GenAI: Fast Data Synthetization with Distribution-free Hierarchical Bayesian Models

Vincent Granville, Ph.D.
vincentg@MLTechniques.com
www.MLTechniques.com
Version 1.0, September 2023

Abstract

Deep learning models such as generative adversarial networks (GAN) require a lot of computing power, and are thus expensive. What if you could produce better data synthetizations, in a fraction of the time, with explainable AI and substantial cost savings? This is what NoGAN2 was designed for. Very different from my first tree-based NoGAN, this new technology relies on resampling, an hierarchical sequence of runs, simulated annealing, and batch processing to boost performance, both in terms of output quality and time requirements. No neural network is involved.

One of the strengths is the use of sophisticated output evaluation metrics for the loss function, and the ability to very efficiently update the loss function at each iteration, with a very small number of computations. In addition, default hyperparemeter values already provide good performance, making the method more stable than neural networks in the context of tabular data generation. It uses an auto-tuning algorithm, to automatically optimize hyperparameters via reinforcement learning. This capability helps you save a lot of time and money.

The purpose of this article is to show the spectacular performance of NoGAN2, using the base model, as some future components are still under construction. One case study involves a dataset with 21 features, to predict student success based on college admission metrics. It includes categorical, ordinal and continuous features as well as missing values. Another case study is a telecom data set to predict customer attrition. Applications are not limited to data synthetization, but also include complex statistical inference problems. Finally, by contrast to most neural network methods, NoGAN2 leads to fully replicable results.

Contents

1 Methodology

The highly generic NoGAN2 technology described here is a powerful and better alternative to neural network synthetization methods for tabular data. It is very different from NoGAN described in [5]. Each offers its own benefits. For reasons that will soon become obvious, I also call this new method hierarchical deep resampling, but I will use the word NoGAN2 for simplicity. Again, it is designed to run much faster, compared to training generative adversarial networks (GAN). Also, the quality of the generated data is far superior to almost all other products available on the market.

Many evaluation metrics to measure faithfulness have critical flaws, sometimes rating synthetic data as excellent, when it is actually a failure, due to relying on low-dimensional indicators. This is especially noticeable on the circle dataset in [4], where all GANs are evaluated as excellent, yet most fail to generate the correct

distribution with points lying on two concentric circles. As I did with NoGAN, here again I fix this problem using the full multivariate empirical distribution (ECDF). It produces much better evaluations. Performance is measured via cross-validation in all my examples.

While the technique is not based on neural networks, it has several features that could also benefit GAN and other deep neural network architectures. Indeed, NoGAN2 can be used as a sandbox to test various features before incorporating them into GAN and similar algorithms. For instance, testing special loss functions, various hierarchical structures and batch processing, or auto-tuning hyperparemeters, is done a lot faster in my NoGAN2 environment. NoGAN2 is also more intuitive and belongs to a set of methods referred to as explainable AI.

1.1 Base algorithm

In a nutshell, the base algorithm is as follows, assuming you want to generate n' synthetic observations, and n is the number of observations in the training set:

Step 1: Initial synthetization. For each feature separately, sample n' values from the univariate ECDF (empirical distribution) computed on the training set, for the feature in question. Put these values in a table, with one column for each feature, and n rows total (the initial synthetic observations).

Step 2: Deep resampling. Pick up one feature randomly, and pick up two rows randomly, from the table created in Step 1. Thus you have two values (possibly identical) for the feature in question. Swap these two values if doing so results in decreasing the loss function; update the table accordingly. Repeat this step until the loss function stops improving.

Thanks to this design, the distribution attached to each feature is correctly replicated at all times during the synthetization process. This includes the means, variances, all statistical moments, percentiles, and all counts and proportions for each categorical variable.

The initial synthetization creates independent features with the correct univariate distributions, as observed in the training set. Then, the goal of Step 2 is to reconstruct the correct dependencies among the features via row shuffles within each feature separately. These shuffles, called swaps, preserve the separate empirical distributions generated in Step 1 at all times, without modifying them at all. In the end, Step 2 consists of keeping the correct univariate distributions, while reconstructing the full multivariate distribution attached to the training set. In the end, the algorithm performs massive permutations or recombinations to minimize the loss function. Thus, it is combinatorial in nature. Optimizing the loss function is done without gradient descent.

1.2 Loss function

The loss function is a distance measuring the similarity between the synthesized data, and the training set. It is minimum and equal to zero when both sets have the same multivariate distribution. However, in the current version, the loss function does not directly compare the two multivariate ECDFs (synthetic data and training set). Instead, it uses proxy measurements to do this job indirectly, leading to extremely fast but approximate computations.

The proxy mechanism works as follows. Let's assume that you have m features denoted as $X_1, \cdots, X_m$ for the training set, and $X'_1, \cdots, X'_m$ for the synthesized data. The functions $g_2(x), h_2(x)$ and so, in the algorithm below, transform a vector (feature) into another vector with the same number of rows, element-wise. The star product is the element-wise product, also known as the Hadamard product [Wiki]. Summations are over all the elements of the vector under the sum, not over i, j or k. Now, here is how to define and compute the loss function:

- Compute $Q_2[i,j] = \sum g_2(X_i) * h_2(X_j)$ for two-way interactions, $Q_3[i,j,k] = \sum g_3(X_i) * h_3(X_j) * k_3(X_k)$ for three-way interactions, and so on, on the training set. Do it for all permutations of $1 \leq i, j, k \leq m$.
- Likewise, compute $Q'_2[i,j] = g_2(X'_i) * h_2(X'_j)$, $Q'_3[i,j,k] = g_3(X'_i) * h_3(X'_j) * k_3(X'_k)$, and so on, for the synthesized data under construction, at each iteration.
- Normalize the quantities so that they lie between -1 and $+1$. For instance, $Q_2[i,j]$ and $Q'_2[i,j]$ become

$$\rho_2[i,j] = \frac{\widetilde{Q}_2[i,j] - \mathrm{E}[g_2(X_i)] \cdot \mathrm{E}[h_2(X_j)]}{\sigma[g_2(X_i)] \cdot \sigma[h_2(X_i)]}, \quad \rho'_2[i,j] = \frac{\widetilde{Q}'_2[i,j] - \mathrm{E}[g_2(X'_i)] \cdot \mathrm{E}[h_2(X'_j)]}{\sigma[g_2(X'_i)] \cdot \sigma[h_2(X'_i)]},$$

where

$$\widetilde{Q}_2[i,j] = \frac{Q_2[i,j]}{n}, \quad \widetilde{Q}'_2[i,j] = \frac{Q'_2[i,j]}{n'}.$$

Here σ stands for the standard deviation. The means and standard deviations are computed respectively on the training set for $\rho_2[i,j]$, and on the synthesized data under construction for $\rho'_2[i,j]$. However they are

computed only once: just after the initial synthetization obtained in Step 1. They will remain unchanged througout the Step 2 iterations. Also, $\rho_2[i,j]$ and $\rho_2'[i,j]$ are correlation coefficients, assuming categorical values are encoded as integers.

- Typical functions are $g_2(x) = x$, denoted as $g_{21}(x)$, and $g_2(x) = x^2$, denoted as $g_{22}(x)$. Likewise for $h_2(x)$. In the current implementation, there is no three-way interactions (the g_3 and h_3). From there, the partial distance functions for two-way interactions are defined as $\Delta_2[i,j] = |\,\rho_2[i,j] - \rho_2'[i,j]\,|$, for $1 \leq i,j \leq m$.

  If using four functions $g_{21}(x), g_{22}(x), h_{21}(x), h_{22}(x)$, then instead of $Q_2[i,j]$ we have

$$Q_{21}[i,j] = \sum g_{21}(X_i) * h_{21}(X_j), \quad Q_{22}[i,j] = \sum g_{22}(X_i) * h_{22}(X_j),$$

  where the sum is not over i,j (assumed to be fixed), but over all the vector elements in each Hadamard product, with each element corresponding to a specific observation. Same for $Q_2'[i,j]$ and $\rho_2[i,j], \rho_2'[i,j]$, each broken down into two pieces. This leads to

$$\Delta_{21}[i,j] = |\,\rho_{21}[i,j] - \rho_{21}'[i,j]\,|, \quad \Delta_{22}[i,j] = |\,\rho_{22}[i,j] - \rho_{22}'[i,j]\,|.$$

- Now let $\Delta_2[i,j] = \max(\alpha_1 \cdot \Delta_{21}[i,j],\, \alpha_2 \cdot \Delta_{21}[i,j])$ with $\alpha_1, \alpha_2 \geq 0$ and $\alpha_1 + \alpha_2 = 1$. If taking into account two-way interactions only, the loss function Δ_2 is the sum of $\Delta_2[i,j]$ over all features i,j with $i \neq j$. An alternative version $\widetilde{\Delta}_2$ consists in using the maximum rather than the sum.

The goal was to design a loss function that maps one-to-one to the multivariate Kolmogorov-Smirnov distance (KS) between synthesized and real data, as KS is the perfect metric for evaluation purposes. We wanted the mapping to be continuous and order-preserving. By focusing on two-way interactions only, and breaking down g_2, h_2 into two components only $(g_{21}, g_{22}, h_{21}, h_{22})$, we get an approximate solution to this problem. But in practice, it works much better and faster than alternatives based on neural networks. In NoGAN3, I will use a different mapping, based on counts per bin, to incorporate the best from the NoGAN technique discussed in [5].

