Efficient Nearest Neighbor Language Models

Junxian He[†], **Graham Neubig**[†], **Taylor Berg-Kirkpatrick**[‡]
[†]Language Technologies Institute, Carnegie Mellon University
[‡]Department of Computer Science and Engineering, University of California San Diego
{junxianh, gneubig}@cs.cmu.edu, tberg@eng.ucsd.edu

Abstract

Non-parametric neural language models (NLMs) learn predictive distributions of text utilizing an external datastore, which allows them to learn through explicitly memorizing the training datapoints. While effective, these models often require retrieval from a large datastore at test time, significantly increasing the inference overhead and thus limiting the deployment of non-parametric NLMs in practical applications. In this paper, we take the recently proposed k-nearest neighbors language model (Khandelwal et al., 2019) as an example, exploring methods to improve its efficiency along various dimensions. Experiments on the standard WikiText-103 benchmark and domain-adaptation datasets show that our methods are able to achieve up to a 6x speed-up in inference speed while retaining comparable performance. The empirical analysis we present may provide guidelines for future research seeking to develop or deploy more efficient non-parametric NLMs.[1]

1 Introduction

Language models (LMs) are one of the most fundamental technologies in NLP, with applications spanning text generation (Bahdanau et al., 2015; Rush et al., 2015), representation learning (Peters et al., 2018; Devlin et al., 2019; Yang et al., 2019), and few-shot learning (Radford et al., 2019; Brown et al., 2020). Modern neural language models (NLMs) based on recurrent (Mikolov et al., 2010; Sundermeyer et al., 2012) or self-attentional (Vaswani et al., 2017; Al-Rfou et al., 2019) neural networks are mostly *parametric*, where the predictions are solely dependent on the model parameters given the input data.

In contrast, recent *non-parametric* LMs (Guu et al., 2018; Khandelwal et al., 2019; He et al., 2020) model text distributions by referencing both the parameters of the underlying model *and* examples from an external datastore. Non-parametric LMs are appealing since they allow for effective language modeling – particularly for rarer patterns – through explicit memorization via a datastore, which mitigates the burden on model parameters to learn to encode all information from a large dataset. One effective and representative example is the k-nearest neighbors LM (kNN-LM, Khandelwal et al. (2019)). The kNN-LM computes the probability of the next token by interpolating a parametric LM with a distribution calculated from the k nearest context-token pairs in the datastore, as demonstrated in Figure 2. This model is particularly notable for its large improvements in performance – it outperforms the previous best parametric LMs by a large margin in standard language modeling benchmarks, in domain adaptation settings, and on other conditional generation tasks such as machine translation (Khandelwal et al., 2020).

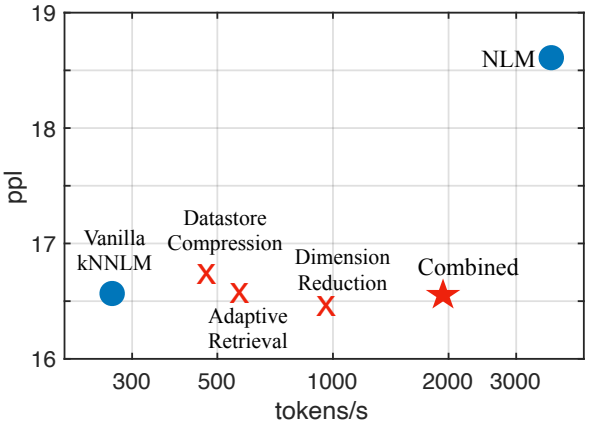

Figure 1: Perplexity (ppl) and evaluation speed for different models. Circled points represent the neural language model (NLM) and k-nearest neighbors language model (kNN-LM) baselines respectively, while others are the methods that we propose and explore in this paper.

However, one downside to the kNN-LM is that the datastore stores high-dimensional dense vectors

[1]Code is available at https://github.com/jxhe/efficient-knnlm.

Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing, pages 5703–5714
November 7–11, 2021.

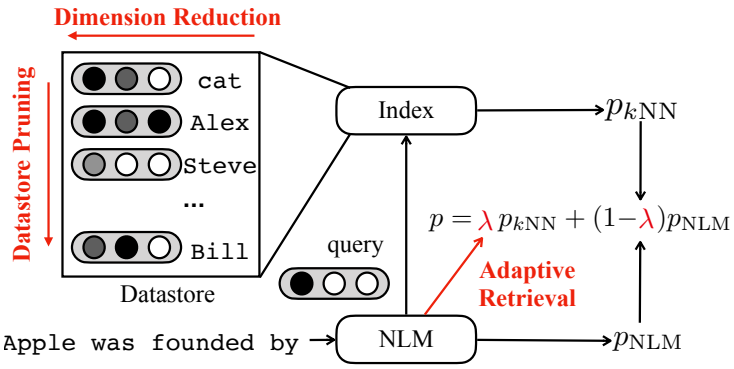

Figure 2: Illustration of kNN-LM . The datastore consists of paired context representations and the corresponding next tokens. The index represents a method that performs approximate k-nearest neighbors search over the datastore. Red and bolded text represent three dimensions that we explore in this paper to improve the efficiency. In adaptive retrieval, we use a predictive model to decide whether or not to query the datastore.

for *each token in the training data*; this can easily scale to hundreds of millions or even billions of records. As a result, the extra retrieval step from such datastores greatly decreases model efficiency at test time. For example, a 100M-entry datastore can lead to an over 10x slow-down compared to parametric models (§3.3) as shown in Figure 1. This issue poses a serious hurdle for the practical deployment of non-parametric LMs, despite their effectiveness.

