

Chapter 2

Fast, efficient NoGAN Tabular Data Synthesis

Here I present two different versions of the best tabular data synthesizer on the market, along with a class of evaluation metrics found nowhere else: the full multivariate Kolmogorov-Smirnov distance between the two empirical cumulative distributions, normalized for the dimension, to compare the generated synthetization with the real data it is mimicking. I also discuss an efficient, adaptive loss function based on the model evaluation metric, as a better alternative to standard loss such as the mean squared error. The material is introduced as projects with solutions, along with real world case studies, applications, illustrations and Python code.

2.1 Fast, high-quality NoGAN synthesizer for tabular data

This project features a very fast synthesizer for tabular data, consistently leading to better synthetizations than those produced by [generative adversarial networks](#) (GAN). It has similarities to [XGboost](#), and does not require fine-tuning. Also, it epitomizes intuitive and [explainable AI](#). For the full description of the technology, see [\[19\]](#). The purpose of this project is to further optimize this new method. Evaluation of the generated data is based on the [multivariate empirical distribution](#) (ECDF), capturing all the patterns found in the real data, spanning across multiple features, categorical and numerical. The full joint ECDF has never been implemented in production mode due to computational complexity. Here it is, for the first time. To avoid [overfitting](#), the real data is split into two parts: the training set to build the synthesizer, and the [validation set](#) to check how it performs outside the training set. We test it on a well-known telecom dataset.

2.1.1 Project description

The project is based on the material [in this paper](#). The password to download the article is MLT12289058. I included the original source code in section [2.1.3](#). It is also on GitHub, [here](#). The goal here is to improve the methodology. The first two steps focus on the multivariate empirical distributions (ECDF), at the core of the [Kolmogorov-Smirnov distance](#) (KS) that evaluates the quality of the synthetization.

The project consists of the following steps:

Step 1: ECDF scatterplot. Create a Jupyter notebook for the code in section [2.1.3](#), and run it “as is”. The code automatically loads the telecom dataset. Section [\[4.3\]](#) in the code produces the scatterplot for the ECDF values (empirical distribution functions): the X-axis and Y-axis represent ECDF values respectively for the real (validation set) and synthetic component: each dot corresponds to a computation of the two ECDFs at a specific random argument, called “location” in the feature space. In particular (`ecdf_real1, ecdf_synth1`) is based on transformed locations, so that the scatterplot is better spread out along the diagonal: see left plot in Figure [2.1](#), compared to the middle plot without the transformation. Apply the same transformation to the ECDF values, instead of the arguments, and plot the result. You should obtain something similar to the right plot in the Figure [2.1](#).

Step 2: Convergence of the KS distance. The KS distance is equal to $\max |F_v(z) - F_s(z)|$ over all locations z in the feature space, where $F_v(z)$ and $F_s(z)$ are the multivariate ECDFs, computed respectively on the validation set and the synthetic data. Here, the real data is split into two parts: the training set to produce the synthetic data, and the validation set to evaluate its quality. In practice, we sample `n_nodes` locations z to compute the ECDFs. Check out if `n_nodes=1000` (the value used in the code) is large enough: see if the KS distance stays roughly the same by increasing `n_nodes` from 10^3 to 10^4 and 10^5 .

See also how the KS distance (denoted as `KS_max` in the code) depends on the number of observations, both in the validation set and the synthetic data.

Step 3: From uniform to Gaussian sampling. The core of the **NoGAN** architecture consists of fixed-size multivariate bins covering all the points in the training set, whether the features are categorical or numerical. For synthetization, a random number of points is generated in each bin: these numbers follow a very specific **multinomial distribution**. In each bin, synthetic observations are uniformly and independently generated. Bins are **hyperrectangles** in the feature space, with sides either parallel or perpendicular to the axes.

For any specific bin, the multivariate median computed on the training set is stored in the array `median`; the list of training set points lying in the bin is stored in `obs_list`, an array where each entry is a multidimensional observation from the training set. The upper and lower bounds of the bin (one per feature), are stored respectively in the arrays `L_bounds` and `U_bounds`, while `count` represents the number of points to generate, in the bin in question. All of this is located towards the bottom of section [2.3] in the code.

The generation of one synthetic observation vector uniformly distributed in the bin is performed separately for each component k (also called feature or dimension) by the instruction

```
new_obs[k] = np.random.uniform(L_bounds[k], U_bounds[k]).
```

In this step, you are asked to replace the uniform distribution by a Gaussian one, with the mean coinciding with the above median. The covariance matrix of the Gaussian may be diagonal for simplicity. About 95% of the generated Gaussian observations should lie within the bin. Those that don't are rejected (try later without **rejection sampling**). In order to produce the required `count` observations within the bin, you need to oversample to be able to meet that `count` after rejection. In addition, do it without as few loops as possible, using **vector operations** instead. You may also replace the nested loops to compute `new_obs[k]`, by vector operations.

Step 4: Speeding up the computations. To find the first value larger or equal to a pre-specified value `arr[idx]` in a sorted list `arr`, I use brute force, sequentially browsing the list until finding the value in question is found, with the following instruction:

```
while obs[k] >= arr[idx] and idx < bins_per_feature[k],
```

incrementing `idx` after each iteration. Replace the `while` loop by a dichotomic search. Measure the gain in computing time, after the change. In short, it improves **time complexity** from linear to logarithmic.

Step 5: Fine-tuning the hyperparameter vector. The main parameter in NoGAN is the vector $[n_1, \dots, n_d]$ named `bins_per_feature` in the code. Here d is the number of features or dimension of the problem, and n_k the desired number of intervals when binning feature k . For each feature, intervals are chosen so that they contain about the same number of observed values: wherever the density is high, intervals are short, and conversely. In the code, the **hyperparameter** is set to `[50, 40, 40, 4]` in section [1.5]. The last value is attached to a binary feature called “Churn”. If you change 4 to 3, there will be no observation with Churn equal to 1 in the synthetic data. Why, and how do you automatically determine the optimum value for this feature?

In the Python code, I only use 4 features, including the three numerical ones. But the telecom dataset contains many more. Add a few more features, and adjust the hyperparameter vector accordingly. For numerical features, small values in the hyperparameter result in artificial linear boundaries in the scatterplots in Figure 2.2 (produced in section [4.1] and [4.2] in the code). Illustrate this fact by reproducing Figure 2.2 but with a different hyperparameter. Can Gaussian sampling, discussed in **step 3**, fix this issue? Very large values in the hyperparameter fix this problem. But too large is not good. Why? Time permitting, you may want to optimize the hyperparameter vector using the smart grid search technique explained in [23].

Finally, for each feature, rather than using intervals based on constant **quantile** increments as in section [2.1] in the code, use arbitrary intervals. In other words, allow the user to provide his own `pc_table2`, rather than the default one based on the hyperparameter. Note that `pc_table2[k]` corresponds to feature k ; it is itself a sub-array with $n_k + 1$ elements, specifying the bounds of the binning intervals for the feature in question.

Step 6: Confidence intervals. This step is optional, and consists of four separate sub-projects.

- Find great hyperparameter vectors using for instance the **smart grid search** technique described in [23]. How do you define and measure “great” in this context?

- Using subsets of the training set, assess the impact of training set size on the KS distance. Reducing the training set while preserving the quality of the output, is a technique frequently used to speed up AI algorithms, see [22]. A more sophisticated version is called **data distillation**.
- Try 100 different seeds (the parameter `seed` in section [1.4] in the code) to generate 100 different synthetizations. Use the generated data to compute confidence intervals of various levels for various statistics (for instance, correlation between tenure and residues), based on the size of the training set.
- Test NoGAN on different datasets, or with much more than four features.

Note that the ECDFs take values between 0 and 1, as it estimates probabilities. Thus the KS distance – the maximum distance between the two ECDFs, synthetic vs validation – also takes values between 0 (best possible synthetization) and 1 (worst case). The dots in the scatterplots in Figure 2.1 should always be close to the main diagonal. When the two ECDFs are identical, the dots lie exactly on the main diagonal.

2.1.2 Solution

The solution to **step 1** consists of elevating the ECDFs `ecdf_real2` and `ecdf_synth2` (taking values between 0 and 1) at the power $1/d$ in section [4.3] in the code. Here d is the number of features, also called dimension, and denoted as `n_features`. The updated version of section [4.3] is on GitHub, [here](#). It produces the 3 plots in Figure 2.1, with the new one on the right-hand side. In case of perfect synthetization, all the dots are on the main diagonal.

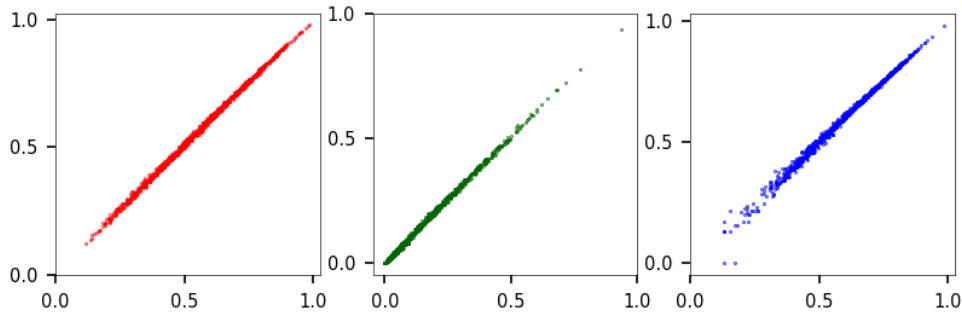


Figure 2.1: ECDF scatterplots (validation vs synthetic) computed three ways

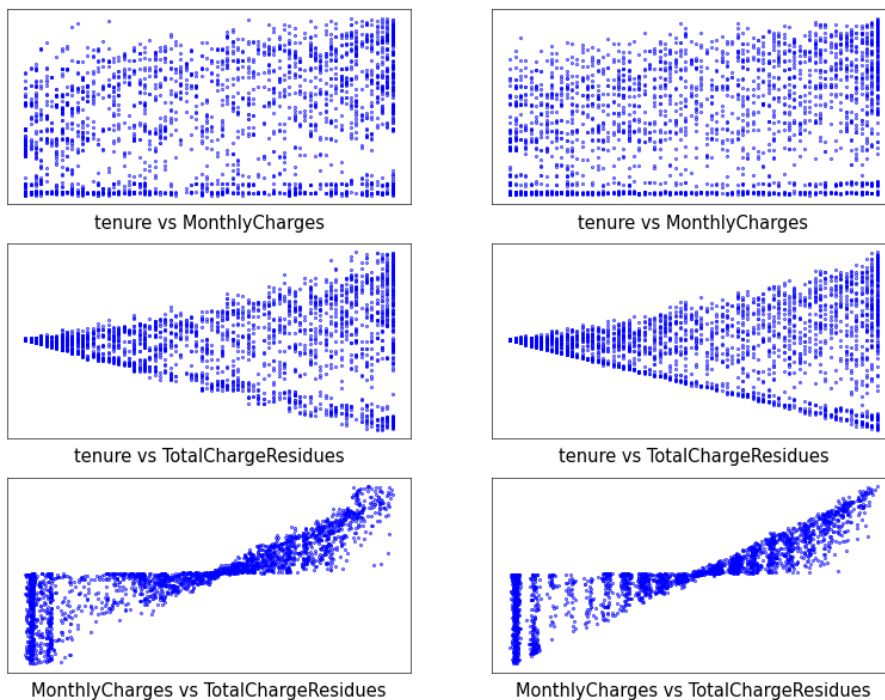


Figure 2.2: Feature scatterplots, synthetic (left) and validation dataset (right)

The NoGAN tab in `telecom.xlsx` features sample synthetic data. This spreadsheet is in the same folder, [here](#). The other tabs in this spreadsheet feature synthetizations obtained via [generative adversarial networks](#) (GAN), for comparison purposes. For more details, see my article “How to Fix a Failing Generative Adversarial Network” [21].

As for [Step 3](#), if you use Gaussian instead of uniform sampling within each multivariate bin, it will reduce edge effects in the synthesized data, especially if using non-truncated Gaussian deviates, with sampled points spilling into neighboring bins. To some extent, this is similar to using [diffusion](#) [Wiki] in neural network models. As an illustration of the edge effect, look at Figure 2.2: you can (barely) see some linear borders between different areas of the plot, in the left middle scatterplot. In particular, on the lower boundary of the cloud point. This happens when the values in the hyperparameter vector, for the features in question, are too low. Here the hyperparameter is `[50, 40, 40, 4]`, with 50 for “tenure”, and 40 for “residues” (the two features in the scatterplot in question). If you decrease these two values to (say) 15, the edge effect will be more pronounced. To the contrary, if you increase it to (say) 80, it won’t be noticeable. High values can lead to overfitting and should be avoided if possible. An implementation of Gaussian NoGAN can be found [here](#). Look at lines 142–150 and 192–200 in the code in question.

I now jump to one of the most important parts: [Step 5](#). I provided answers to some of the questions in the previous paragraph. To choose the hyperparameter vector, the basic rule is this: higher values leads to better synthetizations up to some extent; too high leads to overfitting. If one feature has several categories, and the proportion of observations in the smallest category is p , then the corresponding hyperparameter value must be an integer larger than $1/p$. Otherwise, the smallest category may not be generated in the synthesized data. In practice, for important data segments with very few observations in the training set (such as fraud), you may want to run a separate NoGAN.

