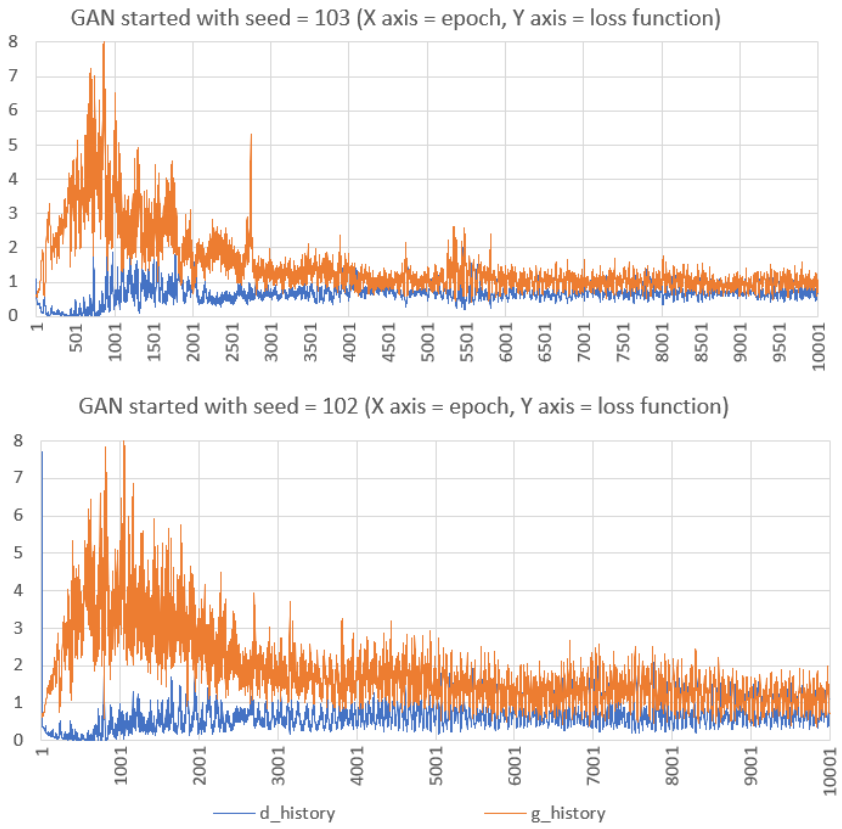
Practical AI & Machine Learning Projects and Datasets



Vincent Granville, Ph.D. | www.MLTechniques.com | Version 4.2, May 2024

Preface

This book is for participants in my AI and machine learning certification program. However, it is now free and available to everyone. With tutorials, enterprise-grade projects and solutions, it covers state-of-the-art material on topics such as generative adversarial networks (GAN), specialized LLM, data synthetization, as well as classical machine learning. It is a work in progress, as I regularly add new projects and new techniques.

The focus is on fast, simple, and better methods, by comparison to vendor solutions. For instance: NoGAN, better evaluation metrics, xLLM (specialized multi-LLM with taxonomies and self-tuning), variable-length embeddings, generating observations outside the training set range, fast probabilistic vector search, or automated SQL generation and execution. In classical ML, I discuss alternatives to traditional methods, for instance to synthesize geo-spatial data or music. Also, I show how to increase automation, speed, size and quality without neural networks.

Step 1: Summary tables. Besides embeddings, what are the summary tables and underlying data structure, for each of them? How are they linked to the output returned to a user query? How to adapt the methodology if the input source is Wikipedia rather than Wolfram?

Step 2: Issues with Python libraries. By looking at my comments in the code or by experimenting yourself, identify bad side effects from standard Python libraries such as NLTK. Examples include stopwords, auto-correct, and singularize. The problems arise due to the specialization of my xLLM, but is otherwise minor in generic applications targeted to non-experts. Suggest workarounds.

Step 3: Improving xLLM. I implemented several enhancements, for instance: ignoring N-grams (permutations of N tokens) [Wiki] not found in the crawled data, using normalized PMI in embeddings as the association metric or weight between two tokens, working with variable-length embeddings (VLE) rather than vectors, managing accented characters, not combining tokens separated by punctuation signs, and minimizing stemming such as singular form. Find other potential enhancements and how to implement them. For instance, making sure that "analysis of variance" and "ANOVA" return the same results, or "San Francisco" is not split into two tokens. Hint: use mappings (synonyms) and double-tokens treated as a single token. Treat uppercase and lowercase differently.

Figure 1: The first three steps in the main xLLM project

This textbook is an invaluable resource to instructors and professors teaching AI, or for corporate training. Also, it is useful to prepare for job interviews or to build a robust portfolio. And for hiring managers, there are plenty of original interview questions. The amount of Python code accompanying the solutions is considerable, using a vast array of libraries as well as home-made implementations showing the inner workings and improving existing black-box algorithms. By itself, this book constitutes a solid introduction to Python and scientific programming. The code is also on my GitHub repository.

About the author

Vincent Granville is a pioneering data scientist and machine learning expert, co-founder of Data Science Central (acquired by TechTarget), founder of MLTechniques.com, former VC-funded executive, author and patent owner.

Vincent's past corporate experience includes Visa, Wells Fargo, eBay, NBC, Microsoft, and CNET. Vincent is also a former post-doc at Cambridge University, and the National Institute of Statistical Sciences (NISS). He published in *Journal of Number Theory*, *Journal of the Royal Statistical Society* (Series B), and *IEEE Transactions on Pattern Analysis and Machine Intelligence*. He is also the author of multiple books, available here. He lives in Washington state, and enjoys doing research on stochastic processes, dynamical systems, experimental math and probabilistic number theory.

Contents

Chapter 1

Getting Started

If you are familiar with Python, you can skip this chapter. It explains different ways to install and work with Python, start with Jupyter Notebook if you want to, and post your projects or notebooks on GitHub. Along the way, you will learn how to produce a video in Python with sample code based on Plotly – a more advanced version of Matplotlib for scientific programming. You will also learn what Google Colab is about: a virtual server where you can run Python remotely in a Jupyter notebook, and sync with GitHub.

1.1 Python, Jupyter Notebook, and Google Colab

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 my books. At this point, you should be able to pursue the learning on your own – a never ending process even for top experts.

The first step is to install Python on your laptop. While it is possible to use Jupyter Notebook 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. Once Python is installed, you can install Notebook from within Python, on the command prompt [Wiki] with the instruction `python -m pip install jupyter`, see here. Or you may want to have it installed on a virtual machine [Wiki] on your laptop: see VMware virtual machine installation here. Or you can access Notebook remotely via Google Colab, see here and here. In my case, Jupyter automatically starts a virtual machine on my laptop: it does not interfere with or modify my standard Python environment, with few exceptions (when installing a third-party library that forces reinstallation of older versions of libraries already in my 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. The benefits is that it recognizes Unix commands. An alternative to Cygwin is Ubuntu [Wiki]. You could also use the Anaconda environment [Wiki].

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, Statsmodels, Scikit-learn, rRndom and Matplotlib. You can create your own too: see my `GD_util` library on GitHub, here and the Python script `PB_NN.py` that uses some of its functions, here. 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. In notebook, the command needs to be preceded by the exclamation point. You can also run Unix commands in a notebook cell: again, it needs to be preceded by the exclamation point, for instance `!pwd` or `!pip install tensorflow` or `!ls -l`. To include plots from Matplotlib to your notebook, add `%matplotlib inline` in

a cell before producing the images. See example here. You can also add HTML and LaTeX formula to make your document look like a webpage, both with internal and external links. Use the Markdown language [Wiki] to create notebook cells that include HTML to document your code (as opposed to Python cells consisting of code). In Markdown, LaTex (for math formulas) start and end with a dollar sign.

If running your notebook on Google Colab, you can automatically save it on GitHub. Or upload it on Colab, from GitHub. Even if you don't use Notebook, I strongly encourage you to create a GitHub account. It will help you with versioning [Wiki] and sharing your code. To copy and paste a piece of you code stored locally, into notebook, use `Ctrl-V`.

You can directly run a Notebook found on GitHub, cell by cell, in your own Colab environment. Try this one. Note that the original Notebook is in my GitHub repository, here. To avoid problems with local files, the dataset used in this Python program is fetched directly from where it is located on the web, in this case, also on my GitHub repository. This is accomplished as follows, using the Pandas library.

```
import pandas as pd

url="https://raw.githubusercontent.com/VincentGranville/Main/main/insurance.csv"
data = pd.read_csv(url)
print(data.head(10))
```

Finally, over time, after installing more and more Python libraries, you are going to face incompatibilities. For instance, TensorFlow automatically installs a number of libraries or relies on existed ones in your environment. Depending on which library versions are installed or exist in your environment, installing TensorFlow may or may not succeed. In many cases, reinstalling specific versions of some other libraries, or an older version of the library that you try to install, may fix the issue. In case of failed installation, check out the error message to see which libraries are causing problems, detect the versions currently on your system with `pip show pandas` (for the Pandas library in this example, assuming it is the cause of the failure) and install a different, compatible version with the command `pip install -I pandas==1.5.3`. In this example, version 1.5.3 and Pandas were mentioned in the error message to help me detect and fix the issue. In my current system (before the fix), I had Pandas 2.0 which was not compatible. The `-I` option forces the new installation and overwrites any other version you have in your system.

In some cases, making the recommended change results in some other library – needed with a specific version for a particular application – not to work anymore. For instance it requires Pandas 2.0 but you had to change it to Pandas 1.5.3 due to another compatibility issue with a different application. It creates a circular error loop impossible to fix. In this case, having two Python implementations with one on a virtual machine, may help.

1.1.1 Online Resources and Discussion Forums

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 my article "The Sound that Data Makes", here), 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. I discovered, in the official online documentation, several settings to improve the performance. Still unsatisfied with the results (due to numerical instability in my case), I searched for alternatives and discovered that the swarm optimization technique (an alternative to `curve_fit`) is available

in the Pyswarms library. In the end, testing these libraries on rich synthetic data allows you to find what works best for your data.

See also how I solve a new problem step by step and find the relevant code, in section 1.3. Below is a list of resources that I regularly use. They have been around for many years Each of them has its own search box, which is useful to identify specific topics in the vast amount of knowledge that they cover.

- StackExchange forum discussions about Python
- Reddit Machine Learning forum
- AnalyticsVidhya originally based in India
- Towards Data Science owned by Medium
- Machine Learning Mastery (popular blog, all in Python)
- Google data sets, Kaggle data sets
- OpenAI and other GPT apps for Python code

1.1.2 Beyond Python

Python has become the standard language for machine learning. Getting familiar with the R programming language will make you more competitive on the job market. In the last chapter in my book on synthetic data, I show how to create videos and better-looking charts in R. Finally, machine learning professionals should know at least the basics of SQL, since many jobs still involve working with traditional databases.

In particular, in one of the companies I was working for, I wrote a script that would accept SQL code as input (in text file format) to run queries against the Oracle databases, and trained analysts on how to use it in place of the dashboard they were familiar with. They were still using the dashboard (Toad in this case) to generate the SQL code, but run the actual queries with my script. The queries were now running 10 times faster: the productivity gain was tremendous.

To summarize, Python is the language of choice for machine learning, R is the language of statisticians, and SQL is the language of business and data analysts.

1.2 Automated data cleaning and exploratory analysis

It is said that data scientists spend 80% of their time on data cleaning and exploratory analysis [Wiki]. This should not be the case. To the beginner, it looks like each new dataset comes with a new set of challenges. Over time, you realize that there are only so many potential issues. Automating the data cleaning step can save you a lot of time, and eliminate boring, repetitive tasks. A good Python script allows you to automatically take care of most problems. Here, I review the most common ones.

First, you need to create a summary table for all features taken separately: the type (numerical, categorical data, text, or mixed). For each feature, get the top 5 values, with their frequencies. It could reveal a wrong or unassigned zip-code such as 99999. Look for other special values such as NaN (not a number), N/A, an incorrect date format, missing values (blank) or special characters. For instance, accented characters, commas, dollar, percentage signs and so on can cause issues with text parsing and regular expression [Wiki]. Compute the minimum, maximum, median and other percentiles for numerical features. Check for values that are out-of-range: if possible, get the expected range from your client before starting your analysis. Use checksums [Wiki] if possible, with encrypted fields such as credit card numbers or ID fields. A few Python libraries can take care of this. In particular: Pandas-Profiling, Sweetviz and D-Tale. See here for details.

The next step is to look at interactions between features. Compute all cross-correlations, and check for redundant or duplicate features that can be ignored. Look for IDs or keys that are duplicate or almost identical. Also two different IDs might have the same individual attached to them. This could reveal typos in your data. Working with a table of common typos can help. Also, collect data using pre-populated fields in web forms whenever possible, as opposed to users manually typing in their information such as state, city, zip-code, or date. Finally, check for misaligned fields. This happens frequently in NLP problems, where data such as URLs are parsed and stored in CSV files before being uploaded in databases. Now you can standardize your data.

Sometimes, the data has issues beyond your control. When I was working at Wells Fargo, internet session IDs generated by the Tealeaf software were broken down into multiple small sessions, resulting in wrong userIDs and very short Internet sessions. Manually simulating such sessions and looking how they were tracked in the database, helped solve this mystery, leading to correct analyses. Sometimes, the largest population segment is entirely missing in the database. For instance, in Covid data, people never tested who recovered on their own (the vast majority of the population in the early days) did not show up in any database, giving a lethality rate

of 6% rather than the more correct 1%, with costly public policy implications. Use common sense and out-of-the-box thinking to detect such issues, and let stakeholders known about it. Alternate data sources should always be used whenever possible. In this case, sewage data – a proxy dataset – provides the answer.

Finally, in many cases, transforming or standardizing your data may be necessary to get meaninful, consistent results. For instance, a log transform for stock prices makes sense. Or you may want all the continuous features to have zero mean and unit variance, possibly even decorrelate them. Is your algorithm invariant under change of scale? If not, using different units (for instance, day instead of hour) may result in different clusters or predictions. Metrics such as Mean Squared Error may be measured in "squared days", and you want to avoid that. As long as the transformation is reversible, you can apply your technique to the transformed data, than map back to the original using the inverse transform.

1.3 Tips to quickly solve new problems

The goal here is to describe the whole process of starting a Python project, facing roadblocks as always, and how to overcome them. It shows how to leverage external resources to quickly get satisfactory results. This is typically what professionals writing code do on a daily basis. You will learn how to create a video in Python, and get exposure to the Plotly library, a more advanced version of Matplotlib, typically used in the context of scientific programming.

1.3.1 Original solution to visualization problem

Assume you are asked to produce good quality, 3D contour plots in Python – not surface plots – and you have no idea how to start. Yet, you are familiar with Matplotlib, but rather new to Python. You have 48 hours to complete this project (two days of work at your office, minus the regular chores eating your time every day). How would you proceed? Here I explain how I did it, as I was in the same situation (self-imposed onto myself). I focus on the 3D contour plot only, as I knew beforehand how to quickly turn it into a video, as long as I was able to produce a decent single plot. My strategy is broken down into the following steps.

- I quickly realized it would be very easy to produce 2D contour or surface plots, but not 3D contour plots. I googled "contour map 3d matplotib". Not satisfied with the results, I searched images, rather than the web. This led me to a Stackoverflow forum question here, which in turn led me to some page on the Matplotlib website, here.
- After tweaking some parameters, I was able to produce Figure 1.1. Unsatisfied with the result and spending quite a bit of time trying to fix the glitch, I asked for a fix on Stackoverflow. You can see my question, and the answers that were posted, here. One participant suggested to use color transparency, but this was useless, as I had tried it before without success. The second answer came with a piece of code, and the author suggested that I used Plotly instead of Matplotlib. I trusted his advice, and got his code to work after installing Plotly (I got an error message asking me to install Kaleido, but that was easy to fix). Quite exciting, but that was far from the end.
- I googled "Matplotlib vs Plotly", to make sure it made sense using Plotly. I was convinced, especially given my interest in scientific computing. Quickly though, I realized my plots were arbitrarily truncated. I googled "plotly 3d contour map" and "plotly truncated 3d contour", which led me to various websites including a detailed description of the `layout` and `scene` parameters. This webpage was particularly useful, as it offered a solution to my problem.
- I spent a bit of time to figure out how to remove the axes and labels, as I feared they could cause problems in the video, changing from one frame to the next one, based on past experience. It took me 30 minutes to find the solution by trial and error. But then I realized that there was one problem left: in the PNG output image, the plot occupied only a small portion, even though it looked fine within the Python environment. Googling "plotly write_image" did not help. I tried to ask for a fix on Stackoverflow, but was not allowed to ask a question for another 24 hours. I asked my question in the Reddit Python forum instead.
- Eventually, shrinking the Z axis, modifying the orientation of the plot, the margins, and the dimensions of the images, I got a substantial improvement. By the time I checked for a potential answer to my Reddit question, my post had been deleted by an admin. But I had finally solved it. Well, almost.
- My concern at this point was using the correct DPI (dots per inch) and FPS (frames per second) for the video, and make sure the size of the video was manageable. Luckily, all the 300 frames (the PNG images, one per plot), now automatically generated, had the exact same physical dimension. Otherwise I would have had to resize them (which can be done automatically). Also, the rendering was good, not pixelized. So I did not have to apply anti-aliasing techniques. And here we are, I produced the video and was happy!

- So I thought. I realized, when producing the animated gif, that there was still a large portion of the images unused (blank). Not as bad as earlier, but still not good enough for me. Now I know how to crop hundreds of images automatically in Python, but instead I opted to load my video on Ezgif, and use the crop option. The final version posted in this chapter is this cropped version. I then produced another video, with 4 mountains, rising up, merging or shrinking according to various schedules. This might be the topic of a future article, as it is going into a new direction: video games.

The first version of my code, using Matplotlib, is available on GitHub here. It is also included in this section, and was used to produce Figure 1.1.

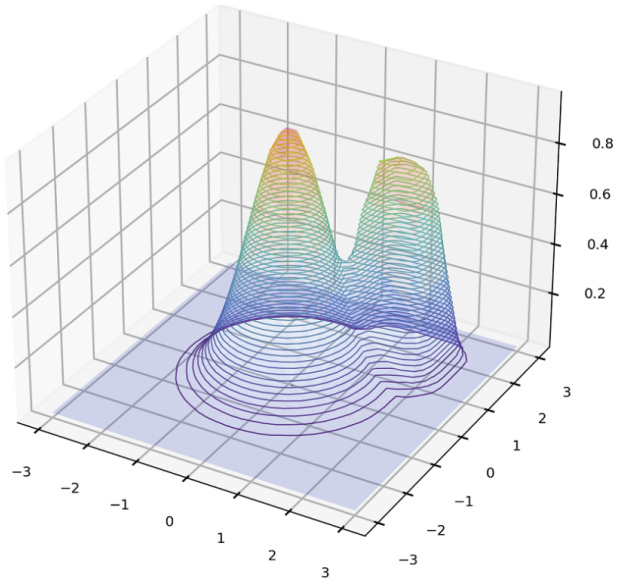


Figure 1.1: Same as Figure 1.2, produced with Matplotlib

```
import numpy as np
import matplotlib.pyplot as plt

plt.rcParams['lines.linewidth']= 0.5
plt.rcParams['axes.linewidth'] = 0.5
plt.rcParams['axes.linewidth'] = 0.5

SMALL_SIZE = 6
MEDIUM_SIZE = 8
BIGGER_SIZE = 10

plt.rc('font', size=SMALL_SIZE) # controls default text sizes
plt.rc('axes', titlesize=SMALL_SIZE) # fontsize of the axes title
plt.rc('axes', labelsize=MEDIUM_SIZE) # fontsize of the x and y labels
plt.rc('xtick', labelsize=SMALL_SIZE) # fontsize of the tick labels
plt.rc('ytick', labelsize=SMALL_SIZE) # fontsize of the tick labels
plt.rc('legend', fontsize=SMALL_SIZE) # legend fontsize
plt.rc('figure', titlesize=BIGGER_SIZE) # fontsize of the figure title

fig = plt.figure()
ax = fig.add_subplot(111, projection="3d")
X, Y = np.mgrid[-3:3:30j, -3:3:30j]
Z= np.exp(-(abs(X)**2 + abs(Y)**2)) + 0.8*np.exp(-4*((abs(X-1.5))**4.2 +
    (abs(Y-1.4))**4.2))

ax.plot_surface(X, Y, Z, cmap="coolwarm", rstride=1, cstride=1, alpha=0.2)
# ax.contourf(X, Y, Z, levels=60, colors="k", linestyles="solid", alpha=0.9,
    antialiased=True)
ax.contour(X, Y, Z, levels=60, linestyles="solid", alpha=0.9, antialiased=True)

plt.savefig('contour3D.png', dpi=300)
plt.show()
```

1.3.2 New solution, after doing some research

You can see the final result in Figure 1.2, and watch the corresponding video here. The code in this section corresponds to the Plotly version, including the production of the video. The choice of the colors is determined by the parameter `colorscale`, set to "Peach" here. The list of available palettes is posted here. You can easily add axes and labels, change font sizes and so on. The parameters to handle this are present in the source code, but turned off in the present version. The code is also available on GitHub, here.

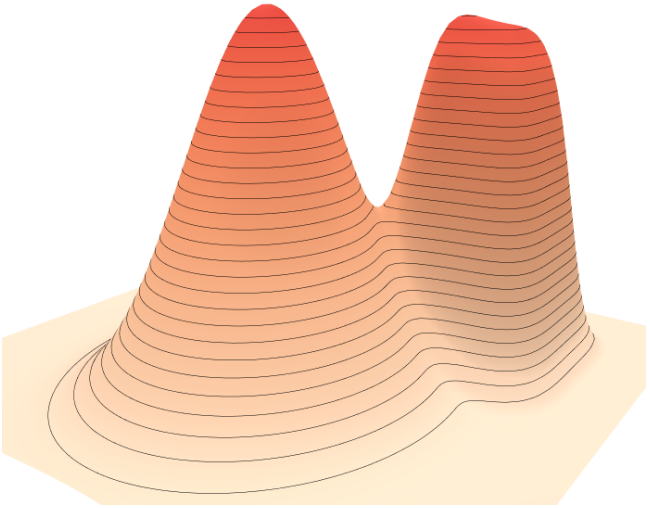


Figure 1.2: Contour plot, 3D mixture model, produced with Plotly

```
import numpy as np
import plotly.graph_objects as go

def create_3Dplot(frame):

   param1=-0.15 + 0.65*(1-np.exp(-3*frame/Nframes)) # height of small hill
   param2=-1+2*frame/(Nframes-1) # rotation, x
   param3=0.75+(1-frame/(Nframes-1)) # rotation, y
   param4=1-0.7*frame/(Nframes-1) # rotation z

   X, Y = np.mgrid[-3:2:100j, -3:3:100j]
   Z= 0.5*np.exp(-(abs(X)**2 + abs(Y)**2)) \
      + param1*np.exp(-4*((abs(X+1.5))**4.2 + (abs(Y-1.4))**4.2))

   fig = go.Figure(data=[
      go.Surface(
         x=X, y=Y, z=Z,
         opacity=1.0,
         contours={
            "z": {"show": True, "start": 0, "end": 1, "size": 1/60,
                 "width": 1, "color": 'black'} # add <"usecolormap": True>
         },
         showscale=False, # try <showscale=True>
         colorscale='Peach')],
   )

   fig.update_layout(
      margin=dict(l=0,r=0,t=0,b=160),
      font=dict(color='blue'),
      scene = dict(xaxis_title='', yaxis_title='',zaxis_title='',
         xaxis_visible=False, yaxis_visible=False, zaxis_visible=False,
         aspectratio=dict(x=1, y=1, z=0.6)),         # resize by shrinkink z
      scene_camera = dict(eye=dict(x=param2, y=param3, z=param4))) # change vantage point

   return(fig)

#-- main
```

```
import moviepy.video.io.ImageSequenceClip # to produce mp4 video
from PIL import Image # for some basic image processing

Nframes=300 # must be > 50
flist=[] # list of image filenames for the video
w, h, dpi = 4, 3, 300 # width and heigh in inches
fps=10 # frames per second

for frame in range(0,Nframes):
   image='contour'+str(frame)+'.png' # filename of image in current frame
   print("Creating image",image) # show progress on the screen
   fig=create_3Dplot(frame)
   fig.write_image(file=image, width=w*dpi, height=h*dpi, scale=1)
   # fig.show()
   flist.append(image)

# output video / fps is number of frames per second
clip = moviepy.video.io.ImageSequenceClip.ImageSequenceClip(flist, fps=fps)
clip.write_videofile('contourvideo.mp4')
```

Chapter 2

Machine Learning Optimization

This chapter covers several fundamental techniques to solve various machine learning problems. The focus is on optimization, either for speed or quality of the results (ideally both simulatneously), featuring state-of-the-art methods that outperform many neural network blackbox systems. As far as possible, the methods in question lead to intuitive and explainable AI. Project 2.1 is actually about generative AI. However, I decided to include it in this chapter as it does not use neural networks.

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 [14]. 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. See also the glossary in appendix C for additional help.

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 $\max |F_{\text{v}}(z) - F_{\text{s}}(z)|$ over all locations z in the feature space, where $F_{\text{v}}(z)$ and $F_{\text{s}}(z)$ 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 $[n_1, \ldots, n_d]$ 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 [18].

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 order 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 [18]. 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 [17]. 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.

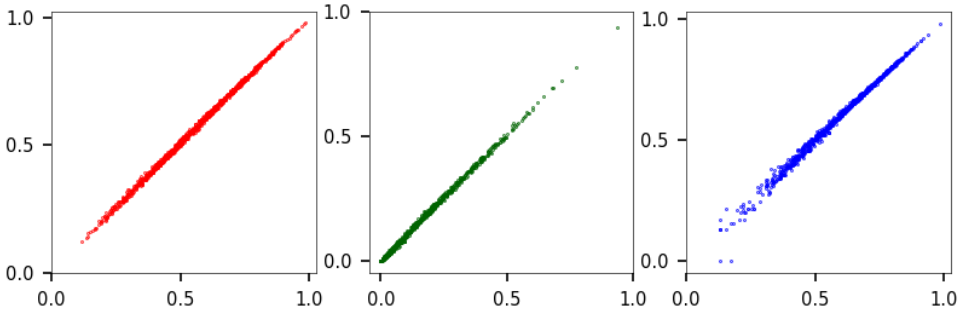


Figure 2.1: ECDF scatterplots (validation vs synthtetic) computed three ways

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" [16].

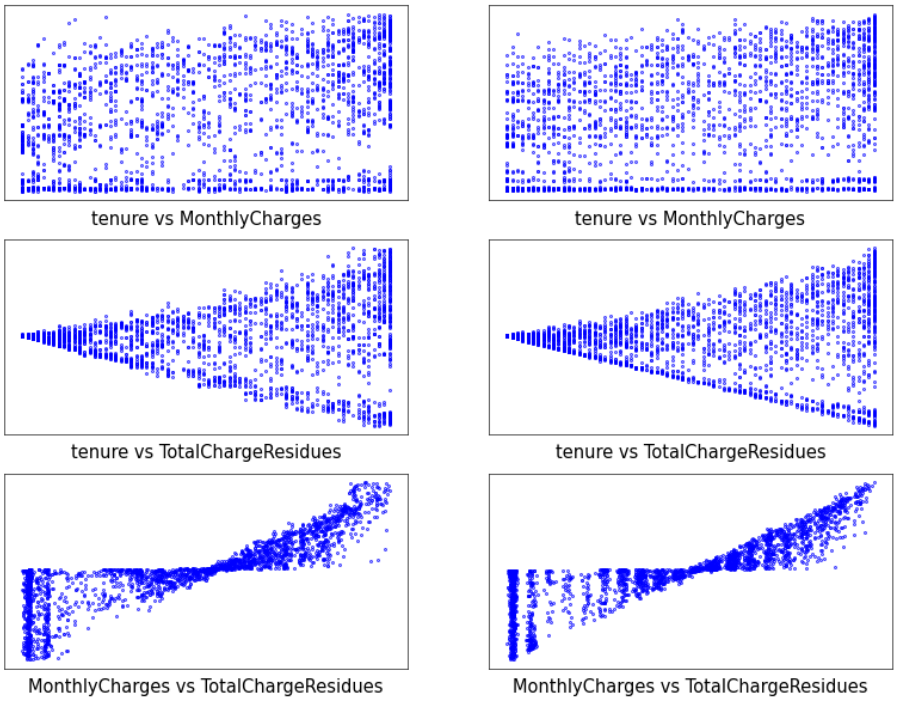


Figure 2.2: Feature scatterplots, synthetic (left) and validation dataset (right)

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. This is illustrated in project 2.2.

Now answering Step 6. First, a great hyperarameter 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 [14]. 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.

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from matplotlib import pyplot
from statsmodels.distributions.empirical_distribution import ECDF

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

#- [1.1] utility functions

def string_to_numbers(string):

    string = string.replace("[", "")
    string = string.replace("]", "")
    string = string.replace(" ", "")
    arr = string.split(',')
    arr = [eval(i) for i in arr]
    return(arr)

def category_to_integer(category):
    if category == 'Yes':
        integer = 1
    elif category == 'No':
        integer = 0
    else:
        integer = 2
    return(integer)

#- [1.2] read data
```

```
url = "https://raw.githubusercontent.com/VincentGranville/Main/main/Telecom.csv"
data = pd.read_csv(url)
features = ['tenure', 'MonthlyCharges', 'TotalCharges','Churn']
data['Churn'] = data['Churn'].map(category_to_integer)
data['TotalCharges'].replace(' ', np.nan, inplace=True)
data.dropna(subset=['TotalCharges'], inplace=True) # remove missing data
print(data.head())
print (data.shape)
print (data.columns)

#- [1.3] transforming TotalCharges to TotalChargeResidues, add to dataframe

arr1 = data['tenure'].to_numpy()
arr2 = data['TotalCharges'].to_numpy()
arr2 = arr2.astype(float)
residues = arr2 - arr1 * np.sum(arr2) / np.sum(arr1) # also try arr2/arr1
data['TotalChargeResidues'] = residues

#- [1.4] set seed for replicability

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

#- [1.5] initialize hyperparameters (bins_per_feature), select features

features = ['tenure','MonthlyCharges','TotalChargeResidues','Churn']
bins_per_feature = [50, 40, 40, 4]

bins_per_feature = np.array(bins_per_feature).astype(int)
data = data[features]
print(data.head())
print (data.shape)
print (data.columns)

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

data_training = data.sample(frac = 0.5)
data_validation = data.drop(data_training.index)
data_training.to_csv('telecom_training_vg2.csv')
data_validation.to_csv('telecom_validation_vg2.csv')

nobs = len(data_training)
n_features = len(features)
eps = 0.0000000001


#--- [2] create synthetic data

#- [2.1] create quantile table pc_table2, one row for each feature

pc_table2 = []
for k in range(n_features):
   label = features[k]
   incr = 1 / bins_per_feature[k]
   pc = np.arange(0, 1 + eps, incr)
   arr = np.quantile(data_training[label], pc, axis=0)
   pc_table2.append(arr)

#- [2.2] create/update bin for each obs [layer 1]
#      Faster implementation: replace 'while' loop by dichotomic search

npdata = pd.DataFrame.to_numpy(data_training[features])
bin_count = {} # number of obs per bin
bin_obs = {} # list of obs in each bin, separated by "~", stored as a string
```

```
for obs in npdata:
   key = []
   for k in range(n_features):
      idx = 0
      arr = pc_table2[k] # percentiles for feature k
      while obs[k] >= arr[idx] and idx < bins_per_feature[k]:
         idx = idx + 1
      idx = idx - 1 # lower bound for feature k in bin[key] attached to obs
      key.append(idx)
      skey = str(key)
   if skey in bin_count:
      bin_count[skey] += 1
      bin_obs[skey] += "~" + str(obs)
   else:
      bin_count[skey] = 1
      bin_obs[skey] = str(obs)

#- [2.3] generate nobs_synth observations (if mode = FixedCounts, nobs_synth = nobs)

def random_bin_counts(n, bin_count):
   # generate multinomial bin counts with same expectation as real counts
   pvals = []
   for skey in bin_count:
      pvals.append(bin_count[skey]/nobs)
   return(np.random.multinomial(n, pvals))

def get_obs_in_bin(bin_obs, skey):
   # get list of observations (real data) in bin skey, also return median
   arr_obs = []
   arr_obs_aux = (bin_obs[skey]).split('~')
   for obs in arr_obs_aux:
      obs = ' '.join(obs.split())
      obs = obs.replace("[ ", "")
      obs = obs.replace("[", "")
      obs = obs.replace(" ]", "")
      obs = obs.replace("]", "")
      obs = obs.split(' ')
      obs = (np.array(obs)).astype(float)
      arr_obs.append(obs)
   arr_obs = np.array(arr_obs)
   median = np.median(arr_obs, axis = 0)
   return(arr_obs, median)


mode = 'RandomCounts' # (options: 'FixedCounts' or 'RandomCounts')
if mode == 'RandomCounts':
   nobs_synth = nobs
   bin_count_random = random_bin_counts(nobs_synth, bin_count)
   ikey = 0

data_synth = []
bin_counter = 0

for skey in bin_count:

   if mode == 'FixedCounts':
      count = bin_count[skey]
   elif mode == 'RandomCounts':
      count = bin_count_random[ikey]
      ikey += 1
   key = string_to_numbers(skey)
   L_bounds = []
   U_bounds = []
   bin_counter += 1

   for k in range(n_features):
```

```
            arr = pc_table2[k]
            L_bounds.append(arr[key[k]])
            U_bounds.append(arr[1 + key[k]])

        # sample new synth obs (new_obs) in rectangular bin skey, uniformily
        # try other distrib, like multivariate Gaussian around bin median
        # the list of real observations in bin[skey] is stored in obs_list (numpy array)
        # median is the vector of medians for all obs in bin skey

        obs_list, median = get_obs_in_bin(bin_obs, skey) # not used in this version

        for i in range(count):
            new_obs = np.empty(n_features) # synthesized obs
            for k in range(n_features):
                new_obs[k] = np.random.uniform(L_bounds[k],U_bounds[k])
            data_synth.append(new_obs)

        str_median = str(["%8.2f" % number for number in median])
        str_median = str_median.replace("'","")
        print("bin ID = %5d | count = %5d | median = %s | bin key = %s"
                %(bin_counter, bin_count[skey], str_median, skey))

data_synth = pd.DataFrame(data_synth, columns = features)

# apply floor function (not round) to categorical/ordinal features
data_synth['Churn'] = data_synth['Churn'].astype('int')
data_synth['tenure'] = data_synth['tenure'].astype('int')

print(data_synth)
data_synth.to_csv('telecom_synth_vg2.csv')


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

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

#- [3.1] compute ecdf on validation set (to later compare with that on synth data)

def compute_ecdf(dataframe, n_nodes, adjusted):

    # Monte-Carlo: sampling n_nodes locations (combos) for ecdf
    #   - adjusted correct for sparsity in high ecdf, but is sparse in low ecdf
    #   - non-adjusted is the other way around
    # for faster computation: pre-compute percentiles for each feature
    # for faster computation: optimize the computation of n_nodes SQL-like queries

    ecdf = {}

    for point in range(n_nodes):

        if point % 100 == 0:
            print("sampling ecdf, location = %4d (adjusted = %s):" % (point, adjusted))
        combo = np.random.uniform(0, 1, n_features)
        if adjusted:
            combo = combo**(1/n_features)
        z = [] # multivariate quantile
        query_string = ""
        for k in range(n_features):
            label = features[k]
            dr = data_validation[label]
            percentile = combo[k]
            z.append(eps + np.quantile(dr, percentile))
            if k == 0:
                query_string += "{} <= {}".format(label, z[k])
            else:
```

```
            query_string += " and {} <= {}".format(label, z[k])

        countifs = len(data_validation.query(query_string))
        if countifs > 0:
            ecdf[str(z)] = countifs / len(data_validation)

    ecdf = dict(sorted(ecdf.items(), key=lambda item: item[1]))

    # extract table with locations (ecdf argument) and ecdf values:
    #    - cosmetic change to return output easier to handle than ecdf

    idx = 0
    arr_location = []
    arr_value = []
    for location in ecdf:
        value = ecdf[location]
        location = string_to_numbers(location)
        arr_location.append(location)
        arr_value.append(value)
        idx += 1

    print("\n")
    return(arr_location, arr_value)


n_nodes = 1000 # number of random locations in feature space, where ecdf is computed
reseed = False
if reseed:
  seed = 555
  np.random.seed(seed)
arr_location1, arr_value1 = compute_ecdf(data_validation, n_nodes, adjusted = True)
arr_location2, arr_value2 = compute_ecdf(data_validation, n_nodes, adjusted = False)

#- [3.2] comparison: synthetic (based on training set) vs real (validation set)

def ks_delta(SyntheticData, locations, ecdf_ValidationSet):

    # SyntheticData is a dataframe
    # locations are the points in the feature space where ecdf is computed
    # for the validation set, ecdf values are stored in ecdf_ValidationSet
    # here we compute ecdf for the synthetic data, at the specified locations
    # output ks_max in [0, 1] with 0 = best, 1 = worst

    ks_max = 0
    ecdf_real = []
    ecdf_synth = []
    for idx in range(len(locations)):
        location = locations[idx]
        value = ecdf_ValidationSet[idx]
        query_string = ""
        for k in range(n_features):
            label = features[k]
            if k == 0:
                query_string += "{} <= {}".format(label, location[k])
            else:
                query_string += " and {} <= {}".format(label, location[k])
        countifs = len(SyntheticData.query(query_string))
        synth_value = countifs / len(SyntheticData)
        ks = abs(value - synth_value)
        ecdf_real.append(value)
        ecdf_synth.append(synth_value)
        if ks > ks_max:
            ks_max = ks
        # print("location ID: %6d | ecdf_real: %6.4f | ecdf_synth: %6.4f"
        #         %(idx, value, synth_value))
    return(ks_max, ecdf_real, ecdf_synth)
```

```
df = pd.read_csv('telecom_synth_vg2.csv')
ks_max1, ecdf_real1, ecdf_synth1 = ks_delta(df, arr_location1, arr_value1)
ks_max2, ecdf_real2, ecdf_synth2 = ks_delta(df, arr_location2, arr_value2)
ks_max = max(ks_max1, ks_max2)
print("Test ECDF Kolmogorof-Smirnov dist. (synth. vs valid.): %6.4f" %(ks_max))

#- [3.3] comparison: training versus validation set

df = pd.read_csv('telecom_training_vg2.csv')
base_ks_max1, ecdf_real1, ecdf_synth1 = ks_delta(df, arr_location1, arr_value1)
base_ks_max2, ecdf_real2, ecdf_synth2 = ks_delta(df, arr_location2, arr_value2)
base_ks_max = max(base_ks_max1, base_ks_max2)
print("Base ECDF Kolmogorof-Smirnov dist. (train. vs valid.): %6.4f" %(base_ks_max))


#--- [4] visualizations

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

    # customized plots, subplot position based on counter

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

def vg_histo(df, feature, counter):

    # customized plots, subplot position based on counter

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

import matplotlib.pyplot as plt
import matplotlib as mpl
mpl.rcParams['axes.linewidth'] = 0.3

#- [4.1] scatterplots for Churn = 'No'

dfs = pd.read_csv('telecom_synth_vg2.csv')
dfs.drop(dfs[dfs['Churn'] == 0].index, inplace = True)
dfv = pd.read_csv('telecom_validation_vg2.csv')
dfv.drop(dfv[dfv['Churn'] == 0].index, inplace = True)

vg_scatter(dfs, 'tenure', 'MonthlyCharges', 1)
vg_scatter(dfv, 'tenure', 'MonthlyCharges', 2)
vg_scatter(dfs, 'tenure', 'TotalChargeResidues', 3)
vg_scatter(dfv, 'tenure', 'TotalChargeResidues', 4)
vg_scatter(dfs, 'MonthlyCharges', 'TotalChargeResidues', 5)
```

```
vg_scatter(dfv, 'MonthlyCharges', 'TotalChargeResidues', 6)
plt.show()

#- [4.2] scatterplots for Churn = 'Yes'

dfs = pd.read_csv('telecom_synth_vg2.csv')
dfs.drop(dfs[dfs['Churn'] == 1].index, inplace = True)
dfv = pd.read_csv('telecom_validation_vg2.csv')
dfv.drop(dfv[dfv['Churn'] == 1].index, inplace = True)

vg_scatter(dfs, 'tenure', 'MonthlyCharges', 1)
vg_scatter(dfv, 'tenure', 'MonthlyCharges', 2)
vg_scatter(dfs, 'tenure', 'TotalChargeResidues', 3)
vg_scatter(dfv, 'tenure', 'TotalChargeResidues', 4)
vg_scatter(dfs, 'MonthlyCharges', 'TotalChargeResidues', 5)
vg_scatter(dfv, 'MonthlyCharges', 'TotalChargeResidues', 6)
plt.show()

#- [4.3] ECDF scatterplot: validation set vs. synth data

plt.xticks(fontsize=7)
plt.yticks(fontsize=7)
plt.scatter(ecdf_real1, ecdf_synth1, s = 0.1, c ="blue")
plt.scatter(ecdf_real2, ecdf_synth2, s = 0.1, c ="blue")
plt.show()

#- [4.4] histograms, Churn = 'No'

dfs = pd.read_csv('telecom_synth_vg2.csv')
dfs.drop(dfs[dfs['Churn'] == 0].index, inplace = True)
dfv = pd.read_csv('telecom_validation_vg2.csv')
dfv.drop(dfv[dfv['Churn'] == 0].index, inplace = True)
vg_histo(dfs, 'tenure', 1)
vg_histo(dfs, 'MonthlyCharges', 2)
vg_histo(dfs, 'TotalChargeResidues', 3)
vg_histo(dfv, 'tenure', 4)
vg_histo(dfv, 'MonthlyCharges', 5)
vg_histo(dfv, 'TotalChargeResidues', 6)
plt.show()

#- [4.5] histograms, Churn = 'Yes'

dfs = pd.read_csv('telecom_synth_vg2.csv')
dfs.drop(dfs[dfs['Churn'] == 1].index, inplace = True)
dfv = pd.read_csv('telecom_validation_vg2.csv')
dfv.drop(dfv[dfv['Churn'] == 1].index, inplace = True)
vg_histo(dfs, 'tenure', 1)
vg_histo(dfs, 'MonthlyCharges', 2)
vg_histo(dfs, 'TotalChargeResidues', 3)
vg_histo(dfv, 'tenure', 4)
vg_histo(dfv, 'MonthlyCharges', 5)
vg_histo(dfv, 'TotalChargeResidues', 6)
plt.show()
```

2.2 Cybersecurity: balancing data with automated SQL queries

While the end goal is to synthesize a very challenging dataset, highly imbalanced, the scope of this project is not data synthetization. Rather, it consists of splitting the dataset into a small number of rather homogeneous parts, each part having specific characteristics. By implementing an algorithm separately to each part, one can expect much better results, whether for data synthetization or any other purpose. At the end, the different outputs are recombined together.

What makes this dataset so special? It has about 117k observations, and 16 features There are two big clusters of duplicate observations, representing about 95% of the dataset: each contains a single observation

vector, repeated time and over. Then a smaller number of clusters, each consisting of 2 to 5 duplicate observations. Most features are categorical, and some are a mixture of categorical and numerical values. Then there are some obvious outliers. This is not an error, it is the way the data is. The case study is about cybersecurity, looking at server data to identify fraud. The amount of fraud is also very small. In this project, most of the heavy work consists of identifying and separating the different parts. It is done using SQL-like statements in Pandas (Python library). See code in section 2.2.3, after completing the project.

2.2.1 Project description

The dataset is located here. First, you need to do some exploratory analysis to find the peculiarities. For each feature, find the distinct values, their type (category or continuous) and count the multiplicity of each value. Also count the multiplicity of each observation vector when all features are combined. We will split the dataset into three subsets named `A`, `C1` and `C2`, and remove some outliers in `C1` (or at least treat them separately). Simulating observations distributed as in `A` or `C1` is straighforward. For `C2`, we will use the NoGAN synthesizer. In the remaining, by observation or observation vector, I mean a full row in the dataset.

The project consists of the following steps:

Step 1: Split the data into two subsets `A` and `B`. Here `A` consists of the two big clusters discussed earlier, each containing one observation vector duplicated thousands of times. Also add to `A` any observation duplicated at least 4 times. Keep one copy of each unique observation in `A`, and add one new feature: the observation count, named `size`. The set `B` contains all the unique observations except those that are now in `A`, with a count (`size`) for each observation.

Step 2: Create set `C` as follows. Remove the columns `scr_port` and `size` from set `B`. Then keep only one copy of each duplicated observation, and add an extra feature to count the multiplicity attached to each observation. Finally, split `C` into `C1` and `C2`, with `C1` consisting of observation vectors with multiplicity larger than one, and `C2` for the other ones (observation vectors that do not have duplicates). Find outliers in `C1` and remove them.

Step 3: The feature `scr_port` is absent in sets `C1` and `C2`, after step 2. Reconstruct the list of values for `scr_port` with the correct count, separately for `C1` and `C2`, as we will need it in step 4. These satellite tables are named `map_C1` and `map_C2` in the Python code. Double check that all the computations, splitting, mapping, counts, uniques, and aggregation, are correct.

Step 4: Generate synthetic observations for `C2`, using the NoGAN algorithm described in project 2.1. Use the following features only:

```
bidirectional_syn_packets
src2dst_syn_packets
application_category_name
application_confidence
src2dst_mean_ps
src2dst_psh_packets
bidirectional_mean_ps
label
```

The feature "label" indicates fraud when the value is not 0. Few observations are labeled as non-fraud in `C2`. How would you proceed to substantially increase the proportion of non-fraud in `C2`, in the generated data? How about generating values for the `src_port` feature, based on the `map_C2` distribution obtained in step 3? Is the distribution in question uniform within its range, between 40,000 and 60,000? If yes, you could generate uniform values for this feature, possibly different from those actually observed.

Step 5: Think of a general strategy to synthesize observations not just for `C2`, but for the full original data. Say you want $N = 10^5$ synthetic observations. You need to generate n_1, n_2, n_3 observations respectively for `A`, `C1`, `C2`, with $N = n_1 + n_2 + n_3$, using a multinomial distribution of parameter $[N; p_1, p_2, p_3]$ where p_1, p_2, p_3 are the the proportions of observations falling respectively in `A`, `C1`, and `C2`.

What are the values of p_1, p_2, p_3? Finally, describe how you would proceed to synthesize observations separately for `A`, `C1`, and `C2`, once the random observation counts n_1, n_2, n_3 have been generated.

2.2.2 Solution

The code to produce `A`, `C1`, and `C2` is in section 2.2.3. It also produces `C1_full` and `C2_full`, identical to `C1` and `C2` except that duplicate observations are now kept as duplicates, rather than aggregated. This completes

Step 1 and Step 2. The same code produces `map_C1` and `map_C2`. All these tables are saved as separate tabs in a spreadsheet `iot_security.xls`, available on GitHub, here.

To check that all the counts are correct, compute the full number of observations in each subset, and verify that the sum matches the number of observations in the original data. For instance, `C1` has 70 unique (distinct) observations or rows, with multiplicity stored in the column `size`. The sum of this column is 7158, representing the actual number of observations. Likewise, `C2` has 87 rows and 110 observations when counting the duplicates. And `A` has 23 rows and 110,467 observations. Finally, $110 + 7158 + 110{,}467 = 117{,}735$, matching the number of rows in the original dataset. This completes Step 3.

To synthesize `C2`, I used a minimalist version of the NoGAN code in project 2.1. This updated version is very generic, with all the selected features declared in section [1.2] in the code, along with the sublist of those that are categorical. The code is on GitHub, here. The results – synthetic data, validation and training sets, along with some evaluation metrics – are in the `C2_Synth_NoGAN` tab in the `iot_security.xlsx` spreadsheet, here. I used the hyperparameter `[80,80,80,80,80,80,80,80]` where each component represents the number of bins used for the corresponding feature, whether continuous or categorical.

The NoGAN synthesizer works as follows. It splits the real data, in this case `C2_full`, into two subsets: the training set to generate new observations, and the validation set to check how good the generated observations are, by comparing the joint empirical distribution function (ECDF) of the synthetic data, with that of the validation set. This cross-validation technique is known as the holdout method: observations in the validation set are held out (that is, not used to train the model), to assess performance outside the training set. The distance between the two multivariate ECDFs, denoted as KS, is the Kolmogorov-Smirnov distance. In addition, the KS distance between the training and validation sets is also computed, and referred to as "Base KS". All the KS distances range from 0 (very good) to 1 (very bad). A synthetization is good when both the KS and Base KS distances are very similar.

The `C2_full` dataset only has 110 observations, including duplicates and outliers. This makes it hard to synthesize, especially since only 50% of these observations are used for training. The Base KS distance between the training and validation sets is rather large: 0.1455. It means that the validation set is quite different from the training set. But the KS distance between the synthesized and validation sets is barely any larger: 0.1727. Thus, NoGAN did a pretty good job. The KS distance between the synthesized data and the training set is a lot smaller. To balance the dataset (increasing the proportion of observations with label equal to 0), create a large enough sample and discard generated observations with non-zero label. This completes the main part of Step 4.

To answer Step 5, based on earlier computations, the proportions p_1, p_2, p_3 are respectively

$$p_1 = \frac{110{,}467}{117{,}735}, \quad p_2 = \frac{7158}{117{,}735}, \quad p_3 = \frac{110}{117{,}735}.$$

Since the set A only has 23 distinct observations repeated time and over (totaling 110,467 when not deduped), to synthesize `A` we can again use a multinomial distribution with the correct probabilities, this time to generate 23 random counts adding to n_1. Each count tells you how many times the corresponding unique observation must be repeated. The probability attached to a unique observation is its frequency measured in the training data. The synthetization of `C1` is left as an exercise.

2.2.3 Python code with SQL queries

The code, also available on GitHub here, splits the original dataset into `A`, `C1`, and `C2`. It also produces `map_C1` and `map_C2`. I did not include the adapted NoGAN code, but you can find it on GitHub, here. It also contains quite a bit of SQL to compute the KS distance. I want to thank Willie Waters for bringing this dataset to my attention.

```
import numpy as np
import pandas as pd

url = "https://raw.githubusercontent.com/VincentGranville/Main/main/iot_security.csv"
data = pd.read_csv(url)
# data = pd.read_csv('iot.csv')
features = list(data.columns)
print(features)
data_uniques = data.groupby(data.columns.tolist(), as_index=False).size()
data_B = data_uniques[data_uniques['size'] <= 3] #
data_A = data_uniques[data_uniques['size'] > 3]
data_A.to_csv('iot_A.csv')
```

```
print(data_A)

data_C = data_B.drop(['src_port','size'], axis=1)
data_C = data_C.groupby(data_C.columns.tolist(), as_index=False).size()
data_C1 = data_C[(data_C['bidirectional_mean_ps'] == 60) |
             (data_C['bidirectional_mean_ps'] == 1078) |
             (data_C['size'] > 1)]
data_C2 = data_C[(data_C['bidirectional_mean_ps'] != 60) &
             (data_C['bidirectional_mean_ps'] != 1078) &
             (data_C['size'] == 1)]
print(data_C)
data_C1.to_csv('iot_C1.csv')
data_C2.to_csv('iot_C2.csv')

data_B_full = data_B.join(data.set_index(features), on=features, how='inner')
features.remove('src_port')
data_C1_full = data_C1.merge(data_B_full, how='left', on=features)
data_C2_full = data_C2.merge(data_B_full, how='left', on=features)
data_C1_full.to_csv('iot_C1_full.csv')
data_C2_full.to_csv('iot_C2_full.csv')

map_C1 = data_C1_full.groupby('src_port')['src_port'].count()
map_C2 = data_C2_full.groupby('src_port')['src_port'].count()
map_C1.to_csv('iot_C1_map.csv')
map_C2.to_csv('iot_C2_map.csv')

data_C1 = data_C1_full.drop(['src_port','size_x', 'size_y'], axis=1)
data_C1 = data_C1.groupby(data_C1.columns.tolist(), as_index=False).size()
data_C2 = data_C2_full.drop(['src_port','size_x', 'size_y'], axis=1)
data_C2 = data_C2.groupby(data_C2.columns.tolist(), as_index=False).size()
data_C1.to_csv('iot_C1.csv')
data_C2.to_csv('iot_C2.csv')
```

2.3 Good GenAI evaluation, fast LLM search, and real randomness

In this section, I cover several topics in detail. First, I introduce one of the best random number generators (PRNG) with infinite period. Then, I show how to evaluate the synthesized numbers using the best metrics. Finally, I illustrate how it applies to other contexts, such as large language models (LLM). In particular, the system is based on words with letters from an arbitrary alphabet, and it can be adapted to any prespecified multivariate distribution, not just uniform: the joint ECDF (empirical distribution) attached to a training set in GenAI systems, for instance. At each step, the focus is both on quality and speed, revisiting old methods or inventing new ones, to get solutions performing significantly better and requiring much less computing time. The three components of this system are:

New powerful random number system

In its simplest form, the random numbers are defined as the binary digits $d_n = x_n \bmod 2$, from the sequence $x_{n+1} = 3 \cdot (x_n // 2)$, where the double slash is the integer division [Wiki]. It is an improvement over binary digits of quadratic irrationals used previously (see section 4.4 in [15]) in the sense that x_n grows only by a factor 3/2 at each iteration, rather than 2. All sequences (x_n) that do not grow indefinitely necessarily result in periodic numbers. This is the case for all PRNGs on the market.

In addition, despite having very long periods, these random generators with finite periods exhibit subtle patterns in rather low dimensions: in short, lack of randomness. They can be quite sensitive to the seed and may require many warm-up iterations before reaching higher randomness. See here how you can crack the Mersenne twister used in the Numpy random function. The question is this: how slowly can x_n grow while preserving perfect randomness, fast implementation, and an infinite period? Read on to see how I managed to reduce the aforementioned exponential growth down to linear, while keeping an infinite period. Proving that the period is infinite is still an open question in number theory, and beyond the scope of this project.

Ultrafast, robust evaluation metrics

The first step is to define what a strongly random sequence is, when it consists of deterministic digits. Details are again in chapter 4 in [15]. The takeaway: you need a metric that captures just that, when testing your system. This is true for all GenAI systems. Indeed, here I am re-using the full multivariate Kolmogorov-Smirnov

distance (KS) specifically implemented in the context of synthetic data generation: see section 6.4.2 in [10] for details. There, I showed how poorly implemented metrics used by vendors fail to capture subtle departures from the target distribution.

In this section, I present a very fast implementation of KS. I also include a few other tests. Very large test batteries exist, for instance Diehard [Wiki]. However, most rely on old statistical practice, offering a large number of disparate, weak tests, rather than a centralized approach to dealing with the problem. You can do a lot better with much fewer tests. This is one of the goals of this project, also with a focus on hard-to-detect patterns.

Also note that the KS distance relies on the CDF rather than the PDF (probability density function). The latter, used in many tests such as Chi-squared, does not work when you have billions of cross-feature buckets in high dimensions, each with very few observations. As in many GenAI systems, this is what we face. To give you an idea, think about counting occurrences of billions of "words" such as

$$3210232010310223034123103323103003110231 02$$

in a sequence of trillions of digits in base 4 (in this case, the alphabet has 4 letters). Most counts will be zero. Likewise, the base (that is, the size of the alphabet) may be a very large integer. The KS distance handles this problem transparently by looking at closest strings found in the digit sequences, themselves having only one occurrence most of the time. Also, it easily takes care of conditional probabilities when needed.

My previous KS implementation involved thousands of Pandas SQL queries spanning across many features. The new version discussed here is based on the radix numeration system [Wiki], turning long strings in big integers (called blocks), allowing for fast retrieval with simple binary search in a list of big numbers. In this context, a block can have many digits: the k-th feature is the k-th digit, although blocks may have a varying number of digits. I implicitly rely on the Python Bignum library [Wiki] to deal with the computations. Finally, the binary search is further improved and called weighted binary search, accelerating the computations by a factor 3 or 4 in the examples tested. So far, I did not compare with other methods such as vector search based on KNN and ANN (approximate nearest neighbors). But these methods are very relevant in this context.

Connection to LLM

The above paragraphs establish the connection to large language models. The problem is strikingly similar to DNA sequence synthetization discussed in section 7.1, where the alphabet has four letters (A, C, G, T) and the words consist of DNA subsequences. The main difference is that DNA sequences are far from random. Yet, the methodology presented here can easily be adapted to arbitrary target distributions. In particular to empirical distributions like those associated to DNA sequencing, or keyword distributions in ordinary text.

Then, as illustrated in the DNA sequencing problem, predictive analytics for GenAI may rely on conditional probabilities such as $P(B_1|B_2)$, where B_1, B_2 are consecutive blocks. Transitioning from KS and the multivariate CDF to conditional probabilities is straightforward with the formula $P(B_1|B_2) = P(B_1, B2)/P(B_2)$.

2.3.1 Project description

We are dealing with sequences of positive integers defined by a recursion $x_{n+1} = f(x_n) \bmod \tau_n$, where x_0 is the initial condition, referred to as the main seed. The choice of f leads to x_n randomly going up and down as n increases. On average, the trend is up: eventually, x_n gets bigger and bigger on average, albeit rather slowly to allow for fast computations. The **digits** – what the generated random numbers are – are simply defined as $d_n = x_n \bmod b$, where b is called the **base**.

The big contrast with classic random generators is that τ_n depends on n rather than being fixed. Here τ_n strictly increases at each iteration, allowing x_n to grow on average. It leads to more random digits, and an infinite period. In the current implementation, $\tau_{n+1} = s + \tau_n$ where s is a moderately large integer. Thus, we have two additional seeds: the step s, and τ_0. The function f depends on two integer parameters p, q. More specifically,

$$x_{n+1} = p \cdot (x_n//q) \bmod \tau_n, \text{ with } p > q > 1,\ x_0, \tau_0 \geq 0, \text{ and } \tau_{n+1} = s + \tau_n. \tag{2.1}$$

Again, the double slash stands for the integer division: $x_n//q = \lfloor x_n/q \rfloor$ where the brackets denote the floor function, or `int` in Python. I use the integer division in Python as it works when x_n becomes extremely large, unlike the standard division combined with `int`.

I implemented three tests of randomness. Before going into the details, let me introduce the concept of **block**. A block consists of a fixed number of digits, used for testing purposes. They are encoded as big integers. For now, let's pretend that a block B has m digits, each with a variable base: the k-th digit is in base b_k, thus with $0 \leq d_k < b_k$. Then B can be uniquely encoded with the one-to-one mapping

$$B = d_1 + d_2 \times b_1 + d_3 \times b_1 b_2 + d_4 \times b_1 b_2 b_3 + \cdots + d_m \times b_1 b_2 \cdots b_{m-1} \tag{2.2}$$

In this case, B is represented by an integer strictly smaller than $b_1 b_2 \cdots b_m$. In the context of real data, each digit represents a feature; the k-th feature can take on b_k different values after approximation or truncation, and a block corresponds to a row in a tabular data set. Of course, the b_k' can be quite large and different, depending on the feature, especially for numerical features. At the other extreme, for a 2-category feature, $b_k = 2$. The numeration system based on (2.2) is called the radix system. It is a generalization of the numeration system in fixed base b. When a new observation is generated, it is first encoded as a block, then the closest match to any row in the training set can be found with a binary search on all long integers B between 0 and $(b_1 \cdots b_m) - 1$, each one representing a potential row in the training set. Note that in general, the existing B's cover a small subset of all potential values: we are dealing with sparsity. We will use the same search algorithm here, with a fixed base b. It makes sense to call it radix search.

Now I can provide a quick overview of the three tests of randomness. The most involved is the block test: it is based on the multivariate KS distance, which in turn relies on radix search. The tests are:

- Run test. The max run test is implemented in section 4.3 for binary digits of quadratic irrationals. The solution presented here is more comprehensive, looking at runs or arbitrary length for all possible digits. These run lengths have a geometric distribution if digits show up randomly. When the base b is very large, there are too many values to fit in a table, so I also compute the following statistics:

$$R(L) = \sum_{d=0}^{b-1} \left(\frac{\rho(L,d) - \rho_0(L,d)}{\sigma(L,d)} \right)^2, \quad L = 1, 2, \ldots \tag{2.3}$$

  where $\rho_0(L,d)$ is the number of runs of length L found in the sequence for digit d, and $\rho(L,d)$ is the expected count if the sequence is random; then, it does not depend on d. Finally, $\sigma^2(L,d)$ is the theoretical variance, not depending on d, to normalize $R(L)$ so that it has a Chi-squared distribution with b degrees of freedom. In short, an unexpectedly large $R(L)$ indicates lack of randomness.

- Spectral test. This test looks at autocorrelations of lag 1, 2, and so on in the digit sequence, to check out whether their behavior is compatible or not with randomness. Because the sequences can be extremely long, the computations are done on the flight, updated one new digit at a time using a buffer, without having to store the whole sequence anywhere.

- Block test. Create a sample of blocks B of size m, for instance equally spaced, and independently of the blocks found in the digit sequence. Here m is the number of digits per block, with each digit in base b. Hence, each block should occur in the infinite digit sequence with frequency b^{-m}, assuming randomness. Compute the CDF value $F(B)$ attached to the underlying theoretical (uniform) distribution, for each of these blocks. Then compute the ECDF or empirical CDF value $F_0(B)$ based on block occurrences: mostly nearest neighbors to B, found in the digit sequence. Finally, KS $= \sup |F(B) - F_0(B)|$, and the supremum is over all blocks B in your sample. To magnify any subtle departure from randomness, plot the following: $\delta(B) = F(B) - F_0(B)$ on the Y-axis, and B (in its integer representation) on the X-axis, for all B in your sample.

There is a considerable amount of new material to master, if you were to implement the whole system on your own. The goal is not to create your own code from scratch, but rather to understand and use mine. You are welcome to improve it and create Python objects for the various components. One of the easy tasks is to use the code to compare different random generators and assess the impact of parameters. Another goal is to generalize the code to other contexts such as synthesizing DNA sequences, where the target distribution is not uniform, but comes as an ECDF (empirical CDF) computed on the training set. This also involves conditional probabilities: predicting the next block given the previous ones, when blocks are auto-correlated.

The project consists of the following steps:

Step 1: Read and understand the code in section 2.3.3. Identify the different components: the three tests, the base, the digits, the generation of the digits, the blocks and block encoding, the seeds and parameters, and the binary search to locate nearby blocks in the sequence, given a specific block. Run the code (it is on GitHub), see and analyze the results. Understand the computations for the autocorrelation coefficients, as it is done on the fly, in a non-traditional way using a buffer.

Step 2: What are the values of $\rho(L,d)$ and $\sigma(L,d)$ in Formula 2.3? How would you optimize a binary search so that it requires fewer steps? To help answer this question, look at the variable `trials`, which counts the combined number of steps used in all the binary searches. Finally, save the generated digits in a file if the sequence is not too long.

Step 3: To improve block tests, work with blocks of various sizes. Also, compute the autocorrelations in the block sequence, in the same way it is done in the digit sequence. Given a small-size block B consisting

of two sub-blocks B_1, B_2, estimate $P(B_2|B_1)$ on the data, and confirm the independence between B_1 and B_2, that is, $P(B_2|B_1) = P(B_2)$.

- **Step 4**: Try different seeds x_0, τ_0 and s to check out how the random generators is sensitive to the seeds. Does it change the quality of the randomness? In particular, try to find the lowest possible seed values that still lead to good random numbers. Then, try different combinations of p, q and b. Some work, some don't. Can you identify necessary conditions on p, q, b to guarantee good random numbers? Finally, can you find a good combination that makes the sequence (x_n) grow as slowly as possible, while still generating great random numbers. Obviously, you need $p > q$, for instance $p = q + 1$, and s as small as possible. A large base b also helps to extract as much as possible from the sequence (x_n), yet some combinations (p, q, b) don't work, and usually $b > q$ causes problems, making $b = q$ ideal.

2.3.2 Solution

Since much of the solution is my code in section 2.3.3, I cover only specific items in this section, with a focus on explaining and interpreting the output tables and visualizations. The main parameters are initialized in section [3] in the code, see lines 278 – 288. In particular, the current values of x_n, τ_n are stored in `x` and `tau` respectively, while s in Formula (2.1) is denoted as `step` in the code.

Figure 2.3 shows the function $\delta(B) = F(B) - F_0(B)$, that is, the difference between a perfectly uniform CDF and the approximate ECDF computed on the generated digits, using 500 equally-spaced test blocks, each with 6 digits. The number of blocks (500 here) is specified by `n_nodes` in the code. The higher the number, the better the approximation. The total number of digits if 640,000. I tried four sets of parameters labeled HM_1 to HM_4. The parameter sets are specific combinations of p, q, b. See lines 294, 316, 337 and 358 for the respective values in the code. I also added Numpy.random for comparison purposes. Note that HM stands for "Home-Made", by contrast to Numpy.

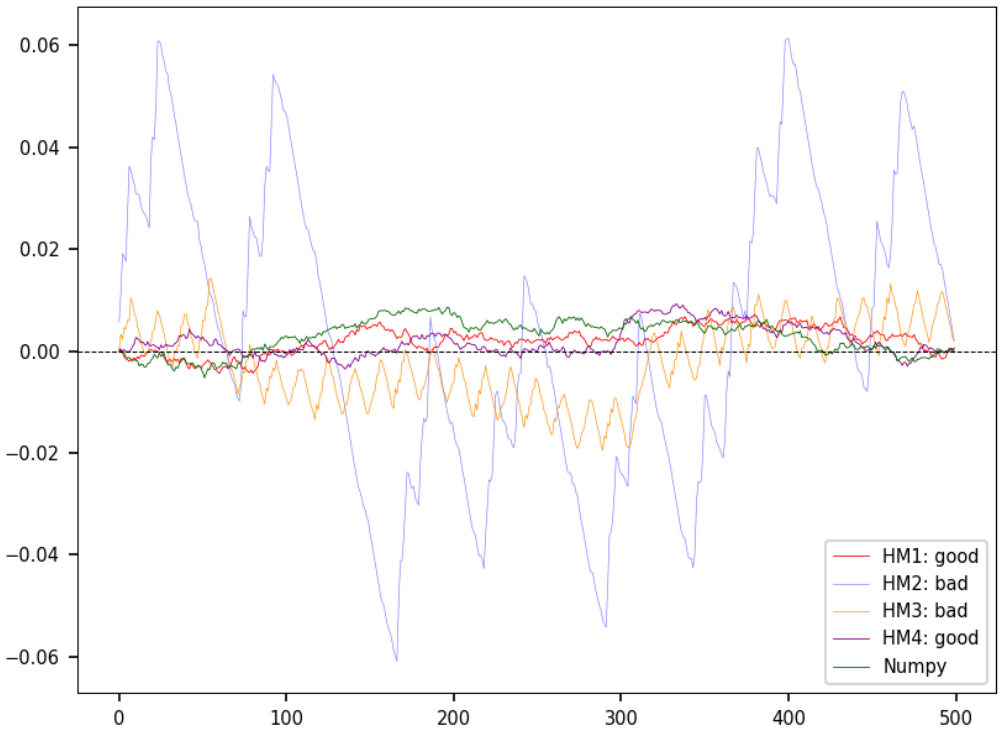


Figure 2.3: Four versions of my generator (HMx) compared to numpy.random

Clearly, HM_2 and HM_3 generate non-random numbers. As for HM_1, HM_4 and Numpy, they pass this test. It does not mean that they generate random digits. More tests are needed for verification, in particular with larger block sizes (the parameter `block_size`) and more nodes (the parameter `n_nodes`).

Table 2.1 focuses on HM_1 only. It shows the number of occurrences for runs of length L, for each of the 4 digits ($b = 4$) and various values of L. The sequence has 640,000 digits. The 4 rightmost columns are summary statistics: `Exp` and `Avg` are respectively the expected and observed counts, while `Norm` indicates whether or not all the counts, for a specific L, are compatible with perfect randomness. If the digits are truly random,

then `Norm` approximately follows a standard normal distribution. In particular, $\texttt{Norm} = (R(L) - b)/(2b)$ where $R(L)$ is defined by (2.3). Clearly, HM_1 passes this test.

L	$d=0$	$d=1$	$d=2$	$d=3$	Exp	Avg	Chi2	Norm
1	8968	8932	9074	8999	9000	8993	1.44	−0.3202
2	2296	2227	2294	2149	2250	2242	6.81	0.3511
3	548	532	620	587	563	572	9.05	0.6315
4	146	132	143	131	141	138	1.44	−0.3204
5	36	39	38	36	35	37	0.69	−0.4136
6	3	11	8	10	9	8	4.61	0.0759
7	2	3	3	2	2	3	0.62	−0.4223
8	0	2	0	0	1	1	3.83	−0.0211

Table 2.1: Runs of length L per digit for HM_1, with summary stats

Table 2.2 shows summary statistics for the 4 home-made generators (HM) and Numpy. One of them (HM_3) has $b = 256$. The normal approximation to `Norm` is especially good when $b > 10$, and bad when the counts are small, that is, when L is large. All the values not compatible with the randomness assumption are highlighted in red. Thus, HM_2 and HM_3 do not produce random digits, confirming the findings from Figure 2.3. Here, AC stands for autocorrelations within digit sequences

Metric	Level	HM_1	HM_2	HM_3	HM_4	Numpy
Norm	$L=1$	−0.3202	1530.7	136.38	0.0885	−0.0471
	$L=2$	0.3511	249.3	95.80	−0.1944	0.3141
	$L=3$	0.6315	80.1	41.37	0.5242	0.8902
AC	Lag 1	0.0010	−0.0046	−0.0731	−0.0002	−0.0007
	Lag 2	−0.0011	0.0022	0.0048	0.0022	−0.0014
KS		0.00676	0.06136	0.01932	0.00929	0.00859

Table 2.2: High-level comparison of HM and Numpy generators, with red flags

Besides listing the parameters used in the simulation, Table 2.3 features two interesting metrics: last x_n, and `Trials`. The former indicates how fast the sequence x_n grows. Numpy is not based on growing sequences, resulting in digits that keep repeating themselves past the finite period. And for random binary digits based on quadratic irrationals, the last x_n would be of the order $2^N \approx 10^{193,000}$, where $N = 640{,}000$ is the number of digits in the sequence. By contrast, for the HM generators explored here, it is around 10^{10}.

Feature	HM_1	HM_2	HM_3	HM_4	Numpy
Last x_n	10^{10}	10^9	10^{10}	10^9	n/a
Trials	178	2197	3497	2293	150
Base	4	4	8	256	4
Block size	6	6	6	6	6
p	7	6	7	401	n/a
q	4	4	4	256	n/a

Table 2.3: Top parameters and special metrics

Finally, `Trials` represents the total number of steps needed in the binary search, combined over the 500 test blocks (500 = `n_nodes`). For each test block, the goal is to find the closest neighbor block in the digit sequence (640,000 digits). The `Trials` value depends on the base, the block size, the number of digits in the sequence, and the number of test blocks. A very small value means that most of the test blocks are already present in the digit sequence: for these test blocks, the binary search is not needed. In the table, a value of 2197 means that on average, each test block requires $2197/500 = 4.39$ trials before finding the closest neighbor

block. The lower `Trials`, the faster the computations. Because I use an optimized binary search, it is already 3–4 times faster than the standard method. The generation of random digits is also just as fast as Numpy.

Regarding the project steps, I now provide answers to selected questions. Let N be the number of digits in base b in the sequence, and π be the probability for a run to be of length $L > 0$, for any digit $d < b$. We are dealing with a binomial distribution of parameter (N, π). Thus,

$$\rho(L,d) = N\pi, \quad \sigma^2(L,d) = N\pi(1-\pi), \quad \text{with } \pi = \left(\frac{b-1}{b}\right)^2 \cdot \left(\frac{1}{b}\right)^L,$$

Note that are $\rho(L,d)$ and $\sigma^2(L,d)$ are respectively the expectation and variance of the binomial distribution. This answers the first question in Step 2. For the second question about the binary search, see lines 184 – 197 in the code. If you set `A=1` and `B=1` respectively in lines 187 and 188, it becomes a standard binary search, with computational complexity $O(\log_2 n)$ in all cases. With my choice of `A` and `B`, it is similar to interpolation search [Wiki], which is $O(\log_2 \log_2 n)$ for the average case when the underlying distribution is uniform. See also the recent article on interpolated binary search [29].

2.3.3 Python implementation

The code is also on GitHub, here.

