Datamap-Driven Tabular Coreset Selection for Classifier Training

AVIV HADAR, Tel Aviv University
TOVA MILO, Tel Aviv University
KATHY RAZMADZE, Tel Aviv University

In the era of data-driven decision-making, efficient machine learning model training is crucial. We present a novel algorithm for constructing *tabular data* coresets using datamaps created for Gradient Boosting Decision Trees models. The resulting coresets, computed within minutes, consistently outperform other baselines and match or exceed the performance of models trained on the entire dataset. Additionally, a training enhancement method leveraging datamap insights during the inference phase improves performance with mathematical guarantees, given a defined property holds. An explainability layer and tools for coreset size optimization further enhance the efficiency of training tabular machine learning models.

1 INTRODUCTION

In data science, successful model training is essential for decision-making and predictive analytics. This involves handling extensive training data and carefully selecting appropriate machine learning (ML) models while optimizing their configurations. However, finding the most suitable model and optimal configuration can be time-consuming due to numerous training iterations. To tackle this issue, various algorithms have been introduced to generate a condensed subset of data, referred as a *coreset*. Initially developed for clustering tabular data [25], and found extensive applications in recent computer vision tasks (e.g., [38]). However, the application of coresets in the context of tabular data remains relatively limited, and mainly focused on coresets that are tailored to classical ML algorithms [30, 37, 50, 53]. Although these works provide guarantees for their usage within the designated classic models, they have limited coverage for the advanced models commonly used in tabular data domain, such as XGBoost [17] or TabNet [12].

Goal. Focusing on classification models, our goal is to create a tabular coreset that achieves high performance on advanced ML models, and could take as an input any type of tabular data (numeric, categorical, etc.). As opposed to some of the previous work, the selected coreset is of a size that the user has provided, and the coreset remains relevant after the addition of new features, which happen frequently during the feature engineering process.

Intuition Behind Our Solution. Our solution draws inspiration from decision trees based ML classification algorithms, where the leaves serve as sets that, once the tree is constructed, contain all the examples the model was trained on. During inference, these leaves or the union of leaves (in case of multiple trees) determine the classification of unseen examples, assigning the same label to all examples within one leaf. Our algorithm capitalizes on the observation that leaves containing many examples, all belonging to the same label, can be viewed as forming equivalence classes. As a result, the algorithms can effectively classify similar instances without the need to train on all these examples. Leveraging this insight, we can select a coreset comprising only a small subset of these examples while excluding the others. However, identifying these *easy to learn* and *hard to learn* regions is non-trivial due to potential variations in splits by different decision trees and the impact of different sampling strategies, leading to diverse leaf nodes and varying results.

Proposed Solution. Our approach addresses the concern by introducing an algorithm for creating a tabular datamap, originally designed for the textual domain [47]. The datamap identifies regions in the data that are easy or hard to learn

Authors' addresses: Aviv Hadar, Tel Aviv University, avivhadar@mail.tau.ac.il; Tova Milo, Tel Aviv University, milo@post.tau.ac.il; Kathy Razmadze, Tel Aviv University, kathyr@mail.tau.ac.il.

during model training. We then develop an algorithm that utilizes the datamap to select a coreset, encompassing hard-to-learn regions and small representatives of easy-to-learn ones. To the best of our knowledge, this is the first application of datamaps for this purpose. This innovative strategy results in a coreset that competes in terms of performance with even complex models trained on the entire dataset. Our efficient implementation, CoreTab, constructs coresets within minutes even for sizable datasets, outperforms competitors in speed and achieves better quality results, up to a 30% increase in some cases compared to the best performing baseline. Additionally, we offer an explainability layer to help users understand the inclusion or condensation of specific data sections in the coreset.

Example 1.1. Consider our approach applied to the *BankLoan* dataset. Here, the goal is to train an ML model for classifying loan approval (*True*) or rejection (*False*) based on various attributes. Imagine a decision tree leaf containing 10% of the data, where all examples share the label *False*. Examining the decision tree path leading to this leaf reveals conditions $bank\_credit \leq 200$ and $Bankruptcies \geq 1$, defining a specific region within the dataset. This region is homogeneous, making predictions easy due to a consistent label. Sampling a small subset from this *easy-to-learn* region is likely highly representative. Conversely, challenging regions for the model require more data. Thus, we create a datamap partitioning the data space into regions of varying complexity based on label homogeneity and the model's ease or difficulty in making predictions within each region.

Training Enhancement. Leveraging the valuable insights encoded within the generated datamap, we introduce an innovative approach to enhance the ML model training process. Our aim is to streamline this intricate and resource-intensive process, particularly beneficial when repeated model training is necessary [26]. We propose a method that significantly improves the performance of models trained on the coreset, often matching or surpassing the performance of models trained on the entire dataset. Moreover, this method provides insights into the trade-offs in results, supported by mathematical guarantees (Section 5), contingent upon a defined property of the data and model. These benefits extend to hyperparameter tuning and cross-validation, both requiring numerous training iterations [56]. To achieve this, we leverage the datamap, a byproduct of the CoreTab algorithm, during the inference phase of a model trained on the coreset. As the datamap algorithm exclusively uses the training set, we achieve a performance boost without data leakage concerns. Our system also includes an investigation tool for users to explore coreset size and accuracy trade-offs, facilitating the selection of the most appropriate coreset size based on desired quality preservation.

Contributions. This work presents several contributions:

(1) Introduction of an *advanced algorithm tailored for creating a tabular datamap*, a crucial element in our coreset generation process, specifically designed for Gradient Boosting Decision Trees (GBDT) models.
(2) Proposal of a *novel algorithm and problem formulation for creating coresets using a datamap* for tabular data, addressing the challenge of efficient coreset creation. This includes an explainability layer to understand patterns within the coreset compared to the full dataset.
(3) Unique approach for *training enhancement* by leveraging the datamap during the inference phase of a model trained over the coreset, supported by theoretical guarantees for model performance. It also includes an explainability layer for optimal coreset size determination.
(4) Provision of *comprehensive experimental results* demonstrating the high performance of CoreTab across various models and datasets, along with thorough comparisons with various baseline methods for tabular coreset selection. Additionally, showcasing the benefits of our approach for training enhancement, revealing significantly reduced training times and high model performance.

2 RELATED WORK

Our work relates to two research areas: (1) data sampling and summarization, and (2) coreset and instance selection techniques. In this section, we highlight distinctions from existing methods and present comparison results in Section 6.

Row Sampling, Data Summariztion and AQP. Row sampling is widely used in various domains, for expediting query results [10, 13] and data visualization for reducing data points [41]. Greedy algorithms are applied for query result diversification [35, 52]. In contrast, our ML model optimization focuses on selecting rows based on labels and their relevance. In AutoML, [32] uses genetic algorithms for a compact and representative data subset, emphasizing general data characteristics. Unlike our label-centric approach, it aims for versatility (see Table 1 for performance comparison).

Although AQP methods such as BAQ [33] and VerdictDB [42] are designed to approximate aggregate query results in large databases, they are not well-suited for coreset creation. This is primarily because they rely on predefined query workloads, which are typically unavailable in ML training, involve extensive preprocessing, and do not optimize based on label-specific importance. Other works, like [36], employ various sampling techniques on data batches due to the large size of the dataset, which resembles sampling strategies used in smaller datasets common in ML training. Another line of work uses Generative Adversarial Networks (GANs [24]) to create small portions of synthetic data that replicate the original dataset, allowing queries on the smaller subset instead of the entire database. We adapted a state-of-the-art method from this line of work, VAE [49], for coreset selection, detailed in Section 6. As shown in Table 1, VAE's performance was inferior to that of CoreTab, underscoring the necessity of specialized methods for coreset creation in ML tasks.

Fundaments of Coresets and Instance Selection. Initially called Instance Selection [34], these techniques involve selecting, generating, and transforming instances to enhance data mining algorithms. Coresets, also known as representative subsets or summarization methods, efficiently approximate complex datasets while retaining crucial structural insights. Works like [25] for clustering tabular data and [22] for statistical mixture models introduced coresets to reduce data size. Notably, these works primarily focus on unsupervised settings, preserving data characteristics without labels.

Recent Coresets Work. CRAIG [38], a foundational technique for selecting supervised ML model coresets that closely approximate the full gradient, was initially introduced for computer vision applications. Subsequent research has expanded on this foundation, predominantly targeting computer vision tasks [14, 39, 43]. Although these methods are not directly applicable to tabular data, we adapted the current state-of-the-art coreset selection method from computer vision [55] for the tabular domain and compared it to CoreTab in Section 6. Our results demonstrate that CoreTab outperforms this adapted model, highlighting its superior effectiveness for tabular data.

Recent works for tabular data have focused on coreset selection for clustering tasks [15, 18, 28, 51]. These clustering-centric approaches may not capture patterns necessary for classification tasks. Additionally, methods for tabular classification coresets [30, 37, 50], primarily tailored to basic ML algorithms, were comprehensively evaluated in our research, highlighting the superior efficacy of our coreset generation approach (Section 6).

Previous work on coresets for relational databases [16, 53] has predominantly employed the gradient change principle. This method constructs coresets by selecting data points that have the most significant impact on the gradient during optimization, thereby preserving the essential characteristics of the original dataset that are crucial for model training, similar to the CRAIG algorithm [38]. In contrast, our research focuses on creating subsets using datamaps derived from Gradient Boosting Decision Tree (GBDT) models. These datamap-informed coresets are highly versatile and applicable to a wide range of complex ML models, making them well-suited for various tabular data scenarios (Section 6). We chose not to include a comparison with the approximation of the CRAIG algorithm designed for multiple tables, as presented in

[53]. The authors of that work have noted that even when applied to a single table, their algorithm is still an approximation, and they suggest that the exact CRAIG algorithm would offer superior performance in single-table scenarios.

Active Learning Positioning. Active Learning (AL) focuses on selecting the most informative data points to improve model performance with minimal labeled data. According to a recent survey [48], AL methods are categorized into Meta Active Learning, Representation-Based Methods, Information-Based Methods, and Random Selection. For example, Meta AL, such as [21], uses reinforcement learning for streaming data, while Representation-Based methods leverage data clustering [29] to select representative samples. Unlike traditional AL, which often does not use labels during selection, our approach in CoreTab assumes labeled data is available and optimizes coreset selectionn according to them. In this paper, we position CoreTab within the AL space and compare it to state-of-the-art AL methods [45, 46] from the Information-Based methods, specifically, *Uncertainty Sampling*, demonstrating CoreTab's superior performance across several datasets.

3 PRELIMINARY

In this section, we establish the groundwork for our upcoming algorithm, both for tabular coresets and tabular datamaps, and discuss essential components of Gradient Boosting Decision Tree (GBDT) algorithms, pivotal to creating datamaps.

Problem Formulation. In line with standard ML conventions, we consider a dataset D with $[R_1, \dots R_N]$ rows and $[C1, \dots C_M]$ columns drawn from distribution $\mathcal{D}$. A coreset is a subset of rows of D, projected over all columns [25].

Definition 3.1 (Tabular Coreset). For a dataset D with row-indices R and column indices C, a tabular coreset of size $n \times m$ is denoted as d and is defined as $D[r, C]$ for any $r \in [R]^n$. Here, $[R]^n$ represents the set of all n-subsets of R, i.e., $[R]^n = \{R' | (R' \subseteq R) \wedge (|R'| = n)\}$.

We focus on binary classification ML models with P and N as the positive and negative classes, respectively. In a typical scenario, a data scientist trains an ML model M using configuration $conf$ to predict labels of the dataset D, defined in the training as column y. We denote this model as $M(D, y, conf)$. Let $Rec(M(D, y, conf))$ and $Prec(M(D, y, conf))$ represent the recall and precision of the trained model, with $Acc(M(D, y, conf))$ being the classification metric to optimize (e.g., recall, precision, F1-score). Next, we introduce two methods for optimizing coreset creation.

Definition 3.2 (Coreset Creation Optimizations). The coreset creation could be optimized based on the following:

- **[Opt_per] Optimization based on models performance:** Given Δ_{Recall}, $\Delta_{Precision}$, a thresholds for performance guarantees, derive the minimal size n and a coreset d of size n, s.t. $n << N$, s.t. $\forall M$: $Rec(M(d, y, conf)) - \Delta_{Recall} \geq Rec(M(D, y, conf))$
  $Prec(M(d, y, conf)) - \Delta_{Precision} \geq Prec(M(D, y, conf))$
- **[Opt_size] Optimization based on coreset size:** Given n, the coreset's d size, find the coreset $d^* \in [R]^n$ s.t. $Acc(M(d^*, y, conf)) \geq Acc(M(d, y, conf)), \forall d \in [R]^n$.