In this paper, we attempt to address this issue of test-time inefficiency and make non-parametric LMs more applicable in real-world settings. We take kNN-LM as an example, first analyzing the evaluation overhead, and raise three questions that we aim to answer in this paper: (1) Do we really need to perform retrieval on the prediction of every single token? (2) Can we identify and prune redundant records from the datastore? (3) Is it possible to further compress the datastore by reducing the vector dimensionality without losing performance? We propose and explore potential solutions for each question to aid efficiency. Specifically, we (1) show that a lightweight network can be learned to automatically prune unnecessary retrieval operations (adaptive retrieval, §4.1), (2) explore several different methods for datastore pruning based on clustering, importance-guided filtering, or greedy merging (§4.2), and (3) empirically demonstrate that simple dimension reduction techniques are able to improve *both* the performance and speed (§4.3). Figure 1 illustrate the overall performance of these methods. Our experiments on the WikiText-103 language modeling benchmark (Merity et al., 2017) and a training-free domain-adaptation setting demonstrate speed improvements of up to 6x with comparable perplexity to the kNN-LM. On a higher level, we expect the empirical results and analysis in the paper to help researchers better understand the speed-performance tradeoff in non-parametric NLMs, and provide a springboard for future research on more efficient non-parametric LMs.

2 k-Nearest Neighbors Language Model

In this section, we overview kNN-LM (Khandelwal et al., 2019) and its implementation details.

Formulation. kNN-LM (Khandelwal et al., 2019) is an LM that estimates token distributions by interpolating a pre-trained autoregressive NLM's distribution with another distribution computed using an external datastore. Specifically, given a sequence of context tokens $c_t = (w_1, \cdots, w_{t-1})$, the kNN-LM's next token probability $p(w_t|c_t)$ is calculated through the interpolation of the probability estimated by a standard parametric NLM $p_{\text{NLM}}(w_t|c_t)$ and a probability computed using an external datastore $p_{kNN}(w_t|c_t)$ (detailed later):[2]

$$\begin{aligned} &p(w_t|c_t) \\ &= \lambda p_{\text{kNN}}(w_t|c_t) + (1-\lambda) p_{\text{NLM}}(w_t|c_t), \end{aligned} \quad (1)$$

where λ is the interpolation hyperparameter. Note that the application of kNN-LM requires no additional training; the parameters of the NLM remain as-is, and Eq. 1 is only applied at test time. The workflow of kNN-LM is shown in Figure 2

Datastore. The datastore in kNN-LM stores context vectors from the pretrained NLM as keys, and their corresponding next tokens as values. Formally, let f be the key function that maps context sequence c to a fixed-size vector, then the datastore $(\mathcal{K}, \mathcal{V})$ contains all the key-value pairs constructed from the entire training examples $\mathcal{D}$:

$$(\mathcal{K}, \mathcal{V}) = \{(f(c_t), w_t) | (c_t, w_t) \in \mathcal{D}\}. \quad (2)$$

The size of such a datastore is almost equal to the number of training tokens because the context c_t is (nearly) unique due to the large context

[2]Below, we sometimes ignore the subscript to simplify notation when there is no confusion.

window size in modern recurrent (Sundermeyer et al., 2012) or self-attentional (Vaswani et al., 2017) NLMs. This suggests that the datastore can easily scale to hundreds of millions or even billions of records. Also, each $f(c_t)$ is a high-dimensional dense vector, which makes the datastore difficult to fit in memory. For example, a datastore from a 100M-token training dataset, using 1024-dimension context vectors at 16-bit precision, could require 200GB of memory.[3]

The Nearest Neighbor Distribution $p_{\text{kNN}}(w_t|c_t)$. At inference time, the kNN-LM (1) computes the context vector $f(c)$ from the current sequence using the pretrained NLM, (2) uses $f(c)$ as the query to retrieve k nearest neighbors $\mathcal{N} = \{(q_i, v_i)|i = 1, \cdots, k\}$ from the datastore, and (3) aggregates the retrieved tokens to form the distribution $p_{\text{kNN}}(w|c)$ to be used in Eq. 1 as:

$$p_{\text{kNN}}(w = y|c) \propto \sum_{(q_i,v_i)\in\mathcal{N}} \mathbb{I}_{v_i=y} \exp(-d(q_i, f(c))). \quad (3)$$

$d(\cdot, \cdot)$ is a distance function between the two vectors, and L^2 was shown to be more effective than other alternatives (Khandelwal et al., 2019). Intuitively, kNN-LM finds context sequences in the datastore that are similar to the test context, and then utilizes the next tokens observed after these contexts to help prediction. Such a mechanism allows language modeling through explicit memorization from the datastore, and may be particularly helpful for patterns rarely seen by the pretrained NLM (Khandelwal et al., 2019, 2020).

Sources of Inference Overhead. The extra inference overhead stems from the kNN search process in $p_{\text{kNN}}(w_t|c_t)$ computation. We denote the inference time per token as $t = t_{\text{NLM}} + t_{k\text{NN}}$. While t_{NLM} remains constant with different datasets, $t_{k\text{NN}}$ unfortunately grows as the datastore scales.

In practice, the kNN search process is often performed only approximately (ANN, Gionis et al. (1999); Muja and Lowe (2009)) to reduce computational cost. Khandelwal et al. (2019) implemented ANN search in kNN-LM[4] using FAISS (Johnson et al., 2019), which combines inverted file systems (Sivic and Zisserman, 2003) and product vector quantization (Jegou et al., 2010). This type of index reduces memory usage by only storing quantized vectors and accelerates kNN search by pre-clustering the datastore vectors; interested readers can refer to (Jegou et al., 2010) for more details. For the purpose of this paper we study kNN-LM using this indexing method as a black box, aiming to improve efficiency in an index-agnostic way. At the same time, we note that building fast and accurate indexing methods remains an active area of research (André et al., 2019; Guo et al., 2020), and selection or improvement of the index itself (possibly in concert with the methods proposed in this paper) is an interesting avenue for future work.

