A Scalable Algorithm for Active Learning

Youguang Chen
University of Texas at Austin
Austin, USA
youguang@utexas.edu

Zheyu Wen
University of Texas at Austin
Austin, USA
zheyw@utexas.edu

George Biros
University of Texas at Austin
Austin, USA
gbiros@acm.org

***Abstract*—FIRAL is a recently proposed deterministic active learning algorithm for multiclass classification using logistic regression. It was shown to outperform the state-of-the-art in terms of accuracy and robustness and comes with theoretical performance guarantees. However, its scalability suffers when dealing with datasets featuring a large number of points n, dimensions d, and classes c, due to its $\mathcal{O}(c^2d^2 + nc^2d)$ storage and $\mathcal{O}(c^3(nd^2 + bd^3 + bn))$ computational complexity where b is the number of points to select in active learning. To address these challenges, we propose an approximate algorithm with storage requirements reduced to $\mathcal{O}(n(d+c) + cd^2)$ and a computational complexity of $\mathcal{O}(bncd^2)$. Additionally, we present a parallel implementation on GPUs. We demonstrate the accuracy and scalability of our approach using MNIST, CIFAR-10, Caltech101, and ImageNet. The accuracy tests reveal no deterioration in accuracy compared to FIRAL. We report strong and weak scaling tests on up to 12 GPUs, for three million point synthetic dataset.**

***Index Terms*—Active learning, contrastive learning, GPU acceleration, iterative solvers, randomized linear algebra, message passing interface, performance analysis**

I. Introduction

Let $\mathbf{X}_o$ be a set of labeled points and $\mathbf{X}_u$ a set of n unlabeled points, both sets sampled from the same distribution. We denote a labeled sample as a pair (x, y), where $x \in \mathbb{R}^d$ is a point and $y \in \{1, 2, \cdots, c\}$ is its label, where c is the number of classes. Our goal of *active learning* is to select b points from $\mathbf{X}_u$ to label and use them along with pairs in $\mathbf{X}_o$ to train a multiclass logistic regression classifier.

Labeling data can be costly, but recent advancements in unsupervised and representation learning [1] enable us to leverage pre-existing feature embeddings combined with shallow learning techniques like logistic regression to develop efficient classification methods [2], [3]. The question is how to select training samples. Active learning addresses this issue by focusing on sample selection [4]. Basic and popular sample selection methods include random sampling and k-means clustering. While these methods are scalable and easy to implement, they can be suboptimal and exhibit high variability due to their inherent randomness, particularly when the labeling budget is limited. We are seeking a method that is scalable, has low variability, and provides accuracy guarantees.

We propose a method for solving this problem based on the FIRAL algorithm (Fisher Information Ratio Active Learning) that appeared in 2023 [5]. FIRAL is an active learning algorithm with theoretical guarantees that outperforms the state of the art in terms of accuracy. However, FIRAL has high complexity due to dense computations. Here we propose an approximate algorithm that dramatically accelerates FIRAL. We dub the new algorithm Approx-FIRAL. A cornerstone in FIRAL is the Fisher information matrix, which is the *Hessian of a negative log-likelihood loss function*. In Approx-FIRAL we exploit the structure of the Hessian and we introduce the following: a matrix-free matrix-vector multiplication "matvec", a preconditioner, randomized trace estimators, and a modified regret minimization scheme. Overall the new components dramatically improve the complexity of the scheme. Combined with GPU and distributed memory parallelism Approx-FIRAL enables active learning for datasets that were intractable for FIRAL. Our contributions can be summarized as follows:

- We exploit structure, randomized linear algebra, and iterative methods to accelerate FIRAL.
- Using Python and CuPy [6],and MPI [7], [8] we support multi-GPU acceleration. Our Python code is open-sourced.
- We compare the accuracy of Approx-FIRAL with the exact FIRAL algorithm as well as several other popular active learning methods; and we test its scalability on multi-GPU systems.

We further test the sensitivity of the method on different input parameters like the dataset size and the number of classes. Overall Approx-FIRAL is orders of magnitude faster that FIRAL without any noticeable difference in accuracy. While FIRAL is limited to datasets with a few thousands of points and up to 50 classes we demonstrate scalability to ImageNet 1.3 million points and 1000 classes, as well as synthetic datasets with several million points.

Related work: There is a substantial body of work on active learning, including approaches such as uncertainty estimation [9], sample diversity [10], [11], [12], Bayesian inference [13], [14], and others. However, these methods lack performance guarantees. FIRAL provides lower and upper bounds of the generalization error for a multinomial logistic regression classifier assuming that the input points follow a sub-Gaussian distribution. It uses convex relaxation (RELAX step), similar to compressed sensing, to first compute weights for each point in $\mathbf{X}_u$ and then uses regret minimization to select b points (ROUND step). Regarding parallel algorithms and GPU implementations, there are many implementations of random sampling and k-means and related combinations but

SC24, November 17-22, 2024, Atlanta, Georgia, USA
979-8-3503-5291-7/24/$31.00 ©2024 IEEE

nothing related to FIRAL-like algorithms.

Outline of the paper: We start with the formulation of FIRAL in § II. We summarize the RELAX step in § II-B and the ROUND step in § II-C. The storage and computational complexity of FIRAL are summarized in § II-D. We introduce Approx-FIRAL in § III: the Hessian structure and the accelerated RELAX step are described in § III-A; and the ROUND solve is described in § III-B. The HPC implementation and complexity analysis are described in § III-C. We report results from numerical experiments in § IV: accuracy and comparisons with other active learning methods are reported in § IV-A; and single and multi-GPU performance results are reported in § IV-B and § IV-C respectively.

II. THE EXACT FIRAL ALGORITHM

A. Formulation

A summary of the main notation used in the paper can be found in Table I. We consider the batch active learning problem with given initial labeled points $\mathbf{X}_o$ and a pool of n unlabeled points $\mathbf{X}_u$. We denote a labeled sample as a pair (x, y), where $x \in \mathbb{R}^d$ is a data point, $y \in \{1, 2, \cdots, c\}$ is its label, and c is the number of classes. We use a multiclass logistic regression model as our classifier. Given x and classifier weights $\theta \in \mathbb{R}^{d\times(c-1)}$, the likelihood of a point x having label y is defined by

$$p(y|x,\theta) = \begin{cases} \frac{\exp(\theta_y^\top x)}{1+\sum_{l\in[c-1]}\exp(\theta_l^\top x)}, & y \in [c-1] \\ \frac{1}{1+\sum_{l\in[c-1]}\exp(\theta_l^\top x)}, & y = c. \end{cases} \tag{1}$$

We denote the vector of all class probabilities for point x by $h(x) \in \mathbb{R}^{c-1}$, with $h_i = p(y = i|x)$. To simplify notation we define $\widetilde{d} = d(c-1)$. The weights θ are found by minimizing the negative log-likelihood: $\ell_{(x,y)}(\theta) \triangleq -\log p(y|x,\theta)$. The Hessian or Fisher information matrix at x is defined by $\mathbf{H}_i := \partial_{\theta\theta}\ell_{(x,y)} \in \mathbb{R}^{\widetilde{d}\times\widetilde{d}}$ and for our classifier is given by

$$\mathbf{H}_i = [\mathrm{diag}(h_i) - h_i h_i^\top] \otimes (x_i x_i^\top). \tag{2}$$

Let $\mathbf{H}_o$ be the summation of Hessians of the initial labeled points, $\mathbf{H}_p$ of the unlabeled points, and $\mathbf{H}_z$ of *weighted* unlabeled points with weights $z \in \mathbb{R}^n$, i.e.

$$\mathbf{H}_o \triangleq \sum_{i\in\mathbf{X}_o}\mathbf{H}_i, \quad \mathbf{H}_p \triangleq \sum_{i\in\mathbf{X}_u}\mathbf{H}_i, \quad \mathbf{H}_z \triangleq \sum_{i\in\mathbf{X}_u} z_i\mathbf{H}_i. \tag{3}$$

Then given a budget of b points to sample (from $\mathbf{X}_u$), an optimal way would be to minimize the Fisher Information Ratio [5]:

$$\operatorname*{arg\,min}_{z\in\{0,1\}^n, \|z\|_1=b} \left(\mathbf{H}_o + \mathbf{H}_z\right)^{-1} \cdot \mathbf{H}_p \triangleq f(z). \tag{4}$$

where "$\cdot$" represents the matrix inner product. Unfortunately, this is an NP-hard combinatorial convex optimization problem. FIRAL proposed an algorithm to solve this problem with near-optimal performance guarantees. The algorithm is composed of two parts: a RELAX step of continuous convex relaxation optimization followed by a ROUND step to select b points.

Table I *Summary of notation.*

Notation	Description
d, c	dimension of point, number of classes
$\widetilde{d}$	$d\,c$
b	budget: number of points to select for labeling
n	number of points in unlabeled pool
$\otimes$	matrix Kronecker product
$\odot$	element-wise multiplication between two vectors
$vec(\cdot)$	vectorization of a matrix by stacking its columns
$\mathbf{X}_o, \mathbf{X}_u$	index sets for initial labeled points and unlabeled points
$\mathbf{H}_i$	Fisher information matrix for point i (Eq. (2))
$\mathbf{H}_o, \mathbf{H}_p, \mathbf{H}_z$	(weighted) sum of Hessians (Eq. (3))
$\mathbf{\Sigma}_z$	sum of Hessians on selected points (Eq. (7))
$f(z)$	objective function (Eq. (4))
g_i	gradient of relaxed objective (Eq. (6))
$\mathcal{B}(\cdot)$	block diagonal operation (Definition 1)
$z_\diamond$	solution of relaxed problem (Eq. (5))
$\widetilde{\cdot}$	matrix transformation (Eq. (8))
η	learning rate in round solver
$\mathbf{A}_t$	matrix in ROUND step (Eq. (10))
$\mathbf{B}_t$	matrix used in Approx-FIRAL (Eq. (17))

B. FIRAL: RELAX step

The first step is to solve a continuous convex optimization problem which is formed by relaxing the constraint for z in Eq. (4):

$$z_\diamond \in \operatorname*{arg\,min}_{z\in[0,1]^n, \|z\|_1=b} \left(\mathbf{H}_o + \mathbf{H}_z\right)^{-1} \cdot \mathbf{H}_p. \tag{5}$$

The gradient of the objective w.r.t z_i is

$$g_i = \frac{\partial f(z)}{\partial z_i} = -\mathbf{H}_i \cdot \mathbf{\Sigma}_z^{-1}\mathbf{H}_p\mathbf{\Sigma}_z^{-1}, \tag{6}$$

where we define

$$\mathbf{\Sigma}_z = \mathbf{H}_o + \mathbf{H}_z. \tag{7}$$

FIRAL uses an entropic mirror descent algorithm to solve the relaxed problem.

