MOPBucket: A Massively OP* algorithm for k-means clustering Bucketloads of data

Jack Spalding-Jamieson
Independent
jacksj@uwaterloo.ca

Eliot W. Robson
Department of Computer Science
University of Illinois Urbana-Champaign
erobson2@illinois.edu

Da Wei Zheng
Department of Computer Science
University of Illinois Urbana-Champaign
dwzheng2@illinois.edu

Abstract

k-means clustering is one of the most popular tools for unsupervised learning. Unfortunately, the existing algorithms for k-means scale poorly with the dataset size n and the number of clusters k, especially when the dimension d of the dataset is moderate or large ($d \geq 100$). In this work, we use techniques inspired by tools for approximate nearest-neighbour search to obtain a highly scalable k-means clustering algorithm.

1 Introduction

k-means clustering is a classical problem in unsupervised learning and computational geometry, with numerous applications across data mining. This and related problems have been extremely well studied over the years, and it holds significant practical importance.

This problem has long been known to be NP-hard, but a local search algorithm commonly attributed to Lloyd [9] is known to produce good results. This algorithm consists of the following basic steps:

1. **Initialize** a set of k centers C and assign each point in P to its closest center in C. The most straightforward method for doing this is to sample points in P uniformly.
2. **Recompute** each center C_i by taking the average of points assigned to it.
3. **Reassign** each point P to its closest center. If no points are reassigned, the algorithm terminates. Otherwise, go back to step 2.

Note that the second and third steps constitute a local search on the problem, and are commonly referred to as "Lloyd iterations".

2 Methodology

In this work, we explore improvements to the empirical performance of Lloyd iterations as previously described. Step 2, recomputing the centers from the clusters assigned to them, is a simple linear-time averaging that is trivially parallelizable, making it difficult to meaningfully optimize. We focus on optimizing step 3, assigning each point in P to its closest center. Specifically, we will maintain an HNSW-based vector search index for the centroids.

HNSW [10] is a heuristic-based algorithm for performing approximate nearest neighbor queries. It consists of a k-NN graph (not the same k we use for clustering) whose edges are pruned according to a popular heuristic [14] paired with a greedy search for queries. The starting point of the greedy search is determined by using the result of the same structure applied to a randomly sampled subset of the points, similar to a skip list. Although techniques of this family have known lower bounds [6, 3], HNSW is known to produce good results while operating in-memory (RAM), comparable to the best algorithms/implementations [2]. Moreover, unlike most other comparably fast algorithms, it does not rely on quantization techniques to achieve its speed.

Note that we will not be applying HNSW as a black-box, but instead we perform some critical improvements specific to this use-case. Without these improvements, we would otherwise obtain much worse results, as we will see in our experimental section.

Each iteration of our clustering algorithm can be broken into three stages:

Build: Compute an HNSW-based vector search index on the centroids.

Reassign: Carefully use the index to compute approximate nearest centroids for each point in the dataset, and assign the point to the corresponding cluster.

*OP is slang for overpowered.

Recompute: Recompute the centroids based on the contents of each cluster.

The recompute step remains unchanged from Lloyd's algorithm. We present a series of improvements to the reassign step. Additionally, we note that the build step in most iterations can be improved by starting with the HNSW graph from the previous iteration, and performing local improvements for the newly-perturbed centroids.

2.1 Better Approximate Reassignments

The naive approach of the reassign step would be to simply use the HNSW index to choose the closest centroids. We present several significant improvements on this approach.

Previous assignments as seed points If the index returns a point further from the previously assigned centroid, the assignment can get *worse* after an iteration. This has a easy fix: simply compare the returned value with the previously assigned centroid, and use the previous centroid if it is better. Rather than doing this, we offer a better solution with a modified HNSW search. We use the previously assigned centroid as an additional seed point in the bottom-most level of the HNSW search.

