789	0.051	0.785	0.322	0.901	0.536
SWEET	3B	0.264	0.792	0.253	0.933	0.722	0.245	0.737	0.293	0.896	0.500
SWEET	1B	0.764	0.764	0.253	0.925	0.615	0.732	0.732	0.293	0.891	0.487
IE	130M	6.709	0.872	0.294	0.941	0.787	5.805	0.755	0.301	0.892	0.534

Table 1: **Main results on HumanEval and MBPP.** "-" indicates either undisclosed parameters (e.g., GPTZero, OpenAI Classifier) or no additional models required (e.g., KGW).

entropy threshold and monitors changes in WR and the number of green tokens $|S|_G$. The optimization process stops when $p > 1$ and $w < 1$, indicating that increasing the entropy threshold improves the sensitivity of the watermarked text to the green token counts while reducing the watermark ratio, thus achieving a balanced and robust distinction. Alg. 4 provides the main procedure. An example is shown in Fig. 2, and additional examples can be found in Fig. 6 in App. A.

5 Experiments

5.1 Tasks and Metrics

We evaluate IE and baselines in two Python code generation tasks: HumanEval (Chen et al., 2021) and MBPP (Austin et al., 2021). We assess IE and baselines in effectiveness and efficiency.

The evaluation of *effectiveness* focuses on both code generation ability and detectability. We assess code generation using Pass@k, and detectability using AUROC, which measures the model's ability to distinguish watermarked from non-watermarked text. We also report the True Positive Rate (TPR), which measures the proportion of correctly identified machine-generated text when the False Positive Rate (FPR) is less than 5%. We propose the Unified Effectiveness Score (UES), averaging the normalized Pass@1 and detectability metrics for overall evaluation: UES $= \frac{\frac{\text{Pass@1}}{\text{Pass@1}_{\text{non}}} + \left(\frac{\text{AUROC}+\text{TPR}}{2}\right)}{2}$, where $\text{Pass@1}_{\text{non}}$ represents the Pass@1 for text without watermark.

From an *efficiency* standpoint, we highlight the number of parameters, denoted as Params, necessary for watermarking in the detection phase. The detection time required by the watermarking methods is also reported in Tab. 8.

To combine effectiveness and efficiency, we introduce a new metric called Performance-to-Params Ratio (PPR), defined as: PPR $= \frac{\text{UES}}{\text{Params}}$.

5.2 Baselines

We compare IE with post-hoc detection baselines and watermarking methods. Post-hoc detection does not require any modification during the generation process, thus maintaining the same text quality as non-watermarked text. LogP(x) and LogRank (Gehrmann et al., 2019), and DetectGPT (Mitchell et al., 2023) are zero-shot detection methods that do not require labeled data. In contrast, GPTZero and OpenAI Classifier (Solaiman et al., 2019) are trained classifiers.

We select KGW (Kirchenbauer et al., 2023), SWEET (Lee et al., 2024), and EWD (Lu et al., 2024) as watermarking methods for comparison. KGW applies watermarking to all tokens during both the generation and detection phases. SWEET only applies watermarking to low-entropy tokens during both phases, resulting in higher text quality and detectability compared to KGW (see details in App. B). EWD improves text watermarking detection by assigning higher influence weights to higher-entropy tokens during detection. To explore the performance SWEET and EWD on smaller surrogate models, we also provide experimental results using StarCoder-3B and StarCoder-1B to compute entropy. In this setting, KGW serves as the watermark generator, while SWEET and EWD act as detectors.

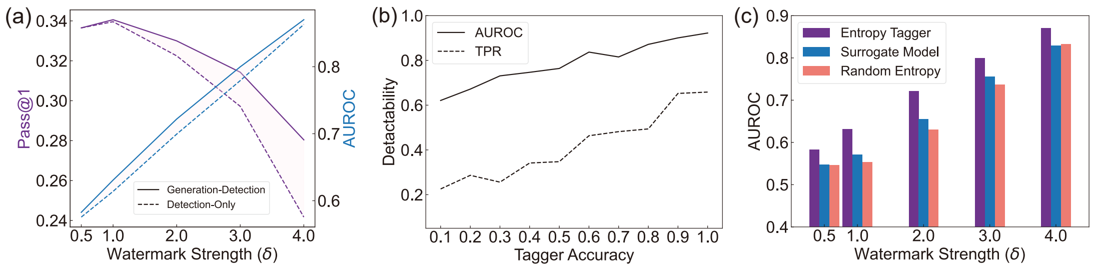


Figure 3: **Analysis of the Entropy Tagger.** (a) Comparison of applying the Entropy Tagger at different stages: generation-detection versus detection-only. (b) The relationship between Entropy Tagger accuracy and its effectiveness in watermarking. (c) Demonstration of the superior performance of the Entropy Tagger compared to a surrogate model and randomly set entropy.

5.3 Implementation

In our implementation, we use Starcoder (Li et al., 2023) as the LLM and SimCSE (Gao et al., 2021) as the embedding model. We use MBPP dataset to train Entropy Tagger, where details are in App. C. For the post-hoc methods, KGW and SWEET, we adopt the optimal hyperparameters reported by Lee et al., 2024. While for EWD, we follow the settings in Lu et al. (2024). Since SWEET provides results corresponding to specific entropy threshold, we calculate the average of the results across these different entropy thresholds. For IE, we use the optimal hyperparameters $\gamma = 0.5$ and $\delta = 3.0$ unless otherwise specified. All experiments can be conducted on one single A100-40G. More detailed settings are provided in App. D.

5.4 Main Results

We show the main results in Tab. 1.

From **Effectiveness** perspective, we can draw the following conclusions: (1) *Post-hoc methods fail to handle machine-generated text in low-entropy scenarios.* The UES of all watermark-based methods exceeds 0.75 on the HumanEval dataset and 0.70 on the MBPP dataset, whereas post-hoc methods remain below 0.70 on both datasets. (2) *Our IE demonstrates strong effectiveness,* outperforming post-hoc methods and achieving comparable performance to SWEET and EWD. (3) *SWEET and EWD suffer performance degradation when applied with smaller models.* When using a surrogate model, IE (130M) significantly outperforms SWEET (1B/3B). While EWD is less sensitive to the choice of surrogate model compared to SWEET, it still underperforms IE on HumanEval. From an **Efficiency** perspective, LogP(x) and LogRank use BERT with 0.12B parameters for detection. DetectGPT relies on SantaCoder (1.1B) or T5-3B (3B). GPTZero and OpenAI Classifier are closed-source, with parameter counts unavailable. KGW requires no additional model, while SWEET and EWD depend on the original LLM (15.5B). In contrast, our method uses an embedding model and a lightweight MLP, totaling 0.13B parameters, comparable to Post-hoc methods.

We finally use PPR to evaluate the **combined effectiveness and efficiency** of the methods. Among all methods, IE achieves the highest PPR, significantly outperforming other watermarking methods. While Post-hoc methods like LogP(x) and LogRank achieve relatively higher PPRs compared to weaker baselines, their effectiveness remains low.