If $g_{21}(x) = h_{21}(x) = x$, then $\rho_{21}[i,j]$ is the correlation coefficient measured on the training set, between features i and j. Here I assume that these features are continuous or dummy variables representing categories. Likewise, $\rho_{21}'[i,j]$ is the same quantity but measured on the synthesized data. A future version will incorporate Cramér's V coefficient: this is the standard metric to represent the association between arbitrary (non-binary) categorical features.

Explaining the loss function is much easier to do in Python than English. So if the topic looks complicated, the Python code may clarify many points. The complexity is due to designing a very efficient architecture, so that tiny changes in the data require only tiny changes in the loss function. In doing so, I relied heavily on tensors, yet without the need to define what it is, and in the code, without using TensorFlow or similar libraries.

1.3 Hyperparemeters and convergence

By contrast to GAN and related techniques, in the current implementation of NoGAN2, the loss function always decreases after swapping two values, albeit more and more slowly over time. It does not oscillate. Swaps become rarer and rarer over time, making progress towards the optimum extremely slow or even impossible after a while. Getting stuck is similar to the vanishing gradient issue [Wiki] in neural networks dealing with tabular data. However, the problem and side effects are typically less pronounced in NoGAN2. In section 2, I discuss how to reduce the risk of getting stuck too early. A future version will occasionally allow swaps even if they result in increasing the loss. It will be performed using a simulated annealing schedule [Wiki], and result in an oscillating loss, with amplitudes decreasing over time, just like in a non-failing GAN.

In practice, despite the combinatorial nature of the problem, a good solution is obtained once each value in the initial synthesized data has been proposed for a swap a couple of times. So the computational complexity is proportional to the number of observations to generate, multiplied by the number of features. Generating small batches of observations separately (by splitting the initial synthesized data in a number of batches), can further improve performance. It is discussed in section 2.

The first version of NoGAN2 (now abandoned) was a step-wise procedure: you generate the second feature leaving the first feature unchanged, then the third feature leaving the first two features unchanged, and so on. Since optimizing the swaps one feature at a time is an assignment problem [Wiki], this approach allows you to use the Hungarian algorithm [Wiki] to efficiently solve this combinatorial problem. However it resulted the inability to make improvements after processing a few features. The final synthetization had the first features very well replicated, and the other ones very poorly rendered.

In the current implementation, features are processed jointly rather than separately. Fine-tuning hyperparameters helps you accelerate the speed of convergence and avoid a local optima. I explain in section 2 how the algorithm can auto-tune itself. Now I discuss the main hyperparameters:

- The seed of the random number generator involved can have an impact on the final synthetization. You may try different seeds to improve the convergence or quality of the results.
- The number of batches discussed earlier.
- The weights α_1 attached to g_{21}, h_{21} and α_2 attached to g_{22}, h_{22} in the loss function, see section 1.2.
- The loss function itself, especially the functions $g_{21}, h_{21}, g_{22}, h_{22}$ and any additional functions that you may use. For instance, g_{23} and h_{23}, then requiring the extra weight α_3.
- When randomly picking up a feature to swap two values, by default, each feature has the same probability of being selected. You can change these probabilities, selecting feature i with probability p_i instead. The probability vector $\mathrm{P} = [p_1, \ldots, p_m]$ is a core hyperparameter. Here m being the number of features.
- If one of the probabilities is set to zero, then the feature in question will never be selected, and won't be updated. You can use this property as follows. Say you have three different groups of features, A B, and C. For instance numerical features in A, ordinal features in B, and continuous features in C. Run NoGAN2 three times: first with non-zero probabilities in A only, then with non-zero probabilities in B only, then with non-zero probabilities in C only.

In the last example, each subsequent run improves on the previous one, by adding features that haven't been processed yet. It is the reason why NoGAN2 is also called hierarchical deep resampling, by analogy to Bayesian hierarchical models [Wiki]. In fact, in probability lingo, the three runs aimed at sampling from $P(A, B, C)$ by doing it first for $P(A)$, then $P(B|A)$, then $P(C|A, B)$, based on the fact that $P(A, B, C) = P(A)P(B|A)P(C|A, B)$.

The same idea can be used in traditional generative adversarial networks, leading to Hi-GAN, for hierarchical GAN. See for instance [3] and [9]. Finally if only one function $g_2(x) = h_2(x) = x$ is used in the loss function, then NoGAN2 is equivalent to the copula technique discussed in chapter 10 in my book [7]. In that case, starting with an initial synthetization with correct marginal distributions but zero cross-correlations, it reconstructs the correct correlation structure attached to the features.

1.4 Acknowledgments

I would like to thank Shakti Chaturvedi for the numerous tests and research that he performed to compare the new technique proposed here, with various generative adversarial networks. He brought the Telecom dataset to my attention, and tested improved versions of GAN and WGAN as well as vendor solutions and related methods. Earlier versions of the NoGAN2 code, along with WCGAN implementations, are available as Jupyter notebooks on his GitHub repository, here, illustrated on various datasets.

I am also very grateful to Rajiv Iyer for turning the multivariate empirical distribution (ECDF) and related KS distance computations into a production code Python library, available here. You can install it with `pip install genAI-evaluation`. See how I use it in section 4. Rajiv also compared NoGAN2 with CTGAN on the student dataset. All comparisons are favorable to NoGAN2.

2 Case studies

I synthesized two datasets. In the first one, observations consist of customer statistics from a Telecom company, to assess attrition rates by segment. I used 4 features and 3500 observations. The feature "total charges" is highly correlated to "tenure" (how many months the customer stayed with the company), and caused more problems when combined with "monthly charges", visible only in higher dimensional plots. I first used GAN and Wasserstein GAN [14], with several data transforms and various tricks to improve the synthetization. In particular, I performed a principal component analysis (PCA) on the training set, then synthesized the transformed data, then applied the inverse PCA transform. The improvements, while substantial and discussed in [6], were not good enough. It led me to create NoGAN (discussed in [5]), and then NoGAN2 presented here. Both provide satisfactory results, in a short amount of time, and with little if any fine-tuning.

The second dataset consists of student admission metrics. The outcome is the success rate at college. I used 21 features and 4400 observations, with a mix of categorical and numerical features, as well as missing data. The structure of the missing data is very simple. In this case, using a separate NoGAN2 for the 300 observations with missing values, is the easiest solution. For more scattered missing values, see how to do it in section 4.2 in [5]. The goal here is not to impute missing values, but to replicate them correctly. For imputation done via synthetization, see the GAIN technique (a variant of GAN) in [15]. This dataset was first synthesized with CTGAN. Some improvement was obtained with NoGAN without using the proper encoding for categorical variables. With proper encoding, the performance was increased by several orders of magnitudes. With NoGAN2, you get good performance (much better than GAN) without any encoding or data transform, in a fraction of the time required by neural networks methods, thus with considerable cost savings.

All the tests involve splitting the real data into two parts: training and validation. The training set is used to generate the synthetic data, while the validation set (also called holdout) is used to assess its quality: how well it mimics the joint empirical distribution (ECDF) observed in the validation set. To achieve this goal, I implemented the powerful Kolmgorov-Smirnov distance to compare the two multivariate ECDFs: training versus validation. Unlike most other distances in the marketplace, it does not miss high dimensional patterns; it will not mark a synthetization as good if it is a failure, undetected by classic evaluation metrics. See section 2.5

Before diving into the actual synthetizations in sections 2.1 and 2.2, I want to mention a few important points. First and obvious, do not check whether two identical values (in two different rows, for a specific feature) benefit from being swapped. Instead, look for different rows with distinct values. This will speed up the algorithm when dealing with numerous binary features such as dummy variables. Then, before fine tuning hyperparameters, read section 2.4. Finally, to generate data outside the observation range, you need to stretch the Numpy quantile function used in the algorithm, or write your own: it does not generate values below the observed minimum, or above the observed maximum. See how to do it in [1].

2.1 Synthesizing the student dataset

The dataset is available on GitHub, here. I also use it in the Python code in section 4. In this section, I explain some techniques to accelerate performance, both in terms of speed and quality of the synthetization. Figures 1 and 2 show the evolution of the loss function and cumulated number of swaps, respectively during the first and last 500k iterations. These iterations start after the initial synthetization, that is, after Step 1 in section 1.1. An iteration consists of randomly selecting a feature, then randomly selecting two rows, and check whether or not the two values (one from each row) must be swapped. The swap takes place if it decreases the loss function. In that case, it increases the number of swaps by one.

