Scaling Up Diffusion and Flow-based XGBoost Models

Jesse C. Cresswell [1] **Taewoo Kim** [1]

Abstract

Novel machine learning methods for tabular data generation are often developed on small datasets which do not match the scale required for scientific applications. We investigate a recent proposal to use XGBoost as the function approximator in diffusion and flow-matching models on tabular data, which proved to be extremely memory intensive, even on tiny datasets. In this work, we conduct a critical analysis of the existing implementation from an engineering perspective, and show that these limitations are not fundamental to the method; with better implementation it can be scaled to datasets 370× larger than previously used. We also propose algorithmic improvements that can further benefit resource usage and model performance, including multi-output trees which are well-suited to generative modeling. Finally, we present results on large-scale scientific datasets derived from experimental particle physics as part of the Fast Calorimeter Simulation Challenge.

1. Introduction

The design of neural network (NN) architectures with appropriate inductive biases for a given data modality has lead to incredible breakthroughs on text (Vaswani et al., 2017), audio (Hochreiter & Schmidhuber, 1997), image (Krizhevsky et al., 2012; He et al., 2016), graph (Kipf & Welling, 2016), and many other modalities. However, tabular data stands out in that tree-based architectures still often outperform NNs (Grinsztajn et al., 2022; McElfresh et al., 2023). This can largely be attributed to the lack of consistent structure in tabular data that NN design typically relies on.

Despite the success of boosted tree architectures like XGBoost (Chen & Guestrin, 2016) on *discriminative* tasks, they are rarely used for *generative* modeling (Nock & Guillame-Bert, 2022; 2023). This is surprising, as XGBoost brings several other meaningful advantages: XGBoost does not require significant data pre-processing (NNs are highly sensitive to data scale and distribution); XGBoost can operate on data that contains null values (NNs require null values to be imputed or entire columns dropped); XGBoost can be trained efficiently on CPU or GPU (NNs usually require GPU training); and XGBoost has superior explainability (Shapley values (Shapley, 1951) are intractable for large NNs, but the TreeSHAP algorithm makes them computable for trees (Lundberg & Lee, 2017; Lundberg et al., 2018)). Similar to MLP networks, XGBoost is a universal function approximator (Friedman et al., 2000; Friedman, 2001) and can be used to fit any function, at least in principle.

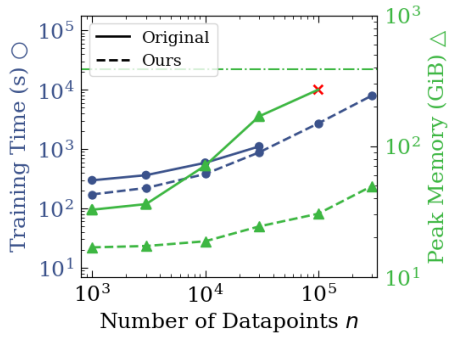


Figure 1: Comparison of training time and memory usage between the original implementation and ours. The × indicates job failure, and the horizontal line indicates the maximum system memory.

Recently, Jolicoeur-Martineau et al. (2024) proposed a method for training score-based diffusion (Song et al., 2021) and flow-matching (Liu et al., 2023; Albergo & Vanden-Eijnden, 2023; Lipman et al., 2023) generative models on tabular data by using XGBoost as the function approximator for a learnable vector field. Given the discussion above, this idea shows great promise, however, the original implementation was only benchmarked on small datasets (up to 11,000 datapoints with 16 features), and proved to be incredibly memory intensive (Figure 1 solid lines). Important scientific and industrial applications of tabular generative modeling typically operate at much larger scales, such as the Fast Calorimeter Simulation Challenge for generative modeling of particle physics interactions (Faucci Giannelli et al., 2022) with tabular datasets 370× larger than those used by Jolicoeur-Martineau et al. (2024).

[1]Layer 6 AI, Toronto, Canada. Correspondence to: Jesse C. Cresswell <jesse@layer6.ai>.

Proceedings of the ICML 2024 AI for Science workshop, Vienna, Austria.

In this work we conduct a deep and critical analysis of the implementation of diffusion and flow-matching models backed by XGBoost using engineering best-practices, and provide a new implementation re-engineered from the ground-up. Our implementation uses more than two orders of magnitude less CPU memory (Figure 1 dashed lines), showing that the algorithm is much more broadly applicable than previously thought. In addition, we demonstrate techniques to improve generative quality, including by using multi-output trees for generative modeling, which can more efficiently represent high-dimensional distributions. Finally, we demonstrate that the methods can scale in practice by applying them to large scientific datasets.

2. Background

First, we briefly review diffusion and flow matching models and then describe how to train them with XGBoost function approximators.[1] For a more extensive description of diffusion and flow matching models, see (Loaiza-Ganem et al., 2024). Finally, we introduce the application of interest from experimental particle physics.

2.1. Diffusion Models

Score-based diffusion models (Song et al., 2021) corrupt data $\mathbf{x}_0 \sim p_0$ by progressively adding noise as $t \in [0,1]$ increases, in a process modeled by a stochastic differential equation (SDE). Reversing this process enables the generation of data from pure noise. The reverse SDE involves a novel term, the score function $\nabla_{\mathbf{x}_t} \log p_t(\mathbf{x}_t)$, where p_t is the density corresponding to data at noise level t. Since the data density p_0 is not known in closed form, neither is p_t, so $\nabla_{\mathbf{x}_t} \log p_t(\mathbf{x}_t)$ cannot be directly computed. However, it can be estimated using a denoising score matching approach (Hyvärinen, 2005; Vincent, 2011) with the loss

$$L_{\text{SM}}(\theta) = \mathbb{E}_{t\sim\mathcal{U}(0,1)} w(t) \mathbb{E}_{\mathbf{x}_0\sim p_0, \mathbf{x}_t\sim p_t(\cdot|\mathbf{x}_0)} \|\mathbf{s}_\theta(\mathbf{x}_t, t) - \nabla_{\mathbf{x}_t} \log p_t(\mathbf{x}_t \mid \mathbf{x}_0)\|_2^2. \quad (1)$$

Here, $\mathbf{s}_\theta$ is a parameterized vector field that is directly regressed on the score function, while $w(t)$ is a positive-valued weighting function that can be chosen freely. In words, for a t sampled uniformly, and $\mathbf{x}_0$ drawn from the data distribution, we sample $\mathbf{x}_t$ from $p_t(\cdot \mid \mathbf{x}_0)$ which is Gaussian (Song et al., 2021),

$$p_t(\mathbf{x}_t \mid \mathbf{x}_0) = \mathcal{N}\Big(\mathbf{x}_t; \sqrt{1-\sigma_t^2}\mathbf{x}_0, \sigma_t^2 \mathbf{I}_D\Big). \quad (2)$$

The standard deviation σ_t depends on the details of the forward SDE. For generation, $\mathbf{s}_\theta$ replaces the score function in the reverse SDE which is then solved numerically.

[1] We refer to XGBoost throughout, but other tree-based regressors could be used. Our implementation takes advantage of XGBoost features that are currently unavailable in other gradient-boosted decision tree libraries.

2.2. Flow Matching

Like continuous normalizing flows (Chen et al., 2018), flow matching interpolates probability densities p_t for $t \in [0,1]$ (Liu et al., 2023; Albergo & Vanden-Eijnden, 2023; Lipman et al., 2023). For generative modeling we consider p_0 as the data, and select a simple prior $p_1 = \mathcal{N}(\mathbf{x}_1 | 0, \sigma^2)$. The interpolation is determined by a vector field at each time $\boldsymbol{\mu}_t$, which transports datapoints $\mathbf{x}_t$ via the ODE $\mathrm{d}\mathbf{x}_t = \boldsymbol{\mu}_t(\mathbf{x}_t)\mathrm{d}t$. When p_t and $\boldsymbol{\mu}_t$ jointly satisfy the continuity equation

$$\frac{\mathrm{d}}{\mathrm{d}t} p_t + \nabla_{\mathbf{x}} \cdot (p_t \boldsymbol{\mu}_t) = 0, \quad (3)$$

then p_t will be a properly normalized density at each t. To perform flow matching, one would train a model $\boldsymbol{\nu}_\theta(\mathbf{x}_t, t)$ of the vector field $\boldsymbol{\mu}_t(\mathbf{x}_t)$ by direct regression,

$$L_{\text{FM}}(\theta) = \mathbb{E}_{t\sim\mathcal{U}(0,1), \mathbf{x}_t\sim p_t} \|\boldsymbol{\nu}_\theta(\mathbf{x}_t, t) - \boldsymbol{\mu}_t(\mathbf{x}_t)\|_2^2. \quad (4)$$

However, in practice neither p_t nor $\boldsymbol{\mu}_t$ is uniquely determined, we can only sample from p_t for $t = 0$ (data) and 1 (prior), and we do not have access to $\boldsymbol{\mu}_t$ to evaluate at $\mathbf{x}_t$.

As a workaround, conditional flow matching (CFM) proposes to use conditional densities $p_t(\mathbf{x}_t \mid (\mathbf{x}_0, \mathbf{x}_1))$ and vector fields $\boldsymbol{\mu}_t(\mathbf{x}_t \mid (\mathbf{x}_0, \mathbf{x}_1))$, where $\mathbf{x}_0 \sim p_0$ is a training datapoint and $\mathbf{x}_1 \sim p_1$ is noise, such that both are tractable. For example (Tong et al., 2024), when we define

$$\begin{aligned} p_t(\mathbf{x}_t \mid (\mathbf{x}_0, \mathbf{x}_1)) &= \mathcal{N}\big(\mathbf{x}_t; t\mathbf{x}_1 + (1-t)\mathbf{x}_0, \sigma^2 \mathbf{I}_D\big), \\ \boldsymbol{\mu}_t(\mathbf{x}_t \mid (\mathbf{x}_0, \mathbf{x}_1)) &= \mathbf{x}_1 - \mathbf{x}_0, \end{aligned} \quad (5)$$

for some $\sigma \geq 0$, the continuity equation (Eq. 3) is satisfied. Now sampling data conditionally as $\mathbf{x}_t \sim p_t(\cdot \mid (\mathbf{x}_0, \mathbf{x}_1))$, the CFM loss has the same gradients as Eq. 4,

$$L_{\text{CFM}} = \mathbb{E}_{t\sim\mathcal{U}(0,1), \mathbf{x}_0\sim p_0, \mathbf{x}_1\sim p_1, \mathbf{x}_t\sim p_t(\cdot|(\mathbf{x}_0,\mathbf{x}_1))} \|\boldsymbol{\nu}_\theta(\mathbf{x}_t, t) - \boldsymbol{\mu}_t(\mathbf{x}_t \mid (\mathbf{x}_0, \mathbf{x}_1))\|_2^2. \quad (6)$$

Therefore, it will lead to the same model $\boldsymbol{\nu}_\theta(\mathbf{x}_t, t)$, but is actually tractable. Finally, new datapoints are generated by solving the ODE starting from $\mathbf{x}_1 \sim p_1$ but using the learned vector field $\boldsymbol{\nu}_\theta(\mathbf{x}_t, t)$ instead of $\boldsymbol{\mu}_t(\mathbf{x}_t)$.

2.3. ForestDiffusion and ForestFlow

There is a clear commonality between the two methods: both regress a parameterized vector field. In almost all applications to date, NNs are used to parameterize the vector field. Jolicoeur-Martineau et al. (2024) made the interesting observation that an XGBoost regressor (Chen & Guestrin, 2016) could be used instead. However, there are several major differences in training that must be overcome.

First, when using NNs Eq. 1 or 6 would be optimized by sampling a minibatch of data $\mathbf{x}_0 \sim p_0$, sampling $t \sim \mathcal{U}(0,1)$ independently for each $\mathbf{x}_0$, sampling fresh noise $\mathbf{x}_1 \sim p_1$

(Eq. 6 only), and then sampling $\mathbf{x}_t$ from the Gaussian in Eq. 2 or 5, respectively. In particular, the timestep t and random vector $\mathbf{x}_t$ would be sampled anew every batch, eventually leading to good coverage of the distributions in the loss function expectations. XGBoost is not trained with minibatches; it requires an entire dataset to be fed in and then minimizes the loss overall. Hence, the random vector $\mathbf{x}_t$ for each training point $\mathbf{x}_0$ would only be sampled once. For better coverage of the distribution, Jolicoeur-Martineau et al. (2024) proposed to duplicate each of the n training datapoints K times, and generate different $\mathbf{x}_t$ for each copy.

Second, whereas with a NN the time step t could be fed in as an additional input to the network during training and generation, Jolicoeur-Martineau et al. (2024) argued that simply adding t as a feature to XGBoost is unlikely to give sufficient emphasis to it, instead proposing to discretize t into n_t uniform steps and train a different XGBoost ensemble for each. The expectation over t is removed in the loss function Eq. 1 or 6, and t is instead treated as a constant for each of n_t separate loss functions.

Third, whereas a NN can easily be designed with a number of outputs equal to the number of features p in $\mathbf{x}$ (the same size as the target vector field), standard decision trees only output a scalar. A brute-force workaround is to train a different XGBoost ensemble to predict each feature.

Fourth, when conditional generation on a class label y is required, a NN can accept y as an input to adapt its behaviour while sharing parameters. Like conditioning on t, conditioning on y is better done by training a separate XGBoost ensemble for each of the n_y classes.

Combining these four solutions, Jolicoeur-Martineau et al. (2024) proposed ForestDiffusion and ForestFlow, aiming to realize the promises of tree-based generative modeling laid out in Section 1. While Jolicoeur-Martineau et al. (2024) reported excellent model performance, there are clear limitations, mainly the memory requirements from data duplication, and inefficient parameter use from conditioning with separate XGBoost ensembles. In total, on a tabular dataset of size $[n, p]$, both methods require training $n_t \cdot n_y \cdot p$ XGBoost ensembles on n_t different datasets of size $[n_i \cdot K, p]$, where n_i is the number of datapoints with label i such that $\sum_{i=1}^{n_y} n_i = n$. The recommended settings are $n_t \approx 50$ and $K \approx 100$, whereas $n_i \approx n/n_y$ for class-balanced data. To emphasize the scaling issues, the *libras* dataset featured in (Jolicoeur-Martineau et al., 2024) with $n = 288$ training datapoints required 151 GiB of CPU memory using the original implementation. We address these scaling issues below with novel techniques and better implementation.

2.4. Calorimeter Simulation

To motivate the need for scalable tabular data generation, we consider an important scientific application - calorimeter simulation. Measuring particle energy with calorimeters is one of the major components in particle accelerator experiments. To understand predictions from theory, physicists simulate interactions within calorimeters, but doing so from first principles is computationally expensive (Agostinelli et al., 2003; Allison et al., 2006; 2016). Generative models have seen remarkable uptake as surrogates for fast simulation, spurred on by the Fast Calorimeter Simulation Challenge (Faucci Giannelli et al., 2022) which provides large tabular datasets of calorimeter measurements, and evaluation metrics that are scientifically relevant. A comparison of the training dataset sizes from (Jolicoeur-Martineau et al., 2024) and from the Challenge is given in Table 1. In Appendix A we provide an extensive review of machine learning methods for calorimeter simulation.

Table 1: **Top:** The largest training datasets from (Jolicoeur-Martineau et al., 2024) in terms of n, p, and np. N/A means y is continuous. In addition, 25% of n is available for testing. **Bottom:** We scale to calorimeter datasets which are up to 370× larger in np.

Dataset	Datapoints n	Columns p	Classes n_y
california	16,512	9	N/A
libras	288	90	15
bean	10,888	16	7
Photons	121,000	368	15
Pions	120,800	533	15

3. Scaling Up

In this section we provide a step-by-step breakdown of the implementation of the main algorithm from (Jolicoeur-Martineau et al., 2024), shown in pseudocode in Algorithm 1, then re-engineer it to scale to calorimeter data. After resolving implementation issues, we offer new techniques to further improve algorithm and model performance, including by completely changing the XGBoost tree structure.

3.1. Limitations of the Existing Implementation

In Algorithm 1 the triple loop must be parallelized well for the method to scale to large datasets. Since the training dataset X_0 is duplicated K times and n_t different versions are generated to represent $\mathbf{x}_t$ at different timesteps, memory can become a severe issue which is compounded by the number of XGBoost ensembles to be trained. Unfortunately, these challenges are not handled well in the implementation published by (Jolicoeur-Martineau et al., 2024). To demonstrate the need for rework, we show the resource usage during training for a small dataset with $n = 1000$, $p = 100$, and $n_y = 10$ in Figure 2. Despite our computer having 385 GiB of CPU memory available, the training job failed. There are three clear problems that this example shows: (1) even on a modest dataset the absolute amount

Algorithm 1 ForestDiffusion and ForestFlow

Input: Dataset X_0 of size $[n, p]$, K, n_t.
$X'_0 \leftarrow K$-fold duplicate of the rows of X_0
$X_1 \leftarrow$ Dataset of $\mathbf{x}_1 \sim \mathcal{N}(0, \mathbf{I}_p)$ with the size of X'_0
for Timestep $t \in$ range(n_t) **do**
  for Class $y \in$ range(n_y) **do**
    for Feature $p_i \in$ range(p) **do**
      $X'_{0,y} \leftarrow$ rows of X'_0 with label y
      $X_{1,y} \leftarrow$ corresponding rows of X_1
      # Create $\mathbf{x}_t$ and regression targets
      $X'_{t,y}, Z_{t,p_i} \leftarrow$ Forward$(X'_{0,y}, X_{1,y}, t, p_i)$
      $f_{t,y,p_i} \leftarrow$Regress XGBoost on Z_{t,p_i} given $X'_{t,y}$
return $\{f_{t,y,p_i}\}$ # Set of $n_t n_y p$ XGBoost ensembles

of memory consumed is unexpectedly high (250 GiB); (2) memory usage grows at a constant rate during training, potentially leading to out-of-memory errors hours into a job; (3) training can fail due to memory issues even when the system maximum has not been reached.

3.2. Outlining the Existing Implementation

To begin unravelling the causes of the three undesirable behaviours above, we consider in detail the Python implementation by (Jolicoeur-Martineau et al., 2024).[2] First, the dataset X_0 of size $[n, p]$ is given as a Numpy array, with discrete labels for conditioning denoted as y. A scaler fits all data X_0 into the range $[-1, 1]$, after which the dataset is duplicated K times, giving X'_0 of size $[nK, p]$, and noise X_1 is sampled with the same shape. For conditioning on classes, n_y Boolean masks are created over X'_0. Next, the regression inputs and targets are created, denoted by X'_t and Z_t respectively. X'_t represents samples $\mathbf{x}_t$ from the distribution in Eq. 2 or 5, while Z_t is either the score function $\nabla_{\mathbf{x}_t} \log p_t(\mathbf{x}_t \mid \mathbf{x}_0)$ from Eq. 1 or the conditional vector field $\boldsymbol{\mu}_t(\mathbf{x}_t \mid (\mathbf{x}_0, \mathbf{x}_1))$ from Eq. 6. Finally, all $n_t \cdot n_y \cdot p$ models are trained in a parallel triple loop with the widely used Python parallelization library Joblib. These steps are shown in the following code snippet, which has been compressed to show only the crucial information, and uses ForestFlow as a representative example.

