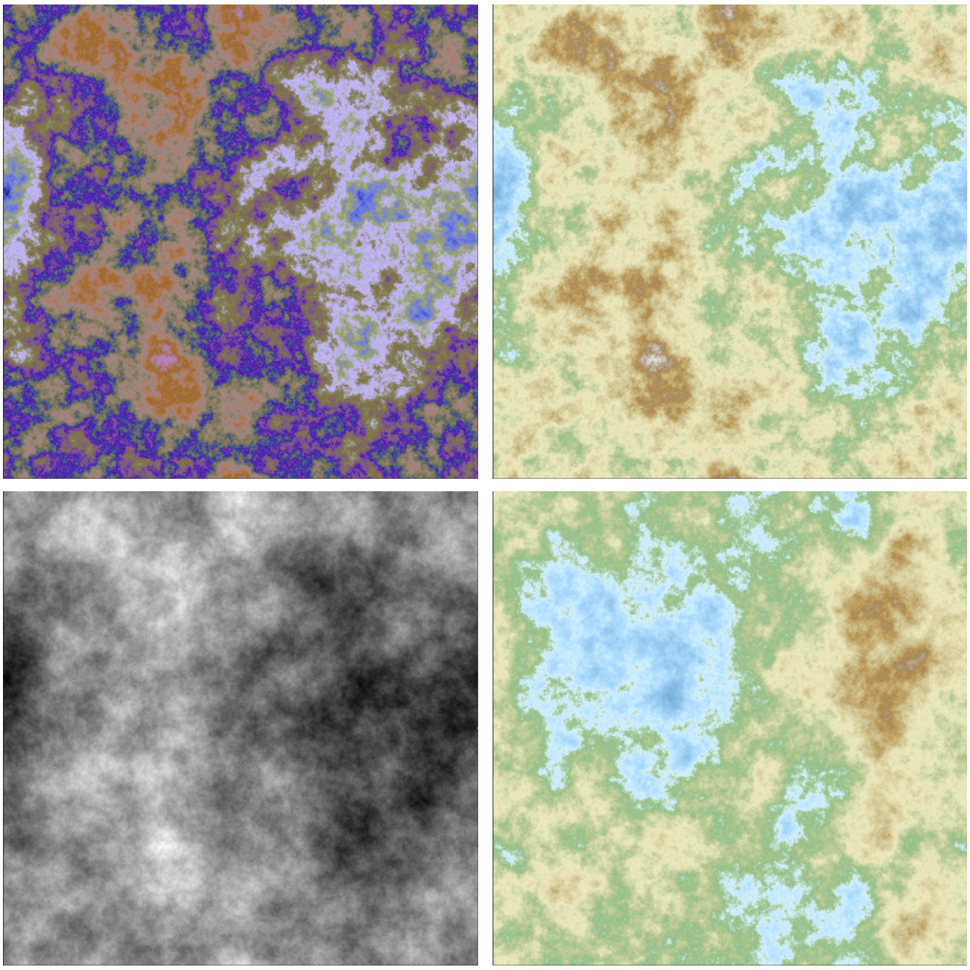
Synthetic Data



Vincent Granville, Ph.D. | www.MLTechniques.com | Version 2.1, December 2022

accent. Alexa always confuses it with 2:59 when I ask her, with my French accent, to set an alarm at 2:39 to go pick up my son at school. A simple fix in this case would be for Alexa to learn directly from me, recognize her error as I train her, and then get it fixed for good. This has the benefit to let Alexa adapt to each customer, offering customized chats rather than relying only on a central training set. Finding counter-examples to your system, to make it fail, in essence to crack it, is referred to as adversarial learning [Wiki]. It helps you better understand how your black-box works, and by integrating these tricky cases, it helps you improve your system.

Finally, generative adversarial networks (GANs) [Wiki] are popular in computer vision. Given a training set, this technique learns to generate new data with the same statistics as the training set. For example, a GAN trained on photographs can generate new photographs that look at least superficially authentic to human observers, having many realistic characteristics. To identify a fake GAN-generated picture from a real one of the same person based on iris parameters, see [52].

