CLUSTERING EMBEDDING TABLES, WITHOUT FIRST LEARNING THEM.

Anonymous authors
Paper under double-blind review

ABSTRACT

Machine learning systems use embedding tables to work with categorical features. These tables may get extremely large in modern recommendation systems, and various methods have been suggested to fit them in memory.

Product- and Residual Vector Quantization (Gray & Neuhoff (1998)) are some of the most successful methods for table compression. They function by substituting table rows with references to "codewords" picked by k-means clustering. Unfortunately, this means that they must first know the table before compressing it, thus they can only save memory at inference time, not training time. Recent work has employed hashing-based approaches to minimize memory usage during training, however the compression obtained is poorer than that achieved by "post-training" quantization.

We demonstrate that combining hashing and clustering based algorithms provides the best of both worlds. By first training a hashing-based "sketch", then clustering it, and then training the clustered quantization, our method may achieve compression ratios close to those of post-training quantization with the training time memory reductions of hashing-based methods. We prove that this technique works rigorously in the least-square setting.

1 INTRODUCTION

Machine learning can model a variety of data types, including continuous, sparse, and sequential features. Categorical features are especially noteworthy since they necessitate an "embedding" of a (typically vast) vocabulary into a smaller linear space in order to facilitate further calculations. IDs of different types, such as user IDs, post IDs on social networks, video IDs, or IP addresses in recommendation systems, are examples of such features. Natural Language Processing is another prominent use for embeddings (usually word embeddings such as Mikolov et al., 2013), however the vocabulary can be significantly reduced by considering subwords or "Byte pair encodings". In recommendation systems like Matrix Factorization or DLRM (see fig. 2) it is typically not possible to reduce the vocabulary this way, and embedding tables can end up requiring hundreds of gigabytes of GPU memory (Naumov et al., 2019). This in effect forces models to be split across may GPUs which creates a communication bottleneck during training and inference.

The traditional approach has been to just hash down to a manageable size using the Hashing Trick, (Weinberger et al., 2009), and accepting that unrelated IDs may wind up with the same representation. This has a natural worsening influence on the model's output quality.

Another option is to quantize the embedding tables. Typically, this entails rounding each individual parameter to 4 or 8 bits. There are, however, quantization methods that work in many dimensions at the same time, such as Product Quantization and Residual Vector Quantization. See Gray & Neuhoff (1998) for a survey of quantization methods. These methods rely on *clustering* (typically k-means) in various ways to create optimal approximations to the vectors being compressed. For example, vectors representing "red", "orange" and "blue" may be stored as simple "dark orange" and "blue" with the two first concepts pointing to the same average embedding. It was recently shown in the theoretical literature that such clustering is indeed necessary for Optimal Euclidean Compression (Indyk & Wagner, 2022). While quantization may generally significantly reduce memory consumption, it has one major drawback: the model is only quantized *after* training, thus memory utilization

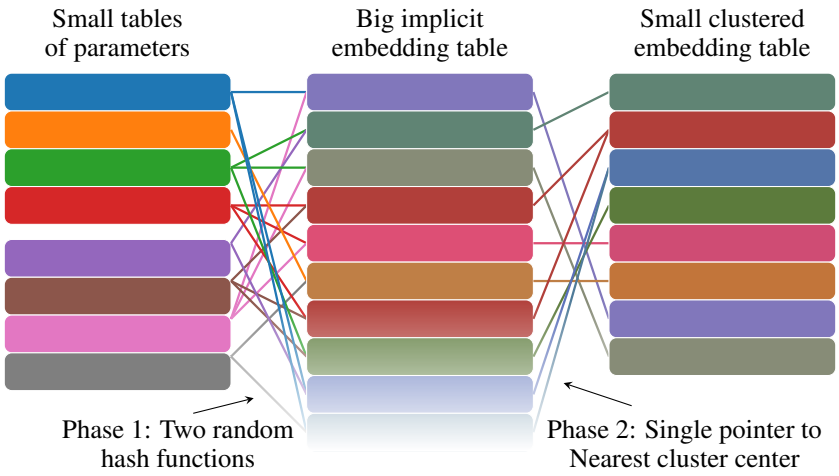


Figure 1: **Cluster QR method in 3 steps: (1)** First each ID is hashed to n_h (here $= 2$) smaller tables, similar to Tito Svenstrup et al. (2017) and the value is taken to be the mean of the target vectors. **(2)** At some point during training, the implicit (large) embedding table is sub-sampled and clustered. **(3)** This leaves a new small table in which each of the IDs are represented by just one vector; and IDs that are "semantically similar" are represented by the same vector.

during training is unaffected. (Note: While it is common to do some "finetuning" of the model after, say, product quantization, the method remains primarily a "post-training" approach.)

Recent authors have considered more advanced ways to use hashing to overcome this problem: Tito Svenstrup et al. (2017); Shi et al. (2020); Desai et al. (2022); Yin et al. (2021); Kang et al. (2021). The common thread has been using multiple hash functions which allow features to get different representations with high probability, while still mapping into a smaller shared table of parameters. While these methods can work better than the hashing trick in some cases, they still fundamentally mix up completely unrelated concepts in a way that introduces large amounts of noise into the remaining machine learning model.

Clearly there is an essential difference between "post-training" compression methods like product quantization which *can utilize similarities between concepts* and "during training" techniques based on hashing, which are forced to randomly mix up concepts. This paper's key contribution is to bridge that gap: *We present a novel compression approach that combines hashing and clustering while retaining the benefits of both methods.* We simultaneously take advantage of similarities between concepts to be encoded and compress data during training, not just at inference.

Our method, depicted in fig. 1, has a very "two phase" feel to it. We initially explored a more "continuous" strategy that interspersed clustering and re-clustering with training. It just so happened that experimentally doing clustering just once produced comparable quality results, while also being easier and more efficient. In spirit, our effort can be likened to methods like RigL (Evci et al., 2020), which discovers the wiring of a sparse neural network during training rather than afterward, as is generally done. Our work can also be seen as a form of "Online Product Quantization", though prior work in this area had a very different focus.[1]

In summary, our main contributions are:

1. We show that it is possible to use clustering at training time, even though we don't have memory enough to learn or even store the concepts be clustered.

   Technically this means we can learn a mapping from the vocabulary $V = [n] = \{1, \ldots, n\}$ to $[k]$ where $n \gg k$ similarly to what would have been the result of learning a vector $v_i \in \mathbb{R}^d$ for every $i \in V$, and then applying k-means clustering. *Why is this hard?* Imagine

[1] In particular the paper of Xu et al. (2018) focused on updating code words already assigned to concept. Our goal is more ambitious: We want to learn *which* concepts to group together without ever knowing the "true" embedding for the concepts.

