Neuro-Symbolic Language Modeling with Automaton-augmented Retrieval

Uri Alon [1] **Frank F. Xu** [1] **Junxian He** [1] **Sudipta Sengupta** [2] **Dan Roth** [3] **Graham Neubig** [1]
[1]Language Technologies Institute, Carnegie Mellon University [2]Amazon AWS [3]AWS AI Labs
{ualon,fangzhex,junxianh,gneubig}@cs.cmu.edu {sudipta,drot}@amazon.com

Abstract

Retrieval-based language models (R-LM) model the probability of natural language text by combining a standard language model (LM) with examples retrieved from an external datastore at test time. While effective, a major bottleneck of using these models in practice is the computationally costly datastore search, which can be performed as frequently as every time step. In this paper, we present RETOMATON – retrieval automaton – which approximates the datastore search, based on (1) saving *pointers* between consecutive datastore entries, and (2) clustering of entries into "states". This effectively results in a *weighted finite automaton* built on top of the datastore, instead of representing the datastore as a flat list. The creation of the automaton is *unsupervised*, and a RETOMATON can be constructed from any text collection: either the original training corpus or from another domain. Traversing this automaton at inference time, in parallel to the LM inference, reduces its perplexity by up to 1.85, or alternatively saves up to 83% of the nearest neighbor searches over kNN-LM (Khandelwal et al., 2020) without hurting perplexity. Our code and trained models are available at `https://github.com/neulab/retomaton` . This is a workshop version of the longer paper that appeared in ICML'2022 (Alon et al., 2022).

1. Introduction

Retrieval-based language models (R-LMs) have recently been shown to improve over standard neural models in a variety of tasks such as unconditional language modeling (Guu et al., 2018; He et al., 2020), machine translation (Zhang et al., 2018; Gu et al., 2018; Khandelwal et al., 2021), question answering (Karpukhin et al., 2020; Ram et al., 2021), and code generation (Hayati et al., 2018; Hashimoto et al., 2018). The key ingredient of R-LMs is their ability to utilize training examples at test time without having to rely on the information encoded in the model's weights only.

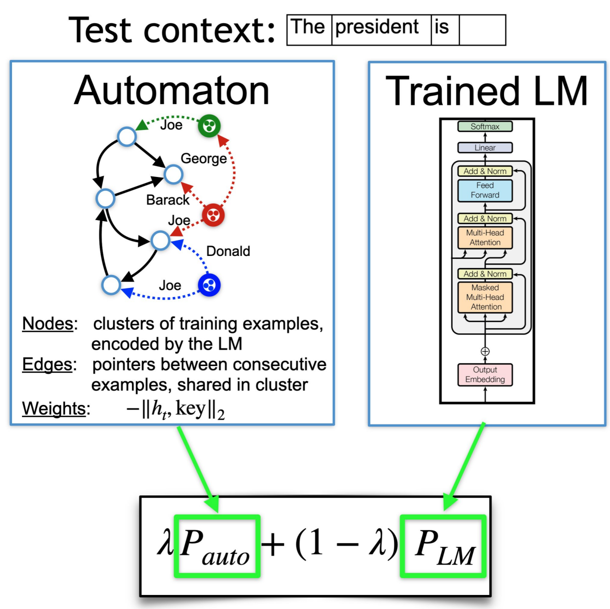

Figure 1: An illustration of our approach – RETOMATON.

In these models, the retrieval component first searches for nearest neighbor examples in an external datastore; then, the base model references these examples during the prediction. This fusion of language models (LMs) and retrieval improves the base language model from several perspectives. Nevertheless, the most critical bottleneck of these models is their *frequent search* over the datastore, which hinders the use of R-LMs in practical settings.

***k*-Nearest Neighbors Language Model** One prominent example of such a retrieval-based model is kNN-LM (Grave et al., 2017; Khandelwal et al., 2020), which predicts a token

* Equal contribution [1]Language Technologies Institute, Carnegie Mellon University [2]Amazon AWS [3]Amazon. Correspondence to: Uri Alon <ualon@cs.cmu.edu>.

ICML workshop on Knowledge Retrieval and Language Models, Baltimore, Maryland, USA, 2022.

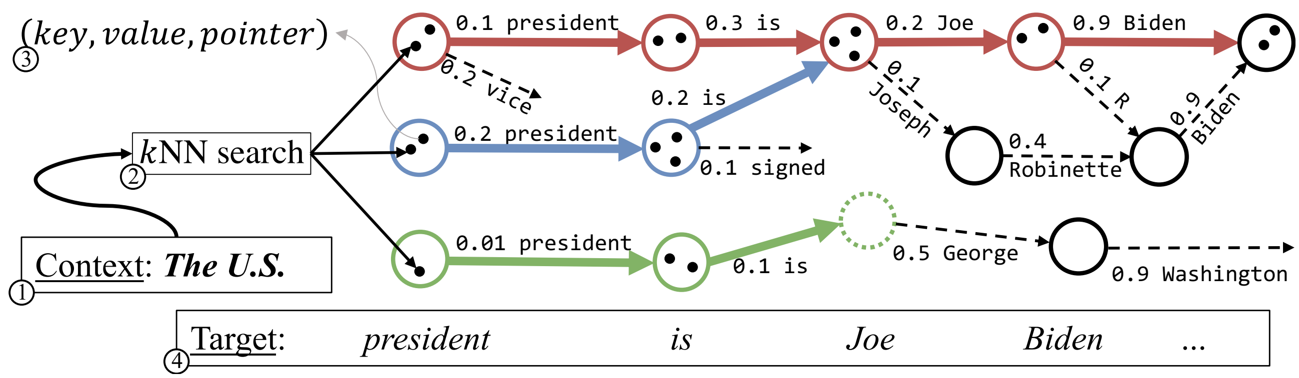

Figure 2: An illustration of RETOMATON. Given a context ① ("*The U.S.*"), a k-nearest neighbor search ② returns the nearest datastore entries. Every datastore entry (• in the figure) is a 3-tuple of $(key, value, pointer)$ ③, where the key is the LM's hidden state and the $value$ is the target token as in Khandelwal et al. (2020); the $pointer$ points to the datastore entry that appears next in the corpus. Close datastore entries are clustered together, and form an automaton state (○,○,○). The pointers of the clustered entries form the state's possible transitions. At inference time, the model decodes ④ while performing multiple parallel traversals (→, →, →) on the automaton to find useful datastore entries, instead of performing a full kNN search. Dashed arrows (⇢) denote allowed automaton transitions that were not taken during the current decoding.

by linearly interpolating the base LM's output with a non-parametric nearest neighbor distribution. This distribution is constructed by searching for the k-nearest neighbors (kNN) in the datastore and weighting them according to their distance to the current test context. Notably, this k-nearest neighbor search is performed *for every generated test token*, introducing severe inference overhead, since this search is significantly slower than the LM's standard "forward pass".

Our Approach: **RETOMATON** Our main insight is that *retrieved neighbors at the current time step also hint at the neighbors that will be retrieved at future time steps*, and can thus save repetitive searches later. Specifically, we construct a *weighted finite automaton* (WFA) on top of an existing datastore, by keeping pointers between datastore entries and clustering similar entries, in a *completely unsupervised* way. This automaton allows sporadic, infrequent, exact kNN searches, and a much cheaper traversal of the automaton at other time steps. We call our model RETOMATON – retrieval automaton. RETOMATON is illustrated in Figure 2.

Concretely, applying RETOMATON to a strong WIKITEXT-103 LM using only the original training set allows for reducing the perplexity, or alternatively, saving 81% of the kNN searches of Khandelwal et al. (2020) without hurting perplexity and without additional training other than clustering. If we do not allow clustering, we can still save 60% of the kNN searches. We also show that RETOMATON allows for effective domain adaptation, by simply constructing a RETOMATON for a different domain. When we construct RETOMATON on top of a *fine-tuned* LM, we decrease the perplexity by more than 17% compared to just fine-tuning. Finally, we perform a thorough ablation study, showing the contribution of the different components and analyzing the tradeoff between coarse- vs. fine-grained clustering.

This is a workshop version of the longer paper that appeared in ICML'2022 (Alon et al., 2022).

2. Building Automata from Datastores

To save kNN searches, we build a WFA on top of the kNN-LM datastore. Then, we traverse the automaton to estimate the next nearest neighbors. We will demonstrate the automaton creation process using Figure 3 as a running example.

2.1. Definitions

Given a finite vocabulary Σ and the set Σ^* of finite sequences over Σ, a trained autoregressive LM defines a distribution p_{LM} over Σ for any context $c \in \Sigma^*$. Given such an LM having $f : \Sigma^* \rightarrow \mathbb{R}^d$ and a text collection $\mathcal{D}$, kNN-LM Khandelwal et al. (2020) constructs a datastore using a single forward pass over $\mathcal{D}$ (detailed in Appendix A). As shown in Figure 3, this results in a set of key-value entries such as (○○●, *president*), where the key is the LM encoding of every prefix in $\mathcal{D}$, and the value is the following token.

Weighted Finite Automaton Our automaton is a tuple $A = \langle Q, \Sigma, q_0, \delta, \phi \rangle$ such that Q is a finite set of states, Σ is the LM's vocabulary, $q_0 \in Q$ is the initial state, $\delta : Q \times \Sigma \rightarrow \mathbb{P}(Q)$ is a transition function where $\mathbb{P}(Q)$ denotes the power set of Q, and $\phi : Q \times \mathbb{R}^d \times \Sigma \rightarrow \mathbb{R}$ is a transition weight function. Unlike standard WFAs, the transition weights are dynamic: notice that ϕ depends also on a vector, in addition to a previous state and an input token. Also, note that δ is defined such that we transition into a set of states.

