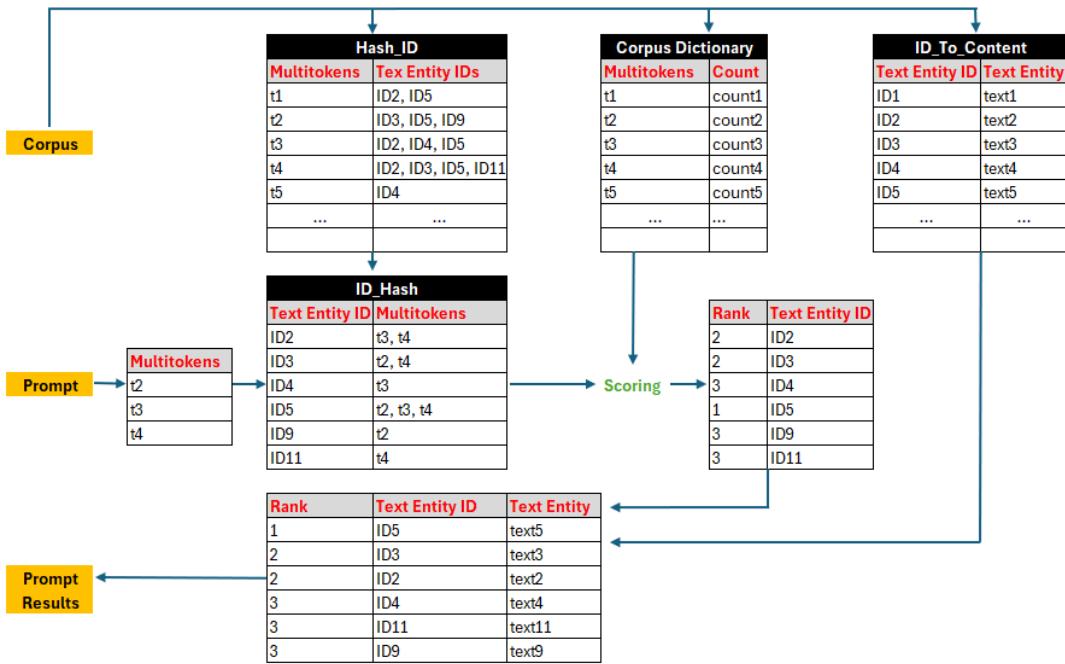
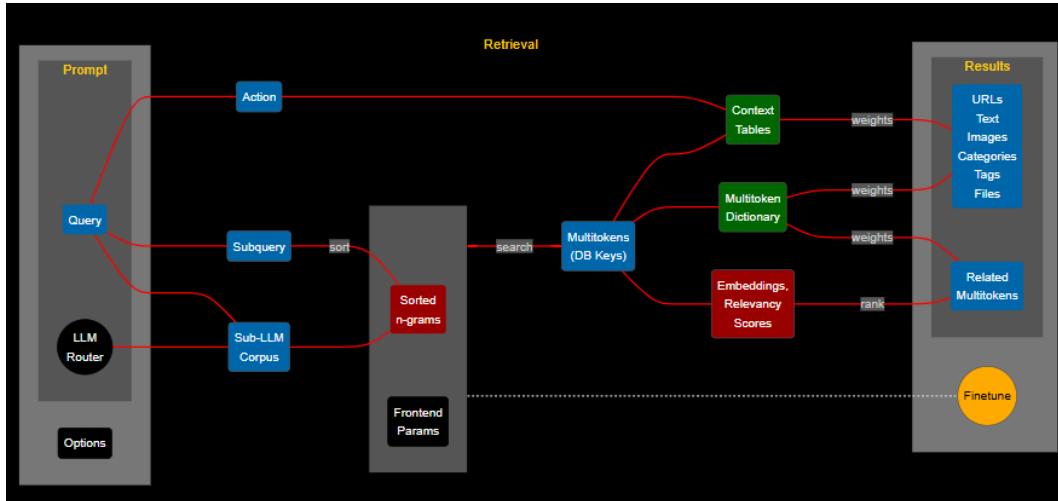


Building Disruptive AI & LLM Technology From Scratch



Contents

I Hallucination-Free LLM with Real-Time Fine-Tuning	7
1 Enterprise xLLM for Search & Retrieval	8
1.1 Efficient database architecture: nested hashes	8
1.2 From frontend prompts to backend tables	9
1.3 What is not covered here	10
2 Parameters, features, and fine-tuning	12
2.1 Backend parameters	12
2.2 Frontend parameters	13
2.3 Agents	13
2.4 Reproducibility	14
2.5 Singularization, stemming, auto-correct	15
2.6 Augmentation, distillation, and frontend tables	15
2.7 In-memory database, latency, and scalability	16
3 Case study	18
3.1 Real-time fine-tuning, prompts, and command menu	19
3.2 Sample session	20
3.3 Web API for enterprise xLLM	30
3.3.1 Left panel: command menu and prompt box	30
3.3.2 Right panel: prompt results	30
3.3.3 Next steps	32
3.4 Conclusions and references	33
4 Appendix	34
4.1 Python code	34
4.2 Thirty features to boost LLM performance	46
4.2.1 Fast search and caching	46
4.2.2 Leveraging sparse databases	47
4.2.3 Contextual tokens	47
4.2.4 Adaptive loss function	47
4.2.5 Contextual tables	47
4.2.6 Smart crawling	47
4.2.7 LLM router, sub-LLMs, and distributed architecture	47
4.2.8 From one trillion parameters down to two	48
4.2.9 Agentic LLMs	48
4.2.10 Data augmentation via dictionaries	48
4.2.11 Distillation done smartly	48
4.2.12 Reproducibility	49
4.2.13 Explainable AI	49
4.2.14 No training, in-memory LLM	49
4.2.15 No neural network	49
4.2.16 Show URLs and references	50
4.2.17 Taxonomy-based evaluation	50
4.2.18 Augmentation via prompt data	50
4.2.19 Variable-length embeddings, indexing, and database optimization	50
4.2.20 Favor backend over frontend engineering	50
4.2.21 Use NLP and Python libraries with caution	51
4.2.22 Self-tuning and customization	51

4.2.23 Local, global parameters, and debugging	51
4.2.24 Displaying relevancy scores, and customizing scores	52
4.2.25 Intuitive hyperparameters	52
4.2.26 Sorted n -grams and token order preservation	52
4.2.27 Blending standard tokens with tokens from the knowledge graph	52
4.2.28 Boosted weights for knowledge-graph and business tokens	53
4.2.29 Versatile command prompt	53
4.2.30 Boost long multitokens and rare single tokens	53
4.2.31 Disambiguation	53
4.2.32 Allow for customized search	53
4.3 LLM glossary	54
II Outperforming Neural Nets and Classic AI	57
5 Building and evaluating a taxonomy-enriched LLM	58
5.1 Project and solution	60
5.2 Python code	61
6 LLM for Clustering and Predictions	67
6.1 Project and solution	68
6.2 Visualizations and discussion	71
6.3 Python code	71
7 Fast, High-Quality NoGAN for Data Synthetization	79
7.1 Project description	79
7.2 Solution	81
7.3 Python implementation	82
8 Hierarchical Bayesian NoGAN for Data Synthetization	88
8.1 Methodology	88
8.1.1 Base algorithm	89
8.1.2 Loss function	89
8.1.3 Hyperparameters and convergence	90
8.1.4 Acknowledgments	91
8.2 Case studies	91
8.2.1 Synthesizing the student dataset	92
8.2.2 Synthesizing the Telecom dataset	94
8.2.3 Other case studies	95
8.2.4 Auto-tuning the hyperparameters	96
8.2.5 Evaluation with multivariate ECDF and KS distance	97
8.3 Conclusion	98
8.4 Python implementation	99
9 Boosting Model Evaluation with Smart Adaptive Loss Functions	108
9.1 Project and solution	109
9.2 Python code	113
10 Deep Retrieval and Multi-Index Chunking for PDFs	118
10.1 PDF contextual parsing and chunking: Nvidia case study	118
10.1.1 Multi-index, bullet lists, images, tables	120
10.1.2 Diagrams, context, agents, auto-tagging	120
10.1.3 Python code	121
10.2 Challenges with standard LLM implementations	125
10.2.1 Training LLMs for the wrong tasks	125
10.2.2 Analogy to clustering algorithms	125
10.2.3 Challenges in evaluating LLMs	126
10.2.4 Benefits of untrained LLMs	127
10.3 Overview of xLLM main differentiators	127

III Innovations in Statistical AI	129
11 Building a Ranking System to Enhance Prompt Results	130
11.1 Relevancy scores and rankings	131
11.2 Case study	132
11.2.1 xLLM for auto-indexing, cataloging and glossary generation	132
11.2.2 xLLM for scientific research	133
11.3 Python code	134
11.4 Appendix: smart ranking in a nutshell	140
12 Probabilistic Search: Alternative to Vector Search	142
12.1 Motivation and architecture	143
12.2 Applications	144
12.2.1 Embeddings and large language models	144
12.2.2 Generating and evaluating synthetic data	145
12.2.3 Clustering, dataset comparisons, outlier detection	146
12.3 Project and solution	147
12.4 Python code	148
13 Strong Random Generators for Reproducible AI	152
13.1 Strong Randomness and reproducibility: two key components	152
13.2 Computing the digits of special math constants	153
13.2.1 P-adic valuations	153
13.2.2 Digit blocks, speed of convergence	154
13.2.3 A plethora of interesting pseudo-random sequences	154
13.3 Testing random number generators	155
13.3.1 Theoretical properties of the digits of $\sqrt{2}$	155
13.3.2 Fast recursion and congruential equidistribution	156
13.3.3 Exponential system: predicting the next block	158
13.4 Python code	160
13.4.1 Fast recursion	160
13.4.2 Main code	161
14 Sampling Outside the Observation Range	166
14.1 Quantile convolution	166
14.2 Truncated Gaussian mixtures and bias detection	168
14.3 Case studies	168
14.3.1 Conclusion	169
14.4 Python code	170
15 Additional Resources	173
15.1 Accelerating convergence of parameter estimates	173
15.1.1 First case study	173
15.1.2 Second case study	173
15.2 Generic, all-in-one curve fitting, regression and clustering	175
15.2.1 Solution, R-squared and backward compatibility	176
15.2.2 Model upgrades	177
15.2.3 Case studies	177
15.2.4 Python code	181
15.3 A simple geospatial interpolation method	182
15.3.1 Problem in two dimensions	182
15.3.2 Spatial interpolation of the temperature dataset	184
15.3.3 Python code	185
15.4 Math-free, parameter-free gradient descent	185
15.4.1 Introduction	186
15.4.2 Implementation details	186
15.4.3 General comments about the methodology and parameters	189
15.4.4 Mathematical version of gradient descent and orthogonal trajectories	190
15.4.5 Python code	191
16 Trading the S&P 500 Index	192
16.1 Non-standard strategies	193

16.2 Selecting a strategy based on its features	194
16.3 Python code and dataset: 40 years' worth of historical data	195
Bibliography	199
Index	201

Introduction

This book features new advances in game-changing AI and LLM technologies built by [GenAItechLab.com](#). Written in simple English, it is best suited for engineers, developers, data scientists, analysts, consultants and anyone with an analytic background interested in starting a career in AI. The emphasis is on scalable enterprise solutions, easy to implement, yet outperforming vendors both in terms of speed and quality, by several orders of magnitude.

Each topic comes with GitHub links, full Python code, datasets, illustrations, and real life case studies, including from Fortune 100 company. Some of the material is presented as enterprise projects with solution, to help you build robust applications and boost your career. You don't need expensive GPU and cloud bandwidth to implement them: a standard laptop works.

Part I: Hallucination-Free LLM with Real-Time Fine-Tuning focuses on high performance in-memory agentic multi-LLMs for professional users and enterprise, with real-time fine-tuning, self-tuning, no weight, no training, no latency, no hallucinations, no GPU. Made from scratch, leading to replicable results, leveraging explainable AI, adopted by Fortune 100. With a focus on delivering concise, exhaustive, relevant, and in-depth search results, references, and links. See also the section on 31 features to substantially boost RAG/LLM performance.

Part II: Outperforming Neural Nets and Classic AI discusses related large-scale systems also benefiting from a light-weight but more efficient architecture. It features LLMs for clustering, classification, and taxonomy creation, leveraging the knowledge graphs embedded in and retrieved from the input corpus when crawling. Then, in chapters 7 and 8, I focus on tabular data synthetization, presenting techniques such as No-GAN, that significantly outperform neural networks, along with the best evaluation metric. The methodology in chapter 9 applies to most AI problems. It offers a generic tool to improve any existing architecture relying on gradient descent, such as deep neural networks. Finally, chapter 10 discusses deep retrieval and multi-index chunking for PDF repositories, capturing elements missed by most RAG/LLM systems, with Nvidia case study.

Part III: Innovations in Statistical AI features a collection of methods that you can integrate in any AI system to boost performance. Based on a modern approach to statistical AI, they cover probabilistic vector search, sampling outside the observation range, strong random number generators, math-free gradient descent, beating the slow statistical convergence of parameter estimates dictated by the Central Limit Theorem, exact geospatial interpolation for non-smooth systems, and more. Efficient chunking and indexing for LLMs is the topic of chapter 11. Finally, chapter 16 shows how to optimize trading strategies to consistently outperform the stock market.

About the author

Vincent Granville is a pioneering GenAI scientist and machine learning expert, co-founder of Data Science Central (acquired by a publicly traded company in 2020), Chief AI Scientist at [MLTechniques.com](#), former VC-funded executive, author and patent owner – one related to LLM. 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 the author of multiple books, available [here](#), including “Synthetic Data and Generative AI” (Elsevier, 2024). Vincent lives in Washington state, and enjoys doing research on stochastic processes, dynamical systems, experimental math and probabilistic number theory.

Part I

Hallucination-Free LLM with Real-Time Fine-Tuning

Chapter 1

Enterprise xLLM for Search & Retrieval

I discuss version 2.0 of xLLM, the in-memory enterprise multi-LLM with zero weight, no training, no transformer, no neural network, no latency, no cost, no GPU, and no hallucination. Based on explainable AI, self-tuned, made from scratch, customizable, and not relying on external API or Python libraries. Version 1.0 is presented in my article entitled “Custom Enterprise LLM/RAG with Real-Time Fine-Tuning”, posted [here](#). Since version 2.0 is backward-compatible and consists of several important additions, I included all the relevant material from the previous article, in this book. New additions include multitoken distillation when processing prompts, agents to meet user intent, singularization, and several improvements such as enhanced command menu. Most importantly, I added several illustrations, featuring xLLM in action as well as important parts of the code.

```
def update_nestedHash(hash, key, value, count=1):

    # 'key' is a word here, value is tuple or single value
    if key in hash:
        local_hash = hash[key]
    else:
        local_hash = {}
    if type(value) is not tuple:
        value = (value,)
    for item in value:
        if item in local_hash:
            local_hash[item] += count
        else:
            local_hash[item] = count
    hash[key] = local_hash
    return(hash)
```

Figure 1.1: Nested hash database, lines 12–27 in the code

1.1 Efficient database architecture: nested hashes

This chapter features an application of xLLM to extract information from a corporate corpus, using prompts referred to as “queries”. The goal is to serve the business user – typically an employee of the company or someone allowed access – with condensed, relevant pieces of information including links, examples, PDFs, tables, charts, definitions and so on, to professional queries. The original xLLM technology is described [in this presentation](#). The main differences with standard LLMs are:

- No training, no neural network involved. Thus, very fast and easy to fine-tune with explainable parameters, and much fewer tokens. Yet, most tokens consist of multiple terms and are called **multitokens**. Also, I use **variable-length embeddings**. The focus is on deliverable concise yet exhaustive results, with enough depth. Cosine similarity and dot products are replaced by customized **pmi** (pointwise mutual information, [\[Wiki\]](#)).
- Parameters have a different meaning in my context. In standard architectures, they represent the weights connecting neurons. You have billions or even trillions of them. But there is no neural network involved here: instead, I use parametric weights governed by a few top-level parameters. The weights – explicitly specified rather than iteratively computed – are not the parameters. My architecture uses two parameter sets: frontend and backend. The former are for scoring and relevancy; they are fine-tuned in real time

with no latency, by the user or with some algorithm. A relevancy score is shown to the user, for each retrieved item.

- I don't use vector or graph databases. Tables are stored as **nested hashes**, and fit in memory (no GPU needed). By nested hashes, I mean **key-value tables**, where the value may also be a key-value table. The format is similar to **JSON** objects, see Figures 1.1 and 1.3. In standard architectures, the central table stores the embeddings. Here, embeddings are one of many backend tables. In addition, there are many contextual tables (taxonomy, knowledge graph, URLs) built during the crawling. This is possible because input sources are well structured, and elements of structure are recovered thanks to **smart crawling**.
- The Python code does not use any library, nor any API call. Not even Pandas, Numpy, or NLTK. So you can run it in any environment without concern for library versioning. Yet it has fewer than 600 lines of code, including the fine-tuning part in real time. I plan to leverage some library functions in the future such as auto-correct, singularize, stem, stopwords and so on. However, home-made solutions offer more customization, such as ad-hoc **stopwords** lists specific to each sub-LLM, for increased performance. For instance, the one-letter word 'p' can not be eliminated if the sub-LLM deals with statistical concepts. The only exception to the "no library" rule is the Requests library, if you choose to download the test enterprise corpus from its GitHub location.
- This chapter focuses only on one part of an enterprise corpus: the internal documentation about how to implement or integrate AI and machine learning solutions. Other parts include marketing, IT, product, sales, legal and HR. A specific sub-LLM is built for each part, using the same architecture. The full LLM consists of these sub-LLMs, glued together with an **LLM router** to redirect user prompts to the specific parts, possibly spanning across multiple sub-LLMs. For instance, 'security' is found in multiple sub-LLMs.

1.2 From frontend prompts to backend tables

The prompt is first stripped of common words such as 'how to', 'example', or 'what is'. The result is called a shortened prompt. The stripped words may be treated separately to determine the user intent, called **action**. They are also stripped from the corpus (crawled data) but again, used to assign an action label to each text entity in the corpus. Then the shortened prompt is sorted in alphabetical order and broken down into sorted ***n*-grams**. A shortened prompt with n words gives rise to $2^n - 1$ sorted n -grams containing from one to n words. Without sorting, that number would be $1! + 2! + \dots + n!$, too large for fast processing.

Sorted n -grams detected in the prompt are then matched against the sorted n -grams found in the backend table **sorted_ngrams** based on the corpus. Each entry in that table is a key-value table. For instance, the entry for the key 'data mining' (a sorted n -gram) might be $\{ \text{'data mining': } 15, \text{'mining data': } 3 \}$. It means that 'data mining' is found 15 times in the corpus, while 'mining data' is found 3 times. Of course, n -grams not found in the corpus are not in that table either. The sorted n -grams table helps retrieve unsorted word combinations found in the corpus and match them back to unsorted n -grams in the prompt. This is in contrast to systems where word order is ignored, leading to problems.

```
tableNames = (
    'dictionary',      # multitokens (key = multitoken)
    'hash_pairs',     # multitoken associations (key = pairs of multitokens)
    'ctokens',        # not adjacent pairs in hash_pairs (key = pairs of multitokens)
    'hash_context1',  # categories (key = multitoken)
    'hash_context2',  # tags (key = multitoken)
    'hash_context3',  # titles (key = multitoken)
    'hash_context4',  # descriptions (key = multitoken)
    'hash_context5',  # meta (key = multitoken)
    'hash_ID',        # text entity ID table (key = multitoken, value is list of IDs)
    'hash_agents',   # agents (key = multitoken)
    'full_content',  # full content (key = multitoken)
    'ID_to_content', # full content attached to text entity ID (key = text entity ID)
    'ID_to_agents',  # map text entity ID to agents list (key = text entity ID)
    'ID_size',        # content size (key = text entity ID)
    'KW_map',         # for singularization, map kw to single-token dictionary entry
    'stopwords',      # stopword list
)
```

Figure 1.2: Primary backend tables, lines 193–210 in the code

From there, each backend table is queried to retrieve the value attached to a specific n -gram found in the prompt. The value in question is also a key-value table: for instance a list of URLs where the key is an URL and the value is the number of occurrences of the n -gram in question, on the landing page. In each section (titles,

URLs, descriptions and so on) results shown to the user are displayed in relevancy order, with a higher weight assigned to n -grams (that is, multitokens) consisting of many words, as opposed to multitokens consisting of one or two words. Embeddings are derived from a backend table called `hash_pairs` consisting of pairs of multitokens found in the same sub-entity in the corpus. Finally, multitokens may or may not be adjacent. Pairs with non-adjacent multitokens are called **contextual pairs**. Occurrences of both multitokens, as well as joint occurrence (when both are simultaneously found in a same sub-entity) are used to compute **pmi**, the core relevancy metric. Embeddings are stored in the `embeddings` key-value backend table, also indexed by multitokens. Again, values are key-value tables, but this time the nested values are **pmi** scores.

```
extraWeights = backendParams['extraWeights']
word = word.lower() # add stemming
weight = 1.0
if word in category:
    weight += extraWeights['category']
if word in tag_list:
    weight += extraWeights['tag_list']
if word in title:
    weight += extraWeights['title']
if word in meta:
    weight += extraWeights['meta']

update_hash(backendTables['dictionary'], word, weight)
update_nestedHash(backendTables['hash_context1'], word, category)
update_nestedHash(backendTables['hash_context2'], word, tag_list)
update_nestedHash(backendTables['hash_context3'], word, title)
update_nestedHash(backendTables['hash_context4'], word, description) # takes space, don't build?
update_nestedHash(backendTables['hash_context5'], word, meta)
update_nestedHash(backendTables['hash_ID'], word, ID)
update_nestedHash(backendTables['hash_agents'], word, agents)
for agent in agents:
    update_nestedHash(backendTables['ID_to_agents'], ID, agent)
update_nestedHash(backendTables['full_content'], word, full_content) # takes space, don't build?
update_nestedHash(backendTables['ID_to_content'], ID, full_content)
```

Figure 1.3: Updating primary backend tables, lines 61–72 in the code

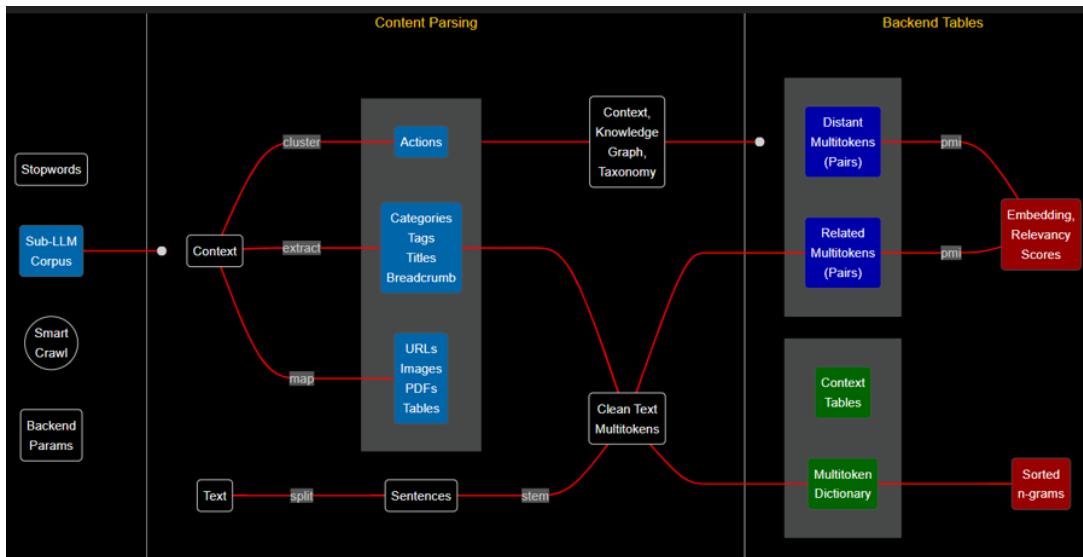


Figure 1.4: From crawl to backend tables (high resolution [here](#))

1.3 What is not covered here

The goal was to create a MVP (minimum viable product) featuring the original architecture and the fine-tuning capability in real time. With compact and generic code, to help you easily add backend tables of your choice, for instance to retrieve images, PDFs, spreadsheets and so on when available in your corpus.

Some features are not yet implemented in this version, but available in the previous version discussed [here](#) and in my book “State of the Art in GenAI & LLMs – Creative Projects, with Solutions”, available [here](#). The following will be available in the next release: auto-correct, stemming, singularization and other text processing

techniques, both applied to the corpus (crawled data) and the prompt. I will also add the ability to use pre-computed backend tables rather than building them from the crawl each time. Backend tables produced with the default backend parameters (see code lines 193–262 in section 4.1) are on GitHub, [here](#).

Also to be included in the next release: corpus augmentation with synonyms and abbreviations dictionaries, as well as contextual multitokens. The latter is implemented in the previous version and discussed in section 6. It consists of tokens containing non-adjacent words in the corpus. However, contextual pairs are included in the current release: it consists of pairs of non-adjacent multitokens, stored in a table called `ctokens` used to produce the embeddings. See lines 183–186 in the code. Then, words such as ‘San Francisco’ must be treated as single tokens.

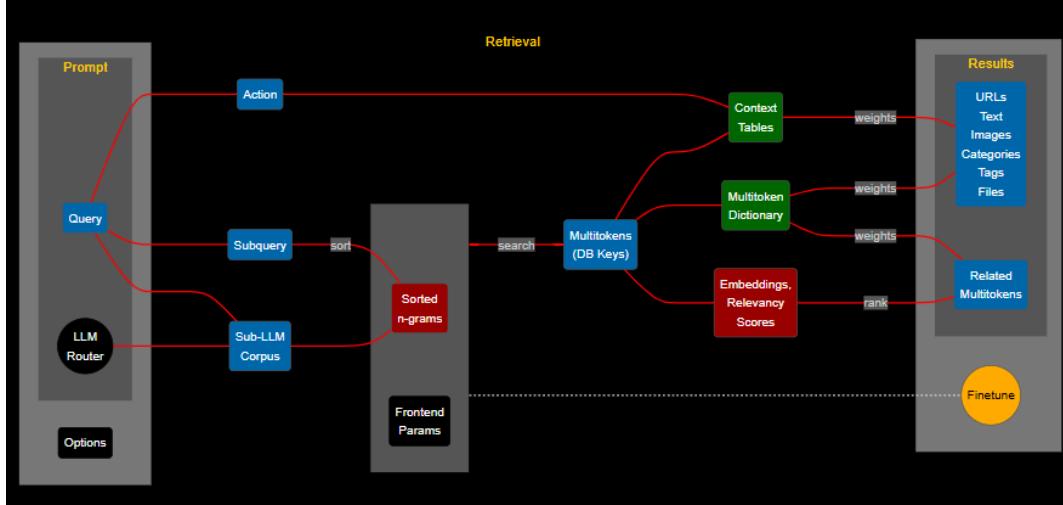


Figure 1.5: From prompt to query results, via backend tables (high resolution [here](#))

Finally, prompts are not broken down into sub-prompts. But the concept of `action` is now implemented. An action determines the user intent: whether he/she is searching for ‘how to’, ‘what is’, ‘examples’, ‘data’, ‘comparisons’, and so on. It requires the addition of an extra backend table, corresponding to the ‘action’ field in the text entities, along with ‘category’, ‘description’, ‘title’ and so on. However, there is no ‘action’ field. It must be constructed with a clustering algorithm applied to the corpus as a pre-processing step, to add action labels to each text entity. My current approach is actually simpler and discussed in chapter 2. Last but not least, the benefit of scoring text entities against prompts, and how to do it, is discussed in section 15.1.2.1.

Chapter 2

Parameters, features, and fine-tuning

In the case study discussed here, the input source consists of about 500 text elements stored as JSON entities, each with a number of fields: title, description, category, tags, URL, ID, and so on. It comes from a Bubble database that populates the website where the corpus is accessible to end-users. In the Python code, the list of entities covering the entire corpus is named `entities`, while a single entry is named `entity`. For each entity, the various fields are stored in a local key-value table called `hash_crawl`, where the key is a field name (for instance, category) and the value is the corresponding content. See lines 292–338 in the code in section 4.1. The full corpus (the anonymized input source) is available as a text file named `repository.txt`, [here](#) on GitHub.

2.1 Backend parameters

Multitokens contain up to 4 terms, as specified by the backend parameter `max_multitokens` in line 265 in the code. The `hash_pairs` table consists of multitokens pairs, each with up to 3 terms: see parameter `maxTerms` in line 267. The maximum gap allowed between two contextual multitokens is 3 terms: see parameter `maxDist` in line 266. These limitations are set to prevent the number of pairs and tokens from exploding. In the end, there are 12,575 multitokens, stored in the `dictionary` table, after removing stopwords. The total number of multitoken pairs is 223,154, while the size of the corpus is 427KB uncompressed.

Stopwords – the words to ignore when building the tables – are manually detected by looking at the most frequent tokens, both in the corpus and in prompt result: see the list in lines 216–222. Finally, when counting multitoken occurrences, appearances in categories, titles and tags get an extra boost, compared to regular text: see lines 268–275 and Figure 1.3. For the full list of backend parameters, see Figure 2.1.

```
backendParams = {
    'max_multitoken': 4, # max. consecutive terms per multi-token for inclusion in dictionary
    'maxDist' : 3,        # max. position delta between 2 multitokens to link them in hash_pairs
    'maxTerms': 3,        # maxTerms must be <= max_multitoken
    'extraWeights' :
        {
            'description': 0.0,
            'category': 0.3,
            'tag_list': 0.4,
            'title': 0.2,
            'meta': 0.1
        }
}
```

Figure 2.1: Backend parameters, lines 697–722 in the code

I did not include `embeddings` and `sorted_ngrams` in the `backendTables` structure in lines 193–214, because they are built on top of primary backend tables, more specifically `dictionary` and `hash_pairs`. The `pmi` values attached to the embeddings are computed as follows:

$$\text{pmi}(t_A, t_B) = \frac{n_{AB}}{\sqrt{n_A \cdot n_B}}, \quad (2.1)$$

where n_A , n_B , n_{AB} are the counts (computed on the full corpus) respectively for multitokens t_A , t_B , and the joint occurrence of t_A , t_B within a same sub-entity (that is, a sentence identified by separators, within a text entity). The user can choose a different formula, or different separators. Primary backend tables are listed in Figure 1.2.

2.2 Frontend parameters

Given the small size of the corpus and backend tables, the backend parameters can be updated in real time. Currently, the code allows the user to easily update the frontend parameters while testing various prompts. The frontend parameters are found in lines 699–721 in the code, and in Figure 3.2. They control the results displayed, including the choice of a customized pmi function, and top keywords to exclude such as ‘data’ found in almost all text entities. Adding ‘data’ to the ignore list does not eliminate results based on multitokens containing ‘data’, as long as the multitokens in question consist of more than one word, such as ‘data asset’.

When entering a prompt, the end-user can choose pre-selected queries listed in lines 760–769, his/her own queries, or simple instructions to update or view the frontend parameters, using one of the options in lines 773–792. The catch-all parameter set (with all values set to zero) yields the largest potential output. Do not use it except for debugging, as the output may be very long. However, if you want to try it, choose the option –f for full results. This is accomplished by entering –f on the command prompt.

```
def default_frontendParams():

    frontendParams = {
        'embeddingKeyMinSize': 1, # try 2
        'embeddingValuesMinSize': 2,
        'min_pmi': 0.00,
        'nABmin': 1,
        'Customized_pmi': True,
        'ContextMultitokenMinSize': 1, # try 2
        'minOutputListSize': 1,
        'bypassIgnoreList': False,
        'ignoreList': ('data',),
        'maxTokenCount': 100, # ignore generic tokens if large enough
        'show': {
            # names of sections to display in output results
            'Embeddings': True,
            'Category': True,
            'Tags': True,
            'Titles': True,
            'Descr.': False, # do not built to save space
            'Whole': False, # do not build to save space
            'ID': True,
            'Agents': True,
        }
    }
    return(frontendParams)
```

Figure 2.2: Default frontend parameters, lines 699–721 in the code

```
agent_map = {
    'template':'Template',
    'policy':'Policy',
    'governance':'Governance',
    'documentation':'Documentation',
    'best practice':'Best Practices',
    'bestpractice':'Best Practices',
    'standard':'Standards',
    'naming':'Naming',
    'glossary':'Glossary',
    'historical data':'Data',
    'overview':'Overview',
    'training':'Training',
    'genai!':'GenAI',
    'gen ai!':'GenAI',
    'example':'Example',
    'example1':'Example',
    'example2':'Example',
}
```

Figure 2.3: Agent map, lines 227–245 in the code

2.3 Agents

Agents determine the user intent to retrieve the appropriate content. For instance:, examples, data, definitions, best practices, standards, on-boarding, and so on. In Figure 1.5, they are represented by the **action** box. One way to create an **agentic LLM** is to add an agent field in each **text entity** when crawling the corpus. See sample text entity in Table 2.1. You can do it using clustering techniques, applied to the corpus. Text entities are

relatively small pieces of content coming straight from the corpus, usually determined by the corpus structure: in this case, a bubble database, but it could also be a repository of PDF documents or web pages.

Getting a list of top multitokens helps you build your agent backend table. In our example, see the list in question Table 2.1, extracted from the dictionary backend table. Another option consists in analyzing dozens, thousands, or millions of user prompts to identify potential actions. The ideal solution is to combine all these options to create agents that correspond not only to user intent, but also to what is actually in the corpus.

The agent map for my case study, is pictured in Figure 3.2. I will improve the format in the next version, and use a many-to-many rather than many-to-one table. In the key-value pairs in the picture, the value on the right is an agent, while the key on the left is a multitoken. The structure thus maps words found in the corpus, to agents. Agents are then incorporated to backend tables for retrieval. In my current implementation, there are two agent backend tables, besides `agent_map` just described:

- `hash_agents` indexed by multitokens found in `dictionary`, to retrieve agents associated to multitokens.
- `ID_to_agents` indexed by text entity IDs (`ID` in the code), to retrieve agents associated to entity IDs.

These two tables are used to produce the agent section in the query results, as shown in Figure 2.4. For details, see lines 679–686 in the code. For instance, the fourth line in the picture tells you that the multitoken ‘data assets’ is associated to agent ‘Governance’ (among others), and that four text entity IDs match this combination: 42, 48, 199, 259, with 259 having the most content with 1153 characters.

In Figure 2.4, the size of each entity ID is also displayed to help the user identify IDs with more content; they might be more valuable. With the command `-i ID` in the prompt box, the user can then retrieve the full content of entity ID, in a format similar to Table 2.1. Two extra backend tables are involved in the process: `hash_size` and `ID_to_content`.

('Data', 'detailed') --> (511, 513)
('Example', 'data assets') --> (90,)
('Example', 'detailed') --> (90,)
('Governance', 'data assets') --> (42, 48, 199, 259)
('Governance', 'detailed') --> (101, 107)
('Governance', 'information assets') --> (223,)
('Policy', 'data assets') --> (42, 48, 199)
('Policy', 'detailed') --> (101,)
('Policy', 'information assets') --> (223,)
('Template', 'detailed') --> (107,)
ID Size
511 690
513 692
90 772
42 948
48 916
199 980
259 1153
101 851
107 1242
223 978

Figure 2.4: Example of agent section shown in query results

Currently, the agent(s) are not automatically detected from the user prompt. I will add this feature in the next version. In the meanwhile, it is possible to display the full list of agents to the user, and let him make his selection. Finally, my agents do not perform actions such as writing messages or solving math problems. Their goal is to deliver more relevant results, based on what users are looking for by analyzing prompt data. A different version of my xLLM performs `clustering`, build taxonomies, and make `predictions` based on text: see [here](#), and Figure 4.1.

2.4 Reproducibility

Most GenAI applications rely on deep neural networks (DNN) such as `GANs` (generative adversarial networks). This is the case for `transformers`, a component of many LLMs. These DNNs rely on random numbers to generate latent variables. The result can be very sensitive to the `seed`.

In many instances, particularly for synthetic data generation and GPU-based apps, the author does not specify seeds for the various PRNG (pseudo-random number generator) involved, be it from the Numpy, Random, Pandas, PyTorch libraries, base Python, or GPU. The result is lack of reproducibility. This is not the case with my algorithms, whether GAN or NoGAN. All of them lead to reproducible results, including the xLLM system described here, which does not rely on transformers or random numbers.

There have been some attempts to improve the situation recently, for instance with the `set_seed` function in some transformer libraries. However, it is not a full fix. Furthermore, the internal PRNGs found in Python libraries are subject to change without control on your side. To avoid these problems, I invite to check out my own PRNGs, some of them faster and better than any other one on the market. See my article “Fast Random Generators with Infinite Period for Large-Scale Reproducible AI and Cryptography”, available [here](#).

2.5 Singularization, stemming, auto-correct

The `KW_map` backend table built in lines 870–888 in the code (see Figure 2.5), is a first attempt at adding NLP functions without using Python libraries. The table is created and saved after running the full code for the first time. Python libraries have glitches that can result in hallucinations, for instance singularizing “hypothesis” to “hypothesi”. They require exception lists such as do-not-singularize as a workaround. Thus the idea to avoid them.

The code featured in Figure 2.5 links the singular and plural version of single-tokens found in the dictionary (when both exist), so that a user looking for (say) “tests” also gets result coming from “test”. See lines 822–823 in the code when processing frontend prompts, and lines 148–149 when building backend tables.

More NLP functions will be added in the next version, including from Python libraries, such as `singularize`, `stemming` and `auto-correct`. To minimize hallucinations, it is better to have a specific list for each sub-LLM. Even then, one must be careful to avoid singularizing (say) “timeliness” to “timelines” or “practices” (noun) to “practice” (verb or noun). In the next version, `KW_map` will also be used as a `synonyms` and `abbreviation` dictionary.

```
def create_KW_map(dictionary):
    # singularization
    # map key to KW_map[key], here key is a single token
    # need to map unseen prompt tokens to related dictionary entries
    #   example: ANOVA -> analysis~variance, ...

    OUT = open("KW_map.txt", "w")

    for key in dictionary:
        if key.count('~') == 0:
            j = len(key)
            keyB = key[0:j-1]
            if keyB in dictionary and key[j-1] == 's':
                if dictionary[key] > dictionary[keyB]:
                    OUT.write(keyB + "\t" + key + "\n")
                else:
                    OUT.write(key + "\t" + keyB + "\n")
        OUT.close()
    return()
```

Figure 2.5: Building the `KW_map` backend table

2.6 Augmentation, distillation, and frontend tables

I build two frontend tables `q_dictionary` and `q_embeddings` each time a new prompt is generated, in order to retrieve the relevant content from the corpus. These tables are similar and linked to backend `dictionary` and `embeddings`, but far smaller and focusing on prompt content only. See lines 828–855 in the code.

Then, I remove single tokens that are part of a multitone when both have the same count in the dictionary. See line 862 in the code, calling the function pictured in Figure 2.6. It makes the output shown to the user, less cluttered. This step is called `distillation`. In standard LLMs, distillation is performed on backend tokens using a different mechanism, since multitone are usually absent; it may result in hallucinations if not done properly. Also, in standard LLMs, the motivation is different: reducing a 500 billion token list, to (say) 50 billion. In xLLM, token lists are at least 1000 times smaller, so there is no real need for backend distillation.

Also, I keep a single copy of duplicate entities, see section 2.7. In the next version, only a limited number selected items will be shown to the user, based on relevancy score, rather than a full list. Even now, it is possible to drastically reduce the size of the output by choosing frontend parameters accordingly.

Finally, you can extend the corpus with external input sources. This step is called **augmentation** in **RAG** (retrieval augmentation generation) systems. The augmented data is split into standard text entities, processed as standard entities, possibly with the ‘Augmented’ tag to distinguish them from organic content, when displaying results. It is also possible to perform **knowledge graph** and **taxonomy augmentation**, as described in my article “Build and Evaluate High Performance Taxonomy-Based LLMs From Scratch”, available [here](#).

```
def distill_frontendTables(q_dictionary, q_embeddings, frontendParams):
    # purge q_dictionary then q_embeddings (frontend tables)

    maxTokenCount = frontendParams['maxTokenCount']
    local_hash = {}
    for key in q_dictionary:
        if q_dictionary[key] > maxTokenCount:
            local_hash[key] = 1
    for keyA in q_dictionary:
        for keyB in q_dictionary:
            nA = q_dictionary[keyA]
            nB = q_dictionary[keyB]
            if keyA != keyB:
                if (keyA in keyB and nA == nB) or (keyA in keyB.split('~')):
                    local_hash[keyA] = 1
    for key in local_hash:
        del q_dictionary[key]

    local_hash = {}
    for key in q_embeddings:
        if key[0] not in q_dictionary:
            local_hash[key] = 1
    for key in local_hash:
        del q_embeddings[key]

    return(q_dictionary, q_embeddings)
```

Figure 2.6: Frontend token distillation before returning results

Table 2.1: Sample text entity from corporate corpus

Field	Value
Entity ID	1682014217673x617007804545499100
Created Date	2023-04-20T18:10:18.215Z
Modified Date	2024-06-04T16:42:51.866Z
Created by	1681751874529x883105704081238400
Title	Business Metadata Template
Description	It outlines detailed instructions for completing the template accurately, covering various sections such as data dictionary, data source, sensitivity information, and roles. After filling out the template, users can interpret the entered data, ensuring clarity on sensitivity classifications, business details, and key roles. Once completed and reviewed, the metadata is uploaded to MLTxQuest, making it accessible through the MLTxQuest portal for all authorized users, thereby centralizing and simplifying access to critical information within the organization.
Tags	metadata, mltxquest, business
Categories	Governance
URLs	

2.7 In-memory database, latency, and scalability

The whole corpus and the backend tables easily fit in memory even on an old laptop. Building the tables takes less than a second. Once the tables are created or loaded, there is no **latency**. This is due to the small size of the corpus, and because the implementation described here deals with only one sub-LLM; the full corpus requires about 15 sub-LLMs. However, for **scalability**, here are some recommendations:

- Pre-load the backend tables once they have been created on the first run; do not build them each time.
- Do not create the `hash_context4` and `full_content` tables; these are among the largest, and redun-

dant with `ID_to_content`.

- Keep only one copy of identical text entities: ideally remove duplicates directly in the corpus, as opposed to using memory-consuming `entity_list` (see lines 296 and 305).
- Unless feasible, do not store `ID_to_content` that maps the entity IDs to their full content, in memory. Only store the list of IDs using small ID tables (`hash_ID`, `ID_size`, `ID_to_agents`). The idea is to search for matching IDs in the backend tables when processing a prompt, and then retrieve the actual content from a database matching IDs to content.
- A distributed architecture can be useful, whereas separate sub-LLMs are stored on different clusters, if needed.

For the time being, my system is a full **in-memory LLM** with **in-memory database**. All the backend tables and text entities (see example in Table 2.1) are stored in memory.