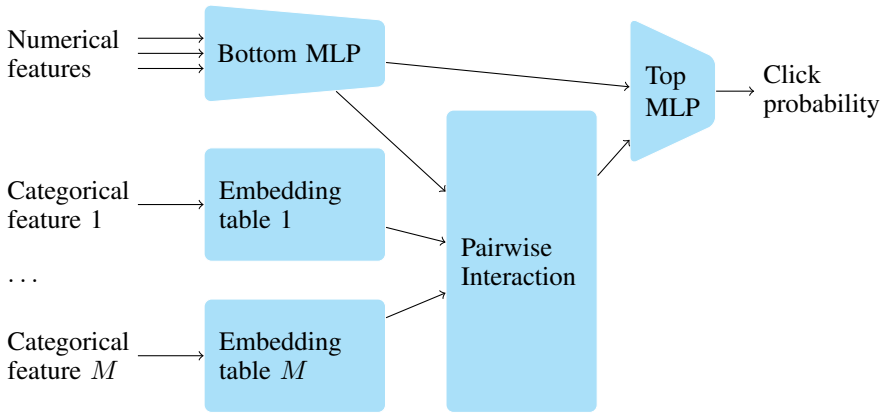


Figure 2: **Typical Recommendation System Architecture:** The DLRM model Naumov et al. (2019) embeds each categorical feature separately and combines the resulting vectors with pairwise dot products. Other architectures use different interaction layers or a single embedding table for all categorical features, but the central role of the embedding table is universal.

at some point in the learning process you decide concepts i and j in V should map to the same vector in $[k]$. For the remaining duration of the training, you can never distinguish the two IDs again, and thus any decision you make is permanent. We circumvent this problem using a sketch of v_i and v_j that can be stored and updated cheaply.

2. Our method bridges the state of the art for quantizing "post-training" (clustering based methods) and during training (hashing methods).

3. We show experimentally that our technique allows compressing the embedding tables of the Deep Learning Recommendation System (DLRM) to less than half of the best previously suggested methods.

1.1 Outline

In section 2 we give a brief overview of the theory and methods used for table compression. In section 3 we introduce our new method, Clustered Compositional Embedding (or CQR for short), in detail. In section 5 we run a comprehensive set of experiments on the Criteo datasets. Finally, in appendix A we sketch some of all the ideas that were tried before we ultimately ended up at the CQR method.

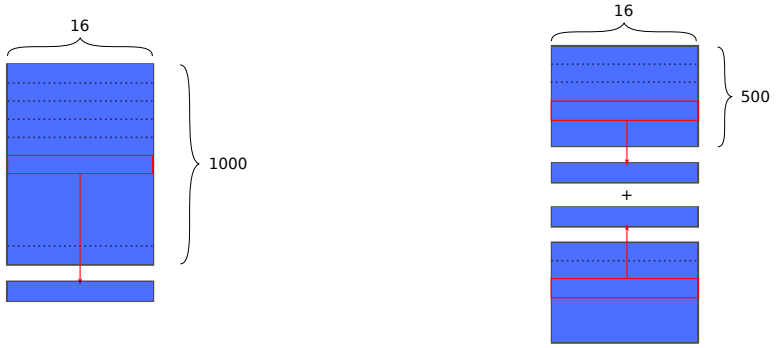
2 Background and Related Work

We show how most previous work on table compression can be seen in the theoretical framework of linear dimensionality reduction. This allows us to generalize many techniques and guide our intuition on how to choose the quality and number of hash functions in the system.

We omit standard common preprocessing tricks, such as weighting entities by frequency, using separate tables and precision for common vs uncommon elements, or completely pruning rare entities. We also don't cover the background of "post training" quantization, but refer to the survey, Gray & Neuhoff (1998). Finally, we keep things reasonably heuristic, but for a deep theoretical understanding of metric compression, we recommend Indyk & Wagner (2022).

2.1 Embedding Tables as Linear Maps

An embedding table is typically expressed as a tall skinny matrix $T \in \mathbb{R}^{d_1 \times d_2}$, where each ID $i \in [d_1]$ is mapped to the i-th row, $T[i]$. Alternatively, i can be expressed as a one-hot row-vector $e_i \in \{0,1\}^{d_1}$ in which case $T[i] = e_i T \in \mathbb{R}^{d_2}$.



(a) **The Hashing Trick:** Each ID is hashed to one location in a table (here with 1000 rows) and it is assigned the embedding vector stored at the location. Many IDs are likely to share the same vector.

(b) **Hash Embeddings:** Each ID is hashed to two rows, one per each table, and its embedding vector is assigned to be the sum of those two vectors. Here, we use two separate tables unlike in Tito Svenstrup et al. (2017).

Figure 3: The Hashing Trick and Hash Embeddings shown side by side with an equal amount of parameters.

Most previous work in the area of table compression is based on the idea of sketching. Here we introduce a sparse matrix $H \in \{0,1\}^{d_1 \times k}$ and a dense matrix $M \in \mathbb{R}^{k \times d_2}$, where $k << d_1$. The idea is to take $T = HM$, so $T[i] = e_i HM$. Because H is sparse it requires very little memory. And because $e_i H$ is also sparse, the computation $(e_i H)M$ takes only constant time.

In this framework we can also express most other approaches to training-time table compression:

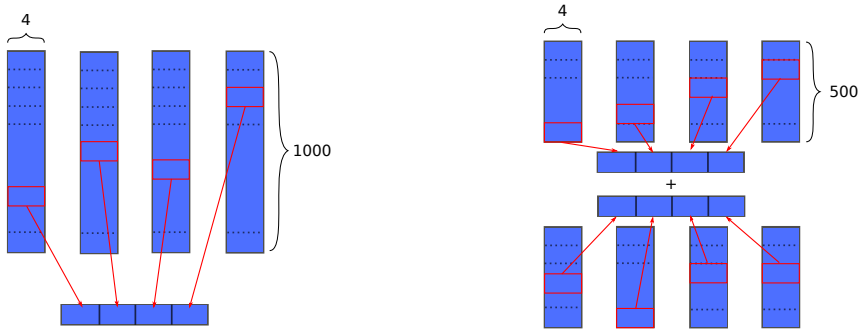
The Hashing Trick (Weinberger et al., 2009) is normally described by a hash function $h : [d_1] \to [k]$, such that i is given the vector $M[h(i)]$, where M is a table with just $k \ll d_1$ rows. Alternatively we can think of this trick as multiplying e_i with a random matrix $H \in \{0,1\}^{k \times d_1}$ which has exactly one 1 in each column. Then the embedding of i is $M[h(i)] = e_i HM$, where $HM \in \mathbb{R}^{d_1 \times d_2}$.

Hash Embeddings (Tito Svenstrup et al., 2017) map each ID $i \in V$ to the sum of a few table rows. For example, if i is mapped to two rows, then its embedding vector is $v = M[h_1(i)] + M[h_2(i)]$. Using the notation of $H \in \{0,1\}^{m \times n}$, one can check that this corresponds to each column having exactly two 1s. In the paper, the authors also consider weighted combinations, which simply means that the non-zero entries of H can be some real numbers.