16.7.1 Sensitivity analysis, imputation and other uses of synthetic data

Synthetic data was originally developed as a method to replace missing values with synthetic ones: this is known as imputation [Wiki]. It did not work well as the missing values typically don't follow the underlying statistical model. One way to improve the technique is as follows. Use real or synthetic data (ideally, both) and remove some values to emulate a data set with missing values. Then replace the introduced missing values by synthetic ones. Try with a large number of parameter-driven simulations to see which parameter values are best at producing missing values. Another benefit of synthetic data is its ability to test model resilience: replace some real observations in your validation set with synthetic ones, and see how your predictions are sensitive to this change. Eliminate models that show lack of resilience. This is an easy way to reduce overfitting.

Another use of synthetic data is for bootstrapping [Wiki] or to compute confidence regions based on parametric bootstrap. Numerous examples are included in this book: see the keywords in question in the index. This simulation-heavy technique requires a large amount of data generated with the same parameter values as those estimated on your original data set. Also, synthetic data is used to benchmark and test algorithms. Again, this book features numerous examples. Finally, one way to correct for imbalanced data or to reduce algorithm biases is to over-sample groups or segments with few observations or representing minorities, when synthesizing your real data. The example in section 16.7.2 shows how to generate synthetic data at the group level, rather than globally. It also allows you to choose how many observations you want to generate, for each group.

16.7.2 Using copulas to generate synthetic data

One method to generate data with the exact same correlation structure and same marginal distributions as in your real dataset is to use copulas [Wiki]. The algorithm to produce n synthesized observations is as follows:

- Step 1: Compute the correlation matrix W associated to your real data.
- Step 2: Generate n deviates from a multivariate Gaussian distribution with zero mean and covariance matrix W. Each deviate is a vector Z_i $(i = 1, \dots, n)$, with the components matching the features in the real data set.
- Step 3: For each generated Z_{ij} (the j-th feature in your i-th vector) compute $U_{ij} = \Phi(Z_{ij})$, where Φ is the CDF (cumulative distribution function) of a univariate standard normal distribution. Thus $0 \leq U_{ij} \leq 1$.
- Step 4: Compute $S_{ij} = Q_j(U_{ij})$ where Q_j is the univariate empirical quantile distribution (the inverse of the empirical distribution) attached to the j-th feature, and computed on the real data.

Assuming W is non singular, your set of feature vectors S_i $(i = 1, \dots, n)$ is your synthesized data, mimicking your real data set. I implicitly used the Gaussian copula here, but other options exist. This method is a direct application of Sklar's theorem [Wiki]. There are various ways to measure the similarity or distance between the synthetic and real version of the data. In this context, the Hellinger distance [Wiki] is popular. You can also compare summary statistics. See also section 6.4 on comparing two datasets.

In Python, the four steps of the algorithm are performed respectively with the functions `np.corrcoef`, `np.random.multivariate_normal`, `norm.cdf` and `np.quantile`. Except for `norm.cdf` (CDF of standard Gaussian distribution) which comes from the Scipy library, the other ones are implemented in Numpy. To generate the exact same synthetic data each time you run the program, use `np.random.seed` with the same seed.

Finally, a few Python libraries deal with copulas, for instance Copulalib. The Python program `copula.py` on my GitHup repository (main folder) shows how it works. It is also possible to avoid copulas and deal directly with the correlation matrix, as in section 7.2.1. Or ignore correlations altogether, and simply add uncorrelated white noise to each feature: this may be the easiest way to generate synthetic data. This approach is significantly

Statistic	Real data	Synthetic 1	Synthetic 2
Mean age	39.21	39.21	38.61
Mean bmi	30.66	30.97	30.79
Mean charges	$13,270	$13,516	$13,253
Min age	18	18	18
Max age	64	64	64
Max charges	$63,770	$49,993	$59,588
Correl age, bmi	0.11	0.06	0.09
Correl age, children	0.04	0.02	0.03
Correl age, charges	0.30	0.29	0.30
Correl bmi, charges	0.20	0.14	0.18
Stdev children	1.21	1.19	1.20
Stdev charges	$12,110	$12,330	$12,132

Table 16.4: Comparing real data with two different synthetic copies

superior to copulas to generate synthetic values outside the range observed in the real data. It also preserves the correlation structure, and in some sense, generates richer data. Another popular method is rejection sampling [Wiki].

