Efficient Coreset Selection with Cluster-based Methods

Chengliang Chai*
ccl@bit.edu.cn
Beijing Institute of Technology
Beijing, China

Jiayi Wang*
jiayi-wa20@mails.tsinghua.edu.cn
Tsinghua University
Beijing, China

Nan Tang
nantang@hkust-gz.edu.cn
HKUST (GZ)
GuangZhou, China

Ye Yuan
Jiabin Liu
yuan-ye@bit.edu.cn
ljb604163320@gmail.com
Beijing Institute of Technology
Beijing, China

Yuhao Deng
Guoren Wang
3220220908@bit.edu.cn
wanggrbit@gmail.com
Beijing Institute of Technology
Beijing, China

ABSTRACT

Coreset selection is a technique for efficient machine learning, which selects a subset of the training data to achieve similar model performance as using the full dataset. It can be performed with or without training machine learning models. Coreset selection with training, which iteratively trains the machine model and updates data items in the coreset, is time consuming. Coreset selection without training can select the coreset before training. Gradient approximation is the typical method, but it can also be slow when dealing with large training datasets as it requires multiple iterations and pairwise distance computations for each iteration. The state-of-the-art (SOTA) results *w.r.t.* effectiveness are achieved by the latter approach, *i.e.,* gradient approximation.

In this paper, we aim to significantly improve the efficiency of coreset selection while ensuring good effectiveness, by improving the SOTA approaches of using gradient descent without training machine learning models. Specifically, we present a highly efficient coreset selection framework that utilizes an approximation of the gradient. This is achieved by dividing the entire training set into multiple clusters, each of which contains items with similar feature distances (calculated using the Euclidean distance). Our framework further demonstrates that the full gradient can be bounded based on the maximum feature distance between each item and each cluster, allowing for more efficient coreset selection by iterating through these clusters. Additionally, we propose an efficient method for estimating the maximum feature distance using the product quantization technique. Our experiments on multiple real-world datasets demonstrate that we can improve the efficiency 3-10× comparing with SOTA almost without sacrificing the accuracy.

*Both authors contributed equally to this research.

KDD '23, August 6–10, 2023, Long Beach, CA, USA

ACM ISBN 979-8-4007-0103-0/23/08…$15.00
https://doi.org/10.1145/3580305.3599326

CCS CONCEPTS

• **Theory of computation** → *Sample complexity and generalization bounds.*

KEYWORDS

Coreset selection; Data-efficient ML; Product quantization

ACM Reference Format:
Chengliang Chai, Jiayi Wang, Nan Tang, Ye Yuan, Jiabin Liu, Yuhao Deng, and Guoren Wang. 2023. Efficient Coreset Selection with Cluster-based Methods. In *Proceedings of the 29th ACM SIGKDD Conference on Knowledge Discovery and Data Mining (KDD '23), August 6–10, 2023, Long Beach, CA, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3580305.3599326

1 INTRODUCTION

Recent advances in machine learning (ML) have addressed the problem of learning from a large corpus of train data using highly scalable solutions and frameworks (for example, Tenorflow [4], Pytorch [40], and so forth). However, it is also known that learning from massive amounts data is expensive. Hence, one challenge is how to learn with less time or from less data, but with the same performance – this is known as *data-efficient* machine learning.

For learning with less time, a classical solution is stochastic gradient decent (SGD), which focuses on accelerating the optimization algorithm by estimating the full gradient via mini-batches of train data. For learning with less data, *coreset* algorithms select a subset of train data such that training on the coreset can perform on par with the entire train data.

In practice, the above two lines of works are complementary, *e.g.,* the selected coreset is often trained using SGD.

In this work, we seek to answer the following question: *whether we can significantly improve the efficiency of coreset selection while ensuring its effectiveness.* A positive answer to this question is important to reducing the overall time of training ML models.

Generally speaking, there are two classes of coreset selection algorithms: *coreset selection with training* and *coreset selection without training*. For coreset selection with training, existing solutions [15, 19, 24, 36] compute the coreset through iterative training, and determine whether to add a training item into the coreset considering its confidence or loss. Because training is needed for each iteration, these solutions are time consuming. In contrast, coreset selection

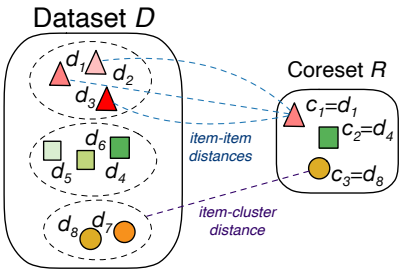

Figure 1: An Overview of Coreset Selection.

without training solutions select the coreset without actually training ML models. Earlier studies on this thread are customized to a particular type, *e.g.,* $k-$means [6, 17], nearest neighbors [46], support vector machine [44], and so on. The SOTA works [38, 39, 41] adopt the idea of gradient approximation, which can compute a good coreset without training in advance for various models optimized via gradient decent. The SOTA results *w.r.t.* effectiveness are achieved by gradient approximation.

Next, let us use an example to illustrate the idea of gradient decent based coreset selection, which uses the coreset to approximate the gradient of full train data and thus leads to similar performance on both the coreset and full train data.

Example 1. *[Gradient approximation for coreset selection.] Figure 1 shows a train set* $D = \{d_1, d_2, ..., d_8\}$. *Each training item* d_i *consists of a feature vector and a label. The full gradient is the sum of gradients of all training items in D. Gradient decent is to find the best parameter of an ML model by iteratively computing the full gradient.*

In order to approximate the full gradient using the coreset, as the size of coreset is smaller than that of full data, we need to compute the weighted sum of gradients of items in the coreset as the approximation. Given D, a good coreset $R \subseteq D$ *with* $|R| = K$ *has the property that every* $d_i \in D$ *ought to be approximated by an item* $c_j \in R$ *(similar to* d_i *in the gradient) such that all items in D can be well approximated by R. The weight of* c_j *equals to the number of items that* c_j *represents in D. For example, in Figure 1, items in similar colors indicate similar gradients, and thus* d_1, d_2 *and* d_3 *can be represented by* c_1 *with a weight of 3.*

Challenge. The reason that coresets can be selected before training is that the gradient difference between every pair of training items can be bounded by the feature distance [18, 47] of this pair (*i.e.,* the item-item distance Figure 1). Nevertheless, current solutions need to enumerate feature distances between every pair, and iterate the large dataset D multiple times, which could be expensive when D is large.

Key observation. Figure 1 shows that data items (*e.g.,* d_1, d_2 and d_3) approximated by an item (c_1) in the coreset are closer to each other than those data items (*e.g.,* $d_5 - d_8$) that cannot be approximated by the corresponding coreset item (c_1). Intuitively, we can cluster the full train dataset and compute the coreset based on clusters. Clearly, this approach can significantly improve the efficiency. We will prove that this approach does not sacrifice effectiveness much and empirically verify it.

Our proposal. In a nutshell, we first partition D into clusters (as shown the dotted circles in Figure 1) such that the feature distance between items in each cluster is close to each other. Afterwards, to select the coreset based on these clusters, we face two challenges. First, the coreset should still approximate the full gradient to achieve good model performance. To address this, we prove that for each item and cluster, if we can compute a maximum feature distance between the item and each item within the cluster, the full gradient can be approximated and bounded. Second, how to efficiently measure the item-cluster distance becomes another challenge. To address this, we propose to leverage the product quantization method that splits the entire feature space, computes the distance in each subspace, and then aggregates the result. In this way, instead of enumerating feature distances among all pairs, we only need to consider these item-cluster distances. Also, during the process of coreset selection, we can just iterate the clusters in D rather than all items, which further improves the efficiency.

Contributions. We make the following notable contributions:

(1) We propose `FastCore`, a cluster-based coreset selection framework that can efficiently approximate the full gradient to produce a well-performed coreset. (Section 3)
(2) We formally define the cluster-based coreset selection problem, prove its hardness, and theoretically analyze the bound of the gradient approximation. (Section 4)
(3) We efficiently measure the distance between an item and a cluster using the product quantization based method to split the feature space. (Section 5)
(4) Experimental results on 5 real-world datasets demonstrate that our method can achieve $3 - 10\times$ acceleration almost without sacrificing the model accuracy. (Section 8)

2 PRELIMINARIES

2.1 Gradient Approximation for Coreset Selection

Suppose that we have a train dataset $D = \{d_1, d_2, ..., d_N\}$. Each $d_i = (\mathbf{x}_i, \mathbf{y}_i)$ denotes the feature vector with m dimensions ($\mathbf{x}_i \in \mathbb{R}^m$) associated with the label ($\mathbf{y}_i$) of the training item d_i, the training objective is to minimize the loss by discovering the best parameters ω^* as follows:

$$\omega^* = \arg\min_{\omega \in \mathbb{W}} \mathcal{F}(\omega),\ \mathcal{F}(\omega) = \frac{1}{N}\sum_{i=1}^{N} f_i(d_i, \omega) \quad (1)$$

where $\mathbb{W}, \mathcal{F}(\cdot)$ and $f_i(\cdot)$ denote the parameter space, overall loss and loss of the i-th item respectively. We abbreviate $f_i(d_i, \omega)$ as $f_i(\omega)$. To optimize Eq. 1, gradient decent is always applied by iteratively computing the full gradient, *i.e.,* $\nabla\mathcal{F}(\omega)$.

Coresets for data-efficient ML. It is always expensive to train with large datasets because of the iterative gradient computation. Although methods like stochastic gradient descent can be used for acceleration, it is still not efficient enough for massive training examples. Therefore, recently, researchers have focused on selecting a subset (*a.k.a.* the *coreset* R) of the full train data (D) such that training on R performs on par with training on D.