Compositional embeddings (or QR embeddings, Shi et al., 2020), define $h_1(i) = \lfloor i/p \rfloor$ and $h_2(i) = i \mod p$ for integer p, and then combines $T[h_1(i)]$ and $T[h_2(i)]$ in various ways. As mentioned by the authors, this choice is, however, not of great importance, and more general hash functions can also be used, which allows for more flexibility in the size and number of tables. Besides using sums, like Hash Embeddings, the authors also propose element-wise multiplication[2] and concatenation. Concatenation $[T[h_1(i)], T[h_2(i)]]$ can again be described with a matrix $H \in \{0,1\}^{d_1 \times k}$ where each column has exactly one 1 in the top half of H and one in the bottom half of H, as well as a block diagonal matrix T. While this restricts the variations in embedding matrices T that are allowed, we usually compensate by picking a larger m, so the difference in entropy is not much different from Hash Embeddings, and the practical results are very similar as well.

ROBE embeddings (Desai et al., 2022) are essentially compositional embeddings with concatenation as described above, but adds some more flexibility in the indexing, including the ability for chunks to "wrap around" in the embedding table. In our experiments ROBE was near indistinguishable from QR-concat.

[2]While combining vectors with element-wise multiplication is not a linear operation, from personal communication with the authors, it is unfortunately hard to train such embeddings in practice. Hence we focus on the two linear variants.



(a) **QR with concatenation:** In the hashing version of compositional embeddings (QR), each ID is hashed to a location in each of, say, 4 different tables. The four vectors stored there are concatenated into the final embedding vector. Given the large number of possible combinations (here 1000^4), it is unlikely that two IDs get assigned the exact same embedding vector, even if they may share each part with some other IDs.

(b) **QR hybrid:** We can combine the sum hashing method of Tito Svenstrup et al. (2017) with the concatenation method of Shi et al. (2020). Each ID then gets assigned a vector that is the concatenation of smaller sums. In itself this method is not particularly interesting, but it is an essential step towards the Cluster QR method we describe in this paper.

Figure 4: Compositional Embeddings with concatenation and the Hybrid method which we will introduce later on.

Deep Hashing Embeddings DHE (Kang et al., 2021) picks 1024 hash functions $h_1, \ldots, h_{1024} : [d_1] \to [-1, 1]$ and feed the vector $(h_1(i), \ldots, h_{1024}(i))$ into a multi-layer perceptron. While the idea of using an MLP to save memory at the cost of larger compute is novel and parts from the sketching framework, the first hashing step of DHE is just sketching with $H \in [-1, 1]^{d_1 \times 1024}$ a dense random matrix. While this is less efficient than a sparse matrix, it can still be applied efficiently to sparse inputs, e_i, and stored in small amounts of memory.

Unfortunately, in our experiments, DHE did not perform as well as the other methods, unless the MLP was taken to have just one layer, in which case it is just a linear transformation, $e_i HM$, and we end up with a more expensive version of Hash Embeddings. See fig. 7b for the details.

Tensor Train (Yin et al., 2021) doesn't use hashing, but like QR it splits the input in a deterministic way, that can be generalized to a random hash function if so inclined. Instead of adding or concatenating chunks, Tensor Train multiplies them together as matrices, which makes it not strictly a linear operation. However like DHE, the first step in reducing the input size is some kind of sketching.

Our algorithm is the first that deviates from random (or fixed) sketching as the first step of embedding. However we can still express the embedding process as $e_i HM$ for a sparse matrix H and a small dense matrix M. The difference is that *we learn H from the data* instead of using a random or fixed matrix. Experimentally and theoretically this allows better learning at the memory usage.

In section 3 we give the practical description of the algorithm, and in section 4 we analyze it mathematically in a simplified setting.

2.2 Other Related Work

Recent parallel work by Ghaemmaghami et al. (2022) presents a different way to learn a clustering without having to learn the exact embedding tables first. Instead of using hashing as a "pseudo table" for clustering to work on, they train a model with a lower dimensional table. They run a special clustering algorithm designed for low dimensional data on this table, and then retrain the model with the clustering known. It is interesting future work to better compare the strengths and weaknesses of our two methods against each other.

Other people have looked at "Learning CountSketch" such as Liu et al. (2020) who gave a formulation in the learning-based sketching paradigm proposed by Indyk et al. (2019). However, these algorithms require repeatedly retraining the model, which for big recommendation systems would be much too slow.

3 Clustered Compositional Embeddings (CQR)

Now we are ready to describe the CQR method. For illustration purposes, we use the setup of one of our experiments on the Kaggle dataset, where the embedding dimension is 16 and the number of parameters we train on is $16 \cdot 1000$.

Step 1: We start by following the QR hybrid method: For each $j = 1, \ldots, 4$, we begin with 4 main embedding tables $T_1, \ldots, T_4$ and 4 helper embedding tables $T'_1, \ldots, T'_4$, each of size 500×4. We also pick random hash functions $h_1, \ldots, h_4, h'_1, \ldots, h'_4 \colon [n] \to [500]$. Then the embedding vector of $i \in [n]$ is assigned to be

$$\begin{pmatrix} T_1[h_1(i)] & & T_2[h_2(i)] & & T_3[h_3(i)] & & T_4[h_4(i)] \\ + & , & + & , & + & , & + \\ T'_1[h'_1(i)] & & T'_2[h'_2(i)] & & T'_3[h'_3(i)] & & T'_4[h'_4(i)] \end{pmatrix}.$$

We then train the embedding tables for either half an epoch or a full epoch.

Step 2: We perform k-means clustering as follows. For each $j = 1, \ldots, 4$, following the Hash Embeddings method, we can re-materialize the vector $T_j[h_j(i)] + T'_j[h'_j(i)]$ for each $i \in [n]$. By picking a sub-sample $S \subseteq [n]$, we re-materialize the vectors for the IDs in S and cluster them to obtain a table T^c_j of 1000 cluster points.[3] With the cluster points, we then compute the index pointer function $h^c_j \colon [n] \to [1000]$ that sends each $i \in [n]$ to the nearest cluster point of $T_j[h_j(i)] + T'_j[h'_j(i)]$, the re-materialized vector of i.

Step 3: We modify the model following the QR concat method: We replace the original embedding tables and hash functions with the tables of cluster points and the index pointer functions, respectively. The embedding vector of each $i \in [n]$ is now assigned to

$$\Big(T^c_1[h^c_1(i)] \quad , \quad T^c_2[h^c_2(i)] \quad , \quad T^c_3[h^c_3(i)] \quad , \quad T^c_4[h^c_4(i)] \Big).$$

In other words, this is just the QR concat method with a set of specially initialized embedding tables and carefully chosen hash functions! The embedding tables are then trained for the remainder of the training duration.