Chapter 3

Case study

I now show how **xLLM** (the name of my LLM) works on one part of a corporate corpus (fortune 100 company), dealing with documentation on internal AI systems and policies. Here, I implemented the **sub-LLM** dedicated to this content. The other parts – marketing, products, finance, sales, legal, HR, and so on – require separate overlapping sub-LLMs not covered here. The anonymized corpus consists of about 300 distinct text entities, and can be found [here](#). Table 2.1 features a sample text entity. The full corpus would be processed with a **multi-LLM** and LLM router.

In addition to the original features described in chapter 2, xLLM comes with a command menu, shown in Figure 3.1. This menu allows you to enter a standard prompt, but also to change the front-end parameters for **real-time fine-tuning**. Figures 1.4 and 1.5 show the main components and workflow for a single sub-LLM. Zoom in for higher resolution. For best resolution, download the original [here](#) on Google Drive for the backend diagram, and [here](#) for the frontend. Finally, the home-made LLM discussed here can be used to create a new taxonomy of the crawled corpus, based on top multitokens. These are listed, from left to right and top to bottom by order of importance, in Table 3.1. Note that here, I did not give a higher weight to multitokens consisting of multiple words. The table was produced using lines 372–375 in the Python code.

Table 3.1: Top multitokens found in corpus, ordered by importance

adls	storage	azure	examples	adf
csa	pipeline	development	framework	architecture
design	mltxdat	process	extract	orc
overview	quality	databricks	data quality	table
guidelines	new	guide	best practices	performance
platform	metadata	solution	business	products
project	resources	create	request	mltxhub
case	zones	key	feature	governance
devops	github	naming	standards	ops
service	monitoring	glossary	global	policy
documentation	data governance	management	document	user
roles	team	onboarding	access	integration
infrastructure	responsibilities	security	engineering	bi
ci	cd	code	learning	support
foundation	admin	timbr	ai	metrics
index	mltxdoc	serving	semantic	layer
applications	environment	mltxquest	deployment	training
api	components	essential	fitness	score
model	genai	machine learning	governance	alpha
ai platform	genai platform	systems		

Now, let's try two prompts, starting with ‘metadata template’. With the default frontend parameters, one text

entity is found: the correct one entitled ‘business metadata template’, because the system tries to detect the joint presence of the two words ‘data’ and ‘template’ within a same **text sub-entity**, whether adjacent or not. A lot more would be displayed if using the catch-all parameter set. The interesting part is the embeddings, linking the prompt to other multitokens, especially ‘instructions completing template’, ‘completing template accurately’, ‘filling out template’ and ‘completed reviewed metadata’. These multitokens, also linked to other text entities, are of precious help. They can be used to extent the search or build **agents**.

My second test prompt is ‘data governance best practices’. It returns far more results, although few clearly stand out based on the relevancy scores. The most relevant category is ‘governance’, the most relevant tags are ‘DQ’ and ‘data quality’, with one text entity dominating the results. Its title is ‘Data Quality Lifecycle’. The other titles listed in the results are ‘Data Literacy and Training Policy’, ‘Audit and Compliance Policy’, ‘Data Governance Vision’, and ‘Data Steward Policy’. Related multitokens include ‘robust data governance’, ‘best practices glossary’, ‘training policy’, ‘data informed decision making’ and ‘data governance practices’.

3.1 Real-time fine-tuning, prompts, and command menu

Here I illustrate a full xLLM session, using a more complex sample query. It also involves fine-tuning front-end parameters in real time. The full session with commands from the command menu, and output results, is listed in section 3.2. Figure 3.1 shows how the command prompt looks like, as well as the result after executing the **-v** command.

```

Command menu:

-q : print last non-command prompt
-x : print sample queries
-p key value : set frontendParams[key] = value
-f : use catch-all parameter set for debugging
-d : use default parameter set
-v : view parameter set
-a multitoken : add multitoken to 'ignore' list
-r multitoken : remove multitoken from 'ignore' list
-l : view 'ignore' list
-i ID1 ID2 ... : print content of text entities ID1 ID2 ...
-s : print size of core backend tables
-c F1 F2 ... : show sections F1 F2 ... in output results

To view available sections for -c command, enter -v command.
To view available keys for -p command, enter -v command.
For -i command, choose IDs from list shown in prompt results.
For standard prompts, enter text not starting with '-' or digit.
-----

Query, command, or integer in [0, 7] for sample query: -v

Key Description Value
2 min_pmi 0.0
3 nABmin 1
4 Customized_pmi True
5 ContextMultitokenMinSize 1
6 minOutputListSize 1
7 bypassIgnoreList False
8 ignoreList ('data',)
9 maxTokenCount 100

Show sections:

Embeddings True
Category True
Tags True
Titles True
Descr. False
Whole False
ID True
Agents True

```

Figure 3.1: Command options and frontend parameters

I started with sample query 6 (the first action in Table 3.2), then looked at the results, fine-tune parameters (actions 5 and 6) and removed some junk (action 3), then rerun the query (action 7) then focused on getting article titles only (action 8) and rerun the query a final time (action 9).

Action	Command	Log Line
1	6	23
2	-i 107 259	377
3	-a detailed	422
4	-v	445
5	-p 6 2	493
6	-p 2 0.50	516
7	6	539
8	-c Titles	688
9	6	711

Table 3.2: Sample xLLM session

The detailed log with executed commands and all the output is shown in section 3.2. In particular, the nine commands in Table 3.2 are found at the corresponding line numbers (rightmost column in Table 3.2), in the log file in section 3.2. Perhaps the most useful results consist of the IDs attached to agents and multitokens related to the prompt, in lines 542–567. Also pictured in Figure 3.2, along with interpretation details in section 2.3. The actual content corresponding to these IDs is shown in lines 593–641. The prompt itself is shown in line 24.

I was particularly interested in finding the articles (text entities) matching my prompt, especially the titles, to check out those that interest me most. This is accomplished with the `-c Titles` command, and the results are shown in lines 988–1001. In the next code release, the corresponding text entity IDs will also be displayed along with the titles, as in the Agents section (Figure 3.2). This way, it is very easy to retrieve the full content corresponding to the titles in question, with the `-i` command.

Since everything is already built for this functionality, adding a few lines of code to retrieve the IDs is straightforward. I encourage you to modify the code accordingly, on your own. This would be a good exercise to help you understand my architecture. The next step is to also add the corresponding IDs in the other sections (Categories, Tags, Descr., Whole, and so on).

3.2 Sample session

Here is the full log obtained by executing the commands in Table 3.2, including standard prompts. The executed program is called `xllm-enterprise-v2.py`, with source code in section 4.1 and on GitHub. The input data source, also on GitHub, is a fully anonymized version of one part of a corporate corpus. Keyword pairs (at the beginning) come from the embeddings backend table. Entries flagged with a star (*) mark contextual pairs. Also,

- Some original word from the prompt, is on the right ('word' column in line 26).
- The related `multitoken` from the `embeddings` backend table, associated to the prompt word in question, is in the middle (the 'token' column). The user may try some of these tokens in a subsequent prompt.
- The 'F' column indicates if the pair is contextual or not.
- The 'pmi' column represents the pointwise mutual information (`PMI`), a measure of association between a word and a token.
- The 'N' column on the left shows the number of joint occurrences of ('token', 'word') in the corpus.

Below is the session log.

```

1
2 Command menu:
3
4 -q      : print last non-command prompt
5 -x      : print sample queries
6 -p key value : set frontendParams[key] = value
7 -f      : use catch-all parameter set for debugging
8 -d      : use default parameter set
9 -v      : view parameter set
10 -a multitoken : add multitoken to 'ignore' list
11 -r multitoken : remove multitoken from 'ignore' list
12 -l      : view 'ignore' list
13 -i ID1 ID2 ... : print content of text entities ID1 ID2 ...
14 -s      : print size of core backend tables
15 -c F1 F2 ... : show sections F1 F2 ... in output results

```

```

16
17 To view available sections for -c command, enter -v command.
18 To view available keys for -p command, enter -v command.
19 For -i command, choose IDs from list shown in prompt results.
20 For standard prompts, enter text not starting with '-' or digit.
21
22 Query, command, or integer in [0, 7] for sample query: 6
23 query: MLTxQuest Data Assets Detailed Information page
24
25 N pmi F token [from embeddings] word [from prompt]
26
27 1 1.00 * confidentiality|availability information|assets
28 1 1.00 * availability|organization information|assets
29 1 1.00 * confidentiality|availability|organization information|assets
30 1 1.00 - availability|organization|information information|assets
31 1 1.00 * integrity|confidentiality|availability information|assets
32 1 1.00 - organization|information information|assets
33 1 1.00 - organization|information|assets information|assets
34 1 1.00 * systems|managed information|assets
35 1 1.00 * managed|mltxdat information|assets
36 1 1.00 * systems|managed|mltxdat information|assets
37 1 1.00 - managed|mltxdat|csa information|assets
38 1 1.00 - platform|against information|assets
39 1 1.00 * platform|against|threats information|assets
40 1 1.00 * threats|such information|assets
41 1 1.00 * data|systems|managed information|assets
42 1 1.00 - csa|platform|against information|assets
43 1 1.00 * against|threats information|assets
44 1 1.00 * against|threats|such information|assets
45 1 0.71 * navigating|data page|mltxquest
46 1 0.71 * efficiently|navigating|data page|mltxquest
47 1 0.71 * navigating|data|assets page|mltxquest
48 1 0.71 - assets|page page|mltxquest
49 1 0.71 - data|assets|page page|mltxquest
50 1 0.71 - page|mltxquest|while page|mltxquest
51 1 0.71 * while|facilitating page|mltxquest
52 1 0.71 * while|facilitating|comprehensive page|mltxquest
53 1 0.71 - assets|page|mltxquest page|mltxquest
54 1 0.71 - mltxquest|while page|mltxquest
55 1 0.71 * mltxquest|while|facilitating page|mltxquest
56 1 0.71 * facilitating|comprehensive page|mltxquest
57 1 0.71 - assets|data page|mltxquest
58 1 0.71 - information|page page|mltxquest
59 1 0.71 - page|mltxquest|data page|mltxquest
60 1 0.71 - information|page|mltxquest page|mltxquest
61 1 0.71 - mltxquest|data page|mltxquest
62 1 0.71 * mltxquest|data|assets page|mltxquest
63 1 0.71 * assets|users page|mltxquest
64 1 0.71 * data|assets|users page|mltxquest
65 1 0.71 - mltxdat|csa|platform information|assets
66 1 0.71 - csa|platform information|assets
67 2 0.67 * users|efficiently data|assets
68 2 0.67 * efficiently|navigating data|assets
69 2 0.67 * users|efficiently|navigating data|assets
70 2 0.67 * aid|users|efficiently data|assets
71 2 0.50 * global|search detailed
72 2 0.50 - detailed|process detailed
73 2 0.50 * process|migrating detailed
74 2 0.50 * detailed|process|migrating detailed
75 2 0.50 * migrating|historical detailed
76 2 0.50 * process|migrating|historical detailed
77 2 0.50 - describes|detailed detailed
78 2 0.50 - describes|detailed|process detailed
79 2 0.47 * data|assets page|mltxquest
80 2 0.47 * page|mltxquest data|assets
81 1 0.45 - mltxdat|csa information|assets
82 2 0.41 - data|migration detailed
83 1 0.35 * guide|global detailed
84 1 0.35 * guide|global|search detailed
85 1 0.35 * information|search detailed
86 1 0.35 * search|data detailed
87 1 0.35 * information|search|data detailed
88
89 N = occurrences of (token, word) in corpus. F = * if contextual pair.
90 If no result, try option '-p f'.
91

```

```

92 SECTION: Category
93
94   Category: 'Products' [6 entries]
95     Linked to: page|mltxquest (2)
96     Linked to: detailed (8)
97     Linked to: information|page|mltxquest|data (1)
98     Linked to: data|assets (9)
99     Linked to: data|assets|page|mltxquest (1)
100    Linked to: page|mltxquest|data|assets (1)
101
102   Category: 'Governance' [3 entries]
103     Linked to: detailed (8)
104     Linked to: information|assets (1)
105     Linked to: data|assets (9)
106
107 SECTION: Tags
108
109   Tags: MLTxQuest [6 entries]
110     Linked to: page|mltxquest (2)
111     Linked to: detailed (8)
112     Linked to: information|page|mltxquest|data (1)
113     Linked to: data|assets (9)
114     Linked to: data|assets|page|mltxquest (1)
115     Linked to: page|mltxquest|data|assets (1)
116
117   Tags: Guideline [3 entries]
118     Linked to: page|mltxquest (2)
119     Linked to: data|assets (9)
120     Linked to: data|assets|page|mltxquest (1)
121
122   Tags: Guidelines [5 entries]
123     Linked to: page|mltxquest (2)
124     Linked to: detailed (8)
125     Linked to: information|page|mltxquest|data (1)
126     Linked to: data|assets (9)
127     Linked to: page|mltxquest|data|assets (1)
128
129   Tags: example1 [2 entries]
130     Linked to: detailed (8)
131     Linked to: data|assets (9)
132
133   Tags: example2 [2 entries]
134     Linked to: detailed (8)
135     Linked to: data|assets (9)
136
137   Tags: governance [2 entries]
138     Linked to: detailed (8)
139     Linked to: data|assets (9)
140
141   Tags: roles [1 entries]
142     Linked to: detailed (8)
143
144   Tags: raci [1 entries]
145     Linked to: detailed (8)
146
147 SECTION: Titles
148
149   Titles: 'MLTxQuest - Data Assets' [3 entries]
150     Linked to: page|mltxquest (2)
151     Linked to: data|assets (9)
152     Linked to: data|assets|page|mltxquest (1)
153
154   Titles: 'MLTxQuest-Data Asset Deta' [5 entries]
155     Linked to: page|mltxquest (2)
156     Linked to: detailed (8)
157     Linked to: information|page|mltxquest|data (1)
158     Linked to: data|assets (9)
159     Linked to: page|mltxquest|data|assets (1)
160
161   Titles: 'MLTxQuest - Global Search' [2 entries]
162     Linked to: detailed (8)
163     Linked to: data|assets (9)
164
165   Titles: 'Roles and Responsibilities Policy' [1 entries]
166     Linked to: detailed (8)
167

```

```

168 Titles: 'Business Metadata Template' [1 entries]
169 Linked to: detailed (8)
170
171 Titles: '[METRICS] Data Products' [1 entries]
172 Linked to: detailed (8)
173
174 Titles: 'Exploration - Monitoring' [1 entries]
175 Linked to: detailed (8)
176
177 Titles: 'Historical data migration' [1 entries]
178 Linked to: detailed (8)
179
180 Titles: 'Data Security Policy' [1 entries]
181 Linked to: information|assets (1)
182
183 Titles: 'Data Privacy Policy' [1 entries]
184 Linked to: data|assets (9)
185
186 Titles: 'Data Steward Policy' [1 entries]
187 Linked to: data|assets (9)
188
189 Titles: 'Data Owner Policy' [1 entries]
190 Linked to: data|assets (9)
191
192 Titles: 'MLTxQuest - Governance Badge' [1 entries]
193 Linked to: data|assets (9)
194
195 SECTION: Entity IDs
196
197 ID: 91 [3 entries]
198 Linked to: page|mltxquest (2)
199 Linked to: data|assets (9)
200 Linked to: data|assets|page|mltxquest (1)
201
202 ID: 92 [5 entries]
203 Linked to: page|mltxquest (2)
204 Linked to: detailed (8)
205 Linked to: information|page|mltxquest|data (1)
206 Linked to: data|assets (9)
207 Linked to: page|mltxquest|data|assets (1)
208
209 ID: 90 [2 entries]
210 Linked to: detailed (8)
211 Agents: ('Example',)
212 Linked to: data|assets (9)
213 Agents: ('Example',)
214
215 ID: 101 [1 entries]
216 Linked to: detailed (8)
217 Agents: ('Policy', 'Governance')
218
219 ID: 107 [1 entries]
220 Linked to: detailed (8)
221 Agents: ('Template', 'Governance')
222
223 ID: 139 [1 entries]
224 Linked to: detailed (8)
225
226 ID: 381 [1 entries]
227 Linked to: detailed (8)
228
229 SECTION: Agents
230
231 Agents: Example [2 entries]
232 Linked to: detailed (8)
233 Linked to: data|assets (9)
234
235 Agents: Policy [3 entries]
236 Linked to: detailed (8)
237 Linked to: information|assets (1)
238 Linked to: data|assets (9)
239
240 Agents: Governance [3 entries]
241 Linked to: detailed (8)
242 Linked to: information|assets (1)
243 Linked to: data|assets (9)

```

```

244
245 Agents: Template [1 entries]
246 Linked to: detailed (8)
247
248 Agents: Data [1 entries]
249 Linked to: detailed (8)
250
251 Above results based on words found in prompt, matched back to backend tables.
252 Numbers in parentheses are occurrences of word in corpus.
253
254
255 SECTION: Agent and Multitoken, with ID list
256 empty unless labels 'ID' and 'Agents' are in 'show'.
257
258 Agent Multitoken ID list
259
260 Data detailed 511, 513
261 Example data|assets 90
262 Example detailed 90
263 Governance data|assets 42, 48, 199, 259
264 Governance detailed 101, 107
265 Governance information|assets 223
266 Policy data|assets 42, 48, 199
267 Policy detailed 101
268 Policy information|assets 223
269 Template detailed 107
270
271 ID Size (of text entity)
272
273 511 690
274 513 692
275 90 772
276 42 948
277 48 916
278 199 980
279 259 1153
280 101 851
281 107 1242
282 223 978
283
284 Command menu:
285
286 -q : print last non-command prompt
287 -x : print sample queries
288 -p key value : set frontendParams[key] = value
289 -f : use catch-all parameter set for debugging
290 -d : use default parameter set
291 -v : view parameter set
292 -a multitoken : add multitoken to 'ignore' list
293 -r multitoken : remove multitoken from 'ignore' list
294 -l : view 'ignore' list
295 -i ID1 ID2 ... : print content of text entities ID1 ID2 ...
296 -s : print size of core backend tables
297 -c F1 F2 ... : show sections F1 F2 ... in output results
298
299 To view available sections for -c command, enter -v command.
300 To view available keys for -p command, enter -v command.
301 For -i command, choose IDs from list shown in prompt results.
302 For standard prompts, enter text not starting with '-' or digit.
303
304 Query, command, or integer in [0, 7] for sample query: -i 107 259
305
306 Entity ID 107
307
308 Modified Date : 2024-07-02T12:51:31.993Z
309 title_text : Business Metadata Template
310 description_text : It outlines detailed instructions for completing the template accurately,
311 covering various sections such as data dictionary, data source, sensitivity information, and
312 roles. After filling out the template, users can interpret the entered data, ensuring clarity
313 on sensitivity classifications, business details, and key roles. Once completed and reviewed,
314 the metadata is uploaded to MLTxQuest, making it accessible through the MLTxQuest portal for
315 all authorized users, thereby centralizing and simplifying access to critical information
316 within the organization.
317 tags_list_text : metadata, mltxquest, business
318 link_list_text :
319 likes_list_text : luiz.lagatosm@abc-mixa.com

```

```

314     category_text : Governance
315
316 Entity ID 259
317
318 Modified Date : 2024-06-27T11:36:39.594Z
319 title_text : MLTxQuest - Governance Badge
320 description_text : The Governance Badge in MLTxQuest is awarded to data assets (tables) that
321 demonstrate exceptional metadata management and data quality. To earn this badge, tables must
322 meet stringent criteria, including robust technical and business metadata descriptions,
323 alongside maintaining a Fitness Index score above 90 consistently. This badge signifies a
324 commitment to high data governance standards, providing users with confidence in data
325 accuracy and transparency in its usage.
326 tags_list_text : badge, governance, metadata
327 link_list_text :
328 likes_list_text : luiz.lagatosm@abc-mixa.com
329 category_text : Governance
330
331 2 text entities found.
332 Completed task: -i 107 259
333
334 Command menu:
335
336 -q      : print last non-command prompt
337 -x      : print sample queries
338 -p key value : set frontendParams[key] = value
339 -f      : use catch-all parameter set for debugging
340 -d      : use default parameter set
341 -v      : view parameter set
342 -a multitoken : add multitoken to 'ignore' list
343 -r multitoken : remove multitoken from 'ignore' list
344 -l      : view 'ignore' list
345 -i ID1 ID2 ... : print content of text entities ID1 ID2 ...
346 -s      : print size of core backend tables
347 -c F1 F2 ... : show sections F1 F2 ... in output results
348
349 To view available sections for -c command, enter -v command.
350 To view available keys for -p command, enter -v command.
351 For -i command, choose IDs from list shown in prompt results.
352 For standard prompts, enter text not starting with '-' or digit.
353
354 Query, command, or integer in [0, 7] for sample query: -a detailed
355 Completed task: -a detailed
356
357 Command menu:
358
359 -q      : print last non-command prompt
360 -x      : print sample queries
361 -p key value : set frontendParams[key] = value
362 -f      : use catch-all parameter set for debugging
363 -d      : use default parameter set
364 -v      : view parameter set
365 -a multitoken : add multitoken to 'ignore' list
366 -r multitoken : remove multitoken from 'ignore' list
367 -l      : view 'ignore' list
368 -i ID1 ID2 ... : print content of text entities ID1 ID2 ...
369 -s      : print size of core backend tables
370 -c F1 F2 ... : show sections F1 F2 ... in output results
371
372 To view available sections for -c command, enter -v command.
373 To view available keys for -p command, enter -v command.
374 For -i command, choose IDs from list shown in prompt results.
375 For standard prompts, enter text not starting with '-' or digit.
376
377 Query, command, or integer in [0, 7] for sample query: -v
378
379 Key Description      Value
380
381 0 embeddingKeyMinSize 1
382 1 embeddingValuesMinSize 2
383 2 min_pmi           0.0
384 3 nABmin            1
385 4 Customized_pmi    True
386 5 ContextMultitokenMinSize 1
387 6 minOutputListSize 1
388 7 bypassIgnoreList   False
389 8 ignoreList         ('data', 'detailed')

```

```

385     9 maxTokenCount      100
386
387 Show sections:
388
389     Embeddings True
390     Category True
391     Tags      True
392     Titles    True
393     Descr.   False
394     Whole    False
395     ID       True
396     Agents   True
397
398 Completed task: -v
399
400 Command menu:
401
402     -q      : print last non-command prompt
403     -x      : print sample queries
404     -p key value : set frontendParams[key] = value
405     -f      : use catch-all parameter set for debugging
406     -d      : use default parameter set
407     -v      : view parameter set
408     -a multitoken : add multitoken to 'ignore' list
409     -r multitoken : remove multitoken from 'ignore' list
410     -l      : view 'ignore' list
411     -i ID1 ID2 ... : print content of text entities ID1 ID2 ...
412     -s      : print size of core backend tables
413     -c F1 F2 ... : show sections F1 F2 ... in output results
414
415 To view available sections for -c command, enter -v command.
416 To view available keys for -p command, enter -v command.
417 For -i command, choose IDs from list shown in prompt results.
418 For standard prompts, enter text not starting with '-' or digit.
419
420 Query, command, or integer in [0, 7] for sample query: -p 6 2
421 Completed task: -p 6 2
422
423 Command menu:
424
425     -q      : print last non-command prompt
426     -x      : print sample queries
427     -p key value : set frontendParams[key] = value
428     -f      : use catch-all parameter set for debugging
429     -d      : use default parameter set
430     -v      : view parameter set
431     -a multitoken : add multitoken to 'ignore' list
432     -r multitoken : remove multitoken from 'ignore' list
433     -l      : view 'ignore' list
434     -i ID1 ID2 ... : print content of text entities ID1 ID2 ...
435     -s      : print size of core backend tables
436     -c F1 F2 ... : show sections F1 F2 ... in output results
437
438 To view available sections for -c command, enter -v command.
439 To view available keys for -p command, enter -v command.
440 For -i command, choose IDs from list shown in prompt results.
441 For standard prompts, enter text not starting with '-' or digit.
442
443 Query, command, or integer in [0, 7] for sample query: -p 2 0.50
444 Completed task: -p 2 0.50
445
446 Command menu:
447
448     -q      : print last non-command prompt
449     -x      : print sample queries
450     -p key value : set frontendParams[key] = value
451     -f      : use catch-all parameter set for debugging
452     -d      : use default parameter set
453     -v      : view parameter set
454     -a multitoken : add multitoken to 'ignore' list
455     -r multitoken : remove multitoken from 'ignore' list
456     -l      : view 'ignore' list
457     -i ID1 ID2 ... : print content of text entities ID1 ID2 ...
458     -s      : print size of core backend tables
459     -c F1 F2 ... : show sections F1 F2 ... in output results
460

```

```

461 To view available sections for -c command, enter -v command.
462 To view available keys for -p command, enter -v command.
463 For -i command, choose IDs from list shown in prompt results.
464 For standard prompts, enter text not starting with '-' or digit.
465
466 Query, command, or integer in [0, 7] for sample query: 6
467 query: MLTxQuest Data Assets Detailed Information page
468
469 N pmi F token [from embeddings] word [from prompt]
470
471 1 1.00 * confidentiality|availability information|assets
472 1 1.00 * availability|organization information|assets
473 1 1.00 * confidentiality|availability|organization information|assets
474 1 1.00 - availability|organization|information information|assets
475 1 1.00 * integrity|confidentiality|availability information|assets
476 1 1.00 - organization|information information|assets
477 1 1.00 - organization|information|assets information|assets
478 1 1.00 * systems|managed information|assets
479 1 1.00 * managed|mltxdat information|assets
480 1 1.00 * systems|managed|mltxdat information|assets
481 1 1.00 - managed|mltxdat|csa information|assets
482 1 1.00 - platform|against information|assets
483 1 1.00 * platform|against|threats information|assets
484 1 1.00 * threats|such information|assets
485 1 1.00 * data|systems|managed information|assets
486 1 1.00 - csa|platform|against information|assets
487 1 1.00 * against|threats information|assets
488 1 1.00 * against|threats|such information|assets
489 1 0.71 * navigating|data page|mltxquest
490 1 0.71 * efficiently|navigating|data page|mltxquest
491 1 0.71 * navigating|data|assets page|mltxquest
492 1 0.71 - assets|page page|mltxquest
493 1 0.71 - data|assets|page page|mltxquest
494 1 0.71 - page|mltxquest|while page|mltxquest
495 1 0.71 * while|facilitating page|mltxquest
496 1 0.71 * while|facilitating|comprehensive page|mltxquest
497 1 0.71 - assets|page|mltxquest page|mltxquest
498 1 0.71 - mltxquest|while page|mltxquest
499 1 0.71 * mltxquest|while|facilitating page|mltxquest
500 1 0.71 * facilitating|comprehensive page|mltxquest
501 1 0.71 - assets|data page|mltxquest
502 1 0.71 - information|page page|mltxquest
503 1 0.71 - page|mltxquest|data page|mltxquest
504 1 0.71 - information|page|mltxquest page|mltxquest
505 1 0.71 - mltxquest|data page|mltxquest
506 1 0.71 * mltxquest|data|assets page|mltxquest
507 1 0.71 * assets|users page|mltxquest
508 1 0.71 * data|assets|users page|mltxquest
509 1 0.71 - mltxdat|csa|platform information|assets
510 1 0.71 - csa|platform information|assets
511 2 0.67 * users|efficiently data|assets
512 2 0.67 * efficiently|navigating data|assets
513 2 0.67 * users|efficiently|navigating data|assets
514 2 0.67 * aid|users|efficiently data|assets
515
516 N = occurrences of (token, word) in corpus. F = * if contextual pair.
517 If no result, try option -p f.
518
519 SECTION: Category
520
521 Category: 'Products' [5 entries]
522 Linked to: page|mltxquest (2)
523 Linked to: information|page|mltxquest|data (1)
524 Linked to: data|assets (9)
525 Linked to: data|assets|page|mltxquest (1)
526 Linked to: page|mltxquest|data|assets (1)
527
528 Category: 'Governance' [2 entries]
529 Linked to: information|assets (1)
530 Linked to: data|assets (9)
531
532 SECTION: Tags
533
534 Tags: MLTxQuest [5 entries]
535 Linked to: page|mltxquest (2)
536 Linked to: information|page|mltxquest|data (1)

```

```

537     Linked to: data|assets (9)
538     Linked to: data|assets|page|mltxquest (1)
539     Linked to: page|mltxquest|data|assets (1)
540
541     Tags: Guideline [3 entries]
542     Linked to: page|mltxquest (2)
543     Linked to: data|assets (9)
544     Linked to: data|assets|page|mltxquest (1)
545
546     Tags: Guidelines [4 entries]
547     Linked to: page|mltxquest (2)
548     Linked to: information|page|mltxquest|data (1)
549     Linked to: data|assets (9)
550     Linked to: page|mltxquest|data|assets (1)
551
552 SECTION: Titles
553
554     Titles: 'MLTxQuest - Data Assets' [3 entries]
555     Linked to: page|mltxquest (2)
556     Linked to: data|assets (9)
557     Linked to: data|assets|page|mltxquest (1)
558
559     Titles: 'MLTxQuest-Data Asset Deta' [4 entries]
560     Linked to: page|mltxquest (2)
561     Linked to: information|page|mltxquest|data (1)
562     Linked to: data|assets (9)
563     Linked to: page|mltxquest|data|assets (1)
564
565 SECTION: Entity IDs
566
567     ID: 91 [3 entries]
568     Linked to: page|mltxquest (2)
569     Linked to: data|assets (9)
570     Linked to: data|assets|page|mltxquest (1)
571
572     ID: 92 [4 entries]
573     Linked to: page|mltxquest (2)
574     Linked to: information|page|mltxquest|data (1)
575     Linked to: data|assets (9)
576     Linked to: page|mltxquest|data|assets (1)
577
578 SECTION: Agents
579
580     Agents: Policy [2 entries]
581     Linked to: information|assets (1)
582     Linked to: data|assets (9)
583
584     Agents: Governance [2 entries]
585     Linked to: information|assets (1)
586     Linked to: data|assets (9)
587
588 Above results based on words found in prompt, matched back to backend tables.
589 Numbers in parentheses are occurrences of word in corpus.
590
591 SECTION: (Agent, Multitoken) --> (ID list)
592
593     empty unless labels 'ID' and 'Agents' are in 'show'.


---


595 Command menu:
596
597     -q      : print last non-command prompt
598     -x      : print sample queries
599     -p key value : set frontendParams[key] = value
600     -f      : use catch-all parameter set for debugging
601     -d      : use default parameter set
602     -v      : view parameter set
603     -a multitoken : add multitoken to 'ignore' list
604     -r multitoken : remove multitoken from 'ignore' list
605     -l      : view 'ignore' list
606     -i ID1 ID2 ... : print content of text entities ID1 ID2 ...
607     -s      : print size of core backend tables
608     -c F1 F2 ... : show sections F1 F2 ... in output results
609
610 To view available sections for -c command, enter -v command.
611 To view available keys for -p command, enter -v command.
612 For -i command, choose IDs from list shown in prompt results.

```

```

613 For standard prompts, enter text not starting with '-' or digit.
614
615 Query, command, or integer in [0, 7] for sample query: -c Titles
616 Completed task: -c Titles
617
618 Command menu:
619
620 -q : print last non-command prompt
621 -x : print sample queries
622 -p key value : set frontendParams[key] = value
623 -f : use catch-all parameter set for debugging
624 -d : use default parameter set
625 -v : view parameter set
626 -a multitoken : add multitoken to 'ignore' list
627 -r multitoken : remove multitoken from 'ignore' list
628 -l : view 'ignore' list
629 -i ID1 ID2 ... : print content of text entities ID1 ID2 ...
630 -s : print size of core backend tables
631 -c F1 F2 ... : show sections F1 F2 ... in output results
632
633 To view available sections for -c command, enter -v command.
634 To view available keys for -p command, enter -v command.
635 For -i command, choose IDs from list shown in prompt results.
636 For standard prompts, enter text not starting with '-' or digit.
637
638 Query, command, or integer in [0, 7] for sample query: 6
639 query: MLTxQuest Data Assets Detailed Information page
640
641 SECTION: Titles
642
643 Titles: 'MLTxQuest - Data Assets' [3 entries]
644 Linked to: page|mltxquest (2)
645 Linked to: data|assets (9)
646 Linked to: data|assets|page|mltxquest (1)
647
648 Titles: 'MLTxQuest-Data Asset Deta' [4 entries]
649 Linked to: page|mltxquest (2)
650 Linked to: information|page|mltxquest|data (1)
651 Linked to: data|assets (9)
652 Linked to: page|mltxquest|data|assets (1)
653
654 Above results based on words found in prompt, matched back to backend tables.
655 Numbers in parentheses are occurrences of word in corpus.
656
657 SECTION: (Agent, Multitoken) --> (ID list)
658
659 empty unless labels 'ID' and 'Agents' are in 'show'.
660
661 Command menu:
662
663 -q : print last non-command prompt
664 -x : print sample queries
665 -p key value : set frontendParams[key] = value
666 -f : use catch-all parameter set for debugging
667 -d : use default parameter set
668 -v : view parameter set
669 -a multitoken : add multitoken to 'ignore' list
670 -r multitoken : remove multitoken from 'ignore' list
671 -l : view 'ignore' list
672 -i ID1 ID2 ... : print content of text entities ID1 ID2 ...
673 -s : print size of core backend tables
674 -c F1 F2 ... : show sections F1 F2 ... in output results
675
676 To view available sections for -c command, enter -v command.
677 To view available keys for -p command, enter -v command.
678 For -i command, choose IDs from list shown in prompt results.
679 For standard prompts, enter text not starting with '-' or digit.
680
681 Query, command, or integer in [0, 7] for sample query:
682

```

3.3 Web API for enterprise xLLM

A web API is available [here](#) on `xllm.GenAItechLab.com` to test the application. The implementation is slightly different from the offline version. But it is based on the same anonymized corporate corpus, dealing with ML and AI policies, integration, definitions, best practices, and references, for corporate users (employees). See how it looks like in Figure 3.2.

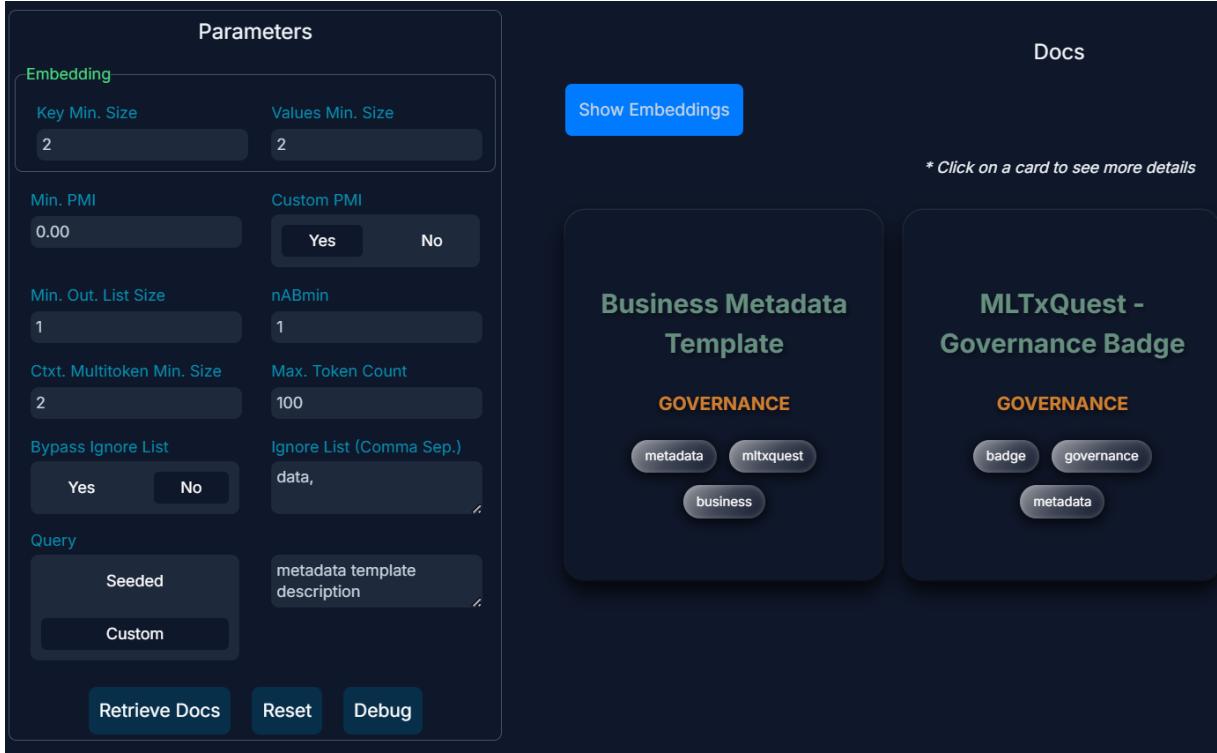


Figure 3.2: Web API for enterprise xLLM, with prompt results for ‘metadata template description’

3.3.1 Left panel: command menu and prompt box

The left panel allows you to fine-tune the front-end parameters in real time, and to enter your prompt at the bottom: either from pre-selected queries with the option `Seeded`, or your own prompt with the option `Custom`. The right panel shows the prompt results. The front-end parameters are the same as in the offline version (see Figure 3.2) except show options that are organized differently, customizable on the right panel.

Initially, the left panel shows no result. After entering any prompt, click on `Retrieve Docs` to display the results. Before trying any new prompt (except the first one), I recommend to click on the `Reset` button at the bottom: it resets the parameters to the default values. The `Debugging` option sets parameters to extreme values that allow you to retrieve everything xLLM is able to find. But the prompt results on the right side can be voluminous. However, it is useful to understand if missing items in the results are due to a glitch, or due to choosing specific parameter values that eliminate some output. In the next version, a `relevancy score` will be attached to each returned item in the prompt results. You will be able to display (say) only the top 10 items, based on score. The user will be able to choose the maximum number of items to display in the results. The score (currently hidden) and the results depend on the parameters.

Finally, parameter values can be modified individually using the top 10 boxes on the left panel, offering custom results and real-time fine-tuning. Lower and upper bounds are specified for each parameter.

3.3.2 Right panel: prompt results

The right panel displays prompt results. Each box represents one item - a `text entity` - called “card” on the UI, and retrieved from the `backend tables` based on its relevancy to the user prompt. See glossary for details.

In our example, two items were retrieved, respectively ‘Business Metadata Template’ and ‘MLTxQuest Governance Badge’. For each item, the green, orange and white fonts represent respectively the title, category, and related tags. If you click on any item, more details show up: see Figure 3.3. You can expand to retrieve

the full raw text: in this case, a JSON entry in the corpus (not shown by default). Also note the **text** entity ID to match back to the corpus, as well as triggered **agents**, at the top in Figure 3.3.



Figure 3.3: First item returned: details

N	PMI	F	Token [from embeddings]	Word [from prompt]
1	1.00		mltxquest~business	metadata~templates
1	1.00	*	templates~accurately	metadata~templates
1	1.00		out~templates	metadata~templates
1	1.00	*	templates~users	metadata~templates
1	1.00	*	out~templates~users	metadata~templates
1	1.00	*	users~interpret	metadata~templates
1	1.00	*	templates~users~interpret	metadata~templates
1	1.00		reviewed~metadata	metadata~templates
1	1.00	*	metadata~uploaded	metadata~templates
1	1.00	*	reviewed~metadata~uploaded	metadata~templates
1	1.00	*	uploaded~mltxquest	metadata~templates
1	1.00	*	metadata~uploaded~mltxquest	metadata~templates
1	1.00		detailed~instructions	metadata~templates
1	1.00		instructions~completing	metadata~templates

Figure 3.4: Top embedding entries for ‘metadata template description’

Finally, you can check out embedding entries related to your prompt, by clicking on the Show Embeddings blue box visible in Figure 3.2. See top embedding entries in Figure 3.4, for ‘metadata template description’ using the default parameter set. The ‘word’ column shows multitokens extracted from the prompt, while the ‘token’ column represents multitokens from the backend tables, related to the ‘word’ in question. Multitokens flagged with a (*) are contextually related to the ‘word’ in question, instead of just based on immediate proximity. The PMI measures the strength of the association, while the leftmost column is another indicator of relevancy. The associations in question may come from different text entities, or from the knowledge graph itself in version 3. These embedding entries are useful to try additional prompts to refine your search, or for debugging purposes.

As a side note, you can try much longer prompts. I chose a short example here for illustration purposes. Prompts with 20 tokens may generate more voluminous output, in about the same amount of time (no perceptible latency).

3.3.3 Next steps

The following features will be added:

- Incorporation of acronyms in the KW_map table, for instance to redirect ‘Doing Business As’ to ‘DBA’ if the former is found in a prompt, but not in the corpus.
- A second dictionary table (or alternate mechanism) for multitokens found in knowledge graph entities: categories, titles, tags, agents, and so on. The end goal is to boost these multitokens, as they have more importance and are of higher quality. In the end, to produce better relevancy scores.
- Working with contextual multitokens, consisting of non-adjacent words found together in a same text sub-entity.
- Data augmentation and more agents, with fewer text entities lacking agents.
- Breaking prompts into sub-prompts. More NLP: stemming, auto-correct, and so on.

3.4 Conclusions and references

My custom sub-LLM designed from scratch does not rely on any Python library or API, and performs better than search tools available on the market, in terms of speed and results relevancy. It offers the user the ability to fine-tune parameters in real time, and can detect user intent to deliver appropriate output. The good performance comes from the quality of the well structured input sources, combined with smart crawling to retrieve the embedded knowledge graph and integrate it in the backend tables. Traditional tools rely mostly on tokens, embeddings, billions of parameters and frontend tricks such as prompt engineering to fix backend issues.

To the contrary, my approach focuses on building a solid backend foundational architecture from the ground up. Tokens and embeddings are not the most important components, by a long shot. Cosine similarity and dot products are replaced by pointwise mutual information. There is no neural network, no training, and a small number of explainable parameters, easy to fine-tune. When you think about it, the average human being has a vocabulary of 30,000 words. Even if you added variations and other pieces of information (typos, plural, grammatical tenses, product IDs, street names, and so on), you end up with a few millions at most, not trillions. Indeed, in expensive multi-billion systems, most tokens and weights are just noise: most are rarely fetched to serve an answer. This noise is a source of hallucinations.

Finally, gather a large number of user queries even before your start designing your architecture, and add prompt elements into your backend tables, as a source of data augmentation. It contributes to enhancing the quality of your system. For additional references, see [19] on mixture of experts, [6] on multitokens, [25, 27] on LLM evaluation, [29] on building your LLM from scratch, [3] on reducing LLM costs, and [5] on variable length embeddings.

