Why do Nearest Neighbor Language Models Work?

Frank F. Xu [1] **Uri Alon** [1] **Graham Neubig** [1]

Abstract

Language models (LMs) compute the probability of a text by sequentially computing a representation of an already-seen context and using this representation to predict the next word. Currently, most LMs calculate these representations through a neural network consuming the immediate previous context. However recently, *retrieval-augmented LMs* have shown to improve over standard neural LMs, by accessing information retrieved from a large datastore, in addition to their standard, parametric, next-word prediction. In this paper, we set out to understand *why* retrieval-augmented language models, and specifically why k-nearest neighbor language models (kNN-LMs) perform better than standard parametric LMs, even when the k-nearest neighbor component retrieves examples from the same training set that the LM was originally trained on. To this end, we perform analysis of various dimensions over which kNN-LM diverges from standard LMs, and investigate these dimensions one by one. Empirically, we identify three main reasons why kNN-LM performs better than standard LMs: using a different input representation for predicting the next tokens, *approximate* kNN search, and the importance of softmax temperature for the kNN distribution. Further, we incorporate some insights into the standard parametric LM, improving performance without the need for an explicit retrieval component. The code is available at https://github.com/frankxu2004/knnlm-why.

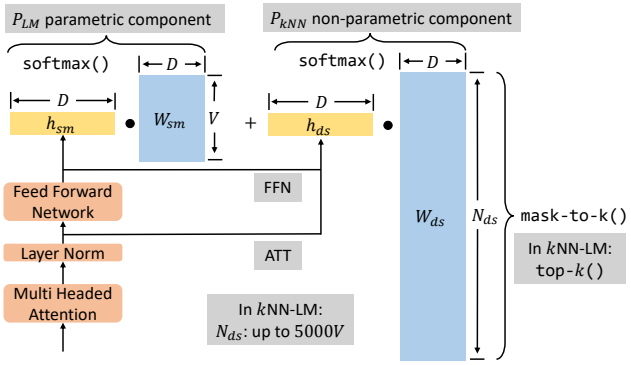


Figure 1: An illustration of the generalized formulation of kNN-LM in Equation 2.

1. Introduction

Language modeling is the task of predicting the probability of a text (often conditioned on context), with broad-spanning applications across natural language processing (Bengio et al., 2003; Merity et al., 2018; Baevski & Auli, 2018; Brown et al., 2020). It is usually done by sequentially encoding a context c_t using a trained neural network function f, and computing the probability of the next word w_t according to $f(c_t)$ and a vector representation of w_t.

Recently, *retrieval-augmented* LMs have shown a series of impressive results (Grave et al., 2017; Guu et al., 2018; He et al., 2020; Khandelwal et al., 2020b; Borgeaud et al., 2022; Alon et al., 2022; Zhou et al., 2022). Retrieval-augmented LMs compute next token distribution based not only on the immediately preceding context c_t and the model parameters, but also on an external datastore, from which examples are retrieved and incorporated into the base LM's prediction. One such model that is notable for both its simplicity and efficacy is the k-nearest neighbor language model (kNN-LM; Khandelwal et al., 2020b). kNN-LM extends a trained base LM by linearly interpolating the output distribution with a kNN model. The nearest neighbors are retrieved according to the distances between the current context embedding of the base LM and all the context embeddings in the datastore. The datastore is created by encoding all contexts from any text, including the original LM training data.

One of the most surprising results from Khandelwal et al.

[1]Language Technologies Institute, Carnegie Mellon University, Pittsburgh, Pennsylvania, United States. Correspondence to: Frank F. Xu <fangzhex@cs.cmu.edu>, Graham Neubig <gneubig@cs.cmu.edu>.

Proceedings of the 40^{th} International Conference on Machine Learning, Honolulu, Hawaii, USA. PMLR 202, 2023.

(2020b) is that kNN-LM reduces the perplexity of the base LM *even when the kNN component is retrieving examples from the same training set that the LM was originally trained on*, indicating that kNN-LM improves the ability to model the training data and is not simply benefiting from access to more data. Intrigued by this finding, we wonder why does kNN-LM work, and how does it improve already-trained strong transformer-based models? In this paper, we set out to understand why kNN-LMs work even in this setting.

In the following sections, we first elucidate connections between the added kNN component and the standard LM component. Specifically, we note that word distributions from both components are calculated using a softmax function, based on the similarity of the current context hidden state with a set of embeddings that corresponds to different next words. With this intuition, we formalize and generalize the non-parametric distribution with the softmax layer and word embedding layer used in parametric LMs. We then show that this generalized form exposes a variety of design choices, e.g., the number of context embeddings in the datastore, the input representation used in softmax layer, different similarity functions, as well as the approximation and sparsification implementations in the kNN search. This provides a general framework for analyzing kNN-LM and similar models and allows us to perform ablation studies that test the importance of various design decisions.

We proceed to propose multiple hypotheses as to why kNN-LM works, which are testable by adjusting the various parameters exposed by our generalized formulation. Based on these hypotheses, we perform ablation experiments and analyze the nuances between different implementations of the generalized version of P_{kNN}. As the answer to our question, "why do kNN-LMs work?", we eventually show that the most probable reasons are threefold:

1. Ensembling the output of softmax using two representations from different layers of the transformer is important; in our experiments, this accounts for 55% of the performance gain of kNN-LM, or 6.5% relative perplexity improvement compared to the base LM.

2. kNN-LM uses *approximate* nearest neighbor search to handle the large number of candidates, and the lack of preciseness in the algorithm actually helps kNN-LM to generalize *better* than exact nearest neighbor search and distance calculation, possibly due to regularization effect. The relative perplexity improvement from this factor is about 2.6%.

3. Depending on the design decisions that are chosen for modeling, adding a temperature term to the kNN non-parametric component can become crucial to the success of modeling (although coincidentally, in the original settings of Khandelwal et al. (2020b), a temperature of 1.0 was close to optimal, which hid the importance of this term). In some settings, the relative perplexity gap between the default and optimal temperature can be as high as 8.4%.

Finally, one significant drawback to the current kNN-LM is the inefficiency of kNN search performed at each step (He et al., 2021; Borgeaud et al., 2022; Alon et al., 2022; Wang et al., 2022). Because of the similarity between kNN-LM and the parametric LM's last layers and the many design choices, we also demonstrate that we are able to make kNN-LM more efficient by substituting the kNN search with another matrix operation that can fit in accelerator memory while maintaining more than half the perplexity improvement, or 6.5% relative improvement to the base LM.

2. Formalizing and Generalizing kNN-LM

kNN-LM (Khandelwal et al., 2020b) is a linear interpolation between a base LM and a kNN model. Given a set of contexts c_i and their corresponding next token w_i as a pair $(c_i, w_i) \in \mathcal{D}$, kNN-LMs create a datastore $(\mathcal{K}, \mathcal{V}) = \{(k_i, v_i)\}$, as a set of key-value pairs $(\mathcal{K}, \mathcal{V}) = \{(f(c_i), w_i) \mid (c_i, w_i) \in \mathcal{D}\}$, where $f(c_i)$ is typically a transformer's hidden state after reading c_i. During inference, the parametric component generates the output distribution $p_{LM}(w_t|c_t;\theta)$ over the next tokens and produces the corresponding context representation $f(c_t)$, given the test input context c_t. Then the non-parametric component queries the datastore with the $f(c_t)$ representation to retrieve its k-nearest neighbors $\mathcal{N}$ with a distance function $d(\cdot,\cdot)$. Next, kNN-LM computes a probability distribution over these neighbors using the softmax of their negative distances, and aggregates the probability mass for each vocabulary item across all of its occurrences in the retrieved targets:

$$p_{\text{kNN}}(w_t|c_t) \propto \sum_{(k_i,v_i)\in\mathcal{N}} \mathbf{1}_{w_t=v_i} \exp(-d(k_i, f(c_t))) \quad (1)$$

Finally, this distribution is interpolated with the parametric LM distribution p_{LM} to produce the final kNN-LM distribution $p(w_t|c_t;\theta) = (1-\lambda)p_{\text{LM}}(w_t|c_t;\theta) + \lambda p_{\text{kNN}}(w_t|c_t)$, where λ is a scalar that controls the weights of the interpolation between two components, with higher λ putting more weight on the non-parametric component.

Looking closely at Equation 1, we notice a similarity between the calculation of P_{kNN} and the standard P_{LM}. The kNN distribution is based on the distances between the current context and the nearest neighbors from the datastore, normalized by a softmax function. Recall that in (standard) parametric language models, the distribution over the vocabulary is also based on a measure of distance, the inner product between the current context embedding and the word embeddings of every token in the vocabulary. Because each context embedding in the datastore $(\mathcal{K}, \mathcal{V})$ corresponds to a target token, we can also view this datastore as a large word embedding matrix with multiple word embeddings for

each of the vocabulary words. Theoretically, given unlimited computation, we could calculate the distribution based on the distances to every embedding in the datastore, and aggregate by vocabulary items, making it more closely resemble P_{LM}. For Equation 1, this will result in $k = |\mathcal{D}|$, the size of the entire datastore, and $\mathcal{N} = \mathcal{D}$, using the distances to every context in the datastore instead of a subset of nearest neighbors. In practice, we use kNN search as a way of approximation, by limiting the calculation to only k nearest neighbors to avoid the computational cost of calculating the distribution over the entire datastore.