C. FIRAL: ROUND step

After the Eq. (5) step, FIRAL rounds $z_\diamond$ into a valid solution to Eq. (4) via regret minimization. Let us denote $\mathbf{\Sigma}_\diamond = \mathbf{H}_o + \mathbf{H}_{z_\diamond}$ and for any matrix $\mathbf{H} \in \mathbb{R}^{\widetilde{d}\times\widetilde{d}}$, we define $\widetilde{\mathbf{H}}$ by

$$\widetilde{\mathbf{H}} \triangleq \mathbf{\Sigma}_\diamond^{-1/2}\mathbf{H}\mathbf{\Sigma}_\diamond^{-1/2}. \tag{8}$$

The round solve has b iterations and at each iteration $t \in [b]$, it selects the point i_t s.t.

$$i_t \in \operatorname*{arg\,min}_{i\in\mathbf{X}_u} \mathrm{Trace}[(\mathbf{A}_t + \frac{\eta}{b}\widetilde{\mathbf{H}}_o + \eta\widetilde{\mathbf{H}}_i)^{-1}], \tag{9}$$

where $\eta > 0$ is a hyperparameters (the learning rate), and $\mathbf{A}_t \in \mathbb{R}^{\widetilde{d}\times\widetilde{d}}$ is a symmetric positive definite matrix defined by the Follow-The-Regularized-Leader algorithm:

$$\mathbf{A}_t = \begin{cases} \sqrt{\widetilde{d}}\mathbf{I}_{\widetilde{d}} & t = 1 \\ \nu_t\mathbf{I} + \eta\widetilde{\mathbf{H}}_{t-1} & t > 1 \end{cases}, \tag{10}$$

Algorithm 1 EXACT-FIRAL

1: RELAX *step:*
2: $z = (1/n, 1/n, \cdots, 1/n) \in \mathbb{R}^n$
3: $\{\beta_t\}_{t=1}^T$: schedule of learning rate for relax solve
4: **for** $t = 1$ to T **do** # T is iteration number
5: $\quad \boldsymbol{\Sigma}_z \leftarrow \mathbf{H}_o + \mathbf{H}_z$
6: $\quad g_i \leftarrow -\operatorname{Trace}(\mathbf{H}_i \boldsymbol{\Sigma}_z^{-1} \mathbf{H}_p \boldsymbol{\Sigma}_z^{-1}), \quad \forall i \in [n]$
7: $\quad z_i \leftarrow z_i \exp(-\beta_t g_i)$
8: $\quad z_i \leftarrow \frac{z_i}{\sum_{j\in[n]} z_j}$
9: $z_\diamond \leftarrow bz$
10: ROUND *step:*
11: $X \leftarrow \emptyset$, $\boldsymbol{\Sigma}_\diamond \leftarrow \mathbf{H}_o + \mathbf{H}_{z_\diamond}$
12: $\mathbf{A}_1 \leftarrow \sqrt{\widetilde{d}}\mathbf{I}_{\widetilde{d}}$, $\widetilde{\mathbf{H}} \leftarrow \mathbf{0}$
13: **for** $t = 1$ to b **do**
14: $\quad i_t \leftarrow \arg\min_{i\in\mathbf{X}_u} \operatorname{Trace}[(\mathbf{A}_t + \frac{\eta}{b}\mathbf{H}_o + \eta\mathbf{H}_i)^{-1}]$
15: $\quad \widetilde{\mathbf{H}} \leftarrow \widetilde{\mathbf{H}} + \frac{1}{b}\widetilde{\mathbf{H}}_o + \widetilde{\mathbf{H}}_{i_t}$
16: $\quad \mathbf{V}\boldsymbol{\Lambda}\mathbf{V}^\top \leftarrow$ eigendecomposition of $\eta\widetilde{\mathbf{H}}$
17: $\quad$ find ν_{t+1} s.t. $\sum_{j\in[\widetilde{d}]}(\nu_{t+1} + \lambda_j)^{-2} = 1$ # bisection
18: $\quad \mathbf{A}_{t+1} \leftarrow \mathbf{V}(\nu_{t+1}\mathbf{I}_{\widetilde{d}} + \boldsymbol{\Lambda})\mathbf{V}^\top$
19: $\quad X \leftarrow X \cup \{x_{i_t}\}$

where $\nu_t \in \mathbb{R}$ is the unique constant s.t. $\operatorname{Trace}(\mathbf{A}_t^{-2}) = 1$, and

$$\widetilde{\mathbf{H}}_{t-1} = \sum_{l=1}^{t-1}\left(\frac{1}{b}\widetilde{\mathbf{H}}_o + \widetilde{\mathbf{H}}_{i_l}\right). \tag{11}$$

The FIRAL algorithm is near-optimal [5] in solving the optimization problem of Eq. (4):

Theorem 1. *[Theorem 10 in [5]] Given $\epsilon \in (0,1)$, let $\eta = 8\sqrt{\widetilde{d}}/\epsilon$, whenever $b \geq 32\widetilde{d}/\epsilon^2 + 16\sqrt{\widetilde{d}}/\epsilon^2$, denote z as the solution corresponding to the points selected by Algorithm 1, then the algorithm is near-optimal: $f(z) \leq (1+\epsilon)f_*$, where f_* is the optimal value of the f in Eq. (4).*

D. Complexity and scalability of FIRAL

Algorithm 1 summarizes FIRAL. Its storage complexity is $\mathcal{O}(c^2d^2 + nc^2d)$ (Table II), which is prohibitively large for large n, d or c. Furthermore, both relax and round solvers involve calculating inverse matrix of size $cd \times cd$. Thus, a scalable algorithm of FIRAL is needed.

III. THE APPROX-FIRAL ALGORITHM

A. The Hessian structure and a fast RELAX step

The new RELAX solver has four components. First, we replace the exact trace operator in line 6 of Algorithm 1 with a randomized trace estimator that only requires matvec operations. Second, we replace the direct solvers with a matrix-free conjugate gradients iterative method (CG). Third, we devise an exact fast matvec approximation for the Hessians. And fourth, we propose an effective preconditioner for the CG scheme. Taken together these components result in a scalable algorithm. We present the pseudo-code for our fast RELAX step in Algorithm 2 and summarize its complexity in Table II.

We first develop an estimator for the gradient g_i in Eq. (6) that avoids constructing dense $\widetilde{d}$-by-$\widetilde{d}$ matrices such as $\boldsymbol{\Sigma}_z$, $\mathbf{H}_p$, and $\boldsymbol{\Sigma}_z^{-1}$. The main idea is to use the Hutchinson trace estimator [15] to approximate the gradient: suppose that we use s Rademacher random vectors $\{v_j \in \mathbb{R}^d\}_{j\in[s]}$, then g_i can be approximated by

$$g_i \approx -\frac{1}{s}\sum_{j\in[s]} v_j^\top \mathbf{H}_i(\boldsymbol{\Sigma}_z^{-1}\mathbf{H}_p\boldsymbol{\Sigma}_z^{-1}v_j). \tag{12}$$

To calculate the vector $\boldsymbol{\Sigma}_z^{-1}\mathbf{H}_p\boldsymbol{\Sigma}_z^{-1}v_j$ in Eq. (12), we can solve two linear systems using CG. Note that this term can be shared for all $i \in \mathbf{X}_u$ in gradient approximation formula. Thus, we only need to calculate the vector once for each mirror descent iteration step.

Fast matrix-free matvec. The trace estimator and CG solvers require Hessian matvecs. The following Lemma gives an exact closed form of the matvec without forming the Hessian matrix explicitly.

Lemma 2 (Matrix-free Hessian matvec)**.** *For any given vector $v \in \mathbb{R}^{dc}$, let $\mathbf{V} \in \mathbb{R}^{d\times c}$ be the reshaped matrix from v such that $vec(\mathbf{V}) = v$. Denote the j-th column of $\mathbf{V}$ by $v_j \in \mathbb{R}^d$, k-th component of h_i by h_i^k. $\mathbf{H}_i$ is given by Eq. (2). Then*

$$\mathbf{H}_i v = \begin{bmatrix} (x_i^\top v_1 - x_i^\top \mathbf{V} h_i)h_i^1 x_i \in \mathbb{R}^d \\ \vdots \\ (x_i^\top v_c - x_i^\top \mathbf{V} h_i)h_i^c x_i \in \mathbb{R}^d \end{bmatrix} \in \mathbb{R}^{\widetilde{d}}.$$

Proof.

$$\begin{aligned}\mathbf{H}_i v &= [\operatorname{diag}(h_i) \otimes (x_i x_i^\top)]v - [(h_i h_i^\top) \otimes (x_i x_i^\top)]v \\ &= vec\big(x_i x_i^\top \mathbf{V} \operatorname{diag}(h_i)\big) - vec\big(x_i x_i^\top \mathbf{V} h_i h_i^\top\big) \\ &= vec\big(\big[(x_i^\top v_1)h_i^1 x_i, \cdots, (x_i^\top v_c)h_i^c x_i\big]\big) \\ &\quad - (x_i^\top \mathbf{V} h_i) vec(x_i h_i^\top),\end{aligned}$$

where the second equality uses a property of the matrix Kronecker product. □

According to Lemma 2, we can compute $\mathbf{H}_i v$ in the following steps: ❶ $\gamma_i \leftarrow \mathbf{V}^\top x_i$, ❷ $\alpha_i \leftarrow \gamma_i^\top h_i$, ❸ $\gamma_i \leftarrow (\gamma_i - \alpha_i) \odot h_i$, and ❹ $\mathbf{H}_i v \leftarrow vec(\gamma_i \otimes x_i)$. It is worth noting that the storage required for the first three steps is only $c+1$ elements, while the last step requires dc elements for storing the result of the matvec operation. A comparison of the complexity between our fast matvec algorithm and direct matvec is provided in Table III.

With the help of the matrix-free matvec, we can calculate $\mathbf{H}_p v$ by

$$\mathbf{H}_p v = \begin{bmatrix} \sum_{i\in\mathbf{X}_u} \gamma_i^1 x_i \\ \vdots \\ \sum_{i\in\mathbf{X}_u} \gamma_i^c x_i, \end{bmatrix} \tag{13}$$

where $\gamma_i^k = (x_i^\top v_1 - x_i^\top \mathbf{V} h_i)h_i^k$ for $k \in [c]$. Based on the previous analysis, the additional storage required is solely for γ_i for all unlabeled points $\mathbf{X}_u$, amounting to $4n(c+1)$ memory cost. We can use the similar calculation for the matvec of $\boldsymbol{\Sigma}_z v$ within the CG iterations.

Preconditioned CG. To further accelerate the calculation, we propose a simple but, as we will see, effective block diagonal preconditioner for the CG solves. We first introduce the block diagonal operation as follows.

Table II *Comparison of algorithm complexity between FIRAL and Approx-FIRAL. n_{relax} is the number of mirror descent iterations in relax solver, n_{CG} is the number of CG iterations in each mirror descent step of the Approx-FIRAL relax solver.*

Complexity	Exact-FIRAL		Approx-FIRAL	
	Relax	Round	Relax	Round
Storage	$\mathcal{O}(c^2d^2 + nc^2d)$	$\mathcal{O}(c^2d^2 + nc^2d)$	$\mathcal{O}(n(d+sc) + cd^2)$	$\mathcal{O}(n(d+c) + cd^2)$
Computation	$\mathcal{O}\big(n_{\text{relax}}nc^3d^2\big)$	$\mathcal{O}\big(bc^3(d^3+n)\big)$	$\mathcal{O}\big(n_{\text{relax}}ncd(d+n_{\text{CG}}s)\big)$	$\mathcal{O}\big(bncd^2\big)$

Table III *Comparison of storage and computational complexity between matrix-free matvec and direct matvec.*

method	storage	computation
direct MatVec	$\mathcal{O}(d^2c^2)$	$\mathcal{O}(d^2c^2)$
fast MatVec	$\mathcal{O}(dc)$	$\mathcal{O}(dc)$

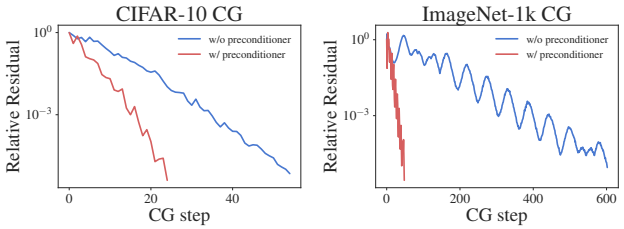

Figure 1 *The impact of preconditioner on CG iterations. The experimental setup is detailed in § IV-A. We showcase the convergence of CG in the initial mirror descent iteration (i.e., Line 6 of Algorithm 2).*

Definition 1 (Block diagonal operation $\mathcal{B}(\cdot)$)**.** *For any matrix $\mathbf{H} \in \mathbb{R}^{\widetilde{d}\times\widetilde{d}}$, define $\mathcal{B}(\mathbf{H}) \in \mathbb{R}^{\widetilde{d}\times\widetilde{d}}$ as the matrix comprising $d \times d$ block diagonals of $\mathbf{H}$; denote the k-th block diagonal matrix by $\mathcal{B}_k(\mathbf{H}) \in \mathbb{R}^{d\times d}$.*

Then, for every Hessian matrix $\mathbf{H}_i$ in Eq. (2), we have its block diagonal as

$$\mathcal{B}(\mathbf{H}_i) = [\text{diag}(h_i \odot (1-h_i))] \otimes (x_i x_i^\top), \tag{14}$$

and its k-th matrix diagonal as

$$\mathcal{B}_k(\mathbf{H}_i) = h_i^k(1-h_i^k) \cdot x_i x_i^\top. \tag{15}$$

We employ $\mathcal{B}(\mathbf{\Sigma}_z)^{-1}$ as the preconditioner for CG to solve the linear system required for gradient estimation in Eq. (12). We illustrate the effectiveness of the CG preconditioner for two datasets in Fig. 1. Using $\mathcal{B}(\mathbf{\Sigma}_z)^{-1}$ as preconditioner accelerates CG convergence due to several factors. Firstly, it improves the conditioning of the matrix. For instance, in the CIFAR-10 test, the condition number of $\mathbf{\Sigma}_z$ is 198, while the condition number of $\mathcal{B}(\mathbf{\Sigma}_z)^{-1}\mathbf{\Sigma}_z$ is 72. Additionally, the majority of eigenvalues of the preconditioned matrix are clustered into small intervals.