3.3. Analysis and Improvement of the Implementation

While the implementation looks innocuous, Figure 2 shows that there are serious engineering issues lurking. We aim to answer the following specific questions based on the observations above: (1) Why are memory requirements high for tiny datasets? (2) Why does memory usage increase during training? (3) Why do jobs fail before memory reaches 100% usage? We proceed by identifying issues, proposing solutions, and quantifying benefits, starting simple.

[2]We accessed `github.com/SamsungSAILMontreal/ForestDiffusion` as of Dec. 1,2023, commit hash `855281b`.

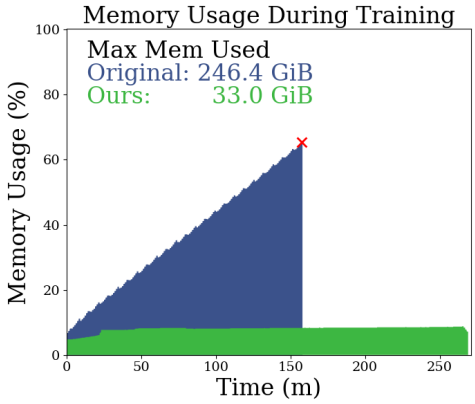


Figure 2: Training memory usage for the original implementation and ours. The red cross × indicates job failure.

Original Python Implementation of ForestFlow

```
1  from sklearn.preprocessing import
   ↪  MinMaxScaler as Scaler
2  import numpy as np, xgboost as xgb
3  from joblib import delayed, Parallel
4
5  X0, y, K, n_t, xgb_kw, n_jobs = inp()
6  n, p = X0.shape
7  # Scale data range to noise variance
8  scaler = Scaler(feature_range=(-1, 1))
9  X0 = scaler.fit_transform(X0)
10 # Duplicate data and generate noise
11 X0 = np.tile(X0, (K, 1)) # [n*K, p]
12 X1 = np.random.normal(size=X0.shape)
13 # Create Boolean masks over classes
14 mask = {}
15 y_uniq = numpy.unique(y)
16 for y_i in y_uniq:
17   mask[y_i] = np.tile(y == y_i, K)
18 # Discretize time into n_t steps
19 t = np.linspace(0, 1, num=n_t)
20 # Create ForestFlow regression targets
21 X_tr = t*X1 + (1-t)*X0 # [n_t, n*K, p]
22 Z_tr = X1 - X0 # [n*K, p]
23 # Train models in parallel triple loop
24 def train_parallel(X_tr_i, Z_tr_i):
25   model = xgb.XGBRegressor(**xgb_kw)
26   return model.fit(X_tr_i, Z_tr_i)
27 regressors = Parallel(n_jobs)(
28   delayed(train_parallel)(
29     X_tr[t_i][mask[y_i], :],
       ↪  Z_tr[mask[y_i], p_i]
30   ) for t_i in range(n_t) for y_i in
     ↪  y_uniq for p_i in range(p)
31 ) # list of n_t*n_y*p XGB ensembles
```

Issue 1: Regression inputs `X_tr` for all timesteps are created in memory at once, which is a large array of shape `[n_t,n*K,p]` (Line 21). Using the recommended values creates an array 5000 times the size of the training dataset, making even small datasets burdensome (Question 1).
Solution 1: Each XGBoost training call requires only the information at a single timestep `X_tr[t_i]`, and so this data should be generated on-the-fly within the `n_t` loop.

Benefit 1: We avoid holding the size `[n_t,n*K,p]` array in memory. This is in fact the first issue encountered when trying to apply the implementation to calorimeter data (Table 1), since a `numpy.float64` array of size `[50,120800*100,533]` requires **2.34 TiB** of memory.

Issue 1: Original

```
t = numpy.linspace(0, 1, num=n_t)
X_tr = t*X1 + (1-t)*X0
# [n_t, n*K, p] array in memory
```

Issue 1: Improvement

```
def train_parallel(X0,X1,Z_tr_i,t_i):
  X_tr_i = t_i*X1 + (1-t_i)*X0
  model = xgboost.XGBRegressor()
  return model.fit(X_tr_i, Z_tr_i)
```

Issue 2: Due to the Python global interpreter lock, parallelization is often handled through multiprocessing wherein the main process spawns worker processes and sends them copies of data. When a large array is assigned to multiple workers, Joblib automatically puts it into shared memory as a memory-mapped file, and passes only the reference to the workers. When an array *slice* like `X_tr[t_i][mask[y_i], :]` is passed to Joblib's `Parallel` call (Line 29), it is treated as a distinct array even though the same slice might appear for many jobs. Hence, Joblib creates a new array in shared memory for every slice throughout every call in the parallel triple loop. By default, Joblib stores the memory-mapped arrays in RAM disk, a virtual disk on RAM, and does not free that memory until all parallel jobs are done. This continuously increases RAM disk usage during training (Question 2) and can lead to out-of-memory errors if RAM disk usage reaches its capacity. For example, the maximum shared memory size on our Ubuntu 22.04 machine had been set to 189 GiB by default, and the failure in Figure 2 was caused by the RAM disk reaching this limit, even though RAM itself (at 385 GiB) was not at 100% usage (Question 3).

Solution 2: Instead of passing array slices within the `Parallel` call, pass the entire array and slice it inside worker processes. Upon the first call, Joblib puts the array into shared memory, but in subsequent calls it identifies the same array being requested and passes only a reference to workers.

Benefit 2: In the original implementation, each `train_parellel` call in the triple loop over `(n_t,n_y,p)` creates a copy of the input slices in shared memory. Looking only at `X_tr`, the triple loop consumes `n_t*p*sizeof(X_tr[t_i])=n_t*p*(n*K*p*8)` bytes in shared memory, which is `p` times more than Issue 1 (Question 1). For the Pions dataset (Table 1), this would amount to **1.22 PiB**. Our solution holds only one copy of `X0` and `X1` in shared memory, a factor of `n_t*p` less.

Issue 2: Improvement

```
def train_parallel(X0, X1, Z_tr, t_i,
↪  mask_i, p_i):
  X_tr_i = t_i*X1[mask_i, :] +
  ↪  (1-t_i)*X0[mask_i, :]
  Z_tr_i = Z_tr[mask_i, p_i]
  model = xgb.XGBRegressor(**xgb_kw)
  return model.fit(X_tr_i, Z_tr_i)
regressors = Parallel(n_jobs)(
  delayed(train_parallel)(
    X0, X1, Z_tr, t_i, mask[y_i], p_i
  ) for t_i in t for y_i in y_uniq for
  ↪  p_i in range(p) )
```

Issue 3: All trained XGBoost models are held in memory until the end of training, causing a steady increase of consumed memory as training progresses (Question 2). The memory consumed by these models is independent of `n`, but increases with `p` and `n_y` (Question 1).

Solution 3: After a model is trained, write it to disk, and delete it from memory. Use the Universal Binary JSON (UBJ) format as it is compatible across XGBoost versions, and is the fastest format overall for reading and writing with the best compression on disk.

Benefit 3: Workers writing their trained model to disk prevents the growth of memory usage over training, and bypasses the need to return models from the worker to the main process via pickling. Furthermore, it serves as a checkpoint so that training can be easily resumed upon system failure. ForestFlow requires `n_t*n_y*p` XGBoost ensembles, each made of `n_tree` trees which themselves have up to `2**(d+1)-1` nodes where `d` is the maximum depth. Each node of an XGBoost tree uses 53 bytes to store parameters, metadata, and training statistics. Using the recommended defaults of `n_tree=100`, `d=7`, and no regularization, essentially all trees will have the maximum 255 nodes, and would require in total **503 GiB** on Pions (Table 1).

Issue 3: Original

```
def train_parallel(...):
  ...
  return model.fit(X_tr_i, Z_tr_i)
```

Issue 3: Improvement

```
def train_parallel(...):
  ...
  model.fit(X_tr_i, Z_tr_i)
  model.save_model(f"{path}.ubj")
```

At this point we have reviewed the most significant training issues which together explain the three problematic observations from Figure 2, and provide the bulk of resource improvements that we report. In Appendix B we list four additional issues and solutions of smaller magnitude, and present a Python implementation with all of our changes. Figure 2 shows that our implementation solves the three observed problems. However, training is only one part of the story. In Appendix B.2 we investigate improvements to generation, with our method proving to be more than an order of magnitude faster.

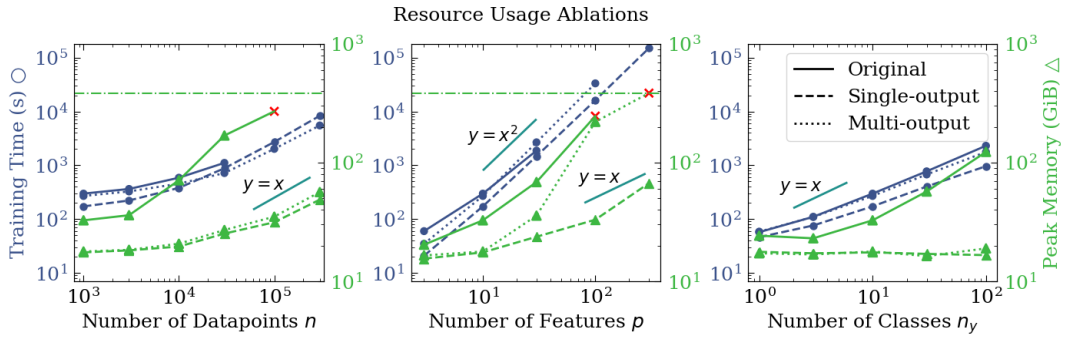


Figure 3: Resource usage of the ForestFlow implementation from (Jolicoeur-Martineau et al., 2024), compared to our implementation, and our extension to multi-output trees. A red cross × indicates job failure. The horizontal line indicates the maximum system memory used for all models at 385 GiB.

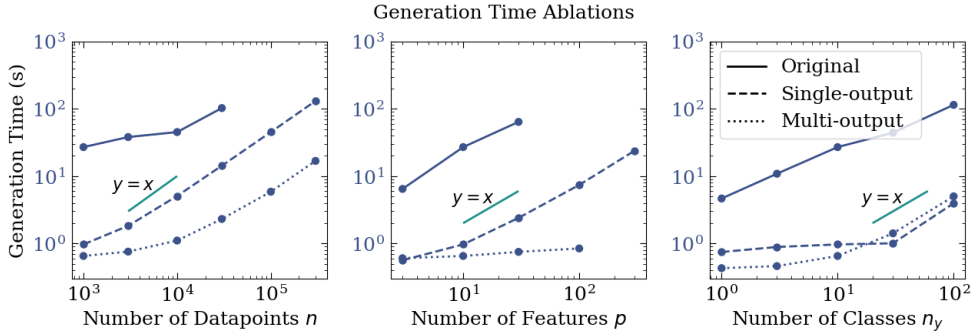


Figure 4: Generation speed of the ForestFlow implementation from (Jolicoeur-Martineau et al., 2024), compared to our implementation, and our extension to multi-output trees. Each datapoint corresponds to a model trained for Figure 3, where missing datapoints mean the training job did not complete.

3.4. Algorithmic Modifications

Beyond improving the implementation of Algorithm 1, we also offer modifications that can improve model performance or resource efficiency. First, we propose class-conditional data scaling, since during training each model only sees data from one class. Second, we find that class-conditional sampling using the labels from the training set improves the distributional characteristics of generated data. Third, for large datasets like Pions, we find the importance of certain hyperparameters changes, and recommend different defaults for better performance with fewer resources.

Finally, we propose a significant change to the structure of trees used by XGBoost which is more suited to the high-dimensional outputs required in generative modelling. Instead of training p single-output trees independently, one for each feature, we propose to use **multi-output trees** (Zhang & Jung, 2021; Ying et al., 2022; März, 2022; Iosipoi & Vakhrushev, 2022; Schmid et al., 2023), where a single tree outputs p values. The advantages are clear: p times fewer XGBoost ensembles are required to represent the vector field, which is a massive benefit for generation speed and trained model memory requirements. The above modifications are fully described in Appendix C.

4. Experimental Results

In this section we demonstrate the improved resource scaling of our implementation, provide benchmarking on small tabular datasets for model performance improvements, then present our results on the much larger calorimeter datasets.

4.1. Resource Usage Scaling

To begin, we quantify resource usage using synthetic datasets of controllable size. Features and labels are randomly generated, with one of n, p, or n_y altered from base values of $n = 1000$, $p = 10$, and $n_y = 10$. The expected behaviour in n, p, and n_y was hinted at in Section 2.3. Increasing n should increase training time, but at most linearly due to XGBoost's `hist` method (Chen & Guestrin, 2016), and linearly increase memory usage. Increasing n_y with n held fixed means increasing the number of ensembles trained, but with each on a smaller dataset, hence memory requirements should slightly decrease. Meanwhile, increasing p leads to quadratic scaling in time since it both increases the number of ensembles required, and the size of each dataset, the latter of which also implies linear memory requirements.

Figure 3 shows our controlled experiment for runtime and memory scaling. Both the Original and our (Single-output) implementations show similar scaling in time as n increases,

Table 2: Average rank (standard error) of generated data quality over 27 datasets. Lower is better.

	$W1_{\text{train}}$	$W1_{\text{test}}$	$\text{Cov}_{\text{train}}$	Cov_{test}	R^2_{gen}	$F1_{\text{gen}}$	P_{bias}	cov_{rate}	Avg.
GaussianCopula (Joe, 2014)	10.0±0.3	10.1±0.3	9.9±0.4	10.0±0.4	9.1±0.1	9.5±0.4	8.6±1.4	10.6±0.5	9.7±0.1
TVAE (Xu et al., 2019)	8.0±0.3	7.6±0.4	8.3±0.3	8.3±0.3	9.7±0.6	8.9±0.6	10.7±0.5	10.1±0.3	9.0±0.0
CTGAN (Xu et al., 2019)	11.4±0.2	11.3±0.2	11.2±0.2	11.1±0.2	11.6±0.2	11.4±0.2	8.6±1.1	10.6±0.5	10.9±0.1
CTAB-GAN+ (Zhao et al., 2024)	9.8±0.3	9.6±0.3	9.9±0.4	9.7±0.4	10.0±0.2	9.9±0.4	10.6±0.6	8.6±1.4	9.7±0.1
STaSy (Kim et al., 2023)	9.0±0.2	9.2±0.2	7.9±0.3	8.0±0.4	8.4±1.3	7.6±0.5	7.0±1.2	6.6±1.4	7.8±0.2
TabDDPM (Kotelnikov et al., 2023)	4.4±0.9	5.8±0.8	4.2±0.7	4.8±0.7	1.9±0.6	5.2±0.9	3.3±1.4	3.1±0.8	4.1±0.1
ForestFlow-original	3.6±0.2	3.4±0.3	2.9±0.3	3.0±0.4	2.5±0.4	4.6±0.4	4.2±0.8	3.5±0.8	3.5±0.1
ForestFlow-SO	1.9±0.2	2.2±0.2	2.4±0.3	2.9±0.4	2.5±0.4	4.1±0.5	4.2±0.8	3.5±0.8	3.0±0.1
ForestFlow-MO	4.2±0.3	3.0±0.3	4.0±0.3	3.7±0.4	3.7±0.5	5.1±0.5	3.6±0.5	3.4±0.3	3.9±0.0
ForestDiffusion-original	5.4±0.3	5.4±0.3	5.7±0.4	5.3±0.4	5.8±0.3	4.3±0.5	5.2±0.9	5.8±0.8	5.4±0.1
ForestDiffusion-SO	3.8±0.3	4.0±0.3	4.4±0.4	4.7±0.4	5.8±0.3	2.4±0.4	5.2±0.9	5.8±0.8	4.5±0.1
ForestDiffusion-MO	6.6±0.3	6.4±0.3	7.1±0.4	6.6±0.4	7.0±0.4	5.3±0.4	6.9±0.9	6.4±0.8	6.5±0.1

as nearly all compute time is spent in calls to XGBoost. However, our implementation is more than two orders of magnitude more memory efficient, and peak memory usage scales linearly at large n. The Original implementation leads to job failures with as few as $n = 10,000$ datapoints. The number of features p has the biggest impact on resource requirements, but our implementation achieves the expected linear memory scaling whereas the Original has quadratic scaling. Finally, for n_y our implementation uses a constant amount of memory, whereas the Original shows worse than linear scaling.

While multi-output trees behave similarly to single-output in n and n_y, they suffer from worse scaling in p. Multi-output trees reduce the number of ensembles required by a factor of p, but each tree's training is more memory intensive as XGBoost must search over a higher dimensional leaf space.

Next, in Figure 4 we show the time required to generate five batches of n datapoints using the same models trained for Figure 3. Not only is our implementation more than an order of magnitude faster for most settings, we see an even greater improvement for multi-output trees. The near-constant scaling in p showcases the benefit of generating all p outputs with a single ensemble. Hence, multi-output trees are a strong candidate for applications that require large volumes of generated data.

4.2. Model Performance on Benchmark Datasets

We directly compare our proposed algorithmic improvements from Section 3.4 to the original ForestDiffusion and ForestFlow on 27 datasets (Muzellec et al., 2020), across 8 metrics, which we averaged over 5 generated datasets for each of 3 seeds. The metrics convey the quality of generated samples along four dimensions: distributional distance (Wasserstein-1 distance to the training or test set), diversity (Coverage (Naeem et al., 2020) of the training or test set), usefulness for training discriminative models (R^2_{gen} and $F1_{\text{gen}}$), and usefulness for statistical inference (P_{bias} and cov_{rate}). For comparison, 6 baseline generative models are shown ranging from statistical methods to tabular diffusion models. The 27 datasets, 8 evaluation metrics, and 6 baseline methods are repeated from (Jolicoeur-Martineau et al., 2024), and described fully in Appendix D along with our hyperparameter settings and experimental details.