```
import numpy as np
from collections import OrderedDict

#--- [1] Functions to generate random digits

def get_next_digit(x, p, q, tau, step, base, option = "Home-Made"):

    if option == "Numpy":
       digit = np.random.randint(0, base)
    elif option == "Home-Made":
       x = ((p * x) // q) % tau # integer division for big int
       tau += step
       digit = x % base
    return(digit, x, tau)


def update_runs(digit, old_digit, run, max_run, hash_runs):

    if digit == old_digit:
       run += 1
    else:
       if (old_digit, run) in hash_runs:
          hash_runs[(old_digit, run)] += 1
       else:
          hash_runs[(old_digit, run)] = 1
       if run > max_run:
          max_run = run
       run = 1
    return(run, max_run, hash_runs)


def update_blocks(digit, m, block, base, block_size, hash_blocks):

    if m < block_size:
       block = base * block + digit
       m += 1
    else:
       if block in hash_blocks:
          hash_blocks[block] += 1
       else:
          hash_blocks[block] = 1
       block = 0
       m = 0
    return(m, block, hash_blocks)

```

```
46
47 def update_cp(digit, k, buffer, max_lag, N, cp_data):
48
49     # processing k-th digit starting at k=2
50     # buffer stores the last max_lag digits
51     # cp stands for the cross-product part (in autocorrelation)
52
53     mu = cp_data[0]
54     cnt = cp_data[1]
55     cp_vals = cp_data[2]
56     cp_cnt = cp_data[3]
57
58     buffer[(k-2) % max_lag] = digit
59     mu += digit
60     cnt += 1
61
62     for lag in range(max_lag):
63        if k-2 >= lag:
64           cp_vals[lag] += digit * buffer[(k-2-lag) % max_lag]
65           cp_cnt[lag] += 1
66
67     cp_data = (mu, cnt, cp_vals, cp_cnt)
68     return(cp_data, buffer)
69
70
71 def generate_digits(N, x0, p, q, tau, step, base, block_size, max_lag, option =
       "Home-Made"):
72
73     # Main function. Also produces output to test randomnes:
74     #    - hash_runs and max_run: input for the run_test function
75     #    - hash_blocks: input for the block_test function
76     #    - cp_data input for the correl_test function
77
78     # for run and block test
79     hash_runs = {}
80     hash_blocks = {}
81     block = 0
82     m = 0
83     run = 0
84     max_run = 0
85     digit = -1
86
87     # for correl_test
88     mu = 0
89     cnt = 0
90     buffer = np.zeros(max_lag)
91     cp_vals = np.zeros(max_lag) # cross-products for autocorrel
92     cp_cnt = np.zeros(max_lag)
93     cp_data = (mu, cnt, cp_vals, cp_cnt)
94
95     x = x0
96
97     for k in range(2, N):
98
99        old_digit = digit
100       (digit, x, tau) = get_next_digit(x, p, q, tau, step, base, option)
101       (run, max_run, hash_runs) = update_runs(digit, old_digit, run, max_run, hash_runs)
102       (m, block, hash_blocks) = update_blocks(digit, m, block, base, block_size,
              hash_blocks)
103       (cp_data, buffer) = update_cp(digit, k, buffer, max_lag, N, cp_data)
104
105    print("----------------")
106    print("PRNG = ", option)
107    print("block_size (digits per block), digit base: %d, %d", block_size, base)
108    if option == "Home-Made":
109       print("p, q: %d, %d" %(p, q))
```

```
      print(len(str(x)), "decimal digits in last x")
   return(hash_runs, hash_blocks, max_run, cp_data)


#--- [2] Functions to perform tests of randomness

def run_test(base, max_run, hash_runs):

   # For each run, chi2 has approx. chi2 distrib. with base degrees of freedom
   # This is true assuming the digits are random

   print()
   print("Digit ", end = " ")
   if base <= 8:
      for digit in range(base):
         print("%8d" %(digit), end =" ")
   print("  Exp", end = " ")
   print("  Avg", end = " ") # count average over all digits
   print(" Chi2", end = " ") # degrees of freedom = base
   print(" norm", end = " ")
   print("\n")

   for run in range(1, max_run+1):

      print("Run %3d" % (run), end = " ")
      prob = ((base-1)/base)**2 * (1/base)**run
      exp = N*prob
      var = N*prob*(1-prob)
      avg = 0
      chi2 = 0

      for digit in range(0, base):
         key = (digit, run)
         count = 0
         if key in hash_runs:
            count = hash_runs[key]
            avg += count
            chi2 += (count - exp)**2 / var
         if base <= 8:
            print("%8d" %(count), end =" ")

      avg /= base
      norm = (chi2 - base) / (2*base)
      print("%8d" %(int(0.5 + exp)), end =" ")
      print("%8d" %(int(0.5 + avg)), end =" ")
      print("%8.2f" %chi2, end =" ")
      print("%8.4f" %norm, end =" ")
      print()

   return()


def get_closest_blocks(block, list, trials):

   # Used in block_test
   # Return (left_block, right_block) with left_block <= block <= right_block,
   #  - If block is in list, left_block = block = right_block
   #  - Otherwise, left_block = list(left), right_block = list(right);
   #            they are the closest neighbors to block, found in list
   #  - left_block, right_block are found in list with weighted binary search
   #  - list must be ordered
   # trials: to compare spped of binary search with weighted binary search

   found = False
   left = 0
   right = len(list) - 1
```

```
    delta = 1
    old_delta = 0

    if block in list:
       left_block = block
       right_block = block

    else:
       while delta != old_delta:
          trials += 1
          old_delta = delta
          A = max(list[right] - block, 0) # in standard binary search: A = 1
          B = max(block - list[left], 0) # in standard binary search: B = 1
          middle = (A*left + B*right) // (A + B)
          if list[middle] > block:
             right = middle
          elif list[middle] < block:
             left = middle
          delta = right - left

       left_block = list[middle]
       right_block = list[min(middle+1, len(list)-1)]

    return(left_block, right_block, trials)


def true_cdf(block, max_block):
    # Used in block_test
    # blocks uniformly distributed on {0, 1, ..., max_block}
    return((block + 1)/(max_block + 1))


def block_test(hash_blocks, n_nodes, base, block_size):

    # Approximated KS (Kolmogorov-Smirnov distance) between true random and PRNG
    # Computed only for blocks with block_size digits (each digit in base system)
    # More nodes means better approximation to KS

    hash_cdf = {}
    hash_blocks = OrderedDict(sorted(hash_blocks.items()))
    n_blocks = sum(hash_blocks.values())
    count = 0
    trials = 0 # total number of iterations in binary search


    for block in hash_blocks:
       hash_cdf[block] = count + hash_blocks[block]/n_blocks
       count = hash_cdf[block]

    cdf_list = list(hash_cdf.keys())
    max_block = base**block_size - 1
    KS = 0
    trials = 0 # total number of iterations in binary search
    arr_cdf = [] # theoretical cdf, values
    arr_ecdf = [] # empirical cdf (PRMG), values
    arr_arg = [] # arguments (block number) associated to cdf or ecdf

    for k in range(0, n_nodes):

       block = int(0.5 + k * max_block / n_nodes)
       (left_block, right_block, trials) = get_closest_blocks(block, cdf_list, trials)
       cdf_val = true_cdf(block, max_block)
       ecdf_lval = hash_cdf[left_block] # empirical cdf
       ecdf_rval = hash_cdf[right_block] # empirical cdf
       ecdf_val = (ecdf_lval + ecdf_rval) / 2 # empirical cdf
       arr_cdf.append(cdf_val)
```

```
242         arr_ecdf.append(ecdf_val)
243         arr_arg.append(block)
244         dist = abs(cdf_val - ecdf_val)
245         if dist > KS:
246            KS = dist
247 
248     return(KS, arr_cdf, arr_ecdf, arr_arg, trials)
249 
250 
251 def autocorrel_test(cp_data, max_lag, base):
252 
253     mu = cp_data[0]
254     cnt = cp_data[1]
255     cp_vals = cp_data[2]
256     cp_cnt = cp_data[3]
257 
258     mu /= cnt
259     t_mu = (base-1) / 2
260     var = cp_vals[0]/cp_cnt[0] - mu*mu
261     t_var = (base*base -1) / 12
262     print()
263     print("Digit mean: %6.2f (expected: %6.2f)" % (mu, t_mu))
264     print("Digit var : %6.2f (expected: %6.2f)" % (var, t_var))
265     print()
266     print("Digit autocorrelations: ")
267     for k in range(max_lag):
268         autocorrel = (cp_vals[k]/cp_cnt[k] - mu*mu) / var
269         print("Lag %4d: %7.4f" %(k, autocorrel))
270 
271     return()
272 
273 
274 #--- [3] Main section
275 
276 # I tested (p, q) in {(3, 2), (7, 4), (13, 8), (401, 256)}
277 
278 N = 64000         # number of digits to generate
279 p = 7             # p/q must > 1, preferably >= 1.5
280 q = 4             # I tried q = 2, 4, 8, 16 and so on only
281 base = q          # digit base, base <= q (base = 2 and base = q work)
282 x0 = 50001        # seed to start the bigint sequence in PRNG
283 tau = 41197       # co-seed of home-made PRNG
284 step = 37643      # co-seed of home-made PRNG
285 digit = -1        # fictitious digit before creating real ones
286 block_size = 6    # digits per block for the block test; must be integer > 0
287 n_nodes = 500     # number of nodes for the block test
288 max_lag = 3       # for autocorrel test
289 seed = 104        # needed only with option = "Numpy"
290 np.random.seed(seed)
291 
292 #- [3.1] Home-made PRNG with parameters that work
293 
294 p, q, base, block_size = 7, 4, 4, 6
295 
296 # generate random digits, home-made PRNG
297 (hash_runs, hash_blocks, max_run, cp_data) = generate_digits(N, x0, p, q, tau, step,
298                                                 base, block_size, max_lag,
299                                                 option="Home-Made")
300 # run_test
301 run_test(base,max_run,hash_runs)
302 
303 # block test
304 (KS, arr_cdf, arr_ecdf1, arr_arg1, trials) = block_test(hash_blocks, n_nodes,
305                                             base ,block_size)
306 # autocorrel_test
307 autocorrel_test(cp_data, max_lag, base)
```

```
308
309 print()
310 print("Trials = ", trials)
311 print("KS = %8.5f\n\n" %(KS))
312
313
314 #- [3.2] Home-made, with parameters that don't work
315
316 p, q, base, block_size = 6, 4, 4, 6
317
318 # generate random digits, home-made PRNG
319 (hash_runs, hash_blocks, max_run, cp_data) = generate_digits(N, x0, p, q, tau, step,
320                                                 base, block_size, max_lag,
321                                                 option="Home-Made")
322 # run_test
323 run_test(base,max_run,hash_runs)
324
325 # block test
326 (KS, arr_cdf, arr_ecdf2, arr_arg2, trials) = block_test(hash_blocks, n_nodes,
327                                             base ,block_size)
328 # autocorrel_test
329 autocorrel_test(cp_data, max_lag, base)
330
331 print()
332 print("Trials = ", trials)
333 print("KS = %8.5f\n\n" %(KS))
334
335 #- [3.3] Home-made, another example of failure
336
337 p, q, base, block_size = 7, 4, 8, 6
338
339 # generate random digits, home-made PRNG
340 (hash_runs, hash_blocks, max_run, cp_data) = generate_digits(N, x0, p, q, tau, step,
341                                                 base, block_size, max_lag,
342                                                 option="Home-Made")
343 # run_test
344 run_test(base,max_run,hash_runs)
345
346 # block test
347 (KS, arr_cdf, arr_ecdf3, arr_arg3, trials) = block_test(hash_blocks, n_nodes,
348                                             base ,block_size)
349 # autocorrel_test
350 autocorrel_test(cp_data, max_lag, base)
351
352 print()
353 print("Trials = ", trials)
354 print("KS = %8.5f\n\n" %(KS))
355
356 #- [3.4] Home-made, using a large base
357
358 p, q, base, block_size = 401, 256, 256, 6
359
360 # generate random digits, home-made PRNG
361 (hash_runs, hash_blocks, max_run, cp_data) = generate_digits(N, x0, p, q, tau, step,
362                                                 base, block_size, max_lag,
363                                                 option="Home-Made")
364 # run_test
365 run_test(base,max_run,hash_runs)
366
367 # block test
368 (KS, arr_cdf, arr_ecdf4, arr_arg4, trials) = block_test(hash_blocks, n_nodes,
369                                             base ,block_size)
370 # autocorrel_test
371 autocorrel_test(cp_data, max_lag, base)
372
373 print()
```

```
print("Trials = ", trials)
print("KS = %8.5f\n\n" %(KS))


#- [3.5] Numpy PRNG (Mersenne twister)

# here p, q are irrelevant
base, block_size = 4, 6

# generate random digits, home-made PRNG
(hash_runs, hash_blocks, max_run, cp_data) = generate_digits(N, x0, p, q, tau, step,
                                                base, block_size, max_lag,
                                                option="Numpy")
# run_test
run_test(base,max_run,hash_runs)

# block test
(KS, arr_cdf, arr_ecdf5, arr_arg5, trials) = block_test(hash_blocks, n_nodes,
                                            base ,block_size)
# autocorrel_test
autocorrel_test(cp_data, max_lag, base)

print()
print("Trials = ", trials)
print("KS = %8.5f\n\n" %(KS))


#--- [4] Scatterplot cdf (true random) versus ecdf (based on the two PRNGs)

import matplotlib.pyplot as plt
import matplotlib as mpl

mpl.rcParams['axes.linewidth'] = 0.5
plt.rcParams['xtick.labelsize'] = 7
plt.rcParams['ytick.labelsize'] = 7

arr_cdf = np.array(arr_cdf)
delta_ecdf1 = (np.array(arr_ecdf1) - arr_cdf)
delta_ecdf2 = (np.array(arr_ecdf2) - arr_cdf)
delta_ecdf3 = (np.array(arr_ecdf3) - arr_cdf)
delta_ecdf4 = (np.array(arr_ecdf4) - arr_cdf)
delta_ecdf5 = (np.array(arr_ecdf5) - arr_cdf)

# print()
# print("blocks (arguments) used to compute ecdf1:\n")
# print(arr_arg1)

plt.plot(delta_ecdf1, linewidth = 0.4, color = 'red', alpha = 1)
plt.plot(delta_ecdf2, linewidth = 0.3, color = 'blue', alpha = 0.5)
plt.plot(delta_ecdf3, linewidth = 0.3, color = 'darkorange', alpha = 1)
plt.plot(delta_ecdf4, linewidth = 0.4, color = 'purple', alpha = 1)
plt.plot(delta_ecdf5, linewidth = 0.4, color = 'darkgreen', alpha = 1)
plt.axhline(y = 0.0, linewidth = 0.5, color = 'black', linestyle = 'dashed')
plt.legend(['HM1: good', 'HM2: bad', 'HM3: bad', 'HM4: good', 'Numpy'],
          loc='lower right', prop={'size': 7}, )
plt.show()
```

2.4 The best GenAI loss function: your evaluation metric!

Most AI and GenAI algorithms aim at optimizing a criterion. The goal is to get good predictions (see project 8.3), 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. 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 groundbreaking 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. Then, I designed an experiment where I knew what the global optimum was. 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 if 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 [19]. This is a resampling method that preservers 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.

2.4.1 Project and solution

The new algorithm is called constrained NoGAN. It blends the best of probabilistic NoGan (based on deep resampling, see chapter 7 in [19] and Python code `DeepResampling_circle.py` here) with the efficient multivariate bin structure found in standard NoGAN (see project 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 how far 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 architecture leading to the spectacular results discussed earler. The code is in section 2.4.2 and also on GitHub, here.

Under construction...

2.4.2 Python code

Under construction...

The code is also on GitHub, here.

Chapter 3

Time Series and Spatial Processes

The first project covers non-periodic times series – more specifically time series with multiple periods – including modeling, simulation and goodness of fit via the autocorrelation structure. The case study is about ocean tides and distances between planets to detect alignments. I then move to random walks and Brownian motions, including integrated Brownian motions, the Hurst exponent to measure smoothness, and ad-hoc smoothing techniques. The last project involves 2D interpolation compared to kriging, applied to the Chicago temperature dataset.

3.1 Time series interpolation: ocean tides

The purpose is to reconstruct daily measurements in a non-periodic time series with monthly observations, and measure the quality of the results. In this project, the daily values can be computed exactly. Let's pretend that we don't know them when we generate them. In the end we will use the hidden exact values to see how good we were at reconstructing (interpolating) them.

The problem is similar to the illustration in Figure 3.1: you are dealing with a time series where the observations are the orange dots, in this case equally spaced by 80-min increments. You don't know what the smooth curves look like; you can't tell from the data. Here, I produced 16 modified copies of the observed data (the 80-min measurements). I used interpolation – one of several techniques to synthetize data – to produce them. Combined together, these 16 sets provide 5-min granularity resulting in the blue curve in Figure 3.1. In fact here, the 5-min data was actually available and represented by the red curve. I pretended it did no exist, but in the end I used it to assess the quality of the results.

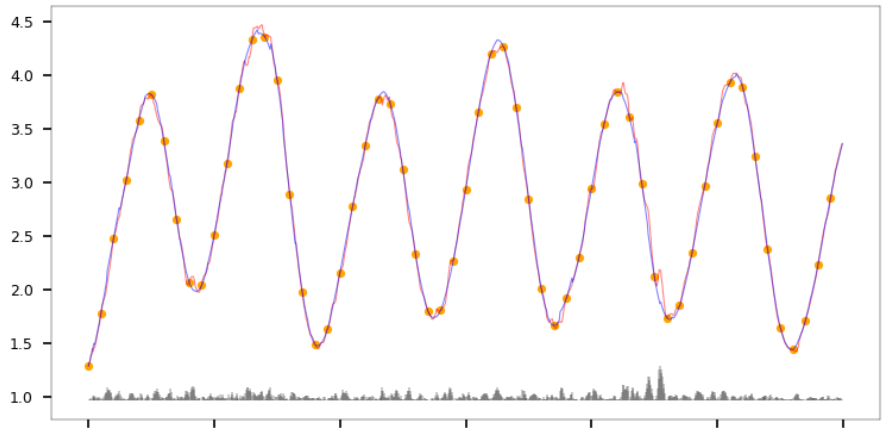


Figure 3.1: Tides at Dublin (5-min data), with 80 mins between interpolating nodes

The project in this section deals with similar time series. You need to get monthly distances between the planets and the Sun, to see how frequently Earth, Venus (or Jupiter) are aligned on the same side of the Sun. For instance, in case of almost perfect alignment, the apparent locations of Jupiter and Mars are identical to the naked eye in the night sky. Is there a chance you might see that event in your lifetime? You'll get an answer to this curious question, but most importantly, the goal is to get you familiar with one aspect of data reconstruction, sometimes called disaggregation. Rather than 80-min observations, we will use monthly or

quarterly observations. And we will reconstruct the more granular data via interpolation. Then, we assessing the quality of the interpolated data, and how more general modeling techniques could be used instead.

We first create a dataset with daily measurements of the distance between Earth and Venus, and interpolate the distances to test how little data is needed for good enough performance: Can you reconstruct daily data from monthly observations of the distances between planets? What about quarterly or yearly observations? Then, the purpose is to assess how a specific class of models is good at interpolating not only this type of data, but at the same time other types of datasets like the ocean tides in Figure 3.1 or the Riemann zeta function in Figure 3.2.

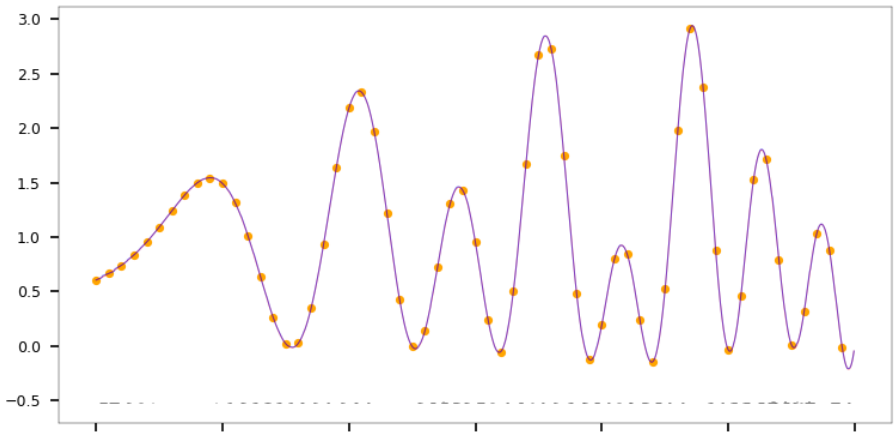


Figure 3.2: Interpolating the real part of $\zeta(\frac{1}{2}+it)$ based on orange points

The planetary fact sheet published by the NASA contains all the information needed to get started. It is available here. I picked up Venus and Earth because they are among the planets with the lowest eccentricities in the solar system. For simplicity, assume that the two orbits are circular. Also assume that at a time denoted as $t = 0$, the Sun, Venus and Earth were aligned and on the same side (with Venus between Earth and the Sun).

Note that all the major planets revolve around the sun in the same direction. Let $\theta_V, \theta_E, R_V, R_E$ be respectively the orbital periods of Venus and Earth, and the distances from Sun for Venus and Earth. From the NASA table, these quantities are respectively 224.7 days, 365.2 days, 108.2×10^6 km, and 149.6×10^6 km. Let $d_V(t)$ be the distance at time t, between Earth and Venus. You first need to convert the orbital periods into angular velocities $\omega_V = 2\pi/\theta_V$ and $\omega_E = 2\pi/\theta_E$ per day. Then elementary trigonometry leads to the formula

$$d_V^2(t) = R_E^2\left[1 + \left(\frac{R_V}{R_E}\right)^2 - 2\frac{R_V}{R_E}\cos\Big((\omega_V - \omega_E)t\Big)\right]. \tag{3.1}$$

The distance is thus periodic, and minimum and equal to $R_E - R_V$ when $(\omega_V - \omega_E)t$ is a multiple of 2π. This happens roughly every 584 days.

3.1.1 Project description

Before starting the project, read section 2.1 about ocean tides in my article "New Interpolation Methods for Synthetization and Prediction" available here, and the Python program `interpol_fourier.py`. The password to open the PDF document is `MLT12289058`. You can use a different interpolation technique, but one of the goals here is to learn how to use code written by a third party, and modify it for your own needs if necessary.

The project consists of the following steps:

Step 1: Use formula (3.1) to generate daily values of $d_V(t)$, for 10 consecutive years, starting at $t = 0$.

Step 2: Use the Python code `interpol_fourier.py` on my GitHub repository here. Interpolate daily data using one out of every 32 observations: using a power of 2 such as $32 = 2^5$ will let the code do everything nicely including producing the chart with the red dots, in the same way that I use one observation out of 16 for the ocean tides (80 minutes = 16×5 minutes). Conclude whether or not using one measurement every 32 days is good enough to reconstruct the daily observations. See how many nodes (the variable `n` in the code) you need to get a decent interpolation.

Step 3: Add planet Mars. The three planets (Venus, Earth, Mars) are aligned with the sun and on the same side when both $(\omega_V - \omega_E)t$ and $(\omega_M - \omega_E)t$ are almost exact multiples of 2π, that is, when both the distance $d_M(t)$ between Earth and Mars, and $d_V(t)$ between Earth and Venus, are minimum. In short, it happens when $g(t) = d_V(t) + d_M(t)$ is minimum. Assume it happened at $t = 0$. Plot the function $g(t)$, for a period of time long enough to see a global minimum (thus, corresponding to an alignment). Here ω_M is the orbital velocity of Mars, and its orbit is approximated by a circle.

Step 4: Repeat steps 1 and 2 but this time for $g(t)$. Unlike $d_V(t)$, the function $g(t)$ is not periodic. Alternatively, use Jupiter instead of Venus, as this leads to alignments visible to the naked eye in the night sky: the apparent locations of the two planets coincide.

Step 5: A possible general model for this type of time series is

$$f(t) = \sum_{k=1}^{m} A_k \sin(\omega_k t + \varphi_k) + \sum_{k=1}^{m} A'_k \cos(\omega'_k t + \varphi'_k) \tag{3.2}$$

where the $A_k, A'_k, \omega_k, \omega'_k, \varphi_k, \varphi'_k$ are the parameters, representing amplitudes, frequencies and phases. Show that this parameter configuration is redundant: you can simplify while keeping the full modeling capability, by setting $\varphi_k = \varphi'_k = 0$ and re-parameterize. Hint: use the angle sum formula (Google it).

Step 6: Try 10^6 parameter configurations of the simplified model based on formula (3.2) with $m = 2$ and $\varphi_k = \varphi'_k = 0$, to simulate time series via Monte-Carlo simulations. For each simulated time series, measure how close it is to the ocean tide data (obtained by setting `mode='Data'` in the Python code), the functions $g(t)$ or $d_V(t)$ in this exercise, or the Riemann zeta function pictured in Figure 3.2 (obtained by setting `mode='Math.Zeta'` in the Python code). Use a basic proximity metric of your choice to asses the quality of the fit, and use it on the transformed time series obtained after normalization (to get zero mean and unit variance). A possible comparison metric is a combination of lag-1, lag-2 and lag-3 autocorrelations applied to the 32-day data (planets) or 16-min data (ocean tides), comparing simulated (synthetic) versus observed data. Also, autocorrelations don't require normalizing the data as they are already scale- and location-invariant.

Step 7: Because of the curse of dimensionality [Wiki], Monte-Carlo is a very poor technique here as we are dealing with 8 parameters. On the other hand, you can get very good approximations with just 4 parameters, with a lower risk of overfitting. Read section 1.3.3 in my book "Synthetic Data and Generative AI" [20] about a better inference procedure, applied to ocean tides. Also read chapter 15 on synthetic universes featuring non-standard gravitation laws to generate different types of synthetic time series. Finally, read chapter 6 on shape generation and comparison: it features a different type of metric to measure the distance between two objects, in this case the time series (their shape: real versus synthetic version).

3.1.2 Note on time series comparison

Regarding the oscillations of the $g(t)$ function in Steps 3 and 4, a 10-year time period is not enough – by a long shot – to find when the next minimum (alignment) will occur. Looking at a 10-year time period is misleading.

One way to check if two time series (after normalization) are similar enough is typically done by comparing their estimated parameters (the ω_k, A_k and so on) to see how "close" they are. This is also how you would measure the similarity between a synthetic time series (or any kind of data for that matter), and the observed data that it is supposed to mimic. You don't compare the shapes: they might be quite different yet represent the same mechanical effect at play. To the contrary, two extracts from different time series may look very similar visually, but may come from very different models: it just happens by coincidence that they look quite similar on some short intervals.

In some sense, what you want to measure is the *stochastic* proximity. Another example is comparing the numbers π and $3/7$. If you search long enough, you will find two sequences of 5000 digits that are identical in both numbers. Yet these two numbers are very different in nature. Now if you compare $5/11$ and $3/7$, you will never find such identical sequences, not even 2-digit long, and you may conclude that the two numbers are very different, while in fact they are of the same kind. Same if you compare π with a rational number very close to π.

There is caveat with comparing datasets based on their estimated parameters: if the parameter set is redundant as in Step 5, you can have two perfectly identical datasets that have very different parameters attached to them. This is known as model identifiability in statistics.

Finally, I encourage you to upgrade my Python code `interpol_fourier.py` to include a sub-function that computes $g(t)$ defined in Step 3. And of course, you can test this program on your own data, not just the

ocean tides or planet-related data. You should try `'mode=Math.Zeta'` and see if you find anything special about the time series generated. There is something very special about it! Don't forget to install the MPmath Python library to make it work.

3.1.3 Solution

I split the solution into two parts. First the computation of daily distances $d_V(t), d_M(t), g(t)$ in million of km, and how they can be recovered via interpolation. Then, the simulations discussed in steps 5–7.

Let's start with Steps 1–4. The distances for $d_V(t), d_M(t), g(t)$ are respectively in blue, orange and green. They are represented by the vertical axis. The time is represented by the horizontal axis. In Figure 3.3, the time unit is a day (daily observations). In Figure 3.4, we are dealing with yearly observations instead. The blue curve shows a minimum about every 584 days, confirming that Venus is closest to Earth every 584 days. As for $g(t)$, there is no periodic minimum. Yet after 2400 days, you get another strong minimum, then minima get higher and higher and you have to wait over 400 years to reach a new low as low as the first one at $t = 0$, when perefct allignment occured by construction.

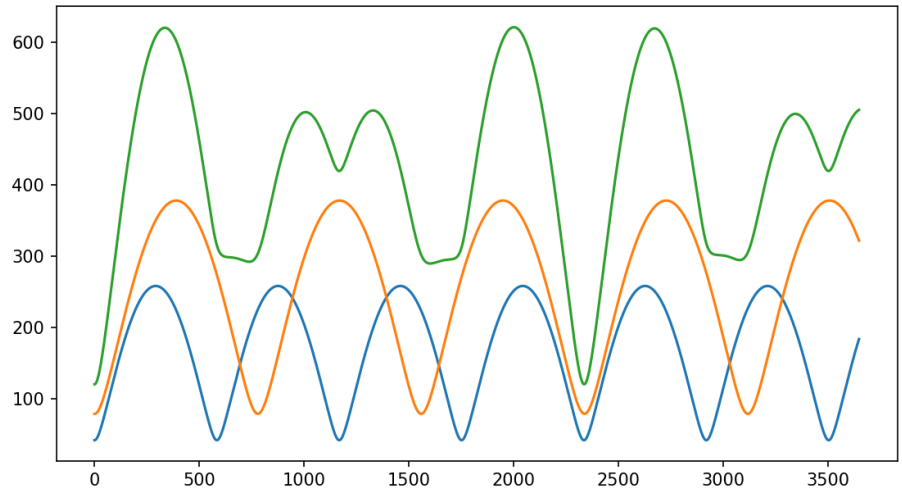


Figure 3.3: $d_V(t), d_M(t), g(t)$ in 10^6 km, first 10×365 days after alignment at $t = 0$

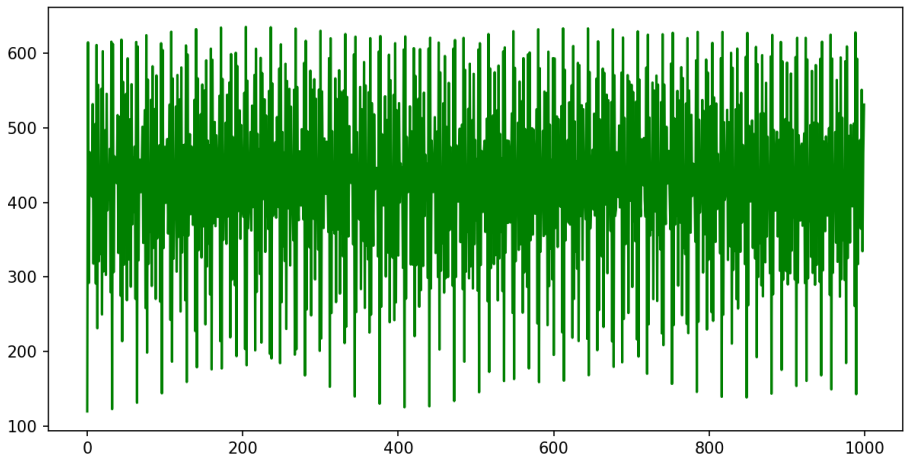


Figure 3.4: $g(t)$ over 1000 years, yearly measurements this time

Clearly, monthly data is well suited for interpolation, to recover daily data. But yearly data is not granular enough, and you can not expect to use (say) 50-year observations to recover or synthetize yearly observations. See below my code to compute the distances. It also produces an output file `gt_distances.txt` of daily observations for $g(t)$, used as input file for the interpolation program.

```
import numpy as np
import matplotlib.pyplot as plt
```

```
R_V = 108.2 # million km
R_E = 149.6 # million km
R_M = 228.0 # million km
ratio_V = R_V / R_E
ratio_M = R_M / R_E

pi2 = 2*np.pi
t_unit = 1 # (in number of days)

# time unit = 32 days, to interpolate daily values (1 obs ever 32 day)
omega_V = t_unit * pi2 / 224.7 # angular velocity per 32 days
omega_E = t_unit * pi2 / 365.2 # angular velocity per 32 days
omega_M = t_unit * pi2 / 687.0 # angular velocity per 32 days

time = []
d_V = []
d_M = []
d_sum = []
t_incr = 1 # (in number of days)
T = 365 * 10 # time period (in number of days)
OUT = open("gt_distances.txt","w")
for t in np.arange(0, T, t_incr):
   time.append(t)
   dist_V = R_E * np.sqrt(1 + ratio_V**2 - 2*ratio_V * np.cos((omega_V - omega_E)*t))
   dist_M = R_E * np.sqrt(1 + ratio_M**2 - 2*ratio_M * np.cos((omega_M - omega_E)*t))
   d_V.append(dist_V)
   d_M.append(dist_M)
   d_sum.append(dist_V + dist_M) # near absolute minimum every ~ 400 years
   OUT.write(str(dist_V + dist_M)+"\n")
OUT.close()
plt.plot(time,d_V)
plt.plot(time,d_M)
plt.plot(time,d_sum,c='green')
plt.show()
```

3.1.3.1 Interpolation

I use `interpol_fourier.py` to interpolate the distances for $g(t)$, using `mode='Data'` and `n=8` (the number of interpolation nodes) as for the ocean tide dataset. Here the imput file was `gt_distances.txt` created by the Python code in the previous section.

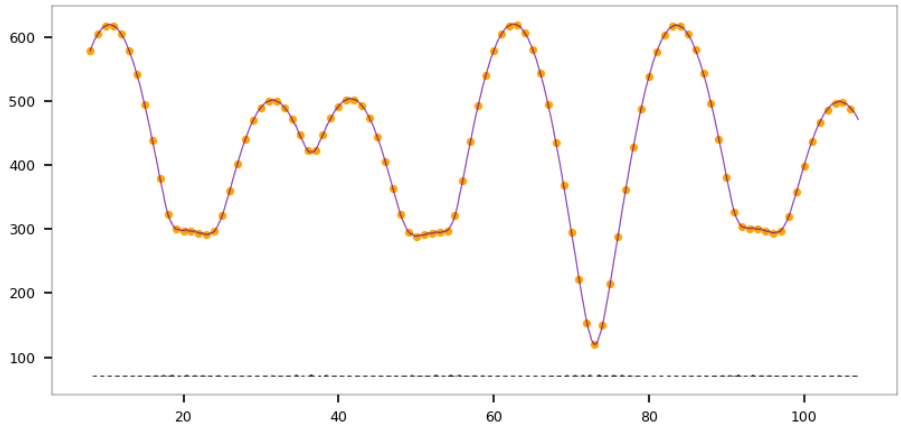


Figure 3.5: Daily interpolated values for $g(t)$, based on exact 32-day data in orange

Reducing the number of interpolation nodes from $n = 8$ to $n = 4$ starts showing a small error visible to the naked eye. With $n = 8$, you can't see the error as illustrated in Figure 3.5. I have no doubt that using one out every 64 days to reconstruct daily data (instead of one every 32) would still do a good job. In the process I created 32 synthetic copies of the orange data to fill the gaps: not identical copies but instead different copies with the right distribution compatible with the orange data. Also keep in mind that $g(t)$ does not have

any period, so any shifted version of it will be different. This is in contrast with the function $d_V(t)$. To run `interpol_fourier.py`, you need to change the input filename to `gt_distances.txt`, and set `t_unit=32`. The resulting plot in Figure 3.3 is truncated on the time (horizontal) axis, compared to Figure 3.5. Also the time unit on the horizontal axis is 32 days instead of one day as in Figure 3.3.

3.1.3.2 Simulations and comparison with real data

The answer to Step 5 is as follows. We have

$$A\cos(\omega t+\varphi)+A'\sin(wt+\varphi')=\alpha\sin\omega t+\beta\sin\omega' t+\alpha'\cos\omega t+\beta'\cos\omega' t,$$

with $\alpha=A\cos\varphi, \beta=-A'\sin\varphi', \alpha'=A\sin\varphi, \beta'=A'\cos\varphi'$. Thus the phase parameters φ, φ' are not necessary. However, removing them requires increasing m in Formula (3.2). Now, ignoring them, let's do the simulations with $m=2$. The Python code below deals with simulating $g(t)$ for the planet dataset. My conclusions follow after the code.

```
import numpy as np
import statsmodels.api as sm
import matplotlib as mpl
from matplotlib import pyplot as plt

# use statsmodel to compute autocorrel lag 1, 2, ..., nlags

#--- read data

IN = open("gt_distances.txt","r")
# IN = open("tides_Dublin.txt","r")
table = IN.readlines()
IN.close()

exact = []
t = 0
for string in table:
   string = string.replace('\n', '')
   fields = string.split('\t')
   value = float(fields[0])
   # value = np.cos(0.15*t) + np.sin(0.73*t)
   if t % 32 == 0: # 16 for ocean tides or Riemann zeta, 32 for planets (also try 64)
      exact.append(value)
   t = t + 1
nobs = len(exact)
time = np.arange(nobs)
exact = np.array(exact) # convert to numpy array

nlags = 8
acf_exact = sm.tsa.acf(exact, nlags=nlags)


#--- generate random params, one set per simulation

np.random.seed(104)
Nsimul = 10000

lim = 20
A1 = np.random.uniform(-lim, lim, Nsimul)
A2 = np.random.uniform(-lim, lim, Nsimul)
B1 = np.random.uniform(-lim, lim, Nsimul)
B2 = np.random.uniform(-lim, lim, Nsimul)
norm = np.sqrt(A1*A1 + A2*A2 + B1*B1 + B2*B2)
A1 = A1 / norm
A2 = A2 / norm
B1 = B1 / norm
B2 = B2 / norm
w1 = np.random.uniform(0, lim, Nsimul)
w2 = np.random.uniform(0, lim, Nsimul)
v1 = np.random.uniform(0, lim, Nsimul)
```

```
v2 = np.random.uniform(0, lim, Nsimul)

#--- generate Nsimul time series each with nobs according to model
#   measure fit between each realization and the real data
#   identify synthetized series with best fit

best_fit = 9999999.9
best_series_idx = 0
for i in range(Nsimul):
   if i % 5000 == 0:
      print("generating time series #",i)
   asimul = A1[i] * np.cos(w1[i]*time) + A2[i] * np.cos(w2[i]*time) + B1[i] *
       np.sin(v1[i]*time) + B2[i] * np.sin(v2[i]*time)
   acf = sm.tsa.acf(asimul, nlags=nlags)
   delta = acf - acf_exact
   metric1 = 0.5 * np.mean(np.abs(delta))
   corrm = np.corrcoef(exact,asimul)
   metric2 = 1 - abs(corrm[0,1])
   fit = metric1 # options: metric1 or metric2
   if fit < best_fit:
      best_fit = fit
      best_series_idx = i
      best_series = asimul
      acf_best = acf

print("best fit with series #",best_series_idx)
print("best fit:",best_fit)
print()

print("autocorrel lags, observed:")
print(acf_exact)
print()

print("autocorrel lags, best fit:")
print(acf_best)
print()

#--- plotting best fit

mu_exact = np.mean(exact)
stdev_exact = np.std(exact)
mu_best_series = np.mean(best_series)
stdev_best_series = np.std(best_series)
best_series = mu_exact + stdev_exact * (best_series - mu_best_series)/stdev_best_series
    # un-normalize

print("min: exact vs best series: %8.3f %8.3f" % (np.min(exact),np.min(best_series)))
print("max: exact vs best series: %8.3f %8.3f" % (np.max(exact),np.max(best_series)))
corrm = np.corrcoef(exact,best_series)
print("|correlation| between both %8.5f" % (abs(corrm[0,1])))

plt.scatter(exact, best_series, c='orange')
plt.show()
```

Monte-Carlo simulations are not a good solution with more than 3 or 4 parameters. Using 4 parameters sometimes lead to better results than the full model with 8 parameters in this case. Likewise using fewer simulations – say 10^4 instead of 10^6 – can lead to better results, especially in a case like this with multiple local minima. Also, using lag-1, lag-2, and lag-3 autorrelations is not enough to measure the "distance" (called `fit` in the Python code), between the real and simulated data, to identify among 10^4 simulated time series which one is the best representative of the type of data we are dealing with. Below is some other highlights and recommendations:

- All the discussion applies to *stationary* time series. It assumes that any non-stationary components (such as trends) have been removed, to apply the methods discussed here. The datasets in this project meet this requirement.

- To measure the quality of fit, it is tempting to use the correlation between simulated and real data. However this approach favors simulated data that is a replicate of the original data. To the contrary, comparing the two autocorrelation structures favors simulated data of the same type as the real data, but not identical. It leads to a richer class of synthetic time series, putting emphasis on structural and stochastic similarity, rather than being "the same". It also minimizes overfitting.
- Try different seeds for the random generator, and see how the solution changes based on the seed. Also, rather than using the sum of absolute value of differences between various autocorrelation lags, try the max, median, or assign a different weight to each lag (such as decaying weights). Or use transformed auto-correlations using a logarithm transform.
- A classic metric to assess the quality of synthetic data is the Hellinger distance, popular because it yields a value between 0 and 1. It measures the proximity between the two marginal distributions – here, that of the simulated and real time series. It is not useful for time series though, because you can have the same marginals and very different auto-correlation structures. Note that the metric I use also yields values between 0 and 1, with zero being best, and 1 being worst.
- The simulation was able to generate values outside the range of observed (real) values. Many synthetic data algorithms fail at that, because they use percentile-based methods (for instance, copulas) for data generation or to measure the quality (Hellinger is in that category). Empirical percentile distributions used for that purpose, including the Python version in the Statsmodels library, have this limitation.

3.2 Temperature data: geospatial smoothness and interpolation

The purpose here is twofold. First, using the pykrige Python library to performs ordinary kriging, to estimate temperatures outside the sensor locations for the Chicago temperature dataset `sensors.csv` available on GitHub, here. The details are in my book [20], in chapter 9. In addition, we will use the osmnx library (Open Street Map) to superimpose a map of the Chicago area on the final output, as seen in Figure 3.10.

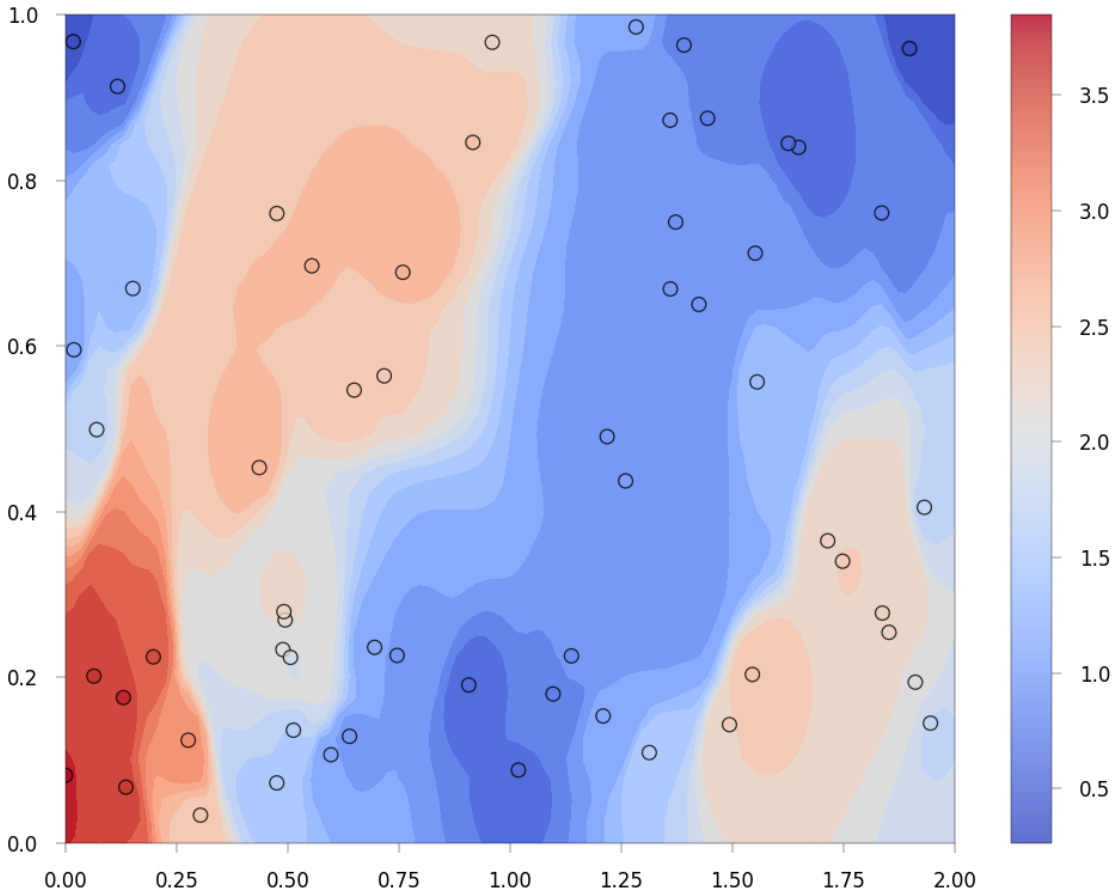


Figure 3.6: Interpolation of the entire grid; dots are training set locations

Then, we will use a generated dataset, with temperatures replaced by an arbitrary math function, in this case a mixture of bivariate Gaussian densities, or Gaussian mixture model. This allows us to simulate hundreds

or thousands of values at arbitrary locations, by contrast with the temperature dataset based on 31 observations only. The math function in question is pictured in Figure 3.7. In this case, rather than kriging, we explore an exact bivariate interpolation method, also featured in chapter 9 in my book. The goal is to compare with kriging, using cross-validation, as shown in Figure 3.8. The solution offered here is a lot faster than the code in my book, thanks to replacing loops with vector and matrix operations. Eventually, we interpolate the entire grid: see Figure 3.6. Picking up random locations on the grid, together with the corresponding interpolated values, produces synthetic geospatial data. It can be used to generate artificial elevation maps, with potential applications in video games.

Last but not least, we define the concept of spatial smoothness for geospatial data, using functions of second-order discrete derivatives. While it is easy to play with parameters of any given algorithm to produce various degrees of smoothness, it is a lot harder to define smoothness in absolute terms, allowing you to compare results produced by two different algorithms. Another important point is overfitting. Despite using exact interpolation for hundreds of locations – akin to using a regression model with hundreds of regression coefficients – we are able to avoid overfitting.

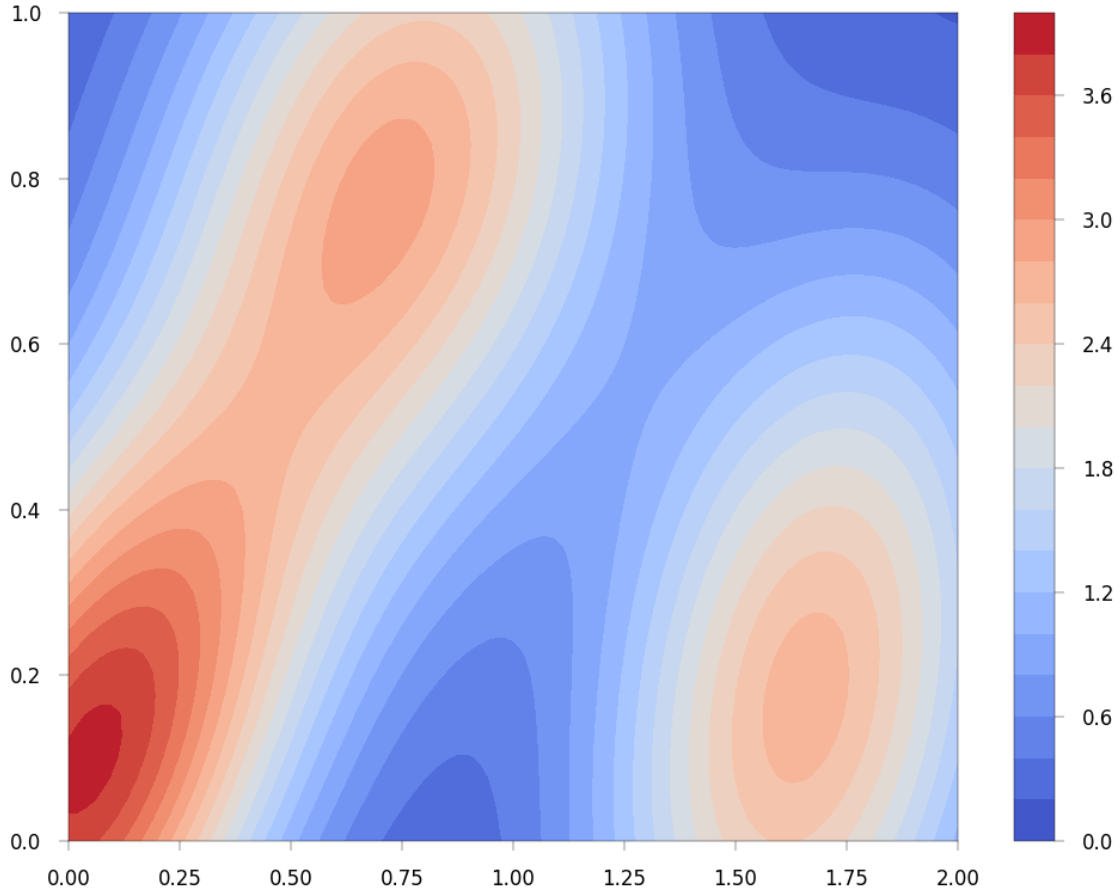


Figure 3.7: Math function used to sample training set locations and values

3.2.1 Project description

The Chicago temperature dataset `sensors.csv` is available here. It contains latitudes, longitudes and temperatures measured at 31 locations, at one point in time. We will use it to illustrate kriging. The algorithm for exact geospatial interpolation is in chapter 9 in my book [20]. The original study of the temperature data set is in the same chapter. A Jupyter notebook, published by the University of Illinois, can be found here. A shorter version of the interpolation code, applied to temperatures in India, can be found on GitHub, here.

The project consists of the following steps:

Step 1: Using the Pykrige library, write Python code to perform ordinary kriging on the temperature data set and interpolate values in the entire area. You may look at the Jypiter notebook aforementioned, to eventually produce a picture similar to Figure 3.10. Try different types of kriging and parameters. Does it always lead to smooth interpolation, similar to time series smoothing? What happens when extrapolating, that is, sampling far outside the training set?

Step 2: Let us consider the following mathematical function, a mixture of m bivariate Gaussian densities defined on $D = [0, 2] \times [0, 1]$ with weights w_k, centers (c_{xk}, c_{yk}), variances σ_{xk}, σ_{yk}, and correlations ρ_k:

$$f(x, y) \propto \sum_{k=1}^{m} \frac{w_k}{\sigma_{xk}\sigma_{yk}\sqrt{1-\rho_k^2}} \exp\left[-\left\{ \frac{(x-c_{xk})^2}{\sigma_{xk}^2} - \frac{2\rho_k(x-c_{xk})(y-c_{yk})}{\sigma_{xk}\sigma_{yk}} + \frac{(y-c_{yk})^2}{\sigma_{yk}^2} \right\}\right]. \quad (3.3)$$

Here the symbol $\propto$ means "proportional to"; the proportionality constant does not matter.

Choose specific values for the parameters, and sample two sets of locations on D: the training set to define the interpolation formula based on the values of the above function at these locations, and the validation set to check how good the interpolated values are, outside the training set.

Step 3: Use the interpolation formula implemented in section 9.3.2 in my book [20] and also available here on GitHub, but this time on the training and validation sets obtained in step 2, rather than on the Chicago temperature dataset. Also interpolate the entire grid, using 10,000 locations evenly spread on D. Plot the results using scatterplots and contour maps. Compute the interpolation error on the validation set. Show how the error is sensitive to the choice of sampled locations and parameters. Also show that the contour maps for interpolated values are considerably less smooth than for the underlying math function, due to using exact interpolation. Would this be also true if using kriging instead? What are your conclusions?

Step 4: The original code in my book runs very slowly. Detect the bottlenecks. How can you improve the speed by several orders of magnitude? Hint: replace some of the loops by array operations, in Numpy.

Step 5: The goal is to define the concept of smoothness, to compare interpolations and contour maps associated to different algorithms, for instance kriging versus my exact interpolation technique. Unlike 1D time series, there is no perfect answer in 2D: many definitions are possible, depending on the type of data. For instance, it could be based on the amount of chaos or entropy. Here we will use a generalization of the 1D metric

$$S(f, D) = \int_D |f''(w)|^2 dw.$$

Explain why definitions based on first-order derivatives are not good. Search the Internet for a potential solution. I did not find any, but you can check out the answer to the question I posted on Mathoverflow, here. You may create a definition based on transformed gradients and Hessians, such as a matrix norm of the Hessian. These objects are respectively the first and second derivatives (a vector for the gradient, a matrix for the Hessian) attached to multivariate functions. Compute the smoothness on different interpolated grids, to see if your definition matches intuition. You will need a discrete version of the gradient or Hessian, as we are dealing with data (they don't have derivatives!) rather than mathematical functions. Numpy has functions such as `gradient` that do the computations in one line of code, when the input is a grid.

Step 6: Optional. How would you handle correlated bivariate values, such as temperature and pressure measured simultaneously at various locations? How about spatio-temporal data: temperatures evolving over time across multiple locations? Finally, turn Figure 3.6 into a video, by continuously modifying the parameters in the function defined by (3.3), over time, with each video frame corresponding to updated parameters. This is how agent-based modeling works.

3.2.2 Solution

The code to solve step 1 is in section 3.2.2.1. In addition, I used the Osmnx library to superimpose the Chicago street map on the temperature 2D grid. For step 2, see the `gm` function in the code featured in section 3.2.2.2. The function `interpolate` in the same program, is the implementation of the interpolation formula discussed in step 3. More about step 3 and step 4 can be found in the same section. As for step 5, I implemented a discrete version of the following formula, to define and compute the smoothness S on $[0, 2] \times [0, 1]$:

$$S = \int_0^1 \int_0^2 |\nabla(|\nabla z(x, y)|)| dxdy$$

where z is the value at location (x, y) on the grid, ∇ is the gradient operator, and $|\cdot|$ is the Eulidean norm. The discrete gradient on the grid is computed in one line of code, with the `gradient` function available in Numpy.

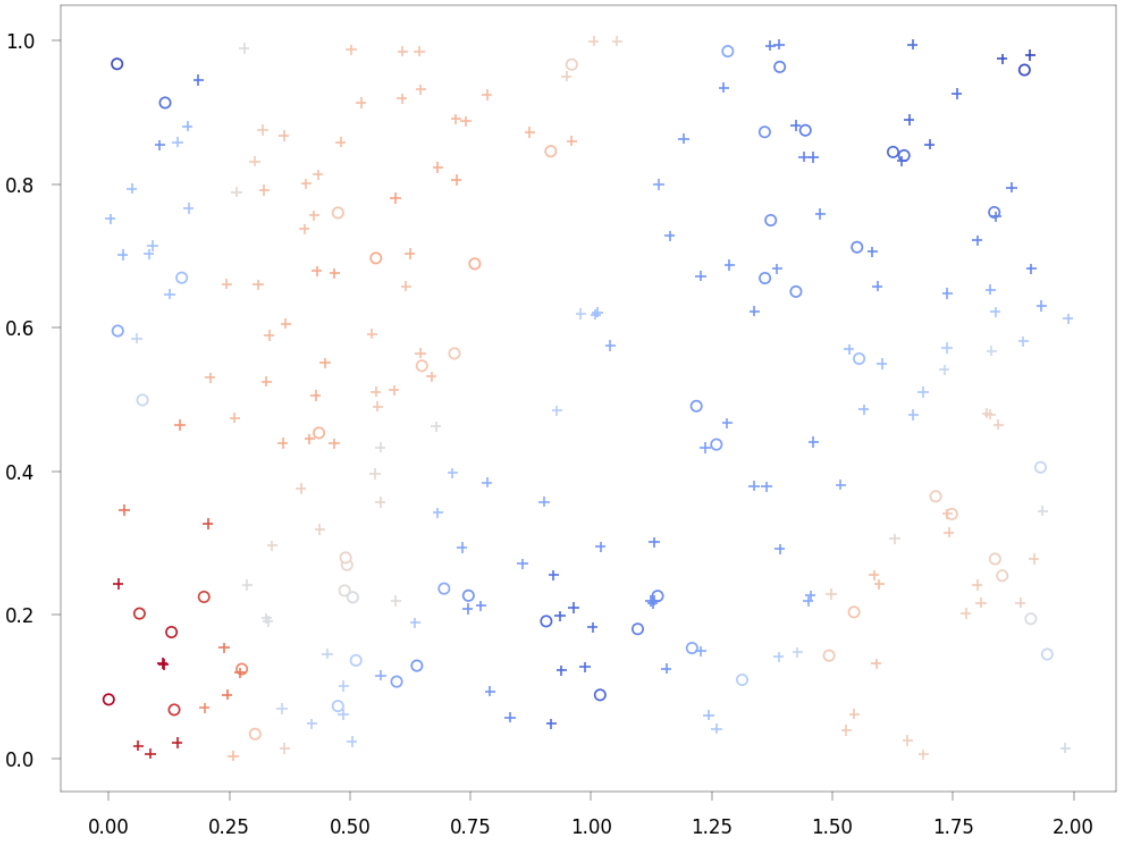


Figure 3.8: Interpolation: dots for training locations, + for validation points

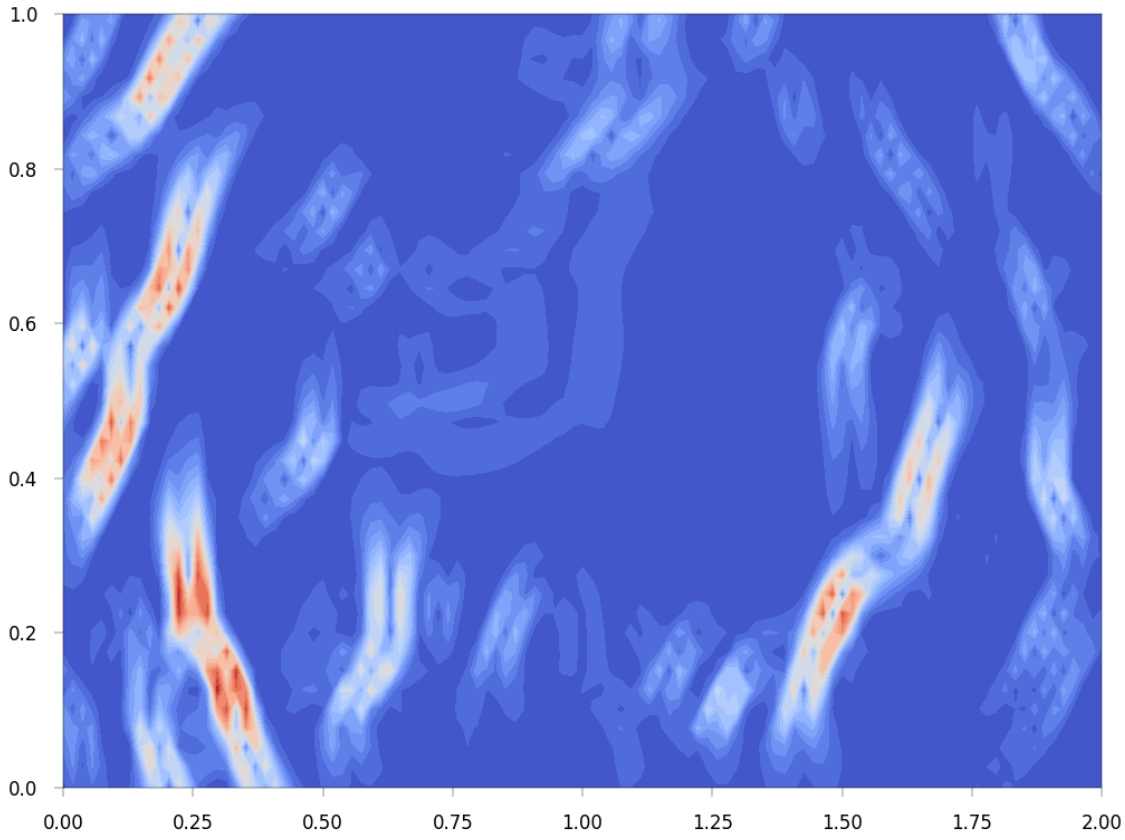


Figure 3.9: Interpolation of entire grid: second-order gradient representing smoothness

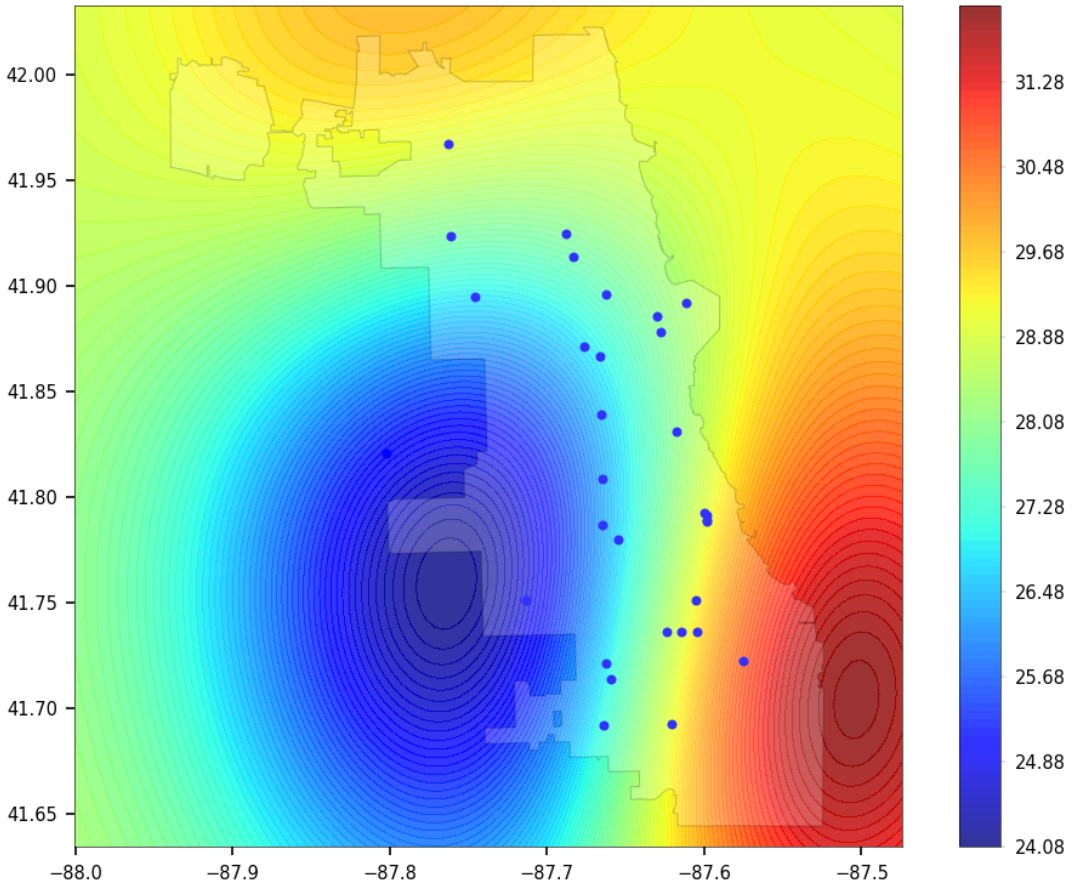


Figure 3.10: Kriging, temperature dataset; dots correspond to actual measurements

Regarding the difference between kriging and my interpolation method (last questions in step 3), kriging tends to produce much smoother results: it is good for measurements such as temperatures, with a smooth gradient. Chaotic processes, for instance the reconstruction of an elevation map or structures similar to Voronoi diagrams [Wiki] are much better rendered with my method, especially when few data points are available. It preserves local variations much better than kriging. Finally, despite offering exact interpolation, my method avoids overfitting, unlike polynomial regression. This is because I use some normalization in the interpolation formula. In short, kriging is a smoothing technique, while my method is best used for data reconstruction or synthetization.

There are many other potential topics to address. I listed below a few suggestions for the reader interested in further exploring this project.

- Play with the parameters $\alpha, \beta, \kappa, \delta$ to increase or decrease smoothness in the exact interpolation method, and see the impact on the error rate (measured on the validation set).
- Add noise to the observed values in the training set, and assess sensitivity of interpolated values to various levels of noise.
- Play with the parameters associated to the `gm` function, to produce many different sets of observed values. Compute the error (`error` in the code in section 3.2.2.2) and relative error, for each parameter set. What is the range of the relative error, depending on the size of the training set?
- Investigate other metrics to measure smoothness (step 5 in the project), for instance 2D generalizations of Hurst exponent [Wiki] used for time series.
- When is it useful to first transform the data, interpolate the transformed data, then apply the inverse transform? For instance, this is done for the Chicago temperature dataset: see chapter 9 in my book [20].
- How far outside the training set locations can you reasonably interpolate without losing too much accuracy? In this case, it is called extrapolation. Check the accuracy of interpolated values for locations in the validation set that are far away from any training set point.
- Compute confidence intervals for the interpolated values (validation set). In order to do so, generate 1000 training sets, each with the same number of points, but different locations. Or use the same training set each time, with a resampling technique such as bootstrapping [Wiki].

Regarding step 6, see how to create a data video in section 1.3.2 in this textbook. An example relevant to this project – an animated elevation map using agent-based modeling – can be found in chapter 14 in my book [20].

3.2.2.1 Kriging

The Chicago temperature dataset is discussed in chapter 9 in my book [20]. My code is inspired from a Jupyter notebook posted here by a team working on this project at the University of Chicago. My Python implementation `kriging_temperatures_chicago.py` listed below is also on GitHub, here.