Distance Recomputation. The distances to the nearest neighbors are required to compute $p_{\text{kNN}}(w_t|c_t)$ as shown in Eq. 3. However, as described above, kNN-LM's nearest neighbor search process performs search over quantized vectors, and as a result it can only return *approximate* distances. While it is possible to compute the accurate distances by reading the full-precision vectors from the datastore after retrieval, this presents challenges as well: (1) storing the entire datastore in memory is not scalable for large datastores, (2) reading the vectors from a large datastore on disk on-the-fly is too slow to be practical (< 1 token per second).[5] Therefore, in this paper we use the approximate distances directly to compute p_{kNN}. This comes at the cost of a minor performance loss, as we will show in §3.3. Similar approximations were adopted to apply kNN-LM to machine translation tasks (Khandelwal et al., 2020).

3 The Efficiency of kNN-LM

In this section, we first introduce the datasets and setup that we will use throughout the paper, and then compare the inference speed of kNN-LM to parametric NLMs.

3.1 Datasets

We study kNN-LM in two different settings: (1) the standard setting where the datastore is constructed from the same data used to train the NLM, and (2) a domain adaptation setting where the datastore is based on the training data in the test domain, in

[3] Note that the dataset to contruct the datastore may not necessarily be the training data that trains the parametric NLM in Eq. 1 – a separate dataset may be used for the datastore construction which would lead to potential applications such as training-free domain adaptation or a gradient-free way to utilize extra training data (Khandelwal et al., 2019).

[4] https://github.com/urvashik/knnlm.

[5] Disk random I/O is another aspect that may be improved by further engineering effort, which is also interesting future work.

which case the NLM never sees the examples included in the datastore. The following two datasets are used for the two settings respectively:

WikiText-103 (Merity et al., 2017) is a standard language modeling benchmark from Wikipedia that has 250K word-level vocabulary. It consists of 103M training tokens, and thus leads to a datastore that has 103M records and takes 200G space. Following (Khandelwal et al., 2019), we use the transformer-based (Vaswani et al., 2017) language model checkpoint released by (Baevski and Auli, 2019) as the underlying pretrained NLM, which is trained on the WikiText-103 training split.

Law-MT is an English-German machine translation dataset in the law domain originally released by (Koehn and Knowles, 2017) and resplit by (Aharoni and Goldberg, 2020). We only use the English text for language modeling. The training set consists of 19M tokens which we use to build the datastore that occupies 55G space. To inspect the domain-adaptation performance, our pretrained NLM is a 12-layer transformer model trained on WMT News Crawl[6] released by (Ng et al., 2019).

3.2 Setup

Throughout the rest of the paper, we adopt the same hyperparameters and index as (Khandelwal et al., 2019) for kNN-LM.[7] Specifically, the number of nearest neighbors is set to 1024 during evaluation.[8] Our pretrained NLMs are the state-of-the-art decoder-only transformers as mentioned above, and the key function $f(c)$ to obtain context vectors is the input to the final layer's feedforward network. The context vectors are 1024-dimensional and 1536-dimensional for WikiText-103 and Law-MT respectively. Given a dataset, we tune the interpolation weight λ on validation set in terms of the vanilla kNN-LM performance, and fix it unless otherwise specified. Complete details on the setup can be found in Appendix A.

Evaluation efficiency is benchmarked on 32 CPU cores (1.5 GHz AMD EPYC 7282) and 1 NVIDIA RTX 3090 GPU which represents a normalized environment – the index searching uses all the CPU cores while neural network computation is based on the GPU. Running retrieval on 32 CPU cores is also used by the FAISS repo[9] as a standard setting to benchmark large-scale retrieval.

Table 1: Evaluation performance and speed of baseline models. The ppl numbers of kNN-LM * (exact) are from (Khandelwal et al., 2019), which recomputes accurate distances.

Model	WikiText-103		Law-MT	
	ppl	tokens/s	ppl	tokens/s
kNN-LM * (exact)	16.12	<1	–	–
NLM	18.66	3847	106.25	28K
kNN-LM	16.65	277	12.32	1052

3.3 Baseline Speed

We measure the perplexity (ppl) and speed of evaluation in term of tested tokens per second, and Table 1 reports the results on the test set of the two datasets. We also include "kNN-LM (exact)" for reference, which represents the kNN-LM variant that re-computes accurate distances as explained in §2. While very effective with 2 ppl points gains on WikiText-103 and over 90 points gains on Law-MT in a domain-adaptation setting, kNN-LM is 10x – 30x slower to evaluate on these datasets because of the extra retrieval step. When exact distances are computed by reading vectors from the disk on-the-fly, kNN-LM (exact) takes over 1 second to evaluate a single token.

4 The Remedies

In this section we propose and explore several different methods that may potentially improve the efficiency of kNN-LM along three axes: (1) adaptive retrieval, (2) datastore pruning, and (3) dimension reduction. We analyze the performance of each method on WikiText-103, trying to conclude the best practices that we will evaluate in §5.

4.1 Adaptive Retrieval

Just as humans refer to books only when they are uncertain in an open-book quiz, the parametric NLMs may not *always* need help from the external datastore. To inspect this hypothesis, we compare $p_{\text{kNN}}(w|c)$ and $p_{\text{NLM}}(w|c)$ for every token in the WikiText-103 validate set. Interestingly, $p_{\text{kNN}}(w|c) \geq p_{\text{NLM}}(w|c)$ only 39% of the time – the likelihood of 61% of the tokens becomes

[6] http://data.statmt.org/news-crawl/

[7] We directly base our experiments on the original kNN-LM implementation.

[8] The perplexity continues improving as k grows as shown in (Khandelwal et al., 2019) and confirmed by us. Yet k does not have an effect on the evaluation speed in the range [8, 1024] from our observation.

[9] https://github.com/facebookresearch/faiss/wiki/Indexing-1G-vectors

Table 2: The features used to train the retrieval adaptor.