Table 2 shows the average rank that each method obtained on each metric,[3] where the average and standard error are computed over the 27 datasets, similar to prior tabular generation papers (Gorishniy et al., 2021; 2023). While the original implementation of ForestFlow already outperforms advanced methods like TabDDPM (Kotelnikov et al., 2023) on several metrics, our performance improvements to the single-output (SO) case further establish it as a state-of-the-art tabular generative model. Multi-output (MO) trees are also competitive with TabDDPM, slightly outperforming it on average. While potentially more expressive in terms of generating correlations between features, multi-output trees have been shown to need thousands of boosting rounds to surpass single-output trees on discriminative tasks (Zhang & Jung, 2021). Here we use the same number of trees per ensemble n_{tree} to make the training times comparable (Figure 3), but also implying that the multi-output versions use roughly p times fewer parameters. Raw metric values are plotted in Appendix D.5.

4.3. CaloForest - Flow-based XGBoost Models for Calorimeter Data

As a scaled-up example, we model the Photons and Pions datasets (Table 1) from the Fast Calorimeter Simulation Challenge (Faucci Giannelli et al., 2022). Each dataset comes with training and test splits of size $\approx 121,000$, where each datapoint represents the energies deposited in voxels of a calorimeter by an incident particle.

Competitive NN-based approaches perform extensive preprocessing to the data to facilitate training (Krause & Shih,

[3] We also include "Avg.", the column-wise average and standard deviation to summarize the table.

Table 3: Model performance on calorimeter data. Lower is better.

Photons	AUC	$E_{\text{dep}}/E_{\text{inc}}$	$E_{\text{dep,L0}}$	$\text{CE}_{\eta,\text{L1}}$	$\text{CE}_{\phi,\text{L1}}$	$\text{Width}_{\eta,\text{L1}}$	$\text{Width}_{\phi,\text{L1}}$
CaloMan (Cresswell et al., 2022)	0.9998	0.0020	0.0001	0.0462	0.0394	0.0366	0.0865
CaloForest (Ours)	0.8392	0.0778	0.0033	0.0056	0.0029	0.0241	0.0228
Pions	AUC	$E_{\text{dep}}/E_{\text{inc}}$	$E_{\text{dep,L0}}$	$\text{CE}_{\eta,\text{L1}}$	$\text{CE}_{\phi,\text{L1}}$	$\text{Width}_{\eta,\text{L1}}$	$\text{Width}_{\phi,\text{L1}}$
CaloMan (Cresswell et al., 2022)	0.9986	0.0404	0.0002	0.0477	0.0282	0.2380	0.2183
CaloForest (Ours)	0.9119	0.0625	0.0384	0.0268	0.0266	0.1935	0.1978

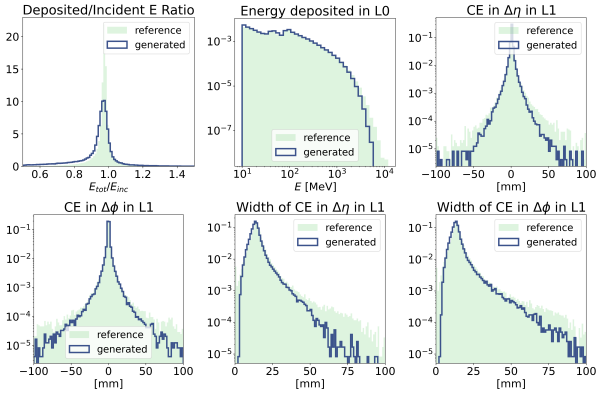


Figure 5: Histograms of high-level features comparing generated Photons samples to the test set. Note the log scale of the y-axis for all but the first plot.

2021b; Mikuni & Nachman, 2022; Cresswell et al., 2022). Since XGBoost is robust to features on different scales and with different distributions we need only perform min-max scaling on each class. We used our single-output variant of ForestFlow as it gave the best performance in Table 2. Based on the hyperparameter importance discussion in Appendix C.3, we discretized time into $n_t = 100$ steps, and duplicated each datapoint $K = 20$ times. Each XGBoost ensemble had $n_{\text{tree}} = 20$ trees of maximum depth 7, a learning rate of 1.5, and all other XGBoost hyperparameters left as defaults. We trained up to 20 XGBoost ensembles in parallel, each with 2 CPUs (see discussion in Appendix B), on a single desktop workstation with 250 GiB RAM and 40 CPUs (Intel Xeon Silver 4114T). In total, for the Photons model 552,000 XGBoost ensembles were trained in 135 hours with a peak memory burden of 54 GiB, while the Pions model used 799,500 ensembles, completed in 281 hours, and required 78 GiB of memory. Generation of n datapoints (matching the number in the training and test sets) took 231 s for Photons (1.91 ms per datapoint), and 347 s for Pions (2.87 ms per datapoint), which can be compared to 40 ms per datapoint for diffusion-based NNs on a GPU (Mikuni & Nachman, 2022), or anywhere from 100 ms to 3 s for the widely used Geant4 simulator (Aad et al., 2021).

The Challenge uses three types of metrics to evaluate models: resource usage, especially generation time as discussed above; distributional closeness to the test set judged by the χ^2 separation power between histograms in features crafted by domain experts; and ROCAUC of a binary classifier trained on a mix of real and generated data. We describe these metrics in full detail in Appendix A.1. The latter two types of metric are shown in Table 3 as compared to a NN-based approach. We see that ForestFlow produces more "realistic" datapoints in that they are harder for a classifier to distinguish from the test set. Feature histograms are shown in Figure 5 which confirms an accurate representation of the true distribution. Complete results for all metrics are given in Appendix A.2).

5. Conclusions and Limitations

In this work we have pushed the boundaries of tabular data generation backed not by neural networks, but by XGBoost. As discussed, XGBoost offers several tantalizing advantages over NNs for generative modelling: XGBoost's better performance on discriminative tabular tasks may translate to better tabular generation; it is robust without data preprocessing; it natively handles missing values; it can be efficiently trained on CPU; and finally it offers improved explainability. However, the great differences in the mechanics of XGBoost training compared to NNs led Jolicoeur-Martineau et al. (2024) to propose overparameterized models that do achieve state-of-the-art performance, but potentially at the cost of practicality and scalability.

Our contributions re-engineered the inner workings of these models leading to peak memory burdens reduced by more than two orders of magnitude, allowing them to scale to datasets 370× bigger than previously tested. We also offered modifications and new techniques that pushed model performance even further. Finally, we proposed the use of multi-output trees which are more suited to the high-dimensional outputs required in generative modelling, and showed that generation time can be reduced by an additional order of magnitude for applications that require large volumes of generated data. Still it is clear that the methods we discuss have limitations. Models trained with ForestDiffusion and ForestFlow are highly overparameterized (Section 4.3), still require significant computational resources (Figure 3), and our proposal to use multi-output trees for fast generation comes at the cost of somewhat reduced performance (Table 2).

References

Congressional Voting Records. UCI Machine Learning Repository, 1987. DOI: https://doi.org/10.24432/C5C01P.

Aad, G., Kupco, A., Chan, J., Principe Martin, M. A., Delsart, P. A., Dreyer, T., Wang, Y., Jakobs, K., Rodriguez Vera, A. M., Shaw, S., et al. AtlFast3: the next generation of fast simulation in ATLAS. Technical report, ATLAS-SIMU-2018-04-003, 2021.

Aeberhard, S. and Forina, M. Wine. UCI Machine Learning Repository, 1991. DOI: https://doi.org/10.24432/C5PC7J.

Agostinelli, S. et al. Geant4—a simulation toolkit. *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment*, 506(3):250–303, 2003. ISSN 0168-9002. doi: https://doi.org/10.1016/S0168-9002(03)01368-8.

Aha, D. Tic-Tac-Toe Endgame. UCI Machine Learning Repository, 1991. DOI: https://doi.org/10.24432/C5688J.

Albergo, M. S. and Vanden-Eijnden, E. Building normalizing flows with stochastic interpolants. In *International Conference on Learning Representations*, 2023.

Allison, J. et al. Geant4 developments and applications. *IEEE Transactions on Nuclear Science*, 53(1):270–278, 2006. doi: 10.1109/TNS.2006.869826.

Allison, J. et al. Recent developments in Geant4. *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment*, 835:186–225, 2016. ISSN 0168-9002. doi: https://doi.org/10.1016/j.nima.2016.06.125.

Amram, O. and Pedro, K. CaloDiffusion with GLaM for High Fidelity Calorimeter Simulation. *arXiv:2308.03876*, 2023.

Bhatt, R. Planning Relax. UCI Machine Learning Repository, 2012. DOI: https://doi.org/10.24432/C5T023.

Bohanec, M. Car Evaluation. UCI Machine Learning Repository, 1997. DOI: https://doi.org/10.24432/C5JP48.

Brooks, T., Pope, D., and Marcolini, M. Airfoil Self-Noise. UCI Machine Learning Repository, 2014. DOI: https://doi.org/10.24432/C5VW2C.

Buckley, M. R., Krause, C., Pang, I., and Shih, D. Inductive CaloFlow. *arXiv:2305.11934*, 2023.

Buhmann, E., Diefenbacher, S., Eren, E., Gaede, F., Kasieczka, G., Korol, A., Korcari, W., Krüger, K., and McKeown, P. CaloClouds: Fast geometry-independent highly-granular calorimeter simulation. *JINST*, 18(11): P11025, 2023. doi: 10.1088/1748-0221/18/11/P11025.

Charytanowicz, M., Niewczas, J., Kulczycki, P., Kowalski, P., and Lukasik, S. seeds. UCI Machine Learning Repository, 2012. DOI: https://doi.org/10.24432/C5H30K.

Chen, R. T. Q., Rubanova, Y., Bettencourt, J., and Duvenaud, D. K. Neural ordinary differential equations. In *Advances in Neural Information Processing Systems*, volume 31, 2018.

Chen, T. and Guestrin, C. XGBoost: A Scalable Tree Boosting System. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2016. ISBN 9781450342322.

Cortez, P., Cerdeira, A., Almeida, F., Matos, T., and Reis, J. Wine Quality. UCI Machine Learning Repository, 2009. DOI: https://doi.org/10.24432/C56S3T.

Cresswell, J. C., Ross, B. L., Loaiza-Ganem, G., Reyes-Gonzalez, H., Letizia, M., and Caterini, A. L. CaloMan: Fast generation of calorimeter showers with density estimation on learned manifolds. *Machine Learning and the Physical Sciences Workshop at NeurIPS 2022*, 2022.

Deterding, D., Niranjan, M., and Robinson, T. Connectionist Bench (Vowel Recognition - Deterding Data). UCI Machine Learning Repository. DOI: https://doi.org/10.24432/C58P4S.

Dias, D., Peres, S., and Bscaro, H. Libras Movement. UCI Machine Learning Repository, 2009. DOI: https://doi.org/10.24432/C5GC82.

Diefenbacher, S., Eren, E., Gaede, F., Kasieczka, G., Krause, C., Shekhzadeh, I., and Shih, D. L2LFlows: Generating high-fidelity 3D calorimeter images. *JINST*, 18(10): P10017, 2023. doi: 10.1088/1748-0221/18/10/P10017.

Ernst, F., Favaro, L., Krause, C., Plehn, T., and Shih, D. Normalizing Flows for High-Dimensional Detector Simulations. *arXiv:2312.09290*, 2023.

Faucci Giannelli, M. and Zhang, R. CaloShowerGAN, a Generative Adversarial Networks model for fast calorimeter shower simulation. *arXiv:2309.06515*, 2023.

Faucci Giannelli, M., Kasieczka, G., Krause, C., Nachman, B., Salamani, D., Shih, D., and Zaborowska, A. Fast Calorimeter Simulation Challenge 2022, 2022. URL https://calochallenge.github.io/homepage/.

Favaro, L., Ore, A., Schweitzer, S. P., and Plehn, T. Calodream - detector response emulation via attentive flow matching. *arXiv:2405.09629*, 2024.

Fisher, R. A. Iris. UCI Machine Learning Repository, 1988. DOI: https://doi.org/10.24432/C56C76.

Flamary, R., Courty, N., Gramfort, A., Alaya, M. Z., Boisbunon, A., Chambon, S., Chapel, L., Corenflos, A., Fatras, K., Fournier, N., Gautheron, L., Gayraud, N. T., Janati, H., Rakotomamonjy, A., Redko, I., Rolet, A., Schutz, A., Seguy, V., Sutherland, D. J., Tavenard, R., Tong, A., and Vayer, T. POT: Python Optimal Transport. *Journal of Machine Learning Research*, 22(78):1–8, 2021.

Freund, Y. and Schapire, R. E. A desicion-theoretic generalization of on-line learning and an application to boosting. In *Computational Learning Theory*, pp. 23–37, Berlin, Heidelberg, 1995. Springer Berlin Heidelberg. ISBN 978-3-540-49195-8.

Friedman, J., Hastie, T., and Tibshirani, R. Additive logistic regression: a statistical view of boosting (With discussion and a rejoinder by the authors). *The Annals of Statistics*, 28(2):337 – 407, 2000. doi: 10.1214/aos/1016218223.

Friedman, J. H. Greedy function approximation: A gradient boosting machine. *The Annals of Statistics*, 29(5), 2001. doi: 10.1214/aos/1013203451.

German, B. Glass Identification. UCI Machine Learning Repository, 1987. DOI: https://doi.org/10.24432/C5WW2P.

Gerritsma, J., Onnink, R., and Versluis, A. Yacht Hydrodynamics. UCI Machine Learning Repository, 2013. DOI: https://doi.org/10.24432/C5XG7R.

Goodfellow, I., Pouget-Abadie, J., Mirza, M., Xu, B., Warde-Farley, D., Ozair, S., Courville, A., and Bengio, Y. Generative adversarial nets. In *Advances in Neural Information Processing Systems*, 2014.

Gorishniy, Y., Rubachev, I., Khrulkov, V., and Babenko, A. Revisiting deep learning models for tabular data. In *Advances in Neural Information Processing Systems*, volume 34, pp. 18932–18943, 2021.

Gorishniy, Y., Rubachev, I., Kartashev, N., Shlenskii, D., Kotelnikov, A., and Babenko, A. TabR: Unlocking the power of retrieval-augmented tabular deep learning. *arXiv:2307.14338*, 2023.

Grinsztajn, L., Oyallon, E., and Varoquaux, G. Why do tree-based models still outperform deep learning on typical tabular data? In *Advances in Neural Information Processing Systems*, volume 35, 2022.

He, K., Zhang, X., Ren, S., and Sun, J. Deep residual learning for image recognition. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, June 2016.

Ho, J., Jain, A., and Abbeel, P. Denoising diffusion probabilistic models. *Advances in Neural Information Processing Systems*, 33:6840–6851, 2020.

Ho, T. K. Random decision forests. In *Proceedings of 3rd International Conference on Document Analysis and Recognition*, volume 1, pp. 278–282 vol.1, 1995. doi: 10.1109/ICDAR.1995.598994.

Hochreiter, S. and Schmidhuber, J. Long short-term memory. *Neural Computation*, 9(8):1735–1780, 1997. doi: 10.1162/neco.1997.9.8.1735.

Hyvärinen, A. Estimation of non-normalized statistical models by score matching. *Journal of Machine Learning Research*, 6(24):695–709, 2005.

Iosipoi, L. and Vakhrushev, A. Sketchboost: Fast gradient boosted decision tree for multioutput problems. In *Advances in Neural Information Processing Systems*, 2022.

Joe, H. *Dependence Modeling with Copulas*. Chapman & Hall/CRC Monographs on Statistics & Applied Probability. Taylor & Francis, 2014. ISBN 9781466583221.

Jolicoeur-Martineau, A., Fatras, K., and Kachman, T. Generating and imputing tabular data via diffusion and flow-based gradient-boosted trees. In *Proceedings of The 27th International Conference on Artificial Intelligence and Statistics*, 2024.

Käch, B. and Melzer-Pellmann, I. Attention to Mean-Fields for Particle Cloud Generation. *arXiv:2305.15254*, 2023.

Kelly, M., Longjohn, R., and Nottingham, K. The UCI Machine Learning Repository. https://archive.ics.uci.edu. Accessed: 2023-12-09.

Kim, J., Lee, C., and Park, N. STaSy: Score-based Tabular data Synthesis. In *The Eleventh International Conference on Learning Representations*, 2023.

Kingma, D. P. and Welling, M. Auto-encoding Variational Bayes. *ICLR*, 2014.

Kipf, T. N. and Welling, M. Semi-Supervised Classification with Graph Convolutional Networks. In *International Conference on Learning Representations*, 2016.

Kobylianskii, D., Soybelman, N., Dreyer, E., and Gross, E. CaloGraph: Graph-based diffusion model for fast shower generation in calorimeters with irregular geometry. *arXiv:2402.11575*, 2024.

Koklu, M. and Ozkan, I. A. Multiclass classification of dry beans using computer vision and machine learning techniques. *Computers and Electronics in Agriculture*, 174:105507, 2020.

Kotelnikov, A., Baranchuk, D., Rubachev, I., and Babenko, A. TabDDPM: Modelling Tabular Data with Diffusion Models. In *Proceedings of the 40th International Conference on Machine Learning*, volume 202 of *Proceedings of Machine Learning Research*, pp. 17564–17579. PMLR, 23–29 Jul 2023.

Krause, C. and Shih, D. CaloFlow: Fast and Accurate Generation of Calorimeter Showers with Normalizing Flows. *arXiv:2106.05285*, 2021a.

Krause, C. and Shih, D. CaloFlow II: Even Faster and Still Accurate Generation of Calorimeter Showers with Normalizing Flows. *arXiv:2110.11377*, 2021b.

Krizhevsky, A., Sutskever, I., and Hinton, G. E. Imagenet classification with deep convolutional neural networks. In *Advances in Neural Information Processing Systems*, volume 25, 2012.

Lipman, Y., Chen, R. T. Q., Ben-Hamu, H., Nickel, M., and Le, M. Flow matching for generative modeling. In *International Conference on Learning Representations*, 2023.

Little, M. Parkinsons. UCI Machine Learning Repository, 2008. DOI: https://doi.org/10.24432/C59C74.

Liu, X., Gong, C., and Liu, Q. Flow straight and fast: Learning to generate and transfer data with rectified flow. In *International Conference on Learning Representations*, 2023.

Loaiza-Ganem, G., Ross, B. L., Hosseinzadeh, R., Caterini, A. L., and Cresswell, J. C. Deep generative models through the lens of the manifold hypothesis: A survey and new connections. *arXiv:2404.02954*, 2024.

Lucas, D., Klein, R., Tannahill, J., Ivanova, D., Brandon, S., Domyancic, D., and Zhang, Y. Climate Model Simulation Crashes. UCI Machine Learning Repository, 2013. DOI: https://doi.org/10.24432/C5HG71.

Lundberg, S. M. and Lee, S.-I. A unified approach to interpreting model predictions. In *Advances in Neural Information Processing Systems*, volume 30, 2017.