```
# compare this with my spatial interpolation:
# https://github.com/VincentGranville/Statistical-Optimization/blob/main/interpol.py
# This kriging oversmooth the temperatures, compared to my method

import numpy as np
import matplotlib as mpl
from matplotlib import pyplot as plt
from matplotlib import colors
from matplotlib import cm # color maps
import osmnx as ox
import pandas as pd
import glob
from pykrige.ok import OrdinaryKriging
from pykrige.kriging_tools import write_asc_grid
import pykrige.kriging_tools as kt
from matplotlib.colors import LinearSegmentedColormap

city = ox.geocode_to_gdf('Chicago, IL')
city.to_file("il-chicago.shp")

data = pd.read_csv(
    'sensors.csv',
    delim_whitespace=False, header=None,
    names=["Lat", "Lon", "Z"])

lons=np.array(data['Lon'])
lats=np.array(data['Lat'])
zdata=np.array(data['Z'])

import geopandas as gpd
Chicago_Boundary_Shapefile = 'il-chicago.shp'
boundary = gpd.read_file(Chicago_Boundary_Shapefile)

# get the boundary of Chicago
xmin, ymin, xmax, ymax = boundary.total_bounds

xmin = xmin-0.06
xmax = xmax+0.05
ymin = ymin-0.01
ymax = ymax+0.01
grid_lon = np.linspace(xmin, xmax, 100)
grid_lat = np.linspace(ymin, ymax, 100)

#------
# ordinary kriging

OK = OrdinaryKriging(lons, lats, zdata, variogram_model='gaussian', verbose=True,
    enable_plotting=False,nlags=20)
z1, ss1 = OK.execute('grid', grid_lon, grid_lat)
print (z1)

#-------
# plots

xintrp, yintrp = np.meshgrid(grid_lon, grid_lat)
plt.rcParams['axes.linewidth'] = 0.3
fig, ax = plt.subplots(figsize=(8,6))
```

```
contour = plt.contourf(xintrp, yintrp, z1,len(z1),cmap=plt.cm.jet,alpha = 0.8)
cbar = plt.colorbar(contour)
cbar.ax.tick_params(width=0.1)
cbar.ax.tick_params(length=2)
cbar.ax.tick_params(labelsize=7)

boundary.plot(ax=ax, color='white', alpha = 0.2, linewidth=0.5, edgecolor='black',
    zorder = 5)
npts = len(lons)

plt.scatter(lons, lats,marker='o',c='b',s=8)
plt.xticks(fontsize = 7)
plt.yticks(fontsize = 7)
plt.show()
```

3.2.2.2 Exact interpolation and smoothness evaluation

The instruction `(za,npt)=interpolate(xa,ya,npdata,0.5*delta)` produces interpolated values `za` for the the validation set locations `(xa,ya)`. The parameter `delta` controls the maximum distance allowed between a location, and a training set point used to interpolate the value at the location in question. The array `npt` stores the number of training set points used to compute each interpolated value. Also, the instruction `z,npt=interpolate(xg,yg,npdata,2.2*delta)` interpolates the entire grid.

The number of loops has been reduced to make the code run faster. In particular, the arguments `xa,ya` can be arrays; accordingly, the output `z` of the `interpolate` function can be an array or even a grid of interpolated values. Likewise, `npt` can be an array or grid, matching the shape of `z`. Finally, `error` measures the mean absolute error between exact and interpolated values on the validation set, while `average_smoothness` measures the smoothness of the grid, original or interpolated. The Python code, `interpol_smooth.py`, is also on GitHub, here.

```
import warnings
warnings.filterwarnings("ignore")

import numpy as np
import matplotlib as mpl
from matplotlib import pyplot as plt
from matplotlib import colors
from matplotlib import cm # color maps

n = 60      # number of observations in training set
ngroups = 3 # number of clusters in Gaussian mixture
seed = 13   # to initiate random number generator

#--- create locations for training set

width = 2
height = 1
np.random.seed(seed)
x = np.random.uniform(0, width, n)
y = np.random.uniform(0, height, n)


#--- create values z attached to these locations

weights = np.random.uniform(0.5, 0.9, ngroups)
sum = np.sum(weights)
weights = weights/sum

cx = np.random.uniform(0, width, ngroups)
cy = np.random.uniform(0, height, ngroups)
sx = np.random.uniform(0.3, 0.4, ngroups)
sy = np.random.uniform(0.3, 0.4, ngroups)
rho = np.random.uniform(-0.3, 0.5, ngroups)
```

```
def f(x, y, cx, cy, sx, sy, rho):

   # bivariate bell curve

   tx = ( (x - cx) / sx)**2
   ty = ( (y - cy) / sy)**2
   txy = rho * (x - cx) * (y - cy) / (sx * sy)
   z = np.exp(-(tx - 2*txy + ty) / (2*(1 - rho**2)) )
   z = z / (sx * sy * np.sqrt(1 - rho**2))
   return(z)

def gm(x, y, weights, cx, cy, sx, sy, rho):

   # mixture of gaussians

   n = len(x)
   ngroups = len(cx)
   z = np.zeros(n)
   for k in range(ngroups):
      z += weights[k] * f(x, y, cx[k], cy[k], sx[k], sy[k], rho[k])
   return(z)

z = gm(x, y, weights, cx, cy, sx, sy, rho)
npdata = np.column_stack((x, y, z))

print(npdata)

#--- model parameters

alpha = 1.0   # small alpha increases smoothing
beta = 2.0    # small beta increases smoothing
kappa = 2.0   # high kappa makes method close to kriging
eps  = 1.0e-8 # make it work if sample locations same as observed ones
delta = eps + 1.2 * max(width, height) # don't use faraway points for interpolation

#--- interpolation for validation set: create locations

n_valid = 200 # number of locations to be interpolated, in validation set
xa = np.random.uniform(0, width, n_valid)
ya = np.random.uniform(0, height, n_valid)

#--- interpolation for validation set

def w(x, y, x_k, y_k, alpha, beta):
   # distance function
   z = (abs(x - x_k)**beta + abs(y - y_k)**beta)**alpha
   return(z)


def interpolate(x, y, npdata, delta):

   # compute interpolated z at location (x, y) based on npdata (observations)
   # also returns npt, the number of data points used for each interpolated value
   # data points (x_k, y_k) with w[(x,y), (x_k,y_k)] >= delta are ignored
   # note: (x, y) can be a location or an array of locations

   if np.isscalar(x): # transform scalar to 1-cell array
      x = [x]
      y = [y]
   sum = np.zeros(len(x))
   sum_coeff = np.zeros(len(x))
   npt = np.zeros(len(x))

   for k in range(n):
      x_k = npdata[k, 0]
```

```
      y_k = npdata[k, 1]
      z_k = npdata[k, 2]
      coeff = 1
      for i in range(n):
         x_i = npdata[i, 0]
         y_i = npdata[i, 1]
         if i != k:
            numerator = w(x, y, x_i, y_i, alpha, beta)
            denominator = w(x_k, y_k, x_i, y_i, alpha, beta)
            coeff *= numerator / (eps + denominator)
      dist = w(x, y, x_k, y_k, alpha, beta)
      coeff = (eps + dist)**(-kappa) * coeff / (1 + coeff)
      coeff[dist > delta] = 0.0
      sum_coeff += coeff
      npt[dist < delta] += 1
      sum += z_k * coeff

   z = sum / sum_coeff
   return(z, npt)

(za, npt) = interpolate(xa, ya, npdata, 0.5*delta)

#--- create 2D grid with x_steps times y_steps locations, and interpolate entire grid

x_steps = 160
y_steps = 80
xb = np.linspace(min(npdata[:,0])-0.50, max(npdata[:,0])+0.50, x_steps)
yb = np.linspace(min(npdata[:,1])-0.50, max(npdata[:,1])+0.50, y_steps)
xc, yc = np.meshgrid(xb, yb)

zgrid = np.empty(shape=(x_steps,y_steps)) # for interpolated values at grid locations
xg = []
yg = []
gmap = {}
idx = 0
for h in range(len(xb)):
   for k in range(len(yb)):
      xg.append(xb[h])
      yg.append(yb[k])
      gmap[h, k] = idx
      idx += 1
z, npt = interpolate(xg, yg, npdata, 2.2*delta)

zgrid_true = np.empty(shape=(x_steps,y_steps))
xg = np.array(xg)
yg = np.array(yg)
z_true = gm(xg, yg, weights, cx, cy, sx, sy, rho) # exact values on the grid

for h in range(len(xb)):
   for k in range(len(yb)):
      idx = gmap[h, k]
      zgrid[h, k] = z[idx]
      zgrid_true[h, k] = z_true[idx]
zgridt = zgrid.transpose()
zgridt_true = zgrid_true.transpose()

#--- visualizations

nlevels = 20 # number of levels on contour plots

def set_plt_params():
   # initialize visualizations
   fig = plt.figure(figsize =(4, 3), dpi=200)
   ax = fig.gca()
   plt.setp(ax.spines.values(), linewidth=0.1)
   ax.xaxis.set_tick_params(width=0.1)
```

```
    ax.yaxis.set_tick_params(width=0.1)
    ax.xaxis.set_tick_params(length=2)
    ax.yaxis.set_tick_params(length=2)
    ax.tick_params(axis='x', labelsize=4)
    ax.tick_params(axis='y', labelsize=4)
    plt.rc('xtick', labelsize=4)
    plt.rc('ytick', labelsize=4)
    plt.rcParams['axes.linewidth'] = 0.1
    return(fig,ax)

# contour plot, interpolated values, full grid

(fig1, ax1) = set_plt_params()
cs1 = plt.contourf(xc, yc, zgridt, cmap='coolwarm',levels=nlevels,linewidths=0.1)
sc1 = plt.scatter(npdata[:,0], npdata[:,1], c=npdata[:,2], s=8, cmap=cm.coolwarm,
     edgecolors='black',linewidth=0.3,alpha=0.8)
cbar1 = plt.colorbar(sc1)
cbar1.ax.tick_params(width=0.1)
cbar1.ax.tick_params(length=2)
plt.xlim(0, width)
plt.ylim(0, height)
plt.show()

# scatter plot: validation set (+) and training data (o)

(fig2, ax2) = set_plt_params()
my_cmap = mpl.colormaps['coolwarm'] # old version: cm.get_cmap('coolwarm')
my_norm = colors.Normalize()
ec_colors = my_cmap(my_norm(npdata[:,2]))
sc2a = plt.scatter(npdata[:,0], npdata[:,1], c='white', s=5, cmap=my_cmap,
    edgecolors=ec_colors,linewidth=0.4)
sc2b = plt.scatter(xa, ya, c=za, cmap=my_cmap, marker='+',s=5,linewidth=0.4)
plt.show()
plt.close()

#--- measuring quality of the fit on validation set
#   zd is true value, za is interpolated value

zd = gm(xa, ya, weights, cx, cy, sx, sy, rho)
error = np.average(abs(zd - za))
print("\nMean absolute error on validation set: %6.2f" %(error))
print("Mean value on validation set: %6.2f" %(np.average(zd)))

#--- plot of original function (true values)

(fig3, ax3) = set_plt_params()
cs3 = plt.contourf(xc, yc, zgridt_true, cmap='coolwarm',levels=nlevels,linewidths=0.1)
cbar1 = plt.colorbar(cs3)
cbar1.ax.tick_params(width=0.1)
cbar1.ax.tick_params(length=2)
plt.xlim(0, width)
plt.ylim(0, height)
plt.show()

#--- compute smoothness of interpolated grid via double gradient
#   1/x_steps and 1/y_steps are x, y increments between 2 adjacent grid locations

h2 = x_steps**2
k2 = y_steps**2
dx, dy = np.gradient(zgrid) # zgrid_true for original function
zgrid_norm1 = np.sqrt(h2*dx*dx + k2*dy*dy)
dx, dy = np.gradient(zgrid_norm1)
zgrid_norm2 = np.sqrt(h2*dx*dx + k2*dy*dy)
zgridt_norm2 = zgrid_norm2.transpose()
average_smoothness = np.average(zgrid_norm2)
print("Average smoothness of interpolated grid: %6.3f" %(average_smoothness))
```

```
(fig4, ax4) = set_plt_params()
cs4 = plt.contourf(xc, yc, zgridt_norm2, cmap=my_cmap,levels=nlevels,linewidths=0.1)
plt.xlim(0, width)
plt.ylim(0, height)
plt.show()
```

Chapter 4

Scientific Computing

Many projects throughout this book feature scientific programming in Python. This section offers a selection that best illustrates what scientific computing is about. In many cases, special libraries are needed, or you have to process numbers with billions of digits. Applications range from heavy simulations to cybersecurity. In several instances, very efficient algorithms are required.

4.1 The music of the Riemann Hypothesis: sound generation

This first project is an introduction to sound generation in Python with wave files, as well as the MPmath Python library to work with special functions including in the complex plane. This library is particularly useful to physicists. No special knowledge of complex functions is required: we work with the real and imaginary part separately, as illustrated in Figure 4.1: in this example, the frequency attached to a musical note is the real part of the complex Dirichlet eta function η [Wiki] (re-scaled to be positive), the duration of a note is the imaginary part of η after re-scaling, and the volume – also called amplitude – is the the modulus [Wiki].

The project consists of the following steps:

Step 1: Read my article "The Sound that Data Makes", available here. It contains Python code to turn random data into music. Also download the technical paper "Math-free, Parameter-free Gradient Descent in Python" available from here. It contains Python code to compute the Dirichlet eta function (among others) using the MPmath library.

Step 2: Using the material gathered in step 1, generate a sound file (wav extension) for the Dirichlet eta function $\eta(\sigma + it)$, with $\sigma = \frac{1}{2}$ and t between 400,000 and 400,020. Create 300 musical notes, one for each value of t equally spaced in the interval in question. Each note has 3 components: the frequency, duration, and volume, corresponding respectively to the real part, imaginary part, and modulus of $\eta(\sigma + it)$ after proper re-scaling.

Step 3: Same as step 2, but this time with a different function of your choice. Or better, with actual data rather than sampled values from mathematical functions. For instance, try with the ocean tide data, or the planet inter-distance data investigated in project 3.1.

Step 4: Optional. Create a sound track for the video featured in Figure 4.2. You can watch the video on YouTube, here. The code to produce this video (with detailed explanations) is in section 4.3.2 in my book [20], and also on GitHub, here. To blend the sound track and the video together, see solution on Stack Exchange, here. An alternative is to convert the wav file to mp4 formart (see how to do it here) then use the Moviepy library to combine them both.

Data visualizations offer colors and shapes, allowing you to summarize multiple dimensions in one picture. Data animations (videos) go one step further, adding a time dimension. See examples on my YouTube channel, and in my book [20]. Then, sound adds multiple dimensions: amplitude, volume and frequency over time. Producing pleasant sound, with each musical note representing a multivariate data point, is equivalent to data binning or bukectization. Stereo and the use of multiple musical instruments (synthesized) add more dimensions. Once you have a large database of data music, you can use it for generative AI: sound generation to mimic existing datasets. Of course, musical AI art is another application, all the way to creating synthetic movies.

Figure 4.1 shows the frequency, duration and volume attached to each of the 300 musical notes in the wav file, prior to re-scaling. The volume is maximum each time the Riemann zeta function hits a zero on the critical line. This is one of the connections to the Riemann Hypothesis. Note that in the solution, I use the

Dirichlet eta function $\eta(\sigma+it)$ with $\sigma=\frac{1}{2}$, corresponding to the critical line [Wiki]. According to the Riemann Hypothesis, this is the only positive value of σ where all the zeros of the Riemann zeta function $\zeta(\sigma+it)$ occur. Indeed, there are infinitely many t for which $\zeta(\frac{1}{2}+it)=0$. You can see the first 100 billion of them, here. The Dirichlet eta function has the same zeros. This is the connection to the Riemann Hypothesis. The notation $\sigma+it$ represents the complex argument of the functions involved, with σ the real part and t the imaginary part. More on this topic in chapter 17 in my book [20].

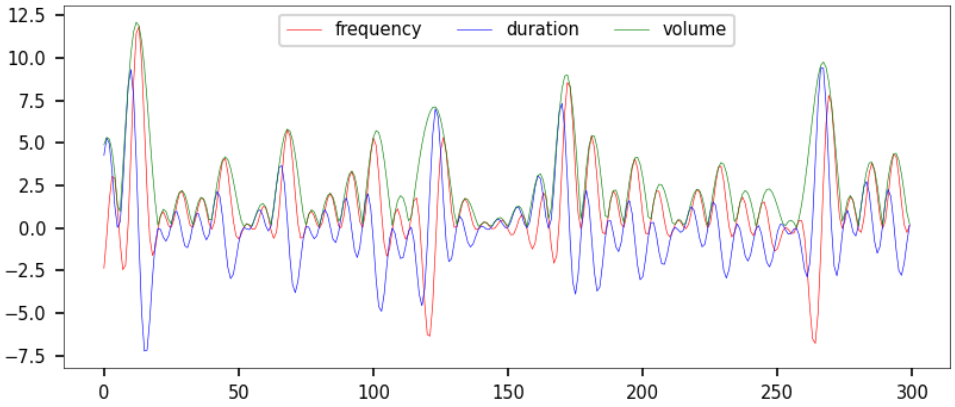


Figure 4.1: 300 musical notes, showing volume, duration and frequency

4.1.1 Solution

The code in this section provides the answer to step 2. The variables `z.real` and `z.imag` correspond respectively to the real and imaginary part of z. The volume in the output wav file (the music) is maximum each time the Riemann zeta or Dirichlet eta function hits a zero on the critical line. The Python code is also on my GitHub repository here.

Figure 4.2 shows the final frame of the video discussed in step 4. It features the convergence path of the Dirichlet eta function in the complex plane, for a specific value of the complex argument $\sigma+it$, when adding more and more terms in the standard sine and cosine series to approximate the function. Here t is very large, and σ is in the critical band $\frac{1}{2}\leq\sigma<1$, where the most interesting action takes place.

Figure 4.2: Last frame from the video featuring the convergence of the Dirichlet eta function

```
import numpy as np
import matplotlib as mpl
import matplotlib.pyplot as plt
from scipy.io import wavfile
import mpmath
```

```
#-- Create the list of musical notes

scale=[]
for k in range(35, 65):
   note=440*2**((k-49)/12)
   if k%12 != 0 and k%12 != 2 and k%12 != 5 and k%12 != 7 and k%12 != 10:
      scale.append(note) # add musical note (skip half tones)
n_notes = len(scale) # number of musical notes


#-- Generate the data

n = 300
sigma = 0.5
min_t = 400000
max_t = 400020

def create_data(f, nobs, min_t, max_t, sigma):
   z_real = []
   z_imag = []
   z_modulus = []
   incr_t = (max_t - min_t) / nobs
   for t in np.arange(min_t, max_t, incr_t):
      if f == 'Zeta':
         z = mpmath.zeta(complex(sigma, t))
      elif f == 'Eta':
         z = mpmath.altzeta(complex(sigma, t))
      z_real.append(float(z.real))
      z_imag.append(float(z.imag))
      modulus = np.sqrt(z.real*z.real + z.imag*z.imag)
      z_modulus.append(float(modulus))
   return(z_real, z_imag, z_modulus)

(z_real, z_imag, z_modulus) = create_data('Eta', n, min_t, max_t, sigma)

size = len(z_real) # should be identical to nobs
x = np.arange(size)

# frequency of each note
y = z_real
min = np.min(y)
max = np.max(y)
yf = 0.999*n_notes*(y-min)/(max-min)

# duration of each note
z = z_imag
min = np.min(z)
max = np.max(z)
zf = 0.1 + 0.4*(z-min)/(max-min)

# volume of each note
v = z_modulus
min = np.min(v)
max = np.max(v)
vf = 500 + 2000*(1 - (v-min)/(max-min))

#-- plot data

mpl.rcParams['axes.linewidth'] = 0.3
fig, ax = plt.subplots()
ax.tick_params(axis='x', labelsize=7)
ax.tick_params(axis='y', labelsize=7)
plt.rcParams['axes.linewidth'] = 0.1
plt.plot(x, y, color='red', linewidth = 0.3)
plt.plot(x, z, color='blue', linewidth = 0.3)
```

```
plt.plot(x, v, color='green', linewidth = 0.3)
plt.legend(['frequency','duration','volume'], fontsize="7",
   loc ="upper center", ncol=3)
plt.show()

#-- Turn the data into music

def get_sine_wave(frequency, duration, sample_rate=44100, amplitude=4096):
   t = np.linspace(0, duration, int(sample_rate*duration))
   wave = amplitude*np.sin(2*np.pi*frequency*t)
   return wave

wave=[]
for t in x: # loop over dataset observations, create one note per observation
   note = int(yf[t])
   duration = zf[t]
   frequency = scale[note]
   volume = vf[t] ## 2048
   new_wave = get_sine_wave(frequency, duration = zf[t], amplitude = vf[t])
   wave = np.concatenate((wave,new_wave))
wavfile.write('sound.wav', rate=44100, data=wave.astype(np.int16))
```

4.2 Cross-correlations in binary digits of irrational numbers

This short off-the-beaten-path project constitutes an excellent preparation for job interviews. In a nutshell, the task consists of implementing the grade-school multiplication algorithm [Wiki] for numbers with millions of digits in base 2, in this case random digits. The practical goal is to assess when and how sequences of bits or binary digits are correlated or not. It is an important issue in the quadratic irrational random number generator (PRNG) discussed in chapter 11 in my book [20].

Indeed, this PRNG relies on blending billions of digits from millions of irrational numbers, using a very fast algorithm. Three of these numbers could be $\sqrt{2341961}, \sqrt{1825361}$ and $\sqrt{2092487}$. It turns out that the binary digits of $\sqrt{2341961}$ and $\sqrt{1825361}$ are correlated, when computing the empirical correlation on a growing sequence of digits. But those of $\sqrt{2341961}$ and $\sqrt{2092487}$ are not, at the limit when the number of digits becomes infinite. In the former case, the correlation is about 1.98×10^{-5}. The exact value is $1/50429$. While very close to zero, it is not zero, and this is a critical cybersecurity issue when designing military-grade PRNGs. Irrational numbers must be selected in such a way that all cross-correlations are zero.

The theoretical explanation is simple, though hard to prove. If X is an irrational number, and p, q are two positive co-prime odd integers, then the correlation between the binary digits of pX and qX, is $1/(pq)$. Again, this is the limit value when the number n of digits in each sequence tends to infinity. In my example, $2341961 = 239^2 \times 41, 1825361 = 211^2 \times 41, 2092487 = 211^2 \times 47$, and $50429 = 239 \times 211$.

4.2.1 Project and solution

Write a program that computes the binary digits of pX using grade-school multiplication. Here p is a positive integer, and X is a random number in $[0, 1]$. Use this program to compute the correlation between the sequences of binary digits of pX and qX, where p, q are positive integers, and X a number in $[0, 1]$ with random binary digits. Focus on the digits after the decimal point (ignore the other ones).

For the solution, see my Python code below. It is also on GitHub, here. It creates the digits of X, then those of pX and qX, starting backward with the last digits. Finally it computes the correlation in question, assuming the digits of X are random. It works if X has a finite number of digits, denoted as `kmax` in the code. By increasing `kmax`, you can approximate any X with infinitely many digits, arbitrarily closely.

```
# Compute binary digits of X, p*X, q*X backwards (assuming X is random)
# Only digits after the decimal point (on the right) are computed
# Compute correlations between digits of p*X and q*X
# Include carry-over when performing grade school multiplication

import numpy as np

# main parameters
```

```
seed = 105
np.random.seed(seed)
kmax = 1000000
p = 5
q = 3

# local variables
X, pX, qX = 0, 0, 0
d1, d2, e1, e2 = 0, 0, 0, 0
prod, count = 0, 0

# loop over digits in reverse order
for k in range(kmax):

   b = np.random.randint(0, 2) # digit of X
   X = b + X/2

   c1 = p*b
   old_d1 = d1
   old_e1 = e1
   d1 = (c1 + old_e1//2) %2 # digit of pX
   e1 = (old_e1//2) + c1 - d1
   pX = d1 + pX/2

   c2 = q*b
   old_d2 = d2
   old_e2 = e2
   d2 = (c2 + old_e2//2) %2 #digit of qX
   e2 = (old_e2//2) + c2 - d2
   qX = d2 + qX/2

   prod += d1*d2
   count += 1
   correl = 4*prod/count - 1

   if k% 10000 == 0:
      print("k = %7d, correl = %7.4f" % (k, correl))

print("\np = %3d, q = %3d" %(p, q))
print("X = %12.9f, pX = %12.9f, qX = %12.9f" % (X, pX, qX))
print("X = %12.9f, p*X = %12.9f, q*X = %12.9f" % (X, p*X, q*X))
print("Correl = %7.4f, 1/(p*q) = %7.4f" % (correl, 1/(p*q)))
```

4.3 Longest runs of zeros in binary digits of $\sqrt{2}$

Studying the longest head runs in coin tossing has a very long history, starting in gaming and probability theory. Today, it has applications in cryptography and insurance [2]. For random sequences or Bernoulli trials, the associated statistical properties and distributions have been studied in details [9], even when the proportions of zero and one are different. Yet, I could barely find any discussion on deterministic sequences, such as the digits or irrational numbers [33]. The case study investigated here fills this gap, focusing on one of the deepest and most challenging problems in number theory: almost all the questions about the distribution of these digits, even the most basic ones such as the proportions of zero and one, are still unsolved conjectures to this day.

In this context, a run is a sequence of successive, identical digits. In random sequences of bits, runs have a specific probability distribution. In particular, the maximum length of a run in a random sequence of n binary digits has expectation $\log_2 n$. For details and additional properties, see [35]. This fact can be used to test if a sequence violates the laws of randomness. Pseudo-random number generators (PRNG) that do not pass this test are not secure.

The focus here is on sequences of binary digits of quadratic irrational numbers of the form $x_0 = \sqrt{p/q}$, where p, q are positive coprime integers. The goal is to show, using empirical evidence, that indeed such sequences pass the test. More precisely, the project consists of computing runs of zeros in billions of successive binary digits of x_0 for specific p and q. Let L_n be the length of such a run starting at position n in the digit expansion of x_0. Do we have $L_n / \log_2 n \leq \lambda$ where λ is a positive constant? Based on the computations, is it reasonable to

conjecture that $\lambda = 1$ if n is large enough? Very little is known about these digits. However, before discussing the project in more details, I share a much weaker yet spectacular result, easy to prove. By contrast to the digits investigated here, there is an abundance of far more accurate theoretical results, proved long ago, for random bit sequences. See for instance [7, 27, 30].

4.3.1 Surprising result about the longest runs

For numbers such as $\sqrt{p/q}$ where p, q are coprime positive integers and p/q is not the square of a rational number, a run of zeros starting at position n can not be longer than $n + C$ where C is a constant depending on p and q. Due to symmetry, the same is true for runs of ones.

Proof

A run of length m starting at position n, consisting of zeros, means that the digit d_n at position n is one, followed by m digits all zero, and then by at least one non-zero digit. For this to happen, we must have

$$\frac{1}{2} < 2^n x_0 - \Big\lfloor 2^n x_0 \Big\rfloor = \frac{1}{2} + \alpha,$$

where

$$x_0 = \sqrt{\frac{p}{q}} \quad \text{and} \quad \frac{1}{2^{m+2}} \leq \alpha < \sum_{k=0}^{\infty} \frac{1}{2^{m+2+k}} = \frac{1}{2^{m+1}}.$$

Here the brackets represent the floor function. Let $K_n = 1 + 2 \cdot \lfloor 2^n x_0 \rfloor$. Then, $K_n < 2^{n+1} x_0 = K_n + 2\alpha$. After squaring both sides, we finally obtain

$$qK_n^2 < 4^{n+1} \cdot p = q(K_n + 2\alpha)^2.$$

Now, let $\delta_n = 4^{n+1}p - qK_n^2$. Note that δ_n is a strictly positive integer, smallest when p, q are coprime. After some simple rearrangements, we obtain

$$\begin{aligned}
2\alpha &= \frac{1}{\sqrt{q}} \left[2^{n+1}\sqrt{p} - \sqrt{4^{n+1}p - \delta_n} \right] \\
&= \frac{1}{\sqrt{q}} \left[\frac{4^{n+1}p - (4^{n+1}p - \delta_n)}{2^{n+1}\sqrt{p} + \sqrt{4^{n+1}p - \delta_n}} \right] \\
&= \frac{1}{\sqrt{q}} \cdot \frac{\delta_n}{2^{n+1}\sqrt{p} + \sqrt{4^{n+1}p - \delta_n}} \\
&\sim \frac{1}{\sqrt{q}} \cdot \frac{\delta_n}{2 \cdot 2^{n+1}\sqrt{p}} \text{ as } n \to \infty.
\end{aligned}$$

In the last line, I used the fact that δ_n is at most of the order 2^n, thus negligible compared to $4^{n+1}p$.

Since a run of length m means $2^{-(m+2)} \leq \alpha < 2^{-(m+1)}$, combining with the above (excellent) asymptotic result, we have, for large n:

$$\frac{1}{2^{m+2}} \leq \frac{\delta_n}{4 \cdot 2^{n+1}\sqrt{pq}} < \frac{1}{2^{m+1}}.$$

Taking the logarithm in base 2, we obtain

$$m + 1 \leq -\log_2 \delta_n + \frac{1}{2}\log_2(pq) + n + 3 < m + 2,$$

and thus, assuming L_n denotes the length of the run at position n and n is large enough:

$$L_n = \Big\lfloor -\log_2 \delta_n + \frac{1}{2}\log_2(pq) + n + 2 \Big\rfloor \leq n + 2 + \frac{1}{2}\log_2(pq). \tag{4.1}$$

This concludes the proof. It provides an upper bound for the maximum possible run length at position n: in short, $L_n \leq n + C$, where C is an explicit constant depending on p and q. ∎

Conversely, a number whose binary digits do not satisfy (4.1) can not be of the prescribed form. Note that if $p = 1$ and q is large, there will be a long run of zeros at the very beginning, and thus C will be larger than usual. I implemented the equality in Formula (4.1) in my Python code in section 4.3.2. It yielded the exact run length in all instances, for all n where a new run starts. In the code, I use the fact that δ_n can be written as $u - qv^2$, where $u = 4^{n+1}p$ and v is a positive odd integer, the largest one that keeps $\delta_n = u - qv^2$ positive. It leads to an efficient implementation where L_n is computed iteratively as n increases, rather than from scratch for each new n.

4.3.2 Project and solution

You need to be able to correctly compute a large number of binary digits of numbers such as $\sqrt{2}$. In short, you must work with exact arithmetic (or infinite precision). This is one of the big takeaways of this project. As previously stated, the goal is to assess whether the maximum run length in binary digits of $x_0 = \sqrt{p/q}$, grows at the same speed as you would expect if the digits were random. We focus on runs of zeros only. A positive answer would be one more indication (when combined with many other tests) that these digits indeed behave like random bits, and can be used to generate random sequences. Fast, secure random number generators based on quadratic irrationals are described in details in [15].

n	L_n	$\log_2 n$	$L_n/\log_2 n$
1	1	0.0000	
8	5	3.0000	
453	8	8.8234	0.9067
1302	9	10.3465	0.8699
5334	10	12.3810	0.8077
8881	12	13.1165	0.9149
24,001	18	14.5508	1.2370
574,130	19	19.1310	0.9932
3,333,659	20	21.6687	0.9230
4,079,881	22	21.9601	1.0018
8,356,568	23	22.9945	1.0002
76,570,752	25	26.1903	0.9546
202,460,869	26	27.5931	0.9423
457,034,355	28	28.7677	0.9733

Table 4.1: Record runs of zeros in binary digits of $\sqrt{2}/2$

The project consists of the following steps:

Step 1: Computing binary digits of quadratic irrationals. Let $x_0 = \sqrt{p/q}$. Here I assume that p, q are positive coprime integers, and p/q is not the square of a rational number. There are different ways to compute the binary digits of x_0, see [15]. I suggest to use the Gmpy2 Python library: it is written in C, thus very fast, and it offers a lot of functions to work with arbitrary large numbers. In particular, `gmpy2.isqrt(z).base(b)` returns the integer square root [Wiki] of the integer `z`, in base `b`. Thus, to get the first N binary digits of x_0, use $z = \lfloor 2^{2N} \cdot p/q \rfloor$ and $b = 2$. The digits are stored as a string.

Step 2: Computing run lengths. Let L_n be the length of the run of zeros starting at position n in the binary expansion of x_0. A more precise definition is offered in section 4.3.1. If there is no such run starting at n, set $L_n = 0$. Once the digits have been computed, it is very easy to obtain the run lengths: see Table 4.2, corresponding to $p = 1, q = 2$. However, I suggest using Formula (4.1) to compute L_n. Not only it shows that the formula is correct, but it also offers an alternative method not based on the binary digits, thus also confirming that the binary digits are correctly computed. The end result should be an extension of Table 4.2, ignoring the columns labeled as s_n for now. Try with the first $N = 10^6$ digits, with $p = 1$ and $q = 2$.

Step 3: Maximum run lengths. Use a fast technique to compute the successive records for L_n, for the first $N = 10^9$ binary digits of $\sqrt{2}/2$. Is it reasonable to conjecture that asymptotically, $L_n/\log_2 n$ is smaller than or equal to 1? In other words, could we have $\limsup L_n/\log_2 n = 1$? See [Wiki] for the definition of lim sup. Then instead of using the binary digits of some number, do the same test for random bits. Is the behavior similar? Do we get the same upper bound for the record run lengths? The records for L_n, and when their occur (location n) in the binary digit expansion, are displayed in Table 4.1. You should get the same values.

A fast solution to compute both the digits and the run lengths is to use the Gmpy2 library, as illustrated in the first code snippet below (also available on GitHub, here). The output is Table 4.1. The conclusion is that indeed, $\lambda_n = L_n/\log_2 n$ seems to be bounded by 1, at least on average as n increases. It does not mean that $\limsup \lambda_n = 1$. For instance, if S_n is the number of zeros in the first n digits, assuming the digits are random, then the ratio $\rho_n = |S_n - n/2|/\sqrt{n}$ seems bounded. However $\limsup \rho_n = \infty$. To get a strictly positive, finite constant for lim sup, you need to further divide ρ_n by $\sqrt{\log\log n}$. This is a consequence of the law of the iterated logarithm [Wiki]. More details are available in my book on dynamical systems [15]. For λ_n applied to random

binary digits, see here. This completes Step 1 and Step 3.

```
import gmpy2

p = 1
q = 2
N = 1000000000 # precision, in number of binary digits

# compute and store in bsqrt (a string) the N first binary digits of sqrt(p/q)
base = 2
bsqrt = gmpy2.isqrt( (2**(2*N) * p) // q ).digits(base)

last_digit = -1
L = 0
max_run = 0

for n in range(0, N):
   d = int(bsqrt[n]) # binary digit
   if d == 0:
      L += 1
   if d == 1 and last_digit == 0:
         run = L
         if run > max_run:
            max_run = run
            print(n-L, run, max_run)
         L = 0
   last_digit = d
```

At the bottom of this section, I share my Python code for Step 2, with the implementation of formula (4.1) to compute L_n. The results are in Table 4.2, and compatible with those obtained in Step 1 and displayed in Table 4.1. The method based on formula (4.1) is a lot slower. So why try it, you may ask? It is slower because Gmpy2 is implemented more efficiently, and closer to machine arithmetic. And the goal is different: formula (4.1) allows you to double check the earlier computations, using a method that does not require producing the binary digits to determine L_n.

n	d_n	L_n	s_n	n	d_n	L_n	s_n	n	d_n	L_n	s_n	n	d_n	L_n	s_n	n	d_n	L_n	s_n
1	1	1	1	21	0		0	41	1		1	61	0		1	81	1	3	1
2	0		0	22	0		1	42	1	1	1	62	1	3	2	82	0		0
3	1		2	23	1		2	43	0		0	63	0		0	83	0		1
4	1	1	1	24	1	2	1	44	1		2	64	0		1	84	0		1
5	0		0	25	0		0	45	1		1	65	0		1	85	1	2	2
6	1	1	2	26	0		1	46	1		1	66	1	1	2	86	0		0
7	0		0	27	1		2	47	1	2	1	67	0		0	87	0		1
8	1	5	2	28	1	2	1	48	0		0	68	1		2	88	1		2
9	0		0	29	0		0	49	0		1	69	1	2	1	89	1	1	1
10	0		1	30	0		1	50	1		2	70	0		0	90	0		0
11	0		1	31	1		2	51	1	2	1	71	0		1	91	1		2
12	0		1	32	1		1	52	0		0	72	1	1	2	92	1	2	1
13	0		1	33	1		1	53	0		1	73	0		0	93	0		0
14	1	2	2	34	1		1	54	1	2	2	74	1		2	94	0		1
15	0		0	35	1		1	55	0		0	75	1		1	95	1		2
16	0		1	36	1		1	56	0		1	76	1		1	96	1	1	1
17	1		2	37	1	2	1	57	1	4	2	77	1		1	97	0		0
18	1		1	38	0		0	58	0		0	78	1	1	1	98	1		2
19	1		1	39	0		1	59	0		1	79	0		0	99	1		1
20	1	2	1	40	1		2	60	0		1	80	1		2	100	1	1	1

Table 4.2: Binary digit d_n, run length L_n for zeros, and steps s_n at position n, for $\sqrt{2}/2$

Most importantly, the slow method is valuable because it is the first step to make progress towards a better (smaller) upper bound than that featured in section 4.3.1. To get a much stronger bound for the run lengths L_n,

one has to investigate δ_n, denoted as `delta` in the code below. The code is also on GitHub, here. Note that the variable `steps` can only take on three values: 0, 1, and 2. It is represented as s_n in Table 4.2. Improving the asymptotic upper bound $L_n/n \leq 1$ in (4.1) as $n \to \infty$, is incredibly hard. I spent a considerable amount of time to non avail, even though anyone who spends a small amount of time on this problem will be convinced that asymptotically, $L_n/\log_2 n \leq 1$ as $n \to \infty$, a much stronger result. Proving the stronger bound, even though verified in Table 4.1 for n up to 10^9, is beyond the capabilities of the mathematical tools currently available. It may as well not be true or undecidable, nobody knows.

```
import math
import gmpy2

# requirement: 0 < p < q
p = 1
q = 2
x0 = math.sqrt(p/q)
N = 1000 # precision, in number of binary digits for x0

# compute and store in bsqrt (a string) the N first binary digits of x0 = sqrt(p/q)
base = 2
bsqrt = gmpy2.isqrt( (2**(2*N) * p) // q ).digits(base)

for n in range(1, N):

   if n == 1:
      u = p * 4**n
      v = int(x0 * 4**n)
      if v % 2 == 0:
         v = v - 1
   else:
      u = 4*u
      v = 2*v + 1
   steps = 0
   while q*v*v < u:
      v = v + 2
      steps += 1 # steps is always 0, 1, or 2
   v = v - 2
   delta = u - q*v*v
   d = bsqrt[n-1] # binary digit of x0 = sqrt(p/q), in position n

   ## delta2 = delta >> (n - 1)
   ## res = 5/2 + n - math.log(delta,2) - math.log(n, 2)

   run = int(n + 1 + math.log(p*q, 2)/2 - math.log(delta, 2) )
   if d == "0" or run == 0:
      run = ""

   print("%6d %1s %2s %1d" % (n, d, str(run), steps))
```

4.4 Quantum derivatives, GenAI, and the Riemann Hypothesis

If you are wondering how close we are to proving the Generalized Riemann Hypothesis (GRH), you should read on. The purpose of this project is to uncover intriguing patterns in prime numbers, and gain new insights on the GRH. I stripped off all the unnecessary math, focusing on the depth and implications of the material discussed here. You will also learn to use the remarkable MPmath library for scientific computing. This is a cool project for people who love math, with the opportunity to work on state-of-the-art research even if you don't have a PhD in number theory.

Many of my discoveries were made possible thanks to pattern detection algorithms (in short, AI) before leading to actual proofs, or disproofs. This data-driven, bottom-up approach is known as experimental math. It contrasts with the top-down, classic theoretical academic framework. The potential of AI and its computing power should not be underestimated to make progress on the most difficult mathematical problems. It offers a big competitive advantage over professional mathematicians focusing on theory exclusively.

My approach is unusual as it is based on the Euler product. The benefit is that you immediately know when

the target function, say the Riemann zeta function $\zeta(s)$, has a root or not, wherever the product converges. Also, these products represent analytic function [Wiki] wherever they converge.

I use the standard notation in the complex plane: $s = \sigma + it$, where σ, t are respectively the real and imaginary parts. I focus on the real part only (thus $t = 0$) because of the following result: if for some $s = \sigma_0$, the product converges, then it converges for all $s = \sigma + it$ with $\sigma > \sigma_0$. Now let's define the Euler product. The finite version with n factors is a function of s, namely

$$f(s,n) = \prod_{p \in P_n} \left(1 - \frac{\chi(p)}{p^s}\right)^{-1} = \prod_{k=1}^{n} \left(1 - \frac{\chi(p_k)}{p_k^s}\right)^{-1}.$$

Here $P_n = \{2, 3, 5, 7, 11, \dots\}$ is the set of the first n prime numbers, and p_k denotes the k-th prime with $p_1 = 2$. The function $\chi(p)$ can take on three vales only: $0, -1, +1$. This is not the most generic form, but the one that I will be working with in this section. More general versions are investigated in chapter 17, in [20]. Of course, we are interested in the case $n \to \infty$, where convergence becomes the critical issue. Three particular cases are:

- Rienmann zeta, denoted as $\zeta(s,n)$ or $\zeta(s)$ when $n = \infty$. In this case $\chi(p) = 1$ for all primes p. The resulting product converges only if $\sigma > 1$. Again, σ is the real part of s.
- Dirichlet L-function $L_4(s,n)$ [Wiki] with Dirichlet modular character $\chi = \chi_4$ [Wiki]. Denoted as $L_4(s)$ when $n = \infty$. Here $\chi_4(2) = 0$, $\chi_4(p) = 1$ if $p - 1$ is a multiple of 4, and $\chi_4(p) = -1$ otherwise. The product is absolutely convergent if $\sigma > 1$, but convergence status is unknown if $\frac{1}{2} < \sigma \leq 1$.
- Unnamed function $Q_2(s,n)$, denoted as $Q_2(s)$ when $n = \infty$. Here $\chi(2) = 0$. Otherwise, $\chi(p_k) = 1$ if k is even, and $\chi(p_k) = -1$ if k is odd. Again, p_k is the k-th prime with $p_1 = 2$. The product is absolutely convergent if $\sigma > 1$, and conditionally convergent [Wiki] if $\frac{1}{2} < \sigma \leq 1$.

All these products can be expanded into Dirichlet series [Wiki], and the corresponding χ expanded into multiplicative functions [Wiki] over all positive integers. Also, by construction, Euler products have no zero in their conditional and absolute convergence domains. Most mathematicians believe that the Euler product for $L_4(s)$ conditionally converges when $\frac{1}{2} < \sigma \leq 1$. Proving it would be a massive accomplishment. This would be make L_4 the first example of a function satisfying all the requirements of the Generalized Riemann Hypothesis. The Unnamed function Q_2 actually achieves this goal, with the exception that its associated χ is not periodic. Thus, Q_2 lacks some of the requirements. The Dirichlet series associated to Q_2 (the product expansion as a series) is known to convergence and thus equal to the product if $\sigma > \frac{1}{2}$.

The rest of the discussion is about building the framework to help solve this centuries-old problem. It can probably be generalized to L-functions other than L_4, with one notable exception: the Riemann function itself, which was the one that jump-started all this vast and beautiful mathematical theory.

4.4.1 Cornerstone result to bypass the roadblocks

The goal here is to prove that the Euler product $L_4(s,n)$ converges to some constant $c(s)$ as $n \to \infty$, for some $s = \sigma_0 + it$, with $t = 0$ and some $\sigma_0 < 1$. In turns, it implies that it converges at $s = \sigma + it$, for all t and for all $\sigma > \sigma_0$. It also implies that $c(s) = L_4(s)$, the true value obtained by analytic continuation [Wiki]. Finally, it implies that $L_4(s)$ has no zero if $\sigma > \sigma_0$. This would provide a partial solution to the Generalized Riemann Hypothesis, for L_4 rather than the Riemann zeta function $\zeta(s)$, and not with $\sigma_0 = \frac{1}{2}$ (the conjectured lower bound), but a least for some $\sigma_0 < 1$. This is enough to make countless mathematical theorems true, rather than "true conditionally on the fact that the Riemann Hypothesis is true". It also leads to much more precise results regarding the distribution of primes numbers: results that to this day, are only conjectures. The implications are numerous, well beyond number theory.

The chain of implications that I just mentioned, follows mainly from expanding the Euler product into a Dirichlet-L series. In this case, the expansion is as follows, with $s = \sigma + it$ as usual:

$$\prod_{k=1}^{\infty} \left(1 - \frac{\chi_4(p_k)}{p_k^s}\right)^{-1} = \sum_{k=0}^{\infty} \frac{(-1)^{k+1}}{(2k+1)^s}. \tag{4.2}$$

The series obviously converges when $\sigma > 0$. The product converges for sure when $\sigma > 1$. It is believed that it converges as well when $\sigma > \frac{1}{2}$. The goal here is to establish that it converges when $\sigma > \sigma_0$, for some $\frac{1}{2} < \sigma_0 < 1$. When both converge, they converge to the same value, namely $L_4(s)$ as the series is the analytic continuation of the product, for all $\sigma > 0$. And of course, the product can not be zero when it converges. Thus $L_4(s) \neq 0$ if $\sigma > \sigma_0$.

The big question is how to find a suitable σ_0, and show that it must be strictly smaller than 1. I now focus on this point, leading to some unknown σ_0, very likely in the range $0.85 < \sigma_0 < 0.95$, for a number of

reasons. The first step is to approximate the Euler product $L_4(s,n)$ with spectacular accuracy around $\sigma = 0.90$, using statistical techniques and a simple formula. This approximation amounts to denoising the irregularities caused by the prime number distribution, including Chebyshev's bias [Wiki]. After this step, the remaining is standard real analysis, trying to establish a new generic asymptotic result for a specific class of functions, and assuring that it encompasses our framework. The new theorem 4.4.1 in question, albeit independent from number theory, has yet to be precisely stated, let alone proved. The current version is as follows:

Theorem 4.4.1 *Let $A_n = \{a_1, \ldots, a_n\}$ and $B_n = \{b_1, \ldots, b_n\}$ be two finite sequences of real numbers, with $a_n \to 0$ as $n \to \infty$. Also assume that $b_{n+1} - b_n \to 0$. Now, define ρ_n as the ratio of the standard deviations, respectively computed on A_n (numerator) and B_n (denominator). If ρ_n converges to a non-zero value as $n \to \infty$, then b_n also converges.*

The issue to finalize the theorem is to make sure that it is applicable in our context, and add any additional requirements needed (if any). Is it enough to require $\inf \rho_n > 0$ and $\sup \rho_n < \infty$, rather than the convergence of ρ_n to non-zero? A stronger version, assuming $\sqrt{n} \cdot a_n$ is bounded and $\liminf \rho_n = \rho > 0$, leads to

$$\rho b_n - a_n \sim c + \frac{\alpha}{\sqrt{n}} + \frac{\beta}{\sqrt{n \log n}} + \cdots \tag{4.3}$$

where c, α, β are constants. As a result, $b_n \to c/\rho$. For the term $\beta/\sqrt{n \log n}$ to be valid, additional conditions on the asymptotic behavior of a_n and b_n may be required. Note that a_n and $\alpha/\sqrt{n}$ have the same order of magnitude. As we shall see, a_n captures most of the chaotic part of $L_4(s,n)$, while the term $\beta/\sqrt{n \log n}$ significant improves the approximation.

The following fact is at the very core of the GRH proof that I have in mind. Let us assume that b_n depends *continuously* on some parameter σ. If $\rho_n \to 0$ when $\sigma = \sigma_1$, and $\rho_n \to \infty$ when $\sigma = \sigma_2$, then there most be some σ_0 with $\sigma_1 \leq \sigma_0 \leq \sigma_2$ such that ρ_n converges to non-zero, or at least $\limsup \rho_n < \infty$ and $\liminf \rho_n > 0$ when $\sigma = \sigma_0$. This in turn allows us to use the proposed theoretical framework (results such as theorem 4.4.1) to prove the convergence of $L_4(s,n)$ at $\sigma = \sigma_0$. The challenge in our case is to show that there is such a σ_0, satisfying $\sigma_0 < 1$. However, the difficulty is not caused by crossing the line $\sigma = 1$, and thus unrelated to the prime number distribution. Indeed, most of the interesting action – including crossing our red line – takes place around $\sigma = 0.90$. Thus the problem now appears to be generic, rather than specific to GRH.

Now I establish the connection to the convergence of the Euler product $L_4(s,n)$. First, I introduce two new functions:

$$\delta_n(s) = L_4(s,n) - L_4(s), \quad \Lambda_n = \frac{1}{\varphi(n)} \sum_{k=1}^{n} \chi_4(p_k), \tag{4.4}$$

with $\varphi(n) = n$ for $n = 2, 3, 4$ and so on. An important requirement is that $\Lambda(n) \to 0$. I also tested $\varphi(n) = n \log n$. Then, in formula (4.3), I use the following:

$$a_n = \Lambda_n, \quad b_n = \delta_n(s). \tag{4.5}$$

Here, $L_4(s)$ is obtained via analytic continuation, not as the limit of the Euler product $L_4(s,n)$. The reason is because we don't know if the Euler product converges if $\sigma < 1$, although all evidence suggests that this is the case. Convergence of $\delta_n(s)$ translates to $c = 0$ in formula (4.3). Finally, in the figures, the X-axis represents n.

4.4.2 Quantum derivative of functions nowhere differentiable

Discrete functions such as $L_4(s,n)$, when s is fixed and n is the variable, can be scaled to represent a continuous function. The same principle is used to transform a random walk into a Brownian motion, as discussed in section 1.3 in my book on chaos and dynamical systems [15], and pictured in Figure 4.5. This is true whether the function is deterministic or random. In this context, n represents the time.

In general, the resulting function is nowhere differentiable. You can use the integrated function rather than the original one to study the properties, as integration turns chaos into a smooth curve. But what if we could use the derivative instead? The derivative is even more chaotic than the original function, indeed it does not even exist! Yet there is a way to define the derivative and make it particularly useful to discover new insights about the function of interest. This new object called quantum derivative is not a function, but rather a set of points with a specific shape, boundary, and configuration. In some cases, it may consist of multiple curves, or a dense set with homogeneous or non-homogeneous point density. Two chaotic functions that look identical to the naked eye may have different quantum derivatives.

The goal here is not to formally define the concept of quantum derivative, but to show its potential. For instance, in section 3.3.2.1 of the same book [15], I compute the moments of a cumulative distribution function (CDF) that is nowhere differentiable. For that purpose, I use the density function (PDF), which of course

is nowhere defined. Yet, I get the correct values. While transparent to the reader, I implicitly integrated weighted quantum derivatives of the CDF. In short, the quantum derivative of a discrete function $f(n)$ is based on $f(n) - f(n-1)$. If the time-continuous (scaled) version of f is continuous, then the quantum derivative corresponds to the standard derivative. Otherwise, it takes on multiple values, called quantum states [Wiki] in quantum physics.

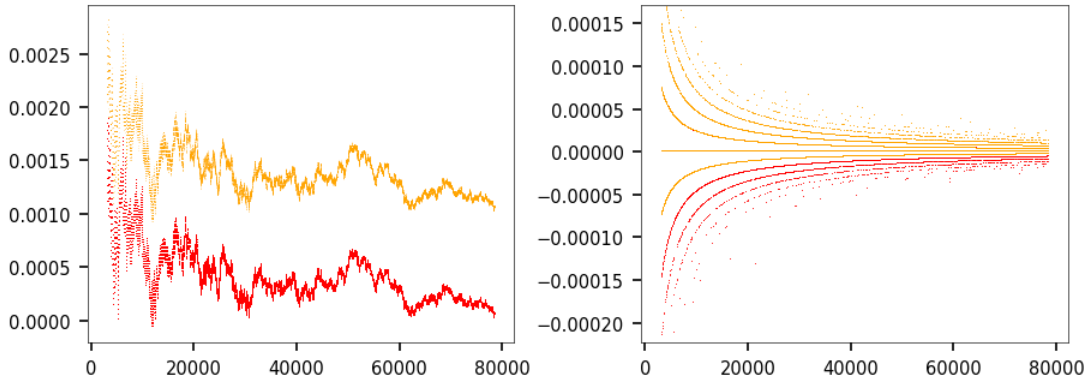


Figure 4.3: Two shifted legs of δ_n (left), and their quantum derivatives (right) [$\sigma = 0.90$]

Now in our context, in Figure 4.3, I show two legs of $\delta_n(s)$: one where $\chi_4(p_n) = +1$, and the other one where $\chi_4(p_n) = -1$. Both give rise to time-continuous functions that are nowhere differentiable, like the Brownian motions in Figure 4.5. Unlike Brownian motions, the variance tends to zero over time. The two functions are almost indistinguishable to the naked eye, so I separated them on the left plot in Figure 4.3. The corresponding quantum derivatives consist of a set of curves (right plot, same figure). They contain a lot of useful information about $L_4(s)$. In particular:

- The left plot in Figure 4.3 shows an asymmetrical distribution of the quantum derivatives around the X-axis. This is caused by the Chebyshev bias, also called prime race: among the first n primes numbers, the difference between the proportion of primes p_k with $\chi_4(p_k) = +1$, and those with $\chi_4(p_k) = -1$, is of the order $1/\sqrt{n}$, in favor of the latter. See [1, 32, 37]. This is known as Littlewood's oscillation theorem [22].
- The various branches in the quantum derivative (same plot) correspond to runs of different lengths in the sequence $\{\chi_4(p_n)\}$: shown as positive or negative depending on the sign of $\chi_4(p_n)$. Each branch has it own point density, asymptotically equal to $2^{-\lambda}$ (a geometric distribution) for the branch featuring runs of length λ, for $\lambda = 1, 2$ and so on. A similar number theory problem with the distribution of run lengths is discussed in section 4.3, for the binary digits of $\sqrt{2}$.

4.4.3 Project and solution

The project consists of checking many of the statements made in section 4.4.1, via computations. Proving the empirical results is beyond the scope of this work. The computations cover not only the case $L_4(s, n)$, but also other similar functions, including synthetic ones and Rademacher random multiplicative functions [23, 24, 25]. It also includes an empirical verification of theorem 4.4.1, and assessing whether its converse might be true. Finally, you will try different functions φ in formula (4.4) to check the impact on approximation (4.3). In the process, you will get familiar with a few Python libraries:

- MPmath to compute $L_4(s)$ for complex arguments when $\sigma < 1$.
- Primepy to obtain a large list of prime numbers.
- Scipy for curve fitting, when verifying the approximation (4.3).

The curve fitting step is considerably more elaborate than the standard implementation in typical machine learning projects. First, it consists of finding the best c, α, β in (4.3) for a fixed n, and identifying the best model: in this case, the choice of $\sqrt{n}$ and $\sqrt{n \log n}$ for the curve fitting function (best fit). Then, using different n, assess whether or not c, α, β, ρ depend on n or not, and whether $c = 0$ (this would suggest that the Euler product converges).

Finally, you will run the same curve fitting model for random functions, replacing the sequence $\{\chi_4(p_n)\}$ by random sequences of independent $+1$ and -1 evenly distributed, to mimic the behavior of $L_4(s, n)$ and Λ_n. One would expect a better fit when the functions are perfectly random, yet the opposite is true. A possible

explanation is the fact that the Chebyshev bias in $L_4(n, s)$ is very well taken care of by the choice of Λ_n, while for random functions, there is no such bias, and thus no correction.

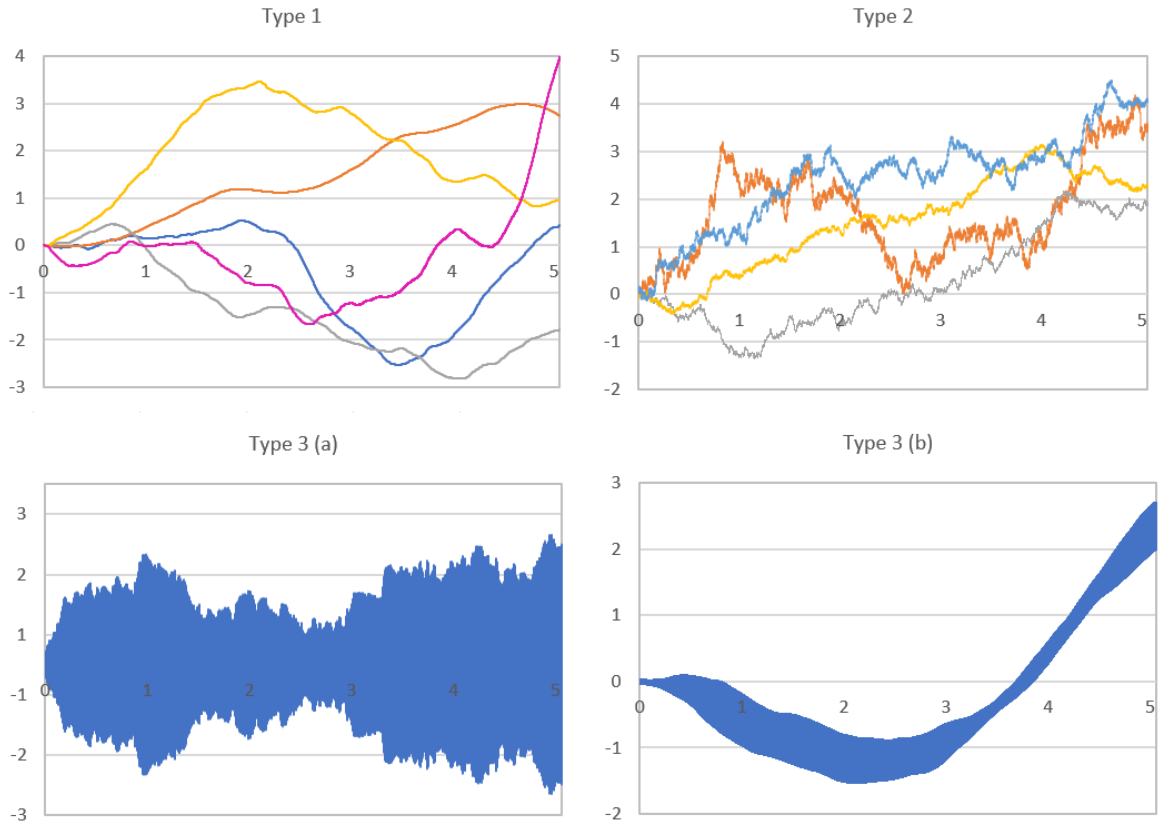


Figure 4.4: Integrated Brownian (top left), Brownian (top right) and quantum derivatives (bottom)

The project consists of the following steps:

Step 1: MPmath library, complex and prime numbers. Compute $L_4(s)$ using the MPmath library. See my code in section 4.4.4. The code is designed to handle complex numbers, even though in the end I only use the real part. Learn how to manipulate complex numbers by playing with the code. Also, see how to create a large list of prime numbers: look at how I use the PrimePy library. Finally, look at the Python `curve_fitting` and `r2_score` functions, to understand how it works, as you will have to use them. The former is from the Scipy library, and the latter (to measure goodness-of-fit) is from the Sklearn library.

Step 2: Curve fitting, part A. Implement Λ_n and the Euler Product $L_4(s, n)$ with n (the number of factors) up to 10^5. My code runs in a few seconds for $n \leq 10^5$. Beyond $n = 10^7$, a distributed architecture may help. In the code, you specify m rather than n, and $p_n \approx m/\log m$ is the largest prime smaller than m. For $s = \sigma + it$, choose $t = 0$, thus avoiding complex numbers, and various values of σ ranging from 0.55 to 1.75. The main step is the curve fitting procedure, similar to a linear regression but for non linear functions. The goal is to approximate $\delta_n(s)$, focusing for now on $\sigma = 0.90$.

The plots of $\delta_n(s)$ and Λ_n at the top in Figure 4.5, with $n = 80{,}000$, look very similar. It seems like there must be some $c_n(s)$ and a strictly positive $\rho_n(s)$ such that $\rho_n(s)\delta_k(s) - \Lambda_k \approx c_n(s)$ for $k = 1, 2, \ldots, n$. Indeed, the scaling factor

$$\rho_n(s) = \frac{\text{Stdev}[\Lambda_1, \ldots, \Lambda_n]}{\text{Stdev}[\delta_1(s), \ldots, \delta_n(s)]},$$

together with $c_n(s) = 0$, work remarkably well. The next step is to refine the linear approximation based on $\rho = \rho_n(s)$, using (4.3) combined with (4.5). This is where the curve fitting takes place; the parameters to estimate are c, α and β, with c close to zero. You can check the spectacular result of the fit, here with $\sigma = 0.90$ and $n \approx 1.25 \times 10^6$, on the bottom left plot in Figure 4.5. Dropping the $\sqrt{n \log n}$ term in (4.3) results in a noticeable drop in performance (test it). For $\rho_n(s)$, also try different options.

Step 3: Curve fitting, part B. Perform the same curve fitting as in Step 2, but this time for different values of n. Keep $s = \sigma + it$ with $t = 0$ and $\sigma = 0.90$. The results should be identical to those in Table 4.3,

where $\gamma_n = \sqrt{n} \cdot \Lambda_n$. The coefficients c, α, β and R^2 (the R-squared or quality of the fit) depend on n and s, explaining the notation in the table. Does $\rho_n(s)$ tend to a constant depending only on s, as $n \to \infty$? Or does it stay bounded? What about the other coefficients?

Now do the same with $\sigma = 0.70$ and $\sigma = 1.10$, again with various values of n. Based on your computations, do you think that $\rho_n(s)$ decreases to zero, stays flat, or increases to infinity, depending on whether $s = 0.70$, $s = 0.90$ or $s = 1.10$? If true, what are the potential implications?

Step 4: Comparison with synthetic functions. First, try $\varphi(n) = n \log n$ rather $\varphi(n) = n$, in (4.4). Show that the resulting curve fitting is not as good. Then, replace $\chi_4(p_k)$, both in $L_4(s, n)$ and Λ_n, by independent Rademacher distributions [Wiki], taking the values $+1$ and -1 with the same probability $\frac{1}{2}$. Show that again, the curve fitting is not as good, especially if $n \leq 10^5$. Then, you may even replace p_k (the k-th prime) by $k \log k$. The goal of these substitutions is to compare the results when χ_4 is replaced by synthetic functions that mimic the behavior of the Dirichlet character modulo 4. Also, you want to assess how much leeway you have in the choice of these functions, for the conclusions to stay valid.

The use of synthetic functions is part of a general approach known as generative AI. If all the results remain valid for such synthetic functions, then the theory developed so far is not dependent on special properties of prime numbers: we isolated that problem, opening the path to an easier proof that the Euler product $L_4(s, n)$ converges to $L_4(s)$ at some location $s = \sigma_0 + it$ with $\sigma_0 < 1$ inside the critical strip.

Step 5: Application outside number theory. Using various pairs of sequences $\{a_n\}$, $\{b_n\}$, empirically verify when the statistical theorem 4.4.1 might be correct, and when it might not.

The Python code in section 4.4.4 allows you to perform all the tasks except Step 5. In particular, for Step 4, set `mode='rn'` in the code. As for the curve fitting plot – the bottom left plot in Figure 4.5 – I multiplied both the target function $\delta_n(s)$ and the fitted curve by $\sqrt{n}$, here with $n = 1.25 \times 10^6$. Both tend to zero, but after multiplication by $\sqrt{n}$, they may or may not tend to a constant strictly above zero. Either way, it seems to indicate that the Euler product converges when $\sigma = 0.90$. What's more, the convergence looks strong, non-chaotic, and the second-order term involving $\sqrt{n \log n}$ in the approximation error, seems to be correct.

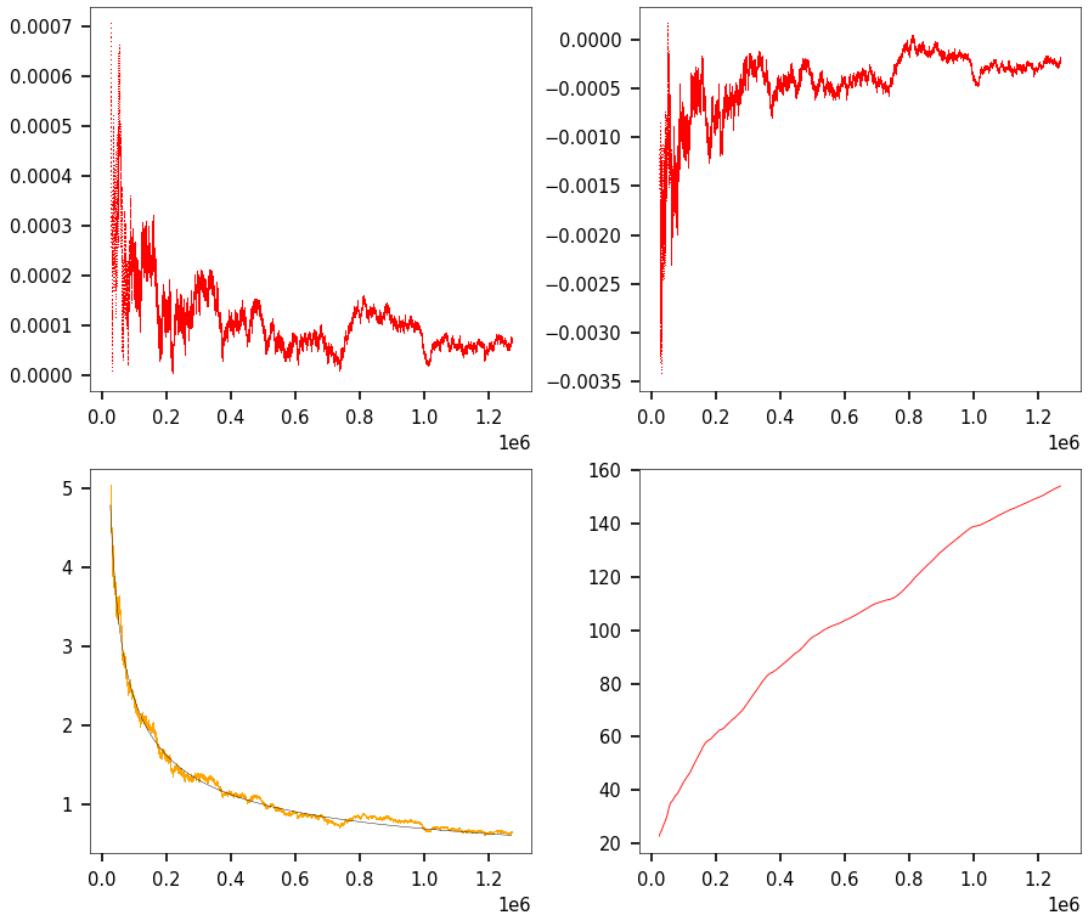


Figure 4.5: Top: δ_n (left), Λ_n (right); bottom: fitting δ_n (left), integrated δ_n (right) [$\sigma = 0.90$]

Regarding Step 3, Table 4.3 is the answer when $\sigma = 0.90$. It seems to indicate that $\rho_n(s)$ convergences (or is at least bounded and strictly above zero) when $\sigma = 0.90$ (remember that $s = \sigma + it$, with $t = 0$). With $\sigma = 0.70$, it seems that $\rho_n(s)$ decreases probably to zero, while with $\sigma = 1.10$, $\rho_n(s)$ is increasing without upper bound. The highest stability is around $\sigma = 0.90$. There, theorem 4.4.1 may apply, which would prove the convergence of the Euler product strictly inside to critical strip. As stated earlier, this would be a huge milestone if it can be proved, partially solving GRH not for $\zeta(s)$, but for the second most famous function of this nature, namely $L_4(s)$. By partial solution, I mean proving it for (say) $\sigma_0 = 0.90 < 1$, but not yet for $\sigma_0 = \frac{1}{2}$.

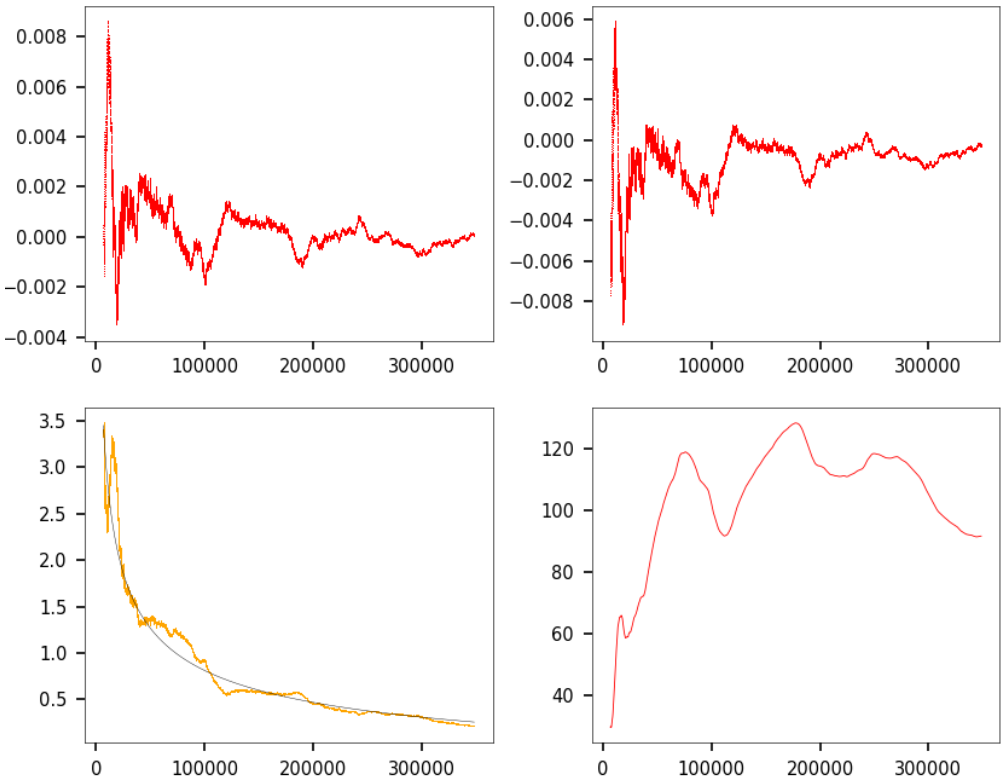


Figure 4.6: Same as Figure 4.5, replacing $\delta_n(s), \Lambda_n$ by synthetic functions

n	γ_n	$c_n(s)$	$\alpha_n(s)$	$\beta_n(s)$	$\rho_n(s)$	R^2
127,040	−0.27	0.00	0.41	0.94	5.122	0.940
254,101	−0.40	0.00	0.39	0.98	5.082	0.976
381,161	−0.35	0.00	0.37	1.03	5.151	0.986
508,222	−0.44	0.00	0.36	1.05	5.149	0.989
635,283	−0.32	0.00	0.36	1.04	5.085	0.989
762,343	−0.22	0.00	0.36	1.03	5.047	0.989
889,404	−0.10	0.00	0.35	1.06	5.123	0.990
1,016,464	−0.44	0.00	0.35	1.07	5.139	0.990
1,143,525	−0.30	0.00	0.34	1.08	5.121	0.991
1,270,586	−0.24	0.00	0.34	1.08	5.111	0.991

Table 4.3: One curve fitting per row, for $\delta_n(s)$ with $\sigma = 0.90$

Unexpectedly, Figure 4.6 shows that the fit is not as good when using a random sequence of $+1$ and -1, evenly distributed, to replace and mimic χ_4. The even distribution is required by the Dirichlet theorem, a generalization of the prime number theorem to arithmetic progressions [Wiki].

Finally, see the short code below as the answer to Step 5. The code is also on GitHub, here. The parameters p, q play a role similar to σ, and `r` represents ρ_n in theorem 4.4.1. The coefficient ρ_n may decrease to zero, increase to infinity, or converge depending on p and q. Nevertheless, in most cases when p, q are not too small, b_n converges. Applied to $L_4(s, n)$, it means that convergence may occur at s even if $\rho_n(s)$ does not converge.

The existence of some σ_1 for which $\rho_n(s)$ decreases to zero, and some σ_2 for which $\rho_n(s)$ increases to infinity, implies that there must be a σ_0 in the interval $[\sigma_1, \sigma_2]$, for which $\rho_n(s)$ converges or is bounded. This in turn implies that the Euler product $L_4(s, n)$ converges at $s = \sigma + it$ if $\sigma > \sigma_0$. The difficult step is to show that the largest σ_1 resulting in $\rho_n(s)$ decreasing to zero, is < 1. Then, $\sigma_0 < 1$, concluding the proof.