6 Analysis and Discussion

6.1 Analysis on Entropy Tagger

Generation-Detection or Detection-only? We compare the performance of the Entropy Tagger in two setups: Detection-only and Generation-Detection. In the Detection-only setup, ground truth entropy values are used to classify tokens as high or low entropy, with a fixed threshold applied for watermark detection. In contrast, the Generation-Detection setup incorporates the Entropy Tagger during the generation phase, predicting entropy values to embed watermarks dynamically. As shown in Fig. 3(a), experimental results indicate that Generation-Detection consistently outperforms Detection-only in both Pass@1 and AUROC across different watermark strengths. This demonstrates that aligning entropy-aware methods during both generation and detection is essential for achieving robust and effective watermarking.

Relationship between Entropy Tagger Accuracy and Watermark Detectability. We investigate how Entropy Tagger accuracy impacts watermark detectability by varying the tagger's accuracy and observing its effect on detection metrics. Under

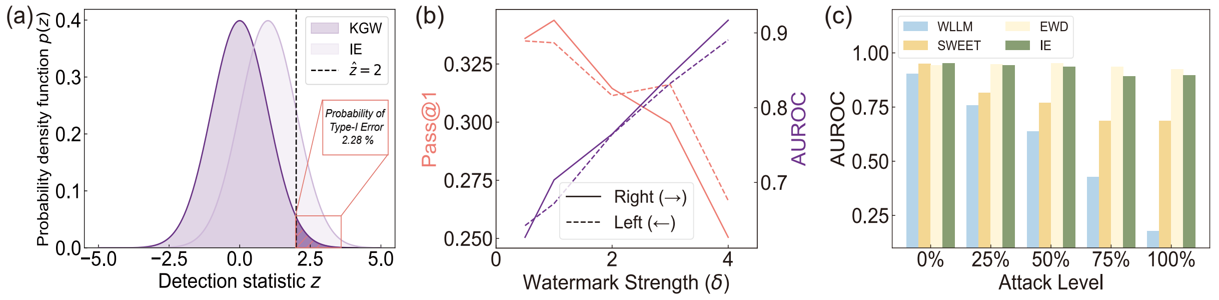


Figure 4: (a) Type-I Error probability and the distribution of detection statistic z for human-written text. (b) Impact of threshold navigator search directions. (c) Robustness of detection to paraphrasing attacks.

a Detection-only setting, we calculate the exact entropy of watermarked text and simulate tagger inaccuracies by introducing disturbances, where the disturbance proportion r (0.0 to 1.0) determines the tagger's accuracy as $1 - r$. Results in Fig. 3(b) show that higher tagger accuracy leads to improved AUROC and TPR, highlighting the importance of precise entropy predictors for robust watermarking.

Comparison with surrogate and random entropy. We also replace the Entropy Tagger in the IE framework with a surrogate model (StarCoder-3B) and with Random Entropy (a floating-point value randomly selected between -5.0 and 5.0), respectively. As shown in Fig. 3(c), the results demonstrate that using the Entropy Tagger significantly outperforms both the Surrogate Model and Random Entropy. Furthermore, the Entropy Tagger contains only 0.13B parameters, making it significantly more cost-effective than the surrogate model. These comparisons confirm the superiority of Entropy Tagger both effectively and efficiently.

6.2 Analysis on Threshold Navigator

Theoretical Validation The primary goal of watermark detection is to minimize Type-I and Type-II errors. Thus, we analyze the impact of the Threshold Navigator on both. Generally, our analysis shows that it has no impact on Type-I Error but significantly reduces Type-II Error.

Type-I Error measures the probability of human-written text being misclassified as watermarked. For human-written text T, each token is assumed to be independent of the watermarking algorithm, and the probability of a token being included in the green list is denoted by γ. As a result, the number of green tokens $|S|_G$ follows a normal distribution: $|S|_G \sim \mathcal{N}(\gamma|T|, \gamma(1-\gamma)|T|)$. In the case of selective watermarking methods such as SWEET, where only a portion of tokens are watermarked, the distribution becomes: $|S|_G \sim \mathcal{N}(\gamma|\tilde{T}|, \gamma(1-\gamma)|\tilde{T}|)$. Here, $|\tilde{T}| = \mathrm{WR} \times |T|$ represents the fraction of the text covered by the watermark. Regardless of the value of WR, the distribution can be standardized using: $z = \frac{|S|_G - \gamma|\tilde{T}|}{\sqrt{\gamma(1-\gamma)|\tilde{T}|}}$. Because $|S|_G$ follows a normal distribution, the standardized variable z follows a standard normal distribution $\mathcal{N}(0, 1)$. The probability density function $p(z)$ describes the likelihood of observing a specific value of z, and the Type-I Error corresponds to the area under the standard normal curve beyond a given threshold (e.g., when $z = 2$, the error is 2.28%, shown as the red region in Fig. 4(a)). Since the probability of $z > \hat{z}$ for human text remains constant across τ, the selection of τ does not affect the Type-I Error rate.

Type-II Error measures the probability of watermarked text being misclassified as human-written text, with lower Type-II Error indicating better detection performance. To show how the Threshold Navigator reduces Type-II Error, we analyze its search criterion ($p > 1$ and $w < 1$) and its effect on the detection statistic z, as higher z values directly lower Type-II Error. Specifically, we examine the relationship between z and two key factors: green token count ($|S|_G$) and watermark ratio (WR). For selective watermarking methods (e.g., IE or SWEET), z can be expressed as:

$$z = \frac{|S|_G - \gamma \cdot \mathrm{WR} \cdot |T|}{\sqrt{\mathrm{WR} \cdot |T| \cdot \gamma(1-\gamma)}}.$$

A higher z for machine-generated text indicate better watermark detectability, as z quantifies the statistical deviation of the green token count from its expected value in human text.

To understand how z changes with $|S|_G$ and WR, we compute the partial derivatives of z:

$$\frac{\partial z}{\partial |S|_G} = \frac{1}{\sqrt{\mathrm{WR} \cdot |T| \cdot \gamma(1-\gamma)}} > 0,$$

showing that z is positively correlated with $|S|_G$.

$$\frac{\partial z}{\partial \mathrm{WR}} = -\frac{|S|_G}{\sqrt{|T| \cdot \gamma(1-\gamma)}} \cdot \frac{1}{2 \cdot \sqrt{\mathrm{WR}^3}} - \sqrt{\frac{\gamma|T|}{1-\gamma}} \cdot \frac{1}{2\sqrt{\mathrm{WR}}} < 0,$$