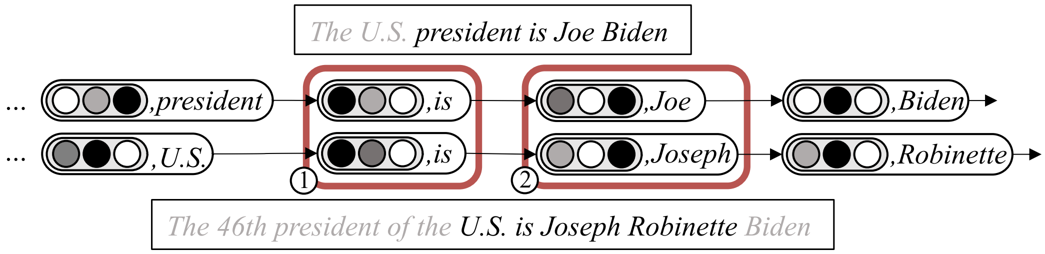

Figure 3: An illustration of our automaton creation process. The text $\mathcal{D}$ contains two sentences: (i) "*The U.S. president is Joe Biden*"; and (ii) "*The 46th president of the U.S. is Joseph Robinette Biden*" . Each datastore entry is a pair of a key vector encoding the prefix and a value representing the next token, such as ,*president*. We save a pointer from every entry to its successor in the text, and we cluster close key vectors to form automaton states (① and ②) that share pointers.

2.2. Constructing the Automaton

Pointers Our main insight is that during the creation of the datastore, we can keep a *pointer* from every datastore entry to the entry that appears next in the text $\mathcal{D}$. Imagine that at time t, the test context is $c^{(t)} = \left(w^{(1)}, ..., w^{(t-1)}\right)$, and the model retrieves a datastore entry $\left(f\left(c_i\right), w_i\right)$. If after the interpolation with the base LM (Equation (9)), the model's generated token $w^{(t)}$ (by sampling or argmax) is equal to w_i, then the entry $\left(f\left(c_{i+1}\right), w_{i+1}\right)$ is likely to be a useful entry at time $t+1$, because c_i was a near neighbor of $c^{(t)}$, and both these contexts were followed by the same token $w_i = w^{(t)}$. That is, our underlying assumption can be formulated as:

$$f\left(c_i\right) \approx f\left(c^{(t)}\right) \implies f\left(c_i \cdot w\right) \approx f\left(c^{(t)} \cdot w\right) \quad (1)$$

where $\approx$ denotes vector similarity, and $c^{(t)} \cdot w$ is the continuation of the context $c^{(t)}$ using the token w.

Thus, given the i-th example $\left(c_i, w_i\right) \in \mathcal{D}$, instead of keeping only the key-value pair $\left(f\left(c_i\right), w_i\right)$, every datastore entry is now $\left(f\left(c_i\right), w_i, p_i\right) \in (\mathcal{K}, \mathcal{V}, \mathcal{P})$, where p_i is a pointer to the next entry, or its index in the datastore. This is illustrated as arrows in Figure 3, where every entry has a pointer to the entry that followed it in the text $\mathcal{D}$. For example, the entry ,*president* points to ,*is*.

States If every entry had only a single outgoing pointer, the automaton would only capture n-grams that appeared in the text *verbatim*, and thus will not be able to generalize to unseen phrases. For example in Figure 3, the sequence "*The 46th president of the U.S. is Joe Biden*" would *not* be captured, because the prefix "*The 46th president of the U.S.*" appeared in one sentence, and the suffix "*Joe Biden*" appeared in another sentence. Thus, to allow entries to *share* their pointers, we cluster entries having close keys into an automaton *state*. A state includes all entries in the cluster, and the entries' pointers are the state's allowed outgoing transitions. For example, in Figure 3, the entry ,*is* is clustered with another entry having the same value, ,*is* (surrounded by ◯ and marked as ①). Furthermore, we can also cluster similar contexts that do *not* have the same value, such as cluster ② containing ,*Joe* and ,*Joseph*, which allows capturing the phrase "*Jospeh Biden*".

This step can be performed using any clustering algorithm such as k-means. We experiment with different values of k_{clust} and clustering algorithms in Appendix D.

Transition States Given a source state $q \in Q$ and an input token $w \in \Sigma$, we define the set of allowable next states as the clusters of entries that are pointed to by a datastore entry in q having the value w. Formally, let $\pi : (\mathcal{K}, \mathcal{V}, \mathcal{P}) \rightarrow Q$ be a function that maps every datastore entry into its containing state, and the function $\pi^{-1} : Q \rightarrow \mathbb{P}\left((\mathcal{K}, \mathcal{V}, \mathcal{P})\right)$, which maps every state into the *set* of datastore entries contained in it. Let $\rho : \mathcal{P} \rightarrow (\mathcal{K}, \mathcal{V}, \mathcal{P})$, be the "dereferencing" operator, which returns a datastore entry given a pointer that points to it. We can now define the allowed transitions as:

$$\delta\left(q, w\right) = \left\{ \pi\left(\rho\left(p\right)\right) \mid \left(k, w, p\right) \in \pi^{-1}\left(q\right) \right\} \quad (2)$$

This is illustrated in Figure 3 as the outgoing pointers of cluster ②, which allow transitioning from ② to different states given the input tokens w ="*Joe*" or w ="*Joseph*". This results in a (currently non-weighted) finite-state automaton, whose nodes are clusters of datastore entries, and whose edges represent the successiveness of entries in $\mathcal{D}$, where successiveness is shared by similar entries.

2.3. Traversal of the Automaton

At test time, given a test context $c^{(t)}$, we traverse the automaton while visiting *multiple* states at every time step, marked in Figure 2 as ➔, ➔, ➔. A traversal begins with a full kNN search of the datastore to retrieve the k-nearest neighbors $\mathcal{N}^{(t)}$ of $f\left(c^{(t)}\right)$. The initial traversal states $\mathcal{S}^{(t)} \subseteq Q$ are the union of the states to which these k-nearest neighbors belong: $\mathcal{S}^{(t)} = \bigcup_{e \in \mathcal{N}^{(t)}} \pi\left(e\right)$.

In the next time step $t+1$, given a token $w^{(t)} \in \Sigma$ that was generated (by argmax or sampling) by the model at time t, we compute the union of all transitions from states $q \in \mathcal{S}^{(t)}$.

We define a function $\hat{\delta} : \mathbb{P}(Q) \times \Sigma \rightarrow \mathbb{P}(Q)$ as follows:

$$\hat{\delta}\left(\mathcal{S}, w\right) = \bigcup_{q \in \mathcal{S}} \delta\left(q, w\right) \tag{3}$$

The decision of whether to continue traversing or start a new traversal can be made in several ways. We experimented with several alternatives, but found the most intuitive way to simply be whether the number of new states is greater than or equal to a threshold τ. That is, we continue traversing if $|\hat{\delta}\left(\mathcal{S}^{(t)}, w^{(t)}\right)| \geq \tau$, or start a new traversal otherwise.

When we *continue traversing*, we take the new states as our states in the next step. When we *start a new traversal*, we perform a new kNN search resulting in $\mathcal{N}^{(t+1)}$, but also include the remaining states we obtained so far $\hat{\delta}\left(\mathcal{S}^{(t)}, w^{(t)}\right)$.

Varying τ allows us to control the trade-off between higher accuracy but frequent traversal restarts, and thus frequent kNN searches (high τ), versus lower accuracy but with rare kNN searches, which saves time (low τ). For additional intuition for τ, see Appendix G.

Transition Weights Given a set of states $\mathcal{S}$, a test context c, and an input token $w \in \Sigma$, we define the transition weight from every $q \in \mathcal{S}$ similarly to Equation (8), except that we sum exponents of negative distances between $f\left(c\right)$ and all entries having value of w that are contained the state q; then, we normalize across all states in $\mathcal{S}^{(t)}$:

$$\phi\left(q, v, w\right) = \sum_{(k_i, w_i, \cdot) \in \pi^{-1}(q)} \mathbb{1}_{w=w_i} \exp\left(-dist\left(v, k_i\right)\right) \tag{4}$$

$$p_{\text{auto}}\left(w \mid c, \mathcal{S}\right) \propto \sum\nolimits_{q \in \mathcal{S}} \phi\left(q, f\left(c\right), w\right) \tag{5}$$

Finally, we interpolate p_{auto} with p_{LM}.

$$p\left(w \mid c, \mathcal{S}\right) = \lambda p_{\text{auto}}\left(w \middle| c, \mathcal{S}\right) + \left(1 - \lambda\right) p_{LM}\left(w \mid c\right) \tag{6}$$

3. Evaluation

We evaluate RETOMATON in two different settings: (i) standard autoregressive language modeling, where the datastore is constructed from the same training corpus that the base LM was trained on ("in-training datastore"); and (ii) domain adaptation, where the datastore is constructed from a *different* domain than the base LM was trained on.

See all evaluation details in Appendix B.

The main metric that we focus on is perplexity with respect to the *fraction of saved searches* (FoSS). Measuring wall-clock time is difficult to reproduce, is brittle to random temporary hardware and system load, and affected by the specific kNN retrieval library. Contrarily, FoSS does not depend on hardware and engineering factors and is thus completely reproducible. In Appendix I, we empirically analyze FoSS with respect to the saved wall-clock time and we show that they are identical up to an additive constant that depends on the hardware and the specific settings. Thus, FoSS serves as a good proxy to the saved wall-clock time.

