

## A Additional Considerations for Section 3.1

**Generalization to sampling.** In Section 3.1, we assume that the generation procedure is deterministic, i.e. that the model performs “greedy inference”. In practice, however, many applications work better with stochastic sampling [Holtzman et al., 2020]. However, this has an obvious caveat for Algorithm 1: if the text generation process is stochastic, a token can be deemed important based not on its actual impact on the model outputs, but on the randomness of the decoding procedure.

To generalize our approach for stochastic generation, we take advantage of the well-know Gumbel-max trick [Gumbel, 1954]. To recap, if we add independent Gumbel-distributed random variables to each predicted logit and take the index of the maximum, the probability that a certain index will be chosen is equal to the softmax of the original logits.

In case of Algorithm 1, we use Gumbel-max trick to reparameterize stochastic sampling from the model with a deterministic sampling conditioned on a pre-generated random state  $s \leftarrow \text{RANDBITS}(N)$ . Given a prompt  $x$ , a response prefix  $y_{1:t}$  and model parameters  $\theta$ , we generate the next token as follows:

$$y_{next} = \arg \max_i \log P(i|x \oplus y_{1:t}, \theta) + \text{GUMBELPRNG}(s \oplus x \oplus y_{1:t}),$$

where GUMBELPRNG is a function that samples a pseudo-random variable from standard Gumbel distribution based on an input seed  $s \oplus x \oplus y_{1:t}$ . To recall,  $\oplus$  denotes concatenation. This way,  $y_{next}$  is distributed as  $P(y_{next}|x \oplus y_{1:t}, \theta)$ , but it is deterministic when conditioned on the random state  $s$ . Hence, we sample a random state  $s$  once at the beginning of Algorithm 1, the entire procedure after that will also be conditionally deterministic (given  $s$ ). We test this further in Appendix F.

**Issues with naïve important token mining.** As we described earlier, Algorithm 1 is inherently sequential because it searches not for individual important tokens, but for important token combinations. In principle, it is tempting to consider a simpler algorithm that considers each token replacement in isolation and can run in parallel. However, when considering [target\_model\_gen\_0, draft\_token, target\_model\_gen\_1] sequences only, a sufficiently strong target model might recover from even a low-quality token and still produce the correct answer. This results in a failure mode where all tokens are individually unimportant, but when all such tokens are *jointly* replaced with their draft versions, the model fails to produce the correct answer. In our preliminary experiments, when using Llama-3.1-70B-Instruct Touvron et al. [2023] as the target model and Llama-3.2-1B-Instruct as the draft model, fewer than 1% of the tokens were labeled as important with this simplified algorithm, whereas our main Algorithm 1 found substantially. One interesting guarantee of Algorithm 1 over its naïve counterpart is that, whenever draft and target models produce different (non-equivalent) answers to a given prompt, our algorithm will find at least one important token, whereas the naïve algorithm may find none.

**On starting conditions for the important token search.** To recall, mining important tokens can be viewed as a shortest path search algorithm in a tree of possible mismatch choices. When performing this type of search, there are two possible directions that one can search from. In Algorithm 1, we start from the target model outputs and iteratively (greedily) replace the mismatching tokens with their draft versions. However, one could also start from the draft model outputs and iteratively swap in target model outputs until the answer becomes equivalent to that of the target model. If we were to use an exhaustive search algorithm, both approaches would converge to the same important token labeling. However, since we are using a semi-greedy algorithm, it is easier to start with an already correct solution and simplify it, as opposed to starting with a wrong one and attempting to fix it.

## B Additional Details on Classifier Training

As we discussed earlier in Section 3.2, there are several important design choices that can affect the performance of an important token classifier in our setting. In this part of supplementary materials, we report the experiments that led us to use a linear classifier based on draft token embeddings encoded with both  $\theta_{draft}$  and  $\theta_{main}$ . To that end, we compare the different classifier variants using the important token embeddings from the GSM8K [Cobbe et al., 2021] training subset (see Section 4.1).