16.7.2.1 The insurance dataset: Python code and results

I used the algorithm in section 16.7.2 to synthesize the insurance dataset shared on Kaggle, here. The spreadsheet `insurance.xlsx` on GitHub (main folder) summarizes all the findings, and contains three datasets: the real data, synthetic data that I produced with `insurance.py` (same folder), and synthetic data generated by Mostly.ai. The dataset has the following fields: age, sex, bmi (body mass index), number of children covered by plan, smoker (yes or no), region (Northeast and so on), and charges incurred by the insurer for the customer in question.

I don't know what algorithm Mostly.ai uses, but their synthetic copy of the real data is strikingly similar to mine. Both have all the hallmarks (quality and defects) of being copula-generated. My version is slightly better because I generated a different copula (and thus, a different correlation matrix) for each group of observations. I grouped the observations by sex, smoker status and region while Mostly.ai applied the same copula across all these groups. Except for "bmi", the features are highly non-Gaussian: "charges" is bimodal, "number of children" has a geometric (discrete) distribution, and "age" is uniform except for the extremes.

Table 16.4 provides some high-level summary statistics: Synthetic 1 is produced by Mostly.ai, and Synthetic 2 using the methodology described here. The correlation structure is well reconstituted. Synthetic 2 is better than Synthetic 1 for "Max charges" (the maximum computed over all observations) because it generates separate copulas for each group. Also, this feature has a bimodal distribution.

The sore point is that copulas are unable to generate values outside the range observed in the real dataset: this is evident when looking at the maxima and minima. All data sets have the same number of observations. Even when I increase the number of synthesized observations by a factor 100, I am still stuck with a maximum charge less than $63,770, even though this ceiling is not an outlier in the real data. The same is true with "age", although this is compounded by the fact that ages 18 and 64 are cut-off points. The issue is that the quantile functions Q_j in section 16.7.2 generate values between the minimum and maximum observed in the real data, for each feature j. A workaround is to introduce uncorrelated white noise either in the real or synthetic data.

The Python code in this section can be optimized for speed as follows: pre-compute the empirical quantiles functions associated to each feature in the real dataset, as well as the CDF of the standard Gaussian distribution. In other words, use a table of pre-computed values.

```
import csv
from scipy.stats import norm
import numpy as np

filename = 'insurance.csv' # make sure fields don't contain commas
# source: https://www.kaggle.com/datasets/teertha/ushealthinsurancedataset

# Fields: age, sex, bmi, children, smoker, region, charges
```

```python
with open(filename, 'r') as csvfile:
   reader = csv.reader(csvfile)
   fields = next(reader) # Reads header row as a list
   rows = list(reader) # Reads all subsequent rows as a list of lists

#-- group by (sex, smoker, region)

groupCount = {}
groupList = {}
for obs in rows:
   group = obs[1] +"\t"+obs[4]+"\t"+obs[5]
   if group in groupCount:
      cnt = groupCount[group]
      groupList[(group,cnt)]=(obs[0],obs[2],obs[3],obs[6])
      groupCount[group] += 1
   else:
      groupList[(group,0)]=(obs[0],obs[2],obs[3],obs[6])
      groupCount[group] = 1

#-- generate synthetic data customized to each group (Gaussian copula)

seed = 453
np.random.seed(seed)
OUT=open("insurance_synth.txt","w")
for group in groupCount:
   nobs = groupCount[group]
   age = []
   bmi = []
   children = []
   charges = []
   for cnt in range(nobs):
      features = groupList[(group,cnt)]
      age.append(float(features[0])) # uniform outside very young or very old
      bmi.append(float(features[1])) # Gaussian distribution?
      children.append(float(features[2])) # geometric distribution?
      charges.append(float(features[3])) # bimodal, not gaussian

   mu = [np.mean(age), np.mean(bmi), np.mean(children), np.mean(charges)]
   zero = [0, 0, 0, 0]
   z = np.stack((age, bmi, children, charges), axis = 0)
   # cov = np.cov(z)
   corr = np.corrcoef(z) # correlation matrix for Gaussian copula for this group

   print("------------------")
   print("\n\nGroup: ",group,"[",cnt,"obs ]\n")
   print("mean age: %2d\nmean bmi: %2d\nmean children: %1.2f\nmean charges: %2d\n"
         % (mu[0],mu[1],mu[2],mu[3]))
   print("correlation matrix:\n")
   print(np.corrcoef(z),"\n")
   nobs_synth = nobs # number of synthetic obs to create for this group
   gfg = np.random.multivariate_normal(zero, corr, nobs_synth)
   g_age = gfg[:,0]
   g_bmi = gfg[:,1]
   g_children = gfg[:,2]
   g_charges = gfg[:,3]

   # generate nobs_synth observations for this group
   print("synthetic observations:\n")
   for k in range(nobs_synth):
      u_age = norm.cdf(g_age[k])
      u_bmi = norm.cdf(g_bmi[k])
      u_children = norm.cdf(g_children[k])
      u_charges = norm.cdf(g_charges[k])
      s_age = np.quantile(age, u_age)     # synthesized age
      s_bmi = np.quantile(bmi, u_bmi)     # synthesized bmi
      s_children = np.quantile(children, u_children) # synthesized children
```

```
    s_charges = np.quantile(charges, u_charges) # synthesized charges

    line =
      group+"\t"+str(s_age)+"\t"+str(s_bmi)+"\t"+str(s_children)+"\t"+str(s_charges)+"\n"
    OUT.write(line)
    print("%3d. %d %d %d %d" %(k, s_age, s_bmi, s_children, s_charges))
OUT.close()
```