Goal. The primary objective of our algorithm, *CoreTab*, consistent with the goals of other coreset algorithms, is to select a tabular coreset that significantly reduces training computation times for a given ML model, without sacrificing, and potentially even enhancing, the model's original performance. Our algorithm optimizes the coreset creation process based on the two methods previously described, allowing for customization according to user preferences. As demonstrated in our experiments in Section 6, training on the coreset, as opposed to the entire dataset, leads to substantially faster training times.

Tabular Datamaps. The Datamap concept, initially introduced in Natural Language Processing (NLP) [47], provides a distinctive view of how ML models perceive and adapt to data during training. It maps and diagnose datasets as they evolve, offering insights into the impact of different data samples on the learning process. In NLP, it creates a map with regions, each representing a set of words and indicating the complexity of learning their representation by the model. Adapting the Datamap concept to tabular data necessitates a shift from neural network-centric gradient changes to delineating regions based on data characteristics (columns values). Unlike NLP models, tabular data includes labels for each row, crucial for model construction. The goal is to capture groups perceived as similar or equivalent by ML models, emphasizing features essential for accurate label-based segregation. Each row is assigned to a region based on similarity within crucial features. We denote the number of examples in a region as its *size*.

Definition 3.3 (Group Homogeneity). Given a threshold value ψ, denote g a group of tuples each labeled by either N or P. We define g_n and g_p, as the set of tuples with the label N and P, respectively. A group is considered homogeneous concerning the label column if either of the following conditions is met: The proportion of tuples with label $N(P)$ in the group, denoted as $\frac{|g_n|}{|g|}$ ($\frac{|g_p|}{|g|}$), is greater than or equal to ψ.

In summary, the devised datamap divides the original dataset into smaller groups, each characterized by the resemblance of several attributes considered meaningful for distinguishing data points based on their labels. It is crucial to emphasize that not all regions have substantial sizes, and not all regions exhibit homogeneity according to predefined thresholds. Adapting the terminology from the original datamap to our case, we refer to regions as *easy to learn* if they are both homogeneous and large based on two thresholds (one for homogeneity and one for size). Conversely, non-homogeneous groups are labeled as *hard-to-learn* since the ML model struggles to segregate the data points within these regions into homogeneous groups. Lastly, there are other regions, specifically small homogeneous ones, designated as *ambiguous* to the model. This implies that the model succeeded in separating the data points into homogeneous regions within these areas, but the challenge was mitigated by their small size. The larger the groups, the more closely they align with the *easy-to-learn* regions. For brevity, the regions referred as *easy*, *hard*, *amb*. The subsequent section elucidates the primary reasons why these traits define the level of data complexity for the model, with further emphasis in Algorithm 1. Formally,

Definition 3.4 (Tabular Datamap). We define a Tabular Datamap dm with k non-overlapping regions as $dm = f_1, \ldots, f_k$, $[r_{i_1}, \ldots, r_{i_j}] \in f_i$, where $1 \leq j \leq N$, and $f_i \cap f_j = \emptyset$ for all $i \neq j \in [1, k]$. Each region f_i is characterized by a set of rules that define its boundaries. Each rule consists of an attribute (from the data space $\mathcal{D}$) and a valid value range for that attribute. Formally, $f_i = \{(c_{i_1}, val_{i_{1_1}}, val_{i_{1_2}}), \ldots, (c_{i_j}, val_{i_{j_1}}, val_{i_{j_2}})\}$. Each data entry $r_p \in \mathcal{D}$ could be assigned to exactly one region based on the region's defined boundaries. This assignment occurs when the attribute values of the entry satisfy the rules of the regions. Formally, $r_p \in f_j \rightarrow val_{j_{b_1}} \leq r_p[c_{j_b}] \leq val_{j_{b_2}}, \forall (c_{j_b}, val_{j_{b_1}}, val_{j_{b_2}}) \in f_j$. The regions' types defined using homogeneity (defined in Sec. 4), and the size of the region (denoted $|f_i|$) with threshold τ:

$$easy\_to\_learn = \{f_i | (homogenous(f_i, \psi) = True) \& (|f_i| > \tau)\}$$

$$hard\_to\_learn = \{f_i | (homogenous(f_i, \psi) = False)\}$$

$$ambiguous\_to\_learn = \{f_i | (homogenous(f_i, \psi) = True) \& (|f_i| \leq \tau)\}$$

Gradient Boosting Decision Trees Utilization. We now explain how *Gradient Boosting Decision Tree* (GBDT) principles are used to construct tabular datamaps and facilitate coreset selection (as detailed in Section 4). GBDT is a powerful ensemble learning technique, first introduced in [23], and has been widely adopted in algorithms like XGBoost [17]. It iteratively builds a series of decision trees, forming additive regression models by fitting a parameterized function (the base learner) to minimize "pseudo"-residuals derived from a specific loss function at each step. By aggregating the

predictions from these trees, the model progressively enhances its predictive accuracy. While the formal algorithm is omitted here for brevity, we focus on how GBDT's initial weak learners (trees generated in the early phases) are utilized to create datamaps with essential attributes. By limiting the number of trees, we encourage the formation of distinct groups characterized by properties significant to at least one weak learner.

Our approach takes advantage of the unique characteristics of gradient boosting trees. The process begins with a shallow initial tree, whose predictions guide the construction of subsequent trees by calculating error gradients. Each tree is divided into leaves, which represent clusters of data points with similar attributes, often related to the label columns. These clusters can be thought of as regions, each defined by a set of rules—boundaries set by the decision tree that created them. Some clusters are completely homogeneous, where the model makes no errors on these samples, while others contain multiple labels, posing challenges. In the later stages, non-homogeneous clusters are refined to reduce error rates. Regions that remain non-homogeneous are difficult for the algorithm to learn and are thus included in the coreset.

4 CORESET AND DATAMAP ALGORITHMS

This section explores *CoreTab*, a system designed for constructing a coreset from the original data. Here, we focus on coreset creation based on user-defined size optimization (opt_size), while the subsequent section discusses coreset creation based on performance optimization (opt_per). We start by elucidating the algorithm (Algorithm 1) responsible for crafting a datamap, a pivotal element in the coreset selection process. Next, we delve into the algorithm (Algorithm 2) generating tabular coresets, leveraging the datamap as a foundational component, focusing on the *hard* regions.

4.1 Tabular Datamap Creation

We delve into the core stages of tabular datamap generation, fully detailed in Algorithm 1. This algorithm adopts an *Optimizing Based on Labels* approach, giving priority to constructing the datamap primarily guided by data point labels. In practical scenarios, a weak learner, typically represented by a decision tree, may fail to accurately classify some data points, resulting in mixed-label leaves that eventually form the examined regions. To mitigate this, a threshold is set for the required homogeneity of a region concerning the label, as formally defined in Section 3. Note that, In Section 3, model M refers to any machine learning (ML) model that is trained on a coreset. The purpose of using this model is to evaluate its performance on a test set and compare it to the performance of the same model trained on the full dataset. This model is different from the ML model employed within our Datamap creation algorithm, defined below.

Initialization. The algorithm's inputs are the Dataset D, the number of trees for GBDT algorithm ($tnum$), the threshold for the size of the considered regions (τ), and the threshold for homogeneous regions (ψ). As explained in Section 6, we have carefully chosen the default values for those parameters, that work best on a wide variety of datasets and tasks. However, the user could change them according to his needs. Then, the GBDT algorithm is run for the given number of trees, and store the resulted trees in $trees$ parameter. Another part of the initialization is creation of two sets of sets. The first, denoted as $datamap$ is initialized as an empty set, and will eventually contain the datamap of the GBDT algorithm. The second set of sets, denoted as $data\_groups$, contains initially a set of all the data points in the given dataset and will be used for storing the intermediate calculated groups to be added later to the datamap, upon reaching certain criterion.

Leveraging Gradient Boosting Decision Trees. We utilize each tree generated in the initial phase of the Gradient Boosting Decision Trees (GBDT) algorithm to refine our data groups. For each tree, we store its leaves in the $leaves$ parameter, where each leaf represents a set of attributes (rules) that led to its creation by the decision tree, along with the data points contained within that leaf. We also initialize an empty set, new_data_groups, for each tree, which will

Algorithm 1 Creation of Datamap

Input: Original Dataset D, $tnum = 30$, $\tau = 5$, $\psi = 1$
Output: $datamap$
initialization: $trees = GBDT(data = D, \#trees = tnum)$
$datamap = [\,]$
$data\_groups = [[r_1, ..., r_N]]$
for every tree $t \in [t_1, t_{tnum}]$ **do**
$leaves = leaves(t)$
$new\_data\_groups = [\,]$
for every $group$ in the $data\_groups$ **do**
for every $leaf$ in $leaves$ **do**
if $leaf \cap group \neq \emptyset$ **then**
add $leaf \cap group$ to new_data_groups
end if
end for
end for
$data\_groups = new\_data\_groups$
for $group \in data\_groups$ **do**
if $(|group| \leq \tau) | (homogeneous(group, \psi) = True)$ **then**
add $group$ to $datamap$
remove $group$ from $data\_groups$
end if
end for
end for
add $data\_groups$ to $datamap$
return $datamap$

Algorithm 2 CoreTab Algorithm - Opt_size

Input: Training Set S, coreset size n, $tnum = 30$, $\tau = 5$, $\psi = 1$, $samp\_ratio = 0.03$
Output: $d = [r_{i_1}, r_{i_2}, ..., r_{i_n}]$ coreset of size n
$coreset = [\,]$
$datamap = datamap\_creation(S, tnum, \tau, \psi)$
for $region \in datamap$ **do**
if $homogeneous(region, \psi) = False$ **then**
$coreset.append(region \cap S)$
end if
end for
if sizeof(coreset) $\geq$ n **then return** $coreset$
end if
$easy\_to\_learn\_candidates = [\,]$
for $region \in datamap$ **do**
if $homogeneous(region, \psi) = True$ **then**
$easy\_to\_learn\_candidates.append(region \cap S)$
end if
end for
for $region \in sort\_by\_size\_desc(easy\_to\_learn\_candidates)$ **do**
$ratio\_for\_region = \min[samp\_ratio, \frac{n - |coreset|}{|region|}]$
$coreset.append(sample(region, ratio\_for\_region))$
remove $region$ from $easy\_to\_learn\_candidates$
if $|coreset| \geq n$ **then return** $coreset$
end if
if $|easy\_to\_learn\_candidates| + |coreset| \leq n$ **then**
$Break$
end if
end for
add $easy\_to\_learn\_candidates$ to $coreset$
return $coreset$

hold the newly formed groups, as described below. For each existing group in *data_groups*, we check whether a leaf in *leaves* contains any of the data points from that group. If such a leaf is found, we create a new group that includes these common data points, and we add it to *new_data_groups*. The rules for this new group are a refinement of the previous rules, merged with the rules of the corresponding leaf. By iterating through all groups in relation to the current tree's

leaves, we form a new collection of data groups. These groups vary in their homogeneity concerning the label, size, and characteristics. This updated collection becomes the new $data\_groups$. Finally, we review the groups within $data\_groups$. If a group is either too small (below a defined threshold) or homogeneous, we add it to the $datamap$ and remove it from $data\_groups$, ensuring that the $datamap$ is composed of the most relevant and refined groups.

The GBDT model used within the datamaps creation algorithm (a critical component of the CoreTab algorithm) is only partially trained on the full dataset. This partial training, inspired by the datamap methodology [47], aids in identifying the significance of each data point in the learning process. However, this step is not the ultimate objective of CoreTab; rather, it is instrumental in constructing a datamap that enables the CoreTab algorithm to select a smaller, representative coreset. The final ML model, whether it be GBDT or another ML model, is then trained on this coreset.

Single Decision Tree Datamap Creation. To further accelerate our algorithm, we propose a streamlined approach for datamap creation using a single decision tree. Instead of iteratively refining groups formed from the intersection of leaves as additional trees are built, we train a single decision tree without constrains. The leaves of this tree are then used to define the regions. Although these regions are not as refined as those produced by the GBDT algorithm, they effectively capture areas that are challenging for a single decision tree to learn, providing a faster datamap generation process. As demonstrated in Section 6, this method reduces runtime by up to 75%, with only a minor impact on the quality of the generated coreset.