Lundberg, S. M., Erion, G. G., and Lee, S.-I. Consistent individualized feature attribution for tree ensembles. *arXiv:1802.03888*, 2018.

Mansouri, K., Ringsted, T., Ballabio, D., Todeschini, R., and Consonni, V. QSAR biodegradation. UCI Machine Learning Repository, 2013. DOI: https://doi.org/10.24432/C5H60M.

März, A. Multi-Target XGBoostLSS Regression. *arXiv:2210.06831*, 2022.

McElfresh, D., Khandagale, S., Valverde, J., Ramakrishnan, G., Goldblum, M., White, C., et al. When do neural nets outperform boosted trees on tabular data? In *Advances in Neural Information Processing Systems*, volume 36, 2023.

Mikuni, V. and Nachman, B. Score-based generative models for calorimeter shower simulation. *Phys. Rev. D*, 106(9): 092009, 2022. doi: 10.1103/PhysRevD.106.092009.

Muzellec, B., Josse, J., Boyer, C., and Cuturi, M. Missing data imputation using optimal transport. In *Proceedings of the 37th International Conference on Machine Learning*, volume 119 of *Proceedings of Machine Learning Research*, pp. 7130–7140. PMLR, 13–18 Jul 2020.

Naeem, M. F., Oh, S. J., Uh, Y., Choi, Y., and Yoo, J. Reliable fidelity and diversity metrics for generative models. In *International Conference on Machine Learning*, pp. 7176–7185. PMLR, 2020.

Nakai, K. Ecoli. UCI Machine Learning Repository, 1996a. DOI: https://doi.org/10.24432/C5388M.

Nakai, K. Yeast. UCI Machine Learning Repository, 1996b. DOI: https://doi.org/10.24432/C5KG68.

Nock, R. and Guillame-Bert, M. Generative trees: Adversarial and copycat. In *Proceedings of the 39th International Conference on Machine Learning*, volume 162, pp. 16906–16951, 2022.

Nock, R. and Guillame-Bert, M. Generative forests. *arXiv:2308.03648*, 2023.

Pace, R. K. and Barry, R. Sparse spatial autoregressions. *Statistics & Probability Letters*, 33(3):291–297, 1997.

Paganini, M., de Oliveira, L., and Nachman, B. CaloGAN: Simulating 3D high energy particle showers in multilayer electromagnetic calorimeters with generative adversarial networks. *Phys. Rev. D*, 97:014021, Jan 2018. doi: 10.1103/PhysRevD.97.014021.

Patki, N., Wedge, R., and Veeramachaneni, K. The synthetic data vault. In *2016 IEEE International Conference on Data Science and Advanced Analytics*, pp. 399–410, 2016. doi: 10.1109/DSAA.2016.49.

Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V., Vanderplas, J., Passos, A., Cournapeau, D., Brucher, M., Perrot, M., and Duchesnay, E. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

Scham, M. A. W., Krücker, D., Käch, B., and Borras, K. DeepTreeGAN: Fast Generation of High Dimensional Point Clouds. *arXiv:2311.12616*, 2023.

Schmid, L., Gerharz, A., Groll, A., and Pauly, M. Tree-based ensembles for multi-output regression: Comparing multivariate approaches with separate univariate ones. *Computational Statistics & Data Analysis*, 179:107628, 2023. ISSN 0167-9473. doi: 10.1016/j.csda.2022.107628.

Schnake, S., Krücker, D., and Borras, K. CaloPoint-Flow II Generating Calorimeter Showers as Point Clouds. *arXiv:2403.15782*, 2024.

Sejnowski, T. and Gorman, P. R. Connectionist Bench (Sonar, Mines vs. Rocks). UCI Machine Learning Repository. DOI: https://doi.org/10.24432/C5T01Q.

Shapley, L. S. Notes on the n-person game—ii: The value of an n-person game. 1951.

Sigillito, V. G., Wing, S. P., Hutton, L. V., and Baker, K. B. Ionosphere. UCI Machine Learning Repository, 1989. DOI: https://doi.org/10.24432/C5W01B.

Sklar, M. *Fonctions de Répartition À N Dimensions Et Leurs Marges*. Université Paris 8, 1959.

Song, Y. and Ermon, S. Generative Modeling by Estimating Gradients of the Data Distribution. In *Advances in Neural Information Processing Systems*, volume 32, 2019.

Song, Y., Sohl-Dickstein, J., Kingma, D. P., Kumar, A., Ermon, S., and Poole, B. Score-based generative modeling through stochastic differential equations. In *International Conference on Learning Representations*, 2021.

Stein, G., Cresswell, J., Hosseinzadeh, R., Sui, Y., Ross, B., Villecroze, V., Liu, Z., Caterini, A. L., Taylor, E., and Loaiza-Ganem, G. Exposing flaws of generative model evaluation metrics and their unfair treatment of diffusion models. In *Advances in Neural Information Processing Systems*, volume 36, pp. 3732–3784, 2023.

Tong, A., Fatras, K., Malkin, N., Huguet, G., Zhang, Y., Rector-Brooks, J., Wolf, G., and Bengio, Y. Improving and generalizing flow-based generative models with minibatch optimal transport. *Transactions on Machine Learning Research*, 2024. ISSN 2835-8856.

van Buuren, S. *Flexible Imputation of Missing Data*. Chapman & Hall/CRC Interdisciplinary Statistics. CRC Press, 2nd edition, 2018. ISBN 9780429960352.

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, Ł., and Polosukhin, I. Attention is all you need. In *Advances in Neural Information Processing Systems*, volume 30, 2017.

Vincent, P. A connection between score matching and denoising autoencoders. *Neural Computation*, 23(7):1661–1674, 2011.

Wolberg, W., Mangasarian, O., Street, N., and Street, W. Breast Cancer Wisconsin (Diagnostic). UCI Machine Learning Repository, 1995. DOI: https://doi.org/10.24432/C5DW2B.

Xu, L., Skoularidou, M., Cuesta-Infante, A., and Veeramachaneni, K. Modeling Tabular data using Conditional GAN. In *Advances in Neural Information Processing Systems*, volume 32, 2019.

Yeh, I.-C. Concrete Compressive Strength. UCI Machine Learning Repository, 2007. DOI: https://doi.org/10.24432/C5PK67.

Yeh, I.-C. Blood Transfusion Service Center. UCI Machine Learning Repository, 2008. DOI: https://doi.org/10.24432/C5GS39.

Yeh, I.-C. Concrete Slump Test. UCI Machine Learning Repository, 2009. DOI: https://doi.org/10.24432/C5FG7D.

Ying, Z., Xu, Z., Li, Z., Wang, W., and Meng, C. MT-GBM: A multi-task gradient boosting machine with shared decision trees. *arXiv:2201.06239*, 2022.

Zhang, Z. and Jung, C. GBDT-MO: Gradient-Boosted Decision Trees for Multiple Outputs. *IEEE Transactions on Neural Networks and Learning Systems*, 32(7):3156–3167, 2021. doi: 10.1109/TNNLS.2020.3009776.

Zhao, Z., Kunar, A., Birke, R., Van der Scheer, H., and Chen, L. Y. CTAB-GAN+: Enhancing Tabular Data Synthesis. *Frontiers in Big Data*, 6, 2024. ISSN 2624-909X. doi: 10.3389/fdata.2023.1296508.

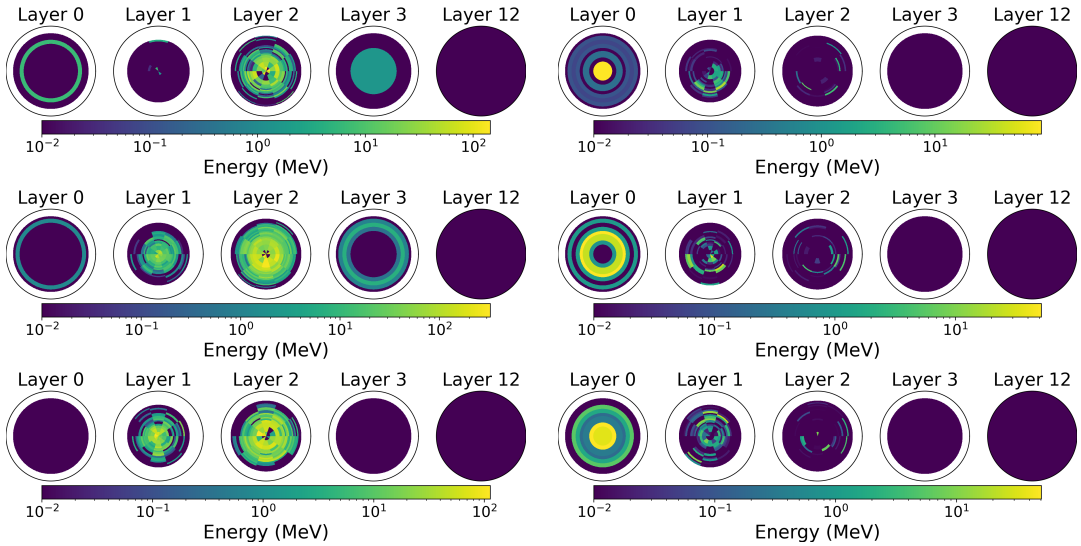


Figure 6: Individual showers shown as energy deposited per voxel for the Photons test dataset (left), and generated by CaloForest (right). Note the nested cylindrical geometry of voxels which is inconsistent across layers, meaning the data must be treated as tabular, rather than as images.

A. Fast Calorimeter Simulation

Particle accelerator experiments in high energy physics utilise several components in their detectors to measure properties of particles created in collisions. Calorimeters are one component that measure the energy of particles. Upon entering the calorimeter, the incident particle begins interacting with the material of the calorimeter and progressively deposits its energy. The interactions form a branching tree-like structure called a shower. Energy deposits are measured in an array of voxels allowing the 3d reconstruction of shower shape. Since nature is inherently probabilistic, a given incident particle with fixed energy gives rise to a probability distribution of possible showers.

Physicists desire to sample from these distributions as one component of detector simulation. By simulating detector responses using known theory, physicists can define their prior for what is expected to be measured when the actual experiment is run. Measured deviations from the prior expectation may indicate new physics, leading to a deeper understanding of nature.

However, sampling calorimeter showers using precise simulation of physical processes from first-principles is incredibly slow. Currently, simulations at the largest particle accelerator, the Large Hadron Collider (LHC), are done with Geant4 (Agostinelli et al., 2003; Allison et al., 2006; 2016), which is CPU-based and can take upwards of ten minutes per shower. Since billions of simulations are needed to provide accurate background statistics, the computational burden of exact simulation is immense.

Generative modelling heralds a solution by directly generating showers using surrogate models instead of simulating them from first-principles. The first method in this line of research used GANs (Paganini et al., 2018), eventually leading to actual deployment of GAN-based generators in the experimental pipeline of the LHC (Aad et al., 2021). Following work explored other deep generative techniques like normalizing flows (Krause & Shih, 2021a;b).

The initial success of these methods at reducing simulation time, while accurately representing the distribution of showers, led to the public release of large-scale training datasets and a call to the community to explore new methods in the Fast Calorimeter Simulation Challenge (Faucci Giannelli et al., 2022). The four datasets represent different types of particles incident on the calorimeter. The datasets represent the calorimeters with a cylindrical pattern of voxels, and each datapoint's features represent the energy deposited in one voxel. This allows the visualization of individual showers (Figure 6), as well as averages across the dataset (Figure 7).

Early submissions to the Challenge branched out to test score-based methods (Mikuni & Nachman, 2022) and models that learned the low-dimensional structure of showers (Cresswell et al., 2022). Overall, submissions can be classified by the generative modelling paradigm they build off of, with GANs (Faucci Giannelli & Zhang, 2023; Käch & Melzer-Pellmann,

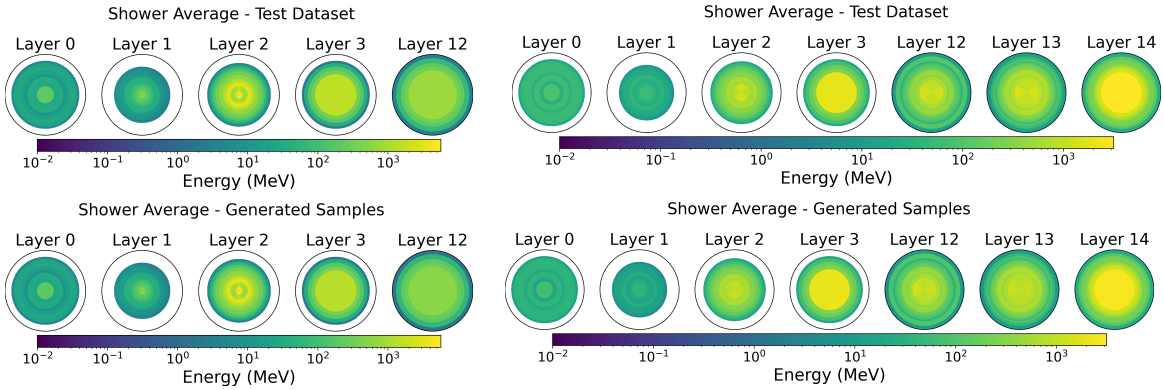


Figure 7: Average deposited energy per voxel for Photons (left) and Pions (right) on the test dataset (top), and samples generated conditionally using the test set class distribution (bottom).

2023; Scham et al., 2023), normalizing flows (Diefenbacher et al., 2023; Buckley et al., 2023; Ernst et al., 2023; Schnake et al., 2024), diffusion models (Amram & Pedro, 2023; Buhmann et al., 2023; Kobylianskii et al., 2024), and conditional flow matching (Favaro et al., 2024) being popular choices.

Notably, every prior submission to the Fast Calorimeter Simulation Challenge uses deep neural networks as function approximators. This is despite the need for GPU resources to train and generate with NNs, whereas existing scientific computing infrastructure for shower simulation is largely CPU-based. Our method, CaloForest, provides an alternative, as it is the only attempt to use tree-based approximators for the Challenge's large-scale tabular datasets.

A.1. Evaluation Metrics

Since the generated showers are meant to be used in actual scientific experiments, custom evaluation metrics have been defined for the Challenge using domain knowledge.

First, computational resources are important (hence "Fast" in the Challenge's title). Training should be accomplished with as little time and memory as possible, but the most important resource metric is shower generation time, as billions of generated showers will be needed in practice. Hence, we track the training time, generation time, and peak memory usage during training. These results are given in Section 4.3.

Second, generated showers must accurately represent the actual distribution of showers predicted by theory. Calculating this distribution in closed form from theory is not feasible, so instead the ground truth is taken from theory-based simulations using Geant4. A test dataset of showers generated in the same way as the training data is provided with each of the Challenge's datasets. Using domain knowledge, physicists defined high-level features from voxel-level information. The one-dimensional distributions of each feature can be compared between the test set and a generated set using the χ^2 separation power between histograms which is defined as

$$\chi^2(h_1, h_2) = \frac{1}{2} \sum_i \frac{(h_{1,i} - h_{2,1})^2}{h_{1,i} + h_{2,i}}, \tag{7}$$

where $h_{j,i}$ is the fraction of all datapoints falling into bin i of histogram j, such that $\sum_i h_{j,i} = 1$. The metric is normalized such that $\chi^2(h_1, h_2) = 0$ if and only if the histograms are the same, $h_1 = h_2$, whereas when the histograms have no overlap $\chi^2(h_1, h_2) = 1$. The high-level features denote the ratio of deposited energy to incident energy, the total deposited energy in each layer of the calorimeter, the center of energy in angular directions η and ϕ per layer, and the width of the center of energy in angular directions per layer. Example χ^2 metrics are given in Table 3 with example histogram plots in Figures 5 and 8, while the complete lists of metrics are shown below in Tables 4 and 5.

Third, a binary classifier is trained to distinguish generated showers from the test set using the architecture and training details provided by the Challenge (Faucci Giannelli et al., 2022). The ROCAUC of the classifier on a balanced, held-out set of samples should be as low as possible, indicating that generated samples are indistinguishable from test datapoints. We present the ROCAUC metrics in Table 3.

Table 4: Photons dataset histogram χ^2 separation powers in domain expert features. L denotes layer. CE is the center of energy. Lower is better.

FEATURE	CaloMan	CaloForest
$E_{\text{dep}}/E_{\text{inc}}$	0.0020	0.0778
E_{dep}, L0	0.00005	0.0033
E_{dep}, L1	0.00008	0.0036
E_{dep}, L2	0.00002	0.0031
E_{dep}, L3	0.00001	0.0018
E_{dep}, L12	0.00002	0.0037
CE in η, L1	0.0462	0.0056
CE in η, L2	0.0419	0.0014
CE in ϕ, L1	0.0394	0.0029
CE in ϕ, L2	0.0367	0.0017
Width in CE in η, L1	0.0366	0.0241
Width in CE in η, L2	0.0696	0.0108
Width in CE in ϕ, L1	0.0865	0.0228
Width in CE in ϕ, L2	0.0649	0.0097

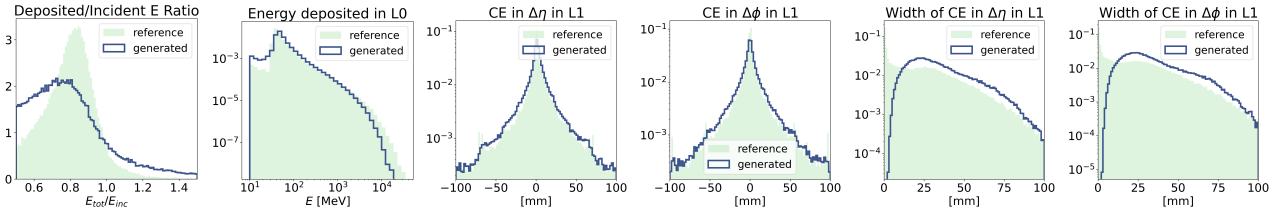


Figure 8: Histograms of high-level features comparing generated Pions samples to the test set. Note the log scale of the y-axis for all but the first plot.

A.2. Extended Results

Here we present complete information on the histogram evaluation metrics for the Challenge that are obtained with our method CaloForest. Tables 4 and 5 show the χ^2 separation powers for histograms of the generated and test set samples. Compared to a NN-based approach designed for the challenge called CaloMan (Cresswell et al., 2022), CaloForest better captures the distribution of Centers of Energy and their Widths. CaloMan was designed with a separate module to predict the deposited energy in each layer, and thus has better performance in those metrics.

We also add feature histogram plots for the Pions dataset in Figure 8 to complement the Photons results shown in the main text (Figure 5).

As mentioned above, average per-voxel energy deposits are shown for the models trained on both datasets in Figure 7 where it is clear that these distributions are learned almost perfectly.