B. The new ROUND step

The difficulty of the ROUND step lies in computing the objective value in Eq. (9) for each point $i \in \mathbf{X}_u$ at each round $t \in [b]$. Even when employing CG with the fast matrix-free matvec introduced in the preceding section, the computational complexity for estimating the objective is $\mathcal{O}(bn_{\text{CG}}n^2cds)$, which is prohibitively large for large-scale problems.

Motivated by the effectiveness of the preconditioner in the RELAX, it is natural to consider some approximation. Notice

Algorithm 2 FAST RELAX SOLVE

$z = (1/n, 1/n, \cdots, 1/n) \in \mathbb{R}^n$
$\{\beta_t\}_{t=1}^T$: schedule of learning rate for relax solve
for $t = 1$ to T **do** # T is iteration number
  $\mathbf{V} = [v_1, v_2, \cdots, v_s] \in \mathbb{R}^{dc\times s}$: matrix of s Rademacher random vectors.
  $\{\mathcal{B}_k(\mathbf{\Sigma}_z)^{-1}\}_{k\in[c]} \leftarrow$preconditioner for CG solve
  $\mathbf{W} \leftarrow \mathbf{\Sigma}_z^{-1}\mathbf{V}$ by preconditioned CG
  $\mathbf{W} \leftarrow \mathbf{H}_p\mathbf{W}$
  $\mathbf{W} \leftarrow \mathbf{\Sigma}_z^{-1}\mathbf{W}$ by preconditioned CG
  $g_i \leftarrow -\frac{1}{s}\sum_{j\in[s]} v_j^\top \mathbf{H}_i w_j, \quad \forall i \in \mathbf{X}_u$
  $z_i \leftarrow z_i \exp(-\beta_t g_i)$
  $z_i \leftarrow \frac{z_i}{\sum_{j\in[n]} z_j}$
$z_\diamond \leftarrow bz$

Algorithm 3 APPROX-FIRAL

$z_\diamond \leftarrow$ solution of RELAX step from Algorithm 2
Diagonal ROUND *step:*
$X \leftarrow \emptyset$, form $\{(\mathbf{\Sigma}_\diamond)_k \in \mathbb{R}^{d\times d}\}_{k\in[c]}$
$\big\{(\mathbf{B}_1)_k^{-1} \leftarrow [\sqrt{\widetilde{d}}(\mathbf{\Sigma}_\diamond)_k + \frac{\eta}{b}(\mathbf{H}_o)_k]^{-1}\big\}_{k\in[c]}$
$\big\{(\mathbf{H})_k \leftarrow \mathbf{0}\big\}_{k\in[c]}$
for $t = 1$ to b **do**
  $i_t \leftarrow$ Eq. (17)
  $\big\{(\mathbf{H})_k \leftarrow (\mathbf{H})_k + \frac{1}{b}(\mathbf{H}_o)_k + h_{i_t}^k(1-h_{i_t}^k)x_{i_t}x_{i_t}^\top\big\}_{k\in[c]}$
  $\big\{[\lambda_{k,j}]_{j=1}^d \leftarrow$ eigenvalues of $(\widetilde{\mathbf{H}})_k\big\}_{k\in[c]}$
  find ν_{t+1} s.t. $\sum_{k\in[c]}\sum_{j\in[d]}(\nu_{t+1} + \eta\lambda_{k,j})^{-2} = 1$
  $\big\{(\mathbf{B}_{t+1})_k^{-1} \leftarrow [\nu_{t+1}(\mathbf{\Sigma}_\diamond)_k + \eta(\mathbf{H})_k + \frac{\eta}{b}(\mathbf{H}_o)_k]^{-1}\big\}_{k\in[c]}$
  $X \leftarrow X \cup \{x_{i_t}\}$

that the ROUND step becomes much easier when considering only the block diagonals of all Hessian matrices. Specifically, we assume that each Hessian matrix $\mathbf{H}_i$ retains only its block diagonal parts, as expressed in Eq. (14). Consequently, all matrices with a size of $\widetilde{d} \times \widetilde{d}$ in the ROUND step are block diagonal. This assumption not only reduces storage requirements but also simplifies calculations. Firstly, we introduce a Sherman-Morrison-like formula for the low-rank updates for the inverse of a block diagonal matrix in Lemma 3. Subsequently, we present a simple yet equivalent objective to the original exact ROUND step in Proposition 4. We outline the pseudo-code in Algorithm 3 and summarize the complexity of the new ROUND step in Table II.

Lemma 3. *Let $\mathbf{A} \in \mathbb{R}^{\widetilde{d}\times\widetilde{d}}$ be a block diagonal positive definite matrix with c block diagonals of $d \times d$ matrices, $x \in \mathbb{R}^d$ and $\gamma \in \mathbb{R}^c$ be vectors. If $\mathbf{A} + \text{diag}(\gamma) \otimes (xx^\top)$ is positive definite, then $\big(\mathbf{A} + \text{diag}(\gamma) \otimes (xx^\top)\big)^{-1}$ is a block diagonal matrix with its k-th block having the following form:*

$$\big(\mathbf{A} + \text{diag}(\gamma) \otimes (xx^\top)\big)_k^{-1} = \mathbf{A}_k^{-1} - \frac{\gamma_k \mathbf{A}_k^{-1}xx^\top\mathbf{A}_k^{-1}}{1 + \gamma_k x^\top \mathbf{A}_k^{-1}x}, \tag{16}$$

where $\mathbf{A}_k^{-1}$ *is the inverse of* k*-th diagonal of* $\mathbf{A}$*,* γ_k *is the* k*-th component of* γ*.*

Proposition 4. *If all Fisher information matrices* $\mathbf{H}_i$ *only preserve the block diagonals of* $d \times d$ *matrices, then at each iteration of the* ROUND *step, the objective defined in Eq. (9) is equivalent to the following:*

$$i_t \in \arg\max_{i \in \mathbf{X}_u} \sum_{k=1}^{c} h_i^k (1 - h_i^k) \cdot \frac{x_i^\top (\mathbf{B}_t)_k^{-1} (\mathbf{\Sigma}_\diamond)_k^{-1} (\mathbf{B}_t)_k^{-1} x_i}{1 + \eta h_i^k (1 - h_i^k) x_i^\top (\mathbf{B}_t)_k^{-1} x_i}, \tag{17}$$

where $\mathbf{B}_t = \mathbf{\Sigma}_\diamond^{1/2} \mathbf{A}_t \mathbf{\Sigma}_\diamond^{1/2} + \frac{\eta}{b} \mathbf{H}_o$.

Proof. We denote the objective for point $i \in \mathbf{X}_u$ in round problem Eq. (9) by r_i, then

$$\begin{aligned} r_i &= \mathrm{Trace}[(\mathbf{A}_t + \frac{\eta}{b}\widetilde{\mathbf{H}}_o + \eta \widetilde{\mathbf{H}}_i)^{-1}] \\ &= \mathrm{Trace}\left[\mathbf{\Sigma}_\diamond^{1/2} \big(\underbrace{\mathbf{\Sigma}_\diamond^{1/2} \mathbf{A}_t \mathbf{\Sigma}_\diamond^{1/2} + \frac{\eta}{b} \mathbf{H}_o}_{\triangleq \mathbf{B}_t} + \eta \mathbf{H}_i \big)^{-1} \mathbf{\Sigma}_\diamond^{1/2} \right] \\ &= \mathrm{Trace}\left[\big(\mathbf{B}_t + \eta \mathbf{H}_i \big)^{-1} \mathbf{\Sigma}_\diamond \right]. \end{aligned} \tag{18}$$

Since $\mathbf{B}_t$ and $\mathbf{H}_i$ are both block diagonal, by Lemma 3, k-th block diagonal of $\big(\mathbf{B}_t + \eta \mathbf{H}_i\big)^{-1}$ has the following form:

$$\big(\mathbf{B}_t + \eta \mathbf{H}_i\big)_k^{-1} = \big(\mathbf{B}_t\big)_k^{-1} - \frac{\eta h_i^k (1 - h_i^k) (\mathbf{B}_t)_k^{-1} x_i x_i^\top (\mathbf{B}_t)_k^{-1}}{1 + \eta h_i^k (1 - h_i^k) x_i^\top (\mathbf{B}_t)_k^{-1} x_i}. \tag{19}$$

Substitute Eq. (19) into Eq. (18), we have

$$\begin{aligned} r_i = &\mathrm{Trace}[\mathbf{B}_t^{-1} \mathbf{\Sigma}_\diamond] \\ &- \eta \sum_{k=1}^{c} h_i^k (1 - h_i^k) \cdot \frac{x_i^\top (\mathbf{B}_t)_k^{-1} (\mathbf{\Sigma}_\diamond)_k^{-1} (\mathbf{B}_t)_k^{-1} x_i}{1 + \eta h_i^k (1 - h_i^k) x_i^\top (\mathbf{B}_t)_k^{-1} x_i}, \end{aligned} \tag{20}$$

which leads to Eq. (17). □

C. HPC implementation and complexity analysis

Our HPC implementation of Approx-FIRAL, as outlined in Algorithms 2 and 3, is GPU-based. We employ `cupy` [6] for computation and `mpi4py` [8] for communication within GPUs. To utilize a GPU-aware Message Passing Interface (MPI), we utilize MVAPICH2-GDR [16]. Our implementation employs single-precision floating point for both storage and computation. Let p be the number of GPUs, we start the parallel implementation by evenly distributing h_i and x_i of n points in $\mathbf{X}_u$ across p GPUs.

Regarding computation, we utilize the built-in functions of the linear algebra routines available in `cupy`. We provide a summary of some of the key functions as follows:

- `cupy.einsum`: In the RELAX step outlined in Algorithm 2, we utilize Einstein summation to construct the block diagonal matrix as a preconditioner in Line 5. In Lines 6-8, we employ Einstein summation for the fast matrix-free matvec developed in § III-A for matrices $\mathbf{\Sigma}_z$ and $\mathbf{H}_p$. For the ROUND step in Algorithm 3, we use this function mainly for the objective calculation in Eq. (17) (Line 7).
- `cupy.linalg.eigvalsh`: In the ROUND step, this function is employed to compute the eigenvalues of the block diagonals of $\widetilde{\mathbf{H}}$ in a batch-wise manner in Line 9 of Algorithm 3. In our implementation, we evenly distribute the computation of eigenvalues for c block diagonals among p GPUs.
- `cupy.linalg.inv`: This function is utilized to calculate the inverse of block diagonal matrices in Line 5 of Algorithm 2 and Lines 4 and 11 of Algorithm 3.

As for communication among GPUs, we outline the primary collective communication operations utilized as follows:

- `MPI_Allreduce`: For RELAX step in Algorithm 2, we need this operation for summation of the block diagonals in Line 5. In Lines 6-8, it is necessary for the summation of the results from the matvec operation. For ROUND step in Algorithm 3, we use `MPI_Allreduce` in Line 7 to find the point with the global maximum objective value across all GPUs.
- `MPI_Allgather`: This operation is employed to collect all eigenvalues in the ROUND step (Line 9 of Algorithm 3).
- `MPI_Bcast`: In Lines 6-8 of Algorithm 2, we distribute $\mathbf{W}$ to each GPU. In Line 11 of Algorithm 3, we utilize this operation to transmit h_{i_t} and x_{i_t} to all GPUs.

In Table IV, we summarize the complexity of storage, computation and communication for our HPC implementation of Approx-FIRAL. The details are outlined as follows. To estimate the cost of collective communications, we rely on the results presented in [17]. We assume that the time used to send a message between two processes is $t_s + mt_w$, where t_s is the latency, t_w is the transfer time per byte, and m denotes the number of bytes transferred. Additionally, we denote the computation cost per byte by t_c for performing the reduction operation locally on any process. The costs associated with the three MPI operations we utilized are as follows: ❶ `MPI_Allreduce`: employing the recursive doubling algorithm, the time complexity is $\log p(t_s + m(t_w + t_c))$. ❷ `MPI_Allgather`: utilizing the recursive doubling algorithm, the time complexity is $\log p t_s + \frac{p-1}{p} m t_w$. ❸ `MPI_Bcast`: using the binomial tree algorithm, the time complexity is $\log p(t_s + mt_w)$.