If we re-write and generalize Equation 1, both the kNN-LM of Khandelwal et al. (2020b) and a large number of related models can be expressed through the following equation:

$$P_{\text{interp}} = (1-\lambda)\underbrace{\text{softmax}(W_{sm} \cdot h_{sm})}_{P_{\text{LM}} \text{ parametric component}} + \lambda \underbrace{M\text{softmax}(\text{mask-to-k}(W_{ds} \otimes h_{ds})/\tau)}_{P_{\text{kNN}} \text{ non-parametric component}}. \quad (2)$$

Figure 1 provides an illustration of Equation 2. The first term of the equation is the standard parametric language model, whereas the second represents a generalized version of utilizing an external datastore. The first component, the output layer of a common parametric language model, is relatively straightforward. W_{sm} of size $V \times D$ is the embedding matrix of the output token, and h_{sm} is the context vector to calculate the distribution of the output token, usually the output of the final feedforward layer in the transformer.

In the second component, W_{ds} represents the datastore, of size $N_{ds} \times D$. N_{ds} is the number of entries in the datastore, and D is the size of each context vector. h_{ds} represents the context vector used to query the datastore. As shown in Figure 1, h_{ds} may come from a different layer of the transformer than h_{sm}. The operator $\otimes$ represents the operation type used to calculate the similarity between context vectors and the query vector, which also has several alternatives that we discuss below. mask-to-k$(\cdot)$ represents a function to sparsify similarity scores across the datastore, setting all but k similarity scores to $-\infty$, which results in probabilities of zero for all masked similarity scores after the softmax. Practically, this is necessary for kNN-LMs because the size of the datastore N_{ds} makes it infeasible to calculate all outputs at the same time. With the masked logits, we apply a more generalized version of softmax with temperature τ. Intuitively adding the temperature can adjust the peakiness or confidence of the softmax probability distribution output. After the softmax, the matrix M of dimension $V \times N_{ds}$ sums the probability of the N_{ds} datastore entries corresponding to each of the V vocabulary entries. Each column in this matrix consists of a one-hot vector with a value of 1 and the index corresponding to the vocabulary item w_i corresponding to the datastore entry for c_i.

Within this formulation, it becomes obvious that there are many design choices for kNN-LM-like models. One important thing to note is that the right side of Equation 2 is actually very similar to the left side representing the standard parametric language model, with a few additional components: M, mask-to-k, and $\otimes$. More specifically, some of the design decisions that go into the kNN-LM, and parallel with standard parametric models are:

Size of W_{ds}: In standard parametric model, the size of W_{sm} is V embeddings, each with D dimensions. In kNN-LM the size of W_{ds} is very large: N_{ds}, the size of the datastore, usually the number of tokens in the training corpus.

Input representation: In the parametric model, h_{sm} is the output from the feedforward layer in the last transformer block, which we abbreviate "ffn". In contrast, kNN-LM rather use as h_{ds} the output from the multi-headed attention layer of the last transformer block (before running the representations through the feed-forward network, and *after* the LayerNorm (Ba et al., 2016)), which we abbreviate as "att".

Similarity & Temperature: In the parametric model, the functional form of $\otimes$ is the inner product (abbreviated IP), whereas kNN-LM use negative squared L2 distance (abbreviated L2) as a similarity function between W_{ds} and h_{ds}. As the similarity scores are turned into probability distributions with the softmax function, the choice of softmax temperature (τ) can control the scaling of the similarity scores and thus the peakiness of the non-parametric distribution.

Approximation & Sparsification: In the parametric model, $k = V$, and no values are masked, but in the kNN-LM, $k \ll V$, and most of the datastore entries are pruned out. The definition of the mask-to-k$(\cdot)$ function, i.e. how to select the important datastore embeddings to include in the similarity calculation (in kNN-LM's case the k nearest neighbors), is a crucial open design choice.

In the following sections, we set out to better understand how each of these design decisions contributes to the improvement in accuracy due to the use of kNN-LMs.

3. Baseline kNN-LM Results

First, we evaluate kNN-LM on Wikitext-103 (Merity et al., 2016), and examine the importance of two design choices: the input representation h_{ds} and the similarity function $\otimes$.

In models examined in this paper, the parametric model is a transformer language model with mostly the same architecture as in Khandelwal et al. (2020b). However, we make slight modifications to the original base LM (Baevski & Auli, 2018) to accommodate our experimentation need. We use BPE tokenization (Sennrich et al., 2015) to train a smaller vocabulary (33K) than the original (260K) on the training corpus of Wikitext-103, as subword tokenization is

	h_{ds}	$\otimes$	+#params	PPL	Interp.	Oracle
Base LM	-	-	0	21.750	-	-
kNN-LM-L2	att	L2	$N_{ds} \times D$	∞	19.174	14.230
kNN-LM-IP	att	IP	$N_{ds} \times D$	∞	19.095	14.077
kNN-LM-L2	ffn	L2	$N_{ds} \times D$	∞	20.734	15.594
kNN-LM-IP	ffn	IP	$N_{ds} \times D$	∞	21.101	16.254

Table 1: Performance of the parametric language model and several kNN-LM variants.

ubiquitous in many state-of-the-art language models (Devlin et al., 2018; Brown et al., 2020). Using subword tokenization also eliminates the need for adaptive softmax (Joulin et al., 2017). This makes the output layer more general, sharing more resemblance to the kNN component as described in Section 2, and facilitates the ablation studies in this paper.[1] This base LM has 268M parameters. To get a perspective on how large the datastore is, it is built on the training data that contains nearly 150M BPE tokens, each paired with a context vector of size 1024. This datastore has a total memory consumption of about 300GB. Following Khandelwal et al. (2020b), at every retrieval step, we take the top 1024 nearest neighbors, i.e., $k = 1024$. The interpolated perplexity is computed with optimal interpolation parameter λ tuned according to the perplexity on the development set, and fixed during inference.

Results comparing multiple kNN-LM variants are shown in Table 1. The first row represents the base parametric language model's perplexity. The second is a formulation analogous to that of kNN-LM, and in the remaining rows, we vary the input representation h_{ds} and distance function $\otimes$ from Equation 2. All variants use a large datastore with size N_{ds}, approximately 5000 times the size of the vocabulary V, as also reflected in "+#params", the number of *additional* parameters other than the base LM.

We report several important quantities. "Interp." shows the interpolated perplexity. "PPL" shows the perplexity of *only* the kNN component of the model $p_{\text{kNN}}()$. This is ∞ for all kNN-LM models, as when the kNN search does not retrieve any datastore entries corresponding to the true target word w_t the probability of it will be zero. "Oracle" shows the lower bound of the interpolated perplexity by choosing the best λ for each token in the evaluation dataset, which will either be $\lambda = 0$ or $\lambda = 1$ depending on whether $P_{LM}(w_t|c_t) > P_{knn}(w_t|c_t)$. From the table, we see that:

1. Using the output of the multi-headed attention layer ("att") as h_{ds} (instead of the standard "ffn" layer) is crucial for better performance of kNN-LM.

2. In general, using negative squared L2 distance or inner product as a similarity function does not result in a large and consistent difference, although in our setting, IP provides slightly better performance when using the "att" inputs, and slightly worse when using "ffn" inputs.

3. Interestingly, when using "ffn" and "IP", the same input and distance metric used in the parametric model, the results are the worst, indicating that kNN-LM particularly benefits from a *different view* of the data than the parametric model.

We found in preliminary experiments that kNN-LM is generalizable to other base language models as well, ranging from small models with 82M parameters to larger models with 774M parameters. The gain from kNN-LM is more significant when used with a smaller, less capable base language model (Appendix A) In this paper, we mainly focus on the factors contributing to the *relative* improvements from kNN-LM, instead of the absolute performance, so we use the 268M model for the remainder of the paper. In the next sections, we perform ablation experiments on the general formulation Equation 2 to elucidate the key elements contributing to the performance improvements in kNN-LM.

4. Effect of Different W_{ds} Formulations

	h_{ds}	N_{ds}	+#params	PPL	Interp.	Oracle
Base LM	-	-	0	21.750	-	-
kNN-LM	att	Big	$N_{ds} \times D$	∞	19.095	14.077
Learned W_{ds}	att	1x	$V \times D$	22.584	20.353	16.954
kNN-LM	ffn	Big	$N_{ds} \times D$	∞	21.101	16.254
Learned W_{ds}	ffn	1x	$V \times D$	20.920	20.694	18.772

Table 2: Performance comparison how the choice of h_{ds}, input representation, affects kNN baselines and models with learnable embeddings as datastore alternative. h_{ds} is the attention layer output. $\otimes$ is IP.

4.1. Replacing Datastore with Trainable Embeddings

From the observation in Section 3, we see that the choice of h_{ds} has a large impact on the performance of kNN-LM. This intrigues us to explore if one key to the improvements of kNN-LM lies in the combination of different input representations, namely the attention output (h_{ds} = att) and feedforward output (h_{ds} = ffn). However, based only the experiments above, it is not possible to disentangle the effect of the choice of h_{ds} and that of other design choices and factors in Equation 2.