B. Re-engineering the ForestDiffusion and ForestFlow Implementation

In this appendix we continue our analysis of the original implementation of ForestDiffusion and ForestFlow training (Algorithm 1) provided by (Jolicoeur-Martineau et al., 2024), and present a summary of our improvements as a unified implementation. We further consider how to optimize data generation.

First, we comment on the pros and cons of parallelization. Most of the memory issues experienced when using ForestDiffusion and ForestFlow are a result of training many XGBoost ensembles in parallel. Parallelization using multiprocessing requires copying data arrays to worker processes, so one may ask about alternatives. Apart from multiprocessing, parallelism in Python can also be achieved through multithreading. This avoids spawning new processes with their own memory spaces and can allow threads to share memory in the main process, however, due to the Python global interpreter lock (GIL), multithreading can only be done on tasks that release the GIL while running. In fact, calls to XGBoost training do release the GIL, as XGBoost runs native C++ code, so multithreading is a potential solution for ForestDiffusion and ForestFlow.

Table 5: Pions dataset histogram χ^2 separation powers in domain expert features. L denotes layer. CE is the center of energy. Lower is better.

FEATURE	CaloMan	CaloForest
$E_{\text{dep}}/E_{\text{inc}}$	0.0404	0.0625
E_{dep}, L0	0.0002	0.0384
E_{dep}, L1	0.0347	0.1440
E_{dep}, L2	0.0052	0.0532
E_{dep}, L3	0.0001	0.0178
E_{dep}, L12	0.0008	0.0046
E_{dep}, L13	0.0001	0.0102
E_{dep}, L14	0.0002	0.0085
CE in η, L1	0.0477	0.0268
CE in η, L2	0.0808	0.0168
CE in η, L12	0.0477	0.0641
CE in η, L13	0.0808	0.1377
CE in ϕ, L1	0.0282	0.0266
CE in ϕ, L2	0.0240	0.0155
CE in ϕ, L12	0.0282	0.0573
CE in ϕ, L13	0.0240	0.1203
Width in CE in η, L1	0.2380	0.1935
Width in CE in η, L2	0.2074	0.1121
Width in CE in η, L12	0.2380	0.1758
Width in CE in η, L13	0.2074	0.2384
Width in CE in ϕ, L1	0.2183	0.1978
Width in CE in ϕ, L2	0.2067	0.1141
Width in CE in ϕ, L12	0.2183	0.1788
Width in CE in ϕ, L13	0.2067	0.2399

However, in our preliminary tests we found that multithreading was more prone to memory not being properly released, causing increased usage over training. We were not able to find a definitive reason for this but we suspect that the Python garbage collector in a multithreaded process does not effectively free up used memory. On the other hand, we observed that multiprocessing is very effective for releasing all used memory when the corresponding job is completed.

Alternatively, one may wonder why creating parallel jobs is necessary at all when a single XGBoost training job can make use of multiple CPUs. However, XGBoost is not perfectly efficient in its use of additional CPUs, especially on small datasets like those from Table 6; training time is reduced by less than 50% when two CPUs are used instead of one, and this efficiency becomes even worse as more CPUs are provided. Hence, there is a tradeoff between speed and memory when training many XGBoost ensembles in parallel: assigning all CPUs to a single job uses the least memory but can be slow, whereas assigning one CPU to N training jobs is much faster, while using roughly N times as much memory. Figure 9 demonstrates this by training our implementation of ForestFlow on a dataset with $n = 1000$, $p = 10$, and $n_y = 10$ for various assignments of CPUs per job (cf. Figures 3 and 4). On our machine with 40 CPUs, we set the number of CPUs assigned to each job to the values $\{1, 2, 4, 10, 20, 40\}$ (and correspondingly set the number of parallel jobs to $\{40, 20, 10, 4, 2, 1\}$). Generally, the number of parallel jobs times the number of CPUs assigned per job should not exceed the number of CPUs available in total, otherwise thread contention can degrade performance. When memory is a limitation, Figure 9 shows that assigning a few CPUs per job and reducing the number of jobs can greatly decrease peak memory requirements at a marginal increase in training time. Hence, we used two CPUs per job when training on large-scale calorimeter data.

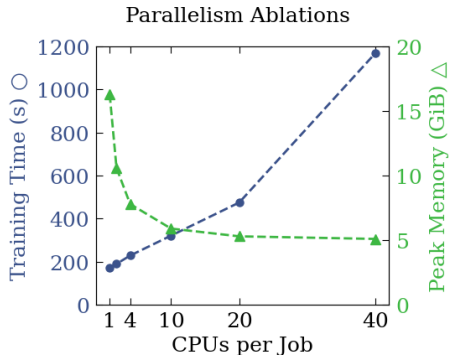


Figure 9: XGBoost is not efficient with multiple CPUs, especially for small datasets. Running single-CPU jobs in parallel is the most time-efficient method as long as adequate memory is available.

For the sake of our resource benchmarking across methods, we always use multiprocessing, and assign one CPU per worker, with the number of workers equal to the available CPUs.

B.1. Continued Analysis and Improvement of the Implementation

In Section 3.3 we began our analysis with the most impactful issues and solutions. We recommended to avoid creating large arrays in memory and instead create slices on-the-fly as needed within parallel loops (Issue 1). We found that improper use of multiprocessing could lead to excessive duplication of arrays in shared memory that could not be freed by the system, and explained how to properly share an array across processes (Issue 2). Finally, we recommended to write XGBoost models to disk as their training completed to prevent them piling up in active memory (Issue 3). These three issues and solutions accounted for the vast majority of memory improvements we observed and explained the three problematic behaviours pointed out in Section 3.1. Nevertheless, we pick up where we left off and present several additional improvements that further optimize memory usage and runtime while adhering to engineering best-practices.

Issue 4: Since worker processes access data saved to shared memory, the main process does not need to occupy memory by holding on to its copy of `X0`, `X1`, and `Z_tr`. These costly array objects are merely used as keys for Joblib to identify the arrays in shared memory.
Solution 4: Explicitly save the arrays in shared memory as memory-mapped files, delete them from the main process, and retain only a reference which can be passed to worker processes.
Benefit 4: The `X0`, `X1`, and `Z_tr` objects can be freed from the main process, amounting to **144 GiB** for the Pions dataset. Technically, we save memory-mapped files on a disk instead of a RAM disk. While a RAM disk occupies RAM space, saving to an actual disk does not, which leaves more available memory during training. Nonetheless, this does not cause slow downs from disk I/O. When a file is saved to a disk, it is first saved to cache memory (part of RAM). When the file is accessed again (potentially by a different process), if it is already in cache, the file in cache is reused. Unlike in-use RAM disk memory, this kind of cache memory can be freed upon memory pressure as it merely serves as cache for a disk.

Issue 4: Improvement

```python
import tempfile, os
from joblib import dump, load
# Create memmap files
temp_folder = tempfile.mkdtemp()
def create_memmap(array, file_name):
  file = os.path.join(temp_folder, file_name)
  dump(array, file)
  return load(file, mmap_mode='r')
X0_mmap = create_memmap(X0, "X0.mmap")
X1_mmap = create_memmap(X1, "X1.mmap")
Z_mmap = create_memmap(Z_tr, "Z.mmap")
# Free memory of X0, X1, Z_tr
del X0, X1, Z_tr
```

Issue 5: Using `n_y` Boolean masks across the duplicated dataset to index each class's data requires `n*K*n_y` bytes, since the `numpy.bool` datatype uses one byte, not one bit. Moreover, indexing a Numpy array in this way creates a copy of underlying data.
Solution 5: First sort the data by class, then use Python's `slice(start, end)` function with the beginning and end indices of each class.
Benefit 5: On the Pions dataset these Boolean masks would occupy **173 MiB** of space. Our solution only requires `2*n_y` integers, and creates a view that does not copy underlying data. However, our solution does involve sorting, although the time involved is minuscule compared to the training time for hundreds of thousands of XGBoost ensembles.

Issue 5: Original

```python
# Create Boolean masks for class conditioning
mask = {} # Boolean mask for which rows of X0 have label y_i
y_uniq = np.unique(y)
for y_i in y_uniq:
  mask[y_i] = np.tile(y == y_i, K)
```

Issue 5: Improvement

```python
# Sort by label and slice for class conditioning
y_arg_sort = np.argsort(y)
y, X0 = y[y_arg_sort], X0[y_arg_sort]
```

```
y_uniq, y_counts = np.unique(y, return_counts=True)
mask = {} # Slice of X0's rows that have label y_i
csum = 0
for y_i, count in zip(y_uniq, y_counts):
  mask[y_i] = slice(csum, csum + count)
  csum += count
y_slice = {} # adjust slices for duplicated data
for y_i, sl in mask.items():
    y_slice[y_i] = slice(sl.start*K, sl.stop*K)
```

Issue 6: In XGBoost training, input data is converted to a `DMatrix`, XGBoost's native data structure, and is reformatted and cached for faster access. For example, features are converted to histograms when using `hist` training, as we do. The histogram computations are redundant across jobs since the same `X_tr_i` is used for all `p_i`.
Solution 6: Ever since XGBoost version 1.6, multiple regressors trained with the same features but different targets can be encapsulated in a single `Booster` object. When the multi-dimensional target is passed to XGBoost's `fit(X,Z)` function, XGBoost internally trains each target sequentially while using the same `DMatrix`, avoiding redundant histogram computations over `p_i`.
Benefit 6: This reduces `DMatrix` constructions and reduces histogram computations by a factor of `p`. Additionally, all ensembles over `p` for a given `n_t` and `n_y` are contained in the same `Booster` object, which, in turn, reduces the number of model files and metadata to be stored, and reduces file I/O overhead.

Issue 6: Original

```
# One Booster for each column p_i
Z_tr_i = Z_tr[mask[y_i], p_i]
model.fit(X_tr_i, Z_tr_i)
```

Issue 6: Improvement

```
# Single Booster for all columns p_i
Z_tr_i = Z_tr[mask[y_i], :]
model.fit(X_tr_i, Z_tr_i)
```

Issue 7: XGBoost internally uses `fp32` regardless of the input data type. However, `numpy.float64` is implicitly used in the original implementation.
Solution 7: Use `fp32` throughout the whole pipeline.
Benefit 7: Using lower precision throughout reduces memory usage without losing model accuracy and avoids implicit data type conversions.

Issue 7: Original

```
X0 = inp()
# loaded as numpy.float64
X1 = np.random.normal(size=X0.shape)
# default dtype is numpy.float64
```

Issue 7: Improvement

```
X0 = inp()
X0 = X0.astype(np.float32)
X1 = np.random.normal(size=X0.shape)
X1 = X1.astype(X0.dtype)
```

To summarize, our implementation making use of all our recommended changes is given below.

Our Implementation of ForestFlow Training with Single-Output Trees

```
from sklearn.preprocessing import MinMaxScaler as Scaler
import numpy as np, xgboost as xgb, tempfile, os
from joblib import delayed, Parallel, dump, load

X0, y, K, n_t, xgb_kw, n_jobs = inp()
n, p = X0.shape
X0 = X0.astype(np.float32) # use XGBoost's native dtype
# Sort by label and slice for class conditioning
y_arg_sort = np.argsort(y)
y, X0 = y[y_arg_sort], X0[y_arg_sort]
y_uniq, y_counts = np.unique(y, return_counts=True)
mask = {} # Slice of X0's rows that have label y_i
csum = 0
for y_i, count in zip(y_uniq, y_counts):
  mask[y_i] = slice(csum, csum + count)
  csum += count
# Scale each class's data so that range matches noise variance
scalers = []
for y_i in y_uniq:
  scalers.append(Scaler(feature_range=(-1, 1))
  X0[mask[y_i], :] = scalers[-1].fit_transform(X0[mask[y_i], :])
# Duplicate data and generate noise
X0 = np.repeat(X0, K, axis=0)
X1 = np.random.normal(size=X0.shape).astype(X0.dtype)
y_slice = {} # adjust slices
for y_i, sl in mask.items():
    y_slice[y_i] = slice(sl.start*K, sl.stop*K)
# Create regression targets (ForestFlow)
Z_tr = X1 - X0 # regression target is constant in t, but input is not
t = np.linspace(0, 1, num=n_t)
# Create memmap files
temp_folder = tempfile.mkdtemp()
def create_memmap(array, file_name):
  file = os.path.join(temp_folder, file_name)
  dump(array, file)
  return load(file, mmap_mode='r')
X0_mmap = create_memmap(X0, "X0.mmap")
X1_mmap = create_memmap(X1, "X1.mmap")
Z_mmap = create_memmap(Z_tr, "Z.mmap")
del X0, X1, Z_tr
# Train models in triple loop over timesteps, classes, and features
def train_parallel(X0_mmap, X1_mmap, Z_mmap, t_i, y_i):
  X_tr_i = t_i*X1_mmap[y_i, :] + (1-t_i)*X0_mmap[y_i, :]
  Z_tr_i = Z_mmap[y_i, :]
  model = xgb.XGBRegressor(**xgb_kw)
  model.fit(X_tr_i, Z_tr_i) # single Booster for all columns p_i
  model.save_model(f"{model_path}.ubj") # path for t_i, y_i
Parallel(n_jobs)(
  delayed(train_parallel)(
    X0_mmap, X1_mmap, Z_mmap, t_i, y_i,
  ) for t_i in t for y_i in y_slice
)
shutil.rmtree(temp_folder) # clean up memmaps
```

For a direct comparison, we show in Figure 2 the memory usage during training using the original implementation as well as ours on the same dataset with $n = 1000$, $p = 100$, and $n_y = 10$. Our implementation does not suffer from the three undesirble behaviours noted in Section 3.1.

B.2. Analysis and Improvement of Data Generation

To this point we have focused on improving the implementation of ForestFlow training. For many applications generation speed is also a critical requirement, including hosted generative model services and our running example of calorimeter simulation for experimental particle physics. In this section we turn our attention to improving the implementation of data generation with a trained ForestFlow model, starting with a summary of the existing implementation from (Jolicoeur-Martineau et al., 2024).

First, for conditional sampling, labels are created using a multinomial distribution with probabilities equal to the relative prevalence of labels in the training set. Boolean masks are created to indicate the conditioning. Gaussian noise `X1` is sampled to seed the generation, and Euler's method over uniformly discretized timesteps is used to solve the ODE using the trained models as the vector field. In particular, a triple `for` loop is used over timesteps, classes, and features in that order.

Original Python Implementation of ForestFlow Generation

```python
import numpy as np

y, n_t, n, p, regressors = inputs()
# Sample labels for conditioning using frequencies from the training dataset
y_uniq, y_counts = np.unique(y, return_counts=True)
y_probs = y_counts / np.sum(y_counts)
y_sample = np.argmax(np.random.multinomial(1, y_probs, size=n), axis=1)
# Create Boolean masks for class-conditioning
label_y = y_uniq[y_sample]
mask = {}
for y_i in y_uniq:
  mask[y_i] = (label_y == y_i)
# Solve ODE with Euler's method starting from noise
X1 = np.random.normal(size=(n, p))
h = 1 / (n_t-1) # size of timestep
for t_i in range(1, 0, h):
  out = np.zeros(shape=X1.shape)
  for y_idx, y_i in enumerate(y_uniq):
    for p_i in range(p):
      model = regressors[t_i][y_idx][p_i]
      out[mask[y_i], p_i] = model.predict(X1[mask[y_i], :])
  X1 = X1 - h * out
X0 = X1
```

Once again, we proceed by pointing out issues, offering solutions, and quantifying the benefits.

Issue 8: XGBoost's core engine is implemented in C++, and there is hidden overhead when the Python wrapper makes a call to its C-API.
Solution 8: Reduce the number of calls to the C-API by reducing the number of distinct `Booster` objects. In our training implementation, all ensembles trained over `p` for a given `n_t` and `n_y` are contained in the same `Booster` object (See Issue 6 in Appendix B.1). Inference on this `Booster` object produces an output shape with `[n_i, p]` containing all features.
Benefit 8: A factor of `p` fewer calls to the XGBoost C-API are made, and we eliminate Python's slow `for` loop over `p`. Additionally, cache locality is utilized more aggressively by XGBoost's C++ inference implementation.

Issue 1: Original

```python
for p_i in range(p):
  model = regressors[t_i][y_idx][p_i]
  out[mask[y_i], p_i] =
    model.predict(X1[mask[y_i], :])
```

Issue 1: Improvement

```python
model = regressors[t_i][y_idx]
out[mask[y_i], :] =
 model.predict(X1[mask[y_i], :])
```

Issue 9: Slow Numpy indexing operations are used in the triple loop.
Solution 9: Conditional generation of datapoints with different `y` labels uses disjoint sets of ensembles. It is not necessary to combine all partially generated datapoints into a single array after every timestep. Instead, concatenate all rows only at the end.
Benefit 9: This eliminates writing intermediate results to non-contiguous memory `out[mask[y_i]]`. It also allows iterating over `y_i` in the outer loop which reduces Numpy indexing that creates a copy of the underlying

data. Data copying is then avoided by replacing the Boolean mask with `slice` as in Issue 5 from Appendix B.1.

Issue 2: Original

```
for t_i in range(1, 0, h):
  out = np.zeros(shape=X1.shape)
  for y_idx, y_i in enumerate(y_uniq):
    for p_i in range(p):
      model=regressors[t_i][y_idx][p_i]
      out[mask[y_i],p_i] =
      ↪ model.predict(X1[mask[y_i],:])
  X1 = X1 - h * out
X0 = X1
```

Issue 2: Improvement

```
results = []
for y_idx, y_i in enumerate(y_uniq):
  X1_i = X1[mask[y_i], :]
  for t_i in range(1, 0, h):
    model = regressors[t_i][y_idx]
    X1_i = X1_i - h*model.predict(X1_i)
  results.append(X1_i)
X0 = np.concatenate(results, axis=0)
```

To summarize, our implementation for ForestFlow generation making use of our recommended changes is given below.

Our Implementation of ForestFlow Generation with Single-Output Trees

```
import numpy as np

y, n_t, n, p, regressors = inp()
# Sample labels for conditioning using frequencies from the training dataset
y_uniq, y_counts = np.unique(y, return_counts=True)
y_probs = y_counts / np.sum(y_counts)
y_sample = np.argmax(np.random.multinomial(1, y_probs, size=n), axis=1)
label_y = y_uniq[y_sample]
# Sort by label and slice each class
label_y.sort()
y_uniq, y_counts = np.unique(label_y, return_counts=True)
mask = {}
csum = 0
for y_i, count in zip(y_uniq, y_counts):
  mask[y_i] = slice(csum, csum + count)
  csum += count
# Solve ODE with Euler's method starting from noise
X1 = np.random.normal(size=(n, p)).astype(np.float32)
h = 1 / (n_t-1) # size of timestep
results = []
for y_idx, y_i in enumerate(y_uniq):
  X1_i = X1[mask[y_i], :]
  for t_i in range(1, 0, h):
    model = regressors[t_i][y_idx]
    X1_i = X1_i - h * model.predict(X1_i)
  results.append(X1_i)
X0 = np.concatenate(results, axis=0)
```

C. Performance Improvements

In this Appendix, we further details methods to improve the generative quality or resource utilization of ForestDiffusion and ForestFlow (Jolicoeur-Martineau et al., 2024) that go beyond implementation changes. This discussion extends Section 3.4 from the main text.