RELAX step. In terms of storage, the parallel implementation of Algorithm 2 requires storing Rademacher random vectors $\mathbf{V}$ (Line 4), the intermediate matrix $\mathbf{W}$ (Lines 6-8), and the inverses of c block-diagonal matrices (Line 5). Hence, the total storage for each GPU amounts to $\mathcal{O}\left(\frac{n}{p}(d + c) + cds + cd^2\right)$ including the storage of x_i and h_i for $\frac{n}{p}$ points.

For building the preconditioner of CG (Line 5), each GPU initially computes the block diagonal matrices $\{\mathcal{B}(\mathbf{\Sigma}_z)_k\}_{k \in [c]}$ with a complexity of $\mathcal{O}(\frac{n}{p} cd^2)$. The `MPI_Allreduce` operation for aggregation of these matrices across all GPUs incur a communication cost of $\mathcal{O}\left(\log p(t_s + cd^2(t_w + t_c)\right)$. Then each GPU calculates the inverse of the block diagonal matrices as the preconditioner, which has a computational complexity

Table IV *Storage, computation and communication complexity of parallel implementation of Approx-FIRAL (Algorithm 3). The detailed derivations are presented in § III-C. n_{relax} represents the number of mirror descent iteration in Algorithm 2, n_{CG} represents the number of CG iterations.*

Complexity	Storage	Computation	Communication
RELAX step	$\mathcal{O}\left(\frac{n}{p}(d+c)+cds+cd^2\right)$	$\mathcal{O}\left(n_{\text{relax}}cd(\frac{n}{p}(d+n_{\text{CG}}s)+d^2)\right)$	$\mathcal{O}\left(n_{\text{relax}}\log p\left(n_{\text{CG}}t_s+cd(n_{\text{CG}}s+d)(t_w+t_c)\right)\right)$
ROUND step	$\mathcal{O}\left(\frac{n}{p}(d+c)+cd^2\right)$	$\mathcal{O}\left(bcd^2(\frac{n}{p}+d)\right)$	$\mathcal{O}\left(b\log p\left(t_s+(d+c)t_w+t_c\right)\right)$

of $\mathcal{O}(cd^3)$. In summary, the computational and communication time required to construct the preconditioner are as follows:

$$T^{\text{comp}}_{\mathcal{B}(\mathbf{\Sigma}_z)} = \mathcal{O}\big(cd^2(\frac{n}{p}+d)\big), \tag{21}$$

$$T^{\text{comm}}_{\mathcal{B}(\mathbf{\Sigma}_z)} = \mathcal{O}\big(\log p(t_s+cd^2(t_w+t_c)\big). \tag{22}$$

Within each preconditioned CG iteration (Lines 6 and 8), the primary time consumption arises from the matvec calculations of $\mathbf{\Sigma}_z\mathbf{V}$ and $\mathcal{B}(\mathbf{\Sigma}_z)\mathbf{V}$. According to the complexity outlined in our fast matvec algorithm in Table III, computation of matvec has a complexity of $\mathcal{O}\left(\frac{n}{p}cds\right)$. Subsequently, the summation of these vectors requires an `MPI_Allreduce` operation with a communication cost of $\mathcal{O}\left(\log p(t_s+cds(t_w+t_c)\right)$. The computation of $\mathcal{B}(\mathbf{\Sigma}_z)\mathbf{V}$ solely demands a computational cost of $\mathcal{O}(cd^2s)$. Let n_{CG} be the CG iteration number, we have

$$T^{\text{comp}}_{\text{CG}} = \mathcal{O}\Big(n_{\text{CG}}\frac{n}{p}cds\Big), \tag{23}$$

$$T^{\text{comm}}_{\text{CG}} = \mathcal{O}\Big(n_{\text{CG}}\log p(t_s+cds(t_w+t_c))\Big). \tag{24}$$

Regarding other components of the relax solver, Line 7 of the matvec operation has a complexity similar to one step of CG. The computation of the gradient g_i (Line 9) and the updating of z (Lines 10-11) necessitate a complexity of $\mathcal{O}\left(\frac{n}{p}cds\right)$.

ROUND **step.** Regarding storage, all matrices utilized in Algorithm 3 are block diagonal matrices, resulting in a storage requirement of $\mathcal{O}(cd^2)$. Furthermore, to compute the objective for each point in Line 7, additional storage of $\mathcal{O}(nc)$ is necessary. As a result, the total storage requirement is $\mathcal{O}(n(c+d)+cd^2)$.

During each iteration of the ROUND step, computing the objective function for each point in Line 7 (Eq. (17)) needs a computational complexity of $\mathcal{O}(\frac{n}{p}cd^2)$. Subsequently, to select the point with the maximum objective, we utilize `MPI_Allreduce` to gather and compare the maximum objective across local processes, resulting in a communication cost of $\mathcal{O}(\log p(t_s+t_w+t_c))$.

To update $\{(\mathbf{H})_k\}_{k\in[c]}$ (Line 8), the process owning i_t broadcasts x_{i_t} and h_{i_t} to other processes using an `MPI_Bcast` operation with a size of $\mathcal{O}(c+d)$. In Line 9, we first compute eigenvalues for $\frac{c}{p}$ matrices for each process, followed by collecting all eigenvalues using `MPI_Allgather`. The computational complexity of this step is $\mathcal{O}(\frac{c}{p}d^3)$, and the communication cost is $\mathcal{O}(\log pt_s+\frac{c}{p}t_w)$. As for Line 11, computing the inverse matrices requires a computational complexity of $\mathcal{O}(cd^3)$. The total computation and communication complexity for the ROUND step are summarized in Table IV.

IV. NUMERICAL EXPERIMENTS

We test the classification accuracy in § IV-A, single node performance in § IV-B and parallel computing performance in § IV-C on the Lonestar6 A100 nodes in the Texas Advanced Computing Center (TACC). Lonestar6 A100 nodes are interconnected with IB HDR (200 Gbps) and have three A100 NVIDIA GPUs per node.

A. Active learning performance

In our accuracy experiments, we attempt to answer the following questions regarding Approx-FIRAL. How does the performance of Approx-FIRAL in active learning tests compare to Exact-FIRAL? How does Approx-FIRAL compare to other active learning methods? Considering we utilize the Hutchinson trace estimator and CG for gradient estimation in RELAX, what impact do variations in the number of Rademacher random vectors and CG termination criteria have on the convergence of RELAX?

Datasets. We demonstrate the effectiveness of Approx-FIRAL using the following real-world datasets: MNIST [18], CIFAR-10 [19], Caltech-101 [20] and ImageNet [21]. First we use unsupervised learning to extract features and then apply active learning to the feature space, that is, we do **not** use any label information in our pre-processing. For MNIST, we calculate the normalized Laplacian of the training data and use the spectral subspace of the 20 smallest eigenvalues. For CIFAR-10, we use a contrastive learning SimCLR model [2] to extract feature; then we compute the normalized Laplacian and select the subspace of the 20 smallest eigenvalues. For Caltech-101 and ImageNet-1k, we use state-of-the-art self-supervised learning model DINOv2 [22] to extract features. We additionally select 50 classes randomly from ImageNet-1k and construct dataset ImageNet-50.

We construct 7 datasets for the active learning tests. A summary of the datasets is outlined in Table V. For the initial labeled set $\mathbf{X}_o$, we randomly pick two samples per class for ImageNet-1k and one per class for all other datasets. To form the unlabeled pool $\mathbf{X}_u$ in MNIST, CIFAR-10, ImageNet-50, and ImageNet-1k, we evenly select points from each class randomly. To simulate a non-i.i.d. scenario, we assemble $\mathbf{X}_u$ in an imbalanced manner for imb-CIFAR-10, imb-ImageNet-50, and Caltech-101. In imb-CIFAR-10 and Caltech-101, the maximum ratio of points between two classes is 10. In imb-ImageNet-50, the maximum ratio of points between two classes is eight. We use the points from the whole training dataset for evaluation.

Table V *Summary of datasets for active learning experiments.*

Name	Type	# classes	dimension	$\mathbf{X}_o$	$\mathbf{X}_u$	# rounds	budget/round	# evaluation points
MNIST	balanced	10	20	10	3,000	3	10	60,000
CIFAR-10 imb-CIFAR-10	balanced imbalanced	10	20	10	3,000	3	10	50,000
ImageNet-50 imb-ImageNet-50	balanced imbalanced	50	50	50	5,000	6	50	64,273
Caltech-101	imbalanced	101	100	101	1,715	6	101	8,677
ImageNet-1k	balanced	1,000	383	2,000	50,000	5	200	1,281,167

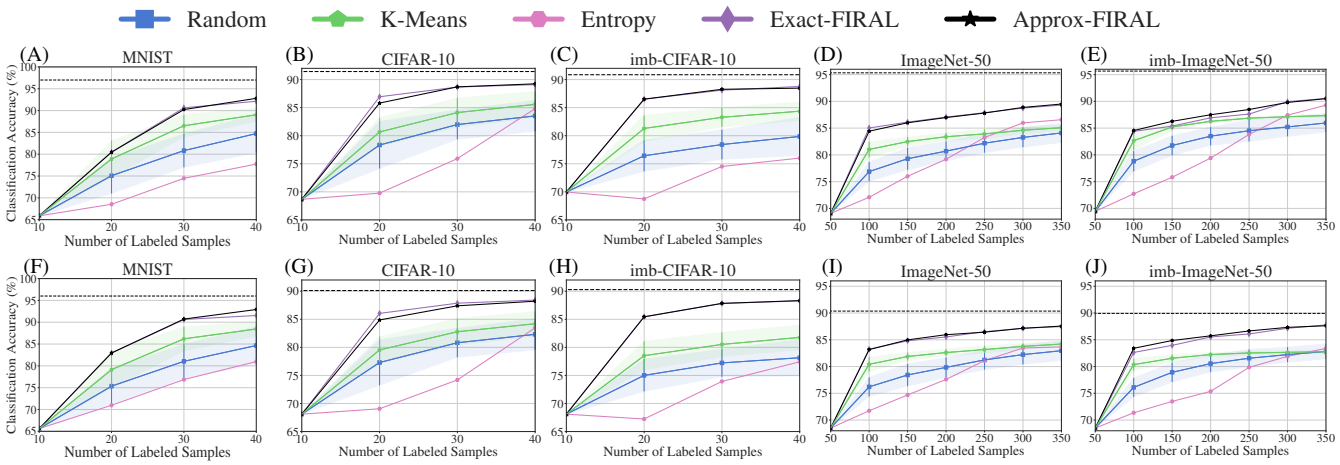

Figure 2 *Classification accuracy for active learning experiments conducted on MNIST, CIFAR-10, imb-CIFAR-10, ImageNet-50, and imb-ImageNet-50 on MNIST, CIFAR-10, imb-CIFAR-10, ImageNet-50 and imb-ImageNet-50. The upper row ((A)-(E)) are plots of pool accuracy on the unlabeled pool $\mathbf{X}_u$, the lower row ((F)-(J)) are plots of evaluation accuracy on the evaluation data.*

Table VI *Time comparison between Exact-FIRAL and Approx-FIRAL on a single A100 GPU. The time reported in the table is in seconds.*

	Exact-FIRAL	Approx-FIRAL
ImageNet-50		
RELAX	33.6	1.3
ROUND	34.8	1.1
Caltech-101		
RELAX	172.3	1.9
ROUND	945.3	4.4

Experimental setup. We compare our proposed Approx-FIRAL with four methods: (1) Random selection, (2) K-means where $k = b$ (b is the budget of the active learning selection per round), (3) Entropy: select top-b points that minimize $\sum_c p(y = c|x) \log p(y = c|x)$, (4) Exact-FIRAL: the original implementation of Algorithm 1. For tests involving larger dimension and number of classes, such as Caltech-101 and ImageNet-1k, we do not conduct tests on Exact-FIRAL due to its demanding storage and computational requirements.

For each of our active learning tests, we use a fixed budget number for selecting points across 3 to 6 rounds. The details are outlined in Table V. We report the average and standard deviation for Random and K-means based on 10 trials.

Regarding the hyperparameters in RELAX, we fix the number of Rademacher vectors at 10, and terminate the CG iteration when the relative residual falls below 0.1. Additionally, we conclude the mirror descent iteration when the relative change of the objective is less than $1.0\text{E}{-4}$. In all of our tests in Table V, this criterion is met within fewer than 100 mirror descent iterations.

The ROUND step requires only one hyperparameter, η. We determine the value of η following the same approach as Exact-FIRAL [5]: we execute the ROUND step with different η values, and then select the one that maximizes $\min_{k\in[c]} \lambda_{\min}(\mathbf{H})_k$, where $\mathbf{H}$ represents the summation of Hessian of the selected b points (Algorithm 3).