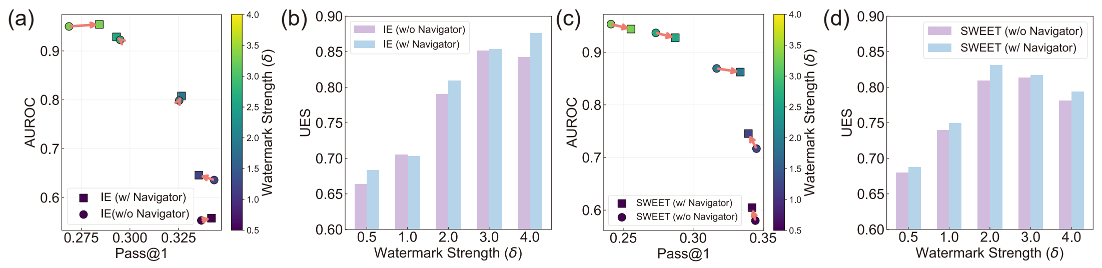


Figure 5: **Effectiveness of the Threshold Navigator.** (a) Improved detectability and quality with the Navigator across δ. (b) Improved UES with the Navigator. (c) Generalizability to SWEET: Pass@1 vs. AUROC, demonstrating similar improvements. (d) UES comparison for SWEET, showing significant gains with the Navigator.

showing that z is negatively correlated with WR.

These results show that increasing $|S|_G$ improves z under the condition $p > 1$, allowing green token counts to grow as thresholds adjust. Simultaneously, decreasing WR enhances z under $w < 1$, reducing watermarked tokens and improving detectability. Therefore, our Threshold Navigator effectively reduces Type-II Error by optimizing $|S|_G$ and WR, leading to improved watermark detection.

Impact of Search Directions Our default threshold search proceeds from high to low. Since different search directions may impact the results, we compare searches starting from high to low ($\leftarrow$) and low to high ($\rightarrow$) to assess their effects. The experimental results are shown in Fig. 4(c). It can be observed that as the watermarking strength increases, navigation towards the Right generally achieves higher AUROC in most cases. Conversely, when the watermarking strength is relatively low, navigation towards the Left results in better code quality. This is because, at higher watermarking strengths, the impact on code quality becomes more significant, and navigation towards the Left, which prioritizes selecting higher entropy thresholds, helps mitigate the degradation of code quality.

Effectiveness and Orthogonality. Fig. 5(a) presents an ablation study where the Threshold Navigator is removed, showing the Pass@1 and AUROC of the watermark under different watermark strengths. Fig. 5(b) illustrates the UES across the same range of watermark strengths. These results demonstrate that the Threshold Navigator significantly enhances the AUROC and UES of IE, enabling the output to strike a balance between quality and detectability. To further evaluate the generalizability of the Threshold Navigator across different watermark backbones, we apply it to the SWEET watermarking method. As shown in Fig. 5(c,d), the Navigator significantly improves SWEET across various watermark strengths. This highlights the versatility of the Threshold Navigator, as it can be seamlessly integrated with existing watermarking methods to enhance their effectiveness.

6.3 Robustness to Paraphrasing Attacks

Malicious users may attempt to remove the watermark by paraphrasing attacks (Krishna et al., 2023; Gao et al., 2025). Here, we conduct variable name paraphrasing attacks on the generated codes at different levels. Specifically, for the generated codes from each watermarking method on the HumanEval dataset, we replace varying proportions of variable names in the watermarked text. Fig. 4(c) shows the detectability of the attacked code, measured by AUROC. The Attack Level denotes the percentage of variable names changed, with 0% meaning none are altered and 25% indicating a quarter are paraphrased. It can be seen that as the Attack Level increases, the detectability of all methods declines. Notably, KGW and SWEET experience the most significant drops, with KGW's detectability falling below 20% and SWEET dropping below 80%. Meanwhile, IE and EWD show better robustness, maintaining around 90%.

7 Conclusion

We introduce IE (Invisible Entropy), a selective watermarking method that overcomes two key limitations: reliance on the original LLM for costly entropy calculations and difficulty watermarking predictable, low-entropy outputs. IE uses a lightweight feature extractor and entropy tagger to predict token entropy without the original LLM and a Threshold Navigator for adaptive entropy thresholds, ensuring balance in effectiveness, naturalness, and detectability. Experiments on HumanEval and MBPP show a 99% parameter reduction with state-of-the-art performance. In the future, we aim to further enhance the accuracy of the entropy tagger to improve watermarking effectiveness and robustness.

Limitations

Although IE offers a safe, efficient and accurate watermarking approach, we identify two limitations and suggest potential solutions to address them.

Entropy Tagger Accuracy Calibration In the App. C, we report the accuracy of the trained Entropy Tagger. Although the current Entropy Tagger performs comparably to the precise entropy calculation, there is still some slight decrease in performance. Therefore, future work could focus on training a more precise Entropy Tagger, such as by incorporating certain specific low-entropy tokens as analyzed in App. G.

Optimization Strategy for Threshold Navigator In § 6.2, we analyze the impact of the two search directions of the Threshold Navigator on watermarking performance. However, in our experiments, the search granularity is fixed at 0.3, which may limit optimization flexibility. Future work could explore adaptive search granularities that dynamically adjust based on context or performance feedback, as well as alternative search directions that better align with different watermarking scenarios to further enhance performance.

References

Mikhail J. Atallah, Victor Raskin, Michael Crogan, Christian Hempelmann, Florian Kerschbaum, Dina Mohamed, and Sanket Naik. 2001. Natural language watermarking: Design, analysis, and a proof-of-concept implementation. In *Information Hiding*, pages 185–200, Berlin, Heidelberg. Springer Berlin Heidelberg.

Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, et al. 2021. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732*.

J.T. Brassil, S. Low, N.F. Maxemchuk, and L. O'Gorman. 1995. Electronic marking and identification techniques to discourage document copying. *IEEE Journal on Selected Areas in Communications*, 13(8):1495–1504.

Liang Chen, Yatao Bian, Yang Deng, Deng Cai, Shuaiyi Li, Peilin Zhao, and Kam-Fai Wong. 2024. WatME: Towards lossless watermarking through lexical redundancy. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 9166–9180, Bangkok, Thailand. Association for Computational Linguistics.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. 2021. Evaluating large language models trained on code. *Preprint*, arXiv:2107.03374.

Miranda Christ, Sam Gunn, and Or Zamir. 2024. Undetectable watermarks for language models. In *The Thirty Seventh Annual Conference on Learning Theory*, pages 1125–1139. PMLR.

Alexandre Duc, Stefan Dziembowski, and Sebastian Faust. 2014. Unifying leakage models: from probing attacks to noisy leakage. In *Advances in Cryptology–EUROCRYPT 2014: 33rd Annual International Conference on the Theory and Applications of Cryptographic Techniques, Copenhagen, Denmark, May 11-15, 2014. Proceedings 33*, pages 423–440. Springer.

Pierre Fernandez, Antoine Chaffin, Karim Tit, Vivien Chappelier, and Teddy Furon. 2023. Three bricks to consolidate watermarks for large language models. In *2023 IEEE International Workshop on Information Forensics and Security (WIFS)*, pages 1–6. IEEE.