Coreset selection by gradient approximation. The SOTA coreset selection frameworks [26, 38, 39, 41] are typically based on

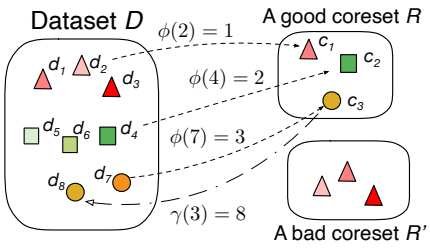

Figure 2: Example of Coreset Selection.

gradient approximation. To be specific, suppose that ω^* is the best parameter trained over the full data D, and ω' is trained over the coreset $R \subseteq D$. The goal of gradient approximation is to select the coreset such that $\nabla\mathcal{F}(\omega')$ is as close as possible to $\nabla\mathcal{F}(\omega^*)$, and thus they will have similar performance. Existing works [26, 38, 39, 41] achieve this goal by minimizing the upper bound of $\|\nabla\mathcal{F}(\omega') - \nabla\mathcal{F}(\omega^*)\|$ ($\|\cdot\|$ denotes the normed difference), which we call gradient approximation error (GA error).

Since $R \subseteq D$ and $|R| \ll |D|$, how can we let the coreset gradient, *i.e.,* $\nabla\mathcal{F}(\omega')$ be close to the full gradient, *i.e.,* $\nabla\mathcal{F}(\omega^*)$? The answer is that each item of R will be associated with a weight λ and $\nabla\mathcal{F}(\omega')$ is a weighted sum of gradients. Formally, the definition of coreset selection based on GA error minimization is as below.

$$R^* = \arg\min_{R\subseteq D, \lambda_j\geq 0} \max_{\omega\in\mathbb{W}} \|\sum_{i=1}^{N} \nabla f_i(\omega) - \sum_{j=1}^{|R|} \lambda_j \nabla f_{\gamma(j)}(\omega)\|, \; s.t. \; |R| \leq K \quad (2)$$

Eq. 2 aims to select a coreset with size at most K through minimizing the GA error, considering the entire parameter space ($\omega \in \mathbb{W}$). We use Figure 2 to better illustrate the equation, where $N = 8$ examples are in D and we aim to select a coreset R with size $K = 3$. Since $R \subseteq D$, we use a mapping $\gamma(j) = i, j \in [1, K], i \in [1, N]$ to denote that the j-th item $c_j \in R$ is the i-th item in D. As shown in Figure 2, given the coreset R, we have $\gamma(1) = 1$, $\gamma(2) = 4$ and $\gamma(3) = 8$. Note that γ is a natural mapping from R to D when R is given, and we need another significant mapping ϕ from D to R, which is highly related to the weight γ in Eq. 2.

Generally speaking, the coreset is selected to represent the full train data, and thus $\phi(i) = j$ indicates that we ask $c_j \in R$ to represent $d_i \in D$ (*a.k.a.* assign d_i to c_j), *i.e.,* using $\nabla f_{\gamma(j)}$ to approximate ∇f_i. Each $d_i \in D$ will be assigned to just one $c_j \in R$, but each c_j might represent multiple items in D. The number of items represented by c_j is defined as the weight of c_j, denoted as λ_j.

Example 2. *In Figure 2, suppose that similar points in shape indicate similar gradients ($\nabla f_1(\omega) \approx \nabla f_2(\omega) \approx \nabla f_3(\omega)$, $\nabla f_4(\omega) \approx \nabla f_5(\omega) \approx \nabla f_6(\omega)$ and $\nabla f_7(\omega) \approx \nabla f_8(\omega)$) in D. In this situation, a good coreset can be $R = \{d_1, d_4, d_8\}$ with mapping $\phi(1) = \phi(2) = \phi(3) = 1, \phi(4) = \phi(5) = \phi(6) = 2$ and $\phi(7) = \phi(8) = 3$. Thus $\lambda_1 = 3$, $\lambda_2 = 3$ and $\lambda_3 = 2$. This coreset is good because $\|\sum_{i=1}^{8} \nabla f_i(\omega) - \sum_{j=1}^{3} \lambda_j \nabla f_{\gamma(j)}(\omega)\|$ is close to 0. However, for another coreset $R' = \{d_1, d_2, d_3\}$, it cannot be good regardless of the mapping ϕ because it is hard to find an item in R' to well represent $d_4 - d_8$, thus leading to a big GA error.*

2.2 Minimize the GA Error

Coreset selection not only selects a subset from D, but should also be associated with the weights of items in the coreset. To minimize the GA error, intuitively, we should set $\phi(i) = j$ where ∇f_i and $\nabla f_{\gamma(j)}$ are close.

Minimizing the upper bound of GA error. To solve Eq. 2, we have to explore the parameter space, which is prohibitively expensive, so typically this is addressed by minimizing the upper bound of the GA error. To be specific, for any particular ω, we apply the triangle equation and have the upper bound [38]:

$$\|\sum_{i=1}^{N} \nabla f_i(\omega) - \sum_{j=1}^{|R|} \lambda_j \nabla f_{\gamma(j)}(\omega)\| \leq \sum_{i=1}^{N} \|\nabla f_i(\omega) - \nabla f_{\gamma(\phi_\omega(i))}(\omega)\| \quad (3)$$

Based on the above observation,the upper bound is minimized when ϕ assigns each d_i to the item in R with most similar gradient, *i.e.,* $\|\sum_{i=1}^{N} \nabla f_i(\omega) - \sum_{j=1}^{|R|} \lambda_j \nabla f_{\gamma(j)}(\omega)\| \leq \sum_{i=1}^{N} \min_{c_j\in R} \|\nabla f_i(\omega) - \nabla f_{\gamma(j)}(\omega)\|$.

To eliminate the parameter ω, it has been proved that using Lipschitz continuity, for convex ML problems [18, 47], *over the entire space $\mathbb{W}$, the normed gradient difference can be bounded by:*

$$\forall i, j, \max_{\omega\in\mathbb{W}} \|\nabla f_i(\omega) - \nabla f_j(\omega)\| \leq \max_{\omega\in\mathbb{W}} O(\|\omega\|) \cdot \|\mathbf{x}_i - \mathbf{x}_j\| \quad (4)$$

where $\|\mathbf{x}_i - \mathbf{x}_j\|$ is the *feature distance,i.e.,* Euclidean distance between feature vectors, and $O(\|\omega\|)$ is a constant. Therefore, we can conclude that *GA error can be bounded independent of the optimization problem,* i.e., *any particular ω*. Putting Eq. 3 and Eq. 4 together, *the coreset selection problem is defiend as:*

$$R^* = \arg\min_{R\subseteq D} \sum_{i=1}^{N} \min_{c_j\in R} \|\mathbf{x}_i - \mathbf{x}_{\gamma(j)}\|, \; \text{s.t. } |R| \leq K \quad (5)$$

Given a coreset R, we can score it by $\sum_{i=1}^{N} \min_{c_j\in R} \|\mathbf{x}_i - \mathbf{x}_{\gamma(j)}\|$. The lower the coreset score, the better the coreset is because it indicates a smaller upper bound of the GA error. To summarize, Eq. 5 is to select the coreset with the lowest score by purely considering the feature vectors of training items rather than training in advance. Note that Eq. 4 holds for items with the same label, so we respectively compute coresets for items with different labels and then combine these coresets.

2.3 Three-loop Framework for Coreset Selection

As solving Eq. 5 is an NP-hard problem with the sub-modular property, typical solutions [38, 39, 41] are based on a 3-loop framework with an approximation ratio $(1 - \frac{1}{e})$.

[1. *Pre-processing.*] The feature distance between every two items should be computed in advance, with the time complexity $O(mN^2)$.

[2. *First loop.*] The item with the maximum *benefit* will be greedily added in each iteration of the first loop. After K iterations, the coreset is constructed. The higher the *benefit*, the lower coreset score we can get if the item is added into current coreset.

[3. *Second loop.*] To pick the most beneficial item, we iterate items in D but are not in current coreset, to compute their benefits.

[4. *Third loop.*] To compute the benefit of each item, we also need to iterate items of D to determine their mapping ϕ based on feature distances.

Algorithm 1: FastCore Framework

Input: Train data D, coreset size K.
Output: A coreset $R \subseteq D$, weight $W = \{\lambda_j\}, |R| = |W| = K$.

1 $R = \emptyset$;
2 Cluster D into $C = \{C_1, C_2, \ldots C_T\}$;
3 Compute the upper bound u_{it} of $s^*_{it} = \max_{k \in C_t} s_{ik}$, where $s_{ik} = \|\mathbf{x}_i - \mathbf{x}_k\|$, $i \in [1, N]$, $t \in [1, T]$;
4 **while** $|R| < K$ **do**
5 /*1st loop*/
6 Sample $D_s \subseteq D \setminus R$;
7 **for** *each item* $d_i \in D_s$ **do**
8 /*2nd loop*/
9 $\mathcal{B}(R \cup \{d_i\}) = 0$;
10 **for** *each cluster* $C_t \in C$ **do**
11 /*3rd loop*/
12 $\mathcal{B}(R \cup \{d_i\}) += \min_{c_j \in R \cup \{d_i\}} u_{\gamma(j)t} \times |C_t|$;
13 $\mathcal{B}(d_i|R) = \mathcal{B}(R) - \mathcal{B}(R \cup \{d_i\})$;
14 $d^* = \arg\max_{d_i \in D_s} \mathcal{B}(d_i|R)$;
15 $R = R \cup \{d^*\}$;
16 **for** $c_j \in R$ **do**
17 $\lambda_j = \sum_{t=1}^{T} \mathbb{I}[c_j = \arg\min_{c_{j'} \in R} u_{\gamma(j')t}] \times |C_t|$;
18 **return** R, W;

As discussed above, to compute a coreset, the 3-loop framework has a time complexity of $O(KN^2)$, associated with a $O(mN^2)$ pre-processing step, which is rather expensive when there is a large N. To overcome this issue, in this paper, we study how to accelerate the coreset computation using cluster-based techniques, still with a theoretical guarantee, *i.e.,* bounded GA error.