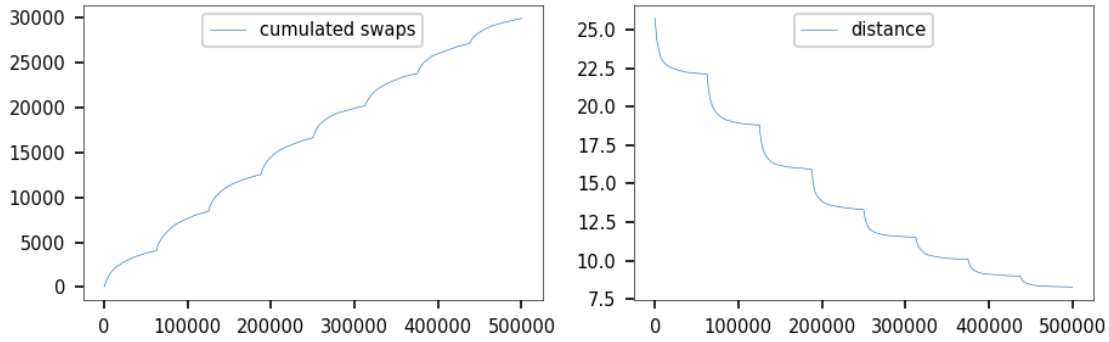

Figure 1: Number of swaps (left) and loss function (right) over time: first run

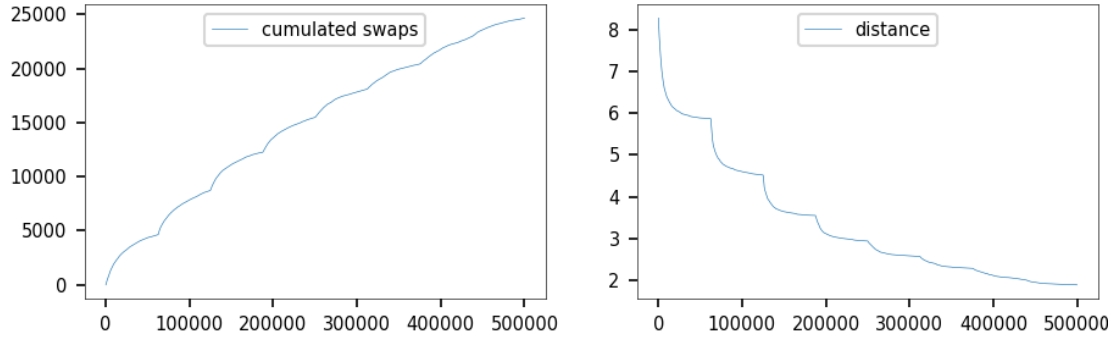

Figure 2: Number of swaps (left) and loss function (right) over time: second run

The first mechanism consists in using batches. I first generate 2000 synthetic observations in the initial synthetization. Then I split the generated data into 8 subsets called batches. Resampling – the swaps – are performed one batch at a time, swapping values internally from within a batch, without modifying the other batches. Because swaps tend to become more and more rare over time, this technique accelerates convergence: this is noticeable in Figures 1 and 2, where the loss function drops quickly at the beginning of each batch but then taper off. By using 8 batches rather than one, overall the loss function spends more time dropping sharply, than staying in a low entropy mode. It also keeps the number of swaps almost constant over time, rather than dropping to zero in late iterations. Bumps happen when moving from one batch to the next one. By contrast,

Figure 5 (Telecom dataset) shows how slowly the loss function decreases after the initial drop, when using one batch only.

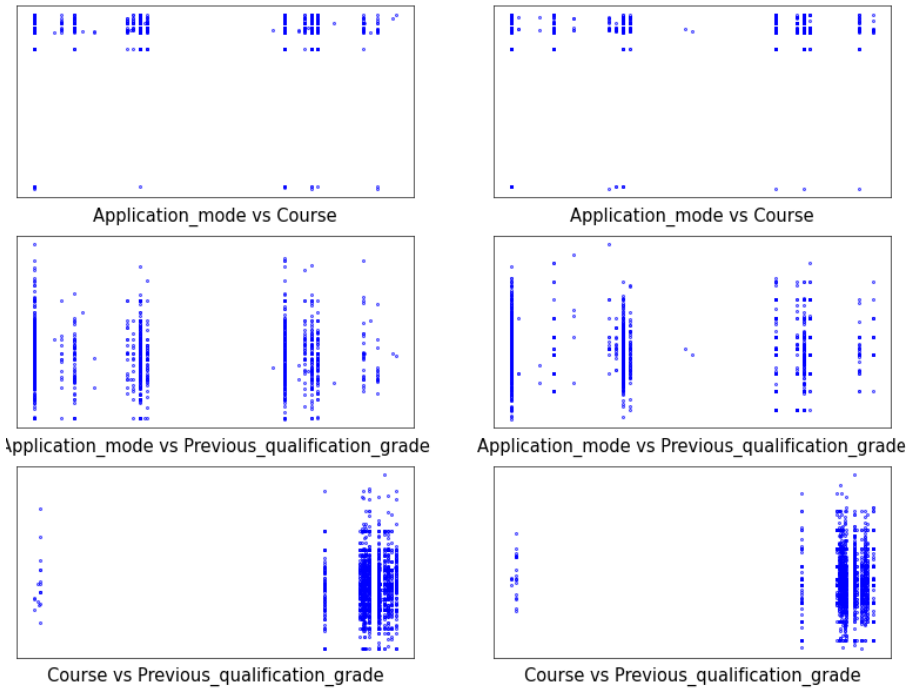

Figure 3: Synthetization (left) vs validation set (right), student dataset

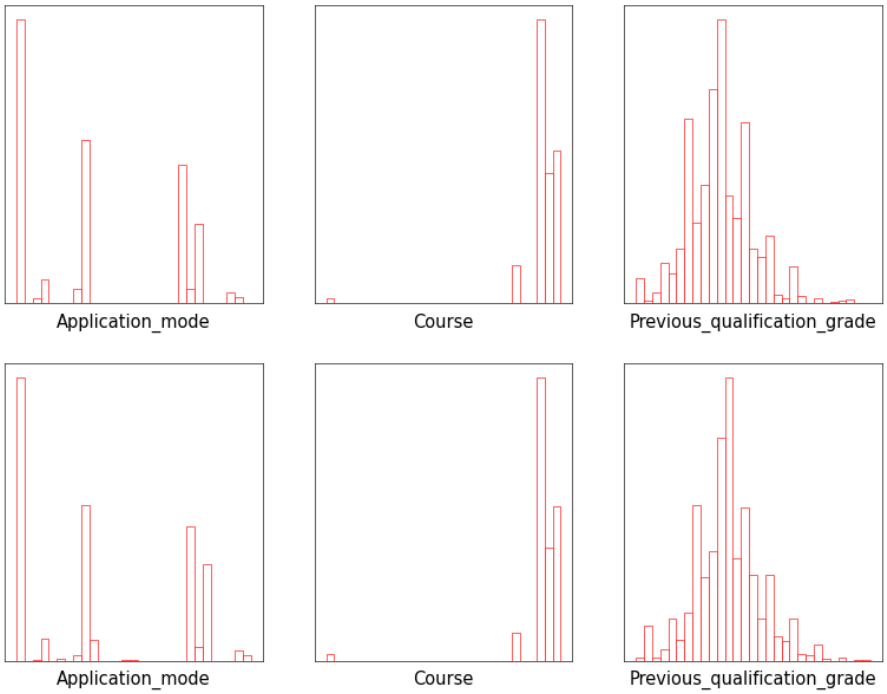

Figure 4: Synthetisation (top) vs validation set (bottom), student dataset

The initial synthetization (before starting deep resampling) correctly replicates each univariate distribution in the training set. Since deep resampling does not modify at all the univariate distributions, all the histograms in Figure 4 always look very good. Thus it is more important to focus on the scatterplots in Figure 3. They look good as well. Despite the 21 features, this data set is actually easy to synthesize, at least with NoGAN2. I used two runs, see part 8 in the code featured in section 4: the first run mostly for the numerical features, and the second run for to remaining features, conditionally to the first run. Thus we are not synthesizing the two sets of features separately, but jointly via a conditional (Bayesian) mechanism. Hyperparameter values are set

to default, except for the weights. You may want to look at my choice for the functions g_{22} and h_{22}, respectively `g` and `h` in the code.

Finally, a small number of observations have missing values, always for the same subset of features. I discarded these observations. In this case, it would be easy to run a separate NoGAN2 on these observations, after removing the features in question. Then putting back these features, where the value is always zero (missing) everywhere. The end result, after removing missing values, is KS = 0.0651, and Base KS = 0.0405. This is within the range of what is considered a good synthetization. For a definition of KS and Base KS, see section 2.5.

2.2 Synthesizing the Telecom dataset

The Python code and dataset are on GitHub, here. With 7000 observations and 4 features including a categorical one, the data is easy to synthesize with NoGAN2. However, despite considerable efforts and testing various improvements, we were unable to produce synthetizations of the same quality with generative adversarial networks (GAN, WGAN and so on). Only NoGAN [5] matches the quality and speed of execution of NoGAN2. The details are in my article "How to Fix a Failing Adversarial Network", see [6]. In short, the problem arises when the features "TotalCharges" and "Tenure" are both present. The latter represents the number of months a customer has stayed with the company; it is highly correlated to the former. But even after decorrelating and scaling transforms, the GAN results, while significantly improved, are well below the performance of NoGAN2.