C.1. Class-conditional Scalers

One advantage of XGBoost as a function approximator is its robustness to data with varying scales and distributions. This stands in stark contrast to deep NNs which require data to be carefully pre-processed for best results. While XGBoost itself is agnostic, ForestDiffusion and ForestFlow do require input data to be on the same scale as the added noise in Eq. 2 and 5. Jolicoeur-Martineau et al. (2024) achieve this by applying min-max scaling over the entire input dataset. However, when using the class-conditional variant, models are trained on disjoint sets of data belonging to each class. If the classes have distinct distributions, which is often the case, then the data subsets actually provided to the training algorithm may not be properly scaled. To rectify this, we propose class-conditional min-max scaling. This is especially beneficial on the calorimeter datasets as the classes represent particle energies increasing on an exponential scale. Class-conditional scaling

centers the data better making it more easily distinguishable as noise is added, ultimately benefiting the model performance.

C.2. Sampling with the Training Set Label Distribution

For class-conditional sampling, Jolicoeur-Martineau et al. (2024) used the relative prevalence of classes in the training set to define a multinomial distribution and sampled from it to create class labels for conditioning. We found it advantageous to directly use the empirical distribution of class labels from the training set for conditioning, especially on the distributional Wasserstein metrics. For the small datasets used in benchmarking (Table 6), multinomial sampling may lead to a skewed distribution by chance; the law of large numbers may not kick in at these sizes. This type of sampling with training set labels is also mandated in the Fast Calorimeter Simulation Challenge (Faucci Giannelli et al., 2022).

C.3. Hyperparameters

The importance of various hyperparameters can change as datasets are scaled up. On small benchmark datasets, Jolicoeur-Martineau et al. (2024) recommend setting the duplication factor K as large as possible (100 in practice) to achieve better coverage of the expectations in Eq. 1 and 6. Larger datasets, on the other hand, are more prone to having duplicates, or datapoints that convey similar information. These will naturally lead to different training examples once noise is added. For this reason, we found that K need not be so large when the dataset itself has large n, and set $K = 20$ in calorimeter experiments. This also serves to reduce the peak memory burden and computation time.

On the other hand increasing n_t directly benefits model performance as it reduces approximation errors when sampling. Still, larger n_t translates to proportionally increased training time, model size, and generation time. In balancing this tradeoff we increased n_t from 50 (Jolicoeur-Martineau et al., 2024) to 100 in calorimeter experiments.

Finally, Jolicoeur-Martineau et al. (2024) noted that their models appeared to be underfitting, even though they are massively overparameterized, and hence avoided regularization. We also observed underfitting, but mitigated it by greatly increasing the learning rate from the default 0.3 to 1.5. On the other hand, Jolicoeur-Martineau et al. (2024) use the default number of trees which is $n_{\text{tree}} = 100$, and due to the lack of regularization, most trees reach their full depth leading to enormous overparameterization with questionable benefit. We found that $n_{\text{tree}} = 20$ still led to sufficiently expressive models but greatly reduces the training time, and number of parameters for calorimeter datasets.

For benchmarking resource usage and performance in Sections 4.1 and 4.2, we still used the default values from Jolicoeur-Martineau et al. (2024) for a fair and direct comparison.

C.4. Multi-output Trees

One obvious downside of using XGBoost regressors is that they output a scalar, whereas for generative modelling we need to output a vector $\mathbf{x}$. In practice $\mathbf{x}$ is often high dimensional, and its dimension p enters multiplicatively into the number of ensembles needed ($n_t \cdot n_y \cdot p$) for ForestDiffusion and ForestFlow.

Our most significant proposal is to replace single-output trees with multi-output trees, also referred to as vector-leaf trees (Zhang & Jung, 2021; Ying et al., 2022; März, 2022; Iosipoi & Vakhrushev, 2022; Schmid et al., 2023). Simply put, each leaf node in the tree outputs a vector, and the training algorithm is modified to fit all output variables at once by maximizing the sum of losses over individual outputs. Not only does this reduce the number of ensembles we require by a factor of p, but it has the potential to increase model performance by better capturing correlations between output variables during generation. Consider how a set of p single-output trees generates a vector output. From identical inputs, each tree independently identifies the appropriate leaf node and outputs a scalar – there is no dependence between elements during generation. This is clearly not desirable for generative models, where for example images have strong correlations between nearby pixels. Multi-output trees can better represent correlations during generation since generated elements do not come from independent trees. However, our experiments have not shown that multi-output trees improve the generative quality of ForestDiffusion and ForestFlow compared to single-output trees at this time. While potentially more expressive, multi-output trees have been shown to need thousands of boosting rounds to surpass their single-output counterparts on discriminative tasks (Zhang & Jung, 2021). Due to our use of the same n_{tree} and maximum depth hyperparameters, our single-output models essentially use p times more parameters which may be the source of their better performance.

Since version 2.0.0, XGBoost has implemented multi-output trees. During our testing we identified a bug in the gain

computation in the official XGBoost codebase and reported it to the maintainers who implemented our proposed fix.[4] Hence, only XGBoost version 2.1.0 or later should be used for multi-output trees. Still, this version's implementation is not yet optimized for time and memory performance, so our measurements in Section 4.1 should be considered preliminary.

D. Experimental Details

In this Appendix we lay out the details of the experiments conducted in Section 4.

D.1. Datasets

For the resource scaling experiments in Section 4.1 we used synthetic data that was randomly generated. The input data X of size $[n, p]$ was simply drawn from a identity covariance Gaussian, while the class label y was randomly drawn from the integers $[0, n_y)$. While this data is meaningless for model performance, it gives us precise control over the dataset size for analysing resource usage. Since the correlations between features are random, unregularized XGBoost regressors will use essentially their entire available capacity in learning which gives us a good upper bound on resource usage. The dataset size parameters were set at $n = 1000$, $p = 10$, and $n_y = 10$ by default, and a single one of these three was modified at a time. We measured training time and peak memory usage for the values $n \in \{100, 300, 1000, 3000, 10000, 30000, 100000, 300000\}$, $p \in \{3, 10, 30, 100, 300\}$, and $n_y \in \{1, 3, 10, 30, 100\}$. For models that trained successfully (i.e. did not fail due to memory issues), we measure the time to generate five batches of data equal in size to the dataset a given model was trained on.

In Section 4.2 we used 27 datasets from the UCI Machine Learning Repository of tabular datasets (Kelly et al.) and from scikit-learn (Pedregosa et al., 2011) that have previously been studied (Muzellec et al., 2020; Jolicoeur-Martineau et al., 2024). These datasets are summarized in Table 6, and showcase a variety of tabular learning tasks with variation in the number of datapoints n, features p, and classes n_y, and target types. In each case, we randomly held-out 20% of the dataset as a test split and trained generative models on the remaining 80%. Categorical variables are one-hot encoded.

Each of the UCI datasets is covered by a CC BY 4.0 license, while the *iris* dataset has a BSD 3-Clause License, and *california housing* has no license.

D.2. Metrics

For a fair and direct comparison in Section 4.2, we use the same eight performance metrics for generated data as in (Jolicoeur-Martineau et al., 2024) which measure quality along four different axes: distributional distance, diversity, usefulness for training discriminative models, and usefulness for statistical inference.

Distributional Distance We measure the Wasserstein-1 distance between the generated data and either the training set ($W1_{\text{train}}$) or test set ($W1_{\text{test}}$). The Wasserstein distance quantifies similarity in distribution - smaller distance to the test set is always desirable while distance to the training set should be similar in magnitude to the distance between the training and test sets, as a much smaller distance here can indicate memorization. Generally $W1_{\text{train}}$ values are not less than the train-test distance, so we treat lower values as better. Computation of Wasserstein distances was done with the Python Optimal Transport library (Flamary et al., 2021). These metrics are omitted for the larger *bean* and *california* datasets as they scale quadratically in dataset size which is prohibitively expensive (Muzellec et al., 2020; Jolicoeur-Martineau et al., 2024).

Diversity Coverage (Naeem et al., 2020) measures to what extent the generated data covers a reference dataset, where a reference datapoint is covered if there is at least one generated datapoint in its neighbourhood. Hence, generated data must be as diverse as the reference data to achieve high Coverage. Coverage is computed as the ratio of covered points to all points (Stein et al., 2023)

$$\text{coverage}\left(\{x_i^g\}_{i=1}^n, \{x_j^r\}_{j=1}^m\right) = \frac{1}{m}\sum_{j=1}^{m} \max_{i=1,\dots,n} \mathbb{1}\left(x_i^g \in B(x_j^r, \text{NND}_k(x_j^r))\right), \tag{8}$$

where x^g are the generated datapoints, x^r are the reference datapoints, $\mathbb{1}(\cdot)$ denotes the indicator function, $B(x, r)$ denotes a ball centered at x with radius r, and $\text{NND}_k(x_j^r)$ is the nearest-neighbour distance between x_j^r and its k^{th} nearest neighbour in $\{x_j^r\}_{j=1}^m$. We use an L1 ball to compute distances as it is more suited for mixed data types typical of tabular data. k

[4] See https://github.com/dmlc/xgboost/issues/9960.

Table 6: Tabular benchmark datasets. Training dataset sizes n are 80% of the total number of datapoints. Continuous and integer targets y are treated as an additional feature.

Dataset	Citation	# Datapoints	# Features p	# Classes n_y	Target y type
airfoil self noise	(Brooks et al., 2014)	1503	6	N/A	Continuous
bean	(Koklu & Ozkan, 2020)	13611	16	7	Categorical
blood transfusion	(Yeh, 2008)	748	4	2	Binary
breast cancer diagnostic	(Wolberg et al., 1995)	569	30	2	Binary
california housing	(Pace & Barry, 1997)	20640	9	N/A	Continuous
car evaluation	(Bohanec, 1997)	1728	6	4	Categorical
climate model crashes	(Lucas et al., 2013)	540	18	2	Binary
concrete compression	(Yeh, 2007)	1030	9	N/A	Continuous
concrete slump	(Yeh, 2009)	103	8	N/A	Continuous
congressional voting	(mis, 1987)	435	16	2	Binary
connectionist bench sonar	(Sejnowski & Gorman)	208	60	2	Binary
connectionist bench vowel	(Deterding et al.)	990	10	2	Binary
ecoli	(Nakai, 1996a)	336	7	8	Categorical
glass	(German, 1987)	214	9	6	Categorical
ionosphere	(Sigillito et al., 1989)	351	33	2	Binary
iris	(Fisher, 1988)	150	4	3	Categorical
libras	(Dias et al., 2009)	360	90	15	Categorical
parkinsons	(Little, 2008)	195	22	2	Binary
planning relax	(Bhatt, 2012)	182	12	2	Binary
qsar biodegradation	(Mansouri et al., 2013)	1055	41	2	Binary
seeds	(Charytanowicz et al., 2012)	210	7	3	Categorical
tic-tac-toe	(Aha, 1991)	958	9	2	Binary
wine	(Aeberhard & Forina, 1991)	178	13	3	Categorical
wine quality red	(Cortez et al., 2009)	1599	11	N/A	Integer
wine quality white	(Cortez et al., 2009)	4898	12	N/A	Integer
yacht hydrodynamics	(Gerritsma et al., 2013)	308	7	N/A	Continuous
yeast	(Nakai, 1996b)	1484	8	10	Categorical

is chosen automatically as the smallest value such that the training data has at least 95% Coverage of the test data. We calculate the Coverage using either the training ($\text{Cov}_{\text{train}}$) or test ($\text{Cov}_{\text{test}}$) dataset as the reference. $\text{Cov}_{\text{train}}$ helps to address "mode dropping" where some parts of the training dataset are ignored, while Cov_{test} helps measure the ability to generalize with sufficient diversity. These metrics were computed for all datasets.

Usefulness for Training Discriminative Models Tabular generative models are often motivated as a way to replace or extend training data for downstream tabular discriminative models (Xu et al., 2019; Kotelnikov et al., 2023). Available training data may be considered private and not suitable for directly training a discriminative model, whereas synthetic data derived from a generative model may be more palatable. Alternatively, synthetic data may be used with the hope that it leads to better performing downstream models. Hence, we measure the usefulness of generative models by training downstream discriminative models on generated data, and evaluating discriminative performance on the test set. Performance is measured either by the F1-score for classification tasks (20 datasets), or the R^2-coefficient for regression tasks (7 datasets), where higher is better. Since these metrics are highly dependent on the type of discriminative model used, we average the performance metrics over four different methods that are commonly used for tabular discriminative modelling: linear/logistic regression, AdaBoost (Freund & Schapire, 1995), Random Forests (Ho, 1995), and, of course, XGBoost (Chen & Guestrin, 2016).

Usefulness for Statistical Inference Whereas the above metrics take a machine learning point of view in aiming to optimize the performance of a model, we can also consider a statistical point of view and measure the usefulness of synthetic data for inferring the importance of features (van Buuren, 2018). By training a linear model on either the training data or generated data we can compare the regression parameters β. If the generated data accurately represents the training data, the learned regression coefficients should be similar. If these coefficients are not similar, one might conclude from the generated data that a given feature is statistically important when the same conclusion would not be reached using the training data. The percent bias measures this difference and is defined as $P_{\text{bias}} = |\mathbb{E}\frac{\hat{\beta}-\beta}{\beta}|$ using the estimated coefficients $\hat{\beta}$ on generated data and actual coefficients β from training data, with the expectation taken over generated data. From another direction, it is desirable for confidence intervals on the estimated coefficients $\hat{\beta}$ to contain the true coefficients β. This is quantified by the coverage rate cov_{rate}, the fraction of β that are contained in the confidence intervals around $\hat{\beta}$. These metrics were computed only for the regression tasks (7 datasets). Lower is better for P_{bias}, but higher is better for cov_{rate}. Coverage rate is not to be confused with Coverage used above as a diversity metric.

D.3. Baseline Methods

In addition to comparing our approach to the original implementation of ForestDiffusion and ForestFlow in Section 4.2, we also compare our improved models to 6 popular baseline methods for tabular generative modelling, including state-of-the-art deep learning methods, as done in (Jolicoeur-Martineau et al., 2024).

GaussianCopula Many deep learning-based generative models learn a mapping between a simple distribution on latent space and the data distribution (Loaiza-Ganem et al., 2024). Generation is done by sampling from the latent distribution and mapping the sample to data space. This overarching idea hearkens back to copula methods (Sklar, 1959) which model any multivariate joint distribution by its univariate marginals along with a copula describing the dependence structure. We use Gaussian copulas (Joe, 2014) implemented by Synthetic Data Vault (SDV) (Patki et al., 2016) using default hyperparameters.

TVAE Variational autoencoders (VAE) (Kingma & Welling, 2014) learn an encoder and decoder with a low dimensional latent space through variational inference. As a typical example we use the tabular VAE (TVAE) from (Xu et al., 2019), again implemented by SDV using default hyperparameters.

CTGAN and CTAB-GAN+ Generative adversarial networks (Goodfellow et al., 2014) train a generator which produces synthetic datapoints, and a discriminator that tries to classify real and synthetic datapoints. The two networks are trained simultaneously in an adversarial manner. As a typical example we use the conditional tabular GAN (CTGAN) from (Xu et al., 2019), also as implemented by SDV. We also employ a more modern tabular GAN called CTAB-GAN+ (Zhao et al., 2024) as implemented by its authors. Both methods use default hyperparameters from their respective implementations.

StaSy and Tab-DDPM More recently score-based (Song & Ermon, 2019) and diffusion models (Ho et al., 2020; Song et al., 2021) have eclipsed VAEs and GANs for generative quality on the image modality. To represent these classes, we use STaSy (Kim et al., 2023), a score-based method, and Tab-DDPM (Kotelnikov et al., 2023) a denoising diffusion model (Ho et al., 2020) adapted for tabular settings. For the former we use hyperparameters found by (Jolicoeur-Martineau et al., 2024), and the latter uses default hyperparameters from the author's implementation.

D.4. Experimental Setup and Hyperparameters

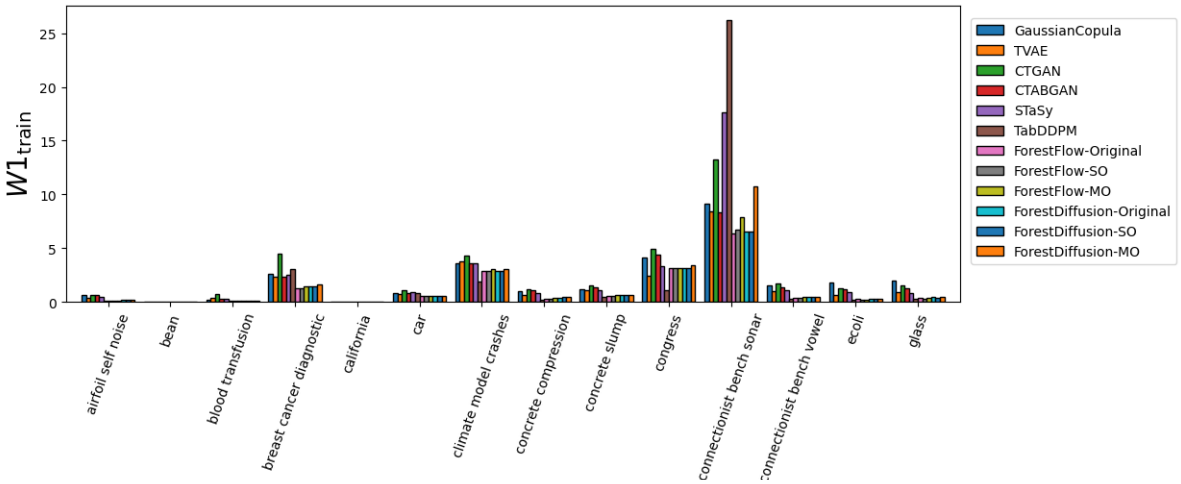
When measuring resource usage in Section 4.1 training time was clocked starting once data had been loaded and pre-processed, and stopped once all models had been trained (i.e. generation and evaluation are not included). Since almost all compute time is spent in calls to XGBoost during training, there was little difference between implementations. For memory, we monitor the used CPU memory every second (or every 10 seconds for long runs taking more than one hour), and report the peak memory burden over the entire training run. This is reasonable, as the peak memory burden determines if a job can successfully complete on a given machine.