To compare different classifier configurations, we further divide the GSM8K training set into classifier training (90%) and validation (10%) subsets. We perform this division at sample level, i.e. all labeled tokens from a given GSM8K sample are used either entirely for classifier training, or entirely for validation. We use the same training and validation subsets throughout this section.

For the first set of experiments (Figure 5), we compare regularizer coefficients for Logistic Regression (left). We also report different classifier types: Logistic Regression, a Random Forest with 128 trees and a multi-layer perceptron (MLP) with a single hidden layer consisting of 128 hidden units with ReLU activation. For consistency, we run all models using Scikit-Learn [Pedregosa et al., 2011] v1.4.2 with all other settings kept to their default values. For MLP, we perform early stopping on yet another 10% subset of the training set with built-in default MLPClassifier early stopping parameters. For this evaluation, all classifiers use draft and target model hidden states (concatenated) encoding the draft token, which is our main setup from Section 3.2.

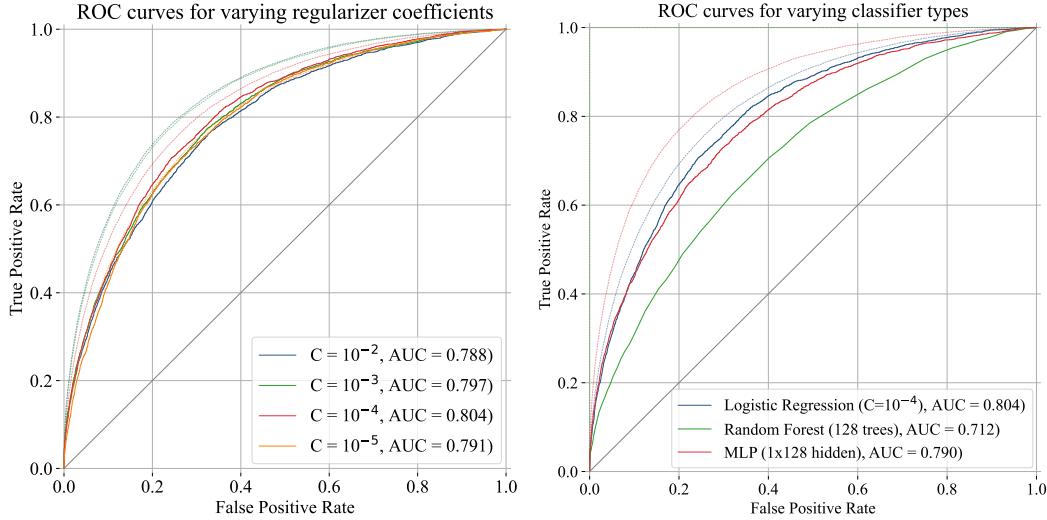


Figure 5: Receiver Operating Characteristics and the corresponding AUC values values for different Logistic Regression regularizers (left) and classifier types (right). Bold lines are validation curves and the dotted lines represent training curves. The AUC is reported in the legend (bottom right).

The results in Figure 5 demonstrate that the classifier quality is fairly robust to the choice of the regularization hyperparameter. It is also fairly robust to the choice of the classifier architecture, barring perhaps the Random Forest classifier, which is overfitting the training data more than other models. Note that this does not necessarily mean that the MLP or tree-based classifiers are generally worse than linear models — only that linear model is enough in our exact setup with a limited training set. We hypothesize that, if allowed to train on much larger dataset, the more complex models will be able to match and possibly outperform logistic regression.

