A Related Work (cont.)

Generative Retrieval Document retrieval traditionally involved training a 2-tower model which mapped both queries and documents to the same high-dimensional vector space, followed by performing an ANN or MIPS for the query over all the documents to return the closest ones. This technique presents some disadvantages like having a large embedding table [22, 23]. Generative retrieval is a recently proposed technique that aims to fix some of the issues of the traditional approach by producing token by token either the title, name, or the document id string of the document. Cao et al. [5] proposed GENRE for entity retrieval, which used a transformer-based architecture to return, token-by-token, the name of the entity referenced to in a given query. Tay et al. [34] proposed DSI for document retrieval, which was the first system to assign structured semantic DocIDs to each document. Then given a query, the model autoregressively returned the DocID of the document token-by-token. The DSI work marks a paradigm shift in IR to generative retrieval approaches and is the first successful application of an end-to-end Transformer for retrieval applications. Subsequently, Lee et al. [23] show that generative document retrieval is useful even in the multi-hop setting, where a complex query cannot be answered directly by a single document, and hence their model generates intermediate queries, in a chain-of-thought manner, to ultimately generate the output for the complex query. Wang et al. [37] supplement the hierarchical k-means clustering based semantic DocIDs of Tay et al. [34] by proposing a new decoder architecture that specifically takes into account the prefixes in semantic DocIDs. In CGR [22], the authors propose a way to take advantage of both the bi-encoder technique and the generative retrieval technique, by allowing the decoder, of their encoder-decoder-based model, to learn separate *contextualized* embeddings which store information about documents intrinsically. To the best of our knowledge, we are the first to use generative Semantic IDs created using an auto-encoder (RQ-VAE [40, 21]) for retrieval models.

Vector Quantization. We refer to Vector Quantization as the process of converting a high-dimensional vector into a low-dimensional tuple of codewords. One of the most straightforward techniques uses hierarchical clustering, such as the one used in [34], where clusters created in a particular iteration are further partitioned into sub-clusters in the next iteration. An alternative popular approach is Vector-Quantized Variational AutoEncoder (VQ-VAE), which was introduced in [35] as a way to encode natural images into a sequence of codes. The technique works by first passing the input vector (or image) through an encoder that reduces the dimensionality. The smaller dimensional vector is partitioned and each partition is quantized separately, thus resulting in a sequence of codes: one code per partition. These codes are then used by a decoder to recreate the original vector (or image).

RQ-VAE [40, 21] applies residual quantization to the output of the encoder of VQ-VAE to achieve a lower reconstruction error. We discuss this technique in more detail in Subsection 3.1. Locality Sensitive Hashing (LSH) [14, 13] is a popular technique used for clustering and approximate nearest neighbor search. The particular version that we use in this paper for clustering is SimHash [2], which uses random hyperplanes to create binary vectors which serve as hashes of the items. Because it has low computational complexity and is scalable [13], we use this as a baseline technique for vector quantization.

B Baselines

Below we briefly describe each of the baselines with which we compare TIGER

- GRU4Rec [11] is the first RNN-based approach that uses a customized GRU for the sequential recommendation task.
- Caser [33] uses a CNN architecture for capturing high-order Markov Chains by applying horizontal and vertical convolutional operations for sequential recommendation.
- HGN [25]: Hierarchical Gating Network captures the long-term and short-term user interests via a new gating architecture.
- SASRec [17]: Self-Attentive Sequential Recommendation uses a causal mask Transformer to model a user's sequential interactions.
- BERT4Rec [32]: BERT4Rec addresses the limitations of uni-directional architectures by using a bi-directional self-attention Transformer for the recommendation task.

- FDSA [42]: Feature-level Deeper Self-Attention Network incorporates item features in addition to the item embeddings as part of the input sequence in the Transformers.
- S^3-Rec [44]: Self-Supervised Learning for Sequential Recommendation proposes pre-training a bi-directional Transformer on self-supervision tasks to improve the sequential recommendation.
- P5 [8]: P5 is a recent method that uses a pretrained Large Language Model (LLM) to unify different recommendation tasks in a single model.