We utilize the logistic regression implementation of `scikit-learn` [23] as our classifier, and we keep the parameters fixed during active learning.

We present the classification accuracy results for both pool accuracy and evaluation accuracy on MNIST, CIFAR-10, imb-CIFAR-10, ImageNet-50 and imb-ImageNet-50 in Fig. 2. Here, pool accuracy refers the accuracy of classifier on the unlabeled pool points $\mathbf{X}_u$, while evaluation accuracy represents the accuracy on the evaluation data (the respective quantities are detailed in Table V). In Fig. 3, we plot the accuracy results obtained from active learning tests conducted on Caltech-101 and ImageNet-1k.

Approx-FIRAL vs. Exact-FIRAL. From the results depicted in Fig. 2, we can observe a very close resemblance in the performance of Approx-FIRAL and Exact-FIRAL. The discrepancies between these two methods are only visible

in a few instances. For example, in the initial round of the CIFAR-10 test (where the number of labeled points is 20 in Fig. 2(B) and (G)), Exact-FIRAL exhibits slightly better performance than Approx-FIRAL. However, Approx-FIRAL surpasses Exact-FIRAL slightly in the imb-ImageNet-50 test (Fig. 2(E) and (J)), as well as in the final round of the MNIST test (Fig. 2(A) and (F)).

In Table VI, we illustrate the time comparison between Exact-FIRAL and Approx-FIRAL for the initial round of ImageNet-50 and Caltech-101 on a single A100 GPU. For ImageNet-50, Approx-FIRAL demonstrates approximately 29 times faster performance than Exact-FIRAL. In the case of Caltech-101, Approx-FIRAL is about 177 times faster compared to Exact-FIRAL.

Approx-FIRAL vs. other methods. It is evident that Approx-FIRAL outperforms other methods in the active learning test results presented in Figs. 2 and 3. Notably, methods such as Random, K-means, and Entropy exhibit an obvious decrease in evaluation accuracy from the balanced CIFAR-10 test (Fig. 2(G)) to the imbalanced CIFAR-10 test (Fig. 2(H)). However, FIRAL maintains a consistent performance level across both CIFAR-10 and imb-CIFAR-10 tests. Further observations include: K-means outperforms Random in all the active learning tests presented in Fig. 2, shows comparable accuracy results to Random in Caltech-101 (Fig. 3(A) and (B)), and exhibits inferior performance compared to Random in ImageNet-1k (Fig. 3(C) and (D)). Additionally, in scenarios where the number of labeled samples is limited (such as in tests on MNIST, CIFAR-10, or the initial rounds of ImageNet-50 in Fig. 2), Random and K-means display considerable variance, and uncertainty-based method such as Entropy performs the poorest.

Parameters in RELAX step. To explore the influence of the number of Rademacher random vectors (s) and the termination tolerance of CG (cg_tol) on RELAX, we analyze the initial round of the active learning test on CIFAR-10 and ImageNet-50. We plot the objective function value Eq. (5) of RELAX against the iteration number of mirror descent in Fig. 4, varying the values of s or cg_tol. Notably, we observe that RELAX does not demonstrate sensitivity to either s or cg_tol.

B. Single-GPU performance

We now turn our attention to the HPC performance evaluation. We start with discussing the performance of our algorithm on a single GPU. We study the performance sensitivity to feature size d and number of class c in ImageNet dataset for both RELAX step and ROUND step. We provide estimates for the theoretical peak time of each major computational component, assuming an ideal peak performance of 19.5TFLOPS for Float32 computation on the GPU A100 [24]. The computation of RELAX solve is broken down into four major components: setting up preconditioner $\mathcal{B}(\mathbf{\Sigma}_z)^{-1}$, performing the conjugate gradient (*CG*), evaluating the *gradient* and *other* related tasks. For ROUND solve, we focus on three components: *computing*

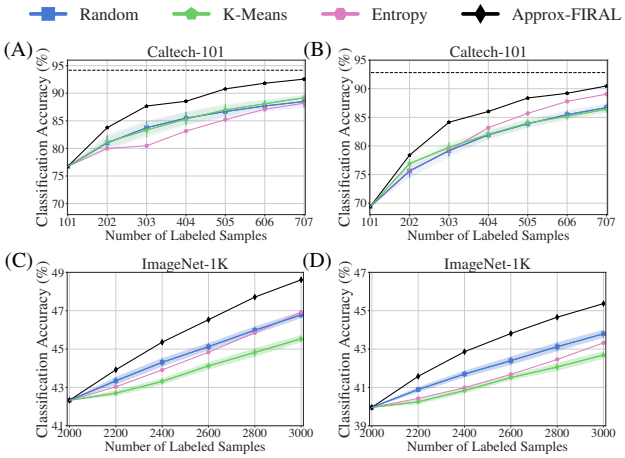

Figure 3 *Classification accuracy for active learning experiments on Caltech-101 and ImageNet-1k. Both (A) and (B) represent the accuracy on evaluation data for Caltech-101. In (A), the accuracy is averaged with each point having the same weight, while in (B), the accuracy is averaged with each class having the same weight. (C) presents the pool accuracy for ImageNet-1k, and (D) presents the evaluation accuracy for ImageNet-1k.*

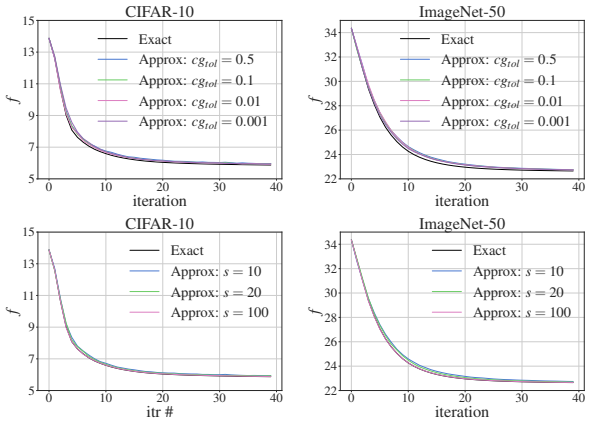

Figure 4 *Effect of the number of Rademacher random vectors (top) and CG termination criteria (bottom) on* RELAX *step (i.e., Algorithm 2). "Exact" refers to the precise* RELAX *solver utilized in Exact-FIRAL, while "Approx" denotes the fast* RELAX *solver employed in Approx-FIRAL. Here, s denotes the number of Rademacher random vectors, and cg_{tol} signifies the relative residual termination tolerance used in the CG solves.*

eigenvalues that is invoked at line 9 of Algorithm 3, evaluating the *objective function*, and *other* related tasks.

Sensitivity to feature size d. As we saw, the computational complexity of the RELAX solve is $\mathcal{O}\left(cd^3 + ncd^2 + n_{CG}ncsd\right)$, where n_{CG} is the number of CG iterations. The major cost lies in the construction of preconditioner $\left\{\mathcal{B}_k(\mathbf{\Sigma}_z^{-1})\right\}_{k\in[c]}$ and CG solving $\mathbf{\Sigma}_z\mathbf{W} = \mathbf{V}$. The construction of $\left\{\mathcal{B}_k(\mathbf{\Sigma}_z^{-1})\right\}_{k\in[c]}$ takes $cd^3 + 2cnd^2$ time. The CG solve involves n_{CG} evaluations of $\mathbf{\Sigma}_z\mathbf{V}$. According to Lemma 2, the time complexity of CG is dominated by $4n_{CG}ncsd$. Time complexity of ROUND solve is $\mathcal{O}(cd^3 + ncd^2)$. The major cost lies in line 9 in Algorithm 3, and evaluation of objective function in Eq. (17). We use

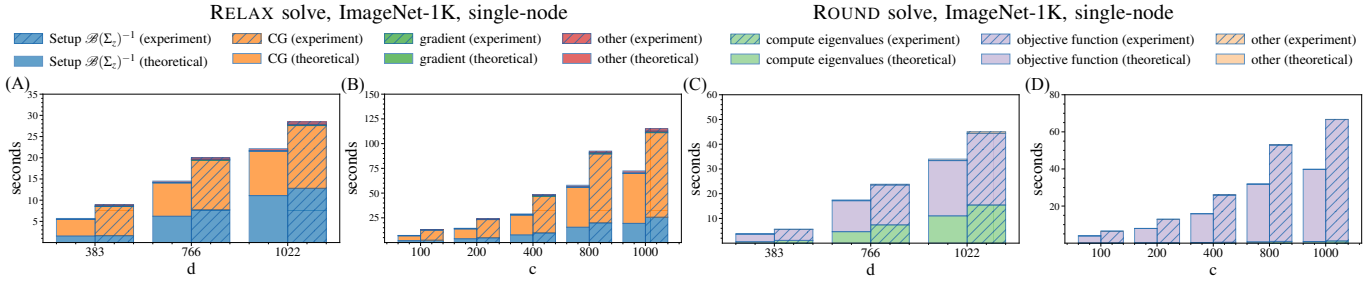

Figure 5 *Wall-clock time dependence of the* RELAX *and* ROUND *solves to the number of features d and the number of classes c using ImageNet-1K. In the run for the d scaling, we fix the number of data points $n = 1.0\text{E}5$ and the number of classes $c = 1000$. We set the number of random vectors to $s = 10$. For each value of d, we run one gradient and fix the number of CG iterations to $n_{CG} = 50$; and the left column represents theoretical time and the right column represents experimental time. In the run to test the algorithmics scalability in c, we fix $n = 1.3\text{E}6$, $d = 383$ and vary c as $[100, 200, 400, 800, 1000]$. The remaining parameters of the algorithm are fixed. We report the results as follows(A)* RELAX *run for d scaling. (B)* RELAX *run for c scaling. (C)* ROUND *solve for d scaling. (D)* ROUND *solve for c scaling.*

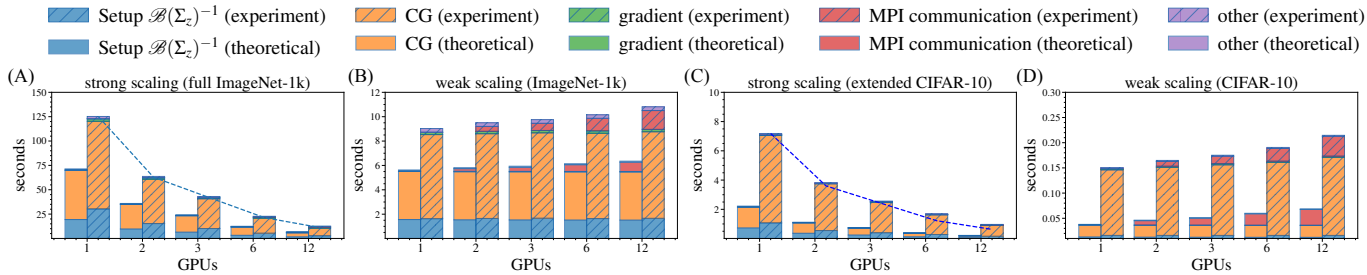

Figure 6 *Strong and weak scaling of the* RELAX *step on CIFAR-10 and ImageNet-1K. The dashed lines indicate ideal scaling performance. (A) Strong scaling on the full ImageNet-1K dataset (*1.3E6 *points). (B) Weak scaling on ImageNet-1K (*1.0E5 *points per rank). (C) Strong scaling on the extended CIFAR-10 dataset (*3.0E6 *points). (D) Weak scaling on CIFAR-10 (*5.0E4 *points per rank).*