We note that unlike the random hash functions used in Step 1, the index pointer functions cannot be stored efficiently. The index pointer functions can, however, be saved on the CPUs or, even better, be used as part of the data pre-processing process as no training is done on them.

4 Analysis

We consider a simplified model of the setup, where $X \in \mathbb{R}^{n \times d_1}$ and $Y \in \mathbb{R}^{n \times d_2}$, and we want to find the $T \in \mathbb{R}^{d_1 \times d_2}$ that minimizes $\|XT - Y\|_2^2$. In our setup $n > d_1 >> d_2$.

Obviously we can easily retrieve T from (X, Y) solving the least squares problem. Let's call this optimal solution $T^* \in \mathbb{R}^{d_1 \times d_2}$. However, we want to save memory, so we instead pick a random matrix $H_0 \in \mathbb{R}^{d_1 \times k}$ and find

$$M_0 = \operatorname*{arg\,min}_{M \in \mathbb{R}^{k \times d_2}} \|XHM - Y\|_2^2.$$

Then $T_1 = H_0 M_0$ is an approximation to T^*, in so far as H_0 preserves the column space of T^*, which is likely if $k > d_2$. Woodruff (2014) showed that indeed H can be a very sparse binary matrix.

The theoretical version of our algorithm can be run for multiple steps, and we show how the difference between T^* and the approximation increases exponentially with more steps.

[3] In k-means clustering, when $n \gg k$, only a subset of vectors is needed for finding the cluster points. In our experiments we sub-sample $256 \cdot k$ many vectors.

Generalized CQR: Repeat for $i \geq 1$:

$$T_i = H_{i-1} M_{i-1}$$
$$H_i = \text{sparse-approximation-of}(T_i) \in \mathbb{R}^{d_1 \times k}.$$
$$M_i = \arg\min_M \|X H_i M - Y\|_2^2 \in \mathbb{R}^{k \times d_2}.$$

We sparsify H_i using the assignment matrix from k-means, but it could also be done in other ways. The important part is that H_i contains a good approximation to T_i plus some random noise, which can be used to improve the approximation in the next step.

To show this, we study a version in which $H_i = [T_{i-1} \mid N(0,1)^{d_1 \times (k-d_2)}]$. That is, H_i is the previous T_i concatenated with a random normal matrix. We show that for this version of the algorithm

Theorem 1.

$$E[\|H_i M_i - T^*\|_F^2] \leq (1 - 1/d_1)^{i(k-d_2)} \|T^*\|_F^2.$$

This means that as $i \to \infty$ we can completely recover T^* starting from pure noise. See appendix B for the proof, as well as experiments with this algorithm and the Generalized CQR with different levels of sparsity.

5 Experiments

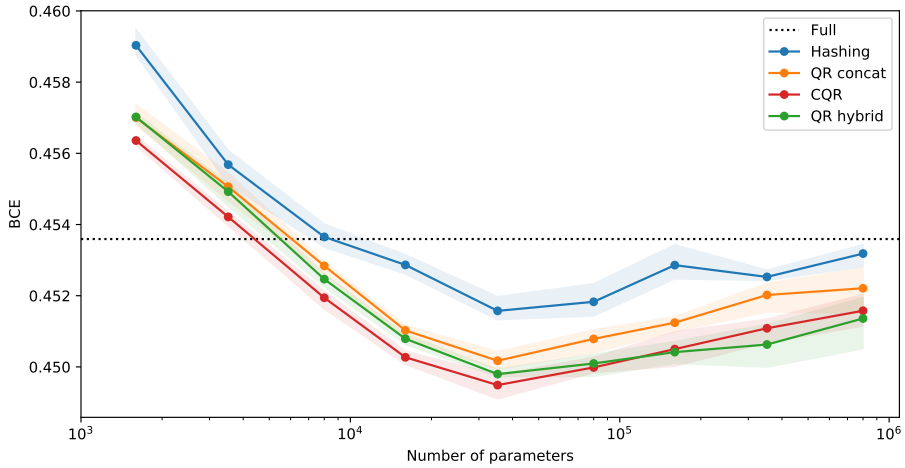


Figure 5: **Clustered QR outperforms Hashing and Compositional methods**. We trained DLRM on the Criteo dataset with different table compression algorithms and different table sizes for 10 epochs and picked the best validation loss. Our method was able to reach the baseline (trained for 1 epoch with $16 \cdot 10^7$ parameters) with just over 4000 total parameters. Meanwhile, the best previously published method (QR concat, Shi et al. (2020)) needed more than 6000. Our hybrid method (QR hybrid) could also slightly outperform QR concat.

The main experimental finding is shown in fig. 5: The CQR method allows training a model with the baseline Binary Cross Entropy (BCE) using a third of the parameters needed with the hashing trick, and 1.5 times less than with the QR concat method. If given the optimal amount of parameters and trained till convergence, it can give a measurably lower BCE.

In this section we explain the experimental setup we used, and show the results of various other experiments highlighting the strengths and weaknesses of the CQR method.

5.1 Experimental Setup

We follow the setup of the open-sourced DLRM code of Naumov et al. (2019).

Dataset. We use Criteo's Kaggle dataset and Criteo's Terabyte dataset of click logs. Both are public datasets and were used for benchmarking in DLRM (Naumov et al. (2019)), our backbone recommendation model. Each dataset has 13 dense features and 26 categorical features.

- The Kaggle dataset has about 45 million samples over 7 days, where each categorical feature has at most 10 million possible values.
- The Terabyte dataset has about 4 billion samples over 24 days. In our experiments, we sub-sampled one-eighth of the data and pre-hashed them such that each categorical feature has at most 10 million possible values.

We use data from the days prior to the last day as the training set. The data of the last data is split into a validation set and a test set.

Evaluation Metric. We use Binary Cross-Entropy (BCE) to evaluate the performance of the recommendation model. This metric is also used in Shi et al. (2020) for evaluating the performance of the QR methods. A small BCE means the prediction probability is close to the actual result.

Epochs. Recommendation systems are typically just trained for one epoch since they are prone to over-fitting. It has however been observed (e.g. by Desai et al. (2022)) that compressed embedding tables benefit from multi-epoch training. We thus give results both for 1 epoch training, 2 epoch training and "best of 10 epochs" (in fig. 5), making sure we get the actual best possible model performance.

Backbone Recommendation Models. We adopt the deep learning recommendation model (DLRM) from Naumov et al. (2019) as the backbone recommendation model; see fig. 2. The DLRM takes dense and sparse features as inputs, where a full embedding table (i.e., one with $|V|$ many rows) is used for each categorical feature. Since the model has an open-sourced implementation, we only have to modify the embedding table part of the code to implement our methods.

Number of runs of each model. On the Kaggle dataset, PQ inference is run once, while all other models are run 3 times. On the Terabyte dataset, each model is run once.