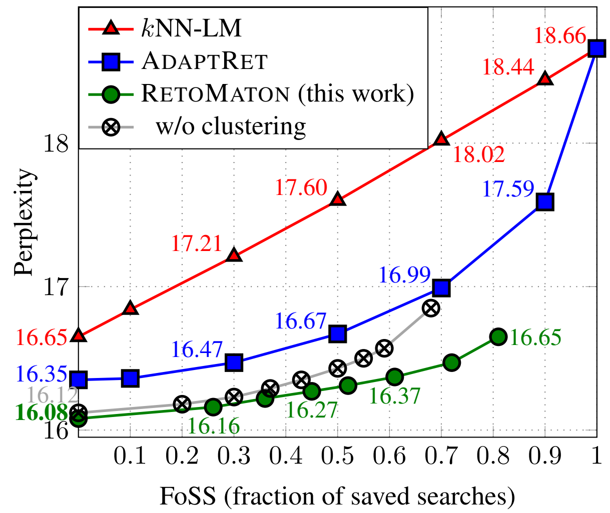

Figure 4: Experiments on WIKITEXT-103, where the datastore is created from the same training set that the base LM was trained on. RETOMATON reduces perplexity across all FoSS values, and even reduces perplexity when FoSS=0.

3.1. In-Domain Datastore

We first experiment with creating the datastore and the automaton from the same data used to train the base LM.

Figure 4 shows how RETOMATON reduces the perplexity on WIKITEXT-103 across different FoSS rates. Specifically, RETOMATON saves 81% of the kNN searches while matching the perplexity of kNN-LM. If we do not perform clustering ("*w/o clustering*"), we can still save more than 60% of the kNN searches while matching kNN-LM. Compared to ADAPTRET, RETOMATON saves 60% of the searches while matching the best perplexity of ADAPTRET.

Surprisingly, even when we do not attempt to save any searches (FoSS=0), RETOMATON reduces perplexity from 16.65 (kNN-LM) and 16.35 (ADAPTRET) to 16.08. The explanation for this is that even when RETOMATON performs kNN search on every step as kNN-LM, it includes the pointers from the previous time step, which are more likely to be correct than the general retrieved nearest neighbors. Some neighbors may be included twice – both as retrieved kNNs, and as pointers from the previous time step; this case is equivalent to increasing the weight of a subset of the retrieved kNNs, which are more likely to be correct.

3.2. Domain Adaptation – shown in Appendix C

Results for domain adaptation are shown in Appendix C.

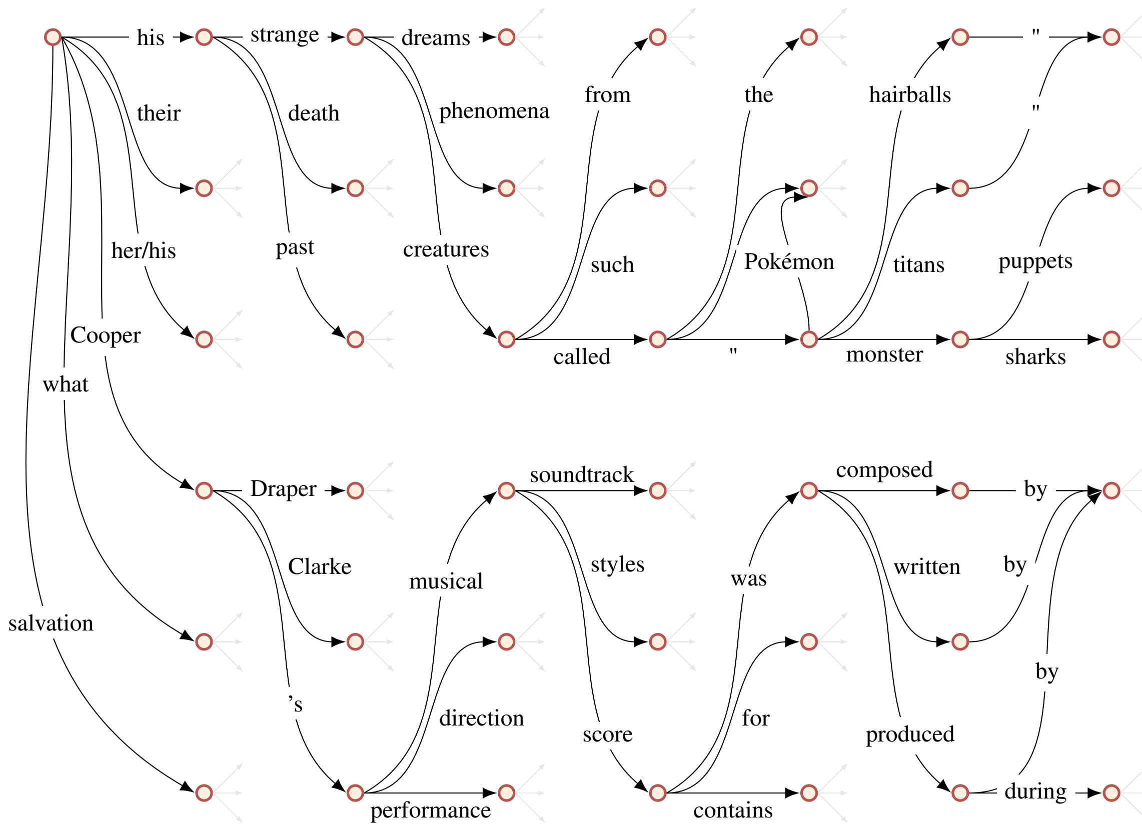

Figure 5: A random sample of the automaton constructed from the training set of WIKITEXT-103

3.3. Improving Fine-Tuning

Table 1: Experiments using a base LM that was fine-tuned on Law-MT. Numbers denote perplexity, and the relative reduction over the fine-tuned LM is shown in parentheses.

Model	FoSS=0		FoSS=0.5	
fine-tuned LM	8.61			
kNN-LM	7.93	(↓7.9%)	8.25	(↓4.2%)
ADAPTRET	7.81	(↓9.2%)	7.91	(↓8.1%)
RETOMATON	**7.10**	**(↓17.5%)**	**7.15**	**(↓17.0%)**

In Appendix C we used RETOMATON to domain-adapt a base LM that was trained on a different domain; however, can RETOMATON improve a *fine-tuned* base LM?

We fine-tuned the base LM of Appendix C on the Law-MT training set, and recreated a datastore and an automaton using the fine-tuned model. As Table 1 shows, while kNN-LM and ADAPTRET reduce the perplexity compared to the fine-tuned model from 8.61 to 7.93 and 7.81, respectively, RETOMATON further reduces perplexity to 7.10, which is a relative reduction of more than 17.5%. This shows how RETOMATON can strengthen even fine-tuned models.

4. Qualitative Analysis

Figure 5 shows a random sample of states and transitions from the automaton construced from the training set of WIKITEXT-103. Further exploration of the automaton can be performed using our publicly available code. Examples and additional analysis are provided in Appendix E.

5. Conclusion

We presented RETOMATON – retrieval automaton. RETOMATON approximates the kNN search of kNN-LM by clustering similar neighbors into automaton states, and keeping pointers from previously found neighbors, which form the transition between states.

Empirically, traversing the automaton at inference time saves up to 83% of the kNN searches, and reduces the perplexity of strong LMs, even after they were fine-tuned.

These results suggest a promising direction for the neuro-symbolic synergy of neural models and symbolic automata. We believe that the principles and the methods presented in this paper are also applicable to other R-LMs, including phrase- and chunk-based retrieval models. To these ends, we make all our code, data, and models publicly available.

References

Aharoni, R. and Goldberg, Y. Unsupervised domain clusters in pretrained language models. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pp. 7747–7763, 2020.

Alon, U., Xu, F. F., He, J., Sengupta, S., Roth, D., and Neubig, G. Neuro-symbolic language modeling with automaton-augmented retrieval. *arXiv preprint arXiv:2201.12431*, 2022.

Baevski, A. and Auli, M. Adaptive input representations for neural language modeling. In *International Conference on Learning Representations*, 2019.

Barrault, L., Bojar, O., Costa-jussà, M. R., Federmann, C., Fishel, M., Graham, Y., Haddow, B., Huck, M., Koehn, P., Malmasi, S., Monz, C., Müller, M., Pal, S., Post, M., and Zampieri, M. Findings of the 2019 conference on machine translation (WMT19). In *Proceedings of the Fourth Conference on Machine Translation (Volume 2: Shared Task Papers, Day 1)*, pp. 1–61, Florence, Italy, August 2019. Association for Computational Linguistics. doi: 10.18653/v1/W19-5301. URL https://aclanthology.org/W19-5301.

Borgeaud, S., Mensch, A., Hoffmann, J., Cai, T., Rutherford, E., Millican, K., Driessche, G. v. d., Lespiau, J.-B., Damoc, B., Clark, A., et al. Improving language models by retrieving from trillions of tokens. *arXiv preprint arXiv:2112.04426*, 2021.

Chen, Q., Wang, H., Li, M., Ren, G., Li, S., Zhu, J., Li, J., Liu, C., Zhang, L., and Wang, J. *SPTAG: A library for fast approximate nearest neighbor search*, 2018. URL https://github.com/Microsoft/SPTAG.

Giles, C. L., Miller, C. B., Chen, D., Chen, H.-H., Sun, G.-Z., and Lee, Y.-C. Learning and extracting finite state automata with second-order recurrent neural networks. *Neural Computation*, 4(3):393–405, 1992.

Grave, E., Cisse, M. M., and Joulin, A. Unbounded cache model for online language modeling with open vocabulary. *Advances in neural information processing systems*, 30, 2017.

Gu, J., Wang, Y., Cho, K., and Li, V. O. Search engine guided neural machine translation. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 32, 2018.

Guo, R., Sun, P., Lindgren, E., Geng, Q., Simcha, D., Chern, F., and Kumar, S. Accelerating large-scale inference with anisotropic vector quantization. In *International Conference on Machine Learning*, pp. 3887–3896. PMLR, 2020.