Chapter 4

Appendix

4.1 Python code

The Python code is also on GitHub, [here](#), along with the crawled input source and backend tables. The enterprise corpus shared on GitHub – actually, a small portion corresponding to the AI section – is fully anonymized.

```
1  #--- [1] Backend: functions
2
3  def update_hash(hash, key, count=1):
4
5      if key in hash:
6          hash[key] += count
7      else:
8          hash[key] = count
9      return(hash)
10
11
12 def update_nestedHash(hash, key, value, count=1):
13
14     # 'key' is a word here, value is tuple or single value
15     if key in hash:
16         local_hash = hash[key]
17     else:
18         local_hash = {}
19     if type(value) is not tuple:
20         value = (value,)
21     for item in value:
22         if item in local_hash:
23             local_hash[item] += count
24         else:
25             local_hash[item] = count
26     hash[key] = local_hash
27     return(hash)
28
29
30 def get_value(key, hash):
31     if key in hash:
32         value = hash[key]
33     else:
34         value = ''
35     return(value)
36
37
38 def update_tables(backendTables, word, hash_crawl, backendParams):
39
40     category = get_value('category', hash_crawl)
41     tag_list = get_value('tag_list', hash_crawl)
42     title = get_value('title', hash_crawl)
43     description = get_value('description', hash_crawl) #
44     meta = get_value('meta', hash_crawl)
45     ID = get_value('ID', hash_crawl)
46     agents = get_value('agents', hash_crawl)
47     full_content = get_value('full_content', hash_crawl) #
48
49     extraWeights = backendParams['extraWeights']
50     word = word.lower() # add stemming
```

```

51     weight = 1.0
52     if word in category:
53         weight += extraWeights['category']
54     if word in tag_list:
55         weight += extraWeights['tag_list']
56     if word in title:
57         weight += extraWeights['title']
58     if word in meta:
59         weight += extraWeights['meta']
60
61     update_hash(backendTables['dictionary'], word, weight)
62     update_nestedHash(backendTables['hash_context1'], word, category)
63     update_nestedHash(backendTables['hash_context2'], word, tag_list)
64     update_nestedHash(backendTables['hash_context3'], word, title)
65     update_nestedHash(backendTables['hash_context4'], word, description) # takes space, don't build?
66     update_nestedHash(backendTables['hash_context5'], word, meta)
67     update_nestedHash(backendTables['hash_ID'], word, ID)
68     update_nestedHash(backendTables['hash_agents'], word, agents)
69     for agent in agents:
70         update_nestedHash(backendTables['ID_to_agents'], ID, agent)
71     update_nestedHash(backendTables['full_content'], word, full_content) # takes space, don't build?
72     update_nestedHash(backendTables['ID_to_content'], ID, full_content)
73
74     return(backendTables)
75
76
77 def clean_list(value):
78
79     # change string "['a', 'b', ...]" to ('a', 'b', ...)
80     value = value.replace("[", "").replace("]", "")
81     aux = value.split("~")
82     value_list = ()
83     for val in aux:
84         val = val.replace("'", "").replace('"', "").lstrip()
85         if val != '':
86             value_list = (*value_list, val)
87     return(value_list)
88
89
90 def get_key_value_pairs(entity):
91
92     # extract key-value pairs from 'entity' (a string)
93     entity = entity[1].replace("}", "", "")
94     flag = False
95     entity2 = ""
96
97     for idx in range(len(entity)):
98         if entity[idx] == '[':
99             flag = True
100        elif entity[idx] == ']':
101            flag = False
102            if flag and entity[idx] == ",":
103                entity2 += "~"
104            else:
105                entity2 += entity[idx]
106
107    entity = entity2
108    key_value_pairs = entity.split(", ")
109    return(key_value_pairs)
110
111
112 def update_dict(backendTables, hash_crawl, backendParams):
113
114     max_multitoken = backendParams['max_multitoken']
115     maxDist = backendParams['maxDist']
116     maxTerms = backendParams['maxTerms']
117
118     category = get_value('category', hash_crawl)
119     tag_list = get_value('tag_list', hash_crawl)
120     title = get_value('title', hash_crawl)
121     description = get_value('description', hash_crawl)
122     meta = get_value('meta', hash_crawl)
123
124     text = category + "." + str(tag_list) + "." + title + "." + description + "." + meta
125     text = text.replace('/', " ").replace('(', ' ').replace(')', ' ').replace('?', ' ')
126     text = text.replace('"', "").replace("'", "").replace('\n', '').replace('!', '')

```

```

127     text = text.replace("\s",'').replace("\t",'').replace(","," ").replace(":", " ")
128     text = text.lower()
129     sentence_separators = ('.',)
130     for sep in sentence_separators:
131         text = text.replace(sep, '_~')
132     text = text.split('_~')
133
134     hash_pairs = backendTables['hash_pairs']
135     ctokens = backendTables['ctokens']
136     KW_map = backendTables['KW_map']
137     stopwords = backendTables['stopwords']
138     hwords = {} # local word hash with word position, to update hash_pairs
139
140     for sentence in text:
141
142         words = sentence.split(" ")
143         position = 0
144         buffer = []
145
146         for word in words:
147
148             if word in KW_map:
149                 word = KW_map[word]
150
151             if word not in stopwords:
152                 # word is single token
153                 buffer.append(word)
154                 key = (word, position)
155                 update_hash(hwords, key) # for word correlation table (hash_pairs)
156                 update_tables(backendTables, word, hash_crawl, backendParams)
157
158                 for k in range(1, max_multitoken):
159                     if position > k:
160                         # word is now multi-token with k+1 tokens
161                         word = buffer[position-k] + "~" + word
162                         key = (word, position)
163                         update_hash(hwords, key) # for word correlation table (hash_pairs)
164                         update_tables(backendTables, word, hash_crawl, backendParams)
165
166                     position +=1
167
168     for keyA in hwords:
169         for keyB in hwords:
170
171             wordA = keyA[0]
172             positionA = keyA[1]
173             n_termsA = len(wordA.split("~"))
174
175             wordB = keyB[0]
176             positionB = keyB[1]
177             n_termsB = len(wordB.split("~"))
178
179             key = (wordA, wordB)
180             n_termsAB = max(n_termsA, n_termsB)
181             distanceAB = abs(positionA - positionB)
182
183             if wordA < wordB and distanceAB <= maxDist and n_termsAB <= maxTerms:
184                 hash_pairs = update_hash(hash_pairs, key)
185                 if distanceAB > 1:
186                     ctokens = update_hash(ctokens, key)
187
188     return(backendTables)
189
190
191 #--- [2] Backend: main (create backend tables based on crawled corpus)
192
193 tableName = (
194     'dictionary', # multitokens (key = multitoken)
195     'hash_pairs', # multitoken associations (key = pairs of multitokens)
196     'ctokens', # not adjacent pairs in hash_pairs (key = pairs of multitokens)
197     'hash_context1', # categories (key = multitoken)
198     'hash_context2', # tags (key = multitoken)
199     'hash_context3', # titles (key = multitoken)
200     'hash_context4', # descriptions (key = multitoken)
201     'hash_context5', # meta (key = multitoken)
202     'hash_ID', # text entity ID table (key = multitoken, value is list of IDs)

```

```

203 'hash_agents', # agents (key = multitoken)
204 'full_content', # full content (key = multitoken)
205 'ID_to_content', # full content attached to text entity ID (key = text entity ID)
206 'ID_to_agents', # map text entity ID to agents list (key = text entity ID)
207 'ID_size', # content size (key = text entity ID)
208 'KW_map', # for singularization, map kw to single-token dictionary entry
209 'stopwords', # stopword list
210 )
211
212 backendTables = {}
213 for name in tableNames:
214     backendTables[name] = {}
215
216 stopwords = ('', '-', 'in', 'the', 'and', 'to', 'of', 'a', 'this', 'for', 'is', 'with', 'from',
217             'as', 'on', 'an', 'that', 'it', 'are', 'within', 'will', 'by', 'or', 'its', 'can',
218             'your', 'be', 'about', 'used', 'our', 'their', 'you', 'into', 'using', 'these',
219             'which', 'we', 'how', 'see', 'below', 'all', 'use', 'across', 'provide', 'provides',
220             'aims', 'one', '&', 'ensuring', 'crucial', 'at', 'various', 'through', 'find', 'ensure',
221             'more', 'another', 'but', 'should', 'considered', 'provided', 'must', 'whether',
222             'located', 'where', 'begins', 'any')
223 backendTables['stopwords'] = stopwords
224
225 # agent_map works, but hash structure should be improved
226 # key is word, value is agent (many-to-one). Allow for many-to-many
227 agent_map = {
228     'template':'Template',
229     'policy':'Policy',
230     'governance':'Governance',
231     'documentation':'Documentation',
232     'best practice':'Best Practices',
233     'bestpractice':'Best Practices',
234     'standard':'Standards',
235     'naming':'Naming',
236     'glossary':'Glossary',
237     'historical data':'Data',
238     'overview':'Overview',
239     'training':'Training',
240     'genai':'GenAI',
241     'gen ai':'GenAI',
242     'example':'Example',
243     'example1':'Example',
244     'example2':'Example',
245 }
246
247 KW_map = {}
248 try:
249     IN = open("KW_map.txt", "r")
250 except:
251     print("KW_map.txt not found on first run: working with empty KW_map.")
252     print("KW_map.txt will be created after exiting if save = True.")
253 else:
254     content = IN.read()
255     pairs = content.split('\n')
256     for pair in pairs:
257         pair = pair.split('\t')
258         key = pair[0]
259         if len(pair) > 1:
260             KW_map[key] = pair[1]
261     IN.close()
262 backendTables['KW_map'] = KW_map
263
264 backendParams = {
265     'max_multitoken': 4, # max. consecutive terms per multi-token for inclusion in dictionary
266     'maxDist' : 3, # max. position delta between 2 multitokens to link them in hash_pairs
267     'maxTerms': 3, # maxTerms must be <= max_multitoken
268     'extraWeights' : # deafault weight is 1
269     {
270         'description': 0.0,
271         'category': 0.3,
272         'tag_list': 0.4,
273         'title': 0.2,
274         'meta': 0.1
275     }
276 }
277
278

```

```

279 local = True # first time run, set to False
280 if local:
281     # get repository from local file
282     IN = open("repository.txt","r")
283     data = IN.read()
284     IN.close()
285 else:
286     # get anonymized repository from GitHub url
287     import requests
288     url = "https://mltblog.com/3y8MXq5"
289     response = requests.get(url)
290     data = response.text
291
292 entities = data.split("\n")
293 ID_size = backendTables['ID_size']
294
295 # to avoid duplicate entities (takes space, better to remove them in the corpus)
296 entity_list = ()
297
298 for entity_raw in entities:
299
300     entity = entity_raw.split("~~")
301     agent_list = ()
302
303     if len(entity) > 1 and entity[1] not in entity_list:
304
305         entity_list = (*entity_list, entity[1])
306         entity_ID = int(entity[0])
307         entity = entity[1].split("{")
308         hash_crawl = {}
309         hash_crawl['ID'] = entity_ID
310         ID_size[entity_ID] = len(entity[1])
311         hash_crawl['full_content'] = entity_raw # do not build to save space
312
313         key_value_pairs = get_key_value_pairs(entity)
314
315         for pair in key_value_pairs:
316
317             if ":" in pair:
318                 key, value = pair.split(": ", 1)
319                 key = key.replace("'", "")
320                 if key == 'category_text':
321                     hash_crawl['category'] = value
322                 elif key == 'tags_list_text':
323                     hash_crawl['tag_list'] = clean_list(value)
324                 elif key == 'title_text':
325                     hash_crawl['title'] = value
326                 elif key == 'description_text':
327                     hash_crawl['description'] = value # do not build to save space
328                 elif key == 'tower_option_tower':
329                     hash_crawl['meta'] = value
330                 if key in ('category_text', 'tags_list_text', 'title_text'):
331                     for word in agent_map:
332                         if word in value.lower():
333                             agent = agent_map[word]
334                             if agent not in agent_list:
335                                 agent_list = (*agent_list, agent)
336
337             hash_crawl['agents'] = agent_list
338             update_dict(backendTables, hash_crawl, backendParams)
339
340 # [2.1] Create embeddings
341
342 embeddings = {} # multitoken embeddings based on hash_pairs
343
344 hash_pairs = backendTables['hash_pairs']
345 dictionary = backendTables['dictionary']
346
347 for key in hash_pairs:
348     wordA = key[0]
349     wordB = key[1]
350     nA = dictionary[wordA]
351     nB = dictionary[wordB]
352     nAB = hash_pairs[key]
353     pmi = nAB/(nA*nB)**0.5 # try: nAB/(nA + nB - nAB)
354     # if nA + nB <= nAB:

```

```

355     #   print(key, nA, nB, nAB)
356     update_nestedHash(embeddings, wordA, wordB, pmi)
357     update_nestedHash(embeddings, wordB, wordA, pmi)
358
359
360 # [2.2] Create sorted n-grams
361
362 sorted_ngrams = {} # to match ngram prompts with embeddings entries
363
364 for word in dictionary:
365     tokens = word.split('`')
366     tokens.sort()
367     sorted_ngram = tokens[0]
368     for token in tokens[1:len(tokens)]:
369         sorted_ngram += "`" + token
370     update_nestedHash(sorted_ngrams, sorted_ngram, word)
371
372 # print top multitokens: useful to build agents, along with sample prompts
373 # for key in dictionary:
374 #     if dictionary[key] > 20:
375 #         print(key, dictionary[key])
376
377
378 #--- [3] Frontend: functions
379
380 # [3.1] custom pmi
381
382 def custom_pmi(word, token, backendTables):
383
384     dictionary = backendTables['dictionary']
385     hash_pairs = backendTables['hash_pairs']
386
387     nAB = 0
388     pmi = 0.00
389     keyAB = (word, token)
390     if word > token:
391         keyAB = (token, word)
392     if keyAB in hash_pairs:
393         nAB = hash_pairs[keyAB]
394         nA = dictionary[word]
395         nB = dictionary[token]
396         pmi = nAB/(nA*nB)**0.5
397     return(pmi)
398
399 # [3.2] update frontend params
400
401 def cprint(ID, entity):
402     # print text_entity (a JSON text string) nicely
403
404     print(" --- Entity %d ---\n" %(ID))
405     keys = (
406         'title_text',
407         'description_text',
408         'tags_list_text',
409         'category_text',
410         'likes_list_text',
411         'link_list_text',
412         'Modified Date',
413     )
414     entity = str(entity).split("``")
415     entity = entity[1].split("{")
416     key_value_pairs = get_key_value_pairs(entity)
417
418     for pair in key_value_pairs:
419         if ":" in pair:
420             key, value = pair.split(": ", 1)
421             key = key.replace("//", "")
422             if key in keys:
423                 print("> ",key,":")
424                 value = value.replace("//",',').split("`")
425                 for item in value:
426                     item = item.lstrip().replace("[", "").replace("]", "")
427                     print(item)
428             print()
429     return()
430

```

```

431 def update_params(option, saved_query, sample_queries, frontendParams, backendTables):
432
433     arr = []
434     ID_to_content = backendTables['ID_to_content']
435     for param in frontendParams:
436         arr.append(param)
437     task = option
438     print()
439
440     if option == '-l':
441         print("Multitoken ignore list:\n", frontendParams['ignoreList'])
442
443     elif option == '-v':
444         print("%3s %s %s\n" %('Key', 'Description'.ljust(25), 'Value'))
445         for key in range(len(arr)):
446             param = arr[key]
447             value = frontendParams[param]
448             if param != 'show':
449                 print("%3d %s %s" %(key, param.ljust(25), value))
450             else:
451                 print("\nShow sections:\n")
452                 for section in value:
453                     print(" %s %s" %(section.ljust(10), value[section]))
454
455     elif option == '-f':
456         # use parameter set to show as much as possible
457         for param in frontendParams:
458             if param == 'ignoreList':
459                 frontendParams[param] = ()
460             elif param == 'Customized_pmi':
461                 # use customized pmi
462                 frontendParams[param] = True
463             elif param == 'show':
464                 showHash = frontendParams[param]
465                 for section in showHash:
466                     # show all sections in output results
467                     showHash[section] = True
468             elif param == 'maxTokenCount':
469                 frontendParams[param] = 999999999
470             else:
471                 frontendParams[param] = 0
472
473     elif option == '-d':
474         frontendParams = default_frontendParams()
475
476     elif '-p' in option:
477         option = option.split(' ')
478         if len(option) == 3:
479             paramID = int(option[1])
480             if paramID < len(arr):
481                 param = arr[paramID]
482                 value = option[2]
483                 if value == 'True':
484                     value = True
485                 elif value == 'False':
486                     value = False
487                 else:
488                     value = float(option[2])
489                     frontendParams[param] = value
490             else:
491                 print("Error 101: key outside range")
492         else:
493             print("Error 102: wrong number of arguments")
494
495     elif '-a' in option:
496         option = option.split(' ')
497         if len(option) == 2:
498             ignore = frontendParams['ignoreList']
499             ignore = (*ignore, option[1])
500             frontendParams['ignoreList'] = ignore
501         else:
502             print("Error 103: wrong number of arguments")
503
504     elif '-r' in option:
505         option = option.split(' ')
506         if len(option) == 2:

```

```

507         ignore2 = ()
508         ignore = frontendParams['ignoreList']
509         for item in ignore:
510             if item != option[1]:
511                 ignore2 = (*ignore2, item)
512         frontendParams['ignoreList'] = ignore2
513     else:
514         print("Error 104: wrong number of arguments")
515
516     elif '-i' in option:
517         option = option.split(' ')
518         nIDs = 0
519         for ID in option:
520             if ID.isdigit():
521                 ID = int(ID)
522                 # print content of text entity ID
523                 if ID in ID_to_content:
524                     cprint(ID, ID_to_content[ID])
525                     nIDs += 1
526         print("\n%d text entities found." % (nIDs))
527
528     elif option == '-s':
529         print("Size of some backend tables:")
530         print(" dictionary:", len(backendTables['dictionary']))
531         print(" pairs : ", len(backendTables['hash_pairs']))
532         print(" ctokens : ", len(backendTables['ctokens']))
533         print(" ID_size : ", len(backendTables['ID_size']))
534
535     elif '-c' in option:
536         show = frontendParams['show']
537         option = option.split(' ')
538         for section in show:
539             if section in option or '*' in option:
540                 show[section] = True
541             else:
542                 show[section] = False
543
544     elif option == '-q':
545         print("Saved query:", saved_query)
546
547     elif option == '-x':
548         print("Index Query\n")
549         for k in range(len(sample_queries)):
550             print(" %3d %s" %(k, sample_queries[k]))
551
552     print("\nCompleted task: %s" %(task))
553     return(frontendParams)
554
555 # [3.3] retrieve info and print results
556
557 def print_results(q_dictionary, q_embeddings, backendTables, frontendParams):
558
559     dictionary = backendTables['dictionary']
560     hash_pairs = backendTables['hash_pairs']
561     ctokens = backendTables['ctokens']
562     ID_to_agents = backendTables['ID_to_agents']
563     ID_size = backendTables['ID_size']
564     show = frontendParams['show']
565
566     if frontendParams['bypassIgnoreList'] == True:
567         # bypass 'ignore' list
568         ignore = ()
569     else:
570         # ignore multitokens specified in 'ignoreList'
571         ignore = frontendParams['ignoreList']
572
573     if show['Embeddings']:
574         # show results from embedding table
575
576         local_hash = {} # used to not show same token 2x (linked to 2 different words)
577         q_embeddings = dict(sorted(q_embeddings.items(),key=lambda item: item[1],reverse=True))
578         print()
579         print("%3s %s %1s %s %s"
580             %('N','pmi'.ljust(4),'F','token [from embeddings]'.ljust(35),
581                'word [from prompt]'.ljust(35)))
582         print()

```

```

583     for key in q_embeddings:
584
585         word = key[0]
586         token = key[1]
587         pmi = q_embeddings[key]
588         ntk1 = len(word.split('`'))
589         ntk2 = len(token.split('`'))
590         flag = " "
591         nAB = 0
592         keyAB = (word, token)
593
594
595         if word > token:
596             keyAB = (token, word)
597         if keyAB in hash_pairs:
598             nAB = hash_pairs[keyAB]
599         if keyAB in ctokens:
600             flag = '*'
601         if (ntk1 >= frontendParams['embeddingKeyMinSize'] and
602             ntk2 >= frontendParams['embeddingValuesMinSize'] and
603             pmi >= frontendParams['min_pmi'] and
604             nAB >= frontendParams['nABmin'] and
605             token not in local_hash and word not in ignore
606         ):
607             print("%3d %4.2f %1s %s %s"
608                  %(nAB,pmi,flag,token.ljust(35),word.ljust(35)))
609             local_hash[token] = 1 # token marked as displayed, won't be shown again
610
611         print()
612         print("N = occurrences of (token, word) in corpus. F = * if contextual pair.")
613         print("If no result, try option '-p f'.")
614         print()
615
616     sectionLabels = {
617         # map section label to corresponding backend table name
618         'Dict' : 'dictionary',
619         'Pairs': 'hash_pairs',
620         'Category': 'hash_context1',
621         'Tags' : 'hash_context2',
622         'Titles': 'hash_context3',
623         'Descr.': 'hash_context4',
624         'Meta' : 'hash_context5',
625         'ID' : 'hash_ID',
626         'Agents': 'hash_agents',
627         'Whole' : 'full_content',
628     }
629     local_hash = {}
630     agentAndWord_to_IDs = {}
631
632     for label in show:
633         # labels: 'Category','Tags','Titles','Descr.','ID','Whole','Agents','Embeddings'
634
635         if show[label] and label in sectionLabels:
636             # show results for section corresponding to label
637
638             tableName = sectionLabels[label]
639             table = backendTables[tableName]
640             local_hash = {}
641             print(">>> RESULTS - SECTION: %s\n" % (label))
642
643             for word in q_dictionary:
644
645                 ntk3 = len(word.split('`'))
646                 if word not in ignore and ntk3 >= frontendParams['ContextMultitokenMinSize']:
647                     content = table[word] # content is a hash
648                     count = int(dictionary[word])
649                     for item in content:
650                         update_nestedHash(local_hash, item, word, count)
651
652             for item in local_hash:
653
654                 hash2 = local_hash[item]
655                 if len(hash2) >= frontendParams['minOutputListSize']:
656                     print(" %s: %s [%d entries]" % (label, item, len(hash2)))
657                     for key in hash2:
658                         print(" Linked to: %s (%s)" %(key, hash2[key]))

```

```

659     if label == 'ID' and item in ID_to_agents:
660         # here item is a text entity ID
661         LocalAgentHash = ID_to_agents[item]
662         local_ID_list = ()
663         for ID in LocalAgentHash:
664             local_ID_list = (*local_ID_list, ID)
665         print(" Agents:", local_ID_list)
666         for agent in local_ID_list:
667             key3 = (agent, key) # key is a multitoken
668             update_nestedHash(agentAndWord_to_IDs, key3, item)
669
670         print()
671     print()
672
673 print("Above results based on words found in prompt, matched back to backend tables.")
674 print("Numbers in parentheses are occurrences of word in corpus.\n")
675
676 print("-----")
677 print(">>> RESULTS - SECTION: (Agent, Multitoken) --> (ID list)")
678 print(" empty unless labels 'ID' and 'Agents' are in 'show'.\n")
679 hash_size = {}
680 for key in sorted(agentAndWord_to_IDs):
681     ID_list = ()
682     for ID in agentAndWord_to_IDs[key]:
683         ID_list = (*ID_list, ID)
684         hash_size[ID] = ID_size[ID]
685     print(key, "-->", ID_list)
686 print("\n ID Size\n")
687 for ID in hash_size:
688     print("%4d %5d" %(ID, hash_size[ID]))
689
690 return()
691
692
693 #--- [4] Frontend: main (process prompt)
694
695 # [4.1] Set default parameters
696
697 def default_frontendParams():
698
699     frontendParams = {
700         'embeddingKeyMinSize': 1, # try 2
701         'embeddingValuesMinSize': 2,
702         'min_pmi': 0.00,
703         'nABmin': 1,
704         'Customized_pmi': True,
705         'ContextMultitokenMinSize': 1, # try 2
706         'minOutputListSize': 1,
707         'bypassIgnoreList': False,
708         'ignoreList': ('data',),
709         'maxTokenCount': 100, # ignore generic tokens if large enough
710         'show': {
711             # names of sections to display in output results
712             'Embeddings': True,
713             'Category': True,
714             'Tags': True,
715             'Titles': True,
716             'Descr.': False, # do not built to save space
717             'Whole': False, # do not build to save space
718             'ID': True,
719             'Agents': True,
720         }
721     }
722     return(frontendParams)
723
724 # [4.2] Purge function
725
726 def distill_frontendTables(q_dictionary, q_embeddings, frontendParams):
727     # purge q_dictionary then q_embeddings (frontend tables)
728
729     maxTokenCount = frontendParams['maxTokenCount']
730     local_hash = {}
731     for key in q_dictionary:
732         if q_dictionary[key] > maxTokenCount:
733             local_hash[key] = 1
734     for keyA in q_dictionary:

```

```

735     for keyB in q_dictionary:
736         nA = q_dictionary[keyA]
737         nB = q_dictionary[keyB]
738         if keyA != keyB:
739             if (keyA in keyB and nA == nB) or (keyA in keyB.split('`')):
740                 local_hash[keyA] = 1
741     for key in local_hash:
742         del q_dictionary[key]
743
744     local_hash = {}
745     for key in q_embeddings:
746         if key[0] not in q_dictionary:
747             local_hash[key] = 1
748     for key in local_hash:
749         del q_embeddings[key]
750
751     return(q_dictionary, q_embeddings)
752
753 # [4.3] Main
754
755 print("\n") #
756 input_ = ""
757 saved_query = ""
758 get_bin = lambda x, n: format(x, 'b').zfill(n)
759 frontendParams = default_frontendParams()
760 sample_queries = (
761     'parameterized datasets map tables sql server',
762     'data load templates importing data database data warehouse',
763     'pipeline extract data eventhub files',
764     'blob storage single parquet file adls gen2',
765     'eventhub files blob storage single parquet',
766     'parquet blob eventhub more files less storage single table',
767     'MLTxQuest Data Assets Detailed Information page',
768     'table asset',
769 )
770
771 while len(input_) > 0:
772
773     print()
774     print("-----")
775     print("Command menu:\n")
776     print(" -q      : print last non-command prompt")
777     print(" -x      : print sample queries")
778     print(" -p key value : set frontendParams[key] = value")
779     print(" -f      : use catch-all parameter set for debugging")
780     print(" -d      : use default parameter set")
781     print(" -v      : view parameter set")
782     print(" -a multitoken : add multitoken to 'ignore' list")
783     print(" -r multitoken : remove multitoken from 'ignore' list")
784     print(" -l      : view 'ignore' list")
785     print(" -i ID1 ID2 ... : print content of text entities ID1 ID2 ...")
786     print(" -s      : print size of core backend tables")
787     print(" -c F1 F2 ... : show sections F1 F2 ... in output results")
788     print("\nTo view available sections for -c command, enter -v command.")
789     print("To view available keys for -p command, enter -v command.")
790     print("For -i command, choose IDs from list shown in prompt results.")
791     print("For standard prompts, enter text not starting with '-' or digit.")
792     print("-----\n")
793
794     input_ = input("Query, command, or integer in [0, %d] for sample query: "
795                  %(len(sample_queries)-1))
796     flag = True # False --> query to change params, True --> real query
797     if input_ != "" and input_[0] == '-':
798         # query to modify options
799         frontendParams = update_params(input_, saved_query,
800                                         sample_queries, frontendParams,
801                                         backendTables)
802         query = ""
803         flag = False
804     elif input_.isdigit():
805         # actual query (prompt)
806         if int(input_) < len(sample_queries):
807             query = sample_queries[int(input_)]
808             saved_query = query
809             print("query:", query)
810         else:

```

```

811     print("Value must be <", len(sample_queries))
812     query = ""
813 else:
814     # actual query (prompt)
815     query = input_
816     saved_query = query
817
818     query = query.split(' ')
819     new_query = []
820     for k in range(len(query)):
821         token = query[k].lower()
822         if token in KW_map:
823             token = KW_map[token]
824         if token in dictionary:
825             new_query.append(token)
826     query = new_query.copy()
827     query.sort()
828     q_embeddings = {}
829     q_dictionary = {}
830
831     for k in range(1, 2*len(query)):
832
833         binary = get_bin(k, len(query))
834         sorted_word = ""
835         for k in range(0, len(binary)):
836             if binary[k] == '1':
837                 if sorted_word == "":
838                     sorted_word = query[k]
839                 else:
840                     sorted_word += "~" + query[k]
841
842         if sorted_word in sorted_ngrams:
843             ngrams = sorted_ngrams[sorted_word]
844             for word in ngrams:
845                 if word in dictionary:
846                     q_dictionary[word] = dictionary[word]
847                     if word in embeddings:
848                         embedding = embeddings[word]
849                         for token in embedding:
850                             if not frontendParams['Customized_pmi']:
851                                 pmi = embedding[token]
852                             else:
853                                 # customized pmi
854                                 pmi = custom_pmi(word, token, backendTables)
855                                 q_embeddings[(word, token)] = pmi
856
857 # if len(query) == 1:
858 #     # single-token query
859 #     frontendParams['embeddingKeyMinSize'] = 1
860 #     frontendParams['ContextMultitokenMinSize'] = 1
861
862 distill_frontendTables(q_dictionary, q_embeddings, frontendParams)
863
864 if len(input_) > 0 and flag:
865     print_results(q_dictionary, q_embeddings, backendTables, frontendParams)
866
867
868 #--- [5] Save backend tables
869
870 def create_KW_map(dictionary):
871     # singularization
872     # map key to KW_map[key], here key is a single token
873     # need to map unseen prompt tokens to related dictionary entries
874     # example: ANOVA -> analysis~variance, ...
875
876 OUT = open("KW_map.txt", "w")
877
878 for key in dictionary:
879     if key.count('~') == 0:
880         j = len(key)
881         keyB = key[0:j-1]
882         if keyB in dictionary and key[j-1] == 's':
883             if dictionary[key] > dictionary[keyB]:
884                 OUT.write(keyB + "\t" + key + "\n")
885             else:
886                 OUT.write(key + "\t" + keyB + "\n")

```

```

887     OUT.close()
888     return()
889
890
891 save = True
892 if save:
893     create_KW_map(dictionary)
894     for tableName in backendTables:
895         table = backendTables[tableName]
896         OUT = open('backend_' + tableName + '.txt', "w")
897         OUT.write(str(table))
898         OUT.close()
899
900 OUT = open('backend_embeddings.txt', "w")
901 OUT.write(str(embeddings))
902 OUT.close()
903
904 OUT = open('backend_sorted_ngrams.txt', "w")
905 OUT.write(str(sorted_ngrams))
906 OUT.close()

```

4.2 Thirty features to boost LLM performance

Many of these features are ground-breaking innovations that make LLMs much faster and not prone to hallucinations. They reduce the cost, latency, and amount of computer resources (GPU, training) by several orders of magnitude. Some of them improve security, making your LLM more attractive to corporate clients. For a larger list, see [here](#).

- The pageview function is denoted as pv . At the basic level, $\text{pv}(A)$ 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 pv because they have not accumulated much traffic yet. To correct for this, see lines 127–136 in the code. From now on, pv refers to normalized pageview counts also adjusted for time. The pageview for a multi-token t is then defined as

$$\text{pv}(t) = \frac{1}{|S(t)|} \cdot \sum_{A \in S(t)} \text{pv}(A), \quad (8.2)$$

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 $\text{pv}(C)$, the pageview count attached to a category C , by averaging 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 $\text{pv}_0(A)$ for an article A inside or outside the training set. The formula is

$$\text{pv}_0(A) = \frac{1}{W_A} \cdot \sum_{t \in T(A)} w_t \cdot \text{pv}(t), \quad (8.3)$$

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 pv_0 for an estimated value or

Figure 4.1: LLM for classification, with only 2 parameters

4.2.1 Fast search and caching

In order to match prompt components (say, embeddings) to the corresponding entities in the backend tables based on the corpus, you need good search technology. In general, you won't find an exact match. The solution consists in using approximate nearest neighbor search ([ANN](#)), together with smart encoding of embedding vectors. See how it works, [here](#). Then, use a [caching](#) mechanism to handle common prompts, to further speed up the processing in real time.

4.2.2 Leveraging sparse databases

While vector and graph databases are popular in this context, they may not be the best solution. If you have two million tokens, you may have as many as one trillion pairs of tokens. In practice, most tokens are connected to a small number of related tokens, typically less than 1000. Thus, the network or graph structure is very sparse, with less than a billion active connections. This is a far cry from a trillion! Hash tables are very good at handling this type of structure.

In my case, I use [nested hash tables](#), a format similar to [JSON](#), that is, similar to the way the input source (HTML pages) is typically encoded. A nested hash is a key-value table, where the value is itself a key-value table. The key in the root hash is typically a word, possibly consisting of multiple tokens. The keys in the child hash may be categories, agents, or URLs associated to the parent key, while values are weights indicating the association strength between a category and the parent key.

4.2.3 Contextual tokens

In standard LLMs, tokens are tiny elements of text, part of a word. In my multi-LLM system, they are full words and even combination of multiple words. This is also the case in other architectures, such as [LLama](#). They are referred to as multi-tokens. When it consists of non-adjacent words found in a same text entity (paragraph and so on), I call them [contextual tokens](#). Likewise, pairs of tokens consisting of non-adjacent tokens are called [contextual pairs](#). When dealing with contextual pairs and tokens, you need to be careful to avoid generating a very large number of mostly irrelevant combinations. Otherwise, you face token implosion.

Note that a word such as “San Francisco” is a single token. It may exist along with other single tokens such as “San” and “Francisco”.

4.2.4 Adaptive loss function

The goal of many deep neural networks ([DNN](#)) is to minimize a loss function, usually via stochastic gradient descent. This is also true for LLM systems based on [transformers](#). The loss function is a proxy to the evaluation metric that measures the quality of your output. In supervised learning LLMs (for instance, those performing supervised classification), you may use the evaluation metric as the loss function, to get better results. One of the best evaluation metrics is the full [multivariate Kolmogorov-Smirnov distance](#) (KS), see [here](#), with Python library [here](#).

But it is extremely hard to design an algorithm that makes billions of atomic changes to KS extremely fast, a requirement in all DNNs as it happens each time you update a weight. A workaround is to use an [adaptive loss function](#) that slowly converges to the KS distance over many epochs. I did not succeed at that, but I was able to build one that converges to the [multivariate Hellinger distance](#), the discrete alternative that is asymptotically equivalent to the continuous KS.

4.2.5 Contextual tables

In most LLMs, the core table is the [embeddings](#). Not in our systems: in addition to embeddings, we have category, tags, related items and various [contextual backend tables](#). They play a more critical role than the embeddings. It is more efficient to have them as backend tables, built during [smart crawling](#), as opposed to reconstructed post-creation as frontend elements.

4.2.6 Smart crawling

Libraries such as BeautifulSoup allow you to easily crawl and parse content such as JSON entities. However, they may not be useful to retrieve the embedded structure present in any good repository. The purpose of [smart crawling](#) is to extract structure elements (categories and so on) while crawling, to add them to your contextual backend tables. It requires just a few lines of ad-hoc Python code depending in your input source, and the result is dramatic. You end up with a well-structured system from the ground up, eliminating the need for prompt engineering.

4.2.7 LLM router, sub-LLMs, and distributed architecture

Good input sources usually have their own taxonomy, with categories and multiple levels of subcategories, sometimes with subcategories having multiple parent categories. You can replicate the same structure in your LLM, having multiple sub-LLMs, one per top category. It is possible to cover the entire human knowledge with 2000 sub-LLMs, each with less than 200,000 multi-tokens. The benefit is much faster processing and more relevant results served to the user.

To achieve this, you need an **LLM router**. It identifies prompt elements and retrieve the relevant information in the most appropriate sub-LLMs. Each one has its set of backend tables, hyperparameters, stopword list, and so on. There may be overlap between different sub-LLMs. **Fine-tuning** can be done locally, initially for each sub-LLM separately, or globally. You may also allow the user to choose a sub-LLM, by having a sub-LLM prompt box, in addition to the standard agent and query prompt boxes.

It is easy to implement this feature using a **distributed architecture**. Sub-LLMs are trained and operated in parallel, using multiple clusters.

4.2.8 From one trillion parameters down to two

By parameter, here I mean the weight between two connected neurons in a deep neural network. How can you possibly replace one trillion parameters by less than 5, and yet get better results, faster? The idea is to use parametric weights. In this case, you update the many weights with a simple formula relying on a handful of explainable parameters, as opposed to neural network activation functions updating (over time) billions of Blackbox parameters — the weights themselves — over and over. I illustrate this in Figure 4.1, featuring material from my coursebook, available [here](#).

4.2.9 Agentic LLMs

An **agent** detects the intent of a user within a prompt and helps deliver results that meet the intent in question. For instance, a user may be looking for definitions, case studies, sample code, solution to a problem, examples, datasets, images, or PDFs related to a specific topic, or links and references. The task of the agent is to automatically detect the intent and guide the search accordingly. Alternatively, the LLM may feature two prompt boxes: one for the standard query, and one to allow the user to choose an agent within a pre-built list.

Either way, you need a mechanism to retrieve the most relevant information in the backend tables. Our approach is as follows. We first classify each **text entity** (say, a web page, PDF document or paragraph) prior to building the backend tables. More specifically, we assign one or multiple agent labels to each text entity, each with its own score or probability to indicate relevancy. Then, in addition to our standard backend tables (categories, URLs, tags, embeddings, and so on), we build an agent table with the same structure: a **nested hash**. The parent key is a multi-token as usual, and the value is also a hash table, where each daughter key is an agent label. The value attached to an agent label is the list of text entities matching the agent in question, each with its own **relevancy score**.

4.2.10 Data augmentation via dictionaries

When designing an LLM system serving professional users, it is critical to use top quality input sources. Not only to get high quality content, but also to leverage its embedded structure (breadcrumbs, taxonomy, knowledge graph). This allows you to create contextual backend tables, as opposed to adding knowledge graph as a top, frontend layer. However, some input sources may be too small, if specialized or if your LLM consists of multiple sub-LLMs, like a **mixture of experts**.

To augment your corpus, you can use dictionaries (synonyms, abbreviations, **acronyms**), indexes, glossaries, or even books. You can also leverage user prompts. They help you identify what is missing in your corpus, leading to corpus improvement or alternate taxonomies. Augmentation is not limited to text. Taxonomy and knowledge graph augmentation can be done by importing external taxonomies. All this is eventually added to your backend tables. When returning results to a user prompt, you can mark each item either as internal (coming from the original corpus) or external (coming from augmentation). This feature will increase the security of your system, especially for enterprise LLMs.

4.2.11 Distillation done smartly

In xLLM, I build two frontend tables `q_dictionary` and `q_embeddings` each time a user generates a new prompt, in order to retrieve the relevant content from the corpus. These tables are similar and linked to the dictionary and embeddings backend tables, but far smaller and serving a single prompt. Then, I remove single tokens that are part of a multi-token when both have the same count in the dictionary. See Figure 2.6. It makes the output results more concise.

This step is called **distillation**. In standard LLMs, you perform distillation on backend rather than frontend tokens using a different mechanism, since multi-tokens are usually absent; it may result in hallucinations if not done properly. Also, in standard LLMs, the motivation is different: reducing a 500 billion token list, to (say) 50 billion. In xLLM, token lists are at least 1000 times smaller, so there is no real need for backend distillation.

Also, I keep a single copy of duplicate text entities. These are the core text elements found in the corpus, for instance paragraphs, PDFs, web pages and so on. As in Google search, when blending content from multiple sources (sometimes even from a single source, or for augmentation purposes), some text entities are duplicated, introducing a bias in the results, by giving too much weight to their tokens.

4.2.12 Reproducibility

Also called **replicability**. Most GenAI systems rely on deep neural networks (DNNs) such as GAN (generative adversarial networks). This is the case for transformers, a component of many LLMs. These DNNs rely on random numbers to generate latent variables. The result can be very sensitive to the seed (to initialize the random number generators). In many instances, particularly for synthetic data generation and GPU-based apps, the author does not specify seeds for the various **PRNG** (pseudo-random number generator) involved, be it from the Numpy, Random, Pandas, PyTorch libraries, base Python, or **GPU**.

The result is lack of **reproducibility**. This is not the case with my algorithms, whether GAN or NoGAN. All of them lead to reproducible results, including the xLLM system described here, which does not rely on transformers or random numbers. There have been some attempts to improve the situation recently, for instance with the `set_seed` function in some **transformer** libraries. However, it is not a full fix. Furthermore, the internal PRNGs found in Python libraries are subject to change without control on your side. To avoid these problems, you can use my PRNGs, some of them faster and better than any other on the market, with one of them requiring just one small line of code. See my article “Fast Random Generators with Infinite Period for Large-Scale Reproducible AI and Cryptography”, available [here](#).

Without sharing the seeds, the only way to make the model reproducible is to save the full model each time, with its billions of weights, instead of a handful of seed parameters. It also makes testing more difficult.

4.2.13 Explainable AI

Do you really need billions of weights (called parameters) that you compute iteratively with a neural network and thousands of epochs? Not to mention a stochastic gradient descent algorithm that may or may not converge? Note that xLLM has zero weight.

The idea consists of using functions that require few if any parameters, such as PMI (pointwise mutual information), an alternative to the cosine similarity and activation functions to measure keyword correlations. It is similar to some regularization methods in regression, with highly constrained or even fixed parameters, drastically reducing the dimension (or degrees of freedom) of the problem. Instead of estimating billions of weights with a deep neural network, the weights are governed by a few explainable **hyperparameters**. It makes fine-tuning much faster and a lot easier. This in turn allows for several benefits, see sections 4.2.14 and 4.2.15.

4.2.14 No training, in-memory LLM

With zero parameter, there is no need for **training**, though fine-tuning is still critical. Without the big neural network machinery, you or the user (thanks to explainable parameters) can fine-tune with in-memory database (the backend tables structured as nested hashes in my case), and in real time, with predictable outcome resulting from any change. There is no risk of **overfitting**. Another benefit is that new words, not part of the training set, do not require extra care, and are not a source of problems.

The result is a full **in-memory LLM** running on a laptop, without GPU. And customized output as the user can play with his favorite set of hyperparameters. Use algorithms such as **smart grid search** [12] to automate the fine-tuning, at least to find the best possible default hyperparameter set. What’s more, your LLM can run locally, which increases security and reduces external dependencies, especially valuable to corporate clients.

4.2.15 No neural network

In the previous section, I described an LLM not powered by a neural network. In particular, it does not need transformers. The concept of transformer-free LLM/RAG is not new. It is gaining in popularity. A side effect, at least in the case of xLLM, is that prompt results are bullet list items grouped in sections instead of long English: references, tags, categories, related keyword, links, datasets, PDFs, titles, but also full text entities coming from the corpus if desired, via the backend tables. With each item having its own **relevancy score**.