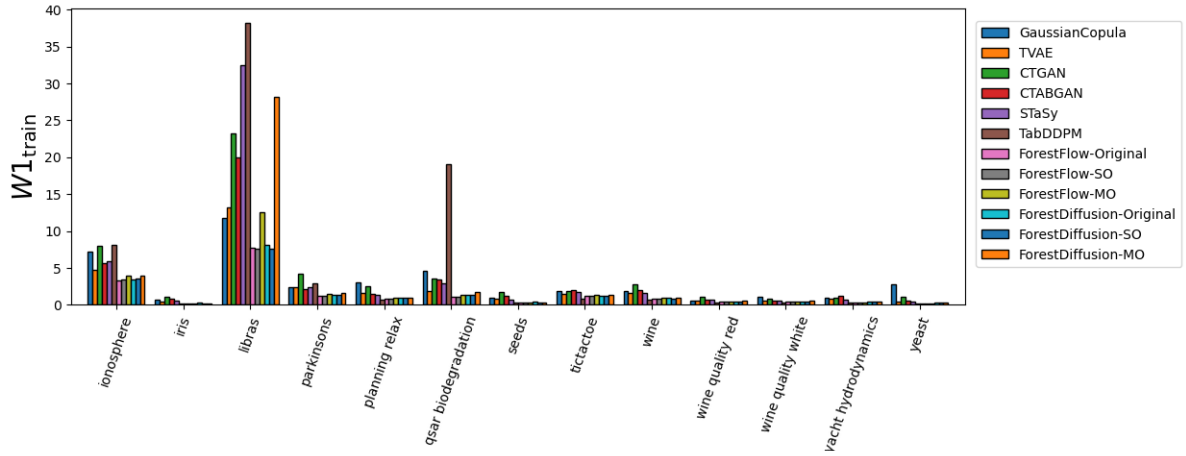
To make the most fair comparison possible, we refrained from tuning hyperparameters between our version of ForestDiffusion and ForestFlow compared to the original (Jolicoeur-Martineau et al., 2024). In particular, for the resource scaling experiments in Section 4.1, all methods use the same learning and XGBoost hyperparameters. The data was duplicated $K = 100$ times with $n_t = 50$ discrete time steps as recommended by (Jolicoeur-Martineau et al., 2024), and models were trained conditional on y whenever $n_y > 1$ (see Table 6). Computationally, 40 parallel training jobs were used (equal to the number of CPUs on our machine), with one CPU assigned to each job. For the sake of benchmarking, we did not reduce the number of parallel jobs when methods began failing due to memory issues. XGBoost hyperparameters were left at their defaults, other than L2 regularization which was set to $\lambda = 0$. Notably, this means $n_{\text{tree}} = 100$ trees were trained per ensemble of max depth 7. The same is true for our performance benchmarking in Section 4.2. The original implementation of ForestDiffusion and ForestFlow used the same hyperparameters as our implementation, with the performance differences coming from our proposed improvements: per-class scaling of data, and sampling using the distribution of labels from the training set. Our multi-output tree variant used a learning rate $\eta = 0.5$ instead of the XGBoost default $\eta = 0.3$, but this was the only parameter changed.

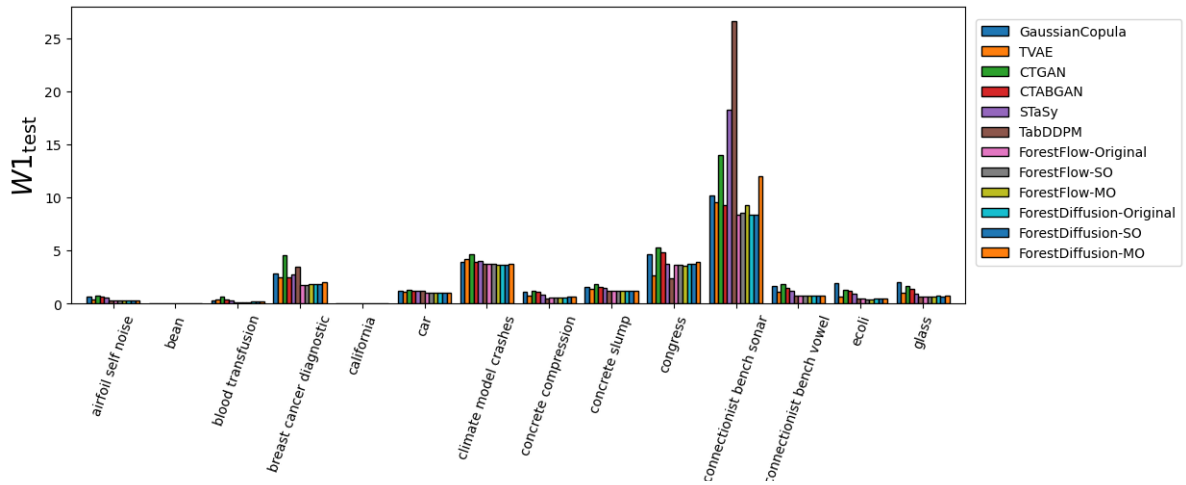
For the performance comparisons in Section 4.2, each method was trained with 3 different random seeds on each dataset, and for each training run 5 sets of data the size of the training dataset were created. Each set of data was used to compute the performance metrics independently, and the results were averaged across the 5 generations per 3 seeds. These averaged performance metrics were then used to compute the relative rankings between methods. For each metric, we computed the ranking of methods on each dataset and then took the mean and standard deviation of rankings across datasets. As discussed in Appendix D.2, not all metrics could be used for all datasets, so the averages over datasets only include applicable datasets where the metric could actually be computed.

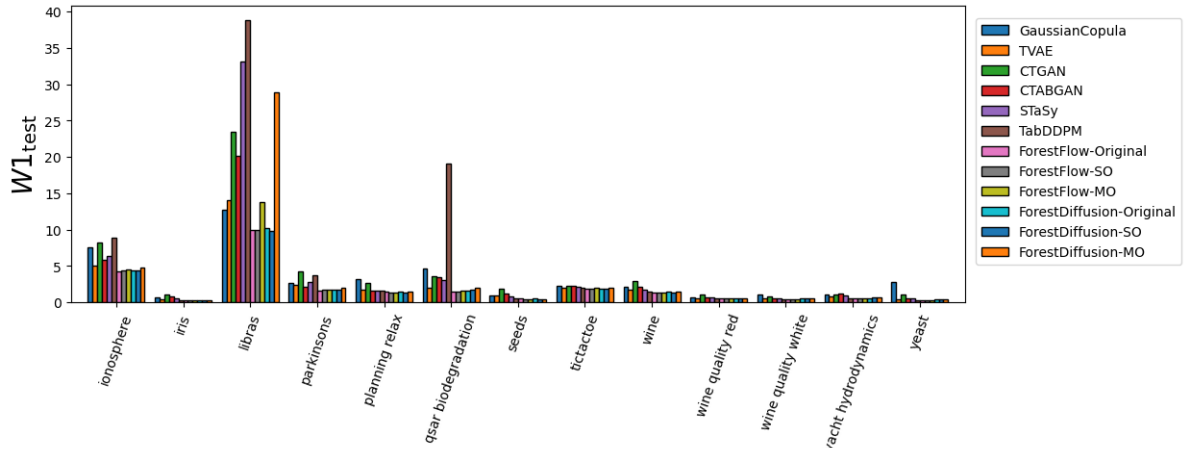
D.5. Additional Performance Benchmarking Results

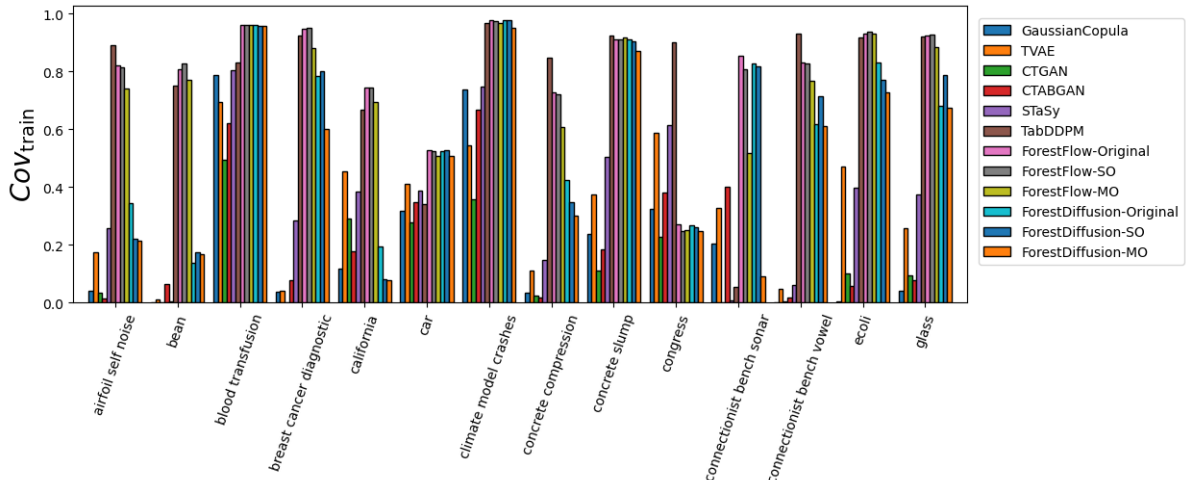
Here we complement the summarized results of Table 2 by plotting the raw metric values averaged over three seeds for each metric, method, and dataset. We remark that the *bean* and *california* datasets were not evaluated with the Wasserstein metrics due to their size. Other plots show missing information for datasets when the metric is suited for either classifcation or regression tasks, but not both.

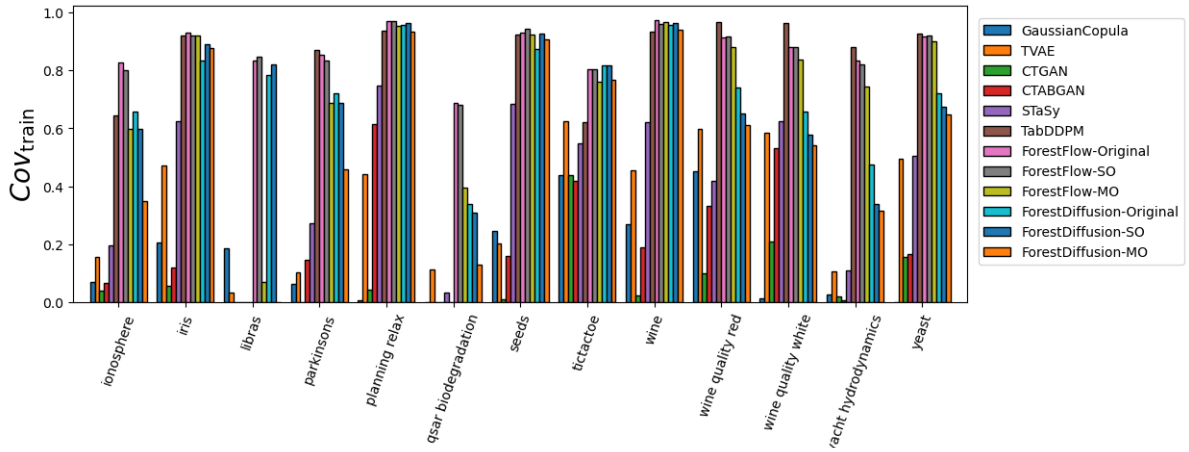


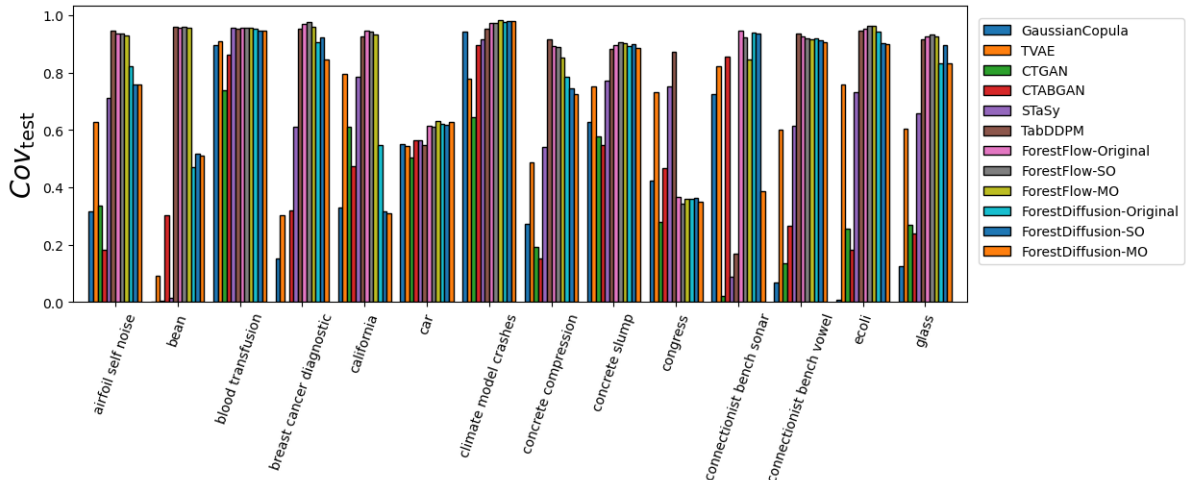

$W1_{train}$
GaussianCopula
TVAE
CTGAN
CTABGAN
STaSy
TabDDPM
ForestFlow-Original
ForestFlow-SO
ForestFlow-MO
ForestDiffusion-Original
ForestDiffusion-SO
ForestDiffusion-MO
ionosphere
iris
libras
parkinsons
planning relax
qsar biodegradation
seeds
tictactoe
wine
wine quality red
wine quality white
yacht hydrodynamics
yeast

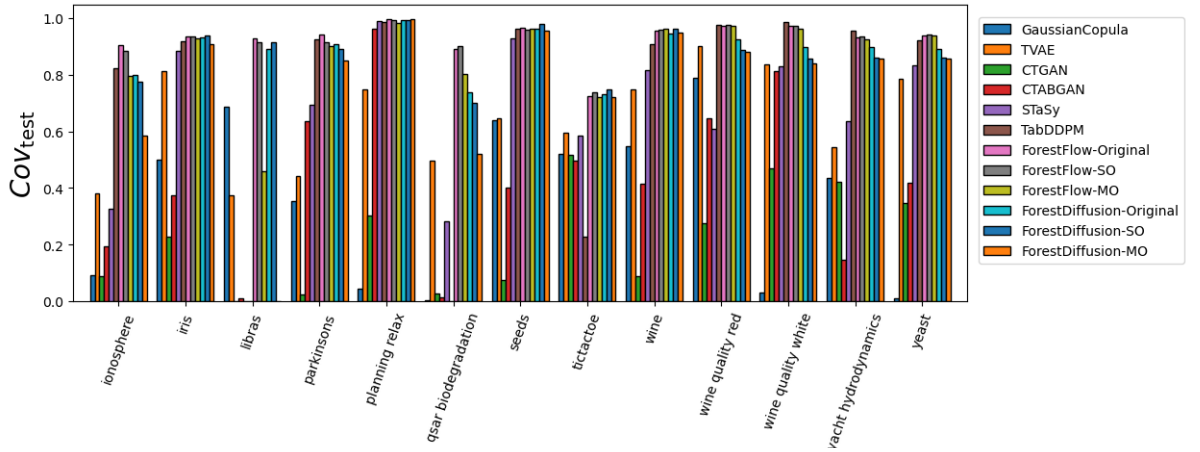

$W1_{test}$
GaussianCopula
TVAE
CTGAN
CTABGAN
STaSy
TabDDPM
ForestFlow-Original
ForestFlow-SO
ForestFlow-MO
ForestDiffusion-Original
ForestDiffusion-SO
ForestDiffusion-MO
airfoil self noise
bean
blood transfusion
breast cancer diagnostic
california
car
climate model crashes
concrete compression
concrete slump
congress
connectionist bench sonar
connectionist bench vowel
ecoli
glass

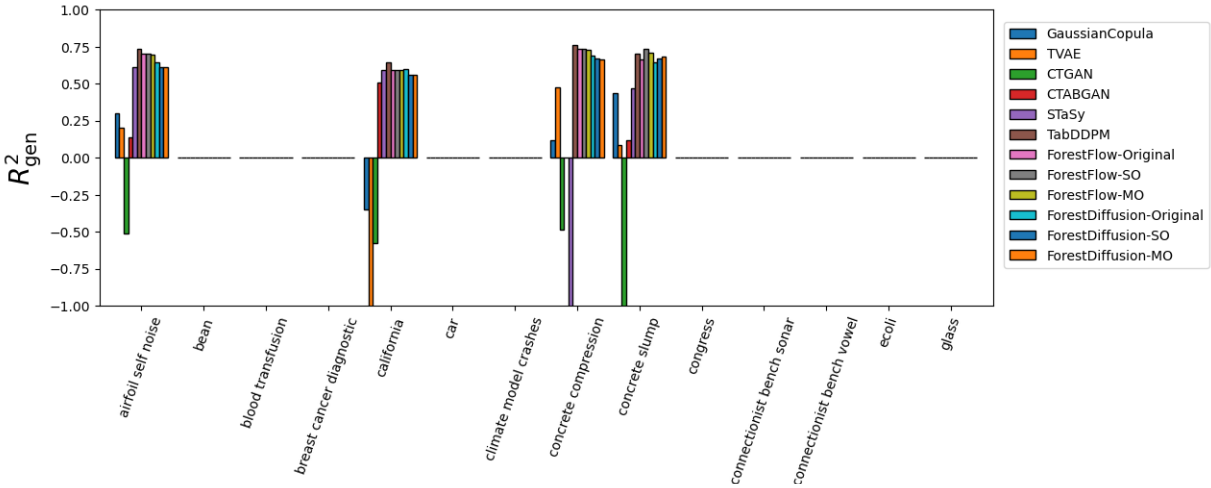

$W1_{test}$
GaussianCopula
TVAE
CTGAN
CTABGAN
STaSy
TabDDPM
ForestFlow-Original
ForestFlow-SO
ForestFlow-MO
ForestDiffusion-Original
ForestDiffusion-SO
ForestDiffusion-MO
ionosphere
iris
libras
parkinsons
planning relax
qsar biodegradation
seeds
tictactoe
wine
wine quality red
wine quality white
yeast