Guu, K., Hashimoto, T. B., Oren, Y., and Liang, P. Generating sentences by editing prototypes. *Transactions of the Association for Computational Linguistics*, 6:437–450, 2018.

Guu, K., Lee, K., Tung, Z., Pasupat, P., and Chang, M.-W. REALM: Retrieval-augmented language model pretraining. *arXiv preprint arXiv:2002.08909*, 2020.

Hashimoto, T. B., Guu, K., Oren, Y., and Liang, P. A retrieve-and-edit framework for predicting structured outputs. In *Proceedings of the 32nd International Conference on Neural Information Processing Systems*, pp. 10073–10083, 2018.

Hayati, S. A., Olivier, R., Avvaru, P., Yin, P., Tomasic, A., and Neubig, G. Retrieval-based neural code generation. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, pp. 925–930, 2018.

He, J., Berg-Kirkpatrick, T., and Neubig, G. Learning sparse prototypes for text generation. *Advances in Neural Information Processing Systems*, 33, 2020.

He, J., Neubig, G., and Berg-Kirkpatrick, T. Efficient nearest neighbor language models. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, pp. 5703–5714, 2021.

Johnson, J., Douze, M., and Jégou, H. Billion-scale similarity search with gpus. *IEEE Transactions on Big Data*, 2019.

Joulin, A., Cissé, M., Grangier, D., Jégou, H., et al. Efficient softmax approximation for gpus. In *International conference on machine learning*, pp. 1302–1310. PMLR, 2017.

Karpukhin, V., Oguz, B., Min, S., Lewis, P., Wu, L., Edunov, S., Chen, D., and Yih, W.-t. Dense passage retrieval for open-domain question answering. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pp. 6769–6781, 2020.

Khandelwal, U., Levy, O., Jurafsky, D., Zettlemoyer, L., and Lewis, M. Generalization through memorization: Nearest neighbor language models. In *International Conference on Learning Representations*, 2020. URL https://openreview.net/forum?id=HklBjCEKvH.

Khandelwal, U., Fan, A., Jurafsky, D., Zettlemoyer, L., and Lewis, M. Nearest neighbor machine translation. In *International Conference on Learning Representations*, 2021. URL https://openreview.net/forum?id=7wCBOfJ8hJM.

Koehn, P. and Knowles, R. Six challenges for neural machine translation. In *Proceedings of the First Workshop on Neural Machine Translation*, pp. 28–39, 2017.

Lin, C.-C., Zhu, H., Gormley, M. R., and Eisner, J. Neural finite-state transducers: Beyond rational relations. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, pp. 272–283, 2019.

Merity, S., Xiong, C., Bradbury, J., and Socher, R. Pointer sentinel mixture models. In *International Conference on Learning Representations*, 2017.

Neubig, G. and Dyer, C. Generalizing and hybridizing count-based and neural language models. In *Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing*, pp. 1163–1172, 2016.

Ng, N., Yee, K., Baevski, A., Ott, M., Auli, M., and Edunov, S. Facebook fair's wmt19 news translation task submission. In *Proceedings of the Fourth Conference on Machine Translation (Volume 2: Shared Task Papers, Day 1)*, pp. 314–319, 2019.

Omlin, C. W. and Giles, C. L. Extraction of rules from discrete-time recurrent neural networks. *Neural networks*, 9(1):41–52, 1996.

Peng, H., Schwartz, R., Thomson, S., and Smith, N. A. Rational recurrences. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, pp. 1203–1214, 2018.

Ram, O., Shachaf, G., Levy, O., Berant, J., and Globerson, A. Learning to retrieve passages without supervision. *arXiv preprint arXiv:2112.07708*, 2021.

Rastogi, P., Cotterell, R., and Eisner, J. Weighting finite-state transductions with neural context. In *Proceedings of the 2016 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pp. 623–633, 2016.

Rijhwani, S., Rosenblum, D., Anastasopoulos, A., and Neubig, G. Lexically-aware semi-supervised learning for ocr post-correction. *Transactions of the Association for Computational Linguistics*, 9:1285–1302, 2021.

Rubin, O., Herzig, J., and Berant, J. Learning to retrieve prompts for in-context learning. *arXiv preprint arXiv:2112.08633*, 2021.

Schwartz, R., Thomson, S., and Smith, N. A. Bridging cnns, rnns, and weighted finite-state machines. In *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 295–305, 2018.

Sennrich, R., Haddow, B., and Birch, A. Neural machine translation of rare words with subword units. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 1715–1725, 2016.

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, Ł., and Polosukhin, I. Attention is all you need. In *Advances in Neural Information Processing Systems*, pp. 6000–6010, 2017.

Weiss, G., Goldberg, Y., and Yahav, E. Extracting automata from recurrent neural networks using queries and counterexamples. In *International Conference on Machine Learning*, pp. 5247–5256. PMLR, 2018.

Weiss, G., Goldberg, Y., and Yahav, E. Learning deterministic weighted automata with queries and counterexamples. *Advances in Neural Information Processing Systems*, 32: 8560–8571, 2019.

Yogatama, D., de Masson d'Autume, C., and Kong, L. Adaptive semiparametric language models. *Transactions of the Association for Computational Linguistics*, 9:362–373, 2021.

Zhang, J., Utiyama, M., Sumita, E., Neubig, G., and Nakamura, S. Guiding neural machine translation with retrieved translation pieces. In *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long Papers)*, pp. 1325–1335, 2018.

Acknowledgments

We thank Lucio Dery and Vincent Hellendoorn for the helpful discussions and thorough feedback. We are also grateful to the anonymous reviewers for their useful comments and suggestions.

A. Background: the kNN-LM Model

kNN-LM (Khandelwal et al., 2020) is a language model that estimates the probability of the next token by interpolating an already-trained base LM with a kNN distribution. The kNN distribution is provided by searching for the k-nearest neighbors in an external datastore and weighting them according to their negative distance.

Datastore Creation Given a context sequence of tokens $c^{(t)} = \left(w^{(1)}, ..., w^{(t-1)}\right)$, the base LM estimates $p_{LM}\left(w \mid c^{(t)}\right)$, the distribution over the target token w. Let f be the function that maps a context c to a fixed-length vector using an already-trained LM. For example, $f\left(c\right)$ can be the output of a Transformer LM's last self-attention layer. kNN-LM constructs the datastore using a single forward pass over a text collection, which can include the training set that the base LM was trained on, or not. Given the i-th example $\left(c_i, w_i\right)$ in the text collection $\mathcal{D}$, the key-value pair $\left(f\left(c_i\right), w_i\right)$ is defined such that $f\left(c_i\right)$ is the *key*, and w_i is the *value*. The datastore $\left(\mathcal{K}, \mathcal{V}\right)$ is the set of all pairs constructed from the examples in the text collection $\mathcal{D}$:

$$\left(\mathcal{K}, \mathcal{V}\right) = \left\{ \left(f\left(c_i\right), w_i\right) \mid \left(c_i, w_i\right) \in \mathcal{D} \right\} \tag{7}$$

Inference At test time, given a test context c, the model queries the datastore $\left(\mathcal{K}, \mathcal{V}\right)$ to retrieve the k-nearest neighbors $\mathcal{N}$ of $f\left(c\right)$, according to a distance function $dist$ between $f\left(c\right)$ and every $f\left(c_i\right)$ in the datastore. $dist$ is typically the squared ℓ_2 distance. These nearest neighbor pairs form a distribution over the target token w, where the probability of every vocabulary item is computed proportionally to the exponent of its negative distance, while summing over all its occurrences in retrieved values of $\mathcal{N}$:

$$p_{k\text{NN}}\left(w \mid c\right) \propto \sum_{\left(f\left(c_i\right), w_i\right) \in \mathcal{N}} \mathbb{1}_{w=w_i} \exp\left(-dist\left(f\left(c\right), f\left(c_i\right)\right)\right) \tag{8}$$

The kNN-LM's next token probability is then the interpolation of p_{LM} and $p_{k\text{NN}}$, using a scalar hyperparameter λ:

$$p\left(w \mid c\right) = \lambda p_{k\text{NN}}\left(w \mid c\right) + \left(1 - \lambda\right) p_{LM}\left(w \mid c\right) \tag{9}$$

Notice that the k-nearest neighbor search is performed *for every target token*. This introduces severe overhead, since the search is significantly slower than the LM's standard forward pass. In the next section, we show how we can avoid the search in the majority of the time steps.

An additional discussion of related work is provided in Appendix F.

B. Experimental Setup

We evaluate RETOMATON in two different settings: (i) standard autoregressive language modeling, where the datastore is constructed from the same training corpus that the base LM was trained on ("in-training datastore"); and (ii) domain adaptation, where the datastore is constructed from a *different* domain than the base LM was trained on.

Our experiments are fully reproducible and our code is available at `https://github.com/neulab/retomaton`.

Implementation We base our experiments on the original kNN-LM implementation that uses the FAISS (Johnson et al., 2019) library to perform kNN search. We also use FAISS for the one-time k-means clustering.

Hyperparameters We used the same settings as the baseline implementations without any special tuning of our model, and always matched the settings to conduct a fair evaluation. We saved half precision (fp16) datastore keys as He et al. (2021). For WIKITEXT-103, which creates a datastore of 103M entries, we use k-means clustering with k_{clust}=1M. For Law-MT, which creates a datastore of 19M entries, we use k_{clust}=200K. We study the difference between clustering algorithms and the number of clusters in Appendix D. Additional details are provided in Appendix H.