To test the effect of the choice of h_{ds} in a more controlled setting, we remove the non-parametric datastore entirely, and initialize W_{ds} in Equation 2 with a randomly initialized

[1] By re-training the base LM from scratch with BPE tokenization and a standard output softmax, our LM's perplexity is worse than reported by Khandelwal et al. (2020b). However, we observe similar relative gains from the additional kNN component, and we argue that the base LM is orthogonal to the study of the factors behind kNN-LM's improvements.

word embedding matrix of the same size ($N_{ds} = V$) as the LM's output embedding W_{sm}, and train W_{ds} with all other parameters fixed.[2] The loss function for training is the cross-entropy loss of softmax($W_{ds} \cdot h_{ds}$) with respect to the ground-truth tokens, identically to how the base LM is trained. We compare how using $h_{ds} =$ att or $h_{ds} =$ ffn affects the interpolated performance. The results are shown in Table 2, with the results of kNN-LMs using these two varieties of input representation for reference. From these experiments we find several interesting conclusions:

Effectiveness of re-training $\mathbf{W_{ds}}$: In the case of "Learned W_{ds} w/ FFN", we are essentially re-learning the weights for the softmax function separately from the underlying LM encoder. Despite this fact, the model achieves a PPL of 20.920, which is 0.83 points better than the base model. This suggests that it is beneficial to learn the parameters of W_{ds} after freezing the transformer encoder.

Effectiveness of ensembling two predictors: In both cases of W_{ds}, the interpolated perplexity is significantly better than that of using a single predictor. This is particularly the case when using the "att" representation for h_{ds}, suggesting that the utility of ensembling predictions from two views of the data is not only useful when using kNN-LM, but also in standard parametric models as well.

Parametric ensembles as an alternative to kNN-LM? Overall, by using a separate word embedding matrix with size $V \times D$ as an alternative to kNN, we can recover about 55% of the performance gain achieved by kNN-LM, with only a limited number of parameters and without the necessity for slow kNN retrieval every time a token is predicted. This suggests that the majority of the gain afforded by kNN-LM could be achieved by other more efficient means.

4.2. Increasing the Softmax Capacity

One premise behind kNN-LM is that the large datastore is the key reason for the kNN-LM's success: the larger the datastore's capacity, the better the performance. We wonder whether such a big datastore is warranted and whether the size and expressivity of W_{ds} leads to better performance. We test the effect of the datastore size for kNN retrieval on kNN-LM interpolated perplexity. If a bigger datastore is better in kNN-LM than a smaller datastore, then the hypothesis of softmax capacity is more probable. We randomly subsample the full datastore in varying percentages and the results are shown in the blue "FAISS mask, FAISS score" series in Figure 3. The full datastore contains more than 150M entries and storing them takes 293GB when using fp16. We see that the perplexity decreases linearly with a higher fraction of the original datastore. Even with just 5% of the datastore size (15G), kNN-LM still provides a benefit over the base LM. However, even when the subsampling percentage reaches 90%, more entrie in the datastore still provide benefits without having significant diminishing returns, suggesting that a large datastore is beneficial.

One possible reason why a larger datastore is helpful is that some words can be difficult to predict. There are several reasons: (1) They are rare, or (2) they are frequent, but they have multiple meanings and appear in different contexts. The softmax bottleneck (Yang et al., 2017) suggests that the final dot product of language model $W_{sm} \cdot h_{sm}$ is capped at D rank, limiting the expressiveness of the output probability distributions given the context; that is, a single output vector of a fixed (1024) size cannot express all the possible mappings between 100M training examples and 33K vocabulary outputs. We hypothesize that kNN-LM improves performance by alleviating the problem, since $M \exp(W_{ds} \otimes h_{ds})$ has a higher rank ($M\cdot$ sums softmax outputs of the same token) and is more expressive than just $\exp(W_{sm} \cdot h_{sm})$. kNN is a sparse approximation of the full softmax over all the embeddings in the datastore W_{ds}. To test this hypothesis, we disentangle the effect of W_{ds} size from the actual saved context embeddings in W_{ds}, by training an embedding matrix of the same size from scratch.

We explore several potential solutions for increasing the capacity of softmax, and examine if they can achieve a similar effect to kNN-LM. The first and easiest solution is to increase the embedding matrix size by adding more embedding vectors for each word type in the vocabulary. To test this, we replace W_{ds} with a much smaller matrix of size $nV \times D$, where we allocate n embedding vectors for each word type. When calculating the probability from this component, we compute the softmax over nV items and sum the probabilities for each vocabulary entry. mask-to-k($\cdot$) is no longer needed, as this formulation is small enough to fit the entire matrix in the GPU. We then finetune the new W_{ds} on the training data until convergence.

Figure 2 compares the base LM using the original kNN-LM with using either the attention layer output ("att") or the feedforward layer output ("ffn") as h_{ds}. We plot the number of embeddings for each word type (nV total embeddings in W_{ds}) versus the interpolated perplexity, with full details found in Appendix B. In both cases, comparing with the top horizontal line which represents the perplexity of the base LM, replacing the datastore with a much smaller weight matrix (from N_{ds} to nV_{ds}) by assigning only a few more embeddings for each word helps, although only about half as effective as kNN-LM. To give a perspective, the original datastore size is about $5000V$. Surprisingly, we find that increasing n does not always bring better performance, even though a larger datastore is better than using a small

[2] Because we previously found little difference between IP and L2 as similarity functions, we use IP in the experiments. For simplicity, we set temperature $\tau = 1$.

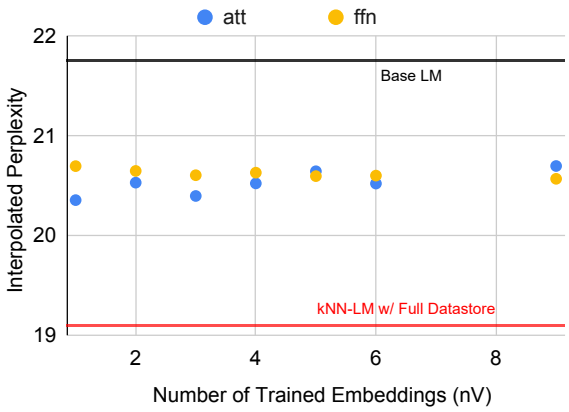


Figure 2: The number of embeddings per word type (nV total embeddings in W_{ds}) versus interpolated perplexity, compared with base LM and kNN-LM.

datastore in kNN-LM. We see that when h_{ds} = ffn, over-parameterization provides limited improvements, while for h_{ds} = att it does not bring consistent improvements at all. Comparing the trend of increasing the embeddings in W_{ds}, with the bottom horizontal line in the plot, which represents the perplexity of the standard kNN-LM using the full datastore (W_{ds} with approx. $5000V$ embeddings), we see no clear trend that more trainable embeddings result in better perplexity, and that the gap between using trained embeddings and using full datastore is still significant. This suggests that simply over-parameterizing W_{ds} is not an effective method of achieving gains similar to kNN-LM.

We hypothesize that this is because by just adding more embeddings, while still using the same training procedure as the original LM, the multiple embeddings for each word type after learning could still be very close to each other, and thus do not increase the softmax capacity much. This suggests that some regularization terms may be needed during training to make the multiple embeddings not converge to the same vector, rendering over-parameterization useless.

Besides simply increasing the number of embedding vectors equally for each word type, we also propose other alternatives to increase softmax capacity. First, we hypothesize that different word types have different difficulties for the language model to predict. For those words that appear very frequently, they may appear in many different contexts. As a result, instead of adding an equal number of additional embeddings to each word type, we propose to adaptively increase the number of embeddings for word types based on word frequency, or total training loss for the word. Second, we try to break the softmax bottleneck. Yang et al. (2017) proposes a solution using Mixture of Softmax (MoS) to produce more linearly independent probability distributions of words given different contexts. Last, instead of training word embeddings of increased size, we also consider compressing the datastore down to a similar-sized embedding matrix for softmax by clustering the datastore and finetuning of the matrix consisting of cluster centroids. However, none of these alternative methods provided additional benefits over the simple multi-embedding approach (Appendix C).

	PPL	Interp.	Oracle
Base LM	21.750	-	-
kNN-LM w/ FAISS mask, FAISS score	∞	19.174	14.230
kNN-LM w/ FAISS mask, real score	∞	19.672	14.393
kNN-LM w/ real mask, real score	∞	19.735	14.480

Table 3: Performance of the parametric language model and comparison of kNN-LMs using the approximate versus ground truth kNN. $\otimes$ is L2. h_{ds} = att.

5. Approximate kNN & Softmax Temperature

5.1. Comparing Approximate kNN Search

To calculate P_{kNN} of the non-parametric component in Equation 2, it is usually prohibitive to use exhaustive kNN search, and thus Khandelwal et al. (2020a) use approximate kNN search using the FAISS library (Johnson et al., 2019). The use of FAISS (similarly to other approximate search libraries) results in two varieties of approximation.

Approximate Neighbors: Because the search for nearest neighbors is not exact, the set of nearest neighbors might not be equivalent to the actual nearest neighbors. Recall that the function mask-to-k$(\cdot)$ in Equation 2 is the function that selects kNN entries from the datastore W_{ds}. We denote "real mask" as the accurate nearest neighbors for mask-to-k$(\cdot)$ selection, and "FAISS mask" as the approximate nearest neighbors returned by the FAISS library.