Now answering [Step 6](#). First, a great hyperparameter vector is one resulting in a small KS distance. The smaller KS, the more faithful your synthetic data is. Then, regarding [confidence intervals](#) (CI), the solution is as follows. To obtain a 90% CI for the correlation ρ between “tenure” and “residues” (the latter named `TotalChargeResidues` in the Python code), compute ρ on each of the 100 synthetizations (one per seed). The 5 and 95 percentiles computed on these ρ ’s, with the Numpy `quantile` function, are respectively the lower and upper bound of your CI. Finally, to test NoGAN on other datasets, try the circle, insurance, and diabetes datasets featured in my article comparing vendor products, available [here](#).

2.1.3 Python implementation

Explanations about the different steps, including a description of the main variables and tables, can be found in [19]. Compared to GAN, this implementation requires very few library functions and only three imports: Numpy, Statsmodels, and Pandas. This significantly reduces incompatibilities between library versions, and increases the chance that you can run it “as is” on any platform, without impacting your own environment. The code `NoGAN.py` is also available on GitHub, [here](#).

```
1 import numpy as np
2 import pandas as pd
3 import matplotlib.pyplot as plt
4 from matplotlib import pyplot
5 from statsmodels.distributions.empirical_distribution import ECDF
6
7 #--- [1] read data and only keep features and observations we want
8
9 #- [1.1] utility functions
10
11 def string_to_numbers(string):
12
13     string = string.replace("[", "")
14     string = string.replace("]", "")
15     string = string.replace(" ", "")
16     arr = string.split(',')
17     arr = [eval(i) for i in arr]
18     return(arr)
19
20 def category_to_integer(category):
21     if category == 'Yes':
22         integer = 1
23     elif category == 'No':
24         integer = 0
25     else:
26         integer = 2
27     return(integer)
28
```

```

29 #- [1.2] read data
30
31 url = "https://raw.githubusercontent.com/VincentGranville/Main/main/Telecom.csv"
32 data = pd.read_csv(url)
33 features = ['tenure', 'MonthlyCharges', 'TotalCharges', 'Churn']
34 data['Churn'] = data['Churn'].map(category_to_integer)
35 data['TotalCharges'].replace(' ', np.nan, inplace=True)
36 data.dropna(subset=['TotalCharges'], inplace=True) # remove missing data
37 print(data.head())
38 print (data.shape)
39 print (data.columns)
40
41 #- [1.3] transforming TotalCharges to TotalChargeResidues, add to dataframe
42
43 arr1 = data['tenure'].to_numpy()
44 arr2 = data['TotalCharges'].to_numpy()
45 arr2 = arr2.astype(float)
46 residues = arr2 - arr1 * np.sum(arr2) / np.sum(arr1) # also try arr2/arr1
47 data['TotalChargeResidues'] = residues
48
49 #- [1.4] set seed for replicability
50
51 pd.core.common.random_state(None)
52 seed = 105
53 np.random.seed(seed)
54
55 #- [1.5] initialize hyperparameters (bins_per_feature), select features
56
57 features = ['tenure', 'MonthlyCharges', 'TotalChargeResidues', 'Churn']
58 bins_per_feature = [50, 40, 40, 4]
59
60 bins_per_feature = np.array(bins_per_feature).astype(int)
61 data = data[features]
62 print(data.head())
63 print (data.shape)
64 print (data.columns)
65
66 #- [1.6] split real dataset into training and validation sets
67
68 data_training = data.sample(frac = 0.5)
69 data_validation = data.drop(data_training.index)
70 data_training.to_csv('telecom_training_vg2.csv')
71 data_validation.to_csv('telecom_validation_vg2.csv')
72
73 nobs = len(data_training)
74 n_features = len(features)
75 eps = 0.0000000001
76
77
78 #--- [2] create synthetic data
79
80 #- [2.1] create quantile table pc_table2, one row for each feature
81
82 pc_table2 = []
83 for k in range(n_features):
84     label = features[k]
85     incr = 1 / bins_per_feature[k]
86     pc = np.arange(0, 1 + eps, incr)
87     arr = np.quantile(data_training[label], pc, axis=0)
88     pc_table2.append(arr)
89
90 #- [2.2] create/update bin for each obs [layer 1]
91 #     Faster implementation: replace 'while' loop by dichotomic search
92
93 npdata = pd.DataFrame.to_numpy(data_training[features])
94 bin_count = {} # number of obs per bin
95 bin_obs = {} # list of obs in each bin, separated by "~", stored as a string
96 for obs in npdata:
97     key = []
98     for k in range(n_features):
99         idx = 0
100         arr = pc_table2[k] # percentiles for feature k
101         while obs[k] >= arr[idx] and idx < bins_per_feature[k]:
102             idx = idx + 1
103         idx = idx - 1 # lower bound for feature k in bin[key] attached to obs
104         key.append(idx)

```

```

105     skey = str(key)
106     if skey in bin_count:
107         bin_count[skey] += 1
108         bin_obs[skey] += "~" + str(obs)
109     else:
110         bin_count[skey] = 1
111         bin_obs[skey] = str(obs)
112
113     #- [2.3] generate nob_synth observations (if mode = FixedCounts, nob_synth = nob)
114
115     def random_bin_counts(n, bin_count):
116         # generate multinomial bin counts with same expectation as real counts
117         pvals = []
118         for skey in bin_count:
119             pvals.append(bin_count[skey]/nob)
120         return(np.random.multinomial(n, pvals))
121
122     def get_obs_in_bin(bin_obs, skey):
123         # get list of observations (real data) in bin skey, also return median
124         arr_obs = []
125         arr_obs_aux = (bin_obs[skey]).split('~')
126         for obs in arr_obs_aux:
127             obs = ' '.join(obs.split())
128             obs = obs.replace("[", "")
129             obs = obs.replace("[", "")
130             obs = obs.replace("]", "")
131             obs = obs.replace("]", "")
132             obs = obs.split(' ')
133             obs = (np.array(obs)).astype(float)
134             arr_obs.append(obs)
135         arr_obs = np.array(arr_obs)
136         median = np.median(arr_obs, axis = 0)
137         return(arr_obs, median)
138
139
140     mode = 'RandomCounts' # (options: 'FixedCounts' or 'RandomCounts')
141     if mode == 'RandomCounts':
142         nob_synth = nob
143         bin_count_random = random_bin_counts(nob_synth, bin_count)
144         ikey = 0
145
146     data_synth = []
147     bin_counter = 0
148
149     for skey in bin_count:
150
151         if mode == 'FixedCounts':
152             count = bin_count[skey]
153         elif mode == 'RandomCounts':
154             count = bin_count_random[ikey]
155             ikey += 1
156         key = string_to_numbers(skey)
157         L_bounds = []
158         U_bounds = []
159         bin_counter += 1
160
161         for k in range(n_features):
162             arr = pc_table2[k]
163             L_bounds.append(arr[key[k]])
164             U_bounds.append(arr[1 + key[k]])
165
166         # sample new synth obs (new_obs) in rectangular bin skey, uniformly
167         # try other distrib, like multivariate Gaussian around bin median
168         # the list of real observations in bin[skey] is stored in obs_list (numpy array)
169         # median is the vector of medians for all obs in bin skey
170
171         obs_list, median = get_obs_in_bin(bin_obs, skey) # not used in this version
172
173         for i in range(count):
174             new_obs = np.empty(n_features) # synthesized obs
175             for k in range(n_features):
176                 new_obs[k] = np.random.uniform(L_bounds[k], U_bounds[k])
177             data_synth.append(new_obs)
178
179     str_median = str(["%8.2f" % number for number in median])
180     str_median = str_median.replace("'", "")

```

```

181     print("bin ID = %5d | count = %5d | median = %s | bin key = %s"
182           %(bin_counter, bin_count[skey], str_median, skey))
183
184 data_synth = pd.DataFrame(data_synth, columns = features)
185
186 # apply floor function (not round) to categorical/ordinal features
187 data_synth['Churn'] = data_synth['Churn'].astype('int')
188 data_synth['tenure'] = data_synth['tenure'].astype('int')
189
190 print(data_synth)
191 data_synth.to_csv('telecom_synth_vg2.csv')
192
193
194 #--- [3] Evaluation synthetization using joint ECDF & Kolmogorov-Smirnov distance
195
196 # dataframes: df = synthetic; data = real data,
197 # compute multivariate ecdf on validation set, sort it by value (from 0 to 1)
198
199 #- [3.1] compute ecdf on validation set (to later compare with that on synth data)
200
201 def compute_ecdf(dataframe, n_nodes, adjusted):
202
203     # Monte-Carlo: sampling n_nodes locations (combos) for ecdf
204     # - adjusted correct for sparsity in high ecdf, but is sparse in low ecdf
205     # - non-adjusted is the other way around
206     # for faster computation: pre-compute percentiles for each feature
207     # for faster computation: optimize the computation of n_nodes SQL-like queries
208
209     ecdf = {}
210
211     for point in range(n_nodes):
212
213         if point % 100 == 0:
214             print("sampling ecdf, location = %4d (adjusted = %s):" % (point, adjusted))
215         combo = np.random.uniform(0, 1, n_features)
216         if adjusted:
217             combo = combo**(1/n_features)
218         z = [] # multivariate quantile
219         query_string = ""
220         for k in range(n_features):
221             label = features[k]
222             dr = data_validation[label]
223             percentile = combo[k]
224             z.append(eps + np.quantile(dr, percentile))
225             if k == 0:
226                 query_string += "{} <= {}".format(label, z[k])
227             else:
228                 query_string += " and {} <= {}".format(label, z[k])
229
230         countifs = len(data_validation.query(query_string))
231         if countifs > 0:
232             ecdf[str(z)] = countifs / len(data_validation)
233
234     ecdf = dict(sorted(ecdf.items(), key=lambda item: item[1]))
235
236     # extract table with locations (ecdf argument) and ecdf values:
237     # - cosmetic change to return output easier to handle than ecdf
238
239     idx = 0
240     arr_location = []
241     arr_value = []
242     for location in ecdf:
243         value = ecdf[location]
244         location = string_to_numbers(location)
245         arr_location.append(location)
246         arr_value.append(value)
247         idx += 1
248
249     print("\n")
250     return(arr_location, arr_value)
251
252
253 n_nodes = 1000 # number of random locations in feature space, where ecdf is computed
254 reseed = False
255 if reseed:
256     seed = 555

```

```

257     np.random.seed(seed)
258     arr_location1, arr_value1 = compute_ecdf(data_validation, n_nodes, adjusted = True)
259     arr_location2, arr_value2 = compute_ecdf(data_validation, n_nodes, adjusted = False)
260
261     #- [3.2] comparison: synthetic (based on training set) vs real (validation set)
262
263     def ks_delta(SyntheticData, locations, ecdf_ValidationSet):
264
265         # SyntheticData is a dataframe
266         # locations are the points in the feature space where ecdf is computed
267         # for the validation set, ecdf values are stored in ecdf_ValidationSet
268         # here we compute ecdf for the synthetic data, at the specified locations
269         # output ks_max in [0, 1] with 0 = best, 1 = worst
270
271         ks_max = 0
272         ecdf_real = []
273         ecdf_synth = []
274         for idx in range(len(locations)):
275             location = locations[idx]
276             value = ecdf_ValidationSet[idx]
277             query_string = ""
278             for k in range(n_features):
279                 label = features[k]
280                 if k == 0:
281                     query_string += "{} <= {}".format(label, location[k])
282                 else:
283                     query_string += " and {} <= {}".format(label, location[k])
284             countifs = len(SyntheticData.query(query_string))
285             synth_value = countifs / len(SyntheticData)
286             ks = abs(value - synth_value)
287             ecdf_real.append(value)
288             ecdf_synth.append(synth_value)
289             if ks > ks_max:
290                 ks_max = ks
291                 # print("location ID: %d | ecdf_real: %6.4f | ecdf_synth: %6.4f"
292                 #       %(idx, value, synth_value))
293         return(ks_max, ecdf_real, ecdf_synth)
294
295     df = pd.read_csv('telecom_synth_vg2.csv')
296     ks_max1, ecdf_real1, ecdf_synth1 = ks_delta(df, arr_location1, arr_value1)
297     ks_max2, ecdf_real2, ecdf_synth2 = ks_delta(df, arr_location2, arr_value2)
298     ks_max = max(ks_max1, ks_max2)
299     print("Test ECDF Kolmogorof-Smirnov dist. (synth. vs valid.): %6.4f" %(ks_max))
300
301     #- [3.3] comparison: training versus validation set
302
303     df = pd.read_csv('telecom_training_vg2.csv')
304     base_ks_max1, ecdf_real1, ecdf_synth1 = ks_delta(df, arr_location1, arr_value1)
305     base_ks_max2, ecdf_real2, ecdf_synth2 = ks_delta(df, arr_location2, arr_value2)
306     base_ks_max = max(base_ks_max1, base_ks_max2)
307     print("Base ECDF Kolmogorof-Smirnov dist. (train. vs valid.): %6.4f" %(base_ks_max))
308
309
310     #--- [4] visualizations
311
312     def vg_scatter(df, feature1, feature2, counter):
313
314         # customized plots, subplot position based on counter
315
316         label = feature1 + " vs " + feature2
317         x = df[feature1].to_numpy()
318         y = df[feature2].to_numpy()
319         plt.subplot(3, 2, counter)
320         plt.scatter(x, y, s = 0.1, c ="blue")
321         plt.xlabel(label, fontsize = 7)
322         plt.xticks([])
323         plt.yticks([])
324         #plt.ylim(0,70000)
325         #plt.xlim(18,64)
326         return()
327
328     def vg_histo(df, feature, counter):
329
330         # customized plots, subplot position based on counter
331
332         y = df[feature].to_numpy()