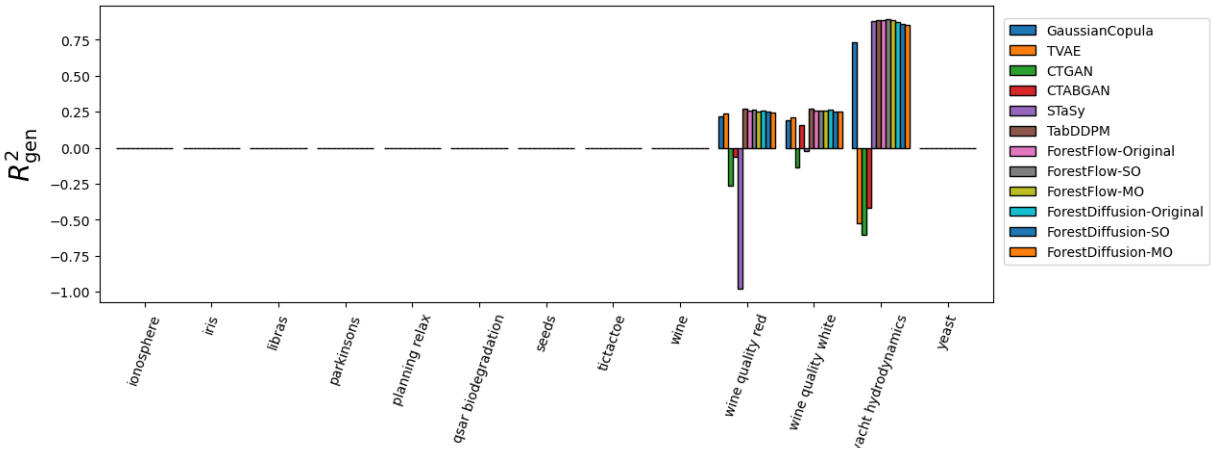

Cov_{train}
1.0
0.8
0.6
0.4
0.2
0.0
airfoil self noise
bean
blood transfusion
breast cancer diagnostic
california
car
climate model crashes
concrete compression
concrete slump
congress
connectionist bench sonar
connectionist bench vowel
ecoli
glass
GaussianCopula
TVAE
CTGAN
CTABGAN
STaSy
TabDDPM
ForestFlow-Original
ForestFlow-SO
ForestFlow-MO
ForestDiffusion-Original
ForestDiffusion-SO
ForestDiffusion-MO

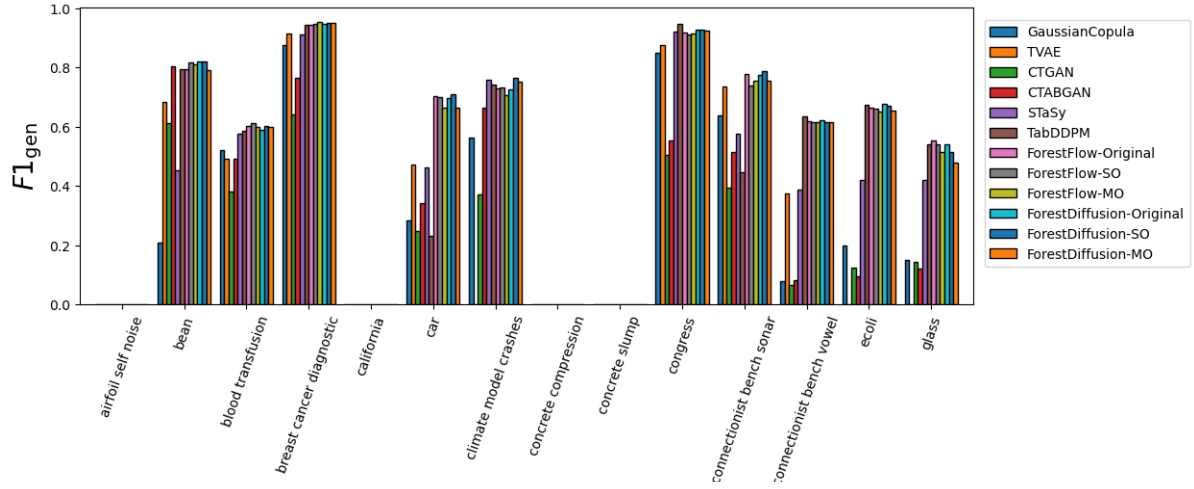

Cov_{train}
1.0
0.8
0.6
0.4
0.2
0.0
ionosphere
iris
libras
parkinsons
planning relax
qsar biodegradation
seeds
tictactoe
wine
wine quality red
wine quality white
yeast
GaussianCopula
TVAE
CTGAN
CTABGAN
STaSy
TabDDPM
ForestFlow-Original
ForestFlow-SO
ForestFlow-MO
ForestDiffusion-Original
ForestDiffusion-SO
ForestDiffusion-MO

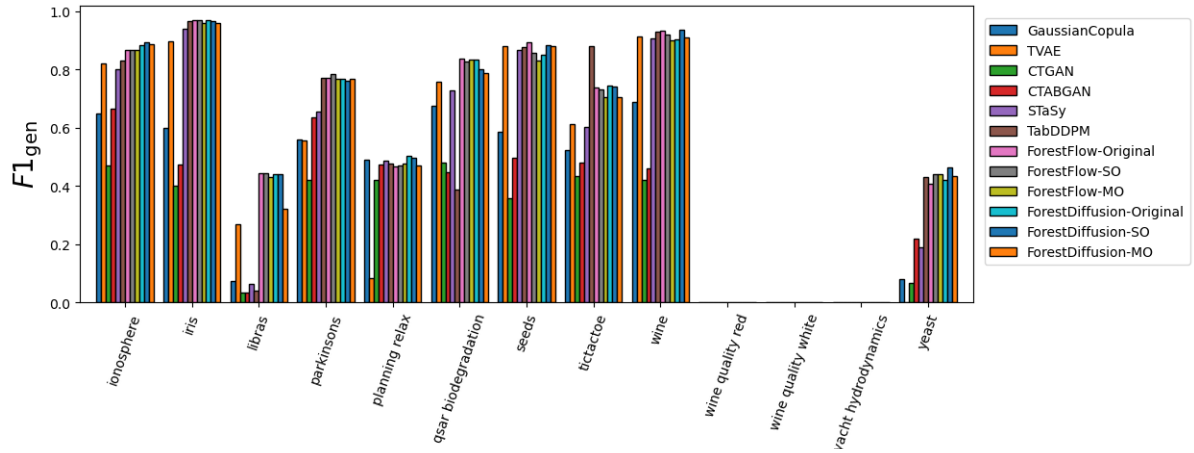

Cov_{test}
1.0
0.8
0.6
0.4
0.2
0.0
airfoil self noise
bean
blood transfusion
breast cancer diagnostic
california
car
climate model crashes
concrete compression
concrete slump
congress
connectionist bench sonar
connectionist bench vowel
ecoli
glass
GaussianCopula
TVAE
CTGAN
CTABGAN
STaSy
TabDDPM
ForestFlow-Original
ForestFlow-SO
ForestFlow-MO
ForestDiffusion-Original
ForestDiffusion-SO
ForestDiffusion-MO

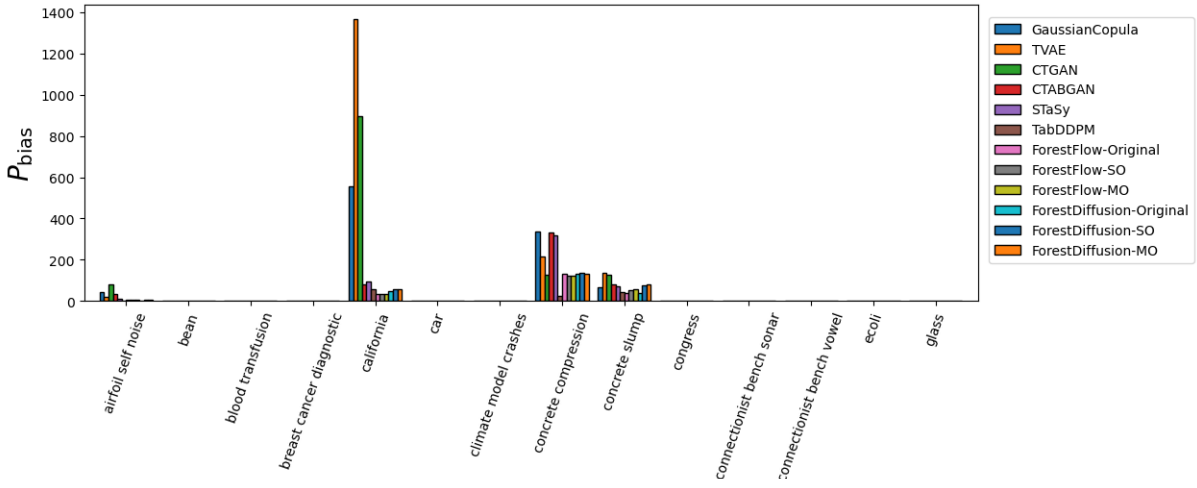

Cov_{test}
1.0
0.8
0.6
0.4
0.2
0.0
ionosphere
iris
libras
parkinsons
planning relax
qsar biodegradation
seeds
tictactoe
wine
wine quality red
wine quality white
yeast
GaussianCopula
TVAE
CTGAN
CTABGAN
STaSy
TabDDPM
ForestFlow-Original
ForestFlow-SO
ForestFlow-MO
ForestDiffusion-Original
ForestDiffusion-SO
ForestDiffusion-MO

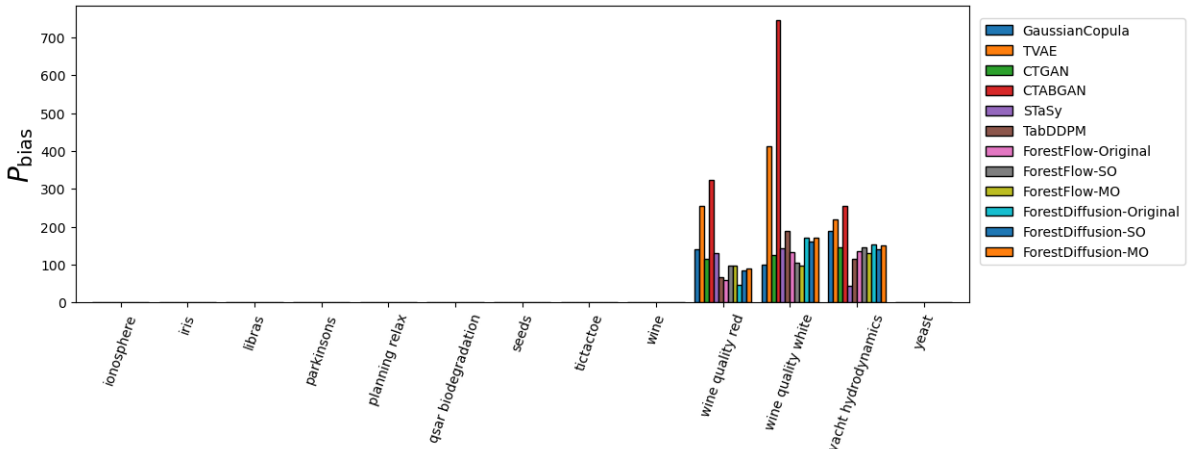

R^2_{gen}
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00
airfoil self noise
bean
blood transfusion
breast cancer diagnostic
california
car
climate model crashes
concrete compression
concrete slump
congress
connectionist bench sonar
connectionist bench vowel
ecoli
glass
GaussianCopula
TVAE
CTGAN
CTABGAN
STaSy
TabDDPM
ForestFlow-Original
ForestFlow-SO
ForestFlow-MO
ForestDiffusion-Original
ForestDiffusion-SO
ForestDiffusion-MO

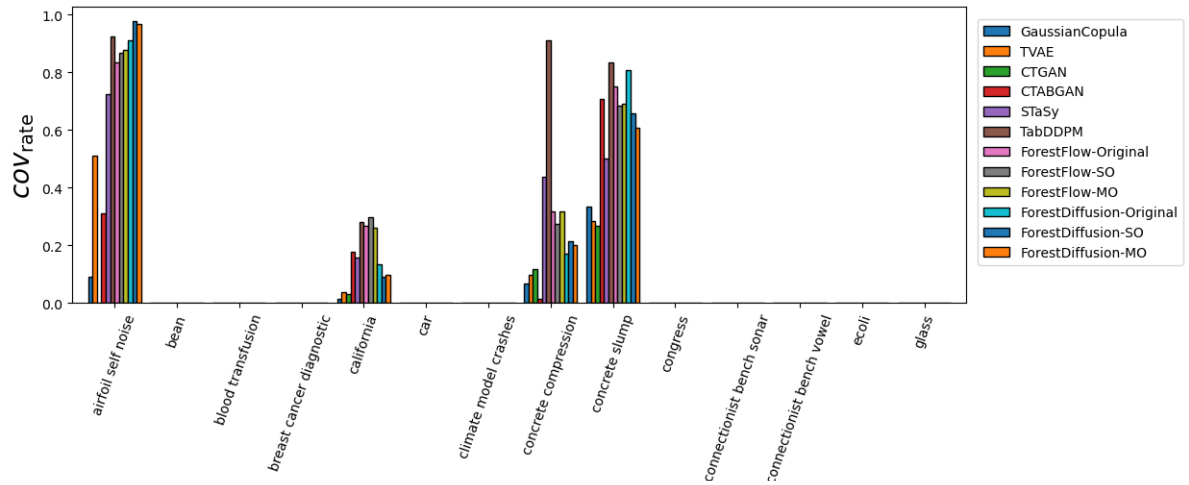

R^2_{gen}
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00
ionosphere
iris
libras
parkinsons
planning relax
qsar biodegradation
seeds
tictactoe
wine
wine quality red
wine quality white
yeast
GaussianCopula
TVAE
CTGAN
CTABGAN
STaSy
TabDDPM
ForestFlow-Original
ForestFlow-SO
ForestFlow-MO
ForestDiffusion-Original
ForestDiffusion-SO
ForestDiffusion-MO

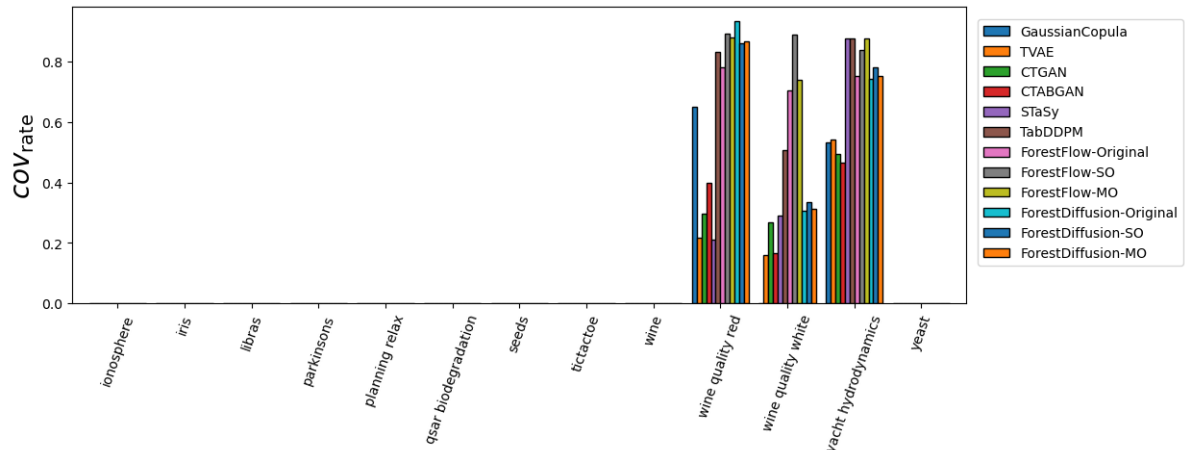

$F1_{gen}$
1.0
0.8
0.6
0.4
0.2
0.0
airfoil self noise
bean
blood transfusion
breast cancer diagnostic
california
car
climate model crashes
concrete compression
concrete slump
congress
connectionist bench sonar
connectionist bench vowel
ecoli
glass
GaussianCopula
TVAE
CTGAN
CTABGAN
STaSy
TabDDPM
ForestFlow-Original
ForestFlow-SO
ForestFlow-MO
ForestDiffusion-Original
ForestDiffusion-SO
ForestDiffusion-MO


$F1_{gen}$
1.0
0.8
0.6
0.4
0.2
0.0
ionosphere
iris
libras
parkinsons
planning relax
qsar biodegradation
seeds
tictactoe
wine
wine quality red
wine quality white
yeast
GaussianCopula
TVAE
CTGAN
CTABGAN
STaSy
TabDDPM
ForestFlow-Original
ForestFlow-SO
ForestFlow-MO
ForestDiffusion-Original
ForestDiffusion-SO
ForestDiffusion-MO


P_{bias}
1400
1200
1000
800
600
400
200
0
airfoil self noise
bean
blood transfusion
breast cancer diagnostic
california
car
climate model crashes
concrete compression
concrete slump
congress
connectionist bench sonar
connectionist bench vowel
ecoli
glass
GaussianCopula
TVAE
CTGAN
CTABGAN
STaSy
TabDDPM
ForestFlow-Original
ForestFlow-SO
ForestFlow-MO
ForestDiffusion-Original
ForestDiffusion-SO
ForestDiffusion-MO


P_{bias}
0
100
200
300
400
500
600
700
ionosphere
iris
libras
parkinsons
planning relax
qsar biodegradation
seeds
tictactoe
wine
wine quality red
wine quality white
yeast
GaussianCopula
TVAE
CTGAN
CTABGAN
STaSy
TabDDPM
ForestFlow-Original
ForestFlow-SO
ForestFlow-MO
ForestDiffusion-Original
ForestDiffusion-SO
ForestDiffusion-MO


COV_{rate}
0.0
0.2
0.4
0.6
0.8
1.0
airfoil self noise
bean
blood transfusion
breast cancer diagnostic
california
car
climate model crashes
concrete compression
concrete slump
congress
connectionist bench sonar
ecoli
glass
GaussianCopula
TVAE
CTGAN
CTABGAN
STaSy
TabDDPM
ForestFlow-Original
ForestFlow-SO
ForestFlow-MO
ForestDiffusion-Original
ForestDiffusion-SO
ForestDiffusion-MO


COV_{rate}
0.0
0.2
0.4
0.6
0.8
ionosphere
iris
libras
parkinsons
planning relax
qsar biodegradation
seeds
tictactoe
wine
wine quality red
wine quality white
yeast
GaussianCopula
TVAE
CTGAN
CTABGAN
STaSy
TabDDPM
ForestFlow-Original
ForestFlow-SO
ForestFlow-MO
ForestDiffusion-Original
ForestDiffusion-SO
ForestDiffusion-MO