Approximate Scores: In addition, FAISS makes some approximations in calculating the distances between the query and the retrieved neighbors for efficiency purposes. We denote "real score" as the scores calculated from ground truth distances between the embeddings, and "FAISS score" as the distances returned by FAISS approximate search.

The comparison of the different approximation settings is shown in Table 3. Quite surprisingly, we actually find that the interpolated perplexity with approximate search is *better* than that with exact search, both with respect to the mask and the score calculation. Intrigued by this counter-intuitive result, we explore the effect of kNN search approximation.

First, we plot the subsampled size of the datastore with the interpolated perplexity Figure 3, but showcasing the comparison between approximate and real masks, approximate and real scores in both the full datastore as well as a small subsampled datastore setting. We find that using an *approximate* FAISS mask to find nearest neighbors performs better

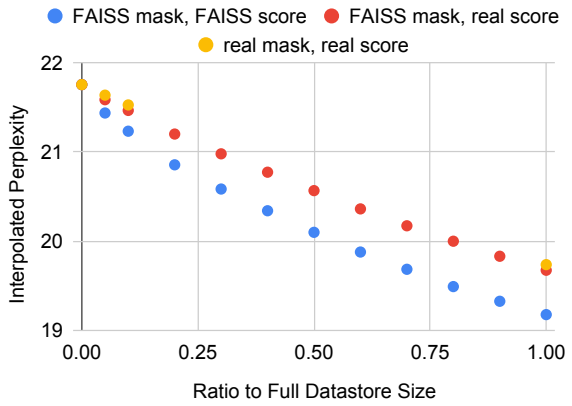
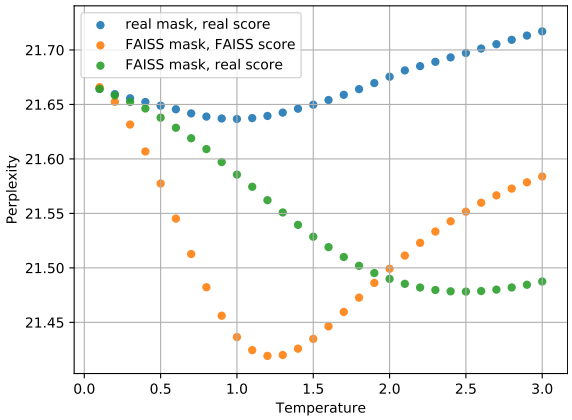
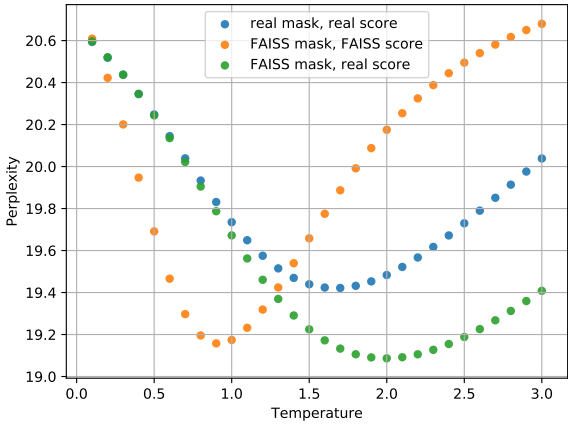


Figure 3: The differences between using approximate and accurate kNN search on varying sizes of the datastore.



(a) On 5% subsampled datastore.



(b) On full datastore.

Figure 4: The interpolated perplexity varies with different softmax temperature τ values.

than using the exact nearest neighbors *both* at 5% and 100% of the datastore. However, using the approximate score returned by FAISS is better than recomputing the exact distances between embeddings for the kNN distribution *only* for the small 5% datastore scenario. Interestingly, the gap between using an approximate score or real score given the same approximate neighbors ("FAISS mask, FAISS score" vs. "FAISS mask, real score") is larger than that between using approximate or real neighbors given the same ground truth method of calculating the distance ("real mask, real score" vs. "FAISS mask, real score").

We hypothesize that this is related to regularization for preventing overfitting, and approximate search provides fuzziness that functions as a regularizer. We can think of the kNN component of kNN-LM as a model, where the datastore size is the model capacity, and the datastore is its training data. Considering that the kNN component uses the exact same training data as the base parametric LM, having ground truth, accurate kNN search may cause the kNN component to overfit the training data.

5.2. Adding Softmax Temperature to kNN Distribution

Because the number of retrieved nearest neighbors, k, is usually much smaller than the vocabulary size V, intuitively, the kNN distribution P_{kNN} used for interpolation tends to be more peaky than the standard LM output distribution. When $k = 1024$ and $V = 33000$, as in our experiments, P_{kNN} will only have a few vocabulary items with a non-zero probability. Furthermore, many of the retrieved neighbors share the same target token and thus make the kNN distribution even peakier. One way to control the entropy, or peakiness of the distribution is to add temperature to the distances that go into the softmax function (Holtzman et al., 2019). We calculate the probability of non-parametric component in Equation 2 where τ is the softmax temperature. In general, the higher the temperature, the less "peaky" the distribution becomes. We experiment with both the 5% as well as the full datastore using different temperatures ranging from 0 to 3 at 0.1 intervals. The results are shown in Figure 4a and Figure 4b respectively.

We see that the default temperature $\tau = 1$ does not always result in the best-interpolated perplexity and tuning the softmax temperature is desirable for all sizes of datastore. The lesson learned here is that tuning the softmax temperature for the kNN distribution is crucial for getting optimal results from each setting. Only coincidentally, a temperature of 1.0 was close to optimal in the original settings of kNN-LM, which hid the importance of this hyperparameter. Even at the optimal temperature of each setting, "real mask, real score" underperforms "FAISS mask, real score". This is consistent with the counter-intuitive phenomenon in Section 5.1. There are also differences between different datastore sizes. With the full datastore, using "real score" outperforms "FAISS score" given the same "FAISS mask". However, the opposite is true when using the 5% datastore. This suggests that as the datastore size grows, using accu-

rate distance values are better than the approximate ones. The smaller gap between using "real score" and "FAISS score" in both datastore settings shows that the main contributor to the improvements is using approximate nearest neighbors ("FAISS mask") rather than using approximate distance values ("FAISS score").

These results emphasize the effect of approximation discussed in Section 5.1, because comparing the small datastore with only 5% with the original datastore, we see that a small datastore means a small training set for the kNN "model" and it thus it benefits more from this regularization, both by using the FAISS mask and FAISS score (at optimal temperature settings). Surprisingly, one of the important ingredients in kNN-LM seems to be *approximate* kNN search, which likely prevents overfitting to the datastore created from the same training set. We further analyze this unexpected result in Appendix D, where we find that longer words and words that appear in many different contexts have slightly better results with approximate nearest neighbors.

Consistently with our findings, He et al. (2021) found that dimensionality reduction using PCA on the datastore vectors (from 1024 to 512 dimensions) improves the perplexity of the original kNN-LM from 16.46 to 16.25, which can be explained by our findings as PCA may provide another source of approximation that contributes to regularization. Notably, similar effects, where an approximation component leads to better generalization, have been reported in other NLP tasks as well, and are sometimes referred to as "beneficial search bias", when modeling errors cause the highest-scoring solution to be incorrect: for example, Meister et al. (2020b) suggest that "quite surprisingly, beam search often returns better results than exact inference due to beneficial search bias for NLP tasks"; Stahlberg & Byrne (2019) also conclude that "vanilla NMT in its current form requires just the right amount of beam search errors, which, from a modeling perspective, is a highly unsatisfactory conclusion indeed, as the model often prefers an empty translation".

6. Probably Wrong Hypotheses for Why kNN-LM Works

The results in the previous sections are the result of extensive analysis and experimentation, in which we also tested a number of hypotheses that did *not* turn out to have a significant effect. Additional details of these hypotheses are detailed in Appendix E, and we hope that they may provide ideas for future improvements of retrieval-based LMs.

Ensemble of Distance Metrics We hypothesized that the ensemble of two distance metrics: the standard inner product distance (which the LM uses) and the L2 distance (which the kNN component uses), is the key to the improvement. However, we found that similar gains can be achieved using the inner-product metric for the retrieved kNN (Appendix E.1).

Ensembling of Two Models We hypothesized that the kNN component merely provides another model for ensembling. The improvement from kNN-LM is *purely* due to the ensembling effect of simply different models. However, we found that kNN-LM's improvement is orthogonal to ensembling with a different base LM (Appendix E.5).

Sparsification The mask-to-k$(\cdot)$ used by kNN retrieval induces sparsity in the distribution over the vocabulary, due to a small k (typically 1024) compared to the size of the vocabulary V (33K in our experiments and 260K in the original setting). We hypothesized that kNN-LM increases the probability of the top-k entries while taking "probability mass" from the long tail of unlikely word types. However, we could not gain any benefits solely from sparsifying the output probability of a standard LM and interpolating it with the original LM (Appendix E.2).

