# HaCore: Efficient Coreset Construction with Locality Sensitive Hashing for Vertical Federated Learning

**Qinbo Zhang[1*], Xiao Yan[2*], Yukai Ding[1], Fangcheng Fu[3], Quanqing Xu[4], Ziyi Li[1], Chuang Hu[1†], Jiawei Jiang[1†]**

[1]School of Computer Science, Wuhan University
[2]Centre for Perceptual and Interactive Intelligence (CPII)
[3]School of Computer Science, Peking University
[4]OceanBase, Ant Group
{qinbo_zhang, yukai.ding, lzy0323, handc, jiawei.jiang}@whu.edu.cn, yanxiaosunny@gmail.com, ccchengff@pku.edu.cn, xuquanqing.xqq@oceanbase.com

## Abstract

Vertical federated learning (VFL) trains model when the features of data samples are scattered over multiple clients. To improve efficiency, a promising approach is to find a coreset of the data samples and use it as a smaller training set. However, existing methods produce a large coreset when there are many clients and have long running time. To address these problems, we propose HaCore for efficient coreset construction in VFL setting. HaCore first employs locality sensitive hashing (LSH) to map features to bit signatures locally on the clients, and then merges the local signatures for $k$-medoids clustering. Data samples that correspond to the medoids are added to the coreset. The core idea is that the distance of original data samples can be approximated by the Hamming distance between their LSH-based bit signatures. To accelerate $k$-medoids, we utilize an inverted index to search the nearest medoid and a bit-counting method to quickly compute the aggregate distance from many signatures to a medoid. We evaluate HaCore on 5 datasets and compare with state-of-the-art coreset construction methods for VFL. The results show that HaCore accelerates the best-performing baseline by over 45× and matches the accuracy of training with all samples.

## Introduction

Federated learning (FL) (Konečnỳ et al. 2016; Li et al. 2020; Zhu et al. 2021) is a distributed machine learning paradigm where data are partitioned over participants and cannot be shared due to privacy concerns. FL has two categories, i.e., horizontal FL (HFL) (Zhang et al. 2021; Wei et al. 2020) and vertical FL (VFL) (Yang et al. 2019; Liu et al. 2019, 2024). In HFL, the participants have different data samples; while in VFL, the participants have different features for the same set of data samples. We focus on VFL, which arises when different institutions hold distinct data about the same entities. VFL has applications in many areas such as healthcare, finance, and the Internet of Things (IoT). For instance, each bank maintains its own financial records of customers, and they can collaborate using VFL to train a model to predict the financial risks of customers.

VFL typically employs the SplitNN (Vepakomma et al. 2018) framework to avoid direct data sharing, which divides the machine learning model into two parts, i.e., a top part and a bottom part. Each client processes local data using its bottom model to generate activations, which are sent to a server. The server concatenates the activations from all clients to train the top model and sends the gradients of the activations back to the clients to perform backward propagation. SplitNN conducts sample-wise communication because it needs to transfer activation and gradient for each training sample, and thus its communication cost increases linearly with the size of the training set. To reduce the communication cost and training time, we can reduce the number of training samples. This is achieved by constructing a coreset that contains representative training samples (Bachem, Lucic, and Krause 2017; Feldman, Schmidt, and Sohler 2020) and often does not compromise model accuracy.

**Existing researches.** V-coreset (Huang et al. 2022) and TreeCSS (Zhang et al. 2024) extend coreset to VFL. V-coreset uses the orthogonal basis and local sensitivity as selection weights to choose training samples for the coreset. TreeCSS employs a clustering-based method to generate fingerprints for each sample on the clients and constructs the coreset by selecting the most valuable samples that share the same fingerprints. However, the two methods have critical limitations. ❶ V-coreset tailors coreset construction for each machine learning model and supports only linear regression and unsupervised clustering but classification models are also popular in practice. ❷ TreeCSS can produce an excessively large coreset when there are many clients. This is because to represent some samples using one representative, TreeCSS requires the samples to share identical signatures over all the clients. The large coreset harms efficiency but using a smaller coreset degrades accuracy. ❸ When dealing with high feature dimension and large data volume, the k-means clustering on each client becomes slow for TreeCSS. Given the shortcomings of existing researches, we pose the following research question:

*Is it possible to design a coreset construction method that*

[*]These authors contributed equally.
[†]Corresponding author.

*preserves model accuracy, allows fast execution, and generalize across different machine learning models?*

**Our solution HaCore.** We address the research question with HaCore by solving two technical problems.

❶ *How to select representative samples for the coreset?* HaCore adopts $k$-medoids (Park and Jun 2009; Schubert and Rousseeuw 2019) and chooses one sample to represent each medoid. This preserves model accuracy since similar samples are clustered together and is widely used in centralized learning (Sener and Savarese 2017; Bachem, Lucic, and Krause 2017). However, in VFL, we cannot transfer all features to a centralized server to compute distance due to privacy concerns. To address this, HaCore adopts locality sensitive hashing (LSH) (Gionis et al. 1999; Andoni et al. 2014; Jafari et al. 2021). In particular, HaCore first utilizes LSH to map the original features to binary signatures locally on each client, then concatenates the binary signatures from all clients for each sample, and finally conducts $k$-medoids on these signatures. The rationale is that LSH ensures that the Hamming distance of the binary signatures approximates the Euclidean distance of the original features. We also devise a method to determine the hash signature length for each client and show that using LSH maintains data privacy.

❷ *How to accelerate the construction of the coreset?* HaCore removes the dependency of running time on feature dimension because the length of the hash signature can be smaller than the feature dimension. That is, LSH can also be used for dimension reduction. However, $k$-medoids still requires to search the nearest medoid for each sample (i.e., assignment) and choose the sample with the minimum distance to all samples as the medoid for each cluster (i.e., selection). Assignment is expensive when there are many medoids, and naive selection costs $O(n^2)$ when a cluster contains $n$ samples. To accelerate assignment, we observe that it is inherently a similarity search problem and thus build an inverted index for the medoids such that each sample only needs to check a small number of medoids. For selection, we devise a bit-counting method to compute the aggregate Hamming distance from a sample to all samples, which reduces the selection cost to $O(n)$.