We make a further observation: Since the centroids do not significantly change over time, if the nearest centroid to a datapoint changes, it is (intuitively) likely to be one of the nearest centroids in the previous iteration. Hence, we can record not just the nearest centroid at each iteration, but several of the nearest centroids. We call this parameter `additional_centroids`. Then, *all* of these points can be used as additional seed points during HNSW.

Minimum iteration count Next, we also add a minimum iteration count to the search at the bottom level of HNSW, guaranteeing that at least a certain number of vertices are visited. This is a feature not present in HNSW, and in fact limited testing revealed that it does not seem to prove helpful for approximate nearest neighbour search. However, since our search graph changes slowly between iterations, we believe this can help escape some critical local optima by diversifying the possible results. We call this parameter `min_iterations`.

Bulk queries for additional seed points We also made use of bulk queries. Specifically, for a set of datapoints Q small enough to be loaded into RAM together (with room for this to occur on many threads simultaneously), we perform these queries together in the following manner:

- Perform the greedy search at all upper levels of the HNSW index to get base seed points for each datapoint in Q.
- Randomly project all the datapoints in Q into a 1-dimensional space.
- Sort all the datapoints in Q first by their base seed points, and then by their projection in the 1-dimensional space.
- Perform the final nearest-neighbour search at the bottom level for each datapoint in order. During each iteration, we add all the previously mentioned additional seed points, and we also add all the top results of the previous iteration as further additional seed points.

3 Experiments

We ran experiments comparing our MOPBucket algorithm (**MopKMeans**) against the SciKit Learn library's implementation of k-means (**ScikitKMeans**) [12] and our own straightforward GPU-accelerated Pytorch implementation of Lloyd's algorithm (**PyTorchKMeanGPU**)[11]. Note that our comparison against a GPU is in fact disadvantageous for MOPBucket, as GPU computation typically scales far better for distance computation on large data sets, while MOPBucket runs solely on CPU. We also compared against a naive black-box implementation of our HNSW-based approach *without* the improvements we discussed in Section 2, using the library faiss as a standard HNSW implementation [4].

3.1 Experimental setup

We conducted the experiments on a workstation machine with Ubuntu 24.04.4 LTS, equipped with an AMD Ryzen 9 7950x CPU, 64GB of RAM, an Nvidia RTX 3090 GPU, and datasets stored in a 2TB SSD. Our algorithm, SciKit, and the faiss HNSW black-box implementation were all run exclusively on the CPU with a maximum of 12 threads. The pytorch implementation was run on the GPU. We note that this GPU is significantly more expensive, and more powerful, than the CPU. Thus, it is particularly notable to obtain better results using just a CPU.

We test on the following datasets:

- Yandex's **Text2Image10M** dataset [13] which consists of images embeddings produced by the `Se-ResNext-101 model` [5]. This data set was used for benchmarking for the NeurIPS 2021 large-scale ANNS competition, and is often very similar to real world data sets.
- The **SIFT1B** dataset [7] and a **20M slice of the SIFT1B** dataset that we will call **SIFT20M**.
- The **DPR10M** dataset generated by [1] from 768-dimensional dense passage retriever model of [8]. To make the dataset of comparable size to the other ones, we created **DPR5M** by taking **a 5M slice of DPR10M**.

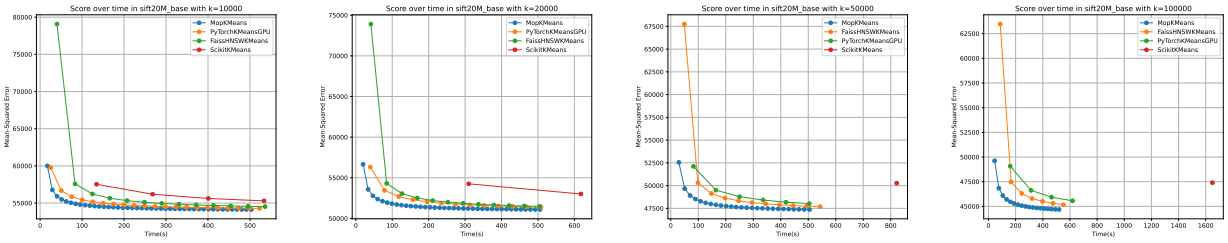

Figure 1: Score over time for values of $k \in \{10\,000, 20\,000, 50\,000, 100\,000\}$ running on SIFT 20M, halting after the first iteration exceeding 500 seconds. Closer to bottom-left is better.