Stolen Probabilities The *stolen probabilities* effect (Demeter et al., 2020) refers to the situation where the output embeddings of an LM are learned such that some words are geometrically placed *inside* the convex hull that is formed by other word embeddings and can thus never be "selected" as the argmax word. We hypothesized that kNN-LM solves the stolen probabilities problem by allowing to assign the highest probability to *any* word, given a test context that is close enough to that word's datastore key. However, we found that *none* of the vectors in our embedding matrix and in the original embedding matrix of Khandelwal et al. (2020b) is located in the convex hull of the others, which is consistent with the findings of Grivas et al. (2022) (Appendix E.4).

Memorization We hypothesized that the kNN component simply provides memorization of the training set. However, we could not improve a standard LM by interpolating its probability with another standard LM that was further trained to overfit the training set (Appendix E.6).

Soft Labels We hypothesized that kNN-LM's improvement lies in reducing the "over-correction" error when training with 1-hot labels, as hypothesized by Yang et al. (2022), and that retrieving neighbors is not important. If only "soft labels" are the key, we could hypothetically improve the performance of another fresh LM with the same model architecture but trained with the soft labels from the base LM, instead of from kNN-LM. This separates the effect of "soft labeling" from the additional guidance provided by kNN. However, this did not help at all (Appendix E.7).

Optimizing Interpolated Loss We hypothesized that the standard LM cross-entropy training loss does not emphasize the examples where base LM performs badly which could benefit from kNN, and directly optimizing the interpolated loss of standard LM and a separate trainable softmax layer could be a better alternative. However, we could not gain

any benefits by training an additional softmax layer together with a base LM using the interpolated loss (Appendix E.8).

7. Conclusion

In this paper, we investigate why kNN-LM improves perplexity, even when retrieving examples from the same training data that the base LM was trained on. By proposing and testing various hypotheses and performing extensive ablation studies, we find that the key to kNN-LM's success is threefold: (1) Ensembling different input representations – the feedforward layer output and the attention layer output – can recover 55% of the performance, even without retrieval. (2) One of the most unexpected discoveries is that using *approximate* nearest neighbor search allows kNN-LMs to generalize better than exact nearest neighbor search, possibly due to a regularization effect. (3) Tuning the softmax temperature for the kNN distribution is crucial to adjust the standard LM output distribution with the distribution created by the retrieved neighbors' distances. These findings are orthogonal to Drozdov et al. (2022) where they discovered kNN-LM works especially well when there is a large n-gram overlap between the training and the test set.

We performed extensive experiments which ruled out other hypotheses as to why kNN-LMs work, such as overparameterization, sparsification, overfitting, ensembling of distance metrics, etc. We believe that this work unlocks a variety of exciting research directions for efficient kNN-LM alternatives in addition to existing improvement models (Zhong et al., 2022). For example, exploring methods that replace the kNN component with trainable parameters and achieve comparable results without the latency burden of kNN-LM.

Acknowledgement

We thank Ramesh Nallapati, Sudipta Sengupta, Dan Roth, Daniel Fried, Xiaosen Zheng, Urvashi Khandelwal, Danqi Chen, and Andrew Drozdov for the helpful discussions and feedback. This project was supported by a gift from AWS AI. Frank F. Xu is supported by the IBM Ph.D. Fellowship.

References

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.

Ba, J. L., Kiros, J. R., and Hinton, G. E. Layer normalization. *arXiv preprint arXiv:1607.06450*, 2016.

Baevski, A. and Auli, M. Adaptive input representations for neural language modeling. *arXiv preprint arXiv:1809.10853*, 2018.

Bengio, Y., Ducharme, R., Vincent, P., and Jauvin, C. A neural probabilistic language model. *Journal of machine learning research*, 3(Feb):1137–1155, 2003.

Borgeaud, S., Mensch, A., Hoffmann, J., Cai, T., Rutherford, E., Millican, K., Van Den Driessche, G. B., Lespiau, J.-B., Damoc, B., Clark, A., et al. Improving language models by retrieving from trillions of tokens. In *International conference on machine learning*, pp. 2206–2240. PMLR, 2022.

Brown, T. B., Mann, B., Ryder, N., Subbiah, M., Kaplan, J., Dhariwal, P., Neelakantan, A., Shyam, P., Sastry, G., Askell, A., et al. Language models are few-shot learners. *arXiv preprint arXiv:2005.14165*, 2020.

Clauset, A., Shalizi, C. R., and Newman, M. E. Power-law distributions in empirical data. *SIAM review*, 51(4): 661–703, 2009.

Demeter, D., Kimmel, G., and Downey, D. Stolen probability: A structural weakness of neural language models. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pp. 2191–2197, 2020.

Devlin, J., Chang, M.-W., Lee, K., and Toutanova, K. BERT: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.

Drozdov, A., Wang, S., Rahimi, R., McCallum, A., Zamani, H., and Iyyer, M. You can't pick your neighbors, or can you? when and how to rely on retrieval in the kNN-LM. In *Findings of the Association for Computational Linguistics: EMNLP 2022*, pp. 2997–3007, Abu Dhabi, United Arab Emirates, December 2022. Association for Computational Linguistics. URL https://aclanthology.org/2022.findings-emnlp.218.

Grave, E., Cissé, M., and Joulin, A. Unbounded cache model for online language modeling with open vocabulary. *arXiv preprint arXiv:1711.02604*, 2017.

Grivas, A., Bogoychev, N., and Lopez, A. Low-rank softmax can have unargmaxable classes in theory but rarely in practice. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 6738–6758, 2022.

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.

He, J., Berg-Kirkpatrick, T., and Neubig, G. Learning sparse prototypes for text generation. *arXiv preprint arXiv:2006.16336*, 2020.

He, J., Neubig, G., and Berg-Kirkpatrick, T. Efficient nearest neighbor language models. *arXiv preprint arXiv:2109.04212*, 2021.

Hinton, G., Vinyals, O., Dean, J., et al. Distilling the knowledge in a neural network. *arXiv preprint arXiv:1503.02531*, 2(7), 2015.

Holtzman, A., Buys, J., Du, L., Forbes, M., and Choi, Y. The curious case of neural text degeneration. *arXiv preprint arXiv:1904.09751*, 2019.

Johnson, J., Douze, M., and Jégou, H. Billion-scale similarity search with GPUs. *IEEE Transactions on Big Data*, 7 (3):535–547, 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.

Khandelwal, U., Fan, A., Jurafsky, D., Zettlemoyer, L., and Lewis, M. Nearest neighbor machine translation. *arXiv preprint arXiv:2010.00710*, 2020a.

Khandelwal, U., Levy, O., Jurafsky, D., Zettlemoyer, L., and Lewis, M. Generalization through Memorization: Nearest Neighbor Language Models. In *International Conference on Learning Representations (ICLR)*, 2020b.

Meister, C., Salesky, E., and Cotterell, R. Generalized entropy regularization or: There's nothing special about label smoothing. *arXiv preprint arXiv:2005.00820*, 2020a.

Meister, C., Vieira, T., and Cotterell, R. Best-first beam search. *Transactions of the Association for Computational Linguistics*, 8:795–809, 2020b.

Merity, S., Xiong, C., Bradbury, J., and Socher, R. Pointer sentinel mixture models. *arXiv preprint arXiv:1609.07843*, 2016.

Merity, S., Keskar, N. S., and Socher, R. Regularizing and optimizing LSTM language models. In *Proceedings of ICLR*, 2018.

Ney, H., Essen, U., and Kneser, R. On structuring probabilistic dependences in stochastic language modelling. *Computer Speech & Language*, 8(1):1–38, 1994.

Pereyra, G., Tucker, G., Chorowski, J., Kaiser, Ł., and Hinton, G. Regularizing neural networks by penalizing confident output distributions. *arXiv preprint arXiv:1701.06548*, 2017.

Radford, A., Wu, J., Child, R., Luan, D., Amodei, D., Sutskever, I., et al. Language models are unsupervised multitask learners. *OpenAI blog*, 1(8):9, 2019.

Sanh, V., Debut, L., Chaumond, J., and Wolf, T. Distilbert, a distilled version of bert: smaller, faster, cheaper and lighter. *arXiv preprint arXiv:1910.01108*, 2019.

Sennrich, R., Haddow, B., and Birch, A. Neural machine translation of rare words with subword units. *arXiv preprint arXiv:1508.07909*, 2015.

Stahlberg, F. and Byrne, B. On nmt search errors and model errors: Cat got your tongue? *arXiv preprint arXiv:1908.10090*, 2019.

Szegedy, C., Vanhoucke, V., Ioffe, S., Shlens, J., and Wojna, Z. Rethinking the inception architecture for computer vision. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 2818–2826, 2016.

Wang, D., Fan, K., Chen, B., and Xiong, D. Efficient cluster-based k-nearest-neighbor machine translation. *ArXiv*, abs/2204.06175, 2022.

Yang, Z., Dai, Z., Salakhutdinov, R., and Cohen, W. W. Breaking the softmax bottleneck: A high-rank rnn language model. *arXiv preprint arXiv:1711.03953*, 2017.

Yang, Z., Sun, R., and Wan, X. Nearest neighbor knowledge distillation for neural machine translation. In *Proceedings of the 2022 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pp. 5546–5556, Seattle, United States, July 2022. Association for Computational Linguistics. doi: 10.18653/v1/2022.naacl-main.406. URL https://aclanthology.org/2022.naacl-main.406.