5.2 Baseline: Full embedding table

Full embedding table means that for each categorical feature, we train an embedding table of the size of the number of categorical values. Then a categorical value is mapped uniquely to a row in the table. For example, if a feature has 10 million possible values, then its embedding table would have 10 million row of embedding vectors. The embedding table is then trained for one epoch. However, using a full embedding table is usually not practical.

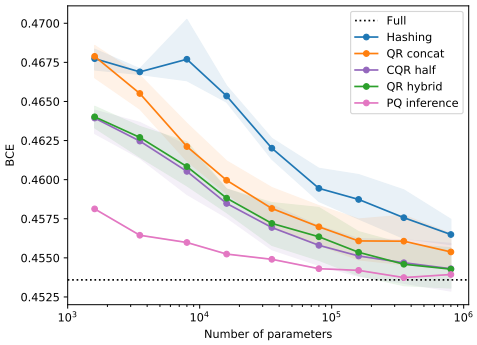
5.3 PQ Inference

As described in the introduction, the CQR method mimics Product Quantization (PQ) on the full embedding table. For that reason, PQ is a strong baseline which we compare with in fig. 6a and fig. 7a. As it turns out we are mostly able to outperform this baseline, which while surprising may be explained by our PQ not being "finetuned" after clustering. We tried training a full table, PQ clustering it and training this clustering for an epoch, but this approach immediately overfit and gave terrible results. Interestingly this suggests CQR works as a regularization method for PQ.

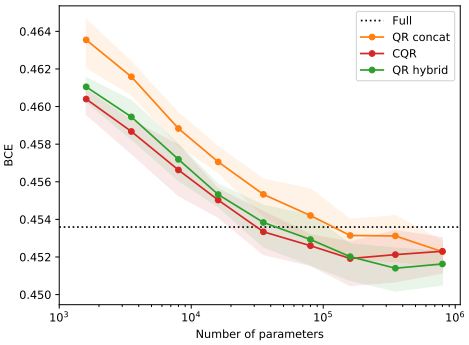
6 Conclusion

We have shown the feasibility of compressing embedding tables at training time using clustering. While there is still work to be done in expanding our theoretical understanding and testing the method in more situations, we believe this shows an exciting new paradigm for dynamic sparsity in neural networks and recommender systems in particular.

Some previous papers have made wildly different claims in terms of the possibility of compressing embedding tables. We believe fig. 6a, fig. 6b and fig. 5 help explain some of that confusion: With standard learning rates, it is not possible to improve over the DLRM baseline much, if at all, at 1 epoch of training. This matches the results of Naumov et al. (2019). However, if you train till

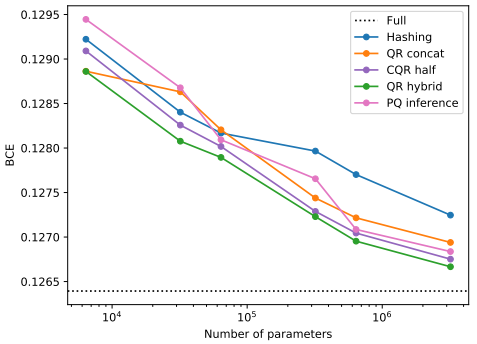


(a) **Kaggle dataset, 1 epoch**: Since CQR, as defined, clusters the table after the first epoch, we here suggest "CQR half" which clusters after only half an epoch. While this outperformed the previous methods (Hashing and QR concat) it was slightly worse than just keeping our own sketching method (QR hybrid) for the entire epoch.

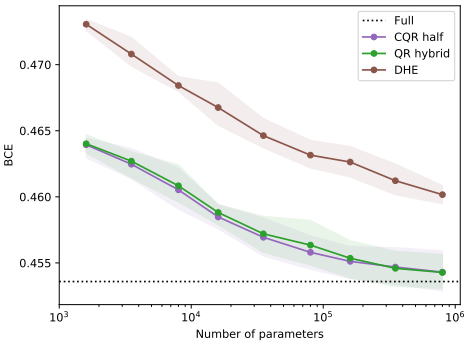


(b) **Kaggle dataset, 2 epoch**: Training for two epochs is enough for CQR to outperform QR hybrid as well as the methods from previous papers, though not by as much as in fig. 5 where we trained till convergence.

Figure 6: All methods trained for 1 or 2 epochs on Kaggle



(a) **Terabyte dataset, 1 epoch**: Like on the Kaggle dataset, running for just 1 epoch was not enough to show an improvement over our sketching method, QR hybrid, but enough to beat QR concat and the Hashing method.



(b) **DHE, Kaggle dataset, 1 epoch**: Using the suggested MLP setup from Kang et al. (2021) we were not able to get competitive performance when fixing the number of layers to match the desired number of parameters. This is surprising, so we have opted to not include DHE on the other plots.

Figure 7: Experiments on Terabyte dataset and with Deep Hashing Embeddings.

convergence, you can match the baseline with 1000x times fewer parameters than the default. Even if you just use the simple hashing trick. This matches the results of Desai et al. (2022).

In all experiments, except fig. 7a on the Terabyte dataset, our method, CQR, outperforms all other methods we tried. We believe it likely would be better on the Terabyte dataset as well, if given time to train until convergence.

Training to convergence is however not very common in practical recommendation systems. Our experiment in fig. 5 may offer an explanation why: Most methods overfit if given too large embedding tables(!) This is quite shocking, since the common knowledge is that bigger is always better and concepts should ideally have their own private embedding vectors. We believe these experiments call for more investigation into overfitting in DLRM style models.

REFERENCES

Andrew R Barron, Albert Cohen, Wolfgang Dahmen, and Ronald A DeVore. Approximation and learning by greedy algorithms. *The annals of statistics*, 36(1):64–94, 2008.

J Lawrence Carter and Mark N Wegman. Universal classes of hash functions. In *Proceedings of the ninth annual ACM symposium on Theory of computing*, pp. 106–112, 1977.

Moses Charikar, Kevin Chen, and Martin Farach-Colton. Finding frequent items in data streams. In *International Colloquium on Automata, Languages, and Programming*, pp. 693–703. Springer, 2002.

Michael B Cohen, TS Jayram, and Jelani Nelson. Simple analyses of the sparse johnson-lindenstrauss transform. In *1st Symposium on Simplicity in Algorithms (SOSA 2018)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2018.

Aditya Desai, Li Chou, and Anshumali Shrivastava. Random offset block embedding (robe) for compressed embedding tables in deep learning recommendation systems. *Proceedings of Machine Learning and Systems*, 4:762–778, 2022.

Martin Dietzfelbinger, Torben Hagerup, Jyrki Katajainen, and Martti Penttonen. A reliable randomized algorithm for the closest-pair problem. *Journal of Algorithms*, 25(1):19–51, 1997.

Utku Evci, Trevor Gale, Jacob Menick, Pablo Samuel Castro, and Erich Elsen. Rigging the lottery: Making all tickets winners. In *International Conference on Machine Learning*, pp. 2943–2952. PMLR, 2020.