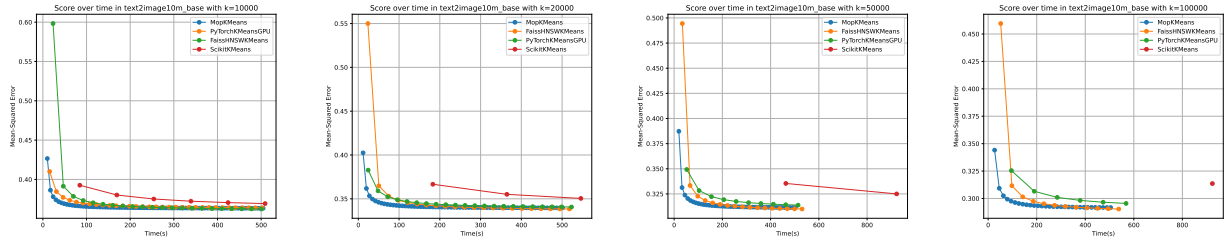

Figure 2: Score over time for values of $k \in \{10\,000, 20\,000, 50\,000, 100\,000\}$ running on text2image10M, halting after the first iteration exceeding 500 seconds. Closer to bottom-left is better.

3.2 Results and Analysis

We ran the following experiments.

- **Score over time**. We compare the performance of the three algorithms with values of $k \in \{10\,000, 20\,000, 50\,000, 100\,000\}$ over time. See Figure 1 for comparison on SIFT 20M and Figure 2 on Text2Image10M. We ran each algorithm until the first iteration that exceeded a runtime of 500 seconds. See also Figure 4 for a plot of the scores achieved with a 500 second timeout as a function of varying values of k. It can be seen that MopKMeans in all cases is both faster and attains better mean squared error.
- **Comparison of time to reach MOPBucket performance**. We compare time it takes for PyTorchKMeansGPU and ScikitKMeans against the baseline score of MopKMeans in 100 seconds (Figure 5). In all cases MOPBucket achieved a higher score; in some cases 12 times faster than ScikitKMeans.
- **Large-scale k-means**. To demonstrate the scalability of our algorithm, we run MopKMeans on the billion-scale dataset SIFT1B with $k = 1\,000\,000$. We obtain a good k-means score (i.e. difficult to improve by more than 10%) in less than 2.5 hours of computation time. We estimate by linear interpolation that ScikitKMeans would take roughly 9.5 days to run a single iteration at this scale of k on our hardware, and would likely obtain a worse score.

4 Conclusion

In this work, we presented a specialization of a family of techniques for approximate nearest neighbours for accelerating Lloyd iterations, and gave a corresponding empirical investigation. We compared our implementation using these methods on a CPU to popular libraries and implementations using existing algorithms on a GPU. The experiments on popular datasets demonstrate a

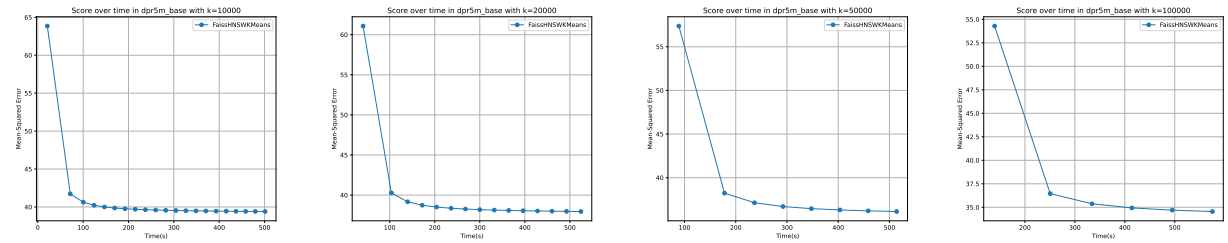

Figure 3: Score over time for values of $k \in \{10\,000, 20\,000, 50\,000, 100\,000\}$ running on DPR5M, halting after the first iteration exceeding 500 seconds. Closer to bottom-left is better.