16.8 Advice to beginners

In this section, I provide guidance to machine learning beginners. After finishing the reading and the accompanying classes (including successfully completing the student projects), you should have a strong exposure to the most important topics, many covered in detail in this book. At this point, you should be able to pursue the learning on your own – a never ending process even for top experts – in particular with the advice provided in section 16.8.1.

16.8.1 Getting started and learning how to learn

The first step is to install Python on your laptop. While it is possible to use Jupyter notebooks instead [Wiki], this option is limited and won't give you the full experience of writing and testing serious code as in a professional, business environment. Installing Python depends on your system. See the official Python.org website here to get started and download Python. On my Windows laptop, I first installed the Cygwin environment (see here how to install it) to emulate a Unix environment. That way, I can use Cygwin windows instead of the Windows command prompt [Wiki]. The benefits is that it recognizes Unix commands. An alternative to Cygwin is Ubuntu. You could also use the Anaconda environment.

Either way, you want to save your first Python program as a text file, say `test.py`. To run it, type in `Python test.py` in the command window. You need to be familiar with basic file management systems, to create folders and sub-folders as needed. Shortly, you will need to install Python libraries on your machine. Some of the most common ones are pandas, scypy, seaborn, numpy, random and matplotlib. You can create your own too, as illustrated in section 5.4.3. In your Python script, only use the needed libraries. Typically, they are listed at the beginning of your code, as in the following example:

```
import numpy as np
import matplotlib.pyplot as plt
import moviepy.video.io.ImageSequenceClip # to produce mp4 video
from PIL import Image # for some basic image processing
```

To install (say) the numpy library, type in `pip install numpy` in the Windowns command prompt.

16.8.1.1 Getting help

One of the easiest ways to learn more and solve new problems using Python or any programming language is to use the Internet. For instance, when I designed my sound generation algorithm in Python (see section 16.1), I googled keywords such as "Python sound processing". I quickly discovered a number of libraries and tutorials, ranging from simple to advanced. Over time, you discover websites that consistently offer solutions suited to your needs, and you tend to stick with them, until you "graduate" to the next level of expertise and use new resources.

It is important to look at the qualifications of people posting their code online, and how recent these posts are. You have to discriminate between multiple sources, and identify those that are not good or outdated. Usually, the best advice comes from little comments posted in discussion forums, as a response to solutions offered by some users. Of course, you can also post your own questions. Two valuable sources here to stay are GitHub and StackExchange. There are also numerous Python groups on LinkedIn and Reddit. In the end, after spending some time searching for sound libraries in Python, I've found solutions that do not require any special library: Numpy can process sound files. It took me a few hours to discover all I needed on the Internet.

Finally, the official documentation that comes with Python libraries can be useful, especially if you want to use special parameters and understand the inner workings (and limitations) rather than using them as black-boxes with the default settings. For instance, when looking at model-free parameter estimation for time series (using optimization techniques), I quickly discovered the `curve_fit` function from the Scipy library. It did not work well on my unusual datasets (see section 1.3.3.2). I discovered, in the official online documentation, several settings to improve the performance. Still unsatisfied with the results (due to numerical instability in