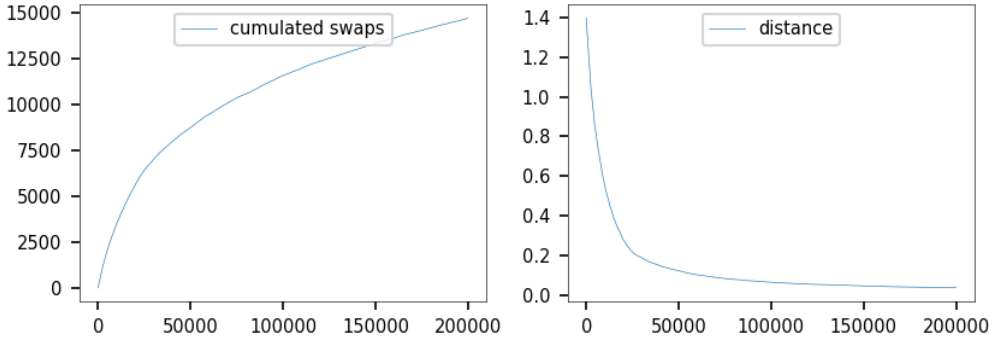

Figure 5: Number of swaps (left) and loss function (right) over time

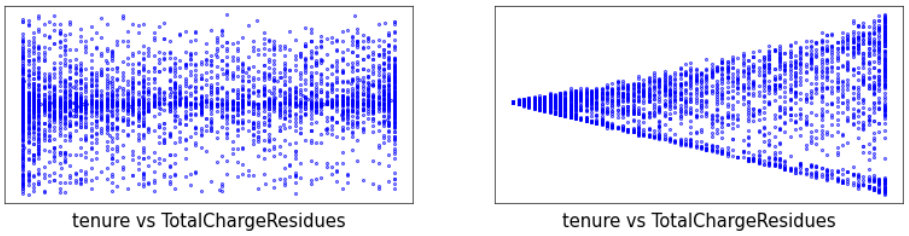

Figure 6: Synthetisation with one term in loss function (left), vs real data (right)

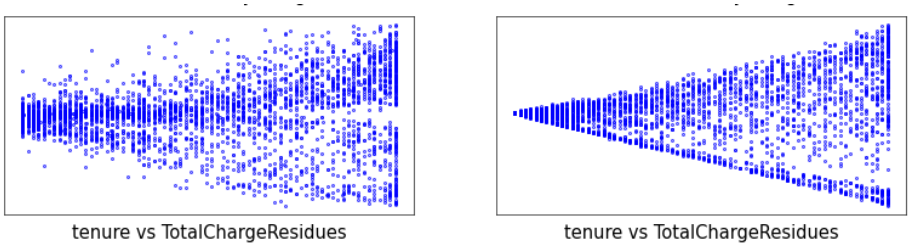

Figure 7: Synthetisation with two terms in loss function (left), vs real data (right)

With this dataset, only one run of deep resampling was needed, processing all features at once. However, unequal weights $\alpha_1 \neq \alpha_2$ in `weights`, combined with uneven values in the probability vector `hyperParam`, significantly improves the quality of the synthesized data. In particular, oversampling the problematic feature

"TotalCharges" really helps. This is done by setting `hyperParm[2]` to an unusually large value. Such fine-tuning is easy to automate, and intuitive.

Finally, Figure 7 shows why we need the two terms in the loss function, rather than just the first one. It significantly increases the quality, when compared to Figure 6. Interestingly, the functions used in the second term are $g_{22}(x) = h_{22}(x) = |x|$. It impacts correlations involving at least one feature with both positive and negative values: in this case, "TotalChargeResidues", which is the decorrelated version of "TotalCharges". Here, KS = 0.0379, and Base KS = 0.0176. As always, the univariate distributions are well rendered, see Figure 8.

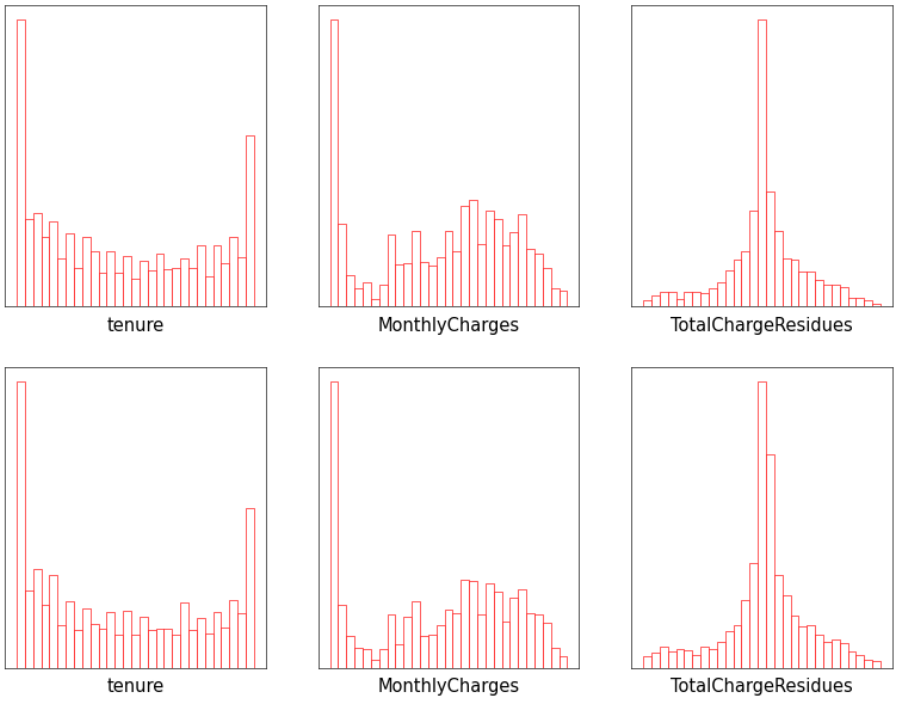

Figure 8: Synthetisation (top) vs validation set (bottom), Telecom dataset

2.3 Other case studies

Besides the Telecom and education domains (the student data), I also explored datasets in the healthcare, insurance and cybersecurity industries. Finally, I tested NoGAN2 on the challenging circle dataset, see Figure 9. The Python code and datasets are on GitHub, here. Look for the documents starting with `DeepResampling` in the filename, for instance `DeepResampling_circle.py`. These examples have both numerical and categorical features. Each one illustrates specific options and hyperparameters. The peculiar cybesecurity case is discussed in the project textbook offered to participants in my Gen AI certification program, available here. The diabetes data also includes missing values, properly rendered by the synthetization.

The general conclusion is that NoGAN2 trains much faster than neural network equivalents, and consistently provide better results, when evaluated using the best distance in a cross-validation setting: the Kolmogorv-Smirnov distance (KS) based on the multivariate ECDF (joint empirical distribution). This evaluation metric is now available as an open-source Python library (`genAI-evaluation`). It captures all the interdependency patterns among the features. By contrast, distances currently used on the marketplace are not implemented in full multivariate mode. It frequently leads to false negatives: synthetizations rated as excellent, when they are actually very poor. This is magnified when synthesizing the circle dataset, as discussed in [4].

NoGAN2 requires less fine-tuning than GAN and other deep neural network techniques. Also, fine-tuning is straightforward, thanks to the explainable nature of the whole system. This allows for auto-tuning as discussed in see section 2.4. In the end, for tabular data generation, the only real competitor to NoGAN2 is NoGAN [5], also developed in my laboratory. Both require very little bandwidth, significantly reducing costs while providing better results.

Finally, categorical features can benefit from being encoded as dummy variables, or from a loss function where standard correlations are replaced by Cramér's V [Wiki] or similar statistics [2]. Most examples also include a label feature (the rightmost column in the dataset) to categorize each observation into various clusters. For instance, the label feature in the circle dataset indicates whether an observation belongs to the inner or outer circle in Figure 9. Using a label feature significantly improves the performance. This is also true for GAN.

The challenges with the circle data is the very small number of observations, combined with nearly duplicate features.

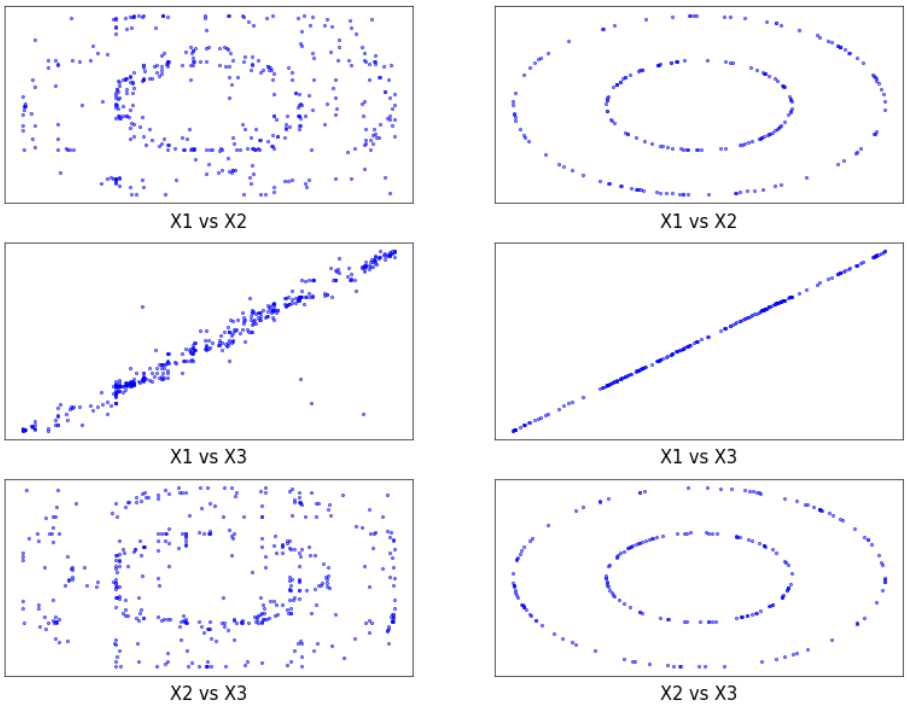

Figure 9: Synthetization (left) vs validation set (right), circle dataset

2.4 Auto-tuning the hyperparameters

Before discussing auto-tuning, let's look at the correlation coefficients involved in the loss function, using the same notation as in section 1.2. For illustration purposes, see Table 1 corresponding to the Telecom dataset with 4 features. The table is organized as follows:

- The first two columns are feature indices. The next three columns are associated to the first term in the loss function, and the rightmost three columns to the second term.
- For each feature pair $\{i, j\}$, the correlations ρ_{12}, ρ_{21} are computed on the validation set, while ρ'_{12}, ρ'_{21} are computed on the synthesized data. The metrics Δ_{12}, Δ_{22} are the absolute difference between correlation coefficients.
- More specifically, the correlation coefficients are defined as

$$\begin{aligned}
\rho_{21}[i,j] &= \text{Corr}[\, g_{21}(X_i), h_{21}(X_j)\,], \\
\rho'_{21}[i,j] &= \text{Corr}[\, g_{21}(X'_i), h_{21}(X'_j)\,], \\
\rho_{22}[i,j] &= \text{Corr}[\, g_{22}(X_i), h_{22}(X_j)\,], \\
\rho'_{22}[i,j] &= \text{Corr}[\, g_{22}(X'_i), h_{22}(X'_j)\,].
\end{aligned}$$

  Here $[X_1, \ldots, X_m]$ and $[X'_1, \ldots, X'_m]$ are respectively the validation and synthetic datasets. Each element represents a column. Also, $g_{21}(x) = h_{21}(x) = x$ for the first term of the loss function. For the second term, I chose $g_{22}(x) = h_{22}(x) = |x|$ component-wise (x is a vector).