Metrics The main metric that we focus on is perplexity with respect to the *fraction of saved searches* (FoSS). In our preliminary experiments, we found that measuring wall-clock time is difficult to reproduce, is brittle to temporary hardware and system load, and affected by the specific kNN retrieval library. Contrarily, FoSS does not depend on hardware and engineering factors and is thus completely reproducible. In Appendix I, we empirically analyze FoSS with respect to the saved wall-clock time and we show that they are identical up to an additive constant that depends on the hardware and the specific settings. Thus, FoSS serves as a good proxy to the saved wall-clock time.

We control FoSS in RETOMATON by running with different values of the $\tau \in [1, \infty)$ threshold, as detailed in Section 2.3. Higher τ results in frequent restarts and thus lower FoSS.

B.1. In-Domain Datastore

Data Following Khandelwal et al. (2020), we use WIKITEXT-103 (Merity et al., 2017), which is a standard benchmark for autoregressive language modeling, having 103M/250K/250K tokens from Wikipedia in its training/validation/test sets, respectively.

Model We use a Transformer (Vaswani et al., 2017) as our base LM, trained by Khandelwal et al. (2020) following the architecture and settings of Baevski & Auli (2019) with adaptive inputs, adaptive softmax (Joulin et al., 2017), and a large 250K word-level vocabulary. This base LM consists of 16 layers, each with 16 self-attention heads, 1024-dimensional hidden states, 4096-dimensional feedforward layers, amounting to 247M parameters.

B.2. Domain Adaptation

Data Following He et al. (2021), we use the English part of Law-MT, which is an English-German translation dataset for the law domain, released by Koehn & Knowles (2017) and resplit by Aharoni & Goldberg (2020). The training set consists of 19M tokens, which we use to build a datastore and an automaton (Appendix C), or fine-tune the base LM and then build a datastore and automaton (Section 3.3).

Model Following He et al. (2021), as our base LM we use a 12-layer, 1536-dimensional transformer with a vocabulary of 42K subword units (Sennrich et al., 2016), amounting to 656M parameters and trained by Ng et al. (2019), on WMT News Crawl (Barrault et al., 2019).

B.3. Baselines

kNN-LM We compare RETOMATON to kNN-LM using the original code and hyperparameters of Khandelwal et al. (2020). The only difference in hyperparameters is that following He et al. (2021), we use the faster approximate kNN distances provided by FAISS, rather than recomputing them, in our model and in the baselines. We vary the FoSS in kNN-LM by uniformly selecting a certain fraction of time steps in which we skip the kNN search and use only the base LM (p_{LM}), as the "Random" baseline in He et al. (2021).

ADAPTRET We also compare RETOMATON to a retrieval-saving approach that is orthogonal to ours. ADAPTRET is the Adaptive Retrieval approach of He et al. (2021), which trains a light MLP to predict whether to perform a kNN search or predict using only the base LM. The main conceptual difference between RETOMATON and ADAPTRET is that RETOMATON skips kNN searches but still computes the interpolation with the non-parametric distribution p_{auto} for *every* token, using the automaton. In contrast, when ADAPTRET skips a kNN search, it also skips the interpolation with the $p_{k\text{NN}}$ distribution of Equation (9) entirely, and backs-off to rely on the base LM solely. For a detailed discussion of the conceptual differences between RETOMATON and ADAPTRET, see Appendix F.1.

Additional analysis and ablation study is provided in Appendix D.

C. Domain Adaptation

We also experiment with domain adaptation, where the base LM was trained on newspapers, and the models are tested on law documents, using datastore and automaton that were constructed from law data as well, as detailed in Appendix B.2.

Figure 6 shows how RETOMATON reduces the perplexity on Law-MT from 12.34 (kNN-LM) and 12.01 (ADAPTRET) to

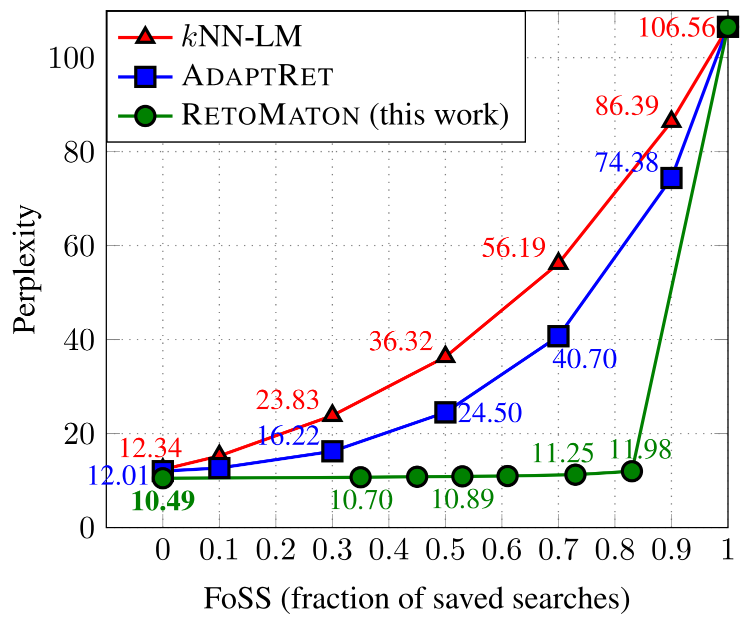

Figure 6: Experiments for domain adaptation, where the datastore is constructed from Law-MT.

10.49 when search is performed every step (FoSS=0). As we increase the fraction of saved searches, RETOMATON shows a very gentle ascent in perplexity, while the perplexity of kNN-LM increases exponentially.

The high perplexity of the ADAPTRET baseline is caused by the high perplexity of the base LM (106.56): in time steps where ADAPTRET does not perform search, its output is identical to the base LM's probability. That is, in domain adaptation, where the base LM performs poorly, interpolating it with the $p_{k\text{NN}}$ distribution (Equation (9)) is *crucial*. Thus, approximating the $p_{k\text{NN}}$ distribution using an automaton is much more effective than pruning it using ADAPTRET, while kNN searches are saved in both cases.

It is also interesting to notice the difference between datasets. In particular, we find that RETOMATON provides a stronger effect in Law-MT, reflected in the very gentle ascent in Figure 6, over its effect in WIKITEXT-103. We believe that one major reason is n-gram repetitiveness between the training and the validation sets. As shown in Figures 13 and 14, there is much higher repetitiveness of n-grams in Law-MT over WIKITEXT-103. For example, 21% of the 5-grams in the validation set of WIKITEXT-103 were seen in the training data; in contrast, in Law-MT – 62% of the 5-grams in the validation set were seen during training.

D. Ablation Study

Pointers vs. clustering The main two contributions in RETOMATON are the use of pointers and the clustering. Here, we tease apart the contribution of each of these.

The *w/o clustering* model is an ablation of RETOMATON, which spares the clustering preprocessing step, and uses only pointers. As shown in Figure 4, even the *w/o clustering* model achieves lower perplexity than the other baselines, with a lowest perplexity of 16.12 at FoSS=0. Up to FoSS=0.4, the base RETOMATON performs only slightly better then *w/o clustering*. Starting from FoSS=0.7, the *w/o clustering* model almost consolidates with ADAPTRET.

From these experiments, we conjecture that RETOMATON's performance in few saved searches (FoSS<0.4) stems mostly from keeping pointers, which provides the most significant boost. Starting from FoSS=0.7, the gap between *w/o clustering* and the base RETOMATON shows the contribution of clustering, which allows longer sequences of consecutive steps without performing kNN search.

Clustering Granularity The only hyperparameter that RETOMATON introduces is the choice of the number of clusters. A higher number of clusters results in smaller, fine-grained clusters. A low number of clusters results in larger, coarse-grained

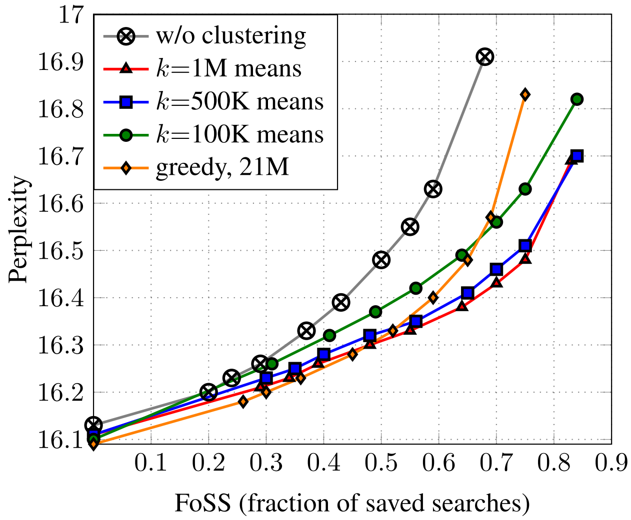

Figure 7: Analysis of the number of clusters on the validation set of WIKITEXT-103. Additional clustering runs can be found in Appendix J (Figure 10).