Next, we compare classifier **inputs**. As we describe in Section 3.2, we use existing LLM hidden states from the last layer of  $\theta_{draft}$  and  $\theta_{target}$  since they are already computed during speculative decoding. This, however, leaves several possible choices about which hidden states should be used:

- **Previous token embeddings**, last hidden states used to predict the mismatching token;
- **Draft token embeddings** are the next embeddings, obtained by encoding the draft token;
- **Target token embeddings** are the next embeddings, obtained by encoding the target token;
- **Both token embeddings** are concatenations of the draft and target token embeddings;

We compare the four input configurations in Figure 6 (left), using Logistic Regression with  $C=10^{-4}$  and both draft and target model hidden states (concatenated) for each case. The results suggest that a classifier that uses mismatching token embeddings (for draft *or* target token) is significantly more accurate than using the preceding token embeddings (the ones used to predict the mismatch). In turn, using both token embeddings results in somewhat better performance than either of them. However, using both token embeddings introduces complications during inference time.

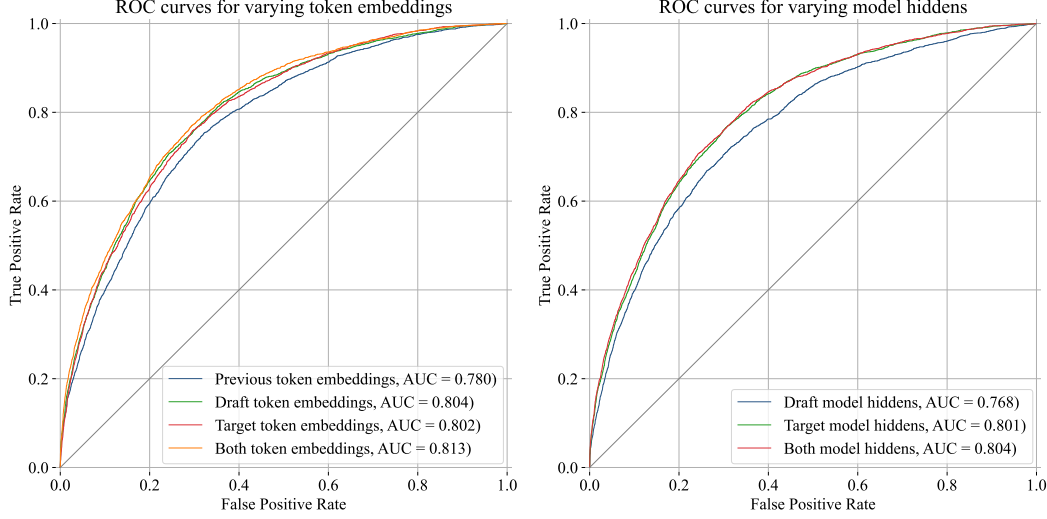


Figure 6: Receiver Operating Characteristics and the corresponding AUC scores (in the plot legend) for different classifier input tokens (left) and models (right). See Appendix B for details.

In normal speculative decoding, the algorithm already computes hidden states for draft tokens with both  $\theta_{draft}$  (during draft generation) and  $\theta_{target}$  (during verification). However, it does not compute embeddings for target tokens since those tokens are not known before the end of the verification stage — and computing them already requires a forward pass with  $\theta_{target}$ . As a result, computing target (or both draft & target) token embeddings would require two sequential forward passes with  $\theta_{target}$  — one to determine the target tokens and detect mismatches, and the other to compute embeddings for those mismatching target tokens. In principle, one could devise a more sophisticated algorithm that computes only the  $\theta_{draft}$  embeddings for mismatching target tokens or guesses the target tokens prior to the verification stage, but doing so would greatly complicate the implementation. Since the increase in the AUC score compared to using just the draft token embeddings is relatively small (Figure 6, on the left), we default to using draft token embeddings.

Additionally, we also test three model hidden states configurations for draft token embeddings: draft model hidden states, target model hidden states, and concatenated hidden states from both models in Figure 6 (right). Here, using the target model hidden states results in superior accuracy to using the draft model. In turn, using both  $\theta_{draft}$  and  $\theta_{target}$  produces an additional, if marginal, increase in accuracy. However, since both hidden states are already available during inference, using them both does not pose additional complications. Though, some real world inference systems may make it more convenient to only use  $\theta_{target}$  for classifier inputs since the AUC difference is within 1%.