The Python code for the Telecom data is available here. The algorithm works best when the maximum values for Δ_{21} and Δ_{22} are similar. Here, the maximum is larger for Δ_{22}. For optimization, increase α_2 in the weight vector $\text{W} = [\alpha_1, \alpha_2]$, keeping $\alpha_1 + \alpha_2 = 1$. This will improve Δ_{22}, but penalize Δ_{21}. Also, choose g_{22} and h_{22} so that the correlations ρ_{21}, ρ_{22} in the first and second terms of the loss function are quite different, with ρ_{22} not too close to zero. This is achieved by testing a few different functions, typically with $g_{22} = h_{22}$. To goal is to build a second term in the loss function, that brings significant improvements over using the first term alone.

The next hyperparameter is the probability vector $\text{P} = [p_1, \ldots, p_m]$ with $p_1 + \cdots + p_m = 1$, also described in section 1.3. It specifies how frequently each feature is selected for a potential swap (swapping the values from two random rows, in the column corresponding to the feature in question). In Table 1, the worst correlation

discrepancies involve features $\{0, 2\}$ and $\{1, 2\}$. This suggests that feature 2 is more challenging. Increasing p_2 may reduce the error. However, it will penalize other features. To choose the optimum p_2, you can let the algorithm do the job, using an adaptive p_2 automatically fine-tuned over time to minimize the loss. The same applies to all the probabilities in P, as well as the two weight parameters in W.

i	j	ρ_{21}	ρ'_{21}	Δ_{21}	ρ_{22}	ρ'_{22}	Δ_{22}
0	1	0.2486	0.2486	0.0000	0.2486	0.2486	0.0000
0	2	0.1225	0.1515	0.0290	0.7334	0.6708	0.0626
0	3	−0.3518	−0.3518	0.0000	−0.3518	−0.3518	0.0000
1	2	0.8186	0.7960	0.0226	−0.0137	−0.0091	0.0046
1	3	0.1815	0.1815	0.0000	0.1815	0.1815	0.0000
2	3	0.1229	0.1229	0.0000	−0.2830	−0.2830	0.0000

Table 1: Correlations in the loss function, Telecom dataset

Finally, the best improvements are obtained by running NoGAN2 twice. First, you split the set of features into two subsets A and B. In the first run, you optimize the loss function for features in A. Then, in the second run, you optimize the full loss function: B conditionally to A, with synthetic values obtained for A in the first run, left unchanged. To implement this mechanism, an extra hyperparameter is needed, the flag vector $F = [q_1, \ldots, q_m]$. In the first run, any q_i set to zero forces feature i to be ignored in the computation of the loss function. Also set p_i to zero in that case. In the code, P and F are represented respectively by `hyperParam` and `flagParam`. To decide which features to include in A, add one feature at a time until you hit a wall and the loss function gets stuck well above zero (convergence failure). This process can be automated.

You can leverage the above mechanism to work with more than two terms in the loss function: to do this, use a different set of functions $g_{21}, h_{21}, g_{22}, h_{22}$ in the second run. If you implement this strategy, before the second run, you need to re-run `initialize_cross_products_tables()` and `compute_univariate_stats()` in the Python code. Increasing the number of terms in the loss function, may help. Because the loss function does not map one-to-one to the KS distance (it is only a proxy for KS), vastly different synthetizations may achieve zero loss, yet have a poor KS. In this case, the only way out is to use a better loss function, for instance with three terms rather than two. The NoGAN3 algorithm will address this issue by using bin counts rather than correlations, in the loss function.

2.5 Evaluation with multivariate ECDF and KS distance

The multivariate empirical distribution (ECDF) and corresponding Kolmogorov-Smirnov distance (KS) between the two ECDFs – validation set vs synthesized data – is described in details, along with the Python implementation, in [5]. It is now available as a Python library, named GenAI-Evalution, available here. In this section, I provide a brief overview.

The joint or *multivariate* ECDF has remained elusive to this day. It is a rather non-intuitive object, hard to visualize and handle even in two dimensions, let alone in higher dimensions with categorical features. For that reason, the empirical probability density function (EPDF) is more popular: it is associated to mixture models, frequently embedded into neural networks, or the Hellinger distance to evaluate the quality of synthesized data. However, the ECDF is more robust. Then, while the Hellinger distance also generalizes to multivariate EPDFs, in practice all the implementations are one-dimensional, with the distance computed for each feature separately. The Hellinger equivalent based on ECDFs instead, is indeed the KS distance. It belongs to a class of measures known as integral probability metrics [11, 13].

It is said that KS does not generalize to the multivariate case. Thus its total absence in applications when the dimension is higher than too, despite the fact that it is the best metric to fully capture all the dependencies among features, especially the non-linear ones. Asymptotics for the multivariate KS distance were investigated only recently [12], while an algorithm for fast computation of the multivariate ECDF is discussed in [10]. To my knowledge, the first practical implementation in high dimensions, tested on real datasets, is found in [5].

As a reminder, given a dataset and location $z = (z_1, \ldots, z_m)$ in the feature space, the ECDF $F(z)$ evaluated at z is defined as the proportion of observations $x = (x_1, \ldots, x_m)$ satisfying $x_i < z_i$ for $i = 1, \ldots, m$. Here m is the number of features or columns, and z_i is any observed value attached to feature i. The KS distance is then defined as

$$\text{KS}(F_s, F_v) = \sup_z |F_s(z) - F_v(z)|,$$

where z is any location in the feature space, and F_s, F_v are the ECDFs, respectively computed on the synthetic and validation set. In practice, F_s, F_v are approximated using a number of locations called interpolation nodes. These nodes are generated according to some stochastic process based on the quantiles of the original features. The goal is to guarantee that the nodes cover the sparse working area – the tiny region of the feature space where the observations lie – with maximum efficiency. How small that area is, compared to the full potential feature space, depends to a large extent on the dimension: the number of features.

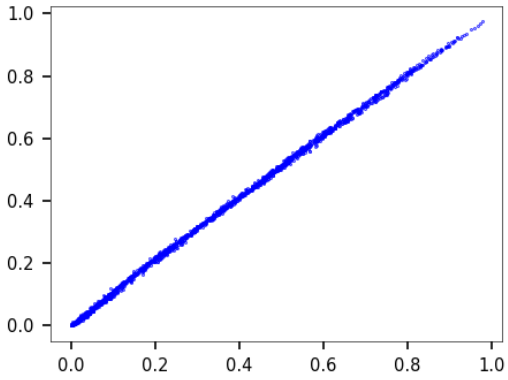

Figure 10: ECDF scatterplot, synthetic vs validation set

Try increasing `n_nodes` (the number of nodes) from 10^3 to 10^4 and 10^5, to see when KS stabilizes. The scatter plot in Figure 10 is produced in part 10 in the code featured in section 4, while the value of `n_nodes` is set in part 9.

Besides computing the KS distance between the synthetic and validation sets, it is useful to also compute KS between the training and validation sets. This distance is called "Base KS" and named `KS_base` in the code. Since the synthetization is based on training data only, if the training and validation sets are very different, you should expect that the synthesized data will not be great at mimicking the real data, outside the training set. This occurs when not properly splitting the real data into training and validation sets. To summarize, the absolute difference between KS and Base KS may be the best indicator of faithfulness. When KS and Base KS are very similar, you may want to increase `n_nodes` to get more accurate values for better discrimination.

3 Conclusion

The context is tabular data generation. NoGAN2, by contrast to NoGAN and despite not being based on neural networks, shares a lot of properties with GAN. For instance, the two terms in the loss function fight against each other in a way similar to discriminator vs generator in GAN; I use weights to give each term a fair chance to win, as in Wasserstein GAN, but with a min-max approach rather than averaging. Batch processing has its equivalent in neural networks, where it is called the same. The first layer in NN architecture corresponds to the first run in the resampling algorithm. Using a second run is similar to having a second (deep) layer in deep neural networks. The quality of the synthetization may be poor even if the loss function reaches zero: this happens when the loss is not a good proxy to the full multivariate KS distance used to evaluate the quality; then it requires working with a different loss function. Or the algorithm may fail to reach a zero loss, getting stuck in a local minimum. In that case, you should change the loss function or the hyperparameters, or allowing the loss to go up and down with a general downward trend. This mechanism is sometimes referred to as simulated annealing [8]. The fast computations, just like in GAN, are based on efficient use of tensors. Features causing problems in GAN (with high entropy or highly correlated to other features) cause similar problems in NoGAN2. It can be addressed using data transforms (called transformers in LLM) such as feature decorrelation, scaling, and PCA, followed by inverse transforms. All these properties make NoGAN2 a good sandbox to test and improve generative adversarial networks.