Zhong, Z., Lei, T., and Chen, D. Training language models with memory augmentation. In *Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing*, pp. 5657–5673, Abu Dhabi, United Arab Emirates, December 2022. Association for Computational Linguistics. URL https://aclanthology.org/2022.emnlp-main.382.

Zhou, S., Alon, U., Xu, F. F., JIang, Z., and Neubig, G. Doccoder: Generating code by retrieving and reading docs. *arXiv preprint arXiv:2207.05987*, 2022.

A. kNN-LM Generalization to Other LMs

	#params	Base LM PPL	kNN-LM PPL	Absolute PPL Gain
Ours	268M	21.75	19.17	2.58
Distilled-GPT2	82M	18.25	14.84	3.41
GPT2-small	117M	14.84	12.55	2.29
GPT2-medium	345M	11.55	10.37	1.18
GPT2-large	774M	10.56	9.76	0.80

Table 4: Performance of kNN-LM applied to other pretrained language models of different sizes.

To test the generalizability of kNN-LM, we follow the same experimental setup as used in Section 3. We select several pretrained models from the GPT2 family (Radford et al., 2019) of various parameter counts, plus a distilled version of GPT2, DistillGPT2. (Sanh et al., 2019) We take the pretrained model checkpoint, build the datastore and evaluate on the Wikitext-103 dataset splits. The results are shown in Table 4. We see that kNN-LMs has good generalizability on other models. It improves the perplexity of all the base LMs tested. However, the larger the model is, and usually the better the base LM's perplexity is, the less gain can be achieved from adding kNN. Note that our model is trained from scratch on Wikitext-103 dataset and thus even with a relatively large model size, the perplexity and perplexity gain from adding kNN is still less than models with pretraining. Without loss of generalizability, we will use our own trained-from-scratch model as the base LM in the following sections for ablation study.

B. Detailed Results for Increasing the Softmax Capacity

h_{ds}	N_{ds}	$\otimes$	+#params	PPL	Interp.	Oracle
-	-	-	0	21.750	-	-
att	Big	IP	$N_{ds} \times D$	∞	19.095	14.077
att	1x	IP	$V \times D$	22.584	20.353	16.954
att	2x	IP	$2V \times D$	21.903	20.529	17.432
att	3x	IP	$3V \times D$	22.434	20.395	17.132
att	4x	IP	$4V \times D$	21.936	20.521	17.423
att	5x	IP	$5V \times D$	22.025	20.643	17.560
att	6x	IP	$6V \times D$	21.972	20.519	17.422
att	9x	IP	$9V \times D$	22.084	20.696	17.631
ffn	Big	IP	$N_{ds} \times D$	∞	21.101	16.254
ffn	1x	IP	$V \times D$	20.920	20.694	18.772
ffn	2x	IP	$2V \times D$	20.889	20.646	18.701
ffn	3x	IP	$3V \times D$	20.829	20.603	18.717
ffn	4x	IP	$4V \times D$	20.769	20.629	18.876
ffn	5x	IP	$5V \times D$	20.720	20.594	18.878
ffn	6x	IP	$6V \times D$	20.726	20.599	18.902
ffn	9x	IP	$9V \times D$	20.687	20.567	18.887

Table 5: Performance comparison of kNN baselines and models with learnable embeddings of increasing size as W_{ds} datastore alternative. h_{ds} is either attention layer output (att) or feedforward layer output (ffn).

C. Alternative Methods for Increasing Softmax Capacity

C.1. Adaptive Increasing Embedding Size

We hypothesize that different word types have different difficulties for the language model to predict. For those words that appear very frequently, they may appear in many different contexts. As a result, instead of adding equal number of additional embeddings to each word type, we propose to adaptively increase the number of embeddings for word types based on word frequency, or total training loss for the word. Based on the intuition of Zipf's law (Clauset et al., 2009), we assign $1 + \log_b f_v$ for each word type $v \in V$, based on either the frequency or the total training loss of the word, f_v. The b is a hyperparameter that could be tuned. To ensure fair comparison, we tune b so that for each experiment the total number

of embeddings matches: $\sum_{v \in V} 1 + \log_b f_v = nV$. The results are shown in Table 6. We see that although nice on paper, given the same number of total embeddings, adaptively increasing the number of embeddings assigned for each word type does not make a significant difference in the final perplexity, when compared with the models that use equal number of embeddings for each word type.

	h_{ds}	N_{ds}	$\otimes$	+#params	PPL	Interp.	Oracle
Base LM	-	-	-	0	21.750	-	-
KNN	att	Big	L2	$N_{ds} \times D$	∞	19.174	14.230
KNN	att	Big	IP	$N_{ds} \times D$	∞	19.095	14.077
Equal Per Word	att	3x	IP	$3V \times D$	22.434	20.395	17.132
Loss Weighted	att	3x	IP	$3V \times D$	21.948	20.440	17.303
Freq. Weighted	att	3x	IP	$3V \times D$	22.507	20.387	17.105
KNN	ffn	Big	L2	$N_{ds} \times D$	∞	20.734	15.594
KNN	ffn	Big	IP	$N_{ds} \times D$	∞	21.101	16.254
Equal Per Word	ffn	3x	IP	$3V \times D$	20.829	20.603	18.717
Loss Weighted	ffn	3x	IP	$3V \times D$	20.764	20.659	18.978
Freq. Weighted	ffn	3x	IP	$3V \times D$	20.757	20.572	18.782

Table 6: Performance comparison of kNN baselines and several configurations that adaptively increase the embedding size with training loss or word frequency.

C.2. Mixture of Softmaxes

(Yang et al., 2017) proposes a solution to the problem using a Mixture of Softmax (MoS) to produce more linearly independent probability distributions of words given different contexts. Suppose that there are a total of R mixture components. MoS first uses R linear layers with weight w_r to transform the current query context vector h_{ds} into $w_r h_{ds}$. With a shared word embedding matrix W_{sm}, we calculate each softmax component's probability distribution with $\text{softmax}(W_{sm} \cdot w_r h_{ds})$. The mixture distribution is then given by:

$$P_{MoS} = \sum_{r}^{R} \pi_{r,h_{ds}} \text{softmax}(W_{sm} \cdot w_r h_{ds}) \tag{3}$$

The prior weights are calculated using another linear layer with weight w_π, as $\pi_{r,h_{ds}} = \text{softmax}(w_\pi h_{ds})$. The softmax ensures that $\sum_r^R \pi_{r,h_{ds}} = 1$. Comparing the MoS with the first term in Equation 2, $M\text{softmax}(\text{mask-to-k}(W_{ds} \otimes h_{ds}))$, we see that there are some connections between the two. MoS eliminates the mask-to-k$(\cdot)$ operation, and replaces the single softmax across a very large vector (size of datastore), into multiple smaller softmaxes, each across only a vector of the size of vocabulary. As a result, the huge W_{ds} is replaced by several linear layers to project the word embedding matrix. Now the first term becomes:

$$M(\oplus_r^R \text{softmax}(W_{sm} \cdot w_r h_{ds})) \tag{4}$$

$$M_{ir} = \pi_{r,h_{ds}}, \forall i \leq V \tag{5}$$

where $\oplus$ represents the vector concatenation operation, and the aggregation matrix M now contains the mixture weights for each softmax being concatenated. We perform experiments with a varying number of mixtures (R), different definitions h_{ds}, and whether to finetune the output word embeddings W_{sm}. We allow finetuning the word embedding when we use attention layer output as context vector, since the word embedding matrix is trained with feedforward layer output originally. The results for this formulation are shown in Table 7. MoS models on its own increase the performance of the language model marginally. When compared with Table 5, we find that these models are worse than those that simply increases the number of embeddings. This is expected because MoS has fewer added parameters compared to those, as it only requires several additional linear projection layers for the embeddings.

C.3. Clustering Datastore

Opposite to training the word embeddings of an increased size, we also consider ways to compress the datastore down to a similar-sized embedding matrix for softmax computation. The intuition is that the datastore contains redundant context vectors, and thus compression could make the datastore smaller without sacrificing too much performance gain. (He et al.,

	h_{ds}	R	$\otimes$	+#params	PPL	Interp.	Oracle
Base LM	-	-	-	0	21.750	-	-
KNN	att	-	L2	$N_{ds} \times D$	∞	19.174	14.230
KNN	att	-	IP	$N_{ds} \times D$	∞	19.095	14.077
KNN	ffn	-	L2	$N_{ds} \times D$	∞	20.734	15.594
KNN	ffn	-	IP	$N_{ds} \times D$	∞	21.101	16.254
Ft. MoS+embed	att	2	IP	$VD + 2D^2 + 2D$	21.986	20.720	17.573
Ft. MoS+embed	att	3	IP	$VD + 3D^2 + 3D$	22.106	20.779	17.609
Ft. MoS Only	att	2	IP	$2D^2 + 2D$	22.552	21.011	17.796
Ft. MoS Only	att	3	IP	$3D^2 + 3D$	22.573	21.024	17.812
Ft. MoS Only	ffn	2	IP	$2D^2 + 2D$	21.351	21.338	20.258
Ft. MoS Only	ffn	3	IP	$3D^2 + 3D$	21.495	21.460	20.322
Ft. MoS Only	ffn	4	IP	$4D^2 + 4D$	21.321	21.321	20.396
Ft. MoS Only	ffn	5	IP	$5D^2 + 5D$	21.371	21.367	20.406

Table 7: Performance comparison of kNN baselines and several MoS configurations. R is the number of mixtures.