## C Precision Matters for Speculative Decoding

When validating the AutoJudge algorithm, we found a peculiar implementation detail that can affect the real world performance of speculative decoding. Namely, *when using the LLM in half precision, token embeddings can differ significantly (up to 10%) between parallel and sequential forward passes on the same data*. In other words, if we record model hidden states as it generates a sequence, then encode the same sequence in parallel to recompute said hidden states, the two sets of hidden states will not match exactly. We attribute this to the fact that encoding tokens in parallel has a different summation order to encoding tokens one by one, which introduces small numeric errors. These errors compound over consecutive layers, resulting in larger errors in the final hidden states.

This is important for AutoJudge since the token labeling Algorithm 1 runs sequential inference with  $\theta_{target}$  and parallel inference on  $\theta_{draft}$ , whereas inference-time speculative decoding does it the other way around: sequential calculations of  $\theta_{draft}$  during the draft generation phase, then parallel forward pass with  $\theta_{target}$  during the verification phase. As a result, the classifier is trained on features that can be significantly different from what they would be during inference. In contrast, running in full precision (float32) does not have such problems.

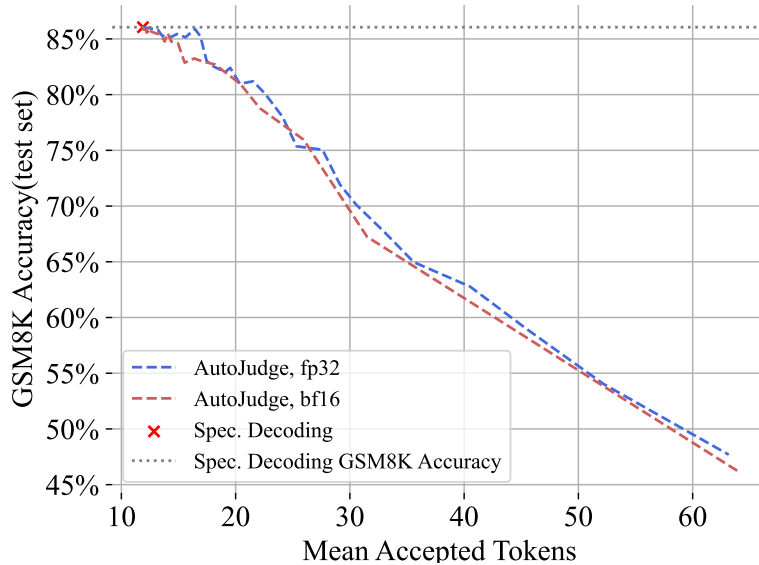


Figure 7: Accuracy on GSM8K and the number of accepted tokens per speculative decoding cycle in float32 and bfloat16 precision. The setup is the same as in Section 4.1.

In Figure 7, we compare accuracy and acceptance rate trade-offs for different classifier thresholds in the same setup as in Section 4.1. There are several ways to circumvent this problem. The most practical one would be to recompute target model embeddings for Algorithm 1 in a parallel forward pass and *not* using the draft model embeddings (since adding them has negligible effect on accuracy, see Figure 6, right). As a result, the classifier would use  $\theta_{target}$  embeddings computed in parallel over draft tokens during both training and inference.

## D Additional Evaluations

In this section, we provide an additional (if limited) evaluation on the MMLU-Pro benchmark [Wang et al., 2024]. This is a general reasoning problem set in the form of multiple choice questions. The model is asked a question with 10 options (A-J) and allowed to reason about which of the options is correct. As before, we evaluate accuracy using the benchmark’s original evaluation protocol.

Due to compute budget restrictions<sup>6</sup>, we train AutoJudge classifier on a small subset of 4925 training samples chosen at random and evaluate on 1024 samples from the test subset, also chosen randomly.

We report our results in Figure 8. To summarize, Autojudge managed to provide some acceleration at the cost of accuracy drawdown, though the accuracy drawdown is a bit faster than for the benchmarks where we trained on the entire training set (see Section 4). We attribute this effect to classifier overfitting.