Yu Fu, Deyi Xiong, and Yue Dong. 2024. Watermarking conditional text generation for ai detection: Unveiling challenges and a semantic-aware watermark remedy. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 38, pages 18003–18011.

Lang Gao, Xiangliang Zhang, Preslav Nakov, and Xiuying Chen. 2025. Shaping the safety boundaries: Understanding and defending against jailbreaks in large language models. *ACL*.

Tianyu Gao, Xingcheng Yao, and Danqi Chen. 2021. SimCSE: Simple contrastive learning of sentence embeddings. In *Empirical Methods in Natural Language Processing (EMNLP)*.

Sebastian Gehrmann, Hendrik Strobelt, and Alexander Rush. 2019. GLTR: Statistical detection and visualization of generated text. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics: System Demonstrations*, pages 111–116, Florence, Italy. Association for Computational Linguistics.

Chenchen Gu, Xiang Lisa Li, Percy Liang, and Tatsunori Hashimoto. 2024. On the learnability of watermarks for language models. In *The Twelfth International Conference on Learning Representations*.

Batu Guan, Yao Wan, Zhangqian Bi, Zheng Wang, Hongyu Zhang, Pan Zhou, and Lichao Sun. 2024. CodeIP: A grammar-guided multi-bit watermark for large language models of code. In *Findings of the Association for Computational Linguistics: EMNLP 2024*, pages 9243–9258, Miami, Florida, USA. Association for Computational Linguistics.

Abe Hou, Jingyu Zhang, Tianxing He, Yichen Wang, Yung-Sung Chuang, Hongwei Wang, Lingfeng Shen, Benjamin Van Durme, Daniel Khashabi, and Yulia Tsvetkov. 2024a. SemStamp: A semantic watermark with paraphrastic robustness for text generation. In *Proceedings of the 2024 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 1: Long Papers)*, pages 4067–4082, Mexico City, Mexico. Association for Computational Linguistics.

Abe Hou, Jingyu Zhang, Yichen Wang, Daniel Khashabi, and Tianxing He. 2024b. k-SemStamp: A clustering-based semantic watermark for detection of machine-generated text. In *Findings of the Association for Computational Linguistics: ACL 2024*, pages 1706–1715, Bangkok, Thailand. Association for Computational Linguistics.

Zhengmian Hu, Lichang Chen, Xidong Wu, Yihan Wu, Hongyang Zhang, and Heng Huang. 2024. Unbiased watermark for large language models. In *The Twelfth International Conference on Learning Representations*.

John Kirchenbauer, Jonas Geiping, Yuxin Wen, Jonathan Katz, Ian Miers, and Tom Goldstein. 2023. A watermark for large language models. In *International Conference on Machine Learning*, pages 17061–17084. PMLR.

John Kirchenbauer, Jonas Geiping, Yuxin Wen, Manli Shu, Khalid Saifullah, Kezhi Kong, Kasun Fernando, Aniruddha Saha, Micah Goldblum, and Tom Goldstein. 2024. On the reliability of watermarks for large language models. In *The Twelfth International Conference on Learning Representations*.

Kalpesh Krishna, Yixiao Song, Marzena Karpinska, John Frederick Wieting, and Mohit Iyyer. 2023. Paraphrasing evades detectors of AI-generated text, but retrieval is an effective defense. In *Thirty-seventh Conference on Neural Information Processing Systems*.

Rohith Kuditipudi, John Thickstun, Tatsunori Hashimoto, and Percy Liang. 2024. Robust distortion-free watermarks for language models. *Transactions on Machine Learning Research*.

Taehyun Lee, Seokhee Hong, Jaewoo Ahn, Ilgee Hong, Hwaran Lee, Sangdoo Yun, Jamin Shin, and Gunhee Kim. 2024. Who wrote this code? watermarking for code generation. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 4890–4911, Bangkok, Thailand. Association for Computational Linguistics.

Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li, Jenny Chim, et al. 2023. Starcoder: may the source be with you! *arXiv preprint arXiv:2305.06161*.

Aiwei Liu, Leyi Pan, Yijian Lu, Jingjing Li, Xuming Hu, Xi Zhang, Lijie Wen, Irwin King, Hui Xiong, and Philip Yu. 2024. A survey of text watermarking in the era of large language models. *ACM Computing Surveys*, 57(2):1–36.

Yepeng Liu and Yuheng Bu. 2024. Adaptive text watermark for large language models. In *Forty-first International Conference on Machine Learning*.

I Loshchilov. 2017. Decoupled weight decay regularization. *arXiv preprint arXiv:1711.05101*.

Yijian Lu, Aiwei Liu, Dianzhi Yu, Jingjing Li, and Irwin King. 2024. An entropy-based text watermarking detection method. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 11724–11735, Bangkok, Thailand. Association for Computational Linguistics.

Hasan Mesut Meral, Bülent Sankur, A. Sumru Özsoy, Tunga Güngör, and Emre Sevinç. 2009. Natural language watermarking via morphosyntactic alterations. *Computer Speech & Language*, 23(1):107–125.

Eric Mitchell, Yoonho Lee, Alexander Khazatsky, Christopher D Manning, and Chelsea Finn. 2023. DetectGPT: Zero-shot machine-generated text detection using probability curvature. In *Proceedings of the 40th International Conference on Machine Learning*, volume 202 of *Proceedings of Machine Learning Research*, pages 24950–24962. PMLR.

Travis Munyer, Abdullah Tanvir, Arjon Das, and Xin Zhong. 2024. Deeptextmark: A deep learning-driven text watermarking approach for identifying large language model generated text. *Preprint*, arXiv:2305.05773.

Ansong Ni, Srini Iyer, Dragomir Radev, Veselin Stoyanov, Wen-Tau Yih, Sida Wang, and Xi Victoria Lin. 2023. LEVER: Learning to verify language-to-code generation with execution. In *Proceedings of the 40th International Conference on Machine Learning*, volume 202 of *Proceedings of Machine Learning Research*, pages 26106–26128. PMLR.

Lip Yee Por, KokSheik Wong, and Kok Onn Chee. 2012. Unispach: A text-based data hiding method using unicode space characters. *Journal of Systems and Software*, 85(5):1075–1082.

Stefano Giovanni Rizzo, Flavio Bertini, and Danilo Montesi. 2016. Content-preserving text watermarking through unicode homoglyph substitution. In *Proceedings of the 20th International Database Engineering & Applications Symposium*, IDEAS '16, page 97–104, New York, NY, USA. Association for Computing Machinery.

Ryoma Sato, Yuki Takezawa, Han Bao, Kenta Niwa, and Makoto Yamada. 2023. Embarrassingly simple text watermarks. *Preprint*, arXiv:2310.08920.

Irene Solaiman, Miles Brundage, Jack Clark, Amanda Askell, Ariel Herbert-Voss, Jeff Wu, Alec Radford, Gretchen Krueger, Jong Wook Kim, Sarah Kreps, Miles McCain, Alex Newhouse, Jason Blazakis, Kris McGuffie, and Jasmine Wang. 2019. Release strategies and the social impacts of language models. *Preprint*, arXiv:1908.09203.