3 FRAMEWORK

Recap that the typical solution mentioned in Section 2.3 is time-consuming mainly because we have to enumerate all feature distances, and iterate all items in D to measure the coreset.

Key idea. To overcome this, we propose to quickly cluster items in D, where each cluster contains similar items (*i.e.,* they have close feature distances). The key idea of FastCore is to select the coreset that can well represent these clusters rather than directly representing D. Since these clusters can capture the distribution of D, our selected coreset is still informative enough to approximate the full gradient of D. We have shown that the coreset selected by FastCore can still lead to a bounded GA error with a theoretical guarantee.

Framework overview. At a high level, we overview FastCore with Algorithm 1, which also consists of pre-processing and 3 loops. [1. *Pre-processing steps (lines 2-3)*] are quite different from the typical solution in Section 2.1. As introduced in the above key idea, we first cluster the D into multiple clusters (line 2-3), each of which $(C_t, t \in [1, T])$ contains similar items. Note that C_t denotes the set of indexes (*i.e.,* indexes in D) of these similar items. We will discuss how to cluster in Section 5.1.

Then, we pre-compute the relation between each item d_i (with feature vector $\mathbf{x}_i$) and each cluster C_t. To be specific, we compute the upper bound of the maximum feature distance between d_i and all items in C_t. As shown in line 3, the maximum feature distance (*i.e.,* the item-cluster distance introduced in Section 1) is denoted by $s^*_{it} = \max_{k \in C_t} s_{ik} = \|\mathbf{x}_i - \mathbf{x}_k\|, i \in [1, N], t \in [1, T]$. As items in each cluster are highly similar, s^*_{it} can well represent relation between each item and all these items. The reason why we use the upper bound u_{it} is that to compute s^*_{it}, we have to enumerate items in each cluster, which is time-consuming. But we can efficiently compute the upper bound, which can still well capture the item-cluster relationship. In Section 4, we will illustrate that using the upper bound to compute the coreset still has a theoretical guarantee *w.r.t.* the bounded GA error. Then in Section 5.2, we will discuss how to compute the upper bound efficiently.

[2. *First loop (lines 4-15)*] is similar to the typical solution (Section 2.1). In each iteration of the loop, we greedily add the most beneficial item (line 14-15) to the coreset.

[3. *Second loop (line 6-13)*] of the typical solution iterates all items in $D \setminus R$ to select the best one. To accelerate this, we can sample a subset $D_s \subseteq D$ to iterate. This sampling approach still has an approximate ratio $(1 - \frac{1}{e} - \epsilon)$ because of the sub-modular property [37], where ϵ is related to the sampling ratio.

[4. *Third loop (lines 9-13)*] is different from that of typical solution, but the objective is the same, which aims to compute the benefit of d_i being added in to current coreset R, denoted by $\mathcal{B}(d_i|R)$. As shown in line 13, the benefit is computed by the coreset score reduction, where $\mathcal{B}(R)$ and $\mathcal{B}(R \cup \{d_i\})$ respectively denote the coreset score of R and $R \cup \{d_i\}$. The main difference is that the typical solution computes the score considering all items in D, each of which is assigned to an item in R, as shown in Eq. 5. But we just consider the clusters, each of which is also assigned to an item in R (line 12) based on the upper bounds computed in the pre-processing steps, which is more efficient because $T \ll N$. We will discuss this step theoretically with more details in Section 4.

[5. *Weight computation (lines 16-17).*] In line 17, the weight λ_j of each item in the coreset equals to the sum of sizes of the clusters that are assigned to c_j (more details can be found in the example of Section 4). Afterwards, we can use the selected coreset to train the model. At each training iteration, when we use $c_j \in R$ to update the gradient, we should compute $\nabla f_{\gamma(j)}$ first, and then use $\lambda_j \nabla f_{\gamma(j)}$ to update the model parameter.

Advantages. Overall, FastCore is fast because (1) it does not need to enumerate every pair of feature distances as the typical solution does, and (2) in the third loop, FastCore just has to iterate the clusters rather than all items in D. Meanwhile, FastCore still has the theoretical guarantee about the gradient approximation.

Next, in Section 4, we will theoretically show why clustering D can bound the gradient and formally define the cluster-based coreset selection problem. Then we prove that the problem is NP-hard and has the sub-modular property. Finally, we give a concrete example of the problem solution (*i.e.,* Algorithm 1).

Remark. The above framework focuses on convex ML problems because the gradient difference should be bounded by Eq. 5. We can extend the framework to deep learning models in Section 6.

4 CLUSTERING-BASED BOUNDED GA ERROR

In this Section, we will theoretically show why Algorithm 1 works, followed by an example to help understanding.

Recap that from Eq. 3 and Eq. 4, we can get the GA error $\|\sum_{i=1}^{N}\nabla f_i(\omega) - \sum_{j=1}^{|R|}\lambda_j\nabla f_{\gamma(j)}(\omega)\|$ is bounded by $\sum_{i=1}^{N}\min_{c_j\in R}\|\mathbf{x}_i-\mathbf{x}_{\gamma(j)}\|$, leading to Eq. 5. After clustering D to $\{C_1, C_2, ...C_T\}$, we have:

$$\begin{aligned}\sum_{i=1}^{N}\min_{c_j\in R}\|\mathbf{x}_i-\mathbf{x}_{\gamma(j)}\| &= \sum_{t=1}^{T}\sum_{k\in C_t}\min_{c_j\in R}\|\mathbf{x}_k-\mathbf{x}_{\gamma(j)}\| \\ &\le \sum_{t=1}^{T}|C_t|\max_{k\in C_t}\min_{c_j\in R}\|\mathbf{x}_k-\mathbf{x}_{\gamma(j)}\| \\ &\le \sum_{t=1}^{T}|C_t|\min_{c_j\in R}\max_{k\in C_t}\|\mathbf{x}_k-\mathbf{x}_{\gamma(j)}\|\end{aligned} \tag{6}$$

In Eq. 6, given these clusters, we can first rewrite the sum of N feature distances (*i.e.,* $\min_{c_j\in R}\|\mathbf{x}_i-\mathbf{x}_{\gamma(j)}\|$) to the sum of T summations, each of which is the sum of $|C_t|$ such distances. Second, for each cluster, the sum can be bounded by the cluster size multiplying the maximum distance ($\max_{k\in C_t}\min_{c_j\in R}\|\mathbf{x}_k-\mathbf{x}_{\gamma(j)}\|$) in the cluster. However, computing this bound is also expensive because we have to iterate every item in each cluster, which is the same as iterating D. To overcome this, third, based on the max-min inequality [8], we have the last inequality in Eq. 6.

In this way, instead of iterating D, we can iterate the much smaller R to derive the maximum feature distances (*i.e.,* $s^*_{\gamma(j)t} = \max_{k\in C_t}\|\mathbf{x}_k-\mathbf{x}_{\gamma(j)}\|, j\in[1,|R|]$) between each $c_j\in R$ and items in each cluster C_t. Then, we can assign the cluster C_t to the item with the minimum distance, *i.e.,* $\min_{c_j\in R} s^*_{\gamma(j)t}$, which means $\forall k\in C_t, \phi(k)=j$.

Upper bound of the maximum feature distance. To achieve efficient coreset selection, given D and clusters C, we should pre-compute all the maximum feature distances $\{s^*_{it}|i\in[1,N], t\in[1,T]\}$. In this way, while selecting the coreset, we can directly get the value of $s^*_{\gamma(j)t}$ within the third loop. However, the time complexity of this pre-computation is still $O(mN^2)$, which is expensive for a large N. To address this, we propose to efficiently compute an upper bound u_{it} of s^*_{it} to replace s^*_{it}, so as to measure the relation between an item and a cluster. Theoretically, following Eq.6, we have:

$$\sum_{t=1}^{T}|C_t|\min_{c_j\in R}\max_{k\in C_t}\|\mathbf{x}_k-\mathbf{x}_{\gamma(j)}\| \le \sum_{t=1}^{T}|C_t|\min_{c_j\in R}u_{\gamma(j)t} \tag{7}$$

Our `Cluster-GA` problem. Overall, putting the above equations together, we can bound the GA error $\|\sum_{i=1}^{N}\nabla f_i(\omega) - \sum_{j=1}^{|R|}\lambda_j\nabla f_{\gamma(j)}(\omega)\|$ with $\sum_{t=1}^{T}|C_t|\min_{c_j\in R}u_{\gamma(j)t}$. Hence, to minimize the bound, we formally define the clustering-based GA error minimization problem (`Cluster-GA`) as below:

$$R^* = \arg\min_{R\subseteq D}\sum_{t=1}^{T}|C_t|\min_{c_j\in R}u_{\gamma(j)t},\ \text{s.t. } |R|\le K \tag{8}$$

Generally speaking, both Eq. 5 and Eq. 8 aims to select a coreset with the lowest coreset score, which denotes the bound of the GA error. The lower the score, the better the coreset can represent the full train data with similar gradients. However, the difference is that given the full data D and a coreset R, Eq. 5 computes the the score considering all items in D, which is time-consuming. Eq. 8 just needs to consider the clusters obtained from D, which is efficient because the number of clusters is much smaller than $N = |D|$. Similar to [38, 39, 41], we can easily prove that `Cluster-GA` is NP-hard with the sub-modular property, and thus we can leverage Algorithm 1 to address it with an approximate ratio. Finally, we give an example to illustrate the algorithm.