4.2 Coreset Creation

Following the introduction of the datamap creation algorithm we present CoreTab (Algorithm 2), the algorithm responsible for selecting a coreset based on the user's preferred coreset size.

Initialization. The algorithm begins by initializing an empty set for the coreset entity and executing Algorithm 1, which is the algorithm for GBDT datamap creation (explained in Section 4.1). Initially, all *hard* regions in the $datamap$ are added to the $coreset$. These regions are defined as non-homogeneous regions formed in the datamap. If the required coreset size is reached, the algorithm stops and outputs the coreset. If the required size is not yet reached, the algorithm proceeds to add samples from the easy regions, starting with the largest *easy* regions. The set of candidates for *easy* regions is constructed initially by adding all homogeneous regions.

Candidates Consideration. An iterative process is then employed to determine the final composition of the coreset. The algorithm examines each region in the *easy* candidates, starting from the largest. For each region, the ratio at which this region will be sampled is defined as the minimum between the given $sample\_ratio$ and the portion of the remaining space in the coreset needed to hold a sample of this region. The region is then sampled according to the defined sample ratio and added to the coreset, while being removed from the *easy* candidates. Taking a sample from a region enables the future ML model, trained solely on the coreset, to gain insight into this region, ensuring coverage even if not fully represented. This loop terminates either when the requested coreset size is reached or when all regions from the *easy* candidates are sampled. Because the candidates are sampled by the order of their size, we first take the *easy* regions and only then we get to the *amb* regions, if the current corest size allows the addition of elements. This strategy (explained in more details in Section 3) is based on the observation that smaller groups are often harder for the algorithm to distinguish from examples of the other class, thus providing more valuable information for the training process.

Performance and Explainability. Section 6 showcases experimental results demonstrating the effectiveness of coresets generated by CoreTab. These coresets consistently outperform those created by other baseline methods across various

ML models, achieving performance levels that closely match models trained on the full dataset. Additionally, CoreTab provides a valuable explainability feature by linking the coreset to the datamap, revealing crucial patterns, including those omitted from the coreset (Section 5.5). This capability allows users to assess the coreset's suitability for their specific tasks and facilitates additional functionalities such as bias detection. Furthermore, Section 5 presents an algorithm for coreset creation that optimizes based on a user-defined quality metric, utilizing an enhanced training method supported by mathematical guarantees.

5 ENHANCING TRAINING FOR IMPROVED MODEL PERFORMANCE

Following coreset creation and ML model training, the inference phase involves predicting labels for unseen data. While our coreset-trained models achieve performance comparable to models trained on the full dataset (see Section 6), we propose a novel enhancement method. This approach extends mathematical guarantees to multiple models with a specific property, overcoming the traditional limitation of guarantees to a single model.

Training Enhancement. Users often experiment with various models before identifying the most effective one, making our method highly relevant for practical applications. To further enhance training outcomes, we introduce an alternative to conventional inference by utilizing the datamap. When a new data entry arrives, we check if it falls within a region of the datamap by evaluating the set of rules (boundaries) that define each region. For every region in the datamap, we maintain information about whether it is classified as easy, hard, or ambiguous to learn, along with the label most commonly associated with that region. If a new entry falls within an *easy to learn* region—characterized by label homogeneity and not represented in the coreset—we predict the label associated with that region, thereby augmenting the model's output. For entries outside these regions, we adhere to the standard inference approach based solely on the coreset-trained model. Efficient implementation is key, especially when managing a large number of groups.[1]

In the remaining part of this section, we outline theoretical guarantees for our algorithm in the training enhancement context, first defining two types of errors when an ML model is solely trained on the coreset during inference.[2] Lastly, we introduce an additional algorithm to generate a coreset based on user-specified required performance, indicated by a quality metric. The promising experimental results are in Section 6.

5.1 Formal Definitions

In the context of our analysis, let's consider a dataset D that can be partitioned into two subsets: P representing all the examples from the positive class and N representing all the examples from the negative class, formally, $D = P \cup N$. Next, let $S, S' \subset D$, denote the training and validation sets, respectively, randomly selected from D.

The datamap generation process (using Algorithm 1) operates on S, creating regions that are then classified into easy, ambiguous and hard to learn for the model. Then, Algorithm 2 creates the coreset, by focusing on the hard to learn regions, and the easy to learn regions are intentionally omitted or significantly under-represented by the coreset. This deliberate exclusion serves a dual purpose: it keeps the coreset small while enabling predictions for new entries during the inference phase based on the region to which they would be assigned in the datamap, particularly when these entries fall within the excluded easy to learn regions. These homogeneous regions are formally defined as follows:

[1]We achieve this efficiency using the *Aho-Corasick* algorithm [11]. By constructing a prefix tree from the leaves of these groups, we enable swift and accurate predictions during the inference phase, even with a substantial number of groups.

[2]Note that although these definitions and guarantees are tailored for the binary classification scenario, a partial extension to multi-class classification is demonstrated in Section 6.

Let $p_1, \ldots, p_{k_p} \subseteq D$ be the *easy to learn* and *ambiguous* regions in datamap that predominantly comprise positive examples, and formally, are positive homogeneous (given ψ homogeneity threshold):

$$\forall i, j \in [1, k_p] \ \frac{|p_i \cap S \cap P|}{|p_i \cap S|} \geq \psi, \ \& \ \ i \neq j \implies p_i, \cap p_j = \emptyset,$$

Similarly, $n_1, \ldots, n_{k_n} \subseteq D$, that primarily contain negative examples, are defined. The unions of these groups, and the complement sets are denoted as:

$$P' = \bigcup_{i=1}^{k_p} p_i \ , P'^c = \{x \in D \wedge x \notin P'\} \ , N' = \bigcup_{i=1}^{k_n} n_i \ , N'^c = \{x \in D \wedge x \notin N'\}$$

We now turn our attention to the two types of errors that can occur when an ML model is trained solely on the coreset, distinct from the entire dataset. To the best of our knowledge, this work is the first to provide theoretical guarantees for these types of errors.

Errors on the Entries from the Coreset Distribution [CrsErr]. These errors occur when the model misclassified an unseen entry taken from the same distribution as the data points in the coreset. We determine if an entry belongs to the coreset distribution by examining the datamaps. An entry is considered part of the coreset distribution if it falls outside the regions in the datamap designated as *easy to learn* and thus intentionally excluded or significantly under represented in the coreset. For simplicity, we assume that the sample ratio is 0. For this type of error, given that the examples falling within the *easy to learn* regions, denoted as P' and N', we define $P' \cup N'$ as the union of all those regions. We denote $(P' \cup N')^c$ as all the possible regions that do not intersect with $(P' \cup N')$. Consequently, we define the coreset as $d = (P' \cup N')^c \cap S$.

Errors on the Entries from the Excluded Data Distribution [ExcErr]. This category of error materializes when the model makes a mis-classification on an entry similar to the portions of the data that were deliberately excluded from the coreset. To determine if an entry is part of this excluded distribution, we rely on the datamaps, specifically checking whether it falls within the excluded regions. The model, having been trained on the coreset, has not encountered these data points during its training phase. Formally, if an entry falls within a region in P' (N'), the probability that this entry belongs to the class N (P) defines the likelihood of this type of error. Our inference method would predict that such an entry belongs to the positive (negative) class. We can formally define these probabilities as $Prob(x \in P' | x \in N)$ and $Prob(x \in N' | x \in P)$, and by referring to x as Bernoulli variables, and S' as a sample of it, we can get an estimation of this probability:

$$N_p := \begin{cases} 1, & \text{if } x \in P', \text{given } x \in N, \\ 0, & \text{otherwise, given } x \in N. \end{cases}, P_n := \begin{cases} 1, & \text{if } x \in N', \text{given } x \in P, \\ 0, & \text{otherwise, given } x \in P. \end{cases}$$

We employ the concept of the *Confidence Interval* for *Bernoulli Variables*, specifically utilizing the *Wilson method* [54]. This method offers an improvement over the normal approximation interval, making it suitable for small samples and skewed observations. With it, we can compute an upper bound for both N_p and P_n. In this calculation, we make use of the validation set S' to estimate these probabilities and their respective boundaries. The Wilson confidence is used because we have just the validation set S' for the evaluation, but we would like to generalize the error for the whole distribution of the data D. Thus, S' is used to create an upper bound for P_n.

Note that confidence interval calculation methods necessitate a user-defined parameter, denoted as δ, which signifies the desired confidence level for the variable to fall within the interval. This same δ value is used to establish the precision and recall boundaries. Importantly, our empirical results consistently show a much higher success rate than the chosen δ value. Therefore, when we refer to "by a probability of at least δ", we are indicating the upper bound of the probability associated with the *ExcErr* error. Formally:

$$\hat{P_n} = WilsonCIUpperBound(P \cap S', \delta) \geq P_n \, , P(x \in P' | x \in N) \leq \hat{N_p}$$
$$\hat{N_p} = WilsonCIUpperBound(N \cap S', \delta) \geq N_p \, , P(x \in N' | x \in P) \leq \hat{P_n}$$

Precision and Recall. The guaranteed boundaries presented in this section pertain to the model's precision and recall. Below is a brief overview of these crucial metrics including formal definitions. Precision Defined as the ratio of true positive predictions to the total number of positive predictions made by the model. formally,

$$\text{Precision} = \frac{\text{True Positives}}{\text{True Positives} + \text{False Positives}} = P(x \in P | x \in P_{pred})$$

Recall Measures the proportion of actual positive instances that are correctly identified by the model. formally,

$$\text{Recall} = \frac{\text{True Positives}}{\text{True Positives} + \text{False Negatives}} = P(x \in P_{pred} | x \in P)$$

Where P_{pred} defined as the group of the positive predictions of the model. These probabilistic definitions aid in subsequent calculations.

5.2 $CrsErr$ Errors

Let us delve into the $CrsErr$ errors, which are the errors on entries taken from the coreset distribution. As previously explained, we define an entry as taken from the coreset distribution when it does not fall within $P' \cup N'$. If a dataset and ML model hold the *Refined-fit* property, we claim that a model trained solely on the coreset $d = (P' \cup N')^c \cap S$ will not be worse then a model trained on the entire set S when making predictions on entries taken from the coreset distribution. Expecting performance stability in the targeted regions during training is not only reasonable but echoes established methodologies, such as [47] emphasis on enhancing performance by focusing on *challenging to learn* examples. Similarly, [31] follows a comparable strategy by ignoring easily learnable examples through the *GOSS* sampling method.

Intuition. During the training of an ML model, the goal is to fit itself to the given data. As perfect fitting for all training data examples is often unattainable, a compromise is necessary to minimize overall loss. Intuitively, a model trained on specific regions, such as $(P' \cap N')^c$, is expected to outperform a model trained on a larger and more diverse set, like the entire training set S. This is because the former model can concentrate on fitting to the specific regions of interest, while the latter must generalize to a broader range of data. This becomes even more accurate when considering that the regions excluded from the coreset are presumed to be "easy" for an ML algorithm to learn. Formally,

Refined-fit Property. A model M, and a data subset B, satisfy the refined fit property if the model trained solely on a set $B \cap S$, where $B \subseteq D$, will not perform worse, in terms of classification performance (Precision and Recall), on a test set taken from the distribution of $B \cap S'$ than the same model trained on the entire S. P_{pred} is the set of all the examples that the model $M(S, conf, y)$ classified as class P. P^*_{pred} is the set of all the examples that the model $M(S \cap B, conf, y)$ classified as class P. And in equivalent formal statement:

$$Prob(x \in P_{pred} \cap B | x \in P) \leq Prob(x \in P^*_{pred} \cap B | x \in P)$$
$$Prob(x \in P_{pred} \cap B | x \in N) \geq Prob(x \in P^*_{pred} \cap B | x \in N)$$

Limitation. The justification for relying on this property is twofold: it is easily verifiable with a given dataset and ML model, and its validation across 48 use-cases (6 datasets and 8 diverse ML models) in Section 6 demonstrates its common occurrence, making it practically applicable for guarantees. While our method shows strong performance in these scenarios, it has certain limitations, particularly when the refined-fit property is not satisfied. The refined-fit property can fail when crucial subsets of data, essential for learning, are removed. For instance, if a significant portion of a minority

class is excluded, the model may struggle to correctly predict this class, leading to decreased accuracy. Additionally, random sampling can inadvertently remove data points that are critical for distinguishing between classes, thereby reducing the model's overall effectiveness. To mitigate these issues, our algorithm carefully selects the regions to remove by focusing on easy-to-learn areas identified through datamaps, ensuring that the most informative and challenging data points are retained for training.