Congzheng Song and Ananth Raghunathan. 2020. Information leakage in embedding models. In *Proceedings of the 2020 ACM SIGSAC conference on computer and communications security*, pages 377–390.

Zhensu Sun, Xiaoning Du, Fu Song, and Li Li. 2023. Codemark: Imperceptible watermarking for code datasets against neural code completion models. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 1561–1572.

Zhensu Sun, Xiaoning Du, Fu Song, Mingze Ni, and Li Li. 2022. Coprotector: Protect open-source code against unauthorized training usage with data poisoning. In *Proceedings of the ACM Web Conference 2022*, pages 652–660.

Mercan Topkara, Umut Topkara, and Mikhail J. Atallah. 2006. Words are not enough: sentence level natural language watermarking. In *Proceedings of the 4th ACM International Workshop on Contents Protection and Security*, MCPS '06, page 37–46, New York, NY, USA. Association for Computing Machinery.

Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, et al. 2023. Llama 2: Open foundation and fine-tuned chat models. *arXiv preprint arXiv:2307.09288*.

Lean Wang, Wenkai Yang, Deli Chen, Hao Zhou, Yankai Lin, Fandong Meng, Jie Zhou, and Xu Sun. 2024. Towards codable watermarking for injecting multi-bits information to LLMs. In *The Twelfth International Conference on Learning Representations*.

Zongqi Wang, Tianle Gu, Baoyuan Wu, and Yujiu Yang. 2025. Morphmark: Flexible adaptive watermarking for large language models. *Preprint*, arXiv:2505.11541.

Bram Wouters. 2024. Optimizing watermarks for large language models. In *Forty-first International Conference on Machine Learning*.

Yihan Wu, Zhengmian Hu, Hongyang Zhang, and Heng Huang. 2023. Dipmark: A stealthy, efficient and resilient watermark for large language models. *arXiv preprint arXiv:2310.07710*.

Hengyuan Xu, Liyao Xiang, Xingjun Ma, Borui Yang, and Baochun Li. 2024a. Hufu: A modality-agnositc watermarking system for pre-trained transformers via permutation equivariance. *arXiv preprint arXiv:2403.05842*.

Xiaojun Xu, Yuanshun Yao, and Yang Liu. 2024b. Learning to watermark llm-generated text via reinforcement learning. *arXiv preprint arXiv:2403.10553*.

Xi Yang, Kejiang Chen, Weiming Zhang, Chang Liu, Yuang Qi, Jie Zhang, Han Fang, and Nenghai Yu. 2023. Watermarking text generated by black-box language models. *Preprint*, arXiv:2305.08883.

Xi Yang, Jie Zhang, Kejiang Chen, Weiming Zhang, Zehua Ma, Feng Wang, and Nenghai Yu. 2022. Tracing text provenance via context-aware lexical substitution. *Proceedings of the AAAI Conference on Artificial Intelligence*, 36(10):11613–11621.

KiYoon Yoo, Wonhyuk Ahn, Jiho Jang, and Nojun Kwak. 2023. Robust multi-bit natural language watermarking through invariant features. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 2092–2115, Toronto, Canada. Association for Computational Linguistics.

A Case study for threshold navigator

In this section, we present a case study on Threshold Navigator. We select different entropy thresholds τ (0.3, 0.6, 0.9, and 1.2), where token below the entropy threshold are not watermarked. The experimental results are shown in Fig. 6. In the watermarked text, tokens are annotated in red, green, or gray to represent red tokens, green tokens, and unwatermarked tokens, respectively. The evaluation is conducted from two perspectives, correct (whether the code correctly answer the question) and detected (whether the watermark is successfully detected).

It is indicated that when τ is set to 0.3, the proportion of watermarked token is relatively high, which tends to result in lower code correctness. Conversely, when τ is set to 1.2, the proportion of watermarked tokens is relatively low. While this helps maintain code correctness to some extent, it also leads to a decrease in watermark detectability. Using the Threshold Navigator algorithm, the results are shown in Tab. 2. When τ is set to 0.3, the values of p and w satisfy the condition $p > 1$ and $w < 1$, respectively. Therefore, a "transition" is required for 0.3, leading to the correct selection of 0.6 as entropy threshold. This is further validated in Fig. 6, where an entropy threshold of 0.6 ensures both correctness and detectability.

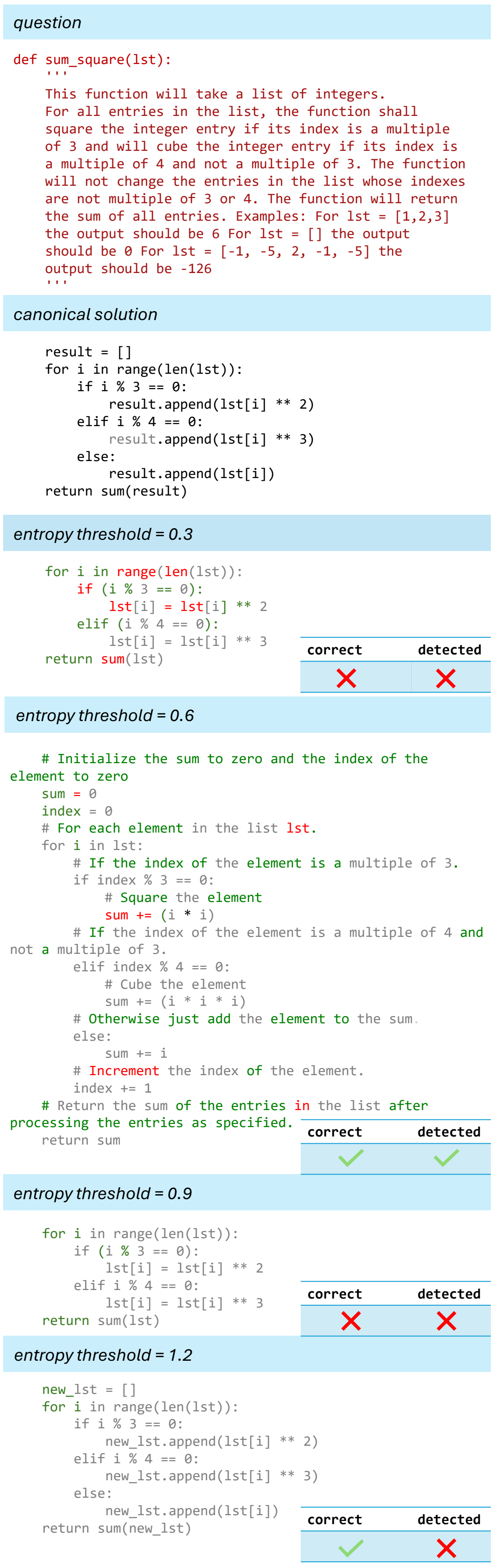
Entropy Threshold	0.3	0.6	0.9
p	3.57	0.12	0.75
w	0.98	0.29	0.58
$p > 1$ and $w < 1$?	Yes	No	No

Table 2: p and w under different entropy thresholds.