```
import numpy as np

N = 10000000
p = 1.00
q = 0.90
stdev = 0.50
seed = 564
np.random.seed(seed)
start = 20

u = 0
v = 0
a = np.zeros(N)
b = np.zeros(N)

for n in range(2, N):

   u += -0.5 + np.random.randint(0, 2)
   v += np.random.normal(0, stdev)/n**q
   a[n] = u / n**p
   b[n] = v

   if n % 50000 == 0:
      sa = np.std(a[start:n])
      sb = np.std(b[start:n])
      r = sa / sb
      c = r * b[n] - a[n]
      print("n = %7d r =%8.5f an =%8.5f bn =%8.5f c =%8.5f sa =%8.5f sb=%8.5f"
            %(n, r, a[n], b[n], c, sa, sb))
```

Important note. When dealing with the Euler product $L_4(s, n)$, the ratio $\rho_n(s)$ is rather stable (bounded strictly above zero, chaos-free, barely depending on n) and may even converge when $\sigma = 0.90$ and $t = 0$. Again, $s = \sigma + it$. Indeed, both the numerator and denominator appear well-behaved and seemingly chaos-free. Both of them tend to zero as n increases, at the same speed as $1/\sqrt{n}$. The chaos is in $L_4(s, n)$ and Λ_n. This fact can be leveraged to make progress towards proving the convergence of $L_4(s, n)$ at $\sigma = 0.90$. If not at $\sigma = 0.90$, there has to be at least one value $\sigma_0 < 1$ (close to 0.90) for which everything I just wrote, apply.

4.4.4 Python code

The implementation of the quantum derivatives is in section [4] in the following code. The coefficient $\rho_n(s)$ is denoted as `r`, while the parameter γ in table 4.3 is denoted as `mu`. In addition to addressing Step 1 to Step 4, the computation of the Dirichlet-L series and the Q_2 function are in section [2.1]. For Step 5, see the Python code at the end of section 4.4.3. Finally, to use synthetic functions rather than χ_4, set `mode='rn'`. The code is also on GitHub, here.

```
# DirichletL4_EulerProduct.py
# On WolframAlpha: DirichletL[4,2,s], s = sigma + it
#    returns Dirichlet L-function with character modulo k and index j.
#
# References:
#    https://www.maths.nottingham.ac.uk/plp/pmzcw/download/fnt_chap4.pdf
#    https://mpmath.org/doc/current/functions/zeta.html
#    f(s) = dirichlet(s, [0, 1, 0, -1]) in MPmath

import matplotlib.pyplot as plt
import matplotlib as mpl
import mpmath
import numpy as np
```

```
from primePy import primes
from scipy.optimize import curve_fit
from sklearn.metrics import r2_score
import warnings
warnings.filterwarnings("ignore")

#--- [1] create tables of prime numbers

m = 1000000 # primes up to m included in Euler product
aprimes = []

for k in range(m):
   if k % 100000 == 0:
      print("Creating prime table up to p <=", k)
   if primes.check(k) and k > 2:
      aprimes.append(k)


#--- [2] Euler product

#--- [2.1] Main function

def L4_Euler_prod(mode = 'L4', sigma = 1.00, t = 0.00):

   L4 = mpmath.dirichlet(complex(sigma,t), [0, 1, 0, -1])
   print("\nMPmath lib.: L4(%8.5f + %8.5f i) = %8.5f + %8.5f i"
        % (sigma, t,L4.real,L4.imag))

   prod = 1.0
   sum_chi4 = 0
   sum_delta = 0
   run_chi4 = 0
   old_chi4 = 0
   DLseries = 0
   flag = 1

   aprod = []
   adelta = []
   asum_delta = []
   achi4 = []
   arun_chi4 = []
   asum_chi4 = []

   x1 = []
   x2 = []
   error1 = []
   error2 = []
   seed = 116 # try 103, 105, 116 & start = 2000 (for mode = 'rn')
   np.random.seed(seed)
   eps = 0.000000001

   for k in range(len(aprimes)):

      if mode == 'L4':
         condition = (aprimes[k] % 4 == 1)
      elif mode == 'Q2':
         condition = (k % 2 == 0)
      elif mode == 'rn':
         condition = (np.random.uniform(0,1) < 0.5)

      if condition:
         chi4 = 1
      else:
         chi4 = -1

      sum_chi4 += chi4
```

```
        achi4.append(chi4)
        omega = 1.00 # try 1.00, sigma or 1.10
        # if omega > 1, asum_chi4[n] --> 0 as n --> infty
        # asum_chi4.append(sum_chi4/aprimes[k]**omega)
        asum_chi4.append(sum_chi4/(k+1)**omega)
        # asum_chi4.append(sum_chi4/(k+1)*(np.log(k+2)))

        if chi4 == old_chi4:
           run_chi4 += chi4
        else:
           run_chi4 = chi4
        old_chi4 = chi4
        arun_chi4.append(run_chi4)

        factor = 1 - chi4 * mpmath.power(aprimes[k], -complex(sigma,t))
        prod *= factor
        aprod.append(1/prod)

        term = mpmath.power(2*k+1, -complex(sigma,t))
        DLseries += flag*term
        flag = -flag

    limit = -eps + 1/prod # full finite product (approx. of the limit)
    if mode == 'L4':
        limit = L4 # use exact value instead (infinite product if it converges)

    for k in range(len(aprimes)):

        delta = (aprod[k] - limit).real # use real part
        adelta.append(delta)
        sum_delta += delta
        asum_delta.append(sum_delta)
        chi4 = achi4[k]

        if chi4 == 1:
           x1.append(k)
           error1.append(delta)
        elif chi4== -1:
           x2.append(k)
           error2.append(delta)

    print("Dirichlet L: DL(%8.5f + %8.5f i) = %8.5f + %8.5f i"
        % (sigma, t, DLseries.real, DLseries.imag))
    print("Euler Prod.: %s(%8.5f + %8.5f i) = %8.5f + %8.5f i\n"
        % (mode, sigma, t, limit.real, limit.imag))

    adelta = np.array(adelta)
    aprod = np.array(aprod)
    asum_chi4 = np.array(asum_chi4)
    asum_delta = np.array(asum_delta)
    error1 = np.array(error1)
    error2 = np.array(error2)

    return(limit.real, x1, x2, error1, error2, aprod, adelta, asum_delta,
           arun_chi4, asum_chi4)

#--- [2.2] Main part

mode = 'L4' # options: 'L4', 'Q2', 'rn' (random chi4)
(prod, x1, x2, error1, error2, aprod, adelta, asum_delta, arun_chi4,
    asum_chi4) = L4_Euler_prod(mode, sigma = 0.90, t = 0.00)


#--- [3] Plots (delta is Euler product, minus its limit)

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

```
plt.rcParams['xtick.labelsize'] = 7
plt.rcParams['ytick.labelsize'] = 7

#- [3.1] Plot delta and cumulated chi4

x = np.arange(0, len(aprod), 1)

# offset < len(aprimes), used to enhance visualizations
offset = int(0.02 * len(aprimes))

# y1 = aprod / prod
# plt.plot(x[offset:], y1[offset:], linewidth = 0.1)
# plt.show()

y2 = adelta
plt.subplot(2,2,1)
plt.plot(x[offset:], y2[offset:], marker=',', markersize=0.1,
  linestyle='None', c='red')

y3 = asum_chi4
plt.subplot(2,2,2)
plt.plot(x[offset:], y3[offset:], marker=',', markersize=0.1,
  linestyle='None', c='red')

#- [3.2] Denoising L4, curve fitting

def objective(x, a, b, c):

   # try c = 0 (actual limit)
   value = c + a/np.sqrt(x) + b/np.sqrt(x*np.log(x))
   return value

def model_fit(x, y2, y3, start, offset, n_max):

   for k in range(n_max):

      n = int(len(y2) * (k + 1) / n_max) - start
      stdn_y2 = np.std(y2[start:n])
      stdn_y3 = np.std(y3[start:n])
      r = stdn_y3 / stdn_y2

      # note: y3 / r ~ mu / sqrt(x) [chaotic part]
      mu = y3[n] * np.sqrt(n) # tend to a constant ?
      y4 = y2 * r - y3
      y4_fit = []
      err = -1

      if min(y4[start:]) > 0:
         popt, pcov = curve_fit(objective, x[start:n], y4[start:n],
                   p0=[1, 1, 0], maxfev=5000)
         [a, b, c] = popt
         y4_fit = objective(x, a, b, c)
         err = r2_score(y4[offset:], y4_fit[offset:])
         print("n = %7d mu =%6.2f c =%6.2f a =%5.2f b =%5.2f r =%6.3f err =%6.3f"
                  %(n, mu, c, a, b, r, err))

   return(y4, y4_fit, err, n)

n_max = 10 # testing n_max values of n, equally spaced
start = 20 # use Euler products with at least 'start' factors
if mode == 'rn':
   start = 1000
if start > 0.5 * offset:
   print("Warning: 'start' reduced to 0.5 * offset")
   start = int(0.5 * offset)
(y4, y4_fit, err, n) = model_fit(x, y2, y3, start, offset, n_max)
```

```
ns = np.sqrt(n)

if err != -1:
   plt.subplot(2,2,3)
   plt.plot(x[offset:], ns*y4[offset:], marker=',', markersize=0.1,
     linestyle='None', c='orange')
   plt.plot(x[offset:], ns*y4_fit[offset:], linewidth = 0.2, c='black')
else:
   print("Can't fit: some y4 <= 0 (try different seed or increase 'start')")

#--- [3.3] Plot integral of delta

y5 = asum_delta
plt.subplot(2,2,4)
plt.plot(x[offset:], y5[offset:], linewidth = 0.4, c='red')
plt.show()


#--- [4] Quantum derivative

#- [4.1] Function to differentiated: delta, here broken down into 2 legs

plt.subplot(1,2,1)
shift = 0.001
plt.plot(x1[offset:], error1[offset:], marker=',', markersize=0.1,
  linestyle='None', alpha = 1.0, c='red')
plt.plot(x2[offset:], shift + error2[offset:], marker=',', markersize=0.1,
  linestyle='None', alpha = 0.2, c='orange')

#- [4.2] Quantum derivative

def d_error(arr_error):

   diff_error = [] # discrete derivative of the error
   positives = 0
   negatives = 0
   for k in range(len(arr_error)-1):
      diff_error.append(arr_error[k+1] - arr_error[k])
      if arr_error[k+1] - arr_error[k] > 0:
         positives +=1
      else:
         negatives += 1
   return(diff_error, positives, negatives)

(diff_error1, positives1, negatives1) = d_error(error1)
(diff_error2, positives2, negatives2) = d_error(error2)
ymin = 0.5 * float(min(min(diff_error1[offset:]), min(diff_error1[offset:])))
ymax = 0.5 * float(max(max(diff_error1[offset:]), max(diff_error2[offset:])))

plt.subplot(1,2,2)
plt.ylim(ymin, ymax)
plt.plot(x1[offset:len(x1)-1], diff_error1[offset:len(x1)-1], marker=',', markersize=0.1,
       linestyle='None', alpha=0.8, c = 'red')
plt.plot(x2[offset:len(x2)-1], diff_error2[offset:len(x2)-1], marker=',', markersize=0.1,
       linestyle='None', alpha=0.8, c = 'orange')
plt.show()

print("\nError 1: positives1: %8d negatives1: %8d" % (positives1, negatives1))
print("Error 2: positives2: %8d negatives2: %8d" % (positives2, negatives2))
```

Chapter 5

Generative AI

The projects in this chapter are related to various aspects of generative AI, such as data synthetization (tabular data), agent-based modeling and model-based generative AI relying on stochastic systems. Emphasis is on assessing the quality of the generated data and reducing the time required to train the underlying algorithms such as GAN (generative adversarial network). For time series generation and geospatial applications, see chapter 3. For LLMs (large language models), see chapter 7. Computer vision, graph and even sound generation, are included in this chapter.

The goal of data synthetization is to produce artificial data that mimics the patterns and features present in existing, real data. Many generation methods and evaluation techniques are available, depending on purposes, the type of data, and the application field. Everyone is familiar with synthetic images in the context of computer vision, or synthetic text in applications such as GPT. Sound, graphs, shapes, mathematical functions, artwork, videos, time series, spatial phenomena — you name it — can be synthesized. In this article, I focus on tabular data, with applications in fintech, the insurance industry, supply chain, and health care, to name a few.

The word "synthetization" has its origins in drug synthesis, or possibly music. Interestingly, the creation of new molecules also benefits from data synthetization, by producing virtual compounds, whose properties (if they could be produced in the real world) are known in advance to some degree. It also involves tabular data generation, where the features replicated are various measurements related to the molecules in question. Historically, data synthetization was invented to address the issue of missing data, that is, as a data imputation technique. It did not work as expected as missing data is usually very different from observed values. But the technique has evolved to cover many applications.

You can synthesize data using interpolation, agent-based modeling, adding correlated zero-mean noise to the real data, using copulas or generative adversarial networks (GANs). All these techniques are discussed in details in my book on Generative AI [20]. For time series synthetization via interpolation, see project 3.1.

5.1 Holdout method to evaluate synthetizations

The goal is to compare synthetic data vendors, in the context of tabular or transactional data generation. Using real datasets, run synthetizations through various vendors and evaluate the quality of the results: how well the real data is reproduced. In order words, the goal is to assess the faithfulness of the synthetizations, using a sound methodology.

The holdout method consists of using 50% of the real data to train the synthesizer, and using the remaing 50% to assess the quality of the synthesized data. It is similar to cross-validation, but applied in the context of synthetic data, where a response or dependent variable may or may not be present. There are many metrics to compare two datasets, in this case real versus synthetic data. Here, I focus on the following metrics:

- The correlation distance matrix Δ. This symmetric $m \times m$ matrix is defined as $\Delta_{ij} = |R_{ij} - S_{ij}|$, where R, S are the correlation matrices respectively measured on the real and synthetic data. Here m is the number of features, including the response or outcome if there is one. The indices i, j represent two features. In particular Δ_{avg} and Δ_{max} are the average and maximum value of Δ. These values are always between 0 (best match) and 1 (worst match).
- The Kolmogorov-Smirnov distance vector K. This vector has m components, one for each feature. Each component represents the normalized distance between two empirical distributions, for the corresponding feature: one attached to the synthetic data, and the other one to the real data. In particular K_{avg} and K_{max} are the average and maximum value of K. These values are always between 0 (best match) and 1 (worst match).

- Additional metrics capturing non-linear inter-dependencies among features, and how well these non-linear patterns are reproduced in the synthetic data. I use scatterplots such as those in Figures 5.1 and 5.2 to show the match (or lack of) between real and synthetic data. These metrics are important, as correlations alone focus on linear dependencies only, and Komogorov-Smirnov are one-dimensional summaries and do not take into account the feature dependencies.

For details and to get started, read my article "Generative AI: Synthetic Data Vendor Comparison and Benchmarking Best Practices" [13], available here. Download the technical document, and look at the Jupyter notebook referenced in the article in question.

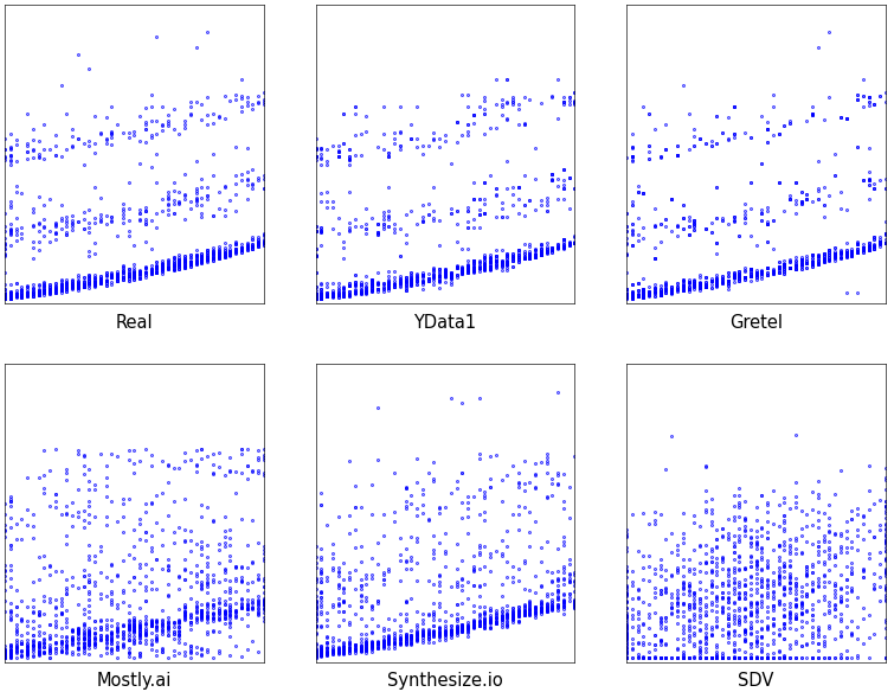


Figure 5.1: Insurance data scatterplot, age (X-axis) versus charges (Y-axis)

My article features 3 datasets and several vendors, as well as a case study with the holdout method: the insurance dataset synthesized with YData.ai. We will start with the insurance dataset. As for the vendors, at the time of writing, Gretel and Mostly.ai offer web APIs. Synthesizing data is as easy as uploading a csv file (the real data), run it through the API, and download the result. Alternatively you can use the YData platform (Fabric). You can run the synthetization with 3 lines of Python code provided in their documentation. You can also use the open source SDV library (synthetic data vault) or my own home-made synthesizers described in chapter 10 in my book on generative AI [20].

5.1.1 Project description

The project consists of the following steps:

Step 1: Use the insurance dataset `incurance.csv` available here. Produce an histogram for each of the non-categorical features: age, charges and bmi (body mass index). The dataset is described in chapter 10 in my book on generative AI [20]. Does it require data cleaning or preprocessing such as transformations to normalize the data?

Step 2: Use the Python code `insurance_compare.py` on my GitHub repository here, to perform evaluations and vendor comparisons. The Δ and K metrics are implemented in this script, as well as histograms and scatterplots.

Step 3: Produce synthetic data using one of the vendors. Update the `insurance_compare.csv` file found in the same GitHub folder, once downloaded in your local environment, by adding the generated data. Proceed as follows: use 50% of the insurance data set (real data) to produce the synthetization, and use the other 50% (the validation set) to compare with the synthesized data. Produce a file similar to `insurance_compare_holdout.csv` (in the same GitHub folder) but for a vendor other than YData.ai, as I already did the analysis for this one.

Step 4: The first 50% of the real data is called the training set. Compare the training set with the validation set, and the synthesized data with the validation set. It is assumed that these three datasets have the same number of observations. Do you observe a loss of quality in the syntheric data, when using the holdout method just described, compared to using the full real data (without validation set)?

Step 5: Run two synthetizations from the same vendor: in other words, produce two synthetic datasets based on the same real data. For each synthetization, use the holdout method to evaluate the quality. The goal is to evaluate not only the difference between a real dataset and its synthetization, but also between two synthetizations of the same dataset. Are differences between two synthetizations of a same dataset larger or smaller than between a real dataset and its synthetization?

The holdout method is used to verify that vendors are not using artifacts to boost performance. If this was the case, it would result in overfitting with very good performance measured against the training data, but poor performance when measured against the validation set. Indeed, the actual performance should always be assessed by comparison with the validation set, not with the data used to train the synthesizer.

5.1.2 Solution

The files `insurance_compare_holdout.csv` (input data) and `insurance_compare_holdout.py` illustrate the holdout method on YData.ai. More details are available in one of my articles posted here: see section 3.2. in the same GitHub repository. The Jupyter notebook `sd_vendors.ipynb` available here illustrates how to compute the evaluation metrics Δ, K and produce the various plots such as Figure 5.2 (scatterplots for the circle data) and Figure 5.3 (histogram for the insurance dataset). For convenience, I also included the Python code in this section.

```
import pandas as pd
import numpy as np
import scipy
from scipy.stats import ks_2samp
from statsmodels.distributions.empirical_distribution import ECDF

dataset = 'insurance_compare.csv'
url = "https://raw.githubusercontent.com/VincentGranville/Main/main/" + dataset
df = pd.read_csv(url)
# df = pd.read_csv(dataset)
if dataset == 'insurance_compare.csv':
   df = df.drop('region', axis=1)
   df = df.dropna(axis='columns')
print(df.head())

data_real = df.loc[df['Data'] == 'Real']
data_real = data_real.drop('Data', axis=1)
data_real = data_real.to_numpy()
print(data_real)

r_corr = np.corrcoef(data_real.T) # need to transpose the data to make sense
print(r_corr)

ltests = df.Data.unique().tolist()
popped_item = ltests.pop(0) # remove real data from the tests
print(ltests)

#--- main loop

for test in ltests:

   data_test = df.loc[df['Data'] == test]
   data_test = data_test.drop('Data', axis=1)
   data_test = data_test.to_numpy()
   t_corr = np.corrcoef(data_test.T)
   delta = np.abs(t_corr - r_corr)
   dim = delta.shape[0] # number of features

   ks = np.zeros(dim)
   out_of_range = 0
```

```
    for idx in range(dim):
        dr = data_real[:,idx]
        dt = data_test[:,idx]
        stats = ks_2samp(dr, dt)
        ks[idx] = stats.statistic
        if np.min(dt) < np.min(dr) or np.max(dt) > np.max(dr):
            out_of_range = 1
    str = "%20s %14s %8.6f %8.6f %8.6f %8.6f %1d" % (dataset, test, np.mean(delta),
            np.max(delta), np.mean(ks), np.max(ks), out_of_range)
    print(str)

#--- visualizing results

def vg_scatter(df, test, counter):

    # customized plots, insurance data
    # one of 6 plots, subplot position based on counter

    data_plot = df.loc[df['Data'] == test]
    x = data_plot[['age']].to_numpy()
    y = data_plot[['charges']].to_numpy()
    plt.subplot(2, 3, counter)
    plt.scatter(x, y, s = 0.1, c ="blue")
    plt.xlabel(test, fontsize = 7)
    plt.xticks([])
    plt.yticks([])
    plt.ylim(0,70000)
    plt.xlim(18,64)
    return()


def vg_histo(df, test, counter):

    # customized plots, insurance data
    # one of 6 plots, subplot position based on counter

    data_plot = df.loc[df['Data'] == test]
    y = data_plot[['charges']].to_numpy()
    plt.subplot(2, 3, counter)
    binBoundaries = np.linspace(0, 70000, 30)
    plt.hist(y, bins=binBoundaries, color='white', align='mid',edgecolor='red',
            linewidth = 0.3)
    plt.xlabel(test, fontsize = 7)
    plt.xticks([])
    plt.yticks([])
    plt.xlim(0,70000)
    plt.ylim(0, 250)
    return()

import matplotlib.pyplot as plt
import matplotlib as mpl
mpl.rcParams['axes.linewidth'] = 0.3

vg_scatter(df, 'Real', 1)
vg_scatter(df, 'YData1', 2)
vg_scatter(df, 'Gretel', 3)
vg_scatter(df, 'Mostly.ai', 4)
vg_scatter(df, 'Synthesize.io', 5)
vg_scatter(df, 'SDV', 6)
plt.show()

vg_histo(df, 'Real', 1)
vg_histo(df, 'YData1', 2)
vg_histo(df, 'Gretel', 3)
vg_histo(df, 'Mostly.ai', 4)
vg_histo(df, 'Synthesize.io', 5)
```

```
vg_histo(df, 'SDV', 6)
plt.show()
```

Table 5.1 compares real with synthetic data using holdout. Each dataset (a row in the table) is compared with the validation set. Thus the row "Validation" is filled with zeros (the best possible fit) as you compare the validation set with itself.

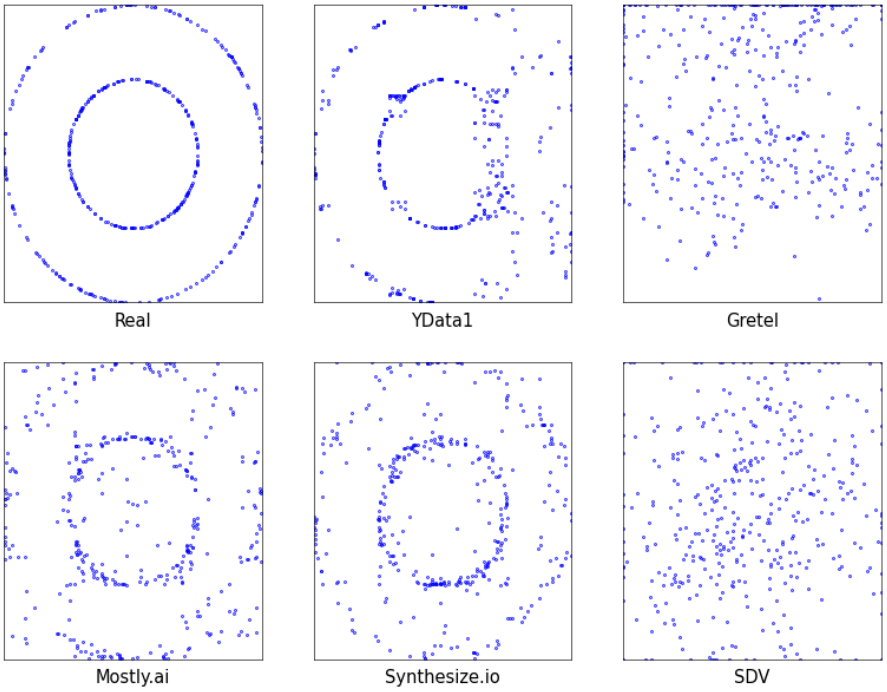


Figure 5.2: Circle data scatterplot, first two coordinates

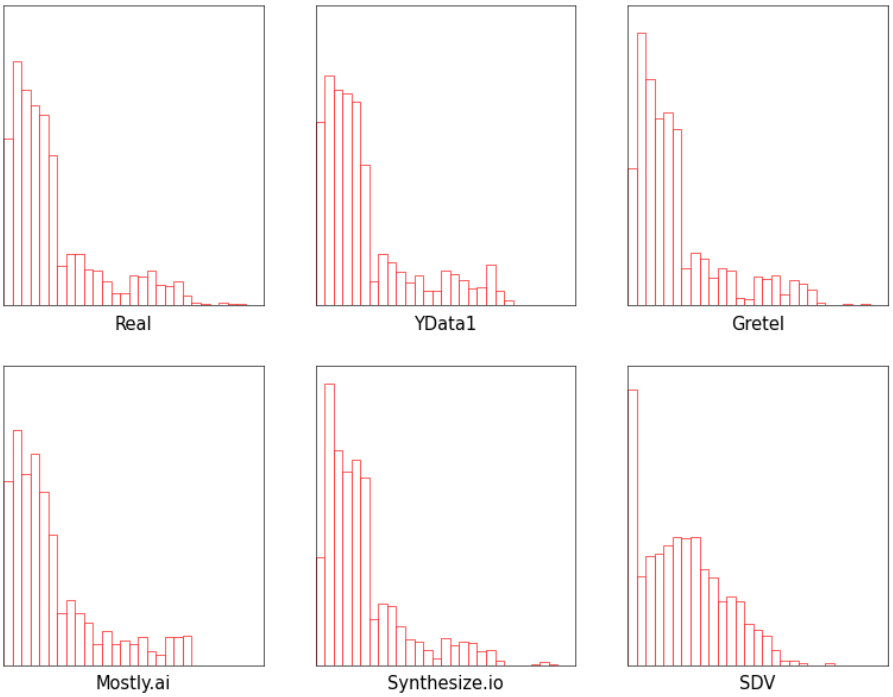


Figure 5.3: Insurance data, charges distribution, real (top left) vs synthetic

Besides the quality metrics investigated in this project, there are other ways to compare vendors. For instance, how long it takes to train a synthesizer, particularly GAN (generative adversarial networks) which is a combination of deep neural networks. Methods to improve the speed are discussed in project 5.2. The ease

of use is also an important factor. For instance, is a Pyhon SDK available? Can you run it on the vendor's platform without interfering with your own environment, and does it require a lot of parameter fine-tuning?

Other factors to consider is replicability and being able to sample outside the observation range. At the time of this writing, none of the vendors offer these features. However, my home-made GAN does, see chapter 10 in my book on generative AI [20].

Data	Type	Δ_{avg}	Δ_{max}	K_{avg}	K_{max}
Training	Real	0.0314	0.0687	0.0361	0.0732
Validation	Real	0.0000	0.0000	0.0000	0.0000
YData1	Synthetic	0.0394	0.1081	0.0433	0.0792
YData2	Synthetic	0.0241	0.0822	0.0386	0.0807

Table 5.1: Insurance data, distances to validation set

5.2 Enhanced synthetizations with GANs and copulas

The goal is to synthesize a real dataset, understand the steps involved (data transformation if necessary, synthesizing, assessing the quality of the results) and then use tools that automate this process. Along the way, we investigate various enhancements to the base technique, and the value that they bring, especially in terms of speeding up the training of the deep neural networks involved, reducing the amount of data needed, replicability, sampling outside the observation range and so on. Two types of synthesizers are considered: GAN and copula-based, in the context of tabular data generation.

5.2.1 Project description

The material about GAN (generative adversarial network) and copula is discussed in detail in chapter 10 in my book on generative AI [20]. This project also features techniques described in separate papers. It covers feature clustering to reduce the dimensionality of the problem, smart grid search for hyperparameter tuning, stopping rules for GAN to speed up training, parametric copulas and alternative to GMM (Gaussian mixture models), customizing the loss function, dealing with categorical features, as well as fast data distillation to reduce the size of the training set. Two important issues are replicability and the ability to generate synthetic data outside the observation range.

This project is rather large, and can easily be broken down into sub-projects. Participants enrolled in the Generative AI certification program may choose to work on two of the steps below. I use the insurance and diabetes datasets in this project. They are available on GitHub, here. Look for `diabetes_clean.csv` and `insurance.csv`.

The project consists of the following steps:

Step 1: **Data transformation**. Use a synthetization tool from one of the vendors listed in project 5.1, or my GAN synthesizer `GAN_diabetes.py` available here and explained in my book [20]. Apply the technique to the diabetes dataset. My GAN is not doing great on the diabetes dataset. Decorrelate the data using PCA (principal component analysis), then synthesize the transformed data, then apply the inverse PCA transform to the synthetic data, to reproduce the correct correlation structure. Measure the improvement over the base synthetization, using the correlation distance metric Δ_{avg} described in section 5.1. See Stack Exchange question on this topic, here. Section 7.1 in my book [20] offers a different perspective on the subject, using square roots of covariance matrices, without advanced matrix algebra.

Step 2: **Data augmentation**. Synthesize the `diabetes_clean.csv` dataset as in step 1. One of the fields named "Outcome" is the binary response: the patient either has cancer, or not. The other features are predictors (called independent variables by statisticians) such as glucose level, body mass index, or age. The goal is to predict the risk of cancer based on the values attached to the predictors. Train a predictive classifier such as logistic regression to predict cancer outcome, on the real data. Then, train the same classifier on augmented data, consisting of 50% of the real data, the other 50% being synthetic data. Assess correct classification rate on the unused portion of the training set, called validation set. Finally, compare the results if doing the same test but without adding synthetic data to the training set.

Step 3: **Data grouping**. Use the Gaussian copula method described in chapter 10 in my book [20] and in my Jupyter notebook here, to synthesize the insurance dataset. Ignore the categorical features "gender", "smoking status" and "region". Instead, for each multivariate bucket, use a separate copula. Then, assess

the improvement over using the same copula across all buckets. An example of bucket, also called bin or flag vector, is `[gender=male,smoking=yes,region=Northeast]`. The bucketization process is performed manually here. But it could be automated using XGboost or similar ensemble methods based on many decision trees, such as my Hidden Decision Trees technique described in chapter 2 in my book [20]. Buckets that are too small (less than 30 observations) can be aggregated into larger buckets or treated separately.

Step 4: **Data distillation**. Can randomly removing 50% of the observations in the training set (real data) speed up the training process by a factor two? Implement this strategy, and evaluate the results, both in term of algorithmic speed and the quality of the generated data. It is possible to be highly selective in the choice of observations to be deleted, for instance to boost the quality of the synthesized data at each deletion. However, this method, known as data distillation [39], can be time consuming and erase all gains in the training process. See also step 7 for stopping the training process when the loss function has stabilized to a low, rather than using a fixed number of epochs.

Step 5: **Feature clustering**. This method is an alternative to principal component analysis. It does not transform the features into meaningless, artificial variables. Thus, it belongs to a set of techniques known as explainable AI. It consists of putting all the highly correlated features into a number of clusters, and creating individual clusters for features barely correlated to any other ones. How woulsd you use this technique on the diabetes data set? While clustering the observations is based on the distance (Euclidean, cosine and so on) between any two observations, clustering features is based on the absolute value of the correlation between any two features. Features barely correlated to any other one can be synthesized separately rather than jointly, thus saving time in the training process, via parallel computing.

Step 6: **Parameter fine-tuning**. Implement the smart grid search algorithm described in [18], to fine-tune the learning rate hyperparameter in `GAN_diabetes.py`. In the end, GANs are gradient descent algorithms to minimize a loss function, and the learning rate applies to the gradient method being used (in my GAN and many other implementations, ADAM is the preferred gradient method).

Step 7: **Loss function and stopping rule**. In some gradient descent implementations such as GANs or linear regression, the loss function is the mean squared or mean absolute error. GANs actually consist of two different neural networks with opposite goals: the generator to synthesize data, and the discriminator to check how close real and synthetic data are to each other. Thus two loss functions are needed, and can be blended into a single one: for details, see here. It is possible to use a customized loss function for each model (generator and discriminator) when calling the `mode.compile` Keras function: see example in `GAN_diabetes.py`.

Another, possibly easier strategy, consists of computing the distance between the real and synthesized data, (say) every 50 epochs, and stop when it is low enough. This is possible in my GAN by setting `mode='Enhanced'`. For instance, I use the distance (or loss function) $L = \Delta_{\text{avg}}$ defined in section 5.1, and implemented as the `GAN_distance` function in the Python code. Modify the code to use $L = \Delta_{\text{avg}} + K_{\text{avg}}$ instead. The first is the average correlation distance, and the second is the average Kolmogorov-Smirnov distance. Computation examples can be found in `insurance_compare.py`, in the same GitHub folder here, and in the `sd_vendors.ipynb` notebook, here.

Step 8: **Synthesizing outliers**. Most of the vendors do not produce synthetizations outside the observation range: for instance, if "age" ranges from 18 to 64 in your real dataset, it will be the same in the synthesized version. My copula method based on empirical quantiles has the same problem. In some contexts, it is important to sample outside the observation range, for instance when testing new algorithms. How would you do to achieve this goal?

Step 9: **Sensitivity analysis**. Add noise to the real data. Evaluate how sensitive your synthetization is to changes in the real data, depending on the amount of noise. Fine-tune parameters in the synthesizer to see how to reduce sensitivity. Identify synthesizers most robust against noise.

Another interesting question is whether your synthesizer leads to replicable results. Those based on neural networks generally don't, and this includes all the solutions offered by the vendors tested. My home-made GAN does: two runs with the same `seed` parameter lead to the same synthetizations. This fact can be leveraged to get better synthetizations, by trying different seeds. Related to this, a potential project consists of testing the variations between two different runs from the same synthesizer on the same data. Are these differences bigger than the average discrepancy between a synthetization and the real data? See Table 5.1 for an answer, based on testing the YData.ai platform. The answer is usually negative.

On a different topic, Figure 5.4 illustrates the evolution of the loss function discussed in step 7, over 10,000 epochs. Note that there are two loss functions: one for the generator (in orange), and one for the discriminator

model (in blue). The two plots correspond to using two different seeds: 103 (top) and 102 (bottom). Clearly, seed 103 is the winner, as it leads to a lower value of the loss function over time, and thus to a stronger local optimum. It is also obvious that we could have stopped training the GAN after about 6000 epochs.

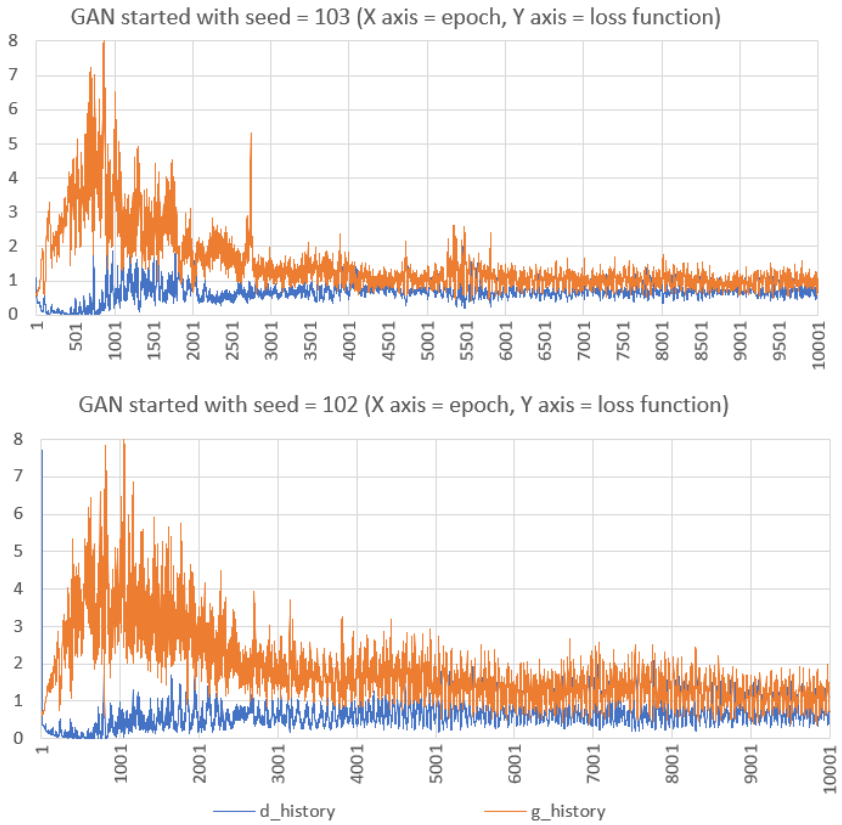


Figure 5.4: Loss function history, two versions of GAN

Finally, one way to accelerate GAN training is to use a fast version of the gradient descent algorithm, such as lightGBM [Wiki]. This is implemented in TabGAN [Wiki], as well as in the light version of SDV (the synthetic data vault library). It may result in a noticeable drop in quality.

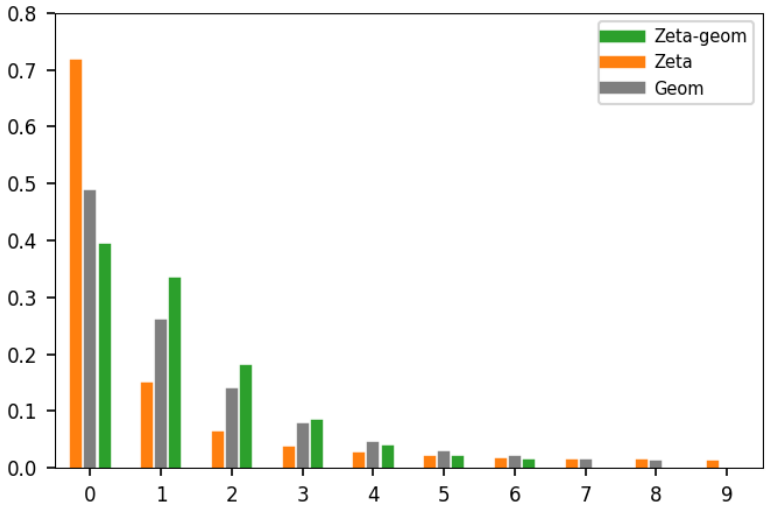


Figure 5.5: Modeling the number of children, insurance data (green is best fit)

5.2.2 Solution

For convenience, this section contains the Python code for the Gaussian copula method based on grouping (step 3), my home-made GAN method (most useful to work on step 7), and the smart grid search related to step 6 and step 8. The choice of the copula – Gaussian or not – has nothing to do with the observed distributions

in the real data: most features are not Gaussian in my examples, some are multimodal and not symmetric. However, non-Gaussian copulas are sometimes preferred when dealing with very thick tails: the reader is invited to check out articles on this topic [21]. Gumbel, Vine, Frank and other copulas are available in Python libraries such as SDV or Copula. See code here. Likewise, the choice of a Gaussian distribution for latent variables in GAN is unimportant, though uniform distributions might be more GAN-friendly.

Copulas methods based on empirical quantiles do not allow you to generate data outside the observation range. This includes my own version. To fix this issue, replace the empirical quantiles by those of a parametric distribution that fits the real data well. The parameters are estimated on the real data. I explain how this works in my article about smart grid search [18]. See Figure 5.5 with the "number of children" – one of the features in the insurance dataset – modeled using a two-parameter zeta-geometric distribution. Univariate and multivariate Gaussian mixture models (GMM) are popular in this context when dealing with continuous variables. Parameters are then estimated via the EM algorithm, and the resulting synthetizations are not limited to the observation range, thus answering the question in step 8. Hierarchical Bayesian models are a generalization of GMM. Adding noise is the easiest way to sample outside the observation range. It is discussed in chapter 7 in my book [20].

The answer to step 5 can be found in my article on feature clustering [12]. I did not include the code here, but you can find two different implementations in section 10.4.6 in my book [20], and here. The first version uses the Scipy library for hierarchical clustering; the second one is based on detecting connected components, a graph theory algorithm. Figure 5.6 shows the 9 features of the diabetes dataset on the X-axis; the Y-axis represents the distance between two feature clusters, measured as $1 - |\rho|$ where ρ is the correlation between features from two different clusters. Clearly, Family history and blood pressure (features 6 and 2 respectively) are the least correlated to other features, and can be treated separately.

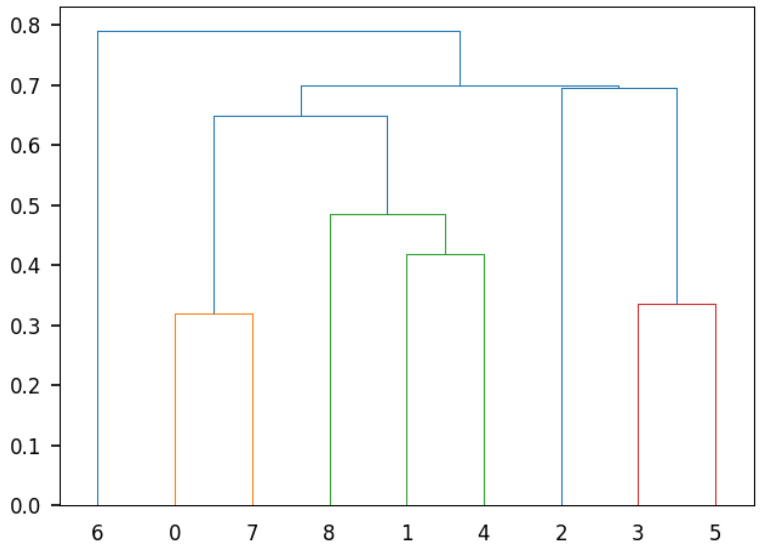


Figure 5.6: Feature clustering on the diabetes dataset

Table 5.2 lists the features (label with description) for the diabetes dataset. Figure 5.7 shows the correlation matrix. Regarding step 4, see my article on stochastic thinning [17].

Finally, regarding step 2, see my code `GAN_diabetes.py` on GitHub, here, also dealing with the same dataset. The feature "Outcome" (cancer status) is the response, and the goal is to predict the risk of cancer given the other features. I use the random forest classifier to perform supervised classification into the two groups (cancer versus no cancer), by calling the `RandomForestClassifier` ensemble method from the Sklearn Python library. I first do it on the real data before training the GAN model, and then again at the end, but this time on the synthetic data for comparison purposes. To complete this step, blend real with synthetic data, then run the random forest classifier on this augmented data, then evaluate the results on a validation set. This latter set with known outcome, part of the real data and also called holdout, is not used to train the random forest classifier, but to evaluate the results. You can generate it with the function `train_test_split` available in Sklearn, as illustrated in the Python code.

5.2.3 Python code

In this section, you will find the Python code for the Gaussian copula method with a separate copula for each group, the GAN method, and the smart grid search algorithm with an application to optimize parameters. The

application in question consists of estimating the parameters in a a 2-parameter distribution to fit the "number of children" feature in the insurance dataset, in the context of parametric copulas. The code and datasets are also available on GitHub: see each subsection for the link to the GitHub locations. For examples using the SDV open source library, including how to handle metadata, see my code snippet `SDV_example` on GitHub, here.

Code	Feature name	Description
0	Pregnancies	Number of pregnancies
1	Glucose	Plasma glucose concentration
2	BloodPressure	Diastolic blood pressure in mm/Hg
3	SkinThickness	Triceps skinfold thickness in mm
4	Insulin	Insulin in U/mL
5	BMI	Body mass index
6	DiabetesPedigreeFunction	Risk based on family history
7	Age	Age of patient
8	Outcome	Patient had diabetes or not

Table 5.2: Feature mapping table for the diabetes dataset

	Pregnanci	Glucose	BloodPres	SkinThickr	Insulin	BMI	DiabetesP	Age	Outcome
Pregnanci	1.00	0.20	0.21	0.09	0.08	-0.03	0.01	0.68	0.26
Glucose	0.20	1.00	0.21	0.20	0.58	0.21	0.14	0.34	0.52
BloodPres	0.21	0.21	1.00	0.23	0.10	0.30	-0.02	0.30	0.19
SkinThickr	0.09	0.20	0.23	1.00	0.18	0.66	0.16	0.17	0.26
Insulin	0.08	0.58	0.10	0.18	1.00	0.23	0.14	0.22	0.30
BMI	-0.03	0.21	0.30	0.66	0.23	1.00	0.16	0.07	0.27
DiabetesP	0.01	0.14	-0.02	0.16	0.14	0.16	1.00	0.09	0.21
Age	0.68	0.34	0.30	0.17	0.22	0.07	0.09	1.00	0.35
Outcome	0.26	0.52	0.19	0.26	0.30	0.27	0.21	0.35	1.00

Figure 5.7: Correlation matrix for the diabetes dataset

5.2.3.1 Gaussian copula by group

This program generates synthetic observations for the insurance dataset, using a different copula for each group. Groups are defined in step 3 in project 5.2. A detailed description of the code is in my Jupyter notebook, here, as well as in chapter 10 in my book [20].

```
import pandas as pd
from scipy.stats import norm
import numpy as np

# source: https://www.kaggle.com/datasets/teertha/ushealthinsurancedataset
# Fields: age, sex, bmi, children, smoker, region, charges

url="https://raw.githubusercontent.com/VincentGranville/Main/main/insurance.csv"
# make sure fields don't contain commas
data = pd.read_csv(url)
print(data.head(10))

groupCount = {}
groupReal = {}
for k in range(0, len(data)):
   obs = data.iloc[k] # get observation number k
   group = obs[1] +"\t"+obs[4]+"\t"+obs[5]
   if group in groupCount:
      cnt = groupCount[group]
      groupReal[(group,cnt)]=(obs[0],obs[2],obs[3],obs[6])
      groupCount[group] += 1
```

```
      else:
         groupReal[(group,0)]=(obs[0],obs[2],obs[3],obs[6])
         groupCount[group] = 1

for group in groupCount:
   print(group, groupCount[group])

print(groupReal[("female\tyes\tsouthwest",0)])
print(groupReal[("female\tyes\tsouthwest",1)])
print(groupReal[("female\tyes\tsouthwest",2)])
print(groupReal[("female\tyes\tsouthwest",3)])
print(groupReal[("female\tyes\tsouthwest",20)])

def create_table(group, groupCount, groupReal):

   # extract data corresponding to specific group, from big table groupReal

   nobs = groupCount[group]
   age = []
   bmi = []
   children = []
   charges = []
   for cnt in range(nobs):
      features = groupReal[(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
      real = np.stack((age, bmi, children, charges), axis = 0)
   return(real)

def gaussian_to_synth(real, gfg, group, nobs_synth, groupSynth):

   # turn multivariate gaussian gfg into synth. data, update groupSynth
   # this is done for a specific group, creating nobs_synth obs.

   age = real[0,:]
   bmi = real[1,:]
   children = real[2,:]
   charges = real[3,:]

   g_age = gfg[:,0]
   g_bmi = gfg[:,1]
   g_children = gfg[:,2]
   g_charges = gfg[:,3]

   for k in range(nobs_synth):

      u_age = norm.cdf(g_age[k])          # u stands for uniform[0, 1]
      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

      # add k-th synth. obs. for group in question, to groupSynth
      groupSynth[(group, k)] = [s_age, s_bmi, s_children, s_charges]

   return()

seed = 453
np.random.seed(seed)
groupSynth = {}
```

```
for group in groupCount:

   real = create_table(group, groupCount, groupReal)
   n_var = real.shape[0]
   zero = np.zeros(n_var)
   corr = np.corrcoef(real) # correlation matrix for Gaussian copula for this group
   nobs_synth = groupCount[group] # number of synthetic obs to create for this group
   gfg = np.random.multivariate_normal(zero, corr, nobs_synth)
   gaussian_to_synth(real, gfg, group, nobs_synth, groupSynth)


for group, k in groupSynth:

   obs = groupSynth[(group,k)] # this is k-th synth. obs. for group in question

   # print synth. data for sample group: age, bmi, children, charges
   if group == "female\tyes\tsouthwest":
      print("%6.2f %7.2f %6.2f %10.2f" % (obs[0], obs[1], obs[2], obs[3]))
```

5.2.3.2 Generative adversarial networks

This Python program below shows the different steps in the implementation of a GAN synthesizer using TensorFlow and Keras, including the architecture and training the two neural networks involved: the generator and discriminator models. The code is also on GitHub, here. Explanations are in chapter 10 in my book [20], and in the corresponding Jupyter notebook, here. I applied the technique to the diabetes dataset, avaliable as `diabetes.csv`, here.

The code does a lot more than training a standard GAN to produce synthetizations. It includes data cleaning, classifying patients as cancer or not using random forests as a predictive model, evaluating the quality of the synthetizations, and stopping before reaching 10,000 epochs when `mode` is set to `'Enhanced'`. As usual, it produces some plots, in this case a time series of historical values for the loss functions, computed at each epoch.

```
import numpy as np
import pandas as pd
import os
import matplotlib.pyplot as plt
import random as python_random
from tensorflow import random
from keras.models import Sequential
from keras.layers import Dense
from keras.optimizers import Adam # type of gradient descent optimizer
from numpy.random import randn
from matplotlib import pyplot
from sklearn.model_selection import train_test_split
from sklearn.ensemble import RandomForestClassifier
from sklearn import metrics

data = pd.read_csv('diabetes.csv')
# rows with missing data must be treated separately: I remove them here
data.drop(data.index[(data["Insulin"] == 0)], axis=0, inplace=True)
data.drop(data.index[(data["Glucose"] == 0)], axis=0, inplace=True)
data.drop(data.index[(data["BMI"] == 0)], axis=0, inplace=True)
# no further data transformation used beyond this point
data.to_csv('diabetes_clean.csv')

print (data.shape)
print (data.tail())
print (data.columns)


seed = 103 # to make results replicable
np.random.seed(seed) # for numpy
random.set_seed(seed) # for tensorflow/keras
```

```
python_random.seed(seed) # for python

adam = Adam(learning_rate=0.001) # also try 0.01
latent_dim = 10
n_inputs = 9 # number of features
n_outputs = 9 # number of features


#--- STEP 1: Base Accuracy for Real Dataset

features = ['Pregnancies', 'Glucose', 'BloodPressure', 'SkinThickness', 'Insulin',
    'BMI', 'DiabetesPedigreeFunction', 'Age']
label = ['Outcome'] # OutCome column is the label (binary 0/1)
X = data[features]
y = data[label]

# Real data split into train/test dataset for classification with random forest

X_true_train, X_true_test, y_true_train, y_true_test = train_test_split(X, y,
    test_size=0.30, random_state=42)
clf_true = RandomForestClassifier(n_estimators=100)
clf_true.fit(X_true_train,y_true_train)
y_true_pred=clf_true.predict(X_true_test)
print("Base Accuracy: %5.3f" % (metrics.accuracy_score(y_true_test, y_true_pred)))
print("Base classification report:\n",metrics.classification_report(y_true_test,
    y_true_pred))


#--- STEP 2: Generate Synthetic Data

def generate_latent_points(latent_dim, n_samples):
   x_input = randn(latent_dim * n_samples)
   x_input = x_input.reshape(n_samples, latent_dim)
   return x_input

def generate_fake_samples(generator, latent_dim, n_samples):
   x_input = generate_latent_points(latent_dim, n_samples) # random N(0,1) data
   X = generator.predict(x_input,verbose=0)
   y = np.zeros((n_samples, 1)) # class label = 0 for fake data
   return X, y

def generate_real_samples(n):
   X = data.sample(n) # sample from real data
   y = np.ones((n, 1)) # class label = 1 for real data
   return X, y

def define_generator(latent_dim, n_outputs):
   model = Sequential()
   model.add(Dense(15, activation='relu', kernel_initializer='he_uniform',
       input_dim=latent_dim))
   model.add(Dense(30, activation='relu'))
   model.add(Dense(n_outputs, activation='linear'))
   model.compile(loss='mean_absolute_error', optimizer=adam,
       metrics=['mean_absolute_error']) #
   return model

def define_discriminator(n_inputs):
   model = Sequential()
   model.add(Dense(25, activation='relu', kernel_initializer='he_uniform',
       input_dim=n_inputs))
   model.add(Dense(50, activation='relu'))
   model.add(Dense(1, activation='sigmoid'))
   model.compile(loss='binary_crossentropy', optimizer=adam, metrics=['accuracy'])
   return model

def define_gan(generator, discriminator):
```

```
    discriminator.trainable = False # weights must be set to not trainable
    model = Sequential()
    model.add(generator)
    model.add(discriminator)
    model.compile(loss='binary_crossentropy', optimizer=adam)
    return model

def gan_distance(data, model, latent_dim, nobs_synth):

    # generate nobs_synth synthetic rows as X, and return it as data_fake
    # also return correlation distance between data_fake and real data

    latent_points = generate_latent_points(latent_dim, nobs_synth)
    X = model.predict(latent_points, verbose=0)
    data_fake = pd.DataFrame(data=X, columns=['Pregnancies', 'Glucose', 'BloodPressure',
        'SkinThickness', 'Insulin', 'BMI', 'DiabetesPedigreeFunction', 'Age', 'Outcome'])

    # convert Outcome field to binary 0/1
    outcome_mean = data_fake.Outcome.mean()
    data_fake['Outcome'] = data_fake['Outcome'] > outcome_mean
    data_fake["Outcome"] = data_fake["Outcome"].astype(int)

    # compute correlation distance
    R_data   = np.corrcoef(data.T) # T for transpose
    R_data_fake = np.corrcoef(data_fake.T)
    g_dist = np.average(abs(R_data-R_data_fake))
    return(g_dist, data_fake)

def train(g_model, d_model, gan_model, latent_dim, mode, n_epochs=10000, n_batch=128,
    n_eval=50):

    # determine half the size of one batch, for updating the discriminator
    half_batch = int(n_batch / 2)
    d_history = []
    g_history = []
    g_dist_history = []
    if mode == 'Enhanced':
        g_dist_min = 999999999.0

    for epoch in range(0,n_epochs+1):

        # update discriminator
        x_real, y_real = generate_real_samples(half_batch) # sample from real data
        x_fake, y_fake = generate_fake_samples(g_model, latent_dim, half_batch)
        d_loss_real, d_real_acc = d_model.train_on_batch(x_real, y_real)
        d_loss_fake, d_fake_acc = d_model.train_on_batch(x_fake, y_fake)
        d_loss = 0.5 * np.add(d_loss_real, d_loss_fake)

        # update generator via the discriminator error
        x_gan = generate_latent_points(latent_dim, n_batch) # random input for generator
        y_gan = np.ones((n_batch, 1))           # label = 1 for fake samples
        g_loss_fake = gan_model.train_on_batch(x_gan, y_gan)
        d_history.append(d_loss)
        g_history.append(g_loss_fake)

        if mode == 'Enhanced':
            (g_dist, data_fake) = gan_distance(data, g_model, latent_dim, nobs_synth=400)
            if g_dist < g_dist_min and epoch > int(0.75*n_epochs):
              g_dist_min = g_dist
              best_data_fake = data_fake
              best_epoch = epoch
        else:
            g_dist = -1.0
        g_dist_history.append(g_dist)

        if epoch % n_eval == 0: # evaluate the model every n_eval epochs
```

```
            print('>%d, d1=%.3f, d2=%.3f d=%.3f g=%.3f g_dist=%.3f' % (epoch, d_loss_real,
                d_loss_fake, d_loss, g_loss_fake, g_dist))
            plt.subplot(1, 1, 1)
            plt.plot(d_history, label='d')
            plt.plot(g_history, label='gen')
            # plt.show() # un-comment to see the plots
            plt.close()

    OUT=open("history.txt","w")
    for k in range(len(d_history)):
        OUT.write("%6.4f\t%6.4f\t%6.4f\n" %(d_history[k],g_history[k],g_dist_history[k]))
    OUT.close()

    if mode == 'Standard':
        # best synth data is assumed to be the one produced at last epoch
        best_epoch = epoch
        (g_dist_min, best_data_fake) = gan_distance(data, g_model, latent_dim,
            nobs_synth=400)

    return(g_model, best_data_fake, g_dist_min, best_epoch)

#--- main part for building & training model

discriminator = define_discriminator(n_inputs)
discriminator.summary()
generator = define_generator(latent_dim, n_outputs)
generator.summary()
gan_model = define_gan(generator, discriminator)

mode = 'Enhanced' # options: 'Standard' or 'Enhanced'
model, data_fake, g_dist, best_epoch = train(generator, discriminator, gan_model,
    latent_dim, mode)
data_fake.to_csv('diabetes_synthetic.csv')


#--- STEP 3: Classify synthetic data based on Outcome field

features = ['Pregnancies', 'Glucose', 'BloodPressure', 'SkinThickness', 'Insulin',
    'BMI', 'DiabetesPedigreeFunction', 'Age']
label = ['Outcome']
X_fake_created = data_fake[features]
y_fake_created = data_fake[label]
X_fake_train, X_fake_test, y_fake_train, y_fake_test = train_test_split(X_fake_created,
    y_fake_created, test_size=0.30, random_state=42)
clf_fake = RandomForestClassifier(n_estimators=100)
clf_fake.fit(X_fake_train,y_fake_train)
y_fake_pred=clf_fake.predict(X_fake_test)
print("Accuracy of fake data model: %5.3f" % (metrics.accuracy_score(y_fake_test,
    y_fake_pred)))
print("Classification report of fake data
    model:\n",metrics.classification_report(y_fake_test, y_fake_pred))


#--- STEP 4: Evaluate the Quality of Generated Fake Data With g_dist and Table_evaluator

from table_evaluator import load_data, TableEvaluator

table_evaluator = TableEvaluator(data, data_fake)
table_evaluator.evaluate(target_col='Outcome')
# table_evaluator.visual_evaluation()

print("Avg correlation distance: %5.3f" % (g_dist))
print("Based on epoch number: %5d" % (best_epoch))
```

5.2.3.3 Smart grid search

The Python code below is also on GitHub, here, with a Jupyter notebook version available here (see section 10 in the notebook in question). It illustrates the smart grid search algorithm to optimize the parameters associated to the distribution fit to "number of children" in the insurance dataset, as illustrated in Figure 5.5. The context is parametric copulas, where empirical quantiles are replaced by those of a known parametric distribution, with parameters estimated on the real data. It is an alternative to gradient descent. The loss function to minimize is pictured in Figure 5.8. The minimum is in the middle of the narrow, elongated basin. Narrow valleys represent a challenge to most optimization techniques. Here, the X-axis represents the value of one of the two parameters, the Y-axis is for the other parameter, and the contour levels represent the value of the loss function. Best fit to real data is attained when loss is minimum.

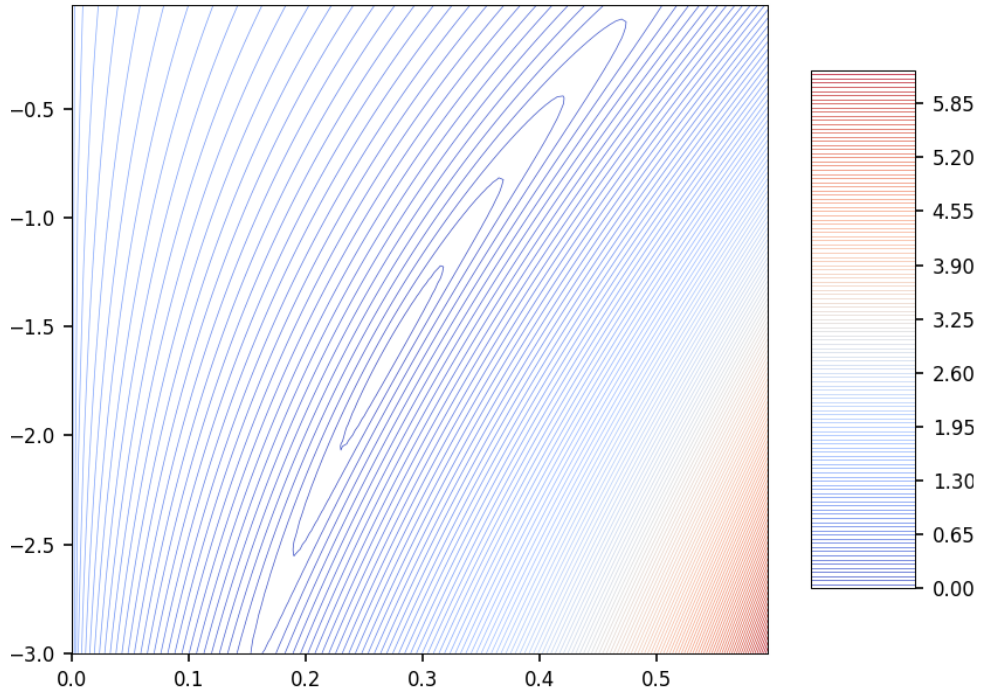


Figure 5.8: Loss function for the 2D smart grid search, with minimum in the basin

```
import numpy as np

#--- compute mean and stdev of ZetaGeom[p, a]

def ZetaGeom(p, a):
    C = 0
    for k in range(200):
        C += p**k/(k+1)**a
    mu = 0
    m2 = 0
    for k in range(200):
        mu += k*p**k/(k+1)**a
        m2 += k*k*p**k/(k+1)**a
    mu /= C
    m2 /= C
    var = m2 - mu*mu
    stdev = var**(1/2)
    return(mu, stdev)

#--- optimized grid search to find optimal p and a

def grid_search(grid_range):
    p_min = grid_range[0][0]
    p_max = grid_range[0][1]
    a_min = grid_range[1][0]
    a_max = grid_range[1][1]
```

```
    p_step = (p_max - p_min)/10
    a_step = (a_max - a_min)/10
    min_delta = 999999999.9
    for p in np.arange(p_min, p_max, p_step):
        for a in np.arange(a_min, a_max, a_step):
            (mu, std) = ZetaGeom(p, a)
            delta = np.sqrt((mu - target_mu)**2 + (std - target_std)**2)
            if delta < min_delta:
                p_best = p
                a_best = a
                mu_best = mu
                std_best = std
                min_delta = delta
    return(p_best, a_best, mu_best, std_best, min_delta)

#--- estimating p and a based on observed mean and standard deviation

target_mu = 1.095 # mean
target_std = 1.205 # standard deviation

p = 0.5
a = 0.0
step_p = 0.4
step_a = 3.0

for level in range(3):
    step_p /= 2
    step_a /= 2
    p_min = max(0, p - step_p)
    p_max = p + step_p
    a_min = a - step_a
    a_max = a + step_a
    grid_range = [(p_min, p_max),(a_min, a_max)]
    (p, a, mu, std, min_delta) = grid_search(grid_range)
    print("delta: %6.4f mu: %6.4f std: %6.4f p: %6.4f a: %6.4f"
        % (min_delta, mu, std, p, a))

# now (p_fit, a_fit) is such that (mean, std) = (target_mu, target_std)
p_fit = p
a_fit = a

# now we found the correct p, a to fit to target_mu, target stdev

#--- sampling from ZetaGeom[p, a]

def CDF(p, a):
    C = 0
    for k in range(100):
        C += p**k/(k+1)**a
    arr_CDF = []
    CDF = 0
    for k in range(100):
        CDF += (p**k/(k+1)**a)/C
        arr_CDF.append(CDF)
    return(arr_CDF)

def sample_from_CDF(p, a):
    u = np.random.uniform(0,1)
    k = 0
    arr_CDF = CDF(p, a)
    while u > arr_CDF[k]:
        k = k+1
    return(k)

#--- sample using estimated p, a to match target mean and stdev
```

```
nobs = 50000 # number of deviates to produce
seed = 500
np.random.seed(seed)
sample1 = np.empty(nobs)
for n in range(nobs):
   k = sample_from_CDF(p_fit, a_fit)
   sample1[n] = k

mean = np.mean(sample1)
std = np.std(sample1)
maxx = max(sample1)
print("\nSample stats: mean: %5.3f std: %5.3f max: %5.3f"
  % (mean, std, maxx))

#--- optional: plotting approximation error for p, a

from mpl_toolkits import mplot3d
import matplotlib as mpl
import matplotlib.pyplot as plt
from matplotlib import cm # color maps

xa = np.arange(0.0, 0.6, 0.005)
ya = np.arange(-3.0, 0.0, 0.025)
xa, ya = np.meshgrid(xa, ya)
za = np.empty(shape=(len(xa),len(ya)))

kk = 0
for p in np.arange(0.0, 0.6, 0.005):
   hh = 0
   for a in np.arange(-3.0, 0.0, 0.025):
      (mu, std) = ZetaGeom(p, a)
      delta = np.sqrt((mu - target_mu)**2 + (std - target_std)**2)
      za[hh, kk] = delta
      hh += 1
   kk += 1

mpl.rcParams['axes.linewidth'] = 0.5
fig = plt.figure()
axes = plt.axes()
axes.tick_params(axis='both', which='major', labelsize=8)
axes.tick_params(axis='both', which='minor', labelsize=8)
CS = axes.contour(xa, ya, za, levels=150, cmap=cm.coolwarm, linewidths=0.35)
cbar = fig.colorbar(CS, ax = axes, shrink = 0.8, aspect = 5)
cbar.ax.tick_params(labelsize=8)
plt.show()

#--- compare with zeta with same mean mu = 1.095

p = 1.0
a = 2.33 # a < 3 thus var is infinite
sample2 = np.empty(nobs)
for n in range(nobs):
   k = sample_from_CDF(p, a)
   sample2[n] = k

mean = np.mean(sample2)
std = np.std(sample2)
maxx = max(sample2)
print("Sample stats Zeta: mean: %5.3f std: %5.3f max: %5.3f"
  % (mean, std, maxx))

#--- compare with geom with same mean mu = 1.095

p = target_mu/(1 + target_mu)
a = 0.0
sample3 = np.empty(nobs)
```

```
for n in range(nobs):
   k = sample_from_CDF(p, a)
   sample3[n] = k

mean = np.mean(sample3)
std = np.std(sample3)
maxx = max(sample3)
print("Sample stats Geom: mean: %5.3f std: %5.3f max: %5.3f"
  % (mean, std, maxx))

#--- plot probability density functions

axes.tick_params(axis='both', which='major', labelsize=4)
axes.tick_params(axis='both', which='minor', labelsize=4)
mpl.rc('xtick', labelsize=8)
mpl.rc('ytick', labelsize=8)
plt.xlim(-0.5,9.5)
plt.ylim(0,0.8)

cdf1 = CDF(p_fit, a_fit)
cdf2 = CDF(1.0, 2.33)
cdf3 = CDF(target_mu/(1+target_mu), 0.0)

for k in range(10):
   if k == 0:
      pdf1 = cdf1[0]
      pdf2 = cdf2[0]
      pdf3 = cdf3[0]
   else:
      pdf1 = cdf1[k] - cdf1[k-1]
      pdf2 = cdf2[k] - cdf2[k-1]
      pdf3 = cdf3[k] - cdf3[k-1]
   plt.xticks(np.linspace(0,9,num=10))
   plt.plot([k+0.2,k+0.2],[0,pdf1],linewidth=5, c='tab:green', label='Zeta-geom')
   plt.plot([k-0.2,k-0.2],[0,pdf2],linewidth=5, c='tab:orange',label='Zeta')
   plt.plot([k,k],[0,pdf3],linewidth=5, c='tab:gray',label='Geom')

plt.legend(['Zeta-geom','Zeta','Geom'],fontsize = 7)
plt.show()
```

5.3 Difference between synthetization and simulation

I focus here on some particular aspects of data synthetizations, that differentiate them from other techniques. Simulations do not simulate joint distributions. Sure, if all your features behave like a mixture of multivariate normal distributions, you can use GMMs (Gaussian mixture models) for synthetization. This is akin to Monte-Carlo simulation. The parameters of the mixture – number of clusters, covariance matrix attached to each Gaussian distribution (one per cluster), and the mixture proportions – can be estimated using the EM algorithm. It is subject to model identifiability issues, but it will work.