C Dataset Statistics

Table 6: Dataset statistics for the three real-world benchmarks.

Dataset	# Users	# Items	Sequence Length	
			Mean	Median
Beauty	22,363	12,101	8.87	6
Sports and Outdoors	35,598	18,357	8.32	6
Toys and Games	19,412	11,924	8.63	6

We use three public benchmarks from the Amazon Product Reviews dataset [10], containing user reviews and item metadata from May 1996 to July 2014. We use three categories of the Amazon Product Reviews dataset for the sequential recommendation task: "Beauty", "Sports and Outdoors", and "Toys and Games". Table 6 summarizes the statistics of the datasets. We use users' review history to create item sequences sorted by timestamp and filter out users with less than 5 reviews. Following the standard evaluation protocol [17, 8], we use the leave-one-out strategy for evaluation. For each item sequence, the last item is used for testing, the item before the last is used for validation, and the rest is used for training. During training, we limit the number of items in a user's history to 20.

D Modifications to the P5 data preprocessing

Table 7: Results for P5[8] with the standard pre-processing.

Methods	Sports and Outdoors				Beauty				Toys and Games			
	Recall@5	NDCG@5	Recall@10	NDCG@10	Recall@5	NDCG@5	Recall@10	NDCG@10	Recall@5	NDCG@5	Recall@10	NDCG@10
P5	0.0061	0.0041	0.0095	0.0052	0.0163	0.0107	0.0254	0.0136	0.0070	0.0050	0.0121	0.0066
P5-ours	0.0107	0.0076	0.01458	0.0088	0.035	0.025	0.048	0.0298	0.018	0.013	0.0235	0.015

The P5 source code [5] pre-processes the Amazon dataset to first create sessions for each user, containing chronologically ordered list of items that the user reviewed. After creating these sessions, the original item IDs from the dataset are remapped to integers $1, 2, 3, \ldots$ [6]. Hence, the first item in the first session gets an id of '1', the second item, if not seen before, gets an id of '2', and so on. Notably, this pre-processing scheme is applied before creating training and testing splits. This results in the creation of a sequential dataset where many sequences are of the form $a, a+1, a+2, \ldots$. Given that P5 uses Sentence Piece tokenizer [30] (See Section 4.1 in [8]), the test and train items in a user session may share the sub-word and can lead to information leakage during inference.

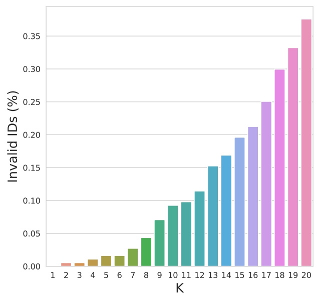
To resolve the leakage issue, instead of assigning sequentially increasing integer ids to items, we assigned random integer IDs, and then created splits for training and evaluation. The rest of the code for P5 was kept identical to the source code provided in the paper. The results for this dataset are reported in Table 7 as the row 'P5'. We also implemented a version of P5 ourselves from scratch, and train the model on only sequential recommendation task. The results for our implementation are depicted as 'P5-ours'. We were also able to verify in our P5 implementation that using consecutive integer sequences for the item IDs helped us get equivalent or better metrics than those reported in P5.

Table 8: The effect of providing user information to the recommender system

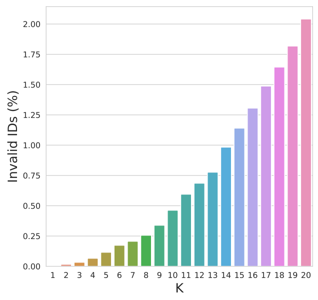
Recall@5	NDCG@5	Recall@10	NDCG@10	
No user information	0.04458	0.0302	0.06479	0.0367
With user id (reported in the paper)	0.0454	0.0321	0.0648	0.0384

Table 9: The mean and stand error of the metrics for different dataset (computed using 3 runs with different random seeds)