To evaluate HaCore, we compare with 4 baselines and experiment with 5 datasets, which encompass both classification and regression tasks. The results show that HaCore outperforms V-coreset and TreeCSS in model accuracy while reducing the end-to-end training time (i.e., coreset construction plus model training) by up to 45x. The results also show that HaCore achieves a model accuracy comparable to full-data training without using many hash functions and coreset samples. Besides, HaCore performs well when varying the number of clients, and our optimizations are effective.

To summarize, we make the following contributions:

- We observe that existing methods for VFL coreset construction suffer from a lack of generality, long running time, and large coreset size.
- We propose HaCore, which uses locality sensitive hashing (LSH) to implement $k$-medoids clustering in VFL setting. HaCore solves the problems of existing methods by clustering on the binary signatures generated by LSH.
- We accelerate the $k$-medoids clustering on binary signatures by using an inverted index for the assignment step and a bit-counting method for the selection step.

## Related Work

**Speeding vertical federated learning (VFL).** Fu (2022b) implements local update techniques to reduce client-server communication rounds in VFL by caching and reusing outdated statistics to estimate model gradients. Feng (2022) introduces VFLFS for collaborative feature selection in VFL, particularly in cases with non-overlapping sample scenarios. It employs $l_2$ constraints on feature weights and uses an autoencoder to identify key features for deep VFL models. Jiang (2022) proposes to select clients that complement each other to carry out VFL, using a mutual information estimator based on the Fagin algorithm for group testing between clients, rather than focusing on reducing the number of clients. Li (2023) proposes a Gaussian stochastic dual-gate to approximate the probability of feature selection while ensuring data privacy with Partially Homomorphic Encryption, without needing a trusted third party. Our approach HaCore is orthogonal to the client and feature selection techniques, as it can be applied after the clients and features have been determined for final model training.

**Coreset.** Coresets select representative samples from a dataset, enabling models trained on this subset to achieve accuracy similar to those trained on the entire dataset. This approach reduces the number of training samples needed and speeds up model training across various tasks, such as unsupervised clustering, regression and low-rank approximation (Feldman and Langberg 2011; Drineas, Mahoney, and Muthukrishnan 2006; Cohen, Musco, and Musco 2017; Lucic et al. 2017). V-coreset introduces coreset techniques to VFL (Huang et al. 2022), customizing methods for linear regression and $k$-means clustering. For linear regression, it selects samples based on weighted importance derived from orthonormal feature bases. For $k$-means, it uses local sensitivity to identify coreset samples. TreeCSS (Zhang et al. 2024) implements a clustering-based strategy where each client clusters its local data and represents features with cluster labels. These labels are then aggregated into global fingerprints for each sample. Representative samples are selected based on their summed distances from the $k$-means medoids for each fingerprint type.

## Preliminaries

**SplitNN.** Here, we introduce the SplitNN framework that provides a general and standard procedure for VFL. Typically, there are $M$ participants (clients) and an aggregation server. Among the clients, there is one special client, called the label owner, that owns all the labels. The goal is to collaboratively train a machine learning model on $N$ data samples $D = \{(x_i \in \mathbb{R}^F, y_i \in \mathbb{N})\}_{i=1}^{N}$, where $x_i$ represents the features of the $i^{\text{th}}$ sample, and $y_i$ is the corresponding label. We use $[N] = \{1, \cdots, N\}$ to denote the identifiers of the samples and $[M] = \{1, \cdots, M\}$ to denote the set of all

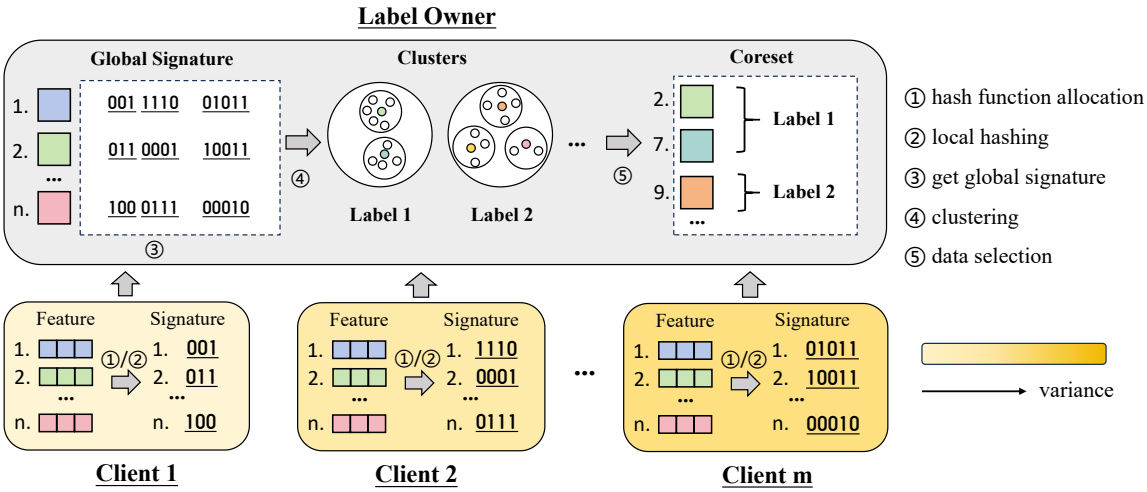



Figure 1: An Illustration of HaCore framework.

clients. Note that, we assume that all participants agree on the same identifier of each data sample.