If the interdependence structure among the features is essentially linear, in other words well captured by the correlation matrix, you can decorrelate the features using a linear transform such as PCA to remove cross-correlations, then sample each feature separately using standard simulation techniques, and finally apply the inverse transform to add the correlations back. This is similar to what the copula method accomplishes. Each decorrelated feature can be modeled using a parametric metalog distribution to fit with various shapes, akin to Monte-Carlo simulations.

5.3.1 Frequently asked questions

The following questions are frequently asked by participants working on the data synthetizations projects. I added the most popular ones in this section, with my answer.

5.3.1.1 Dealing with a mix of categorical, ordinal, and continuous features

This is when synthetization becomes most useful. The copula method can handle it easily. For categorical variables, you can create buckets also called flag vectors. For instance, `[smoker=yes,region=South,gender=F]` is a bucket. Frequency counts are computed for each bucket in the real dataset. Generate the estimated frequencies for each bucket when synthetizing data. You may aggregate all small buckets into one catch-all bucket. This method, similar to decision trees and XGboost, is a good alternative to turning your categorical features into a large number of binary, numerical dummy variables.

To deal with non-linear interdependencies among your features, GAN synthetizations are usually superior to copula-based methods. Of course, the two methods can be blended: remove the cross-correlations first, then synthetize decorrelated features using GAN, then add back the cross-correlations, with the same linear transform and inverse transform as discussed earlier. One issue is how to measure the correlation between categorical features, or between categorical and numerical features. Metrics such as Cramér's V accomplish this, returning a value between 0 and 1, instead of between -1 and 1 for standard correlations.

5.3.1.2 Do Gaussian copulas work on non-Gaussian observations?

Copulas and GANs aimed at replicating the whole joint distribution, not each component separately, but also the correlations (for copulas) and non-linear feature interactions (GANs). It works with discrete and multimodal distributions combined together, regardless of the underlying distribution (copulas are based on empirical quantiles, although parametric versions are available). Whether you use a Gaussian or Frank or Vine copula does not really matter, except when dealing with distributions with very long tails. Same with GAN: you use Gaussian distributions for latent variables regardless of the actual distributions in the real data.

5.3.1.3 My simulations do as well as synthetizations, how so?

You need to compare the feature dependencies structures as well, not just feature-to-feature (1D) comparisons. Perfect replication of univariate distributions is easy, but replication of cross-interdepencies is the challenging part. See how I measure the quality of the results using the Δ_{avg} in section 5.1 .

Basically, I compute the correlation matrix M_1 on real data, then M_2 on synthetic data, then $\Delta = M_1 - M_2$ except that I take the absolute value of the difference for each entry in $M_1 - M_2$. This correlation distance matrix is denoted as Δ. Then Δ_{avg} is the value averaged over all elements or Δ. So if there are m features, the matrix Δ is $m \times m$, symmetric, the main diagonal is zero, and each element has a value between 0 (great fit) and 1 (bad fit). Note that this method focuses on bivariate comparisons only, and linear interactions only. There are methods to compare more complex multidimensional interactions. More on this in my book on synthetic data and generative AI [20]. See also [6].

5.3.1.4 Sensitivity to changes in the real data

To avoid over-fitting, you assess the quality of the resulting synthetization using the holdout method. Say 50% of your real data is used to train GAN or set up the copula, and the remaining 50% (called validation set) used to compare with synthetic data. This cross-validation technique makes your comparison more meaningful. Sensitivity to the original distribution should not be too large unless you introduce a lot of noise to assess sensitivity to the point that the difference between D_1 and D_2 is larger than between S_1 and D_1 or S_2 and D_2. Here D_1, D_2 are the real data and real data after adding noise, while S_1, S_2 are the corresponding synthetizations.

My own GAN is quite sensitive (compared to vendors) but there are ways to reduce this problem by choosing the loss function and other techniques. My copula is more robust than the open source SDV library, as SDV is a combo GAN/Copula and Lite SDV (the version tested) uses a fast but poor gradient descent algorithm for GAN. Some parameter fine-tuning might be needed to reduce sensitivity. On the circle data, my GAN does better than the copula, with some vendors (YData.ai in particular) doing even much better, and some vendors including SDV Lite doing much worse. Wasserstein GANs [Wiki] is an alternative [36], also designed to avoid mode collapse. This happens when the underlying gradient descent method gets stuck in some local optimum. See example of implementation, here.

5.3.2 Project: synthetizations with categorical features

The goal is to compare two synthetizations based on the insurance dataset: one obtained without ignoring the dependencies between the categorical and quantitative features, and the other one treating the categorical and quantitative features independently. Here GroupID represents a bucket, as illustrated in Table 5.3. Since "gender" and "smoking status" have 2 potential values, while "region" has 4, the total number of buckets is at most $2 \times 2 \times 4 = 16$. Thus GroupID ranges from 0 to 15.

The project consists of the following steps:

Step 1: Generate 1300 observations, synthesizing the categorical features "gender", "smoking status", and "region", but ignoring the quantitative features "charges", "bmi", "number of children", and "age". Analyze the computational complexity of your method. Can you improve it?

Step 2: For each observation, in addition to the categorical features synthesized in step 1, generate the quantitative features. Use a separate copula for each GroupID in table 5.3.

Step 3: Same as step 2, but this time, for the quantitative features use the same global copula for all the observations regardless of GroupID.

Step 4: Compute the average charge per GroupID, both for the synthetizations obtained in steps 2 and 3, and for the real data. Conclude that step 2 yields superior results compared to step 3, because step 3 ignores the dependencies between the categorical and quantitative features.

5.3.3 Solution

The Python code in this section answers step 1. The computational complexity of my solution is equal to the square of the number of buckets (here 16^2) multiplied by the number of observations to generate. Note that the counts in the synthetization follow a multinomial distribution with 16 parameters: the frequencies attached to each GroupID. Thus the 1300 observations could be generated faster, without a loop, using the multinomial generator available in the Numpy library.

In this project, each of the 16 potential buckets have at least 20 observations. However, when granular categorical features such as zip code are present, some buckets may have too few or no observations, even if the dataset is very large. You can bundle these small buckets with the closest ones, or use county rather than zip code. Or treat these small buckets separately, as explained in my hidden decision trees algorithm: see chapter 2 in my book [20].

GroupID	Gender	Smoker	Region	real	synth.
0	female	yes	southwest	21	29
1	male	no	southeast	134	141
2	male	no	northwest	132	152
3	female	no	southeast	139	129
4	female	no	northwest	135	137
5	male	no	northeast	125	126
6	female	yes	southeast	36	34
7	male	no	southwest	126	106
8	male	yes	southeast	55	47
9	female	no	northeast	132	130
10	male	yes	southwest	37	29
11	female	no	southwest	141	132
12	female	yes	northeast	29	25
13	male	yes	northeast	38	44
14	male	yes	northwest	29	28
15	female	yes	northwest	29	33

Table 5.3: Group counts, real versus synthesized

To answer step 2, see the data grouping step in Project 5.2.1. It is based on the same dataset. The results from step 1 tells you how many observations to synthetisize for each GroupID, that is, for each copula. The improvement obtained by using a separate copula for each GroupID, as opposed to a same global copula, is discussed in section 4 (assessing quality of synthetized data) in this notebook. This provides the answer to step 4. The Python code below, solving step 1, is also on GitHub, here.

Categorical features in GAN can be handled with softmax output [Wiki]. See the section "Wasserstein GAN on categorical data" in this paper, and "Generating Multi-Categorical Samples with Generative Adversarial Networks" [4], with the accompanying code here. Finally, to evaluate the quality of synthetic data, you should not focus on raw features only, but compare ratios. For instance, in the diabetes dataset, compare the cancer rate per age group, between real and synthesized data. In this case, cancer and age are the raw features: the former being binary (yes/no, for each patient), and the latter being ordinal.

```
import pandas as pd
import numpy as np

url="https://raw.githubusercontent.com/VincentGranville/Main/main/insurance.csv"
# make sure fields don't contain commas
data = pd.read_csv(url)
print(data.head(10))

groupID = {}
groupLabel = {}
groupCount = {}
ID = 0

Nobs = len(data)
for k in range(0, Nobs):
   obs = data.iloc[k] # get observation number k
   group = obs[1] +"\t"+obs[4]+"\t"+obs[5]
   if group in groupID:
      groupCount[group] += 1
   else:
      groupCount[group] = 1
      groupID[group] = ID
      groupLabel[ID] = group
      ID += 1
Ngroups = len(groupID)

Nobs_synth = Nobs
seed = 453
np.random.seed(seed)

GroupCountSynth = {}
Synth_group = {}
for k in range(Nobs_synth):
   u = np.random.uniform(0.0, 1.0)
   p = 0
   ID = -1
   while p < u:
      ID = ID + 1
      group = groupLabel[ID]
      p += groupCount[group]/Nobs
   group = groupLabel[ID]
   if group in GroupCountSynth:
      GroupCountSynth[group] += 1
   else:
      GroupCountSynth[group] = 0
   Synth_group[k] = group # GroupID assigned to synthetic observation k

for group in groupCount:
   print(group, groupCount[group], GroupCountSynth[group])
```

5.4 Music, synthetic graphs, LLM, and agent-based models

Besides the projects described in this chapter, there are other projects throughout this book, related to generative AI, yet not involving generative adversarial networks. In particular, sections 3.1 and 3.2 deal respectively with time series and geospatial synthetizations, using interpolation techniques. Then, section 2.1 features a very fast synthetizer for tabular data, consistently leading to better synthetizations than those produced by GANs. For DNA sequence synthetization based on LLM techniques, see section 7.1. To synthesize music, see section 4.1. Finally, synthetic graphs and agent-based models are covered in section 6.5.

Chapter 6

Data Visualizations and Animations

This chapter is organized differently. Rather than working on specific project steps, the goal is to learn how to understand, reproduce and fine-tune advanced visualizations. In the process, you will learn various visual techniques and how to optimize their parameters. Most importantly, you want to apply what you learned to your own datasets. To illustrate the visualizations, I use various case studies. Typically, each section has the following components:

- An AI algorithm performing some task, for instance curve fitting, using some input data, and producing some output. You don't need to understand how the AI algorithm works. You only need to think about potential insightful visualizations, to tell what the algorithm accomplishes. In short, thinking in terms of images rather than words.
- The visualization part, typically at the end of the Python code, once the output is created. In some cases, the visualization is a data animation (video). In this case, I provide a link to the actual video on YouTube. But I also include a number of frames in this textbook, to give you an idea. For instance, see Figure 6.1

The topics are selected based both on the usefulness of the associated AI algorithm, and the value of the accompanying visualizations. Everything is in Python. I do not cover dashboards and techniques based on BI tools. The focus is on high quality, insightful visual output, as well as on covering fundamental computer vision concepts: DPI, frames per second, RGB channels, color opacity, optimum palettes, graph visualization, bitmaps and grids, contour levels, orthogonal trajectories and so on. As a starter, I encourage you to read section 1.3.1, featuring a 3D visualization with contour levels and rotating shapes seen from various angles over time.

6.1 Synthesizing data outside the observation range

Given a continuous feature such as annual charges per policy holder for an insurance company, how do you synthesize data mimicking the distribution observed in the training set? In this case, one observation corresponds to one policy holder. Using the empirical cumulative distribution (ECDF), the solution is straightforward. In fact, many synthetic data generators rely on the quantile function – the inverse of the ECDF – available in various Python libraries.

But there is a big caveat. All implementations allow you to sample between the minimum and maximum observed in the training set. If you synthesize a large number of observations, this becomes a serious issue. For instance, if your training data has 1000 observations, you want the 0.9995 quantile to be above the maximum, and the 0.0005 quantile to be below the minimum. But by how much? In short, you want to write you own quantile function: one that generates extrapolated quantiles. I discuss how to do it here. This is the core AI algorithm used in this section, with Python code in section 6.1.2.

6.1.1 Animated histograms for extrapolated quantiles

The algorithm generating the extrapolated quantiles has one important parameter: v. When $v = 0$, the output is identical to traditional quantiles. As v increases, the quantiles get more and more extrapolated, with a smoother distribution and wider range. Figure 6.1 illustrates this transition, with $v = 0$ for the top left histogram, and maximum for the bottom right. The goal is to produce a video showing the exptrapolated quantiles, with 500 frames. Each frame corresponds to a specific value of v, with v (starting at zero) increasing over time. You can watch the video here. Note that unlike the selected 64 frames in Figure 6.1, the video has ticks and labels on both axes (vertical, horizontal), as well as a legend showing the value of v at any time, thus, updated in real time when you watch it.

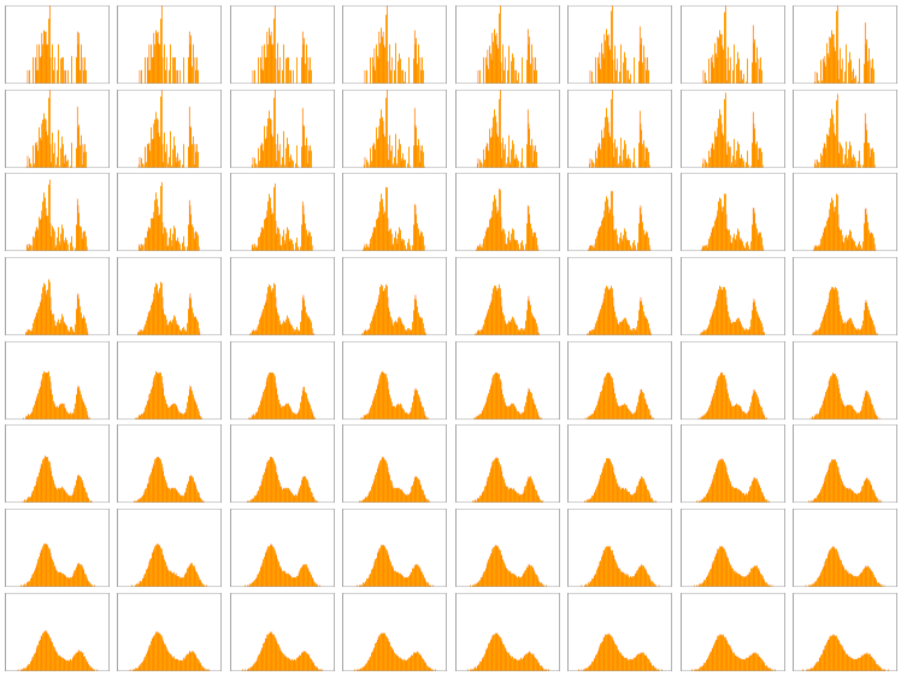


Figure 6.1: From raw data histogram (top left) to extrapolated quantiles (bottom right)

The code in section 6.1.2 produces both the video and Figure 6.1. Now, let's discuss the graphical elements. For easy presentation, I broke them down into the following categories:

- **Video library and parameters**. I used the MoviePy library, in particular `ImageSequenceClip` that reads PNG images and combine them into a video. You might want to check options such as conversion to GIF, compression mode or resolution (dots per inch or DPI). Here I use `fps=10`, that is, 10 frames per second.
- **Matplotlib parameters**. The options and plot types (histograms, scatterplots, contour maps, 3D, grids, and so on) are too numerous to list. The goal in this chapter is to cover the most important ones. Here, note the use of `plt.savefig` to save the video frames produced by `plt.hist`, as PNG images. General Matplotlib options are specified using `mpl.rcParams` and `plt.rcParams`.
- **Adaptive legend**. In `plt.hist`, the option `label='v=%6.4f'%v`, combined in the following line of code with the instruction `plt.legend(loc='upper right',prop={'size':6})`, allows you to display the legend in the upper right corner in each frame, with a different value of v depending on the frame. The formatting (float with 4 digits) is specified by `'v=%6.4f'` while the actual argument v ia passed via `%v`. The argument `'size':6` specifies the size of the font.
- **Image transforms**. Images must have even dimensions (length and height, in pixels), and they must all have the same size, to produce a viewable video. See how I do it my `save_image` function in the code. In other cases, I had to use antialiasing techniques to eliminate pixelation effects. See how I did it, here.
- **GIF format**. You can use the MoviePy library to produce animated GIFs instead of videos. But I was not satisfied with the results. Instead I used free online tools such as Ezgif to convert MP4 videos (produced by MoviePy) to GIF.

6.1.2 Python code: video, thumbnails

The code listed here produces both the video and Figure 6.1. The code is also on GitHub, here.

```
# equantileThumbnails.py: extrapolated quantiles, with thumbnails production

import numpy as np
import matplotlib.pyplot as plt
import matplotlib as mpl
import pandas as pd
from PIL import Image
```

```
import moviepy.video.io.ImageSequenceClip

seed = 76
np.random.seed(seed)

def get_test_data(n=100):
   data = []
   for k in range(n):
      u = np.random.uniform(0, 1)
      if u < 0.2:
         x = np.random.normal(-1, 1)
      elif u < 0.7:
         x = np.random.normal(0, 2)
      else:
         x = np.random.normal(5.5, 0.8)
      data.append(x)
   data = np.array(data)
   return(data)

def get_real_data():
   url = "https://raw.githubusercontent.com/VincentGranville/Main/main/insurance.csv"
   data = pd.read_csv(url)
   # features = ['age', 'sex', 'bmi', 'children', 'smoker', 'region', 'charges']
   data = data['bmi'] # choose 'bmi' or 'charges'
   data = np.array(data)
   return(data)

#--

def truncated_norm(mu, sigma, minz, maxz):
   z = np.random.normal(mu, sigma)
   if minz < maxz:
      while z < minz or z > maxz:
         z = np.random.normal(mu, sigma)
   return(z)

#- sample from mixture

def mixture_deviate(N, data, f, sigma, minz, maxz, verbose=False):
   sample = []
   point_idx = np.random.randint(0, len(data), N)
   mu = data[point_idx]
   for k in range(N):
      z = truncated_norm(mu[k], sigma, minz, maxz)
      sample.append(z)
      if verbose and k%10 == 0:
         print("sampling %6d / %6d" %(k, N))
   sample = np.array(sample)
   sample = np.sort(sample)
   return(sample)

#--- Main part

data = get_test_data(100)
# data = get_real_data()
N = 1000000
truncate = False

# minz > maxz is the same as (minz = -infinity, maxz = +infinity)
if truncate == True:
   minz = 0.50 * np.min(data) # use 0.95 for 'charges', 0.50 for 'bmi'
   maxz = 1.50 * np.max(data) # use 1.50 for 'charges', 1.50 for 'bmi'
else:
   minz = 1.00
   maxz = 0.00

```

```
#--- Making video

mpl.rcParams['axes.linewidth'] = 0.9
plt.rcParams['xtick.labelsize'] = 7
plt.rcParams['ytick.labelsize'] = 7
bins=np.linspace(-7.0, 10.0, num=100)

pbins = 1000
step = N / pbins # N must be a multiple of pbins
my_dpi = 300     # dots per each for images and videos
width = 2400     # image width
height = 1800    # image height
flist = []       # list image filenames for video
nframes = 100
velocity = 1.75

def save_image(fname,frame):
   global fixedSize
   plt.savefig(fname, bbox_inches='tight')
   # make sure each image has same size and size is multiple of 2
   # required to produce a viewable video
   im = Image.open(fname)
   if frame == 0:
      # fixedSize determined once for all in the first frame
      width, height = im.size
      width=2*int(width/2)
      height=2*int(height/2)
      fixedSize=(width,height)
   im = im.resize(fixedSize)
   im.save(fname,"PNG")
   return()

for frame in range(nframes):

   print("Processing frame", frame)
   v = 0.4*(frame/nframes)**velocity ### np.log(1 + frame)/100
   sigma = v * np.std(data)
   sample = mixture_deviate(N, data, truncated_norm, sigma, minz, maxz)
   equant = []

   for k in range(pbins):
      p = (k + 0.5) / pbins
      eq_index = int(step * (k + 0.5))
      equant.append(sample[eq_index])

   plt.figure(figsize=(width/my_dpi, height/my_dpi), dpi=my_dpi)
   plt.hist(equant,color='orange',edgecolor='red',bins=bins,linewidth=0.3,label='v=%6.4f'
       %v)
   # plt.legend(loc='upper right', prop={'size': 6}, )
   plt.ylim(0,60)
   plt.xticks([]) # comment out for the video
   plt.yticks([]) # comment out for the video
   fname='equant_frame'+str(frame)+'.png'
   flist.append(fname)
   save_image(fname,frame)
   plt.close()

clip = moviepy.video.io.ImageSequenceClip.ImageSequenceClip(flist, fps=10)
clip.write_videofile('equant.mp4')


#--- Making picture with thumbmails
#
# note: nframes must be a multiple of n_thumbnails

columns = 10
```

```
rows = 10
n_thumbnails = columns * rows
increment = int(nframes / n_thumbnails)

import matplotlib.image as mpimg
cnt = 1

for frame in range(0, nframes, increment):

   fname = flist[frame]
   img = mpimg.imread(fname)
   plt.subplot(rows, columns, cnt)
   plt.xticks([])
   plt.yticks([])
   plt.axis('off')
   plt.subplots_adjust(wspace = 0.05, hspace = 0.05)
   plt.imshow(img)
   cnt += 1

plt.show()
```

6.2 Curve fitting in bulk

I decided to include this material for two reasons. First, the underlying AI algorithm is a particular case of a generic technique that can perform any kind of regression, supervised or not, under one umbrella. In particular, there is no dependent variable in the example featured in this section. The technique uses gradient descent with a Lagrange multiplier to satisfy some arbitrary constraints on the regression coefficients. In neural network contexts, the constraints are called regularization. The generic method, called cloud regression, is described in chapter 1 in my book on synthetic data [11]. Here I focus on a particular application: curve fitting.

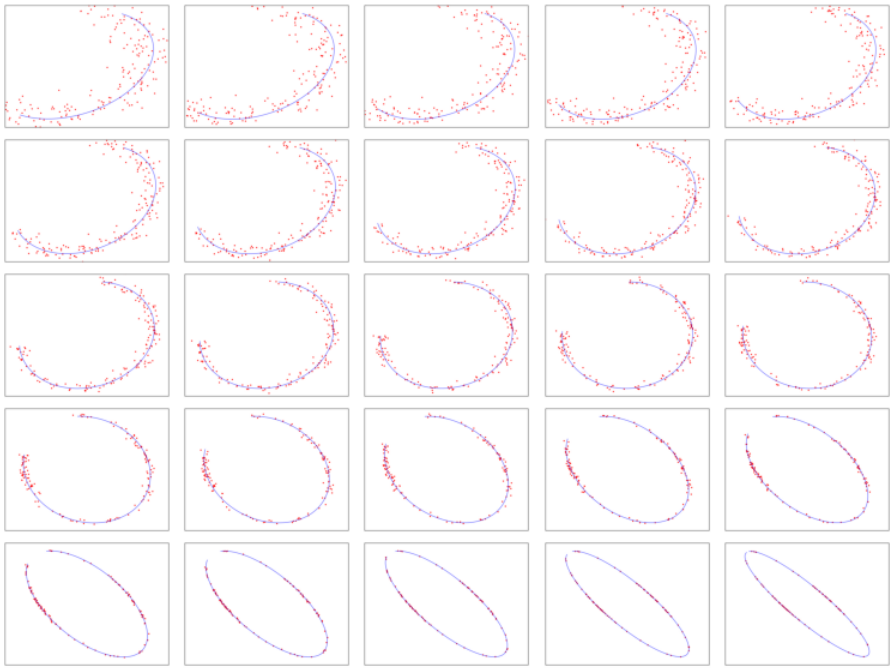


Figure 6.2: Twenty five frames from the curve fitting video

Then, the visualization features a powerful method to summarize a large number of tests, with just one short data animation. To illustrate the potential, I tested the curve fitting procedure on 500 training sets, each one generated with a specific set of parameters and a different amount of noise. The goal is to check how well the curve fitting technique works depending on the data. The originality lies in the choice of a continuous path in the parameter space, so that moving from one training set to the other (that is, from one frame to the next in the video) is done smoothly, yet covering a large number of possible combinations. See 25 of the 500 video

frames in Figure 6.2. Note how the transitions are smooth, yet over time cover various situations: a changing rotation angle, training sets (the red dots) of various sizes, ellipse eccentricity that varies over time, and noise ranging from strong to weak.

6.2.1 Regression without dependent variable

The curve fitting method is an example of regression without dependent variable. You don't predict a response y given a set of features x, using a formula such as $y = f(x, \theta)$ where θ is the model parameter (a vector). Instead, you fit $f(y, x, \theta) = 0$ under some constraint such as $\theta \cdot \theta^T = 1$.

If you look at Figure 6.2, the curve f is a multivalued function [Wiki]. Thus, the method is more general than regression. At the limit, when the eccentricity (one of the parameters) is infinite, the ellipse becomes a straight line. Assuming $\theta \cdot \theta^T = 1$, the problem is equivalent to a Deming regression [Wiki]. The regression line can be vertical: standard regression fails in that case. Then, depending on θ, the method encompasses polynomial regression. If the features are properly scaled, taking values between -1 and $+1$, it is actually a *robust* polynomial regression with no risk of overfitting. Also you can have a curve consisting of two ellipses rather than one. Then, it becomes a clustering problem! Finally, see in Figure 6.2 how it works with partial ellipse arcs.

Now I focus on the visualizations. What I wrote in section 6.1.1 regarding video generation, also applies here. In the code in section 6.2.2, you should focus on the following:

- Line 293 produces the animated GIF, while 295 – 310 generates Figure 6.2. The video frames are created in lines 251 – 286, with the video produced in lines 289 – 290. Watch the video on YouTube, here.
- Parameters controlling the video (DPI, number of frames) are set in lines 240 and 243. Frames per second (`fps`) is set in line 289, and the name of the PNG images in line 256. I invite you to fine-tune them.
- Parameters controlling the curve fitting algorithm are set in lines 259 – 271. Some parameters depend on a real number `p` between 0 and 1 (see line 266), representing the time in the video from beginning to end. Again, you can play with these parameters.

6.2.2 Python code

This version is different from the original listed here. In the original version, you can also generate confidence intervals for the fitted ellipse, that is, for the estimated parameters. This feature is not included in the code below. However, this new version produces better visualizations. You can find it on GitHub, here.

```
# Fitting ellipse via least squares

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

def fit_ellipse(x, y):

    # Fit the coefficients a,b,c,d,e,f, representing an ellipse described by
    # the formula F(x,y) = ax^2 + bxy + cy^2 + dx + ey + f = 0 to the provided
    # arrays of data points x=[x1, x2, ..., xn] and y=[y1, y2, ..., yn].

    # Based on the algorithm of Halir and Flusser, "Numerically stable direct
    # least squares fitting of ellipses'.

    D1 = np.vstack([x**2, x*y, y**2]).T
    D2 = np.vstack([x, y, np.ones(len(x))]).T
    S1 = D1.T @ D1
    S2 = D1.T @ D2
    S3 = D2.T @ D2
    T = -np.linalg.inv(S3) @ S2.T
    M = S1 + S2 @ T
    C = np.array(((0, 0, 2), (0, -1, 0), (2, 0, 0)), dtype=float)
    M = np.linalg.inv(C) @ M
    eigval, eigvec = np.linalg.eig(M)
    con = 4 * eigvec[0]* eigvec[2] - eigvec[1]**2
    ak = eigvec[:, np.nonzero(con > 0)[0]]
    return np.concatenate((ak, T @ ak)).ravel()

```

```
def cart_to_pol(coeffs):

    # Convert the cartesian conic coefficients, (a, b, c, d, e, f), to the
    # ellipse parameters, where F(x, y) = ax^2 + bxy + cy^2 + dx + ey + f = 0.
    # The returned parameters are x0, y0, ap, bp, e, phi, where (x0, y0) is the
    # ellipse centre; (ap, bp) are the semi-major and semi-minor axes,
    # respectively; e is the eccentricity; and phi is the rotation of the semi-
    # major axis from the x-axis.

    # We use the formulas from https://mathworld.wolfram.com/Ellipse.html
    # which assumes a cartesian form ax^2 + 2bxy + cy^2 + 2dx + 2fy + g = 0.
    # Therefore, rename and scale b, d and f appropriately.
    a = coeffs[0]
    b = coeffs[1] / 2
    c = coeffs[2]
    d = coeffs[3] / 2
    f = coeffs[4] / 2
    g = coeffs[5]

    den = b**2 - a*c
    if den > 0:
       raise ValueError('coeffs do not represent an ellipse: b^2 - 4ac must'
                        ' be negative!')

    # The location of the ellipse centre.
    x0, y0 = (c*d - b*f) / den, (a*f - b*d) / den

    num = 2 * (a*f**2 + c*d**2 + g*b**2 - 2*b*d*f - a*c*g)
    fac = np.sqrt((a - c)**2 + 4*b**2)
    # The semi-major and semi-minor axis lengths (these are not sorted).
    ap = np.sqrt(num / den / (fac - a - c))
    bp = np.sqrt(num / den / (-fac - a - c))

    # Sort the semi-major and semi-minor axis lengths but keep track of
    # the original relative magnitudes of width and height.
    width_gt_height = True
    if ap < bp:
       width_gt_height = False
       ap, bp = bp, ap

    # The eccentricity.
    r = (bp/ap)**2
    if r > 1:
       r = 1/r
    e = np.sqrt(1 - r)

    # The angle of anticlockwise rotation of the major-axis from x-axis.
    if b == 0:
       phi = 0 if a < c else np.pi/2
    else:
       phi = np.arctan((2.*b) / (a - c)) / 2
       if a > c:
          phi += np.pi/2
    if not width_gt_height:
       # Ensure that phi is the angle to rotate to the semi-major axis.
       phi += np.pi/2
    phi = phi % np.pi

    return x0, y0, ap, bp, phi

def sample_from_ellipse_even(x0, y0, ap, bp, phi, tmin, tmax, npts):

    npoints = 1000
    delta_theta=2.0*np.pi/npoints
    theta=[0.0]
    delta_s=[0.0]
```

```
    integ_delta_s=[0.0]
    integ_delta_s_val=0.0
    for iTheta in range(1,npoints+1):
       delta_s_val=np.sqrt(ap**2*np.sin(iTheta*delta_theta)**2+ \
                      bp**2*np.cos(iTheta*delta_theta)**2)
       theta.append(iTheta*delta_theta)
       delta_s.append(delta_s_val)
       integ_delta_s_val = integ_delta_s_val+delta_s_val*delta_theta
       integ_delta_s.append(integ_delta_s_val)
    integ_delta_s_norm = []
    for iEntry in integ_delta_s:
       integ_delta_s_norm.append(iEntry/integ_delta_s[-1]*2.0*np.pi)

    x=[]
    y=[]
    for k in range(npts):
       t = tmin + (tmax-tmin)*k/npts
       for lookup_index in range(len(integ_delta_s_norm)):
          lower=integ_delta_s_norm[lookup_index]
          upper=integ_delta_s_norm[lookup_index+1]
          if (t >= lower) and (t < upper):
             t2 = theta[lookup_index]
             break
       x.append(x0 + ap*np.cos(t2)*np.cos(phi) - bp*np.sin(t2)*np.sin(phi))
       y.append(y0 + ap*np.cos(t2)*np.sin(phi) + bp*np.sin(t2)*np.cos(phi))

    return x, y

def sample_from_ellipse(x0, y0, ap, bp, phi, tmin, tmax, npts):

    x = np.empty(npts)
    y = np.empty(npts)
    x_unsorted = np.empty(npts)
    y_unsorted = np.empty(npts)
    angle = np.empty(npts)

    global urs, vrs

    if frame == 0:
       cov=[[ap,0],[0,bp]]
       urs, vrs = np.random.multivariate_normal([0, 0], cov, size = npts_max).T

    # sample from multivariate normal, then rescale
    count = 0
    index = 0
    while count < npts:
       u = urs[index]
       v = vrs[index]
       index += 1
       d=np.sqrt(u*u/(ap*ap) + v*v/(bp*bp))
       u=u/d
       v=v/d
       t = np.pi + np.arctan2(-ap*v,-bp*u)
       if t >= tmin and t <= tmax:
          x_unsorted[count] = x0 + np.cos(phi)*u - np.sin(phi)*v
          y_unsorted[count] = y0 + np.sin(phi)*u + np.cos(phi)*v
          angle[count]=t
          count=count+1

    # sort the points x, y for nice rendering with mpl.plot
    hash={}
    hash = dict(enumerate(angle.flatten(), 0)) # convert array angle to dictionary
    idx=0
    for w in sorted(hash, key=hash.get):
       x[idx]=x_unsorted[w]
       y[idx]=y_unsorted[w]
```

```
163         idx=idx+1
164
165     return x, y
166
167 def get_ellipse_pts(params, npts=100, tmin=0, tmax=2*np.pi, sampling='Standard'):
168
169     # Return npts points on the ellipse described by the params = x0, y0, ap,
170     # bp, e, phi for values of the parametric variable t between tmin and tmax.
171
172     x0, y0, ap, bp, phi = params
173
174     if sampling=='Standard':
175         t = np.linspace(tmin, tmax, npts)
176         x = x0 + ap * np.cos(t) * np.cos(phi) - bp * np.sin(t) * np.sin(phi)
177         y = y0 + ap * np.cos(t) * np.sin(phi) + bp * np.sin(t) * np.cos(phi)
178     elif sampling=='Enhanced':
179         x, y = sample_from_ellipse(x0, y0, ap, bp, phi, tmin, tmax, npts)
180     elif sampling=='Even':
181         x, y = sample_from_ellipse_even(x0, y0, ap, bp, phi, tmin, tmax, npts)
182
183     return x, y
184
185 def vgplot(x, y, color, npts, tmin, tmax):
186
187     plt.plot(x, y, linewidth=0.8, color=color) # plot exact ellipse
188     # fill gap (missing segment in the ellipse plot) if plotting full ellipse
189     if tmax-tmin > 2*np.pi - 0.001:
190         gap_x=[x[nlocs-1],x[0]]
191         gap_y=[y[nlocs-1],y[0]]
192         plt.plot(gap_x, gap_y, linewidth=0.8, color=color)
193     plt.xticks([])
194     plt.yticks([])
195     plt.xlim(-1 + min(x), 1 + max(x))
196     plt.ylim(-1 + min(y), 1 + max(y))
197     return()
198
199 def main(npts, noise, seed, tmin, tmax, params, sampling):
200
201     # params = x0, y0, ap, bp, phi (input params for ellipse)
202     global ur, vr
203
204     # Get points x, y on the exact ellipse and plot them
205     x, y = get_ellipse_pts(params, npts, tmin, tmax, sampling)
206
207     # perturb x, y on the ellipse with some noise, to produce training set
208     if frame == 0:
209       cov = [[1,0],[0,1]]
210       np.random.seed(seed)
211       ur, vr = np.random.multivariate_normal([0, 0], cov, size = npts_max).T ### npts).T
212     x += noise * ur[0:npts]
213     y += noise * vr[0:npts]
214
215     # get and print exact and estimated ellipse params
216     coeffs = fit_ellipse(x, y) # get quadratic form coeffs
217     print('True ellipse : x0, y0, ap, bp, phi = %+.5f %+.5f %+.5f %+.5f %+.5f' % params)
218     fitted_params = cart_to_pol(coeffs) # convert quadratic coeffs to params
219     print('Estimated values: x0, y0, ap, bp, phi = %+.5f %+.5f %+.5f %+.5f %+.5f' %
          fitted_params)
220     print()
221
222     # plot training set points in red
223     plt.scatter(x, y,s = 3.5,color = 'red')
224
225     # get nlocs points on the fitted ellipse and plot them
226     x, y = get_ellipse_pts(fitted_params, nlocs, tmin, tmax, sampling)
227     vgplot(x, y,'blue', nlocs, tmin, tmax)
```

```
228
229     # save plots in a picture [filename is image]
230     plt.savefig(image, bbox_inches='tight',dpi=dpi)
231     plt.close() # so, each video frame contains one curve only
232     return()
233
234 #--- Main Part: Initializationa
235
236 sampling= 'Enhanced'  # options: 'Enhanced', 'Standard', 'Even'
237 npts_max = 50000   # max size of random arrays
238 nlocs = 2500       # number of points used to represent true ellipse
239
240 dpi =240    # image resolution in dpi (100 for gif / 300 for video)
241 flist = [] # list of image filenames for the video
242 gif = []    # used to produce the gif image
243 nframes = 500 # number of frames in video
244
245 # intialize plotting parameters
246 plt.rcParams['axes.linewidth'] = 0.8
247 plt.rc('axes',edgecolor='black') # border color
248 plt.rc('xtick', labelsize=6) # font size, x axis
249 plt.rc('ytick', labelsize=6) # font size, y axis
250
251 #--- Main part: Main loop
252
253 for frame in range(0,nframes):
254
255     # Global variables: dpi, frame, image
256     image='ellipse'+str(frame)+'.png' # filename of image in current frame
257     print("Creating image",image) # show progress on the screen
258
259     # params = (x0, y0, ap, bp, phi) : first two coeffs is center of ellipse, last one
260     # is rotation angle, the two in the middle are the semi-major and semi-minor axes.
261     #
262     # Also: 0 <= tmin < tmax <= 2 pi determine start / end of ellipse arc
263
264     # parameters used for current frame
265     seed = 100       # same seed (random number generator) for all images
266     p = frame/(nframes-1) # assumes nframes > 1
267     noise = (1-p)*(1-p) # amount of noise added to to training set
268     npts = int(100*(2-p)) # number of points in training set, < npts_max
269     tmin= (1-p)*np.pi # training set: ellipse arc starts at tmin >= 0
270     tmax= 2*np.pi    # training set: ellipse arc ends at tmax < 2*Pi
271     params = 4, -3.5, 7, 1+6*(1-p), (p+np.pi/3) # ellipse parameters
272
273     # call to main function
274     main(npts, noise, seed, tmin, tmax, params, sampling)
275
276     # processing images for video and animated gif production (using pillow library)
277     im = Image.open(image)
278     if frame==0:
279        width, height = im.size # determines the size of all future images
280        width=2*int(width/2)
281        height=2*int(height/2)
282        fixedSize=(width,height) # even number of pixels for video production
283     im = im.resize(fixedSize) # all images must have same size to produce video
284     gif.append(im)  # to produce Gif image [uses lots of memory if dpi > 100]
285     im.save(image,"PNG") # save resized image for video production
286     flist.append(image)
287
288 # output video / fps is number of frames per second
289 clip = moviepy.video.io.ImageSequenceClip.ImageSequenceClip(flist, fps=20)
290 clip.write_videofile('ellipseFitting.mp4')
291
292 # output video as gif file
293 gif[0].save('ellipseFitting.gif',save_all=True, append_images=gif[1:],loop=0)
```

```
294
295 #--- Making picture with thumbmails
296 #
297 # note: nframes must be a multiple of n_thumbnails
298
299 columns = 5
300 rows = 5
301 n_thumbnails = columns * rows
302 increment = int(nframes / n_thumbnails)
303
304 import matplotlib.image as mpimg
305 cnt = 1
306
307 for frame in range(0, nframes, increment):
308
309    fname = flist[frame]
310    img = mpimg.imread(fname)
311    plt.subplot(rows, columns, cnt)
312    plt.xticks([])
313    plt.yticks([])
314    plt.axis('off')
315    plt.subplots_adjust(wspace = 0.05, hspace = 0.05)
316    plt.imshow(img)
317    cnt += 1
318
319 plt.show()
```

6.3 Gradient descent, grids, and maps

Under construction.

6.4 Supervised classification with image bitmaps

6.5 Miscellaneous topics

6.5.1 Agent-based modeling: collision graphs

6.5.2 Terrain generation: image and palette morphing

6.5.3 Mathematical art

Chapter 7

NLP and Large Language Models

If you tried apps such as GPT (generative pre-training transformer), you may be surprised by the quality of the sentences, images, sound, videos, or code generated. Yet, in the end, the value is the depth and relevance or the content generated, more than the way it is presented. My interest started when I did a Google search for "variance of the range for Gaussian distributions". I vaguely remember that it is of the order $1/\sqrt{n}$ where n is the number of observations, but could not find the reference anymore. Indeed I could not find anything at all on this topic. The resources I found on the subject 10 years ago are all but gone, or at least very hard to find. As search evolved over time, it now caters to a much larger but less educated audience. As a result, none of the search results were even remotely relevant to my question. This is true for pretty much any research question that I ask.

Using OpenAI, I found the answer I was looking for, even with more details than expected, yet with no link to an actual reference, no matter how I change my prompt. OpenAI could not find my answer right away, and I had to rephrase my prompt as "what is the asymptotic variance of the range for Gaussian distributions". More general prompts on specific websites, such as "asymptotic distribution of sample variance" lead to a number of articles which in turn lead to some focusing on Gaussian distributions. Even today, automatically getting a good answer in little time, with a link, is still a challenge.

In this chapter, I focus on building better LLMs to address the issues in question. Project 7.1 is peculiar in the sense that the alphabet consists of 4 symbols. The goal is to synthesize sequences that exhibit the same autocorrelation structure as found in the original data, and then evaluate the results. Then, in projects 7.2.1, 7.2.2, and 7.2.3, I discuss in details a new multi-LLM app based on crawled data and reconstructed taxonomies, with specialized sub-LLMs (one per top category), self-tuning, variable-length embeddings with multi-token words, and much smaller tables. The material covers the core components of this new, energy-efficient RAG architecture, including smart crawling. There is no neural network involved.

7.1 Synthesizing DNA sequences with LLM techniques

This project is not focused on genome data alone. The purpose is to design a generic solution that may also work in other contexts, such as synthesizing molecules. The problem involves dealing with a large amount of "text". Indeed, the sequences discussed here consist of letter arrangements, from an alphabet that has 5 symbols: A, C, G, T and N. The first four symbols stand for the types of bases found in a DNA molecule: adenine (A), cytosine (C), guanine (G), and thymine (T). The last one (N) represents missing data. No prior knowledge of genome sequencing is required.

The data consists of DNA sub-sequences from a number of individuals, and categorized according to the type of genetic patterns found in each sequence. Here I combined the sequences together. The goal is to synthesize realistic DNA sequences, evaluate the quality of the synthetizations, and compare the results with random sequences. The idea is to look at a DNA string S_1 consisting of n_1 consecutive symbols, to identify potential candidates for the next string S_2 consisting of n_2 symbols. Then, assign a probability to each string S_2 conditionally on S_1, use these transition probabilities to sample S_2 given S_1, then move to the right by n_2 symbols, do it again, and so on. Eventually you build a synthetic sequence of arbitrary length. There is some analogy to Markov chains. Here, n_1 and n_2 are fixed, but arbitrary.

7.1.1 Project and solution

Let's look at 3 different DNA sequences. The first one is from a real human being. The second one is synthetic, replicating some of the patterns found in real data. The third one is purely random, with each letter indepen-

```
Real DNA

ATGCCCCAACTAAATACTACCGTATGGCCCACCATAATTACCCCCATACTCCTTACACTATTCCTCATCACCCAACTA
AAAATATTAAACACAAACTACCACCTACCTCCCTCACCAAAGCCCATAAAAATAAAAAATTATAACAAACCCTGAGAA
CCAAAATGAACGAAAATCTGTTCGCTTCATTCATTGCCCCCACAATCCTAGNATGAACGAAAATCTGTTCGCTTCATT
CATTGCCCCCACAATCCTAGGCCTACCCGCCGCAGTACTGATCATTCTATTTCCCCCTCTATTGATCCCCACCTCCAA
ATATCTCATCAACAACCGACTAATCACCACCCAACAATGACTAATCAAACTAACCTCAAAACAAATGATAACCATACA

Synthetic DNA

TTGTTTTCTTCACCTAAATGCACAAGAATGGTGGGCCGAGGAGCCATGTCAAGTGGGGATGGGTCTATCGAACCTGAG
GGCCCCCCACTTCAGATGCTTCGTACTGTCTTTGGGACTTCTCACCGTCTCATGGTCTGCCCTGCCCCGCAGTGTGGC
CTGGTATTTTTAACCCTATTATAGAAACAACAATTTATGGGCTCCTTGAAGCTTATACAATACAACAGTAAAGGGCCC
CTCCTCCAGTCAGCCTCTTTCCCTCTTAGGGTAAATGAGGATATCCAAGTGCCCACCTCATCATCAACTCCGCCACCA
GTTTGCAGCCCTTGCAGGAGATTTCTGGTGATGAAAGTTCAGTGGACTTGGGAAAAGCCGTCATGCTGTCTGCCAACC

Random DNA

ATCCTGCTTCATATGTAGGAAGGGTTGTAGGTTCCCGGAGGGCGCATTGCAAAGACCGGCCAGACTACTTATGGCCGC
GTCCTAAGCACCATATGCTAAGCCTGATTAACATCGCGCGGATGTAACTACACGCGCGCTACGTGAATCCTAGGCAGC
CGTCACGATTGACTCCTCATACTCATCGAGGCGCTCGCGTCATAGACCGACCATCGCGTCACCATAATAAGTAGAGTC
TTTACGGTAGGCCTTCAAAATACGGACAAGGCATTTGTATTCTTCATGTCATGTAGCTGAAGAATACCATTAAGTTTA
TAGGCGGGTGTACGACAAGACTGCCAGGTGGCAGTGTCGTCACAAGAGCGCGTAAACTTTTTGCCGGTAATAGACCGT
```

Table 7.1: Genome data, three DNA sub-sequences: real, synthetic, and random

dently distributed from each other, with the same 25% marginal frequency. Can you tell the differences just by looking at the 3 sequences in Table 7.1? If not, see Figure 7.1 and accompanying description.

For this project, I start with real sequences to train the DNA synthesizer. I then evaluate the quality, and show how synthetic DNA is superior to random sequences. Any symbol other than A, C, G, or T must be labeled as N. It represents missing data and I ignore it in the Python code. Figures 7.1 and 7.2 illustrate the end result: comparing string frequencies in real, synthetic and random DNA sequences, for specific "words" consisting of n_3 consecutive symbols. The main diagonal (red line) represents perfect fit with real DNA. The number of "words" (dots in the pictures) varies from 4000 to 10,000, and are called `nodes` in the Python code. It shows that these DNA sequences are anything but random, with synthetic fitting real data quite well.

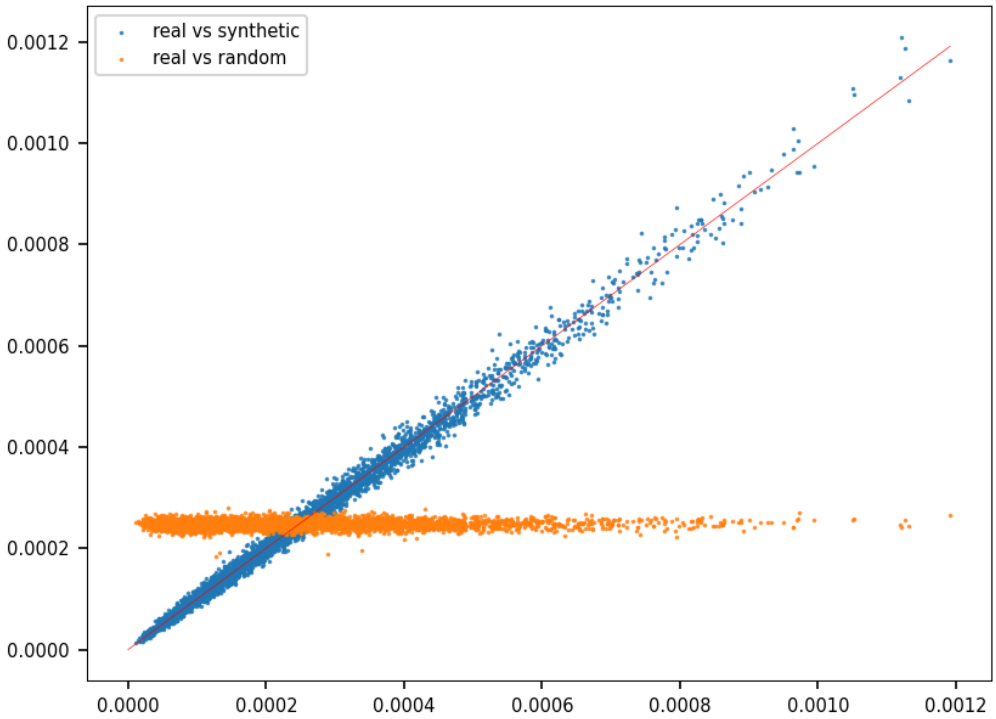


Figure 7.1: PDF scatterplots, $n_3 = 6$: real DNA vs synthetic (blue), and vs random (orange)

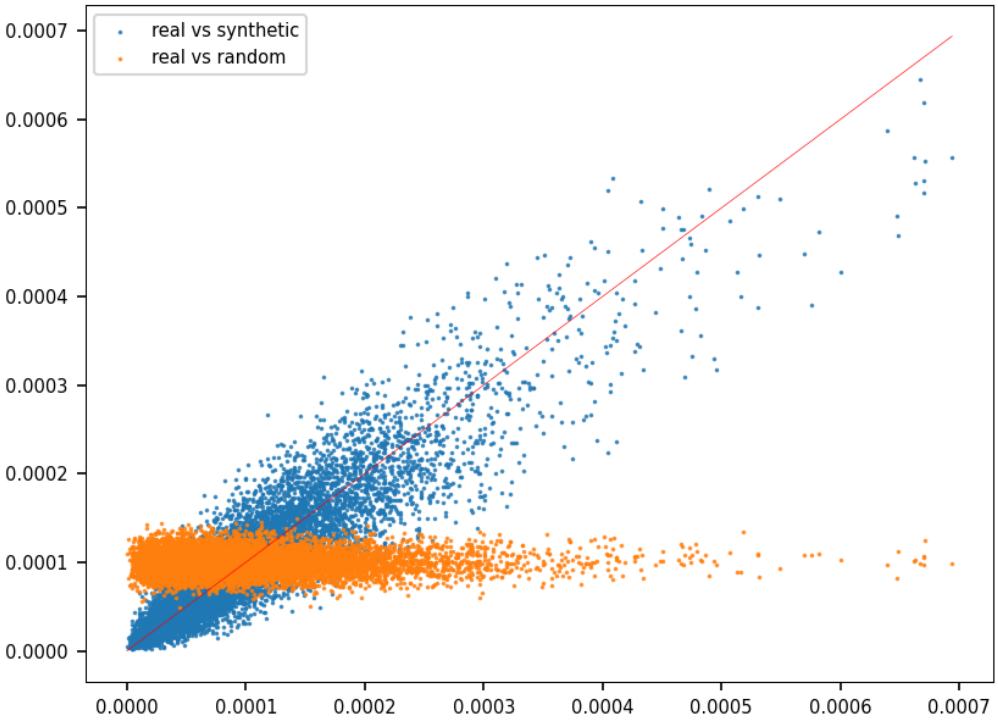


Figure 7.2: PDF scatterplots, $n_3 = 8$: real DNA vs synthetic (blue), and vs random (orange)

The project consists of the following steps:

Step 1: Understanding the data. Look at the URL in the Python code in section 7.1.2 to access the data. Ignore the "class" feature as the purpose here is not to classify DNA sequences. A small extract of a real DNA sequence is featured in Table 7.1, at the top. Note that the order of the letters is important.

Step 2: Compute summary statistics. For n_1 and n_2 fixed, extract all distinct strings S_1, S_2 of length respectively n_1 and n_2, and compute their occurrences. Do the same for strings S_{12} of length $n_1 + n_2$. The goal is to predict, given a string S_1 of length n_1, the probability to be followed by a specific string S_2 of length n_2. That is, $P[S_2 = s_2|S_1 = s_1]$. Here a string is a sub-sequence of letters. Strings are called words, and letters are also called characters. The four letters are 'A', 'B', 'C', 'D'. Ignore the letter 'N'. Then, do the same when the strings S_1 and S_2 are separated by a gap of g letters, for $g = 1, 2, 3$.

Step 3: String associations. Two specific strings S_1, S_2 may frequently be found together, or rarely. Characterize this string association using the pointwise mutual information (PMI) [Wiki]. Rare occurrences may indicate a rare genetic condition. Order the (S_1, S_2) found in the data according to the PMI metric.

Step 4: Synthesize a DNA sequence. Proceed as follows. Start with an arbitrary string S_1 of length n_1. Then add n_2 letters at a time, sequentially, to the DNA sequence being generated. In other words, at each step, sample S_2 from $P(S_2|S_1)$, where S_2 is the new string of length n_2 to be added, and S_1 is the last string of length n_1 built so far.

Step 5: Evaluate the quality. In this context, the Hellinger distance [Wiki] is simple and convenient, to assess the quality of the synthetic DNA sequence, that is, how well it replicates the patterns found in real DNA. The value is between 0 (great fit) and 1 (worst fit). Randomly select $n = 10{,}000$ strings S_3 of length n_3 found in real DNA. These strings are referred to as *nodes*. Compute the frequency $P_{\text{real}}(S_3)$ for each of them. Also compute the frequency $P_{\text{synth}}(S_3)$ on the synthetic sequence. The Hellinger distance is then

$$\text{HD} = \sqrt{1 - \sum \sqrt{P_{\text{real}}(S_3) \cdot P_{\text{synth}}(S_3)}},$$

where the sum is over all the selected strings S_3. Also compare real DNA with a random sequence, using HD. Show that synthetic DNA is a lot better than random sequences, to mimic real DNA. Finally, try different values of n_1, n_2, n_3 and check whether using $n = 1000$ nodes provides a good enough approximation to HD (it is much faster than $n = 10{,}000$, especially when n_3 is large).

The solution to the first four steps correspond to steps [1–4] in the Python code in section 7.1.2, while Step 5 corresponds to step [6] in the code. To compute summary statistics (Step 2) when S_1 and S_2 are separated by a gap of g letters, replace `string2=obs[pos1:pos2]` by `string2=obs[pos1+g:pos2+g]` in step [2] in the code. The interest in doing this is to assess whether there are long-range associations between strings. By default, in the current version of the code, $g = 0$.

Figures 7.1 and 7.2 show scatterplots with probability vectors $[P_{\text{real}}(S_3), P_{\text{synth}}(S_3)]$ in blue, for thousands of strings S_3 found in the real DNA sequence. For orange dots, the second component $P_{\text{synth}}(S_3)$ is replaced by $P_{\text{rand}}(S_3)$, the value computed on a random sequence. Clearly, the synthetic DNA is much more realistic than the random DNA, especially when $n_3 = 6$. Note that the probabilities are associated to overlapping events: for instance, the strings 'AACT' and 'GAAC' are not independent, even in the random sequence. The Hellinger distance used here is not adjusted for this artifact.

To dive deeper into DNA synthetization, you might want to investigate problems such as the minimal DNA sequence that determines the gender, or some other genetic features.

7.1.2 Python code

The code is also on GitHub, here. For explanations, see section 7.1.1.

```
# genome.py : synthesizing DNA sequences
# data: https://www.kaggle.com/code/tarunsolanki/classifying-dna-sequence-using-ml

import pandas as pd
import numpy as np
import re # for regular expressions


#--- [1] Read data

url = "https://raw.githubusercontent.com/VincentGranville/Main/main/dna_human.txt"
human = pd.read_table(url)
# human = pd.read_table('dna_human.txt')
print(human.head())


#--- [2] Build hash table architecture
#
# hash1_list[string1] is the list of potential string2 found after string1, separated by
    ~

nobs = len(human)
print(nobs)
hash12 = {}
hash1_list = {}
hash1 = {}
hash2 = {}
count1 = 0
count2 = 0
count12 = 0
sequence = ''

for k in range(nobs):
  obs = human['sequence'][k]
  sequence += obs
  sequence += 'N'
  type = human['class'][k]
  length = len(obs)
  string1_length = 4
  string2_length = 2
  pos0 = 0
  pos1 = pos0 + string1_length
  pos2 = pos1 + string2_length

  while pos2 < length:

```

```
      string1 = obs[pos0:pos1]
      string2 = obs[pos1:pos2]

      if string1 in hash1:
         if string2 not in hash1_list[string1] and 'N' not in string2:
            hash1_list[string1] = hash1_list[string1] + '~' + string2
         hash1[string1] += 1
         count1 += 1
      elif 'N' not in string1:
         hash1_list[string1] = '~' + string2
         hash1[string1] = 1
      key = (string1, string2)

      if string2 in hash2:
         hash2[string2] += 1
         count2 += 1
      elif 'N' not in string2:
         hash2[string2] = 1

      if key in hash12:
         hash12[key] += 1
         count12 += 1
      elif 'N' not in string1 and 'N' not in string2:
         hash12[key] = 1

      pos0 += 1
      pos1 += 1
      pos2 += 1

   if k % 100 == 0:
      print("Creating hash tables: %6d %6d %4d" %(k, length, type))


#--- [3] Create table of string associations, compute PMI metric

print()
index = 0
for key in hash12:
   index +=1
   string1 = key[0]
   string2 = key[1]
   n1 = hash1[string1] # occurrences of string1
   n2 = hash2[string2] # occurrences of string2
   n12 = hash12[key] # occurrences of (string1, string2)
   p1 = n1 / count1 # frequency of string1
   p2 = n2 / count2 # frequency of string2
   p12 = n12 / count12 # frequency of (string1, string2)
   pmi = p12 / (p1 * p2)
   if index % 100 == 0:
      print("Computing string frequencies: %5d %4s %2s %4d %8.5f"
            %(index, string1, string2, n12, pmi))
print()


#--- [4] Synthetization
#
# synthesizing word2, one at a time, sequencially based on previous word1

n_synthetic_string2 = 2000000
seed = 65
np.random.seed(seed)

synthetic_sequence = 'TTGT' # starting point (must be existing string1)
pos1 = len(synthetic_sequence)
pos0 = pos1 - string1_length

```

```
112
113 for k in range(n_synthetic_string2):
114
115     string1 = synthetic_sequence[pos0:pos1]
116     string = hash1_list[string1]
117     myList = re.split('~', string)
118
119     # get target string2 list in arr_string2, and corresponding probabilities in arr_proba
120     arr_string2 = []
121     arr_proba = []
122     cnt = 0
123     for j in range(len(myList)):
124        string2 = myList[j]
125        if string2 in hash2:
126           key = (string1, string2)
127           cnt += hash12[key]
128           arr_string2.append(string2)
129           arr_proba.append(hash12[key])
130     arr_proba = np.array(arr_proba)/cnt
131
132     # build cdf and sample word2 from cdf, based on string1
133     u = np.random.uniform(0, 1)
134     cdf = arr_proba[0]
135     j = 0
136     while u > cdf:
137        j += 1
138        cdf += arr_proba[j]
139     synthetic_string2 = arr_string2[j]
140     synthetic_sequence += synthetic_string2
141     if k % 100000 == 0:
142        print("Synthesizing %7d/%7d: %4d %8.5f %2s"
143               % (k, n_synthetic_string2, j, u, synthetic_string2))
144
145     pos0 += string2_length
146     pos1 += string2_length
147
148 print()
149 print("Real DNA:\n", sequence[0:1000])
150 print()
151 print("Synthetic DNA:\n", synthetic_sequence[0:1000])
152 print()
153
154
155 #--- [5] Create random sequence for comparison purposes
156
157 print("Creating random sequence...")
158 length = (1 + n_synthetic_string2) * string2_length
159 random_sequence = ""
160 map = ['A', 'C', 'T', 'G']
161
162 for k in range(length):
163    random_sequence += map[np.random.randint(4)]
164    if k % 100000 == 0:
165       print("Creating random sequence: %7d/%7d" %(k,length))
166 print()
167 print("Random DNA:\n", random_sequence[0:1000])
168 print()
169
170
171 #--- [6] Evaluate quality: real vs synthetic vs random DNA
172
173 max_nodes = 10000 # sample strings for frequency comparison
174 string_length = 6 # length of sample strings (fixed length here)
175
176 nodes = 0
177 hnodes = {}
```

```
iter = 0

while nodes < max_nodes and iter < 5*max_nodes:
   index = np.random.randint(0, len(sequence)-string_length)
   string = sequence[index:index+string_length]
   iter += 1
   if string not in hnodes and 'N' not in string:
      hnodes[string] = True
      nodes += 1
      if nodes % 1000 == 0:
         print("Building nodes: %6d/%6d" %(nodes, max_nodes))
print()

def compute_HD(hnodes, sequence, synthetic_sequence):

   pdf1 = []
   pdf2 = []
   cc = 0

   for string in hnodes:
      cnt1 = sequence.count(string)
      cnt2 = synthetic_sequence.count(string)
      pdf1.append(float(cnt1))
      pdf2.append(float(cnt2))
      ratio = cnt2 / cnt1
      if cc % 100 == 0:
         print("Evaluation: computing EPDFs: %6d/%6d: %5s %8d %8d %10.7f"
               %(cc, nodes, string, cnt1, cnt2, ratio))
      cc += 1

   pdf1 = np.array(pdf1) # original dna sequence
   pdf2 = np.array(pdf2) # synthetic dna sequence
   pdf1 /= np.sum(pdf1)
   pdf2 /= np.sum(pdf2)

   HD = np.sum(np.sqrt(pdf1*pdf2))
   HD = np.sqrt(1 - HD)
   return(pdf1, pdf2, HD)

pdf_dna, pdf_synth, HD_synth = compute_HD(hnodes, sequence, synthetic_sequence)
pdf_dna, pdf_random, HD_random = compute_HD(hnodes, sequence, random_sequence)

print()
print("Total nodes: %6d" %(nodes))
print("Hellinger distance [synthetic]: HD = %8.5f" %(HD_synth))
print("Hellinger distance [random] : HD = %8.5f" %(HD_random))

#--- [7] Visualization (PDF scatterplot)

import matplotlib.pyplot as plt
import matplotlib as mpl

mpl.rcParams['axes.linewidth'] = 0.5
plt.rcParams['xtick.labelsize'] = 7
plt.rcParams['ytick.labelsize'] = 7

plt.scatter(pdf_dna, pdf_synth, s = 0.1, color = 'red', alpha = 0.5)
plt.scatter(pdf_dna, pdf_random, s = 0.1, color = 'blue', alpha = 0.5)
plt.legend(['real vs synthetic', 'real vs random'], loc='upper left', prop={'size': 7}, )
plt.plot([0, np.max(pdf_dna)], [0, np.max(pdf_dna)], c='black', linewidth = 0.3)
plt.show()
```

7.2 Creating high quality LLM embeddings

The purpose of this project is twofold. First, learning how to efficient crawl large, well structured websites to extract and reconstruct comprehensive keyword taxonomies, as well as gathering vast amounts of text. Here, the focus is on one particular component of human knowledge: statistics and probability theory. Then use the crawled data, structure it, and provide high quality answers – along with the sources – to specific questions or prompts. At the time of this writing, OpenAI fails at both.

Unlike OpenAI, the goal is not to produce wordy English prose explaining rudimentary principles at length, blended with more advanced material. Instead, the output may consist of a few bullet points, keywords and links. That is, what is most useful to busy professionals who are experts in their field. The motivation behind this project is to eliminate bottlenecks in standard search technology. In particular, Google search (whether targeted to a specific website or not) as well as search boxes found on Wikipedia, Stack Exchange, ArXiv, Wolfram and other websites, including "related content" links found on these websites, are of limited value.

Likewise, GPT frequently provides slightly wrong answers to scientific questions, without providing the sources that would allow you to easily fix the errors. This may result in considerable time spent in searching specialized information, using methods that could be automated. The example at the origin of this project was my search to find the asymptotic expectation of the range for Gaussian distributions. Why I was interested in this topic, and the problems that I faced even with GPT, are described here.

Pair ID	Depth	Category	Parent Category
19393	4	Pappus's Centroid Theorem	Surfaces of Revolution
11589	3	Connecting Homomorphism	Cohomology
202	2	Belongie	MathWorld Contributors
16755	4	Nonstandard Methods	Nonstandard Analysis
7877	3	Positive Linear Functional	Moslehian
20970	5	Sinhc Function	Transcendental Root Constants
24829	5	de Finetti Diagram	Triangle Properties
24237	5	Inverse Curve	Polar Curves
23013	5	Moore-Penrose Pseudoinverse	Matrix Operations
25751	5	Medial Hexagonal Hexecontahedron	Uniform Polyhedra
5552	3	LerchPhi	Wolfram Language Commands
466	2	Levai	MathWorld Contributors
18508	4	Convex Polygon	Polygons
13327	5	Damped Simple Harmonic Motion--Underdamping	Ordinary Differential Equations
12757	4	Durand's Rule	Numerical Integration
21063	5	17	Small Numbers
10212	4	Connell Sequence	Parity
22606	4	Almost Alternating Link	Alternating Knots
23564	5	Polydrafter	Miscellaneous Polyshapes
15653	4	One-One Complete	Theory of Computation
3792	3	Rule 30	A New Kind of Science

Figure 7.3: Extract from reconstructed taxonomy structure, Wolfram website

7.2.1 Smart, efficient, and scalable crawling

Here, the purpose is to identify target web sites to crawl, identify the kind of information to gather, and perform the actual initial crawling. Enterprise-grade crawling must meet the following requirements:

- Smart: Extracting all the relevant data and nothing more. Avoid getting stuck in infinite loops. For large repositories, start by crawling the category pages when possible (building a knowledge taxonomy in the process), then crawl the actual content. For the latter, assign a category, parent category, and category depth to each crawled URL. A web page may have multiple categories and parent categories. Choose an appropriate organization for the crawling algorithm: tree, graph, or stack, and heavy use of hash tables (key-value pairs).
- Efficient: You may start with a small crawl, for instance a subcategory, to optimize the parameters. Do not crawl too fast to avoid being blocked. Add a timeout to each URL request, to avoid getting stuck on occasions. Assign a status to each URL: queued (added to crawling list but not crawled yet), parsed (crawled), final (no downstream URLs attached to it), or failed. Failed requests can be re-tried later. Use a distributed architecture. Finally, you want to store the raw crawled content efficiently, typically in compressed format and organized for easy retrieval.

- Scalable: Using a stack architecture leads to simple distributed implementation and avoids recursivity. Saving each web page to a file right after being crawled, along with other summary statistics, provides the ability to easily resume from where your algorithm stopped in case of crash, as the crawling may last for months or years. Then you want to avoid being blocked by the target website or being able to recover if it happens. Using anonymous proxies may help with this. You may skip images and other binary files (video, audio) when crawling, if you don't plan on using them.

In this project, I crawled the entire Wolfram website, consisting of thousands of categories and about 15,000 web pages, totaling 150 MB in compressed format. Because web pages can be modified by the owner over time, my Python code may need to be adapted. However, I stored all the crawled content and full taxonomy, here. So, you can skip Part 1 (crawling) and move to Part 2 (building the LLM engine) if you experience crawling issues.

The project consists of the following steps:

Step 1: Introduction. The goal is to crawl mathworld.wolfram.com, more specifically the Probability & Statistics section. Also identify other relevant websites worth crawling. Besides the directory structure and actual content, what other types of pages should you consider, to build your LLM app? Which Python libraries should you use? How about using anonymous proxies and other strategies to avoid blocking when crawling?