```

```

333     plt.subplot(2, 3, counter)
334     min = np.min(y)
335     max = np.max(y)
336     binBoundaries = np.linspace(min, max, 30)
337     plt.hist(y, bins=binBoundaries, color='white', align='mid', edgecolor='red',
338             linewidth = 0.3)
339     plt.xlabel(feature, fontsize = 7)
340     plt.xticks([])
341     plt.yticks([])
342     return()
343
344 import matplotlib.pyplot as plt
345 import matplotlib as mpl
346 mpl.rcParams['axes.linewidth'] = 0.3
347
348 #- [4.1] scatterplots for Churn = 'No'
349
350 dfs = pd.read_csv('telecom_synth_vg2.csv')
351 dfs.drop(dfs[dfs['Churn'] == 0].index, inplace = True)
352 dfv = pd.read_csv('telecom_validation_vg2.csv')
353 dfv.drop(dfv[dfv['Churn'] == 0].index, inplace = True)
354
355 vg_scatter(dfs, 'tenure', 'MonthlyCharges', 1)
356 vg_scatter(dfv, 'tenure', 'MonthlyCharges', 2)
357 vg_scatter(dfs, 'tenure', 'TotalChargeResidues', 3)
358 vg_scatter(dfv, 'tenure', 'TotalChargeResidues', 4)
359 vg_scatter(dfs, 'MonthlyCharges', 'TotalChargeResidues', 5)
360 vg_scatter(dfv, 'MonthlyCharges', 'TotalChargeResidues', 6)
361 plt.show()
362
363 #- [4.2] scatterplots for Churn = 'Yes'
364
365 dfs = pd.read_csv('telecom_synth_vg2.csv')
366 dfs.drop(dfs[dfs['Churn'] == 1].index, inplace = True)
367 dfv = pd.read_csv('telecom_validation_vg2.csv')
368 dfv.drop(dfv[dfv['Churn'] == 1].index, inplace = True)
369
370 vg_scatter(dfs, 'tenure', 'MonthlyCharges', 1)
371 vg_scatter(dfv, 'tenure', 'MonthlyCharges', 2)
372 vg_scatter(dfs, 'tenure', 'TotalChargeResidues', 3)
373 vg_scatter(dfv, 'tenure', 'TotalChargeResidues', 4)
374 vg_scatter(dfs, 'MonthlyCharges', 'TotalChargeResidues', 5)
375 vg_scatter(dfv, 'MonthlyCharges', 'TotalChargeResidues', 6)
376 plt.show()
377
378 #- [4.3] ECDF scatterplot: validation set vs. synth data
379
380 plt.xticks(fontsize=7)
381 plt.yticks(fontsize=7)
382 plt.scatter(ecdf_reall1, ecdf_synth1, s = 0.1, c ="blue")
383 plt.scatter(ecdf_real2, ecdf_synth2, s = 0.1, c ="blue")
384 plt.show()
385
386 #- [4.4] histograms, Churn = 'No'
387
388 dfs = pd.read_csv('telecom_synth_vg2.csv')
389 dfs.drop(dfs[dfs['Churn'] == 0].index, inplace = True)
390 dfv = pd.read_csv('telecom_validation_vg2.csv')
391 dfv.drop(dfv[dfv['Churn'] == 0].index, inplace = True)
392 vg_histo(dfs, 'tenure', 1)
393 vg_histo(dfs, 'MonthlyCharges', 2)
394 vg_histo(dfs, 'TotalChargeResidues', 3)
395 vg_histo(dfv, 'tenure', 4)
396 vg_histo(dfv, 'MonthlyCharges', 5)
397 vg_histo(dfv, 'TotalChargeResidues', 6)
398 plt.show()
399
400 #- [4.5] histograms, Churn = 'Yes'
401
402 dfs = pd.read_csv('telecom_synth_vg2.csv')
403 dfs.drop(dfs[dfs['Churn'] == 1].index, inplace = True)
404 dfv = pd.read_csv('telecom_validation_vg2.csv')
405 dfv.drop(dfv[dfv['Churn'] == 1].index, inplace = True)
406 vg_histo(dfs, 'tenure', 1)
407 vg_histo(dfs, 'MonthlyCharges', 2)
408 vg_histo(dfs, 'TotalChargeResidues', 3)

```

```
409 vg_histo(dfv, 'tenure', 4)
410 vg_histo(dfv, 'MonthlyCharges', 5)
411 vg_histo(dfv, 'TotalChargeResidues', 6)
412 plt.show()
```

2.2 Hierarchical Bayesian NoGAN for data synthesis

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? Very different from the tree-based NoGAN described in chapter 2.1, this new technology, abbreviated as NoGAN2, 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 hyperparameter 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 chapter is to show the spectacular performance of NoGAN2. 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.

2.2.1 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 chapter 2.1. 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 [18], 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 hyperparameters, 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**.

2.2.1.1 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**.

2.2.1.2 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, \dots, X_m for the training set, and X'_1, \dots, 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 $Q'_2[i, j] = \sum g_2(X'_i) * h_2(X'_j)$, $Q'_3[i, j, k] = \sum g_3(X'_i) * h_3(X'_j) * k_3(X'_k)$, 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

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

where

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

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 throughout 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 $\Delta_2[i, j] = \max(\alpha_1 \cdot \Delta_{21}[i, j], \alpha_2 \cdot \Delta_{22}[i, j])$ 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 $\tilde{\Delta}_2$ 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.

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.

2.2.1.3 Hyperparameters 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 [2.2.2](#), 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 [2.2.2](#).

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 [2.2.2](#) 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 [2.2.1.2](#).
- 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 $P = [p_1, \dots, p_m]$ 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 [11] and [40]. 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 [27]. 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.

2.2.2 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 [60], 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 chapter 9 in [26], were not good enough. It led to the creation of NoGAN discussed in chapter 2.1, and then NoGAN2 presented here. Both provide satisfactory results, in a short amount of time, and with little if any fine-tuning.

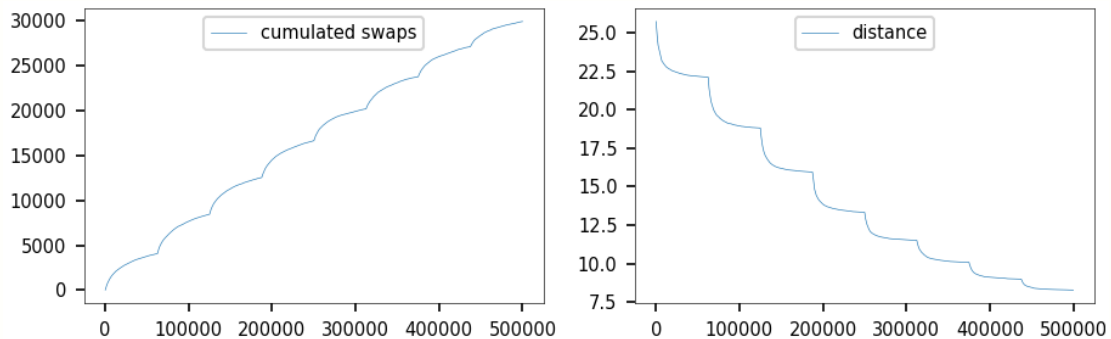


Figure 2.3: Number of swaps (left) and loss function (right) over time: first run

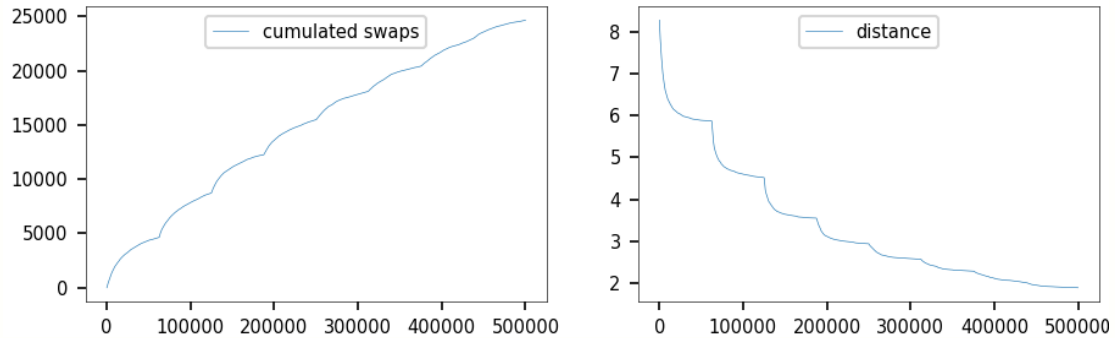


Figure 2.4: Number of swaps (left) and loss function (right) over time: second run

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 6.4.2 in [26]. 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 [70]. 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 **Kolmogorov-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 2.2.2.5

Before diving into the actual synthetizations in sections 2.2.2.1 and 2.2.2.2, 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 2.2.2.4. 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 [6].

2.2.2.1 Synthesizing the student dataset

The dataset is available on GitHub, [here](#). I also use it in the Python code in section 2.2.4. In this section, I explain some techniques to accelerate performance, both in terms of speed and quality of the synthetization. Figures 2.3 and 2.4 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 2.2.1.1. 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.

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 2.3 and 2.4, 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 2.7 (Telecom dataset) shows how slowly the loss function decreases after the initial drop, when using one batch only.

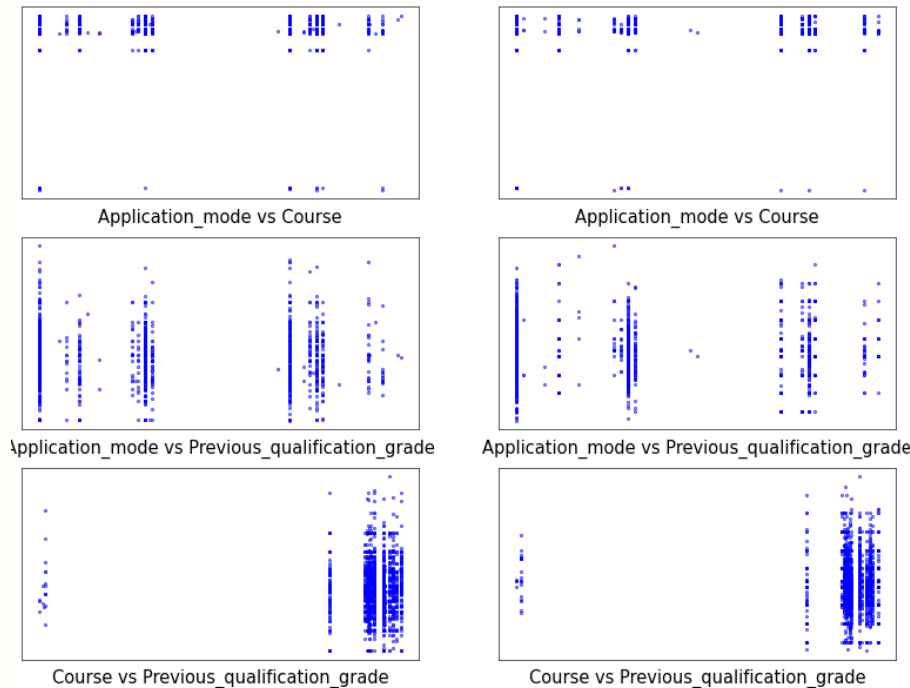


Figure 2.5: Synthetization (left) vs validation set (right), student dataset

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 2.6 always look very good. Thus it is more important to focus on the scatterplots in Figure 2.5. 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 2.2.4: 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.

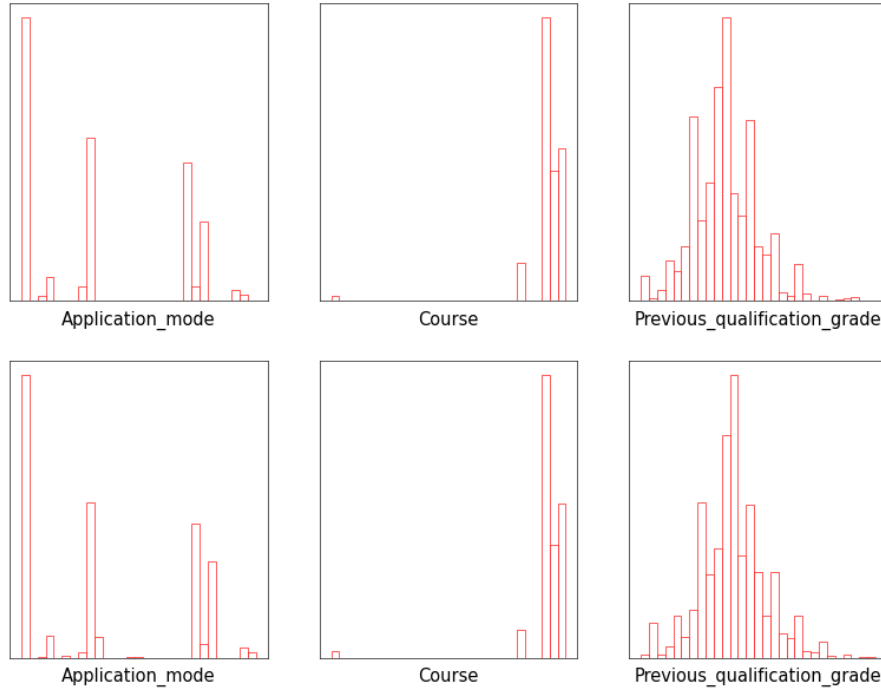


Figure 2.6: Synthetisation (top) vs validation set (bottom), student dataset

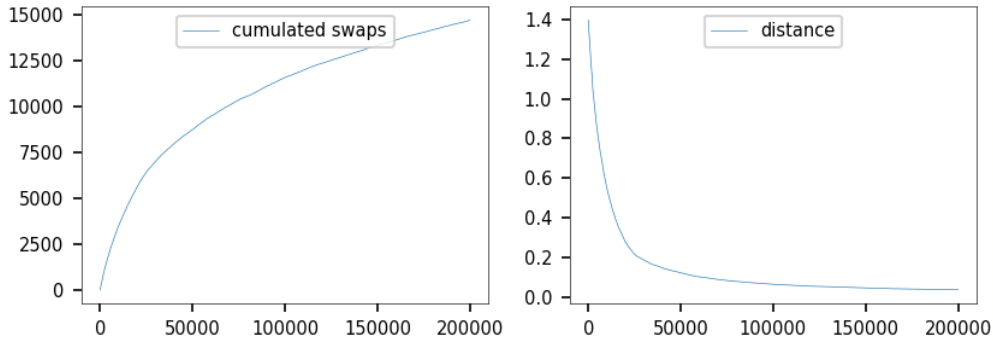


Figure 2.7: Number of swaps (left) and loss function (right) over time

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 2.2.2.5.