Due to the vertical partition scheme, every feature vector $x_i$ is partitioned over the $M$ clients, and a client $m$ holds some features of all the samples, i.e., $D_m = \{x_i^m \in \mathbb{R}^{F_m} : m \in [M]\}_{i=1}^N$, where $F_m$ denotes the number of local features on client $m$. This gives $\sum_{m=1}^M F_m = F$. To facilitate training, the global model $f(\theta)$ is partitioned into a bottom model $f_b(\theta_b)$ and a top model $f_t(\theta_t)$. Each client $m$ owns a segment of the bottom model $f_b^m(\theta_b^m)$ that works on its local features. Thus, the loss function can be expressed as:

$$\begin{aligned} \mathcal{L}(\mathbf{Y}', \mathbf{Y}) &:= \mathcal{L}(f_t([h_1, \cdots, h_m, \cdots, h_M], \theta_t), \mathbf{Y}) \\ with \quad h_m &= f_b^m(\mathbf{X}_m, \theta_b^m) \end{aligned} \tag{1}$$

A min-batch of model training works as follows:

- The local feature vectors $\mathbf{X}_m$ (i.e., $\{x_i^m\}_{i=1}^N$) are processed by the clients using their bottom models (i.e., $f_b^m(\theta_b^m)$) to produce intermediate outputs $h_m$.
- The server concatenates the activations as $h = [h_1, \cdots, h_m, \cdots, h_M]$, processes them with top model: $\mathbf{Y}' = f_t(h, \theta_t)$, and sends $\mathbf{Y}'$ to the label owner.
- The label owner computes the loss function $\mathcal{L}(\mathbf{Y}', \mathbf{Y})$ with $\mathbf{Y}$ (i.e., $\{y_i\}_{i=1}^N$ ) and generates $\nabla_{Y'}\mathcal{L}(\mathbf{Y}', \mathbf{Y})$, which is the gradient for $\mathbf{Y}'$. It is then sent to the server.
- The server updates the top model with the gradient $\nabla_{Y'}\mathcal{L}(\mathbf{Y}', \mathbf{Y})$, and computes gradients $\nabla_{h_m}\mathcal{L}(\mathbf{Y}', \mathbf{Y})$ for each client $m$, which are then used by the clients to update their local bottom models.

As shown in this procedure, the activation/gradient exchange is sample-wise. Thus, if we can lessen the number of samples, we can reduce the overall training complexity.

**Locality sensitive hashing (LSH).** LSH maps data points into buckets using a set of hash functions, with nearby points more likely to be hashed into the same bucket. A hash collision occurs when two points share the same hash value. Under a LSH function, the chance of such collision, also called collision probability, depends on how similar they are. We call a hash family $\mathcal{H}$ is $(r_1, r_2, p_1, p_2)$-sensitive for distance function $d(x, y)$ if for any $h \in \mathcal{H}$:

$$\begin{aligned} &\text{if } d(x,y) \le r_1 \text{ then } Pr[h(x) = h(y)] \ge p_1, \\ &\text{if } d(x,y) \ge r_2 \text{ then } Pr[h(x) = h(y)] \le p_2, \end{aligned} \tag{2}$$

where $r_1$ and $r_2$ are distance thresholds, and $p_1, p_2$ are probabilities. In order to make a locality sensitive family useful, it requires $r_1 < r_2$ and $p_1 > p_2$. The hashing process for a vector $x \in \mathbb{R}^{1\times F}$ can be specified as: $h(x) = x \times \mathbf{A}$, where $\mathbf{A} \in \mathbb{R}^{F\times|\mathcal{H}|}$ is a Gaussian distributed matrix. In HaCore, we map the data into binary signatures, as this hash theme is locally sensitive in Euclidean space (Gionis et al. 1999). Thus the Hamming distance of the binary signatures could approximate the similarity between original samples.

## HaCore Methodology

### Method Overview

In this part, we introduce our HaCore coreset construction method. In general, we first use LSH to map different client's data into local bit signatures, and aggregate them into global signatures. To select the representative samples, we apply the $k$-medoids algorithm to the global signatures, and add the medoid-samples to the coreset. We reduce the computational cost of the Hamming space $k$-medoids algorithm by employing an inverted-index-based optimization method to narrow the nearest medoid search space and a bit-counting method to efficiently update the medoid of a cluster.

### Coreset Construction Procedure

To construct a coreset, our method follows these steps:

**Step 1: Allocate hash functions** (①)**.** In vertical federated learning (VFL), clients often have various data due to their different areas of expertise. Assigning the same number of hash functions to each client can be problematic: clients with more variance data can not fully capture their information,

while those with less informative data waste hash functions. To address this, we allocate hash functions by evaluating the variance of data samples on each client. Here, we define the variance $V_m$ of client $m$ as:

$$V_m = \frac{1}{N}\sum_{i=1}^{N}\|x_i^m - \frac{1}{N}\sum_{i=1}^{N}x_i^m\|_2, \tag{3}$$

the number of hash functions $h_m$ allocated to client $m$ is determined by its proportion to all clients combined:

$$h_m = \frac{V_m}{\sum_{i=1}^{n} V_i} \cdot |\mathcal{H}|, \tag{4}$$

where $\mathcal{H}$ is the family of hash functions. We assign more hash functions to the clients with more variable features. By doing this, we can better capture the similarity of original samples, and apply it to the binary scheme.

**Step 2: Local hashing (②).** After confirming the number of hash functions for each client, we perform local hashing. Specifically, we conduct the hashing process using a private Gaussian projection matrix $\mathbf{A}_m \in \mathbb{R}^{F_m \times h_m}$ generated by each client individually. We derive local signatures by taking the sign of the hashed results. For each data sample $x_i \in D$ , the local signature $s_m(x_i)$ on client $m$ is obtained as follows:

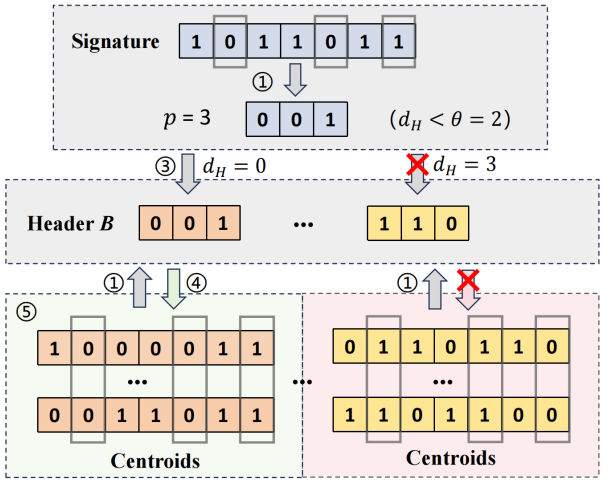
$$s_m(x_i) = \begin{cases} 1 & \text{if } (x_i^m \times \mathbf{A}_m)_j \geq 0 \\ 0 & \text{if } (x_i^m \times \mathbf{A}_m)_j < 0 \end{cases} j = 1, 2, \cdots, h_m. \tag{5}$$

Since the sign can only be 0 or 1, the signature can be compressed into a memory-efficient bit string.