B Algorithms for entropy-based selective watermark (SWEET)

In this section, we present the algorithms for entropy-based selective watermark generation and detection (SWEET), as shown in Alg. 1 and Alg. 2. The core idea has already been introduced in the § 3, while watermarking is applied only to the tokens with entropy greater than τ during generation and detection process.

The algorithm for text generation with entropy-based selective watermarking is built on KGW, as shown in Alg. 1. Initially, the language model processes the preceding tokens to compute the probability distribution $p^{(t)}$ over the vocabulary for the



question

```
def sum_square(lst):
    '''
    This function will take a list of integers.
    For all entries in the list, the function shall
    square the integer entry if its index is a multiple
    of 3 and will cube the integer entry if its index is
    a multiple of 4 and not a multiple of 3. The function
    will not change the entries in the list whose indexes
    are not multiple of 3 or 4. The function will return
    the sum of all entries. Examples: For lst = [1,2,3]
    the output should be 6 For lst = [] the output
    should be 0 For lst = [-1, -5, 2, -1, -5] the
    output should be -126
    '''
```

canonical solution

```
    result = []
    for i in range(len(lst)):
        if i % 3 == 0:
            result.append(lst[i] ** 2)
        elif i % 4 == 0:
            result.append(lst[i] ** 3)
        else:
            result.append(lst[i])
    return sum(result)
```

entropy threshold = 0.3

```
    for i in range(len(lst)):
        if (i % 3 == 0):
            lst[i] = lst[i] ** 2
        elif (i % 4 == 0):
            lst[i] = lst[i] ** 3
    return sum(lst)
```

correct	detected
✗	✗

entropy threshold = 0.6

```
    # Initialize the sum to zero and the index of the
element to zero
    sum = 0
    index = 0
    # For each element in the list lst.
    for i in lst:
        # If the index of the element is a multiple of 3.
        if index % 3 == 0:
            # Square the element
            sum += (i * i)
        # If the index of the element is a multiple of 4 and
not a multiple of 3.
        elif index % 4 == 0:
            # Cube the element
            sum += (i * i * i)
        # Otherwise just add the element to the sum.
        else:
            sum += i
        # Increment the index of the element.
        index += 1
    # Return the sum of the entries in the list after
processing the entries as specified.
    return sum
```

correct	detected
✓	✓

entropy threshold = 0.9

```
    for i in range(len(lst)):
        if (i % 3 == 0):
            lst[i] = lst[i] ** 2
        elif i % 4 == 0:
            lst[i] = lst[i] ** 3
    return sum(lst)
```

correct	detected
✗	✗

entropy threshold = 1.2

```
    new_lst = []
    for i in range(len(lst)):
        if i % 3 == 0:
            new_lst.append(lst[i] ** 2)
        elif i % 4 == 0:
            new_lst.append(lst[i] ** 3)
        else:
            new_lst.append(lst[i])
    return sum(new_lst)
```

correct	detected
✓	✗

Figure 6: Results for various entropy thresholds.

next token s_t (Line 3). The entropy of this distribution determines whether the watermark is applied (Line 4). If the entropy H_t exceeds a threshold τ, the vocabulary is partitioned into a "green list" and a "red list" using a hash function seeded by the previous token. The size of the green list is controlled by a proportion parameter γ, and its logits are increased by a hardness parameter δ to influence token selection. The final token is sampled from the adjusted probability distribution (Lines 5 to 9). If the entropy H_t is below the threshold, the token is sampled from the original distribution without modification (Line 11).

The detection phase for entropy-based selective watermarking is similar to the generation phase, as shown in Alg. 2. It initializes counters for green list tokens ($|S|_G$), scored tokens ($|\hat{T}|$), total generated tokens ($|T|$), and the Watermark Ratio (WR) (Line 2). For each token, the entropy H_t is computed (Line 4). If H_t exceeds the threshold τ, a hash of the previous token seeds a random number generator to partition the vocabulary into a green list G and a red list R. Tokens in the green list increment the green token count, while all scored tokens update the scored token count (Lines 5 to 9). After processing all tokens, a standardized score z is calculated to measure the deviation in green token frequency and the Watermark Ratio WR (Line 13). If z exceeds a predefined threshold $\hat{z}$, the text is classified as watermarked; otherwise, it is considered unwatermarked (Lines 14 to 18).

C Training details of entropy tagger

C.1 Preprocess

The statistics of HumanEval and MBPP datasets is shown in Tab. 3. During the preprocessing phase, we use the training split of the MBPP dataset to construct the training dataset for the Entropy Tagger, with the preprocessing algorithm described in Alg. 3. Specifically, we first concatenate the prompt with the code. (Line 4) Next, we truncate the sequence starting from the beginning, adding one token at a time, and compute the exact entropy using LLM as the label. Then, we use the Unified Feature Extractor to extract features from the truncated sequence to obtain the feature vector v. (Lines 5 to 12) Finally, we obtain the preprocessed dataset $\hat{D} = \{(X_i, y_i)\}$, where X_i represents the i-th feature vector v, and y_i represents the corresponding actual entropy for X_i. The dataset size for each split is shown in Tab. 3.

Algorithm 1 Text Generation with entropy-based selective watermark

```
1:  Input: prompt, $s_{-N_p}, \ldots, s_{-1}$
           entropy threshold, $\tau$
           green list size, $\gamma \in (0, 1)$
           hardness parameter, $\delta > 0$
2:  for $t = 0, 1, \ldots$ do
3:    Apply the language model to prior tokens $s_{-N_p}, \ldots, s_{-1}$ to get a probability vector $p^{(t)}$ over the vocabulary.
4:    Calculate the entropy $H_t$ for next token $s_t$.
5:    if $H_t > \tau$ then
6:      Compute a hash of token $s_{t-1}$, and use it to seed a random number generator.
7:      Using this random number generator, randomly partition the vocabulary into a "green list" $G$ of size $\gamma|V|$, and a "red list" $R$ of size $(1-\gamma)|V|$.
8:      Add $\delta$ to each green list logit. Apply the softmax operator to these modified logits to get a probability distribution over the vocabulary.
          $$\hat{p}_k^{(t)} = \begin{cases} \frac{e^{\left(l_k^{(t)}+\delta\right)}}{\sum_{i\in R} e^{l_i^{(t)}} + \sum_{i\in G} e^{l_i^{(t)}+\delta}}, & k \in G \\ \frac{e^{l_k^{(t)}}}{\sum_{i\in R} e^{l_i^{(t)}} + \sum_{i\in G} e^{l_i^{(t)}+\delta}}, & k \in R \end{cases}$$
9:      Sample the next token, $s_t$, using the marked distribution $\hat{p}^{(t)}$.
10:   else
11:     Sample the next token, $s_t$, using the origin distribution $p^{(t)}$.
12:   end if
13: end for
```

Dataset	Split	# Samples	# Converted
HumanEval	test	164	32,168
MBPP	train	374	29,747
	validation	90	7,391
	test	500	40,571

Table 3: **Statistics of HumanEval and MBPP.** #Samples indicates the number of samples in each split of the dataset, while #Converted represents the number of samples in each split after preprocessing.