2.2.2.2 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 (chapter 2.1) matches the quality and speed of execution of NoGAN2. The details are in chapter 9 in [26] entitled “How to Fix a Failing Adversarial Network”. 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.

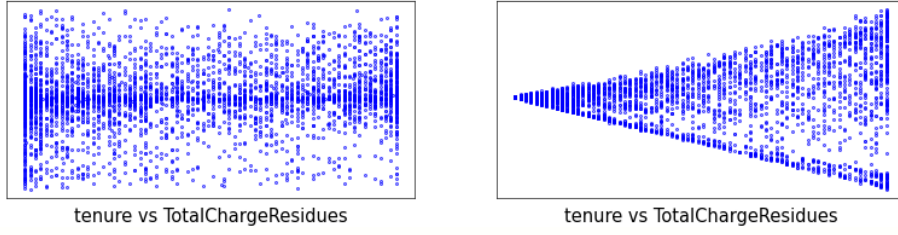


Figure 2.8: Synthetisation with one term in loss function (left), vs real data (right)

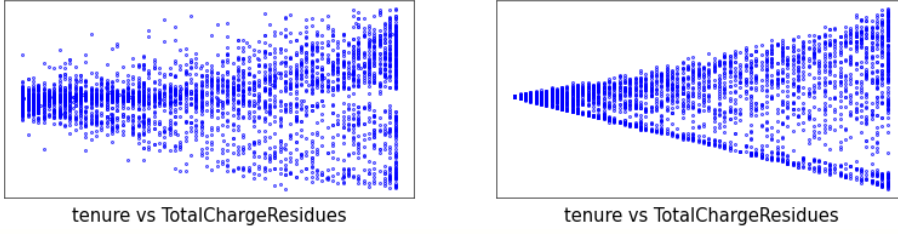


Figure 2.9: Synthetisation with two terms in loss function (left), vs real data (right)

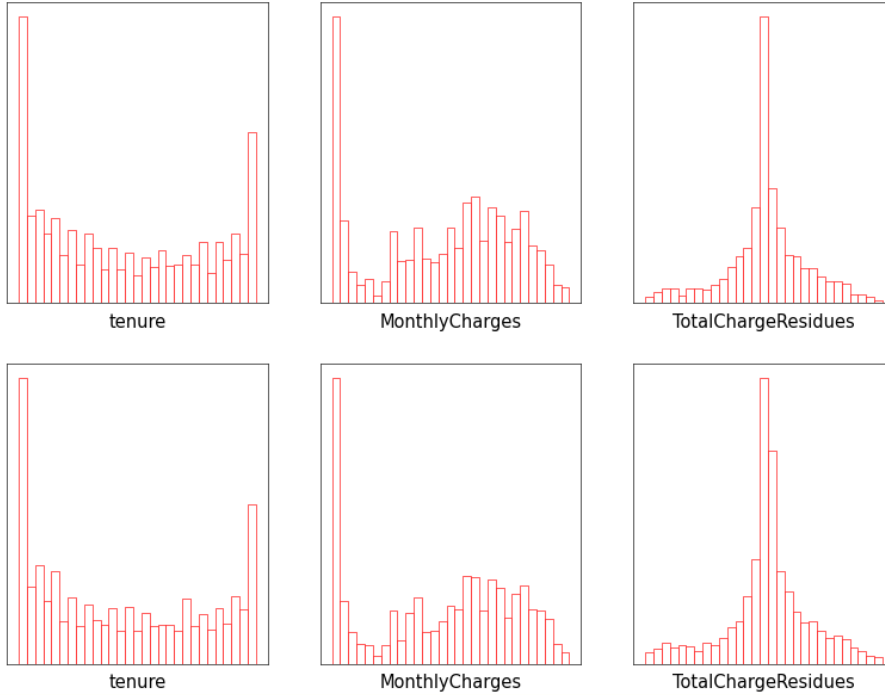


Figure 2.10: Synthetisation (top) vs validation set (bottom), Telecom dataset

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 hyperParam[2] to an unusually large value. Such fine-tuning is easy to automate, and intuitive.

Finally, Figure 2.9 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 2.8. 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 2.10.

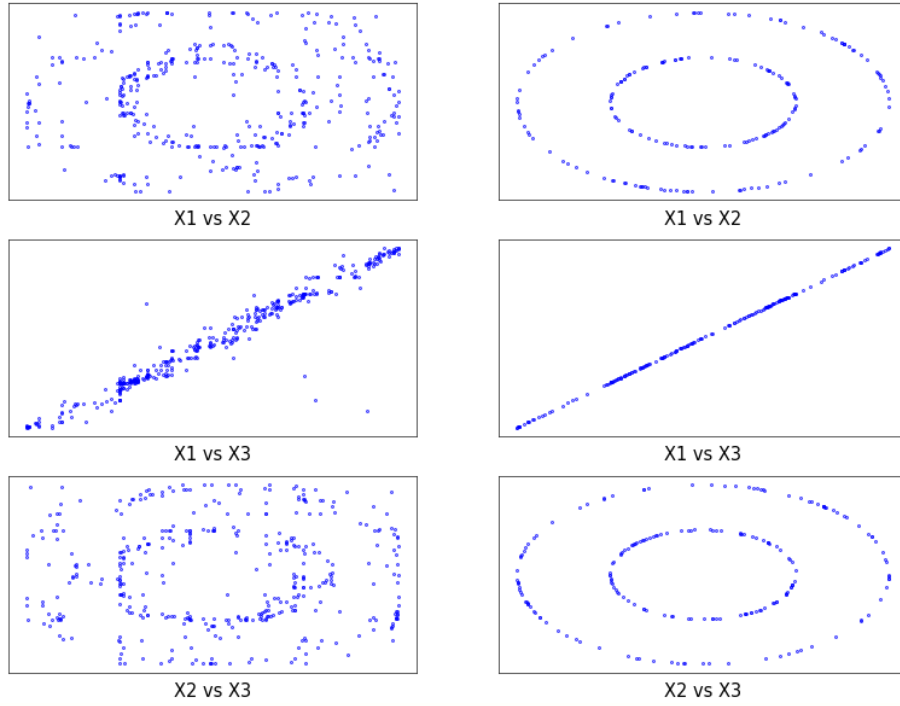


Figure 2.11: Synthetization (left) vs validation set (right), circle dataset

2.2.2.3 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 2.11. 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 cybersecurity 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 [18].

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 2.2.2.4. In the end, for tabular data generation, the only real competitor to NoGAN2 is NoGAN explored in chapter 2.1, 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 [8]. 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 2.11. 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.

2.2.2.4 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 2.2.1.2. For illustration purposes, see Table 2.1 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 $[X_1, \dots, X_m]$ and $[X'_1, \dots, X'_m]$ 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 $W = [\alpha_1, \alpha_2]$, 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 $P = [p_1, \dots, p_m]$ with $p_1 + \dots + p_m = 1$, also described in section 2.2.1.3. 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 2.1, 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 .

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 $F = [q_1, \dots, q_m]$. 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.

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 2.1: Correlations in the loss function, Telecom dataset

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.

2.2.2.5 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 chapter 2.1. It is now available as the GenAI-Evaluation Python library on PyPi, [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** [44, 59].

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 [49], while an algorithm for fast computation of the multivariate ECDF is discussed in [41]. To my knowledge, this is the first practical implementation in high dimensions, tested on real datasets.

As a reminder, given a dataset and location $z = (z_1, \dots, z_m)$ in the feature space, the ECDF $F(z)$ evaluated at z is defined as the proportion of observations $x = (x_1, \dots, x_m)$ satisfying $x_i < z_i$ for $i = 1, \dots, m$. 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

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

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.

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 2.12 is produced in part 10 in the code featured in section 2.2.4, 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.

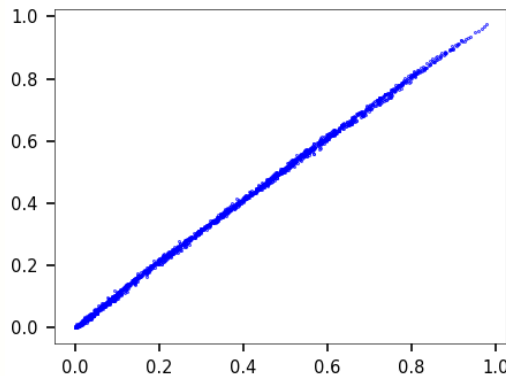


Figure 2.12: ECDF scatterplot, synthetic vs validation set

2.2.3 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](#) [31]. 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, though 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. NoGAN3 discussed in section [2.3](#) solves this problem, using bin counts in the loss function as in NoGAN (section [2.1](#)), 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](#).

2.2.4 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.

```

1 import numpy as np
2 import pandas as pd
3
4 import matplotlib.pyplot as plt
5 import matplotlib as mpl
6 import genai_evaluation as ge
7 from matplotlib import pyplot
8 from statsmodels.distributions.empirical_distribution import ECDF
9 import warnings
10 warnings.simplefilter("ignore")
11
12
13 #--- [1] read data and only keep features and observations we want
14
15 def category_to_integer(category):
16     if category == 'Enrolled':
17         integer = 1
18     elif category == 'Dropout':
19         integer = 0
20     else:
21         integer = 2
22     return(integer)
23
24 #- [1.1] read data
25
26 url = "https://raw.githubusercontent.com/VincentGranville/Main/main/students.csv"
```

```

27 # data = pd.read_csv('students_C2_full_nogan.csv')
28 data = pd.read_csv(url)
29 print(data)
30
31 # all features used here
32
33 features = [
34     'Application_mode',          # 0, categorical
35     'Course',                    # 1, categorical
36     'Previous_qualification_grade', # 2, ordinal
37     'Mother_qualification',       # 3, categorical
38     'Father_qualification',       # 4, categorical
39     'Mother_occupation',         # 5, categorical
40     'Father_occupation',         # 6, categorical
41     'Admission_grade',           # 7, ordinal
42     'Tuition_fees_up_to_date',    # 8, binary
43     'Age_at_enrollment',         # 9, ordinal [outliers]
44     'Curricular_units_1st_sem_evaluations', # 10, ordinal [0 = missing]
45     'Curricular_units_1st_sem_approved', # 11, ordinal [0 = missing]
46     'Curricular_units_1st_sem_grade', # 12, ordinal [0 = missing]
47     'Curricular_units_2nd_sem_enrolled', # 13, ordinal [0 = missing]
48     'Curricular_units_2nd_sem_evaluations', # 14, ordinal [0 = missing]
49     'Curricular_units_2nd_sem_approved', # 15, ordinal [0 = missing]
50     'Curricular_units_2nd_sem_grade', # 16, ordinal [0 = missing]
51     'Unemployment_rate',         # 17, ordinal
52     'Inflation_rate',            # 18, ordinal, can be < 0
53     'GDP',                       # 19, ordinal, can be < 0
54     'Target'                     # 20, categorical [outcome]
55 ]
56
57 data['Target'] = data['Target'].map(category_to_integer)
58
59 # remove rows with missing values
60 data = data[data['Curricular_units_1st_sem_evaluations'] != 0]
61
62 print(data.head())
63 print (data.shape)
64 print (data.columns)
65
66 #- [1.2] set seed for replicability
67
68 pd.core.common.random_state(None)
69 seed = 106 ## 105
70 np.random.seed(seed)
71
72 #- [1.3] select features
73
74 data = data[features]
75 data = data.sample(frac = 1) # shuffle rows to break artificial sorting
76
77 #- [1.4] split real dataset into training and validation sets
78
79 data_training = data.sample(frac = 0.5)
80 data_validation = data.drop(data_training.index)
81 data_training.to_csv('training_vg2.csv')
82 data_validation.to_csv('validation_vg2.csv')
83 data_train = pd.DataFrame.to_numpy(data_training)
84
85 nobs = len(data_training)
86 n_features = len(features)
87
88
89 #--- [2] create initial synthetic data
90
91 def create_initial_synth(nobs_synth):
92
93     eps = 0.000000001
94     n_features = len(features)
95     data_synth = np.empty(shape=(nobs_synth, n_features))
96
97     for i in range(nobs_synth):
98         pc = np.random.uniform(0, 1 + eps, n_features)
99         for k in range(n_features):
100             label = features[k]
101             data_synth[i, k] = np.quantile(data_training[label], pc[k], axis=0)
102     return(data_synth)