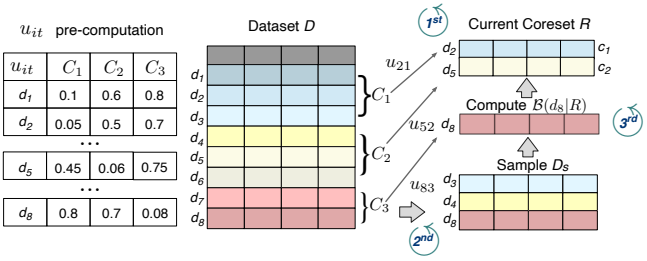

Figure 3: Illustration of Example 3.

Hence, we can use the greedy algorithm (Algorithm 1) to solve the `Cluster-GA` problem with an approximate ratio. We use an example in Figure 3 to help understand it.

Example 3. *Suppose that given D with 8 items, where similar colors indicate they have similar feature distances, we aim to select a coreset with $K = 3$.*

Pre-processing. *We first cluster D to $C = \{C_1, C_2, C_3\}$, and then pre-compute the upper bounds u_{it} between each $d_i \in D$ and the three clusters (line 3 of Algorithm 1). For example, as $d_5 \in C_2$, $u_{52} = 0.06$ is smaller than u_{51} and u_{53}. Then, we begin to select the coreset.*

First loop. *The right part of the figure shows the status that in the greedy algorithm, two items ($d_2(c_1)$ and $d_5(c_2)$) have been added into R, and we are in the third iteration of the first loop.*

Second loop. *Next, we begin the second loop with sampling $D_s \subseteq D\backslash R$ (line 6), and then iterate $D_s = \{d_3, d_4, d_8\}$. For each $d_i \in D_s$, we aim to compute its benefit ($\mathcal{B}(d_i|R)$) using the third loop, and select the one with the largest benefit to add into R (line 14-15).*

Third loop. *Within the third loop, taking $\mathcal{B}(d_8|R)$ as an example, we temporarily add d_8 to R and initialize the coreset score $\mathcal{B}(R\cup\{d_8\}) = 0$. Afterwards, we iterate C_1, C_2, C_3 to compute the score. For instance, for C_3, since $u_{83} < u_{53} < u_{23}$, we will assign C_3 to $d_8(c_3)$, i.e., $\phi(7) = 3, \phi(8) = 3$. Similarly, $C_1(C_2)$ is assigned to $d_2(d_5)$. Therefore, $\mathcal{B}(R\cup\{d_8\}) = u_{21}\times 3 + u_{52}\times 3 + u_{83}\times 2$ (line 12). Since $\mathcal{B}(R)$ has been computed at the last iteration of the first loop, we can then compute $\mathcal{B}(d_8|R) = \mathcal{B}(R) - \mathcal{B}(R\cup\{d_8\})$(line 13). Then, after the second loop, d_8 will be added since $\mathcal{B}(d_8|R)$ is larger than $\mathcal{B}(d_3|R)$ and $\mathcal{B}(d_4|R)$. Now, the coreset size is 3 and we have selected the items in the coreset.*

Weights computation. *Finally, we compute the weights of items in R. In this example, three items in C_1 are assigned to c_1, so $\lambda_1 = 3$. Similarly, $\lambda_2 = 3$ and $\lambda_3 = 2$.*

Convergence rate. In ML optimization, convergence rate indicates the speed of the ML algorithm to find the minimizer. The higher the rate, the smaller number of iterations the model required for convergence. It can be calculated by comparing the the parameter ω_k at the k-th iteration to the optimal parameter ω^*, and ω_{k+1} at the $(k+1)$-th iteration to ω^*.

Theorem 1. *The convergence rate of Algorithm 1 is at the same rate of $O(\frac{1}{\sqrt{k}})$ as the convergence rate for incremental gradient descent algorithms like SGD on D.*

The proof is shown in our technical report [3] due to the space limitation. Therefore, the convergence rate of `FastCore` is at the same as the rate for SGD on the full data. Hence, `FastCore` needs the same number of epochs to converge as training over the full data. Since the coreset is much smaller than the full train set, the efficiency is much improved.

5 PRE-PROCESSING STEPS

In this Section, we will discuss the important pre-processing steps in `FastCore`,*i.e.,* clustering and upper bound computation.

5.1 Clustering

To cluster D efficiently, we use the local sensitive hashing [5, 12] (LSH) to efficiently hash similar items to the same cluster. The basic idea is to hash D with p random hyperplanes $\{\mathbf{h}_1, \mathbf{h}_2, \ldots, \mathbf{h}_p\}$ in the m-dimensional space. Each hyperplane divides the entire space into two half subspaces, and LSH just hashes the items according to which subspace they fall in. Specifically, for hyperplane $\mathbf{h}_k, k \in [1, p]$, LSH hashes item $d_i \in D$ to hash value 1 if $\mathbf{h}_k \cdot \mathbf{x}_i > 0$, and 0 otherwise. In this way, each d_i is hashed into a p-bit 0-1 hash code by p hyperplanes. Since the items with the same hash code are highly similar, we take them as a cluster. Clustering by LSH is highly efficient as it has a complexity of $O(Npm)$, linear with $|D|$.

5.2 Upper Bound Computation

Given an item $d_i \in D$ and a cluster $C_t \in C$, recap that it is too expensive to compute s^*_{it}, so we compute an upper bound u_{it}. Next, we propose two approaches to address this, both of which rely on the key idea that splits the m dimension feature space into M low dimensional subspaces. The feature vector $\mathbf{x}_i$ of d_i is also splitted into M subvectors, each of which is denoted by $\mathbf{x}^l_i$. Then we respectively compute the upper bound of the maximum distance between l-th subvector and vectors of l-th feature subspace in $C_t, l \in [1, M]$, and sum them up, producing the upper bound u_{it} between d_i and C_t. Formally, following Eq. 7, we have:

$$\sum_{t=1}^{T} |C_t| \min_{c_j \in R} \max_{k \in C_t} \|\mathbf{x}_k - \mathbf{x}_{\gamma(j)}\| \le \sum_{t=1}^{T} |C_t| \min_{c_j \in R} \sum_{l=1}^{M} \max_{k \in C_t} \|\mathbf{x}^l_k - \mathbf{x}^l_{\gamma(j)}\| \tag{9}$$

we use $s^l_{\gamma(j)t} = \max_{k \in C_t} \|\mathbf{x}^l_k - \mathbf{x}^l_{\gamma(j)}\|$ to denote the aforementioned maximum feature distance *w.r.t.* the l-th subspace. We can take $u_{\gamma(j)t}$ as $\sum_{l=1}^{M} s^l_{\gamma(j)t}$. To compute $u_{\gamma(j)t}$, next, we discuss how to efficiently compute $s^l_{it}, i \in [1, N]$.

Convex hull based method. To efficiently compute s^l_{it}, we can compute a convex hull $H^l_t \subseteq C^l_t = \{\mathbf{x}^l_k | k \in C_t\}$, and obviously the subvector in C^l_t that has the largest feature distance with $\mathbf{x}^l_i$ must be in H^l_t, *i.e.,* $\arg\max_{\mathbf{x} \in C^l_t} \|\mathbf{x} - \mathbf{x}^l_i\| \in H^l_t$. Then we just have to iterate H^l_t and get $s^l_{it} = \arg\max_{\mathbf{x} \in H^l_t} \|\mathbf{x} - \mathbf{x}^l_i\|$, which will be used to compute the upper bound u_{it}. Assume an ideal case ($M = 1$) that we can directly compute the convex hull over D without dividing into subspaces, and thereby we can compute the best upper bound which is exactly s^*_{it}. However, it is prohibitively time-consuming to compute the convex hull over long vectors (when $\frac{m}{M} > 3$), so we have to divide it into short sub-vectors considering the efficiency. But unlike long vectors (*e.g.,* M is close to 1) can produce a bound close to s^*_{it}, short vectors always compute an upper bound that is likely to be much larger than s^*_{it} because the upper bound is computed by summing up all s^l_{it}, leading to inaccurate coreset scores.

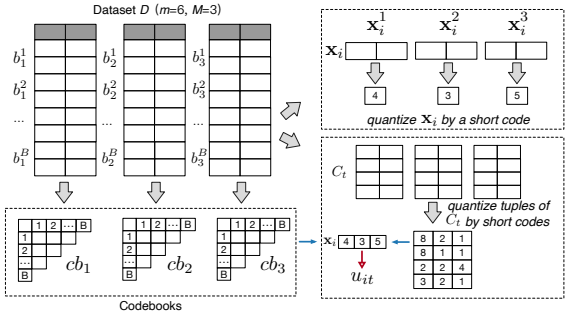

Figure 4: Example of Product Quantization Method.

Time complexity analysis. When $\frac{m}{M} \le 3$, the convex hulls can be computed in $O(M \sum_{t=1}^{T} |C_t| log(|C_t|)) = O(MNlogN)$ [42]. Then, to compute u_{it} for each $i \in [1, N]$, we need to iterate all H^l_t, totally taking $O(N \cdot \sum_{l=1}^{M} \sum_{t=1}^{T} \frac{m}{M} \cdot |H^l_t|) = O(mN \cdot \sum_{t=1}^{T} |H^l_t|)$. Since convex hulls always contain a small fraction of items, *i.e.,* $\sum_{t=1}^{T} |H^l_t| \ll N$, it is more efficient than the method in Section 2.1 with the complexity of $O(mN^2)$. However, in the worst case when $H^l_t = C^l_t$ holds for any $t \in [1, T]$ and $l \in [1, M]$, *i.e.,* all the items are on the convex hulls in each subspace, it obtains no efficiency improvement comparing with the brute-force method.