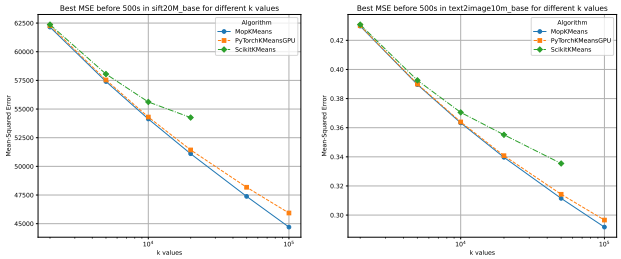

Figure 4: Best score achieved with a 500 second timeout on different algorithms and numbers of clusters. Missing datapoints for ScikitKMeans indicates a single iteration failed to complete before the timeout.

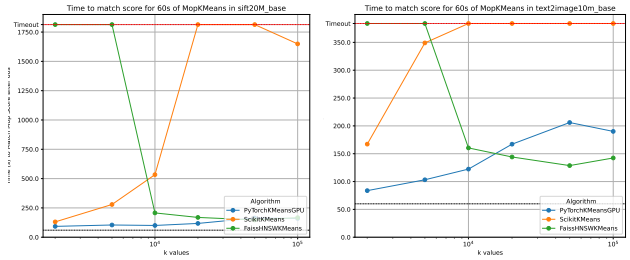

Figure 5: For various values of k, plots of the time it takes other algorithms to reach the same score that MOPBucket can achieve in at most 60 seconds, at a timeout of 500 seconds. Note that ScikitKMeans usually times out.

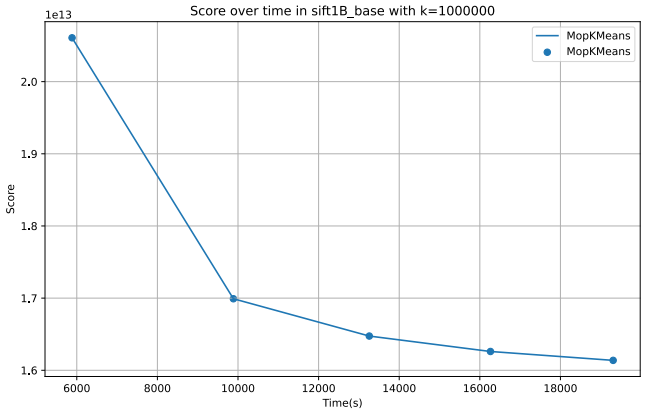

Figure 6: The score over time of our algorithm on SIFT1B with $k = 1\,000\,000$. We estimate that scikit would take approximately 9.5 days to run even a single iteration with these values.

clear performance improvement using our techniques, even when comparing a (relatively) cheap CPU to an expensive consumer GPU. This provides substantial evidence of improved performance using our methods over existing ones.

References

[1] Cecilia Aguerrebere, Ishwar Bhati, Mark Hildebrand, Mariano Tepper, and Ted Willke. Similarity search in the blink of an eye with compressed indices. *Proceedings of the VLDB Endowment*, 16(11):3433–3446, 2023.

[2] Martin Aumüller, Erik Bernhardsson, and Alexander Faithfull. Ann-benchmarks: A benchmarking tool for approximate nearest neighbor algorithms. *Information Systems*, 87:101374, 2020.

[3] Haya Diwan, Jinrui Gou, Cameron Musco, Christopher Musco, and Torsten Suel. Navigable graphs for high-dimensional nearest neighbor search: Constructions and limits. *arXiv preprint arXiv:2405.18680*, 2024.