Despite the similarities, there are major differences, besides the absence of gradient descent. First, NoGAN2 is much faster, thought it can be tempting to use a large number of iterations, to increase the number of swaps (the analog of neuron activation). It usually improves the results, although only marginally after a while. A better strategy is to use batches. Compared to GAN, the convergence issues are much less pronounced. Indeed, I managed to synthesize all my datasets rather quickly and with superior quality, consistently. NoGAN2 is also much more stable, and easier to fine-tune because of the intuitive nature of the hyperparameters. It illustrates explainable AI. Because of this, it leads to auto-tuning, where fine-tuning is automated, possibly with reinforcement learning, in little time. Ideally, you want to use the KS distance as your loss function, rather than a proxy mimicking the approximation of a multivariate function by the first few terms of its Taylor series.

The challenge is how to design an efficient architecture to achieve this goal. The upcoming NoGAN3 algorithm will solve this problem, using bin counts in the loss function as in the first NoGAN [5], rather than correlations between transformed features.

Finally, NoGAN2 starts with an initial synthetization where all the features, taken separately, are perfectly replicated but lacking the cross-dependencies structure. It gives NoGAN2 a good head start, as all univariate statistical summaries are preserved throughout the algorithm. The deep resampling consists of reconstructing these interdependencies. If the loss function has one term only, then NoGAN2 is simply the copula method, implemented differently. The addition of a second term allows you to replicate not just feature correlations, but much more complex dependencies. A second or third run makes it an hierarchical Bayesian model.