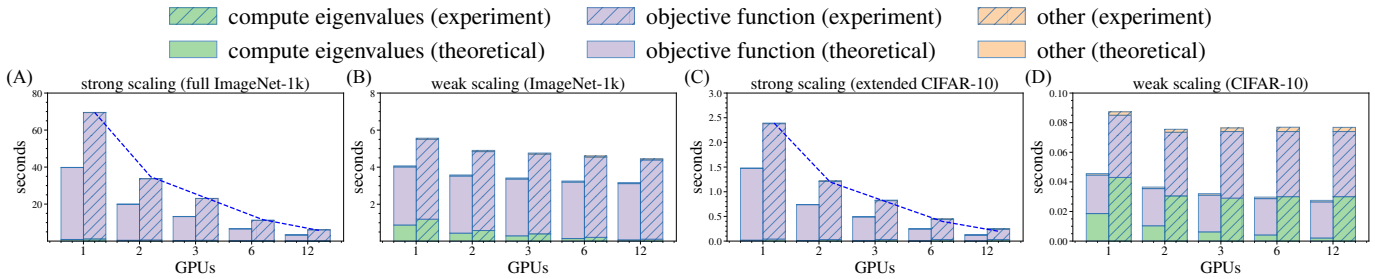

Figure 7 *Strong and weak scaling of the* ROUND *step on CIFAR-10 and ImageNet-1K. The dashed lines indicate ideal scaling performance. (A) Strong scaling on the full ImageNet-1K dataset (*1.3E6 *points). (B) Weak scaling on ImageNet-1K (*1.0E5 *points per rank). (C) Strong scaling on the extended CIFAR-10 dataset (*3.0E6 *points). (D) Weak scaling on CIFAR-10 (*5.0E4 *points per rank).*

the `cupy.linalg.eigvalsh` to compute eigenvalues which takes $\mathcal{O}(cd^3)$. We fit the prefactor to 300 doing a few experiments isolated to this function. The evaluation of Eq. (17) has time complexity $3cd^3 + 4ncd^2$. We utilize features from ImageNet-1K extracted using the pretrained self-supervised ViT models DINOv2 [22] with varying dimensions. Specifically, we explore feature dimensions d of 383, 766, and 1022. The number of classes is fixed at $c = 1000$, and we maintain a consistent number of points at 500,000. We set the number of random vectors $s = 10$ in $\mathbf{V}$. Fig. 5(A)(C) show the sensitivity results for both RELAX and ROUND steps. Each d value is represented by two adjacent columns. The left column displays the theoretical peak time for each d, while the right column shows the actual test time. Specifically, Fig. 5 (A) presents the results for the RELAX step. We conduct the RELAX step for one mirror descent iteration while keeping the number of CG iterations fixed at $n_{CG} = 50$. Increasing d from 383 to 766 leads to a 4.72× increase in the wall time of the preconditioner $\{\mathcal{B}_k(\mathbf{\Sigma}_z^{-1})\}_{k\in[c]}$, while the CG time increases by 1.7×. When d increases from 766 to 1022, the wall time of the preconditioner increases by 1.66×, and the CG time increases by 1.26×.

In the ROUND step, we conduct one iteration and showcase the results in Fig. 5(C). Increasing d from 383 to 766 results in a 6.6× increase in eigenvalue computation time. Additionally, the evaluation time for the objective function increases by 3.65×. Upon further increasing d from 766 to 1022, the time

required for eigenvalue decomposition increases by 2.08×, while the evaluation time for the objective function rises by 1.79×.

Sensitivity to class number c. Similarly, we examine the algorithm's sensitivity to the number of classes, c. As observed, the complexity of the RELAX step scales linearly with the number of classes, as does the construction of the preconditioner. Similarly, the two primary components of the ROUND step, namely, computing eigenvalues (line 9 in Algorithm 3) and evaluating the objective function (line 7 in Algorithm 3), also show linear scale to the number of classes. We conduct tests on the ImageNet dataset with 1.3 million points and a feature dimension of $d = 383$. The number of classes c varies from $100, 200, 400, 800, 1000$. Fig. 5 (B) illustrates the results of the RELAX step. When c increases from 100 to 200, the preconditioner cost increases by 2×, and the CG time increases by 1.79×. Conversely, for the scenario where c increases from 100 to 1000, the preconditioner time increases by 10.6×, and the CG time increases by 8.3×. In the ROUND step, we execute one iteration and present the results in Fig. 5 (D). As c increases from 100 to 200, the eigenvalue decomposition time increases by 2.08×, and the time for evaluating the objective function increases by 1.99×. Conversely, when c increases from 100 to 1000, the eigenvalue decomposition time increases by 10×, and the time for evaluating the objective function increases by 10.37×. Overall, the solver exhibits the expected scaling behavior.

C. Parallel scalability

We perform strong and weak scaling tests on our parallel implementation of Approx-FIRAL using two datasets. ❶ ImageNet-1K: the dimension of points is $d = 383$, and the number of classes is $c = 1000$. For the strong scaling test, we use the entire ImageNet-1K dataset with an unlabeled pool $\mathbf{X}_u$ containing $n = 1.3$ million points. In the weak scaling test, we allocate 0.1 million points to each GPU. ❷ CIFAR-10: the dataset has points with dimension of $d = 512$ and number of classes $c = 10$. In the strong scaling test, we expand CIFAR-10 by introducing random noise from ∼50K to 3 million points. For the weak scaling test, we allocate 50,000 points to each GPU.

We present strong and weak scaling results for the RELAX steps in Fig. 6 and the ROUND step in Fig. 7, employing up to 12 GPUs for both tests. In the RELAX step, we present the time for one mirror descent iteration. For the ROUND step, we report the time for selecting one point. For estimating the theoretical collective communication time costs for MPI operations, we assume a latency of $t_s = 1.0\text{E}{-4}$s, a bandwidth of $1/t_w = 2.0\text{E}10$ byte/s, and a computation cost per byte of $t_c = 1.0\text{E}{-10}$ s/byte. Additionally, for computation estimation, we maintain the use of 19.5TFLOPS peak performance of GPU A100 as in the previous section.

Scalability of RELAX step. The main computational cost in the RELAX step stem from the preconditioner setup and the CG solve. Regarding strong scaling results presented in Fig. 6(A) for ImageNet-1K and (C) for CIFAR-10, utilizing 12 GPUs leads to a speedup of 10.9× for the preconditioner setup and 11.3× for the CG solve in the case of ImageNet-1K. For CIFAR-10, the speedup for the preconditioner is 6.7×, while for CG, it reaches 8 when employing 12 GPUs. As for the weak scaling, with the number of GPUs raised to 12, the time increases by less than 10% for ImageNet-1K (Fig. 6(B)), and within 20% for CIFAR-10 (Fig. 6(D)). The primary increases in time are attributed to MPI communications. We present the ideal speedup as dashed lines in Fig. 6, with negligible variance in performance.

Scalability of ROUND step. In the ROUND step, communication costs are negligible, so we include the time in the "other" category in the plots of Fig. 7. In the strong scaling tests, employing 12 GPUs results in an 11.4× speedup for ImageNet-1K, as shown in Fig. 7(A), and achieves an 11.1× speedup for CIFAR-10, as seen in Fig. 7(C). Regarding weak scaling, the time slightly decreases when we increase the number of GPUs. This occurs because we evenly distribute the eigenvalue calculations across all GPUs. This effect is more pronounced in the case of ImageNet-1K, as shown in Fig. 7(B), compared to CIFAR-10 (Fig. 7(D)), since ImageNet-1K has 1000 classes while CIFAR-10 has only 10 classes. Similarly, we present the ideal speedup as dashed lines in Fig. 7, with negligible variance in performance.

Regarding the discrepancy between theoretical and experimental performance shown in Figs. 6 and 7, one cause is the performance of `cupy.einsum`, which is impacted by memory management and suboptimal kernel performance for certain input sizes. Additionally, the theoretical analysis includes certain constants related to specific kernels that have not been calibrated, such as the prefactors in the eigenvalue solvers, contributing to the gap.

V. CONCLUSIONS

We presented Approx-FIRAL, a new algorithm that is orders of magnitude faster than FIRAL. This improvement is achieved by replacing FIRAL's exact solutions with inexact iterative methods or block diagonal approximations, using randomized approximations for matrix traces, and approximating eigenvalue solves with block diagonal methods. Empirical results show that these approximations have minimal impact on sample selection effectiveness, as demonstrated by test accuracy across seven diverse datasets, including those with class imbalances. Furthermore, our muli-GPU implementation allows efficient scaling to large datasets such as ImageNet. Our open-source Python implementation allows interoperability with existing machine learning workflows.

Our approach has **several limitations**. First, we still use direct solvers in some parts of the code. Specifically, eigenvalue solves in the ROUND step and block factorization for our Hessian preconditioner are performed exactly. These methods are not scalable for certain parameters and could be replaced with sparsely preconditioned iterative solvers to enhance both performance and scalability of Approx-FIRAL. We aim to incorporate these improvements in future versions

of the algorithm. Second, we have not extended the theoretical results of FIRAL to the approximate version. While most of the matrices involved are symmetric positive definite and our approximations are stable perturbations, deriving precise error bounds requires detailed estimates of the approximation error.Third, despite its efficiency, Approx-FIRAL is still more resource-intensive compared to other methods. It performs best when the number of classes is relatively small, the feature embeddings are excellent, and only a few examples are needed for classification. As the number of classes grows, simpler methods may be more appropriate. Fourth, our testing has been limited to NVIDIA GPUs. Although the code is theoretically portable—CuPy supports AMD GPUs and it can be adapted for CPUs using NumPy—these alternative implementations have not yet been carried out.

A final, more fundamental limitation of the generic FIRAL approach is its inability to accommodate changes in the feature embedding as new examples are introduced. Typically, empirical methods address this by retraining or fine-tuning the embedding whenever new labels are obtained. In such cases, since the embedding evolves with new data, the data points themselves change, rendering the FIRAL theory inapplicable. Active learning with theoretical guarantees for such setups remains an open problem that probably requires an entirely different approach.

Acknowledgements

This material is based upon work supported by NSF award OAC 2204226; by the U.S. Department of Energy, Office of Science, Office of Advanced Scientific Computing Research, Applied Mathematics program, Mathematical Multifaceted Integrated Capability Centers (MMICCS) program, under award number DE-SC0023171; by the U.S. Department of Energy, National Nuclear Security Administration Award Number DE-NA0003969; and by the U.S. National Institute on Aging under award number R21AG074276-01. Any opinions, findings, and conclusions or recommendations expressed herein are those of the authors and do not necessarily reflect the views of the DOE, NIH, and NSF. Computing time on the Texas Advanced Computing Centers Stampede system was provided by an allocation from TACC and the NSF.

References

[1] Y. Bengio, A. Courville, and P. Vincent, "Representation learning: A review and new perspectives," *IEEE transactions on pattern analysis and machine intelligence*, vol. 35, no. 8, pp. 1798–1828, 2013.

[2] T. Chen, S. Kornblith, M. Norouzi, and G. E. Hinton, "A simple framework for contrastive learning of visual representations," 2020. [Online]. Available: https://arxiv.org/abs/2002.05709

[3] F. Zhuang, Z. Qi, K. Duan, D. Xi, Y. Zhu, H. Zhu, H. Xiong, and Q. He, "A comprehensive survey on transfer learning," *Proceedings of the IEEE*, vol. 109, no. 1, pp. 43–76, 2020.

[4] P. Ren, Y. Xiao, X. Chang, P.-Y. Huang, Z. Li, B. B. Gupta, X. Chen, and X. Wang, "A survey of deep active learning," *ACM computing surveys (CSUR)*, vol. 54, no. 9, pp. 1–40, 2021.

[5] Y. Chen and G. Biros, "Firal: An active learning algorithm for multinomial logistic regression," *Advances in Neural Information Processing Systems*, vol. 36, 2024.

[6] R. Nishino and S. H. C. Loomis, "CuPy: A numpy-compatible library for NVIDIA GPU calculations," *31st conference on neural information processing systems*, vol. 151, no. 7, 2017.

[7] W. Gropp, E. Lusk, and A. Skjellum, *Using MPI: portable parallel programming with the message-passing interface*. MIT press, 1999, vol. 1.

[8] L. Dalcin and Y.-L. L. Fang, "mpi4py: Status update after 12 years of development," *Computing in Science & Engineering*, vol. 23, no. 4, pp. 47–54, 2021.

[9] X. Li and Y. Guo, "Adaptive active learning for image classification," in *2013 IEEE Conference on Computer Vision and Pattern Recognition*, 2013, pp. 859–866.

[10] O. Sener and S. Savarese, "Active learning for convolutional neural networks: A core-set approach," *arXiv preprint arXiv:1708.00489*, 2017.

[11] D. Gissin and S. Shalev-Shwartz, "Discriminative active learning," *arXiv preprint arXiv:1907.06347*, 2019.

[12] G. Citovsky, G. DeSalvo, C. Gentile, L. Karydas, A. Rajagopalan, A. Rostamizadeh, and S. Kumar, "Batch active learning at scale," *Advances in Neural Information Processing Systems*, vol. 34, pp. 11 933–11 944, 2021.

[13] Y. Gal, R. Islam, and Z. Ghahramani, "Deep bayesian active learning with image data," in *International conference on machine learning*. PMLR, 2017, pp. 1183–1192.

[14] R. Pinsler, J. Gordon, E. Nalisnick, and J. M. Hernández-Lobato, "Bayesian batch active learning as sparse subset approximation," *Advances in neural information processing systems*, vol. 32, 2019.