This conciseness and **exhaustivity** is particularly useful to business professionals or advanced users. It acts as a search tool, much better than Google or internal search boxes found on countless websites. However, beginners prefer well-worded, long, coherent English sentences that form a “story”. In short, generated rather than imported text, even though the quality of the imported text (full sentences) is high, because it comes from professional websites.

To satisfy beginners or any user fond on long English sentences, you would need to add an extra layer on top of the output. This may require a neural network, or not. Currently, xLLM returns items with a text entity ID attached to them, rather than the full content. A typical prompt may result in 20 IDs grouped into sections and clusters. The user can choose the IDs most relevant to his interests, then request the full content attached to these IDs, from the prompt menu. Displaying the full content by default would result in the user facing a flood of output text, defeating the purpose of conciseness.

4.2.16 Show URLs and references

xLLM returns URLs, references, and for each corpus entry (a text entity), even the email address of the employee maintaining the material in question in your organization. Other benefits include concise yet exhaustive results, relevancy scores attached to each item in the results, and local implementation.

4.2.17 Taxonomy-based evaluation

Assessing the quality of LLM search results is difficult. Usually, there is no “perfect answer” to compare with. Even if the results are correct (no [hallucination](#)), you don’t know if they are exhaustive. The problem is similar to evaluating clustering algorithms: both solve unsupervised learning problems. In special cases such as LLM for predictive analytics (a supervised learning technique), [evaluation](#) is possible via standard [cross-validation](#) techniques, see [here](#). Reversible translators from one language to another (English to German, or Python to Java) are easier to evaluate: translate from English to German, then from German back to English. Repeat this cycle 20 times and compare the final English version, with the original one.

Since xLLM mostly deals with [knowledge graphs](#), one way to assess quality is to have it reconstruct the internal taxonomy of the corpus, pretending we don’t know it. Then, you can compare the result with the actual taxonomy embedded in the corpus and retrieved during the crawl. Even then, the problem is not simple. In one example, the reconstructed taxonomy was less granular than the original one, and possibly better depending on the judge. But definitely different to some significant extent.

4.2.18 Augmentation via prompt data

A list of one million user prompts is a data gold mine, not just for [augmentation](#). You can use it to build agents, create an external taxonomy for taxonomy augmentation, detect what is missing in the corpus and address the missing content (possibly via augmentation). Or create lists of synonyms and abbreviations to improve your LLM. Imagine a scenario where users are searching for PMI, when that word is nowhere mentioned in your corpus, replaced instead by its expansion “pointwise mutual information”. Now, thanks to user queries, you can match them both.

4.2.19 Variable-length embeddings, indexing, and database optimization

Embeddings of static length work well with [vector databases](#). The price to pay is time efficiency (slow [vector search](#)) due to the large size of these vectors. With [variable length embeddings](#) and [nested hash databases](#), you can speed up search dramatically. Nested hashes are very similar to [JSON databases](#).

Also, in xLLM, the backend tables store [text entity](#) IDs along with embeddings, but not lengthy sentences (the full content). When retrieving results, the full original text associated to various items is not immediately displayed. Only the category, title, URL, related words and other short pieces of content, along with IDs. To retrieve the full content, the user must select IDs from prompt results. Then ask (from the command prompt) to fetch the full content attached to these IDs, from the larger database. You can automate this step, though I like the idea of the user selecting himself which IDs to dig in (based on score, category, and so on). The reason is because there may be many IDs shown in the prompt results, and the user’s choice may be different from algorithmic decisions. The mechanism behind accessing content via IDs is called [indexing](#).

Figure 3.1 shows the [command prompt](#) in xLLM. Note the `-p` option for real-time fine-tuning, and also the `-i` option to retrieve full content from a list of text entity IDs. To further optimize search, you can use [quantized embeddings](#) or [probabilistic nearest neighbor search](#). The latter is discussed [here](#). The word [retrieval](#) is sometimes used instead of search.

4.2.20 Favor backend over frontend engineering

The need for [prompt engineering](#) is due in part to faulty backend implementation. Too many tokens (most of them being noise), the choice of poor input sources (for instance, Reddit), too much reliance on embeddings only, and failure to detect and retrieve the embedded structure (knowledge graph, taxonomy) when crawling

the corpus. Instead, knowledge graphs are built on top rather than from the ground up. Prompt engineering is the fix to handle the numerous glitches. Some glitches come from the Python libraries themselves, see section 4.2.21.

By revisiting the fundamentals, even crawling and the choice of input sources, you can build a better architecture from the beginning. You may experience fewer hallucinations (if any) and avoid prompt engineering to a large extent. Your token list can be much smaller. In our case, embeddings is just one of the many backend tables, and not the most important one. The use of large contextual tokens and multiple sub-LLMs with ad-hoc parameters and stopword lists for each one, also contributes to the quality and robustness of the system.

4.2.21 Use NLP and Python libraries with caution

Python libraries such as auto-correct, singularize, stopwords, and stemming, have numerous glitches. You can use them, but I recommend having do-not-auto-correct, do-not-singularize lists and so on, specific to each sub-LLM. Examples of problems encountered include “*hypothesis*” singularized to “*hypothesi*”, “*Feller*” auto-corrected to “*seller*”, and the token “*p*” discarded even though in the sub-LLM that covers statistical science, it cannot be ignored (representing a probability, as in $p = 0.80$).

It is tempting to ignore punctuation, special or accented characters and upper cases to standardize the text. But watch out for potential side effects, especially when dealing with lastnames. These special text elements can be of great value if you keep them. Then, some words such as “*San Francisco*” are single-tokens disguised as double-tokens.

4.2.22 Self-tuning and customization

If your LLM – or part of it such as a sub-LLM – is light enough so that your backend tables and token lists fit in memory occupying little space, then it opens up many possibilities. For instance, the ability to fine-tune in real time, either via automated algorithms, or by letting the end-user doing it on his own. The latter is available in xLLM, with intuitive parameters: when fine-tuning, you can easily predict the effect of lowering or increasing some values. In the end, two users with the same prompt may get different results if working with different parameter sets. It leads to a high level of **customization**.

Now if you have a large number of users, with a few hundred allowed to fine-tune the parameters, you can collect valuable information. It becomes easy to detect the popular combinations of values from the customized parameter sets. The system can auto-detect the best parameter values and offer a small selection as default or recommended combinations. More can be added over time based on user selection, leading to organic **reinforcement learning** and **self-tuning**.

Self-tuning is not limited to parameter values. Some metrics such as **PMI** (a replacement to the dot product and cosine similarity) depend on some parameters that can be fine-tuned. But even the whole formula itself (a Python function) can be customized.

4.2.23 Local, global parameters, and debugging

In multi-LLM systems (sometimes called **mixture of experts**), whether you have a dozen or hundreds of sub-LLMs, you can optimize each sub-LLM locally, or the whole system. Each sub-LLM has its own backend tables to deal with the specialized corpus that it covers, typically a top category. However, you can have either local or global parameters:

- Global parameters are identical for all sub-LLMs. They may not perform as well as local parameters, but they are easier to maintain. Also, they can be more difficult to fine-tune. However, you can fine-tune them first on select sub-LLMs, before choosing the parameter set that on average, performs best across multiple high-usage sub-LLMs.
- Local parameters boost performance but require more time to fine-tune, as each sub-LLM has a different set. At the very least, you should consider using ad-hoc **stopwords** lists for each sub-LLM. These are built by looking at top tokens prior to filtering or distillation, and letting an expert determine which tokens are worth ignoring, for the topic covered by the sub-LLM in question.

You can have a mix of global and local parameters. In xLLM, there is a catch-all parameter set that returns the maximum output you can possibly get from any prompt. It is the same for all sub-LLMs. See option `-f` on the command prompt menu in Figure 3.1. You can use this parameter set as starting point, and modify values until the output is concise enough and shows the most relevant items at the top. The `-f` option is also used for **debugging**.

4.2.24 Displaying relevancy scores, and customizing scores

By showing a **relevancy score** to each item returned to a prompt, it helps the user determine which pieces of information are most valuable, or which text entities to view (similar to deciding whether clicking on a link or not, in classic search). It also helps with fine-tuning, as scores depend on the parameter set. Finally, some items with lower score may be of particular interest to the user; it is important not to return top scores exclusively. We are all familiar with Google search, where the most valuable results typically do not show up at the top.

Currently, this feature is not yet implemented in xLLM. However, many statistics are attached to each item, from which one can build a score. The **PMI** is one of them. In the next version, a custom score will be added. Just like the PMI function, it will be another function that the user can customize.

4.2.25 Intuitive hyperparameters

If your LLM is powered by a deep neural network (DNN), parameters are called **hyperparameters**. It is not obvious how to fine-tune them jointly unless you have considerable experience with DNNs. Since xLLM is based on **explainable AI**, parameters – whether backend or frontend – are intuitive. You can easily predict the impact of lowering or increasing values. Indeed, the end-user is allowed to play with frontend parameters. These parameters typically put restrictions on what to display in the results. For instance:

- Minimum PMI threshold: the absolute minimum is zero, and depending on the PMI function, the maximum is one. Do not display content with a PMI below the specified threshold.
- Single tokens to ignore because they are found in too many text entities, for instance ‘data’ or ‘table’. This does not prevent ‘data governance’ from showing up, as it is a multitoken of its own.
- Maximum gap between two tokens to be considered as related. Also, list of text separators to identify text sub-entities (a relation between two tokens is established anytime they are found in a same sub-entity).
- Maximum number of words allowed per multitoken. Minimum and maximum word count for a token to be integrated in the results.
- Amount of boost to add to tokens that are also found in the knowledge graph, taxonomy, title, or categories. Amount of stemming allowed.

4.2.26 Sorted n -grams and token order preservation

To retrieve information from the backend tables in order to answer a user query (prompt), the first step consists of cleaning the query and breaking it down into sub-queries. The cleaning consists of removing stopwords, some stemming, adding acronyms, auto-correct and so on. Then only keep the tokens found in the dictionary. The dictionary is a backend table built on the augmented corpus.

Let’s say that after cleaning, we have identified a subquery consisting of tokens A, B, C, D, in that order in the original prompt. For instance, (A, B, C, D) = (‘new’, ‘metadata’, ‘template’, ‘description’). The next step consists in looking at all 15 combinations of any number of these tokens, sorted in alphabetical order. For instance ‘new’, ‘metadata new’, ‘description metadata’, ‘description metadata template’. These combinations are called **sorted n -grams**. In the xLLM architecture, there is a key-value backend table, where the key is a sorted n -gram. And the value is a list of multtokens found in the dictionary (that is, in the corpus), obtained by rearranging the tokens in the parent key. For a key to exist, at least one rearrangement must be in the dictionary. In our example, ‘description template’ is a key (sorted n -gram) and the corresponding value is a list consisting of one element: ‘template description’ (a multitoken).

This type of architecture indirectly preserves to a large extent the order in which tokens show up in the prompt, while looking for all potential re-orderings in a very efficient way. In addition, it allows you to retrieve in the corpus the largest text element (multitoken) matching, up to token order, the text in the cleaned prompt. Even if the user entered a token not found in the corpus, provided that an acronym exists in the dictionary.

4.2.27 Blending standard tokens with tokens from the knowledge graph

In the corpus, each text entity is linked to some knowledge graph elements: tags, title, category, parent category, related items, and so on. These are found while crawling, and consist of text. I blend them with the ordinary text. They end up in the embeddings, and contribute to enrich the multitoken associations, in a way similar to augmented data. They also add contextual information not limited to token proximity. Also, you could have two embedding tables: one for regular text, and one for text found in the knowledge graph (taxonomies and so on).

4.2.28 Boosted weights for knowledge-graph and business tokens

Not all tokens are created equal, even those with identical spelling. Location is important: tokens come from ordinary text, or from the knowledge graph, see section 4.2.27. The latter yields higher quality. When a token is found while parsing a text entity, its counter is incremented in the dictionary, typically by 1. However, in the xLLM architecture, you can add an extra boost to the increment if the token is found in the knowledge graph, as opposed to ordinary text. Some backend parameters allow you to choose how much boost to add, depending on whether the graph element is a tag, category, title, and so on. Another strategy is to use two counters: one for the number of occurrences in ordinary text, and a separate one for occurrences in the knowledge graph.

See chapter 11 for details about the implementation of [graph tokens](#) originating from the knowledge graph. Another type of multi-tokens is [business tokens](#). A sub-LLM dealing with part of your corporate corpus can benefit from a dictionary consisting of business terms. Many are found in the corpus itself, others come from augmentation to help with prompt processing. Each business term has its own categories and synonyms. It can be handled in a way similar to graph tokens, or added as tags in the tag field attached to each text entity.

4.2.29 Versatile command prompt

Most commercial LLM apps that I am familiar with offer limited options besides the standard prompt. Sure, you can input your corpus, work with an API or SDK. In some cases, you can choose specific deep neural network (DNN) hyperparameters for fine-tuning. But for the most part, they remain a Blackbox. One of the main reasons is that they require training, and training is a laborious process when dealing with DNNs with billions of weights.

With in-memory LLMs such as xLLM, there is no training. Fine-tuning is a lot easier and faster, thanks to explainable AI: see section 4.2.25. In addition to standard prompts, the user can enter command options in the prompt box, for instance `-p key value` to assign `value` to parameter `key`. See Figure 3.1. The next prompt will be based on the new parameters, without any delay to return results based on the new configuration.

There are many other options besides fine-tuning: an agent box allowing the user to choose a specific agent, and a category box to choose which sub-LLMs you want to target. You can even check the sizes of the main tables (embeddings, dictionary, contextual), as they depend on the backend parameters.

4.2.30 Boost long multitokens and rare single tokens

I use different mechanisms to give more importance to multitokens consisting of at least two terms. The higher the number of terms, the higher the importance. For instance, if a cleaned prompt contains the multitokens (A, B), (B, C), and (A, B, C) and all have the same count in the dictionary (this happens frequently), xLLM with display results related to (A, B, C) only, ignoring (A, B) and (B, C). Some frontend parameters allow you to set the minimum number of terms per multitoken, to avoid returning generic results that match just one token. Also, the PMI metric can be customized to favor long multi-tokens.

Conversely, some single tokens, even consisting of one or two letters depending on the sub-LLM, may be quite rare, indicating that they have high informative value. There is an option in xLLM not to ignore single tokens with fewer than a specified number of occurrences. Note that in many LLMs on the market, tokens are very short. They consist of parts of a word, not even a full word, and multitokens are absent. In xLLM, very long tokens are favored, while tokens that are less than a word, don't exist. Yet, digits like 1 or 2, single letters, IDs, symbols, codes, and even special characters can be token if they are found *as is* in the corpus.

4.2.31 Disambiguation

[Disambiguation](#) consists in offering multiple choices to the user, when the prompt is ambiguous. It happens when a keyword or keyword combination has multiple meanings, depending on the context. Asking the user to choose a specific category may alleviate hallucinations.

4.2.32 Allow for customized search

Allow the user to choose agents and sub-LLMs from a pre-built list. This is better than having the system guess the user intent (the action) and automatically re-route the query to appropriate sub-LLMs without user input other than the prompt. Also, it is a good idea to offer the following options in the user interface:

- Allow the user to choose recent results: favoring material recently added to the corpus. Almost all corpuses have timestamps attached to each document. More generally, offer different ranking criteria to the user, besides [recency](#). For instance, favoring long over short text entities (or the other way around) when retrieving content.

- Allow for negative words in the prompt, via a separate “negative search box” in addition to the prompt box. What I mean here is this: if the prompt contains the multi-tokens A, B, C and the negative search contains the multi-tokens D, E, return results related to A, B, C, but only those that do not contain D or E. In this case, D, E are called **negative tokens**.

4.3 LLM glossary

Some terms are unique to xLLM, others such as decoder or transformer are only found in standard LLMs. Here, DNN stands for deep neural network. In DNNs, that is, in traditional LLMs, the parameters are the weights connecting neurons. In xLLM, there is usually zero or few weights; parameters play the role of hyperparameters.

For a more comprehensive list, check out the index at the end of this book. Each index entry or sub-entry has backlinks pointing to where it is discussed in the book, usually in multiple sections. Indexed terms are easy to recognize, as they appear in orange in the text. Interestingly, one possible use of the xLLM technology is to automatically create large indexes spanning across multiple documents.

Table 4.1: LLM glossary

agent	A mechanism to detect user intent in a prompt, to retrieve the most appropriate content. An agent determines what the user is looking for: definition, examples, search results, data, best practices, references, URLs, and so on. Different from an action , which consists of running a separate app for instance to write an email, do some data analysis, or perform some computations. LLMs that can handle various agents are called multi-agent .
ANN	Approximate nearest neighbor search. 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 12.
backend	The xLLM architecture is split into backend (Figure 1.4) and frontend (Figure 1.5). The backend (parameters, tables) deals with the corpus, knowledge graph, and augmentation. It does not see what is in the prompt. The frontend deals with the prompt. It does not see what is in the backend. The LLM is an interface that connects frontend content, to backend tables.
contextual token	A multitoken consisting of multiple single tokens, say (A, B, C), where the tokens A and B, or B and C, are not adjacent to each other in the corpus. Still, A, B and C are found in a same text sub-entity, for instance a paragraph in a larger text entity (web page or JSON entity).
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.
distillation	Data distillation is a technique used to reduce the size of your dataset with minimum loss of information, sometimes even improving predictive algorithms, even via random deletions. It is also used in the context of LLMs, to reduce the size of token lists, and to remove noise or garbage.
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 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 popular.
exhaustivity	One of the most overlooked evaluation metrics when assessing LLM performance or in benchmarking tests. It depends on the corpus and the knowledge of the expert judging the prompt results. For exhaustivity, consider corpus augmentation and using acronyms to identify all possible name variations for words found in the prompt, to map them to what is in the corpus.

Continued on next page

Table 4.1: LLM glossary (Continued)

explainable AI	LLMs powered by deep neural networks are Blackboxes governed by hyperparameters . To the contrary, xLLM does not need training and does not use weights estimated via DNNs. Instead, it is fine-tuned using intuitive parameters, a feature known as explainable AI .
fine-tuning	Fine-tuning consists in testing various parameter values (hyperparameters when neural networks are involved) to increase performance (speed, exhaustivity or relevancy in prompt results). Different from training . In xLLM, there is no training, except if used as a classifier, taxonomy builder, or for predictions. Instead, the user can fine-tune frontend parameters in real time. You can also automatically determine the optimum parameters based on user preferences. This is called self-tuning .
frontend	See backend.
GAN	Generative adversarial network . One of the many types of DNN (deep neural network). 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. To synthetize tabular data, NogAN is better: see chapter 7.
GPT	GPT stands for Generative Pre-trained Transformer. 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 .
hash database	See also key-value database. The most sophisticated is a nested hash , where the key can be any structure (typically a t-uple or a list) and the value is itself a nested hash. They can be updated very quickly in memory, and the structure is similar to JSON databases . It is extensively used in xLLM, see Figure 1.1.
hyperparameter	In LLMs powered by deep neural networks (DNN), the hyperparameters are those of the DNN: number of epochs, seed, number of layers, batch size, type a gradient descent, learning rate, parameters attached to the loss function, and so on. In xLLM, hyperparameters – actually called parameters – govern the type of results returned to a user prompt. There are two types: backend and frontend parameters. The former are linked to retrieval in the corpus when crawling. The latter are linked to prompt processing and you can fine-tune them in real time, from the command prompt.
in-memory LLM	When backend tables, weights, and token lists fit in memory, it makes sense to load everything in memory to boost speed. This is the case with xLLM. It can be done with large LLMs, especially those not relying on neural networks and billions of weights.
key-value database	Also known as hash table or dictionary in Python. In xLLM, 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.
knowledge graph	A structure connecting high-level text elements such as categories and sub-categories, or tags. A typical example is a taxonomy . It can be retrieved from the corpus itself when crawling, thanks to breadcrumbs or categorization embedded in the corpus. Or built on top of it, or augmented via external input sources. Categories can have multiple parent categories, and multiple subcategories.
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, here . 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	LLM architecture offering different types of agents, for instance to solve math problems or to generate Python code. In xLLM, the word agent represents a mechanism to detect user intent and to serve results matching the intent. For instance, showing definitions, examples, references, tables, best practices and so on.

Continued on next page

Table 4.1: LLM glossary (Continued)

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 [14]. 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.
text entity	The main text unit in xLLM, for instance a JSON entry such as pictured in Table 2.1, with raw text, various fields, and contextual information such as category. Sometimes augmented with agent labels. It could also be (say) a Wikipedia web page. Sub-entities are shorter and delimited by pre-specified separators such as period or semi-colon. For two multitokens to be connected (other than via the knowledge graph), they must reside within a same sub-entity.
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 Table 12.1. 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	Used to retrieve corpus embeddings identical or close to embeddings detected in a user prompt. 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 .
weight	Also called parameters in deep neural networks (DNN). They are the weights attached to connections between neurons. Thus, LLMs based on DNNs may have billions or even trillions of them, while xLLM has zero, except in the version that performs clustering and predictions (less than 5 weights). Another way to look at it is that weights are implicit in xLLM (not estimated) and governed by a few high-level parameters. See section 4.2.8.

Part II

Outperforming Neural Nets and Classic AI

Chapter 5

Building and evaluating a taxonomy-enriched LLM

In Part I of this book, I 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 Likelihood	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 5.1: 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 5.2: “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 5.1: 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.

5.1 Project and solution

The goal is not to write code from scratch to solve the problem, but to look at my implementation in section 5.2, 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`. 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 5.2, 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 5.1 and 5.2. 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 5.2, 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 5.1 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.

5.2 Python code

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

```

1 # taxonomy.py: vincentg@mltechniques.com
2
3 import requests
4
5 # Unlike xllm6.py, xllm6_short.py does not process the (huge) crawled data.
6 # Instead, it uses the much smaller summary tables produced by xllm6.py
7
8
9 #--- [1] get tables if not present already
10
11 # First, get xllm6_util.py from GitHub and save it locally as xllm6_util.py
12 #   note: this python code does that automatically for you
13 # Then import everything from that library with 'from xllm6_util import *'
14 # Now you can call the read_xxx() functions from that library
15 # In addition, the tables stopwords and utf_map are also loaded
16 #
17 # Notes:
18 #   - On first use, download all locally with overwrite = True
19 #   - On next uses, please use local copies: set overwrite = False
20
21 # Table description:
22 #
23 # unless otherwise specified, a word consists of 1, 2, 3, or 4 tokens
24 # word_pairs is used in xllm6.py, not in xllm6_short.py
25 #
26 # dictionary = {} words with counts: core (central) table
27 # word_pairs = {} pairs of 1-token words found in same word, with count
28 # word2_pairs = {} pairs of multi-token words found on same URL, with count
29 # url_map = {} URL IDs attached to words in dictionary
30 # arr_url = [] maps URL IDs to URLs (one-to-one)
31 # hash_category = {} categories attached to a word
32 # hash_related = {} related topics attached to a word
33 # hash_see = {} topics from "see also" section, attached to word
34 # ngrams_table = {} ngrams of word found when crawling
35 # compressed_ngrams_table = {} only keep ngram with highest count
36 # utf_map = {} map accented characters to non-accented version
37 # stopwords = () words (1 or more tokens) not accepted in dictionary
38 # word_hash = {} list of 1-token words associated to a 1-token word
39 # word2_hash = {} list of multi-token words associated to a multi-token word
40 # compressed_word2_hash = {} shorter version of word2_hash
41 # embeddings = {} key is a 1-token word; value is hash of 1-token:weight
42 # embeddings2 = {} key is a word; value is hash of word:weight
43
44
45 path = "https://raw.githubusercontent.com/VincentGranville/Large-Language-Models/main/xllm6/"
46
47 overwrite = False # if True, get tables from GitHub, otherwise use local copy
48
49 if overwrite:
50
51     response = requests.get(path + "xllm6_util.py")
52     python_code = response.text

```

```

53
54     local_copy = "xllm6_util"
55     file = open(local_copy + ".py", "w")
56     file.write(python_code)
57     file.close()
58
59 # get local copy of tables
60
61 files = [ 'xllm6_arr_url.txt',
62           'xllm6_compressed_ngrams_table.txt',
63           'xllm6_compressed_word2_hash.txt',
64           'xllm6_dictionary.txt',
65           'xllm6_embeddings.txt',
66           'xllm6_embeddings2.txt',
67           'xllm6_hash_related.txt',
68           'xllm6_hash_category.txt',
69           'xllm6_hash_see.txt',
70           'xllm6_url_map.txt',
71           'xllm6_word2_pairs',
72           'stopwords.txt'
73       ]
74
75 for name in files:
76     response = requests.get(path + name)
77     content = response.text
78     file = open(name, "w")
79     file.write(content)
80     file.close()
81
82 import xllm6_util as llm6
83
84 # if path argument absent in read_xxx(), read from GitHub
85 # otherwise, read from copy found in path
86
87 arr_url    = llm6.read_arr_url("xllm6_arr_url.txt", path="")
88 dictionary  = llm6.read_dictionary("xllm6_dictionary.txt", path="")
89 stopwords   = llm6.read_stopwords("stopwords.txt", path="")
90
91 compressed_ngrams_table = llm6.read_table("xllm6_compressed_ngrams_table.txt",
92                                              type="list", path="")
93 compressed_word2_hash = llm6.read_table("xllm6_compressed_word2_hash.txt",
94                                            type="hash", path="")
95 embeddings   = llm6.read_table("xllm6_embeddings.txt", type="hash", path="",
96                                 format="float")
97 embeddings2  = llm6.read_table("xllm6_embeddings2.txt", type="hash", path="",
98                                 format="float")
99 hash_related = llm6.read_table("xllm6_hash_related.txt", type="hash", path="")
100 hash_see    = llm6.read_table("xllm6_hash_see.txt", type="hash", path="")
101 hash_category = llm6.read_table("xllm6_hash_category.txt", type="hash", path="")
102 url_map    = llm6.read_table("xllm6_url_map.txt", type="hash", path="")
103 word2_pairs = llm6.read_table("xllm6_word2_pairs.txt", type="list", path="")
104
105
106 #--- [2] Create/save taxonomy tables if overwrite = True, otherwise read them
107
108 from collections import OrderedDict
109
110 ignoreWords = { "term", "th", "form", "term", "two", "number", "meaning", "normally",
111                 "summarizes", "assumed", "assumes", "p", "s", "et", "possible",
112                 "&#9671;", ";", "denoted", "denotes", "computed", "other" }
113
114 def create_taxonomy_tables(threshold, thresh2, ignoreWords, dictionary):
115
116     topWords = {}      # words with highest counts, from dictionary
117     wordGroups = {}    # hash of hash: key = topWord; value = hash of words
118                 # containing topWord (can be empty)
119     connectedTopWords = {} # key = (wordA, wordB) where wordA and wordB contains
120                 # a topWord; value = occurrences count
121     smallDictionary = {} # dictionary entries (words) containing a topWord
122     connectedByTopWord = {} # same as connectedTopWords, but in flattened hash format;
123                           # key = topWord
124     missingConnections = {} # if this table is not empty, reduce threshold and/or thresh2
125
126     for word in dictionary:
127         n = dictionary[word] # word count
128         tokens = word.count('`')

```

```

129     if n > threshold and word not in ignoreWords: # or tokens > 1 and n > 1:
130         topWords[word] = n
131
132     for topWord in topWords:
133         n1 = dictionary[topWord]
134         hash = {}
135         for word in dictionary:
136             n2 = dictionary[word]
137             if topWord in word and n2 > thresh2 and word != topWord:
138                 hash[word] = n2
139     if hash:
140         hash = dict(sorted(hash.items(), key=lambda item: item[1], reverse=True))
141     else:
142         missingConnections[topWord] = 1
143     wordGroups[topWord] = hash
144
145     for topWord in topWords:
146         for word in dictionary:
147             if topWord in word:
148                 smallDictionary[word] = dictionary[word]
149
150     counter = 0
151     for topWordA in topWords:
152         if counter % 10 == 0:
153             print("Create connectedTopWords: ", counter, "/", len(topWords))
154         counter += 1
155         hash = {}
156         for topWordB in topWords:
157             key = (topWordA, topWordB)
158             if topWordA != topWordB:
159                 connectedTopWords[key] = 0
160                 for word in smallDictionary:
161                     if topWordA in word and topWordB in word:
162                         connectedTopWords[key] += 1
163                         if topWordB in hash:
164                             hash[topWordB] += 1
165                         else:
166                             hash[topWordB] = 1
167         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="")
233     for line in data:
234         line = line.split('\t')
235         count = int(line[1])
236         key = llm6.text_to_list(line[0])
237         connectedTopWords[key] = count
238
239
240 #--- [3] Play with taxonomy tables to get insights and improve them
241
242 topWords = dict(sorted(topWords.items(), key=lambda item: item[0]))
243
244 def show_menu(n, dict_mode):
245
246     # option 'o' useful to check if topWordA, topWordB are connected or not
247     # option 'c' shows all topWordB connected to topWordA = topWord
248
249     print("Command line menu: \n")
250     print("<Enter>      - exit")
251     print("h            - help: show menu options")
252     print("a            - show all top words")
253     print("ds           - select short dictionary")
254     print("df           - select full dictionary")
255     print("n integer    - display entries with count >= integer")
256     print("f string     - find string in dictionary")
257     print("g topWord    - print groupWords[topWord]")
258     print("c topWord    - print connectedByTopWord[topWord]\n")
259     print("l topWordA topWordB - (topWordA, topWordB) connections count\n")
260     print("current settings: n = %3d, dictionary = %s" %(n, dict_mode))
261     print()
262     return()
263
264 topWords = dict(sorted(topWords.items(), key=lambda item: item[0]))
265
266 dict_mode = 'short'
267 dict = smallDictionary
268 query = "o"
269 n = 0 # return entries with count >= n
270 show_menu(n, dict_mode)
271
272 while query != "":
273
274     query = input("Enter command, ex: <c hypothesis> [h for help]: ")
275     queries = query.split(' ')
276     action = queries[0]
277     if len(queries) > 2:
278         queries[1] = queries[1] + " " + queries[2]
279
280     if action == 'h':

```

```

281     show_menu(n, dict_mode)
282
283     elif action == 'ds':
284         dict = smallDictionary
285         dict_mode = 'short'
286
287     elif action == 'df':
288         dict = dictionary
289         dict_mode = 'full'
290
291     elif action == 'a':
292         for topWord in topWords:
293             count = topWords[topWord]
294             print(count, topWord)
295         print()
296
297     elif action in ('f', 'g', 'c', 'l', 'n') and len(queries) > 1:
298         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         astrig = 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 #--- [4] Build local taxonomy using external taxonomy
355

```

```

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
417     for category in categories:
418         depth = categories[category]
419         similarity = compute_similarity(dictionary, word, category)
420         if similarity > max_similarity:
421             max_similarity = similarity
422             max_depth = depth
423             NN_category = category
424     assignedCategories[word] = (NN_category, max_depth, max_similarity)
425     if counter % 200 == 0:
426         print("%5d / %5d: %d %4.2f %s | %s"
427               %(counter, len(dictionary), max_depth, max_similarity, word, NN_category))
428     counter += 1
429
430 OUT = open("xllm6_assignedCategories.txt", "w")
431 for word in assignedCategories:
432     OUT.write(word+"\t"+str(assignedCategories[word])+"\n")
433

```

Chapter 6

LLM for Clustering and Predictions

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 5.

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 6.3. 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** [6], 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 pv . At the basic level, $\text{pv}(A)$ 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 pv because they have not accumulated much traffic yet. To correct for this, see lines 127–136 in the code. From now on, pv refers to normalized pageview counts also adjusted for time. The pageview for a multi-token t is then defined as

$$\text{pv}(t) = \frac{1}{|S(t)|} \cdot \sum_{A \in S(t)} \text{pv}(A), \quad (6.1)$$

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 $\text{pv}(C)$, the pageview count attached to a category C , by averaging 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 $\text{pv}_0(A)$ for an article A inside or outside the training set. The formula is

$$\text{pv}_0(A) = \frac{1}{W_A} \cdot \sum_{t \in T(A)} w_t \cdot \text{pv}(t), \quad (6.2)$$

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 pv_0 for an estimated value or prediction, and pv for an observed value. In some cases, $T(A)$ is empty and thus Formula (6.2) is meaningless. The solution consists in replacing the predicted value by $\text{pv}_0(A) = \text{pv}(C_A)$, where C_A is the category attached to article A .

The prediction formula (6.2) 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

$$L(G) = \bigcup_{t \in G} S(t). \quad (6.3)$$

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 project 5.2 in [13]), designed to work with hash tables rather than matrices. The similarity metric is defined as

$$s(t_1, t_2) = \frac{|S(t_1) \cap S(t_2)|}{|S(t_1) \cup S(t_2)|} \in [0, 1]. \quad (6.4)$$

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](#).

6.1 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 6.3. 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?

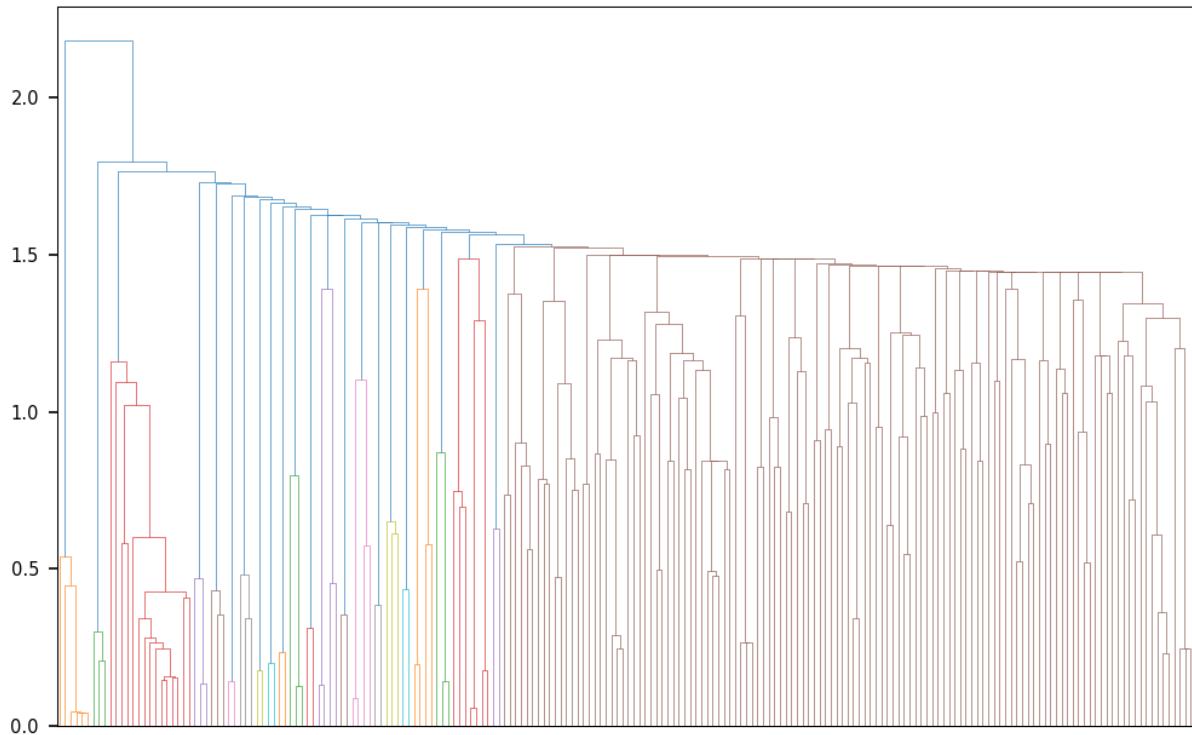


Figure 6.1: Multi-tokens hierarchical clustering: dendrogram (20 groups, 104 multi-tokens)

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 (6.1) by a weighted sum as in (6.2), 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 6.1: Cluster linked to $\{\text{'Learning}\sim\text{vs}', \text{'Machine}^\wedge\text{vs}', \text{'Machine}\sim\text{Learning}\sim\text{vs}'\}$

The remaining of this project is devoted to visualizations pertaining to the various components discussed earlier. Table 6.1 features one of the 20 article clusters based on hierarchical clustering of multi-tokens. In this case, the multi-token group, automatically detected, is $\{\text{'Learning}\sim\text{vs}', \text{'Machine}^\wedge\text{vs}', \text{'Machine}\sim\text{Learning}\sim\text{vs}'\}$. It is a mix of standard and contextual tokens. The corresponding article cluster was automatically retrieved using Formula (6.3). 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.

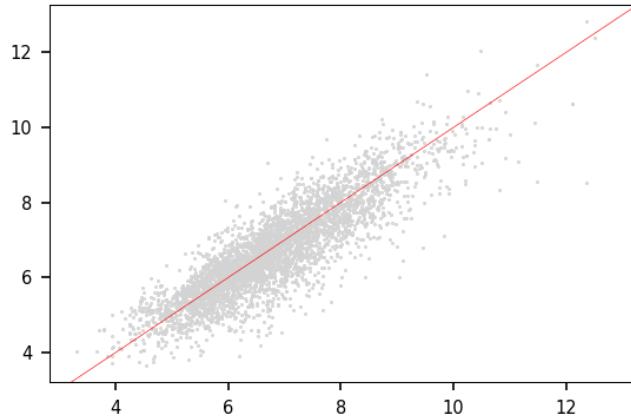


Figure 6.2: Scatterplot, observed vs predicted article pv (4000 articles)

For [token prediction](#), that is, predicting the next token in a sentence – what LLMs were designed for initially – see section 7.1 entitled “Synthesizing DNA Sequences with LLM Techniques”, in [13]. In this case, the alphabet consists of 4 letters: the 4 nucleotides A, T, G, C that make up the DNA molecule. In [13], Figures 7.1 and 7.2 show how the predictions significantly outperform the baseline.

6.2 Visualizations and discussion

Figure 6.1 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 6.1 for an example combining 3 multi-tokens, corresponding to one of the small clusters in the dendrogram.

Figure 6.2 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 6.3, 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 6.4 shows the corrected numbers: it represents the reality more faithfully; the transform used here is non linear.