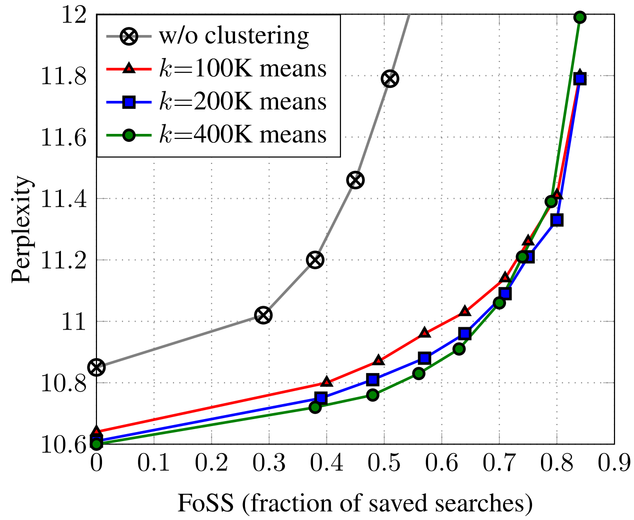

Figure 8: Analysis of the number of clusters on the validation set of Law-MT. A larger version of this figure can be found in Appendix J (Figure 11).

length=3	length=6	length=10
, and the	*the Streets Have No Name "*	*. As a result , there was not a single*
, but the	*Department of Transportation (MDOT)"*	*and some were moved to new locations . Before its*
roughly bounded by	*In the United States ,*	*but it was not until the following day that a*
in the first	*the end of the song ,*	*the end of the Second World War was completed in*
a number of	*not occur until 27 May 1915*	*to lack of evidence ; however , the decision was*
, when the	*fired the shots that caused the*	*the Streets Have No Name ' is more like the*

Table 2: Some of the sequences from the WIKITEXT-103 validation set that our automaton captured without performing kNN search. We selected length=10 sequences that did not appear in the training data.

clusters, where every cluster is possibly more noisy.

Here, we vary the number of clusters and also experiment with the "greedy" clustering algorithms from He et al. (2021). The advantage of the greedy algorithm is that it is computationally cheaper: it requires searching for each datastore entry's nearest neighbors, and then performing a single pass of merging, while k-means requires multiple iterations over the entire datastore.

The results of these experiments are shown in Figure 7 for WIKITEXT-103 and Figure 8 for Law-MT. As shown in Figure 7, k=500K means and k=1M means achieve similar perplexities, while k=100K is too coarse-grained. The *greedy* algorithm presents an interesting tradeoff, as it achieves lower perplexity than the others at FoSS=0, but degrades as FoSS increases, since each cluster is smaller. Figure 8 shows a similar tradeoff in Law-MT: the most fine-grained clustering using k=400K means performs best for FoSS=0, but achieves a higher perplexity than others at FoSS>0.7. Additional clustering runs are shown in Appendix J.

E. Qualitative Analysis

What are the sequences of tokens that RETOMATON captured and predicted consecutively, without kNN search?

Table 2 shows examples of sequences among those that were given the highest probability (p_{auto} in Equation (5)) from the validation set of WIKITEXT-103. Naturally, short sequences (length=3) are usually common 3-grams such as "*, and the*". As length increases (to 6 and 10 tokens), the sequences become more specific; for example, two of them contain part of the name of the song "*Where the Streets Have No Name*" by the band U2. Nevertheless, we selected sequences into the list of length=10 such that *none of them appeared as-is in the training set*, to show that RETOMATON does not only memorize

Training Sequence from: `https://en.wikipedia.org/wiki/Oh,_What_a_Knight!`
The writer of the scenario is unknown , but it was most likely Lloyd Lonergan . He was an experienced newspaperman employed by The New York Evening World while ... A surviving film still gives the possibility of identifying *three of the actors in the film ...*
Validation Sequence from: `https://en.wikipedia.org/wiki/Home_Made_Mince_Pie`
The writer of the scenario is unknown , but it was most likely Lloyd Lonergan . He was an experienced newspaperman employed by The New York Evening World while ... A surviving film still gives the possibility of identifying *eight actors ...*

Figure 9: An example for a 236-token long sequence that RETOMATON was able to capture from the WIKITEXT-103 training set and apply to the validation set. Although most of the paragraph appears as-is in both sets, they have different endings, as these are articles about different silent films from 1910. This shows how RETOMATON allows to dynamically retrieve chunks of text from the training data using a single kNN search, rather than a single token at a time as kNN-LM.

n-grams, but instead, clustering allows it to compose multiple small n-grams into longer n-grams.

Figure 9 shows a 217-token long passage that appears in both training and validation sets, but with different endings. RETOMATON could predict this passage consecutively, without performing search. This shows how RETOMATON can retrieve single tokens from the datastore, as well as adaptively construct much longer chunks.

Figure 12 (Appendix J) shows a histogram of the lengths of these sequences. The vast majority (98%) of the validation set tokens are included in n-grams having n>1.

F. Related Work

F.1. Comparison to ADAPTRET (He et al., 2021)

The closest work to ours is ADAPTRET (He et al., 2021). ADAPTRET saves kNN searches as well, but it suffers from conceptual weaknesses compared to our work:

When the model performs *k*NN search: ADAPTRET uses only the neighbors retrieved by the kNN search. In contrast, RETOMATON uses the set of retrieved nearest neighbors, but also includes the remaining pointers from the previous time step. Apparently, these remaining pointers have a higher likelihood of predicting the correct token than the general set of retrieved nearest neighbors.

When the model does not perform *k*NN search: ADAPTRET skips the interpolation of $p_{k\text{NN}}$ with p_{LM}, and uses p_{LM} solely. In contrast, RETOMATON uses pointers, and still computes the interpolation of p_{auto} with p_{LM}. As an interesting direction for future work, we expect that learning a dynamic interpolation factor λ, similarly to ADAPTRET, will even further improve RETOMATON's results.

Data Efficiency ADAPTRET requires training on a dataset that is disjoint from the corpus that the datastore was built from, to prevent overfitting. Thus, He et al. had to train their MLP *on the validation set*, which is less data-efficient – spending 90% of the original validation set for additional training. In contrast, our approach is completely *unsupervised*, and thus does not require additional data.

F.2. Retrieval and Neuro-Symbolic Methods

Granularity of Retrieval While Khandelwal et al. (2020) and Yogatama et al. (2021) retrieve a token at a time step, other work retrieved a sentence (Hashimoto et al., 2018; Gu et al., 2018; Rubin et al., 2021), a prototype (Guu et al., 2018; He et al., 2020), or a chunk (Guu et al., 2020; Borgeaud et al., 2021). RETOMATON implicitly generalizes these approaches by dynamically constructing the retrieved sequence, essentially being able to retrieve individual tokens as well as constructing search-free longer passages.

Hybrid Models Combining n-grams (Neubig & Dyer, 2016) and automata (Rijhwani et al., 2021) with neural language models has usually led to "static", count-based, transitions weights. In contrast, states in our automaton contain hidden representations of the neural LM, which allows RETOMATON to *dynamically* weigh transitions. Other work scored automata with RNNs (Rastogi et al., 2016; Lin et al., 2019), or constructed RNNs from automata (Schwartz et al., 2018; Peng et al., 2018); RETOMATON differs from these approaches by providing with *retrieved instances* from a datastore, instead of enforcing structure on the neural LM.

Automata Extraction The extraction of automata from neural networks goes back to Giles et al. (1992) and Omlin & Giles (1996), mainly for synthetic and simple regular languages. Later, Weiss et al. (2018; 2019) scaled up the extraction to larger GRU and LSTM architectures. In this work, we do not only extract an automaton, but also combine it to improve the accuracy of neural LMs.

G. Intuition for τ

When traversing the automaton at test-time, the set of states of the next time step is determined according to:

$$\mathcal{S}^{(t+1)} = \begin{cases} \hat{\delta}\left(\mathcal{S}^{(t)}, w^{(t)}\right) & |\hat{\delta}\left(\mathcal{S}^{(t)}, w^{(t)}\right)| \geq \tau \\ \hat{\delta}\left(\mathcal{S}^{(t)}, w^{(t)}\right) \cup \bigcup\limits_{e \in \mathcal{N}^{(t+1)}} \pi\left(e\right) & \text{otherwise} \end{cases}$$

Intuitively, the larger $|\hat{\delta}\left(\mathcal{S}^{(t)}, w^{(t)}\right)|$ is, the more "correct entries" – entries whose values were equal to $w^{(t)}$ that we had at time t, and the more we can rely on their pointers at time $t+1$. Thus, if $|\hat{\delta}\left(\mathcal{S}^{(t)}, w^{(t)}\right)| \geq \tau$, it means that we have more states to continue traversing to, and we can avoid the kNN search. If $|\hat{\delta}\left(\mathcal{S}^{(t)}, w^{(t)}\right)| < \tau$, it means that many of the entries at time t were "incorrect" (their value was not equal to $w^{(t)}$), and the set of new states $\hat{\delta}\left(\mathcal{S}^{(t)}, w^{(t)}\right)$ is small (or even empty); in this case, we perform a new kNN search and restart an automaton traversal.

The minimal value is $\tau = 1$, which means that as long as we have *at least one* automaton state to continue traversing from, we use it without performing kNN search. The maximal value is $\tau = \infty$, which means that $|\hat{\delta}\left(\mathcal{S}^{(t)}, w^{(t)}\right)|$ will always be lower than τ, and thus we will perform kNN search at every step.

H. Evaluation Details

Implementation and Hyperparameters We used the exact hyperparameters of (Khandelwal et al., 2020) including $k_{\text{neigh}} = 1024$ (the number of retrieved nearest neighbors when performing a full search) and the same FAISS (Johnson et al., 2019) kNN library. Following He et al. (2021), we loaded the index to the GPU, and used half precision (fp16) datastore keys.

During the automaton traversal, when reaching automaton states – there might be too many datastore entries in it, which can make the computation slow. We thus always computed Equation (5) over at most $\text{max\_knns} = 1024$ datastore entries (1024 datastore entries overall, in the union of all states $q \in \mathcal{S}^{(t)}$), preferring entries that were directly pointed by pointers from the previous state ($\rho\left(p\right)$), and otherwise choosing randomly. We chose 1024 to match the number of nearest neighbors retrieved when performing a full search $k_{\text{neigh}} = 1024$, although they are not coupled to each other and denote different things.

We used PyTorch Sparse[1] to perform a GPU-based lookup of datastore entries belonging to a cluster.