Moving forward, we will furnish guarantees for $ExcErr$ errors, specifically addressing errors on entries from the excluded data distribution, provided they adhere to the property.

5.3 $ExcErr$ Errors

Relying on our established property, we extend assurances to this error type, ensuring comprehensive performance guarantees for a model trained solely on the coreset. Below, we provide theoretical assurances regarding Recall and Precision, outlining the expected differences in these metrics between a model trained on the entire dataset and one trained solely on the coreset.

Recall Guarantees. We define the difference in the recall:

PROPOSITION 5.1. *If the Refined-fit property holds, the disparity in recall between a model trained on the entire dataset ($recall$) and the recall of a model trained on the coreset d ($recall^*$), with the addition of datamap utilization during inference, is governed by a probability denoted as δ. This difference is bounded by $\hat{P_n}$, which represents the upper limit on the fraction of the intersection between all the positive class examples denoted as P and the negative easy to learn set N'. In mathematical terms:*

$$recall - recall^* = \Delta_{recall} \leq Prob(x \in N' | x \in P) = \hat{P_n}$$

PROOF. Given that P_{pred} is the set of all the examples that the model classified as class P, $recall$ is defined as: $recall = P(x \in P_{pred} | x \in P)$. Next, we bound the $recall^*$ as all the entries that we have predicted are from the positive class, put aside the entries that were part of N' and thus classified as false class, although their real classification is positive. Using the Refined-fit property:

$$recall^* = Prob(x \in P^*_{pred} | x \in P) = Prob(x \in P^*_{pred} \cap (P' \cup N')^c | x \in P)$$
$$+Prob(x \in P^*_{pred} \cap (P' \cup N') | x \in P) \geq Prob(x \in P_{pred} \cap (P' \cup N')^c | x \in P) + Prob(x \in P' | x \in P)$$
$$\geq Prob(x \in P_{pred} \cap (P' \cup N')^c | x \in P) + Prob(x \in P_{pred} \cap (P' \cup N') | x \in P) - Prob(x \in N' | x \in P) =$$
$$= Prob(x \in P_{pred} | x \in P) - Prob(x \in N' | x \in P)$$

Overall, the difference of the recall of the two models is:

$$\Delta_{recall} = recall - recall^* \leq Prob(x \in P_{pred} | x \in P) - Prob(x \in P_{pred} | x \in P)$$
$$+Prob(x \in N' | x \in P) = Prob(x \in N' | x \in P)$$

With δ probability we get: $\Delta_{recall} \leq Prob(x \in N' | x \in P) = \hat{P_n}$ □

Precision Guarantees. We proceed to provide guarantees define the difference in precision:

PROPOSITION 5.2. *If the Refined-fit property holds, the difference in the precision of a model that was trained on all the data and the precision of the model that was trained on the coreset d and using the datamap in the inference, by probability of δ, is:*

$$\hat{P_n} Prob(x \in P) + \frac{\hat{N_p}}{\frac{Prob(x \in P)}{1 - Prob(x \in P)} + \hat{N_p}}$$

This ratio highlights the connection between the percentage of the positive vs. negative, and $\hat{N_p}, \hat{P_n}$ which represent errors of type *ExcErr* that our algorithm will make.

PROOF. We denote P_{pred} as all the examples that were classified as class P by the model. Thus, the precision of a model M trained over the whole training set S is:

$$precision = Prob(x \in P | x \in P_{pred}) =$$

$$\frac{Prob(P \cap P_{pred})}{P_{pred}} = \frac{Prob(P \cap P_{pred} \cap (P' \cup N')^c) + Prob(P \cap P_{pred} \cap P') + Prob(P \cap P_{pred} \cap N')}{Prob(P_{pred} \cap (P' \cup N')^c) + Prob(P_{pred} \cap P') + Prob(P_{pred} \cap N')}$$

For brevity, we denote:

$$\gamma := +Prob(P \cap P^c_{pred} \cap P') + Prob(P_{pred} \cap N')$$

Now we will add the expression $Prob(P \cap P^c_{pred} \cap P')$ to both the numerator and denominator, and as $0 \leq precision \leq 1$, the term will increase:

$$precision \leq$$

$$\leq \frac{Prob(P \cap P_{pred} \cap (P' + N')^c) + Prob(P \cap P_{pred} \cap P') + Prob(P \cap P^c_{pred} \cap P') + Prob(P \cap P_{pred} \cap N')}{Prob(P_{pred} \cap (P' \cup N')^c) + Prob(P_{pred} \cap P') + \gamma} =$$

$$= \frac{Prob(P \cap P_{pred} \cap (P' \cup N')^c) + Prob(P \cap P') + Prob(P \cap P_{pred} \cap N')}{Prob(P_{pred} \cap (P' \cup N')^c) + (Prob(N \cap P_{pred} \cap P') + Prob(P \cap P_{pred} \cap P')) + \gamma} =$$

$$= \frac{Prob(P \cap P_{pred} \cap (P' \cup N')^c) + Prob(P \cap P') + Prob(P \cap P_{pred} \cap N')}{Prob(P_{pred} \cap (P' \cap N')^c) + Prob(N \cap P_{pred} \cap P') + Prob(P \cap P') + Prob(P_{pred} \cap N')} =$$

$$\leq \frac{Prob(P \cap P_{pred} \cap (P' \cup N')^c) + Prob(P \cap P') + Prob(P \cap P_{pred} \cap N')}{Prob(P_{pred} \cap (P' \cup N')^c) + Prob(P \cap P') + Prob(P_{pred} \cap N')} =$$

$$= \frac{Prob(P \cap P_{pred} \cap (P' \cup N')^c) + Prob(P \cap P') + Prob(P \cap P_{pred} \cap N')}{(Prob(P \cap P_{pred} \cap (P' \cup N')^c) + Prob(N \cap P_{pred} \cap (P' \cup N')^c)) + Prob(P \cap P') + Prob(P_{pred} \cap N')} \leq$$

Because the fraction is between 0 and 1, and using the refined-fit property, the same increase in both the numerator and denominator will increase the entire expression:

$$\leq \frac{Prob(P \cap P^*_{pred} \cap (P' \cup N')^c) + Prob(P \cap P') + Prob(P \cap P_{pred} \cap N')}{(Prob(P \cap P^*_{pred} \cap (P' \cup N')^c) + Prob(N \cap P_{pred} \cap (P' \cup N')^c)) + Prob(P \cap P') + Prob(P_{pred} \cap N')} \leq$$

Using the refined-fit property:

$$\leq \frac{Prob(P \cap P^*_{pred} \cap (P' \cup N')^c) + Prob(P \cap P') + Prob(P \cap P_{pred} \cap N')}{(Prob(P \cap P^*_{pred} \cap (P' \cup N')^c) + Prob(N \cap P^*_{pred} \cap (P' \cup N')^c)) + Prob(P \cap P') + Prob(P_{pred} \cap N')} \leq$$

$$\leq \frac{Prob(P \cap P^*_{pred} \cap (P' \cup N')^c) + Prob(P \cap P') + Prob(P \cap N')}{Prob(P^*_{pred} \cap (P' \cup N')^c) + Prob(P \cap P')} =$$

Because $Prob(P^*_{pred} \cap N') = 0$

$$= \frac{Prob(P \cap P^*_{pred} \cap P'^c) + Prob(P \cap P') + Prob(P \cap N')}{Prob(P^*_{pred} \cap P'^c) + Prob(P \cap P')} = \frac{B + Prob(P \cap N')}{A}$$

when $1 \geq A \geq B$ and $B \leq Prob(x \in P)$

Next we defined $precision*$, the precision achieved by a model that was trained on S without N', P' and in the inference phase, follows the training enhancement (the use of the datamap for predicting the labels of the entities that falls in the regions of N', P'):

$$precision^* = \frac{Prob(P \cap P^*_{pred} \cap P'^c) + Prob(P \cap P')}{Prob(P^*_{pred} \cap P'^c) + Prob(P^*_{pred} \cap P')} =$$

$$= \frac{Prob(P \cap P^*_{pred} \cap P'^c) + Prob(P \cap P')}{Prob(P^*_{pred} \cap P'^c) + Prob(P \cap P') + Prob(x \in P' | x \in N) Prob(x \in N)}$$

With probability δ we get:

$$precision^* \geq \frac{Prob(P \cap P^*_{pred} \cap P'^c) + Prob(P \cap P')}{Prob(P^*_{pred} \cap P'^c) + Prob(P \cap P') + \hat{N_p}} = \frac{B}{A + \hat{N_p}} = \frac{1}{\frac{A}{B} + \frac{\hat{N_p}}{B}}$$

$$precision = \frac{B + Prob(x \in N' | x \in P) Prob(x \in P)}{A} \leq \frac{B + \hat{P_n} P(x \in P)}{A}$$

Thus, the difference between the precision of an ML model that has been trained over all S and the precision of a model that has been trained on a coreset d is:

$$\Delta_{precision} = precision - precision^* \leq \frac{\hat{P_n} Prob(x \in P)}{A} + \frac{1}{\frac{A}{B}} - \frac{1}{\frac{A}{B} + \frac{\hat{N_p}}{B}}$$

$$\hat{P_n} Prob(x \in P) + 1 - \frac{1}{1 + \frac{\hat{N_p}}{B}} \leq \hat{P_n} Prob(x \in P) + 1 - \frac{1}{1 + \frac{\hat{N_p} Prob(x \in N)}{Prob(x \in P)}}$$

$$= \hat{P_n} Prob(x \in P) + \frac{\hat{N_p} Prob(x \in N)}{Prob(x \in P) + \hat{N_p} Prob(x \in N)} = \hat{P_n} Prob(x \in P) + \frac{\hat{N_p}}{\frac{Prob(x \in P)}{Prob(x \in N)} + \hat{N_p}}$$

$$= \hat{P_n} Prob(x \in P) + \frac{\hat{N_p}}{\frac{Prob(x \in P)}{1 - Prob(x \in P)} + \hat{N_p}}$$

□

Further Discussion. Note that specific algorithm performance (precision and recall) are not needed to calculate those boundaries, they are generic for all algorithms that satisfy the property. Also, it is crucial to highlight that there are no restrictions on the groups N' and P', and the mathematical guarantees remain valid regardless of how they are selected. These guarantees naturally tend to improve when the groups exhibit higher homogeneity concerning the labels. Additionally, the coreset construction can be based on a subset of the features, not necessarily the entire feature set of the labeled data entries. Importantly, the mathematical guarantees provided above still hold in these scenarios. This flexibility allows the addition of features to an existing coreset during the feature engineering process while maintaining the mathematical guarantees. These claims have been proven in the experiments presented in Section 6.1.

5.4 opt_per Algorithm

Now, having defined the mathematical guarantees for performance when the training enhancement approach is used, we present an algorithm for coreset creation that optimizes the coreset based on the quality metric provided by users. The algorithm is similar to Algorithm 2, with modifications for separating the *easy* regions into mostly positive or negative homogeneous groups, as explained in this section. Algorithm 3 is designed to construct a coreset ($coreset$) from an original dataset (D), divided into training and validation sets (S and S' respectively), ensuring high-quality performance metrics based on specified differences in recall and precision from an ML model trained on the entire data while holding the defined property. The process starts with the creation of a $datamap$, dividing the dataset into regions that capture relevant patterns. The algorithm then iterates through these regions, evaluating their homogeneity and ease of learning for positive and negative instances. Homogeneous regions are added to the coreset, with particular attention given to those conducive to learning positive and negative instances.

For negative instances, the algorithm selectively adds regions to the coreset until the desired recall threshold is met or exceeded. Sampling techniques maintain efficiency, and the selected regions are then removed from consideration. A similar process is applied to positive instances, ensuring the coreset achieves the specified precision threshold. This adaptive approach leverages validation sets and iterative adjustments to the coreset composition, dynamically accommodating different regions based on their learning complexities. The resulting coreset is a representative subset

that optimally balances data inclusivity with computational efficiency, providing a valuable tool for subsequent machine learning tasks. The algorithm concludes by returning the generated coreset.