### D.1 Additional Evaluations for Section 4.1

In Figure 9, we report accuracy and the number of accepted tokens for 1B draft / 8B target and 1B draft / 70B target model pairs in 8 shot setup. In Figure 10, we report additional threshold configurations for AutoJudge and additional values of  $K$  for the Top- $K$  baseline, extending Figures 3 and 4. Additionally, we evaluate the AutoJudge classifier trained on LiveCodeBench on GSM8K and vice versa to gauge the effect of task-specific training. Predictably, these out-of-domain classifiers perform significantly worse. We attribute this to the fact that the GSM8K-trained classifier likely did not see any Python source code, whereas the LiveCodeBench classifier did not perform arithmetic operations and did not solve equations that are common in GSM8K.

<sup>6</sup>we use preemptible GPU instances throughout our experiments and cannot control their availability

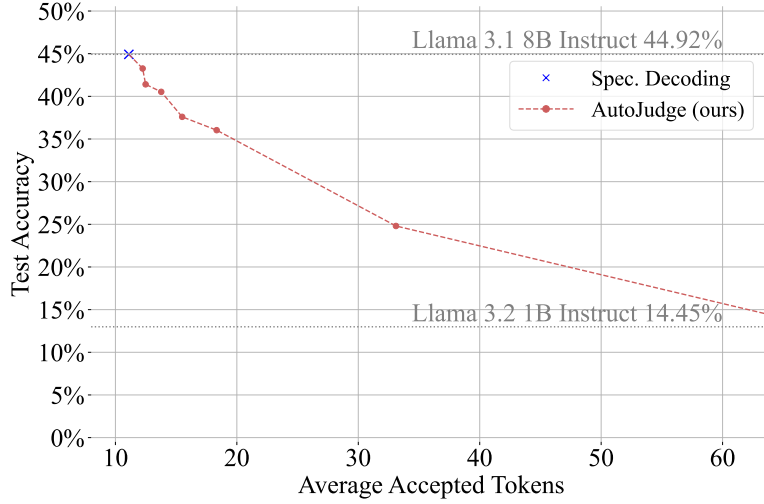


Figure 8: MMLU-Pro accuracy (a subset of 1024 test samples) and mean accepted tokens for Llama 3.2 1B draft and Llama 3.1 8B target model pair, when trained on a subset of training examples.

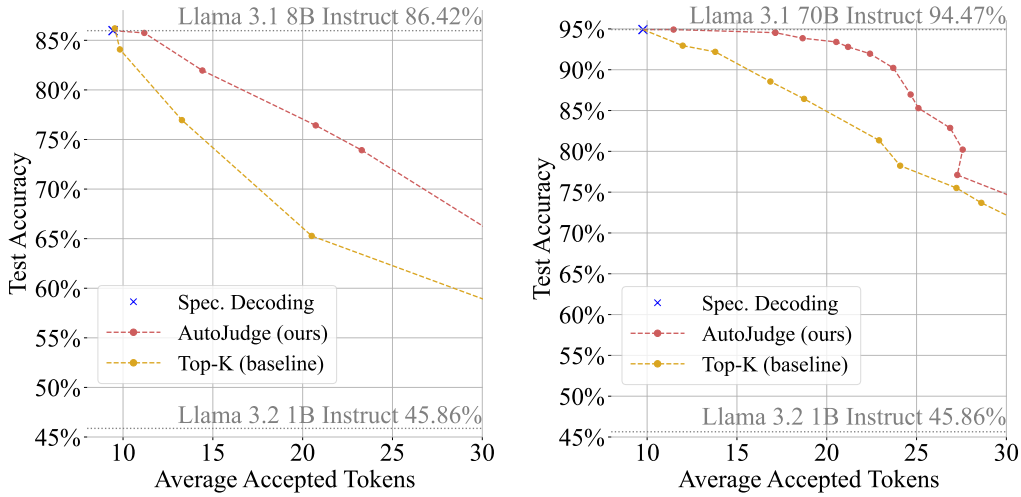


Figure 9: Downstream accuracy and the average number of accepted tokens for GSM8K 8-shot with Llama-3.1-8B-Instruct target and Llama-3.2-1B-Instruct draft models (left) and Llama-3.1-70B-Instruct target and Llama-3.2-1B-Instruct draft models (right)