Hardware We ran all experiments on 32 CPU cores, and RTX 3090 or v100 GPUs. Since our main metric is the fraction of saved searches (FoSS), a different GPU will not change our results. The experiments in Appendix I (Figure 15) were performed on the same machine using a RTX 3090 GPU.

I. Fraction of Saved Searches (FoSS) vs. Wall-clock Saved Time

In our experiments in Section 3, we reported perplexity compared to FoSS (the fraction of saved searches). The other alternative of measuring wall-clock time is difficult to reproduce, is brittle to temporary hardware and system load, and affected by the specific kNN retrieval library such as FAISS as used in Khandelwal et al. (2020), ScaNN (Guo et al., 2020) as used in Borgeaud et al. (2021), or SPTAG (Chen et al., 2018), etc. Further, it depends on factors that are orthogonal to our contribution, such as whether the RAM is large enough to store the datastore, and the random-access reading latency of the hard-drive.

FoSS, in contrast, does not depend on hardware, engineering factors, or temporary system load. FoSS is also completely reproducible while wall-clock time is not. Thus, FoSS serves as a good proxy to wall-clock time that enables reproducible

[1] http://github.com/rusty1s/pytorch_sparse

experiments. Here, we perform an empirical analysis of FoSS with respect to the saved wall-clock time.

Figure 15 shows a comparison between the fraction of saved wall-clock time and FoSS. As shown, the saved wall-clock time and FoSS are identical up to an additive constant that depends on the hardware and the specific settings. Searching for k-nearest neighbors using a CPU FAISS index results in a curve that is very close to the optimal $y = x$, meaning that almost the entire reduction in searches is translated directly into saving of wall-clock time. Using a GPU index without clustering (only pointers) results in a penalty of 17%, but the curve is almost parallel to the optimal $y = x$. Using a GPU index with clustering results in a penalty of 24%, and begins to be beneficial in terms of wall-clock time starting from FoSS=0.32.

The experiments in Figure 15 were performed in the setting that we load the datastore to memory, to prevent the hard drive's latency from being the bottleneck. Another option is to approximate the key vectors using the FAISS index, but currently FAISS's `reconstruct` API is implemented only for a CPU index (rather than a GPU index),[2] and for a single key ID at a time,[3] and thus does not support batching.

We expect that as the datastore size increases to the scales of Borgeaud et al. (2021), and as the number of neighbors retrieved increases (k_{neigh}) – the more pressure that will be put on the kNN search, the more of a bottleneck that it will become, and the larger relative benefit that saving kNN searches will provide to wall-clock time.

J. Additional Results

Comparison of Datasets Figure 13 and Figure 14 show the overlap of n-grams between the training and validation set of WIKITEXT-103 and Law-MT. As shown, for all values of n, more n-grams from the validation set were seen in the training set in Law-MT compared to WIKITEXT-103. We see this as the explanation for the better scaling of RETOMATON on Law-MT, where the perplexity only gently increases as we increase FoSS, compared WIKITEXT-103.

Ablation Study Figure 10 and Figure 11 show results for different clustering algorithms and granularities, on WIKITEXT-103 and Law-MT, respectively. These figures are similar to Figure 7 and Figure 8, except that we include here more runs of k-means with more values of k_{clust} and more runs of the greedy clustering of He et al. (2021).

[2] `https://github.com/facebookresearch/faiss/issues/314`
[3] `https://github.com/facebookresearch/faiss/issues/1163`

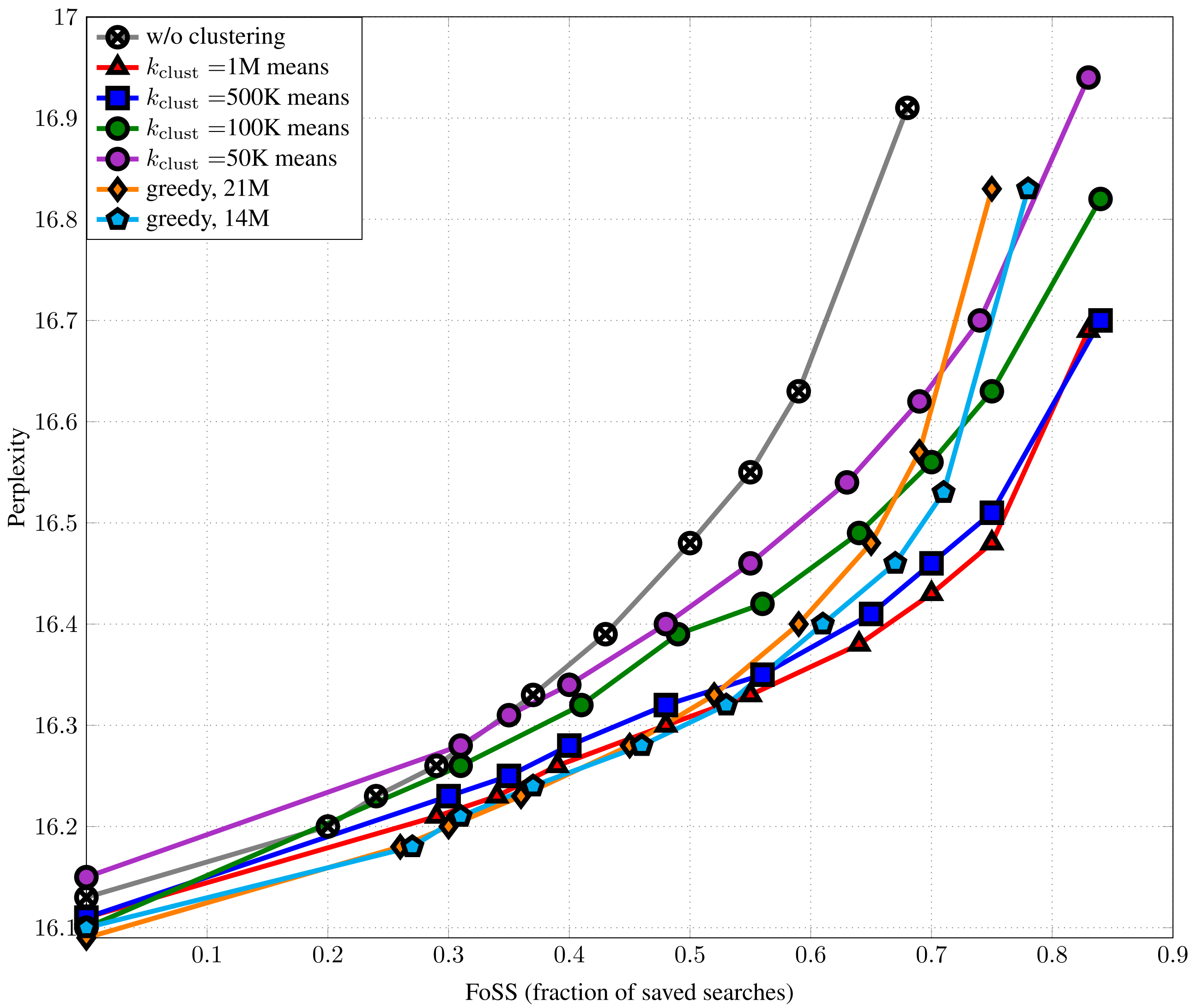

Figure 10: Analysis of the number of clusters on the validation set of WIKITEXT-103.