Product quantization (PQ) based method. To overcome the above issues, we propose a more efficient and effective PQ-based solution to estimate an upper bound. We first introduce the basic concept of PQ widely used in the era of approximate nearest neighbor (ANN) search, which also uses the idea of decomposing the entire feature space to a Cartesian product of subspaces with low dimensions [20]. Each subspace is quantized separately. Hence, a feature vector can be represented as a short code, and the l-th element of the code denotes the quantization index of the l-th subspace of the vector. Finally, in ANN search, the Euclidean distance between two items can be efficiently estimated from their codes. Next, we will see how to leverage PQ to address our problem.

As shown in Figure 4, given the full train data D with $m = 6$, suppose that we divide it into $M = 3$ subspaces, each of which has 2 dimensions (the upper left part of Figure 4). For each subspace, we can sample some items and run $k-$means algorithm [35] over this subspace. Suppose that B denotes the number of clusters produced by $k-$means, so in Figure 4, $\{b^1_1, b^2_1, ..., b^B_1\}$ denote the $k-$means cluster centers in the first subspace, associated with indexes (codes) $\{1, 2, ..., B\}$. Then, a codebook cb_1 is computed, which records the feature distance between each pair of cluster centers, and we have M codebooks in total (the lower left part of Figure 4). We use $cb_1[x][y]$ to denote the distance between the x-th and y-th centers. In this way, any $d_i(\mathbf{x}_i) \in D$ can be quantized to a short code q_i with length M, and we use $q^l_i, l \in [1, M]$ to denote the l-th code, which means

the q_i^l-th center is the closest one to $\mathbf{x}_i^l$ among all clusters in the l-th subspace (the upper right part of Figure 4). For example, $q_i^1 = 4$ because b_1^4 is the closest cluster with $\mathbf{x}_i^1$. For any d_i, d_j, their feature distance can be approximated by $s_{ij} = \sum_{l=1}^{M} cb_l[q_i^l][q_j^l]$.

Given d_i and C_t, as shown in the lower right part of Figure 4, we compute an approximation of s_{it}^l by first quantizing $\mathbf{x}_i$ and $\forall \mathbf{x} \in C_t$ to short codes. Then based on the codebooks, for the l-th subspace, we compute the largest distance between the corresponding code of d_i and codes of items in C_t, *i.e.,* $\hat{s}_{it}^l = \max_{k \in C_t} cb_l[q_i^l][q_k^l]$ as the approximation. Then we approximately compute an upper bound $u_{it} = \sum_{l=1}^{M} \hat{s}_{it}^l$ by summing the M distances up.

Although $\hat{s}_{it}^l$ may be a little smaller than s_{it}^l because of the quantization bias, the above summation u_{it} is always larger than s_{it}^* as each $\hat{s}_{it}^l$ is close to s_{it}^l. Besides, since the length of sub-vectors (*i.e.,* $\frac{m}{M}$) of the PQ-based method can be much longer than that of convex hull based method, the upper bound will be better (closer and larger than s_{it}^*) as longer sub-vectors result in more accurate distance computation, leading to a well-performed coreset.

Time complexity analysis. First, the cluster centers in PQ can be computed using only a small sampling of D, which takes negligible time. Then the codebooks can be computed in $O(mB^2)$ and the short codes of D can be computed in $O(mNB)$. Recap that for the l-th subspace, we compute the largest distance between the corresponding code of d_i and codes of items in C_t by $\hat{s}_{it}^l = \max_{k \in C_t} cb_l[q_i^l][q_k^l]$. Since q_i^l only takes from B different values $\{1, 2, \ldots, B\}$, we can precompute $\max_{k \in C_t} cb_l[j][q_k^l], j \in [1, B]$ for each C_t, which takes $O(TB^2)$. In this way, we can compute each $u_{it} = \sum_{l=1}^{M} \hat{s}_{it}^l$ in $O(M)$, and all the upper bounds u_{it}, $i \in [1, N]$, $t \in [1, T]$ can be computed in $O(MNT)$, much faster than the $O(mN^2)$ typical 3-loop method and the convex hull based method since $M \ll m$ and $T \ll N$.

Overall time complexity. Besides the above pre-processing steps, recap that the typical 3-loop algorithm has the complexity of $O(KN^2)$, and if we select the most beneficial item from D_s in the second loop, the complexity becomes $O(KSN)$, where $S = |D_s|$ denotes the sample size. For `FastCore`, the time complexity is $O(KST)$, which is more efficient because $T \ll N$. Hence, overall, `FastCore` is more efficient on both pre-processing steps and the 3-loop framework.

6 FASTCORE FOR DEEP LEARNING

As discussed above, Eq. 4 only applies to the convex ML problems, and thus we can achieve coreset selection without training in advance through the equation. But for deep neural networks (DNNs), how to bound the gradient difference becomes a challenge.

As studied in [22, 23], for DNNs, the gradient difference between two items can be efficiently bounded by the difference between gradients of the loss *w.r.t.* the input to the last network layer as:

$$\|\nabla f_i(\omega) - \nabla f_j(\omega)\| \leq \|\nabla f_i^{(L)}(\omega) - \nabla f_j^{(L)}(\omega)\| \cdot c_1 + c_2 \qquad (10)$$

where $\nabla f_i^{(L)}(\omega)$ denotes the gradient of loss of the L−th layer (the last layer) for d_i, and c_1, c_2 are constants. Note that the bound changes with parameter ω, so instead of selecting the coreset absolutely before training for convex problems, we have to iteratively update the coreset using Algorithm 1 with parameters updating.

In this case, we need to train for several epochs for deep learning. Specifically, we first initialize the model parameters and train few epochs over D. We can easily extract the gradient of the layer for each item. We then have the gradients for all items in D, regard these gradients as the feature vectors discussed in previous sections and run Algorithm 1 to compute a coreset. Afterwards, we train on the coreset for several epochs to get new parameters. Next, we feed forward items in D using the new parameters to get new gradients, so as tox update the coreset. We repeat the above steps for multiple times and a good coreset can be computed. Note that although we need iterative training in this scenario, the training is taken over the much smaller coreset, for just few epochs, so in general, it is still efficient. More details are illustrated in Appendix B.2.

7 RELATED WORK

Coresets can be used in many different data-driven problems [10, 11, 31–34] and computed in different ways. Some existing studies [13, 15, 24, 36] select the coreset during training. DeepFool [15] proposes to iteratively add items with low confidence (*i.e.,* close to the decision boundary) which are computed by the trained model into the coreset. Glister [24] iteratively selects the coreset that optimizes the generalization error using bi-level optimization. Huang et al. [13] also considered loss of training items in the coreset, which is used to approximate the overall training loss of the entire dataset. These method are not efficient enough because they need iterative training. Also, they are hard to achieve good performance because they generally use their own heuristic criteria to select the coreset, rather than directly optimizing the gradients closely related to the ML performance. There exist works [9, 17, 44, 46] that can select coresets without training in advance, but they are customized to particular model types (k−means [6, 17], SVM [44], Naive Bayes [46], Gaussian mixture model [30] and Bayesian inference [9] etc.). Gradient approximation methods [38, 39, 41] can achieve good performance because the selected weighted coreset can approximate the full gradient with theoretical guarantee, but they are not efficient because of the time-consuming multiple loops during the coreset selection process. In addition, Wang el al. [45] studied how to efficiently compute a coreset for feature augmentation by joining multiple tables. Chai el al. [16] studied how to select a good coreset over incomplete data.

Incremental gradient methods aim to accelerate the typical gradient decent algorithm that needs to iteratively compute the full gradient inefficiently. Most commonly-used methods are SGD and its variants (*e.g.,* momentum-based SGD [43], Adam [25], Adagrad [14] etc.). They provide an unbiased estimation for the full gradient by iteratively computing from subsets of the full data. `FastCore` can also run based on these incremental gradient methods with the same convergence rate, so our method is complementary to them.

8 EXPERIMENTS

8.1 Experimental Settings

Dataset. We evaluate on 5 real-world datasets with size varying from the magnitude of 10^5 to 10^7. Table 1 shows the statistics. (1) `Brazil` [1] is a multi-classification dataset to predict *the customer's review score of an order.* There are 98463 orders in total, each of which has 9 attributes (*e.g.,* price, freight value, etc.) with the label

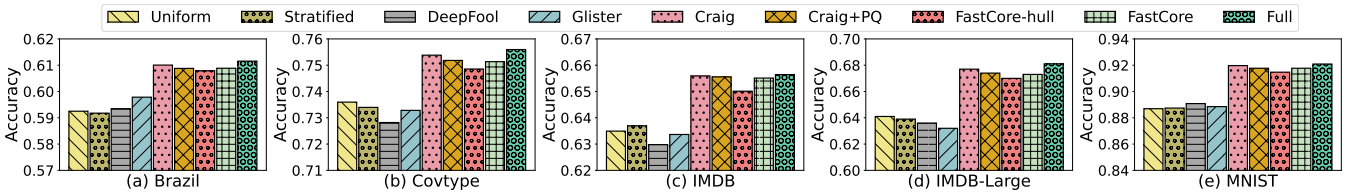

Figure 5: Effectiveness Comparison.

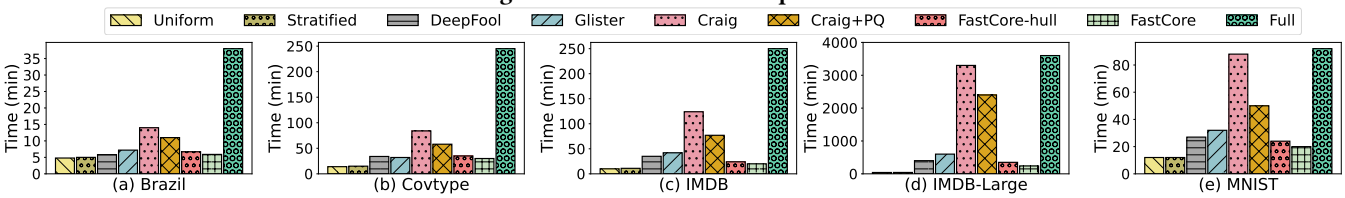

Figure 6: Efficiency Comparison.