Algorithm 2 Detection with entropy-based selective watermark

```
1: Input: prompt, $s_{-N_p}, \dots, s_{-1}$
          entropy threshold, $\tau$
          green list size, $\gamma \in (0, 1)$
          z threshold, $\hat{z}$
2: Initialize: green token counts, $|S|_G \leftarrow 0$
               scored tokens counts, $|\hat{T}| \leftarrow 0$
               generated tokens counts, $|T| \leftarrow 0$
               watermark ratio, WR $\leftarrow 0$
3: for $t = 0, 1, ...$ do
4:     Compute the entropy $H_t$ of the next token $s_t$.
5:     if $H_t > \tau$ then
6:         Compute a hash of $s_{t-1}$, and use it to seed a random number generator.
7:         Using the random number generator, randomly partition the vocabulary into a "green list" $G$ of size $\gamma|V|$, and a "red list" $R$ of size $(1-\gamma)|V|$.
8:         Increment $|S|_G$ if $s_t$ in green list.
               $$|S|_G \leftarrow \begin{cases} |S|_G + 1, & \text{if } s_t \in G \\ |S|_G, & \text{otherwise} \end{cases}$$
9:         Increment $|\hat{T}| \leftarrow |\hat{T}| + 1$
10:    end if
11:    Increment $|T| \leftarrow |T| + 1$
12: end for
13: Compute $z$ and WR.
               $$z = \frac{|S|_G - \gamma|\hat{T}|}{\sqrt{\gamma(1-\gamma)|\hat{T}|}}, \quad \text{WR} = \frac{|\hat{T}|}{|T|}$$
14: return $z > \hat{z}$, WR, $|S|_G$
```

Entropy Threshold	MBPP Validation	MBPP Test	HumanEval Test
0.3	83.89	81.93	68.47
0.6	82.52	81.06	66.61
0.9	83.45	82.31	68.51
1.2	84.54	83.73	70.95
1.5	86.97	86.79	75.71

Table 4: Accuracy of Entropy Tagger for different entropy thresholds.

Algorithm 3 Algorithm for preprocessing of Entropy Tagger

```
1: Input: Original dataset $D = \{T_i\}$, where $T_i$ represents a sample containing a prompt and corresponding code.
2: Output: Preprocessed dataset $\hat{D} = \{(X_i, y_i)\}$, where $X_i$ is the feature vector and $y_i$ is the actual entropy.
3: for each sample $T_i$ in $D$ do
4:     Concatenate the prompt and code in $T_i$ to form a single sequence $S$.
5:     Initialize an empty list $\hat{S} = []$ to store truncated sequences.
6:     for $k = 1$ to length$(S)$ do
7:         Truncate $S$ to the first $k$ tokens to create $S_k$.
8:         Append $S_k$ to $\hat{S}$.
9:     end for
10:    for each truncated sequence $S_k$ in $\hat{S}$ do
11:        Compute the exact entropy $y_k$ of $S_k$ using StarCoder.
12:        Extract the feature vector $v_k$ for $S_k$ using the Unified Feature Extractor.
13:        Add $(v_k, y_k)$ to $\hat{D}$.
14:    end for
15: end for
16: Return: Preprocessed dataset $\hat{D}$.
```

C.2 Training

Ablation study on training objective We evaluate the accuracy of the Entropy Tagger under two training objectives: classification and regression. In the classification setting, the model is trained as a binary classifier to directly predict whether each token is low entropy. Accuracy is computed by comparing the predicted class label $\hat{y}_i \in \{0, 1\}$, with the ground-truth label $y_i \in \{0, 1\}$:

$$\text{Acc.}_{\text{cls}} = \frac{1}{N} \sum_{i=1}^{N} \mathbf{1}[\hat{y}_i = y_i] \tag{2}$$

In the regression setting, the model predicts a scalar entropy value $\hat{e}_i \in \mathbb{R}$. The ground-truth entropy value $e_i \in \mathbb{R}$ is also provided. We discretize both values into bins of width 0.3, capping the maximum bin value at 1.5, and evaluate accuracy by comparing the resulting discrete labels:

$$\text{Bin}(x) = \min\left(\left\lfloor \frac{x}{0.3} \right\rfloor \times 0.3,\ 1.5\right) \tag{3}$$

$$\text{Acc}_{\text{reg}} = \frac{1}{N}\sum_{i=1}^{N} \mathbf{1}\left[\text{Bin}(\hat{e}_i) = \text{Bin}(e_i)\right] \tag{4}$$

Tab. 5 demonstrates that the regression-based Entropy Tagger consistently underperforms the classification-based version in terms of accuracy across all three datasets. Consequently, we adopt the classification objective for training the Entropy Tagger.

Details on Training Entropy Tagger During the training phase, we construct a binary classification MLP, and then, based on the threshold τ, we map y in $\hat{D}$ to True or False. If $y_i < \tau$, it is set to True, otherwise False. We then train using BCELoss and optimize with AdamW (Loshchilov, 2017). The hyperparameter settings are shown in the Tab. 6. Finally, the epoch with the highest accuracy on the MBPP validation split is selected as the Entropy Tagger.

Hyperparameter	Setting
# epochs	100
batch_size	32
lr	1e-4
optimizer	AdamW
weight_decay	2e-5

Table 6: The hyperparameter settings for training the Entropy Tagger.

C.3 Validation

We use the MBPP test and HumanEval test as the test sets, representing the in-domain and out-of-domain scenarios, respectively. The test results are shown in Tab. 4. The results show that the accuracy of the Entropy Tagger is consistent across different splits of the same dataset (in-domain), achieving over 80%. When applied across datasets (out-of-domain), using the Entropy Tagger for prediction also achieves an accuracy of over 66.61%, with an accuracy of 75.71% at the $\tau = 1.5$.

D Implementation details

All methods can be implemented on a single *NVIDIA A100-SXM4-40GB*. For Post-hoc methods and KGW, we follow the implementation provided in the Lee et al., 2024. For EWD, we adopt the recommended hyperparameters from Lu et al., 2024. However, to ensure a fair comparison, we use the same hash key in KGW for EWD. For SWEET, we use the settings recommended in the original paper. Since the Threshold Navigator automatically selects a fixed threshold, we report the averaged results across all thresholds for SWEET. As SWEET consider the trade-off between code generation ability and detectability, two results are reported for MBPP. We select the one with the highest AUROC. For IE, we report the result with the highest UES under the condition that Pass@1 is allowed to drop by up to 20%. Detailed settings for each method on each dataset can be found in Tab. 7.