Feature	Description
$f(c)$	the context embeddings from pretrained NLM
conf(c)	the maximal value (confidence) of p_{NLM}
ent(c)	the entropy of the distribution p_{NLM}
log freq$(c[-n:])$	log of frequency of the immediate n context tokens computed from the training data. $n = 1, 2, 3, 4$ which leads to four scalar features.
log fert$(c[-n:])$	fert$(c[-n:])$ is the number of unique word (fertility) that succeeds the immediate n context tokens computed from the training data. $n = 1, 2, 3, 4$ which leads to four scalar features.

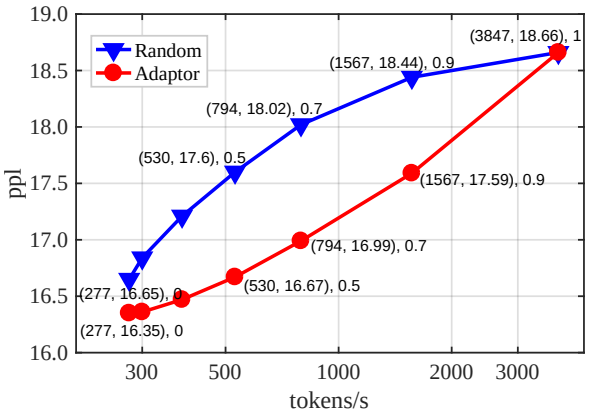

Figure 3: Perplexity and speed results of adaptive retrieval on WikiText-103 test set. We annotate the coordinates of some points and the third number in the annotation is the fraction of retrieval operations that are removed.

worse after interpolation despite the overall improvement. This indicates that if we were able to identify these locations perfectly, 61% of the retrieval operations could be removed completely and we would achieve even better perplexity. Inspired by this observation, we aim to automatically identify and prune unnecessary retrieval operations to speed up inference.

Methods: We propose to train a light neural network, the retrieval adaptor, to identify when we should remove the retrieval operation. Specifically, given the context c as the input, the retrieval adaptor may be trained with either (1) a classification objective to predict whether $p_{\text{kNN}}(w|c) \geq p_{\text{NLM}}(w|c)$, or (2) a likelihood maximization objective to predict the interpolation weight $\lambda(c)$ and maximize the overall likelihood of kNN-LM as in Eq. 1. In our preliminary results the classification method performs only on par with a random removal baseline, partially due to the discretized noisy supervision. Therefore, we directly maximize the kNN-LM log likelihood by modeling λ as a function of the context:

$$\mathcal{L} = \frac{1}{T} \sum_t \left[\log p(w_t|c_t; \lambda_\theta(c_t)) - a \cdot \lambda_\theta(c_t) \right], \tag{4}$$

where only θ – the parameters of the retrieval adaptor – are updated. The second term is an L^1 regularizer that encourages learning sparse weights for $p_{k\text{NN}}$, which we find helpful to prune unnecessary retrievals. At inference time, we prune a given fraction of retrievals with the smallest kNN weight $\lambda(c)$ by resetting $\lambda(c)$ to zero. The hyperparameters of the retrieval adaptor network including the regularizer coefficient, a, are tuned on the validation set in terms of perplexity at 50% retrieval pruning. Learning the interpolation weights to prune is related to (Johansen and Socher, 2017) where they learn to skip text for classification tasks. Optimizing the interpolation weights in kNN-LM has also been applied at training time to train the NLM jointly (Yogatama et al., 2021).

Architecture and Input Features: The retrieval adaptor is a light MLP network with linear transformation followed by ReLU activation at each layer. The output layer maps the hidden representation to a 2-dimensional vector followed by a `LogSoftmax` layer to yield $\log(\lambda)$ and $\log(1-\lambda)$ respectively. Complete details on the retrieval adaptor can be found in Appendix A.2. We concatenate several neural and count-based features as input to the retrieval adaptor as shown in Table 2. For the scalar features (basically all the features excluding $f(c)$) , we found it helpful to map them to a vector with a small network before concatenation. We note that all the features are trivial to obtain at test time – the neural features are from intermediate computation of $p_{\text{NLM}}(w|c)$ and count-based features are looked-up values. Ablation analysis on these features can be found in Appendix B.

Training: During training, only the retrieval adaptor is updated while the pretrained NLM is fixed. Note that it is inappropriate to train the retrieval adaptor on the training dataset, which would lead to biased solutions since p_{NLM} may have already overfit on the training data and the datastore includes the training example itself. To generalize to the test data, we hold out 10% of the validation data for validation and use the remaining 90% to train the retrieval adaptor. The retrieval adaptor is light and converges quickly; it took several minutes to train it on WikiText-103 with a single GPU.

Results: Figure 3 shows the perplexity and evaluation speed of adaptive retrieval on the test set of WikiText-103, varying the percent of removed retrieval operations. The different threshold values of λ used to cut off retrieval is selected based on the synthetic validation set mentioned above. We also add a random retrieval baseline which uniformly selects a certain fraction of retrieval operations to discard. We observe that adaptive retrieval (AR) exhibits a much flatter increase of perplexity than the random baseline when the number of removed retrievals grows. Notably, AR is able to achieve comparable perplexity to the original kNN-LM model (16.67 vs. 16.65) while being nearly 2x faster (530 vs. 277 tokens/s) through removing 50% of the operations. AR's gain comes from both the smart pruning mask and optimized λ. We perform an ablation study on this in Appendix B.