```

```

103
104
105 #--- [3] loss functions Part 1
106
107 def compute_univariate_stats():
108
109     # 'dt' for training data, 'ds' for synth. data
110
111     # for first tem in loss function
112     dt_mean = np.mean(data_train, axis=0)
113     dt_stdev = np.std(data_train, axis=0)
114     ds_mean = np.mean(data_synth, axis=0)
115     ds_stdev = np.std(data_synth, axis=0)
116
117     # for g(arr)
118     dt_mean1 = np.mean(g(data_train), axis=0)
119     dt_stdev1 = np.std(g(data_train), axis=0)
120     ds_mean1 = np.mean(g(data_synth), axis=0)
121     ds_stdev1 = np.std(g(data_synth), axis=0)
122
123     # for f(arr)
124     dt_mean2 = np.mean(h(data_train), axis=0)
125     dt_stdev2 = np.std(h(data_train), axis=0)
126     ds_mean2 = np.mean(h(data_synth), axis=0)
127     ds_stdev2 = np.std(h(data_synth), axis=0)
128
129     values = [dt_mean, dt_stdev, ds_mean, ds_stdev,
130               dt_mean1, dt_stdev1, ds_mean1, ds_stdev1,
131               dt_mean2, dt_stdev2, ds_mean2, ds_stdev2]
132     return(values)
133
134 def initialize_cross_products_tables():
135
136     # the core structure for fast computation when swapping 2 values
137     # 'dt' for training data, 'ds' for synth. data
138     # 'prod' is for 1st term in loss, 'prod12' for 2nd term
139
140     dt_prod = np.empty(shape=(n_features,n_features))
141     ds_prod = np.empty(shape=(n_features,n_features))
142     dt_prod12 = np.empty(shape=(n_features,n_features))
143     ds_prod12 = np.empty(shape=(n_features,n_features))
144
145     for k in range(n_features):
146         for l in range(n_features):
147             dt_prod[l, k] = np.dot(data_train[:,l], data_train[:,k])
148             ds_prod[l, k] = np.dot(data_synth[:,l], data_synth[:,k])
149             dt_prod12[l, k] = np.dot(g(data_train[:,l]), h(data_train[:,k]))
150             ds_prod12[l, k] = np.dot(g(data_synth[:,l]), h(data_synth[:,k]))
151     products = [dt_prod, ds_prod, dt_prod12, ds_prod12]
152     return(products)
153
154
155 #--- [4] loss function Part 2: managing loss function
156
157 # Weights hyperparameter:
158 #
159 # 1st value is for 1st term in loss function, 2nd value for 2nd term
160 # each value should be between 0 and 1, all adding to 1
161 # works best when loss contributions from each term are about the same
162
163 #- [4.1] loss function contribution from features (k, l) jointly
164
165 # before calling functions in sections [4.1], [4.2] and [4.3], first intialize
166 # by calling compute_univariate_stats() and compute_cross_products() before;
167 # this initialization needs to be done only once at the beginning
168
169 def get_distance(k, l, weights):
170
171     dt_prodn = dt_prod[k, l] / nobs
172     ds_prodn = ds_prod[k, l] / nobs_synth
173     dt_r = (dt_prodn - dt_mean[k]*dt_mean[l]) / (dt_stdev[k]*dt_stdev[l])
174     ds_r = (ds_prodn - ds_mean[k]*ds_mean[l]) / (ds_stdev[k]*ds_stdev[l])
175
176     dt_prodn12 = dt_prod12[k, l] / nobs
177     ds_prodn12 = ds_prod12[k, l] / nobs_synth
178     dt_r12 = (dt_prodn12 - dt_mean1[k]*dt_mean2[l]) / (dt_stdev1[k]*dt_stdev2[l])

```

```

179     ds_r12 = (ds_prodn12 - ds_mean1[k]*ds_mean2[l]) / (ds_stdev1[k]*ds_stdev2[l])
180
181     # dist = weights[0]*abs(dt_r - ds_r) + weights[1]*abs(dt_r12 - ds_r12)
182     dist = max(weights[0]*abs(dt_r - ds_r), weights[1]*abs(dt_r12 - ds_r12))
183     return(dist, dt_r, ds_r, dt_r12, ds_r12)
184
185 def total_distance(weights, flagParam):
186
187     eval = 0
188     max_dist = 0
189     super_max = 0
190     lmax = n_features
191
192     for k in range(n_features):
193         if symmetric:
194             lmax = k
195         for l in range(lmax):
196             if l != k and flagParam[k] > 0 and flagParam[l] > 0:
197                 values = get_distance(k, l, weights)
198                 dist2 = max(abs(values[1] - values[2]), abs(values[3] - values[4]))
199                 eval += values[0]
200                 if values[0] > max_dist:
201                     max_dist = values[0]
202                 if dist2 > super_max:
203                     super_max = dist2
204     return(eval, max_dist, super_max)
205
206 #- [4.2] updated loss function when swapping rows idx1 and idx2 in feature k
207 #     contribution from feature l jointly with k
208
209 def get_new_distance(k, l, idx1, idx2, weights):
210
211     tmp1_k = data_synth[idx1, k]
212     tmp2_k = data_synth[idx2, k]
213     tmp1_l = data_synth[idx1, l]
214     tmp2_l = data_synth[idx2, l]
215
216     #-- first term of loss function
217
218     remove1 = tmp1_k * tmp1_l
219     remove2 = tmp2_k * tmp2_l
220     add1 = tmp1_k * tmp2_l
221     add2 = tmp2_k * tmp1_l
222     new_ds_prod = ds_prod[l, k] - remove1 - remove2 + add1 + add2
223
224     dt_prodn = dt_prod[k, l] / nobs
225     ds_prodn = new_ds_prod / nobs_synth
226     dt_r = (dt_prodn - dt_mean[k]*dt_mean[l]) / (dt_stdev[k]*dt_stdev[l])
227     ds_r = (ds_prodn - ds_mean[k]*ds_mean[l]) / (ds_stdev[k]*ds_stdev[l])
228
229     #-- second term of loss function
230
231     remove1 = g(tmp1_k) * h(tmp1_l)
232     remove2 = g(tmp2_k) * h(tmp2_l)
233     add1 = g(tmp1_k) * h(tmp2_l)
234     add2 = g(tmp2_k) * h(tmp1_l)
235     new_ds_prodn12 = ds_prodn12[k, l] - remove1 - remove2 + add1 + add2
236
237     dt_prodn12 = dt_prodn12[k, l] / nobs
238     ds_prodn12 = new_ds_prodn12 / nobs_synth
239     dt_r12 = (dt_prodn12 - dt_mean1[k]*dt_mean2[l]) / (dt_stdev1[k]*dt_stdev2[l])
240     ds_r12 = (ds_prodn12 - ds_mean1[k]*ds_mean2[l]) / (ds_stdev1[k]*ds_stdev2[l])
241
242     #--
243
244     # new_dist = weights[0]*abs(dt_r - ds_r) + weights[1]*abs(dt_r12 - ds_r12)
245     new_dist = max(weights[0]*abs(dt_r - ds_r), weights[1]*abs(dt_r12 - ds_r12))
246     return(new_dist, dt_r, ds_r, dt_r12, ds_r12)
247
248
249 #- [4.3] update prod tables after swapping rows idx1 and idx2 in feature k
250 #     update impacting feature l jointly with k
251
252 def update_product(k, l, idx1, idx2):
253
254     tmp1_k = data_synth[idx1, k]

```

```

255     tmp2_k = data_synth[idx2, k]
256     tmp1_l = data_synth[idx1, l]
257     tmp2_l = data_synth[idx2, l]
258
259     #-- first term of loss function
260
261     remove1 = tmp1_k * tmp1_l
262     remove2 = tmp2_k * tmp2_l
263     add1 = tmp1_k * tmp2_l
264     add2 = tmp2_k * tmp1_l
265     ds_prod[k, l] = ds_prod[k, l] - remove1 - remove2 + add1 + add2
266     ds_prod[l, k] = ds_prod[k, l]
267
268     #-- second term of loss function
269
270     remove1 = g(tmp1_k) * h(tmp1_l)
271     remove2 = g(tmp2_k) * h(tmp2_l)
272     add1 = g(tmp1_k) * h(tmp2_l)
273     add2 = g(tmp2_k) * h(tmp1_l)
274     ds_prod12[k, l] = ds_prod12[k, l] - remove1 - remove2 + add1 + add2
275
276     remove1 = h(tmp1_k) * g(tmp1_l)
277     remove2 = h(tmp2_k) * g(tmp2_l)
278     add1 = h(tmp1_k) * g(tmp2_l)
279     add2 = h(tmp2_k) * g(tmp1_l)
280     ds_prod12[l, k] = ds_prod12[l, k] - remove1 - remove2 + add1 + add2
281
282     return()
283
284
285 #--- [5] feature sampling
286
287 def sample_feature(mode, hyperParameter):
288
289     # Randomly pick up one column (a feature) to swap 2 values from 2 random rows
290     # One feature is assumed to be in the right order, thus ignored
291
292     if mode == 'Equalized':
293         u = np.random.uniform(0, 1)
294         cutoff = hyperParam[0]
295         feature = 0
296         while cutoff < u:
297             feature += 1
298             cutoff += hyperParam[feature]
299     else:
300         feature = np.random.randint(1, n_features) # ignore feature 0
301     return(feature)
302
303
304 #--- [6] functions: deep synthetization, plot history, print stats
305
306 #- [6.1] main function
307
308 def deep_resampling(hyperParameter, run, loss_type, n_batches,
309                    n_iter, nob_synth, weights, flagParam, mode):
310
311     # main function
312
313     batch = 0
314     batch_size = nob_synth // n_batches
315     niter_per_batch = n_iter // n_batches
316     lower_row = 0
317     upper_row = batch_size
318     nswaps = 0
319     cgain = 0 # cumulative gain
320
321     arr_swaps = []
322     arr_history_quality = []
323     arr_history_max_dist = []
324     arr_time = []
325     print()
326
327     for iter in range(n_iter):
328
329         k = sample_feature(mode, hyperParameter)
330         batch = iter // niter_per_batch

```

```

331     lower_row = batch * batch_size
332     upper_row = lower_row + batch_size
333     idx1 = np.random.randint(lower_row, upper_row) % nobs_synth
334     tmp1 = data_synth[idx1, k]
335     tmp2 = tmp1
336     counter = 0
337     while tmp2 == tmp1 and counter < 20:
338         idx2 = np.random.randint(lower_row, upper_row) % nobs_synth
339         tmp2 = data_synth[idx2, k]
340         counter += 1
341
342     g_param = 0.5
343     h_param = g_param
344
345     delta = 0
346     delta2 = 0
347     for l in range(n_features):
348         if l != k and flagParam[l] > 0:
349             values = get_distance(k, l, weights)
350             delta += values[0]
351             if values[0] > delta2:
352                 delta2 = values[0]
353             if not symmetric: # if functions g, h are different
354                 values = get_distance(l, k, weights)
355                 delta += values[0]
356                 if values[0] > delta2:
357                     delta2 = values[0]
358
359     new_delta = 0
360     new_delta2 = 0
361     for l in range(n_features):
362         if l != k and flagParam[l] > 0:
363             values = get_new_distance(k, l, idx1, idx2, weights)
364             new_delta += values[0]
365             if values[0] > new_delta2:
366                 new_delta2 = values[0]
367             if not symmetric: # if functions g, h are different
368                 values = get_new_distance(l, k, idx1, idx2, weights)
369                 new_delta += values[0]
370                 if values[0] > new_delta2:
371                     new_delta2 = values[0]
372
373     if loss_type == 'sum_loss':
374         gain = delta - new_delta
375     elif loss_type == 'max_loss':
376         gain = delta2 - new_delta2
377     if gain > 0:
378         cgain += gain
379     for l in range(n_features):
380         if l != k:
381             update_product(k, l, idx1, idx2)
382             # update_product(l, k, idx1, idx2)
383     data_synth[idx1, k] = tmp2
384     data_synth[idx2, k] = tmp1
385     nswaps += 1
386
387     if iter % 500 == 0:
388         quality, max_dist, super_max = total_distance(weights, flagParam)
389         arr_swaps.append(nswaps)
390         arr_history_quality.append(quality)
391         arr_history_max_dist.append(max_dist)
392         arr_time.append(iter)
393         if iter % 5000 == 0:
394             print("Iter: %6d Distance: %8.4f SupDist: %8.4f Gain: %8.4f Swaps: %6d"
395                   %(iter, quality, super_max, cgain, nswaps))
396
397     return(nswaps, arr_swaps, arr_history_quality, arr_history_max_dist, arr_time)
398
399 #- [6.2] save synthetic data, show some stats
400
401 def evaluate_and_save(filename, weights, run, flagParam):
402
403     print("\nMetrics after deep resampling\n")
404     quality, max_dist, super_max = total_distance(weights, flagParam)
405     print("Distance: %8.4f" %(quality))
406     print("Max Dist: %8.4f" %(max_dist))