**Step 3: Global signature construction (③).** We let all the clients jointly run a secure multi-party shuffling protocol (Movahedi, Saia, and Zamani 2015; Chase, Ghosh, and Poburinnaya 2020) on their local hash signatures before sending them to the aggregation server. The server concatenates the local signatures for the same sample into a global signature, which is then sent to the label owner for further processing: $s_i = \bigoplus_{m=1}^{M} s_m(x_i)$, where $s_i$ is the global signature for the $i^{\text{th}}$ sample and the $\bigoplus$ denotes concatenation.

**Step 4: $k$-medoids clustering (④).** After the label owner receives all the global signatures $S = \{s_i\}_{i=1}^{N}$, it then performs $k$-medoids algorithm to cluster these global signatures into multiple clusters. The $k$-medoids algorithm is similar to $k$-means, but with a key difference in how medoids are updated. Instead of using the mean of the data points, $k$-medoids selects the point within the cluster that has the smallest total distance to all other points as the new medoid. Given that the signatures are bit strings, Hamming distance is used as the distance metric. This updating method ensures that real samples represent their clusters. We perform $k$-medoids clustering based on data labels, clustering sample signatures with different labels into distinct groups.

Assuming there are $L$ kinds of labels and $k$ clusters in total, we cluster the signatures for each kind of label into $k/L$ clusters, yielding medoids $\{cent_j^l\}_{j=1}^{k/L}$ and their corresponding signature clusters $\{c_j^l\}_{j=1}^{k/L}$. Formally, for a signature $s$




① sample bits ② compare with headers ③ find qualified headers

④ go to inverted list ⑤ compare with centroids

Figure 2: An illustration of Hamming space $k$-medoids optimization by inverted index.

belonging to label $l$ ($l = 1, 2, \ldots, L$), we first identify its nearest medoid $\hat{c}$ and assign it to the corresponding cluster:

$$\hat{c} = \arg\min_{1<j<k/L} d_H(s, cent_j^l), \tag{6}$$

after finishing this process for all signatures, we update the medoids, as the items within each cluster have changed:

$$cent_j^l = \arg\min_{s \in c_j^l} \sum_{s' \in c_j^l} d_H(s, s'), \tag{7}$$

where $d_H(\cdot, \cdot)$ represents the Hamming distance function. This process allows us to group similar signatures under the same label together, and record the most representative ones.

**Step 5: Data selection (⑤).** After completing the $k$-medoids clustering, we gather all the medoids and add the corresponding data samples and labels to the coreset $D_c$. Specifically, we retrieve the data sample identifiers $[N_c]$ associated with the medoids. These identifiers are then sent back to the clients to form the coreset together: $D_c = \{(x_i \in \mathbb{R}^F, y_i \in \mathbb{N}) : i \in [N_c]\}$. The coreset size can be adjusted by users through manually setting the number of medoids.

## Efficiency Optimizations

In our framework, the most computationally intensive step is performing $k$-medoids clustering based on the Hamming distance of the signatures. It is essential to reduce computation costs while maintaining clustering quality. The $k$-medoids clustering process involves two main steps: ❶ find the nearest medoid for each sample and assign it to the corresponding cluster according to Equation 6, ❷ update a new medoid for each cluster, as outlined in Equation 7. We have proposed optimization methods for each of these two steps: for ❶, we construct an inverted index on the medoids to reduce unnecessary computations between samples and

medoids; for ❷, we utilize a bit-counting method to reduce the time complexity from $O(n^2)$ to $O(n)$.

**Optimization 1: Inverted-index on medoids (❶).** The core idea is to build an inverted index on the medoids to efficiently exclude distant medoids for each signature. Let $\mathcal{B}$ represents the set of all possible $p$-bit strings. Each element in $\mathcal{B}$ serves as a header in the inverted index. As illustrated in Figure 2, a header corresponds to a list of medoids, with each list containing medoids whose sampled bit strings match the header. For each medoid, specific bits are sampled from predetermined positions to create a substring of $p$ bits. Similarly, for a given signature $s_i$, we extract bits from the same positions as used for the medoids to form its $p$-bit substring $s_i^p$ (①). We then compare $s_i^p$ with all the headers in $\mathcal{B}$ (②). Based on that, only those headers that the Hamming distance between it and the signature is below a predetermined threshold $\theta$ are considered as the qualified ones (③), which means that their corresponding medoids are more likely to have the true nearest medoid than other headers'. The design rationale behind this is:

**Theorem 1.** *The similarity between sampled $p$ bits $a^p$ and $b^p$, derived from original $n$ bit strings $a$ and $b$ is an unbiased estimator of the true similarity between $a$ and $b$. $Sim(\cdot,\cdot)$ denotes the similarity of two bit strings.*

$$\mathbb{E}[Sim(a^p, b^p)] = Sim(a, b).$$

After comparing all the headers in $\mathcal{B}$, the signature goes to the qualified inverted list (green section in Figure 2) (④) to gather the medoids that belong to it, and compare with them to find the nearest medoid (⑤). This eliminates the need to evaluate those medoids that are unlikely to be the nearest.