[15] M. Hutchinson, "A stochastic estimator of the trace of the influence matrix for laplacian smoothing splines," *Communications in Statistics - Simulation and Computation*, vol. 19, no. 2, pp. 433–450, 1990. [Online]. Available: https://doi.org/10.1080/03610919008812866

[16] D. K. Panda, H. Subramoni, C.-H. Chu, and M. Bayatpour, "The mvapich project: Transforming research into high-performance mpi library for hpc community," *Journal of Computational Science*, vol. 52, p. 101208, 2021, case Studies in Translational Computer Science. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1877750320305093

[17] R. Thakur, R. Rabenseifner, and W. Gropp, "Optimization of collective communication operations in mpich," *Int. J. High Perform. Comput. Appl.*, vol. 19, no. 1, p. 49–66, feb 2005. [Online]. Available: https://doi.org/10.1177/1094342005051521

[18] L. Deng, "The mnist database of handwritten digit images for machine learning research," *IEEE Signal Processing Magazine*, vol. 29, no. 6, pp. 141–142, 2012.

[19] A. Krizhevsky, V. Nair, and G. Hinton, "Cifar-10 (canadian institute for advanced research)." [Online]. Available: http://www.cs.toronto.edu/~kriz/cifar.html

[20] F.-F. Li, M. Andreeto, M. Ranzato, and P. Perona, "Caltech 101," Apr 2022.

[21] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, A. C. Berg, and L. Fei-Fei, "ImageNet Large Scale Visual Recognition Challenge," *International Journal of Computer Vision (IJCV)*, vol. 115, no. 3, pp. 211–252, 2015.

[22] M. Oquab, T. Darcet, T. Moutakanni, H. V. Vo, M. Szafraniec, V. Khalidov, P. Fernandez, D. Haziza, F. Massa, A. El-Nouby, R. Howes, P.-Y. Huang, H. Xu, V. Sharma, S.-W. Li, W. Galuba, M. Rabbat, M. Assran, N. Ballas, G. Synnaeve, I. Misra, H. Jegou, J. Mairal, P. Labatut, A. Joulin, and P. Bojanowski, "Dinov2: Learning robust visual features without supervision," 2023.

[23] 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*, vol. 12, pp. 2825–2830, 2011.

[24] Nvidia, "Nvidia a100 tensor core gpu architecture," https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/nvidia-ampere-architecture-whitepaper.pdf, 2020, accessed: Apr. 2, 2024.

Appendix: Artifact Description/Artifact Evaluation

Artifact Description (AD)

I. Overview of Contributions and Artifacts

A. Paper's Main Contributions

C_1 We propose Approx-FIRAL to accelerate Exact-FIRAL by exploiting matrix structure, randomized linear algebra, and iterative methods.

C_2 We evaluated the accuracy of Approx-FIRAL by comparing to Exact-FIRAL and several other popular active learning methods.

C_3 We used CuPy and MPI to support multi-GPU acceleration in Python.

C_4 We tested the scalability of our parallel implementation on multi-GPU systems.

B. Computational Artifacts

The computational artifacts A_1 and A_2 can be accessed at the following link:

https://zenodo.org/doi/10.5281/zenodo.10981845.

There are two subdirectories included. The directory "accuracy_tests/" contains artifact A_1, the directory "scalability_tests/" contains artifact A_2.

Artifact ID	Contributions Supported	Related Paper Elements
A_1	C_1, C_2	Figures 2-4
A_2	C_3, C_4	Figures 6-8

II. Artifact Identification

A. Computational Artifact A_1

Relation To Contributions

This artifact includes the implementation of our proposed algorithm, Approx-FIRAL, and code for other active learning methods such as Exact-FIRAL, Random, K-means, and Entropy. Through numerical experiments, we demonstrate that ❶ Approx-FIRAL exhibits similar performance to Exact-FIRAL in active learning, ❷ Approx-FIRAL accelerates the Exact-FIRAL significantly.

Expected Results

By conducting active learning experiments on multiple datasets (refer to Table V), we reproduce classification accuracy results presented in Figures 2 and 4. It is expected that the newly proposed Approx-FIRAL will exhibit performance similar to Exact-FIRAL and surpass the other methods being compared. By comparing the execution time of the RELAX step and ROUND step between Approx-FIRAL and Exact-FIRAL on the ImageNet-50 dataset, we replicate the time comparison results within Figure 3. It is expected that Approx-FIRAL significantly accelerates Exact-FIRAL in both the RELAX step and the ROUND step.

Expected Reproduction Time (in Minutes)

All tests associated with artifact A_1 can be performed on a single A100 GPU node within the TACC Lonestar6 system. Each A100 GPU node is equipped with 3 NVIDIA A100 PCIE 40GB GPUs. Subdirectory `/job_accuracy/` contains all bash scripts for the experiments to replicate Figures 2-4. The estimated execution time for all scripts are listed in the table below.

Figure	bash script	time (min)
Figure 2	`mnist.sh`	2
	`cifar10.sh`	4
	`imagenet50.sh`	55
Figure 3	`time_imagenet50.sh`	2
	`time_caltech101.sh`	21
Figure 4	`caltech101.sh`	4
	`imagenet1k.sh`	100

Artifact Setup (incl. Inputs)

Hardware: We conducted experiments by executing code on the NVIDIA A100 GPU on the TACC Lonestar6 system.

Software: We set the TACC environment as follows:

intel/19.1.1	impi/19.0.9	autotools/1.4
cmake/3.24.2	pmix/3.2.3	xalt/2.10.32
TACC		

Furthermore, the following packages are necessary to run the active learning experiments: python/3.11.8, numpy/1.24.3, scipy/1.10.1, cupy/13.0.0, mpi4py/3.1.5, absl-py/1.3.0, scikit-learn/1.2.2. For plotting results, we utilize matplotlib/3.7.1 and seaborn/0.12.2.

Datasets / Inputs: We have seven datasets obtained by real-world image datasets with features acquired through self-supervised learning methods: MNIST, CIFAR-10, imb-CIFAR-10, ImageNet-50, imb-ImageNet-50, Caltech-101, and ImageNet-1k. The specifics of each dataset are outlined in Table V. Our datasets can be accessed via the following link: https://zenodo.org/doi/10.5281/zenodo.10977375.

Installation and Deployment Once the testing environment has been configured (refer to `set_env.sh`), we proceed to set the path for downloaded data and the path for saving active learning results by exporting them (refer to `set_path.sh`).

Artifact Execution

To replicate the active learning accuracy performance results depicted in Figures 2 and 4, the execution of the following five scripts is required: `mnist.sh`, `cifar10.sh`, `imagenet50.sh`, `caltech101.sh`, and `imagenet1k.sh`. Besides, running `time_imagenet50.sh` provides the results necessary to replicate the time comparison results of ImageNet-50 as depicted in Figure 3.

For each dataset, the respective active learning bash script carries out active learning experiments utilizing methods like Approx-FIRAL, Random, K-means, and Entropy. With the exclusion of tests conducted on Caltech-101 and ImageNet-1k, each script also runs Exact-FIRAL. In the case of random number-based methods such as Random and K-means, results are stored for 10 different random seeds.

For the details about the dataset size, dimension, number of classes, active learning rounds and batch size, one can refer to Table V.

Both Exact-FIRAL and Approx-FIRAL involve RELAX and ROUND steps. For the RELAX step of Exact-FIRAL, we set the mirror descent iteration number to be 200. we configure the number of mirror descent iterations to be 200. This iteration count is deemed sufficient, as the relative change of the objective typically falls below 1.E-6 in all tests. For the RELAX step of Approx-FIRAL, we establish the termination condition for mirror descent as reaching either 40 iterations or achieving a relative change in the objective function of less than 1.E-4. In addition, we use 10 Rademacher random vectors for trace estimation and we terminate CG iterations when the relative residual falls below 0.1. In the ROUND step of both Exact-FIRAL and Approx-FIRAL, we perform a grid search for the hyperparameter learning rate η to maximize the minimum eigenvalue of the sum of Hessian matrices of the selected points. To replicate Figures 2 and 4, we provide the values for η in our datasets.

In order to ensure fair comparison for the computation time depicted in Figure 3, we maintain a fixed mirror descent iteration number of 40. All other hyperparameter settings remain consistent with those previously stated.

Artifact Analysis (incl. Outputs)

Once all the bash scripts associated with the active learning experiments have been executed, running `plot_results.sh` will handle the post-processing and visualization of the active learning results, thereby reproducing the results depicted in Figures 2-4.

B. Computational Artifact A_2

Relation To Contributions

This artifact leverages CuPy and MPI to enable parallel execution across multiple GPUs. We used NVIDIA's nvtx to profile our codes. It applies to contributions C_3 and C_4.

Expected Results

We expect nearly perfect scaling for both strong and weak scaling runs as the communication costs comprise `MPI_Allreduce` operations of modest size. Therefore for strong scaling we expect $\mathcal{O}(1/p)$ costs where p is the number of GPUs. For weak scaling we expect constant time for fixed grain size (problem size per GPU). We also test algorithmic scaling on a single A100 when varying the feature dimension d and the number of classes c.

Expected Reproduction Time (in Minutes)

We time RELAX and ROUND separately. RELAX is tested for a single iteration of mirror descent, while ROUND is tested for its second iteration.

RELAX: In the ImageNet-1k dataset, the anticipated strong scaling computational time for this artifact is approximately 2 minutes for 1 GPU, 1 minute for 2 GPUs, 0.6 minutes for 3 GPUs, 0.3 minutes for 6 GPUs, and 0.15 minutes for 12 GPUs. The expected weak scaling computational time from 1 to 12 GPUs is approximately 0.17 minutes.

For the CIFAR-10 dataset, the expected strong scaling computational time is about 0.117 minutes for 1 GPU, 0.067 minutes for 2 GPUs, 0.05 minutes for 3 GPUs, 0.03 minutes for 6 GPUs, and 0.016 minutes for 12 GPUs. The anticipated weak scaling computational time from 1 to 12 GPUs is approximately 0.003 minutes.

We conduct sensitivity tests on RELAX regarding feature size d and the number of classes c using a single GPU-A100 and the ImageNet-1k dataset. For varying feature sizes d, the computational times are approximately 0.13 minutes for $d = 383$, 0.32 minutes for $d = 766$, and 0.45 minutes for $d = 1022$. Regarding the number of classes c, the computational times are approximately 0.21 minutes for $c = 100$ and 1.91 minutes for $c = 1000$.

ROUND: In the ImageNet-1k dataset, the anticipated strong scaling computational time for this artifact is approximately 1.17 minutes for 1 GPU, 0.58 minutes for 2 GPUs, 0.38 minutes for 3 GPUs, 0.2 minutes for 6 GPUs, and 0.12 minutes for 12 GPUs. The expected weak scaling computational time from 1 to 12 GPUs is around 0.09 minutes.

For the CIFAR-10 dataset, the expected strong scaling computational time is approximately 0.038 minutes for 1 GPU, 0.02 minutes for 2 GPUs, 0.015 minutes for 3 GPUs, 0.008 minutes for 6 GPUs, and 0.004 minutes for 12 GPUs. The anticipated weak scaling computational time from 1 to 12 GPUs is about 0.0013 minutes.

We test the sensitivity of the single-GPU wall-clock time when varying the feature dimension d and the number of classes c using the ImageNet-1k dataset. With varying d, the computational times are approximately 0.1 minutes for $d = 383$, 0.42 minutes for $d = 766$, and 0.78 minutes for $d = 1022$. With varying c, the computational times are approximately 0.12 minutes for $c = 100$ and 1.17 minutes for $c = 1000$.

Artifact Setup (incl. Inputs)

Hardware: we test our results in GPU-A100 of TACC Lonestar6 system.
Software: we set environment of TACC as below

gcc/11.2.0	mvapich2-gdr/2.3.7	cmake/3.24.2
pmix/3.2.3	xalt/2.10.32	TACC
cuda/11.4	nccl/2.11.4	

In addition, we install python/3.11.8, cupy/13.0.0, mpi4py/3.1.5, numpy/1.24.3 and nvtx packages.

Dataset: In the strong scaling experiments, we used the entire ImageNet-1k dataset and an extended version of the CIFAR-10 dataset. The CIFAR-10 dataset is expanded by introducing Gaussian random noise to its features, thereby increasing the dataset size from 50,000 data points to 3 million data points. These datasets are available here: https://zenodo.org/doi/10.5281/zenodo.10977375.