```

```

407
408 data_synthetic = pd.DataFrame(data_synth, columns = features)
409 data_synthetic.to_csv(filename)
410 print("\nSynthetic data, first 10 rows\n",data_synthetic.head(10))
411
412 print("\nBivariate feature correlation:")
413 print("...dt_xx for training set, ds_xx for synthetic data")
414 print("...xx_r for correl[k, l], xx_r12 for correl[g(k), h(l)]\n")
415 print("%2s %2s %8s %8s %8s %8s"
416       % ('k', 'l', 'dist', 'dt_r', 'ds_r', 'dt_r12', 'ds_r12'))
417 print("-----")
418 for k in range(n_features):
419     for l in range(n_features):
420         condition = (flagParam[k] >0 and flagParam[l] > 0)
421         if k != l and condition:
422             values = get_distance(l, k, weights)
423             dist = values[0]
424             dt_r = values[1] # training, 1st term of loss function
425             ds_r = values[2] # synth., 1st term of loss function
426             dt_r12 = values[3] # training, 2nd term of loss function
427             ds_r12 = values[4] # synth., 2nd term of loss function
428             print("%2d %2d %8.4f %8.4f %8.4f %8.4f"
429                   % (k, l, dist, dt_r, ds_r, dt_r12, ds_r12))
430     return()
431
432 #- [6.3] plot history of loss function, and cumulated number of swaps
433
434 def plot_history(history):
435
436     arr_swaps = history[1]
437     arr_history_quality = history[2]
438     arr_history_max_dist = history[3]
439     arr_time = history[4]
440
441     mpl.rcParams['axes.linewidth'] = 0.3
442     plt.rc('xtick', labelsiz=7)
443     plt.rc('ytick', labelsiz=7)
444     plt.xticks(fontsize=7)
445     plt.yticks(fontsize=7)
446     plt.subplot(1, 2, 1)
447     plt.plot(arr_time, arr_swaps, linewidth = 0.3)
448     plt.legend(['cumulated swaps'], fontsize="7",
449              loc="upper center", ncol=1)
450     plt.subplot(1, 2, 2)
451     plt.plot(arr_time, arr_history_quality, linewidth = 0.3)
452     # plt.plot(arr_time, arr_history_max_dist, linewidth = 0.3)
453     plt.legend(['distance'], fontsize="7",
454              loc="upper center", ncol=1)
455     plt.show()
456     return()
457
458
459 #--- [7] initializations
460
461 #- create initial synthetization
462
463 nob_synth = 2000
464 data_synth = create_initial_synth(nob_synth)
465
466 #- specify 2nd part of loss function (argument is a number or array)
467
468 # do not use g(arr) = f(arr) = arr: this is pre-built already as 1st term in loss fct
469 # these two functions f, g are for the second term in the loss function
470
471 def g(arr):
472     return(1/(0.01 + np.absolute(arr)))
473 def h(arr):
474     return(1/(0.01 + np.absolute(arr)))
475
476 symmetric = True # set to True if functions g and h are identical
477 # 'symmetric = True' twice as fast as 'symmetric = False'
478
479 #- initializations: product tables and univariate stats
480
481 products = initialize_cross_products_tables()
482 dt_prod = products[0]

```

```

483 ds_prod = products[1]
484 dt_prod12 = products[2]
485 ds_prod12 = products[3]
486
487 values = compute_univariate_stats()
488 dt_mean = values[0]
489 dt_stdev = values[1]
490 ds_mean = values[2]
491 ds_stdev = values[3]
492 dt_mean1 = values[4]
493 dt_stdev1 = values[5]
494 ds_mean1 = values[6]
495 ds_stdev1 = values[7]
496 dt_mean2 = values[8]
497 dt_stdev2 = values[9]
498 ds_mean2 = values[10]
499 ds_stdev2 = values[11]
500
501
502 #--- [8] deep resampling
503
504 mode = 'Equalized' # options: 'Standard', 'Equalized'
505 eps2 = 0.0
506
507 #- first run
508
509 run = 1
510 n_iter = 500001
511 n_batches = 8
512 loss_type = 'sum_loss' # options: 'max_loss' or 'sum_loss'
513 weights = [0.90, 0.10]
514 hyperParam = np.zeros(len(features))
515 hyperParam[8] = 1
516 hyperParam[10] = 1
517 hyperParam[11] = 1
518 hyperParam[12] = 1
519 hyperParam[13] = 1
520 hyperParam[14] = 1
521 hyperParam[15] = 1
522 hyperParam[16] = 1
523 hyperParam = hyperParam / np.sum(hyperParam)
524 flagParam = np.ones(len(features))
525 history = deep_resampling(hyperParam, run, loss_type, n_batches, n_iter,
526                          nobs_synth, weights, flagParam, mode)
527 evaluate_and_save('synth_vg2.csv', weights, run, flagParam)
528 plot_history(history)
529
530 #- second run
531
532 run = 2
533 n_iter = 500001
534 n_batches = 8
535 loss_type = 'sum_loss' # options: 'max_loss' or 'sum_loss'
536 weights = [0.90, 0.10]
537 hyperParam = np.ones(len(features))
538 hyperParam[8] = 0
539 hyperParam[10] = 0
540 hyperParam[11] = 0
541 hyperParam[12] = 0
542 hyperParam[13] = 0
543 hyperParam[14] = 0
544 hyperParam[15] = 0
545 hyperParam[16] = 0
546 hyperParam = hyperParam / np.sum(hyperParam)
547 flagParam = np.ones(len(features))
548 history = deep_resampling(hyperParam, run, loss_type, n_batches, n_iter,
549                          nobs_synth, weights, flagParam, mode)
550 evaluate_and_save('synth_vg2.csv', weights, run, flagParam)
551 plot_history(history)
552
553
554 #--- [9] Evaluation synthetization using joint ECDF & Kolmogorov-Smirnov distance
555
556 #     dataframes: df = synthetic; data = real data,
557 #     compute multivariate ecdf on validation set, sort it by value (from 0 to 1)
558

```

```

559 print("\nMultivariate ECDF computations...\n")
560 n_nodes = 1000 # number of random locations in feature space, where ecdf is computed
561 seed = 50
562 np.random.seed(seed)
563
564 df_validation = pd.DataFrame(data_validation, columns = features)
565 df_synthetic = pd.DataFrame(data_synth, columns = features)
566 df_training = pd.DataFrame(data_train, columns = features)
567 query_lst, ecdf_val, ecdf_synth = ge.multivariate_ecdf(df_validation, df_synthetic, n_nodes,
    verbose = True)
568 query_lst, ecdf_val, ecdf_train = ge.multivariate_ecdf(df_validation, df_training, n_nodes, verbose
    = True)
569
570 ks = ge.ks_statistic(ecdf_val, ecdf_synth)
571 ks_base = ge.ks_statistic(ecdf_val, ecdf_train)
572 print("Test ECDF Kolmogorof-Smirnov dist. (synth. vs valid.): %6.4f" %(ks))
573 print("Base ECDF Kolmogorof-Smirnov dist. (train. vs valid.): %6.4f" %(ks_base))
574
575
576 #--- [10] visualizations (based on Matplotlib version: 3.7.1)
577
578 def vg_scatter(df, feature1, feature2, counter):
579
580     # customized plots, subplot position based on counter
581
582     label = feature1 + " vs " + feature2
583     x = df[feature1].to_numpy()
584     y = df[feature2].to_numpy()
585     plt.subplot(3, 2, counter)
586     plt.scatter(x, y, s = 0.1, c ="blue")
587     plt.xlabel(label, fontsize = 7)
588     plt.xticks([])
589     plt.yticks([])
590     #plt.ylim(0,70000)
591     #plt.xlim(18,64)
592     return()
593
594 def vg_histo(df, feature, counter):
595
596     # customized plots, subplot position based on counter
597
598     y = df[feature].to_numpy()
599     plt.subplot(2, 3, counter)
600     min = np.min(y)
601     max = np.max(y)
602     binBoundaries = np.linspace(min, max, 30)
603     plt.hist(y, bins=binBoundaries, color='white', align='mid',edgecolor='red',
604         linewidth = 0.3)
605     plt.xlabel(feature, fontsize = 7)
606     plt.xticks([])
607     plt.yticks([])
608     return()
609
610 mpl.rcParams['axes.linewidth'] = 0.3
611
612 #- [10.1] scatterplots
613
614 dfs = pd.read_csv('synth_vg2.csv')
615 dfv = pd.read_csv('validation_vg2.csv')
616 vg_scatter(dfs, features[0], features[1], 1)
617 vg_scatter(dfv, features[0], features[1], 2)
618 vg_scatter(dfs, features[0], features[2], 3)
619 vg_scatter(dfv, features[0], features[2], 4)
620 vg_scatter(dfs, features[1], features[2], 5)
621 vg_scatter(dfv, features[1], features[2], 6)
622 plt.show()
623
624 #- [10.2] histograms
625
626 dfs = pd.read_csv('synth_vg2.csv')
627 dfv = pd.read_csv('validation_vg2.csv')
628 vg_histo(dfs, features[0], 1)
629 vg_histo(dfs, features[1], 2)
630 vg_histo(dfs, features[2], 3)
631 vg_histo(dfv, features[0], 4)
632 vg_histo(dfv, features[1], 5)

```

```
633 vg_histo(dfv, features[2], 6)
634 plt.show()
```

2.3 Boosting Model Evaluation with Smart Adaptive Loss Functions

Most AI and GenAI algorithms aim at optimizing a criterion. The goal is to get good predictions, relevant and exhaustive answers (LLMs) or realistic synthetizations. The quality of the output is assessed using some **evaluation metric**. However, the algorithm tries to minimize a **loss function**. Both are very distinct: one cannot expect to get the best output by optimizing a criterion other than the model evaluation metric.

So, why are we dealing with two different criteria: loss and evaluation? The reason is simple: good global evaluation metrics are unable to handle atomic updates extremely fast, billions of times, for instance each time a neuron is activated in a **deep neural network**. Instead, loss functions are very good at that, and serve as a proxy to the evaluation metric. But what if we could find a great evaluation metric that can be updated at the speed of light, at each tiny change? The goal of this project is to answer this question, with a case study in the context of tabular **synthetic data** generation.

This problem is typically solved with generative adversarial networks (**GANs**), see project 5.2 in [25]. The best evaluation metric, to assess the faithfulness of the generated data, is arguably the **multivariate Kolmogorov-Smirnov distance** (KS). It compares two multivariate empirical distributions (**ECDF**), corresponding to the training and synthetic datasets. But GAN does not optimize KS. Instead it tries to minimize a loss function such as **Wasserstein**, via **stochastic gradient descent**.

It would have been fantastic to replace the loss function by KS, but I could not find any way to make it computationally feasible: any tiny change to the data – which happens billions of times with GAN or my other algorithms – requires to recompute the full KS, a time-consuming task. But I found an even more ground-breaking solution. It involves comparing the densities (**EPDF**) rather than the distributions (**ECDF**). Easier said than done, as it brings new challenges with continuous features; the solution is a lot easier with categorical features.

The seminal idea consists of using more and more granular approximations to the EPDF as the iterative algorithm progresses, eventually converging to the exact yet singular EPDF. The resulting evaluation metric is a variant of the **Hellinger distance**, rather than KS. Thus, it amounts to using an **adaptive loss function** [46] that converges to the Hellinger distance. Then, I tested the method using a special framework where the global optimum is known in advance, hoping to retrieve it. The result is spectacular:

- Brute-force combinatorial approach requires 10^{869} steps (at most) to find the global optimum.
- I found it in less than 2 millions operations.
- A typical GAN is nowhere close to the optimum after 4 millions weight updates, and it requires more computing time than my method.
- The classic **Hungarian algorithm** solves this assignment problem [Wiki] in 64 millions operations at most.

The global optimum is unique up to a permutation of the generated observations. I found it with no GAN, no neural network. Instead, with a probabilistic algorithm. These algorithms tend to progress very fast initially, but quite slowly after a while. One would think that if using neural networks, replacing the fixed Wasserstein loss function with adaptive Hellinger approximations (that is, model evaluations that get more accurate over time), you would need much less than 2 millions steps. However, it remains to be seen if a gradient descent – the core of any neural network – is suitable in this case.

My solution does not use gradient descent: instead, at each step, a random move is made (thus, not based on a continuous path governed by derivatives of a function), but nevertheless always in the right direction: the move is accepted only if further minimizing the loss. In addition, there are four more important components to my method:

- The algorithm starts with an initial synthetic dataset where all marginal distributions match those in the training set. So, the univariate Hellinger distances are perfect to begin with. The goal is then to optimize the multivariate Hellinger distance, which takes into account the full dependency structure in the feature space.
- The algorithm is a version of the hierarchical Bayesian **NoGAN** described in chapter 7 in [26]. This is a resampling method that preserves the marginal distributions throughout all iterations, while attempting to approach the joint distribution in the training set, using a continuous loss function as in neural networks. However, here the loss function is replaced by the evaluation metric (Hellinger).

- The granularity of the EDPF lattice support increases over time, resulting in an exponential explosion in the number of bins, especially in high dimensions. However the sparsity increases in the same proportion: the vast majority of bins are empty and not even stored or visited. The number of active bins (stored in hash tables) is not larger than the number of observations, at any given time.
- Pre-computation of square roots, stored in tables. Each of the 2 million atomic updates requires the computation of 16 square roots. The tables in question significantly contribute to speeding up the algorithm.