Table 1: Statistics of datasets

Dataset	# of items	# of features	Type
Brazil	98,463	9	Tabular
CovType	581,012	54	Tabular
IMDB	674,466	40	Tabular
IMDB-Large	21,303,410	40	Tabular
MNIST	60,000	28×28	Image

Rating 1-5. (2) CovType [2] is to predict *forest cover type*. There are 581012 data points with 54 attributes. (3) IMDB [28] refers to a dataset that *predicts the rating (1-10) of movies*, which contains the basic information of movies, *e.g.,* movie_id, title, production_year. (4) IMDB-Large [28] is the large version of IMDB, which contains 21303410 records with the same attributes. (5) MNIST [27] is a typical image classification dataset to predict the handwritten digits.

Methods. We evaluate with the following 9 methods.

(1) Full refers to the baseline that trains over the full dataset D.

(2) Uniform samples K items uniformly as the training set.

(3) Stratified uses stratified sampling to sample K items in total from clusters produced by LSH.

(4) DeepFool [15] selects data points close to the decision boundary as a coreset.

(5) Glister [24] selects the coreset by optimizing its generalization error using bi-level optimization.

(6) Craig [38] uses the typical 3-loop gradient approximation method to select a coreset with size K.

(7) Craig+PQ is similar to Craig, but uses PQ to measure the feature distance between two items.

(8) FastCore-hull uses the convex hull to measure the distance between an item and a cluster.

(9) FastCore uses PQ to measure the item-cluster distance.

Hyper-parameter Setting. We train logistic regression for classification by default, where L2-regularization (coefficient=10^{-4}) and SGD with momentum of 0.9 are applied. The learning rate and batch size are tuned as hyperparameters independently for different methods and we decay the learning rate between epochs use *cosine annealing* [29]. S is set as 500. For FastCore-hull, we divide the feature space into 2-dimensional subspaces (*i.e.,* $\frac{m}{M} = 2$). For FastCore, we divide the feature space into $M = 3, 3, 4, 4$ and 4 subspaces for Brazil, CovType, IMDB, IMDB-Large and MNIST respectively. For each subspace, $B = 256$ clusters will be computed for quantization. More details are illustrated in Appendix B.1.

Evaluation Metrics. We mainly evaluate the efficiency and effectiveness of FastCore and other baselines. For efficiency, we report the end-to-end runtime including selecting the coreset and training. For effectiveness, we report the prediction accuracy.

8.2 Comparison with Baselines

8.2.1 Effectiveness. Figure 5 shows the accuracy of FastCore comparing with the baselines. We use $\rho = \frac{K}{N}$ to denote the proportion of coreset size to the full data. For Brazil, CovType andIMDB, we set $\rho = 1\%$. For MNIST, we set $\rho = 10\%$. For the larger one, IMDB-Large, we set $\rho = 0.1\%$. We can observe that Full achieves the best accuracy because it trains over the full data, but it is rather inefficient, which will be discussed in Section 8.2.2. But we can observe that Craig, Craig+PQ and FastCore are comparable to Full on all datasets. For example, the three methods and Full have an accuracy of around 61%, 75% and 65.5% on Brazil, CovType and IMDB respectively. The reason is that they use gradient approximation method to select the coreset that can approximate the full gradient of the full train data. Craig+PQ is competitive to Craig because the product quantization method can accurately estimate the feature distance. FastCore is also competitive because our cluster-based method contains similar items in each cluster and the gradient approximation error is also bounded. In addition, FastCore outperforms FastCore-hull because the product quantization method can provide a better estimation for the bound of maximum distance than the convex hull method. As shown in Table 2, we sample some item-cluster pairs from datasets and compute the mean relative error ($\frac{|u_{it} - s^*_{it}|}{s^*_{it}}$) of item-cluster distances. We can observe that PQ-based method (FastCore) computes a better bound closer to the ground truth (s^*_{it}) than FastCore-hull, and thus FastCore can compute a more accurate coreset score, leading to a good coreset. FastCore outperforms other baselines because it considers to approximate the full gradient, which is closely related to the training result, but other baselines just use their own simple heuristic ways. For example, Stratified focuses on selecting diverse set of training items to better capture the training data distribution, while DeepFool selects training items close to the decision boundary to

Table 2: Mean Relative Error of item-cluster distances.

	Brazil	CovType	IMDB	IMDB-Large
FastCore-hull	1.51	1.24	0.83	0.12
FastCore	0.02	0.12	0.17	0.03

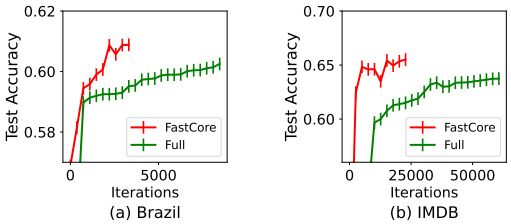

Figure 7: Convergence Analysis.

fine-tune the boundary. They are likely to be sub-optimal compared with us because they are lack of a guiding principle (*i.e.,* gradient approximation) to select the coreset.

8.2.2 Efficiency. In this part, we compare the efficiency of FastCore with baselines. As shown in Figure 6, Full takes the longest time because it trains over the large train data, and FastCore is always 10 times faster than Full. Craig and Craig+PQ are almost more efficient than Full because they train over the much smaller coreset. But for large datasets, like IMDB-Large (Figure 6(d)), Craig and Craig+PQ are inefficient because they have to enumerate every pair of training items with a high time complexity. Hence, that is the reason why we cluster the train set, and the time complexity is reduced. Overall, we can improve the efficiency $3-10\times$ comparing with Craig and Craig+PQ. FastCore is more efficient than FastCore-hull because computing the convex hull is time-consuming. Besides, Glister and DeepFool are less efficient because the need iterative training while selecting the coreset. Finally, Uniform and Stratified are very efficient obviously because they use simple heuristic, but fail to achieve high accuracy.

8.3 Convergence Evaluation

In this part, we evaluate the convergence of FastCore, comparing with Full. In Section 4, we have proved that the convergence rate of FastCore is the same as that of Full theoretically. In Figure 7, the X-axis denotes the number of iterations, and the Y-axis denotes the test accuracy. We can observe that training on the coreset converges much faster than training on the full data. As Full still has to take a number of iterations to converge, we do not plot the entire process in the figure. For example, on dataset IMDB, FastCore takes 2.5×10^4 iterations to converge, but Full takes 2×10^6 iterations. The reason is that they have the same convergence rate, so they spend similar number of epochs converging. Since the coreset is much smaller than the full data, FastCore just needs a smaller number of iterations to converge.

8.4 Deep Learning Evaluation

In this part, we test how FastCore performs comparing with baselines when the downstream model is deep learning. As shown in Figures 8 and 9, we use multilayer perceptron (MLP) for datasets

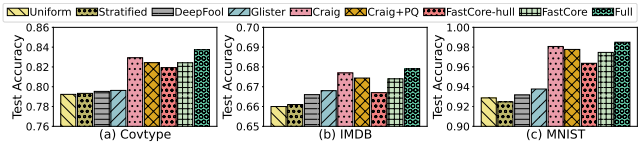

Figure 8: Effectiveness Comparison for DNN.

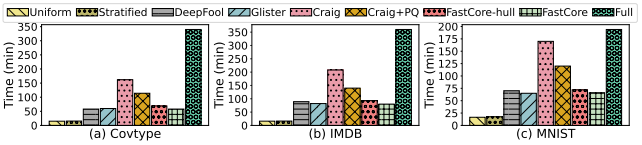

Figure 9: Efficiency Comparison for DNN.

CovType and IMDB, convolutional neural network (CNN) for MNIST to evaluate the effectiveness and efficiency. The MLP model consists of 2 hidden linear layers with 128 nodes and a softmax output layer. The CNN model consists of a single convolutional layer with 32 (3,3) filters followed by a max pooling layer. The output is then flattend and connected to the softmax output layer by a linear layer with 128 nodes. For both models, ReLU is used as the activation function, and L2 regularization with a coefficient=10^{-4} is used. Cross entropy is used as the loss function for training. We can observe that for effectiveness (Figure 8), FastCore also performs well, *i.e.,* competitively with Full and gradient approximation baselines, and much faster than them (Figure 9) for efficiency. This demonstrates that we can also efficiently approximate the gradient to get a good coreset for deep neural networks.

For the extensive ablation studies with respect to coreset size, ML model types, clustering methods, etc, please refer to the Appendix.

9 CONCLUSION

In this paper, we study an efficient coreset selection framework FastCore using the cluster-based method to accelerate ML training. Specifically, we first split the large full training set into multiple clusters, and then efficiently compute the bound of maximum feature distances between each cluster and item using PQ. Afterwards, a typical 3-loop coreset selection is applied to judiciously select a coreset that can well approximate the full gradient of original train set. Experimental results show that FastCore is more efficient than other baselines without much sacrificing the effectiveness.

Limitations and Opportunities. Despite the efficacy and time efficiency our method is limited to ML problems optimized by gradient descent algorithms and cannot handle other models such as decision trees, KNN, and XGBoost. It is crucail to design a coreset selection method that supports more model types or different methods customized to different models. Furthermore, in reality, data may contain noise, which can lead to inaccurate distance computations and a bad coreset. Therefore, it is important to investigate a coreset selection framework that can handle noisy data.

ACKNOWLEDGMENTS

Chengliang Chai is supported by NSF of China (62102215). Ye Yuan is supported by the National Key R&D Program of China (2022YFB2702100), the NSFC (61932004, 62225203, U21A20516) and the DITDP (JCKY2021211B017). Guoren Wang is supported by the NSFC (61732003, U2001211). Ye Yuan is the corresponding author.