4 Python implementation

The code in this section corresponds to `DeepResampling_students.py` on GitHub, here. There are other examples in the same folder, all starting with `DeepResampling`, corresponding to other case studies, each featuring a different set of hyperparameters. There is also a Jupyter notebook, available here. The KS distance is computed using the GenAI-Evaluation library. You can install it like any other Python library.

```
import numpy as np
import pandas as pd

import matplotlib.pyplot as plt
import matplotlib as mpl
import genai_evaluation as ge
from matplotlib import pyplot
from statsmodels.distributions.empirical_distribution import ECDF
import warnings
warnings.simplefilter("ignore")


#--- [1] read data and only keep features and observations we want

def category_to_integer(category):
   if category == 'Enrolled':
      integer = 1
   elif category == 'Dropout':
      integer = 0
   else:
      integer = 2
   return(integer)

#- [1.1] read data

url = "https://raw.githubusercontent.com/VincentGranville/Main/main/students.csv"
# data = pd.read_csv('students_C2_full_nogan.csv')
data = pd.read_csv(url)
print(data)

# all features used here

features = [
   'Application_mode',            # 0, categorical
   'Course',                      # 1, categorical
   'Previous_qualification_grade', # 2, ordinal
   'Mother_qualification',        # 3, categorical
   'Father_qualification',        # 4, categorical
   'Mother_occupation',           # 5, categorical
   'Father_occupation',           # 6, categorical
   'Admission_grade',             # 7, ordinal
   'Tuition_fees_up_to_date',     # 8, binary
   'Age_at_enrollment',           # 9, ordinal [outliers]
   'Curricular_units_1st_sem_evaluations', # 10, ordinal [0 = missing]
   'Curricular_units_1st_sem_approved', # 11, ordinal [0 = missing]
   'Curricular_units_1st_sem_grade', # 12, ordinal [0 = missing]
   'Curricular_units_2nd_sem_enrolled', # 13, ordinal [0 = missing]
```

```
    'Curricular_units_2nd_sem_evaluations', # 14, ordinal [0 = missing]
    'Curricular_units_2nd_sem_approved', # 15, ordinal [0 = missing]
    'Curricular_units_2nd_sem_grade', # 16, ordinal [0 = missing]
    'Unemployment_rate',           # 17, ordinal
    'Inflation_rate',              # 18, ordinal, can be < 0
    'GDP',                         # 19, ordinal, can be < 0
    'Target'                       # 20, categorical [outcome]
 ]

data['Target'] = data['Target'].map(category_to_integer)

# remove rows with missing values
data = data[data['Curricular_units_1st_sem_evaluations'] != 0]

print(data.head())
print (data.shape)
print (data.columns)

#- [1.2] set seed for replicability

pd.core.common.random_state(None)
seed = 106 ## 105
np.random.seed(seed)

#- [1.3] select features

data = data[features]
data = data.sample(frac = 1) # shuffle rows to break artificial sorting

#- [1.4] split real dataset into training and validation sets

data_training = data.sample(frac = 0.5)
data_validation = data.drop(data_training.index)
data_training.to_csv('training_vg2.csv')
data_validation.to_csv('validation_vg2.csv')
data_train = pd.DataFrame.to_numpy(data_training)

nobs = len(data_training)
n_features = len(features)


#--- [2] create initial synthetic data

def create_initial_synth(nobs_synth):

    eps = 0.000000001
    n_features = len(features)
    data_synth = np.empty(shape=(nobs_synth,n_features))

    for i in range(nobs_synth):
        pc = np.random.uniform(0, 1 + eps, n_features)
        for k in range(n_features):
            label = features[k]
            data_synth[i, k] = np.quantile(data_training[label], pc[k], axis=0)
    return(data_synth)


#--- [3] loss functions Part 1

def compute_univariate_stats():

    # 'dt' for training data, 'ds' for synth. data

    # for first tem in loss function
    dt_mean = np.mean(data_train, axis=0)
    dt_stdev = np.std(data_train, axis=0)
```

```
    ds_mean = np.mean(data_synth, axis=0)
    ds_stdev = np.std(data_synth, axis=0)

    # for g(arr)
    dt_mean1 = np.mean(g(data_train), axis=0)
    dt_stdev1 = np.std(g(data_train), axis=0)
    ds_mean1 = np.mean(g(data_synth), axis=0)
    ds_stdev1 = np.std(g(data_synth), axis=0)

    # for f(arr)
    dt_mean2 = np.mean(h(data_train), axis=0)
    dt_stdev2 = np.std(h(data_train), axis=0)
    ds_mean2 = np.mean(h(data_synth), axis=0)
    ds_stdev2 = np.std(h(data_synth), axis=0)

    values = [dt_mean, dt_stdev, ds_mean, ds_stdev,
            dt_mean1, dt_stdev1, ds_mean1, ds_stdev1,
            dt_mean2, dt_stdev2, ds_mean2, ds_stdev2]
    return(values)

def initialize_cross_products_tables():

    # the core structure for fast computation when swapping 2 values
    # 'dt' for training data, 'ds' for synth. data
    # 'prod' is for 1st term in loss, 'prod12' for 2nd term

    dt_prod = np.empty(shape=(n_features,n_features))
    ds_prod = np.empty(shape=(n_features,n_features))
    dt_prod12 = np.empty(shape=(n_features,n_features))
    ds_prod12 = np.empty(shape=(n_features,n_features))

    for k in range(n_features):
        for l in range(n_features):
            dt_prod[l, k] = np.dot(data_train[:,l], data_train[:,k])
            ds_prod[l, k] = np.dot(data_synth[:,l], data_synth[:,k])
            dt_prod12[l, k] = np.dot(g(data_train[:,l]), h(data_train[:,k]))
            ds_prod12[l, k] = np.dot(g(data_synth[:,l]), h(data_synth[:,k]))
    products = [dt_prod, ds_prod, dt_prod12, ds_prod12]
    return(products)


#--- [4] loss function Part 2: managing loss function

# Weights hyperparameter:
#
#   1st value is for 1st term in loss function, 2nd value for 2nd term
#   each value should be between 0 and 1, all adding to 1
#   works best when loss contributions from each term are about the same

#- [4.1] loss function contribution from features (k, l) jointly

# before calling functions in sections [4.1], [4.2] and [4.3], first intialize
# by calling compute_univariate_stats() and compute_cross_products() before;
# this initialization needs to be done only once at the beginning

def get_distance(k, l, weights):

    dt_prodn = dt_prod[k, l] / nobs
    ds_prodn = ds_prod[k, l] / nobs_synth
    dt_r = (dt_prodn - dt_mean[k]*dt_mean[l]) / (dt_stdev[k]*dt_stdev[l])
    ds_r = (ds_prodn - ds_mean[k]*ds_mean[l]) / (ds_stdev[k]*ds_stdev[l])

    dt_prodn12 = dt_prod12[k, l] / nobs
    ds_prodn12 = ds_prod12[k, l] / nobs_synth
    dt_r12 = (dt_prodn12 - dt_mean1[k]*dt_mean2[l]) / (dt_stdev1[k]*dt_stdev2[l])
    ds_r12 = (ds_prodn12 - ds_mean1[k]*ds_mean2[l]) / (ds_stdev1[k]*ds_stdev2[l])
```

```
    # dist = weights[0]*abs(dt_r - ds_r) + weights[1]*abs(dt_r12 - ds_r12)
    dist = max(weights[0]*abs(dt_r - ds_r), weights[1]*abs(dt_r12 - ds_r12))
    return(dist, dt_r, ds_r, dt_r12, ds_r12)

def total_distance(weights, flagParam):

    eval = 0
    max_dist = 0
    super_max = 0
    lmax = n_features

    for k in range(n_features):
        if symmetric:
            lmax = k
        for l in range(lmax):
            if l != k and flagParam[k] > 0 and flagParam[l] >0:
                values = get_distance(k, l, weights)
                dist2 = max(abs(values[1] - values[2]), abs(values[3] - values[4]))
                eval += values[0]
                if values[0] > max_dist:
                    max_dist = values[0]
                if dist2 > super_max:
                    super_max = dist2
    return(eval, max_dist, super_max)

#- [4.2] updated loss function when swapping rows idx1 and idx2 in feature k
#      contribution from feature l jointly with k

def get_new_distance(k, l, idx1, idx2, weights):

    tmp1_k = data_synth[idx1, k]
    tmp2_k = data_synth[idx2, k]
    tmp1_l = data_synth[idx1, l]
    tmp2_l = data_synth[idx2, l]

    #-- first term of loss function

    remove1 = tmp1_k * tmp1_l
    remove2 = tmp2_k * tmp2_l
    add1 = tmp1_k * tmp2_l
    add2 = tmp2_k * tmp1_l
    new_ds_prod = ds_prod[l, k] - remove1 - remove2 + add1 + add2

    dt_prodn = dt_prod[k, l] / nobs
    ds_prodn = new_ds_prod / nobs_synth
    dt_r = (dt_prodn - dt_mean[k]*dt_mean[l]) / (dt_stdev[k]*dt_stdev[l])
    ds_r = (ds_prodn - ds_mean[k]*ds_mean[l]) / (ds_stdev[k]*ds_stdev[l])

    #-- second term of loss function

    remove1 = g(tmp1_k) * h(tmp1_l)
    remove2 = g(tmp2_k) * h(tmp2_l)
    add1 = g(tmp1_k) * h(tmp2_l)
    add2 = g(tmp2_k) * h(tmp1_l)
    new_ds_prod12 = ds_prod12[k, l] - remove1 - remove2 + add1 + add2

    dt_prodn12 = dt_prod12[k, l] / nobs
    ds_prodn12 = new_ds_prod12 / nobs_synth
    dt_r12 = (dt_prodn12 - dt_mean1[k]*dt_mean2[l]) / (dt_stdev1[k]*dt_stdev2[l])
    ds_r12 = (ds_prodn12 - ds_mean1[k]*ds_mean2[l]) / (ds_stdev1[k]*ds_stdev2[l])

    #--

    # new_dist = weights[0]*abs(dt_r - ds_r) + weights[1]*abs(dt_r12 - ds_r12)
    new_dist = max(weights[0]*abs(dt_r - ds_r), weights[1]*abs(dt_r12 - ds_r12))
```

```
    return(new_dist, dt_r, ds_r, dt_r12, ds_r12)


#- [4.3] update prod tables after swapping rows idx1 and idx2 in feature k
#      update impacting feature l jointly with k

def update_product(k, l, idx1, idx2):

    tmp1_k = data_synth[idx1, k]
    tmp2_k = data_synth[idx2, k]
    tmp1_l = data_synth[idx1, l]
    tmp2_l = data_synth[idx2, l]

    #-- first term of loss function

    remove1 = tmp1_k * tmp1_l
    remove2 = tmp2_k * tmp2_l
    add1 = tmp1_k * tmp2_l
    add2 = tmp2_k * tmp1_l
    ds_prod[k, l] = ds_prod[k, l] - remove1 - remove2 + add1 + add2
    ds_prod[l, k] = ds_prod[k, l]

    #-- second term of loss function

    remove1 = g(tmp1_k) * h(tmp1_l)
    remove2 = g(tmp2_k) * h(tmp2_l)
    add1 = g(tmp1_k) * h(tmp2_l)
    add2 = g(tmp2_k) * h(tmp1_l)
    ds_prod12[k, l] = ds_prod12[k, l] - remove1 - remove2 + add1 + add2

    remove1 = h(tmp1_k) * g(tmp1_l)
    remove2 = h(tmp2_k) * g(tmp2_l)
    add1 = h(tmp1_k) * g(tmp2_l)
    add2 = h(tmp2_k) * g(tmp1_l)
    ds_prod12[l, k] = ds_prod12[l, k] - remove1 - remove2 + add1 + add2

    return()


#--- [5] feature sampling

def sample_feature(mode, hyperParameter):

    # Randomly pick up one column (a feature) to swap 2 values from 2 random rows
    # One feature is assumed to be in the right order, thus ignored

    if mode == 'Equalized':
        u = np.random.uniform(0, 1)
        cutoff = hyperParam[0]
        feature = 0
        while cutoff < u:
            feature += 1
            cutoff += hyperParam[feature]
    else:
        feature = np.random.randint(1, n_features) # ignore feature 0
    return(feature)


#--- [6] functions: deep synthetization, plot history, print stats

#- [6.1] main function

def deep_resampling(hyperParameter, run, loss_type, n_batches,
                    n_iter, nobs_synth, weights, flagParam, mode):

    # main function
```

```
batch = 0
batch_size = nobs_synth // n_batches
niter_per_batch = n_iter // n_batches
lower_row = 0
upper_row = batch_size
nswaps = 0
cgain = 0 # cumulative gain

arr_swaps = []
arr_history_quality = []
arr_history_max_dist = []
arr_time = []
print()

for iter in range(n_iter):

   k = sample_feature(mode, hyperParameter)
   batch = iter // niter_per_batch
   lower_row = batch * batch_size
   upper_row = lower_row + batch_size
   idx1 = np.random.randint(lower_row, upper_row) % nobs_synth
   tmp1 = data_synth[idx1, k]
   tmp2 = tmp1
   counter = 0
   while tmp2 == tmp1 and counter < 20:
      idx2 = np.random.randint(lower_row, upper_row) % nobs_synth
      tmp2 = data_synth[idx2, k]
      counter += 1

   g_param = 0.5
   h_param = g_param

   delta = 0
   delta2 = 0
   for l in range(n_features):
      if l != k and flagParam[l] > 0:
         values = get_distance(k, l, weights)
         delta += values[0]
         if values[0] > delta2:
            delta2 = values[0]
         if not symmetric: # if functions g, h are different
            values = get_distance(l, k, weights)
            delta += values[0]
            if values[0] > delta2:
               delta2 = values[0]

   new_delta = 0
   new_delta2 = 0
   for l in range(n_features):
      if l != k and flagParam[l] > 0:
         values = get_new_distance(k, l, idx1, idx2, weights)
         new_delta += values[0]
         if values[0] > new_delta2:
            new_delta2 = values[0]
         if not symmetric: # if functions g, h are different
            values = get_new_distance(l, k, idx1, idx2, weights)
            new_delta += values[0]
            if values[0] > new_delta2:
               new_delta2 = values[0]

   if loss_type == 'sum_loss':
      gain = delta - new_delta
   elif loss_type == 'max_loss':
      gain = delta2 - new_delta2
   if gain > 0:
```

```
            cgain += gain
            for l in range(n_features):
               if l != k:
                  update_product(k, l, idx1, idx2)
                  # update_product(l, k, idx1, idx2)
            data_synth[idx1, k] = tmp2
            data_synth[idx2, k] = tmp1
            nswaps += 1

         if iter % 500 == 0:
            quality, max_dist, super_max = total_distance(weights, flagParam)
            arr_swaps.append(nswaps)
            arr_history_quality.append(quality)
            arr_history_max_dist.append(max_dist)
            arr_time.append(iter)
            if iter % 5000 == 0:
               print("Iter: %6d Distance: %8.4f SupDist: %8.4f Gain: %8.4f Swaps: %6d"
                     %(iter, quality, super_max, cgain, nswaps))

   return(nswaps, arr_swaps, arr_history_quality, arr_history_max_dist, arr_time)

#- [6.2] save synthetic data, show some stats

def evaluate_and_save(filename, weights, run, flagParam):

   print("\nMetrics after deep resampling\n")
   quality, max_dist, super_max = total_distance(weights, flagParam)
   print("Distance: %8.4f" %(quality))
   print("Max Dist: %8.4f" %(max_dist))

   data_synthetic = pd.DataFrame(data_synth, columns = features)
   data_synthetic.to_csv(filename)
   print("\nSynthetic data, first 10 rows\n",data_synthetic.head(10))

   print("\nBivariate feature correlation:")
   print("....dt_xx for training set, ds_xx for synthetic data")
   print("....xx_r for correl[k, l], xx_r12 for correl[g(k), h(l)]\n")
   print("%2s %2s %8s %8s %8s %8s %8s"
         % ('k', 'l', 'dist', 'dt_r', 'ds_r', 'dt_r12', 'ds_r12'))
   print("-------------------------------------------------")
   for k in range(n_features):
      for l in range(n_features):
         condition = (flagParam[k] >0 and flagParam[l] > 0)
         if k != l and condition:
            values = get_distance(l, k, weights)
            dist = values[0]
            dt_r = values[1] # training, 1st term of loss function
            ds_r = values[2] # synth., 1st term of loss function
            dt_r12 = values[3] # training, 2nd term of loss function
            ds_r12 = values[4] # synth., 2nd term of loss function
            print("%2d %2d %8.4f %8.4f %8.4f %8.4f %8.4f"
                  % (k, l, dist, dt_r, ds_r, dt_r12, ds_r12))
   return()

#- [6.3] plot history of loss function, and cumulated number of swaps

def plot_history(history):

   arr_swaps = history[1]
   arr_history_quality = history[2]
   arr_history_max_dist = history[3]
   arr_time = history[4]

   mpl.rcParams['axes.linewidth'] = 0.3
   plt.rc('xtick',labelsize=7)
   plt.rc('ytick',labelsize=7)
```

```
    plt.xticks(fontsize=7)
    plt.yticks(fontsize=7)
    plt.subplot(1, 2, 1)
    plt.plot(arr_time, arr_swaps, linewidth = 0.3)
    plt.legend(['cumulated swaps'], fontsize="7",
       loc ="upper center", ncol=1)
    plt.subplot(1, 2, 2)
    plt.plot(arr_time, arr_history_quality, linewidth = 0.3)
    # plt.plot(arr_time, arr_history_max_dist, linewidth = 0.3)
    plt.legend(['distance'], fontsize="7",
       loc ="upper center", ncol=1)
    plt.show()
    return()


#--- [7] initializations

#- create intitial synthetization

nobs_synth = 2000
data_synth = create_initial_synth(nobs_synth)

#- specify 2nd part of loss function (argument is a number or array)

# do not use g(arr) = f(arr) = arr: this is pre-built already as 1st term in loss fct
# these two functions f, g are for the second term in the loss function

def g(arr):
    return(1/(0.01 + np.absolute(arr)))
def h(arr):
    return(1/(0.01 + np.absolute(arr)))

symmetric = True # set to True if functions g and h are identical
# 'symmetric = True' twice as fast as 'symmetric = False'

#- initializations: product tables and univariate stats

products = initialize_cross_products_tables()
dt_prod = products[0]
ds_prod = products[1]
dt_prod12 = products[2]
ds_prod12 = products[3]

values = compute_univariate_stats()
dt_mean = values[0]
dt_stdev = values[1]
ds_mean = values[2]
ds_stdev = values[3]
dt_mean1 = values[4]
dt_stdev1 = values[5]
ds_mean1 = values[6]
ds_stdev1 = values[7]
dt_mean2 = values[8]
dt_stdev2 = values[9]
ds_mean2 = values[10]
ds_stdev2 = values[11]


#--- [8] deep resampling

mode = 'Equalized' # options: 'Standard', 'Equalized'
eps2 = 0.0

#- first run

run = 1
```

```
n_iter = 500001
n_batches = 8
loss_type = 'sum_loss' # options: 'max_loss' or 'sum_loss'
weights = [0.90, 0.10]
hyperParam = np.zeros(len(features))
hyperParam[8] = 1
hyperParam[10] = 1
hyperParam[11] = 1
hyperParam[12] = 1
hyperParam[13] = 1
hyperParam[14] = 1
hyperParam[15] = 1
hyperParam[16] = 1
hyperParam = hyperParam / np.sum(hyperParam)
flagParam = np.ones(len(features))
history = deep_resampling(hyperParam, run, loss_type, n_batches, n_iter,
                    nobs_synth, weights, flagParam, mode)
evaluate_and_save('synth_vg2.csv', weights, run, flagParam)
plot_history(history)

#- second run

run = 2
n_iter = 500001
n_batches = 8
loss_type = 'sum_loss' # options: 'max_loss' or 'sum_loss'
weights = [0.90, 0.10]
hyperParam = np.ones(len(features))
hyperParam[8] = 0
hyperParam[10] = 0
hyperParam[11] = 0
hyperParam[12] = 0
hyperParam[13] = 0
hyperParam[14] = 0
hyperParam[15] = 0
hyperParam[16] = 0
hyperParam = hyperParam / np.sum(hyperParam)
flagParam = np.ones(len(features))
history = deep_resampling(hyperParam, run, loss_type, n_batches, n_iter,
                    nobs_synth, weights, flagParam, mode)
evaluate_and_save('synth_vg2.csv', weights, run, flagParam)
plot_history(history)


#--- [9] Evaluation synthetization using joint ECDF & Kolmogorov-Smirnov distance

#      dataframes: df = synthetic; data = real data,
#      compute multivariate ecdf on validation set, sort it by value (from 0 to 1)

print("\nMultivariate ECDF computations...\n")
n_nodes = 1000 # number of random locations in feature space, where ecdf is computed
seed = 50
np.random.seed(seed)

df_validation = pd.DataFrame(data_validation, columns = features)
df_synthetic = pd.DataFrame(data_synth, columns = features)
df_training = pd.DataFrame(data_train, columns = features)
query_lst, ecdf_val, ecdf_synth = ge.multivariate_ecdf(df_validation, df_synthetic,
    n_nodes, verbose = True)
query_lst, ecdf_val, ecdf_train = ge.multivariate_ecdf(df_validation, df_training,
    n_nodes, verbose = True)

ks = ge.ks_statistic(ecdf_val, ecdf_synth)
ks_base = ge.ks_statistic(ecdf_val, ecdf_train)
print("Test ECDF Kolmogorof-Smirnov dist. (synth. vs valid.): %6.4f" %(ks))
print("Base ECDF Kolmogorof-Smirnov dist. (train. vs valid.): %6.4f" %(ks_base))
```

```
#--- [10] visualizations (based on MatPlotLib version: 3.7.1)

def vg_scatter(df, feature1, feature2, counter):

   # customized plots, subplot position based on counter

   label = feature1 + " vs " + feature2
   x = df[feature1].to_numpy()
   y = df[feature2].to_numpy()
   plt.subplot(3, 2, counter)
   plt.scatter(x, y, s = 0.1, c ="blue")
   plt.xlabel(label, fontsize = 7)
   plt.xticks([])
   plt.yticks([])
   #plt.ylim(0,70000)
   #plt.xlim(18,64)
   return()

def vg_histo(df, feature, counter):

   # customized plots, subplot position based on counter

   y = df[feature].to_numpy()
   plt.subplot(2, 3, counter)
   min = np.min(y)
   max = np.max(y)
   binBoundaries = np.linspace(min, max, 30)
   plt.hist(y, bins=binBoundaries, color='white', align='mid',edgecolor='red',
           linewidth = 0.3)
   plt.xlabel(feature, fontsize = 7)
   plt.xticks([])
   plt.yticks([])
   return()

mpl.rcParams['axes.linewidth'] = 0.3

#- [10.1] scatterplots

dfs = pd.read_csv('synth_vg2.csv')
dfv = pd.read_csv('validation_vg2.csv')
vg_scatter(dfs, features[0], features[1], 1)
vg_scatter(dfv, features[0], features[1], 2)
vg_scatter(dfs, features[0], features[2], 3)
vg_scatter(dfv, features[0], features[2], 4)
vg_scatter(dfs, features[1], features[2], 5)
vg_scatter(dfv, features[1], features[2], 6)
plt.show()

#- [10.2] histograms

dfs = pd.read_csv('synth_vg2.csv')
dfv = pd.read_csv('validation_vg2.csv')
vg_histo(dfs, features[0], 1)
vg_histo(dfs, features[1], 2)
vg_histo(dfs, features[2], 3)
vg_histo(dfv, features[0], 4)
vg_histo(dfv, features[1], 5)
vg_histo(dfv, features[2], 6)
plt.show()
```