Algorithm 3 Creation of Datamap - Quality optimized

Input: Training set $S \subset D$,Validation set $S' \subset D$, recall threshold τ_{recall} or precision threshold $\tau_{precision}$, $tnum = 30$, $\tau = 5$, sample ratio $samp\_ratio$
Output: $d = [r_{i_1}, r_{i_2}, ..., r_{i_n}]$ coreset with performance τ_{recall} or $\tau_{precision}$

```
coreset = [ ]
p_easy_to_learn_cand = [ ]
n_easy_to_learn_cand = [ ]
datamap = datamap_creation(S, tnum, τ)
for region ∈ datamap do
    if homogeneous(region) = False then
        coreset.append(region ∩ S)
    end if
    if homogeneous(region) = True&positive(region) = True then
        p_easy_to_learn_cand.append(region)
    end if
    if homogeneous(region) = True&negative(region) = True then
        n_easy_to_learn_cand.append(region)
    end if
end for
p_val = {}
n_val = {}
Δ_recall = 0
Δ_precision = 0
for region ∈ sort_by_size_desc(n_easy_to_learn_cand) do
    n_val.append(region ∩ S')
    Δ_recall = calc_delta_recall(n_val)
    if Δ_recall > τ_recall then
        break
    end if
    coreset.append(sample(region ∩ S, samp_ratio))
    remove region from n_easy_to_learn_cand
end for
add n_easy_to_learn_cand ∩ S to coreset

for region ∈ sort_by_size_desc(p_easy_to_learn_cand) do
    p_val.append(region ∩ S')
    Δ_precision = calc_delta_precision(p_val, n_val)
    if Δ_precision > τ_precision then
        break
    end if
    coreset.append(sample(region ∩ S, samp_ratio))
    remove region from p_easy_to_learn_cand
end for
add p_easy_to_learn_cand ∩ S to coreset
return coreset
```

5.5 Explainability

This section concludes with an exploration of the explainability layer, which aids users in navigating the intricate trade-off between coreset size and recall or precision guarantees. We provide insights into the datamap that characterizes the selected coreset and introduce a set of investigation tools designed to help users select the optimal coreset size for their specific needs. These tools also offer deeper insights into the created coreset, enabling a comprehensive understanding of the selection process and its impact on model performance.

Coreset Explainability. To enhance user comprehension, we have developed a visualization tool that simplifies the datamap into a single decision tree, making it more readable. This tool highlights data patterns represented by regions in the datamap that were either omitted from the coreset or included in significantly reduced proportions due to their ease of

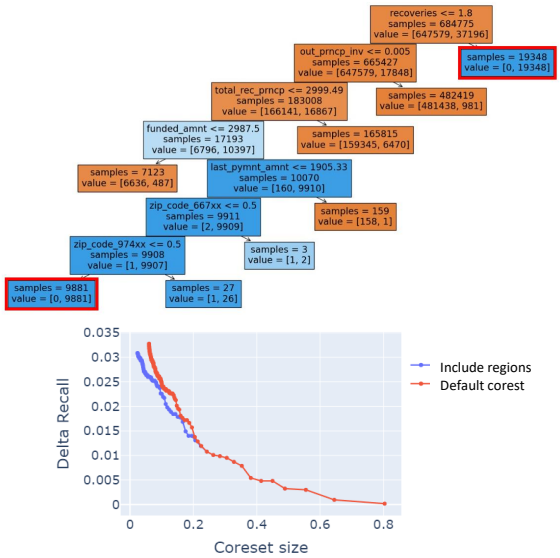


Fig. 1. CoreTab Explainabilty Layer

learning by the algorithm. Figure 1 presents an example of this investigation tool for the *LN* dataset. Each leaf in the decision tree corresponds to one or several regions sharing similar properties, displayed in a compact format. The number of samples indicates the size of each region, and red frames mark regions excluded from the coreset because they are "easy-to-learn."

Trade-off Visualization. Unlike other approaches that rely on default coreset size percentages or require user input, our method allows users to explore the impact of various coreset sizes on the model's *recall* and *precision*, guided by the mathematical guarantees outlined in Section 5. Additionally, users can investigate the effects of selecting different proportions of positive and negative examples, providing flexibility in coreset customization. Our approach also enables users to assess how including or excluding specific regions from the coreset impacts model performance. Figure 1(bottom) illustrates how *recall* changes when a user-selected region is added to the coreset, helping users determine the most suitable coreset size for their task.

6 EXPERIMENTS

Our system underwent thorough evaluation, assessing runtime performance and coreset quality with the F1-score metric across diverse models and datasets. We then evaluated the training enhancement method, including cross-validation testing, hyper-parameter tuning, and presenting empirical results for theoretical guarantees. Lastly, ablation studies are discussed.

Experimental Setup. Our system, implemented in Python 3.9 as a local library [2] is compatible with common EDA environments like Jupyter notebooks. We utilize the XGBoost library [9] for Gradient Boosting Decision Tree and scikit-learn [8] for the decision tree in the lighter version. The Aho-Corasick algorithm for inference uses the Python

implementation [11]. Experiments ran on an Intel Xeon CPU server with 24 cores and 1024GB of RAM, using default parameters of carefully tested, elaborated in this Section.

Datasets To demonstrate the versatility of our system, we used a variety of datasets with different characteristics, including balanced and unbalanced classes, varying sizes, and different combinations of column types (numeric, categorical, textual), all were previously used for coreset creation:

(1) Credit Cards [4] (*CC*): 250K rows, 31 columns, 0.17% positive (*P*) class.
(2) Loans [7] (*LN*): 856K rows, 1145 columns, 5.4% *P* class.
(3) Hepmass [6] (*HP*): 7M rows, 29 columns, 50% *P* class.
(4) Bank Fraud [1] (*BF*): 1M rows, 59 columns, 1.1% *P* class.
(5) Diabetes [5] (*DI*): 254K rows, 22 columns, 14% *P* class.
(6) Covertype [3] (*CT*): 581K rows, 55 columns, 36% *P* class.

For brevity, results are presented for representative datasets in each experiment, showcasing diversity in class balance, column types, and size. Evaluation also includes a multi-class dataset (*CT*), utilizing the one-versus-all training method for the first class (class 1 in the dataset).

Baselines. In our comparative analysis, we evaluate various baseline methods for creating data samples or subsets and constructing tabular coresets tailored for model training. For all baselines, we imposed a limit on the coreset size of 30,000 examples per class. All baselines, extensively discussed in Section 2, include:

Naive methods:

(1) RAN (*Random Sampling*): Randomly selects data tuples for forming a coreset.

Instance selection, Sampling and AQP Methods:

(2) IS-CNN (*Instance Selection - Condensed Nearest Neighbors*): Coreset selection for Condensed Nearest Neighbors classification [34], limited in adjusting coreset sizes.
(3) IS-CLUS (*Instance Selection - Cluster Centroids*): Coreset selection using centroids of data clusters generated with a specified number of clusters [34].
(4) VAE (*VAE*): An adaptation to a leading work from AQP field that uses a GAN to create synthetic data similar to the original distribution [49].

Coresets selection methods:

(4) CR (*CRAIG*): Coresets optimized for Logistic Regression [38].
(5) SBT (*SubStrat*): Genetic-based tuples selection for auto-ML training [32].
(6) TC (*Tree Coreset*): Coresets for k decision trees [30].
(7) FDMat (*FDMat*): The Computer Vision's state-of-the-art coreset creation method, with tabular adaptation [55].
(8) CoreTab (*CoreTab*): Our implementation of Algorithm 2, with *CoreTab-GBT* using the GBDT datamap for coreset selection and *CoreTab-DT* using the datamap for a single Decision Tree. both with $\tau = 5, \psi = 1$

Active Learning methods:

(9) EPIG (*EPIG*): Prediction-Oriented Bayesian Active Learning approach [46].
(10) UCS (*Uncertainty Sampling*): Adaptation of [45], adding iteratively examples with the least amount of confidence using the xgboost probabilities.

Default (*Default*) is referred to training the algorithm on the entire dataset. Although it is expected to outperform models trained on coresets, it requires significantly more time for training compared to models trained on a coreset alone.

ML models tested We test the created coreset over the performance of different ML models, all those models were used to train over the coresets that were created for different datasets, and by different baselines. We divide the models tested by type of ML model.

Tree based:

(1) XGBoost [17] (XGB)
(2) LightGBM [31] (LGBM)
(3) CATboost [44] (CAT)
(4) Random Forest [27] (RF)

Classic ML:

(5) Logistic Regression [20] (LR)
(6) Support Vector Machine (with RBF kernel) [19] (SVM).

Neural Networks for tabular data:

(7) TabNet[12] (TABNET).

LLMs for tabular data:

(8) GPT-4o model[3] [40] was fine-tuned on our training set (or coreset) using the OpenAI fine-tuning API. We selected 1,500 examples from each class to create the tailored model, applying the default hyperparameters provided by OpenAI. To enhance performance, we adjusted the batch size to 30.

Evaluation Metric To assess the performance of classifiers trained on each coreset and the entire dataset, we use the *F1-score* for the positive class, which in unbalanced datasets is the smaller class. The F1-score is a crucial metric for evaluating classification models, especially in imbalanced datasets, as it considers both precision and recall, providing a balanced measure of model performance. This ensures that the model accurately identifies all classes. The F1-score is calculated using the harmonic mean of precision and recall, defined as follows: F1-score $= 2 \cdot \frac{Precision \cdot Recall}{Precision + Recall}$, with precision and recall definitions provided in Section 5.

6.1 CoreTab Evaluation

Next, we compare baseline performance using our quality metrics and execution time. The results are divided into two sections: baselines comparison and usability across different ML models. Since other baselines can only generate coresets of predetermined sizes, we included only the version of CoreTab-Opt_size in the experiments to ensure a fair comparison, as this is the standard task.

Baseline Comparison. Table 1 presents the F1 scores for the XGBoost model when trained on coresets generated by various baselines. For all baselines and datasets we provide the F1 scores of models trained on the baselines' coreset and the running times (in seconds). Across all datasets, CoreTab consistently achieves the highest F1 scores (tied with *Default* that used the entire dataset). These scores are, on average, up to 10% higher than those of the other baselines. CoreTab also exhibits the second-best running times (excluding RAN), with coreset creation and model training typically completed in just a few minutes. Only FDMat had a faster runtime than ours; however, its performance was significantly worse. For example, in the *CT* dataset, it produced the poorest F1 score. This demonstrates the effectiveness of our sampling algorithm in capturing relevant portions of large datasets with respect to their labels. Note that the baseline TC was unable

[3] gpt-4o-mini-2024-07-18

to finish coreset creation within the reasonable time limit of 24 hours. Furthermore, the CRAIG algorithm generates a similarity matrix between all data points during execution, resulting in a matrix size of 20TB for the *LN* dataset, exceeding the available memory, denoted as *OOM* (Out of memory). Further comparisons involving the Logistic Regression model yield similar results, thus omitted. The poor performance of the AQP baseline (VAE) on most datasets, suggests that while the GAN may have been able to generate data broadly similar to the original, it likely failed to capture the critical nuances that distinguish positive and negative examples. This highlights a key limitation of GANs in this context: generating data that is superficially similar to the overall distribution, but missing the subtle, decision-driving patterns necessary for effective coreset construction for classification. The active learning baseline (UCS) beat our algorithm only once (*CT* dataset), due to the fact that easy to learn regions were not learned properly, as the under-sampling was not created optimally. However, when incorporating the TE factor, CoreTab outperforms the baselines across all datasets, including the UCS baseline by a large margin.

Note that, the performance did preserve more significantly on the unbalanced datasets. This is likely because unbalanced data tends to have larger homogeneous regions, which can lead to a greater reduction in sample size in these easy-to-learn regions.

Performance over different ML models. Table 2 presents the performance of various ML models, both classic and complex, when trained on coresets generated by the two versions of CoreTab. These results are compared to the performance of the exact models and configurations trained on the entire dataset, rather than just the coreset. The effectiveness of CoreTab across different ML models is attributed to the datamap-guided selection process, which prioritizes the retention of diverse and informative data points. This approach ensures that the coreset captures essential characteristics of the data that are beneficial for training a wide range of models, not just decision trees. As a result, CoreTab in both versions either performs on par with the default training on the full dataset or even outperforms it across all models and datasets. Moreover, the lightweight version of our algorithm, CoreTabDT, exhibits nearly the same performance as the full version while significantly reducing computation time. In some cases, it even outperforms the full version. Notably, the SVM model did not complete training within 24 hours on *BF*, *LN* and *HP*, and is thus absent from Table 2. However, it successfully completed training on the coresets in less than an hour.

Cross-Validation. While a single run is significantly quicker than training the model on the entire dataset for some datasets, for others, training over the whole dataset takes less time than creating the coreset and then training the model on it. However, the time saved using our coresets during cross-validation is noteworthy, as shown in Figure 4b. For an XGBoost model on the *LN* dataset, even with a small number of folds, cross-validation takes nearly half the time when trained on the coreset. With an increasing number of folds, the time differences become more pronounced, resulting in only 25% of the runtime for a 10-fold cross-validation.