4.2 Datastore Pruning

The information present in a large training dataset is often redundant, which suggests that a datastore constructed from training tokens may be pruned with no or only minor performance cost. To validate this hypothesis, we propose several different methods to prune the number of entries and reduce the datastore size:

Random Pruning: As a simple baseline, a certain fraction of the datastore entries are randomly selected. Random pruning has been shown to work well with a billion-scale datastore in machine translation tasks (Khandelwal et al., 2020).

k-Means Pruning: Clustering is a common technique to prune redundant vectors by only keeping the centroids of the clusters. Yet in our task specifically, we note that a general clustering on the context vectors is not directly applicable since the vectors in the same cluster may still correspond to various target tokens, as language use in context is not deterministic. Therefore, we propose to perform target-aware k-means clustering – for a word w_i in the vocabulary, we perform a separate k-means clustering for all the context vectors that have w_i as the target token, then we only keep centroids of each cluster as well as saving the cluster size s. The `(centroid vector, cluster size, target token)` triples form a new compressed datastore. Since we approximate multiple vectors in the same cluster with the centroid and only save the centroid vector once in the new datastore, the computation of the kNN distribution $p_{k\text{NN}}$ needs to be rectified as:

$$p_{\text{kNN}}(w = y|c) \propto \sum_{(q_i,v_i)\in\mathcal{N}} \mathbb{I}_{v_i=y}\, s_i \cdot \exp(-d(q_i, f(c))), \quad (5)$$

the cluster size s_i acts like weights for each datastore entry. Eq. 5 recovers Eq. 1 when every cluster is of size 1.[10] In practice, we perform 5000 separate k-means clustering passes only for the most frequent 5000 words due to high computational cost, which accounts for 84% of all the training tokens. For other vectors we treat each of them as a separate clusters with size 1. The number of clusters in k-means are set to 1/20 of the number of vectors to be clustered, which produces a 5x smaller datastore overall. We did not intensively tune the k-means hyperparameters due to the computational burden. We note that the clustering here is different from the pre-clustering in the ANN index with inverted file systems mentioned in §2– the index's pre-clustering does not actually reduce size and is just for lookup.

Algorithm 1 Greedy Merging

$(\mathcal{K}, \mathcal{V}) = \{(q_i, v_i)\}_{i=1}^{N} \leftarrow$ the old datastore
$s \leftarrow \mathbf{1}$ ▷ weight vector with size N
for $(q_i, v_i) \in (\mathcal{K}, \mathcal{V})$ **do**
  retrieve K neighbors of (q_i, v_i) as $\{q_{t_k}, v_{t_k}\}_{k=1}^{K}$
  for $k = 1, 2, \cdots, K$ **do**
    if $s_{t_k} = 1$ & $v_i = v_{t_k}$ & $t_k \neq i$ **then** ▷ merge condition
      $s_i \leftarrow s_i + 1$ ▷ merge (q_{t_k}, v_{t_k}) into (q_i, v_i)
      $s_{t_k} \leftarrow s_{t_k} - 1$ ▷ Remove record t_k
    end if
  end for
end for
Save datastore $\{(q_i, v_i, s_i) | s_i > 0, i = 1, \cdots, N\}$

Greedy Merging: Generally we aim to merge records that share the same target token while being close to each other in vector space. Token-aware clustering is an attempt to achieve this goal, but forcing all points to participate in clustering – and the resulting large clusters – causes some points within the same cluster to be distant in some clusters with high variance. Thus approximating all the vectors with the cluster centroids may lead to large errors. To address this issue, we propose a simple approach, greedy merging (GM), which inspects every record in the datastore and greedily

[10] In addition, the centroid formulation is roughly equivalent to saving vectors within the same cluster as the centroids multiple times without pruning in the original formulation.

merges their nearest neighbors if a merging condition is satisfied. The detailed algorithm is shown in Algorithm 1. Intuitively, GM is density-based to group points with nearest neighbors, but the merging operation only happens *locally* between a point and its nearest neighbors – it never propagates to merge the nearest neighbors of nearest neighbors unlike typical density-based clustering methods (Ester et al., 1996) which may amplify errors. Similar to k-means pruning, we also compute the weights s_i of each entry in the compressed datastore to correct $p_{k\text{NN}}$ computation using Eq. 5. Without a global clustering mechanism, this approach ensures that the merging vectors are close enough by inspecting only a small number of nearest neighbors. In the following analysis we vary the number of nearest neighbors K within range [2,100] to achieve different compression rates.

Rank-based Pruning: It is well known that embedding spaces contain "hubs" which are nearest neighbors of many other embeddings (Tomasev et al., 2013), and other points that are not nearest neighbors of any other points. We hypothesize that these entries which are rarely nearest neighbors may be removed without significant impact on the performance. To verify this assumption, we iterate every (c_i, w_i) pair in the training data as queries to search their k nearest neighbors from the datastore (k is set to a large number as 1024 here). In this process we compute an "importance score" for every entry in the datastore as $g = \sum_i 1/\text{rank}_i$, where rank_i is the rank of this entry among the nearest neighbors of the query $f(c_i)$. rank$= +\infty$ if it is not in the retrieval results. Intutively, the "importance score" up-weights the datastore records that appear more often with lower ranks in the retrieval results. Then we sort all the datastore records in terms of g and remove the ones with small scores, varying the compression rate. This method shares spirit with the technique in (Min et al., 2020) which filters out the articles that are never retrieved in memory-constrained open-domain question answering tasks.

Results: Figure 4 demonstrates the perplexity v.s. speed results on Wikitext-103 validation set of different datastore pruning methods described above. Only one solution point is reported for k-means since we do not vary the hyperparameters of k-means for different compression rate, given that its computational cost is much higher than other methods. Using 20% of the original datstore, k-means

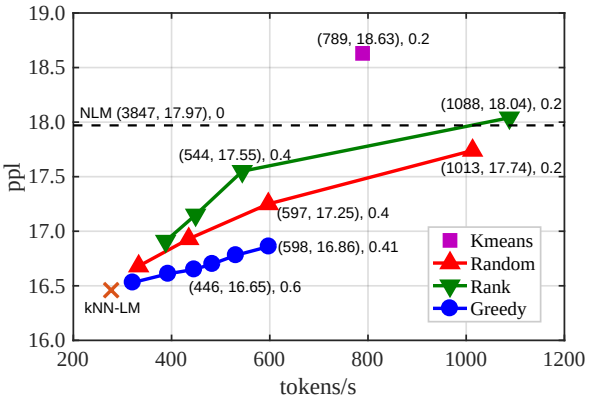

Figure 4: Perplexity and speed results of datastore pruning methods on WikiText-103 validation set. We annotate the coordinates of some points and the third number in the annotation is the compression rate (fraction of records remained).