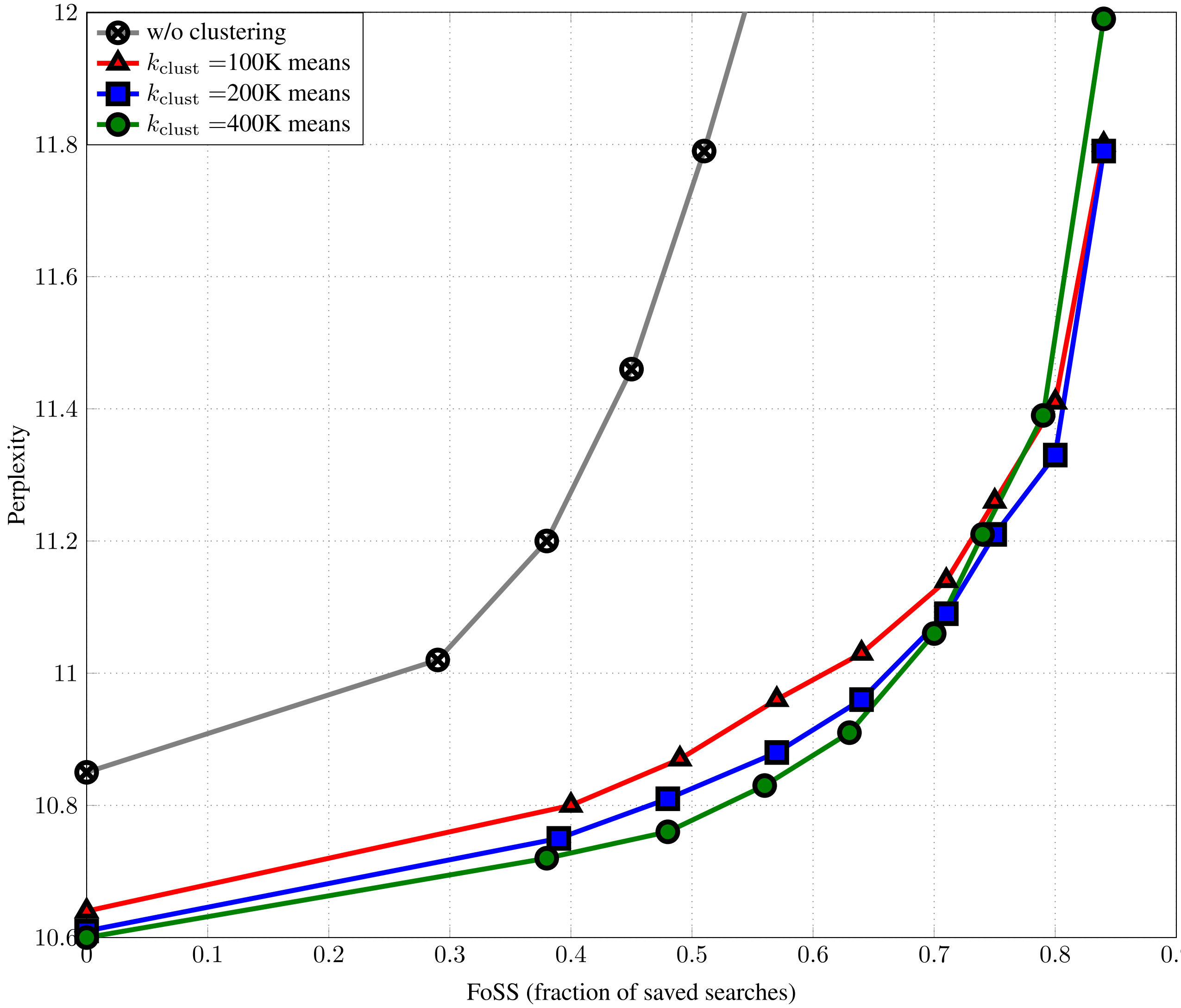

Figure 11: Analysis of the number of clusters on the validation set of Law-MT.

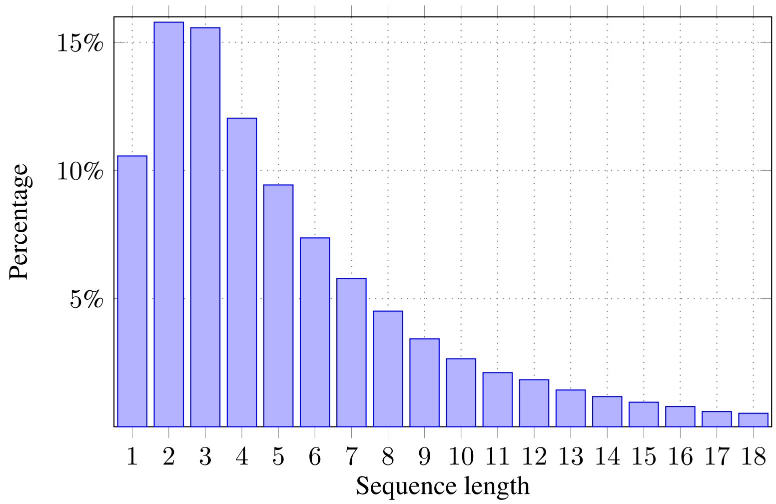

Figure 12: Histogram of the lengths of sequences that were predicted consecutively, without kNN search, in WIKITEXT-103.

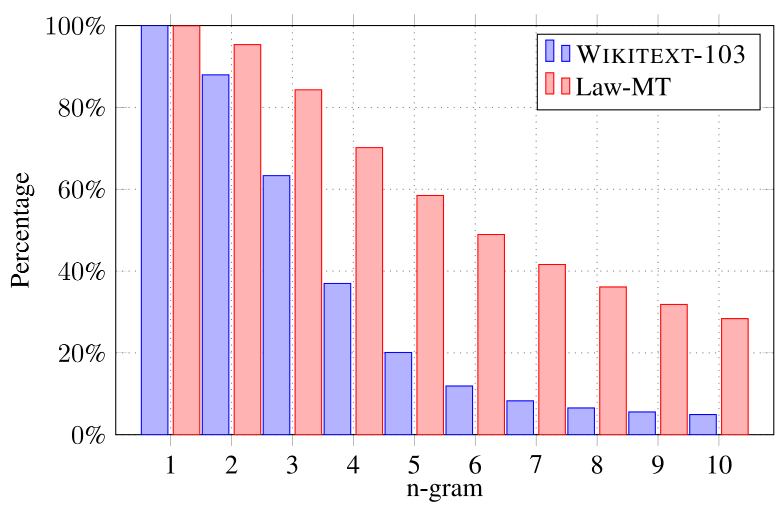

Figure 13: The fraction of *n-gram types* in the validation set that appeared verbatim in the training set in each dataset, for different values of n.

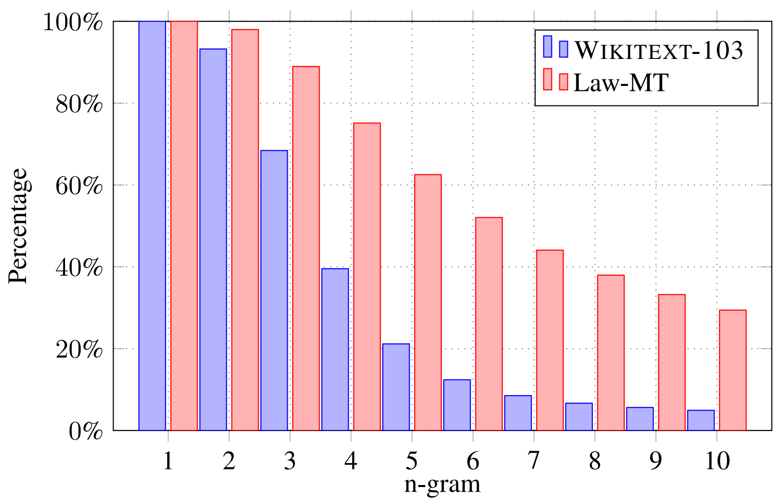

Figure 14: The fraction of *n-gram occurrences* in the validation set that appeared verbatim in the training set in each dataset, for different values of n.

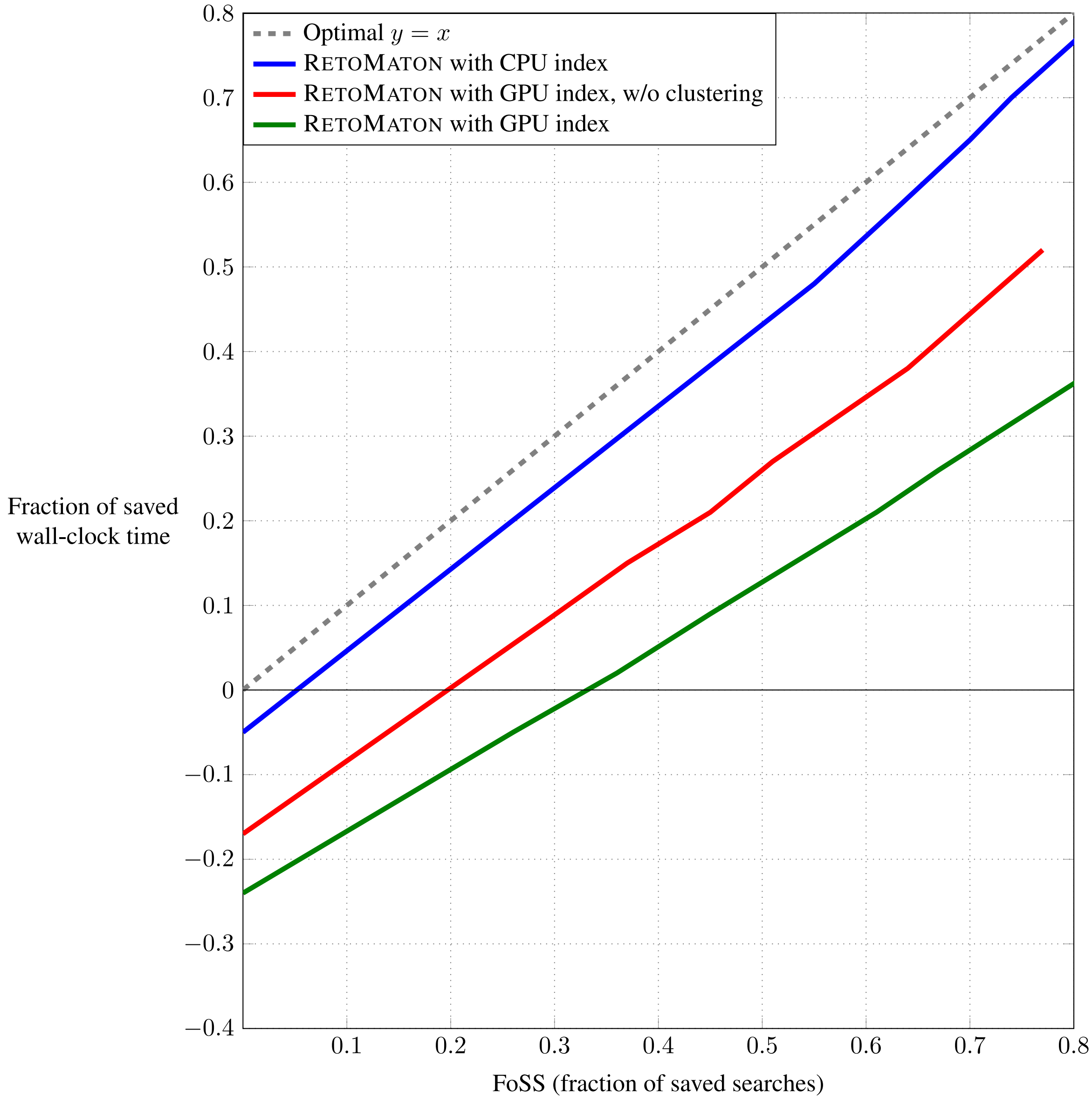

Figure 15: A comparison between the fraction of saved wall-clock time vs. FoSS, the fraction of saved searches. The fraction of saved wall-clock time was computed relatively to the baseline kNN-LM.