2021) shows that we can safely compress the datastore by clustering to 50% of the original size without losing performance. We test this idea further by clustering the entire datastore into a size that could fit in GPU memory (e.g. $2V$, $3V$) and thus could be easily finetuned further and use the resulting centroids to replace W_{ds}. Within each cluster, there will be a distribution of different words with contexts, and we use the frequency of words within each cluster to calculate the aggregation matrix M in Equation 2. This would have the added benefit of "multi-sense" embedding, which allows similar meanings to be clustered to form a new "meta word" while the same word with different meanings would form different "meta words". A notable example is bank, shore, and financial institution. However, this does not work, mostly because of the high compression loss after clustering and the imbalanced distribution of word types among each cluster.

D. Which Words Benefit from Approximation?

To further understand the unexpected results when using the different kNN approximate retrieval settings in Section 5.1 and Section 5.2, we analyze on a token level, based on how many times each ground truth token's probability in the evaluation set are helped by each kNN setting. It means that for each ground truth token in the evaluation, we count the times when the kNN distribution is higher than the base LM distribution P_{LM}, i.e., $P_{kNN} > P_{LM}$.

Since we found previously that approximate kNN provides an additional performance boost compared to ground truth kNN, we thus compare "real mask, real score" versus "FAISS mask, real score" in this analysis. To prevent outliers, we filter out words with less than 10 occurrences in the evaluation set. For each setting, we calculate the percentage of occurrences in the evaluation set where each token in the vocabulary where the kNN module achieves a better probability than base LM. We then plot the absolute difference between the percentages of the two settings, with respect to various possible attributes of the token that achieves better probability using each setting.

Figure 5 shows that the longer the token is, which usually suggests proper nouns and harder and less common words in English, are better with approximate neighbors than ground truth ones, and vice versa. We hypothesize that this is due to longer words are more prone to overfitting in kNN-LM and thus using approximate kNN provides an effect similar to smoothing and regularization.

We also compare words that could appear in more diverse contexts with words that co-occur with few distinct contexts. To measure how diverse the contexts of each word in the vocabulary is, we calculate both the forward and backward bigram entropy for each word in the evaluation set that has more than 10 occurrences. The bigram entropy is a simple yet good indicator of context diversity for a given word, as used in Kneser–Ney smoothing (Ney et al., 1994). We calculate both the forward and backward bigram entropy for each word w as follows, where w_{after} and w_{before} represent the word after and before the given word w.

$$H_{\text{forward}}(w) = -\sum_{w_{\text{after}}} p(w_{\text{after}}|w) \log p(w_{\text{after}}|w) \tag{6}$$

$$H_{\text{backward}}(w) = -\sum_{w_{\text{before}}} p(w_{\text{before}}|w) \log p(w_{\text{before}}|w) \tag{7}$$

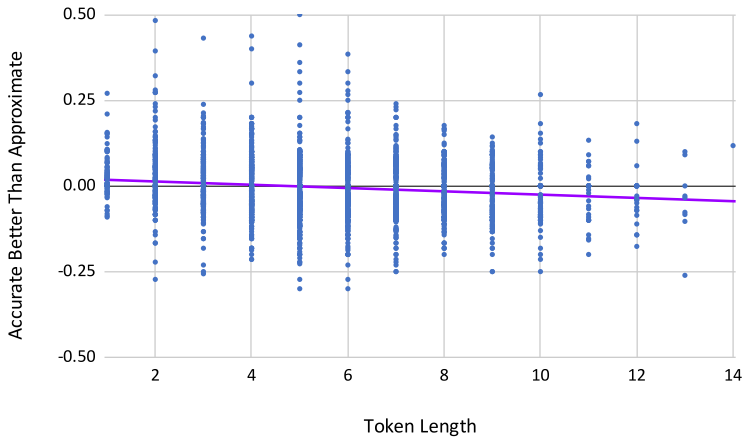


Figure 5: The effect of the token character length on how much accurate nearest neighbors are better than approximate FAISS neighbors. Negative values mean worse. The trend line of the scatter points is shown.

Forward and backward entropy represents how diverse the context after and before the given word is. Intuitively, bigram entropy is supposed to indicate words that can appear in lots of different contexts. The higher the entropy of a word, the more diverse its context is, and vice versa. For example, words like "Francisco" would have a low entropy because it mostly comes after "San".

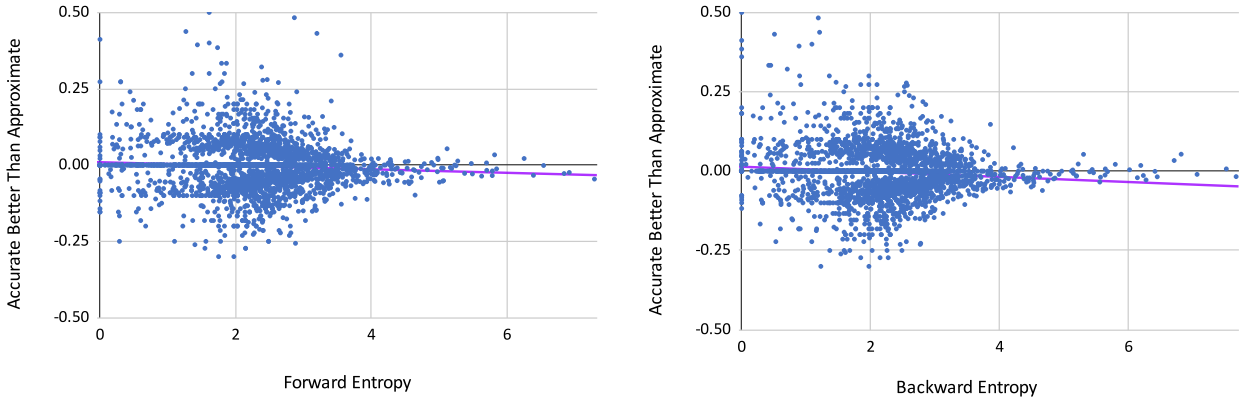


Figure 6: The effect of the forward and backward entropy of words on how accurate nearest neighbors are better than approximate FAISS neighbors. Negative values mean worse. The trend line of the scatter points are shown.

The comparison is shown in Figure 6. We see that the higher the entropy in both forward and backward cases, the better using approximate nearest neighbor search becomes. This suggests that words that appear in many different contexts are better off with an approximate kNN, and "easy-to-predict" examples such as "Jersey" and "Fransisco" is better with accurate kNN, possibly because these examples are less prone to overfitting errors and thus requires less regularization from approximation.

E. Failed Hypotheses

E.1. Distance Metric

We hypothesize that the key to kNN-LM's performance gain is the ensemble of two distance metrics: the standard dot product distance (which the LM uses) with the L2 distance (which the kNN component uses as $\otimes$). We tried to replace the kNN component with a component that just takes the tokens retrieved by the kNN search and returns their L2 distance to the LM output word embeddings: $W_{sm} \otimes h_{ds}$ instead of $W_{ds} \otimes h_{ds}$, where $\otimes$ represents the negative L2 distance. We tried

this with both variants of h_{ds}, attention layer output, and feedforward layer output. None of these helped.

E.2. Sparsification

In Equation 2, mask-to-k$(\cdot)$ used by kNN retrieval induces sparsity in the distribution over the vocabulary, due to a small k compared to the number of vocabulary V. We hypothesize that the in kNN-LM, the kNN distribution is sparse, practically increasing the probability of the top-k entries. The kNN distribution has up to 1024 entries that are non-zero, concentrating more probability mass over the most likely tokens. This effect is similar to the redistribution of probability mass for text generation in (Holtzman et al., 2019). We test this hypothesis only by taking top 32, 64, 128, 512, or 1024 tokens in the parametric LM probability and zeroing out the probabilities of the rest of the tokens. To compensate, we experiment with different softmax temperatures and then interpolate with the parametric LM probability. This isolates the effect of the datastore and retrieval at all, and this does not help at all, suggesting that sparsification of the output probability alone is not enough.

Another attempt is to hypothesize that the key in kNN-LM is that it selects "which tokens to include" in the kNN distribution, and not their distances. The intuition behind is that maybe the selection of the top tokens according to the kNN search is better than that from the dot-product distance between the language model's output vector and all the vocabulary embeddings. We perform experiments similar to the previous attempt, sparsifying the output probability with the tokens retrieved by the kNN search (but ignoring the distances provided by the kNN search) rather than the top k tokens of the LM, with and without removing duplicates. In the best case, they manage to reduce the perplexity by 0.5 (whereas kNN-LM reduces by nearly 2).

E.3. Location within Context Window

Supposedly, words in the beginning of the "context window" of the transformer at test time have less contextual information than words toward the end of context window.

We hypothesized that maybe the base LM performs worse in one of these (beginning vs. end of the context window), and maybe kNN-LM provides a higher improvement in one of these. We measured the per-token test perplexity with respect to the location of each token in the context window. However, we did not find any significant correlation between the performance of the base LM and the location, and no significant correlation between the difference between kNN-LM and the base LM and the location.