## D.2 Additional Evaluations for Sections 4.2

In this section, we provide additional classifier threshold evaluations for both model pairs in our LiveCodeBench setup. These results are reported in Figure 11, with 1B draft / 8B target models on the left and 8B draft / 70B target on the right. We use the same setup as in Section 4.2 and the portion of the results at the top are exactly the same values that we reported in Figure 4.

## E vLLM Inference Implementation Details

To measure time efficiency of AutoJudge, we incorporate it into vLLM inference library. The integration is built upon `vllm==0.8.5`, `torch==2.7.0` with CUDA 12.6 and `transformers==4.51.3`. We set window size 32 and batch size 1. The implementation allows to perform batched inference without modifications, but we choose to evaluate with batch size 1 for accurate measurements. For efficiency the current implementation predicts important tokens using only hidden state of the target model, as we have shown that adding hidden states of the draft model does not substantially increase

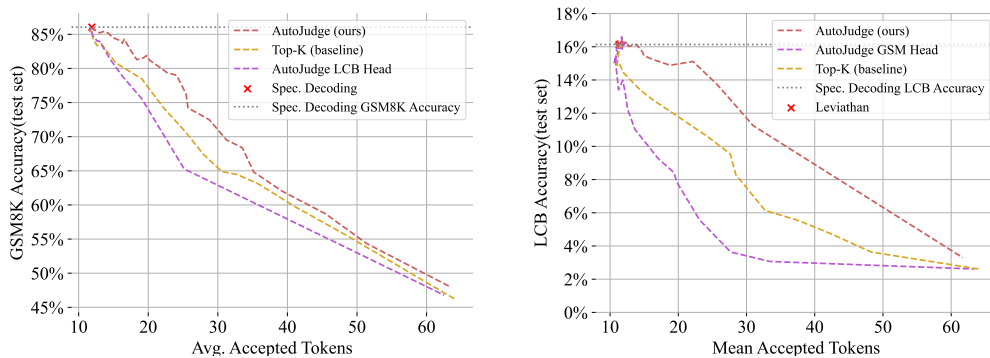


Figure 10: Downstream accuracy and the average number of accepted tokens for GSM8K (left) and LiveCodeBench (right) with Llama-3.1-8B-Instruct target and Llama-3.2-1B-Instruct draft models.