The ImageNet-1k dataset can be prepared by executing

```
python -u prepare_data.py --mode
'strong_scaling' --dataset
'imagenet1M_d383'
```

The CIFAR-10 dataset can be prepared (on 3 GPUs) by executing

```
python -u prepare_data.py --mode
'weak_scaling' --dataset 'cifar10',
--nproc 3
```

The code has an option `--output` to save the dataset in the given output path.

Deployment: Before running the following test, users should ensure they have installed all required packages and prepared the dataset as described above.
RELAX: for strong scaling tests with the ImageNet dataset with `X` number of GPUs, users run the following code

```
ibrun -n X nsys profile -t
nvtx,osrt --force-overwrite=true
--output=nvtx_mpi_imagenet_relax
python -u relax_test.py --nproc X
--mode 'strong_scaling' --dataset
'imaegnet1M_d383'
```

For weak scaling tests with the CIFAR-10 dataset with `X` number of GPUs, users run

```
ibrun -n X nsys profile -t
nvtx,osrt --force-overwrite=true
--output=nvtx_mpi_cifar_relax python
-u relax_test.py --nproc X --mode
'weak_scaling' --dataset 'cifar10'
```

ROUND: for strong scaling tests with the ImageNet dataset with `X` number of GPUs, users run

```
ibrun -n X nsys profile -t nvtx,osrt
--force-overwrite=true --stats=true
--output=nvtx_mpi_imagenet_round
python -u round_test.py --nproc X
--mode 'strong_scaling' --dataset
'imagenet1M_d383' --d 383
```

For weak scaling tests with the CIFAR-10 dataset with `X` number of GPUs, users run

```
ibrun -n X nsys profile -t nvtx,osrt
--force-overwrite=true --stats=true
--output=nvtx_mpi_cifar_round python
-u round_test.py --nproc X --mode
'weak_scaling' --dataset 'cifar10'
```

Both RELAX and ROUND codes support testing sensitivity to d. Users can modify the `--dataset` option to be `imagenet1M_d383`, `imagenet1M_d766`, or `imagenet1M_d1022`, and adjust the corresponding `--d` option to `383`, `766`, or `1022`. To test sensitivity on c, users can vary the `--c` option from `100` to `1000`.

Artifact Execution

Users first run `prepare_data.py` to generate inputs of RELAX and ROUND codes, and then test the strong/weak scaling or the sensitivity experiment.

Artifact Analysis (incl. Outputs)

Users can retrieve the `*.qdrep` file from the specified path in the `--output` options mentioned above. The timing performance is obtained using NVIDIA Nsight Systems, and the results should correspond to those depicted in Figures 6-8.

Artifact Evaluation (AE)

A. Computational Artifact A_1

Artifact Setup (incl. Inputs)

Download the unzip the file from https://zenodo.org/doi/10.5281/zenodo.10981845. The subdirectory `/accuracy_tests/` contains artifact A_1. All tests associated with artifact A_1 can be performed on a single A100 GPU node within the TACC Lonestar6 system. Each A100 GPU node is equipped with 3 NVIDIA A100 PCIE 40GB GPUs. The setup process for the artifact involves the following steps:

1) Install anaconda.

2) Create an environment named `firal` in conda and install all packages listed in `requirements.txt` (or using `requirements_pip.txt` for pip install):

```
conda create -n firal -c conda-forge
python=3.11.0
conda activate firal
conda install --yes --file
requirements.txt
```

3) Install `mpi4py`:

```
module unload python3
env MPICC=mpicc
python3 -m pip install --user mpi4py
```

4) Download and unzip `data_accuracy.tar.gz` from https://zenodo.org/doi/10.5281/zenodo.10977375. This directory contains all datasets for the tests related to Artifact A_1 and replicate Figures 2-4. In addition, set the environment variable for the data directory:

```
export DATA=''the absolute path of the
data directory''
```

5) Create a directory to save results and set the environment variable for the results directory:

```
mkdir results
export SAVE=''the absolute path of the
results directory''
```

Artifact Execution

The directory `/jobs_accuracy/` contains all the bash scripts needed to run the experiments for replicating Figures 2-4 in the paper. Each script operates **independently**. The experiments related to each figure are explained as follows:

1) To replicate Figure 2, run the bash scripts `mnist.sh`, `cifar10.sh`, and `imagenet50.sh`. Each script executes an active learning task on one/two different datasets using the following active learning methods: Exact-FIRAL, Approx-FIRAL, Random, K-means, and Entropy.

   For instance, using the `mnist.sh` script as an example, the following command executes one round of an active learning task using the Exact-FIRAL method for the MNIST dataset by first running the RELAX solve followed by the ROUND solve. "test" represents the dataset name, r indicates which round of active learning is to run, b denotes the budget for sampling, "task" means RELAX solve or ROUND solve is to run. Note that for each round of the active learning, it depends on the results of the previous rounds, ROUND solve results depends on the RELAX solve results. The execution of Approx-FIRAL method is similar to Exact-FIRAL.

```
python3 ../run_exact_firal.py
--test='mnist' --r=1 --b=10
--task='relax'
python3 ../run_exact_firal.py
--test='mnist' --r=1 --b=10
--task='round'
```

   The Random and K-means methods are random approaches. For each dataset, these two methods are executed with 10 different random seeds within the script.

2) To replicate Figure 3, run the bash scripts `time_imagenet50.sh` and `time_caltech101.sh` for the time comparison between Exact-FIRAL and Approx-FIRAL on one round of active learning task using datasets ImageNet-50 and Caltech-101.

3) To replicate Figure 4, run the bash scripts `caltech101.sh` and `imagenet1k.sh` for caltech-101 and ImageNet-1K, respectively.

   Each script executes active learning tests on a specific dataset using the following active learning methods: Approx-FIRAL, Random, K-means, and Entropy. Exact-FIRAL is not included here due to its high time requirements and the limited storage available for these two large-scale datasets.

   Note that the execution of the ROUND solve for ImageNet-1K in `imagenet1k.sh` uses multiple GPUs by `mpi4py`. For example, there are 3 GPUs available in an A100 GPU node on TACC Lonstar6, we can run the following command by set $X = 3$:

```
ibrun -n X python3 ../run_round_approx_mpi.py
--test='imagenet1k' --ppn=X --r=1
--print_unit=50 --b=200 --log_C=0.1
```

Artifact Analysis (incl. Outputs)

After running experiments, results for Figures 2-4 can be plotted by using python script `plot_results.py`. Note that all the plots for Figures 2-4 will be saved in the directory `results/accuracy_plot`. We illustrate the details for the analysis of each figure as follows:

1) To replicate Figure 2, run:

```
python3 plot_results.py --figureid=2
```

The expected outcomes are twofold. Firstly, our proposed fast algorithm, Approx-FIRAL, achieves similar active learning accuracy to Exact-FIRAL. Additionally, both Approx-FIRAL and Exact-FIRAL outperform other active learning methods across different datasets. This relates to the evaluation of our contribution C_2.

2) To replicate Figure 3, run:

```
python3 plot_results.py --figureid=3
```

The key result is that Approx-FIRAL is much faster than the Exact-FIRAL on the active learning test on datasets ImageNet-50 and Caltech-101. This is related to the evaluation of our contribution C_1.

3) To replicate Figure 4, run:

```
python3 plot_results.py --figureid=4
```

The anticipated result is that Approx-FIRAL will outperform other active learning methods on tests involving large datasets such as Caltech-101 and ImageNet-1K. This relates to the evaluation of our contribution C_2.

B. Computational Artifact A_2

Artifact Setup (incl. Inputs)

Download and unzip the file from https://zenodo.org/doi/10.5281/zenodo.10981845. The subdirectory `/scalability_tests/` contains artifact A_2. Associated tests are performed on various number of GPU-A100 (1, 2, 3, 6, 12 GPUs) from TACC Lonestar6 system. We deploy the code following these steps:

1) Load the following modules in TACC: gcc/11.2.0, mvapich2-gdr/2.3.7, cmake/3.24.2 pmix/3.2.3, xalt/2.10.32, TACC, cuda/11.4 and nccl/2.11.4.

2) Create an environment named `firal_mpi` in conda and install packages as following:

   - `conda create -n firal_mpi -c conda-forge python=3.11.0`
   - `conda activate firal_mpi`
   - `conda install -c conda-forge cupy cuda-version=11.4`
   - `CC=gcc CXX=g++ pip install mpi4py --no-cache-dir --no-binary :all:`
   - `conda install -c conda-forge nvtx`

3) Set the following:

   - `export MV2_USE_CUDA=1`
   - `export MV2_USE_ALIGNED_ALLOC=1`
   - `export LD_PRELOAD= /opt/apps/gcc11_2/mvapich2-gdr/2.3.7/lib64/libmpi.so`
   - `export CUDA_HOME=$TACC_CUDA_DIR`

4) Download and unzip the following files into a folder named `data_scalability/` from https://zenodo.org/doi/10.5281/zenodo.10977375. These datasets are necessary for running artifact A_2 and replicating Figures 6-8.

   - cifar10-data.tar.gz
   - dinov2-features-small-transform.tar.gz
   - dinov2-features-base-transform.tar.gz
   - dinov2-features-large-transform.tar.gz
   - imagenet_idx.tar.gz

5) Load necessary environment by running:

```
source set_env.sh
```

6) Set variable DATASET as the absolute path of `data_scalability/` by:

```
export DATASET=''absolute path of data_scalability/''
```

Artifact Execution

The directory `jobs_scalability/` contains all the bash scripts needed to run the experiments for Figures 6-8. The experiments for Figure 6 test scaling performance on a single GPU, while the experiments for Figures 7-8 test scaling performance on up to 12 GPUs. **We recommend that users warm up the machine by running one of our scalability test scripts multiple times before conducting any tests. Additionally, we advise repeating the test until the performance stabilizes.** Users should run the bash files under directory `scalability_tests/jobs_scalability/`. We split all experiments into 2 main steps:

STEP 1: we run jobs in interactive session by using 2 A100 GPU nodes in TACC Lonestar6 equipped with 6 GPUs:

```
idev -p gpu-a100-dev -N 2 -n 6 -t 02:00:00
```

Then we run jobs as follows:

1) First of all, run bash script `prepare_dataset.sh` to prepare data necessary for the tasks followed.

2) To replicate Figure 6, run the bash `RELAX_step_sensitivity_test.sh` and `ROUND_step_sensitivity_test.sh`. The first bash file runs sensitivity test regarding variable feature size d and number of class c on RELAX step. The second bash file runs for ROUND step. Within the bash file, user will see several options for our python programs. We profile our python code by nvtx for timing purpose, and which is executed by `nsys`. `--output` represents where to save nvtx file. `--mode` helps define size of the dataset we use to test. `--dataset` represents the name of the dataset we use for test. `--d` and `--c` are the variables of interest of our sensitivity test.

3) To replicate part of the scaling tests in Figure 7 upto 6 GPUs, run the bash files `RELAX_step_scaling_test_imagenet1M.sh` and `RELAX_step_scaling_test_cifar10.sh`. These two files test strong and weak scaling for RELAX solve on datasets ImageNet-1K and CIFAR-10.

4) To replicate part of the scaling tests in Figure 8 upto 6 GPUs, run the bash file `ROUND_step_scaling_test_imagenet1M.sh` and `ROUND_step_scaling_test_cifar10.sh`. These two files test strong and weak scaling on two datasets ROUND solve on datasets ImageNet-1K and CIFAR-10.

STEP 2: To finish the scaling tests in Figure 7 and Figure 8 with 12 GPUs, we first exit the interactive session after finishing STEP 1. Then we submit the Slurm job script to TACC server by running:

```
sbatch job_MPI_ngpu12.sh
```

Artifact Analysis (incl. Outputs)

After running the experiments, we can generate plots for Figures 6-8 by using the Jupyter notebook file `FIRAL_scalability.ipynb` (or equivalently, using python script `FIRAL_scalability_plot.py`). The plot results will be saved in the folder `scalability_plot_results`. Using the NVTX profile, we can identify the time consumption for each component of the artifact. These files can be read by NVIDIA Nsight Systems. To automate the evaluation, we use `cupy.cuda.get_elapsed_time()` and save the time consumption in .npz files under the `time_folder`. The Jupyter notebook `FIRAL_scalability.ipynb` or python script `FIRAL_scalability_plot.py` reads these .npz files to plot Figures 6-8. We expect nearly perfect scaling performance for both strong and weak scaling tests. The sensitivity tests on feature size d and the number of classes c should also align with the complexity order presented in the paper.