Benjamin Ghaemmaghami, Mustafa Ozdal, Rakesh Komuravelli, Dmitriy Korchev, Dheevatsa Mudigere, Krishnakumar Nair, and Maxim Naumov. Learning to collide: Recommendation system model compression with learned hash functions. *arXiv preprint arXiv:2203.15837*, 2022.

Robert M. Gray and David L. Neuhoff. Quantization. *IEEE transactions on information theory*, 44 (6):2325–2383, 1998.

Piotr Indyk and Tal Wagner. Optimal (euclidean) metric compression. *SIAM Journal on Computing*, 51(3):467–491, 2022.

Piotr Indyk, Ali Vakilian, and Yang Yuan. Learning-based low-rank approximations. *Advances in Neural Information Processing Systems*, 32, 2019.

Jeff Johnson, Matthijs Douze, and Hervé Jégou. Billion-scale similarity search with GPUs. *IEEE Transactions on Big Data*, 7(3):535–547, 2019.

Wang-Cheng Kang, Derek Zhiyuan Cheng, Tiansheng Yao, Xinyang Yi, Ting Chen, Lichan Hong, and Ed H Chi. Learning to embed categorical features without embedding tables for recommendation. In *Proceedings of the 27th ACM SIGKDD Conference on Knowledge Discovery & Data Mining*, pp. 840–850, 2021.

Simin Liu, Tianrui Liu, Ali Vakilian, Yulin Wan, and David P Woodruff. Learning the positions in countsketch. *arXiv preprint arXiv:2007.09890*, 2020.

Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781*, 2013.

Maxim Naumov, Dheevatsa Mudigere, Hao-Jun Michael Shi, Jianyu Huang, Narayanan Sundaraman, Jongsoo Park, Xiaodong Wang, Udit Gupta, Carole-Jean Wu, Alisson G Azzolini, et al. Deep learning recommendation model for personalization and recommendation systems. *arXiv preprint arXiv:1906.00091*, 2019.

F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

Hao-Jun Michael Shi, Dheevatsa Mudigere, Maxim Naumov, and Jiyan Yang. Compositional embeddings using complementary partitions for memory-efficient recommendation systems. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pp. 165–175, 2020.

Dan Tito Svenstrup, Jonas Hansen, and Ole Winther. Hash embeddings for efficient word representations. *Advances in neural information processing systems*, 30, 2017.

Kilian Weinberger, Anirban Dasgupta, John Langford, Alex Smola, and Josh Attenberg. Feature hashing for large scale multitask learning. In *Proceedings of the 26th annual international conference on machine learning*, pp. 1113–1120, 2009.

David P. Woodruff. Sketching as a tool for numerical linear algebra. *Found. Trends Theor. Comput. Sci.*, 10(1-2):1–157, 2014. doi: 10.1561/0400000060. URL https://doi.org/10.1561/0400000060.

Donna Xu, Ivor W Tsang, and Ying Zhang. Online product quantization. *IEEE Transactions on Knowledge and Data Engineering*, 30(11):2185–2198, 2018.

Chunxing Yin, Bilge Acun, Xing Liu, and Carole-Jean Wu. Tt-rec: Tensor train compression for deep learning recommendation models. *CoRR*, abs/2101.11714, 2021. URL https://arxiv.org/abs/2101.11714.

REPRODUCIBILITY STATEMENT

The backbone recommendation model, DLRM by Naumov et al. (2019), has an open-sourced PyTorch implementation of the model available on github, with instructions on how to use them in the benchmarking folder. It also has instructions on how to download the public datasets. We follow very closely to their instructions, so reproducing the baseline result should be straightforward.

For the QR methods, we only need to change two functions in the code: `create_emb` and `apply_emb`. We suggest using a class for each QR method; see fig. 4. For the random hash function, one could use a universal hash function or `numpy.random.randint`.

For the CQR method, apart from using the classes for the QR concat method and the QR hybrid method, we also need to add a section in the code for k-means clustering after some training. We recommend using the open-sourced k-means clustering algorithm by Johnson et al. (2019), since it is much faster than Scikit-learn's k-means clustering algorithm (Pedregosa et al., 2011).

A WHAT DIDN'T WORK

Here are the ideas we tried but didn't work at the end.

Using multiple helper tables It is a natural idea use more than one helper table. However, in our experiments, the effect of having more helper tables is not apparent.

Circular clustering Based on the QR concat method, the circular clustering method would use information from other columns to do clustering. However, the resulting index pointer functions are too similar to each other, meaning that this method is essentially the hashing trick. We further discuss this issue in appendix C.

Continuous clustering We originally envisioned our methods in a tight loop between training and (re)clustering. It turned out that reducing the number of clusterings didn't impact performance, so we eventually reduced it all the way down to just one. In practical applications, with distribution shift over time, doing more clusterings may still be useful, as we discuss in section 4.

Changing the number of columns In general, increasing the number of columns leads to better results. However the marginal benefits quickly decrease, and as the number of hash functions grow, so does the training and inference time. We found that 4 columns / hash-functions was a good spot.

Residual vector quantization The CQR method combines Product Quantization (PQ) with the QR concat method. We tried combining Residual vector quantization (RVQ) with the Hash Embeddings method from Tito Svenstrup et al. (2017). This method does not perform significantly better than the Hash Embeddings method.

Seeding with PQ We first train a full embedding table for one epoch, and then do Product Quantization (PQ) on the table to obtain the index pointer functions.

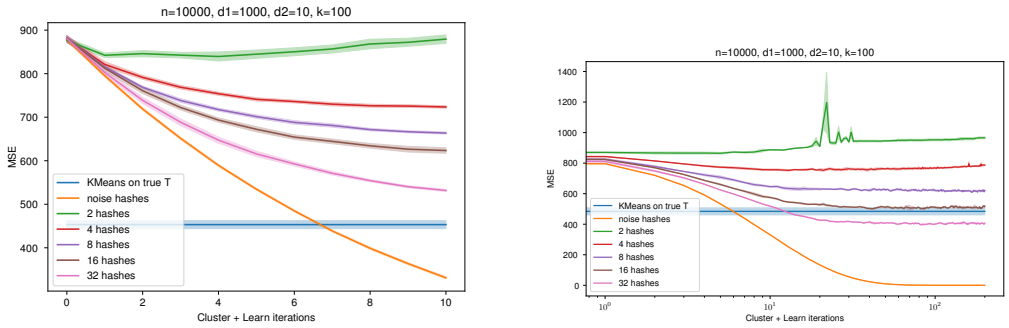
We then use the index pointer functions instead of random hash functions in the QR concat method. This method turned out performing badly: The training loss quickly diverges from the test loss after training on just a few batches of data.