Most tabular data synthesizers have challenges to generate observations with the prescribed distribution. By replacing the loss function with the evaluation metric, I face the opposite problem: generating observations too similar to those in the training set. So instead of trying to achieve a low value for the Hellinger distance as in GAN, I must do the opposite: preventing the Hellinger distance from being too low. This is accomplished by setting up constraints. I call my approach **constrained synthetization**. See Figure 2.13 where the Hellinger distance H between the synthetic data and training set (real data) is kept above 0.60 at all times.

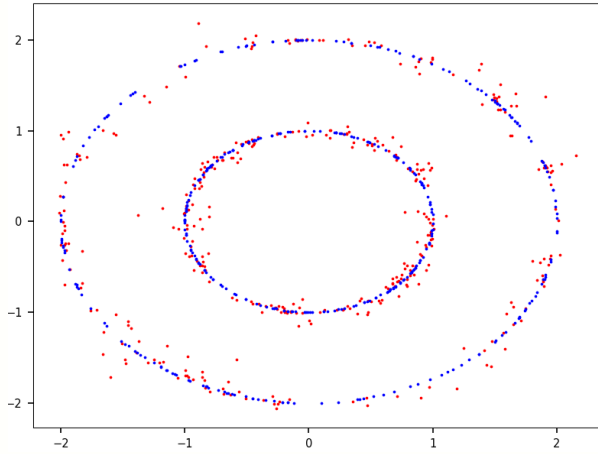


Figure 2.13: Synth. ($H > 0.6$, red) vs real (blue) with `NoGAN_Hellinger.py`

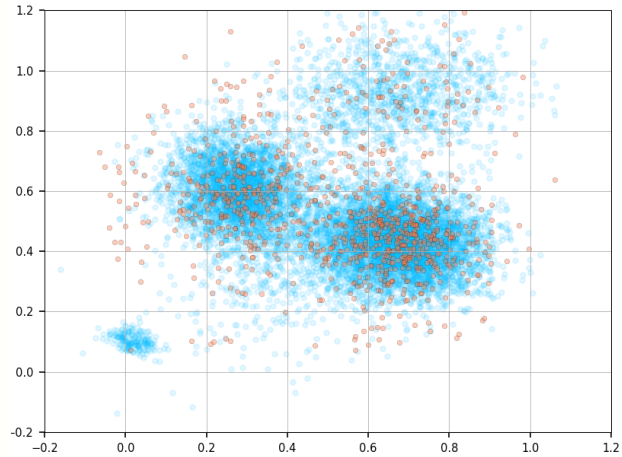


Figure 2.14: First 2 dimensions of mixture data (blue) with synthetization in orange

2.3.1 Project and solution

The name for the new algorithm is **constrained NoGAN**. It blends the best of **probabilistic NoGAN** (based on deep **resampling**, see section 2.2) and Python code `DeepResampling_circle.py` [here](#)) with the efficient multivariate **bin structure** found in **standard NoGAN** (see section 2.1 and corresponding code `NoGAN.py` [here](#)). The bin structure is used to compute the Hellinger distance. For the simplest version of constrained NoGAN, see `NoGAN_Hellinger.py`, [here](#). This script is great to synthetize data featuring hidden structures (in this case, concentric circles), to check whether or not pattern detection algorithms can detect them. The higher the Hellinger distance threshold used in the synthetization, the less visible the circles, making detection harder.

In the remaining of this project, I focus on `NoGAN_hellinger2.py`, with a slightly different architecture leading to the spectacular results discussed earlier. The code is in section 2.3.2 and also on GitHub, [here](#). The resampling part works as follows:

- Randomly select two observations $x = (x_1, x_2, x_3)$ and $y = (y_1, y_2, y_3)$ from the synthetic dataset under construction.
- Propose the update $x' = (x_1, y_2, x_3)$ and $y' = (y_1, x_2, y_3)$ and compute the change in Hellinger distance, would the proposed update be accepted. Here the Hellinger distance is measured on the first two features both in the training and synthetic data, while ignoring the third feature.
- Accept the change if it decreases the Hellinger distance, and repeat these steps millions of times, until the Hellinger distance barely decreases anymore.

The above describes the first pass: that is, updating the second feature conditionally to the first one. In a second pass, you update the third feature conditionally to the first two ones. This time the Hellinger distance is computed on all 3 features. You have more passes if you have more than 3 features. In practice, each feature is actually a block of features. For simplicity, in this project I stop after the first pass.

You need to start with an initial synthetic dataset where each feature taken separately has about the same empirical distribution as its sister in the training set. This is accomplished with lines 93–97 in the code, using the option `mode='Quantiles'`. No matter the number of iterations, the final synthetic data set will be close

but different to the training set: it is constrained on the initial synthetization. However, if you use the option `mode='Shuffle'`, the program uses a scrambled version of the training set for the initial synthetization: each feature column is shuffled separately. In this case, the only synthetization minimizing the Hellinger distance (up to a permutation of the rows) is the training set itself! You would think that it will take billions of years of computing time to reach it *exactly* given the probabilistic nature of the algorithm, albeit getting a very good approximation very quickly. However, thanks to the small size of the training set (400 observations), it takes just 2 millions atomic updates (also called swaps) to reach the unique, known global optimum. This is faster than any other algorithm including neural networks.

Mode	Dim	Iterations	KS(S, T)	KS(S_0, T)	KS(S_0, S)	Swaps
Shuffle	2	2,000,000	0.0000	0.0525	0.0500	12,429
Quantiles	2	2,000,000	0.0350	0.0725	0.0500	17,764
Shuffle	3	2,000,000	0.0200	0.2275	0.2150	12,177
Quantiles	3	2,000,000	0.0475	0.2075	0.2025	12,801

Table 2.2: KS stats: S = synth., S_0 = initial synth., and T = training set

Table 2.2 shows the quality of the synthetization S , depending on the dimension (the number of features), the number of iterations, and whether the initial synthetization S_0 is independent from the training set T (mode set to ‘Quantiles’) or a scrambled version of the training set (mode set to ‘Shuffle’). The parameter `dim` is specified in line 18 in the code. The values for `granularity` and `reset_granularity` depend on `dim` and are set in the code respectively in line 151 and 149. The multivariate **Kolmogorov-Smirnov distance** (KS) takes values between 0 and 1, with zero for perfect match. It is computed in lines 283–305 in the code, using the **GenAI-evaluation** Python library. The quality of the synthetization is assessed by comparing the base value $\text{KS}(S_0, T)$ with $\text{KS}(S, T)$. The lower the latter, the better the result.

The training set used in the illustrations so far is the circle dataset available [here](#). It has 9 features: the last one is categorical (binary) and the other ones are concentrated in a tiny portion of the feature space. For instance, in the first two dimensions, the points are concentrated on two concentric circles. Combined with the fact that it contains only 400 observations, it is a challenging set for synthetization purposes. All the cross-correlations are either 0, 1 or -1. It tricks many synthesizers based on simple loss functions to generate points in a full 2-dimensional square rather than on the one-dimensional circles. I compared the solutions offered by several vendors in an notorious article, available [here](#).

The project consists of the following steps:

Step 1: Hyperparameters and notations. There are four top parameters: `granularity`, `shift`, `reset_granularity`, and `reset_shift`, denoted respectively as $\gamma, \tau, \Delta_\gamma$ and Δ_τ . The grid determining the granularity of the multivariate bins is initialized as follows: each feature is binned into γ **quantile** intervals of equal length. Every Δ_γ iterations, γ is increased by one. Then, in the main loop starting at line 157, the Hellinger distance (its square) between the synthetic data under construction and the training set, is discretized using the grid in question. Finally, the vector τ determines the location of the grid, that is, how much it is shifted from the origin, for each feature separately. It is initially set to zero (the origin) and then changed to a random location every Δ_τ iterations.

Depending on the number `dim` of features (the dimension), if you increase or decrease γ, Δ_γ or Δ_τ , what is the expected impact on the final results? Can you retrieve the values used to produce Table 2.2, by running the code? Also, does it make sense to plot the values of the Hellinger loss function over time? Why not? Would this plot still convey some useful information?

Step 2: Higher dimensions, bigger data. Create a training set consisting of 20 features and 10,000 observations, for instance a **Gaussian mixture**. Feel free to use **OpenAI** to generate the code (I did). Or use my code `mixture.py` or its output data `mixture.csv`, available on GitHub respectively [here](#) and [here](#). When running the synthesizer `NoGAN.Hellinger2.py` (code in section 2.3.2) on the mixture dataset, what challenges are you facing when the dimension is larger than 10, and how to address them? Note that when I asked OpenAI to produce `mixture.py`, the **covariance matrices** in the mixture are not symmetric and **positive semidefinite**. I had to use a second prompt to fix it.

Step 3: Fine-tuning and evaluation. It is easy to assess the impact of the hyperparameters, on the generated (synthetic) data. Design a scheme to efficiently test different combinations of values for $\gamma, \tau, \Delta_\gamma$ and Δ_τ . Then, find combinations that do well, based on the number of features (the dimension). What about the number of observations to synthesize? For example, when synthesizing a large number, it may

be better to split the output data into multiple batches generated separately. Assess the impact of the batch size on the results, especially the speed of convergence. How do you evaluate the results?

Step 4: Comparing NoGAN methods. That is, `NoGAN_hellinger2.py` (the code in section 2.3.2) with `NoGAN_hellinger.py` (also using Hellinger as the loss function), `DeepResampling_circle.py` based on a traditional loss function (discussed in chapter 7 in [26]), and the original `NoGAN.py` described in section 2.1. The latter has no loss function. Finally, also compare with GAN-based `GAN_circle8d.py` described in section 5.2 in [25]. Focus on quality, speed, and overfitting in particular.

Now my answers. Regarding **Step 1**, I used $\gamma = 20$, $\Delta_\gamma = 10,000$ in two dimensions, and then $\gamma = 2$, $\Delta_\gamma = 50,000$ in three dimensions, together with `mode='Shuffle'` in line 89 to produce the results in Table 2.2. I did not use the shift parameters τ , Δ_τ on the circle dataset. Starting with a high value for γ speeds up convergence in low dimensions (2 or 3) but can get you stuck in higher dimensions. Use a lower value of γ_τ if you get stuck. Use $\Delta_\tau \leq 0.5 \times \gamma_\tau$ to visit the feature space more thoroughly: it can speed up convergence and get you unstuck in higher dimensions. A new τ is similar to trying another location or starting point in **gradient descent** to avoid local minima. In higher dimensions, use $\gamma = 2$, $\Delta_\gamma > 100,000$ and $\Delta_\tau < 50,000$. In general, the more swaps, the better. Since the loss function is different each time the value of γ or τ changes, its graph is meaningless. Still, it can help you identify when the moves (called *swaps*) are becoming rare or absent, to fix it: see Figure 2.15, where the slope dramatically changes each time γ is modified, that is, each time the loss stops decreasing fast enough.

As for **Step 2**, in dimensions above 10, it is customary to work with groups of features, one group at a time. Do the first 10, then the next 10 conditionally on the first 10, then the next 10 conditionally on first 20, and so on. The computing time increases linearly with the dimension, keeping the same number of iterations, say 2 million, regardless of dimension. This is similar to neural networks, where computing time also increases linearly with the dimension, not exponentially. See details in chapter 7 in [26]. Finally, I used `mode='Quantiles'` on the mixture dataset that I created, available [here](#), working with the first 8 features only. In Figure 2.14, the training set has 10,000 observations. I generated 1000 synthetic ones, thus the reason why orange dots are much rarer than blue ones.

γ	Δ_γ	Δ_τ	$KS(S, T)$	$KS(S_0, T)$	Swaps
2	200,000	∞	0.0537	0.1949	8758
4	∞	50,000	0.0425	0.1352	32,786
2	200,000	50,000	0.0536	0.1352	23,127

Table 2.3: Hyperparam impact on KS, mixture dataset

Now **Step 3**. Why evaluate the results when the loss function is actually the evaluation metric? Yet, it still makes sense to have another evaluation with a different metric, if anything to confirm that all computations are done correctly. In the code, I use the KS distance for final evaluation, see lines 283–305. Table 2.3 is based on first 8 columns of the mixture dataset, using `mode='Quantiles'`. Here S, S_0, T stand respectively for the final synthetization, initial synthetization, and the training set. The table features some of the best hyperparameter combinations that I tried. For a systematic approach to find these parameters, you can use **smart grid search**, described in [23]. As in table 2.2, I used 2 million iterations.

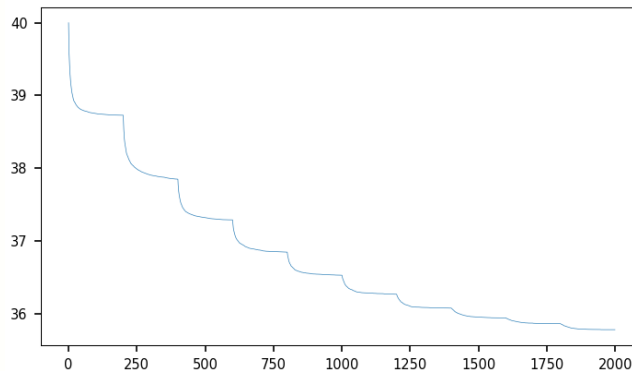


Figure 2.15: Adaptive loss function: steps showing when γ changed

Regarding **Step 4**, `NoGAN.py` is the best, followed by `NoGAN_Hellinger.py` (a probabilistic version of the former). This is because they generate observations directly into the bin structure attached to the training

set. The drawback is potential overfitting, which can be mitigated with **constrained synthetization** as discussed earlier. Then NoGAN_Hellinger2.py usually beats DeepResampling_circle.py because the former uses the evaluation metric as the loss function, while the latter uses a proxy. Finally, GAN_circle8d.py is the last in my ranking: hard to train and fine-tune, very sensitive to the seed and to the dataset (volatile results), non-replicable, and non-explainable AI.