**Optimization 2: Bit-counting in clusters (❷).** After assigning signatures to new clusters, the next step is to compute the medoid for each cluster. The medoid is the signature with the shortest total Hamming distance to all other signatures in the cluster. A naive approach calculates the Hamming distance between each pair in a loop and sums them, resulting in an inefficient $O(n^2)$ time complexity. Our optimization approach iterates through all signatures in the cluster to count the number of 0s and 1s at each bit position and maintains a count list $C^j = [num(0, j), num(1, j)]$ for each position $j$. For a given signature, if the bit value at position $j$ is 0, the total distance is increased by the count of 1s at that position; if the bit value is 1, the count of 0s is added to the total distance. This optimization reduces the time complexity to $O(n)$ as it requires only two passes through all the signatures. Our method is empowered by the following theorem:

**Theorem 2.** *The sum of distances from a $n$-bit hash code $a$ to all other $n$-bit hash codes $(b_1, b_2, \ldots, b_T)$ is equal to the sum of the number of bits in the hash code that differ from the corresponding bit positions in all other hash codes.*

$$\sum_{i=1}^{T} d_H(a, b_i) = \sum_{j=1}^{n} C^j[1 - b^j].$$

## Privacy Analysis

In this work, we assume the aggregation server and all participants follow an honest-but-curious model, a widely recognized threat model in federated learning research (Shokri and Shmatikov 2015; Bonawitz et al. 2017; Ma et al. 2020). First, each participant's hash functions remain private, ensuring data security if there is no server-client collusion. To further enhance privacy, the clients jointly run a secure multi-party shuffling protocol (Chase, Ghosh, and Poburinnaya 2020; Movahedi, Saia, and Zamani 2015) on local hash signatures before sending them to the server. This ensures that neither the clients nor the server can determine the exact ordering of the hash bits. During the coreset construction process, the transmitted data, apart from the hash bits (signatures), are the selected sample identifiers, which do not compromise privacy. The entire process is secure against a curious server. For stricter privacy requirements, the process can be encrypted using Homomorphic Encryption (Acar et al. 2018), although this would reduce efficiency.

| Dataset | SU | HI | CO | YP | HP |
|---|---|---|---|---|---|
| # samples | 1.0M | 1.0M | 0.6M | 0.5M | 0.5M |
| # features | 18 | 28 | 54 | 90 | 11 |
| # classes | 2 | 2 | 7 | / | / |

Table 1: Statistics of the datasets, first 3 for classification, last 2 for regression.

# Experimental Evaluation

## Experiment Settings

**Datasets and models.** Table 1 reports the statistics of the datasets used in our experiments. Among them, SU (Whiteson 2014b) and HI (Whiteson 2014a) are for binary classification, and we randomly select 1 million samples from their original datasets; CO (Blackard 1998) has seven distinct categories; YP (Bertin-Mahieux 2011) and HP (Sleem 2018) are for regression. For each dataset, we group the samples into a training set (70%), a validation set (10%), and a test set (20%). We use a multi-layer perceptron (MLP) and a logistic regression (LR) as the downstream models. The MLP has two hidden layers with a default dimension of 128. The machine learning model in VFL is inherently less complex (Khan, ten Thij, and Wilbik 2022; Jiang et al. 2022; Fu et al. 2022a) because it requires splitting the data and the model into several parts. For instance, it is uncommon for different organizations to hold different portions of images in real-world scenarios. In addition, complex networks like CNNs are rarely used due to the lack of reasonable approaches to perform convolution operations in VFL.

**Baselines.** We compare our proposed HaCore with the following baselines:

- Full: It uses all the training samples in the dataset to collaboratively train a VFL model.
- Random: It uniformly samples a subset from the original training set to train the VFL model.
- TreeCSS: It utilizes a clustering-based scheme to clusters the features locally on each participant and then merges the local clustering results to form the coreset.

| | Classification (ACC (%) ↑) | | | | | Regression (RMSE ↓) | | | |
|---|---|---|---|---|---|---|---|---|---|
| | SU | | HI | | CO | YP | | HP | |
| | LR | MLP | LR | MLP | MLP | LR | MLP | LR | MLP |
| Full | 76.93±0.12 | 79.53±0.16 | 68.98±0.11 | 73.56±0.23 | 84.57±0.21 | 9.52±0.03 | 8.98±0.07 | 0.09±0.03 | 0.08±0.01 |
| Random | 58.26±1.35 | 60.82±1.24 | 50.39±1.68 | 61.17±2.26 | 63.92±0.57 | 12.83±2.89 | 10.64±1.61 | 64.98±5.46 | 61.84±3.81 |
| TreeCSS | 75.92±0.21 | 78.45±0.15 | 67.02±0.18 | 71.01±0.14 | 83.69±0.19 | 9.59±0.13 | 9.12±0.11 | 0.31±0.02 | 0.25±0.06 |
| V-coreset | 75.28±0.12 | 78.89±0.09 | 66.62±0.15 | 70.49±0.07 | 82.45±0.15 | 9.54±0.06 | 9.12±0.13 | 0.11±0.03 | 0.10±0.05 |
| HaCore | **76.71±0.15** | **79.44±0.12** | **68.76±0.17** | **73.42±0.16** | **84.52±0.17** | **9.52±0.04** | **9.01±0.09** | **0.10±0.02** | **0.09±0.02** |

Table 2: Accuracy comparison for classification task and regression task with the coreset size is **2%** of the training set. RMSE (smaller is better) and Accuracy (higher is better). We highlight the best result of coreset construction method for each dataset.