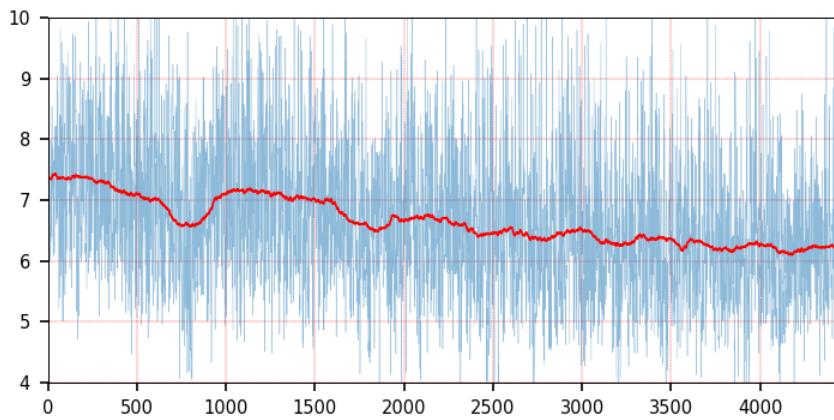


Figure 6.3: Article pv over time, before adjusting for time

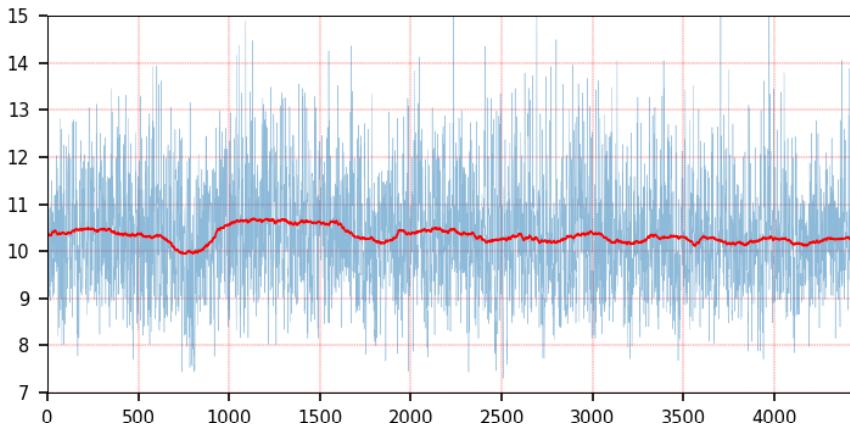


Figure 6.4: Article pv over time, after time adjustment

6.3 Python code

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

```

1 # nlp_scoring.py | vincentg@mltechniques.com
2
3 import pandas as pd
4 import numpy as np
```

```

5 import matplotlib as mpl
6 from matplotlib import pyplot as plt
7
8 mpl.rcParams['lines.linewidth'] = 0.3
9 mpl.rcParams['axes.linewidth'] = 0.5
10 plt.rcParams['xtick.labelsize'] = 7
11 plt.rcParams['ytick.labelsize'] = 7
12
13
14 #--- [1] Read data
15
16 # read data either from GitHub, or use local copy, depending on path
17
18 # path = "https://raw.githubusercontent.com/VincentGranville/Statistical-Optimization/main/"
19 path = ""
20 filename = "Articles-Pageviews.txt"
21 url = path + filename
22
23 data = pd.read_csv(url, sep='\t', engine='python', encoding='cp1252')
24 header = ['Title', 'URL', 'Author', 'Page views', 'Creation date', 'Status']
25
26
27 #--- [2] Core functions and tables
28
29 # pv is aggregated normalized pageview; pv_rel is average pv for a specific group
30
31 arr_titles = data['Title'] # article titles
32 arr_status = data['Status'] # blog post, forum question, etc.
33 arr_pv = data['Page views'] # article absolute (un-normalized) pageviews
34 arr_url = data['URL'] # article URL
35 arr_author = data['Author'] # article author
36 arr_categories = {} # article category (combo of author, status, URL extracts)
37
38 category_pv = {} # key: article category; value: average pv per category
39 category_count = {} # key: article category; value: number of articles in that category
40 hash_words_count = {} # key: word; value: number of occurrences across all articles
41 hash_pv = {} # key: word; value: aggregated pv across all those articles
42 hash_titles = {} # key: word; value: hash of articles containing word (value: pv)
43 hash_authors_count = {} # key: author;
44
45 def compress_status(status):
46     status = status.lower()
47     if 'forum' in status:
48         status = 'forum_question'
49     elif 'resource' in status:
50         status = 'resource'
51     else:
52         status = 'blog'
53     return(status)
54
55 def compress_url(url):
56     url = url.lower()
57     if 'datasciencecentral' in url:
58         url = 'DSC'
59     else:
60         url = 'Other'
61     return(url)
62
63 def update_hash(titleID, word, hash_words_count, hash_pv, hash_titles):
64
65     if word in hash_words_count:
66         hash_words_count[word] += 1
67         hash_pv[word] += pv # what if words found twice in same title ???
68         hash = hash_titles[word]
69         hash[titleID] = pv
70         hash_titles[word] = hash
71     else:
72         hash_words_count[word] = 1
73         hash_pv[word] = pv
74         hash_titles[word] = { titleID : pv }
75     return(hash_words_count, hash_pv, hash_titles)
76
77 def update_single_tokens(titleID, words, hash_words_count, hash_pv, hash_titles):
78
79     # words is an array of words (found in title)
80

```

```

81     for word in words:
82         hash_words_count, hash_pv, hash_titles = update_hash(titleID, word,
83                         hash_words_count, hash_pv, hash_titles)
84     return(hash_words_count, hash_pv, hash_titles)
85
86 def update_joint_tokens(titleID, words, hash_words_count, hash_pv, hash_titles):
87
88     # words is an array of words (found in title)
89     # capturing adjacent tokens
90
91     for idx in range(len(words)):
92         word = words[idx]
93         if idx < len(words)-1 and words[idx+1] != '':
94             word += " " + words[idx+1]
95         hash_words_count, hash_pv, hash_titles = update_hash(titleID, word,
96                         hash_words_count, hash_pv, hash_titles)
97         if idx < len(words)-2 and words[idx+2] != '':
98             word += " " + words[idx+2]
99         hash_words_count, hash_pv, hash_titles = update_hash(titleID, word,
100                         hash_words_count, hash_pv, hash_titles)
101     return(hash_words_count, hash_pv, hash_titles)
102
103 def update_disjoint_tokens(titleID, words, hash_words_count, hash_pv, hash_titles):
104
105     # words is an array of words (found in title)
106     # word1 and word2 cannot be adjacent:
107     #   if they were, this is already captured in the update_joint_tokens function
108
109     param_D = 1 # word1 and word2 must be separated by at least param_D tokens
110
111     for k in range(len(words)):
112         for l in range(len(words)):
113             word1 = words[k]
114             word2 = words[l]
115             distance = abs(k - l)
116             if word1 < word2 and distance > param_D and word1 != '' and word2 != '':
117                 word12 = word1 + " " + word2
118                 hash_words_count, hash_pv, hash_titles = update_hash(titleID,
119                                 word12, hash_words_count, hash_pv, hash_titles)
120     return(hash_words_count, hash_pv, hash_titles)
121
122 def get_article_pv(titleID, arr_pv):
123     # using log: it gives a better normalization and fit than sqrt, for pv distribution
124     return(np.log(float(arr_pv[titleID])))
125
126
127 #--- [3] De-trend pv
128
129 param_T1 = 0.80
130 param_T2 = 0.11
131 arr_pv_new = np.zeros(len(arr_pv))
132
133 for k in range(len(arr_pv)):
134     energy_boost = param_T1 * np.sqrt(k + param_T2 * len(arr_pv))
135     arr_pv_new[k] = arr_pv[k] * (1 + energy_boost)
136 arr_pv = np.copy(arr_pv_new)
137
138
139 #--- [4] Populate core tables
140
141 for k in range(len(data)):
142     author = arr_author[k]
143     if author in hash_authors_count:
144         hash_authors_count[author] +=1
145     else:
146         hash_authors_count[author] =1
147
148 param_A = 50 # authors with fewer than param_A articles are bundled together
149
150 for k in range(len(data)):
151
152     pv = get_article_pv(k, arr_pv)
153     cstatus = compress_status(arr_status[k])
154     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
221 # Surprisingly, word pv's have much more predictive power than category pv's
222 # For new titles with no decent word in historical tables, it is very useful
223
224 for k in range(len(data)):
225     category = arr_categories[k]
226     pv = get_article_pv(k, arr_pv)
227     if category in category_count:
228         category_pv[category] += pv
229         category_count[category] += 1
230     else:
231         category_pv[category] = pv
232         category_count[category] = 1

```

```

233
234     print()
235     input("> Press <Enter> to continue")
236     print()
237     print("== CATEGORIES: titles count, pv, category name\n")
238
239     for category in category_count:
240         count = category_count[category]
241         category_pv[category] /= count
242         print("%5d %6.3f %s" %(count, category_pv[category], category))
243     print()
244
245
246 #--- [7] Create short list of frequent words with great performance
247
248 # This reduces the list of words for the word clustering algo in next step
249 # Goal: In next steps, we cluster groups of words with good pv to
250 #       categorize various sources of good performance
251
252 short_list = {}
253 keep = 'good' # options: 'bad' or 'good'
254
255 param_G1 = 1.10 # must be above 1, large value to get titles with highest pv
256 param_G2 = 0.90 # must be below 1, low value to get articles with lowest pv
257 param_C1 = 10 # single-token word with count <= param_C1 not included in short_list
258 param_C2 = 4 # multi-token words with count <= param_C2 not included in short_list
259
260 for word in hash_pv_deduped:
261     count = hash_words_count[word]
262     pv = hash_pv[word]/count
263     if keep == 'good':
264         flag = bool(pv > param_G1 * mean_pv)
265     elif keep == 'bad':
266         flag = bool(pv < param_G2 * mean_pv)
267     if flag and (count > param_C1 or count > param_C2 and '^' in word):
268         short_list[word] = 1
269
270
271 #--- [8] compute similarity between words in short list, based on common titles
272
273 # Find list of articles S1 and S2, containing respectively word1 and word2
274 # word1 is similar to word2 if |S1 intersection S2| / |S1 union S2| is high
275
276 hash_pairs = {}
277 aux_list = {}
278 param_S = 0.20 # if similarity score about this threshold, word1 and word2 are linked
279
280 for word1 in short_list:
281     for word2 in short_list:
282
283         set1 = set()
284         for titleID1 in hash_titles[word1]:
285             set1.add(titleID1)
286         set2 = set()
287         for titleID2 in hash_titles[word2]:
288             set2.add(titleID2)
289
290         count1 = len(set1)
291         count2 = len(set2)
292         count12 = len(set1.union(set2))
293         similarity = len(set1.intersection(set2)) / count12
294
295         if similarity > param_S and word1 < word2:
296             hash_pairs[(word1, word2)] = similarity
297             hash_pairs[(word2, word1)] = similarity
298             hash_pairs[(word1, word1)] = 1.00
299             hash_pairs[(word2, word2)] = 1.00
300             aux_list[word1] = 1
301             aux_list[word2] = 1
302
303
304 #--- [9] Turn hash_pairs{} into distance matrix dist_matrix, then perform clustering
305
306 # Keyword clustering based on similarity metric computed in previous step
307 # Alternative to exploite sparse matrix: connected components algorithm on hash_pairs
308 # Connected components is much faster, works with big graphs

```

```

309 # In addition, not subject to deprecated parameters unlike Sklearn clustering
310 # Source code for connected components: https://mltblog.com/3UDJ2tR
311
312 param_N = 20 # prespecified number of clusters in word clustering
313 n = len(aux_list)
314 dist_matrix = [[0 for x in range(n)] for y in range(n)]
315 arr_word = []
316
317 i = 0
318 for word1 in aux_list:
319     arr_word.append(word1)
320     j = 0
321     for word2 in aux_list:
322         key = (word1, word2)
323         if key in hash_pairs:
324             # hash_pairs is based on similarity; dist_matrix = 1 - hash_pairs is distance
325             dist_matrix[i][j] = 1 - hash_pairs[(word1, word2)]
326         else:
327             # assign maximum possible distance if i, j are not linked
328             dist_matrix[i][j] = 1.00 # maximum possible distance
329         j = j+1
330     i = i+1
331
332 #- Clustering, two models: hierachical and k-medoids, based on distance matrix
333
334 from sklearn.cluster import AgglomerativeClustering
335 hierarch = AgglomerativeClustering(n_clusters=param_N, linkage='average').fit(dist_matrix)
336
337 # !pip install scikit-learn-extra
338 from sklearn_extra.cluster import KMedoids
339 kmmedoids = KMedoids(n_clusters=param_N, random_state=0).fit(dist_matrix)
340
341 #- Now showing the clusters obtained from each model
342
343 def show_clusters(model, hash_titles, arr_titles, arr_pv):
344
345     groups = model.labels_
346     hash_group_words = {}
347     for k in range(len(groups)):
348         group = groups[k]
349         word = arr_word[k]
350         if group in hash_group_words:
351             hash_group_words[group] = (*hash_group_words[group], word)
352         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 dendrogram 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 = {}
419         rhash[word] = pv_rel
420         reversed_hash_titles[titleID] = rhash
421
422 # Now predicting pv
423
424 observed = []
425 predicted = []
426 missed = 0
427 n = 0
428 param_W1 = 1 # low value increases overfitting, should be >= 1
429 param_W2 = 2.00 # chosen to minimize error between pv and estimated_pv
430
431 for titleID in reversed_hash_titles:
432     pv = get_article_pv(titleID, arr_pv)
433     rhash = reversed_hash_titles[titleID]
434     n += 1
435     count = 0
436     sum = 0
437     for word in rhash:
438         weight = hash_words_count[word]
439         booster = 1.00
440         if '**' in word:
441             booster = 1.00 # test a different value
442         elif '^' in word:
443             booster = 1.00 # test a different value
444         if weight > param_W1:
445             count += booster * (1/weight)**param_W2
446             sum += booster * (1/weight)**param_W2 * rhash[word]
447     if count > 0:
448         estimated_pv = sum / count
449     else:
450         missed += 1
451         category = arr_categories[titleID]
452         estimated_pv = category_pv[category]
453         observed.append(pv)
454         predicted.append(estimated_pv)
455
456 observed = np.array(observed)
457 predicted = np.array(predicted)
458 mean_pv = np.mean(observed)
459 min_loss = 999999999.99
460 param_Z = 0.00

```

```

461
462     for test_param_Z in np.arange(-0.50, 0.50, 0.05):
463
464         scaled_predicted = predicted + test_param_Z * (predicted - mean_pv)
465         loss = 0
466         for q in (.10, .25, .50, .75, .90):
467             delta_ecdf = abs(np.quantile(observed,q)-np.quantile(scaled_predicted,q))
468             if delta_ecdf > loss:
469                 loss = delta_ecdf
470         if loss < min_loss:
471             min_loss = loss
472         param_Z = test_param_Z
473
474     predicted = predicted + param_Z * (predicted - mean_pv)
475     loss = min_loss
476     mean_estimated_pv = np.mean(predicted)
477     mean_error = np.mean(np.abs(observed-predicted))
478     correl = np.corrcoef(observed, predicted)
479
480     plt.axline((min(observed),min(observed)),(max(observed),max(observed)),c='red')
481     plt.scatter(predicted, observed, s=0.2, c ="lightgray", alpha = 1.0)
482
483     plt.show()
484
485     print()
486     print("== PREDICTIONS\n")
487     print("Predicted vs observed pageviews, for 4000 articles\n")
488     print("Loss: %6.3f" %(loss))
489     print("Missed titles [with pv estimated via category]: ",missed, "out of", n)
490     print("Mean pv (observed) : %8.3f" %(mean_pv))
491     print("Mean pv (estimated): %8.3f" %(mean_estimated_pv))
492     print("Mean absolute error: %8.3f" %(mean_error))
493     print("Correl b/w observed and estimated pv: %8.3f" %(correl[0][1]))
494     print()
495     print("Observed quantiles (left) vs prediction-based (right)")
496     print("P.10: %8.3f %8.3f" %(np.quantile(observed, .10), np.quantile(predicted, .10)))
497     print("P.25: %8.3f %8.3f" %(np.quantile(observed, .25), np.quantile(predicted, .25)))
498     print("P.50: %8.3f %8.3f" %(np.quantile(observed, .50), np.quantile(predicted, .50)))
499     print("P.75: %8.3f %8.3f" %(np.quantile(observed, .75), np.quantile(predicted, .75)))
500     print("P.90: %8.3f %8.3f" %(np.quantile(observed, .90), np.quantile(predicted, .90)))
501
502     #- Plot normalized pv of articles over time
503
504     y = np.zeros(len(arr_pv))
505     for k in range(len(arr_pv)):
506         y[k] = get_article_pv(k, arr_pv)
507
508     z = np.zeros(len(arr_pv))
509     window = 120 # for moving average
510     for k in range(len(arr_pv)):
511         if k>window < 0:
512             z[k] = np.mean(y[0:k+window])
513         elif k+window > len(arr_pv):
514             z[k] = np.mean(y[k-window:len(arr_pv)-1])
515         else:
516             z[k] = np.mean(y[k-window:k+window])
517
518     plt.plot(range(len(arr_pv)), y, linewidth = 0.2, alpha = 0.5)
519     plt.plot(range(len(arr_pv)), z, linewidth = 0.8, c='red', alpha = 1.0)
520
521     plt.xlim(0, len(arr_pv))
522     #plt.ylim(7, 15)
523     plt.grid(color='red', linewidth = 0.2, linestyle='--')
524     plt.show()

```

Chapter 7

Fast, High-Quality NoGAN for Data Synthetization

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 [8]. 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.

7.1 Project description

The project is based on the material in paper 29, [here](#). The password to download the article is MLT12289058. I included the original source code in section 7.3. It is also on GitHub, [here](#). The goal here is to improve the methodology. The first two steps focus on the multivariate empirical distributions (ECDF), at the core of the Kolmogorov-Smirnov distance (KS) that evaluates the quality of the synthetization.

The project consists of the following steps:

Step 1: ECDF scatterplot. Create a Jupyter notebook for the code in section 7.3, 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 7.1, 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 7.1.

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

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

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

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

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

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

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

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

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

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

In the Python code, I only use 4 features, including the three numerical ones. But the telecom dataset contains many more. Add a few more features, and adjust the hyperparameter vector accordingly. For numerical features, small values in the hyperparameter result in artificial linear boundaries in the scatterplots in Figure 7.2 (produced in section [4.1] and [4.2] in the code). Illustrate this fact by reproducing Figure 7.2 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 [12].

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

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

- Find great hyperparameter vectors using for instance the `smart grid search` technique described in [12]. 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 [11]. 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 7.1 should always be close to the main diagonal. When the two ECDFs are identical, the dots lie exactly on the main diagonal.

7.2 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 7.1, with the new one on the right-hand side. In case of perfect synthetization, all the dots are on the main diagonal.

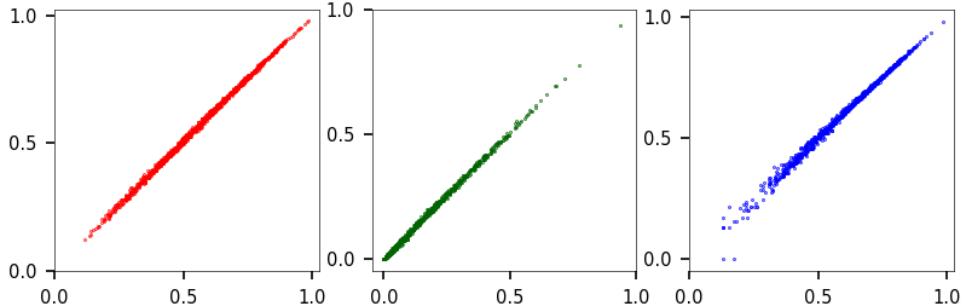


Figure 7.1: ECDF scatterplots (validation vs synthetic) computed three ways

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” [10].

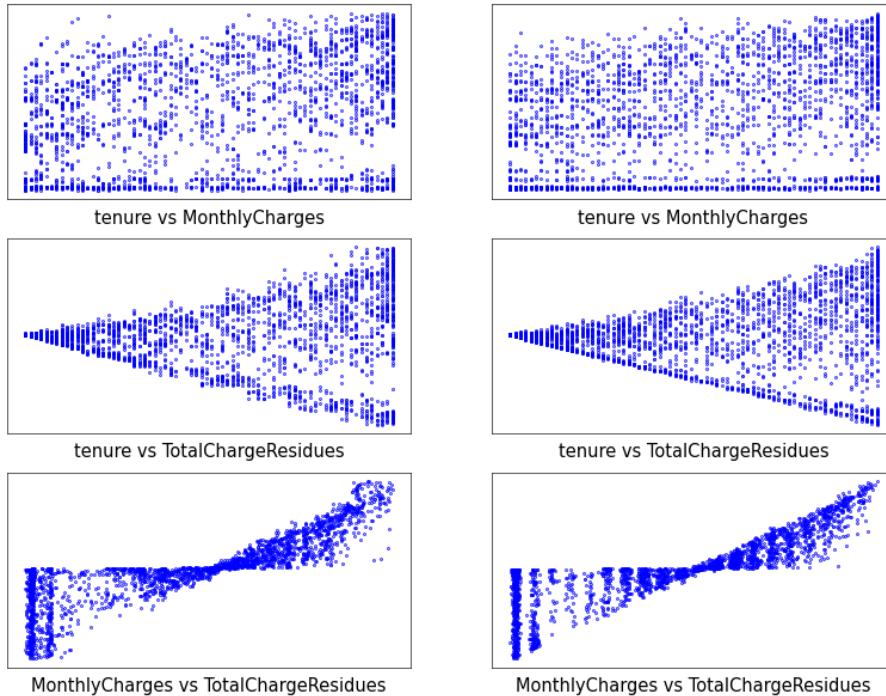


Figure 7.2: Feature scatterplots, synthetic (left) and validation dataset (right)

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 7.2: you can (barely) see some linear borders between

different areas of the plot, in the left middle scatterplot. In particular, on the lower boundary of the cloud point. This happens when the values in the hyperparameter vector, for the features in question, are too low. Here the hyperparameter is $[50, 40, 40, 4]$, with 50 for “tenure”, and 40 for “residues” (the two features in the scatterplot in question). If you decrease these two values to (say) 15, the edge effect will be more pronounced. To the contrary, if you increase it to (say) 80, it won’t be noticeable. High values can lead to overfitting and should be avoided if possible. An implementation of Gaussian NoGAN can be found [here](#). Look at lines 142–150 and 192–200 in the code in question.

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

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

7.3 Python implementation

Explanations about the different steps, including a description of the main variables and tables, can be found in [8]. Compared to GAN, this implementation requires very few library functions and only three imports: Numpy, Statsmodels, and Pandas. This significantly reduces incompatibilities between library versions, and increases the chance that you can run it “as is” on any platform, without impacting your own environment. The code `NoGAN.py` is also available on GitHub, [here](#).

```

1 import numpy as np
2 import pandas as pd
3 import matplotlib.pyplot as plt
4 from matplotlib import pyplot
5 from statsmodels.distributions.empirical_distribution import ECDF
6
7 #--- [1] read data and only keep features and observations we want
8
9 #- [1.1] utility functions
10
11 def string_to_numbers(string):
12
13     string = string.replace("[", " ")
14     string = string.replace("]", " ")
15     string = string.replace(" ", " ")
16     arr = string.split(',')
17     arr = [eval(i) for i in arr]
18     return(arr)
19
20 def category_to_integer(category):
21     if category == 'Yes':
22         integer = 1
23     elif category == 'No':
24         integer = 0
25     else:
26         integer = 2
27     return(integer)
28
29 #- [1.2] read data
30
31 url = "https://raw.githubusercontent.com/VincentGranville/Main/main/Telecom.csv"
32 data = pd.read_csv(url)
33 features = ['tenure', 'MonthlyCharges', 'TotalCharges', 'Churn']
34 data['Churn'] = data['Churn'].map(category_to_integer)
35 data['TotalCharges'].replace(' ', np.nan, inplace=True)
36 data.dropna(subset=['TotalCharges'], inplace=True) # remove missing data
37 print(data.head())

```

```

38 print (data.shape)
39 print (data.columns)
40
41 #- [1.3] transforming TotalCharges to TotalChargeResidues, add to dataframe
42
43 arr1 = data['tenure'].to_numpy()
44 arr2 = data['TotalCharges'].to_numpy()
45 arr2 = arr2.astype(float)
46 residues = arr2 - arr1 * np.sum(arr2) / np.sum(arr1) # also try arr2/arr1
47 data['TotalChargeResidues'] = residues
48
49 #- [1.4] set seed for replicability
50
51 pd.core.common.random_state(None)
52 seed = 105
53 np.random.seed(seed)
54
55 #- [1.5] initialize hyperparameters (bins_per_feature), select features
56
57 features = ['tenure', 'MonthlyCharges', 'TotalChargeResidues', 'Churn']
58 bins_per_feature = [50, 40, 40, 4]
59
60 bins_per_feature = np.array(bins_per_feature).astype(int)
61 data = data[features]
62 print(data.head())
63 print (data.shape)
64 print (data.columns)
65
66 #- [1.6] split real dataset into training and validation sets
67
68 data_training = data.sample(frac = 0.5)
69 data_validation = data.drop(data_training.index)
70 data_training.to_csv('telecom_training_vg2.csv')
71 data_validation.to_csv('telecom_validation_vg2.csv')
72
73 nobs = len(data_training)
74 n_features = len(features)
75 eps = 0.0000000001
76
77
78 #--- [2] create synthetic data
79
80 #- [2.1] create quantile table pc_table2, one row for each feature
81
82 pc_table2 = []
83 for k in range(n_features):
84     label = features[k]
85     incr = 1 / bins_per_feature[k]
86     pc = np.arange(0, 1 + eps, incr)
87     arr = np.quantile(data_training[label], pc, axis=0)
88     pc_table2.append(arr)
89
90 #- [2.2] create/update bin for each obs [layer 1]
91 #     Faster implementation: replace 'while' loop by dichotomic search
92
93 npdata = pd.DataFrame.to_numpy(data_training[features])
94 bin_count = {} # number of obs per bin
95 bin_obs = {} # list of obs in each bin, separated by "~", stored as a string
96 for obs in npdata:
97     key = []
98     for k in range(n_features):
99         idx = 0
100        arr = pc_table2[k] # percentiles for feature k
101        while obs[k] >= arr[idx] and idx < bins_per_feature[k]:
102            idx = idx + 1
103        idx = idx - 1 # lower bound for feature k in bin[key] attached to obs
104        key.append(idx)
105        skey = str(key)
106        if skey in bin_count:
107            bin_count[skey] += 1
108            bin_obs[skey] += "~" + str(obs)
109        else:
110            bin_count[skey] = 1
111            bin_obs[skey] = str(obs)
112
113 #- [2.3] generate nobs_synth observations (if mode = FixedCounts, nobs_synth = nobs)

```

```

114
115 def random_bin_counts(n, bin_count):
116     # generate multinomial bin counts with same expectation as real counts
117     pvals = []
118     for skey in bin_count:
119         pvals.append(bin_count[skey]/nobs)
120     return(np.random.multinomial(n, pvals))
121
122 def get_obs_in_bin(bin_obs, skey):
123     # get list of observations (real data) in bin skey, also return median
124     arr_obs = []
125     arr_obs_aux = (bin_obs[skey]).split('`')
126     for obs in arr_obs_aux:
127         obs = ' '.join(obs.split())
128         obs = obs.replace("[ ", " ")
129         obs = obs.replace(" [", " ")
130         obs = obs.replace(" ]", " ")
131         obs = obs.replace("]", " ")
132         obs = obs.split(' ')
133         obs = (np.array(obs)).astype(float)
134         arr_obs.append(obs)
135     arr_obs = np.array(arr_obs)
136     median = np.median(arr_obs, axis = 0)
137     return(arr_obs, median)
138
139
140 mode = 'RandomCounts' # (options: 'FixedCounts' or 'RandomCounts')
141 if mode == 'RandomCounts':
142     nobs_synth = nobs
143     bin_count_random = random_bin_counts(nobs_synth, bin_count)
144     ikey = 0
145
146 data_synth = []
147 bin_counter = 0
148
149 for skey in bin_count:
150
151     if mode == 'FixedCounts':
152         count = bin_count[skey]
153     elif mode == 'RandomCounts':
154         count = bin_count_random[ikey]
155         ikey += 1
156     key = string_to_numbers(skey)
157     L_bounds = []
158     U_bounds = []
159     bin_counter += 1
160
161     for k in range(n_features):
162         arr = pc_table2[k]
163         L_bounds.append(arr[key[k]])
164         U_bounds.append(arr[1 + key[k]])
165
166     # sample new synth obs (new_obs) in rectangular bin skey, uniformly
167     # try other distrib, like multivariate Gaussian around bin median
168     # the list of real observations in bin[skey] is stored in obs_list (numpy array)
169     # median is the vector of medians for all obs in bin skey
170
171     obs_list, median = get_obs_in_bin(bin_obs, skey) # not used in this version
172
173     for i in range(count):
174         new_obs = np.empty(n_features) # synthesized obs
175         for k in range(n_features):
176             new_obs[k] = np.random.uniform(L_bounds[k],U_bounds[k])
177         data_synth.append(new_obs)
178
179     str_median = str(["%8.2f" % number for number in median])
180     str_median = str_median.replace("/", "")
181     print("bin ID = %5d | count = %5d | median = %s | bin key = %s"
182           %(bin_counter, bin_count[skey], str_median, skey))
183
184 data_synth = pd.DataFrame(data_synth, columns = features)
185
186 # apply floor function (not round) to categorical/ordinal features
187 data_synth['Churn'] = data_synth['Churn'].astype('int')
188 data_synth['tenure'] = data_synth['tenure'].astype('int')
189

```

```

190 print(data_synth)
191 data_synth.to_csv('telecom_synth_vg2.csv')
192
193
194 #--- [3] Evaluation synthetization using joint ECDF & Kolmogorov-Smirnov distance
195
196 # dataframes: df = synthetic; data = real data,
197 # compute multivariate ecdf on validation set, sort it by value (from 0 to 1)
198
199 #- [3.1] compute ecdf on validation set (to later compare with that on synth data)
200
201 def compute_ecdf(dataframe, n_nodes, adjusted):
202
203     # Monte-Carlo: sampling n_nodes locations (combos) for ecdf
204     #   - adjusted correct for sparsity in high ecdf, but is sparse in low ecdf
205     #   - non-adjusted is the other way around
206     # for faster computation: pre-compute percentiles for each feature
207     # for faster computation: optimize the computation of n_nodes SQL-like queries
208
209     ecdf = {}
210
211     for point in range(n_nodes):
212
213         if point % 100 == 0:
214             print("sampling ecdf, location = %4d (adjusted = %s):" % (point, adjusted))
215             combo = np.random.uniform(0, 1, n_features)
216
217             if adjusted:
218                 combo = combo***(1/n_features)
219
220             z = [] # multivariate quantile
221             query_string = ""
222             for k in range(n_features):
223                 label = features[k]
224                 dr = data_validation[label]
225                 percentile = combo[k]
226                 z.append(eps + np.quantile(dr, percentile))
227
228                 if k == 0:
229                     query_string += "{} <= {}".format(label, z[k])
230                 else:
231                     query_string += " and {} <= {}".format(label, z[k])
232
233             countifs = len(data_validation.query(query_string))
234             if countifs > 0:
235                 ecdf[str(z)] = countifs / len(data_validation)
236
237     ecdf = dict(sorted(ecdf.items(), key=lambda item: item[1]))
238
239     # extract table with locations (ecdf argument) and ecdf values:
240     #   - cosmetic change to return output easier to handle than ecdf
241
242     idx = 0
243     arr_location = []
244     arr_value = []
245     for location in ecdf:
246         value = ecdf[location]
247         location = string_to_numbers(location)
248         arr_location.append(location)
249         arr_value.append(value)
250         idx += 1
251
252     print("\n")
253     return(arr_location, arr_value)
254
255 n_nodes = 1000 # number of random locations in feature space, where ecdf is computed
256 reseed = False
257 if reseed:
258     seed = 555
259     np.random.seed(seed)
260 arr_location1, arr_value1 = compute_ecdf(data_validation, n_nodes, adjusted = True)
261 arr_location2, arr_value2 = compute_ecdf(data_validation, n_nodes, adjusted = False)
262
263 #- [3.2] comparison: synthetic (based on training set) vs real (validation set)
264
265 def ks_delta(SyntheticData, locations, ecdf_ValidationSet):
266
267     # SyntheticData is a dataframe

```

```

266     # locations are the points in the feature space where ecdf is computed
267     # for the validation set, ecdf values are stored in ecdf_ValidationSet
268     # here we compute ecdf for the synthetic data, at the specified locations
269     # output ks_max in [0, 1] with 0 = best, 1 = worst
270
271     ks_max = 0
272     ecdf_real = []
273     ecdf_synth = []
274     for idx in range(len(locations)):
275         location = locations[idx]
276         value = ecdf_ValidationSet[idx]
277         query_string = ""
278         for k in range(n_features):
279             label = features[k]
280             if k == 0:
281                 query_string += "{} <= {}".format(label, location[k])
282             else:
283                 query_string += " and {} <= {}".format(label, location[k])
284         countifs = len(SyntheticData.query(query_string))
285         synth_value = countifs / len(SyntheticData)
286         ks = abs(value - synth_value)
287         ecdf_real.append(value)
288         ecdf_synth.append(synth_value)
289         if ks > ks_max:
290             ks_max = ks
291     # print("location ID: %d | ecdf_real: %.4f | ecdf_synth: %.4f"
292     #       %(idx, value, synth_value))
293     return(ks_max, ecdf_real, ecdf_synth)
294
295 df = pd.read_csv('telecom_synth_vg2.csv')
296 ks_max1, ecdf_real1, ecdf_synth1 = ks_delta(df, arr_location1, arr_value1)
297 ks_max2, ecdf_real2, ecdf_synth2 = ks_delta(df, arr_location2, arr_value2)
298 ks_max = max(ks_max1, ks_max2)
299 print("Test ECDF Kolmogorof-Smirnov dist. (synth. vs valid.): %.4f" %(ks_max))
300
301 #- [3.3] comparison: training versus validation set
302
303 df = pd.read_csv('telecom_training_vg2.csv')
304 base_ks_max1, ecdf_real1, ecdf_synth1 = ks_delta(df, arr_location1, arr_value1)
305 base_ks_max2, ecdf_real2, ecdf_synth2 = ks_delta(df, arr_location2, arr_value2)
306 base_ks_max = max(base_ks_max1, base_ks_max2)
307 print("Base ECDF Kolmogorof-Smirnov dist. (train. vs valid.): %.4f" %(base_ks_max))
308
309
310 #--- [4] visualizations
311
312 def vg_scatter(df, feature1, feature2, counter):
313
314     # customized plots, subplot position based on counter
315
316     label = feature1 + " vs " + feature2
317     x = df[feature1].to_numpy()
318     y = df[feature2].to_numpy()
319     plt.subplot(3, 2, counter)
320     plt.scatter(x, y, s = 0.1, c ="blue")
321     plt.xlabel(label, fontsize = 7)
322     plt.xticks([])
323     plt.yticks([])
324     #plt.ylim(0,70000)
325     #plt.xlim(18,64)
326     return()
327
328 def vg_histo(df, feature, counter):
329
330     # customized plots, subplot position based on counter
331
332     y = df[feature].to_numpy()
333     plt.subplot(2, 3, counter)
334     min = np.min(y)
335     max = np.max(y)
336     binBoundaries = np.linspace(min, max, 30)
337     plt.hist(y, bins=binBoundaries, color='white', align='mid', edgecolor='red',
338               linewidth = 0.3)
339     plt.xlabel(feature, fontsize = 7)
340     plt.xticks([])
341     plt.yticks([])
```

```

342     return()
343
344 import matplotlib.pyplot as plt
345 import matplotlib as mpl
346 mpl.rcParams['axes.linewidth'] = 0.3
347
348 #- [4.1] scatterplots for Churn = 'No'
349
350 dfs = pd.read_csv('telecom_synth_vg2.csv')
351 dfs.drop(dfs[dfs['Churn'] == 0].index, inplace = True)
352 dfv = pd.read_csv('telecom_validation_vg2.csv')
353 dfv.drop(dfv[dfv['Churn'] == 0].index, inplace = True)
354
355 vg_scatter(dfs, 'tenure', 'MonthlyCharges', 1)
356 vg_scatter(dfv, 'tenure', 'MonthlyCharges', 2)
357 vg_scatter(dfs, 'tenure', 'TotalChargeResidues', 3)
358 vg_scatter(dfv, 'tenure', 'TotalChargeResidues', 4)
359 vg_scatter(dfs, 'MonthlyCharges', 'TotalChargeResidues', 5)
360 vg_scatter(dfv, 'MonthlyCharges', 'TotalChargeResidues', 6)
361 plt.show()
362
363 #- [4.2] scatterplots for Churn = 'Yes'
364
365 dfs = pd.read_csv('telecom_synth_vg2.csv')
366 dfs.drop(dfs[dfs['Churn'] == 1].index, inplace = True)
367 dfv = pd.read_csv('telecom_validation_vg2.csv')
368 dfv.drop(dfv[dfv['Churn'] == 1].index, inplace = True)
369
370 vg_scatter(dfs, 'tenure', 'MonthlyCharges', 1)
371 vg_scatter(dfv, 'tenure', 'MonthlyCharges', 2)
372 vg_scatter(dfs, 'tenure', 'TotalChargeResidues', 3)
373 vg_scatter(dfv, 'tenure', 'TotalChargeResidues', 4)
374 vg_scatter(dfs, 'MonthlyCharges', 'TotalChargeResidues', 5)
375 vg_scatter(dfv, 'MonthlyCharges', 'TotalChargeResidues', 6)
376 plt.show()
377
378 #- [4.3] ECDF scatterplot: validation set vs. synth data
379
380 plt.xticks(fontsize=7)
381 plt.yticks(fontsize=7)
382 plt.scatter(ecdf_real1, ecdf_synth1, s = 0.1, c ="blue")
383 plt.scatter(ecdf_real2, ecdf_synth2, s = 0.1, c ="blue")
384 plt.show()
385
386 #- [4.4] histograms, Churn = 'No'
387
388 dfs = pd.read_csv('telecom_synth_vg2.csv')
389 dfs.drop(dfs[dfs['Churn'] == 0].index, inplace = True)
390 dfv = pd.read_csv('telecom_validation_vg2.csv')
391 dfv.drop(dfv[dfv['Churn'] == 0].index, inplace = True)
392 vg_histo(dfs, 'tenure', 1)
393 vg_histo(dfs, 'MonthlyCharges', 2)
394 vg_histo(dfs, 'TotalChargeResidues', 3)
395 vg_histo(dfv, 'tenure', 4)
396 vg_histo(dfv, 'MonthlyCharges', 5)
397 vg_histo(dfv, 'TotalChargeResidues', 6)
398 plt.show()
399
400 #- [4.5] histograms, Churn = 'Yes'
401
402 dfs = pd.read_csv('telecom_synth_vg2.csv')
403 dfs.drop(dfs[dfs['Churn'] == 1].index, inplace = True)
404 dfv = pd.read_csv('telecom_validation_vg2.csv')
405 dfv.drop(dfv[dfv['Churn'] == 1].index, inplace = True)
406 vg_histo(dfs, 'tenure', 1)
407 vg_histo(dfs, 'MonthlyCharges', 2)
408 vg_histo(dfs, 'TotalChargeResidues', 3)
409 vg_histo(dfv, 'tenure', 4)
410 vg_histo(dfv, 'MonthlyCharges', 5)
411 vg_histo(dfv, 'TotalChargeResidues', 6)
412 plt.show()

```

Chapter 8

Hierarchical Bayesian NoGAN for Data Synthetization

Deep learning models such as generative adversarial networks (GAN) require a lot of computing power, and are thus expensive. What if you could produce better data synthetizations, in a fraction of the time, with explainable AI and substantial cost savings? Very different from the tree-based NoGAN described in chapter 7, this new technology, abbreviated as NoGAN2, relies on resampling, an hierarchical sequence of runs, simulated annealing, and batch processing to boost performance, both in terms of output quality and time requirements. No neural network is involved.

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

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

8.1 Methodology

The highly generic NoGAN2 technology described here is a powerful and better alternative to neural network synthetization methods for tabular data. It is very different from NoGAN described in chapter 7. Each offers its own benefits. For reasons that will soon become obvious, I also call this new method **hierarchical deep resampling**, but I will use the word NoGAN2 for simplicity. Again, it is designed to run much faster, compared to training **generative adversarial networks** (GAN). Also, the quality of the generated data is far superior to almost all other products available on the market.

Many evaluation metrics to measure faithfulness have critical flaws, sometimes rating synthetic data as excellent, when it is actually a failure, due to relying on low-dimensional indicators. This is especially noticeable on the circle dataset in [7], where all GANs are evaluated as excellent, yet most fail to generate the correct distribution with points lying on two concentric circles. As I did with NoGAN, here again I fix this problem using the full **multivariate empirical distribution** (ECDF). It produces much better evaluations. Performance is measured via **cross-validation** in all my examples.

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

8.1.1 Base algorithm

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

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

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

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

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

8.1.2 Loss function

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

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

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

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

where

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

Here σ stands for the standard deviation. The means and standard deviations are computed respectively on the training set for $\rho_2[i, j]$, and on the synthesized data under construction for $\rho'_2[i, j]$. However they are computed only once: just after the initial synthetization obtained in Step 1. They will remain unchanged throughout the Step 2 iterations. Also, $\rho_2[i, j]$ and $\rho'_2[i, j]$ are correlation coefficients, assuming categorical values are encoded as integers.

- Typical functions are $g_2(x) = x$, denoted as $g_{21}(x)$, and $g_2(x) = x^2$, denoted as $g_{22}(x)$. Likewise for $h_2(x)$. In the current implementation, there is no three-way interactions (the g_3 and h_3). From there, the partial distance functions for two-way interactions are defined as $\Delta_2[i, j] = |\rho_2[i, j] - \rho'_2[i, j]|$, for $1 \leq i, j \leq m$.

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

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

where the sum is not over i, j (assumed to be fixed), but over all the vector elements in each Hadamard product, with each element corresponding to a specific observation. Same for $Q'_2[i, j]$ and $\rho_2[i, j], \rho'_2[i, j]$, each broken down into two pieces. This leads to

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

- Now let $\Delta_2[i, j] = \max(\alpha_1 \cdot \Delta_{21}[i, j], \alpha_2 \cdot \Delta_{22}[i, j])$ with $\alpha_1, \alpha_2 \geq 0$ and $\alpha_1 + \alpha_2 = 1$. If taking into account two-way interactions only, the **loss function** Δ_2 is the sum of $\Delta_2[i, j]$ over all features i, j with $i \neq j$. An alternative version $\tilde{\Delta}_2$ consists in using the maximum rather than the sum.

The goal was to design a loss function that maps one-to-one to the multivariate **Kolmogorov-Smirnov distance** (KS) between synthesized and real data, as KS is the perfect metric for evaluation purposes. We wanted the mapping to be continuous and order-preserving. By focusing on two-way interactions only, and breaking down g_2, h_2 into two components only ($g_{21}, g_{22}, h_{21}, h_{22}$), we get an approximate solution to this problem. But in practice, it works much better and faster than alternatives based on neural networks. In chapter 9, I use a different mapping, based on counts per bin, to incorporate the best from the NoGAN technique discussed in chapter 7.

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

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

8.1.3 Hyperparameters and convergence

By contrast to GAN and related techniques, in the current implementation of NoGAN2, the loss function always decreases after swapping two values, albeit more and more slowly over time. It does not oscillate. Swaps become rarer and rarer over time, making progress towards the optimum extremely slow or even impossible after a while. Getting stuck is similar to the **vanishing gradient** issue [Wiki] in neural networks dealing with tabular data. However, the problem and side effects are typically less pronounced in NoGAN2. In section 8.2, I discuss how to reduce the risk of getting stuck too early. A future version will occasionally allow swaps even if they result in increasing the loss. It will be performed using a **simulated annealing** schedule [Wiki], and result in an oscillating loss, with amplitudes decreasing over time, just like in a non-failing GAN.

In practice, despite the combinatorial nature of the problem, a good solution is obtained once each value in the initial synthesized data has been proposed for a swap a couple of times. So the computational complexity is proportional to the number of observations to generate, multiplied by the number of features. Generating small **batches** of observations separately (by splitting the initial synthesized data in a number of batches), can further improve performance. It is discussed in section 8.2.

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

In the current implementation, features are processed jointly rather than separately. Fine-tuning **hyperparameters** helps you accelerate the speed of convergence and avoid a local optima. I explain in section 8.2 how the algorithm can **auto-tune** itself. Now I discuss the main hyperparameters:

- The **seed** of the random number generator involved can have an impact on the final synthetization. You may try different seeds to improve the convergence or quality of the results.
- The number of **batches** discussed earlier.
- The weights α_1 attached to g_{21}, h_{21} and α_2 attached to g_{22}, h_{22} in the loss function, see section 8.1.2.
- The **loss function** itself, especially the functions $g_{21}, h_{21}, g_{22}, h_{22}$ and any additional functions that you may use. For instance, g_{23} and h_{23} , then requiring the extra weight α_3 .

- When randomly picking up a feature to swap two values, by default, each feature has the same probability of being selected. You can change these probabilities, selecting feature i with probability p_i instead. The probability vector $P = [p_1, \dots, p_m]$ is a core hyperparameter. Here m being the number of features.
- If one of the probabilities is set to zero, then the feature in question will never be selected, and won't be updated. You can use this property as follows. Say you have three different groups of features, A B, and C. For instance numerical features in A, ordinal features in B, and continuous features in C. Run NoGAN2 three times: first with non-zero probabilities in A only, then with non-zero probabilities in B only, then with non-zero probabilities in C only.

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

The same idea can be used in traditional generative adversarial networks, leading to Hi-GAN, for **hierarchical GAN**. See for instance [4] and [21]. Finally if only one function $g_2(x) = h_2(x) = x$ is used in the loss function, then NoGAN2 is equivalent to the **copula** technique discussed in chapter 10 in [15]. In that case, starting with an initial synthetization with correct marginal distributions but zero cross-correlations, it reconstructs the correct correlation structure attached to the features.

8.1.4 Acknowledgments

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

I am also very grateful to [Rajiv Iyer](#) for turning the multivariate empirical distribution (ECDF) and related KS distance computations into a production code Python library, available [here](#). You can install it with `pip install genAI-evaluation`. See how I use it in section 8.4. Rajiv also compared NoGAN2 with CTGAN on the student dataset. All comparisons are favorable to NoGAN2.

8.2 Case studies

I synthesized two datasets. In the first one, observations consist of customer statistics from a Telecom company, to assess attrition rates by segment. I used 4 features and 3500 observations. The feature "total charges" is highly correlated to "tenure" (how many months the customer stayed with the company), and caused more problems when combined with "monthly charges", visible only in higher dimensional plots. I first used GAN and Wasserstein GAN [31], with several data transforms and various tricks to improve the synthetization. In particular, I performed a **principal component analysis** (PCA) on the training set, then synthesized the transformed data, then applied the inverse PCA transform. The improvements, while substantial and discussed in chapter 9 in [14], were not good enough. It led to the creation of NoGAN discussed in chapter 7, and then NoGAN2 presented here. Both provide satisfactory results, in a short amount of time, and with little if any fine-tuning.