References

[1] Fabiola Banfi, Greta Cazzaniga, and Carlo De Michele. Nonparametric extrapolation of extreme quantiles: a comparison study. *Stochastic Environmental Research and Risk Assessment*, 36:1579–1596, 2022. [Link]. 5

[2] Marc G. Bellemare et al. The Cramer distance as a solution to biased Wasserstein gradients. *Preprint*, pages 1–20, 2017. arXiv:1705.10743 [Link]. 8

[3] Wei Chen and Mark Fuge. Synthesizing designs with interpart dependencies using hierarchical generative adversarial networks. *Journal of Mechanical Design*, 141:1–11, 2019. [Link]. 4

[4] Vincent Granville. Generative AI: Synthetic data vendor comparison and benchmarking best practices. *Preprint*, pages 1–13, 2023. MLTechniques.com [Link]. 1, 8

[5] Vincent Granville. Generative AI technology break-through: Spectacular performance of new synthesizer. *Preprint*, pages 1–16, 2023. MLTechniques.com [Link]. 1, 3, 4, 7, 8, 10, 12

[6] Vincent Granville. How to fix a failing generative adversarial network. *Preprint*, pages 1–10, 2023. MLTechniques.com [Link]. 4, 7

[7] Vincent Granville. *Synthetic Data and Generative AI*. Elsevier, 2024. [Link]. 4

[8] Vincent Granville, Mirko Krivanek, and Jean-Paul Rasson. Simulated annealing: A proof of convergence. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 16:652–656, 1996. 11

[9] Jogendra Nath Kundu et al. GAN-Tree: An incrementally learned hierarchical generative framework for multi-modal data distributions. *IEEE/CVF International Conference on Computer Vision*, pages 8190–8199, 2019. arXiv:1908.03919 [Link]. 4

[10] Nicolas Langrené and Xavier Warin. Fast multivariate empirical cumulative distribution function with connection to kernel density estimation. *Computational Statistics & Data Analysis*, 162:1–16, 2021. [Link]. 10

[11] Tengyuan Liang. Estimating certain integral probability metric (IPM) is as hard as estimating under the IPM. *Preprint*, pages 1–15, 2019. arXiv:1911.00730 [Link]. 10

[12] Michael Naaman. On the tight constant in the multivariate Dvoretzky–Kiefer–Wolfowitz inequality. *Statistics & Probability Letters*, 173:1–8, 2021. [Link]. 10

[13] Bharath Sriperumbudur et al. On the empirical estimation of integral probability metrics. *Electronic Journal of Statistics*, pages 1550–1599, 2012. [Link]. 10

[14] Chang Su, Linglin Wei, and Xianzhong Xie. Churn prediction in telecommunications industry based on conditional Wasserstein GAN. *IEEE International Conference on High Performance Computing, Data, and Analytics*, pages 186–191, 2022. IEEE HiPC 2022 [Link]. 4

[15] Jinsung Yoon et al. GAIN: Missing data imputation using generative adversarial nets. *Preprint*, pages 1–10, 2018. arXiv:1806.02920 [Link]. 4