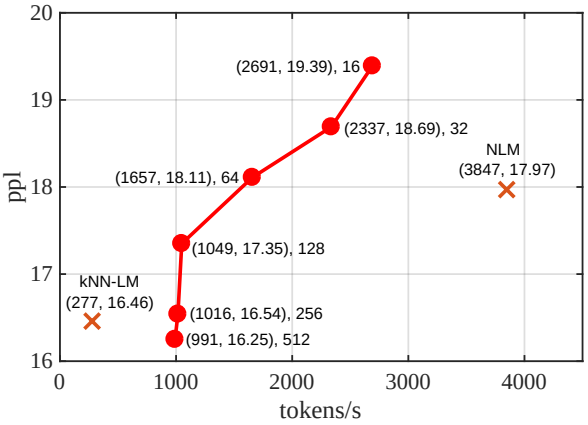

Figure 5: PCA dimension reduction results on WikiText-103 validation set. We annotate the coordinates of the points and the third number in the annotation is the PCA dimensions.

even underperforms the vanilla NLM baseline, suggesting that the cluster centroids approximation may lead to large distance errors which reduce the accuracy of the kNN distribution. Surprisingly, the simple random pruning method outperforms more complicated ones such as k-means and rank-based pruning. The best approach is greedy merging, which demonstrates a relatively flat curve compared with others.

4.3 Dimension Reduction

The context vectors $f(c)$ from large NLMs are often high-dimensional. For example, the pretrained NLMs that we use produce vectors of 1024 and 1536 dimensions in WikiText-103 and Law-MT respectively, which incurs significant datastore space and distance computation cost. To mitigate this issue, we empirically explore the effect of dimension reduction in kNN-LM . Specifically, we use prin-

cipal component analysis (PCA), an efficient and scalable dimension reduction algorithm, to reduce the dimensions and generate a new compressed datastore. We vary the new PCA dimensions as the hyperparameter and report the results.

Results: As shown in Figure 5, the evaluation becomes faster as expected with smaller dimensions, yet a too aggressive compression (dimension < 256) incurs large perplexity cost and even loses advantages over NLM when the dimension is smaller than 128. However, at 256 and 512 dimensions PCA is able to achieve comparable or even better performance than the original 1024-dim vectors, while attaining 3x-4x speed-up.[11]

5 Putting it All Together

Based on the analysis results in §4, in this section we combine best practices in adaptive retrieval, datastore pruning, and dimension reduction to assess the performance. We select the retrieval pruning rate r, datastore pruning rate n, and the reduced dimensions d on the validation set,[12] so that they achieve the largest speed-up at the cost of $<= 0.1$ perplexity compared to vanilla kNN-LM . We report the results on the test set.

Results: Table 3 shows the results on the test set of WikiText-103 and Law-MT, where we assess the combination of all three different strategies. Separate performance for each strategy is also included for reference points. On WikiText-103, adaptive retrieval is able to remove 50% of the retrieval and achieve nearly 2x speed-up, greedy merging prunes 40% of the datastore at the cost of 0.2 perplexity points. The dimension reduction method PCA leads to a minor improvement of perplexity over kNN-LM while being 3.6x faster. Combination of all the three techniques yields comparable perplexity to vanilla kNN-LM (16.67 v.s. 16.65) and a 6.6x speed-up (1835 v.s. 277).

Different from WikiText-103 where the datastore is contructed from the data that trains the pretrained NLM, in the Law-MT domain adaptation setting the datastore represents the domain-specific knowledge that the pretrained NLM never sees during training and thus is critical to produce good perplexity. This may be inferred from by the large ppl gains that the datastore offers (94 points). From another perspective though, the big improvement from the datastore retrieval leads to difficulties removing retrieval operations adaptively[13] – our learned retrieval adaptor is able to remove only 10% of the retrieval operations costing 0.1 ppl points. Greedy merging is able to prune 40% of the datastore losing 0.7 ppl points. We suspect that the Law-MT datastore is more vulnerable to pruning than the WikiText-103 one because of its smaller size (19M v.s. 103M) and corresponding lack of redundancy. Interestingly, the PCA dimension reduction yields 1 point ppl gain over the vanilla kNN-LM while achieving 3.3x speed-up, consistent with WikiText-103. This implies that a PCA transformation may be able to produce a new vector space that is more appropriate for defining $p_{k\text{NN}}$ with L^2 distances, we leave the underlying reasons for future work to discuss. Finally, a combination of the three allows kNN-LM to be evaluated 5.4x faster and even obtain superior perplexity.

Table 3: Perplexity and speed results on the test set of WikiText-103 and Law-MT. AR, GM, DR denote adaptive retrieval, datastore pruning, and dimension reduction respectively, "+All" denotes the combination of all the three technique.

Methods	ppl	tokens/s	speedup
WikiText-103			
NLM	18.66	3847	13.9x
kNN-LM	16.65	277	1x
+AR ($r = 0.5$)	16.67	530	1.9x
+GM ($n = 0.6$)	16.86	446	1.6x
+DR ($d = 512$)	16.40	991	3.6x
+All	16.67	1835	6.6x
Law-MT			
NLM	106.56	27.8K	264.3x
NLM (fine-tuned)	8.61	27.8K	264.3x
kNN-LM	12.64	1052	1x
+AR ($r = 0.1$)	12.74	1290	1.2x
+GM ($n = 0.6$)	13.33	1451	1.4x
+DR ($d = 512$)	11.59	3420	3.3x
+All	12.29	5708	5.4x

[11] The tool we use for PCA, the FAISS PCA implementation, applies random rotation to the PCA output vectors by default to re-balance variances of components of a vector (Gong et al., 2012), which may provide additional benefits over vanilla PCA on product vector quantization inside the index.