Here are some variations of the CQR method:

Earlier clustering We currently have two versions of the CQR method: CQR half, where clustering happens at the middle of the first epoch, and CQR, where clustering happens at the end of the first epoch. We observe that when we cluster earlier, the result is slightly worse. Though in our case the CQR half method still outperforms the QR concat method.

More parameters before clustering The CQR method allows using two number of parameters, one in Step 1 where we follow the QR hybrid method to get a sketch, and one in Step 3 where we follow the QR concat method. We thought that by using more parameters at the beginning, we would be able to get a better set of index pointer tables. However, the experiment suggested that the training is faster but the terminal performance is not significantly better.

B Proof of the main theorem



(a) **Multi-step CQR with 10 steps**: All methods, dense as sparse, get nearly the same error after the zero steps. This means we haven't done any clustering, but simply run a more or less sparse Count Sketch. At 1 step all sparsities still do quite well, which is why the real CQR method proposed in this paper uses just 2 hash functions per column.

(b) **Multi-step CQR with 200 steps**: All sparse methods converge before reaching zero loss. This is expected, since the k-means baseline run on the true, optimal T^* matrix also doesn't achieve zero loss. Meanwhile, the dense method analyzed in this section does eventually achieve zero loss as predicted by the theorem.
Interestingly k-means stopped being an effective CQR sparsification algorithm at higher number of steps, and we here instead assign each ID to the nearest codewords from a random Gaussian codebook.

Figure 8: **Multi-step CQR on Least Squares**: We compare the Multi-step CQR method with sparse H (n hashes per row) with the dense model analyzed in theorem 1. We also show a baseline of first solving for the optimal T^* and then running k-means to reduce the space usage. As hoped, and expected, they all quickly improve with more iterations, though the more sparse version eventually converge.

Recall that $n \gg d_1 > k > d_2$ are positive integers, X is a matrix of size $n \times d_1$, and Y is a matrix of size $n \times d_2$. We want to find a matrix T^* of size $d_1 \times d_2$ such that $\|XT^* - Y\|_F$ is minimized. In particular, our method, as described in section 4, finds an optimal M_i at each step such that $\|XH_iM_i - Y\|_F$ is minimized, where $H_i = [T_{i-1}|N_i]$ and N_i is a random matrix where each column is uniformly sampled from the unit sphere, and estimates T^* with $T_i := H_iM_i$.

Recall the theorem:

Theorem 1.

$$E[\|H_iM_i - T^*\|_F^2] \leq (1 - 1/d_1)^{i(k-d_2)}\|T^*\|_F^2.$$

We note that our choice of M_i would give a good estimate $T_i = H_iM_i$ of T: Recall that $n \gg d_1 > k$. Hence we can assume that X has linear independent columns and the random matrix H_i has linear independent rows. Then using the Moore-Penrose inverse,

$$\begin{aligned}
\arg\min_{M_i}\|XH_iM_i - Y\|_F &= (XH_i)^+Y \\
&= H_i^+X^+Y \quad \text{(Since } H_I \text{ and } X \text{ are both full rank)} \\
&= H_i^+T^* \quad \text{(By definition of } T^*\text{)} \\
&= \arg\min_{M_i}\|H_iM_i - T^*\|_F.
\end{aligned}$$

In particular, the content of the main theorem is that $\|H_iM_i - T^*\|_F$ converges to 0 as i increases.

Let's consider a non-optimal choice of M_i first. Suppose we set M_i to be

$$M_i = \begin{bmatrix} I_{k-d_2} \\ M_i' \end{bmatrix},$$

where M_i' is chosen such that $\|H_iM_i - T^*\|_F$ is minimized. By direct multiplication, we have $H_iM_i = T_{i-1} + N_iM_i'$. Hence in this case minimizing $\|H_iM_i - T^*\|_F$ is equivalent to finding M_i' at each step such that $\|N_iM_i' - (T^* - T_{i-1})\|_F$ is minimized. In other words, we are trying to estimate T^* with $\sum_i H_iM_i'$, where each H_i is random and each M_i' is greedily chosen at each step. This is similar to, for example, the approaches in Barron et al. (2008), though they use a concrete list of H_i's.

Let's solve a sub-problem first.

Lemma 1. *Let $v \in \mathbb{R}^d$ be a vector. Let $\hat{n} \in \mathbb{R}^d$ be a random vector uniformly sampled from the unit sphere. Let $a \in \mathbb{R}$ such that $\|v + a\hat{n}\|_2$ is minimized. Then*

$$E\left[\|v + a\hat{n}\|_2^2\right] = (1 - 1/d)\|v\|_2^2.$$

Proof. First, we have

$$a = \langle v, \hat{n}\rangle.$$

Furthermore, because $\hat{n}$ is uniformly sampled from the sphere, we have

$$E\left[\langle v, \hat{n}\rangle^2\right] = \|v\|_2^2/d.$$

Now,

$$\begin{aligned}
E\left[\|v - a\hat{n}\|_2^2\right] &= E\left[\langle v, v\rangle - 2\langle v, \hat{n}\rangle^2 + \langle v, \hat{n}\rangle^2\right] \\
&= \|v\|_2^2 - E\left[\langle v, n\rangle^2\right] \\
&= (1 - 1/d)\|v\|_2^2.
\end{aligned}$$

□

Lemma 2. *Let $V \in \mathbb{R}^{d_1 \times d_2}$ be a matrix. Let N be a random matrix in $\mathbb{R}^{d_1 \times 1}$, where the column is uniformly sampled from the unit sphere S^{d_1-1}. We then pick $A \in \mathbb{R}^{1 \times d_2}$ such that $\|NA - V\|_F$ is minimized. Then*

$$E\left[\|NA - V\|_F^2\right] = (1 - 1/d_1)\|V\|_F^2.$$

Proof. Let $A = [a_1|\dots|a_{d_2}]$. We note that

$$\|V\|_F^2 = \sum_{i=1}^{d_2} \|V_{:,i}\|_2^2.$$

By lemma 1, we have

$$E\left[\|V_{:,i} + a_i N\|_2^2\right] = (1 - 1/d_1)\|V_{:,i}\|_2^2.$$

Hence the linearity of expectation gives

$$E\left[\|V + NA\|_F^2\right] = (1 - 1/d_1)\|V\|_F^2. \qquad \square$$

Lemma 3. *Let $V \in \mathbb{R}^{d_1 \times d_2}$ be a matrix. Let N be a random matrix in $\mathbb{R}^{d_1 \times (k-d_2)}$, where each column is uniformly sampled from the unit sphere S^{d_1-1}. We then pick $A \in \mathbb{R}^{(k-d_2)\times d_2}$ such that $\|NA - V\|_F$ is minimized. Then*

$$E\left[\|NA - V\|_F^2\right] = (1 - 1/d_1)^{k-d_2}\|V\|_F^2.$$

Proof. If $k = d_2 + 1$, this is just lemma 2. If $k > d_2 + 1$, then

$$\begin{aligned}
\mathrm{E}\left[\|NA - V\|_F^2\right] &= \mathrm{E}\left[\|(\sum_{i=1}^{k-d_2} N_{:,i}A_i) - V\|_F^2\right] \\
&= \mathrm{E}\left[\|N_{:,1}A_1 + (\sum_{i=2}^{k-d_2} N_{:,i}A_i) - V\|_F^2\right] \\
&= (1 - 1/d_1)\,\mathrm{E}\left[\|(\sum_{i=2}^{k-d_2} N_{:,i}A_i) - V\|_F^2\right] \quad \text{by lemma 2} \\
&= (1 - 1/d_1)^{k-d_2}\,\mathrm{E}[\|-V\|_F^2] \quad \text{by induction} \\
&= (1 - 1/d_1)^{k-d_2}\|-V\|_F^2. \qquad \square
\end{aligned}$$

Now we prove our main theorem.