REFERENCES

[1] https://www.kaggle.com/datasets/olistbr/brazilian-ecommerce/, 2023. Accessed: 2023-02-01.

[2] https://archive.ics.uci.edu/ml/datasets/covertype, 2023. Accessed: 2023-02-01.

[3] Effcient coreset selection with cluster-based methods [technical report]. https://github.com/for0nething/FCTR, 2023. Last accessed: 2023-06-03.

[4] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, et al. Tensorflow: a system for large-scale machine learning. In *Osdi*, volume 16, pages 265–283. Savannah, GA, USA, 2016.

[5] A. Andoni and P. Indyk. Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions. *Commun. ACM*, 51(1):117–122, 2008.

[6] O. Bachem, M. Lucic, and A. Krause. Scalable k -means clustering via lightweight coresets. In Y. Guo and F. Farooq, editors, *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining, KDD 2018, London, UK, August 19-23, 2018*, pages 1119–1127. ACM, 2018.

[7] C. B. Barber, D. P. Dobkin, and H. Huhdanpaa. The quickhull algorithm for convex hulls. *ACM Trans. Math. Softw.*, 22(4):469–483, 1996.

[8] S. Boyd, S. P. Boyd, and L. Vandenberghe. *Convex optimization*. Cambridge university press, 2004.

[9] T. Campbell and T. Broderick. Bayesian coreset construction via greedy iterative geodesic ascent. In *ICML 2018*, volume 80 of *Proceedings of Machine Learning Research*, pages 697–705. PMLR, 2018.

[10] C. Chai, L. Cao, G. Li, J. Li, Y. Luo, and S. Madden. Human-in-the-loop outlier detection. In *SIGMOD Conference 2020*, pages 19–33. ACM, 2020.

[11] C. Chai, J. Liu, N. Tang, G. Li, and Y. Luo. Selective data acquisition in the wild for model charging. *Proc. VLDB Endow.*, 15(7):1466–1478, 2022.

[12] M. Charikar. Similarity estimation techniques from rounding algorithms. In J. H. Reif, editor, *Proceedings on 34th Annual ACM Symposium on Theory of Computing, May 19-21, 2002, Montréal, Québec, Canada*, pages 380–388. ACM, 2002.

[13] C. Coleman, C. Yeh, S. Mussmann, B. Mirzasoleiman, P. Bailis, P. Liang, J. Leskovec, and M. Zaharia. Selection via proxy: Efficient data selection for deep learning. In *ICLR 2020*. OpenReview.net, 2020.

[14] J. C. Duchi, E. Hazan, and Y. Singer. Adaptive subgradient methods for online learning and stochastic optimization. *J. Mach. Learn. Res.*, 12:2121–2159, 2011.

[15] M. Ducoffe and F. Precioso. Adversarial active learning for deep networks: a margin based approach. *CoRR*, abs/1802.09841, 2018.

[16] C. C. et al. Goodcore: Data-effective and data-efficient machine learning through coreset selection over incomplete data. In *SIGMOD*. ACM, 2023.

[17] S. Har-Peled and S. Mazumdar. On coresets for k-means and k-median clustering. In L. Babai, editor, *Proceedings of the 36th Annual ACM Symposium on Theory of Computing, Chicago, IL, USA, June 13-16, 2004*, pages 291–300. ACM, 2004.

[18] T. Hofmann, A. Lucchi, S. Lacoste-Julien, and B. McWilliams. Variance reduced stochastic gradient descent with neighbors. In *Advances in Neural Information Processing Systems 28: Annual Conference on Neural Information Processing Systems 2015, December 7-12, 2015, Montreal, Quebec, Canada*, pages 2305–2313, 2015.

[19] J. Huang, R. Huang, W. Liu, N. M. Freris, and H. Ding. A novel sequential coreset method for gradient descent algorithms. In *ICML 2021*, volume 139 of *Proceedings of Machine Learning Research*, pages 4412–4422. PMLR, 2021.

[20] H. Jegou, M. Douze, and C. Schmid. Product quantization for nearest neighbor search. *IEEE transactions on pattern analysis and machine intelligence*, 33(1):117–128, 2010.

[21] J. Johnson, M. Douze, and H. Jégou. Billion-scale similarity search with GPUs. *IEEE Transactions on Big Data*, 7(3):535–547, 2019.

[22] A. Katharopoulos and F. Fleuret. Not all samples are created equal: Deep learning with importance sampling. In *ICML 2018*, volume 80 of *Proceedings of Machine Learning Research*, pages 2530–2539. PMLR, 2018.

[23] K. Killamsetty, D. Sivasubramanian, G. Ramakrishnan, A. De, and R. K. Iyer. GRAD-MATCH: gradient matching based data subset selection for efficient deep model training. In M. Meila and T. Zhang, editors, *Proceedings of the 38th International Conference on Machine Learning, ICML 2021, 18-24 July 2021, Virtual Event*, volume 139 of *Proceedings of Machine Learning Research*, pages 5464–5474. PMLR, 2021.

[24] K. Killamsetty, D. Sivasubramanian, G. Ramakrishnan, and R. K. Iyer. GLISTER: generalization based data subset selection for efficient and robust learning. In *AAAI 2021*, pages 8110–8118. AAAI Press, 2021.

[25] D. P. Kingma and J. Ba. Adam: A method for stochastic optimization. In Y. Bengio and Y. LeCun, editors, *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, 2015.

[26] K. Kirchhoff and J. A. Bilmes. Submodularity for data selection in machine translation. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing, EMNLP 2014, October 25-29, 2014, Doha, Qatar, A meeting of SIGDAT, a Special Interest Group of the ACL*, pages 131–141. ACL, 2014.

[27] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *Proc. IEEE*, 86(11):2278–2324, 1998.

[28] V. Leis, A. Gubichev, A. Mirchev, P. A. Boncz, A. Kemper, and T. Neumann. How good are query optimizers, really? *Proc. VLDB Endow.*, 9(3):204–215, 2015.

[29] I. Loshchilov and F. Hutter. SGDR: stochastic gradient descent with warm restarts. In *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*. OpenReview.net, 2017.

[30] M. Lucic, M. Faulkner, A. Krause, and D. Feldman. Training gaussian mixture models at scale via coresets. *J. Mach. Learn. Res.*, 18:160:1–160:25, 2017.

[31] Y. Luo, C. Chai, X. Qin, N. Tang, and G. Li. Interactive cleaning for progressive visualization through composite questions. In *36th IEEE International Conference on Data Engineering, ICDE*, pages 733–744. IEEE, 2020.

[32] Y. Luo, X. Qin, N. Tang, and G. Li. Deepeye: Towards automatic data visualization. In *34th IEEE International Conference on Data Engineering, ICDE 2018, Paris, France, April 16-19, 2018*, pages 101–112, 2018.

[33] Y. Luo, N. Tang, G. Li, C. Chai, W. Li, and X. Qin. Synthesizing natural language to visualization (NL2VIS) benchmarks from NL2SQL benchmarks. In G. Li, Z. Li, S. Idreos, and D. Srivastava, editors, *SIGMOD '21: International Conference on Management of Data, China, June 20-25, 2021*, pages 1235–1247. ACM, 2021.

[34] Y. Luo, N. Tang, G. Li, J. Tang, C. Chai, and X. Qin. Natural language to visualization by neural machine translation. *IEEE Trans. Vis. Comput. Graph.*, 28(1):217–226, 2022.

[35] J. MacQueen. Classification and analysis of multivariate observations. In *5th Berkeley Symp. Math. Statist. Probability*, pages 281–297. University of California Los Angeles LA USA, 1967.

[36] K. Margatina, G. Vernikos, L. Barrault, and N. Aletras. Active learning by acquiring contrastive examples. In *EMNLP 2021*, pages 650–663. Association for Computational Linguistics, 2021.

[37] B. Mirzasoleiman, A. Badanidiyuru, A. Karbasi, J. Vondrák, and A. Krause. Lazier than lazy greedy. In *Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence, January 25-30, 2015, Austin, Texas, USA*, pages 1812–1818. AAAI Press, 2015.

[38] B. Mirzasoleiman, J. A. Bilmes, and J. Leskovec. Coresets for data-efficient training of machine learning models. In *Proceedings of the 37th International Conference on Machine Learning, ICML 2020, 13-18 July 2020, Virtual Event*, volume 119 of *Proceedings of Machine Learning Research*, pages 6950–6960. PMLR, 2020.

[39] B. Mirzasoleiman, K. Cao, and J. Leskovec. Coresets for robust training of deep neural networks against noisy labels. In *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual*, 2020.

[40] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, et al. Pytorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems*, 32, 2019.

[41] O. Pooladzandi, D. Davini, and B. Mirzasoleiman. Adaptive second order coresets for data-efficient machine learning. In *International Conference on Machine Learning, ICML 2022, 17-23 July 2022, Baltimore, Maryland, USA*, volume 162 of *Proceedings of Machine Learning Research*, pages 17848–17869. PMLR, 2022.

[42] F. P. Preparata and S. J. Hong. Convex hulls of finite sets of points in two and three dimensions. *Communications of the ACM*, 20(2):87–93, 1977.

[43] N. Qian. On the momentum term in gradient descent learning algorithms. *Neural Networks*, 12(1):145–151, 1999.

[44] M. Tukan, C. Baykal, D. Feldman, and D. Rus. On coresets for support vector machines. *Theor. Comput. Sci.*, 890:171–191, 2021.

[45] J. Wang, C. Chai, N. Tang, J. Liu, and G. Li. Coresets over multiple tables for feature-rich and data-efficient machine learning. *Proc. VLDB Endow.*, 16(1):64–76, 2022.