The second dataset consists of student admission metrics. The outcome is the success rate at college. I used 21 features and 4400 observations, with a mix of categorical and numerical features, as well as missing data. The structure of the missing data is very simple. In this case, using a separate NoGAN2 for the 300 observations with missing values, is the easiest solution. For more scattered missing values, see how to do it in section 6.4.2 in [14]. The goal here is not to impute missing values, but to replicate them correctly. For **imputation** done via synthetization, see the GAIN technique (a variant of GAN) in [35]. This dataset was first synthesized with CTGAN. Some improvement was obtained with NoGAN without using the proper encoding for categorical variables. With proper encoding, the performance was increased by several orders of magnitudes. With NoGAN2, you get good performance (much better than GAN) without any encoding or data transform, in a fraction of the time required by neural networks methods, thus with considerable cost savings.

All the tests involve splitting the real data into two parts: training and validation. The training set is used to generate the synthetic data, while the **validation set** (also called **holdout**) is used to assess its quality: how well it mimics the **joint empirical distribution** (ECDF) observed in the validation set. To achieve this goal, I implemented the powerful **Kolmogorov-Smirnov distance** to compare the two multivariate ECDFs: training versus validation. Unlike most other distances in the marketplace, it does not miss high dimensional patterns; it will not mark a synthetization as good if it is a failure, undetected by classic evaluation metrics. See section 8.2.5

Before diving into the actual synthetizations in sections 8.2.1 and 8.2.2, I want to mention a few important points. First and obvious, do not check whether two identical values (in two different rows, for a specific feature) benefit from being swapped. Instead, look for different rows with distinct values. This will speed up the algorithm when dealing with numerous binary features such as dummy variables. Then, before fine tuning hyperparameters, read section 8.2.4. Finally, to generate data outside the observation range, you need to stretch the Numpy [quantile function](#) used in the algorithm, or write your own: it does not generate values below the observed minimum, or above the observed maximum. See how to do it in [1].

8.2.1 Synthesizing the student dataset

The dataset is available on GitHub, [here](#). I also use it in the Python code in section 8.4. In this section, I explain some techniques to accelerate performance, both in terms of speed and quality of the synthetization. Figures 8.1 and 8.2 show the evolution of the loss function and cumulated number of swaps, respectively during the first and last 500k iterations. These iterations start after the initial synthetization, that is, after Step 1 in section 8.1.1. An iteration consists of randomly selecting a feature, then randomly selecting two rows, and check whether or not the two values (one from each row) must be swapped. The swap takes place if it decreases the loss function. In that case, it increases the number of swaps by one.

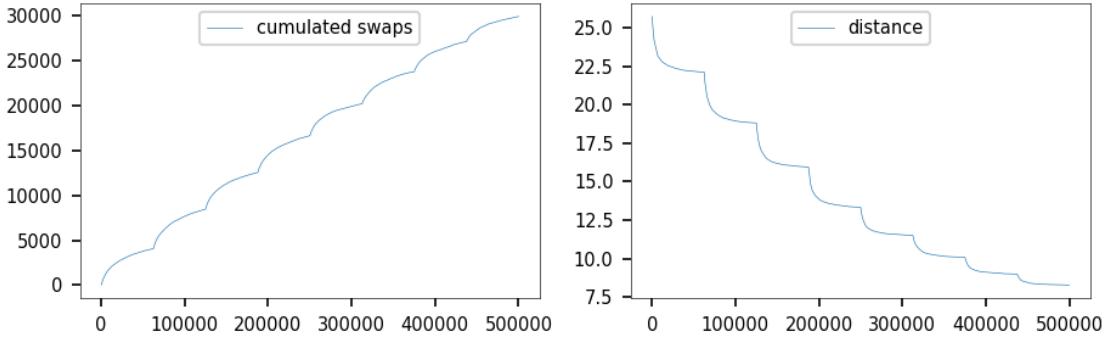


Figure 8.1: Number of swaps (left) and loss function (right) over time: first run

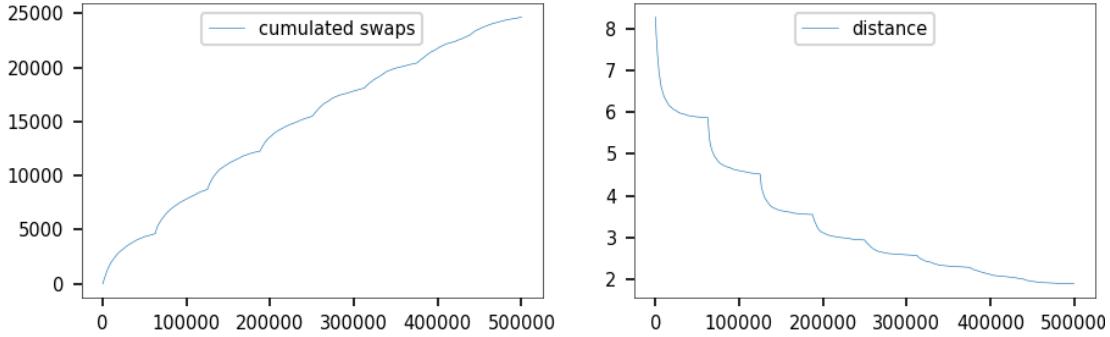


Figure 8.2: Number of swaps (left) and loss function (right) over time: second run

The first mechanism consists in using [batches](#). I first generate 2000 synthetic observations in the initial synthetization. Then I split the generated data into 8 subsets called batches. Resampling – the swaps – are performed one batch at a time, swapping values internally from within a batch, without modifying the other batches. Because swaps tend to become more and more rare over time, this technique accelerates convergence: this is noticeable in Figures 8.1 and 8.2, where the loss function drops quickly at the beginning of each batch but then taper off. By using 8 batches rather than one, overall the loss function spends more time dropping sharply, than staying in a low entropy mode. It also keeps the number of swaps almost constant over time, rather than dropping to zero in late iterations. Bumps happen when moving from one batch to the next one. By contrast, Figure 8.5 (Telecom dataset) shows how slowly the loss function decreases after the initial drop, when using one batch only.

The initial synthetization (before starting deep resampling) correctly replicates each univariate distribution in the training set. Since deep resampling does not modify at all the univariate distributions, all the histograms in Figure 8.4 always look very good. Thus it is more important to focus on the scatterplots in Figure 8.3. They look good as well. Despite the 21 features, this data set is actually easy to synthesize, at least with NoGAN2. I

used two runs, see part 8 in the code featured in section 8.4: the first run mostly for the numerical features, and the second run for the remaining features, conditionally to the first run. Thus we are not synthesizing the two sets of features separately, but jointly via a conditional (Bayesian) mechanism. Hyperparameter values are set to default, except for the weights. You may want to look at my choice for the functions g_{22} and h_{22} , respectively g and h in the code.

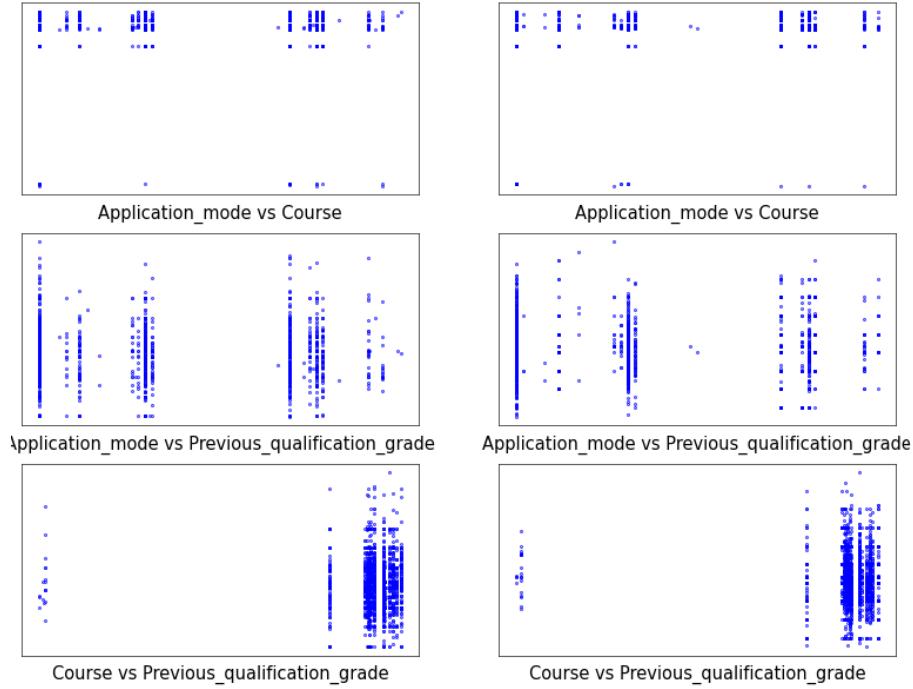


Figure 8.3: Synthetization (left) vs validation set (right), student dataset

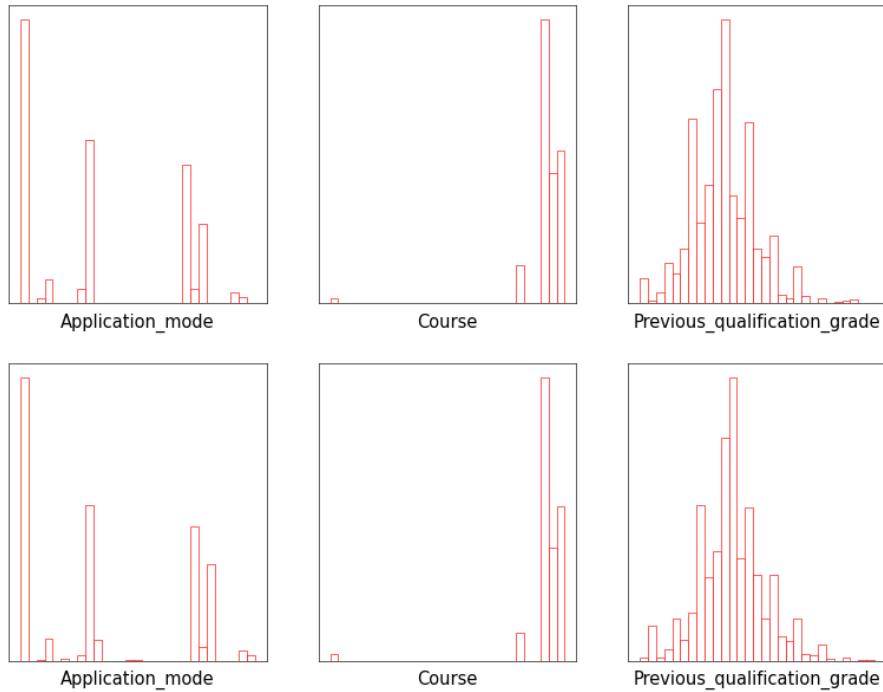


Figure 8.4: Synthetisation (top) vs validation set (bottom), student dataset

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

This is within the range of what is considered a good synthetization. For a definition of KS and Base KS, see section 8.2.5.

8.2.2 Synthesizing the Telecom dataset

The Python code and dataset are on GitHub, [here](#). With 7000 observations and 4 features including a categorical one, the data is easy to synthesize with NoGAN2. However, despite considerable efforts and testing various improvements, we were unable to produce synthetizations of the same quality with generative adversarial networks (GAN, WGAN and so on). Only NoGAN (chapter 7) matches the quality and speed of execution of NoGAN2. The details are in chapter 9 in [14] entitled “How to Fix a Failing Adversarial Network”. In short, the problem arises when the features “TotalCharges” and “Tenure” are both present. The latter represents the number of months a customer has stayed with the company; it is highly correlated to the former. But even after decorrelating and scaling transforms, the GAN results, while significantly improved, are well below the performance of NoGAN2.

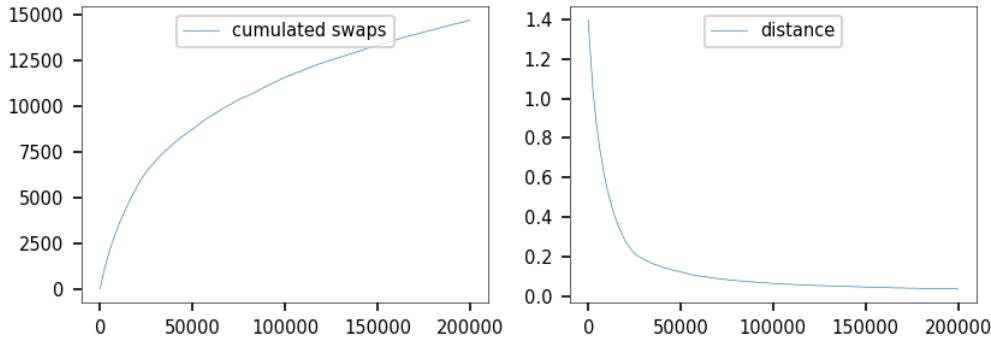


Figure 8.5: Number of swaps (left) and loss function (right) over time

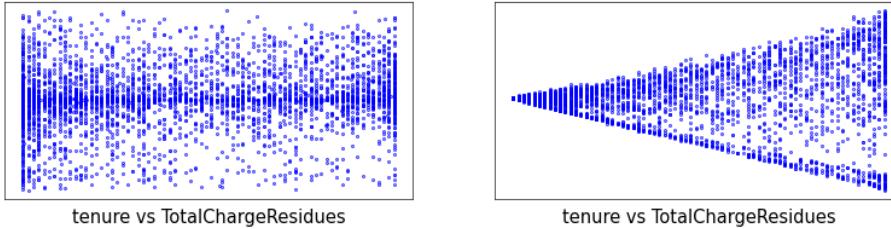


Figure 8.6: Synthetisation with one term in loss function (left), vs real data (right)

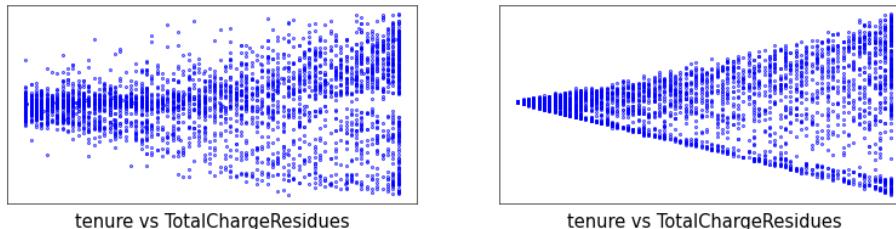


Figure 8.7: Synthetisation with two terms in loss function (left), vs real data (right)

With this dataset, only one run of deep resampling was needed, processing all features at once. However, unequal weights $\alpha_1 \neq \alpha_2$ in `weights`, combined with uneven values in the probability vector `hyperParam`, significantly improves the quality of the synthesized data. In particular, oversampling the problematic feature “TotalCharges” really helps. This is done by setting `hyperParam[2]` to an unusually large value. Such fine-tuning is easy to automate, and intuitive.

Finally, Figure 8.7 shows why we need the two terms in the loss function, rather than just the first one. It significantly increases the quality, when compared to Figure 8.6. Interestingly, the functions used in the second term are $g_{22}(x) = h_{22}(x) = |x|$. It impacts correlations involving at least one feature with both positive and

negative values: in this case, “TotalChargeResidues”, which is the decorrelated version of “TotalCharges”. Here, KS = 0.0379, and Base KS = 0.0176. As always, the univariate distributions are well rendered, see Figure 8.8.

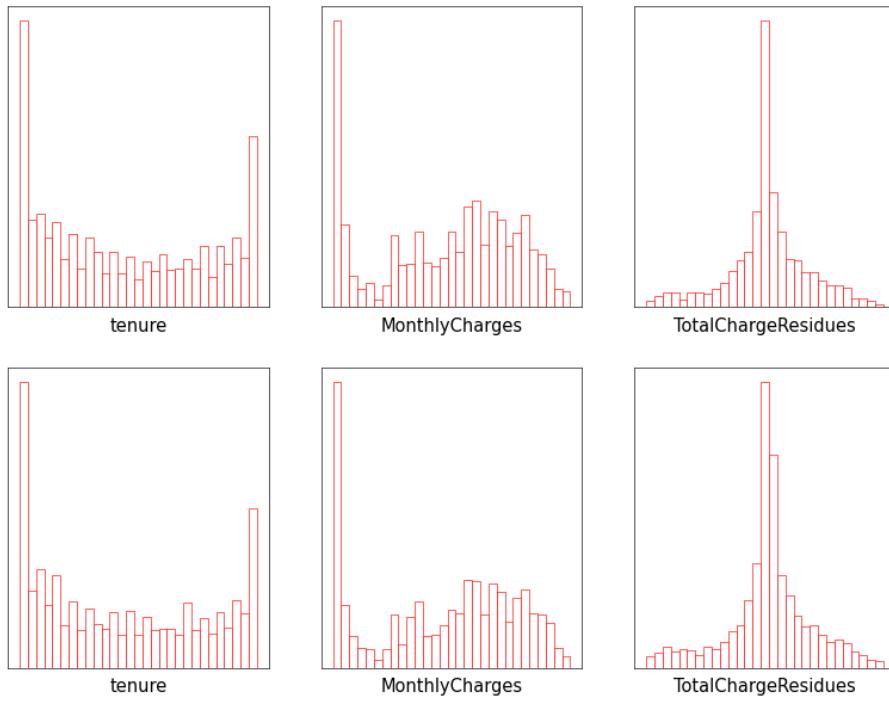


Figure 8.8: Synthetisation (top) vs validation set (bottom), Telecom dataset

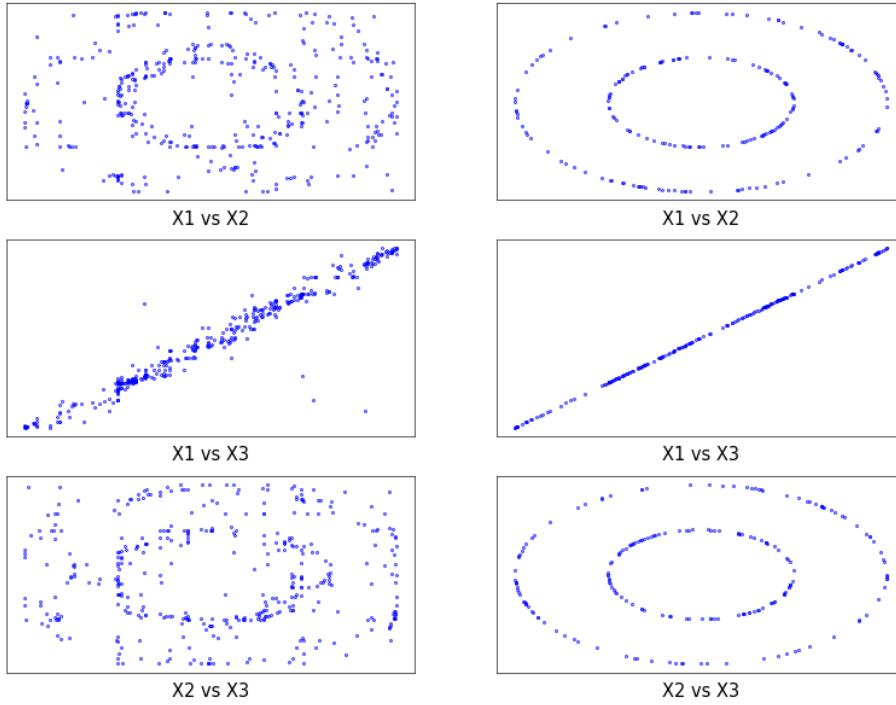


Figure 8.9: Synthetization (left) vs validation set (right), circle dataset

8.2.3 Other case studies

Besides the Telecom and education domains (the student data), I also explored datasets in the healthcare, insurance and cybersecurity industries. Finally, I tested NoGAN2 on the challenging circle dataset, see Figure 8.9. The Python code and datasets are on GitHub, [here](#). Look for the documents starting with DeepResampling in the filename, for instance DeepResampling_circle.py. These examples have both numerical and categorical features. Each one illustrates specific options and hyperparameters. The peculiar cybersecurity case is

discussed in the project textbook offered to participants in my Gen AI certification program, available [here](#). The diabetes data also includes missing values, properly rendered by the synthetization.

The general conclusion is that NoGAN2 trains much faster than neural network equivalents, and consistently provide better results, when evaluated using the best distance in a cross-validation setting: the Kolmogorov-Smirnov distance (KS) based on the multivariate ECDF (joint empirical distribution). This evaluation metric is now available as an open-source Python library (`genAI-evaluation`). It captures all the interdependency patterns among the features. By contrast, distances currently used on the marketplace are not implemented in full multivariate mode. It frequently leads to false negatives: synthetizations rated as excellent, when they are actually very poor. This is magnified when synthesizing the circle dataset, as discussed in [7].

NoGAN2 requires less fine-tuning than GAN and other deep neural network techniques. Also, fine-tuning is straightforward, thanks to the explainable nature of the whole system. This allows for auto-tuning as discussed in see section 8.2.4. In the end, for tabular data generation, the only real competitor to NoGAN2 is NoGAN explored in chapter 7, also developed in my laboratory. Both require very little bandwidth, significantly reducing costs while providing better results.

Finally, categorical features can benefit from being encoded as dummy variables, or from a loss function where standard correlations are replaced by Cramér's V [\[Wiki\]](#) or similar statistics [2]. Most examples also include a **label feature** (the rightmost column in the dataset) to categorize each observation into various clusters. For instance, the label feature in the circle dataset indicates whether an observation belongs to the inner or outer circle in Figure 8.9. Using a label feature significantly improves the performance. This is also true for GAN. The challenges with the circle data is the very small number of observations, combined with nearly duplicate features.

8.2.4 Auto-tuning the hyperparameters

Before discussing **auto-tuning**, let's look at the correlation coefficients involved in the loss function, using the same notation as in section 8.1.2. For illustration purposes, see Table 8.1 corresponding to the Telecom dataset with 4 features. The table is organized as follows:

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

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

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

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

The next hyperparameter is the probability vector $P = [p_1, \dots, p_m]$ with $p_1 + \dots + p_m = 1$, also described in section 8.1.3. It specifies how frequently each feature is selected for a potential swap (swapping the values from two random rows, in the column corresponding to the feature in question). In Table 8.1, the worst correlation discrepancies involve features $\{0, 2\}$ and $\{1, 2\}$. This suggests that feature 2 is more challenging. Increasing p_2 may reduce the error. However, it will penalize other features. To choose the optimum p_2 , you can let the algorithm do the job, using an adaptive p_2 automatically fine-tuned over time to minimize the loss. The same applies to all the probabilities in P , as well as the two weight parameters in W .

Finally, the best improvements are obtained by running NoGAN2 twice. First, you split the set of features into two subsets A and B. In the first run, you optimize the loss function for features in A. Then, in the second

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

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

Table 8.1: Correlations in the loss function, Telecom dataset

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

8.2.5 Evaluation with multivariate ECDF and KS distance

The [multivariate empirical distribution](#) (ECDF) and corresponding [Kolmogorov-Smirnov distance](#) (KS) between the two ECDFs – validation set vs synthesized data – is described in details, along with the Python implementation, in chapter 7. It is now available as the GenAI-Evalution Python library on PyPi, [here](#). In this section, I provide a brief overview.

The joint or *multivariate* ECDF has remained elusive to this day. It is a rather non-intuitive object, hard to visualize and handle even in two dimensions, let alone in higher dimensions with categorical features. For that reason, the [empirical probability density function](#) (EPDF) is more popular: it is associated to [mixture models](#), frequently embedded into neural networks, or the [Hellinger distance](#) to evaluate the quality of synthesized data. However, the ECDF is more robust. Then, while the Hellinger distance also generalizes to multivariate EPDFs, in practice all the implementations are one-dimensional, with the distance computed for each feature separately. The Hellinger equivalent based on ECDFs instead, is indeed the KS distance. It belongs to a class of measures known as [integral probability metrics](#) [24, 30].

It is said that KS does not generalize to the multivariate case. Thus its total absence in applications when the dimension is higher than two, despite the fact that it is the best metric to fully capture all the dependencies among features, especially the non-linear ones. Asymptotics for the multivariate KS distance were investigated only recently [28], while an algorithm for fast computation of the multivariate ECDF is discussed in [22]. To my knowledge, this is the first practical implementation in high dimensions, tested on real datasets.

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

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

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

Try increasing `n_nodes` (the number of nodes) from 10^3 to 10^4 and 10^5 , to see when KS stabilizes. The scatter plot in Figure 8.10 is produced in part 10 in the code featured in section 8.4, while the value of `n_nodes` is set in part 9.

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

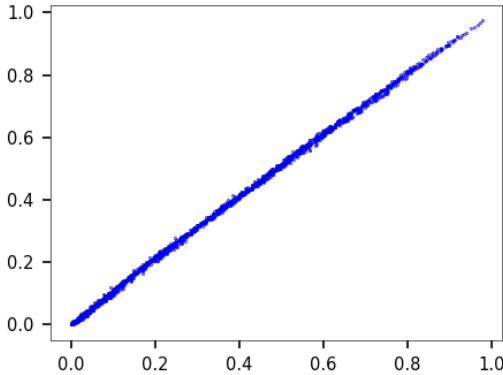


Figure 8.10: ECDF scatterplot, synthetic vs validation set

8.3 Conclusion

The context is tabular data generation. NoGAN2, by contrast to NoGAN and despite not being based on neural networks, shares a lot of properties with GAN. For instance, the two terms in the loss function fight against each other in a way similar to discriminator vs generator in GAN; I use weights to give each term a fair chance to win, as in Wasserstein GAN, but with a min-max approach rather than averaging. Batch processing has its equivalent in neural networks, where it is called the same. The first layer in NN architecture corresponds to the first run in the resampling algorithm. Using a second run is similar to having a second (deep) layer in deep neural networks. The quality of the synthetization may be poor even if the loss function reaches zero: this happens when the loss is not a good proxy to the full multivariate KS distance used to evaluate the quality; then it requires working with a different loss function. Or the algorithm may fail to reach a zero loss, getting stuck in a local minimum. In that case, you should change the loss function or the hyperparameters, or allowing the loss to go up and down with a general downward trend. This mechanism is sometimes referred to as simulated annealing [16]. The fast computations, just like in GAN, are based on efficient use of tensors. Features causing problems in GAN (with high entropy or highly correlated to other features) cause similar problems in NoGAN2. It can be addressed using data transforms (called transformers in LLM) such as feature decorrelation, scaling, and PCA, followed by inverse transforms. All these properties make NoGAN2 a good sandbox to test and improve generative adversarial networks.

Despite the similarities, there are major differences, besides the absence of gradient descent. First, NoGAN2 is much faster, thought it can be tempting to use a large number of iterations, to increase the number of swaps (the analog of neuron activation). It usually improves the results, although only marginally after a while. A better strategy is to use batches. Compared to GAN, the convergence issues are much less pronounced. Indeed, I managed to synthesize all my datasets rather quickly and with superior quality, consistently. NoGAN2 is also much more stable, and easier to fine-tune because of the intuitive nature of the hyperparameters. It illustrates explainable AI. Because of this, it leads to auto-tuning, where fine-tuning is automated, possibly with reinforcement learning, in little time. Ideally, you want to use the KS distance as your loss function, rather than a proxy mimicking the approximation of a multivariate function by the first few terms of its Taylor series. The challenge is how to design an efficient architecture to achieve this goal. NoGAN3 discussed in chapter 9 solves this problem, using bin counts in the loss function as in NoGAN (chapter 7), rather than correlations between transformed features.

Finally, NoGAN2 starts with an initial synthetization where all the features, taken separately, are perfectly replicated but lacking the cross-dependencies structure. It gives NoGAN2 a good head start, as all univariate statistical summaries are preserved throughout the algorithm. The deep resampling consists of reconstructing

these interdependencies. If the loss function has one term only, then NoGAN2 is simply the [copula method](#), implemented differently. The addition of a second term allows you to replicate not just feature correlations, but much more complex dependencies. A second or third run makes it an [hierarchical Bayesian model](#).

8.4 Python implementation

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

```

1 import numpy as np
2 import pandas as pd
3
4 import matplotlib.pyplot as plt
5 import matplotlib as mpl
6 import genai_evaluation as ge
7 from matplotlib import pyplot
8 from statsmodels.distributions.empirical_distribution import ECDF
9 import warnings
10 warnings.simplefilter("ignore")
11
12
13 #--- [1] read data and only keep features and observations we want
14
15 def category_to_integer(category):
16     if category == 'Enrolled':
17         integer = 1
18     elif category == 'Dropout':
19         integer = 0
20     else:
21         integer = 2
22     return(integer)
23
24 #- [1.1] read data
25
26 url = "https://raw.githubusercontent.com/VincentGranville/Main/main/students.csv"
27 # data = pd.read_csv('students_C2_full_nogan.csv')
28 data = pd.read_csv(url)
29 print(data)
30
31 # all features used here
32
33 features = [
34     'Application_mode',           # 0, categorical
35     'Course',                   # 1, categorical
36     'Previous_qualification_grade', # 2, ordinal
37     'Mother_qualification',      # 3, categorical
38     'Father_qualification',      # 4, categorical
39     'Mother_occupation',         # 5, categorical
40     'Father_occupation',         # 6, categorical
41     'Admission_grade',          # 7, ordinal
42     'Tuition_fees_up_to_date',   # 8, binary
43     'Age_at_enrollment',         # 9, ordinal [outliers]
44     'Curricular_units_1st_sem_evaluations', # 10, ordinal [0 = missing]
45     'Curricular_units_1st_sem_approved', # 11, ordinal [0 = missing]
46     'Curricular_units_1st_sem_grade', # 12, ordinal [0 = missing]
47     'Curricular_units_2nd_sem_enrolled', # 13, ordinal [0 = missing]
48     'Curricular_units_2nd_sem_evaluations', # 14, ordinal [0 = missing]
49     'Curricular_units_2nd_sem_approved', # 15, ordinal [0 = missing]
50     'Curricular_units_2nd_sem_grade', # 16, ordinal [0 = missing]
51     'Unemployment_rate',          # 17, ordinal
52     'Inflation_rate',            # 18, ordinal, can be < 0
53     'GDP',                      # 19, ordinal, can be < 0
54     'Target'                     # 20, categorical [outcome]
55 ]
56
57 data['Target'] = data['Target'].map(category_to_integer)
58
59 # remove rows with missing values
60 data = data[data['Curricular_units_1st_sem_evaluations'] != 0]
61
62 print(data.head())

```

```

63 print (data.shape)
64 print (data.columns)
65
66 #- [1.2] set seed for replicability
67
68 pd.core.common.random_state(None)
69 seed = 106 ## 105
70 np.random.seed(seed)
71
72 #- [1.3] select features
73
74 data = data[features]
75 data = data.sample(frac = 1) # shuffle rows to break artificial sorting
76
77 #- [1.4] split real dataset into training and validation sets
78
79 data_training = data.sample(frac = 0.5)
80 data_validation = data.drop(data_training.index)
81 data_training.to_csv('training_vg2.csv')
82 data_validation.to_csv('validation_vg2.csv')
83 data_train = pd.DataFrame.to_numpy(data_training)
84
85 nobs = len(data_training)
86 n_features = len(features)
87
88
89 #--- [2] create initial synthetic data
90
91 def create_initial_synth(nobs_synth):
92
93     eps = 0.000000001
94     n_features = len(features)
95     data_synth = np.empty(shape=(nobs_synth,n_features))
96
97     for i in range(nobs_synth):
98         pc = np.random.uniform(0, 1 + eps, n_features)
99         for k in range(n_features):
100             label = features[k]
101             data_synth[i, k] = np.quantile(data_training[label], pc[k], axis=0)
102     return(data_synth)
103
104
105 #--- [3] loss functions Part 1
106
107 def compute_univariate_stats():
108
109     # 'dt' for training data, 'ds' for synth. data
110
111     # for first term in loss function
112     dt_mean = np.mean(data_train, axis=0)
113     dt_stdev = np.std(data_train, axis=0)
114     ds_mean = np.mean(data_synth, axis=0)
115     ds_stdev = np.std(data_synth, axis=0)
116
117     # for g(arr)
118     dt_mean1 = np.mean(g(data_train), axis=0)
119     dt_stdev1 = np.std(g(data_train), axis=0)
120     ds_mean1 = np.mean(g(data_synth), axis=0)
121     ds_stdev1 = np.std(g(data_synth), axis=0)
122
123     # for f(arr)
124     dt_mean2 = np.mean(h(data_train), axis=0)
125     dt_stdev2 = np.std(h(data_train), axis=0)
126     ds_mean2 = np.mean(h(data_synth), axis=0)
127     ds_stdev2 = np.std(h(data_synth), axis=0)
128
129     values = [dt_mean, dt_stdev, ds_mean, ds_stdev,
130               dt_mean1, dt_stdev1, ds_mean1, ds_stdev1,
131               dt_mean2, dt_stdev2, ds_mean2, ds_stdev2]
132     return(values)
133
134 def initialize_cross_products_tables():
135
136     # the core structure for fast computation when swapping 2 values
137     # 'dt' for training data, 'ds' for synth. data
138     # 'prod' is for 1st term in loss, 'prod12' for 2nd term

```

```

139
140     dt_prod = np.empty(shape=(n_features,n_features))
141     ds_prod = np.empty(shape=(n_features,n_features))
142     dt_prod12 = np.empty(shape=(n_features,n_features))
143     ds_prod12 = np.empty(shape=(n_features,n_features))
144
145     for k in range(n_features):
146         for l in range(n_features):
147             dt_prod[l, k] = np.dot(data_train[:,l], data_train[:,k])
148             ds_prod[l, k] = np.dot(data_synth[:,l], data_synth[:,k])
149             dt_prod12[l, k] = np.dot(g(data_train[:,l]), h(data_train[:,k]))
150             ds_prod12[l, k] = np.dot(g(data_synth[:,l]), h(data_synth[:,k]))
151     products = [dt_prod, ds_prod, dt_prod12, ds_prod12]
152     return(products)
153
154
155 #---- [4] loss function Part 2: managing loss function
156
157 # Weights hyperparameter:
158 #
159 #   1st value is for 1st term in loss function, 2nd value for 2nd term
160 #   each value should be between 0 and 1, all adding to 1
161 #   works best when loss contributions from each term are about the same
162
163 #- [4.1] loss function contribution from features (k, l) jointly
164
165 # before calling functions in sections [4.1], [4.2] and [4.3], first initialize
166 # by calling compute_univariate_stats() and compute_cross_products() before;
167 # this initialization needs to be done only once at the beginning
168
169 def get_distance(k, l, weights):
170
171     dt_prodn = dt_prod[k, l] / nobs
172     ds_prodn = ds_prod[k, l] / nobs_synth
173     dt_r = (dt_prodn - dt_mean[k]*dt_mean[l]) / (dt_stdev[k]*dt_stdev[l])
174     ds_r = (ds_prodn - ds_mean[k]*ds_mean[l]) / (ds_stdev[k]*ds_stdev[l])
175
176     dt_prodn12 = dt_prod12[k, l] / nobs
177     ds_prodn12 = ds_prod12[k, l] / nobs_synth
178     dt_r12 = (dt_prodn12 - dt_mean1[k]*dt_mean2[l]) / (dt_stdev1[k]*dt_stdev2[l])
179     ds_r12 = (ds_prodn12 - ds_mean1[k]*ds_mean2[l]) / (ds_stdev1[k]*ds_stdev2[l])
180
181     # dist = weights[0]*abs(dt_r - ds_r) + weights[1]*abs(dt_r12 - ds_r12)
182     dist = max(weights[0]*abs(dt_r - ds_r), weights[1]*abs(dt_r12 - ds_r12))
183     return(dist, dt_r, ds_r, dt_r12, ds_r12)
184
185 def total_distance(weights, flagParam):
186
187     eval = 0
188     max_dist = 0
189     super_max = 0
190     lmax = n_features
191
192     for k in range(n_features):
193         if symmetric:
194             lmax = k
195         for l in range(lmax):
196             if l != k and flagParam[k] > 0 and flagParam[l] >0:
197                 values = get_distance(k, l, weights)
198                 dist2 = max(abs(values[1] - values[2]), abs(values[3] - values[4]))
199                 eval += values[0]
200                 if values[0] > max_dist:
201                     max_dist = values[0]
202                 if dist2 > super_max:
203                     super_max = dist2
204     return(eval, max_dist, super_max)
205
206 #- [4.2] updated loss function when swapping rows idx1 and idx2 in feature k
207 #       contribution from feature l jointly with k
208
209 def get_new_distance(k, l, idx1, idx2, weights):
210
211     tmp1_k = data_synth[idx1, k]
212     tmp2_k = data_synth[idx2, k]
213     tmp1_l = data_synth[idx1, l]
214     tmp2_l = data_synth[idx2, l]

```

```

215
216     #-- first term of loss function
217
218     remove1 = tmp1_k * tmp1_l
219     remove2 = tmp2_k * tmp2_l
220     add1 = tmp1_k * tmp2_l
221     add2 = tmp2_k * tmp1_l
222     new_ds_prod = ds_prod[l, k] - remove1 - remove2 + add1 + add2
223
224     dt_prodn = dt_prod[k, l] / nobs
225     ds_prodn = new_ds_prod / nobs_synth
226     dt_r = (dt_prodn - dt_mean[k]*dt_mean[l]) / (dt_stdev[k]*dt_stdev[l])
227     ds_r = (ds_prodn - ds_mean[k]*ds_mean[l]) / (ds_stdev[k]*ds_stdev[l])
228
229     #-- second term of loss function
230
231     remove1 = g(tmp1_k) * h(tmp1_l)
232     remove2 = g(tmp2_k) * h(tmp2_l)
233     add1 = g(tmp1_k) * h(tmp2_l)
234     add2 = g(tmp2_k) * h(tmp1_l)
235     new_ds_prod12 = ds_prod12[k, l] - remove1 - remove2 + add1 + add2
236
237     dt_prodn12 = dt_prod12[k, l] / nobs
238     ds_prodn12 = new_ds_prod12 / nobs_synth
239     dt_r12 = (dt_prodn12 - dt_mean1[k]*dt_mean2[l]) / (dt_stdev1[k]*dt_stdev2[l])
240     ds_r12 = (ds_prodn12 - ds_mean1[k]*ds_mean2[l]) / (ds_stdev1[k]*ds_stdev2[l])
241
242     #--
243
244     # new_dist = weights[0]*abs(dt_r - ds_r) + weights[1]*abs(dt_r12 - ds_r12)
245     new_dist = max(weights[0]*abs(dt_r - ds_r), weights[1]*abs(dt_r12 - ds_r12))
246     return(new_dist, dt_r, ds_r, dt_r12, ds_r12)
247
248
249 #- [4.3] update prod tables after swapping rows idx1 and idx2 in feature k
250 #      update impacting feature l jointly with k
251
252 def update_product(k, l, idx1, idx2):
253
254     tmp1_k = data_synth[idx1, k]
255     tmp2_k = data_synth[idx2, k]
256     tmp1_l = data_synth[idx1, l]
257     tmp2_l = data_synth[idx2, l]
258
259     #-- first term of loss function
260
261     remove1 = tmp1_k * tmp1_l
262     remove2 = tmp2_k * tmp2_l
263     add1 = tmp1_k * tmp2_l
264     add2 = tmp2_k * tmp1_l
265     ds_prod[k, l] = ds_prod[k, l] - remove1 - remove2 + add1 + add2
266     ds_prod[l, k] = ds_prod[k, l]
267
268     #-- second term of loss function
269
270     remove1 = g(tmp1_k) * h(tmp1_l)
271     remove2 = g(tmp2_k) * h(tmp2_l)
272     add1 = g(tmp1_k) * h(tmp2_l)
273     add2 = g(tmp2_k) * h(tmp1_l)
274     ds_prod12[k, l] = ds_prod12[k, l] - remove1 - remove2 + add1 + add2
275
276     remove1 = h(tmp1_k) * g(tmp1_l)
277     remove2 = h(tmp2_k) * g(tmp2_l)
278     add1 = h(tmp1_k) * g(tmp2_l)
279     add2 = h(tmp2_k) * g(tmp1_l)
280     ds_prod12[l, k] = ds_prod12[l, k] - remove1 - remove2 + add1 + add2
281
282     return()
283
284
285     #--- [5] feature sampling
286
287 def sample_feature(mode, hyperParameter):
288
289     # Randomly pick up one column (a feature) to swap 2 values from 2 random rows
290     # One feature is assumed to be in the right order, thus ignored

```

```

291     if mode == 'Equalized':
292         u = np.random.uniform(0, 1)
293         cutoff = hyperParam[0]
294         feature = 0
295         while cutoff < u:
296             feature += 1
297             cutoff += hyperParam[feature]
298     else:
299         feature = np.random.randint(1, n_features) # ignore feature 0
300     return(feature)
301
302
303
304 #--- [6] functions: deep synthetization, plot history, print stats
305
306 #- [6.1] main function
307
308 def deep_resampling(hyperParameter, run, loss_type, n_batches,
309                      n_iter, nobs_synth, weights, flagParam, mode):
310
311     # main function
312
313     batch = 0
314     batch_size = nobs_synth // n_batches
315     niter_per_batch = n_iter // n_batches
316     lower_row = 0
317     upper_row = batch_size
318     nswaps = 0
319     cgain = 0 # cumulative gain
320
321     arr_swaps = []
322     arr_history_quality = []
323     arr_history_max_dist = []
324     arr_time = []
325     print()
326
327     for iter in range(n_iter):
328
329         k = sample_feature(mode, hyperParameter)
330         batch = iter // niter_per_batch
331         lower_row = batch * batch_size
332         upper_row = lower_row + batch_size
333         idx1 = np.random.randint(lower_row, upper_row) % nobs_synth
334         tmp1 = data_synth[idx1, k]
335         tmp2 = tmp1
336         counter = 0
337         while tmp2 == tmp1 and counter < 20:
338             idx2 = np.random.randint(lower_row, upper_row) % nobs_synth
339             tmp2 = data_synth[idx2, k]
340             counter += 1
341
342         g_param = 0.5
343         h_param = g_param
344
345         delta = 0
346         delta2 = 0
347         for l in range(n_features):
348             if l != k and flagParam[l] > 0:
349                 values = get_distance(k, l, weights)
350                 delta += values[0]
351                 if values[0] > delta2:
352                     delta2 = values[0]
353                 if not symmetric: # if functions g, h are different
354                     values = get_distance(l, k, weights)
355                     delta += values[0]
356                     if values[0] > delta2:
357                         delta2 = values[0]
358
359         new_delta = 0
360         new_delta2 = 0
361         for l in range(n_features):
362             if l != k and flagParam[l] > 0:
363                 values = get_new_distance(k, l, idx1, idx2, weights)
364                 new_delta += values[0]
365                 if values[0] > new_delta2:
366                     new_delta2 = values[0]