Proof of the main theorem. We prove by induction. Suppose by induction that T_{i-1} is obtained from our method and satisfies the statement. Recall that $H_i = [T_{i-1}|N_i]$. If we pick M_i' such that $\|H_i M_i - T^*\|_F$ is minimized, then by lemma 3,

$$E\left[\|T_{i-1} + N_i M_i' - T^*\|_F^2\right] = (1 - 1/d_1)^{k-d_2}\|T^* - T_{i-1}\|_F^2.$$

Since $T_{i-1} + N_i M_i' = H_i \cdot [I_{k-d_2}; M_i']$, an optimal M_i satisfies

$$\begin{aligned}
E\left[\|H_i M_i - T^*\|_F^2\right] &\le E\left[E\left[\|H_i M_i - T^*\|_F^2 \mid T_{i-1}\right]\right] \\
&\le (1 - 1/d_1)^{k-d_2} E\left[\|T^* - T_{i-1}\|_F^2\right] \\
&\le (1 - 1/d_1)^{i(k-d_2)}\|T^*\|_F^2 \quad \text{by induction.} \qquad \square
\end{aligned}$$

C TABLE COLLAPSE

Table collapsing was a problem we encountered for the circular clustering method as described in appendix A. We describe the problem and the metric we used to detect it here, since we think they may be of interest to the community.

Suppose we are doing k-means clustering on a table of 3 partitions in order to obtain 3 index pointer functions h_j^c. These functions can be thought as a table, where the (i, j)-entry is given by $h_j^c(i)$.

There are multiple failure modes we have to be aware of. The first one is column-wise collapse:

1	0	0
1	1	2
1	0	3
⋮	⋮	⋮
1	3	1

In this table the first column has collapsed to just one cluster. Because of the way k-means clustering works, this exact case of complete collapse isn't actually possible, but we might get arbitrarily low entropy as measured by H_1, which we define as follows: For each column j, its column entropy is defined to be the entropy of the probability distribution $p_j : h_j^c([n]) \to [0, 1]$ defined by

$$p_j(x) = \frac{\#\{i : h_j^c(i) = x\}}{n}.$$

Then we define H_1 to be the minimum entropy of the (here 3) column-entropies.

The second failure mode is pairwise collapse:

1	0	1
2	2	3
1	0	3
3	1	0
2	2	1

In this case the second column is just a permutation of the first column. This means the expanded set of possible vectors is much smaller than we would expect. We can measure pairwise collapse by computing the entropy of the histogram of pairs, where the entropy of the column pair (j_1, j_2) is defined by the column entropy of $h_{j_1}^c(\cdot) + \max(h_{j_1}^c)h_{j_2}^c(\cdot)$. Then we define H_2 to be the minimum of such pair-entropies for all $\binom{3}{2}$ pairs of columns.

Pairwise entropy can be trivially generalized to triple-wise and so on. If we have c columns we may compute each of $H_1, \ldots, H_c$. In practice H_1 and H_2 may contain all the information we need.

C.1 What entropies are expected?

The maximum value for H_1 is $\log k$, in the case of a uniform distribution over clusters. The maximum value for H_2 is $\log \binom{k}{2} \approx 2 \log k$. (Note $\log n$ is also an upper bound, where n is the number of points in the dataset / rows in the table.)

With the QR method we expect all the entropies to be near their maximum. However, for the Circular Clustering method this is not the case! That would mean we haven't been able to extract any useful cluster information from the data.

Instead we expect entropies close to what one gets from performing Product Quantization (PQ) on a complete dataset. In short:

1. Too high entropy: We are just doing QR more slowly.
2. Too low entropy: We have a table collapse.
3. Golden midpoint: Whatever entropy normal PQ gets.

D Hashing

If $h : [n] \to [m]$ and $s : [n] \to \{-1, 1\}$ are random functions, a Count Sketch is a matrix $H \in \{0, -1, 1\}^{m \times n}$ where $H_{i,j} = s(i)$ if $h(i) = j$ and 0 otherwise. Charikar et al. (2002) showed that if m is large enough, the matrix H is a dimensionality reduction in the sense that the norm $\|x\|_2$ of any vector in $\mathbb{R}^n$ is approximately preserved, $\|Hx\|_2 \approx \|x\|_2$.[4]

This gives a simple theoretical way to think about the algorithms above: The learned matrix $T' = H^T T$ is simply a lower dimensional approximation to the real table that we wanted to learn. While the theoretical result requires the random "sign function" s for the approximation to be unbiased, in practice this appears to not be necessary when directly learning T'. Maybe because the vectors can simply be slightly shifted to debias the result.

There are many strong theoretical results on the properties of Count Sketches. For example, Woodruff (2014) showed that they are so called "subspace embeddings" which means the dimensionality reduction is "robust" and doesn't have blind spots that SGD may accidentally walk into.

[4]This also implies that inner products are approximately preserved by the dimensionality reduction.

However, the most practical result is that one only needs h to be a "universal hash function" ala Carter & Wegman (1977), which can be as simple and fast as the "multiply shift" hash function by Dietzfelbinger et al. (1997).

If Count Sketch shows that hashing each $i \in [n]$ to a single row in $[m]$, we may wonder why methods like Hash Embeddings use multiple hash functions (or DHE uses more than a thousand.) The answer can be seen in the theoretical analysis of the "Johnson Lindenstrauss" transformation and in particular the "Sparse Johnson Lindenstrauss" as analyzed by Cohen et al. (2018). The analysis shows that if the data being hashed is not very uniform, it is indeed better to use more than one hash function (more than 1 non-zero per column in H.) The exact amount depends on characteristics in the data distribution, but one can always get away with a sparsity of ϵ when looking for a $1 + \epsilon$ dimensionality reduction. Hence we speculate that DHE could in general replace the 1024 hash functions with something more like Hash Embeddings with an MLP on top. Another interesting part of the Cohen et al. (2018) analysis is that one should ideally split $[m]$ in segments, and have one hash function into each segment. This matches the implementations we based our work on below.