[46] K. Wei, R. K. Iyer, and J. A. Bilmes. Submodularity in data subset selection and active learning. In *ICML 2015*, volume 37 of *JMLR Workshop and Conference Proceedings*, pages 1954–1963. JMLR.org, 2015.

[47] Z. A. Zhu, Y. Yuan, and K. Sridharan. Exploiting the structure: Stochastic gradient methods using raw clusters. In *Advances in Neural Information Processing Systems 29: Annual Conference on Neural Information Processing Systems 2016, December 5-10, 2016, Barcelona, Spain*, pages 1642–1650, 2016.

A VARIANCE EVALUATION

A.0.1 Coreset size. In this part, we test the performance of varying the coreset size. As shown in Figure 10, we vary ρ from 0.1% to 10% on dataset `IMDB`, `Brazil` and `CovType`. We can observe that with the size increasing, the accuracy increases first and then remains stable, which demonstrates that a small coreset can achieve good performance because of the gradient approximation. In practice, one may consider how to select an appropriate coreset size. Here, we propose a simple yet effective solution. We begin with a very small size, *e.g.*, $\rho = 0.1\%$, and train over the coreset to test the performance. Then we enlarge the coreset like for 10 times and test again. This repeats until the performance remains stable. Since the coreset size increases exponentially, it just needs few iterations to be stable. This is also efficient because in each iteration, we just train over a small coreset.

A.0.2 ML models. We test different convex models for an ablation study. As shown in Figure 11, we respectively train on logistic regression, SVM and Lasso. We can observe that although different models may perform differently, the performance difference between `FastCore` and `Full` of the same model is not significant. Hence, `FastCore` can select the coreset that approximates the full gradient accurately regardless of user-specified ML models.

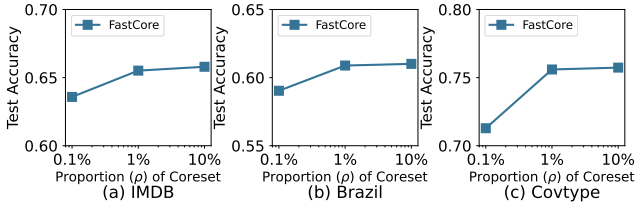

Figure 10: Varying Coreset Size.

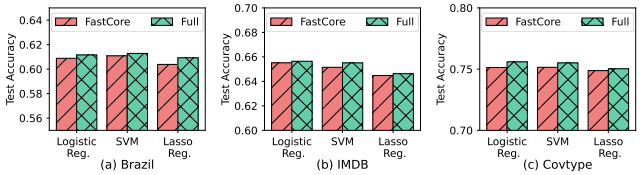

Figure 11: Varying ML models.

A.0.3 Hyperparameters in LSH. We test the performance of varying the number of hyperplanes in LSH. As shown in Figure 12(a), with the number increasing, more clusters are generated, and the items within each cluster are closer, so the accuracy increases at the beginning. Afterwards, the accuracy remains stable because items in each cluster are similar enough for gradient approximation. Therefore, empirically, using about 64 hyperplanes is appropriate because more clusters will reduce the efficiency. As shown in Table 4, we found that when the hyperplane number achieves 64, the average score is above 0.82 on all datasets, which also verifies our empirical finding. Besides, We also test the consuming time of this hyperparameter search, as show in Table 5, we found that the hyper-parameter search time can be ignored compared with the entire process.

A.0.4 Hyperparameters in PQ. We test the performance of varying $\frac{m}{M}$ for PQ. In Figure 12(b)-(c), at the beginning, $\frac{m}{M} = 1$ means that in each subspace, the length of all sub-vectors is 1. With $\frac{m}{M}$ increasing, the accuracy increases first because each sub-vector is longer, which keeps more information when adding up these $\hat{s}_{it}^{l}$, leading to a more precise bound. But if each sub-vector is too long, which means that each vector is quantized to a very short code, the accuracy decreases because in this situation, the PQ method is not informative enough to give accurate Euclidean distance estimation. Empirically, $M \approx 3$ is always an appropriate choice.

A.0.5 Cluster Methods. We test other typical clustering methods such as k-means and the Gaussian mixture model (GMM) and find that they achieve similar model performance compared to LSH which we used in our framework. However, as show in Table 6, we set the number of clusters for k-means and GMM to be the same as in our setting, we can see that these clustering methods have much higher time complexity than LSH and therefore are not suitable for our scenario with high-efficiency requirements on three datasets.

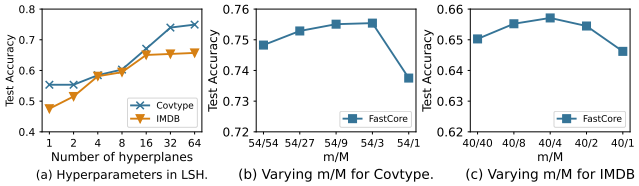

Figure 12: Varying Hyperparameters in LSH and PQ.

B IMPLEMENTATION DETAILS.

B.1 More Experimental Settings

Implementation. For `FastCore-hull`, we build the convex hulls using the QHull library [7]. For `FastCore` and `Craig+PQ`, we use the FAISS [21] library for Product Quantization. We split the each dataset to 75%/25% as train/test set, and we select the coreset based on the 75% train set. The source code is available at the anonymous link https://github.com/AnonymousCodesRepo/FastCore.

Environment. All experiments were implemented in Python, performed on a Ubuntu Server with an Intel Xeon Silver 4110 2.10GHz CPU having 32 cores and 128GB DDR4 main memory without SSD.

Detailed Hyperparameters. Detailed hyperparameters used in Section 8.2 of `FastCore` are shown in Table 3. We also include the dimensionality and the data size of each dataset for ease of reference. By default, the number of training epochs is set to 40 for each dataset. We train each model using SGD with momentum of 0.9. The hyperparameters for each dataset are tuned independently, and their optimal values are shown in the table. We decay the learning rate between epochs using cosine annealing schedule. For `FastCore` and `FastCore-hull`, their hyperparameters are set as the empirical optimal values as discussed in Section A. In Section 8.2, since `Glister` and `DeepFool` are methods that select coresets with training, we update their selected coresets every 10 epochs for a fair comparison.

Table 3: Hyperparameters of FastCore in Section 8.2.

	Brazil	CovType	IMDB	IMDB-Large	MNIST
DIMENSION	9	54	40	40	28× 28
DATA SIZE	98,463	581,012	674,466	21,303,410	60,000
TRAINING EPOCHS	40	40	40	40	40
LEARNING RATE	0.1	0.1	0.1	0.1	0.01
NUMBER of HYPERPLANES	48	64	48	48	64
NUMBER of SUBSPACES	3	3	4	4	4
NUMBER of CLUSTERS in PQ	256	256	256	256	256

Algorithm 2: FastCore for Deep Learning

Input: D, K, deep model $\mathcal{M}(\omega)$, κ.
Output: A coreset $R \subseteq D$, weight $W = \{\lambda_j\}, |R| = |W| = K$.

1 $R = \emptyset$;
2 $count = 0$;
3 Train $\mathcal{M}(\omega)$ over D for few epochs;
4 **for** *each item* $d_i \in D$ **do**
5 $\quad \nabla f_i^{(L)}(\omega)$ = ComputeGradient($\mathcal{M}(\omega)$, d_i);
6 Taking $\{\nabla f_i^{(L)}(\omega) | i \in [1, N]\}$ as input, call Algorithm 1 to compute a coreset and the weight ($\mathcal{M}(\omega)$, R, W);
7 **while** $count < \kappa$ **do**
8 $\quad \mathcal{M}(\omega)$ = Train_with_Coreset($\mathcal{M}(\omega)$, R, W);
9 $\quad$ **for** *each item* $d_i \in D$ **do**
10 $\quad\quad \nabla f_i^{(L)}(\omega)$ = ComputeGradient($\mathcal{M}(\omega)$, d_i);
11 $\quad$ (R, W) = Call Algorithm 1 to update the coreset;
12 $\quad count+ = 1$;
13 **return** R, W;

Table 4: The Relationship Between Hyperparameter Number and Average Score.

# Hyperplanes	Brazil	CovType	IMDB	IMDB-Large	MNIST
8	0.15	0.08	0.22	0.19	0.14
16	0.57	0.56	0.49	0.46	0.53
32	0.81	0.80	0.83	0.79	0.80
64	0.82	0.84	0.85	0.83	0.82
128	0.83	0.85	0.85	0.84	0.82

Table 5: Efficiency of Hyper-Parameter Search.

	Brazil	CovType	IMDB	IMDB-Large	MNIST
Hyper-parameter search time (s)	0.57	1.84	1.46	11.88	1.59
Total time (s)	355	1802	1217	14412	1240

Table 6: Cluster Time and Test Accuracy of IMDB.

Method	Cluster Time (s)	Test Accuracy
K-MEANS	2898	65.7
GMM	3791	65.4
LSH	1.35	65.7

B.2 Algorithm of FastCore for Deep Learning

Algorithm 2 shows the FastCore framework for deep learning, which takes the original train data D and a deep learning model with initialized parameters as input. First, it trains over D for few epochs (line 3). In practice, 20 epochs are used. Then we calculate the gradient ($\nabla f_i^{(L)}(\omega)$) of loss of the last layer for each d_i (line 5). Then we take these gradients as the feature vectors of all items in D, and trigger Algorithm 1 to compute an initial coreset (line 6). Afterwards, we train with the coreset to update the model parameters (line 8), iterate D to update the gradients (line 10) and call Algorithm 1 to update the coreset (line 11) for κ times. In practice, we select a new coreset based on the updated model parameters every 20 epochs and train with $\kappa = 10$, *i.e.,* train the network on coreset for 200 epochs in total.