| | SU | | HI | | CO | YP | | HP | |
|---|---|---|---|---|---|---|---|---|---|
| | LR | MLP | LR | MLP | MLP | LR | MLP | LR | MLP |
| Full($10^4$) | 3.69±0.14 | 4.22±0.15 | 3.98±0.06 | 4.63±0.13 | 2.79±0.28 | 2.89±0.16 | 3.44±0.11 | 2.21±0.12 | 2.68±0.05 |
| Random($10^2$) | 0.43±0.01 | 1.12±0.22 | 0.62±0.01 | 1.31±0.17 | 1.48±0.31 | 1.38±0.12 | 1.71±0.19 | 1.01±0.19 | 1.26±0.21 |
| TreeCSS($10^4$) | 1.79±0.21 | 2.01±0.12 | 3.18±0.18 | 3.88±0.23 | 1.89±0.21 | 1.41±0.12 | 1.68±0.18 | 1.78±0.21 | 1.99±0.10 |
| HaCore($10^2$) | **6.88±0.16** | **7.72±0.07** | **7.48±0.14** | **8.65±0.06** | **6.34±0.19** | **5.62±0.09** | **6.91±0.16** | **5.11±0.11** | **6.17±0.09** |
| **Speed up** | 26× | 26× | 43× | 45× | 30× | 25× | 24× | 35× | 32× |

Table 3: End-to-end training time comparison. The numbers are in seconds, and the magnitude of time duration is marked alongside the method. We highlight the best result of coreset construction method for each dataset.

Note that TreeCSS accelerates data alignment and coreset construction in VFL. We focus solely on coreset construction and the subsequent model training.

- V-coreset: It computes the orthonormal basis on each client and uses it for importance sampling to form the coreset. However, since V-coreset lacks a distributed implementation (Huang et al. 2022), our comparison is limited to model quality metrics.

**Evaluation protocol.** We launch a cluster with four clients and one label owner. The dataset is equally partitioned into four vertical portions, and each portion is held by one client. Meanwhile, the label owner has all the labels of the dataset. We choose Adam (Kingma and Ba 2014) as the optimization algorithm for both classification and regression tasks. We perform grid search to tune the learning rate within $\{1, 0.1, 0.01, 0.001\}$ across all datasets. We tune the batch size from 0.1% to 10% of the training data. We continue model training until the relative change in the loss is below a threshold (set as $1e-4$) for five epochs. For regression tasks, we use the Root-Mean-Square-Error (RMSE) to evaluate model performance, where smaller value indicates better accuracy. For classification tasks, we use classification accuracy, and higher value is better.

**Implementation.** We run our experiments on a cluster, where each machine has a Intel-i9 CPU and 24GB memory, and the machines are connected via 10GBps Ethernet. The machines communicate using gRPC with the proto3 library, and model training is implemented using PyTorch. Each machine serves as a client, and one machine serves as the server.

## Main results

Table 2 compares the model accuracy of HaCore and baselines. Table 3 reports the end-to-end training time consumption (coreset construction plus model training). Note that for each method, we set the default coreset size to 2% of the training data and the default client number to 4. The number of hash functions is fixed at 128. For the optimization of $k$-medoids, we randomly sample *9-bits* from the original signatures to form the headers of the inverted index.

**Model accuracy.** Table 2 shows that HaCore closely matches the performance of the Full model and is more effective than other coreset construction methods. The tiny performance gap between HaCore and Full in both classification and regression tasks indicates that HaCore provides good model accuracy. In contrast, Random exhibits significantly higher RMSE in regression tasks and lower accuracy in classification tasks, as it uniformly samples data without considering sample quality. Compared to other coreset construction methods, HaCore consistently outperforms them, with improvements of up to 2.41% in classification accuracy and 0.21 in RMSE score for regression.

**Time consumption.** Table 3 shows that the time consumption of HaCore is much lower than Full, and the time reduction can be as mush as 98%. Random has the shortest time consumption among the methods, as it simply selects samples randomly. In comparison with other coreset construction methods, HaCore consistently outperforms TreeCSS. It can accelerate TreeCSS by up to 45× on the HI dataset and by an average of 28× on other datasets.

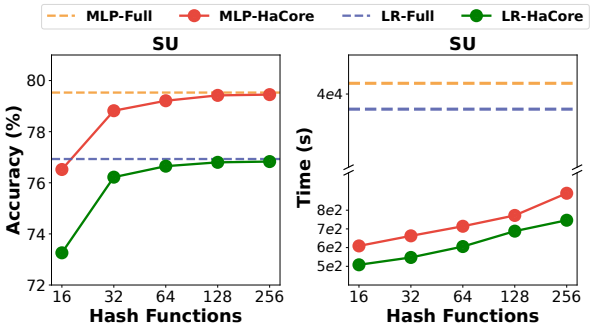



Figure 3: The influence of number of hash functions.

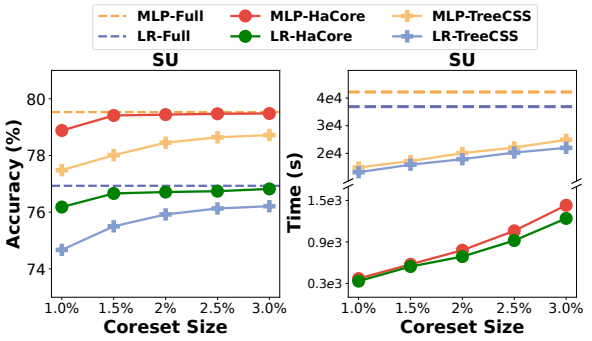



Figure 4: The influence of varying coreset size.

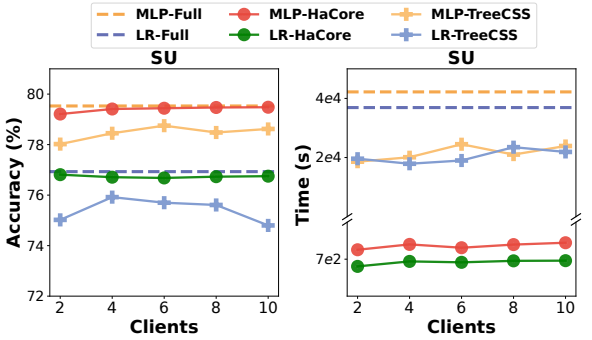



Figure 5: The influence of number of clients.

## Ablation Study and Micro Results

We select one representative dataset SU for further analysis.