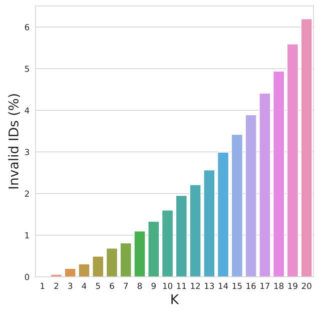
Datasets	Recall@5	NDCG@5	Recall@10	NDCG@10
Beauty	0.0441 ± 0.00069	0.0309 ± 0.00062	0.0642 ± 0.00092	0.0374 ± 0.00061
Sports and Outdoors	0.0278 ± 0.00069	0.0189 ± 0.00043	0.0419 ± 0.0010	0.0234 ± 0.00048
Toys and Games	0.0518 ± 0.00064	0.0375 ± 0.00039	0.0698 ± 0.0013	0.0433 ± 0.00047



(a) Sports and Outdoors



(b) Beauty



(c) Toys and Games

Figure 6: Percentage of invalid IDs when generating Semantic IDs using Beam search for various values of K. As shown, $\sim 0.3\% - 6\%$ of the IDs are invalid when retrieving the top-20 items.

E Discussion

Effects of Semantic ID length and codebook size. We tried varying the Semantic ID length and codebook size, such as having an ID consisting of 6 codewords each from a codebook of size 64. We noticed that the recommendation metrics for TIGER were robust to these changes. However, note that the input sequence length increases with longer IDs (i.e., more codewords per item ID), which makes the computation more expensive for our transformer-based sequence-to-sequence model.

Scalability. To test the scalability of Semantic IDs, we ran the following experiment: We combined all the three datasets and generated Semantic IDs for the entire set of items from the three datasets. Then, we used these Semantic IDs for the recommendation task on the Beauty dataset. We compare the results from this experiment with the original experiment where the Semantic IDs are generated only from the Beauty dataset. The results are provided in Table 10. We see that there is only a small decrease in performance here.

Inference cost. Despite the remarkable success of our model on the sequential recommendation task, we note that our model can be more computationally expensive than ANN-based models during inference due to the use of beam search for autoregressive decoding. We emphasize that optimizing the computational efficiency of TIGER was not the main objective of this work. Instead, our work opens up a new area of research: *Recommender Systems based on Generative Retrieval*. As part of future work, we will consider ways to make the model smaller or explore other ways of improving the inference efficiency.

Memory cost of lookup tables. We maintain two lookup hash tables for TIGER: an Item ID to Semantic ID table and a Semantic ID to Item ID table. Note that both of these tables are generated only once and then frozen: they are generated after the training of the RQ-VAE-based Semantic ID generation model, and after that, they are frozen for the training of the sequence-to-sequence

[5] https://github.com/jeykigung/P5

[6] https://github.com/jeykigung/P5/blob/0aaa3fd8366bb6e708c8b70708291f2b0ae90c82/preprocess/data_preprocess_amazon.ipynb

Table 10: Testing scalability by generating the Semantic IDs on the combined dataset vs generating the Semantic IDs on only the Beauty dataset.

	Recall@5	NDCG@5	Recall@10	NDCG@10
Semantic ID [Combined datasets]	0.04355	0.3047	0.06314	0.03676
Semantic ID [Amazon Beauty]	0.0454	0.0321	0.0648	0.0384

transformer model. Each Semantic ID consists of a tuple of 4 integers, each of which are stored in 8 bits, hence totalling to 32 bits per item. Each item is represented by an Item ID, stored as a 32 bit integer. Thus, the size of each lookup table will be of the order of $64N$ bits, where N is the number of items in the dataset.

Memory cost of embedding tables. TIGER uses much smaller embedding tables compared to traditional recommender systems. This is because where traditional recommender systems store an embedding for each item, TIGER only stores an embedding for each semantic codeword. In our experiments, we used 4 codewords each of cardinality 256 for Semantic ID representation, resulting in 1024 (256$\times$4) embeddings. For traditional recommender systems, the number of embeddings is N, where N is the number of items in the dataset. In our experiments, N ranged from 10K to 20K depending on the dataset. Hence, the memory cost of TIGER's embedding table is $1024d$, where d is the dimension of the embedding, whereas the memory cost for embedding lookup tables in traditional recommendation systems is Nd.