Step 2: Initial taxonomy. Identify three types of pages: non-terminal category pages (pointing to subcategories), terminal category pages (pointing to content pages), and actual content pages. Reconstruct the full list of categories, creating a table such as the one featured in Figure 7.3. Produce a file with all the content pages found (URLs), with category, parent category, and category depth for each URL (one row per URL). These URLs will be crawled in the next step. Make sure that accented, non-English and special characters are preserved. Also, prove that the crawling algorithm used in the Python code, always ends without missing any URL.

Step 3: Extracting content. Crawl all the URLs obtained in the previous step. Use the smart, efficient, scalable methodology. The output should be text files, each one containing 500 URLs, with one (very long) line per URL, with the following fields: URL, category, parent category, category depth, and full content attached to the URL in question. Use appropriate separators for these fields. The HTML content, consisting of text, should have all line breaks (the `\n` character) replaced by (say) a space, so that one web page fits in one row. Make sure that accented, non-English and special characters are preserved.

Let's start with the answer to Step 1. Content worth crawling includes search result pages linked to pre-specified keywords (for instance, 'quantile' on Wolfram search, using this URL), website-targeted Google search (for instance, 'quantile + Wolfram.com' using this URL), exact search as opposed to broad search, lists of "related material" or "related questions" found on crawled pages, indexes or glossaries when available (see example here), metada [Wiki], and tags. All these elements help build relationships between keywords or concepts.

As for websites, consider crawling Wikipedia (with its own taxonomy), Stack Exchange (each website focusing on one topic, for instance mathematics), ArXiv, Google Scholar, and GitHub. Finally, I used the Requests Python library for crawling. For crawling examples with anonymous proxies, see here with the Torpy library, and here with the Bright Data library. The latter is not free.

Regarding Step 2, sample page types can be found here (non-terminal category page referred to as Type 1 in the code), here (terminal category page referred to as Type 2), and here (actual content page, with metadata at the top). The first two types require ad-hoc parsing to recursively build the categories and to extract the final URLs: the actual content pages. For output file management, note the `encoding='utf-8'` directive to handle non-standard characters, and the use of the `flush` command to avoid buffering.

As for Step 3, see the Python code below. Crawling the content pages (lines 135 – 171) can be done with a stand-alone script, as all the necessary input data is stored in a text file in lines 124 – 132. The output files are on GitHub, here. This repository contains sample logfiles monitoring crawling progress, the taxonomy structure featured in Figure 7.3, as well as the full text (HTML code) of all the content web pages. Both for the entire Wolfram website, and the Statistics & Probability category. The source code is also on GitHub, here.

As a final note, OpenAI uses the Common Crawl repository [Wiki]. This resource is claimed to contain "all the Internet" and it is also available as open source. You might consider it as an alternative to running your own crawling algorithm. Also, while OpenAI did not provide useful links to answer my questions, Edge (Bing) and Bard have that capability. Bing blends AI results with standard search result; in the AI-generated section, it provided this useful link to my question. I was less lucky with Bard. However, keep in mind that these apps keep being improved all the time. The tests reported here were performed in January 2024.

```
import requests
import time

URL_list = []
URL_parent_Category = {}
categoryLevel = {}
history = {}
final_URLs = {}

URL_base1 = "https://mathworld.wolfram.com/topics/" # for directory pages (root)
URL_base2 = "https://mathworld.wolfram.com/" # for final pages

seed_URL = "https://mathworld.wolfram.com/topics/ProbabilityandStatistics.html"
seed_category = "Probability and Statistics" # "Root" if starting at URL_base1
categoryLevel[seed_category] = 1 # set to 0 if starting at URL_base1

#seed_URL = "https://mathworld.wolfram.com/topics/"
#seed_category = "Root" # "Root" if starting at URL_base1
#categoryLevel[seed_category] = 0 # set to 0 if starting at URL_base1

URL_list.append(seed_URL) # URL stack
URL_parent_Category[seed_URL] = seed_category

parsed = 0   # number of URLs already parsed
n_URLs = 1   # total number of URLs in the queue
max_URLs = 5000 # do not crawl more than max_URLs directory pages

def validate(string):
   Ignore = ['about/','classroom/','contact/','whatsnew/','letters/']
   validated = True
   if len(string) > 60 or string in Ignore or string.count('topics') > 0:
      validated = False
   return(validated)

def update_lists(new_URL, new_category, parent_category, file):
   URL_parent_Category[new_URL] = new_category
   # if new_category was encountered before, special processing required
   #  --> in this case, not a one-to-one mapping (ignore here)
   categoryLevel[new_category] = 1 + categoryLevel[parent_category]
   level = str(categoryLevel[new_category])
   file.write(level+"\t"+new_category+"\t"+parent_category+"\n")
   file.flush()
   return()


#---[1] Creating category structure and list of webpages

# file1 allows you to resume from where it stopped in case of crash

file1 = open("crawl_log.txt","w",encoding="utf-8")
file2 = open("crawl_categories.txt","w",encoding="utf-8")

while parsed < min(max_URLs, n_URLs):

   URL = URL_list[parsed] # crawl first non-visited URL on the stack
   parent_category = URL_parent_Category[URL]
   level = categoryLevel[parent_category]
   time.sleep(5.0) # slow down crawling to avoid being blocked
   parsed += 1

   if URL in history:

      # do not crawl twice the same URL
      print("Duplicate: %s" %(URL))
      file1.write(URL+"\tDuplicate\t"+parent_category+"\t"+str(level)+"\n")
```

```
66
67      else:
68
69          print("Parsing: %5d out of %5d: %s" % (parsed, n_URLs, URL))
70          # req = requests.get(server, auth=('user','pass'))
71          resp = requests.get(URL, timeout=5)
72          history[URL] = resp.status_code
73
74          if resp.status_code != 200:
75
76              print("Failed: %s" %(URL))
77              file1.write(URL+"\tError:"+str(resp.status_code)+"\t"+
78                                          parent_category+"\t"+str(level)+"\n")
79              file1.flush()
80
81          else: # URL successfully crawled
82
83              file1.write(URL+"\tParsed\t"+parent_category+"\t"+str(level)+"\n")
84              page = resp.text
85              page = page.replace('\n', ' ')
86              page1 = page.split("<a href=\"/topics/")
87              page2 = page.split("<a href=\"/")
88              n_URLs_old = n_URLs
89
90              # scraping Type-1 page (intermediate directory node)
91
92              for line in page1:
93                  line = line.split("<span>")
94                  line = line[0]
95                  if line.count(">") == 1:
96                      line = line.split("\">")
97                      if len(line) > 1:
98                          new_URL = URL_base1 + line[0]
99                          new_category = line[1]
100                         URL_list.append(new_URL)
101                         update_lists(new_URL, new_category, parent_category, file2)
102                         file1.write(new_URL+"\tQueued\t"+new_category+"\t"+str(level+1)+"\n")
103                         file1.flush()
104                         n_URLs += 1
105
106             # scraping Type-2 page (final directory node)
107
108             if n_URLs == n_URLs_old:
109
110                 for line in page2:
111                     line = line.split("</a>")
112                     line = line[0].split("\">")
113                     if validate(line[0]) and len(line) > 1:
114                         new_URL = URL_base2 + line[0]
115                         new_category = line[1]
116                         update_lists(new_URL, new_category, parent_category, file2)
117                         file1.write(new_URL+"\tEndNode\t"+new_category+"\t"+str(level+1)+"\n")
118                         file1.flush()
119                         final_URLs[new_URL] = (new_category, parent_category, level+1)
120
121 file1.close()
122 file2.close()
123
124 # save list of final URLs to use in step [2]
125
126 count = 0
127 file = open("list_final_URLs.txt","w",encoding="utf-8")
128 for URL in final_URLs:
129     count += 1
130     file.write(str(count)+"\t"+URL+"\t"+str(final_URLs[URL])+"\t\n")
131 file.close()
```

```
print()


#---[2] Extracting content from final URLs

# file_log + file_input allows you to resume from where it stopped (in case of crash)

file_input = open("list_final_URLs.txt","r",encoding="utf-8")
file_log = open("crawl_content_log.txt","w",encoding="utf-8")
file_output = open("crawl_final.txt","w",encoding="utf-8")
separator = "\t~"

Lines = file_input.readlines()
file_input.close()
n = len(Lines) # number of final URLs (final pages)

for line in Lines:

   line = line.split("\t")
   count = int(line[0])
   URL = line[1]
   category = line[2]
   category.replace('\n','')
   print("Page count: %d/%d %s" %(count, n, URL))
   time.sleep(2.5) # slow down crawling to avoid being blocked

   resp = requests.get(URL, timeout=5)

   if resp.status_code == 200:
      # add page content: one line per page in the output file
      page = resp.text
      page = page.replace('\n', ' ')
      file_output.write(URL+"\t"+category+separator+page+"\t\n")
      file_output.flush()

   file_log.write(str(count)+"\t"+URL+"\t\n")
   file_log.flush()

file_log.close()
file_output.close()
https://drive.google.com/file/d/1H_xhfhzIPnO8oe9xlwCDWWM9OR5m81wd/view
```

7.2.2 User queries, category-specific embeddings and related tables

Extreme LLM (xLLM) is a multi-LLM architecture with a separate, specialized and simple LLM for each top category. Each LLM has its own set of tables: embeddings, subcategories, related contents, stopwords, URLs, stemmed words, and so on. Some tables are identical across all categories, for instance the mapping between accented characters and the ASCII version. The idea is to use the best Internet websites covering the entire human knowledge, reconstruct the structure and taxonomy of each website, blend the taxonomies, and extract condensed content. The other component consists of processing user queries and returning results in real-time, including links. The user selects or suggests top categories, along with his prompt, to get more relevant results, faster.

So far, I crawled the entire Wolfram website to cover all math and statistics categories. This was accomplished in project 7.2.1. Here I focus on one portion of this content: the top category "Probability & Statistics". The crawled data – the input to the Python code – is available as a text file named `crawl_final_stats.txt`, on Google Drive here, consisting of about 600 webpages.

The starting point is Appendix C, providing a lot more details and featuring `xllm5_util.py` (see section C.2), a utility with several functions used in this project as well as in project 7.2.3. This library is also on GitHub, here. The whole xLLM application is broken down into multiple parts, see Table 7.2.

Project 7.2.2 (this section) focuses on the third bullet point in Table 7.2. While xLLM may look like the most advanced and complicated project in this entire book, it is actually relatively simple and light. First, there is no neural network or training involved. Reinforcement learning is accomplished via self-tuning, described later,

relying on the most popular parameter sets based on usage. Then, the apparent complexity is due to minimal use of existing Python libraries. The reason is because I designed much more efficient, faster, smaller yet better solutions: for instance variable-length embeddings (VLE) encoded as hash tables (though graphs would be a great alternative), rather than vectors. Also, there are many other summary tables besides embeddings; these satellite tables are derived from the reconstructed taxonomy, and deliver the best improvements. By contrast, in traditional LLMs, embeddings are the central table. The way n-grams are treated, or the weights attached to tokens, are non-standard and work better, but require customized code rather than available libraries.

Component	Description
Web crawling	Project 7.2.1. It nncludes `crawl_directory.py`, also on GitHub, here.
Utilities	Appendix C. It includes `xllm5_util.py`, also on GitHub, here
Creating & updating tables (embeddings and so on), processing user queries	Project 7.2.2. Main program, for developers: `xllm5.py`, also on GitHub, here. See section 8.1.2.1 for details about embeddings and cosine distance.
Adding other sources (Wikipedia, books), processing user queries	Project 7.2.3. Based on `xllm5_short.py` (here on GitHub), a short version of `xllm5.py` that uses the summary tables created in project 7.2.2 (embeddings and so on), but does not create them.
Fast embedding search	A better alternative to vector search. See projects 2.3 and 8.1.
Glossary	Appendix B is a glossary focusing on LLM, and worth reading before moving forward.

Table 7.2: xLLM: projects breakdown

7.2.2.1 Project and solution

When I crawled mathworld.wolfram.com, I found different types of pages: hierarchical category listings including the home page (see here) and actual content pages (see here). I discuss the details in Step 2 in project 7.2.1. Content pages have important navigation features, such as "related pages" and "See also" references. I isolated them to build special summary tables, besides embeddings. Many websites, for instance Wikipedia, have a very similar structure.

See Figure 8.1 for embedding examples. I added words and tokens found in categories and other navigation features, to the main summary tables including embeddings. Parsing content pages and extraction of these navigation features is performed in the `split_page` function, line 76 in the python code in section 7.2.2.2. For partial xLLM sample output – extract from an answer to a user query – see Figure C.1. Embeddings are not included.

The goal here is not to create an LLM from scratch, but to understand the different components, adapt it to other sources, augment my current version of xLLM with extra sources, and improve some of the features. Thus, you should start by looking at the code `xllm5.py` in section 7.2.2.2 as well as the accompanying library `xllm5_util.py` in section C.2.

The project consists of the following steps:

Step 1: Summary tables. Besides embeddings, what are the summary tables and underlying data structure, for each of them? How are they linked to the output returned to a user query? How to adapt the methodology if the input source is Wikipedia rather than Wolfram?

Step 2: Best practices. By looking at my comments in the code or by experimenting yourself, identify bad side effects from standard Python libraries such as NLTK. Examples include stopwords, auto-correct, and singularize. The problems arise due to the specialization of my xLLM, but is otherwise minor in generic applications targeted to non-experts. Suggest workarounds.

Step 3: Improving xLLM. I implemented several enhancements, for instance: ignoring N-grams (permutations of N tokens) [Wiki] not found in the crawled data, using normalized PMI in embeddings as the association metric or weight between two tokens, working with variable-length embeddings (VLE) rather than vectors, managing accented characters, not combining tokens separated by punctuation signs, and minimizing stemming such as singular form. Find other potential enhancements and how to implement them. For instance, making sure that "analysis of variance" and "ANOVA" return the same results, or "San Francisco" is not split into two tokens. Hint: use mappings (synonyms) and double-tokens treated as a single tokens. Treat uppercase and lowercase differently.

Step 4: Taxonomy creation. The architecture of xLLM relies heavily on having a great categorization of the crawled content. In the case of Wolfram, the underlying man-made taxonomy (the category hierarchy) can easily be reconstructed. How would you proceed if you do not find any pre-built taxonomy? How to build one from scratch? How to detect subcategories that should not be included? In the case of Wolfram, a sub-category can have multiple parent categories, albeit rarely. How do you handle this?

Step 5: Parameters and evaluation. In the code, what are compressed N-grams and their purpose? How are user queries matched to embeddings and other back-end tables? Is there any training in the algorithm? How would you evaluate the results? What are the main parameters? Hint: allow some users to play with the parameters; automatically detect the most popular values across these users; popular values become the default setting. For customized results, allow users to keep playing with their favorite values. The key concept here is self-tuning.

Step 6: Monetization. How would you monetize the app, while keeping it free and open-source?

I now answer my own questions, starting with Step 1. For summary tables, see lines 22-38 in `xllm5_short.py` in section 7.2.3, and the corresponding text files (saved tables) on GitHub, here. To check out the size, format, or key-value elements for hash tables, click on the corresponding file. For hash tables, the key can be a token as in `word_list`, or a word (a combination of up to 4 tokens, separated by the tilde symbol) as in `dictionary` and `ngram`. In some cases (`embeddings`, `related`) the value is also a hash table, or a list (`ngram`). There are about 3500 valid tokens, and 46,000 valid words in the dictionary table, for the top-category that I am interested in.

To process a user query, see the code starting at line 371 in section 7.2.2.2. The main function for this task is `process_query` (line 330). First, I transform the query in the same way the crawled data was originally transformed (auto-correct, removing stopwords, and so on), and then sort it by token: the query "normal and Poisson distributions" becomes "distribution normal poisson". I ignore tokens not in `dictionary`, and then generate all token combinations (lines 337-346). In this case: distribution, normal, poisson, distribution normal, distribution poisson, normal poisson, distribution normal poisson. I then match these combinations with keys in the compressed N-gram table. For instance, "distribution normal" has an entry (key) and the corresponding value is "normal distribution" (the most common N-gram for this word, as found when crawling the data). That is, "normal distribution" is also a key in the summary tables (at least some of them). In particular, I can retrieve links, embeddings, categories, and related content associated to "normal distribution", respectively in the tables `url_map`, `hash_embeddings`, `hash_category`, and `hash_related`.

Finally, the Wikipedia website also has a structure quite similar to Wolfram. For the machine learning top category, see the directory page, here: this is the starting point for crawling, visiting content pages, and to reconstruct the taxonomy. Like Wolfram, content pages sometimes have "See also" navigation links, and related categories are listed at the bottom of the page. For sample code crawling Wikipedia, see here.

Now moving to Step 2. Examples of problems are auto-correct libraries erroneously changing "feller" to "seller", or "hypothesis" singularized to "hypothesi", or "even" flagged as a stopword by some libraries and thus removed, while you need it to distinguish between even and odd functions in your specialized LLM. San Francisco should map to one single token, not two. To avoid these problems, do not singularize a word in the user query if the plural (but not the presumed singular) is found in the dictionary table. Do not auto-correct proper names such as "Feller". Create a table of words that can not be corrected or singularized, especially high frequency words such as "hypothesis". Find words frequently associated, such as "San" and "Francisco", or use available tables. And build your own stopword list. Stopwords, do-not-stem, do-not-auto-correct, and do-not-singularize tables should be specific to a top category, that is, to a sub LLM in your multi-LLM architecture.

To answer Step 3, I recommend creating a synonym dictionary. It would have an entry mapping "ANOVA" to "analysis of variance", and conversely. You can build such a dictionary by browsing glossaries, or use one from a Python library. Then allowing tokens to be double, for instance considering "Gaussian", "distribution" and "Gaussian+distribution" as three tokens. The latter is a double token and represents two terms frequently found together. In this case, "Gaussian" and "distribution" are both tokens on their own, and sub-tokens.

Regarding Step 4, you can build a taxonomy by identifying the top 200 words (up to 4 tokens) in the crawled sources, and manually assign labels to those that best represent a top category. At level two, look at top words attached to a specific top category (using embeddings or other tables), to create level-2 sub-categories. Do the same for level-3 and so on. The process can be automated to some extent. To give you an idea, the Wolfram taxonomy has about 5000 categories.

Note that the pre-built Wolfram taxonomy is not a tree: some sub-categories have multiple parent categories. Use a graph rather than a tree to represent such structures. Finally, some sub-categories such as "linear algebra", that naturally fit under "Probability & Statistics" (the top category in this project), are found under the top category "Algebra" in the Wolfram taxonomy. However, the "Algebra" category is filled with irrelevant

material ("Lie algebra"), so you want to eliminate all top categories from "Algebra" that are not relevant to statistics. Then, the Wolfram sub-category "Bayesian statistics" is rather weak: you need to augment it with other sources, such as Wikipedia. If you want to add "gradient descent" to "Probability & Statistics", you need to look at the top category "Calculus".

Part of the answer to Step 5 (N-grams and matching user queries to summary tables) is discussed in the second paragraph in my solution to Step 1. The compressed N-gram table is a key-value hash, where the key is an ordered N-gram (with tokens in alphabetical order) and the value is a list of non-ordered versions of the N-gram in question, found in the sources during crawling. Thus these non-ordered versions or natural words are present as keys in the `dictionary` and satellite summary key-value tables. To check if a user query matches some words in the summary tables, extract ordered N-grams from the query, look for them in the compressed N-gram table, and from there find the associated non-ordered versions. Finally, these non-ordered versions exist as keys in the summary tables.

There is no actual training in the algorithm. The idea is to identify the best parameters based on usage, and set them as default. Important parameters are the minimum counts or thresholds to decide whether or not to include some entries in the output results. For instance: `ccnt1`, `ccnt2`, and `maxprint` in lines 360-362. The max number of tokens per word is set to 4, but could be changed. Separators include comma and point, but could be expanded. Instead of `pmi`, you can use `word2_weight` in line 321 in `xllm5_util.py` (see code in section C.2). Or you can change the `exponent` value in the computation of `pmi` (same code, line 310). If you use multiple sources, say Wolfram and Wikipedia, you can assign a separate trust score to each one and allow the user to fine-tune it. Finally, you can offer the user to choose between different stopword lists.

Finally, for Step 6 (monetization), there are two options. First, blend organic results with relevant, sponsored links from advertisers. Then offer a paid version with API access, and larger summary tables (compared to the free version) based on additional input sources. Or give access to fresh summary tables (regularly updated) and parameter tuning, to paid customers only.

7.2.2.2 Python code

The program, called `xllm5.py`, is also on GitHub, here. The input file `crawl_final_stats.txt` (Wolfram crawled pages, "Probability & Statistics" top category) is on my Google drive, here. Then the `xllm5_util.py` library is discussed in Appendix C.2, and also on GitHub, here.

```
import numpy as np
import xllm5_util as llm5
from autocorrect import Speller
from pattern.text.en import singularize


#--- [1] some utilities

# words can not be any of these words
stopwords = ( "of", "now", "have", "so", "since", "but", "and",
            "thus", "therefore", "a", "as", "it", "then", "that",
            "with", "to", "is", "will", "the", "if", "there", "then,",
            "such", "or", "for", "be", "where", "on", "at", "in", "can",
            "we", "on", "this", "let", "an", "are", "has", "how", "do",
            "each", "which", "nor", "any", "all", "al.", "by", "having",
            "therefore", "another", "having", "some", "obtaining",
            "into", "does", "union", "few", "makes", "occurs", "were",
            "here", "these", "after", "defined", "takes", "therefore,",
            "here,", "note", "more", "considered", "giving", "associated",
            "etc.", "i.e.,", "Similarly,", "its", "from", "much", "was",
            "given", "Now,", "instead", "above,", "rather", "consider",
            "found", "according", "taking", "proved", "now,", "define",
            "showed", "they", "show", "also", "both", "must", "about",
            "letting", "gives", "their", "otherwise", "called", "descibed",
            "related", "content", "eg", "needed", "picks", "yielding",
            "obtained", "exceed", "until", "complicated", "resulting",
            "give", "write", "directly", "good", "simply", "direction",
            "when", "itself", "ie", "al", "usually", "whose", "being",
            "so-called", "while", "made", "allows", "them", "would", "keeping",
            "denote", "implemented", "his", "shows", "chosen", "just",
            "describes", "way", "stated", "follows", "approaches", "known",
            "result", "sometimes", "corresponds", "every", "referred",
```

```
            "produced", "than", "may", "not", "exactly", " ", "whether",
            "illustration", ",", ".", "...", "states", "says", "known", "exists",
            "expresses", "respect", "commonly", "describe", "determine", "refer",
            "often", "relies", "used", "especially", "interesting", "versus",
            "consists", "arises", "requires", "apply", "assuming", "said",
            "depending", "corresponding", "calculated", "depending", "associated",
            "corresponding", "calculated", "coincidentally", "becoming", "discussion",
            "varies", "compute", "assume", "illustrated", "discusses", "notes",
            "satisfied", "terminology", "scientists", "evaluate", "include", "call",
            "implies", "although", "selected", "however", "between", "explaining",
            "featured", "treat", "occur", "actual", "authors", "slightly",
            "specified"
          )

# map below to deal with some accented / non-standard characters
utf_map = { " " : " ",
          "&oacute;" : "o",
          "&eacute;" : "e",
          "&ouml;" : "o",
          "&ocirc;" : "o",
          "&#233;" : "e",
          "&#243;" : "o",
          " "      : " ",
          "'s"     : "", # example: Feller's --> Feller
        }

def get_top_category(page):

    # useful if working with all top categories rather than just one
    # create one set of tables (dictionary, ngrams...) for each top category
    # here we mostly have only one top category: 'Probability & Statistics'
    # possible cross-links between top categories (this is the case here)

    read = (page.split("<ul class=\"breadcrumb\">"))[1]
    read = (read.split("\">"))[1]
    top_category = (read.split("</a>"))[0]
    return(top_category)


def trim(word):
    return(word.replace(".", "").replace(",",""))


def split_page(row):

    line = row.split("<!-- Begin Content -->")
    header = (line[0]).split("\t~")
    header = header[0]
    html = (line[1]).split("<!-- End Content -->")
    content = html[0]
    related = (html[1]).split("<h2>See also</h2>")
    if len(related) > 1:
        related = (related[1]).split("<!-- End See Also -->")
        related = related[0]
    else:
        related = ""
    see = row.split("<p class=\"CrossRefs\">")
    if len(see) > 1:
        see = (see[1]).split("<!-- Begin See Also -->")
        see = see[0]
    else:
        see = ""
    return(header, content, related, see)


def list_to_text(list):
```

```
    text = " " + str(list) + " "
    text = text.replace("'", " ")
    text = text.replace("\"", " ")
    # text = text.replace("-", " ")
    text = text.replace("(", "( ")
    text = text.replace(")", ". )").replace(" ,",",")
    text = text.replace(" |",",").replace(" |",",")
    text = text.replace(" .", ".")
    text = text.lower()
    return(text)


#--- [2] Read Wolfram crawl and update main tables

file_html = open("crawl_final_stats.txt","r",encoding="utf-8")
Lines = file_html.readlines()

# unless otherwise specified, a word consists of 1, 2, 3, or 4 tokens

dictionary = {}       # words with counts
word_pairs = {}       # pairs of 1-token words found in same word, with count
url_map = {}          # URL IDs attached to words in dictionary
arr_url = []          # maps URL IDs to URLs (one-to-one)
hash_category = {}    # categories attached to a word
hash_related = {}     # related topics attached to a word
hash_see = {}         # topics from "see also" section, attached to a word
word_list = {}        # list of 1-token words associated to a 1-token word

url_ID = 0 # init for first crawled page

# process content from Wolfram crawling, one page at a time
# for Probability & Statistics category

for row in Lines:

   #-- cleaning; each row is a full web page + extra info

   category = {}

   for key in utf_map:
      row = row.replace(key, utf_map[key])

   (header, content, related, see) = split_page(row)
   url = (header.split("\t"))[0]
   cat = (header.split("\t"))[1]
   cat = cat.replace(",", " |").replace("(","").replace(")","")
   cat = cat.replace("'","").replace("\"","")
   category[cat] = 1

   # top_category not always "Probability & Statistics"
   top_category = get_top_category(row)

   # processing "related content" on web page
   list = related.split("\">")
   related = ()
   for item in list:
      item = (item.split("<"))[0]
      if item != "" and "mathworld" not in item.lower():
         related = (*related, item)

   # processing "see also" on web page
   if see != "":
      list = see.split("\">")
      see = ()
      for item in list:
         item = (item.split("<"))[0]
```

```
165           if item != "" and item != " ":
166              see = (*see, item)
167
168     text_category = list_to_text(category)
169     text_related = list_to_text(related)
170     text_see = list_to_text(see)
171     content += text_category + text_related + text_see
172
173     # skip all chars between 2 quotes (it's just HTML code)
174     flag = 0
175     cleaned_content = ""
176     for char in content:
177        if char == "\"":
178           flag = 1 - flag
179        if flag == 0:
180           cleaned_content += char
181
182     cleaned_content = cleaned_content.replace(">", "> ")
183     cleaned_content = cleaned_content.replace("<", ". <")
184     cleaned_content = cleaned_content.replace("(", "( ")
185     cleaned_content = cleaned_content.replace(")", ". )")
186     cleaned_content = cleaned_content.lower()
187     data = cleaned_content.split(" ")
188     stem_table = llm5.stem_data(data, stopwords, dictionary)
189
190     # update tables after each parsed webpage
191     # data is an array containing cleaned words found in webpage
192
193     url_ID = llm5.update_core_tables(data, dictionary, url_map, arr_url, hash_category,
194                               hash_related, hash_see, stem_table, category, url,
195                               url_ID, stopwords, related, see, word_pairs, word_list)
196
197
198 #--- [3] create embeddings and ngrams tables, once all sources are parsed
199
200 pmi_table         = llm5.create_pmi_table(word_pairs, dictionary)
201 embeddings        = llm5.create_embeddings(word_list, pmi_table)
202 ngrams_table      = llm5.build_ngrams(dictionary)
203 compressed_ngrams_table = llm5.compress_ngrams(dictionary, ngrams_table)
204
205
206 #--- [4] print some stats (utilities)
207
208 def hprint(title, hash, maxprint, output_file, format = "%3d %s"):
209    print("\n%s\n" %(title))
210    output_file.write("\n%s\n\n" %(title))
211    hash_sorted = dict(sorted(hash.items(),
212                   key=lambda item: item[1], reverse=True))
213    printcount = 0
214    for item in hash_sorted:
215       value = hash[item]
216       if "URL" in title:
217          item = arr_url[int(item)]
218       if item != "" and printcount < maxprint:
219          print(format % (value, item))
220          output_file.write(str(value) + " " + str(item) + "\n")
221          printcount += 1
222
223    return()
224
225
226 def word_summary(word, ccnt1, ccnt2, maxprint, output_file):
227
228    if word not in dictionary:
229       print("No result")
230       output_file.write("No result\n")
```

```
231         cnt = 0
232     else:
233         cnt = dictionary[word]
234 
235     if cnt > ccnt1:
236 
237         dashes = "-" * 60
238         print(dashes)
239         output_file.write(dashes + "\n")
240         print(word, dictionary[word])
241         output_file.write(word + " " + str(dictionary[word]) + "\n")
242 
243         hprint("ORGANIC URLs", url_map[word], maxprint, output_file)
244         hprint("CATEGORIES & LEVELS", hash_category[word], maxprint, output_file)
245         hprint("RELATED", hash_related[word], maxprint, output_file)
246         hprint("ALSO SEE", hash_see[word], maxprint, output_file)
247 
248         if word in word_list and word in embeddings:
249 
250             # print embedding attached to word
251 
252             hash = {}
253             weight_list = llm5.collapse_list(word_list[word])
254             embedding_list = embeddings[word]
255 
256             for word2 in embedding_list:
257               if word2 != word:
258                  pmi = embedding_list[word2]
259                  count = weight_list[word2]
260                  product = pmi * count
261                  hash[word2] = product
262 
263             hprint("EMBEDDINGS", hash, maxprint, output_file, "%8.2f %s")
264 
265     print()
266     return()
267 
268 
269 #--- [5] main loop
270 
271 def save_tables():
272 
273     list = { "dictionary" : dictionary,
274              "ngrams_table" : ngrams_table,
275              "compressed_ngrams_table" : compressed_ngrams_table,
276              "word_list" : word_list,
277              "embeddings" : embeddings,
278              "url_map" : url_map,
279              "hash_category" : hash_category,
280              "hash_related" : hash_related,
281              "hash_see" : hash_see,
282            }
283 
284     for table_name in list:
285         file = open("xllm5_" + table_name + ".txt", "w")
286         table = list[table_name]
287         for word in table:
288             file.write(word + "\t" + str(table[word]) + "\n")
289         file.close()
290 
291     file = open("xllm5_arr_url.txt", "w")
292     for k in range(0, len(arr_url)):
293         file.write(str(k) + "\t" + arr_url[k] + "\n")
294     file.close()
295 
296     file = open("stopwords.txt","w")
```

```
    file.write(str(stopwords))
    file.close()
    file = open("utf_map.txt","w")
    file.write(str(utf_map))
    file.close()
    return()


dump = False # to save all potential query results (big file)
save = True # to save all tables

if save:
    save_tables()

if dump:

    # hperparameters
    ccnt1 = 0 # 5
    ccnt2 = 0 # 1
    maxprint = 200 # up to maxprint rows shownn per word/section

    dump_file = open("xllm5_dump.txt", "w")
    for word in dictionary:
        word_summary(word, ccnt1, ccnt2, maxprint, dump_file)
    dump_file.close()

print("Words (up to 4 tokens):", len(dictionary))
print("1-token words with a list:", len(word_list))
print("1-token words with an embedding:", len(embeddings))


#--- [6] process sample user queries

def process_query(query, ccnt1, ccnt2, maxprint, output_file = ""):
    # query is a sorted word, each token is in dictionary
    # retrieve all sub-ngrams with a dictionary entry, print results for each one

    get_bin = lambda x, n: format(x, 'b').zfill(n)
    n = len(query)

    for k in range(1, 2**n):

        binary = get_bin(k, n)
        sorted_word = ""
        for k in range(0, len(binary)):
            if binary[k] == '1':
                if sorted_word == "":
                    sorted_word = query[k]
                else:
                    sorted_word += "~" + query[k]

        if sorted_word in compressed_ngrams_table:
            list = compressed_ngrams_table[sorted_word]
            # the word below (up to 4 tokens) is in the dictionary
            word = list[0]
            print("Found:", word)
            output_file.write("Found:" + word + "\n")
            word_summary(word, ccnt1, ccnt2, maxprint, output_file)

    return()


# hyperparameters
ccnt1   = 0
ccnt2   = 0  # 2
maxprint = 10 # up to maxprint rows shownn per word/section
```

```
spell = Speller(lang='en')


query = " "
print("\n")
output_file = open("xllm5_results.txt", "w")

while query != "":

   # entries separated by commas are treated independently
   query = input("Enter queries (ex: Gaussian distribution, central moments): ")
   queries = query.split(',')
   token_list = []
   token_clean_list = []

   for query in queries:

      tokens = query.split(' ')
      for token in tokens:
         # note: spell('feller') = 'seller', should not be autocorrected
         token = token.lower()
         if token not in dictionary:
            token = spell(token)
         token_list.append(token)
      stemmed = llm5.stem_data(token_list, stopwords, dictionary, mode = 'Internal')

      for old_token in stemmed:
         token = stemmed[old_token]
         if token in dictionary:
            token_clean_list.append(token)
      token_clean_list.sort()

      if not token_clean_list:
         if query != "":
            print("No match found")
            output_file.write("No match found\n")
      else:
         print("Found: ", token_clean_list)
         output_file.write("Found: " + str(token_clean_list) + "\n")
         process_query(token_clean_list, ccnt1, ccnt2, maxprint, output_file)

output_file.close()
```

7.2.3 RAG: retrieval augmented generation using book catalogs

The purpose is to augment the xLLM system introduced in section 7.2.2, with external input data consisting of books. The focus is still on one particular sub-LLM: the "probability & statistics" category, initially built on the Wolfram crawl. The additional input (the books) leads to an alternate taxonomy, complementing sub-categories that don't have much coverage in Wolfram, such as "Bayesian analysis". The alternate taxonomy based on the books may lead to a separate sub-LLM, or used for fusion with the existing one built earlier. Other input sources to augment the back-end tables include the metadata attached to crawled documents, and the front-end user prompts. For more on LLM fusion, see [34].

In the process, I also create a synonyms table built on glossaries and other structure found in the books, to further enhance output results to user prompts. Then, I introduce double-tokens and scores: the latter measures the relevancy of the results returned to the user, for a specific query. Finally, I further investigate self-tuning, a better alternative to standard LLM evaluation metrics described in [28].

The project consists of the following steps:

Step 1: xLLM pluses and minuses. The benefits of xLLM are listed and contrasted to standard LLMs in appendix C.3. What are the potential negatives?

Step 2: Data augmentation. How to incorporate additional input sources such as PDFs? In short, explain how to reconstruct the underlying taxonomy of a PDF repository, and blend the augmented data

with the existing summary tables built in Step 1 in project 7.2.2. What other sources could be used? The LaTeX sources of my books are on GitHub, here. The PDF version of this document can be found here. In these documents, what elements would you use to create / update the summary tables such as categories or embeddings? These additional input sources (PDFs and so on) are referred to as augmented data, corresponding to the letter **A** in the RAG architecture (retrieval-augmentation-generation).

Step 3: Scoring, evaluation, self-tuning. The results displayed to the user, broken down per section (URLs, categories, tokens, related items and so on as in Figure C.1) are sorted according to relevancy: see the numbers on the left column, attached to each item in Figure C.1. Lines 131-147 in the Python script `xllm5_short.py` at the end of this section, take care of this.

How to create a global quality score attached to the output displayed to the user, based on the various relevancy scores assigned to each returned item? How do you weight the various sources (original crawl, augmented data such as PDFs)? Finally, if you allow the user to play with the hyperparameters (see Step 5 in project 7.2.2), you can deliver customized results: two users with the same query get different outputs. Discuss the challenges of creating a universal evaluation metric to compare various LLMs such as OpenAI, Mistral, Claude, xLLM and so on. How would you identify the optimum set of parameters and set it as default, based on user preferences? That is, based on the individual favorite hyperparameter sets selected by the users.

Step 4: Enhancements. Some queries such as "Bayes" or "Anova" return very little, while "Bayesian" or "analysis of variance" return a lot more. How to address this issue? How do you improve or build a good stopwords list for a specific top category? Finally, what are the benefits of double tokens such as "Gaussian+distribution", in addition to the simple tokens "Gaussian" and "distribution"? How to implement double tokens in embeddings and other tables?

Step 5: Combining multiple specialized LLMs. How do you manage multiple LLMs, one per top category? The solution here is known as multi-agent system.

I now offer my answers, starting with Step 1. Not working with the right in-house experts (people who know your repository very well and what users are looking for) may result in poor design. Not enough testing may lead to many bugs. You must create a list of who can access what to avoid data leaks and security issues. Other issues: poor documentation; the expert engineer who knows all the details leaves your company.

Now my answer to Step 2. To augment the Wolfram "Probability & Statistics" top category (the specialized sub-LLM of interest here), you can parse my statistics books: either the PDF documents, or the LaTeX sources that I share on GitHub, here. For PDF documents, you can use Python libraries to retrieve tables, images, and content: see my sample code here, based on the PyPDF2 library. But I recommend working with the LaTeX sources, for two reasons. First, the process that produces the PDFs may result in information loss. Then, it may be easier to retrieve the hidden structure from the original sources: index keywords, glossary terms, table of contents, bibliography items and references, formatting including color (for instance to separate standard text from Python code elements) as well as titles, sections and subsections to help build a taxonomy.

With the newest version of xLLM, it is easier to reconstruct the underlying taxonomy, or build one from scratch. See details in section C.4. You may use the existing Wolfram taxonomy to match multi-token words or some n-gram permutation found in the LaTeX sources, with related Wolfram-based x-embeddings entries in the `embeddings2` table, and their corresponding category in the `hash_category` table. More specifically, for $\texttt{word}_1$ found in the source text, look for all $\texttt{word}_2$ found in $\texttt{embedding2[word}_1\texttt{]}$, and then look for $\texttt{hash\_category[word}_1\texttt{]}$ and $\texttt{hash\_category[word}_2\texttt{]}$. Note that x-embeddings (the `embeddings2` table) are available in xLLM6 discussed in section C.4, not in the xLLM5 code in this section. Words with high count and not found in Wolfram-based tables may lead to new categories or sub-categories. The `compressed_ngrams_table` indexed by sorted n-grams, helps you retrieve existing token permutations found in Wolfram tables, when the original word is not present in these tables.

Finally, other input sources for data augmentation include metadata, synonyms dictionaries, Wikipedia (featuring a taxonomy like Wolfram, see here) and the prompts entered by the users. Some Wikipedia sub-categories are irrelevant and should be skipped, otherwise it will add noise in the back-end tables. Some are missing.

Regarding Step 3, different types of users (expert versus layman) may rate the results differently. To build a default set of hyperparameters, look at all the values favored by your users. This assumes that users can select hyperparameters to fine-tune results according to their liking. Based on these choices, perform smart grid search using the technique described in section 5.2.3.3.

As for evaluating the results, in the output displayed to the user, each item in each section has a relevancy scored z shown on the left: see example in Figure C.5. The first step is to normalize these scores, for instance with the transform $z \mapsto 1 - \exp(-\lambda z)$ where $\lambda > 0$ is an hyperparameter. The normalized score is then between

0 and 1. Compute an aggregate score per section, also taking into account the total number of items returned for a specific section. Then compute a global score for each query, as a weighted average over the sections. Use a different weight for each section. More about LLM evaluation can be found in [28] and [31], published in 2024. See also the GitHub repository on this topic, here.

Now moving to Step 4 and Step 5. Several major enhancements are discussed in section C.4, featuring the differences between xLLM5 (the Python code in this section) and xLLM6. To deal with multiple specialized LLMs – one per top category – do the same separately for each of the 15 top categories on Wolfram. Here I focused on just one of them. Allow the user to specify a category along with his query, so that retrieving the content from the backend tables (x-embeddings and so on) is focused and fast, boosting relevancy. There will be redundancy among the various LLMs. Also, a top layer can be added to jointly manage the separate LLMs. This is accomplished with a multi-agent system architecture.

You may use the `xllm5_short.py` program below to do various testing and implement the enhancements discussed in this section. The code is also on GitHub, here. For the most recent version `xllm6_short.py`, see section C.4.

```
# xllm5_short.py : Extreme LLM (light version), vincentg@mltechniques.com

import requests
from autocorrect import Speller
from pattern.text.en import singularize

# Unlike xllm5.py, xllm5_short.py does not process the (huge) crawled data.
# Instead, it uses the much smaller summary tables produced by xllm5.py

#--- [1] get tables if not present already

# First, get xllm5_util.py from GitHub and save it locally as xllm5_util.py
#    note: this python code does that automatically for you
# Then import everything from that library with 'from xllm5_util import *'
# Now you can call the read_xxx() functions from that library
# In addition, the tables stopwords and utf_map are also loaded
#
# Notes:
#   - On first use, dowload all locally with overwrite = True
#   - On next uses, please use local copies: set overwrite = False

# Table description:
#
# unless otherwise specified, a word consists of 1, 2, 3, or 4 tokens
# word_pairs is used in xllm5.py, not in xllm5_short.py
#
# dictionary = {} words with counts: core (central) table
# word_pairs = {} pairs of 1-token words found in same word, with count
# url_map = {}    URL IDs attached to words in dictionary
# arr_url = []    maps URL IDs to URLs (one-to-one)
# hash_category = {} categories attached to a word
# hash_related = {} related topics attached to a word
# hash_see = {}   topics from "see also" section, attached to word
# word_list = {}  list of 1-token words associated to a 1-token word
# ngrams_table = {} ngrams of word found when crawling
# compressed_ngrams_table = {} only keep ngram with highest count
# utf_map = {}    map accented characters to non-accented version
# stopwords = ()  1-token words not accepted in dictionary

path =
    "https://raw.githubusercontent.com/VincentGranville/Large-Language-Models/main/llm5/"

overwrite = False

if overwrite:

   response = requests.get(path + "xllm5_util.py")
   python_code = response.text

```

```
    local_copy = "xllm5_util"
    file = open(local_copy + ".py", "w")
    file.write(python_code)
    file.close()

    # get local copy of tables

    files = [ 'xllm5_arr_url.txt',
            'xllm5_compressed_ngrams_table.txt',
            'xllm5_word_list.txt',
            'xllm5_dictionary.txt',
            'xllm5_embeddings.txt',
            'xllm5_hash_related.txt',
            'xllm5_hash_category.txt',
            'xllm5_hash_see.txt',
            'xllm5_url_map.txt',
            'stopwords.txt'
          ]

    for name in files:
       response = requests.get(path + name)
       content = response.text
       file = open(name, "w")
       file.write(content)
       file.close()

import xllm5_util as llm5

# if path argument absent in read_xxx(), read from GitHub
# otherwise, read from copy found in path

arr_url    = llm5.read_arr_url("xllm5_arr_url.txt", path="")
dictionary = llm5.read_dictionary("xllm5_dictionary.txt", path="")
stopwords  = llm5.read_stopwords("stopwords.txt", path="")

compressed_ngrams_table = llm5.read_table("xllm5_compressed_ngrams_table.txt",
                                          type="list", path="")
word_list  = llm5.read_table("xllm5_word_list.txt", type="list", path="")
embeddings = llm5.read_table("xllm5_embeddings.txt", type="hash", path="",
                                                format="float")
hash_related = llm5.read_table("xllm5_hash_related.txt", type="hash", path="")
hash_see   = llm5.read_table("xllm5_hash_see.txt", type="hash", path="")
hash_category = llm5.read_table("xllm5_hash_category.txt", type="hash", path="")
url_map    = llm5.read_table("xllm5_url_map.txt", type="hash", path="")


#--- [2] some utilities

def singular(data, mode = 'Internal'):

    stem_table = {}

    for word in data:
       if mode == 'Internal':
          n = len(word)
          if n > 2 and "~" not in word and \
               word[0:n-1] in dictionary and word[n-1] == "s":
             stem_table[word] = word[0:n-1]
          else:
             stem_table[word] = word
       else:
          # the instruction below changes 'hypothesis' to 'hypothesi'
          word = singularize(word)

          # the instruction below changes 'hypothesi' back to 'hypothesis'
          # however it changes 'feller' to 'seller'
```

```
115           # solution: create 'do not singularize' and 'do not autocorrect' lists
116           stem_table[word] = spell(word)
117
118     return(stem_table)
119
120
121 #--- [3] print some stats (utilities)
122
123 def fformat(value, item, format):
124     format = format.split(" ")
125     fmt1 = format[0].replace("%","")
126     fmt2 = format[1].replace("%","")
127     string = '{:{fmt1}} {:{fmt2}}'.format(value,item,fmt1=fmt1,fmt2=fmt2)
128     return(string)
129
130
131 def hprint(title, hash, maxprint, output_file, format = "%3d %s"):
132     print("\n%s\n" %(title))
133     output_file.write("\n%s\n\n" %(title))
134     hash_sorted = dict(sorted(hash.items(),
135                     key=lambda item: item[1], reverse=True))
136     printcount = 0
137     for item in hash_sorted:
138         value = hash[item]
139         if "URL" in title:
140             item = arr_url[int(item)]
141         if item != "" and printcount < maxprint:
142             print(format % (value, item))
143             string = fformat(value, item, format)
144             output_file.write(string + "\n")
145             printcount += 1
146
147     return()
148
149
150 def word_summary(word, ccnt1, ccnt2, maxprint, output_file):
151
152     if word not in dictionary:
153         print("No result")
154         output_file.write("No result\n")
155         cnt = 0
156     else:
157         cnt = dictionary[word]
158
159     if cnt > ccnt1:
160
161         dashes = "-" * 60
162         print(dashes)
163         output_file.write(dashes + "\n")
164         print(word, dictionary[word])
165         output_file.write(word + " " + str(dictionary[word]) + "\n")
166
167         hprint("ORGANIC URLs", url_map[word], maxprint, output_file)
168         hprint("CATEGORIES & LEVELS", hash_category[word], maxprint, output_file)
169         hprint("RELATED", hash_related[word], maxprint, output_file)
170         hprint("ALSO SEE", hash_see[word], maxprint, output_file)
171
172         if word in word_list and word in embeddings:
173
174             # print embedding attached to word
175
176             hash = {}
177             weight_list = llm5.collapse_list(word_list[word])
178             embedding_list = embeddings[word]
179
180             for word2 in embedding_list:
```

```
            if word2 != word:
               pmi = embedding_list[word2]
               count = weight_list[word2]
               product = pmi * count
               hash[word2] = product

         hprint("EMBEDDINGS", hash, maxprint, output_file, "%8.2f %s")

   print()
   return()


#--- [4] main loop

dump = False

if dump:

   # hperparameters
   ccnt1 = 0 # 5
   ccnt2 = 0 # 1
   maxprint = 200 # up to maxprint rows shownn per word/section

   dump_file = open("xllm5_dump.txt", "w")
   for word in dictionary:
      word_summary(word, ccnt1, ccnt2, maxprint, dump_file)
   dump_file.close()

print("Words (up to 4 tokens):", len(dictionary))
print("1-token words with a list:", len(word_list))
print("1-token words with an embedding:", len(embeddings))


#--- [5] process sample user queries

def process_query(query, ccnt1, ccnt2, maxprint, output_file = ""):
   # query is a sorted word, each token is in dictionary
   # retrieve all sub-ngrams with a dictionary entry, print results for each one

   get_bin = lambda x, n: format(x, 'b').zfill(n)
   n = len(query)

   for k in range(1, 2**n):

      binary = get_bin(k, n)
      sorted_word = ""
      for k in range(0, len(binary)):
         if binary[k] == '1':
            if sorted_word == "":
               sorted_word = query[k]
            else:
               sorted_word += "~" + query[k]

      if sorted_word in compressed_ngrams_table:
         list = compressed_ngrams_table[sorted_word]
         # the word below (up to 4 tokens) is in the dictionary
         word = list[0]
         print("Found:", word)
         output_file.write("Found:" + word + "\n")
         word_summary(word, ccnt1, ccnt2, maxprint, output_file)

   return()


# hyperparameters
ccnt1   = 0
```

```
ccnt2   = 0  # 2
maxprint = 10 # up to maxprint rows shownn per word/section

spell = Speller(lang='en')

query = " "
print("\n")
output_file = open("xllm5_results.txt", "w")

while query != "":

   # entries separated by commas are treated independently
   query = input("Enter queries (ex: Gaussian distribution, central moments): ")
   queries = query.split(',')
   token_list = []
   token_clean_list = []

   for query in queries:

      tokens = query.split(' ')
      for token in tokens:
         # note: spell('feller') = 'seller', should not be autocorrected
         token = token.lower()
         if token not in dictionary:
            token = spell(token)
         token_list.append(token)
      stemmed = singular(token_list, mode = 'Internal')

      for old_token in stemmed:
         token = stemmed[old_token]
         if token in dictionary:
            token_clean_list.append(token)
      token_clean_list.sort()

      if not token_clean_list:
         if query != "":
            print("No match found")
            output_file.write("No match found\n")
      else:
         print("Found: ", token_clean_list)
         output_file.write("Found: " + str(token_clean_list) + "\n")
         process_query(token_clean_list, ccnt1, ccnt2, maxprint, output_file)

output_file.close()
```

Chapter 8

Miscellaneous Projects

This chapter features projects that span across multiple categories, such as generative AI, large language models and NLP, or machine learning optimization and scientific computing. They allow you to broaden your horizon in multiple directions. You don't need to be an expert in many areas to work on them. Typically, the first step consists of checking my Python code, create a minimum viable version, then add features, and use the code on your own data. Each project is self-contained, with an introduction presenting relevant material. References to literature and other chapters are provided as needed.

8.1 Fast probabilistic nearest neighbor search (pANN)

ANN – approximate nearest neighbors – is at the core of fast vector search, itself central to GenAI, especially GPT and LLM. My new methodology, abbreviated as pANN, has many other applications: clustering, classification, measuring the similarity between two datasets (images, soundtracks, time series, and so on), tabular data synthetization (improving poor synthetizations), model evaluation, and even detecting extreme observations.

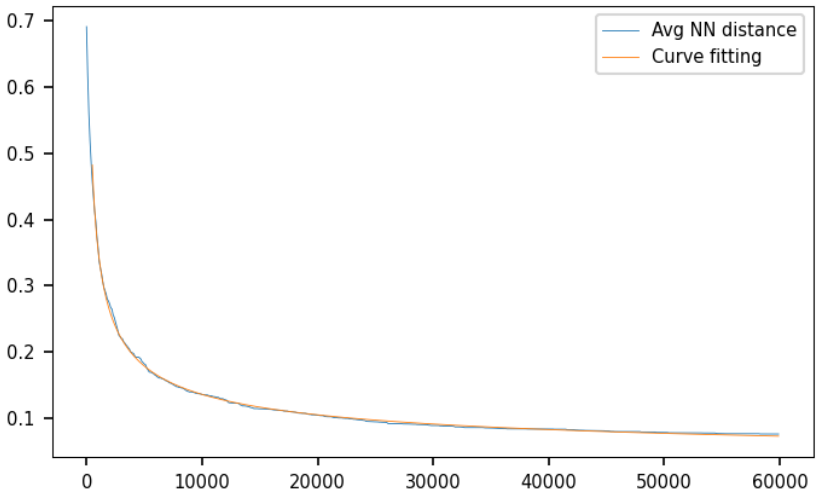


Figure 8.1: Average NN distance over time, with probabilistic ANN

Just to give an example, you could use it to categorize all time series without statistical theory. Parametric statistical models are subject to identifiability issues (redundancy) and less explainable, leading to definitions less useful to developers, and math-heavy. pANN avoids that. Fast and simple, pANN (for Probabilistic ANN) does not involve training or neural networks, and it is essentially math-free. Its versatility comes from four features:

- Most algorithms aim at minimizing a loss function. Here I also explore what you can achieve by maximizing the loss.
- Rather than focusing on one set of datasets, I use two sets S and T. For instance, K-NN looks for nearest neighbors within a set S. What about looking for nearest neighbors in T, to observations in S? This leads to far more applications than the one-set approach.

- Some algorithms are very slow and may never converge. No one looks at them. But what if the loss function drops very fast at the beginning, fast enough that you get better results in a fraction of the time, by stopping early, compared to using the "best" method?
- In many contexts, a good approximate solution obtained in little time from an otherwise non-converging algorithm, may be as good for practical purposes as a more accurate solution obtained after far more steps using a more sophisticated algorithm.

Figure 8.1 shows how quickly the loss function drops at the beginning. In this case, the loss represents the average distance to the approximate nearest neighbor, obtained so far in the iterative algorithm. The X-axis represents the iteration number. Note the excellent curve fitting (in orange) to the loss function, allowing you to predict its baseline (minimum loss, or optimum) even after a small number of iterations. To see what happens if you maximize the loss instead, read the full technical document. Another example of non-converging algorithm doing better than any kind of gradient descent is discussed in chapter 13 in [20].

8.1.1 Motivation and architecture

With the increasing popularity of RAG, LLM, and ANN-based fast vector search including in real time, it was time for me to figure out how I could improve the existing technology. In this context, ANN stands for approximated nearest neighbors, a better alternative to K-NN.

What I came up with covers a large number of applications: matching embeddings to prompts or user queries, data synthetization, GenAI model evaluation, measuring the similarity or distance between two datasets, detection of extremes in arbitrary dimensions, finding the envelop of a dataset, or classification (supervised and unsupervised). The technique was first introduced in the context of NoGAN tabular data synthetization, see chapter 7 in [19]. The goal is to find a good approximation to the solution, as quickly as possible. You can think of it as a gradient descent method, with very steep descent at the beginning, leading to a satisfactory solution in a fraction of the time most sophisticated algorithms require. Speed is further increased by using absolute differences rather than square roots of sums of squares. Also, there is no gradient and no neural networks: thus, no math beyond random numbers.

The following architecture and algorithm are common to all applications. You have two sets of multivariate vectors, S and T respectively with n and m elements, each elements being a vector with d components:

$$S = \{x_1, \ldots, x_n\}$$
$$T = \{y_1, \ldots, y_m\}.$$

For each element x_k in S, you want to find the closest neighbor $y_{\sigma(k)}$ in T. Thus, the problem consists of finding the function σ_0 that minimizes the loss function $L(\sigma)$ defined by

$$L(\sigma) = \sum_{k=1}^{n} \|x_k - y_{\sigma(k)}\|. \tag{8.1}$$

The minimum is over all integer functions σ defined on $\{1, \ldots, n\}$ with values in $\{1, \ldots, m\}$. There are m^n such functions. The one minimizing $L(\sigma)$ is denoted as σ_0. It might not be unique, but this is unimportant. In some cases, we are interested in maximizing $L(\sigma)$ instead, which is identical to minimizing $-L(\sigma)$. And frequently, to be admissible as a solution, a function σ must satisfy $x_k \neq y_{\sigma(k)}$ for $1 \leq k \leq n$.

The oldest application in recent times, also the origin for the abbreviation ANN, is the K-NN problem, or K nearest neighbors. In this case, S consists of K copies of T. As we shall see, my algorithm results in a different solution, with a variable number of neighbors per observation, rather than the fixed value K. Also, when $K = 1$, the trivial solution is $\sigma(k) = k$ for $1 \leq k \leq n$. That is, the closest neighbor to x_k is x_k itself. Thus the aforementioned constraint $x_k \neq y_{\sigma(k)}$ to eliminate this solution.

An ancient version dating back to 1890 is the assignment problem. It was solved in polynomial time in 1957, using the Hungarian algorithm [Wiki]. These days, we want something much faster than even quadratic time. My method will provide a good approximation much faster than quadratic if you stop early. Brute force would solve this problem in $n \times m$ steps, by finding the closest $y_{\sigma(k)}$ to each x_k separately. Note that unlike in the original assignment problem, here the function σ does not need to be a permutation, allowing for faster, one-to-many neighbor allocation.

The solution can be an excellent starting point for an exact search, or used as a final, good enough result. The algorithm processes the data set S a number of times. Each completed visit of S is called an epoch. In a given epoch, for each observation x_k (with $1 \leq k \leq n$), a potential new neighbor $y_{\sigma'(k)}$ is randomly selected. If

$$\|x_k - y_{\sigma'(k)}\| < (1 - \tau) \cdot \|x_k - y_{\sigma(k)}\|,$$

then $y_{\sigma'(k)}$ becomes the new, closest neighbor to x_k, replacing the old neighbor $y_{\sigma(k)}$. In this case, $\sigma(k) \leftarrow \sigma'(k)$. Otherwise, $\sigma(k)$ is unchanged, but $y_{\sigma'(k)}$ is flagged as unsuitable neighbor in the list of potential neighbors to x_k. For each x_k, the list of unsuitable neighbors starts empty and grows very slowly, at least at the beginning. The parameter τ is called the temperature. The default value is zero, but positive values that decay over time may lead to an accelerated schedule. Negative values always underperform, but it makes the loss function goes up and down, with oscillations of decreasing amplitude over time, behaving very much like the loss function in stochastic gradient descent and deep neural networks.

Another mechanism to accelerate the convergence at the beginning (what we are interested in) is as follows. At the start of each epoch, sort S in reverse order based on distance to nearest neighbors in T, obtained so far. In a given epoch, do not process all observations x_k, but only a fraction of them, for instance the top 50% with the largest NN distances.

Figure 8.1 illustrates the convergence. The power function $\varphi(t) = \alpha + \beta t^{-\gamma}$ provides an excellent fit. Here $\varphi(t)$ is the average nearest neighbor distance at time t. The time represents the number of steps performed so far, on a dataset with $n = m = 200$. Interestingly, $\gamma \approx 0.50$, but on some datasets, I was able to get faster convergence, with $\gamma \approx 0.80$. The coefficient α represents the average NN distance at the limit, if you were to do an exact search. In other words, $\alpha \approx L(\sigma_0)/n$. If you are only interested in α, you can get a good approximation in a fraction of the time it takes to compute the exact NN distances. To do it even faster by interpolating the curve fitting function based on the first few hundred measurements only, see Figure 4.5 and section 4.4.3.

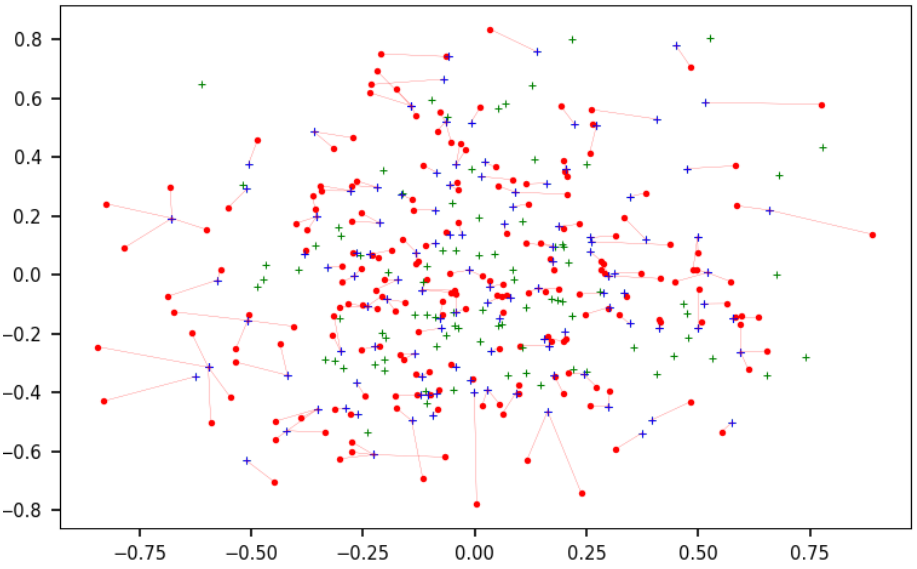


Figure 8.2: Approximate NNs from T (blue) to points in S (red) after a few thousand steps

Figure 8.2 shows the dataset used in Figure 8.1, with red segments linking points in S (red) to their closest neighbor in T (blue) obtained at the current iteration. View video here showing how the approximate nearest neighbors get more and more accurate over time, from beginning to end.

8.1.2 Applications

The methodology presented here is useful in several contexts. Now, I describe how to leverage my algorithm in various applications, ranging from traditional to GenAI and LLM.

8.1.2.1 Embeddings and large language models

In large language models, embeddings are lists of tokens attached to a keyword. For instance, Table 8.1 is based on my specialized xLLM that answers research questions about statistics and probability: see section 7.2. The table features embeddings, for 7 one-token keywords. For each keyword, the top row shows the top 10 tokens. The bottom one shows the importance of each token: the numbers represent the normalized pointwise mutual information (PMI) between a token and a keyword. This metric measures the strength of the association between a token and a keyword. Here, I use variable-length embeddings [5] to reduce the size of the embedding tables.

To measure the similarity between two words, first compute the dot product (the $\bullet$ product) [Wiki] between the two word embeddings. Tokens with a PMI of zero for either word (that is, absent for one of the two words)

are ignored in the computations. Then, compute the norm $\|\cdot\|$ of each word. The norm is the square root of the sum of squared PMIs. For instance, based on Table 8.1:

$$\text{normal} \bullet \text{Gaussian} = 0.43903 \times 0.00858 + 0.05885 \times 0.01164 = 0.004452$$
$$\text{binomial} \bullet \text{Gaussian} = 0.11796 \times 0.00858 + 0.01117 \times 0.01164 = 0.001142$$

$$\|\text{normal}\| = 0.49678, \quad \|\text{Gaussian}\| = 0.05853, \quad \|\text{binomial}\| = 0.13998.$$

There are many different ways to define the similarity between two words. The cosine similarity [Wiki] is one of them. It normalizes the dot products, but does not capture magintudes. It is computed as follows:

$$\rho(\text{normal}, \text{Gaussian}) = \frac{\text{normal} \bullet \text{Gaussian}}{\|\text{normal}\| \cdot \|\text{Gaussian}\|} = 0.15311,$$
$$\rho(\text{binomial}, \text{Gaussian}) = \frac{\text{binomial} \bullet \text{Gaussian}}{\|\text{normal}\| \cdot \|\text{Gaussian}\|} = 0.13940.$$

Whether using the dot product or cosine similarity, "normal" is closer to "Gaussian" than "binomial". The distance may then be defined as $1 - \rho$. The goal, given two sets of embeddings S and T, is to find, for each embedding in S, its closest neighbor in T. For instance, S may consist of the top 1000 standardized user queries with associated embeddings (stored in cache for fast real-time retrieval), and T maybe the full list of embeddings based on crawling and/or parsing your entire repository.

word	token 1	token 2	token 3	token 4	token 5	token 6	token 7	token 8	token 9	token 10
hypothesis	alternative	null	statistical	false	test	nested	testing	type	bourget	chinese
	0.05070	0.03925	0.03539	0.03177	0.01885	0.01661	0.01358	0.01056	0.01011	0.01011
test	statistical	wilcoxon	negative	alternative	alpha	fisher	kolmogorov	contingency	type	false
	0.09546	0.05842	0.03206	0.02700	0.02519	0.02456	0.02224	0.02099	0.02066	0.01924
normal	distribution	bivariate	standard	log	multivariate	variate	ratio	trivariate	sum	difference
	0.43903	0.15486	0.10019	0.09719	0.05885	0.05204	0.03569	0.03368	0.03240	0.03074
Gaussian	inverse	joint	increment	multivariate	physicists	spectrum	noisy	distribution	board	polygon
	0.04340	0.02718	0.01164	0.01164	0.01164	0.01006	0.00964	0.00858	0.00832	0.00774
walk	random	self-avoiding	wiener	connective	polya	levy	two-dim	lattice	trajectories	confined
	0.16104	0.10019	0.04138	0.02888	0.01691	0.01491	0.01447	0.01344	0.01004	0.01004
random	walk	variable	number	sequence	independent	set	constant	polya	one-dim	process
	0.16104	0.10245	0.08385	0.06631	0.05068	0.03509	0.03230	0.03028	0.02939	0.02844
binomial	distribution	negative	approximation	integer	beta	multivariate	discrete	trial	rise	infinity
	0.11796	0.06830	0.01455	0.01327	0.01133	0.01117	0.01039	0.00990	0.00944	0.00886

Table 8.1: Embeddings (one per word) with normalized PMI score attached to each token

When the goal is to compute all nearest neighbors withing T (in this case, $S = T$), the xLLM architecture is especially efficient. It uses a separate embedding table for each top category. Assuming q tables respectively with $N_1, \ldots, N_q$ embeddings, standard k-NN over categories bundled together is $O(N^2)$ with $N = N_1 + \cdots + N_q$, versus the much lower $O(N_1^2 + \cdots + N_q^2)$ when the q categories are treated separately. With the ANN algorithm described in section 8.1, these computing times are significantly reduced. However, with q categories, you must add a little overhead time and memory as there is a top layer for cross-category management. When a category has more than (say) 5000 embeddings, further acceleration is achieved by splitting its table into smaller batches, and compute nearest neighbors on each batch separately. The solid gain in speed usually outweighs the small loss in accuracy. For prompt compression to reduce the size of the input user queries, see [3].

8.1.2.2 Generating and evaluating synthetic data

My first use of probabilistic ANN (pANN) was for synthesizing tabular data, see chapter 7 in [19]. It led to a faster and better alternative to GAN (generative adversarial networks), and was actually called NoGAN as it does not require neural networks. But it also helps with various related GenAI problems. For instance:

- **Improving poor synthetic data**. Say you have a real dataset T, and and you created a synthetic version of it, the S set. You can generate much more observations than needed in your synthetic data, then only keep the best ones. To do this, only keep in S observations with a nearest neighbor in T that is close enough. In short, you discard synthetic observations that are too far away from any real observation. This simple trick will improve the quality of your synthetic data, if the goal is good enough replication of the underlying distribution in the real data. pANN is particularly handy to solve this problem.

- **Evaluating the quality of synthetic data**. The best metrics to evaluate the faithfulness of synthetic data are typically based on the multivariate empirical cumulative distributions (ECDF), see section 2.1. The ECDF is evaluated at various locations z in the feature space, computed both on the synthetic data S, and the real data T. In particular, the Kolmogorov-Smirnov distance is defined as

$$\text{KS}(S,T) = \sup_z |F_\text{s}(z) - F_\text{r}(z)|,$$

where F_s, F_r are the ECDFs, respectively for the synthetic and real data. It involves finding the closest neighbors to each z, both in S and T. Again, the pANN algorithm can help accelerate the computations.

For an alternative to pANN, based on interpolated binary search and radix encoding, see section 2.3. Several nearest neighbor search methods are discussed in the article "Comprehensive Guide To Approximate Nearest Neighbors Algorithms" [38].

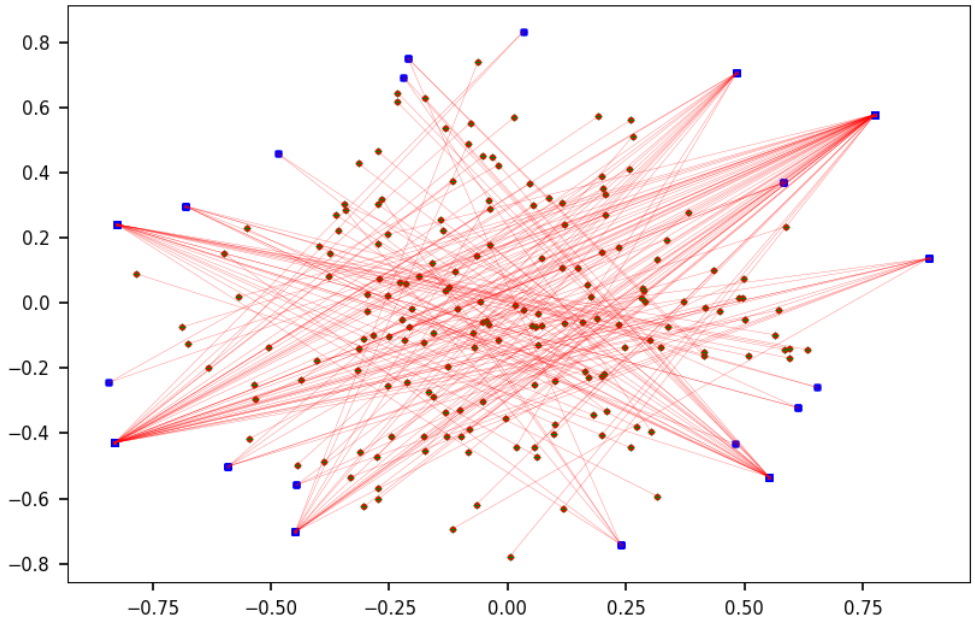


Figure 8.3: Extreme points in blue ($S = T$) obtained by maximizing the loss $L(\sigma)$

8.1.2.3 Clustering, dataset comparisons, outlier detection

Nearest neighbor (NN) methods were first used in classification algorithms, with $S = T$: the ancestor of pANN is K-NN. To adapt pANN to $K > 1$, proceed as follows. If you want the three approximate nearest neighbors ($K = 3$) to observation x_{k_1} in S, keep two extra copies of x_{k_1}, say x_{k_2} and x_{k_3}, in S. At any time in the iterative algorithm, flag any nearest neighbor assigned to one of the copies, say x_{k_2}, as nonassignable to the two other copies, in this case x_{k_1} and x_{k_3}. To do so, add the nearest neighbor in question (its index in T) to the lists `hask[`k_1`]` and `hash[`k_3`]` in line 138 in the Python code in section 8.1.4. You also need to set `optimize='speed'` in line 117 so that `hash` is active. In the code, the nearest neighbor to x_{k_1} in T, is y_j with $j =$`arr_NN[`k_1`]`. Its index is j.