**Number of hash functions.** We evaluate HaCore by varying the number of hash functions while keeping the coreset size constant. Figure 3 shows that more hash functions improve test accuracy by enhancing the representational capacity of sample signatures. However, this also increases the overall training time due to the extended duration required for more computation in $k$-medoids clustering.

**Number of coreset samples.** We also examine how HaCore and TreeCSS perform with different coreset sizes while keeping the number of hash functions constant. As shown in Figure 4, increasing the coreset size improves model accuracy, but also extends the end-to-end time due to the longer clustering time needed for more clusters and the increased training time from the larger coreset. HaCore keeps performing better than TreeCSS in both accuracy and efficiency.

| Method | SU | YP |
|---|---|---|
| | ACC (%) ↑ | RMSE ↓ |
| All | 79.46 | 9.02 |
| Average | 79.21 | 9.10 |
| Variance | 79.44 | 9.01 |

Table 4: Effect of variance-based hash function allocation under default experiment settings.

| Method | SU | | YP | |
|---|---|---|---|---|
| | ACC (%) ↑ | Time | RMSE ↓ | Time |
| w/o optimization | 79.45 | 3118s | 9.01 | 2967s |
| optimization | 79.44 | 772s | 9.01 | 691s |

Table 5: Effect of optimizations on end-to-end training time under default experiment settings.

**Number of clients.** We test the stability of HaCore and TreeCSS with varying client numbers. As shown in Figure 5, TreeCSS becomes highly unstable with changes in client numbers, impacting both accuracy and time efficiency. This instability arises because TreeCSS requires identical signatures across all clients. To keep the coreset size at 2% of the training set, we adjust the number of clusters per client, reducing cluster variety and compromising coreset quality. Unstable $k$-means runtimes also cause variations in end-to-end time. In contrast, the feature mapping of LSH in HaCore remains stable regardless of the number of clients.

**Effect of variance-based allocation.** We validate the effect of variance-based hash function allocation. As shown in Table 4, All denotes that hash functions are used for mapping all the features into global signatures in centralized scenarios. And Average means that hash functions are allocated averagely to each clients. The results indicate that, compared to Average, Variance can better retain the knowledge from variable clients and achieve better accuracy than Average.

**Effect of optimizations.** As shown in Table 5, in terms of accuracy, optimization has almost no impact. However, the optimization reduces the time required to just 23% of the original, resulting in a significant improvement in efficiency.

## Conclusions

In this paper, we propose HaCore as an efficient coreset construction method for VFL. This approach aims to reduce the coreset construction time while providing solid quality of the coreset samples. It first utilizes locality sensitive hashing to map the original features into Hamming space and selects the medoid samples after $k$-medoids clustering. We also build an inverted index and apply a bit-counting method to accelerate the clustering process. Extensive experiments show that HaCore significantly outperforms the baselines.

## Acknowledgments

This work was sponsored by National Science and Technology Major Project (2022ZD0116315), Key R&D Program of Hubei Province (2023BAB077, 2023BAB170), and National Natural Science Foundation of China (62472327, 62402011). This work was supported by Ant Group through CCF-Ant Research Fund (CCF-AFSG RF20240104, CCF-AFSG RF20230106). Chuang Hu and Jiawei Jiang are the corresponding authors.

## References

Acar, A.; Aksu, H.; Uluagac, A. S.; and Conti, M. 2018. A survey on homomorphic encryption schemes: Theory and implementation. *ACM Computing Surveys (Csur)*, 51(4): 1–35.

Andoni, A.; Indyk, P.; Nguyen, H. L.; and Razenshteyn, I. 2014. Beyond locality-sensitive hashing. In *Proceedings of the twenty-fifth annual ACM-SIAM symposium on Discrete algorithms*, 1018–1028. SIAM.

Bachem, O.; Lucic, M.; and Krause, A. 2017. Practical coreset constructions for machine learning. *arXiv preprint arXiv:1703.06476*.

Bertin-Mahieux, T. 2011. YearPredictionMSD Dataset. UCI Machine Learning Repository. DOI: https://doi.org/10.24432/C50K61.

Blackard, J. 1998. Covertype. UCI Machine Learning Repository. DOI: https://doi.org/10.24432/C50K5N.

Bonawitz, K.; Ivanov, V.; Kreuter, B.; Marcedone, A.; McMahan, H. B.; Patel, S.; Ramage, D.; Segal, A.; and Seth, K. 2017. Practical Secure Aggregation for Privacy-Preserving Machine Learning. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, 1175–1191.

Chase, M.; Ghosh, E.; and Poburinnaya, O. 2020. Secret-shared shuffle. In *Advances in Cryptology–ASIACRYPT 2020: 26th International Conference on the Theory and Application of Cryptology and Information Security, Daejeon, South Korea, December 7–11, 2020, Proceedings, Part III 26*, 342–372. Springer.

Cohen, M. B.; Musco, C.; and Musco, C. 2017. Input sparsity time low-rank approximation via ridge leverage score sampling. In *SODA*, 1758–1777. SIAM.

Drineas, P.; Mahoney, M. W.; and Muthukrishnan, S. 2006. Sampling algorithms for l 2 regression and applications. In *SODA*, 1127–1136.

Feldman, D.; and Langberg, M. 2011. A unified framework for approximating and clustering data. In *STOC*, 569–578.

Feldman, D.; Schmidt, M.; and Sohler, C. 2020. Turning big data into tiny data: Constant-size coresets for k-means, PCA, and projective clustering. *SIAM Journal on Computing*, 49(3): 601–657.

Feng, S. 2022. Vertical federated learning-based feature selection with non-overlapping sample utilization. *Expert Systems with Applications*, 208: 118097.

Fu, C.; Zhang, X.; Ji, S.; Chen, J.; Wu, J.; Guo, S.; Zhou, J.; Liu, A. X.; and Wang, T. 2022a. Label inference attacks against vertical federated learning. In *31st USENIX security symposium (USENIX Security 22)*, 1397–1414.