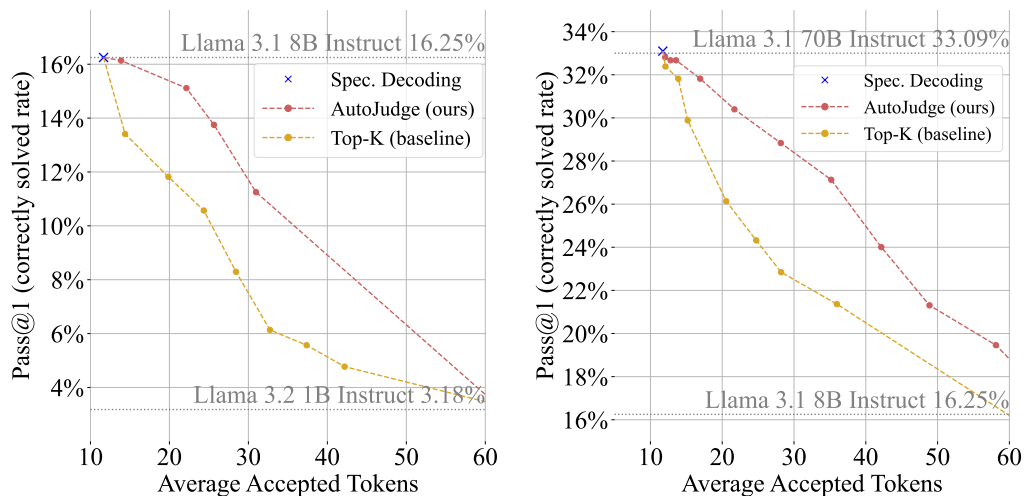


Figure 11: Downstream accuracy and the average number of accepted tokens for LiveCodeBench with Llama-3.1-8B-Instruct target and Llama-3.2-1B-Instruct draft models (left) and Llama-3.1-70B-Instruct target and Llama-3.1-8B-Instruct draft models (right)

the accuracy. Evaluations are run on NVIDIA A100-SXM4-80GB GPUs: one for 8B/1B model pair and four for 70B/8B pair in tensor parallel setup.

## F Additional Inference Benchmarks

In this section, we report additional runtime benchmarks that extend our evaluations from Section 4.3. Namely, we run speculative decoding with AutoJudge classifier on LiveCodeBench dataset and for the GSM8K in 8-shot (as opposed to zero-shot) configuration. We report our results for GSM8K (8-shot) in Table 2 and our results for LiveCodeBench in Table 3. In both figures, we use the same experiment configuration as in Section 4.3. The results follow the same trend we observed earlier, with AutoJudge decoding achieving significant speedups over standard speculative decoding, up to  $1.96\times$  in Table 2 (right) for the 8B draft / 70B target model pair. The slower speed of both speculative decoding and our approach is due to the longer prefill phase in the 8-shot setup compared to the 0-shot setup that is provided in Table 1 (right).

Additionally, in Table 2 (left) we evaluate a variant of AutoJudge decoding that uses stochastic sampling during both important token search and actual decoding. We use the Gumbel-max trick to implement conditionally deterministic sampling as described in Appendix A. Note that this change only affects the important token mining phase and the sampling procedure during actual inference and evaluation does not need to be changed. Our results suggest that AutoJudge speculative decoding can indeed generalize to stochastic sampling, making it applicable for models and tasks where this

Table 2: (left) Inference speed benchmarks on GSM8K (0-shot) for 1B draft / 8B target model with **stochastic sampling** (Appendix A) and (right) on GSM8K (8-shot) for 8B draft / 70B target model with greedy inference. In (left), threshold 0.0 represents speculative decoding setup.

Llama 3.1 1B draft / 3.1 8B target <b>sampling</b>					Llama 3.1 8B draft / 3.1 70B target <b>greedy</b>				
Threshold	0.0	0.175	0.25	<b>0.3</b>	Threshold	0.03	0.08	0.13	<b>0.26</b>
Accuracy, %	82	79	78	<b>77</b>	Accuracy, %	95	95	95	<b>93</b>
Speed, tokens/s	79.7	84	87	<b>90</b>	Speed, tokens/s	57	69	75	<b>81</b>
<i>Speculative Decoding: 79.7 tokens/s</i>					<i>Speculative Decoding: 36.7 tokens/s</i>				
Speedup(ours)	1.0	1.05	1.09	<b>1.13</b>	Speedup(ours)	1.56	1.88	2.05	<b>2.19</b>

Table 3: Inference speed benchmarks on LiveCodebench with vLLM implementation for 1B draft / 8B target models (left) and for 8B draft / 70B target models (right).

Llama 3.2 1B draft / 3.1 8B target					Llama 3.1 8B draft / 3.1 70B target				
Threshold	0.03	<b>0.06</b>	0.08	0.11	Threshold	0.05	0.075	0.1	<b>0.125</b>
Accuracy, %	16	<b>15</b>	13	3	Accuracy, %	33	32	30	<b>29</b>
Speed, tokens/s	115	<b>157</b>	178	192	Speed, tokens/s	58	65	89	<b>96</b>
<i>Speculative Decoding: 75.4 tokens/s</i>					<i>Speculative Decoding: 40.5 tokens/s</i>				
Speedup(ours)	1.53	<b>2.08</b>	2.36	2.55	Speedup(ours)	1.43	1.60	2.19	<b>2.37</b>

1101 type of inference is preferable. Sampling with different seeds can also be seen as a way to generate  
1102 more training data.