For classification purposes, a new point x_{k_1} in S, outside the training set T, gets assigned using majority vote, by looking at the class assigned to its nearest neighbors in T. For clustering (unsupervised classification) the same rule applies, but there is no class label: the end result is a clustering structure that groups points in unlabeled clusters based on proximity.

Beyond classification, pANN is also helpful to find similar datasets in a database. For example, images or soundtracks. Each dataset hast its feature vector, consisting not necessarily of pixels or waves, but instead, where vector components are summary statistics about the image or soundtrack. This is an extension of what I discussed in section 8.1.2.2. It also applies to comparing time series, where vectors consist of autocorrelations of various lags, with one vector per time series. Finally, if you maximize rather than minimize the loss function, you can detect extreme points as opposed to nearest neighbors: see Figure 8.3.

8.1.3 Project and solution

Rather than asking you to write an algorithm and Python code from scratch, the goal is to understand my methodology, simplify and test the code in section 8.1.4, add features, and investigate particular cases.

The project consists of the following steps.

Step 1: Accelerating speed. Simplify the code by removing the acceleration mechanisms: get rid of the overhead attached to `optimize='speed'` (for instance, `hash`) and set `optimize='memory'`. Also, comment out line 80, lines 170–171, and replace line 130 by `k=iter%N`. Then you don't need the function `sort_x_by_NNdist_to_y`. Finally, set `decay=0`. Also identify and remove all the code linked to producing the video, for instance `flist`, `frame`, the `save_image` function, and lines 158-168.

Step 2: Performance of accelerators. Understand the difference between `iter`, `steps`, and `swaps` in the code. Play with the different accelerators. How would you assess the performance of each accelerator? Finally, allowing the loss function to go up and down with decaying oscillations and downward trend, that is, stochastic descent similar to the orange curve in Figure 5.4, is always worse than going straight down, or steepest descent. Explain why, since for neural networks, the opposite is true. Here, stochastic descent is emulated with negative `decay` in line 119. .

Step 3: Evaluation. The key metric Δ_t linked to the loss function (Formula 8.1) is the average nearest neighbor distance at any given iteration t, starting with a rather large value and decreasing over time: the nearest neighbors in T to points in S become more and more accurate as t increases, starting with random locations. However, Δ_t depends on m, the number of points in T: intuitively, the larger m, the smaller Δ_t. How would you adjust Δ_t to make it independent of m?

To answer the second part of Step 2, in deep neural networks, the loss function is a proxy to the performance or quality metric. To the contrary, here the loss function and model evaluation metric are identical. Also, there is no risk in strictly decreasing the loss function at each iteration: eventually the algorithm must reach a global minimum. This is not true in deep neural networks, where you can get stuck in a local minimum if you don't allow the loss function to go up and down. As for the terminology, a `swap` is when a change (nearest neighbor re-assignment) actually occurs during an iteration. Swaps become rarer and rarer over time. A `step` within an iteration is when a nearest neighbor candidate is not accepted (for instance, because it has already been rejected in the past), forcing the algorithm to choose another candidate. Steps are more numerous towards the end, and used only when `optimize` is set to `'speed'`. Otherwise steps and iterations are the same.

Regarding Step 3, if the points are independently and uniformly distributed in a d-dimensional feature space and $S = T$, then Δ_t is almost proportional to $m^{-1/d}$ when t is large enough. Thus the adjusted $\Delta'_t = m^{1/d}\Delta_t$ is nearly independent of m. The factor $m^{-1/d}$ can be obtained via simulation and curve fitting. However, there is a theoretical explanation. Let $S = T$ and let us assume that the points follow a Poisson process of intensity λ in d dimensions. The probability that there is no point within a distance R to a given, arbitrary location is $G(R) = \exp(-\lambda\nu R^d)$ where ν is the volume of the d-dimensional unit ball [Wiki]. Thus, the CDF (cumulative distribution) for the distance R to the nearest neighbor is $F_R(r) = 1 - G(r)$, for $r \geq 0$.

So, R has a Weibull distribution. Its expectation $\mathrm{E}(R)$ is proportional to $\lambda^{-1/d}$, that is, to $m^{-1/d}$ since the intensity λ is the expected number of points per unit volume (or per unit area in 2D). The peculiarity here is that I use the taxicab distance [Wiki] rather than the traditional Euclidean norm: see line 145 in Python code in section 8.1.4. The reason is for faster computations; the choice of the distance has very little impact. Then, the volume of the unit ball is $\nu = 2^d/d!$

8.1.4 Python code

See section 8.1.2.3 for details about `hash`, `arr_NN` and the parameter `optimize`. The `sort_x_by_NNdist_to_y` function is used together with `K=int(0.5*N)` in line 80 to accelerate the speed of the algorithm, by processing only the top `K` observations in S at each epoch, sorted by nearest distance to T. An epoch (same meaning as in neural networks) is a full run of S. However here, only `K` observations out of `N` are processed during an epoch. Yet, due to re-sorting S at the end of each epoch, the `K` observations change at each epoch. Thus over time, all observations are used. To not use this feature, set `K=N`.

The parameter `decay` in line 151 offers a different acceleration mechanism. The default value is zero. A positive value provides a boost at the beginning, where it is most needed. A negative value always yields lower performance, yet the resulting loss function goes up and down (rather than only down), in a way similar to stochastic gradient descent in deep neural networks: there is no benefit to it, other than educational.

Acceleration mechanisms offer a modest boost, about 10% in my tests. But I haven't investigated them thoroughly. If you remove them, it will reduce the length of the code and make it easier to understand. There is also a significant amount of code to produce the visualizations and the video. If you remove this, the code will be significantly shorter. The code is also on GitHub, here, with a shorter version here.

```
1 # Probabilistic ANN, can be used for clustering / classification
```

```
2
3  import numpy as np
4  import matplotlib.pyplot as plt
5  from scipy.optimize import curve_fit
6  import matplotlib as mpl
7  from PIL import Image
8  import moviepy.video.io.ImageSequenceClip
9
10
11 #--- [1] Parameters and functions for visualizations
12
13 def save_image(fname,frame):
14
15     # back-up function in case of problems with plt.savefig
16     global fixedSize
17
18     plt.savefig(fname, bbox_inches='tight')
19     # make sure each image has same size and size is multiple of 2
20     # required to produce a viewable video
21     im = Image.open(fname)
22     if frame == 0:
23        # fixedSize determined once for all in the first frame
24        width, height = im.size
25        width=2*int(width/2)
26        height=2*int(height/2)
27        fixedSize=(width,height)
28     im = im.resize(fixedSize)
29     im.save(fname,"PNG")
30     return()
31
32 def plot_frame():
33
34     plt.scatter(x[:,0], x[:,1], color='red', s = 2.5)
35     z = []
36
37     for k in range(N):
38
39        neighbor = arr_NN[k]
40        x_values = (x[k,0], y[neighbor,0])
41        y_values = (x[k,1], y[neighbor,1])
42        plt.plot(x_values,y_values,color='red',linewidth=0.1,marker=".",markersize=0.1)
43        z_obs = (y[neighbor,0], y[neighbor,1])
44        z.append(z_obs)
45
46     z = np.array(z)
47     plt.scatter(y[:,0], y[:,1], s=10, marker = '+', linewidths=0.5, color='green')
48     plt.scatter(z[:,0], z[:,1], s=10, marker = '+', linewidths=0.5, color='blue')
49     return()
50
51 mpl.rcParams['axes.linewidth'] = 0.5
52 plt.rcParams['xtick.labelsize'] = 7
53 plt.rcParams['ytick.labelsize'] = 7
54
55
56 #--- [2] Create data, initial list of NN, and hash
57
58 def sort_x_by_NNdist_to_y(x, y, arr_NN):
59
60     NNdist = {}
61     x_tmp = np.copy(x)
62     arr_NN_tmp = np.copy(arr_NN)
63     for k in range(N):
64        neighbor = arr_NN_tmp[k]
65        NNdist[k] = np.sum(abs(x_tmp[k] - y[neighbor]))
66     NNdist = dict(sorted(NNdist.items(), key=lambda item: item[1],reverse=True ))
67
```

```
    k = 0
    for key in NNdist:
        arr_NN[k] = arr_NN_tmp[key]
        x[k] = x_tmp[key]
        k += 1
    return(x, arr_NN)

seed = 57
np.random.seed(seed)
eps = 0.00000000001

N = 200        # number of points in x[]
K = int(0.5 * N) # sort x[] by NN distance every K iterations
M = 200        # number of points in y[]

niter = 10000
mean = [0, 0]
cov = [(0.1, 0),(0, 0.1)]
x = np.random.multivariate_normal(mean, cov, size=N)
y = np.random.multivariate_normal(mean, cov, size=M)
# y = np.copy(x)
np.random.shuffle(x)
np.random.shuffle(y)

arr_NN = np.zeros(N)
arr_NN = arr_NN.astype(int)
hash = {}
sum_dist = 0

for k in range(N):

    # nearest neighbor to x[k] can't be identical to x[k]
    dist = 0

    while dist < eps:
      neighbor = int(np.random.randint(0, M))
      dist = np.sum(abs(x[k] - y[neighbor]))

    arr_NN[k] = neighbor
    sum_dist += np.sum(abs(x[k] - y[neighbor]))
    hash[k] = (-1,)

x, arr_NN = sort_x_by_NNdist_to_y(x, y, arr_NN)
low = sum_dist


#--- [3] Main part

mode   = 'minDist' # options: 'minDist' or 'maxDist'
optimize = 'speed' # options: 'speed' or 'memory'
video  = False   # True if you want to produce a video
decay  = 0.0

history_val = []
history_arg = []
flist = []
swaps = 0
steps = 0
frame = 0

for iter in range(niter):

    k = iter % K
    j = -1
    while j in hash[k] and len(hash[k]) <= N:
        # if optimized for memory, there is always only one iter in this loop
```

```
        steps += 1
        j = np.random.randint(0, M) # potential new neighbor y[j], to x[k]

    if optimize == 'speed':
        hash[k] = (*hash[k], j)

    if len(hash[k]) <= N:

        # if optimized for memory, then len(hash[k]) <= N, always
        old_neighbor = arr_NN[k]
        new_neighbor = j
        old_dist = np.sum(abs(x[k] - y[old_neighbor]))
        new_dist = np.sum(abs(x[k] - y[new_neighbor]))
        if mode == 'minDist':
            ratio = new_dist/(old_dist + eps)
        else:
            ratio = old_dist/(new_dist + eps)
        if ratio < 1-decay/np.log(2+iter) and new_dist > eps:
            swaps += 1
            arr_NN[k] = new_neighbor
            sum_dist += new_dist - old_dist
            if sum_dist < low:
                low = sum_dist

            if video and swaps % 4 == 0:

                fname='ann_frame'+str(frame)+'.png'
                flist.append(fname)
                plot_frame()

                # save image: width must be a multiple of 2 pixels, all with same size
                # use save_image(fname,frame) in case of problems with plt.savefig
                plt.savefig(fname, dpi = 200)
                plt.close()
                frame += 1

    if iter % K == K-1:
        x, arr_NN = sort_x_by_NNdist_to_y(x, y, arr_NN)

    if iter % 100 == 0:
        print("%6d %6d %6d %8.4f %8.4f"
              % (iter, swaps, steps, low/N, sum_dist/N))
        history_val.append(sum_dist/N)
        history_arg.append(steps) # try replacing steps by iter


history_val = np.array(history_val)
history_arg = np.array(history_arg)

if video:
    clip = moviepy.video.io.ImageSequenceClip.ImageSequenceClip(flist, fps=6)
    clip.write_videofile('ann.mp4')


#--- [4] Visualizations (other than the video)

plot_frame()
plt.show()

#- curve fitting for average NN distance (Y-axis) over time (X-axis)

# works only with mode == 'minDist'

def objective(x, a, b, c):
    return(a + b*(x**c))

```

```
# ignore first offset iterations, where fitting is poor
offset = 5

x = history_arg[offset:]
y = history_val[offset:]

# param_bounds to set bounts on curve fitting parameters
if mode == 'minDist':
   param_bounds=([0,0,-1],[np.inf,np.infty,0])
else:
   param_bounds=([0,0,0],[np.inf,np.infty,1])

param, cov = curve_fit(objective, x, y, bounds = param_bounds)
a, b, c = param
# is c = -1/2 the theoretical value, assuming a = 0?
print("\n",a, b, c)

y_fit = objective(x, a, b, c)
## plt.plot(x, y, linewidth=0.4)
plt.plot(history_arg, history_val, linewidth=0.4)
plt.plot(x, y_fit, linewidth=0.4)
plt.legend(['Avg NN distance','Curve fitting'],fontsize = 7)
plt.show()
```

8.2 Building and evaluating a taxonomy-enriched LLM

Section 7.2 and appendix C discuss xLLM, my multi-LLM architecture to deal with knowledge retrieval, referencing, and summarization. It heavily relies on structures found in the corpus, such as taxonomies. I tested it on the Wolfram website, to create a specialized sub-LLM for the "Probability & Statistics" section. The root directory is accessible here on Wolfram. This is the starting point for crawling all the content. Note that the strength of xLLM comes from using structured text such as taxonomies, consisting of concise, high-value text elements, superior to standard embeddings and tokens based on raw content only.

Wolfram top categories	Top categories based on parsed content		
Bayesian Analysis	L_1 Methods	Proportions	Rank Statistics
Descriptive Statistics	Analysis of Variance	Gaming Theory	Standardization
Error Analysis	Bias	Indices	Sums of Squares
Estimators	Bivariate Methods	Least Squares	Sampling Methods
Markov Processes	Central Limit Theorem	Linear Algebra	Small Data
Moments	Central Tendency Metrics	Markov Chains	Statistical Laws
Multivariate Statistics	Characteristic Function	Maximum Likehood	Statistical Moments
Nonparametric Statistics	Conditional Probas, Bayes	Measure Theory	Statistical Tests
Probability	Confidence Intervals	Measures of Spread	Stepwise Methods
Random Numbers	Continuous Distributions	Model Fitting	Tabular Data
Random Walks	Correlations Analysis	Multivariate Methods	Tests of Hypothesis
Rank Statistics	Curve Fitting	Normal Distribution	Time Series
Regression	Data	Optimization	Trials
Runs	Degrees of Freedom	Outliers	Plots
Statistical Asymptotic	Density Functions	Parametric Estimation	Regression
Statistical Distributions	Descriptive Statistics	Point Estimation	
Statistical Indices	Discrete Distributions	Poisson Processes	
Statistical Plots	Error Analysis	Probability Theory	
Statistical Tests	Estimation Theory	Quantiles & Simulation	
Time-Series Analysis	Exponential Family	Random Numbers	
Trials	Extreme Value Theory	Random Walks	

Table 8.2: Top categories, Wolfram taxonomy (leftmost column) vs content-based

Words found in high-quality taxonomies, glossaries, titles, synonyms dictionaries or indexes, are well-formed and important. They should be assigned higher weights than words found in ordinary content, and contribute to the quality and conciseness of the output returned to user prompts. The quality of the input sources is also very important. Here the goal is to build a faithful taxonomy, one that matches the crawled data. In the case of Wolfram, the content has a strong emphasis on mathematical statistics. To build a specialized LLM more focused on applied statistics, you need content augmentation with (say) Wikipedia, or design a separate LLM based on different sources. Here I stick with the original Wolfram content. To avoid hallucinations, the resulting sub-LLM provides answers only if the relevant information exists in the specialized corpus.

"Sampling Methods" sub-categories		
Aggregated, Grouped Data	Estimation	Statistical Summaries
Bernoulli Trials	Large, Small Samples	Sample Types
Central Limit Theorem	Moments	Random Sampling
Combinatorics	Normalization	Sampling Error
Confidence Intervals	Order Statistics, Ranks	Variance Sources
Data Gathering Process	Population	Time Series Sampling
Empirical Distribution	Sample Bias	Observations Types
Equal / Unequal Samples	Sample Size	Outliers

Table 8.3: "Sampling Methods" sub-categories based on parsed content

https://mathworld.wolfram.com/Stem-and-LeafDiagram.html	https://mathworld.wolfram.com/BonferroniCorrection.html
Detected category: longitudinal~data (score: 405)	Detected category: bonferroni~correction (score: 207)
Wolfram category: stem-and-leaf~diagram	Wolfram category: bonferroni~correction
https://mathworld.wolfram.com/StemLeafPlot.html	https://mathworld.wolfram.com/Chi-SquaredTest.html
Detected category: stem-and-leaf~diagram (score: 18)	Detected category: beta~distribution (score: 144)
Wolfram category: stemleafplot	Wolfram category: chi-squared~test
https://mathworld.wolfram.com/TukeyMean-DifferencePlot.htm	https://mathworld.wolfram.com/Fisher-BehrensProblem.html
Detected category: q-q~plot (score: 27)	Detected category: reversion~to~the~mean (score: 63)
Wolfram category: tukey~mean-difference~plot	Wolfram category: fisher-behrens~problem
https://mathworld.wolfram.com/AlphaValue.html	https://mathworld.wolfram.com/FishersExactTest.html
Detected category: alpha~value (score: 54)	Detected category: fisher~z-distribution (score: 576)
Wolfram category: alpha~value	Wolfram category: fishers~exact~test
https://mathworld.wolfram.com/AlternativeHypothesis.html	https://mathworld.wolfram.com/HotellingsT-SquaredTest.html
Detected category: hypothesis (score: 99)	Detected category: hotelling~t^2~test (score: 45)
Wolfram category: alternative~hypothesis	Wolfram category: hotellings~t^2~test
https://mathworld.wolfram.com/Anderson-DarlingStatistic.html	https://mathworld.wolfram.com/Hypothesis.html
Detected category: statistic (score: 36)	Detected category: hypothesis (score: 216)
Wolfram category: anderson-darling~statistic	Wolfram category: hypothesis
https://mathworld.wolfram.com/BalancedANOVA.html	https://mathworld.wolfram.com/HypothesisTesting.html
Detected category: anova (score: 36)	Detected category: hypothesis~testing (score: 189)
Wolfram category: balanced~anova	Wolfram category: hypothesis~testing

Figure 8.4: Wolfram true vs re-assigned categories based on content (extract from full table)

The Wolfram "Probability & Statistics" ontology has about 600 categories, subcategories and so on. Here we want to accomplish three goals:

- Pretend that we crawled the Wolfram website but did not retrieve the taxonomy. Build one from scratch, based on the crawled content. Again, here we are only interested in the "Probability & Statistics" section.
- Pretend that we have some external taxonomy that covers the Wolfram crawl. For instance, the machine learning taxonomy from Wikipedia (see here). Use that taxonomy to categorize each Wolfram webpage

and top keyword (consisting of multiple tokens) found in the crawled content. Of course, not all keywords can be uniquely categorized.

- Pretend that the external taxonomy is actually that from Wolfram, but we are not aware of it. Use that taxonomy as it if was an external source, to categorize each Wolfram webpage. Then compare with the actual category assigned by Wolfram. Then compare the two taxonomies: original versus reconstructed. This last step allows you to evaluate the quality of your LLM.

8.2.1 Project and solution

The goal is not to write code from scratch to solve the problem, but to look at my implementation in section 8.2.2, understand the different tables and components, test it, and see how it works. It is a bonus if you can improve my algorithms.

My code is significantly shorter than it appears at first glance. First, lines 1–105 reads all the tables produced by `xLLM6.py`, described in appendix C.4 and in the solution to Step 1 in project 7.2.3. You won't need all of them, but they are there for convenience and consistency with the main xLLM architecture. Then, lines 240–352 allows you to understand and play with some of the new tables built to facilitate the creation of the taxonomy. The core of the code is split in two sections. First, lines 108–237 to create the new tables needed to create a taxonomy from scratch: see description in the comments in lines 116-124. Once produced, you can load them in memory next time you run the code, to boost speed. Then, lines 355–432 deals with integrating an external taxonomy into xLLM. All these tables are also on GitHub, here. The code also requires the `xllm6_util.py` library, available here. The project consists of the following steps:

Step 1: Build taxonomy from scratch. Using only the `dictionary` table in the code in section 8.2.2, see how I build the `topWords` table, and then `connectedTopWords`, in lines 126–168. How can you use `topWords` to create categories, and `connectedTopWords` to create subcategories? The solution requires human intelligence to assign a meaningful label or name to each category and subcategory, and to discard non-valid entries. Thus, the process is semi-automated.

Do not use the Wolfram category. That is, do no use the `hash_category` table featuring the Wolfram taxonomy. The goal here is to create an alternate taxonomy from scratch. See what I came up with in Tables 8.2 and 8.3. Your solution will be different.

Step 2: Leverage external taxonomy. In line 402, I retrieve a list of categories from `hash_category`. While these are actually Wolfram categories, let's pretend that they are external. Now, in lines 407–427, see how I assign a category from that list, to each word found in `dictionary`. For that purpose, I use a similarity metric that compares two text elements: a dictionary word on one side, and a category entry on the other side; see lines 380–400. Try with different similarity metrics, or allow for multiple categories per word. My results are stored in the `assignedCategories` table, available here. The relevancy score is included. Note that some words are uncategorized.

Step 3: LLM evaluation. Assign a category to each crawled webpage using `assignedCategories` built in Step 2. This table available here plays the role of an ontology based on the crawled content augmented with the Wolfram taxonomy, while ignoring how categories are assigned to webpages by Wolfram. Then, look at the Wolfram categories assigned to webpages, using this table. Note that in this table, each URL is linked to two categories: the actual category on the left in the parentheses, and the parent category on the right (one level above). In the rare instances where a category has multiple parent categories, only one is kept. Each webpage is assigned to just one category.

You will need: the `arr_url` table (here) mapping URL IDs to URLs, and also the `url_map` table (here) indexed by `dictionary` words. The latter helps you retrieve the list of webpages (URL IDs) containing any word in the dictionary. By inverting it, you can retrieve, for each URL, all the dictionary words that it contains. Note that the Wolfram category table (here) is indexed by URL, while `assignedCategories` is indexed by words from the `dictionary` table (both use the same "word" index). To evaluate the LLM, for each webpage, compare the two category assignments: one coming from here versus the other one derived from `assignedCategories`. The former is supposed to be the correct one. Quantify the amount of mismatch.

In the end, content-based categories found in `assignedCategories` also come from Wolfram (indirectly as if using an external taxonomy). But they may be assigned to webpages differently, compared to the real Wolfram taxonomy. The fewer mismatches between the two assignments (across all the webpages), the better your algorithm. In short, this step reallocates Wolfram categories to webpages, not knowing how Wolfram does the allocation.

Step 4: Taxonomy augmentation. How would you create an augmented taxonomy, by blending two different ones? For instance, the Wolfram taxonomy with the one obtained in Step 1.

In addition to the code in section 8.2.2, the Excel spreadsheet `Wolfram-stats2.xlsx`, available here, illustrates some of the steps. The `TopWords` and `TopWord Pairs` tabs are related to Step 1. Then, I also use a list of words to ignore when building the fundamental `TopWords` table: see the man-made `ignoredWords` list in lines 110–112 in the Python code.

To answer Step 2 and Step 3, I produced a separate piece of code `reallocate.py`, available on GitHub, here. Figure 8.4 shows an extract from the output. The full table `detectedCategories.txt` is on GitHub, here. In essence, this Python script uses the Wolfram taxonomy as external data to the crawled content, and assigns a category to each URL based on words found on the webpage in question.

Actually, for each page, the script assigns multiple categories, each with its relevancy score. In the end, the candidate category with the highest score is selected as the target category. Two different scoring mechanisms are offered, determined by the tuning parameter `mode`. The best fit is obtained with `mode='depth'`, favoring deeper over general sub-categories to better mimic the way Wolfram assigns categories to pages. Even then, only 141 URLs are perfectly matched to their true Wolfram category. However, mismatch does not mean error: it means a close but not perfect match. Indeed, the reconstructed taxonomy may be superior to the true one, in the sense that the Wolfram page categorization is too granular. It also illustrates how evaluation metrics are subject to argumentation.

A better evaluation metric would take into account the distance or similarity between the actual Wolfram category, and the reconstructed one. If the reconstructed category is the parent (one level above) of the Wolfram category, the penalty for not having a perfect match, should be small.

8.2.2 Python code

The code is also on GitHub, here.

```
# taxonomy.py: vincentg@mltechniques.com

import requests

# Unlike xllm6.py, xllm6_short.py does not process the (huge) crawled data.
# Instead, it uses the much smaller summary tables produced by xllm6.py


#--- [1] get tables if not present already

# First, get xllm6_util.py from GitHub and save it locally as xllm6_util.py
#    note: this python code does that automatically for you
# Then import everything from that library with 'from xllm6_util import *'
# Now you can call the read_xxx() functions from that library
# In addition, the tables stopwords and utf_map are also loaded
#
# Notes:
#   - On first use, dowload all locally with overwrite = True
#   - On next uses, please use local copies: set overwrite = False

# Table description:
#
# unless otherwise specified, a word consists of 1, 2, 3, or 4 tokens
# word_pairs is used in xllm6.py, not in xllm6_short.py
#
# dictionary = {} words with counts: core (central) table
# word_pairs = {} pairs of 1-token words found in same word, with count
# word2_pairs = {} pairs of multi-token words found on same URL, with count
# url_map = {}    URL IDs attached to words in dictionary
# arr_url = []    maps URL IDs to URLs (one-to-one)
# hash_category = {} categories attached to a word
# hash_related = {} related topics attached to a word
# hash_see = {}  topics from "see also" section, attached to word
# ngrams_table = {} ngrams of word found when crawling
# compressed_ngrams_table = {} only keep ngram with highest count
# utf_map = {}    map accented characters to non-accented version
```

```
# stopwords = () words (1 or more tokens) not accepted in dictionary
# word_hash = {} list of 1-token words associated to a 1-token word
# word2_hash = {} list of multi-token words associated to a multi-token word
# compressed_word2_hash = {} shorter version of word2_hash
# embeddings = {} key is a 1-token word; value is hash of 1-token:weight
# embeddings2 = {} key is a word; value is hash of word:weight


path =
    "https://raw.githubusercontent.com/VincentGranville/Large-Language-Models/main/xllm6/"

overwrite = False # if True, get tables from GitHub, otherwise use local copy

if overwrite:

   response = requests.get(path + "xllm6_util.py")
   python_code = response.text

   local_copy = "xllm6_util"
   file = open(local_copy + ".py", "w")
   file.write(python_code)
   file.close()

   # get local copy of tables

   files = [ 'xllm6_arr_url.txt',
           'xllm6_compressed_ngrams_table.txt',
           'xllm6_compressed_word2_hash.txt',
           'xllm6_dictionary.txt',
           'xllm6_embeddings.txt',
           'xllm6_embeddings2.txt',
           'xllm6_hash_related.txt',
           'xllm6_hash_category.txt',
           'xllm6_hash_see.txt',
           'xllm6_url_map.txt',
           'xllm6_word2_pairs',
           'stopwords.txt'
         ]

   for name in files:
      response = requests.get(path + name)
      content = response.text
      file = open(name, "w")
      file.write(content)
      file.close()

import xllm6_util as llm6

# if path argument absent in read_xxx(), read from GitHub
# otherwise, read from copy found in path

arr_url    = llm6.read_arr_url("xllm6_arr_url.txt", path="")
dictionary = llm6.read_dictionary("xllm6_dictionary.txt", path="")
stopwords  = llm6.read_stopwords("stopwords.txt", path="")

compressed_ngrams_table = llm6.read_table("xllm6_compressed_ngrams_table.txt",
                                          type="list", path="")
compressed_word2_hash = llm6.read_table("xllm6_compressed_word2_hash.txt",
                                          type="hash", path="")
embeddings = llm6.read_table("xllm6_embeddings.txt", type="hash", path="",
                                            format="float")
embeddings2 = llm6.read_table("xllm6_embeddings2.txt", type="hash", path="",
                                            format="float")
hash_related = llm6.read_table("xllm6_hash_related.txt", type="hash", path="")
hash_see   = llm6.read_table("xllm6_hash_see.txt", type="hash", path="")
hash_category = llm6.read_table("xllm6_hash_category.txt", type="hash", path="")
```

```
url_map    = llm6.read_table("xllm6_url_map.txt", type="hash", path="")
word2_pairs = llm6.read_table("xllm6_word2_pairs.txt", type="list", path="")


#--- [2] Create/save taxonomy tables if overwrite = True, otherwise read them

from collections import OrderedDict

ignoreWords = { "term", "th", "form", "term", "two", "number", "meaning", "normally",
            "summarizes", "assumed", "assumes", "p", "s", "et", "possible",
            "&#9671;", ";", "denoted", "denotes", "computed", "other"}

def create_taxonomy_tables(threshold, thresh2, ignoreWords, dictionary):

   topWords = {}      # words with highest counts, from dictionary
   wordGroups = {}    # hash of hash: key = topWord; value = hash of words
                      #      containing topWord (can be empty)
   connectedTopWords = {} # key = (wordA, wordB) where wordA and wordB contains
                      #       a topWord; value = occurrences count
   smallDictionary = {} # dictionary entries (words) containing a topWord
   connectedByTopWord = {} # same as connectedTopWords, but in flattened hash format;
                      #       key = topWord
   missingConnections = {} # if this table is not empty, reduce threshold and/or thresh2

   for word in dictionary:
      n = dictionary[word] # word count
      tokens = word.count('~')
      if n > threshold and word not in ignoreWords: # or tokens > 1 and n > 1:
         topWords[word] = n

   for topWord in topWords:
      n1 = dictionary[topWord]
      hash = {}
      for word in dictionary:
         n2 = dictionary[word]
         if topWord in word and n2 > thresh2 and word != topWord:
            hash[word] = n2
      if hash:
         hash = dict(sorted(hash.items(), key=lambda item: item[1], reverse=True))
      else:
         missingConnections[topWord] = 1
      wordGroups[topWord] = hash

   for topWord in topWords:
      for word in dictionary:
         if topWord in word:
            smallDictionary[word] = dictionary[word]

   counter = 0
   for topWordA in topWords:
      if counter % 10 == 0:
         print("Create connectedTopWords: ", counter, "/", len(topWords))
      counter += 1
      hash = {}
      for topWordB in topWords:
         key = (topWordA, topWordB)
         if topWordA != topWordB:
            connectedTopWords[key] = 0
            for word in smallDictionary:
               if topWordA in word and topWordB in word:
                  connectedTopWords[key] += 1
                  if topWordB in hash:
                     hash[topWordB] += 1
                  else:
                     hash[topWordB] = 1
      hash = dict(sorted(hash.items(), key=lambda item: item[1], reverse=True))
```

```
168         connectedByTopWord[topWordA] = hash
169 
170     taxonomy_tables = [topWords, wordGroups, connectedTopWords,
171                        smallDictionary, connectedByTopWord, missingConnections]
172     return(taxonomy_tables)
173 
174 
175 def save_taxonomy_tables():
176 
177     list = { "topWords" : topWords,
178              "wordGroups" : wordGroups,
179              "smallDictionary" : smallDictionary,
180              "connectedByTopWord" : connectedByTopWord,
181              "missingConnections" : missingConnections,
182            }
183 
184     for table_name in list:
185         file = open("xllm6_" + table_name + ".txt", "w")
186         table = list[table_name]
187         for word in table:
188             file.write(word + "\t" + str(table[word]) + "\n")
189         file.close()
190 
191     file = open("xllm6_connectedTopWords.txt", "w")
192     for key in connectedTopWords:
193         file.write(str(key) + "\t" + str(connectedTopWords[key]) + "\n")
194     file.close()
195 
196     return()
197 
198 
199 #--- Get taxonomy tables
200 
201 build_taxonomy_tables = True # if True, create and save these tables locally (slow)
202 
203 if build_taxonomy_tables:
204 
205     threshold = 30 # minimum word count to qualify as topWord
206     thresh2 = 2    # another word count threshold
207 
208     taxonomy_tables = create_taxonomy_tables(threshold, thresh2, ignoreWords, dictionary)
209     topWords       = taxonomy_tables[0]
210     wordGroups     = taxonomy_tables[1]
211     connectedTopWords = taxonomy_tables[2]
212     smallDictionary = taxonomy_tables[3]
213     connectedByTopWord = taxonomy_tables[4]
214     missingConnections = taxonomy_tables[5]
215 
216     connectedTopWords = dict(sorted(connectedTopWords.items(),
217                              key=lambda item: item[1], reverse=True))
218 
219     save_taxonomy_tables()
220     for topWord in missingConnections:
221         print(topWord)
222     print()
223 
224 else:
225 
226     smallDictionary = llm6.read_dictionary("xllm6_smallDictionary.txt", path="")
227     topWords       = llm6.read_dictionary("xllm6_topWords.txt", path="")
228     wordGroups     = llm6.read_table("xllm6_wordGroups.txt", type="hash", path="")
229     connectedByTopWord = llm6.read_table("xllm6_connectedByTopWord.txt", type="hash",
        path="")
230 
231     connectedTopWords = {}
232     data = llm6.get_data("xllm6_connectedTopWords.txt", path="")
```

```
    for line in data:
        line = line.split('\t')
        count = int(line[1])
        key = llm6.text_to_list(line[0])
        connectedTopWords[key] = count


#--- [3] Play with taxonomy tables to get insights and improve them

topWords = dict(sorted(topWords.items(), key=lambda item: item[0]))

def show_menu(n, dict_mode):

    # option 'o' useful to check if topWordA, topWordB are connected or not
    # option 'c' shows all topWordB connected to topWordA = topWord

    print("Command line menu: \n")
    print("<Enter>          - exit")
    print("h                - help: show menu options")
    print("a                - show all top words")
    print("ds               - select short dictionary")
    print("df               - select full dictionary")
    print("n integer        - display entries with count >= integer")
    print("f string         - find string in dictionary")
    print("g topWord        - print groupWords[topWord]")
    print("c topWord        - print connectedByTopWord[topWord]")
    print("l topWordA topWordB - (topWordA, topWordB) connections count\n")
    print("current settings: n = %3d, dictionary = %s" %(n, dict_mode))
    print()
    return()

topWords = dict(sorted(topWords.items(), key=lambda item: item[0]))

dict_mode = 'short'
dict = smallDictionary
query = "o"
n = 0 # return entries with count >= n
show_menu(n, dict_mode)

while query != "":

    query  = input("Enter command, ex: <c hypothesis> [h for help]: ")
    queries = query.split(' ')
    action = queries[0]
    if len(queries) > 2:
        queries[1] = queries[1] + " " + queries[2]

    if action == 'h':
        show_menu(n, dict_mode)

    elif action == 'ds':
        dict = smallDictionary
        dict_mode = 'short'

    elif action == 'df':
        dict = dictionary
        dict_mode = 'full'

    elif action == 'a':
        for topWord in topWords:
            count = topWords[topWord]
            print(count, topWord)
        print()

    elif action in ('f', 'g', 'c', 'l', 'n') and len(queries) > 1:
        string = queries[1]
```

```
299
300         if action == 'n':
301             n = int(string)
302
303         elif action == 'f':
304             for word in dict:
305                 count = dict[word]
306                 if string in word and count >= n:
307                     print(count, string, word)
308             print()
309
310         elif action == 'g':
311             topWord = string
312             if topWord in wordGroups:
313                 hash = wordGroups[topWord]
314                 countA = dictionary[topWord]
315                 for word in hash:
316                     countB = dictionary[word]
317                     if countB >= n:
318                         print(countA, countB, topWord, word)
319             else:
320                 print("topWord not in wordGroups")
321             print()
322
323         elif action == 'c':
324             topWord = string
325             if topWord in connectedByTopWord:
326                 hash = connectedByTopWord[topWord]
327                 countA = dictionary[topWord]
328                 for word in hash:
329                     countB = dictionary[word]
330                     countAB = hash[word]
331                     if countAB >= n:
332                         print(countA, countB, countAB, topWord, word)
333             else:
334                 print("topWord not in wordGroups")
335             print()
336
337         elif action == 'l':
338             astring = string.split(' ')
339             if len(astring) == 1:
340                 print("needs 2 topWords, space-separated")
341             else:
342                 key = (astring[0], astring[1])
343                 if key in connectedTopWords:
344                     count = connectedTopWords[key]
345                 else:
346                     count = 0
347                 print(count, key)
348
349     elif action != '':
350         print("Missing arguments")
351
352 print()
353
354
355 #--- [4] Build local taxomomy using external taxonomy
356
357 ## extract categories from external category table...
358 ## assign category to sample page based on words in page
359 ## stem/plural
360
361 def get_external_taxonomy(hash_category):
362
363     categories = {}
364     parent_categories = {}
```

```
365
366     for word in hash_category:
367        for category_item in hash_category[word]:
368           category_item = category_item.lower()
369           category_item = category_item.replace(' ',' ').split(' | ')
370           category1 = category_item[0].replace(' ', '~')
371           category2 = category_item[1].replace(' ', '~')
372           level1 = int(category_item[2])
373           level2 = level1 - 1
374           categories[category1] = level1
375           categories[category2] = level2
376           parent_categories[category1] = category2
377     return(categories, parent_categories)
378
379
380  def compute_similarity(dictionary, word, category):
381
382     tokensA = word.split("~")
383     tokensB = category.split("~")
384     normA = 0
385     normB = 0
386     for tokenA in tokensA:
387        if tokenA in dictionary:
388           normA += dictionary[tokenA]**0.50
389     for tokenB in tokensB:
390        if tokenB in dictionary:
391           normB += dictionary[tokenB]**0.50
392
393     similarity = 0
394     for tokenA in tokensA:
395        for tokenB in tokensB:
396           if tokenA == tokenB and tokenA in dictionary and tokenB in dictionary:
397              weight = dictionary[tokenA]
398              similarity += weight**0.50
399     similarity /= max(normA, normB)
400     return(similarity)
401
402  categories, parent_categories = get_external_taxonomy(hash_category)
403
404
405  #--- Main loop
406
407  assignedCategories = {}
408  counter = 0
409
410  print("Assign categories to dictionary words\n")
411
412  for word in dictionary:
413     max_similarity = 0
414     max_depth = 0
415     NN_category = ""
416     for category in categories:
417        depth = categories[category]
418        similarity = compute_similarity(dictionary, word, category)
419        if similarity > max_similarity:
420           max_similarity = similarity
421           max_depth = depth
422           NN_category = category
423     assignedCategories[word] = (NN_category, max_depth, max_similarity)
424     if counter % 200 == 0:
425        print("%5d / %5d: %d %4.2f %s | %s"
426          %(counter, len(dictionary), max_depth, max_similarity, word, NN_category))
427     counter += 1
428
429  OUT = open("xllm6_assignedCategories.txt", "w")
430  for word in assignedCategories:
```

```
    OUT.write(word+"\t"+str(assignedCategories[word])+"\n")
OUT.close()
```

8.3 Predicting article performance and clustering using LLMs

The dataset consists of 4000 articles published between 2015 and 2020, on Data Science Central. For each article, the following information is available: title, publication date, author, URL, type of article (forum question or blog post), and the number of pageviews measured at the end of the time period in question. The goal is to identify patterns that lead to unusual pageview numbers, to automatically suggest great titles to contributing authors. The dataset does not contain the full articles, only the titles. It would be useful to assess conversion rates (new subscribers) or sales based on the wording found in the subject and content, and the match between both, to maximize pageviews and conversions simultaneously.

Nevertheless, titles alone lead to very interesting results. I also use efficient techniques to predict pageview numbers given the title and category, and to cluster high performing articles into meaningful overlapping groups. A category is defined as a combination of attributes: author if among the top 10, blog or forum question, and URL pointing either to Data Science Central or a satellite channel. The dataset is on GitHub, here.

The methodology relies heavily on text processing and may be extended to much larger datasets. In this case, to boost efficiency, I recommend splitting the data into multiple subsets treated separately: a subset may correspond to a specific channel. This approach is similar to the multi-LLM architecture discussed in section 8.2.

The main concepts and notations are as follows:

- Instead of vector or graph databases, tables are structured as nested hashes. These are key-value tables (dictionary in Python), where the value is also a hash. For instance, the distance matrix used in standard clustering algorithms is a nested hash transformed into a matrix. See lines 314–330 in the code in section 8.3.3. Another useful operation is hash inversion, see lines 409–420 in the code. Nested hashes are very efficient to represent sparse data; they can be compared to matrices where both rows and columns have a variable number of elements. Hash inversion is similar to matrix transposition.
- Tokens are replaced by multi-tokens [8], referred to as "words" or contextual tokens. For instance, 'data', 'science', 'data∼science' and 'data^science' are multi-tokens. Of course, the first two are also single tokens. In 'data^science', the tokens 'data' and 'science' are not adjacent in the title; I use it to leverage large context, for titles that contain both tokens in arbitrary locations other than adjacent. But 'data∼science' corresponds to the classical word with adjacent tokens.
- The pageview function is denoted as $\texttt{pv}$. At the basic level, $\texttt{pv}(A)$ is the pageview number of article A, based on its title and categorization. It must be normalized, taking the logarithm: see lines 122–123 in the code. Then, the most recent articles have a lower $\texttt{pv}$ because they have not accumulated much traffic yet. To correct for this, see lines 127–136 in the code. From now on, $\texttt{pv}$ refers to normalized pageview counts also adjusted for time. The pageview for a multi-token t is then defined as

$$\texttt{pv}(t) = \frac{1}{|S(t)|} \cdot \sum_{A \in S(t)} \texttt{pv}(A), \tag{8.2}$$

  where $S(t)$ is the set of all article titles containing t, and $|\cdot|$ is the function that counts the number of elements in a set. Sometimes, two different tokens t_1, t_2 have $S(t_1) = S(t_2)$. In this case, to reduce the number of tokens, I only keep the longest one. This is done in lines 193–206 in the code.
- Likewise, you can define $\texttt{pv}(C)$, the pageview count attached to a category C, by averaging $\texttt{pv}$'s over all articles assigned to that category. Finally, $T(A)$ denotes the set of multi-tokens attached to an article A.

With the notations and terminology introduced so far, it is very easy to explain how to predict the pageview count $\texttt{pv}_0(A)$ for an article A inside or outside the training set. The formula is

$$\texttt{pv}_0(A) = \frac{1}{W_A} \cdot \sum_{t \in T(A)} w_t \cdot \texttt{pv}(t), \tag{8.3}$$

with:

$$W_A = \sum_{t \in T(A)} w_t, \quad w_t = 0 \text{ if } |S(t)| \leq \alpha, \quad w_t = \frac{1}{|S(t)|^\beta} \text{ if } |S(t)| > \alpha.$$

Here $\alpha, \beta > 0$ are parameters. I use $\alpha = 1$ and $\beta = 2$. The algorithm puts more weights on rare tokens, but a large value of β or a small value of α leads to overfitting. Also, I use the notation $\texttt{pv}_0$ for an estimated value or

prediction, and pv for an observed value. In some cases, $T(A)$ is empty and thus Formula (8.3) is meaningless. The solution consists in replacing the predicted value by $\texttt{pv}_0(A) = \texttt{pv}(C_A)$, where C_A is the category attached to article A.

The prediction formula (8.3) looks like a regression, but with no regression coefficients to estimate. Also, the dimension of the feature vector is variable: it depends on A. Despite not being a minimization problem, thus the absence of loss function, gradient descent, or neural networks, the predictions are remarkably accurate, with a fairly small bias. It illustrates the power of explainable AI. You can further improve the results using the recalibration technique in lines 459–475 in the code. It now involves one parameter, and a loss function that is a fast approximation to the Kolmogorov-Smirnov distance between two empirical distributions (ECDF): one based on observed pv, and the other one on predicted pv.

I now describe the unsupervised clustering procedure, used to detect groups of articles that perform well. First, you define a similarity metric $s(t_1, t_2)$ between two multi-tokens t_1, t_2. You use one of the clustering methods to group the multi-tokens into clusters, based on the similarity. Then, for each multi-token group G, you retrieve the list $L(G)$ of articles belonging to G with the formula

$$L(G) = \bigcup_{t \in G} S(t). \tag{8.4}$$

The similarity metric is stored in the hash_pairs table in the code, indexed by multi-token pairs: see lines 276–301. The hash structure is far more efficient than a matrix because most pairs $\{t_1, t_2\}$ have $s(t_1, t_2) = 0$ and are not even stored in the hash. But standard clustering procedures in Python libraries use a distance matrix instead. I turned the similarity hash into a distance matrix with the formula $d(t_1, t_2) = 1 - s(t_1, t_2)$. The resulting matrix is big and extremely sparse, resulting in memory problems in some cases. The workaround is to use a different clustering technique, such as my very fast connected components algorithm (see section 5.2.2), designed to work with hash tables rather than matrices. The similarity metric is defined as

$$s(t_1, t_2) = \frac{|S(t_1) \cap S(t_2)|}{|S(t_1) \cup S(t_2)|} \in [0, 1]. \tag{8.5}$$

Note that the multi-token groups are not overlapping, but the article groups are. Finally, I tried two clustering methods: hierarchical clustering (also called agglomerative) from the Sklearn library, and k-medoids [Wiki] from the Sklearn_extra library. The latter is a variant of k-means that works with any distance matrix. The detected clusters for both methods are on GitHub, respectively here and here.

8.3.1 Project and solution

The goal is not to write code from scratch to solve the problems discussed earlier. Instead, I invite you to read my code, understand it, and run it. Then identify and play with the parameters, the impact they have on various components, and avoid overfitting. In the end, you should be able to improve some of the algorithms, especially by replacing Python libraries used for clustering, with methods that can handle sparse graphs efficiently, such as connected components. Other key points to consider: testing different loss functions, automating multi-feature category detection, fine-tuning, sample size determination, confidence intervals for predicted pv, and trying other similarity metrics, tokens reduction and time-adjustment techniques.

The project consists of the following steps:

Step 1: High performance keywords. Download and explore the input dataset available here, run my code, and identify keywords found in article titles, with either above or below average pageview counts. Understand how I assign a category to a title. What are the best and worst performing categories? See lines 224–243 in the code in section 8.3.3. How are categories used to predict title pv? Does it make sense to work with more granular categories?

Step 2: Clustering articles. What are the top parameters influencing the clustering algorithms in the code, besides param_N specifying the number of clusters? How do they influence the cluster structure? Note that one of the clusters is very large, no matter which clustering method you use: see cluster with label 0, here. How would you fix this problem? Finally, replace the Python libraries used for clustering, by a far more efficient procedure that efficiently handles sparse graphs.

Step 3: Predictions, sample size, and evaluation. How would you compute confidence intervals for each predicted pv, based on sample size? Predicted pv for each article are stored in the predicted array, and computed in lines 431–454 in the code. The observed values are stored in the observed array. Finally, use cross-validation to better assess the quality of predictions, and to detect when overfitting is present.

Step 4: Fine-tuning. Identify all the parameters that may have an impact on predicted pv: param_W1, param_W2, param_G and so on. To normalize pv, I use a log transform, see lines 122–124 in the code. Why? Then, to take into account the impact of time on pv, that is, to work with time-adjusted pv, I use a square root transform of the time with two parameters param_T1 and param_T2, see lines 133–136. Finally, I use a linear transform with one parameter param_Z to remove the bias in predicted pv, see line 474. Try other parametric transforms. How would you find the best transforms and best parameters?

Now, here is my answer to Step 1. Best performing keywords include: 'cheat sheet', 'free book', 'learn Python', 'machine learning algorithms', 'R + Python', 'explained in one picture', 'explained in simple English', and 'great articles'. Among the worst performers: 'data security', 'IoT', 'Apache', 'business data', 'C++', 'web scraping' and 'big data'. Keep in mind that the data was collected in 2020. As for categories, blog posts do a lot better than forum questions.

The category table nlp_scoring_categories.txt is stored here on GitHub, and the multi-tokens table nlp_scoring_multi_tokens.txt, here in the same folder. Due to the small number of articles, I bundled all authors but the most prolific into one group. Also, it makes sense to further group small categories together. Categories span across multiple categorical features, and are jointly encoded using the smart encoding technique described here. How much granularity you can afford depends on the sample size: here, about 4000. You want to make sure each category has at least 50 titles so that its average pv is stable enough. Categories with high pv variance should be split or re-arranged to keep variance low.

To answer Step 2, there are several parameters influencing the final number of multi-tokens, which in turn also impacts the clusters. But the one directly related to the clustering algorithm is param_S in line 278, taking a value between 0 and 1. The closer to 1, the more homogeneous the clusters, and the smaller the dimension of the distance matrix. Finally, to split the large cluster, you need to access the dendrogram and linkage structure produced in line 394, or perform clustering on the large cluster to obtain subgroups, or use my connected components algorithm for clustering: the last option gives you full control over the clustering procedure.

Now moving to Step 3. A simple method to compute model-free confidence intervals (CI), in this case for pv, consists of selecting 100 subsamples each with (say) $N = 3000$ articles from the training set (out of 4000), and compute $\text{pv}_0(A)$ for each article A in each subsample. This gives you a range of values for $\text{pv}_0(A)$. The minimum and maximum determines your CI at level $1 - 1/100 = 99\%$. This method is known as bootstrapping [Wiki]. The length L_N of your CI depends on N, and in general, L_N is almost proportional to $1/\sqrt{N}$. Despite the numerous computations involved, in this case the procedure is very fast as there is no neural network involved.

Finally, regarding Step 4, see lines 402–404 for the parameters influencing the predicted pv. It is tempting to use a gradient descent algorithm to optimize all parameters jointly by minimizing the loss function defined in lines 465-471. However, this may lead to overfitting. It is better to decouple the problem: separately optimize the parameters attached to time adjustment (lines 129–136), categorization (line 148), pv normalization (lines 122–124), and predictions (see lines 459–475). Also, it is possible to replace the summation (8.2) by a weighted sum as in (8.3), for instance giving a different weight to each article based on its category or on the title length.

pv	Title
9.042	AI vs Deep Learning vs Machine Learning
8.242	Machine Learning vs. Traditional Statistics: Different philosophies, Different Approaches
9.422	Machine Learning vs. Traditional Statistics: Different philosophies, Different Approaches
5.635	A Comparative Roundup: Artificial Intelligence vs. Machine Learning vs. Deep Learning
7.168	Artificial Intelligence vs. Machine Learning vs. Deep Learning
6.715	AI vs. Machine Learning vs. Deep Learning: What is the Difference?
7.717	Machine Learning vs Statistics vs Statistical Learning in One Picture
7.855	Supervised Learning vs Unsupervised & Semi Supervised in One Pi...
9.185	Python vs R: 4 Implementations of Same Machine Learning Technique
6.907	MS Data Science vs MS Machine Learning / AI vs MS Analytics

Table 8.4: Cluster linked to {'Learning~vs', 'Machine^vs', 'Machine~Learning~vs'}

The remaining of this project is devoted to visualizations pertaining to the various components discussed earlier. Table 8.4 features one of the 20 article clusters based on hierarchical clustering of multi-tokens. In this case, the multi-token group, automatically detected, is {'Learning~vs', 'Machine^vs', 'Machine~Learning~vs'}. It is a mix of standard and contextual tokens. The corresponding article cluster was automatically retrieved using Formula (8.4). Note that two titles are identical but have different pv's. This is due to posting the same article twice; each has a different URL.

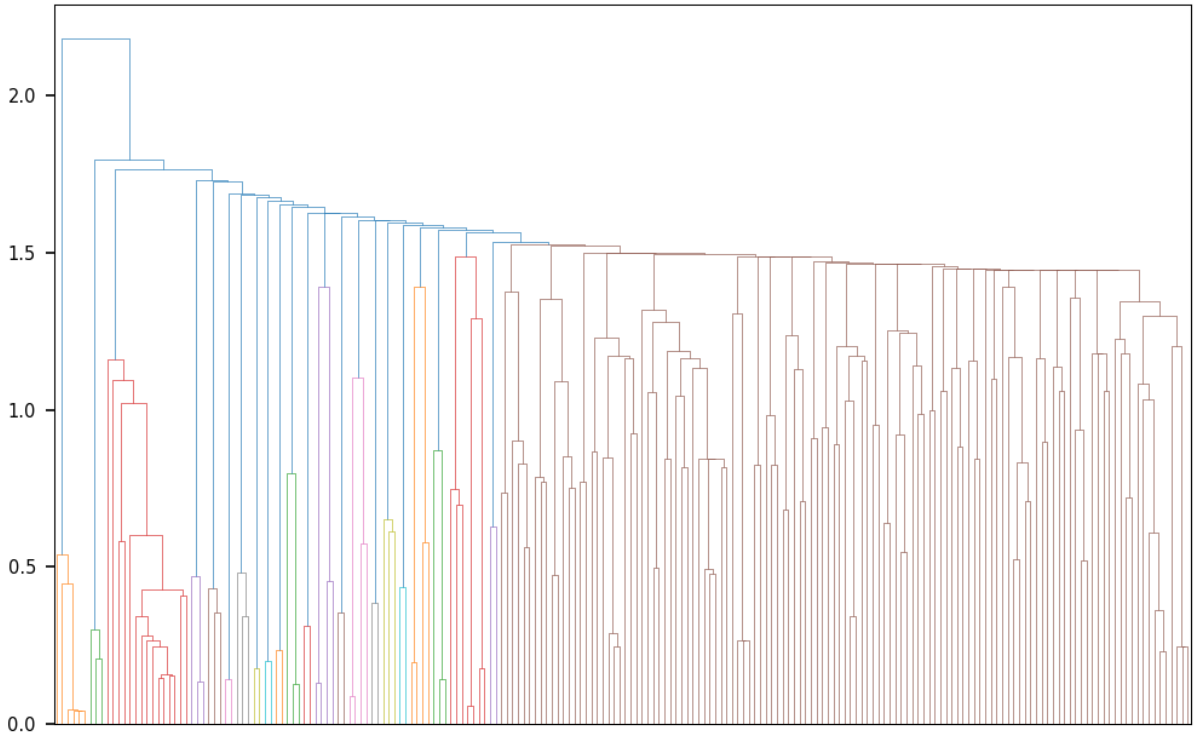


Figure 8.5: Multi-tokens hierarchical clustering: dendrogram (20 groups, 104 multi-tokens)

8.3.2 Visualizations and discussion

Figure 8.5 shows the dendrogram associated to the clustering algorithm. The goal here was to cluster good performing articles, thus the number of multi-tokens is small (104) because I only used those with highest `pv`'s. The rightmost group in brown is unusually large and could be split into 5 subgroups, based on the dendrogram. Each multi-token – a final node in the dendrogram tree – has several articles connected to it: this is not shown in the picture, but see Table 8.4 for an example combining 3 multi-tokens, corresponding to one of the small clusters in the dendrogram.

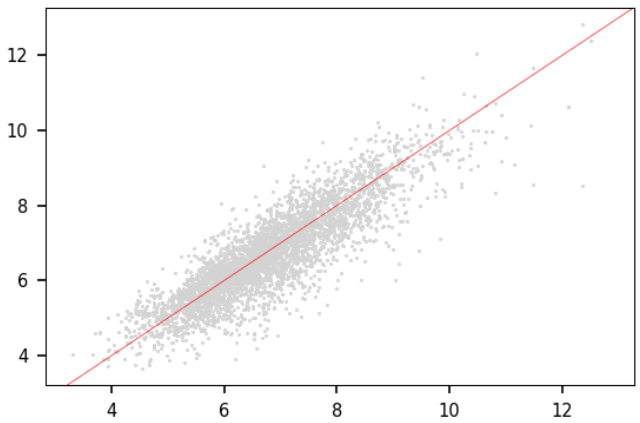


Figure 8.6: Scatterplot, observed vs predicted article `pv` (4000 articles)

Figure 8.6 shows predicted versus observed `pv`'s in a scatterplot. It is based on the initial version of the algorithm, without parameters nor the loss function. You can barely see a small bias, with large `pv`'s slightly underestimated, and smaller ones overestimated. I also observed the same bias in various tests. That was the reason why I introduced the parameter `param_Z` and a loss function to recalibrate the empirical distribution of predicted `pv`'s. That said, the fit is pretty good even before the upgrade. Multi-tokens found in only one article were excluded to avoid overfitting, by setting `param_W1=1`.

In Figure 8.7, the decline in pageviews over time is quite noticeable, but seems to taper off at the end. However, it is artificially created by the fact that the most recent articles haven't accumulated much traffic yet.

Figure 8.8 shows the corrected numbers: it represents the reality more faithfully; the transform used here is non linear.

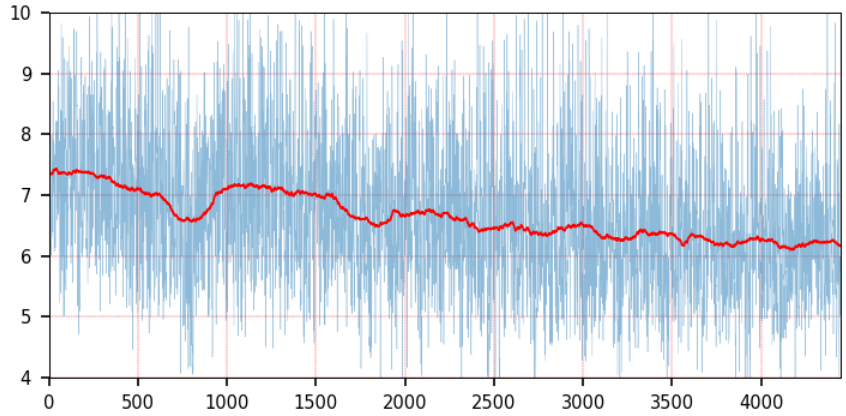


Figure 8.7: Article pv over time, before adjusting for time

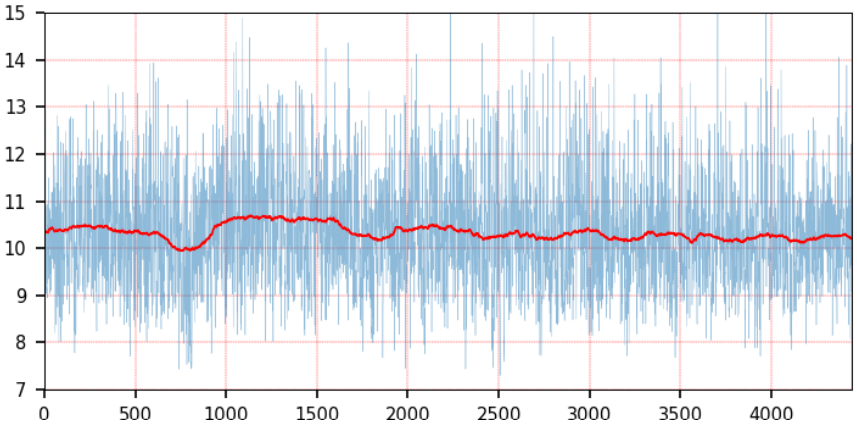


Figure 8.8: Article pv over time, after time adjustment

8.3.3 Python code

The code is also on GitHub, here.

```
# nlp_scoring.py | vincentg@mltechniques.com

import pandas as pd
import numpy as np
import matplotlib as mpl
from matplotlib import pyplot as plt

mpl.rcParams['lines.linewidth'] = 0.3
mpl.rcParams['axes.linewidth'] = 0.5
plt.rcParams['xtick.labelsize'] = 7
plt.rcParams['ytick.labelsize'] = 7


#--- [1] Read data

# read data either from GitHub, or use local copy, depending on path

# path =
    "https://raw.githubusercontent.com/VincentGranville/Statistical-Optimization/main/"
path = ""
filename = "Articles-Pageviews.txt"
url = path + filename

```

```
data = pd.read_csv(url, sep='\t', engine='python', encoding='cp1252')
header = ['Title', 'URL', 'Author', 'Page views', 'Creation date', 'Status']


#--- [2] Core functions and tables

# pv is aggregated normalized pageview; pv_rel is average pv for a specific group

arr_titles = data['Title'] # article titles
arr_status = data['Status'] # blog post, forum question, etc.
arr_pv  = data['Page views'] # article absolute (un-normalized) pageviews
arr_url = data['URL']    # article URL
arr_author = data['Author'] # article author
arr_categories = {}      # article category (combo of author, status, URL extracts)

category_pv = {}  # key: article category; value: average pv per category
category_count = {} # key: article category; value: number of articles in that category
hash_words_count = {} # key: word; value: number of occurrences across all articles
hash_pv = {}      # key: word; value: aggregated pv across all those articles
hash_titles = {}  # key: word; value: hash of articles containing word (value: pv)
hash_authors_count = {} # key: author;

def compress_status(status):
   status = status.lower()
   if 'forum' in status:
      status = 'forum_question'
   elif 'resource' in status:
      status = 'resource'
   else:
      status = 'blog'
   return(status)

def compress_url(url):
   url = url.lower()
   if 'datasciencecentral' in url:
      url = 'DSC'
   else:
      url = 'Other'
   return(url)

def update_hash(titleID, word, hash_words_count, hash_pv, hash_titles):

   if word in hash_words_count:
      hash_words_count[word] += 1
      hash_pv[word] += pv # what if words found twice in same title ??
      hash = hash_titles[word]
      hash[titleID] = pv
      hash_titles[word] = hash
   else:
      hash_words_count[word] = 1
      hash_pv[word] = pv
      hash_titles[word] = { titleID : pv }
   return(hash_words_count, hash_pv, hash_titles)

def update_single_tokens(titleID, words, hash_words_count, hash_pv, hash_titles):

   # words is an array of words (found in title)

   for word in words:
      hash_words_count, hash_pv, hash_titles = update_hash(titleID, word,
                           hash_words_count, hash_pv, hash_titles)
   return(hash_words_count, hash_pv, hash_titles)

def update_joint_tokens(titleID, words, hash_words_count, hash_pv, hash_titles):

   # words is an array of words (found in title)
```

```
    # capturing adjacent tokens

    for idx in range(len(words)):
       word = words[idx]
       if idx < len(words)-1 and words[idx+1] != '':
          word += "~" + words[idx+1]
          hash_words_count, hash_pv, hash_titles = update_hash(titleID, word,
                                hash_words_count, hash_pv, hash_titles)
       if idx < len(words)-2 and words[idx+2] != '':
          word += "~" + words[idx+2]
          hash_words_count, hash_pv, hash_titles = update_hash(titleID, word,
                                hash_words_count, hash_pv, hash_titles)
    return(hash_words_count, hash_pv, hash_titles)

def update_disjoint_tokens(titleID, words, hash_words_count, hash_pv, hash_titles):

    # words is an array of words (found in title)
    # word1 and word2 cannot be adjacent:
    #   if they were, this is already captured in the update_joint_tokens function

    param_D = 1 # word1 and word2 must be separated by at least param_D tokens

    for k in range(len(words)):
       for l in range(len(words)):
          word1 = words[k]
          word2 = words[l]
          distance = abs(k - l)
          if word1 < word2 and distance > param_D and word1 != '' and word2 != '':
             word12 = word1 + "^" + word2
             hash_words_count, hash_pv, hash_titles = update_hash(titleID,
                           word12, hash_words_count, hash_pv, hash_titles)
    return(hash_words_count, hash_pv, hash_titles)

def get_article_pv(titleID, arr_pv):
    # using log: it gives a better normalization and fit than sqrt, for pv distribution
    return(np.log(float(arr_pv[titleID])))


#--- [3] De-trend pv

param_T1 = 0.80
param_T2 = 0.11
arr_pv_new = np.zeros(len(arr_pv))

for k in range(len(arr_pv)):
    energy_boost = param_T1 * np.sqrt(k + param_T2 * len(arr_pv))
    arr_pv_new[k] = arr_pv[k] * (1 + energy_boost)
arr_pv = np.copy(arr_pv_new)


#--- [4] Populate core tables

for k in range(len(data)):
    author = arr_author[k]
    if author in hash_authors_count:
       hash_authors_count[author] +=1
    else:
       hash_authors_count[author] =1

param_A = 50 # authors with fewer than param_A articles are bundled together

for k in range(len(data)):

    pv = get_article_pv(k, arr_pv)
    cstatus = compress_status(arr_status[k])
    curl = compress_url(arr_url[k])
```

```
155     category = curl + "~" + cstatus
156     author = arr_author[k]
157     if hash_authors_count[author] > param_A:
158        arr_categories[k] = category + "~" + author
159     else:
160        arr_categories[k] = category
161
162     words = str(arr_titles[k]).replace(',',' ').replace(':',' ').replace('?', ' ')
163     words = words.replace('.',' ').replace('(',' ').replace(')', ' ')
164     words = words.replace('-',' ').replace(' ',' ').replace('\xa0', ' ')
165     # words = words.lower()
166     words = words.split(' ')
167
168     if 'DSC~resource' in category or 'DSC~blog' in category:
169        hash_words_count, hash_pv, hash_titles = update_single_tokens(k, words,
170                              hash_words_count, hash_pv, hash_titles)
171        hash_words_count, hash_pv, hash_titles = update_joint_tokens(k, words,
172                              hash_words_count, hash_pv, hash_titles)
173        hash_words_count, hash_pv, hash_titles = update_disjoint_tokens(k, words,
174                              hash_words_count, hash_pv, hash_titles)
175
176 mean_pv = sum(hash_pv.values()) / sum(hash_words_count.values())
177 print("Mean pv: %6.3f" % (mean_pv))
178
179
180 #--- [5] Sort, normalize, and dedupe hash_pv
181
182 # Words with identical pv are all attached to the same set of titles
183 # We only keep one of them (the largest one) to reduce the number of words
184
185 eps = 0.000000000001
186 hash_pv_rel = {}
187
188 for word in hash_pv:
189    hash_pv_rel[word] = hash_pv[word]/hash_words_count[word]
190
191 hash_pv_rel = dict(sorted(hash_pv_rel.items(), key=lambda item: item[1], reverse=True))
192
193 hash_pv_deduped = {}
194 old_pv = -1
195 old_word = ''
196 for word in hash_pv_rel:
197    pv = hash_pv_rel[word]
198    if abs(pv - old_pv) > eps:
199       if old_pv != -1:
200          hash_pv_deduped[old_word] = old_pv
201       old_word = word
202       old_pv = pv
203    else:
204       if len(word) > len(old_word):
205          old_word = word
206 hash_pv_deduped[old_word] = old_pv
207
208 print()
209 print("=== DEDUPED WORDS: titles count, relative pv (avg = 1.00), word\n")
210 input("> Press <Enter> to continue")
211
212 for word in hash_pv_deduped:
213    count = hash_words_count[word]
214    if count > 20 or count > 5 and '~' in word:
215       print("%6d %6.3f %s" %(count, hash_pv_rel[word]/mean_pv, word))
216
217
218 #--- [6] Compute average pv per category
219
220 # Needed to predict title pv, in addition to word pv's
```

```
# Surprisingly, word pv's have much more predictive power than category pv's
# For new titles with no decent word in historical tables, it is very useful

for k in range(len(data)):
   category = arr_categories[k]
   pv = get_article_pv(k, arr_pv)
   if category in category_count:
      category_pv[category] += pv
      category_count[category] += 1
   else:
      category_pv[category] = pv
      category_count[category] = 1

print()
input("> Press <Enter> to continue")
print()
print("=== CATEGORIES: titles count, pv, category name\n")

for category in category_count:
   count = category_count[category]
   category_pv[category] /= count
   print("%5d %6.3f %s" %(count, category_pv[category], category))
print()


#--- [7] Create short list of frequent words with great performance

# This reduces the list of words for the word clustering algo in next step
# Goal: In next steps, we cluster groups of words with good pv to
#     categorize various sources of good performance

short_list = {}
keep = 'good' # options: 'bad' or 'good'

param_G1 = 1.10 # must be above 1, large value to get titles with highest pv
parma_G2 = 0.90 # must be below 1, low value to get articles with lowest pv
param_C1 = 10 # single-token word with count <= param_C1 not included in short_list
param_C2 = 4 # multi-token words with count <= param_C2 not included in short_list

for word in hash_pv_deduped:
   count = hash_words_count[word]
   pv = hash_pv[word]/count
   if keep == 'good':
      flag = bool(pv > param_G1 * mean_pv)
   elif keep == 'bad':
      flag = bool(pv < param_G2 * mean_pv)
   if flag and (count > param_C1 or count > param_C2 and '~' in word):
      short_list[word] = 1


#--- [8] compute similarity between words in short list, based on common titles

# Find list of articles S1 and S2, containing respectively word1 and word2
# word1 is similar to word2 if |S1 intersection S2| / |S1 union S2| is high

hash_pairs = {}
aux_list = {}
param_S = 0.20 # if similarity score about this threshold, word1 and word2 are linked

for word1 in short_list:
   for word2 in short_list:

      set1 = set()
      for titleID1 in hash_titles[word1]:
         set1.add(titleID1)
      set2 = set()
```

```
        for titleID2 in hash_titles[word2]:
            set2.add(titleID2)

        count1 = len(set1)
        count2 = len(set2)
        count12 = len(set.union(set1, set2))
        similarity = len(set.intersection(set1, set2)) / count12

        if similarity > param_S and word1 < word2:
            hash_pairs[(word1, word2)] = similarity
            hash_pairs[(word2, word1)] = similarity
            hash_pairs[(word1, word1)] = 1.00
            hash_pairs[(word2, word2)] = 1.00
            aux_list[word1] = 1
            aux_list[word2] = 1


#--- [9] Turn hash_pairs{} into distance matrix dist_matrix, then perform clustering

# Keyword clustering based on similarity metric computed in previous step
# Alternative to exploite sparse matrix: connected components algorithm on hash_pairs
# Connected components is much faster, works with big graphs
# In addition, not subject to deprecated parameters unlike Sklearn clustering
# Source code for connected components: https://mltblog.com/3UDJ2tR

param_N = 20 # prespecified number of clusters in word clustering
n = len(aux_list)
dist_matrix = [[0 for x in range(n)] for y in range(n)]
arr_word = []

i = 0
for word1 in aux_list:
    arr_word.append(word1)
    j = 0
    for word2 in aux_list:
        key = (word1, word2)
        if key in hash_pairs:
            # hash_pairs is based on similarity; dist_matrix = 1 - hash_pairs is distance
            dist_matrix[i][j] = 1 - hash_pairs[(word1, word2)]
        else:
            # assign maximum possible distance if i, j are not linked
            dist_matrix[i][j] = 1.00 # maximum possible distance
        j = j+1
    i = i+1

#- Clustering, two models: hierarchichal and k-medoids, based on distance matrix

from sklearn.cluster import AgglomerativeClustering
hierarch = AgglomerativeClustering(n_clusters=param_N,linkage='average').fit(dist_matrix)

# !pip install scikit-learn-extra
from sklearn_extra.cluster import KMedoids
kmedoids = KMedoids(n_clusters=param_N,random_state=0).fit(dist_matrix)

#- Now showing the clusters obtained from each model

def show_clusters(model, hash_titles, arr_titles, arr_pv):

    groups = model.labels_
    hash_group_words = {}
    for k in range(len(groups)):
        group = groups[k]
        word = arr_word[k]
        if group in hash_group_words:
            hash_group_words[group] = (*hash_group_words[group], word)
        else:
```

```
353             hash_group_words[group] = (word,)
354
355     hash_group_titles = {}
356     for group in hash_group_words:
357         words = hash_group_words[group]
358         thash = {}
359         for word in words:
360             for titleID in hash_titles[word]:
361                 thash[titleID] = 1
362         hash_group_titles[group] = thash
363
364     for group in hash_group_words:
365         print("-----------------------------------------")
366         print("Group", group)
367         print()
368         print("keywords:", hash_group_words[group])
369         print()
370         print("Titles with normalized pv on the left:")
371         print()
372         for titleID in hash_group_titles[group]:
373             pv = get_article_pv(titleID, arr_pv)
374             print("%6.3f %s" %(pv, arr_titles[titleID]))
375         print("\n")
376
377     return(hash_group_words, hash_group_titles)
378
379 print("\n\n=== CLUSTERS obtained via hierarchical clustering\n")
380 input("> Press <Enter> to continue")
381 show_clusters(hierarch, hash_titles, arr_titles, arr_pv)
382
383 print("\n\n=== CLUSTERS obtained via k-medoid clustering\n")
384 input("> Press <Enter> to continue")
385 show_clusters(kmedoids, hash_titles, arr_titles, arr_pv)
386
387 input(">Press <Enter> to continue")
388
389 #- plot dendogram related to dist_matrix
390
391 from scipy.cluster.hierarchy import dendrogram, linkage
392 # Doc for Scipy dendograms: https://mltblog.com/3UG0C08
393
394 Z = linkage(dist_matrix) # Exercise: use Z to split the large group
395 dendrogram(Z)
396 plt.show()
397
398
399 #--- [10] Predicting pv
400
401 # Need to do cross-validation in the future
402 # Influenced by: param_W1, param_W2, param_G1, param_C1, param_C2, param_A, param_D
403 #             param_T1, param_T2
404 # Not influenced by: param_N, param_S
405 # Large param_W2 combined with small param_W1 may lead to overfitting
406
407 # We need the build inverted ("transposed") hash_titles, named reversed_hash_titles
408
409 reversed_hash_titles = {}
410
411 for word in hash_titles:
412     pv_rel = hash_pv_rel[word]
413     hash = hash_titles[word]
414     for titleID in hash:
415         if titleID in reversed_hash_titles:
416             rhash = reversed_hash_titles[titleID]
417         else:
418             rhash ={}
```

```
        rhash[word] = pv_rel
        reversed_hash_titles[titleID] = rhash

# Now predicting pv

observed = []
predicted = []
missed = 0
n = 0
param_W1 = 1 # low value increases overfitting, should be >= 1
param_W2 = 2.00 # chosen to minimize error between pv and estimated_pv

for titleID in reversed_hash_titles:
   pv = get_article_pv(titleID, arr_pv)
   rhash = reversed_hash_titles[titleID]
   n += 1
   count = 0
   sum = 0
   for word in rhash:
      weight = hash_words_count[word]
      booster = 1.00
      if '~' in word:
         booster = 1.00 # test a different value
      elif '^' in word:
         booster = 1.00 # test a different value
      if weight > param_W1:
         count += booster * (1/weight)**param_W2
         sum += booster * (1/weight)**param_W2 * rhash[word]
   if count > 0:
      estimated_pv = sum / count
   else:
      missed += 1
      category = arr_categories[titleID]
      estimated_pv = category_pv[category]
   observed.append(pv)
   predicted.append(estimated_pv)

observed = np.array(observed)
predicted = np.array(predicted)
mean_pv = np.mean(observed)
min_loss = 999999999.99
param_Z = 0.00

for test_param_Z in np.arange(-0.50, 0.50, 0.05):

   scaled_predicted = predicted + test_param_Z * (predicted - mean_pv)
   loss = 0
   for q in (.10, .25, .50, .75, .90):
      delta_ecdf = abs(np.quantile(observed,q)-np.quantile(scaled_predicted,q))
      if delta_ecdf > loss:
         loss = delta_ecdf
   if loss < min_loss:
      min_loss = loss
      param_Z = test_param_Z

predicted = predicted + param_Z * (predicted - mean_pv)
loss = min_loss
mean_estimated_pv = np.mean(predicted)
mean_error = np.mean(np.abs(observed-predicted))
correl = np.corrcoef(observed, predicted)

plt.axline((min(observed),min(observed)),(max(observed),max(observed)),c='red')
plt.scatter(predicted, observed, s=0.2, c ="lightgray", alpha = 1.0)

plt.show()
```

```
print()
print("=== PREDICTIONS\n")
print("Predicted vs observed pageviews, for 4000 articles\n")
print("Loss: %6.3f" %(loss))
print("Missed titles [with pv estimared via category]: ",missed, "out of", n)
print("Mean pv (observed) : %8.3f" %(mean_pv))
print("Mean pv (estimated): %8.3f" %(mean_estimated_pv))
print("Mean absolute error: %8.3f" %(mean_error))
print("Correl b/w observed and estimated pv: %8.3f" %(correl[0][1]))
print()
print("Observed quantiles (left) vs prediction-based (right)")
print("P.10: %8.3f %8.3f" %(np.quantile(observed, .10), np.quantile(predicted, .10)))
print("P.25: %8.3f %8.3f" %(np.quantile(observed, .25), np.quantile(predicted, .25)))
print("P.50: %8.3f %8.3f" %(np.quantile(observed, .50), np.quantile(predicted, .50)))
print("P.75: %8.3f %8.3f" %(np.quantile(observed, .75), np.quantile(predicted, .75)))
print("P.90: %8.3f %8.3f" %(np.quantile(observed, .90), np.quantile(predicted, .90)))

#- Plot normalized pv of articles over time

y = np.zeros(len(arr_pv))
for k in range(len(arr_pv)):
  y[k] = get_article_pv(k, arr_pv)

z = np.zeros(len(arr_pv))
window = 120 # for moving average
for k in range(len(arr_pv)):
  if k-window < 0:
     z[k] = np.mean(y[0:k+window])
  elif k+window > len(arr_pv):
     z[k] = np.mean(y[k-window:len(arr_pv)-1])
  else:
     z[k] = np.mean(y[k-window:k+window])

plt.plot(range(len(arr_pv)), y, linewidth = 0.2, alpha = 0.5)
plt.plot(range(len(arr_pv)), z, linewidth = 0.8, c='red', alpha = 1.0)

plt.xlim(0, len(arr_pv))
#plt.ylim(7, 15)
plt.grid(color='red', linewidth = 0.2, linestyle='--')
plt.show()
```