[4] Matthijs Douze, Alexandr Guzhva, Chengqi Deng, Jeff Johnson, Gergely Szilvasy, Pierre-Emmanuel Mazaré, Maria Lomeli, Lucas Hosseini, and Hervé Jégou. The faiss library. *CoRR*, abs/2401.08281, 2024. doi: 10.48550/ARXIV.2401.08281. URL `https://doi.org/10.48550/arXiv.2401.08281`.

[5] Jie Hu, Li Shen, Samuel Albanie, Gang Sun, and Enhua Wu. Squeeze-and-excitation networks. *IEEE Trans. Pattern Anal. Mach. Intell.*, 42(8):2011–2023, 2020. doi: 10.1109/TPAMI.2019.2913372. URL `https://doi.org/10.1109/TPAMI.2019.2913372`.

[6] Piotr Indyk and Haike Xu. Worst-case performance of popular approximate nearest neighbor search implementations: Guarantees and limitations. *Advances in Neural Information Processing Systems*, 36:66239–66256, 2023.

[7] Hervé Jégou, Romain Tavenard, Matthijs Douze, and Laurent Amsaleg. Searching in one billion vectors: Re-rank with source coding. In *Proceedings of the IEEE International Conference on Acoustics, Speech, and Signal Processing, ICASSP 2011, May 22-27, 2011, Prague Congress Center, Prague, Czech Republic*, pages 861–864. IEEE, 2011. doi: 10.1109/ICASSP.2011.5946540. URL `https://doi.org/10.1109/ICASSP.2011.5946540`.

[8] Vladimir Karpukhin, Barlas Oguz, Sewon Min, Patrick Lewis, Ledell Wu, Sergey Edunov, Danqi Chen, and Wen-tau Yih. Dense passage retrieval for open-domain question answering. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 6769–6781, Online, November 2020. Association for Computational Linguistics. doi: 10.18653/v1/2020.emnlp-main.550. URL `https://www.aclweb.org/anthology/2020.emnlp-main.550`.

[9] Stuart P. Lloyd. Least squares quantization in PCM. *IEEE Trans. Inform. Theory*, 28(2):129–137, 1982. ISSN 0018-9448. doi: 10.1109/TIT.1982.1056489. URL https://doi.org/10.1109/TIT.1982.1056489.

[10] Yu A Malkov and Dmitry A Yashunin. Efficient and robust approximate nearest neighbor search using hierarchical navigable small world graphs. *IEEE transactions on pattern analysis and machine intelligence*, 42(4):824–836, 2018.

[11] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Z. Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library. In Hanna M. Wallach, Hugo Larochelle, Alina Beygelzimer, Florence d'Alché-Buc, Emily B. Fox, and Roman Garnett, editors, *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, December 8-14, 2019, Vancouver, BC, Canada*, pages 8024–8035, 2019. URL https://proceedings.neurips.cc/paper/2019/hash/bdbca288fee7f92f2bfa9f7012727740-Abstract.html.

[12] 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.

[13] Harsha Vardhan Simhadri, George Williams, Martin Aumüller, Matthijs Douze, Artem Babenko, Dmitry Baranchuk, Qi Chen, Lucas Hosseini, Ravishankar Krishnaswamny, Gopal Srinivasa, Suhas Jayaram Subramanya, and Jingdong Wang. Results of the neurips'21 challenge on billion-scale approximate nearest neighbor search. In Douwe Kiela, Marco Ciccone, and Barbara Caputo, editors, *Proceedings of the NeurIPS 2021 Competitions and Demonstrations Track*, volume 176 of *Proceedings of Machine Learning Research*, pages 177–189. PMLR, 06–14 Dec 2022. URL https://proceedings.mlr.press/v176/simhadri22a.html.

[14] Godfried T Toussaint. The relative neighbourhood graph of a finite planar set. *Pattern recognition*, 12(4):261–268, 1980.