```

```

367         if not symmetric: # if functions g, h are different
368             values = get_new_distance(l, k, idx1, idx2, weights)
369             new_delta += values[0]
370             if values[0] > new_delta2:
371                 new_delta2 = values[0]
372
373         if loss_type == 'sum_loss':
374             gain = delta - new_delta
375         elif loss_type == 'max_loss':
376             gain = delta2 - new_delta2
377         if gain > 0:
378             cgain += gain
379             for l in range(n_features):
380                 if l != k:
381                     update_product(k, l, idx1, idx2)
382                     # update_product(l, k, idx1, idx2)
383             data_synth[idx1, k] = tmp2
384             data_synth[idx2, k] = tmp1
385             nswaps += 1
386
387         if iter % 500 == 0:
388             quality, max_dist, super_max = total_distance(weights, flagParam)
389             arr_swaps.append(nswaps)
390             arr_history_quality.append(quality)
391             arr_history_max_dist.append(max_dist)
392             arr_time.append(iter)
393             if iter % 5000 == 0:
394                 print("Iter: %6d Distance: %8.4f SupDist: %8.4f Gain: %8.4f Swaps: %6d"
395                     %(iter, quality, super_max, cgain, nswaps))
396
397     return(nswaps, arr_swaps, arr_history_quality, arr_history_max_dist, arr_time)
398
399 #- [6.2] save synthetic data, show some stats
400
401 def evaluate_and_save(filename, weights, run, flagParam):
402
403     print("\nMetrics after deep resampling\n")
404     quality, max_dist, super_max = total_distance(weights, flagParam)
405     print("Distance: %8.4f" %(quality))
406     print("Max Dist: %8.4f" %(max_dist))
407
408     data_synthetic = pd.DataFrame(data_synth, columns = features)
409     data_synthetic.to_csv(filename)
410     print("\nSynthetic data, first 10 rows\n",data_synthetic.head(10))
411
412     print("\nBivariate feature correlation:")
413     print("....dt_xx for training set, ds_xx for synthetic data")
414     print("....xx_r for correl[k, l], xx_r12 for correl[g(k), h(l)]\n")
415     print("%2s %2s %8s %8s %8s %8s %8s"
416         % ('k', 'l', 'dist', 'dt_r', 'ds_r', 'dt_r12', 'ds_r12'))
417     print("-----")
418     for k in range(n_features):
419         for l in range(n_features):
420             condition = (flagParam[k] >0 and flagParam[l] > 0)
421             if k != l and condition:
422                 values = get_distance(l, k, weights)
423                 dist = values[0]
424                 dt_r = values[1] # training, 1st term of loss function
425                 ds_r = values[2] # synth., 1st term of loss function
426                 dt_r12 = values[3] # training, 2nd term of loss function
427                 ds_r12 = values[4] # synth., 2nd term of loss function
428                 print("%2d %2d %8.4f %8.4f %8.4f %8.4f %8.4f"
429                     % (k, l, dist, dt_r, ds_r, dt_r12, ds_r12))
430
431     return()
432
433 #- [6.3] plot history of loss function, and cumulated number of swaps
434
435 def plot_history(history):
436
437     arr_swaps = history[1]
438     arr_history_quality = history[2]
439     arr_history_max_dist = history[3]
440     arr_time = history[4]
441
442     mpl.rcParams['axes.linewidth'] = 0.3
443     plt.rc('xtick',labelsize=7)

```

```

443     plt.rc('ytick',labelsize=7)
444     plt.xticks(fontsize=7)
445     plt.yticks(fontsize=7)
446     plt.subplot(1, 2, 1)
447     plt.plot(arr_time, arr_swaps, linewidth = 0.3)
448     plt.legend(['cumulated swaps'], fontsize="7",
449                loc ="upper center", ncol=1)
450     plt.subplot(1, 2, 2)
451     plt.plot(arr_time, arr_history_quality, linewidth = 0.3)
452     # plt.plot(arr_time, arr_history_max_dist, linewidth = 0.3)
453     plt.legend(['distance'], fontsize="7",
454                loc ="upper center", ncol=1)
455     plt.show()
456     return()
457
458
459 #--- [7] initializations
460
461 #- create intitial synthetization
462
463 nobs_synth = 2000
464 data_synth = create_initial_synth(nobs_synth)
465
466 #- specify 2nd part of loss function (argument is a number or array)
467
468 # do not use g(arr) = f(arr) = arr: this is pre-built already as 1st term in loss fct
469 # these two functions f, g are for the second term in the loss function
470
471 def g(arr):
472     return(1/(0.01 + np.absolute(arr)))
473 def h(arr):
474     return(1/(0.01 + np.absolute(arr)))
475
476 symmetric = True # set to True if functions g and h are identical
477 # 'symmetric = True' twice as fast as 'symmetric = False'
478
479 #- initializations: product tables and univariate stats
480
481 products = initialize_cross_products_tables()
482 dt_prod = products[0]
483 ds_prod = products[1]
484 dt_prod12 = products[2]
485 ds_prod12 = products[3]
486
487 values = compute_univariate_stats()
488 dt_mean = values[0]
489 dt_stdev = values[1]
490 ds_mean = values[2]
491 ds_stdev = values[3]
492 dt_mean1 = values[4]
493 dt_stdev1 = values[5]
494 ds_mean1 = values[6]
495 ds_stdev1 = values[7]
496 dt_mean2 = values[8]
497 dt_stdev2 = values[9]
498 ds_mean2 = values[10]
499 ds_stdev2 = values[11]
500
501
502 #--- [8] deep resampling
503
504 mode = 'Equalized' # options: 'Standard', 'Equalized'
505 eps2 = 0.0
506
507 #- first run
508
509 run = 1
510 n_iter = 500001
511 n_batches = 8
512 loss_type = 'sum_loss' # options: 'max_loss' or 'sum_loss'
513 weights = [0.90, 0.10]
514 hyperParam = np.zeros(len(features))
515 hyperParam[8] = 1
516 hyperParam[10] = 1
517 hyperParam[11] = 1
518 hyperParam[12] = 1

```

```

519 hyperParam[13] = 1
520 hyperParam[14] = 1
521 hyperParam[15] = 1
522 hyperParam[16] = 1
523 hyperParam = hyperParam / np.sum(hyperParam)
524 flagParam = np.ones(len(features))
525 history = deep_resampling(hyperParam, run, loss_type, n_batches, n_iter,
526                             nobs_synth, weights, flagParam, mode)
527 evaluate_and_save('synth_vg2.csv', weights, run, flagParam)
528 plot_history(history)
529
530 #- second run
531
532 run = 2
533 n_iter = 500001
534 n_batches = 8
535 loss_type = 'sum_loss' # options: 'max_loss' or 'sum_loss'
536 weights = [0.90, 0.10]
537 hyperParam = np.ones(len(features))
538 hyperParam[8] = 0
539 hyperParam[10] = 0
540 hyperParam[11] = 0
541 hyperParam[12] = 0
542 hyperParam[13] = 0
543 hyperParam[14] = 0
544 hyperParam[15] = 0
545 hyperParam[16] = 0
546 hyperParam = hyperParam / np.sum(hyperParam)
547 flagParam = np.ones(len(features))
548 history = deep_resampling(hyperParam, run, loss_type, n_batches, n_iter,
549                             nobs_synth, weights, flagParam, mode)
550 evaluate_and_save('synth_vg2.csv', weights, run, flagParam)
551 plot_history(history)
552
553
554 #--- [9] Evaluation synthetization using joint ECDF & Kolmogorov-Smirnov distance
555
556 #     dataframes: df = synthetic; data = real data,
557 #     compute multivariate ecdf on validation set, sort it by value (from 0 to 1)
558
559 print("\nMultivariate ECDF computations...\n")
560 n_nodes = 1000 # number of random locations in feature space, where ecdf is computed
561 seed = 50
562 np.random.seed(seed)
563
564 df_validation = pd.DataFrame(data_validation, columns = features)
565 df_synthetic = pd.DataFrame(data_synth, columns = features)
566 df_training = pd.DataFrame(data_train, columns = features)
567 query_lst, ecdf_val, ecdf_synth = ge.multivariate_ecdf(df_validation, df_synthetic, n_nodes,
568             verbose = True)
569 query_lst, ecdf_val, ecdf_train = ge.multivariate_ecdf(df_validation, df_training, n_nodes, verbose
570             = True)
571 ks = ge.ks_statistic(ecdf_val, ecdf_synth)
572 ks_base = ge.ks_statistic(ecdf_val, ecdf_train)
573 print("Test ECDF Kolmogorof-Smirnov dist. (synth. vs valid.): %6.4f" %(ks))
574 print("Base ECDF Kolmogorof-Smirnov dist. (train. vs valid.): %6.4f" %(ks_base))
575
576 #--- [10] visualizations (based on Matplotlib version: 3.7.1)
577
578 def vg_scatter(df, feature1, feature2, counter):
579
580     # customized plots, subplot position based on counter
581
582     label = feature1 + " vs " + feature2
583     x = df[feature1].to_numpy()
584     y = df[feature2].to_numpy()
585     plt.subplot(3, 2, counter)
586     plt.scatter(x, y, s = 0.1, c ="blue")
587     plt.xlabel(label, fontsize = 7)
588     plt.xticks([])
589     plt.yticks([])
590     #plt.ylim(0,70000)
591     #plt.xlim(18,64)
592     return()

```

```
593
594 def vg_histo(df, feature, counter):
595
596     # customized plots, subplot position based on counter
597
598     y = df[feature].to_numpy()
599     plt.subplot(2, 3, counter)
600     min = np.min(y)
601     max = np.max(y)
602     binBoundaries = np.linspace(min, max, 30)
603     plt.hist(y, bins=binBoundaries, color='white', align='mid', edgecolor='red',
604               linewidth = 0.3)
605     plt.xlabel(feature, fontsize = 7)
606     plt.xticks([])
607     plt.yticks([])
608     return()
609
610 mpl.rcParams['axes.linewidth'] = 0.3
611
612 #- [10.1] scatterplots
613
614 dfs = pd.read_csv('synth_vg2.csv')
615 dfv = pd.read_csv('validation_vg2.csv')
616 vg_scatter(dfs, features[0], features[1], 1)
617 vg_scatter(dfv, features[0], features[1], 2)
618 vg_scatter(dfs, features[0], features[2], 3)
619 vg_scatter(dfv, features[0], features[2], 4)
620 vg_scatter(dfs, features[1], features[2], 5)
621 vg_scatter(dfv, features[1], features[2], 6)
622 plt.show()
623
624 #- [10.2] histograms
625
626 dfs = pd.read_csv('synth_vg2.csv')
627 dfv = pd.read_csv('validation_vg2.csv')
628 vg_histo(dfs, features[0], 1)
629 vg_histo(dfs, features[1], 2)
630 vg_histo(dfs, features[2], 3)
631 vg_histo(dfv, features[0], 4)
632 vg_histo(dfv, features[1], 5)
633 vg_histo(dfv, features[2], 6)
634 plt.show()
```

Chapter 9

Boosting Model Evaluation with Smart Adaptive Loss Functions

Most AI and GenAI algorithms aim at optimizing a criterion. The goal is to get good predictions (see project 6), relevant and exhaustive answers (LLMs) or realistic synthetizations. The quality of the output is assessed using some **evaluation metric**. However, the algorithm tries to minimize a **loss function**. Both are very distinct: one cannot expect to get the best output by optimizing a criterion other than the model evaluation metric.

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

This problem is typically solved with generative adversarial networks (**GANs**), see project 5.2 in [13]. 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. Thus, it amounts to using an **adaptive loss function** [26] that converges to the Hellinger distance. Then, I tested the method using a special framework where the global optimum is known in advance, hoping to retrieve it. The result is spectacular:

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

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

My solution does not use gradient descent: instead, at each step, a random move is made (thus, not based on a continuous path governed by derivatives of a function), but nevertheless always in the right direction: the

move 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 [14]. This is a resampling method that preserves the marginal distributions throughout all iterations, while attempting to approach the joint distribution in the training set, using a continuous loss function as in neural networks. However, here the loss function is replaced by the evaluation metric (Hellinger).
- The granularity of the EDPF lattice support increases over time, resulting in an exponential explosion in the number of bins, especially in high dimensions. However the sparsity increases in the same proportion: the vast majority of bins are empty and not even stored or visited. The number of active bins (stored in hash tables) is not larger than the number of observations, at any given time.
- Pre-computation of square roots, stored in tables. Each of the 2 million atomic updates requires the computation of 16 square roots. The tables in question significantly contribute to speeding up the algorithm.

Most tabular data synthesizers have challenges to generate observations with the prescribed distribution. By replacing the loss function with the evaluation metric, I face the opposite problem: generating observations too similar to those in the training set. So instead of trying to achieve a low value for the Hellinger distance as in GAN, I must do the opposite: preventing the Hellinger distance from being too low. This is accomplished by setting up constraints. I call my approach **constrained synthetization**. See Figure 9.1 where the Hellinger distance H between the synthetic data and training set (real data) is kept above 0.60 at all times.

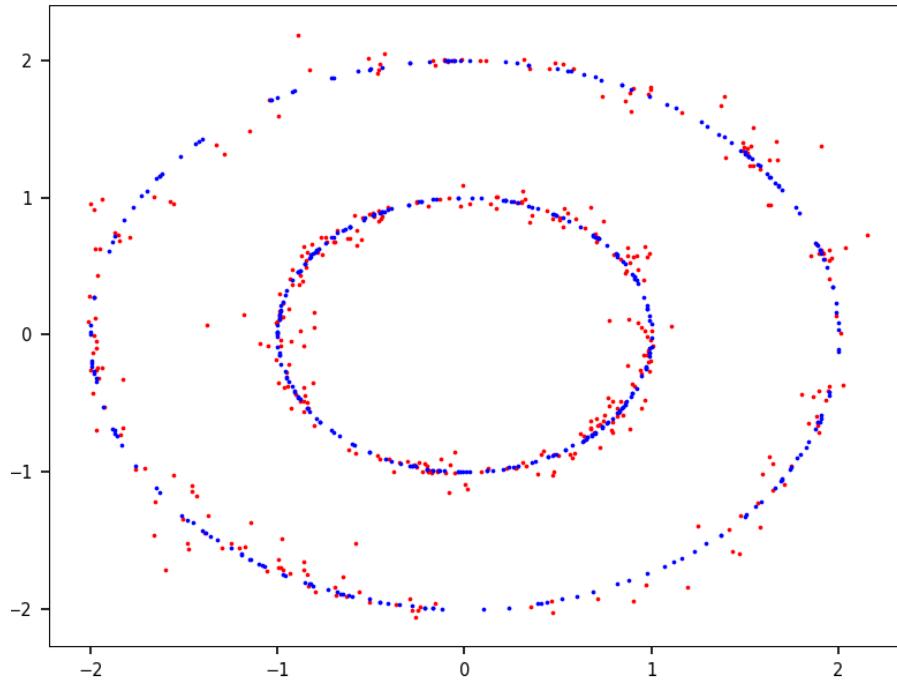


Figure 9.1: Synth. ($H > 0.6$, red) vs real (blue) with [NoGAN_Hellinger.py](#)

9.1 Project and solution

The name for the new algorithm is **constrained NoGAN**. It blends the best of **probabilistic NoGan** (based on deep **resampling**, see chapter 8) and Python code `DeepResampling_circle.py` [here](#)) with the efficient multivariate **bin structure** found in **standard NoGAN** (see chapter [?] and corresponding code `NoGan.py` [here](#)). The bin structure is used to compute the Hellinger distance. For the simplest version of constrained NoGAN, see `NoGAN_Hellinger.py`, [here](#). This script is great to synthetize data featuring hidden structures (in this case, concentric circles), to check whether or not pattern detection algorithms can detect them. The higher the Hellinger distance threshold used in the synthetization, the less visible the circles, making detection harder.

In the remaining of this project, I focus on `NoGAN_hellinger2.py`, with a slightly different architecture

leading to the spectacular results discussed earlier. The code is in section 9.2 and also on GitHub, [here](#). The resampling part works as follows:

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

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

You need to start with an initial synthetic dataset where each feature taken separately has about the same empirical distribution as its sister in the training set. This is accomplished with lines 93–97 in the code, using the option `mode='Quantiles'`. No matter the number of iterations, the final synthetic data set will be close but different to the training set: it is constrained on the initial synthetization. However, if you use the option `mode='Shuffle'`, the program uses a scrambled version of the training set for the initial synthetization: each feature column is shuffled separately. In this case, the only synthetization minimizing the Hellinger distance (up to a permutation of the rows) is the training set itself! You would think that it will take billions of years of computing time to reach it *exactly* given the probabilistic nature of the algorithm, albeit getting a very good approximation very quickly. However, thanks to the small size of the training set (400 observations), it takes just 2 millions atomic updates (also called swaps) to reach the unique, known global optimum. This is faster than any other algorithm including neural networks.

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

Table 9.1: KS stats: S = synth., S_0 = initial synth., and T = training set

Table 9.1 shows the quality of the synthetization S , depending on the dimension (the number of features), the number of iterations, and whether the initial synthetization S_0 is independent from the training set T (mode set to ‘Quantiles’) or a scrambled version of the training set (mode set to ‘Shuffle’). The parameter `dim` is specified in line 18 in the code. The values for `granularity` and `reset_granularity` depend on `dim` and are set in the code respectively in line 151 and 149. The multivariate [Kolmogorov-Smirnov distance](#) (KS) takes values between 0 and 1, with zero for perfect match. It is computed in lines 283–305 in the code, using the [GenAI-evaluation](#) Python library. The quality of the synthetization is assessed by comparing the base value $KS(S_0, T)$ with $KS(S, T)$. The lower the latter, the better the result.

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

The project consists of the following steps:

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

Depending on the number `dim` of features (the dimension), if you increase or decrease γ , Δ_γ or Δ_τ , what is the expected impact on the final results? Can you retrieve the values used to produce Table 9.1, by running the code? Also, does it make sense to plot the values of the Hellinger loss function over time? Why not? Would this plot still convey some useful information?

Step 2: Higher dimensions, bigger data. Create a training set consisting of 20 features and 10,000 observations, for instance a Gaussian mixture. Feel free to use OpenAI to generate the code (I did). Or use my code `mixture.py` or its output data `mixture.csv`, available on GitHub respectively [here](#) and [here](#). When running the synthesizer `NoGAN_Hellinger2.py` (code in section 9.2) on the mixture dataset, what challenges are you facing when the dimension is larger than 10, and how to address them? Note that when I asked OpenAI to produce `mixture.py`, the covariance matrices in the mixture are not symmetric and positive semidefinite. I had to use a second prompt to fix it.

Step 3: Fine-tuning and evaluation. It is easy to assess the impact of the hyperparameters, on the generated (synthetic) data. Design a scheme to efficiently test different combinations of values for $\gamma, \tau, \Delta_\gamma$ and Δ_τ . Then, find combinations that do well, based on the number of features (the dimension). What about the number of observations to synthesize? For example, when synthesizing a large number, it may be better to split the output data into multiple batches generated separately. Assess the impact of the batch size on the results, especially the speed of convergence. How do you evaluate the results?

Step 4: Comparing NoGAN methods. That is, `NoGAN_hellinger2.py` (the code in section 9.2) with `NoGAN_hellinger.py` (also using Hellinger as the loss function), `DeepResampling_circle.py` based on a traditional loss function (discussed in chapter 7 in [14]), and the original `NoGAN.py` described in section 7. The latter has no loss function. Finally, also compare with GAN-based `GAN_circle8d.py` described in section 5.2 in [13]. Focus on quality, speed, and overfitting in particular.

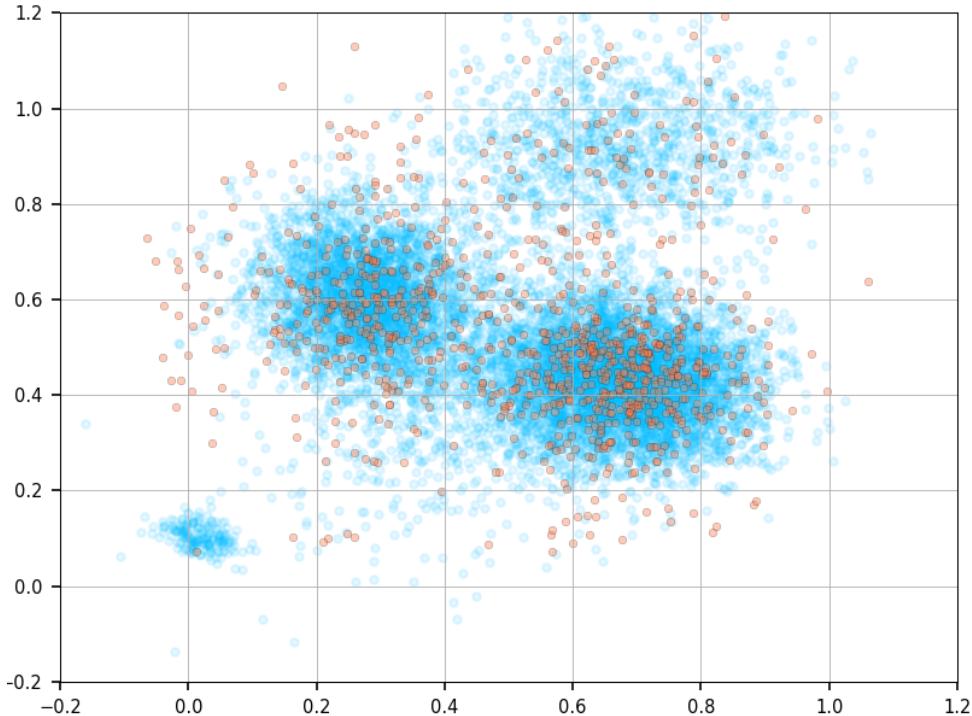


Figure 9.2: First 2 dimensions of mixture data (blue) with synthetization in orange

Now my answers. Regarding Step 1, I used $\gamma = 20, \Delta_\gamma = 10,000$ in two dimensions, and then $\gamma = 2, \Delta_\gamma = 50,000$ in three dimensions, together with `mode='Shuffle'` in line 89 to produce the results in Table 9.1. I did not use the shift parameters τ, Δ_τ on the circle dataset. Starting with a high value for γ speeds up convergence in low dimensions (2 or 3) but can get you stuck in higher dimensions. Use a lower value of γ_τ if you get stuck. Use $\Delta_\tau \leq 0.5 \times \gamma_\tau$ to visit the feature space more thoroughly: it can speed up convergence and get you unstuck in higher dimensions. A new τ is similar to trying another location or starting point in gradient descent to avoid local minima. In higher dimensions, use $\gamma = 2, \Delta_\gamma > 100,000$ and $\Delta_\tau < 50,000$. In general, the more swaps, the better. Since the loss function is different each time the value of γ or τ changes, its graph is meaningless. Still, it can help you identify when the moves (called *swaps*) are becoming rare or absent, to fix it: see Figure 9.3,

where the slope dramatically changes each time γ is modified, that is, each time the loss stops decreasing fast enough.

As for **Step 2**, in dimensions above 10, it is customary to work with groups of features, one group at a time. Do the first 10, then the next 10 conditionally on the first 10, then the next 10 conditionally on first 20, and so on. The computing time increases linearly with the dimension, keeping the same number of iterations, say 2 million, regardless of dimension. This is similar to neural networks, where computing time also increases linearly with the dimension, not exponentially. See details in chapter 7 in [14]. Finally, I used `mode='Quantiles'` on the mixture dataset that I created, available [here](#), working with the first 8 features only. In Figure 9.2, the training set has 10,000 observations. I generated 1000 synthetic ones, thus the reason why orange dots are much rarer than blue ones.

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

Table 9.2: Hyperparam impact on KS, mixture dataset

Now **Step 3**. Why evaluate the results when the loss function is actually the evaluation metric? Yet, it still makes sense to have another evaluation with a different metric, if anything to confirm that all computations are done correctly. In the code, I use the KS distance for final evaluation, see lines 283–305. Table 9.2 is based on first 8 columns of the mixture dataset, using `mode='Quantiles'`. Here S, S_0, T stand respectively for the final synthetization, initial synthetization, and the training set. The table features some of the best hyperparameter combinations that I tried. For a systematic approach to find these parameters, you can use `smart grid search`, described in [12]. As in table 9.1, I used 2 million iterations.

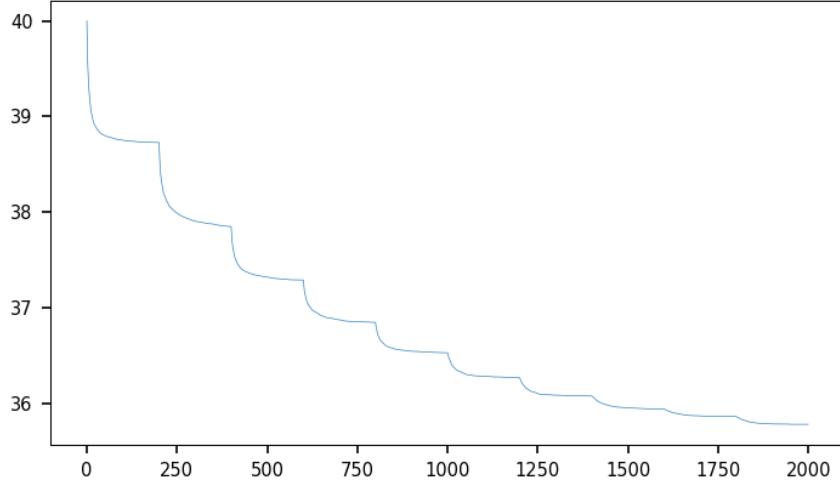


Figure 9.3: Adaptive loss function: steps showing when γ changed

Regarding **Step 4**, `NoGAN.py` is the best, followed by `NoGAN_Hellinger.py` (a probabilistic version of the former). This is because they generate observations directly into the bin structure attached to the training set. The drawback is potential overfitting, which can be mitigated with `constrained synthetization` as discussed earlier. Then `NoGAN_Hellinger2.py` usually beats `DeepResampling_circle.py` because the former uses the evaluation metric as the loss function, while the latter uses a proxy. Finally, `GAN_circle8d.py` is the last in my ranking: hard to train and fine-tune, very sensitive to the seed and to the dataset (volatile results), non-reproducible, and non-explainable AI.

Perhaps one of the greatest benefits of `NoGAN_Hellinger2.py` is its ability to test very fast new features that could be added to deep neural networks, such as using the evaluation metric as the loss function. Also, by choosing `mode='Shuffle'`, the ability to check whether the system can retrieve the known global optimum, and if not, how close it gets. Finally, an adaptive loss function may prevent you from getting stuck in a local optimum, by changing γ or Δ_τ when the loss stops decreasing: this is visible in Figure 9.3.

9.2 Python code

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

```

1 import numpy as np
2 import pandas as pd
3 import matplotlib.pyplot as plt
4 import matplotlib as mpl
5 from matplotlib import pyplot
6
7
8 #--- [1] Read data and select dim
9
10 # you can read data from URL below
11 # https://raw.githubusercontent.com/VincentGranville/Main/main/circle8d.csv
12 data = pd.read_csv('circle8d.csv')
13 features = list(data.columns.values)
14 X = data.to_numpy()
15 features = np.array(features)
16
17 # use the first dim columns only
18 dim = 2
19 X = X[:, 0:dim]
20 features = features[0:dim]
21 nobs_real, dim = X.shape
22
23
24 #--- [2] Functions to build bin structure
25
26 def create_quantile_table(x, Hyperparam, shift):
27
28     arr_q = []
29     for d in range(dim):
30         n = Hyperparam[d]
31         arr_qd = np.zeros(n+1)
32         for k in range(n):
33             q = shift[d] + k/n
34             arr_qd[k] = np.quantile(x[:,d], q % 1)
35         arr_qd[n] = max(x[:,d])
36         arr_qd.sort()
37         arr_q.append(arr_qd)
38     return(arr_q)
39
40
41 def find_quantile_index(x, arr_quantiles):
42     k = 0
43     while k < len(arr_quantiles) and x > arr_quantiles[k]:
44         k += 1
45     return(max(0, k-1))
46
47
48 def create_bin_structure(x, arr_q):
49
50     hash_bins = {}
51     hash_bins_median = {}
52     hash_index = []
53
54     for n in range(x.shape[0]):
55
56         key = ()
57         for d in range(dim):
58             kd = find_quantile_index(x[n,d], arr_q[d])
59             key = (*key, kd)
60         hash_index.append(key)
61
62         if key in hash_bins:
63             hash_bins[key] += 1
64             points = hash_bins_median[key]
65             points.append(x[n,:])
66             hash_bins_median[key] = points
67         else:
68             hash_bins[key] = 1
69             hash_bins_median[key] = [x[n,:]]
70
71     for key in hash_bins:
72         points = hash_bins_median[key]
```

```

73     # beware: even number of points -> median is not one of the points
74     median = np.median(points, axis = 0)
75     hash_bins_median[key] = median
76
77     return (hash_bins, hash_index, hash_bins_median)
78
79
80 #--- [3] Generate initial nobs_synth obs
81
82 # if nobs_synth > 1000, split in smaller batches, do one batch at a time
83 nobs_synth = nobs_real
84 seed = 155
85 np.random.seed(seed)
86
87 # get initial synth. data with same marginal distributions as real data
88
89 mode = 'Shuffle' # options: 'Shuffle', 'Quantiles'
90 synth_X = np.empty(shape=(nobs_synth,dim))
91
92 if mode == 'Quantiles':
93     for k in range(nobs_synth):
94         pc = np.random.uniform(0, 1.00000001, dim)
95         for d in range(dim):
96             synth_X[k, d] = np.quantile(X[:,d], pc[d], axis=0)
97
98 elif mode == 'Shuffle':
99     nobs_synth == nobs_real # both must be equal
100    synth_X = np.copy(X)
101    for d in range(dim):
102        col = synth_X[:, d]
103        np.random.shuffle(col)
104        synth_X[:, d] = col
105
106 synth_X_init = np.copy(synth_X)
107
108
109 #--- [4] Main part: create synth obs to minimize Hellinger loss function
110
111 def in_bin(x, key, arr_q):
112     # test if vector x is in bin attached to key
113     status = True
114     for d in range(dim):
115         arr_qd = arr_q[d]
116         kd = key[d]
117         if x[d] < arr_qd[kd] or x[d] >= arr_qd[kd+1]:
118             status = False # x is not in the bin
119     return(status)
120
121 def array_to_tuple(arr):
122     list = ()
123     for k in range(len(arr)):
124         list = (*list, arr[k])
125     return(list)
126
127 Hellinger = 40.0 # arbitrary value
128 swaps = 0
129 history_log_H = []
130 history_log_swaps = []
131 flist = [] # list of image filenames for the video
132 frame = 0 # frame number, for video
133
134 # to accelerate computations (pre-computed sqrt)
135 sqrt_real = np.sqrt(nobs_real)
136 sqrt_synth = np.sqrt(nobs_synth)
137 n_sqrt = max(nobs_real, nobs_synth)
138 arr_sqrt = np.sqrt(np.arange(n_sqrt))
139
140 # visualization: graphic parameters
141 mpl.rcParams['lines.linewidth'] = 0.3
142 mpl.rcParams['axes.linewidth'] = 0.5
143 plt.rcParams['xtick.labelsize'] = 7
144 plt.rcParams['ytick.labelsize'] = 7
145
146 video_mode = False
147
148 # Hyperparameters

```

```

149 reset_granularity = 10000 # set to 50000 if dim = 3, set to 10000 if dim = 2
150 reset_shift = 999999999999
151 granularity = 20 # set to 2 if dim > 2, set to 20 if dim = 2
152 Hyperparam = np.full(dim, granularity)
153 shift = np.zeros(dim)
154 n_iter = 2000000
155
156
157 for iter in range(n_iter):
158
159     if iter % reset_granularity == 0 or iter % reset_shift == 0 or iter == 0:
160
161         # Get more granular Hellinger approximation
162
163         if iter % reset_granularity == 0:
164             Hyperparam = 1 + Hyperparam
165         if iter % reset_shift == 0:
166             shift = np.random.uniform(0, 1, dim)
167         arr_q = create_quantile_table(X, Hyperparam, shift)
168         (hash_bins_real,
169          hash_index_real,
170          hash_bins_median_real
171         ) = create_bin_structure(X, arr_q)
172         (hash_bins_synth,
173          hash_index_synth,
174          hash_bins_median_synth, # unused
175         ) = create_bin_structure(synth_X, arr_q)
176
177         k = np.random.randint(0, nobs_synth)
178         key_k = hash_index_synth[k]
179         scount1 = hash_bins_synth[key_k]
180         if key_k in hash_bins_real:
181             rcount1 = hash_bins_real[key_k]
182         else:
183             rcount1 = 0
184
185         l = np.random.randint(0, nobs_synth)
186         key_l = hash_index_synth[l]
187         scount2 = hash_bins_synth[key_l]
188         if key_l in hash_bins_real:
189             rcount2 = hash_bins_real[key_l]
190         else:
191             rcount2 = 0
192
193         d = np.random.randint(1, dim) # column 0 can stay fixed
194
195         new_key_k = np.copy(key_k)
196         new_key_l = np.copy(key_l)
197         new_key_k[d] = key_l[d]
198         new_key_l[d] = key_k[d]
199         new_key_k = array_to_tuple(new_key_k)
200         new_key_l = array_to_tuple(new_key_l)
201
202         if new_key_k in hash_bins_synth:
203             scount3 = hash_bins_synth[new_key_k]
204         else:
205             scount3 = 0
206         if new_key_k in hash_bins_real:
207             rcount3 = hash_bins_real[new_key_k]
208         else:
209             rcount3 = 0
210
211         if new_key_l in hash_bins_synth:
212             scount4 = hash_bins_synth[new_key_l]
213         else:
214             scount4 = 0
215         if new_key_l in hash_bins_real:
216             rcount4 = hash_bins_real[new_key_l]
217         else:
218             rcount4 = 0
219
220         A = arr_sqrt[scount1] / sqrt_synth - arr_sqrt[rcount1] / sqrt_real
221         B = arr_sqrt[scount1-1] / sqrt_synth - arr_sqrt[rcount1] / sqrt_real
222         C = arr_sqrt[scount2] / sqrt_synth - arr_sqrt[rcount2] / sqrt_real
223         D = arr_sqrt[scount2-1] / sqrt_synth - arr_sqrt[rcount2] / sqrt_real
224         E = arr_sqrt[scount3] / sqrt_synth - arr_sqrt[rcount3] / sqrt_real

```

```

225 F = arr_sqrt[scount3+1]/sqrt_synth - arr_sqrt[rcount3]/sqrt_real
226 G = arr_sqrt[scount4] /sqrt_synth - arr_sqrt[rcount4]/sqrt_real
227 H = arr_sqrt[scount4+1]/sqrt_synth - arr_sqrt[rcount4]/sqrt_real
228 delta_H = - A*A + B*B - C*C + D*D - E*E + F*F - G*G + H*H
229
230 if delta_H < -0.00001:
231
232     Hellinger += delta_H
233     swaps += 1
234
235 # update hash_index_synth and hash_bins_synth
236
237 hash_index_synth[k] = new_key_k
238 if new_key_k in hash_bins_synth:
239     hash_bins_synth[new_key_k] +=1
240 else:
241     hash_bins_synth[new_key_k] = 1
242 if hash_bins_synth[key_k] == 1:
243     del hash_bins_synth[key_k]
244 else:
245     hash_bins_synth[key_k] -= 1
246
247 hash_index_synth[l] = new_key_l
248 if new_key_l in hash_bins_synth:
249     hash_bins_synth[new_key_l] += 1
250 else:
251     hash_bins_synth[new_key_l] =1
252 if key_l in hash_bins_synth:
253     hash_bins_synth[key_l] -= 1
254 else:
255     hash_bins_synth[key_l] = 1
256
257 # update synthetic data
258
259 aux = synth_X[k, d]
260 synth_X[k, d] = synth_X[l, d]
261 synth_X[l, d] = aux
262
263 if video_mode and swaps % 25 == 0:
264
265     # save image for future inclusion in video
266     fname='nogan3_frame'+str(frame)+'.png'
267     flist.append(fname)
268     plt.scatter(synth_X[:,0], synth_X[:,1], s = 1.0)
269     plt.savefig(fname, dpi = 200)
270     plt.close()
271     frame += 1
272
273 if iter % 1000 == 0:
274
275     print("Iter: %d | Loss: %9.6f | Swaps: %d"
276           %(iter, Hellinger, swaps))
277     history_log_H.append(Hellinger)
278     history_log_swaps.append(swaps)
279
280 #--- [5] Evaluation with KS distance
281
282 import genai_evaluation as ge
283
284 n_nodes = 1000
285
286 df_init = pd.DataFrame(synth_X_init, columns = features)
287 df_synth = pd.DataFrame(synth_X, columns = features)
288 df_train = pd.DataFrame(X, columns = features)
289
290 query_lst, ecdf_train, ecdf_init = ge.multivariate_ecdf(df_train,
291               df_init, n_nodes, verbose = True)
292 ks_base = ge.ks_statistic(ecdf_train, ecdf_init)
293
294 query_lst, ecdf_train, ecdf_synth = ge.multivariate_ecdf(df_train,
295               df_synth, n_nodes, verbose = True)
296 ks = ge.ks_statistic(ecdf_train, ecdf_synth)
297
298 query_lst, ecdf_init, ecdf_synth = ge.multivariate_ecdf(df_init,
299               df_synth, n_nodes, verbose = True)

```

```

301 ks_diff = ge.ks_statistic(ecdf_init, ecdf_synth)
302
303 print("Test ECDF Kolmogorof-Smirnov dist. (synth. vs train.): %6.4f" %(ks))
304 print("Base ECDF Kolmogorof-Smirnov dist. (init. vs train.): %6.4f" %(ks_base))
305 print("Diff ECDF Kolmogorof-Smirnov dist. (init. vs synth.): %6.4f" %(ks_diff))
306
307
308 #--- [6] Plot some results and create video
309
310 mpl.rc('hatch', color='k', linewidth=0.3)
311 plt.scatter(X[:,0],X[:,1],marker='o',c='deepskyblue',alpha=0.1,s=10)
312 plt.scatter(synth_X[:,0],synth_X[:,1],marker='o',c='coral',alpha=0.4,s=10,
            edgecolors='black',lw=0.2)
313 plt.grid(linewidth = 0.4, alpha = 1)
314 plt.show()
315
316 x_axis = range(len(history_log_H))
317 plt.plot(x_axis, history_log_H)
318 plt.show()
319 plt.plot(x_axis, history_log_swaps)
320 plt.show()
321
322 if video_mode:
323     import moviepy.video.io.ImageSequenceClip
324     clip = moviepy.video.io.ImageSequenceClip.ImageSequenceClip(flist, fps=6)
325     clip.write_videofile('nogan3.mp4')

```

Chapter 10

Deep Retrieval and Multi-Index Chunking for PDFs

In this chapter, I discuss the preprocessing steps used to turn a PDF repository into input suitable for xLLM. It includes chunking, indexing text entities with hierarchical multi-index system, retrieving contextual elements including font type, color, and size, building additional contextual information such as agents to add to text entities, as well as retrieving images and tables – some not detected by standard Python libraries, relying instead on proprietary technology.

Section 10.2 includes a discussion on why training an LLM for token prediction is not needed, how you can do without to save considerable training time and minimize costs (no GPU, zero weight), as well as current challenges with model evaluation. Section 10.3 summarizes key features of xLLM, contrasting them with other systems, and justifying the label LLM 2.0 for our innovative, radically different architecture with its numerous next-gen subsystems.

10.1 PDF contextual parsing and chunking: Nvidia case study

Here, I illustrate how you can perform [chunking](#), [multi-indexing](#), table and image retrieval on a PDF repository featuring public financial reports from Nvidia. The goal is to turn the PDFs into a format amenable to xLLM, with contextual chunks similar – at least from a design standpoint – to the one featured in Table 2.1. Although the Python code handles multiple PDFs at once, I focus on a single PDF document, posted [here](#).

I use the PyMuPDF Python library also known as Fitz, with home-made algorithms to retrieve tables that it fails to detect. Another useful library is LlamaParse. Yet, a different approach consists of saving each PDF page (a slide in this case) as an image, then using computer vision technology to retrieve the various elements from the images, and turn them into text when appropriate. Indeed, lines 286–300 in the code in section 10.1.3 turn the slides into images. However, I will not follow this approach. Instead, I convert the PDFs to [JSON](#) (line 158 in the code) and then parse the JSON elements, including images. Note that it includes non-standard characters: see how to process them in lines 85–91 in the code.

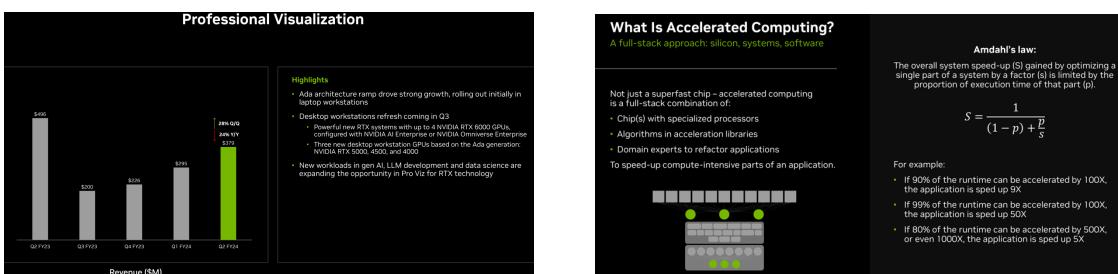


Figure 10.1: Nvidia PDF: slide with sublist, hidden table (left); with 2 lists and formula (right)

Two slides (thumbnails) are featured in Figure 10.1. For full resolution, click on the corresponding thumbnail, or zoom in. The resulting text entities are shown respectively in Tables 10.1 and 10.2, with entries (rows) listed in the order they were retrieved: sometimes, the slide title is not the first element to be recovered, and sometimes the title is absent. The remaining of the discussion focuses on how to build these text entities, also adding [multimodal](#) elements such as tables and images, and how to integrate them with xLLM.