Perhaps one of the greatest benefits of NoGAN_Hellinger2.py is its ability to test very fast new features that could be added to deep neural networks, such as using the evaluation metric as the loss function. Also, by choosing mode='Shuffle', the ability to check whether the system can retrieve the known global optimum, and if not, how close it gets. Finally, an adaptive loss function may prevent you from getting stuck in a local optimum, by changing γ or Δ_τ when the loss stops decreasing: this is visible in Figure 2.15.

2.3.2 Python code

The code below (NoGAN_Hellinger2.py) is also on GitHub, [here](#).

```

1 import numpy as np
2 import pandas as pd
3 import matplotlib.pyplot as plt
4 import matplotlib as mpl
5 from matplotlib import pyplot
6
7
8 #--- [1] Read data and select dim
9
10 # you can read data from URL below
11 # https://raw.githubusercontent.com/VincentGranville/Main/main/circle8d.csv
12 data = pd.read_csv('circle8d.csv')
13 features = list(data.columns.values)
14 X = data.to_numpy()
15 features = np.array(features)
16
17 # use the first dim columns only
18 dim = 2
19 X = X[:, 0:dim]
20 features = features[0:dim]
21 nobs_real, dim = X.shape
22
23
24 #--- [2] Functions to build bin structure
25
26 def create_quantile_table(x, Hyperparam, shift):
27
28     arr_q = []
29     for d in range(dim):
30         n = Hyperparam[d]
31         arr_qd = np.zeros(n+1)
32         for k in range(n):
33             q = shift[d] + k/n
34             arr_qd[k] = np.quantile(x[:,d], q % 1)
35         arr_qd[n] = max(x[:,d])
36         arr_qd.sort()
37         arr_q.append(arr_qd)
38     return(arr_q)
39
40
41 def find_quantile_index(x, arr_quantiles):
42     k = 0
43     while k < len(arr_quantiles) and x > arr_quantiles[k]:
44         k += 1
45     return(max(0, k-1))
46
47
48 def create_bin_structure(x, arr_q):
49
50     hash_bins = {}
51     hash_bins_median = {}
52     hash_index = []
53
54     for n in range(x.shape[0]):
55
56         key = ()
57         for d in range(dim):

```

```

58         kd = find_quantile_index(x[n,d], arr_q[d])
59         key = (*key, kd)
60         hash_index.append(key)
61
62         if key in hash_bins:
63             hash_bins[key] += 1
64             points = hash_bins_median[key]
65             points.append(x[n,:])
66             hash_bins_median[key] = points
67         else:
68             hash_bins[key] = 1
69             hash_bins_median[key] = [x[n,:]]
70
71     for key in hash_bins:
72         points = hash_bins_median[key]
73         # beware: even number of points -> median is not one of the points
74         median = np.median(points, axis = 0)
75         hash_bins_median[key] = median
76
77     return(hash_bins, hash_index, hash_bins_median)
78
79
80 #--- [3] Generate initial nobs_synth obs
81
82 # if nobs_synth > 1000, split in smaller batches, do one batch at a time
83 nobs_synth = nobs_real
84 seed = 155
85 np.random.seed(seed)
86
87 # get initial synth. data with same marginal distributions as real data
88
89 mode = 'Shuffle' # options: 'Shuffle', 'Quantiles'
90 synth_X = np.empty(shape=(nobs_synth,dim))
91
92 if mode == 'Quantiles':
93     for k in range(nobs_synth):
94         pc = np.random.uniform(0, 1.00000001, dim)
95         for d in range(dim):
96             synth_X[k, d] = np.quantile(X[:,d], pc[d], axis=0)
97
98 elif mode == 'Shuffle':
99     nobs_synth == nobs_real # both must be equal
100     synth_X = np.copy(X)
101     for d in range(dim):
102         col = synth_X[:, d]
103         np.random.shuffle(col)
104         synth_X[:, d] = col
105
106 synth_X_init = np.copy(synth_X)
107
108
109 #--- [4] Main part: create synth obs to minimize Hellinger loss function
110
111 def in_bin(x, key, arr_q):
112     # test if vector x is in bin attached to key
113     status = True
114     for d in range(dim):
115         arr_qd = arr_q[d]
116         kd = key[d]
117         if x[d] < arr_qd[kd] or x[d] >= arr_qd[kd+1]:
118             status = False # x is not in the bin
119     return(status)
120
121 def array_to_tuple(arr):
122     list = ()
123     for k in range(len(arr)):
124         list = (*list, arr[k])
125     return(list)
126
127 Hellinger = 40.0 # arbitrary value
128 swaps = 0
129 history_log_H = []
130 history_log_swaps = []
131 flist = [] # list of image filenames for the video
132 frame = 0 # frame number, for video
133

```

```

134 # to accelerate computations (pre-computed sqrt)
135 sqrt_real = np.sqrt(nobs_real)
136 sqrt_synth = np.sqrt(nobs_synth)
137 n_sqrt = max(nobs_real, nobs_synth)
138 arr_sqrt = np.sqrt(np.arange(n_sqrt))
139
140 # visualization: graphic parameters
141 mpl.rcParams['lines.linewidth'] = 0.3
142 mpl.rcParams['axes.linewidth'] = 0.5
143 plt.rcParams['xtick.labelsize'] = 7
144 plt.rcParams['ytick.labelsize'] = 7
145
146 video_mode = False
147
148 # Hyperparameters
149 reset_granularity = 10000 # set to 50000 if dim = 3, set to 10000 if dim = 2
150 reset_shift = 99999999999
151 granularity = 20 # set to 2 if dim > 2, set to 20 if dim = 2
152 Hyperparam = np.full(dim, granularity)
153 shift = np.zeros(dim)
154 n_iter = 2000000
155
156
157 for iter in range(n_iter):
158
159     if iter % reset_granularity == 0 or iter % reset_shift == 0 or iter == 0:
160
161         # Get more granular Hellinger approximation
162
163         if iter % reset_granularity == 0:
164             Hyperparam = 1 + Hyperparam
165         if iter % reset_shift == 0:
166             shift = np.random.uniform(0, 1, dim)
167         arr_q = create_quantile_table(X, Hyperparam, shift)
168         ( hash_bins_real,
169           hash_index_real,
170           hash_bins_median_real
171         ) = create_bin_structure(X, arr_q)
172         ( hash_bins_synth,
173           hash_index_synth,
174           hash_bins_median_synth, # unused
175         ) = create_bin_structure(synth_X, arr_q)
176
177         k = np.random.randint(0, nobs_synth)
178         key_k = hash_index_synth[k]
179         scount1 = hash_bins_synth[key_k]
180         if key_k in hash_bins_real:
181             rcount1 = hash_bins_real[key_k]
182         else:
183             rcount1 = 0
184
185         l = np.random.randint(0, nobs_synth)
186         key_l = hash_index_synth[l]
187         scount2 = hash_bins_synth[key_l]
188         if key_l in hash_bins_real:
189             rcount2 = hash_bins_real[key_l]
190         else:
191             rcount2 = 0
192
193         d = np.random.randint(1, dim) # column 0 can stay fixed
194
195         new_key_k = np.copy(key_k)
196         new_key_l = np.copy(key_l)
197         new_key_k[d] = key_l[d]
198         new_key_l[d] = key_k[d]
199         new_key_k = array_to_tuple(new_key_k)
200         new_key_l = array_to_tuple(new_key_l)
201
202         if new_key_k in hash_bins_synth:
203             scount3 = hash_bins_synth[new_key_k]
204         else:
205             scount3 = 0
206         if new_key_k in hash_bins_real:
207             rcount3 = hash_bins_real[new_key_k]
208         else:
209             rcount3 = 0

```

```

210
211 if new_key_l in hash_bins_synth:
212     scount4 = hash_bins_synth[new_key_l]
213 else:
214     scount4 = 0
215 if new_key_l in hash_bins_real:
216     rcount4 = hash_bins_real[new_key_l]
217 else:
218     rcount4 = 0
219
220 A = arr_sqrt[scount1] /sqrt_synth - arr_sqrt[rcount1]/sqrt_real
221 B = arr_sqrt[scount1-1]/sqrt_synth - arr_sqrt[rcount1]/sqrt_real
222 C = arr_sqrt[scount2] /sqrt_synth - arr_sqrt[rcount2]/sqrt_real
223 D = arr_sqrt[scount2-1]/sqrt_synth - arr_sqrt[rcount2]/sqrt_real
224 E = arr_sqrt[scount3] /sqrt_synth - arr_sqrt[rcount3]/sqrt_real
225 F = arr_sqrt[scount3+1]/sqrt_synth - arr_sqrt[rcount3]/sqrt_real
226 G = arr_sqrt[scount4] /sqrt_synth - arr_sqrt[rcount4]/sqrt_real
227 H = arr_sqrt[scount4+1]/sqrt_synth - arr_sqrt[rcount4]/sqrt_real
228 delta_H = - A*A + B*B - C*C + D*D - E*E + F*F - G*G + H*H
229
230 if delta_H < -0.00001:
231
232     Hellinger += delta_H
233     swaps += 1
234
235     # update hash_index_synth and hash_bins_synth
236
237     hash_index_synth[k] = new_key_k
238     if new_key_k in hash_bins_synth:
239         hash_bins_synth[new_key_k] +=1
240     else:
241         hash_bins_synth[new_key_k] = 1
242     if hash_bins_synth[key_k] == 1:
243         del hash_bins_synth[key_k]
244     else:
245         hash_bins_synth[key_k] -= 1
246
247     hash_index_synth[l] = new_key_l
248     if new_key_l in hash_bins_synth:
249         hash_bins_synth[new_key_l] += 1
250     else:
251         hash_bins_synth[new_key_l] =1
252     if key_l in hash_bins_synth:
253         hash_bins_synth[key_l] -= 1
254     else:
255         hash_bins_synth[key_l] = 1
256
257     # update synthetic data
258
259     aux = synth_X[k, d]
260     synth_X[k, d] = synth_X[l, d]
261     synth_X[l, d] = aux
262
263     if video_mode and swaps % 25 == 0:
264
265         # save image for future inclusion in video
266         fname='nogan3_frame'+str(frame)+'.png'
267         flist.append(fname)
268         plt.scatter(synth_X[:,0], synth_X[:,1], s = 1.0)
269         plt.savefig(fname, dpi = 200)
270         plt.close()
271         frame += 1
272
273 if iter % 1000 == 0:
274
275     print("Iter: %7d | Loss: %9.6f | Swaps: %5d"
276           %(iter, Hellinger, swaps))
277     history_log_H.append(Hellinger)
278     history_log_swaps.append(swaps)
279
280
281 #--- [5] Evaluation with KS distance
282
283 import genai_evaluation as ge
284
285 n_nodes = 1000

```

```

286
287 df_init = pd.DataFrame(synth_X_init, columns = features)
288 df_synth = pd.DataFrame(synth_X, columns = features)
289 df_train = pd.DataFrame(X, columns = features)
290
291 query_lst, ecdf_train, ecdf_init = ge.multivariate_ecdf(df_train,
292     df_init, n_nodes, verbose = True)
293 ks_base = ge.ks_statistic(ecdf_train, ecdf_init)
294
295 query_lst, ecdf_train, ecdf_synth = ge.multivariate_ecdf(df_train,
296     df_synth, n_nodes, verbose = True)
297 ks = ge.ks_statistic(ecdf_train, ecdf_synth)
298
299 query_lst, ecdf_init, ecdf_synth = ge.multivariate_ecdf(df_init,
300     df_synth, n_nodes, verbose = True)
301 ks_diff = ge.ks_statistic(ecdf_init, ecdf_synth)
302
303 print("Test ECDF Kolmogorof-Smirnov dist. (synth. vs train.): %6.4f" %(ks))
304 print("Base ECDF Kolmogorof-Smirnov dist. (init. vs train.): %6.4f" %(ks_base))
305 print("Diff ECDF Kolmogorof-Smirnov dist. (init. vs synth.): %6.4f" %(ks_diff))
306
307
308 #--- [6] Plot some results and create video
309
310 mpl.rc('hatch', color='k', linewidth=0.3)
311 plt.scatter(X[:,0],X[:,1],marker='o',c='deepskyblue',alpha=0.1,s=10)
312 plt.scatter(synth_X[:,0],synth_X[:,1],marker='o',c='coral',alpha=0.4,s=10,
313     edgecolors='black',lw=0.2)
314 plt.grid(linewidth = 0.4, alpha = 1)
315 plt.show()
316
317 x_axis = range(len(history_log_H))
318 plt.plot(x_axis, history_log_H)
319 plt.show()
320 plt.plot(x_axis, history_log_swaps)
321 plt.show()
322
323 if video_mode:
324     import moviepy.video.io.ImageSequenceClip
325     clip = moviepy.video.io.ImageSequenceClip.ImageSequenceClip(flist, fps=6)
326     clip.write_videofile('nogan3.mp4')

```