Relevance of CoreTab to the Feature Addition Scenario. When working with tabular data, where features are often added incrementally, it's crucial to understand how well CoreTab performs under such conditions. To evaluate this, we conducted experiments focusing on feature addition and the robustness of CoreTab in these scenarios. In our initial experiment, we randomly excluded $20\% - 60\%$ of the columns from all datasets. This partial dataset was used to create a coreset and a datamap specific to the subset of features. We then reintroduced the excluded columns and trained an ML model using the coreset created from the initial subset. Notably, the added features significantly enhanced model performance in the default setting (i.e., training on all available features), suggesting that these columns provided essential information for the model, rather than being redundant or simply correlated with the existing features.

Baselines	*CC*(12%)			*LN*(8%)			*CT*(11%)			*DI*(22%)			*BF*(7%)			*HP*(1%)		
	F1-score	CCT (s)	MTT (s)	F1-score	CCT (s)	MTT (s)	F1-score	CCT (s)	MTT (s)	F1-score	CCT (s)	MTT	F1-score	CCT (s)	MTT	F1-score	CCT (s)	MTT
CoreTabDT	**0.87±0.02**	13.62 ± 1	0.35 ± 0.01	**0.985 ± 0.001**	92.5 ± 3	6.3 ± 0.3	0.793 ± 0.002	23.9 ± 0.5	0.64 ± 0.02	**0.343±0.007**	24.5 ± 0.3	0.7 ± 0.3	**0.192 ± 0.006**	52.3 ± 0.7	1.3 ± 0.4	**0.875 ± 0.001**	1870 ± 50	3.5 ± 0.5
CoreTabGBT	**0.87±0.02**	6.8 ± 0.5	0.35 ± 0.01	**0.985 ± 0.001**	102 ± 1	6.3 ± 0.3	0.849 ± 0.003	85 ± 4	0.64 ± 0.02	0.310 ± 0.003	40 ± 1	0.7 ± 0.3	0.119 ± 0.007	204 ± 7	1.3 ± 0.4	0.874 ± 0.001	8.1K ± 100	3.5 ± 0.5
TC	N/A	$> 24h$	N/A	N/A	$> 24h$	N/A	N/A	$> 24h$	N/A	N/A	$> 24h$	N/A	N/A	$> 24h$	N/A	N/A	$> 24h$	N/A
CR	0.75 ± 0.05	10K ± 100	0.35 ± 0.01	N/A	OOM	N/A	0.839 ± 0.002	20K ± 1.6K	0.64 ± 0.02	0.271 ± 0.004	39K ± 1	0.7 ± 0.3	N/A	OOM	N/A	N/A	OOM	N/A
SBT	0.80 ± 0.03	10.5K ± 200	0.35 ± 0.01	0.978 ± 0.001	48K ± 400	6.3 ± 0.3	0.832 ± 0.002	11.4K ± 300	0.64 ± 0.02	0.258 ± 0.007	10.6K ± 200	0.7 ± 0.3	0.055 ± 0.008	12K ± 700	1.3 ± 0.4	N/A	$> 24h$	N/A
RAN	0.79 ± 0.02	0.2 ± 0.01	0.35 ± 0.01	0.978 ± 0.001	0.1 ± 0.001	6.3 ± 0.3	0.835 ± 0.001	0.01 ± 0.02	0.64 ± 0.02	0.241 ± 0.004	0.7 ± 0.3	0.7 ± 0.3	0.054 ± 0.005	1.3 ± 0.4	1.3 ± 0.4	0.859 ± 0.001	3.5 ± 0.5	3.5 ± 0.5
IS-CNN	0.50 ± 0.04	238 ± 10	0.35 ± 0.01	N/A	$> 24h$	N/A	N/A	$> 24h$	N/A	N/A	$> 24h$	N/A	N/A	$> 24h$	N/A	N/A	$> 24h$	N/A
IS-CLUS	0.81 ± 0.01	719 ± 20	0.35 ± 0.01	0.94 ± 0.03	5K ± 70	6.3 ± 0.3	0.835 ± 0.003	830 ± 20	0.64 ± 0.02	0.334 ± 0.04	279 ± 6	0.7 ± 0.3	0.15 ± 0.06	4K ± 100	1.3 ± 0.4	0.847 ± 0.01	373 ± 12	3.5 ± 0.5
FDMat	0.86 ± 0.02	0.70 ± 0.03	0.35 ± 0.01	0.970 ± 0.008	62 ± 3	6.3 ± 0.3	0.68 ± 0.01	2.02 ± 0.04	0.64 ± 0.02	0.265 ± 0.003	0.87 ± 0.01	0.7 ± 0.3	0.064 ± 0.005	4.3 ± 0.05	1.3 ± 0.4	0.800 ± 0.006	50 ± 30	3.5 ± 0.5
UCS	0.86 ± 0.02	4.5 ± 0.1	0.35 ± 0.01	**0.985 ± 0.001**	160 ± 3	6.3 ± 0.3	**0.872 ± 0.003**	8.9 ± 0.2	0.64 ± 0.02	0.265 ± 0.004	3.7 ± 0.1	0.7 ± 0.3	0.063 ± 0.004	17.5 ± 0.4	1.3 ± 0.4	0.866 ± 0.008	41 ± 1	3.5 ± 0.5
EPIG	0.86 ± 0.02	620 ± 10	0.35 ± 0.01	0.982 ± 0.001	9K ± 600	6.3 ± 0.3	0.833 ± 0.003	2290 ± 40	0.64 ± 0.02	0.266 ± 0.005	861 ± 4	0.7 ± 0.3	0.066 ± 0.007	2570 ± 20	1.3 ± 0.4	0.857 ± 0.006	27K ± 3K	3.5 ± 0.5
VAE	0.003 ± 0.001	3700 ± 100	0.35 ± 0.01	0.0 ± 0.0	11K ± 70	6.3 ± 0.3	0.45 ± 0.08	3400 ± 20	0.64 ± 0.02	0.25 ± 0.01	2240 ± 50	0.7 ± 0.3	0.010 ± 0.005	4800 ± 100	1.3 ± 0.4	0.0 ± 0.0	26K ± 3K	3.5 ± 0.5
Default	**0.87±0.2**	N/A	7.1 ± 0.3	**0.985 ± 0.001**	N/A	165 ± 9	0.849 ± 0.003	N/A	6.6 ± 0.1	0.26 ± 0.01	N/A	1.86 ± 0.03	0.077 ± 0.006	N/A	23.8 ± 0.2	0.874 ± 0.001	N/A	160 ± 70

Table 1. Baseline Comparison: When the XGBoost model is trained on different coresets (including the entire dataset for the Default baseline), for various datasets (CC, LN, CT, DI, BF), the coresets created by CoreTab outperform all other baselines in terms of F1-score. CoreTab also achieves significantly lower coreset creation times (CCT) compared to the more complex baselines. However, for a single run, it may sometimes take more time than training the default setting. This advantage becomes more apparent with repeated training. Model training time (MTT) remains consistent for all baselines that could create a coreset, except for the Default setting, which was trained on the entire dataset.The percentage near the dataset represent the size of the created coreset.

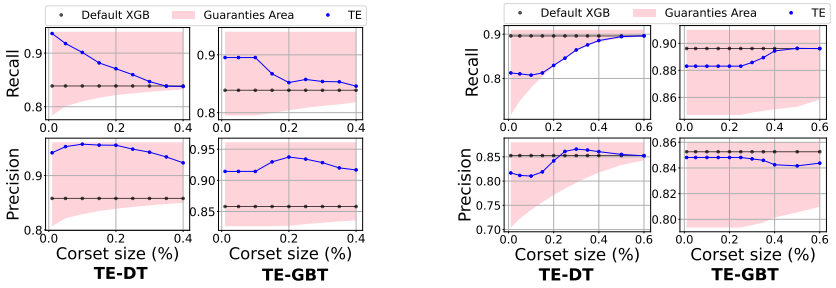
Data	Baselines	LR	SVM	XGB	LGBM	CAT	TABNET	RF	GPT-4
CC(12%)	CoreTabDT	**0.74±0.03**	**0.84 ± 0.02**	**0.87 ± 0.02**	0.84 ± 0.02	**0.87 ± 0.01**	0.1 ± 0.1	**0.86 ± 0.02**	**0.3 ± 0.2**
	CoreTabGBT	0.73 ± 0.03	**0.84 ± 0.02**	**0.87 ± 0.02**	**0.86 ± 0.02**	**0.87 ± 0.02**	0.1 ± 0.1	0.85 ± 0.02	0.12 ± 0.05
	Default	0.73 ± 0.04	0.78 ± 0.02	**0.87 ± 0.02**	0.83 ± 0.02	**0.87 ± 0.02**	**0.2 ± 0.2**	**0.86±0.02**	0.06 ± 0.01
BF(6%)	CoreTabDT	**0.12±0.01**	0.088 ± 0.002	**0.192 ± 0.006**	0.103 ± 0.002	**0.216 ± 0.008**	0.083 ± 0.003	**0.14±0.01**	**0.07 ± 0.01**
	CoreTabGBT	0.035 ± 0.004	**0.104 ± 0.0.002**	0.119 ± 0.007	**0.124 ± 0.003**	0.108 ± 0.005	**0.094±0.005**	0.018 ± 0.003	**0.07 ± 0.01**
	Default	0.02 ± 0.003	$N/A(24h+)$	0.077 ± 0.006	0.096 ± 0.002	0.064 ± 0.005	0.081 ± 0.004	0.003 ± 0.001	0.058 ± 0.003
LN(8%)	CoreTabDT	0.958 ± 0.003	0.70 ± 0.01	0.984± 0.001	0.984 ± 0.001	0.983± 0.001	**0.891 ± 0.3**	0.951 ± 0.004	0.93 ± 0.05
	CoreTabGBT	0.952 ± 0.004	**0.82 ± 0.0.01**	**0.985±0.001**	**0.985±0.001**	0.983 ±0.001	0.87 ±0.04	**0.967 ±0.006**	**0.971 ±0.007**
	Default	**0.961±0.001**	$N/A(24h+)$	**0.985±0.001**	0.980 ±0.001	**0.984±0.001**	0.85 ±0.11	0.963 ± 0.001	0.967 ± 0.021
CT(11%)	CoreTabDT	**0.683±0.003**	**0.923± 0.001**	0.793 ± 0.003	0.783 ± 0.003	0.823 ± 0.002	0.745 ± 0.006	0.948 ± 0.001	0.55 ± 0.04
	CoreTabGBT	0.675 ± 0.004	0.873 ± 0.002	**0.849± 0.003**	**0.815±0.002**	0.876 ±0.002	0.833 ± 0.004	0.937 ± 0.001	0.35 ± 0.25
	Default	0.676 ± 0.002	$N/A(24h+)$	**0.849 ± 0.003**	0.801 ± 0.002	**0.896±0.001**	**0.869 ± 0.003**	**0.952 ± 0.002**	**0.57 ± 0.04**
DI(22%)	CoreTabDT	**0.364±0.005**	0.422 ± 0.004	**0.343± 0.007**	0.427 ± 0.003	**0.35± 0.01**	0.42 ± 0.01	**0.323 ± 0.006**	**0.426 ± 0.001**
	CoreTabGBT	0.37 ± 0.01	**0.453 ± 0.003**	0.310 ± 0.003	**0.461±0.003**	0.263 ±0.003	**0.456±0.005**	0.267 ± 0.005	0.39 ± 0.03
	Default	0.241 ± 0.004	0.144 ± 0.004	0.26 ± 0.01	0.443 ± 0.003	0.259 ± 0.004	0.439 ± 0.005	0.254 ± 0.003	0.407 ± 0.003
HP(1%)	CoreTabDT	0.829 ± 0.001	0.858 ± 0.001	**0.875± 0.001**	0.867 ± 0.001	0.880 ± 0.001	0.878 ± 0.004	**0.872 ± 0.001**	0.62 ± 0.18
	CoreTabGBT	0.828 ± 0.001	**0.861 ± 0.001**	0.874 ± 0.001	0.865 ± 0.001	**0.881 ±0.001**	**0.880±0.001**	**0.872 ± 0.001**	0.77 ± 0.01
	Default	**0.837 ± 0.001**	$N/A(24h+)$	0.874 ± 0.001	**0.870 ± 0.001**	**0.881 ± 0.001**	0.879 ± 0.009	**0.872 ± 0.001**	**0.79 ± 0.02**

Table 2. Performance Comparison of Various ML Models: Performance (in terms of F1-Score) of various ML models, both classic and complex, when trained on coresets created by two versions of CoreTab and on the entire dataset (Default). CoreTab in both versions either performs on par with the default training on the full dataset or even outperforms it across all models and datasets.