[12] Adaptive retrieval uses part of the validation data to training the retrieval adaptor network, thus we select r separately on its own held-out validation and then combine it to others.

[13] This can be reflected from the oracle comparison: $p_{\text{kNN}}(w|c) \geq p_{\text{NLM}}(w|c)$ 76% of the time compared to 39% in WikiText-103.

6 Implications and Future Work

In this paper, we explore several different ways to improve efficiencies of the k-nearest neighbors language model, achieving up to 6x speed-up while attaining comparable performance. As for future work, it is interesting to explore features from the datastore side to better know when to retrieve, and the gap between retrieval-based NLMs and parametric NLMs may be further reduced by combining more optimized indexing methods and the approaches in this paper.

Acknowledgements

We thank the anonymous reviewers for their comments, Emma Strubell, André Martins, Pedro Martins, and Uri Alon for helpful advice and discussions, and Wanzhen He for help with figure plotting. This material is based upon work supported by the National Science Foundation under Grant 1815287.

References

Roee Aharoni and Yoav Goldberg. 2020. Unsupervised domain clusters in pretrained language models. In *Proceedings of ACL*.

Rami Al-Rfou, Dokook Choe, Noah Constant, Mandy Guo, and Llion Jones. 2019. Character-level language modeling with deeper self-attention. In *Proceedings of AAAI*.

Fabien André, Anne-Marie Kermarrec, and Nicolas Le Scouarnec. 2019. Quicker adc: Unlocking the hidden potential of product quantization with simd. *IEEE transactions on pattern analysis and machine intelligence*.

Alexei Baevski and Michael Auli. 2019. Adaptive input representations for neural language modeling. In *Proceedings of ICLR*.

Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. 2015. Neural machine translation by jointly learning to align and translate. In *Proceedings of ICLR*.

Tom B Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. 2020. Language models are few-shot learners. *arXiv preprint arXiv:2005.14165*.

Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. Bert: Pre-training of deep bidirectional transformers for language understanding. In *Proceedings of NAACL*.

Martin Ester, Hans-Peter Kriegel, Jörg Sander, and Xiaowei Xu. 1996. A density-based algorithm for discovering clusters in large spatial databases with noise. In *Proceedings of KDD*.

Aristides Gionis, Piotr Indyk, Rajeev Motwani, et al. 1999. Similarity search in high dimensions via hashing. In *Proceedings of VLDB*.

Yunchao Gong, Svetlana Lazebnik, Albert Gordo, and Florent Perronnin. 2012. Iterative quantization: A procrustean approach to learning binary codes for large-scale image retrieval. *IEEE transactions on pattern analysis and machine intelligence*, 35(12):2916–2929.

Ruiqi Guo, Philip Sun, Erik Lindgren, Quan Geng, David Simcha, Felix Chern, and Sanjiv Kumar. 2020. Accelerating large-scale inference with anisotropic vector quantization. In *Proceedings of ICML*.

Kelvin Guu, Tatsunori B Hashimoto, Yonatan Oren, and Percy Liang. 2018. Generating sentences by editing prototypes. *Transactions of the Association for Computational Linguistics*, 6:437–450.

Junxian He, Taylor Berg-Kirkpatrick, and Graham Neubig. 2020. Learning sparse prototypes for text generation. In *Proceeings of NeurIPS*.

Sepp Hochreiter and Jürgen Schmidhuber. 1997. Long short-term memory. *Neural computation*, 9(8):1735–1780.

Herve Jegou, Matthijs Douze, and Cordelia Schmid. 2010. Product quantization for nearest neighbor search. *IEEE transactions on pattern analysis and machine intelligence*, 33(1):117–128.

Alexander Johansen and Richard Socher. 2017. Learning when to skim and when to read. In *Proceedings of the 2nd Workshop on Representation Learning for NLP*.

Jeff Johnson, Matthijs Douze, and Hervé Jégou. 2019. Billion-scale similarity search with gpus. *IEEE Transactions on Big Data*.

Urvashi Khandelwal, Angela Fan, Dan Jurafsky, Luke Zettlemoyer, and Mike Lewis. 2020. Nearest neighbor machine translation. *arXiv preprint arXiv:2010.00710*.

Urvashi Khandelwal, Omer Levy, Dan Jurafsky, Luke Zettlemoyer, and Mike Lewis. 2019. Generalization through memorization: Nearest neighbor language models. In *Proceedings of ICLR*.

Diederik P Kingma and Jimmy Ba. 2015. Adam: A method for stochastic optimization. In *Proceedings of ICLR*.

Philipp Koehn and Rebecca Knowles. 2017. Six challenges for neural machine translation. In *Proceedings of the First Workshop on Neural Machine Translation*.

Stephen Merity, Caiming Xiong, James Bradbury, and Richard Socher. 2017. Pointer sentinel mixture models. In *Proceedings of ICLR*.

Tomáš Mikolov, Martin Karafiát, Lukáš Burget, Jan Černockỳ, and Sanjeev Khudanpur. 2010. Recurrent neural network based language model. In *Eleventh annual conference of the international speech communication association*.

Sewon Min, Jordan Boyd-Graber, Chris Alberti, Danqi Chen, Eunsol Choi, Michael Collins, Kelvin Guu, Hannaneh Hajishirzi, Kenton Lee, Jennimaria Palomaki, et al. 2020. Neurips 2020 efficientqa competition: Systems, analyses and lessons learned. *arXiv preprint arXiv:2101.00133*.

Marius Muja and David G Lowe. 2009. Fast approximate nearest neighbors with automatic algorithm configuration. In *Proceedings of the International Conference on Computer Vision Theory and Applications*.

Nathan Ng, Kyra Yee, Alexei Baevski, Myle Ott, Michael Auli, and Sergey Edunov. 2019. Facebook fair's wmt19 news translation task submission. In *Proceedings of the Fourth Conference on Machine Translation (Volume 2: Shared Task Papers, Day 1)*.