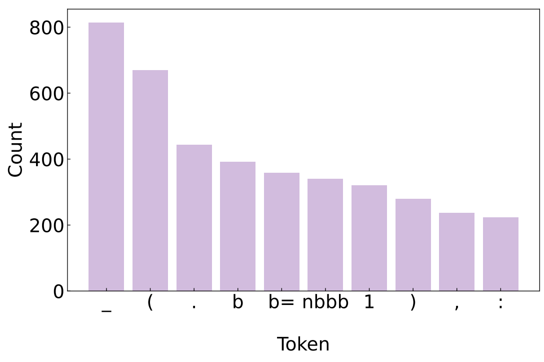
E Computational Time Used Analysis

To evaluate the computational efficiency of each method, we measure the total runtime required to complete evaluation on the `HumanEval` benchmark. Due to the variation in generated text lengths across different methods, all watermarking approaches are applied exclusively during the detection phase to ensure a fair comparison. Each method is evaluated three times under the same hardware conditions, and the average total runtime is reported. The results are summarized in Tab. 8.

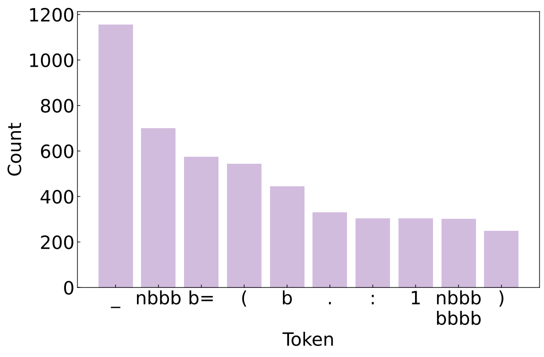
As shown in the Tab. 8, our method achieves the lowest total runtime, demonstrating its practical advantage in terms of computational efficiency.

F Algorithms for Threshold Navigator

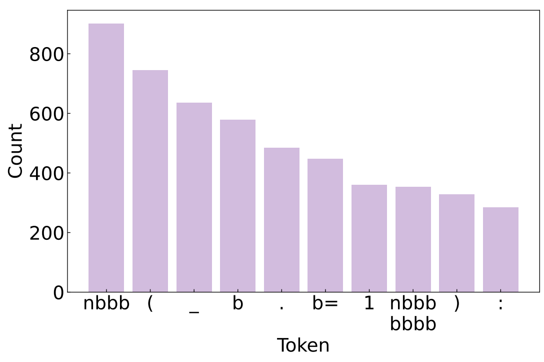
The algorithmic details of Threshold Navigator are shown in Alg. 4. Given a prompt sequence, green list size, and search granularity, we begin by initializing the entropy threshold τ_0 and computing the corresponding Watermark Ratio (WR_0) and the number of green tokens $|S|_{G_{\tau_0}}$ under this threshold. (Lines 3-5). Then, we enumerate downward from the initial entropy threshold (e.g., 1.5) to lower values (e.g., 1.2, 0.9, 0.6, 0.3). For each new entropy threshold, we compute the updated Watermark Ratio ($\text{WR}_{\tau_{i-1}}$) and green token count ($|S|_{G_{\tau_i}}$). (Lines 6-8) For every pair of adjacent entropy thresholds, we calculate the green token change ration p (Line 9) and the Watermark Ratio change ratio w (Line 10). The search stops when the condition $p > 1$ and $w < 1$ is met for the first time, and the previous entropy threshold is selected as the final threshold and returned (Lines 11-13).



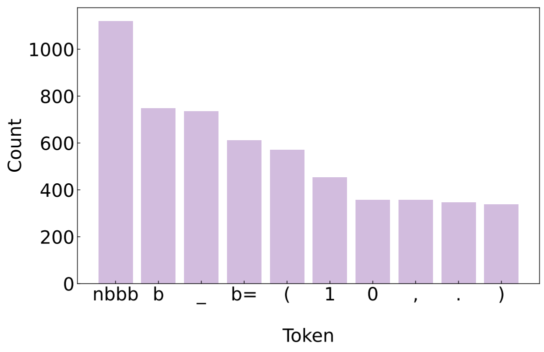
(a) entropy threshold (τ) = 0.3



(b) entropy threshold (τ) = 0.6



(c) entropy threshold (τ) = 0.9



(d) entropy threshold (τ) = 1.2

Figure 7: Top-K tokens most frequently classified as low entropy tokens.

Methods	**MBPP Validation**	**MBPP Test**	**HumanEval Test**
Regression	38.16	36.90	37.94
Classification	84.27	83.16	70.05

Table 5: Accuracy of Entropy Tagger for different training objectives.

Algorithm 4 Threshold Navigator

```
1: Input: prompt, $s_0, \dots, s_{t-1}$
          green list size, $\gamma \in (0, 1)$
          search granularity, $\Delta$
2: Output: $\hat{\tau}$ (final entropy threshold)
3: Initialize $\tau_0$ (initial entropy threshold)
4: $\hat{\tau} \leftarrow \tau_0$
5: Calculate WR$_0$ (Watermark Ratio) and $|S|_{G_{\tau_0}}$ (green token count).
6: for $i = 1$ to $\lfloor \tau_0 / \Delta \rfloor$ do
7:     $\tau_i \leftarrow \tau_{i-1} - \Delta$
8:     Calculate WR$_{\tau_i}$ and $|S|_{G_{\tau_i}}$.
9:     $p \leftarrow |S|_{G_{\tau_{i-1}}} / |S|_{G_{\tau_i}}$
10:    $w \leftarrow$ WR$_{\tau_{i-1}}$/WR$_{\tau_i}$
11:    if $p > 1$ and $w < 1$ then
12:        $\hat{\tau} \leftarrow \tau_{i-1}$
13:        break
14:    end if
15: end for
16: return $\hat{\tau}$
```

Dataset	Method	γ	δ
HumanEval	KGW	0.25	3.0
	EWD	0.5	2.0
	SWEET	0.25	3.0
	IE	0.5	3.0
MBPP	KGW	0.25	3.0
	EWD	0.5	2.0
	SWEET	0.5	2.0
	IE	0.25	3.0

Table 7: Detailed settings for each watermark methods.

Table 8: Total runtime (in seconds) on the `HumanEval` benchmark for each method.

Method	Total Time (s)
KGW	55.86(± 10.67)
EWD	118.83(± 11.82)
SWEET	110.75(± 11.29)
IE	100.36(± 6.53)

G Analysis on low entropy tokens

We rank the frequency of tokens classified as low entropy token under $\gamma = 0.25$ and $\delta = 3.0$ across different entropy thresholds and report the top 10 tokens. To enhance clarity, we use "b" to represent spaces and "n" to represent newlines. The results are shown in Fig. 7. It can be observed that, despite varying entropy thresholds, certain tokens frequently appear as low entropy tokens, such as "_", ".", ":", "1", "(", ")", spaces, newlines, and their combinations.