Appendix A

Glossary: GAN and Tabular Data Synthetization

The following list features the most important concepts related to tabular data synthetization and evaluation methods, with a focus on generative adversarial networks.

activation function	Function transforming values from the last layer of a deep neural network such as GAN, into actual output. For dummy variables, it is customary to use softmax.
algorithmic bias	Algorithm are designed by architects with their own biases, train on data reflecting these biases (for instance, pictures of mostly white people) and decision from blackbox systems (who gets a loan) impacted by these biases. Synthetic data can help address this issue.
base distance	When evaluating generated data, you compare your synthetic data with the validation data, a subset of the real data not use for training. The base distance is the distance between the part of the real data not used for training (the validation set), and the part of the real data actually used for training.
batch	In GAN implementations, during each epoch (a full run of the dataset), you synthetize small batches of data and evaluate these batches separately one at a time, as it is a lot faster than doing it on the whole data at once.
binning	Many algorithms such as XGboost work on binned data, where feature values - either jointly or separately - are aggregated into buckets called bins of flag vectors. Bin counts also work well with categorical data.
categorical feature	A non numerical feature sometimes represented by dummy variables, one per category value, such as disease type or keyword. It can lead to a large number of features, artificially increasing the dimension of the problem. Grouping and aggregation techniques can reduce the dimensionality.
copula	Data synthetization technique based on empirical quantiles and the feature correlation matrix, generalizing the inverse transform sampling method to multivariate data.
correlation matrix	The distance between two correlation matrices, one computed on the real data and the other one on the synthetic data, is a fundamental evaluation metric to measure the quality of the generated data.
Cramer's V	A generalization of the correlation coefficient to measure the association between categorical features, or between a categorical and numerical feature. The value is between 0 (no association) and 1 (strong association).
data augmentation	The method consists of adding synthetic observations to your training set, to produce more robust predictions or classifications. By enriching the training set, your algorithm will be better trained to deal with future real data not in the training set.

data cleaning	Required step before using any modeling technique, to detect outliers, missing values, duplicates, wrong formatting, and so on. Can be automated to a large extent.
dummy variable	Binary feature with two values (0 and 1) to represent categorical information, for instance California = 0/1 to indicate whether the location is in California or not. In this case, you may have 50 dummy variables, one for each state. It allows you to use numerical algorithms on categorical data.
EDA	Exploratory data analysis. Used to detect outliers, unique values with count and frequency (for each feature), percentiles, duplicated and missing values, correlation between features, and empirical distributions. Also used to bin the data.
ECDF	Empirical cumulative distribution function uniquely characterizing the underlying distribution in a dataset. Works with numerical and categorical features. The one-dimensional version is computed for each feature separately.
EPDF	Empirical probability density function. The discrete derivative of the ECDF, and more difficult to handle than ECDF. For discrete variables, there is a one-to-one mapping between ECDF and EPDF.
epoch	Also called iteration. One full run of your real data when training a GAN model. The loss functions (generator and discriminator) are computed at each epoch and should stabilize to low values after thousands of epochs, depending on the hyperparameters.
explainable AI	Set of methods leading to easy interpretation, with simple explanations whenever the blackbox system makes a decision. Explainability can be increased using feature importance scores. Some algorithms such as NoGAN are fully explainable by design.
faithfulness	One of the goals of synthetization is to correctly mimic the statistical distributions and patterns found in the real data. Faithfulness metrics such as KS distance measure how well this is accomplished. Metrics measuring the quality of predictions (via training set augmentation and cross-validation), are called utility metrics. Security metrics measure how well personal information has been transformed.
GAN	Generative adversarial network. Data synthetization technique based on 3 deep neural networks: the generator to generate synthetic observations, the discriminator to distinguish between fake and real data (competing with the generator), and the full model.
gradient descent	Most machine learning algorithms including GAN, aim to minimize a loss function, or equivalently, maximize model fitting to data. Gradient descent performs this task. It may or may not succeed depending on parameters such as learning rate. Neural networks use stochastic gradient descent. Discretized versions are available.
Hellinger distance	Metric to evaluate the quality of synthetizations. Based on the probability density functions (EPDF), computed on the real and synthetic data, then compared. Typically performed on each feature separately.
hexagonal bin	Hexagonal bin plots are scatterplots where each dot is replaced by a fixed-size bin containing a variable number of observations. The color intensity represents the number of observations in each bin. Each bin is hexagonal: this is the optimum shape. The hexagons are arranged in an tessellation with underlying hexagonal lattice.
holdout	The holdout method consists of using a portion of your real data (called training set) to train a synthesizer, and the remaining (called validation set) to evaluate the quality of the generated data.

hyperparameter	In neural networks, parameters are the weights attached to the synapses connecting the neurons. Hyperparameters control the behavior of the whole system, and specify its architecture. For instance: number of epochs, batch size, loss functions, activation functions, learning rate, type of gradient descent, number and type of layers (dense or sparse), and so on.
imbalance	In a dataset, segments with few observations (for instance, fraudulent transactions) cause imbalance. Synthetization allows you to generate more observations for these segments, to balance the dataset to improve the performance of some algorithms.
KS distance	Kolmogorov-Smirnov distance. To evaluate the quality of synthesized data. While the Hellinger distance is based on the density (EPDF) and averaged deviations, KS is based on the maximum deviation between the two ECDFs: real versus synthetic. It is more robust than Hellinger.
latent variable	In GANs, feature values that we cannot interpret directly, but which encode a meaningful internal representation of externally observed events.
learning rate	Parameter that governs increments in gradient descent algorithms. Small values means slow convergence and possibly getting stuck around a local minimum. Large values may lead to missing the optimum or lack of convergence.
loss function	The function to minimize in a gradient descent algorithm. For instance, the maximum KS distance between the generated and real data, in a synthetization problem.
metadata	Information attached to a tabular dataset, specifying the type of data for each column: categorical, ordinal (integer), text, timestamp, continuous feature, and so on.
missing values	Can be encoded as NaN, a blank cell, the empty string ‘’, a large integer, or zero. NoGAN easily handles them. Techniques to retrieve missing values are called imputation methods.
mode collapse	In GANs, mode collapse happens when the generator can only produce a single type of output or a small set of outputs. This may happen due to problems in training, such as the generator finding a type of data that can easily fools the discriminator and thus keeps generating that one type.
multivariate ECDF	Same as ECDF but in this case computed jointly for multiple features, rather than separately for each feature. The computation is not straightforward.
NoGAN	Synthesizer not based on GAN or neural networks. A very efficient one, both in terms of speed and quality of the output, sharing some similarities with XGboost, is described in [14]. The copula and interpolation methods also fall in that category.
overfitting	Synthetic data that looks too good to be true, could be the result of overfitting. This can happen when fine-tuning the hyperparameters to work on one particular dataset. To reduce overfitting, evaluate the quality of a synthetization on a validation set using the holdout method. Or assess performance of predictions based on augmented data, using cross-validation.
oversampling	Consists of producing a larger proportion of synthetic observations for underrepresented segments in the real data (for instance fraudulent transactions), to fix the imbalance problem.
PCA	Principal component analysis. Used as a transform to decorrelate the features in the real data, prior to training GAN, as this can improve synthetizations. The correct correlation structure is then put back into the synthetization, using the inverse PCA transform, after running GAN.
quantile	The empirical quantile function is the inverse of the ECDF. It generalizes percentiles.
reinforcement learning	Machine learning classification technique where correct allocation of future observations (outside the training set) is rewarded, enabling the system to self-learn via trial and error.

replicability	A replicable neural network is one that can produce the exact same results when run multiple times on the same data, regardless of the platform. Usually controlled by a seed parameter: using the same seed leads to the same results.
scaling	A transformation that keeps the values of each feature within the same range, or with the same variance in the real data, before using GAN. A measurement, whether in yards or miles, will be scale-free after the transformation. It can dramatically improve the quality of the generated data. Inverse scaling is then applied to the generated data, after the GAN synthetization.
seed	Parameter used to initialize the various random number generators involved in the GAN architecture, typically one for each Python library that generates random numbers. It produces replicable results, at least with CPU implementations. In GPU, the problem is different.
stopping rule	A criterion to decide when to stop training a GAN, typically when an epoch produces an unusually good synthetization, based on quality evaluation metrics such as the KS distance. It produces much better results than stopping after a fixed number of epochs.
synthetization	Production of generated observations, also called synthetic data, with statistical properties mimicking those computed on a pre-specified real data set.
tabular data	Data arranged in tables, where columns represent features, and rows represent observations. Typically used for transactional data. Time series are treated with specific algorithms.
training set	The portion of your real data used to train your synthesizer. The other part is called the validation set, and used to evaluate the quality of the synthetic data (how well it mimics real data). This setting known as holdout allows you to test you synthetisizer on future data and avoid overfitting.
transform	Similar to transformers in large language models. Consists of using an invertible transform on your real data prior to GAN processing, to improve GAN performance. You need to apply the inverse transform on the generated data, after GAN. Example of transforms: scaling, PCA, standardization (transformed features having the same variance and zero mean), and normalization (to eliminate skewness).
validation set	See training set.
vanishing gradient	When the gradient gets close to zero in a gradient descent algorithm, it can prevent further progress towards locating the optimum. In the worst case, this may completely stop the neural network from further training.
Wasserstein loss	The GAN Wasserstein loss function seeks to increase the gap between the scores for real and generated data. It is one of the many loss functions to improve the gradient descent algorithm, avoiding mode collapse and similar problems in some synthetizations.
WGAN	Wasserstein GAN, based on the Wasserstein loss function.

Appendix B

Glossary: GenAI and LLMs

ANN	Approximate nearest neighbor. Similar to the K-NN algorithm used in supervised classification, but faster and applied to retrieving information in vector databases, such as LLM embeddings stored as vectors. I designed a probabilistic version called pANN, especially useful for model evaluation and improvement, with applications to GenAI, synthetic data, and LLMs. See section 8.1.
diffusion	Diffusion models use a Markov chain with diffusion steps to slowly add random noise to data and then learn to reverse the diffusion process to construct desired data samples from the noise. The output is usually a dataset or image similar but different from the original ones. Unlike variational auto-encoders, diffusion models have high dimensionality in the latent space (latent variables): the same dimension as the original data. Very popular in computer vision and image generation.
embedding	In LLMs, embeddings are typically attached to a keyword, paragraph, or element of text; they consist of tokens. The concept has been extended to computer vision, where images are summarized in small dimensions by a number of numerical features (far smaller than the number of pixels). Likewise, in LLMs, tokens are treated as the features in your dataset, especially when embeddings are represented by fixed-size vectors. The dimension is the number of tokens per embedding. See Figure 8.1 and the token entry.
encoder	An auto-encoder is (typically) a neural network to compress and reconstruct unlabeled data. It has two parts: an encoder that compacts the input, and a decoder that reverses the transformation. The original transformer model was an auto-encoder with both encoder and decoder. However, OpenAI (GPT) uses only a decoder. Variational auto-encoders (VAE) are very popular.
GAN	Generative adversarial network. One of the many types of DNN (deep neural network) architecture. It consists of two DNNs: the generator and the discriminator, competing against each other until reaching an equilibrium. Good at generating synthetic images similar to those in your training set (computer vision). Key components include a loss function, a stochastic gradient descent algorithm such as Adam to find a local minimum to the loss function, and hyperparameters to fine-tune the results. Not good at synthesizing tabular data, thus the reason I created NoGAN: see section 2.1.
GPT	In case you did not know, GPT stands for Generative Pre-trained Transformer. The main application is LLMs. See transformer.
graph database	My LLMs rely on taxonomies attached to the crawled content. Taxonomies consist of categories, subcategories and so on. When each subcategory has exactly one parent category, you use a tree to represent the structure. Otherwise, you use a graph database.

key-value database	Also known as hash table or dictionary in Python. In my LLMs, embeddings have variable size. I store them as short key-value tables rather than long vectors. Keys are tokens, and a value is the association between a token, and the word attached to the parent embedding.
LangChain	Available as a Python library or API, it helps you build applications that read data from internal documents and summarize them. It allows you to build customized GPTs, and blend results to user queries or prompts with local information retrieved from your environment, such as internal documentation or PDFs.
LLaMA	An LLM model that predicts the next word in a word sequence, given previous words. See how I use them to predict the next DNA subsequence in DNA sequencing, in section 7.1. Typically associated to auto-regressive models or Markov chains.
LLM	Large language model. Modern version of NLP (natural language processing) and NLG (natural language generation). Applications include chatbots, sentiment analysis, text summarization, search, and translation.
multi-agent system	LLM architecture with multiple specialized LLMs. The input data (a vast repository) is broken down into top categories. Each one has its own LLM, that is, its own embeddings, dictionary, and related tables. Each specialized LLM is sometimes called a simple LLM. See my own version named xLLM , in section 7.2.2.
multimodal	Any architecture that blends multiple data types: text, videos, sound files, and images. The emphasis is on processing user queries in real-time, to return blended text, images, and so on. For instance, turning text into streaming videos.
normalization	Many evaluation metrics take values between 0 and 1 after proper scaling. Likewise, weights attached to tokens in LLM embeddings have a value between -1 and +1. In many algorithms and feature engineering, the input data is usually transformed first (so that each feature has same variance and zero mean), then processed, and finally you apply the inverse transform to the output. These transforms or scaling operations are known as normalization.
parameter	This word is mostly used to represent the weights attached to neuron connections in DNNs. Different from hyperparameters. The latter are knobs to fine-tune models. Also different from the concept of parameter in statistical models despite the same spelling.
RAG	Retrieval-augmentation-generation. In LLMs, retrieving data from summary tables (embeddings) to answer a prompt, using additional sources to augment your training set and the summary tables, and then generating output. Generation focuses on answering a user query (prompt), on summarizing a document, or producing some content such as synthesized videos.
regularization	Turning a standard optimization problem or DNN into constrained optimization, by adding constraints and corresponding Lagrange multipliers to the loss function. Potential goals: to obtain more robust results, or to deal with over-parameterized statistical models and ill-conditioned problems. Example: Lasso regression. Different from normalization.
reinforcement learning	A semi-supervised machine learning technique to refine predictive or classification algorithms by rewarding good decisions and penalizing bad ones. Good decisions improve future predictions; you achieve this goal by adding new data to your training set, with labels that work best in cross-validation testing. In my LLMs, I let the user choose the parameters that best suit his needs. This technique leads to self-tuning and/or customized models: the default parameters come from usage.

Synthetic data	Artificial tabular data with statistical properties (correlations, joint empirical distribution) that mimic those of a real dataset. You use it to augment, balance or anonymize data. Few methods can synthesize outside the range observed in the real data (your training set). I describe how to do it in section 10.4 in [19]. A good metric to assess the quality of synthetic data is the full, multivariate Kolmogorov-Smirnov distance, based on the joint empirical distribution (ECDF) computed both on the real and generated observations. It works both with categorical and numerical features. The word synthetic data is also used for generated (artificial) time series, graphs, images, videos and soundtracks in multimodal applications.
token	In LLMs or NLP, a token is a single word; embeddings are vectors, with each component being a token. A word such as "San Francisco" is a single token, not two. In my LLMs, I use double tokens, such as "Gaussian distribution" for terms that are frequently found together. I treat them as ordinary (single) tokens. Also, the value attached to a token is its "correlation" (pointwise mutual information) to the word representing its parent embedding, see Figure 8.1. But in traditional LLMs, the value is simply the normalized token frequency computed on some text repository.
transformer	A transformer model is an algorithm that looks for relationships in sequential data, for instance, words in LLM applications. Sometimes the words are not close to each other, allowing you to detect long-range correlations. It transforms original text into a more compact form and relationships, to facilitate further processing. Embeddings and transformers go together.
vector search	A technique combined with feature encoding to quickly retrieve embeddings in LLM summary tables, most similar to prompt-derived embeddings attached to a user query in GPT-like applications. Similar to multivariate "vlookup" in Excel. A popular metric to measure the proximity between two embeddings is the cosine similarity. To accelerate vector search, especially in real-time, you can cache popular embeddings and/or use approximate search such as ANN.

Appendix C

Introduction to Extreme LLM and Customized GPT

In this chapter, I discuss elements of architecture related to the large language models featured in section 7.2. The goal is to crawl some large websites, and create an application that returns specialized results to user queries or prompts. Named xLLM, it involves the following steps.

xLLM architecture: main steps

- Crawl specialized websites: Wolfram or a major category in Wikipedia. Focus on one top category.
- Reconstruct the taxonomy and create word associations and keyword embeddings.
- Parse user queries, retrieve the information, and return results, based on embeddings and other tables.
- Augment your data by adding other sources, such as parsed books.
- Add more top categories, each one with its separate crawling / sources, set of embeddings, and tables.
- Get the system to self-tune itself based on user feedback (favorite parameter values selected by the users). This leads to user-customized results.

The keyword used to describe this type of system is RAG: retrieve, augment, and generate. The xLLM project is broken down into major components, with separate pieces of code. In particular:

Python code used in the xLLM project

- To read the embedding and other tables, see `xllm5_util.py`, is in section C.2. Also on GitHub, here.
- The program `xllm5_short.py` reads the tables, process the user queries, and return the results. It is used in section in section 7.2.3, and available on GitHub, here.
- The program `xllm5.py` reads the crawled data and produces the input tables for `xllm5_short.py`. It is on GitHub, here. This is the main code, for developers, and discussed in section 7.2.2.
- Crawling is done with `crawl_directory.py`, available here and used in section 7.2.1.

Section C.1 is an introduction to the topic. Figures C.1 – C.2 show how xLLM compares to the Wolfram search box, even though both are based on the exact same content (the Wolfram website). Google is not better than Wolfram search, displaying rudimentary output only, even if you ask Google to search Wolfram exclusively. And OpenAI / GPT pictured in Figure C.3 is not better either.

C.1 Best practices

I explain how to improve the performance of GPT-like apps, both in terms of quality and speed (thus costs), using specialized tables, one set per top category. For instance, if you want to cover AI, machine learning, statistics and mathematics, some of your top categories include machine learning, probability and statistics, discrete mathematics, and so on. Some category overlap is expected. The components below dramatically improve the results.

Customized Embeddings. Break down your information database (what you crawled for instance) in 20 or so top categories. Have a separate embedding table for each category. In my case, it's not just embeddings but other tables such as local taxonomies. Have a separate one for each top category, allow for overlap. In order words, avoid silos: they will dilute the quality of your output. Twenty embedding tables, each with 50k keywords, is better than a single one with 1 million keywords.

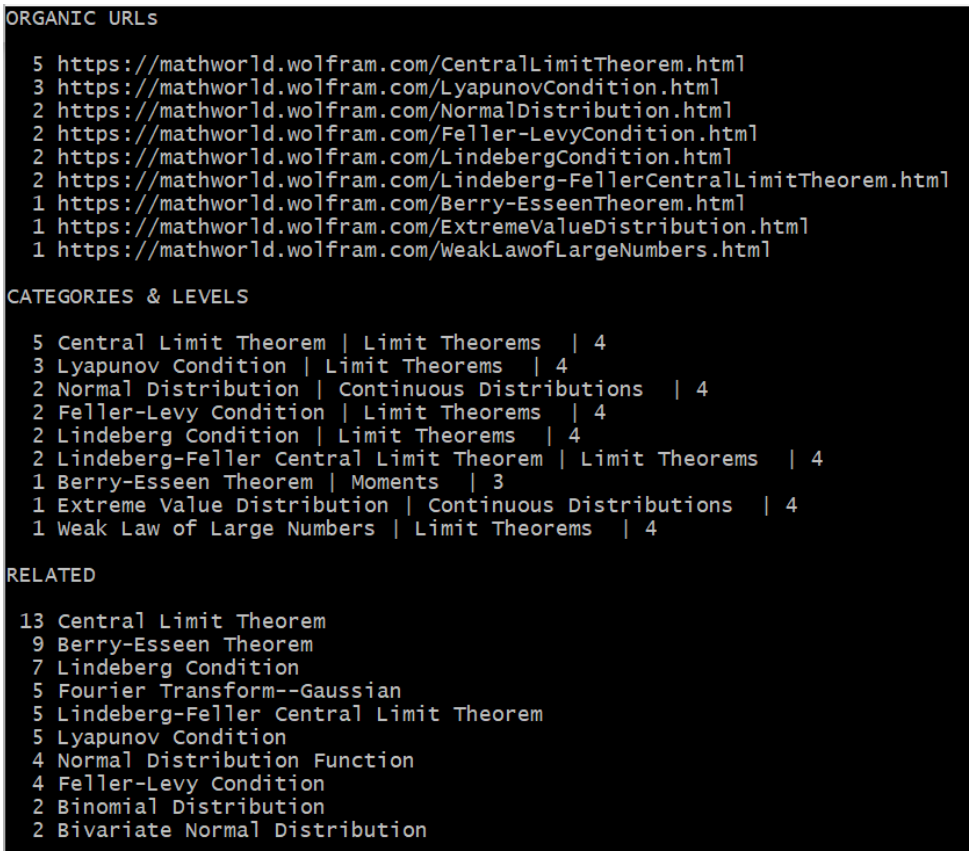


Figure C.1: Home-made xLLM: results for query "central limit theorem"

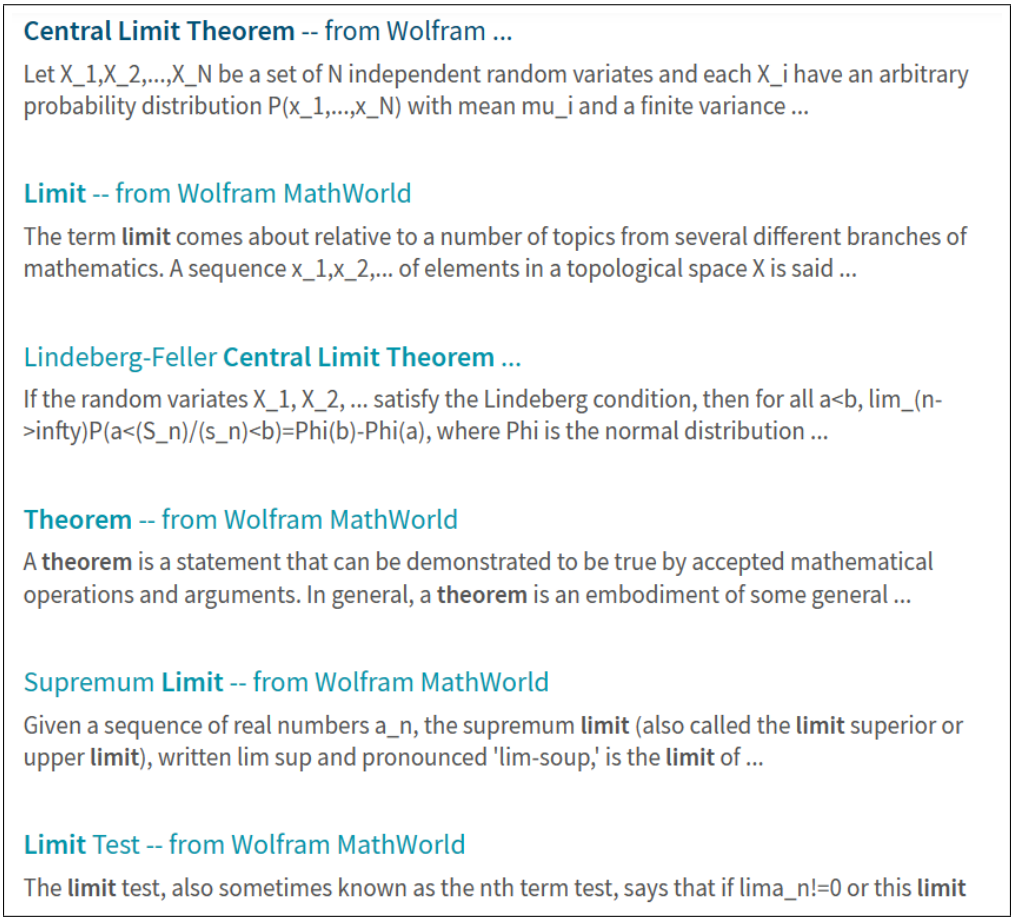


Figure C.2: Wolfram search box: results for query "central limit theorem"

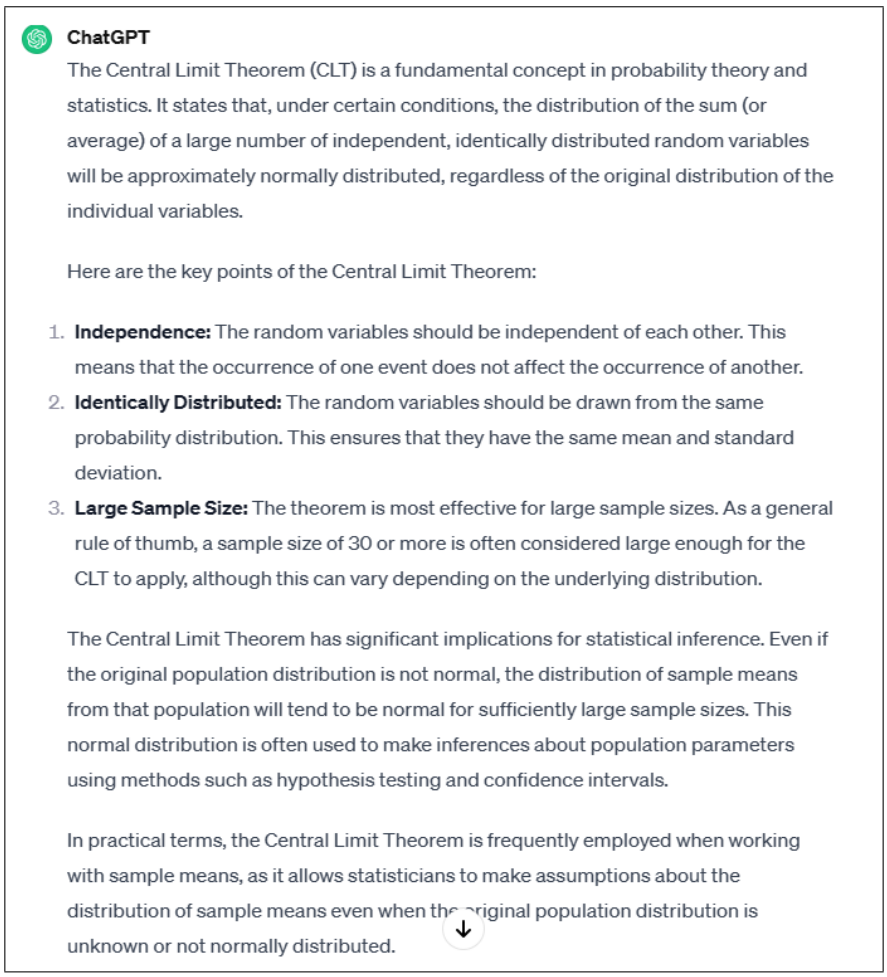


Figure C.3: OpenAI GPT: results for query "central limit theorem"

Variable-length embeddings. Abbreviated as VLE [5]. Many systems have a fixed but very large number of tokens per embedding. A fixed size may help with KNN vector search. But if you only keep the most relevant tokens for each embedding, the size of your tables will decrease significantly, boosting scalability. You can still achieve very efficient search: for instance, using the radix search algorithm described in section 2.3. Also, the quantities attached to each token – the relevancy metric in particular – do not need to be a frequency between 0 and 1, or a value between -1 to +1. I use pointwise mutual information instead. It is easier to interpret and compare, especially when you have multiple embedding tables.

High-quality taxonomy. Creating or relying on a good taxonomy helps you create better embeddings and better results. The words found in category and sub-category titles should be added to the embeddings, with a higher weight. Category titles are cleaner than raw text found on web pages. In case of parsing books, sections and subsection titles could carry a higher weight than raw text. When I crawled Wolfram, I retrieve the full taxonomy with 5000+ entries, all man-made by experts. It is one of the main contributors to the output quality.

Self-tuning. All GPT-like apps have several parameters, transparent to the user. For instance, the user can't choose which thresholds to apply to the embeddings. Allow the user to set all the parameters to her liking. This way, you can collect the most popular choices for your parameters, based on user feedback. Of course, this is done automatically on a permanent basis. In the end, you come up with optimum parameters. Trained in real-time by human beings! (this is what I meant by no algorithmic training; it is replaced by humans)

Even better: offer the user the ability to keep his favorite, self-customized set of parameter values. In the end, there is no one-size-fits-all evaluation metric. My xLLM is terrible for the novice looking for basic definitions, while GPT is a lot better. Conversely, for professional users looking for research results or serious references, the opposite is true.

Offer two prompt boxes. One for the standard query. And one where the user can suggest a category (or two) of his own. You can offer a selection of 10 or 20 pre-selected categories. But you might as well let the user enter a category himself, and process that information as a standard text string. Then match it to existing categories in your system. And then, process the user query or prompt, and match it to the right category to

return the most relevant results. Remember, each top category has its own embeddings! You want to use the correct embedding table(s) before returning results.

Multi-agent system. This is becoming a hot topic! Some say 2024 will be the year of the customized GPT. A multi-agent system is simply a top layer in your system, controlling all the top categories and embedding tables. In short, it glues the various customized embeddings together, allowing them to "communicate" with each other. In other words, it controls the cross-interactions. It is similar to multimodal (blending images and text) but for text only (blending multiple top categories). Related to mixture of experts [26].

Weighted sources. Your app will blend multiple sources together. Say one of your top categories is statistical science, and it has one specialized embedding table. The content may consist of crawled books and crawled sub-categories both in Wolfram and Wikipedia. Not all sources carry the same weight. You want a well-balanced embedding table. If (say) Wikipedia has more stuff, typically of lower quality, you want to weight that source appropriately.

Find structured data. The Internet and most websites are considerably more structured than most people think. You just have to find where the structure is hiding. In the case of Wolfram, it comes with a nice taxonomy, among other structures. Wikipedia has its own too. If you crawl books, look for indexes or glossaries and match index terms back to entries in the text. Indexes also have sub-entries and cross-links between entries, that you can leverage.

Even better: each entry (index term) is in some sections or subsections, in addition to being in sentences. Use the table of content and the sectioning as a pseudo-taxonomy. Associate section keywords with index terms found in it. And voila! Now you have strong keyword associations, in addition to the loose associations when focusing on raw (unstructured) text only.

C.2 Python utility for xLLM

This library contains the functions to read, create, or update embeddings and related tables needed to retrieve data when processing a user query. These tables are stored as dictionaries, also known as hash or key-value databases. The key is either a word consisting of tilde-separated tokens (up to 4 tokens), or a single token. The value is either a count, a list or a dictionary itself. The tables in question are on GitHub, here. The library also contains NLP and transform functions, for instance to turn a list into a dictionary. The Python code is also on GitHub, here. The library is named `xllm5_util.py`.

```
import numpy as np
import requests
from autocorrect import Speller
from pattern.text.en import singularize
spell = Speller(lang='en')

#--- [1] functions to read core tables (if not produced by you script)

pwd =
    "https://raw.githubusercontent.com/VincentGranville/Large-Language-Models/main/llm5/"

#--- [1.1] auxiliary functions

def text_to_hash(string, format = "int"):
   string = string.replace("'","").split(', ')
   hash = {}
   for word in string:
      word = word.replace("{","").replace("}","")
      if word != "":
         word = word.split(": ")
         value = word[1]
         if format == "int":
            value = int(value)
         elif format == "float":
            value = float(value)
         hash[word[0]] = value
   return(hash)


```

```
def text_to_list(string):
   if ', ' in string:
      string = string.replace("'","").split(', ')
   else:
      string = string.replace("'","").split(',')
   list = ()
   for word in string:
      word = word.replace("(","").replace(")","")
      if word != "":
         list = (*list, word)
   return(list)


def get_data(filename, path):
   if 'http' in path:
      response = requests.get(path + filename)
      data = (response.text).replace('\r','').split("\n")
   else:
      file = open(filename, "r")
      data = [line.rstrip() for line in file.readlines()]
      file.close()
   return(data)


#--- [1.2] functions to read the tables

def read_table(filename, type, format = "int", path = pwd):
   table = {}
   data = get_data(filename, path)
   for line in data:
      line = line.split('\t')
      if len(line) > 1:
        if type == "hash":
           table[line[0]] = text_to_hash(line[1], format)
        elif type == "list":
           table[line[0]] = text_to_list(line[1])
   return(table)


def read_arr_url(filename, path = pwd):
   arr_url = []
   data = get_data(filename, path)
   for line in data:
      line = line.split('\t')
      if len(line) > 1:
         arr_url.append(line[1])
   return(arr_url)


def read_stopwords(filename, path = pwd):
   data = get_data(filename, path)
   stopwords = text_to_list(data[0])
   return(stopwords)


def read_dictionary(filename, path = pwd):
   dictionary = {}
   data = get_data(filename, path)
   for line in data:
      line = line.split('\t')
      if len(line) > 1:
         dictionary[line[0]] = int(line[1])
   return(dictionary)


#--- [2] core function to create/update dictionary and satellite tables
```

```
95
96  def trim(word):
97     return(word.replace(".", "").replace(",",""))
98
99
100 def reject(word, stopwords):
101
102    # words can not contain any of these
103    # note: "&" and ";" used in utf processing, we keep them
104    flaglist = ( "=", "\"", "(", ")", "<", ">", "}", "|", "&quot;",
105              "{", "[", "]", "^", "/", "%", ":", "_",
106             )
107
108    # words can not start with any of these chars
109    bad_start = ("-",)
110
111    rejected = False
112    for string in flaglist:
113       if string in word:
114          rejected = True
115    if len(word) == 0:
116       rejected = True
117    elif word[0].isdigit() or word[0] in bad_start:
118       rejected = True
119    if word.lower() in stopwords:
120       rejected = True
121    return(rejected)
122
123
124 def create_hash(list):
125    hash = {}
126    for item in list:
127       if item in hash:
128          hash[item] += 1
129       elif item != "":
130          hash[item] = 1
131    return(hash)
132
133
134 def update_hash(word, hash_table, list):
135    if list != "":
136       hash = hash_table[word]
137       for item in list:
138          if item in hash:
139             hash[item] += 1
140          elif item != "":
141             hash[item] = 1
142       hash_table[word] = hash
143    return(hash_table)
144
145
146 def add_word(word, url_ID, category, dictionary, url_map, hash_category,
147           hash_related, hash_see, related, see, word_pairs, word_list):
148
149    # word is either 1-token, or multiple tokens separated by ~
150
151    urllist = (str(url_ID),)
152
153    if word in dictionary:
154
155       dictionary[word] += 1
156       url_map = update_hash(word, url_map, urllist)
157       hash_category = update_hash(word, hash_category, category)
158       hash_related = update_hash(word, hash_related, related)
159       hash_see = update_hash(word, hash_see, see)
160
```

```
161     else:
162
163        dictionary[word] = 1
164        urlist = (url_ID,)
165        url_map[word] = create_hash(urllist)
166        hash_category[word] = create_hash(category)
167        hash_related[word] = create_hash(related)
168        hash_see[word] = create_hash(see)
169
170     # generate association between 2 tokens of a 2-token word
171     # this is the starting point to create word embeddings
172
173     if word.count('~') == 1:
174
175        # word consists of 2 tokens word1 and word2
176        string = word.split('~')
177        word1 = string[0]
178        word2 = string[1]
179
180        pair = (word1, word2)
181        if pair in word_pairs:
182           word_pairs[pair] += 1
183        else:
184           word_pairs[pair] = 1
185        pair = (word2, word1)
186        if pair in word_pairs:
187           word_pairs[pair] += 1
188        else:
189           word_pairs[pair] = 1
190
191        if word1 in word_list:
192           word_list[word1] = (*word_list[word1], word2)
193        else:
194           word_list[word1] = (word2,)
195        if word2 in word_list:
196           word_list[word2] = (*word_list[word2], word1)
197        else:
198           word_list[word2] = (word1,)
199
200     return()
201
202
203  def stem_data(data, stopwords, dictionary, mode = 'Internal'):
204
205     # input: raw page (array containing the 1-token words)
206     # output: words found both in singular and plural: we only keep the former
207     # if mode = 'Singularize', use singularize library
208     # if mode = 'Internal', use home-made (better)
209
210     stem_table = {}
211     temp_dictionary = {}
212
213     for word in data:
214        if not reject(word, stopwords):
215           trim_word = trim(word)
216           temp_dictionary[trim_word] = 1
217
218     for word in temp_dictionary:
219        if mode == 'Internal':
220           n = len(word)
221           if n > 2 and "~" not in word and \
222                word[0:n-1] in dictionary and word[n-1] == "s":
223              stem_table[word] = word[0:n-1]
224           else:
225              stem_table[word] = word
226        else:
```

```
227           # the instruction below changes 'hypothesis' to 'hypothesi'
228           word = singularize(word)
229
230           # the instruction below changes 'hypothesi' back to 'hypothesis'
231           # however it changes 'feller' to 'seller'
232           # solution: create 'do not singularize' and 'do not autocorrect' lists
233           stem_table[word] = spell(word)
234
235     return(stem_table)
236
237
238 def update_core_tables(data, dictionary, url_map, arr_url, hash_category, hash_related,
239                        hash_see, stem_table, category, url, url_ID, stopwords, related,
240                        see, word_pairs, word_list):
241
242     # data is a word array built on crawled data (one webpage, the url)
243     # url_ID is incremented at each call of update_core_tables(xx)
244     # I/O: dictionary, url_map, word_list, word_pairs,
245     #      hash_see, hash_related, hash_category
246     # these tables are updated when calling add_word(xxx)
247
248     arr_word = [] # list of words (1 to 4 tokens) found on this page, local array
249     k = 0
250
251     for word in data:
252
253        if not reject(word, stopwords):
254
255           raw_word = word
256           trim_word = trim(word)
257           trim_word = stem_table[trim_word]
258
259           if not reject(trim_word, stopwords):
260
261              arr_word.append(trim_word)
262              add_word(trim_word, url_ID, category, dictionary, url_map, hash_category,
263                     hash_related, hash_see, related, see, word_pairs, word_list)
264
265              if k > 0 and trim_word == raw_word:
266                 # 2-token word
267                 if arr_word[k-1] not in trim_word:
268                    word = arr_word[k-1] + "~" + trim_word
269                    add_word(word, url_ID, category, dictionary, url_map, hash_category,
270                           hash_related, hash_see, related, see, word_pairs, word_list)
271
272              if k > 1 and trim_word == raw_word:
273                 # 3-token word
274                 if arr_word[k-2] not in word:
275                    word = arr_word[k-2] + "~" + word
276                    add_word(word, url_ID, category, dictionary, url_map, hash_category,
277                           hash_related, hash_see, related, see, word_pairs, word_list)
278
279              if k > 2 and trim_word == raw_word:
280                 # 4-token word
281                 if arr_word[k-3] not in word:
282                    word = arr_word[k-3] + "~" + word
283                    add_word(word, url_ID, category, dictionary, url_map, hash_category,
284                           hash_related, hash_see, related, see, word_pairs, word_list)
285              k += 1
286
287     arr_url.append(url)
288     url_ID += 1
289     return(url_ID)
290
291
292 #--- [3] simple text processsing
```

```
293
294 def collapse_list(list):
295    # group by item and get count for each item
296    clist = {}
297    for item in list:
298       if item in clist:
299          clist[item] += 1
300       elif item != '':
301          clist[item] = 1
302    return(clist)
303
304
305 #--- [4] create embeddings and ngrams tables, once all sources are parsed
306
307 def create_pmi_table(word_pairs, dictionary):
308
309    pmi_table = {} # pointwise mutual information
310    exponent = 1.0
311
312    for pair in word_pairs:
313
314       word1 = pair[0]
315       word2 = pair[1]
316       f1 = dictionary[word1] / len(dictionary)
317       f2 = dictionary[word2] / len(dictionary)
318       f12 = word_pairs[pair] / len(word_pairs)
319       pmi = np.log2(f12 / (f1 * f2)**exponent)
320       word2_weight = word_pairs[pair] / dictionary[word1]
321       pmi_table[pair] = pmi
322
323    return(pmi_table)
324
325
326 def create_embeddings(word_list, pmi_table):
327
328    embeddings = {}
329
330    for word in word_list:
331
332       list = word_list[word]
333       clist = collapse_list(list)
334       embedding_list = {}
335
336       for word2 in clist:
337          count = clist[word2]
338          pair = (word, word2)
339
340          if pair in pmi_table:
341
342             pmi = pmi_table[pair]
343             embedding_list[word2] = pmi
344
345       embeddings[word] = embedding_list
346
347    return(embeddings)
348
349
350 def build_ngrams(dictionary):
351
352    ngrams_table = {}
353    for word in dictionary:
354       tokens = word.split("~")
355       tokens.sort()
356       sorted_word = tokens[0]
357       for k in range(1, len(tokens)):
358          sorted_word += "~" + tokens[k]
```

```
        if sorted_word in ngrams_table:
            ngrams_table[sorted_word] = (*ngrams_table[sorted_word], word,)
        else:
            ngrams_table[sorted_word] = (word,)
    return(ngrams_table)


def compress_ngrams(dictionary, ngrams_table):
    # for each sorted_word, keep most popular ngram only

    compressed_ngrams_table = {}
    for sorted_word in ngrams_table:
        ngrams = ngrams_table[sorted_word]
        max_count = 0
        for ngram in ngrams:
            if dictionary[ngram] > max_count:
                max_count = dictionary[ngram]
                best_ngram = ngram
        compressed_ngrams_table[sorted_word] = (best_ngram, )
    return(compressed_ngrams_table)
```

C.3 Comparing xLLM with standard LLMs

The xLLM architecture is so different in so many foundational respects that it is almost a different animal. Here, I highlight some important differences. Figure C.4 shows that embeddings are just one of the many summary tables needed to generate answers to user prompts. Each table is unique to each sub-LLM, though redundancy is allowed. Even for embeddings, the storage is non-standard, based on variable-length and key-values or graph databases, rather than vector databases. Not the least, xLLM is energy friendly, as it does not need GPU, neural networks, training (self-tuned instead), or trillions of weights.

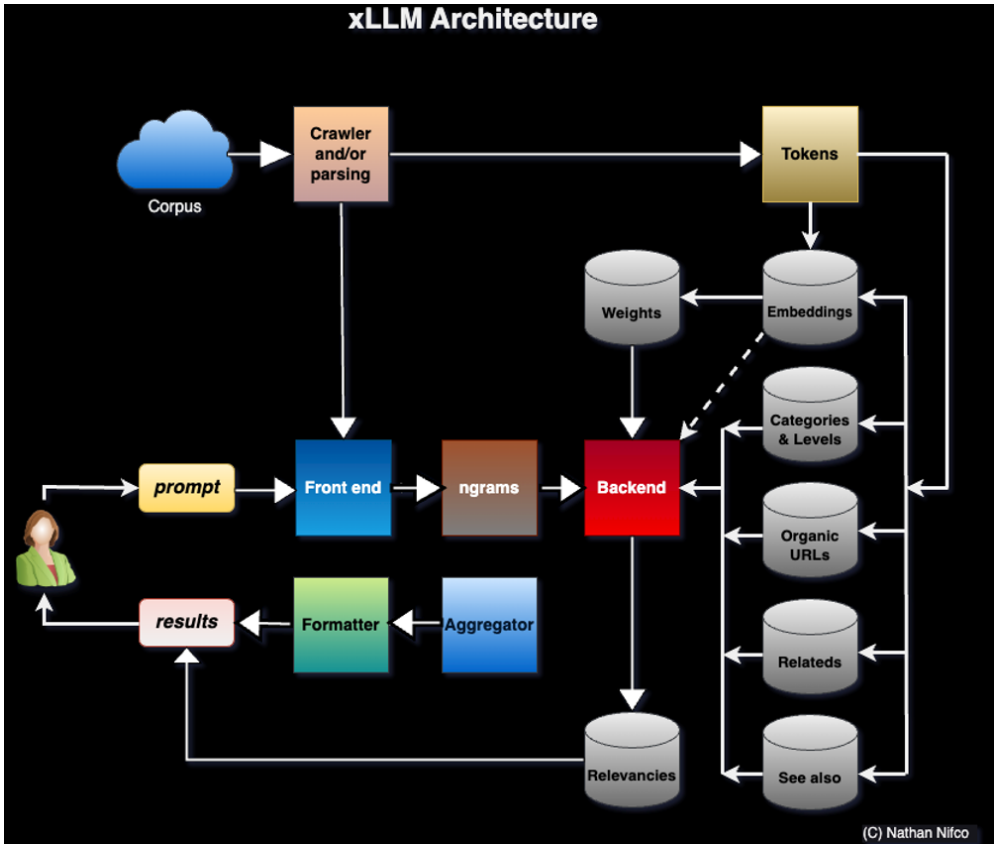


Figure C.4: xLLM: Satellite tables and variable-length embeddings (one set per sub-LLM)

xLLM	Standard LLM
Multiple specialized LLMs: one per category to increase accuracy and relevance as well as latency while mitigating hallucination and therefore liability	One monolithic LLM, all users get same output; prompt engineering (front-end) needed to fix the resulting back-end problems
Raw input + reconstructed (or discoverable) taxonomy to potentially provide a knowledge graph; UI navigation feature to further increase accuracy and relevancy	Based on raw input only
Embeddings is just one of the main summary tables to increase accuracy and relevancy while reducing latency and cost	Embeddings is the core summary table; no effort to discover or leverage internal structure of the corpus (input sources)
User enters prompt and selected categories to increase accuracy and relevancy	User enters prompt only; results in slower response (latency) as data retrieval is not targeted to specific tables
Internal relevancy score attached to each item displayed to the user	No relevancy score provided to the user
Output broken down into sections, links, categories, related concepts, internal versus external sources to reduce liability, increase explainability and observability while mitigating hallucinations	In many instances, links not provided (input sources kept secret resulting in liability issues)
User can customize hyperparameters and relevancy metrics to provide customized calibration	Hyperparameters not accessible to user resulting in training issues
Fast and variable-length embeddings to reduce latency, mitigate hallucination and reduce liability	Massive tables, trillions of weights, fixed-length embeddings slowing down vector search, delicate and time consuming training with deep neural networks
Inexpensive, lightweight, energy-efficient, ideal for implementation to increase user customization and diverse user scalability	Needs lots of GPU and cloud time, thus expensive especially if hosted externally; not eco-friendly
Self-tuned with reinforcement learning based on user preferences	There is no universal evaluation metric because satisfaction depends on type of user
Explainable AI (meaningful parameters)	Neural networks are blockboxes, notorious for lack of explainability
Consise output targeted to professional users, faithful representation of input text elements	Excessive stemming and glitches in standard libraries (autocorrect, singularize, stopwords) leads to hallucinations, prompt engineering needed tofix issues
Ability to orchestrate outcomes employing a multi-agent architecture	One LLM: no top layer to manage multiple LLMs
Ability to process outcomes from multiple prompts in parallel or sequentially	Bulk processing only available in paid version via API
Secure, reduced risk of hallucination or data leakage, especially in local implementations	Hallucinations may lead to errors and liability; confidential input data in prompts may not be protected

C.4 Comparing xLLM5 with xLLM6

Unless otherwise specified, all LLM projects in this textbook – 7.2.2 and 7.2.3 in particular – are based on xLLM5. The code and datasets are also on GitHub, here. A new version, namely xLLM6, is now available and still being further developed. I did not include the code in this book, but you can find it on GitHub, here. The overall architecture is similar, with the following main differences:

- The introduction of multi-token words in embedding tables. The new embeddings, called x-embeddings, are stored in a key-value table named `embeddings2` in the Python code. The key is a multi-token word, and the value is a key-value table itself. The old embeddings are still stored in the `embeddings` table with the same data structure; they may be discarded moving forward. See the x-embeddings on GitHub, here, compared to the old embeddings, here.

- A new counter named `paragraph` in the function `update_core_tables2` in `xllm6_util.py`. This new function replaces the old version `update_core_tables` in xLLM5. Most of the main upgrades are implemented there, including a call to new functions `update_word2_hash` and `update_word2_pairs`.

  Words need to be in a same paragraph to appear jointly in x-embeddings. Long-range dependencies are still included, via categories, breadcrumbs, metadata, and other navigation features. The `word2_hash` table, a precursor to the x-embeddings, replaces the old `world_list` table: not only the format is streamlined (key-value hash rather than list), but it now handles multi-token words. Due to the potential explosion in size, the shorter version `compressed_word2_hash` is the central table. The `word2_pairs` table contains similar content, but organized in a different way to facilitate taxonomy creation. All these tables are local to a specific top category, in other words, to a sub-LLM in a multi-LLM architecture.

  See on GitHub the `compressed_word2_hash`, here, and the `word2_pairs`, here.

- I also added two sections to the output results delivered to user prompts: "x-Embeddings", and "Linked Words". The former is based on pmi (pointwise mutual information) just like in the old "Embeddings" section, and the latter on raw word counts.

- The stopwords list, again specific to each sub-LLM, can now handle multiple-token words, with tokens separated by the tilde character "~" within a same word. These lists are built by looking at meaningless or useless words with highest counts in the `dictionary` table. They significantly contribute to improving the quality of embeddings, and need to be built from scratch. Do not use stopwords from Python libraries unless you blend them with customized do-not-exclude lists.

C.4.1 Conclusion

The x-embeddings, made up of multi-token words, is a significant improvement over standard single-token embeddings. They can dramatically increase the number of weights, from 5000 to over 1 million for a specialized sub-LLM corresponding to a top category. Applied to the whole human knowledge, it would result in a few trillion weights.

```
ORGANIC URLs

 18 https://mathworld.wolfram.com/Hypothesis.html
 14 https://mathworld.wolfram.com/HypothesisTesting.html
  9 https://mathworld.wolfram.com/NullHypothesis.html
  8 https://mathworld.wolfram.com/AlternativeHypothesis.html
  5 https://mathworld.wolfram.com/StatisticalHypothesis.html
  4 https://mathworld.wolfram.com/NestedHypothesis.html
  4 https://mathworld.wolfram.com/StatisticalTest.html
  3 https://mathworld.wolfram.com/TypeIError.html
  3 https://mathworld.wolfram.com/TypeIIError.html
  1 https://mathworld.wolfram.com/FisherSignTest.html

CATEGORIES & LEVELS

 18 Hypothesis | Statistical Tests  | 3
 14 Hypothesis Testing | Statistical Tests  | 3
  9 Null Hypothesis | Statistical Tests  | 3
  8 Alternative Hypothesis | Statistical Tests  | 3
  5 Statistical Hypothesis | Statistical Tests  | 3
  4 Nested Hypothesis | Statistical Tests  | 3
  4 Statistical Test | Statistical Tests  | 3
  3 Type I Error | Statistical Tests  | 3
  3 Type II Error | Statistical Tests  | 3
  1 Fisher Sign Test | Statistical Tests  | 3

RELATED

 46 Null Hypothesis
 41 Hypothesis Testing
 41 Alternative Hypothesis
 31 Hypothesis
 21 Statistical Test
 21 Type I Error
 21 Type II Error
 20 Fisher Sign Test
 19 Paired
 19 Wilcoxon Signed Rank Test

ALSO SEE

  5 Alternative Hypothesis
  5 Hypothesis
  5 Hypothesis Testing
  5 Null Hypothesis

LINKED WORDS

 29 test
 24 null
 20 statistical
 15 testing
 10 alternative
  6 hypothesis~statistical
  6 type
  6 error
  5 statistic
  4 fisher

EMBEDDINGS

   28.47 statistical
   27.50 alternative
   26.15 null
   19.16 testing
   18.25 rejection
   13.60 effect
    9.12 truth
    9.12 evidence
    9.12 determines
    8.58 type

X-EMBEDDINGS

   68.69 null
   36.70 testing
   19.36 alternative
   17.17 hypothesis~statistical
    9.25 test
    8.59 hypothesis~null
    8.22 nested
    5.72 paired~statistical
    5.72 alternative~hypothesis
    5.72 alternative~hypothesis~statistical
```

Figure C.5: xLLM6, top results for "hypothesis"

However, much of it is garbage and never fetched during similarity search to match front-end user prompts with back-end x-embeddings built on crawls. In short, there is a waste of space, computer time, and bandwidth.

In addition, it can result in hallucinations in standard LLMs (OpenAI and the likes). Compressed tables extract the essence and are much smaller. Compression is achieved via stopwords and by not linking words together unless they are found in a same, short paragraph. Of course, besides x-embeddings, xLLM uses many other small tables that take care of long-range word dependencies, such as taxonomies. Much of the compression results from using a separate LLM for each top category, allowing the user to select categories.

While Figure C.5 mostly shows 1-token words in the x-embeddings, you can change the parameters to favor multi-token words. For instance, ignoring single-token words, doubling the count (weight) attached to multi-token words, or doubling the count when a word is found in the title, taxonomy, or metadata as opposed to the main text. The latter approach also favors natural *n*-grams: those with highest occurrences among all permutations of a same multi-token word. For instance, "statistical hypothesis" favored over "hypothesis statistical". To further boost natural *n*-grams, consider the following two options:

- Return to the front-end user prompt the most common *n*-gram permutation attached to any multi-token word, based on counts stored in the `dictionary` table.
- Or normalize *n*-grams to their most common permutation, in the various back-end tables. To perform this task, the hash table `compressed_ngrams_table` is very useful. Its key is a sorted *n*-gram, while the corresponding value is a vector featuring the permutations of the *n*-gram in question, each with its own count. Normalized *n*-grams further reduces the size of the largest summary tables such as `word2_pairs`.

Finally, you can use the `dictionary` and `word2_pairs` tables to create a taxonomy, if you don't already have one. Top multi-token dictionary entries, say $\texttt{word}_1$ and $\texttt{word}_2$, where $\texttt{(word}_1\texttt{, word}_2\texttt{)}$ is not a key of `word2_pairs`, correspond to different categories. To the contrary, if $\texttt{word2\_pairs[(word}_1\texttt{, word}_2\texttt{)]}$ is a large integer (counting the simultaneous occurrences of both words across many paragraphs), it means that $\texttt{word}_1$ and $\texttt{word}_2$ belong to the same category. Based on this, you can reconstruct the underlying taxonomy. The first step is to ignore top words not representing any category. Also, words found in titles should be given extra weight.

It is interesting to note that despite not using a synonyms or abbreviations dictionary yet, some of the issues in xLLM5, such as "ANOVA" and "analysis of variance", or "Bayes" and "Bayesian" not returning similar results, are not present in xLLM6. Other improvements not yet implemented consist of handling capital letters separately, and treating words such as "Saint Petersburg" as one token. To conclude, Figure C.5 displays only the top 10 entries in each section, ordered by relevancy. Some entries not shown because not in the top 10, are quite relevant. Fine-tuning the hyperparameters may result in a different ordering, better or worse depending on the user.

Bibliography

[1] Adel Alamadhi, Michel Planat, and Patrick Solé. Chebyshev's bias and generalized Riemann hypothesis. *Preprint*, pages 1–9, 2011. arXiv:1112.2398 [Link]. 67

[2] K. Binswanger and P. Embrechts. Longest runs in coin tossing. *Insurance: Mathematics and Economics*, 15:139–149, 1994. [Link]. 60

[3] Iulia Brezeanu. How to cut RAG costs by 80% using prompt compression. *Blog post*, 2024. TowardsData-Science [Link]. 139

[4] Ramiro Camino, Christian Hammerschmidt, and Radu State. Generating multi-categorical samples with generative adversarial networks. *Preprint*, pages 1–7, 2018. arXiv:1807.01202 [Link]. 96

[5] Johnathan Chiu, Andi Gu, and Matt Zhou. Variable length embeddings. *Preprint*, pages 1–12, 2023. arXiv:2305.09967 [Link]. 138, 177

[6] Fida Dankar et al. A multi-dimensional evaluation of synthetic data generators. *IEEE Access*, pages 11147–11158, 2022. [Link]. 95

[7] Antónia Földes. The limit distribution of the length of the longest head-run. *Periodica Mathematica Hungarica*, 10:301–310, 1979. [Link]. 61

[8] Fabian Gloeckle et al. Better & faster large language models via multi-token prediction. *Preprint*, pages 1–29, 2024. arXiv:2404.19737 [Link]. 155

[9] Louis Gordon, Mark F. Schilling, and Michael S. Waterman. An extreme value theory for long head runs. *Probability Theory and Related Fields*, 72:279–287, 1986. [Link]. 60

[10] Vincent Granville. *Statistics: New Foundations, Toolbox, and Machine Learning Recipes*. Data Science Central, 2019. 25

[11] Vincent Granville. *Synthetic Data and Generative AI*. MLTechniques.com, 2022. [Link]. 102

[12] Vincent Granville. Feature clustering: A simple solution to many machine learning problems. *Preprint*, pages 1–6, 2023. MLTechniques.com [Link]. 84

[13] Vincent Granville. Generative AI: Synthetic data vendor comparison and benchmarking best practices. *Preprint*, pages 1–13, 2023. MLTechniques.com [Link]. 77

[14] Vincent Granville. Generative AI technology break-through: Spectacular performance of new synthesizer. *Preprint*, pages 1–16, 2023. MLTechniques.com [Link]. 12, 15, 170

[15] Vincent Granville. *Gentle Introduction To Chaotic Dynamical Systems*. MLTechniques.com, 2023. [Link]. 24, 62, 66

[16] Vincent Granville. How to fix a failing generative adversarial network. *Preprint*, pages 1–10, 2023. ML-Techniques.com [Link]. 14

[17] Vincent Granville. Massively speed-up your learning algorithm, with stochastic thinning. *Preprint*, pages 1–13, 2023. MLTechniques.com [Link]. 13, 84

[18] Vincent Granville. Smart grid search for faster hyperparameter tuning. *Preprint*, pages 1–8, 2023. ML-Techniques.com [Link]. 13, 82, 84

[19] Vincent Granville. *Statistical Optimization for AI and Machine Learning*. MLTechniques.com, 2024. [Link]. 36, 37, 137, 139, 174

[20] Vincent Granville. *Synthetic Data and Generative AI*. Elsevier, 2024. [Link]. 40, 45, 46, 47, 49, 50, 56, 57, 59, 65, 76, 77, 81, 82, 84, 85, 87, 95, 96, 137

[21] Elisabeth Griesbauer. *Vine Copula Based Synthetic Data Generation for Classification*. 2022. Master Thesis, Technical University of Munich [Link]. 84

[22] Emil Grosswald. Oscillation theorems of arithmetical functions. *Transactions of the American Mathematical Society*, 126:1–28, 1967. [Link]. 67

[23] Adam J. Harper. Moments of random multiplicative functions, II: High moments. *Algebra and Number Theory*, 13(10):2277–2321, 2019. [Link]. 67

[24] Adam J. Harper. Moments of random multiplicative functions, I: Low moments, better than squareroot cancellation, and critical multiplicative chaos. *Forum of Mathematics, Pi*, 8:1–95, 2020. [Link]. 67

[25] Adam J. Harper. Almost sure large fluctuations of random multiplicative functions. *Preprint*, pages 1–38, 2021. arXiv [Link]. 67

[26] Albert Jiang et al. Mixtral of experts. *Preprint*, pages 1–13, 2024. arXiv:2401.04088 [Link]. 178

[27] Zsolt Karacsony and Jozsefne Libor. Longest runs in coin tossing. teaching recursive formulae, asymptotic theorems and computer simulations. *Teaching Mathematics and Computer Science*, 9:261–274, 2011. [Link]. 61

[28] Andrei Lopatenko. Evaluating LLMs and LLM systems: Pragmatic approach. *Blog post*, 2024. [Link]. 129, 131

[29] Adnan Saher Mohammed, Şahin Emrah Amrahov, and Fatih V. Çelebi. Interpolated binary search: An efficient hybrid search algorithm on ordered datasets. *Engineering Science and Technology*, 24:1072–1079, 2021. [Link]. 29

[30] Tamas Mori. The a.s. limit distribution of the longest head run. *Canadian Journal of Mathematics*, 45:1245–1262, 1993. [Link]. 61

[31] Hussein Mozannar et al. The RealHumanEval: Evaluating Large Language Models' abilities to support programmers. *Preprint*, pages 1–34, 2024. arXiv:2404.02806 [Link]. 131

[32] Michel Planat and Patrick Solé. Efficient prime counting and the Chebyshev primes. *Preprint*, pages 1–15, 2011. arXiv:1109.6489 [Link]. 67

[33] M.S. Schmookler and K.J. Nowka. Bounds on runs of zeros and ones for algebraic functions. *Proceedings 15th IEEE Symposium on Computer Arithmetic*, pages 7–12, 2001. ARITH-15 [Link]. 60

[34] Sergey Shchegrikovich. How do you create your own LLM and win The Open LLM Leaderboard with one Yaml file? *Blog post*, 2024. [Link]. 129

[35] Mark Shilling. The longest run of heads. *The College Mathematics Journal*, 21:196–207, 2018. [Link]. 60

[36] Chang Su, Linglin Wei, and Xianzhong Xie. Churn prediction in telecommunications industry based on conditional Wasserstein GAN. *IEEE International Conference on High Performance Computing, Data, and Analytics*, pages 186–191, 2022. IEEE HiPC 2022 [Link]. 95

[37] Terence Tao. Biases between consecutive primes. *Tao's blog*, 2016. [Link]. 67

[38] Eyal Trabelsi. Comprehensive guide to approximate nearest neighbors algorithms. *Blog post*, 2020. TowardsDataScience [Link]. 140

[39] Ruonan Yu, Songhua Liu, and Xinchao Wang. Dataset distillation: A comprehensive review. *Preprint*, pages 1–23, 2022. Submitted to IEEE PAMI [Link]. 82

Index