Baselines	*CC* (12%)			*BF* (6%)		
	F1-score 20%	F1-score 40%	F1-score 60%	F1-score 20%	F1-score 40%	F1-score 60%
CoreTabDT	**0.87±0.01**	**0.87±0.01**	0.86 ± 0.01	**0.187 ± 0.004**	**0.184 ± 0.004**	**0.185±0.003**
CoreTabGBT	**0.87±0.01**	**0.87±0.01**	0.86 ± 0.01	0.127 ± 0.005	0.121 ± 0.007	0.12 ± 0.01
TE-DT	**0.87±0.01**	0.86 ± 0.02	0.86 ± 0.02	0.121 ± 0.004	0.12 ± 0.01	0.10 ± 0.01
TE-GBT	**0.87±0.01**	**0.87±0.01**	0.86 ± 0.01	0.13 ± 0.01	0.124 ± 0.008	0.12 ± 0.01
Default	0.78 ± 0.07	0.82 ± 0.03	0.85 ± 0.02	0.0002 ± 0.0001	0.010 ± 0.005	0.04 ± 0.01
Default All	**0.87±0.01**	**0.87±0.01**	**0.87±0.01**	0.08 ± 0.01	0.08 ± 0.01	0.08 ± 0.01

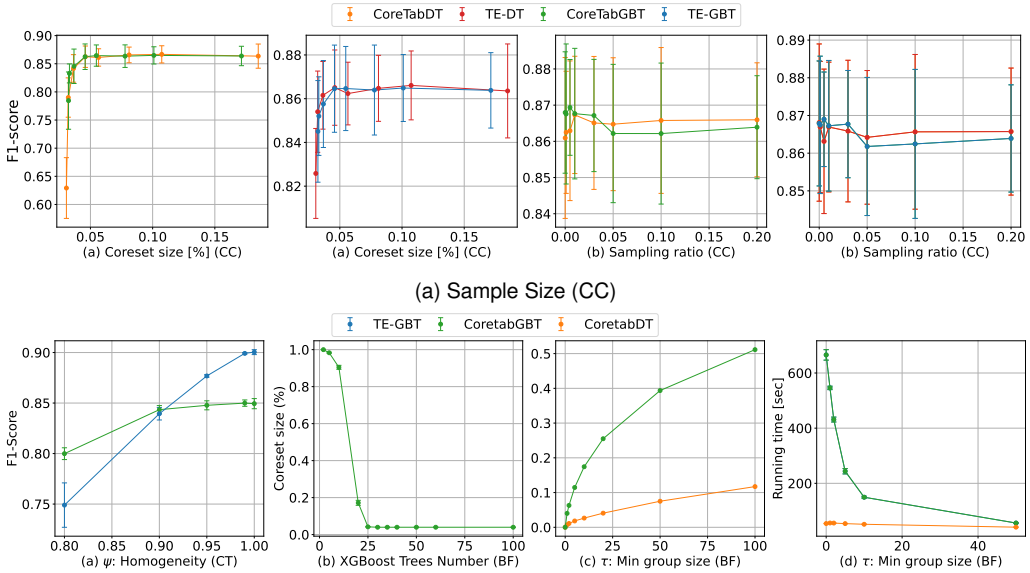
Table 3. Robustness to Feature Addition: The percentage near the dataset represent the size of the created coreset, and the F1-Score, indicates the portion of features remained in the training dataset (Including the entire dataset training both with the subset of features and all the features for the default, default all baselines). Even though CoreTab was trained on only a subset of the features, it selected rows representative of the entire dataset, resulting in model performance that either matched or exceeded that of the default model trained on the full dataset. This demonstrates CoreTab's remarkable robustness to scenarios involving frequent feature addition. Also the difference in performance between the default and the default all baselines shows that although the added features contain useful information, lacking them didn't harm CoreTab performance.

As shown in Table 3, our results demonstrate that the coreset remained relevant, whether the newly added features were correlated or uncorrelated with the original ones. The performance metrics show that models trained on coresets performed comparably to (*CC*), or even better than (*BF*), those trained on the full dataset. The reasoning behind this is



(a) Theoretical Guaranties Validation (CT) (b) Theoretical Guaranties Validation (HP)

Fig. 2. Theoretical Guarantees Validation: The pink shaded region represents the guaranteed bounds for either the *recall* or *precision* results of the model trained on the coreset, for both versions of CoreTab, and for different coreset sizes. The performance of CoreTab consistently falls within the guaranteed bounds for all tested coreset sizes, often outperforming the default XGB model trained on the entire dataset.



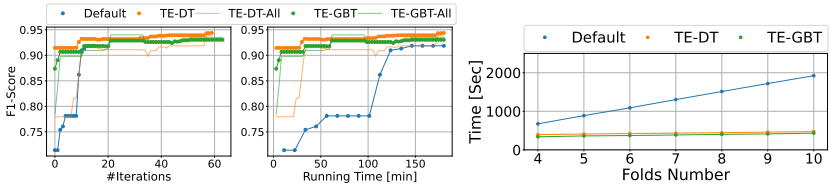
(a) Sample Size (CC)

(b) Datamap parameters

Fig. 3. Ablation Studies of CoreTab Parameters: We conducted an extensive ablation study to examine the influence of various parameters in our algorithm. By systematically altering individual parameters while keeping others at their default values, we assessed the impact on coreset size, model performance, and running time. In subsection 6.2, we provide a detailed analysis of each parameter. The dataset of each figure appears in parentheses in the X-axis

Model	Baselines	Default Recall	Coreset Recall	Default Precision	Coreset Precision
XGB	TE-DT	0.576 ± 0.012	0.69 ± 0.0126	0.653 ± 0.011	0.66 ± 0.008
	TE-GBT	0.696 ± 0.005	0.759 ± 0.009	0.669 ± 0.011	0.737 ± 0.012
LR	TE-DT	0.61 ± 0.003	0.645 ± 0.007	0.533 ± 0.009	0.528 ± 0.011
	TE-GBT	0.473 ± 0.016	0.48 ± 0.038	0.574 ± 0.013	0.573 ± 0.006

Table 4. Refined-fit property validation (CT)



(a) Hyper-Parameters Optimization (CT) (b) Cross Validation Running Time (LN)

Fig. 4. Reduced Running Times in Multiple Training: While CoreTab may not always be faster than the default setting of training an ML model on the entire dataset, it demonstrates significant benefits in reducing running times during multiple training scenarios.

Data	Baselines	XGB	LGBM	CAT	TABNET	GPT-4
CC (12%)	TE-DT	**0.87±0.02**	**0.86 ± 0.02**	**0.87 ± 0.01**	**0.3 ± 0.1**	**0.3 ± 0.2**
	TE-GBT	**0.87±0.02**	**0.86 ± 0.02**	**0.87 ± 0.02**	**0.3 ± 0.1**	0.23 ± 0.06
	Default	**0.87±0.02**	0.83 ± 0.02	**0.87 ± 0.02**	0.2 ± 0.2	0.06 ± 0.01
BF (6%)	TE-DT	**0.127±0.007**	0.143 ± 0.005	**0.13 ± 0.01**	0.13 ± 0.01	**0.16 ± 0.07**
	TE-GBT	0.083 ± 0.004	**0.155± 0.004**	0.079 ± 0.003	**0.141 ± 0.003**	0.12 ± 0.07
	Default	0.076 ± 0.006	0.096 ± 0.002	0.064 ± 0.005	0.081 ± 0.004	0.058 ± 0.005
CT (11%)	TE-DT	**0.934±0.001**	**0.930 ± 0.001**	**0.935 ± 0.001**	**0.926 ± 0.001**	**0.923 ± 0.007**
	TE-GBT	0.900 ± 0.004	0.887 ± 0.002	0.908 ± 0.001	0.900 ± 0.001	0.85 ± 0.05
	Default	0.849 ± 0.003	0.802 ± 0.001	0.897 ± 0.001	0.869 ± 0.003	0.57 ± 0.04

Table 5. Training Enhancement Performance: ML models trained on the coresets generated by CoreTab, using the Training Enhancement (TE) method, achieve F1 scores that match or exceed those of the default setting where the model is trained on the entire dataset.

that once the *easy-to-learn* regions are identified in the datamap, adding new features does not impact them. However, the *ambiguous-to-learn* and *hard-to-learn* regions, which are part of the coreset, can still benefit from the new information.

6.2 Training Enhancement Evaluation

Performance. Table 5 evaluates the training enhancement method's effectiveness throughout the training phase of four advanced algorithms trained on the CoreTab coreset, (Section 5). The performance is comparable to models trained on the entire dataset ($Default$) and even improves for most models. Examining a specific model, like XGBoost, reveals enhanced results with only a marginal increase in runtime. The coresets in this section are created with the same size limit mentioned in the experimental setup.

Hyper-Parameters Optimization. Next, we highlight the utility of training enhancement, extending beyond the fundamental training of a single algorithm. In Figure 4a, we illustrate the hyperparameter optimization process for an XGBoost model on the *CT* dataset. The optimization is constrained to a 3-hour runtime. It's important to note that XGBoost often requires extensive parameter tuning, and our optimization uses the Optuna algorithm. The F1-score, representing model performance, is measured on a fixed test set. This process was repeated and employed 5-fold cross-validation for result reliability. The results show that the optimization process on the coreset outperforms the full dataset in terms of efficiency and time. Importantly, training the model with the optimized coreset configuration on the entire dataset yields excellent performance, highlighting the adaptability of coreset-generated configurations.

Property Validation. In Section 5, we introduced the mathematical guarantees of the training enhancement method, relying on the Refined-fit property of the model and data. This property has been thoroughly validated across various models and datasets, confirming its commonality and reliability. Table 5 illustrates the validation of this property using

the *CT* dataset, in conjunction with XGB and LR models. Notably, both versions of CoreTab either match or exceed the performance of the default algorithm trained on the entire dataset.

Theoretical Guarantees Validation. Next we provide empirical validation of the mathematical guarantees by Figure 2. The pink shaded region represents the guaranteed bounds for either the *recall* or *precision* results of the model trained on the coreset, for both versions of CoreTab, and for different coreset sizes. As shown, the performance of CoreTab consistently falls within the guaranteed bounds for all tested coreset sizes, often outperforming the default XGB model trained on the entire dataset. Note that the mathematical assurances for TE-GBT for small coreset sizes surpass those of TE-DT, this is TE-GBT main advantage over TE-DT.

6.3 Parameter Tuning and Ablation Studies

Here, we delve into the influence of critical parameters in our algorithm, which affect both coreset size and datamap creation.

CoreTab *Algorithm.* We conducted experiments to assess the influence of different parameters in the CoreTab algorithm by measuring the average F1-score achieved while varying one parameter and using default values for the others. The results, summarized in Figure 3a, highlight several key findings. Firstly, it is evident that the coreset should be at most 5% of the entire dataset, as increasing the coreset size beyond this point does not significantly improve model performance. Additionally, we explored the impact of sampling from *easy-to-learn* regions and including them in the coreset. This parameter primarily affects when training enhancement method is not used. The experiments reveal that including more than 5% of entries from easy-to-learn regions in the coreset does not lead to substantial overall performance improvement.

Datamap Algorithm. Our ablation study assessed the impact of different parameters in Datamap algorithm on coreset size, model performance, and runtime (Figure 3b c-f). The parameter ψ (Definition 3.3) controls the retention of difficult-to-learn regions. For imbalanced datasets, ψ becomes irrelevant when set below $\frac{N}{N+P}$, as the dataset is treated as a one homogeneous group. high values of ψ (above 99% for TE-GBT and 90% for CoreTabGBT) ensure optimal performance. Lower ψ can alleviate coreset size limitations in CoreTabGBT and CoreTabDT. In our experiments, we set $\psi = 1$, though decreasing it may help when coreset size is restrictive in balanced datasets. The parameter $tnum$ (number of trees) used during CoreTabGBT coreset creation affects the datamap's ability to under-sample easy-to-learn regions. As shown in Figure 3b (d), using around 25 trees avoids coreset size limitations, with consistent results across datasets. We recommend $tnum = 30$ for added robustness in various scenarios. The parameter τ (Definition 3.4) balances coreset size and computational efficiency. Lower τ prevents size limitations, while higher τ reduces computation time, as shown in Figures 3b (e, f). τ has a stronger impact on CoreTabGBT than CoreTabDT. We used $\tau = 5$ in our experiments, but recommend a range of $2 \leq \tau \leq 10$ for optimal performance. Higher τ values may be acceptable for CoreTabDT, depending on the dataset.