Matthew Peters, Mark Neumann, Mohit Iyyer, Matt Gardner, Christopher Clark, Kenton Lee, and Luke Zettlemoyer. 2018. Deep contextualized word representations. In *Proceedings of NAACL*.

Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. 2019. Language models are unsupervised multitask learners. *OpenAI blog*, 1(8):9.

Alexander M Rush, Sumit Chopra, and Jason Weston. 2015. A neural attention model for abstractive sentence summarization. In *Proceedings of EMNLP*.

Josef Sivic and Andrew Zisserman. 2003. Video google: A text retrieval approach to object matching in videos. In *Proceedings of ICCV*.

Martin Sundermeyer, Ralf Schlüter, and Hermann Ney. 2012. Lstm neural networks for language modeling. In *Thirteenth annual conference of the international speech communication association*.

Nenad Tomasev, Milos Radovanovic, Dunja Mladenic, and Mirjana Ivanovic. 2013. The role of hubness in clustering high-dimensional data. *IEEE transactions on knowledge and data engineering*, 26(3):739–751.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Lukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. In *Proceedings of NeurIPS*.

Zhilin Yang, Zihang Dai, Yiming Yang, Jaime Carbonell, Russ R Salakhutdinov, and Quoc V Le. 2019. XLNet: Generalized autoregressive pretraining for language understanding. In *Proceedings of NeurIPS*.

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

A Experimental Setup Details

A.1 General Setup Details

The interpolation hyperparameter λ is tuned in the range [0,1, 0.9] with interval 0.05 on the validation split of each dataset separately. As a result, $\lambda = 0.25$ in WikiText-103 and $\lambda = 0.9$ in Law-MT.

A.2 Adaptive Retreival

We use the same adaptive retrieval configuration hyperparameters for different datasets, which are validated on the WikiText-103 dev set: the retrieval adaptor is a MLP network with 4 hidden layers, 1 input layer and 1 output layer. Each layer is a linear transformation followed by the ReLU non-linear activation, and a dropout layer with 0.2 dropout probability, except for the output layer where the hidden unites are transformed to 2 dimensions followed by a log softmax to produce $\log \lambda$ and $\log(1 - \lambda)$. The number of hidden units in each layer is 128. Before passing the input features to MLP, we transform each of the scalar features (all the features except for $f(c)$) into an m-dim vector, where $m = \dim(f(c))/n$ and n is the number of scalar feature types. This is to balance the context vector feature and other features. The scalar-feature transformation is performed with an one-layer `Linear(in, out)-ReLU-Linear(out, out)` network. We also tried using LSTM (Hochreiter and Schmidhuber, 1997) network to capture the temporal relations yet found it leads to very unstable training and fails to converge, though we note that MLP is faster at test time and the $f(c)$ feature already captures the temporal correlations between tokens. The coefficient of the L^1 regularizer a is tuned on WikiText-103 validation set among $\{0.01, 0.05, 0.1, 0.2, 0.5, 1\}$ and fixed as 0.05 for both WikiText-103 and Law-MT. The model is trained using the Adam optimizer (Kingma and Ba, 2015) with learning rate 0.0005. The checkpoint with the best validation perplexity at 50% pruning is saved.

B Ablation Analysis

Input features to retrieval adaptor: We analyze the effect of different input features to the retrieval adaptor by removing a subset of features. We report the perplexities at 50% retrieval pruning, because using different features only has a marginal effect on the evaluation speed. Results on the WikiText-103 test set are shown in Table 4. All features together produce the best results, while the perplexity is relatively robust to removal of a single feature. In our experiments (§4 and §5) we drop off the log freq feature and use the others to save memory while achieving comparable perplexities to using all features.

Table 4: Results of kNN-LM + adaptive retrieval using different input features. The perplexities are based on removing 50% of the retrieval operations after training retrieval adaptor.

Features	ppl
$f(c)$ + conf + ent + log freq + log fert	16.63
-log freq	16.67
-log fert	16.72
-ent, conf	16.77
-$f(c)$	16.71
-conf, ent, log freq, log fert	17.03

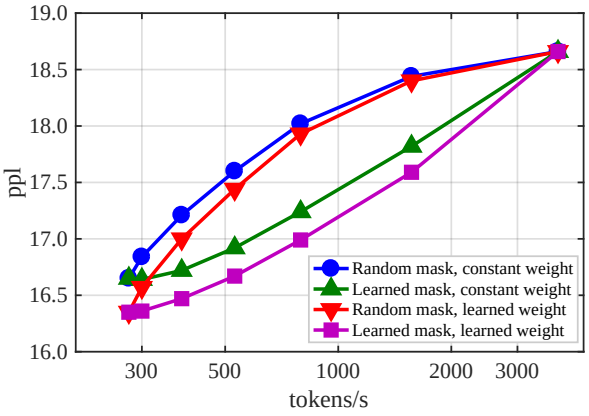

Figure 6: Perplexity and speed results of adaptive retrieval on WikiText-103 test set. This figure includes different variants of adaptive retrieval for ablation analysis.

Effect of learnable interpolation weights: In the adaptive retrieval analysis (§4.1), we observed gains of a learned retrieval adaptor over a random baseline at different fractions of retrieval prunning. However, the advantages may come from two sources: (1) the automatically identified prunning masks agains the random masks, and (2) the learned interpolation weights on the remaining retrievals against the constant weights that random baseline uses. To separate the two effects, we perform an ablation study to analyze the results of (1) random mask, constant weight ("Random" in §4.1), (2) random mask, learned weight – the weights are from the trained retrieval adaptor, (3) learned mask, constant weight, and (4) learned mask, learned weight ("Adaptor" in §4.1). The results are shown in Figure 6, "learned mask, learned weight" performs

the best. While minor gains are from the automatically learned weights ("Random mask, learned weights"), most of the superiority can be attained with the smart pruning strategy even with constant weights ("Learned mask, constant weights").