We also hypothesized that maybe the beginning of every Wikipedia article is more "predictable", and the text becomes more difficult to predict as the article goes into details. However, we also did not find any correlation with the location of the word within the *document* it appears in.

E.4. Stolen Probabilities

The *stolen probabilities* effect (Demeter et al., 2020) refers to the situation where the output embeddings of an LM are learned such that some words are geometrically placed *inside* the convex hull that is formed by other word embeddings. Since language models generate a score for every output word by computing the dot product of a hidden state with all word embeddings, Demeter et al. (2020) prove that in such a case, it is impossible for words inside the convex hull to be predicted as the LM's most probable word (the "argmax").

We hypothesized that kNN-LM solves the stolen probabilities problem by allowing to assign the highest probability to *any* word, given a test hidden state that is close enough to that word's datastore key. Nevertheless, as shown by Grivas et al. (2022), although this problem might happen in small RNN-based language models, in modern transformers it rarely happens in practice. Using the code of Grivas et al. (2022), we checked the embeddings matrix of our model and of the checkpoint provided by Khandelwal et al. (2020b). Indeed, we found that in both models – *no word is un-argmaxable*.

E.5. Are kNN-LM Just Ensembling?

Our hypothesis is that kNN component only provides another model for ensembling. The interpolation process is basically an ensemble model. Technically it is unsurprising that kNN-LM will have the benefit from ensembling, but we perform experiments to see how it compares to other ensembling. We trained another language model with the same architecture as the base LM we used throughout the experiments, with some variants having more than one embedding vector for each

word (similar to Section 4.2). We interpolate the models with the original base LM, and the results are shown in Table 8. We see that even just ensembling the base LM with another identical model, but trained with a different random seed, provides a huge performance boost, both on interpreted perplexity and on oracle perplexity.

Prev. Layers	h_{ds}	N_{ds}	$\otimes$	+#params	PPL	Interp.	Oracle
same	-	-	-	0	21.750	-	-
same	att	Big	L2	$N_{ds} \times D$	∞	19.174	14.230
same	att	Big	IP	$N_{ds} \times D$	∞	19.095	14.077
same	ffn	Big	L2	$N_{ds} \times D$	∞	20.734	15.594
same	ffn	Big	IP	$N_{ds} \times D$	∞	21.101	16.254
diff	ffn	1x	IP	$F + V \times D$	21.569	18.941	14.980
diff	ffn	2x	IP	$F + 2V \times D$	21.914	18.948	14.885
diff	ffn	3x	IP	$F + 3V \times D$	22.206	18.981	14.853

Table 8: Performance comparison of kNN baselines and models with different size output embeddings re-trained from scratch.

However, just because ensembling two LMs of the same architecture provides better performance than interpolating the base LM with kNN does not necessarily suggest that kNN's performance improvement can be fully replaced by model ensembling. In other words, we are interested in whether the kNN performance improvements are orthogonal to that of model ensembling. To test this, we compare the performance of the ensemble of K multiple LMs versus the ensemble of $K-1$ multiple LMs plus the kNN component. The comparison is fair because we have the same number of models in the ensemble, and the only difference is whether the kNN component is included. The results are shown in Figure 7. For the "LM" series, each point is K LMs ensemble, and for the "kNN" series, each point is $K-1$ LMs plus kNN. We see that even at 4-ensemble, the ensemble that contain kNN as a component still have a considerable edge over the 4-ensemble that contain just LMs.

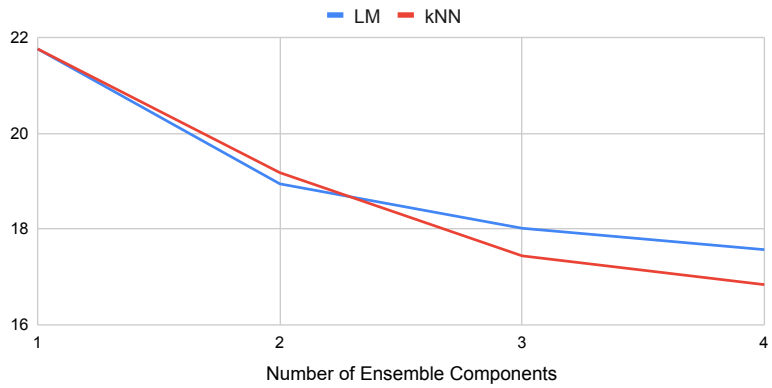


Figure 7: Ensembling effect comparison, between multiple base LMs and multiple base LMs plus kNN component.

E.6. Are kNN-LM Just Overfitting?

Since kNN-LM improves perplexity even with the same training dataset as datastore, we are curious if kNN-LM works by only "memorizing" the training data. The hypothesis is that the datastore and the kNN search are trying to memorize the training data. In other words, the parametric LM is under-fitting some tokens. The intuition behind this is that the kNN component retrieves examples directly from the training set. What if we could retrieve the same examples using an overfitted LM? We took the trained LM, removed the dropout, and continued training until almost perfect fit (very small training loss). We then interpolated the overfitted transformer with the original LM. The results are shown in Table 9. F represents the number of parameters in the base LM, minus the output embedding matrix. We see that overfitting can provide very little help after interpolation. Looking at the oracle performance, we think that the overfitted model memorizes some rare contexts and tokens in the training set where it could be useful during evaluation. However, the overfitting hurts the performance on other tokens too much so that even interpolation is not able to balance the performance.

	Prev. Layers	h_{ds}	N_{ds}	$\otimes$	+#params	PPL	Interp.	Oracle
Base LM	same	-	-	-	0	21.750	-	-
kNN-LM	same	att	Big	L2	$N_{ds} \times D$	∞	19.174	14.230
kNN-LM	same	att	Big	IP	$N_{ds} \times D$	∞	19.095	14.077
kNN-LM	same	ffn	Big	L2	$N_{ds} \times D$	∞	20.734	15.594
kNN-LM	same	ffn	Big	IP	$N_{ds} \times D$	∞	21.101	16.254
Overfit@92	diff	ffn	V	IP	$F + V \times D$	1702.806	21.732	17.764
Overfit@129	diff	ffn	V	IP	$F + V \times D$	8966.508	21.733	17.814

Table 9: Performance comparison of several baselines with two overfitted models, at 92 and 129 additional epochs.

E.7. Are kNN-LM Just Soft-Label Training?

(Yang et al., 2022) claims that using "soft labels" during training is the key to kNN's success, that interpolates the ground truth labels with kNN-LM model outputs, effectively "distilling" kNN-LM. It is based on the hypothesis that the room for kNN-LM's improvement over base LM lies in the "over-correction" when training with a 1-hot labels. This is related to the effect from label smoothing methods (Szegedy et al., 2016; Pereyra et al., 2017; Meister et al., 2020a). However, we believe that this explanation is not satisfactory. If the key is training with soft-labels, why do these soft labels must be provided specifically by a kNN search? If soft labels were the key, then soft-label training where the labels come from the base LM itself should have worked as well. To separate the effect of soft labeling from the kNN's additional guidance, we train another LM with the same model architecture as the base LM, with the soft labels from the base LM. This teacher-student training is to distill the knowledge from the base LM (Hinton et al., 2015). We find that by just training with "soft labels" from the base LM to alleviate the alleged "over-correction" problem is not the key, as this does not help with the interpolated perplexity at all. This suggests that even with the same training data, kNN still provides valuable additional guidance.

E.8. Are kNN-LM Just Training to Optimize Interpolated Loss?

In Section 4.2, we discover that using over-parameterization with standard LM training loss does not further close the gap towards kNN-LM. This suggests that some regularization term may be needed during training to make the multiple embeddings not converge to the same vector, rendering over-parameterization useless.

From Table 2, we see that a better interpolated perplexity may not require a very low perplexity when measured only with the extra input representation. However, we still use a standard LM loss to only train the additional embedding matrix, that directly minimizes the perplexity using only the extra input representation. This discrepancy between training and the evaluation with interpolation suggests that training with an alternative loss function that interpolates the base LM's output with the output using the extra input representation may be beneficial.

To test the hypothesis that standard LM training loss do not emphasize the examples where base LM performs badly, we train the extra model's parameter W_{ds}, with interpolated loss L:

$$L = \text{CrossEntropy}(\lambda \text{softmax}(W_{ds} \cdot h_{ds}) + (1 - \lambda)\text{softmax}(W_{sm} \cdot h_{sm}), y) \tag{8}$$

y represents the ground truth label for each context. We only learn the parameter W_{ds} while freezing all other parameters, similar to all other experiments. We choose $\lambda = 0.25$ as it is the best hyper-parameter for kNN-LM experiments and our goal for this training is to mimic the loss of kNN-LM after interpolation. This training loss effectively assigns a higher value to the training examples where the base LM's loss is high, suggesting the need for the extra W_{ds} to help with these hard cases. However, for either "att" for "ffn" for h_{ds}, either V or $3V$ for the number of embeddings in W_{ds}, we are unable to achieve a better perplexity than just the base LM. This suggests that, while nice on paper, the interpolated loss optimization process is not trivial.