## 1103 G Hardware Configurations

1104 We run our experiments primarily on A100-SXM4 GPUs with 80GB DRAM on servers with dual  
1105 Epyc 7742 CPU and 1TiB RAM. For model pairs that do not fit on a single GPU, we use distributed  
1106 inference with naive model parallelism (`device_map="auto"`) when using `transformers` and  
1107 tensor parallelism for vLLM experiments.

1108 The actual time per experiment varies by the dataset and model pair: 1B draft / 8B target model pair  
1109 takes, on average, 65.6 seconds to process a GSM8K example, whereas the 8B draft / 70B target  
1110 takes up an average of 706.4 seconds per sample. On LiveCodeBench, the same 8B draft / 70B target  
1111 model takes up 449 seconds per sample. Since Algorithm 1 can run independently for each sample,  
1112 we were able to run our code on low-priority preemptible hardware. However, this also makes it hard  
1113 to measure the exact amount of computations used in our experiments since some of them were lost  
1114 due to preemption. For running on a single A100 server, please refer to the time per sample above  
1115 to estimate the total runtime requirements. Please also note that there are ways to mine important  
1116 tokens more efficiently using APIs (below).

### 1117 G.1 Scaling Up Algorithm 1 with API Calls

1118 We would also like to mention that it is possible to scale up the important token discovery in  
1119 AutoJudge by reframing it in terms of API calls. Note that the search algorithm only ever runs regular  
1120 generation (greedy or sampling) with the target model and runs parallel forward pass on  $\theta_{draft}$ .

1121 Hence, we can run Algorithm 1 by replacing `GENERATE(...)` on lines 4 and 10 with a call to an  
1122 LLM generation API with the specified input. With this reframing, we can mine important tokens by  
1123 querying LLM API providers even if they cannot inference the large target model locally. This can  
1124 help in a number of use cases: for instance, when developing a speculative decoding algorithm for  
1125 use with offloading, e.g. [Svirshchevski et al., 2024, Miao et al., 2023].

1126 Additionally, modern open-source inference libraries for LLMs often expose an OpenAI-  
1127 compatible API. This way, one can run the important token mining algorithm efficiently over a  
1128 pool of LLM inference servers, taking advantage of server-side batching and optimized kernels.

1129 Note, however, that this regime requires tokenizing and detokenizing the target model’s messages  
1130 received from API calls, since most public services operate on non-tokenized text strings. Since there  
1131 are several ways to spell the same text with a given BPE merge table, this can sometimes lead to  
1132 unintentional “token healing”.

1133 As a part of our supplementary code, we release a variant of important token mining algorithm that  
1134 uses an API to run the target model, allowing the user to learn AutoJudge classifier on top of models  
1135 that do not fit on their local hardware.

## 1136 **H Dataset and Model Licenses**

1137 As encouraged by NeurIPS paper checklist, we summarize our use of licensed models and datasets.

- 1138 • GSM8K benchmark [Cobbe et al., 2021] is licensed under MIT License;
- 1139 • LiveCodeBench benchmark [Jain et al., 2024] is licensed under MIT License;
- 1140 • MMLU-Pro benchmark [Wang et al., 2024] is licensed under Apache-2.0 license;
- 1141 • Llama 3.1 and 3.2 models Dubey et al. [2024] are under Llama Community License Agreement.

1142 Our work would also be impossible without the open-source software including (but not limited to)  
1143 PyTorch [Paszke et al., 2019], Hugging Face Transformers [Wolf et al., 2019], vLLM [Kwon et al.,  
1144 2023] and hundreds of other open-source packages in the Python data science & machine learning  
1145 ecosystem. Enumerating and acknowledging all these individual packages would take up several  
1146 pages, but they can be recovered automatically from our supplementary code.