Fu, F.; Miao, X.; Jiang, J.; Xue, H.; and Cui, B. 2022b. Towards Communication-efficient Vertical Federated Learning Training via Cache-enabled Local Updates. *arXiv preprint arXiv:2207.14628*.

Gionis, A.; Indyk, P.; Motwani, R.; et al. 1999. Similarity search in high dimensions via hashing. In *Vldb*, volume 99, 518–529.

Huang, L.; Li, Z.; Sun, J.; and Zhao, H. 2022. Coresets for Vertical Federated Learning: Regularized Linear Regression and $K$-Means Clustering. *Advances in Neural Information Processing Systems*, 29566–29581.

Jafari, O.; Maurya, P.; Nagarkar, P.; Islam, K. M.; and Crushev, C. 2021. A survey on locality sensitive hashing algorithms and their applications. *arXiv preprint arXiv:2102.08942*.

Jiang, J.; Burkhalter, L.; Fu, F.; Ding, B.; Du, B.; Hithnawi, A.; Li, B.; and Zhang, C. 2022. Vf-PS: How to Select Important Participants in Vertical Federated Learning, Efficiently and Securely? *Advances in Neural Information Processing Systems*, 2088–2101.

Khan, A.; ten Thij, M.; and Wilbik, A. 2022. Communication-efficient vertical federated learning. *Algorithms*, 15(8): 273.

Kingma, D. P.; and Ba, J. 2014. Adam: A Method for Stochastic Optimization. *arXiv preprint arXiv:1412.6980*.

Konečnỳ, J.; McMahan, H. B.; Yu, F. X.; Richtárik, P.; Suresh, A. T.; and Bacon, D. 2016. Federated Learning: Strategies for Improving Communication Efficiency. *arXiv preprint arXiv:1610.05492*.

Li, A.; Peng, H.; Zhang, L.; Huang, J.; Guo, Q.; Yu, H.; and Liu, Y. 2023. FedSDG-FS: Efficient and Secure Feature Selection for Vertical Federated Learning. *arXiv preprint arXiv:2302.10417*.

Li, T.; Sahu, A. K.; Talwalkar, A.; and Smith, V. 2020. Federated Learning: Challenges, Methods, and Future Directions. *IEEE Signal Processing Magazine*, 37(3): 50–60.

Liu, Y.; Kang, Y.; Zhang, X.; Li, L.; Cheng, Y.; Chen, T.; Hong, M.; and Yang, Q. 2019. A Communication Efficient Collaborative Learning Framework for Distributed Features. *arXiv preprint arXiv:1912.11187*.

Liu, Y.; Kang, Y.; Zou, T.; Pu, Y.; He, Y.; Ye, X.; Ouyang, Y.; Zhang, Y.-Q.; and Yang, Q. 2024. Vertical federated learning: Concepts, advances, and challenges. *IEEE Transactions on Knowledge and Data Engineering*.

Lucic, M.; Faulkner, M.; Krause, A.; and Feldman, D. 2017. Training gaussian mixture models at scale via coresets. *The Journal of Machine Learning Research*, 18(1): 5885–5909.

Ma, C.; Li, J.; Ding, M.; Yang, H. H.; Shu, F.; Quek, T. Q.; and Poor, H. V. 2020. On Safeguarding Privacy and Security in the Framework of Federated Learning. *IEEE Network*, 34(4): 242–248.

Movahedi, M.; Saia, J.; and Zamani, M. 2015. Shuffle to baffle: Towards scalable protocols for secure multi-party shuffling. In *2015 IEEE 35th International Conference on Distributed Computing Systems*, 800–801. IEEE.

Park, H.-S.; and Jun, C.-H. 2009. A simple and fast algorithm for K-medoids clustering. *Expert systems with applications*, 36(2): 3336–3341.

Schubert, E.; and Rousseeuw, P. J. 2019. Faster k-medoids clustering: improving the PAM, CLARA, and CLARANS algorithms. In *Similarity Search and Applications: 12th International Conference, SISAP 2019, Newark, NJ, USA, October 2–4, 2019, Proceedings 12*, 171–187. Springer.

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

Shokri, R.; and Shmatikov, V. 2015. Privacy-Preserving Deep Learning. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, 1310–1321.

Sleem, A. 2018. HousePricing. Kaggle Big Data Competition Platform. DOI: https://www.kaggle.com/datasets/greenwing1985/housepricing/data.

Vepakomma, P.; Gupta, O.; Swedish, T.; and Raskar, R. 2018. Split learning for health: Distributed deep learning without sharing raw patient data. *arXiv preprint arXiv:1812.00564*.

Wei, S.; Tong, Y.; Zhou, Z.; and Song, T. 2020. Efficient and Fair Data Valuation for Horizontal Federated Learning. *Federated Learning: Privacy and Incentive*, 139–152.

Whiteson, D. 2014a. HIGGS. UCI Machine Learning Repository. DOI: https://doi.org/10.24432/C5V312.

Whiteson, D. 2014b. SUSY. UCI Machine Learning Repository. DOI: https://doi.org/10.24432/C54606.

Yang, Q.; Liu, Y.; Chen, T.; and Tong, Y. 2019. Federated Machine Learning: Concept and Applications. *ACM Transactions on Intelligent Systems and Technology (TIST)*, 10(2): 1–19.

Zhang, C.; Xie, Y.; Bai, H.; Yu, B.; Li, W.; and Gao, Y. 2021. A survey on federated learning. *Knowledge-Based Systems*, 216: 106775.

Zhang, Q.; Yan, X.; Ding, Y.; Xu, Q.; Hu, C.; Zhou, X.; and Jiang, J. 2024. TreeCSS: An Efficient Framework for Vertical Federated Learning. In *International Conference on Database Systems for Advanced Applications*, 425–441. Springer.

Zhu, H.; Xu, J.; Liu, S.; and Jin, Y. 2021. Federated learning on non-IID data: A survey. *Neurocomputing*, 465: 371–390.