7 CONCLUSION AND FUTURE WORK

In conclusion, we introduce CoreTab, an innovative algorithm designed for constructing data coresets optimized for training ML models using datamaps for GBDT models. Our experiments consistently showcase that these coresets, computed within minutes, surpass competing methods and even models trained on the full dataset. Moreover, a training enhancement technique leveraging datamap insights enhances performance with mathematical assurances, provided a defined property holds. Future research directions may include extending our method to handle multiple datasets, exploring its applicability

to various model types, accommodating diverse data types including unstructured data, and adapting to multi-class classification.

REFERENCES

[1] [n. d.]. Bank Fraud Dataset. https://www.kaggle.com/datasets/sgpjesus/bank-account-fraud-dataset-neurips-2022.

[2] [n. d.]. CoreTab git repository. https://anonymous.4open.science/r/coretab-4131/.

[3] [n. d.]. Cover Type Dataset. https://archive.ics.uci.edu/dataset/31/covertype.

[4] [n. d.]. Credit Card Dataset. https://www.kaggle.com/datasets/mlg-ulb/creditcardfraud.

[5] [n. d.]. Diabetes Dataset. https://archive.ics.uci.edu/dataset/34/diabetes.

[6] [n. d.]. Hempass Dataset. https://archive.ics.uci.edu/dataset/347/hepmass.

[7] [n. d.]. Loan Dataset. https://www.kaggle.com/deepanshu08/prediction-of-lendingclub-loan-defaulters.

[8] [n. d.]. Scikit-Learn Decision Tree Classifier. https://scikit-learn.org/stable/modules/generated/sklearn.tree.DecisionTreeClassifier.html.

[9] [n. d.]. Xgboost Library. https://xgboost.readthedocs.io/en/stable/.

[10] Sameer Agarwal, Barzan Mozafari, Aurojit Panda, Henry Milner, Samuel Madden, and Ion Stoica. 2013. BlinkDB: queries with bounded errors and bounded response times on very large data. In *Proceedings of the 8th ACM European Conference on Computer Systems*.

[11] Alfred V. Aho and Margaret J. Corasick. 1975. Efficient String Matching: An Aid to Bibliographic Search. *Commun. ACM* 18, 6 (June 1975), 333–340. https://doi.org/10.1145/360825.360855

[12] Sercan Ö Arik and Tomas Pfister. 2021. Tabnet: Attentive interpretable tabular learning. In *Proceedings of the AAAI conference on artificial intelligence*, Vol. 35. 6679–6687.

[13] Brian Babcock, Surajit Chaudhuri, and Gautam Das. 2003. Dynamic sample selection for approximate query processing. In *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*.

[14] Zalán Borsos, Mojmir Mutny, and Andreas Krause. 2020. Coresets via bilevel optimization for continual learning and streaming. *Advances in neural information processing systems* 33 (2020), 14879–14890.

[15] Vladimir Braverman, Vincent Cohen-Addad, H-C Shaofeng Jiang, Robert Krauthgamer, Chris Schwiegelshohn, Mads Bech Toftrup, and Xuan Wu. 2022. The power of uniform sampling for coresets. In *2022 IEEE 63rd Annual Symposium on Foundations of Computer Science (FOCS)*. IEEE, 462–473.

[16] Jiaxiang Chen, Qingyuan Yang, Ruomin Huang, and Hu Ding. 2022. Coresets for Relational Data and The Applications. *Advances in Neural Information Processing Systems* 35 (2022), 434–448.

[17] Tianqi Chen, Tong He, Michael Benesty, Vadim Khotilovich, Yuan Tang, Hyunsu Cho, Kailong Chen, Rory Mitchell, Ignacio Cano, Tianyi Zhou, et al. 2015. Xgboost: extreme gradient boosting. *R package version 0.4-2* 1, 4 (2015), 1–4.

[18] Vincent Cohen-Addad, Kasper Green Larsen, David Saulpic, Chris Schwiegelshohn, and Omar Ali Sheikh-Omar. 2022. Improved Coresets for Euclidean k-Means. *Advances in Neural Information Processing Systems* 35 (2022), 2679–2694.

[19] Corinna Cortes and Vladimir Vapnik. 1995. Support-vector networks. *Machine learning* 20 (1995), 273–297.

[20] David R Cox. 1958. The regression analysis of binary sequences. *Journal of the Royal Statistical Society: Series B (Methodological)* 20, 2 (1958), 215–232.

[21] Meng Fang, Yuan Li, and Trevor Cohn. 2017. Learning how to active learn: A deep reinforcement learning approach. *arXiv preprint arXiv:1708.02383* (2017).

[22] Dan Feldman, Matthew Faulkner, and Andreas Krause. 2011. Scalable training of mixture models via coresets. *Advances in neural information processing systems* 24 (2011).

[23] Jerome H Friedman. 2002. Stochastic gradient boosting. *Computational statistics & data analysis* 38, 4 (2002), 367–378.

[24] Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. 2020. Generative adversarial networks. *Commun. ACM* 63, 11 (2020), 139–144.

[25] Sariel Har-Peled and Soham Mazumdar. 2004. On Coresets for K-Means and k-Median Clustering. In *Proceedings of the Thirty-Sixth Annual ACM Symposium on Theory of Computing* (Chicago, IL, USA) *(STOC '04)*. Association for Computing Machinery, New York, NY, USA, 291–300.

[26] Xin He, Kaiyong Zhao, and Xiaowen Chu. 2021. AutoML: A survey of the state-of-the-art. *Knowledge-Based Systems* 212 (2021), 106622.

[27] Tin Kam Ho. 1995. Random decision forests. In *Proceedings of 3rd international conference on document analysis and recognition*, Vol. 1. IEEE, 278–282.

[28] Lingxiao Huang, Shaofeng H-C Jiang, Jianing Lou, and Xuan Wu. 2022. Near-optimal coresets for robust clustering. *arXiv preprint arXiv:2210.10394* (2022).

[29] Dino Ienco, Albert Bifet, Indrė Žliobaitė, and Bernhard Pfahringer. 2013. Clustering based active learning for evolving data streams. In *International Conference on Discovery Science*. Springer, 79–93.

[30] Ibrahim Jubran, Ernesto Evgeniy Sanches Shayda, Ilan I Newman, and Dan Feldman. 2021. Coresets for decision trees of signals. *Advances in Neural Information Processing Systems* 34 (2021), 30352–30364.

[31] Guolin Ke, Qi Meng, Thomas Finley, Taifeng Wang, Wei Chen, Weidong Ma, Qiwei Ye, and Tie-Yan Liu. 2017. Lightgbm: A highly efficient gradient boosting decision tree. *Advances in neural information processing systems* 30 (2017).

[32] Teddy Lazebnik, Amit Somech, and Abraham Itzhak Weinberg. 2022. SubStrat: A Subset-Based Optimization Strategy for Faster AutoML. *Proceedings of the VLDB Endowment* 16, 4 (2022), 772–780.

[33] Kaiyu Li, Yong Zhang, Guoliang Li, Wenbo Tao, and Ying Yan. 2019. Bounded Approximate Query Processing. *IEEE Transactions on Knowledge and Data Engineering* 31, 12 (2019), 2262–2276. https://doi.org/10.1109/TKDE.2018.2877362

[34] Huan Liu and Hiroshi Motoda. 2013. *Instance selection and construction for data mining*. Vol. 608. Springer Science & Business Media.

[35] Ziyang Liu, Peng Sun, and Yi Chen. 2009. Structured search result differentiation. *PVLDB* 2, 1 (2009).

[36] Qingzhi Ma and Peter Triantafillou. 2019. Dbest: Revisiting approximate query processing engines with machine learning models. In *Proceedings of the 2019 International Conference on Management of Data*. 1553–1570.

[37] Tung Mai, Cameron Musco, and Anup Rao. 2021. Coresets for classification–simplified and strengthened. *Advances in Neural Information Processing Systems* 34 (2021), 11643–11654.

[38] Baharan Mirzasoleiman, Jeff Bilmes, and Jure Leskovec. 2020. Coresets for data-efficient training of machine learning models. In *International Conference on Machine Learning*. PMLR, 6950–6960.

[39] Baharan Mirzasoleiman, Kaidi Cao, and Jure Leskovec. 2020. Coresets for robust training of deep neural networks against noisy labels. *Advances in Neural Information Processing Systems* 33 (2020), 11465–11477.

[40] OpenAI. 2023. GPT-4 Technical Report. *OpenAI* (2023). https://openai.com/research/gpt-4

[41] Y. Park, M. Cafarella, and B. Mozafari. 2016. Visualization-aware sampling for very large databases. In *ICDE*.

[42] Yongjoo Park, Barzan Mozafari, Joseph Sorenson, and Junhao Wang. 2018. Verdictdb: Universalizing approximate query processing. In *Proceedings of the 2018 International Conference on Management of Data*. 1461–1476.

[43] Omead Pooladzandi, David Davini, and Baharan Mirzasoleiman. 2022. Adaptive second order coresets for data-efficient machine learning. In *International Conference on Machine Learning*. PMLR, 17848–17869.

[44] Liudmila Prokhorenkova, Gleb Gusev, Aleksandr Vorobev, Anna Veronika Dorogush, and Andrey Gulin. 2018. CatBoost: unbiased boosting with categorical features. *Advances in neural information processing systems* 31 (2018).

[45] Manali Sharma and Mustafa Bilgic. 2017. Evidence-based uncertainty sampling for active learning. *Data Mining and Knowledge Discovery* 31 (2017), 164–202.

[46] Freddie Bickford Smith, Andreas Kirsch, Sebastian Farquhar, Yarin Gal, Adam Foster, and Tom Rainforth. 2023. Prediction-oriented bayesian active learning. In *International Conference on Artificial Intelligence and Statistics*. PMLR, 7331–7348.

[47] Swabha Swayamdipta, Roy Schwartz, Nicholas Lourie, Yizhong Wang, Hannaneh Hajishirzi, Noah A Smith, and Yejin Choi. 2020. Dataset Cartography: Mapping and Diagnosing Datasets with Training Dynamics. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*. 9275–9293.

[48] Alaa Tharwat and Wolfram Schenck. 2023. A survey on active learning: State-of-the-art, practical challenges and research directions. *Mathematics* 11, 4 (2023), 820.

[49] Saravanan Thirumuruganathan, Shohedul Hasan, Nick Koudas, and Gautam Das. 2020. Approximate query processing for data exploration using deep generative models. In *2020 IEEE 36th international conference on data engineering (ICDE)*. IEEE, 1309–1320.

[50] Murad Tukan, Cenk Baykal, Dan Feldman, and Daniela Rus. 2021. On coresets for support vector machines. *Theoretical Computer Science* 890 (2021), 171–191.

[51] Murad Tukan, Xuan Wu, Samson Zhou, Vladimir Braverman, and Dan Feldman. 2022. New coresets for projective clustering and applications. In *International Conference on Artificial Intelligence and Statistics*. PMLR, 5391–5415.

[52] Marcos R Vieira, Humberto L Razente, Maria CN Barioni, Marios Hadjieleftheriou, Divesh Srivastava, Caetano Traina, and Vassilis J Tsotras. 2011. On query result diversification. In *ICDE*.

[53] Jiayi Wang, Chengliang Chai, Nan Tang, Jiabin Liu, and Guoliang Li. 2022. Coresets over multiple tables for feature-rich and data-efficient machine learning. *Proceedings of the VLDB Endowment* 16, 1 (2022), 64–76.

[54] Edwin B Wilson. 1927. Probable inference, the law of succession, and statistical inference. *J. Amer. Statist. Assoc.* 22, 158 (1927), 209–212.

[55] Weiwei Xiao, Yongyong Chen, Qiben Shan, Yaowei Wang, and Jingyong Su. 2024. Feature Distribution Matching by Optimal Transport for Effective and Robust Coreset Selection. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 38. 9196–9204.

[56] Tong Yu and Hong Zhu. 2020. Hyper-parameter optimization: A review of algorithms and applications. *arXiv preprint arXiv:2003.05689* (2020).