Table 10.1: Turning the left slide in Figure 10.1 into a multi-index xLLM text entity

Type	ID1	ID2	ID3	Size	Text
Note	0	-1	-1	36	Highlights
List	0	0	-1	36	Growth fueled by GeForce RTX 40 series GPUs for laptops and desktops
List	0	1	-1	36	Large upgrade opportunity ahead: just 47% of installed base have upgraded to RTX; ~20% have a GPU with RTX 3060 or higher
List	0	2	-1	36	NVIDIA GPU-powered laptops have gained in popularity; shipments now outpace desktop GPUs in several regions around the world
SubList	0	2	0	32	Likely to shift overall Gaming seasonality a bit, with Q2 and Q3 stronger, reflecting the back-to-school and holiday build schedules
List	0	3	0	36	RTX/DLSS ecosystem continues to expand; 35 new games added DLSS support, including Diablo IV and Baldur's Gate 3; there are now over 330 RTX accelerated games and apps
List	0	4	0	36	Bringing generative AI to gaming with NVIDIA Avatar Cloud Engine (ACE) for Games
Title	1	4	0	72	Gaming
Note	3	-1	-1	40	Revenue (\$M)
Data	4	TD0	-1	24	\$2,042 \$1,574 \$1,831 \$2,240 \$2,486
Note	5	TL0	-1	24	Q2 FY23 Q3 FY23 Q4 FY23 Q1 FY24 Q2 FY24
Data	6	-1	-1	24	22% Y/Y
Note	7	-1	-1	24	and
Data	8	-1	-1	24	11% Q/Q

Table 10.2: Turning the right slide in Figure 10.1 into a multi-index xLLM text entity

Type	ID1	ID2	ID3	Size	Text
Title	0	-1	-1	72	What Is Accelerated Computing?
Note	1	-1	-1	44	Not just a superfast chip – accelerated computing is a full-stack combination of:
List	1	0	-1	44	Chip(s) with specialized processors
List	1	1	-1	44	Algorithms in acceleration libraries
List	1	2	-1	44	Domain experts to refactor applications To speed-up compute-intensive parts of an application.
Note	3	-1	-1	48	A full-stack approach: silicon, systems, software
Note	4	-1	-1	44	For example:
List	4	0	-1	44	If 90% of the runtime can be accelerated by 100X, the application is sped up 9X
List	4	1	-1	44	If 99% of the runtime can be accelerated by 100X, the application is sped up 50X
List	4	2	-1	44	If 80% of the runtime can be accelerated by 500X, or even 1000X, the application is sped up 5X
Note	6	-1	-1	48	Amdahl's law:
Note	7	-1	-1	44	The overall system speed-up (S) gained by optimizing a single part of a system by a factor (s) is limited by the proportion of execution time of that part (p).
Note	8	-1	-1	60	\\U0001d446=
Data	9	TD0	-1	60	1 1\u2212 \\U0001d45d+
Note	10	TL0	-1	60	\\U0001d45d \\U0001d460

10.1.1 Multi-index, bullet lists, images, tables

A **multi-index** is a vector index, in contrast to univariate indexes discussed in chapter 11 and found in standard LLMs. In our example, the columns ID1, ID2, ID3 and Size in Tables 10.1 and 10.2 are part of the multi-index. In the Python code, they corresponds respectively to `block_ID`, `item_ID`, `sub_ID`, and `font_size`: see code line 240. Other multi-index components generated by the Python script in the same code line: `doc_ID` (reference to the parent PDF document), `page_num` (the page number associated to the text entity in the PDF document in question), `font_type`, `font_color` and `block_number`.

10.1.1.1 Multi-index components

I now describe the main components ID1, ID2, ID3 and so forth built into the multi-index.

- ID1 (`block_ID`) is an home-made alternate block identifier to the JSON `block_number` generated by the PyMuPDF library. It puts together broken sentences having multiple block numbers in JSON. Typically, the text sections separated by “|” in the Text column in Tables 10.1 and 10.2 are assigned different block numbers, yet are considered to be part of a same `block_ID`.
- ID2 (`item_ID`) and ID3 (`sub_ID`): see section 10.1.1.5
- Size (`font_size`) identifies the size of the font, an indicator of importance. Also, color and font type (for instance, italics or bold) convey special information. The Python code retrieves these elements.
- In addition, `doc_ID` is used to identify the parent document in a repository with multiple PDFs, while `page_num` identifies the page number in a same PDF, whether pages are numbered or not.

10.1.1.2 Images

Images also have their own multi-index, consisting of `doc_ID`, `page_num`, and the image number in the text entity, represented by `image_index` (line 261 in the code). To save these images to output files, uncomment lines 263–264 in the code. These images are internal to the PDFs and should not be confused with images created for each slide in lines 288–300. Typically, they represent logos, part of a diagram, or full-fledged valuable images. By looking at the size and type distribution (lines 259–260), it is easy to detect and discard useless images that appear on many slides. Indexed images (type + size) are listed in the output file, at the bottom of each text entity: see [here](#).

10.1.1.3 Tables

The tables detected by the PyMuPDF library in lines 160–170 are not indexed yet. However, those internally detected have a special ID2 in the multi-index, consisting of the letter T (for table), followed by the letter D for data or L for labels, and then followed by the table number (`table_ID`, line 52) in the text entity in question. See example in Table 10.1.

10.1.1.4 Text entities, sub-entities

I use the words **chunk** and **text entity** interchangeably, with the same meaning throughout this book. Note that in many standard LLMs, chunks are not determined by the document structure (they are fixed-size) and do not include contextual information. This is not the case with xLLM. Finally, a sub-entity is one element in a text entity: in our example, one row in Tables 10.1 or 10.2. More on text entities in section 10.1.2.

10.1.1.5 Bullet lists

Bullet lists are not recognized by the PyMuPDF library, yet are everywhere. A bullet list is identified by three IDs, namely ID1, ID2, and ID3. All items within a same list (including sub-lists) have the same ID1, representing the list number within a same text entity, itself identified by `page_num`. For instance, we have two distinct bullet lists in Table 10.2, and one list with a sub-list in Table 10.1. Sub-list items have an ID3 distinct from -1, and the same ID2 identifying the parent list item. Different list items at the top level have different ID2's but the same ID1.

10.1.2 Diagrams, context, agents, auto-tagging

Each text entity is assigned a type: data, labels, title, note, list, or sub-list. In particular, “title” is a contextual sub-entity, as opposed to a regular one. In xLLM, **multi-tokens** found in contextual elements are called **graph multitokens** as they are related to the underlying **knowledge graph**, and are treated separately: see section 11.2. In the Nvidia corpus, another type of contextual information are keywords in green font. These are index

keywords and play the same role as the words highlighted in orange in this book (all of them are index terms). Additional contextual sub-entities will be added later, including **agents**, tags, and categories, using an **auto-tagging** algorithm without human interaction. The first step consists of creating a multi-token dictionary and clustering techniques along with analyzing the token distribution, to identify candidate tags, categories, and agents. The process is similar to the one already in place for the corporate corpus discussed in section 2.3.

As for diagrams, they are detected as images and available in the individual slides stored as images. In many cases (histograms), the underlying table is retrieved using the mechanism described in section 10.1.1.3. It allows for automated predictive analytics on multiple tables blended together. Some diagrams can be rendered as HTML or JSON, as the PyMuPDF library allows you to convert PDF to JSON or HTML.

Finally, the PDFs contain many special characters, including those found in mathematical formulas. For instance, the bullet symbol indicating a bullet list, or the bottom three rows in Table 10.2. In this case, the special characters are converted to a code number, starting with \u or \U. See lines 85–91 in the Python code. Interestingly, line 86 in the Python code contains the symbol • without causing problems when executing it.

10.1.3 Python code

The Python code, input PDF and output (indexed contextual text entities in text format) are on GitHub, respectively [here](#) (code), [here](#) (input), and [here](#) (output).

```

1 # Input PDF for this script: https://drive.google.com/file/d/1Daa9oZJm4-b6NqUsVGxK2euemcFnf8jH/
2
3 # https://pymupdf.readthedocs.io/en/latest/page.html#Page.find_tables
4 #
5     https://stackoverflow.com/questions/56155676/how-do-i-extract-a-table-from-a-pdf-file-using-pymupdf
6
7 import fitz # PyMuPDF
8
9
9 def update_item_ID(k, entity_idx, type, table_ID):
10
11     idx = entity_idx[k]
12     if type == 'Data':
13         # table: data row
14         flag = 'TD' #
15     elif type == 'Note':
16         # table: labels
17         flag = 'TL'
18     idx_list = list(idx)
19     idx_list[1] = flag + str(table_ID)
20     entity_idx[k] = tuple(idx_list)
21
22     return(entity_idx)
23
24
25 def detect_table(xLLM_entity):
26
27     # detect and flag simple pseudo-tables
28
29     entity_txt = xLLM_entity[0]
30     entity_type = xLLM_entity[1]
31     entity_idx = xLLM_entity[2]
32     table_ID = -1
33     table_flag = False
34
35     for k in range(1, len(entity_type)):
36
37         type = entity_type[k]
38         text = entity_txt[k]
39         old_text = entity_txt[k-1]
40         old_type = entity_type[k-1]
41
42         if ( (
43             (type == 'Data' and old_type == 'Note') or
44             (type == 'Note' and old_type == 'Data') or
45             (type == 'Data' and old_type == 'Data')
46         )
47             and old_text.count('|') == text.count('|')
48             and text.count('|') > 2
49         ):
50             print("detected table", table_ID + 1)
51             if not table_flag:

```

```

52         table_ID += 1
53         table_flag = True
54         idx = entity_idx[k]
55         old_idx = entity_idx[k-1]
56
57         # update item_ID (idx[1] and old_idx[1]) in current and previous row
58         # item_ID starts with letter D if data, or N if labels
59
60         update_item_ID(k, entity_idx, type, table_ID)
61         update_item_ID(k-1, entity_idx, old_type, table_ID)
62
63     else:
64         #table_ID = -1
65         table_flag = False
66
67     return(xLLM_entity)
68
69
70 def cprint_page(xLLM_entity, OUT):
71
72     entity_txt = xLLM_entity[0]
73     entity_type = xLLM_entity[1]
74     entity_idx = xLLM_entity[2]
75
76     for k in range(len(entity_type)):
77
78         type = entity_type[k]
79         text = entity_txt[k]
80         text = text.strip()
81         text = text.replace(" ", " ")
82         text = text.replace(" |", " |")
83         text = text.replace("||", " |")
84         text = text.replace("|||", " |")
85         text = text.encode('unicode-escape').decode('ascii')
86         text = text.replace('\u2022', '•')
87         text = text.replace('\u2013', '--')
88         text = text.replace('\u2014', '--')
89         text = text.replace('\u2019', "'")
90         text = text.replace('\u201c', '"')
91         text = text.replace('\u201d', '"')
92
93         idx = entity_idx[k]
94         doc_ID = idx[3]
95         block_ID = idx[0]
96         item_ID = idx[1]
97         sub_ID = idx[2]
98         pn = idx[4] # page number
99         fs = idx[5] # font size
100        fc = idx[6] # font color
101        ft = idx[7] # font typeface
102        #- print(k, type, idx, text)
103        OUT.write(f"\{type:<8}\{block_ID:>3}\{item_ID:>5}\{sub_ID:>3}\{pn:>3}\"
104                      f"\{fs:>5}\{fc:>9}\ {ft:<20}\{text:<80}\n")
105
106    OUT.write("\n")
107    return()
108
109
110 def update_page(text, type, entity, idx):
111
112     entity_txt = entity[0]
113     entity_type = entity[1]
114     entity_idx = entity[2]
115     block_ID = idx[0]
116     item_ID = idx[1]
117     sub_ID = idx[2]
118     k = len(entity_txt)
119     if k > 0:
120         old_type = entity_type[k-1]
121         old_idx = entity_idx[k-1]
122         old_block_ID = old_idx[0]
123         old_item_ID = old_idx[1]
124         old_sub_ID = old_idx[2]
125     else:
126         old_type = ""
127         old_block_ID = ""

```

```

128     old_item_ID = ""
129     old_sub_ID = ""
130     if type in ('Note', 'Data'):
131         sep = "|"
132     else:
133         sep = " "
134
135     if (type == old_type and block_ID == old_block_ID
136         and item_ID == old_item_ID and sub_ID == old_sub_ID):
137         new_text = entity_txt[k-1] + text + sep
138         entity_txt[k-1] = new_text
139     else:
140         entity_txt.append(sep + text + sep)
141         entity_type.append(type)
142         entity_idx.append(idx)
143
144     return(entity)
145
146 def convert_pdf_to_json(pdf_path, json_path, doc_ID, text_path):
147     # Open the PDF file
148     pdf_document = fitz.open(pdf_path)
149     content = ""
150
151     # Iterate through the pages
152     for page_num in range(len(pdf_document)):
153         OUT.write("\n-----\n")
154         OUT.write("Processing page " + str(page_num) + "\n\n")
155         print("Page:", page_num)
156         page = pdf_document.load_page(page_num)
157
158         text_data = page.get_text("dict") # also extract as "json" to get tokens in green font
159
160         tabs = page.find_tables()
161         for tabs_index, tab in enumerate(tabs):
162             # iterate over all tables
163             index = (page_num, tabs_index)
164             table_data = tab.extract() # extracting tabs[i], the i-th table in this page
165             if len(table_data) > 0:
166                 # if not, ignore this table (note the important parameter threshold here)
167                 OUT.write("Table " + str(index) + ":\n")
168                 for row in table_data:
169                     OUT.write(str(row) + "\n")
170                 OUT.write("\n")
171
172         itemize = False
173         item_ID = -1
174         sub_ID = -1
175         block_ID = -1
176         item = ""
177         fsm = -1 # top level font size in bullet list
178         fst = 64 # min title font size (top parameter)
179         title = ""
180         notes = ""
181         old_block_number = -1
182         old_font_size = -1
183         entity_txt = []
184         entity_idx = []
185         entity_type = []
186         type = ""
187         old_text = ""
188         old_type = ""
189
190         for block in text_data["blocks"]:
191             if block["type"] == 0: # Text block
192                 block_number = block["number"]
193                 for line in block["lines"]:
194                     for span in line["spans"]:
195
196                         text = span["text"]
197                         font_name = span["font"]
198                         font_size = span["size"]
199                         font_size = round(font_size,1)
200                         font_color = span["color"]
201
202                         if font_size > fst:
203                             type = 'Title'

```

```

204         elif ord(text[0]) == 8226:
205             itemize = True
206             if fsm == -1:
207                 fsm = font_size
208                 if font_size > 0.98 * fsm:
209                     item_ID += 1
210                     type = 'List'
211                 else:
212                     sub_ID +=1
213                     type = 'SubList'
214             elif itemize:
215                 #- itemize = ((0.99 < font_size/old_font_size < 1.01) and
216                 #-(not text[0].isupper() or ord(old_text[0]) == 8226))
217                 itemize = ((0.99 < font_size/old_font_size < 1.01) or
218                             (ord(old_text[0]) == 8226))
219
220             if not itemize:
221                 item_ID = -1
222                 sub_ID = -1
223                 block_ID += 1
224                 type = 'Note'
225             else:
226                 if not text[0].isdigit() and text[0] not in ('$', '+', '-'):
227                     type = 'Note'
228                 #- elif block_number != old_block_number:
229                 else:
230                     type = 'Data'
231
232             if block_ID == -1:
233                 block_ID += 1
234             elif ((type not in (old_type, 'List', 'SubList')) or
235                   (not (0.99 < font_size/old_font_size < 1.01))):
236                 if (old_text != "" and ord(old_text[0]) != 8226 and
237                     type not in ('List', 'SubList')):
238                     block_ID += 1
239
240             idx = (block_ID, item_ID, sub_ID, doc_ID, page_num, font_size,
241                   font_color, font_name, block_number)
242             entity = (entity_txt, entity_type, entity_idx)
243             update_page(text, type, entity, idx)
244
245             old_font_size = font_size
246             old_text = text
247             old_type = type
248
249             old_block_number = block_number
250
251             entity = detect_table(entity)
252             cprint_page(entity, OUT)
253
254             image_list = page.get_images()
255             for image_index, img in enumerate(image_list, start=1):
256                 xref = img[0]
257                 base_image = pdf_document.extract_image(xref)
258                 image_bytes = base_image["image"]
259                 size = len(image_bytes)
260                 ext = base_image["ext"]
261                 index = (page_num, image_index)
262                 OUT.write(f"Image {str(index):<8}{str(ext):>5} size = {str(size):>6}\n")
263                 #- with open(f"image_{page_num + 1}_{image_index}.{ext}", "wb") as image_file:
264                 # image_file.write(image_bytes)
265
266             content += "__P" + str(page_num) + "\n" + page.get_text("json") # "text", "html", "json"
267
268             # Write the json content to a file
269             with open(json_path, 'w', encoding='utf-8') as json_file:
270                 json_file.write(content)
271
272             # --- Main ---
273
274             doc_ID = 0 # to identify the PDF doc
275             filename = 'nvda-f2q24-investor-presentation-final-1'
276             pdf_path = filename + '.pdf'
277             json_path = filename + '.json'
278             text_path = filename + '.txt'
279

```

```

280
281 OUT = open(text_path, "wt", encoding="utf-8")
282 convert_pdf_to_json(pdf_path, json_path, doc_ID, OUT)
283 OUT.close()
284
285
286 # --- PDF to Images ---
287
288 from PIL import Image
289
290 pdf_document = fitz.open(pdf_path)
291 zoom = 2 # to increase the resolution
292 mat = fitz.Matrix(zoom, zoom)
293
294 for page_num in range(len(pdf_document)):
295     page = pdf_document.load_page(page_num)
296     pix = page.get_pixmap(matrix = mat) # or (dpi = 300)
297     img = Image.frombytes("RGB", [pix.width, pix.height], pix.samples)
298     filename = "PDF" + str(page_num) + '.png' # you could change image format accordingly
299     img.save(filename)
300     print('Converting PDFs to Image ... ' + filename)

```

10.2 Challenges with standard LLM implementations

LLMs are trained on tasks irrelevant to what they will do for the user. It's like training a plane to efficiently operate on the runway, but not to fly. In short, it is almost impossible to train an LLM, and evaluating is just as challenging. Then, training is not even necessary. In this article, I dive on all these topics.

10.2.1 Training LLMs for the wrong tasks

Since the beginnings with Bert, training an LLM typically consists of predicting the next tokens in a sentence, or removing some tokens and then have your algorithm fill the blanks. You optimize the underlying deep neural networks to perform these **supervised learning** tasks as well as possible. Typically, it involves growing the list of tokens in the training set to billions or trillions, increasing the cost and time to train. However, recently, there is a tendency to work with smaller datasets, by distilling the input sources and token lists. After all, out of one trillion tokens, 99% are noise and do not contribute to improving the results for the end-user; they may even contribute to hallucinations. Keep in mind that human beings have a vocabulary of about 30,000 keywords, and that the number of potential standardized prompts on a specialized corpus (and thus the number of potential answers) is less than a million.

In addition, training relies on minimizing a loss function that is just a proxy to the model evaluation function. So, you don't even truly optimize next token prediction, itself a task unrelated to what LLMs must perform for the user. To directly optimize the evaluation metric, see my approach in chapter 9. And while I don't design LLMs for next token prediction, see one exception in section 7.1 in [13], to synthesize DNA sequences.

The fact is that LLM optimization is an **unsupervised machine learning** problem, thus not really amenable to training. I compare it to clustering, in contrast to supervised classification. There is no perfect answer except for trivial situations. In the context of LLMs, laymen may like OpenAI better than my own technology, while the converse is true for busy business professionals and advanced users. Also, new keywords keep coming regularly. Acronyms and synonyms that map to keywords in the training set yet absent in the corpus, are usually ignored. All this further complicates training.

10.2.2 Analogy to clustering algorithms

As an example, this time in the context of clustering, see Figure 10.2. It features a dataset with three clusters. How many do you see, if all the dots had the same color? Ask someone else, and you may get a different answer. What characterizes each of these clusters? You cannot train an algorithm to correctly identify all cluster structures, though you could for very specific cases like this one. The same is true for LLMs. Interestingly, some vendors claim that their LLM can solve clustering problems. I would be happy to see what they say about the dataset in Figure 10.2.

For those interested, the dots in Figure 1 represent points on three different orbits of the Riemann zeta function. The blue and red ones are related to the critical band. The red dots correspond to the critical line with infinitely many roots. It has no hole. The yellow dots are on an orbit outside the critical band; the orbit in question is bounded (unlike the other ones) and has a hole in the middle of the picture. The blue orbit also

appears to not cover the origin where red dots are concentrated; its hole center — the “eye” as in the eye of a hurricane — is on the left side, to the right of the dense red cluster but left to the center of the hole in the yellow cloud. Proving that its hole encompasses the origin, amounts to proving the Riemann Hypothesis. The blue and red orbits are unbounded.

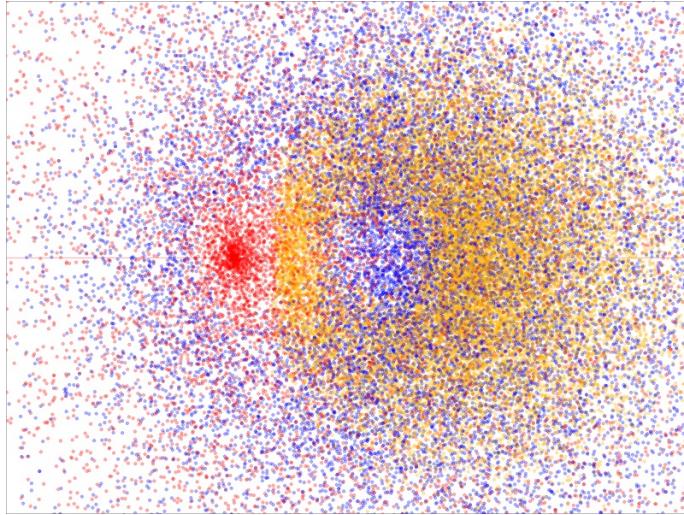


Figure 10.2: Training an LLM is as hard as training a clustering algorithm

Can LLMs figure this out? Can you train them to answer such questions? The answer is no. They could succeed if appropriately trained on this very type of example but fail on a dataset with a different structure (the equivalent of a different kind of prompt). One reason why it makes sense to build specialized LLMs rather than generic ones.

10.2.3 Challenges in evaluating LLMs

There are numerous evaluation metrics to assess the quality of an LLM, each measuring a specific type of performance. It is reminiscent of test batteries for random number generators (PRNGs) to assess how random the generated numbers are. But there is a major difference. In the case of PRNGs, the target output has a known, prespecified uniform distribution. The reason to use so many tests is because no one has yet implemented a **universal test**, based for instance on the full multivariate Kolmogorov-Smirnov distance. Actually, I did recently and will publish an article about it. It even has its Python library, [here](#).

For LLMs that answer arbitrary prompts and output English sentences, there is no possible universal test. Even though LLMs are trained to correctly replicate the full multivariate token distribution — a problem similar to PRNGs thus with a universal evaluation metric — the goal is not token prediction, and thus evaluation criteria must be different. There will never be a universal evaluation metric except for very peculiar cases, such as LLMs for predictive analytics. Below is a list of important features lacking in current evaluation and benchmarking metrics.

10.2.3.1 Overlooked criteria, hard to measure

Some of the important features rarely present in LLM benchmarks include:

- **Exhaustivity**: Ability to retrieve absolutely every relevant item, given the limited input corpus. Full exhaustivity is the ability to retrieve everything on a specific topic, like chemistry, even if not in the corpus. This may involve crawling in real-time to answer the prompt.
- **Inference**: Ability to invent correct answers for problems with no known solution, for instance proving or creating a theorem. I asked GPT if the sequence (a^n) is equidistributed modulo 1 if $a = 3/2$. Instead of providing references as I had hoped — it usually does not — GPT created a short proof on the fly, and not just for $a = 3/2$; Perplexity failed at that. The use of synonyms and links between knowledge elements can help achieve this goal.
- Depth, conciseness and **disambiguation** are not easy to measure. Some users see long English sentences superior to concise bullet lists grouped into sections in the prompt results. For others, the opposite is true. Experts prefer depth, but it can be overwhelming to the layman. Disambiguation can be achieved by asking the user to choose between different contexts; this option is not available in most LLMs. As a

result, these qualities are rarely tested. The best LLMs, in my opinion, combine depth and conciseness, providing in much less text far more information than their competitors. Metrics based on entropy could help in this context.

- Ease of use is rarely tested, as all UIs consist of a simple search box. However, xLLM is different, allowing the user to select agents, negative keywords, and sub-LLMs (subcategories). It also allows you to fine-tune or debug in real-time thanks to intuitive parameters. In short, it is more user-friendly. For details, see [here](#).
- Security is also difficult to measure. But LLMs with local implementation, not relying on external APIs and libraries, are typically more secure. Finally, a good evaluation metric should take into account the **relevancy scores** displayed in the results. As of now, only xLLM displays such scores, telling the user how confident it is in its answer.

10.2.4 Benefits of untrained LLMs

In view of the preceding arguments, one might wonder if training is necessary, especially since training is not done to provide better answers, but to better predict the next tokens. Without training, its heavy computations and black-box deep neural networks machinery, LLMs can be far more efficient yet even more accurate. To generate answers in English prose based on the retrieved material, one may use template, pre-made answers where keywords can be plugged in. This can turn specialized sub-LLMs into giant Q&A's with a million questions and answers covering all potential inquiries. And deliver real ROI to the client by not requiring GPU and expensive training.

This is why I created xLLM, and the reason why I call it LLM 2.0, with its radically different architecture and next-gen features including the unique UI. I describe it in detail in Part I in this book. Additional research papers are available [here](#).

That said, xLLM still requires fine-tuning, for back-end and front-end parameters. It also allows the user to test and select his favorite parameters. Then, it blends the most popular ones to automatically create default parameter sets. I call it **self-tuning**. It is possibly the simplest **reinforcement learning** strategy.

10.3 Overview of xLLM main differentiators

Due to the innovative architecture and next gen features, xLLM constitutes a milestone in LLM development, moving away from the deep neural network (DNN) machinery and its expensive black-box training and GPU reliance, while delivering more accurate results to professional users, especially for enterprise applications. In Table 10.3, I summarize some of the most innovative contributions, justifying the label LLM 2.0. In the table below, KG stands for knowledge graph.

Table 10.3: Comparing LLM 2.0 with LLM 1.0

xLLM (LLM 2.0)	Standard LLMs (LLM 1.0)
Solid foundations to design robust back-end architecture from the ground up, retrieve and leverage the knowledge graph from the corpus (smart crawling). Hallucination-free, no need for prompt engineering. Zero weight. Suggested alternate prompts based on embeddings.	Poor back-end architecture. Knowledge graph built on top (top-down rather than bottom-up approach). Needs prompt engineering and billions of weights. Yet, the success depends more on auxiliary subsystems rather than on the core DNN engine.
Few tokens: “real estate San Francisco” is 2 tokens. Contextual chunks, KG and contextual tokens with non-adjacent words, sorted n -grams, customizable PMI metric for keyword associations, variable-length embeddings, in-memory nested hashes for KG back-end DB.	Tons of tiny tokens. Fixed-size chunks and embeddings are common. Vector databases, dot product and cosine similarity instead of PMI. Reliance on faulty Python libraries for NLP. One type of token: no KG or contextual tokens.
Focus on conciseness, accuracy, depth and exhaustivity in prompt results. Normalized relevancy scores displayed to the user, to warn him of potential poor answers when corpus has gaps. Augmentation and use of synonyms to map prompt keywords to tokens in backend tables, to boost exhaustivity and minimize gaps. Prompt results (front-end) distillation.	Focus on lengthy English prose aimed at novices, in prompt results. Evaluation metrics do not measure exhaustivity or depth. No relevancy scores shown to the user, or used in model evaluation. No mechanism to reduce gaps other than augmentation. Back-end distillation needed to fix poor corpus or oversized token lists.

Specialized sub-LLMs with LLM router. User can choose categories, agents (built in the back-end), and negative keywords. Or fine tune front-end intuitive parameter in real-time, with debugging option. Process prompts in bulk. Fine tune back-end parameters. Popular user-chosen parameters used for self-tuning, to generate default parameter sets. No training needed. Parameters local to sub-LLM, or global.	User interface limited to basic search box, doing one prompt at a time. No real-time fine-tuning, little if any customization available: the system guesses user intents (the agents). Fine-tuning for developers only, may require re-training the full model (costly), and it is based on black-box parameters rather than explainable AI. Needs regular training as new keywords show up and the model is not trained on them.
Use of multi-index and deep retrieval techniques (e.g. for PDFs). Highly secure (local, authorized users). Can connect to other LLMs or call home-made apps (NoGAN, LLM for clustering or predictions). Taxonomy and KG augmentation. Pre-made template answers with keyword plugging to cover many prompts.	Single index. Proprietary and standard libraries may miss some tables, graphs and other elements in PDFs: shallow retrieval. No KG Augmentation. Data leakage; security and liability issues (hallucinations). Long answers favored over conciseness and structured output.

Part III

Innovations in Statistical AI

Chapter 11

Building a Ranking System to Enhance Prompt Results

This chapter features the most recent developments to the enterprise version of xLLM described in Part I. The focus here is on returning the best extracts from the corpus, to a prompt or user query, in the context of search and retrieval (**RAG**). These extracts are called **text entities** and may consist of a webpage, part of a webpage, a few text sentences, part of a PDF document, or a JSON entity extracted from the input corpus. Given a prompt and the size of the corpus, their number can be large, thus the need to only return those that are most relevant to the prompt.

Before digging into the **relevancy score** architecture, I introduce important updates to the original enterprise model. I now call it **e-xLLM**, to emphasize its application to enterprise corpuses, and to differentiate it from other versions used for scientific research (see section 11.2.2), or for clustering and predictions (see chapter 6).

First, I augmented the corporate corpus used in my case study. The new corpus named `repository3.txt`, is on GitHub, [here](#). Text entities from the original corpus have an ID (also called **index**) between 0 and 1000, while the new ones have an ID between 2000 and 4000. I created a new tag to make it easy for the user to know whether some output comes from the original data, or from the augmentation.

Then, I created a library `xllm_enterprise_util.py` to move some code outside the main program `xllm-enterprise-v2.py`. The new program `xllm-enterprise-v2-user.py` imports that library, making it much shorter and easier to understand. But the main upgrade is in `xllm-enterprise-v2-dev.py`. It also imports the same library. Yet, rather than fine-tuning in real time, the goal is to process many prompts with known answer, in bulk. It also includes the relevancy scores discussed earlier, the main topic in this section.

Another improvement is the ability to link tokens from the prompt, to tokens in the corpus, via the `KW_map` table. Sometimes, a user enters a term such as ‘doing business as’, while in the corpus, the corresponding word is ‘dba’. The goal of `KW_map` is to address this issue, to increase **exhaustivity** when returning results. Perhaps the most striking new feature is the introduction of **graph tokens** from the **knowledge graph**, to boost relevancy by leveraging the global **context**. See section 11.1.

Finally, key components built in the first version include **multitokens** and contextual information such as tags, titles, categories, URLs, or related items, in text entities. Contextual elements are detected during the initial crawl and added to the **backend tables**. The scope of the context is global, not local as in most other LLMs. In some instances, contextual elements, for instance **agents**, are created after the initial crawl, typically via a clustering algorithm applied to the corpus. In the end, they are attached to the JSON-like **text entities**, as separate entries from the main text: see an example in Table 2.1. In-memory **nested hashes** and **sub-LLMs** (sometimes called **mixture of experts**), including the **LLM router**, are discussed in Part I.

I now describe the multitokens, as they significantly differ from tokens in other LLM architectures, both in terms on format and usage, as xLLM does not rely on neural networks, has zero weight and no training:

- **Standard multitokens**, for instance ‘San Francisco~real estate’, consist of one or multiple single tokens separated by ‘~’, in this case two single tokens ‘San Francisco’ and ‘real estate’. They may overlap: a same single token can be found in different multitokens, or split into sub-tokens.
- **Contextual multitokens** consist of tokens that are not adjacent in the prompt or corpus, yet found in a same text sub-entity. In this case, the single tokens are separated by `\wedge`, as in ‘San Francisco\wedge real estate’.
- **Graph multitokens** (a new addition) are multitokens found in the contextual fields attached to a text entity, such as sub-category. They start with ‘`_`’ as in ‘`_San Francisco~real estate`’. They are not stored

in the dictionary to save space. Instead, they are found in `hash_ID`, a key-value table where the key is a text entity ID, and the value is a list of multitokens attached to it. See the `update_tables` function in `xllm_enterprise_util.py`.

In the code, I use `gword` to represent graph multitokens.

11.1 Relevancy scores and rankings

The corporate corpus tested here consists of definitions and policies regarding AI integration. It is broken down in text entities (see example in Table 2.1). Each entity is indexed – it has an ID attached to it for easy retrieval – and consists of two types of data: raw text (the description field) and knowledge graph or contextual elements (the other fields, such as title, tag, or category). Some of the contextual fields such as `agent` have been added post-crawling, using a clustering algorithm applied to the corpus. Then, `hash_ID[ID]` lists all the multitokens attached to text entity `ID`, with occurrence count for each. The full content attached to the ID is in `ID_to_content[ID]`, mapping IDs to text entities.

When processing a prompt, xLLM extracts all multitokens that are also present in the backend dictionary (itself built on the corpus), using a synonyms and acronyms table `KW_map` for exhaustivity. The goal is to match them to multitokens found in text entities. This step does not involve `embeddings`, which are used for a different purpose: finding related words.

The data gathered prior to returning the query results consists of entity IDs and their multitokens that match those from the prompt. Several metrics are linked to this data. In particular, whether a multitone is in the contextual elements or in the regular text attached to a specific text entity, the length of the text entity and multitone, the number of single tokens in the multitokens, the occurrence count attached to the multitone across the whole corpus (stored in the multitone dictionary), and whether a multitone consists of adjacent tokens or not. From there, I build 4 scores S_A, S_B, S_C, S_D to measure the fit between a text entity (represented by its ID), and the prompt:

- S_A measures the importance of the multitokens found both in the text entity, and in the prompt.
- S_B is the number of multitokens found both in the text entity, and in the prompt (intersection).
- S_C is same as S_A , but for multitokens also found in the contextual fields in the text entity.
- S_D is same as S_B , but for multitokens also found in the contextual fields in the text entity.

These scores are computed in lines 326–341 in the code in section 11.3. In particular, the formula for S_A , for a specific text entity ID, is as follows:

$$S_A(\text{ID}) = \sum_{t \in M(\text{ID}, \text{P})} \lambda_t w_t^{-\beta_t}, \quad (11.1)$$

where $M(\text{ID}, \text{P})$ is the set of multitokens found both in prompt P and in the text entity ID. Here $\lambda_t = 1$ and $\beta_t = 0.50$. Note the analogy with Formula (6.2) used in xLLM for predictions, also based on inverse powers. It favors rare tokens, which bear more weight in specialized search.

Traditional LLMs may use a negative value for β_t , and cosine metrics and/or parameters λ_t obtained via gradient descent, typically with neural networks. There is an implicit step activation function in Formula (11.1): the fact that tokens outside $M(\text{ID}, \text{P})$ are ignored, further speeds up the computations. Such step functions are not recommended in neural networks, as it breaks the continuity and differentiability assumptions required for convergence of gradient descent. With xLLM, it is not an issue. Also neural networks are over-parameterized with millions of parameter sets producing the same output, to increase the odds of landing in a parameter configuration close to the optimum, during gradient descent. Again, xLLM does not face this problem.

One of the benefits is that no training is needed. It works with new tokens as the system is not pre-trained. Also, xLLM could be used as a starting point – a very good approximation close to the optimum – for LLMs relying on deep neural networks. Part of it is because xLLM bypasses all the continuity, linearity (tensors) and differentiability constraints attached to DNNs. But it remains to be seen whether DNNs can capitalize on this. In the case of tabular data synthetization, my NoGAN approach (see chapter 7) outperforms all DNNs both in terms of speed and quality, based on the best evaluation metrics.

I now describe how to build a ranking system for text entities, based on the four scores S_A, S_B, S_C, S_D . First, I sort the local `ID_hash` (a transposed of `hash_ID`) containing all the text entity IDs relevant to the prompt, in descending order, for each of the 4 scores. For each score, I only keep the ID ranks. See how `ID_hash` is built, in lines 315–324 in the code. Then I compute a global rank for each ID, using the function `rank_ID`. See line 345. The `rank_ID` function is listed in lines 214–239. In short, the global rank is a weighted combination of the 4 ranks originating from the 4 sorted scores: see line 236. The lower weights attached

to scores S_B, S_C, S_D gives them more importance over S_A , as they are related either to multitokens found in knowledge graph elements, or to multiple tokens consisting of several tokens.

11.2 Case study

Figures 11.1 and 11.2 show two prompts, and the text entity IDs retrieved by xLLM. To see the full content of these text entities, look at the anonymized input corpus `repository3.txt` on GitHub, [here](#). Each row is a text entity, starting with its ID. Note that IDs below 2000 (at the bottom of the file on GitHub) correspond to the original corpus, while IDs 2000 and higher comes from some augmentation.

```
-----
Prompt: List some data assets under the 'people' Business domain.
Cleaned: ['assets', 'business', 'data', 'domains', 'list', 'people']
-----
Most relevant text entities:

      ID wRank ID_Tokens
24      7 {'business~domains': 1, 'business~domains~list': 1}
433     10 {'domains~data': 1}
151     10 {'business~data': 1}
2701    12 {'people': 1, '__people': 1}
42      12 {'data~assets': 1}
48      12 {'data~assets': 1}
90      12 {'data~assets': 1}
91      12 {'data~assets': 1}
92      12 {'data~assets': 1}
199     12 {'data~assets': 1}
```

Figure 11.1: Text entities retrieved by xLLM and associated multitokens, first example

```
-----
Prompt: Identify the parent DLZ of the geographical zone AFR in the MLTxQuest hierarchy.
Cleaned: ['afr', 'dlz', 'geographical', 'hierarchy', 'identify', 'mltxquest', 'zone']
-----
Most relevant text entities:

      ID wRank ID_Tokens
2919    9 {'zone': 1, 'hierarchy': 1}
433     9 {'zone': 1, 'hierarchy': 1}
2012    10 {'zone': 1, 'mltxquest': 1, 'afr': 1}
2655    14 {'hierarchy': 1}
2075    14 {'geographical': 1}
2479    14 {'geographical': 1}
2844    14 {'geographical': 1}
3252    15 {'mltxquest': 1, 'afr': 1}
2154    16 {'zone': 1, 'mltxquest': 1, 'dlz': 1}
2220    19 {'zone': 1, 'dlz': 1}
```

Figure 11.2: Text entities retrieved by xLLM and associated multitokens, second example

Multitoken counters are all equal to 1 here because the text entities are short and do not contain a same multitoken more than once. The order in which IDs are shown depends heavily on β (see `beta=0.50` in line 249 in the code) and the weight vector [2, 1, 1, 1] attached to the 4 scores S_A, S_B, S_C, S_D , initialized in line 236 in the code. It also depends on the size and number of text entities, that is, on the `chunking` mechanism. Significant improvement can be achieved by adding entries to `KW_map`, for instance ‘pple’ pointing to ‘people’, and ‘PeopleDomain’ pointing to ‘people domain’. Or via additional NLP when parsing the corpus.

The 10 text entity IDs shown in Figures 11.1 and 11.2 are only a small subset of what xLLM found, selected via the ranking mechanism. Also note the token starting with ‘`_`’ in Figure 11.1, identifying a `graph token`. It comes from a knowledge graph element in text entity 2701, rather than from regular text.

11.2.1 xLLM for auto-indexing, cataloging and glossary generation

In the end, one has to question the need for billions of tokens and weights, when less than a few thousands cover more than what could come from prompts. The total number of meaningful prompts of any length, after cleaning, is probably well below a million for this sub-LLM. This includes all past and future prompts. One

might generate all of them with a **prompt synthesizer** and then create the answer for each of them, resulting in Q&A list with a million entries, easily manageable.

Besides search and retrieval, another possible application of xLLM is to automatically update the corpus by adding relevant material in text entities, based on augmentation, or detecting and deduping redundant entries. Or to build a taxonomy on the corpus, possibly seeded using some external taxonomy, via **taxonomy augmentation**. You can also use xLLM for **cataloging** or as an **auto-indexer** and **glossary generation** for books, large websites or repositories: it will automatically detect index entries and sub-entries, create the full index or glossary, and flag the corresponding terms in the corpus using a mechanism similar to text entity IDs. At the time of writing, the only product offering this capability is **notebookML**.

Another topic of interest is to study the patterns found in the corpus, versus those found in prompts, to eventually increase the compatibility between what the user is looking for, and what is available in the corpus. Stopwords lists attached to prompts may be different from those attached to the corpus.

Finally, xLLM can be used along with other LLMs to re-inforce or judge each other. Large companies do not stick to just one product. They like to have more than just one tool. One of the challenges is automated **model evaluation**: xLLM returns concise yet exhaustive bullet lists with a score attached to each item, see Figure 11.4. How do you assess exhaustivity? How to take into account the relevancy scores in your evaluation metric? In chapter 5, I discuss an approach based on the ability to correctly reconstruct the underlying taxonomy in the corpus. Another idea is to use xLLM to generate an index, and compare it with the existing one in the corpus.

11.2.2 xLLM for scientific research

The first version of xLLM was not intended for corporate clients, but for scientific research. The goal was to find references, links, and related topics to specific research questions or keywords. Yet it is based on the same architecture, refined over time with the introduction of **nested hashes** as the core database structure. See section 7.2 in [13] for a detailed description, including smart crawling to retrieve the taxonomy embedded in the corpus. In this example, the Wolfram website and its large, high-quality taxonomy.

```

ORGANIC URLs

5 https://mathworld.wolfram.com/CentralLimitTheorem.html
3 https://mathworld.wolfram.com/LyapunovCondition.html
2 https://mathworld.wolfram.com/NormalDistribution.html
2 https://mathworld.wolfram.com/Feller-LevyCondition.html
2 https://mathworld.wolfram.com/LindebergCondition.html
2 https://mathworld.wolfram.com/Lindeberg-FellerCentralLimitTheorem.html
1 https://mathworld.wolfram.com/Berry-EsseenTheorem.html
1 https://mathworld.wolfram.com/ExtremeValueDistribution.html
1 https://mathworld.wolfram.com/WeakLawofLargeNumbers.html

CATEGORIES & LEVELS

5 Central Limit Theorem | Limit Theorems | 4
3 Lyapunov Condition | Limit Theorems | 4
2 Normal Distribution | Continuous Distributions | 4
2 Feller-Levy Condition | Limit Theorems | 4
2 Lindeberg Condition | Limit Theorems | 4
2 Lindeberg-Feller Central Limit Theorem | Limit Theorems | 4
1 Berry-Esseen Theorem | Moments | 3
1 Extreme Value Distribution | Continuous Distributions | 4
1 Weak Law of Large Numbers | Limit Theorems | 4

RELATED

13 Central Limit Theorem
9 Berry-Esseen Theorem
7 Lindeberg Condition
5 Fourier Transform--Gaussian
5 Lindeberg-Feller Central Limit Theorem
5 Lyapunov Condition
4 Normal Distribution Function
4 Feller-Levy Condition
2 Binomial Distribution
2 Bivariate Normal Distribution

```

Figure 11.3: xLLM for scientific research, looking for ‘central limit theorem’

It turns out that many websites including Wikipedia or Amazon, as well as corporate corpuses, have a very similar structure. It makes it easy to adapt the system to a variety of environments. The reason to include

xLLM for research in this section, is to show how it performs compared to others. If you look at Figure 11.3 and 11.4, it is obvious that xLLM serves professional users a lot better than OpenAI. Other comparisons with Google search, Gemini, Bing AI, and even the internal Wolfram search box, lead to the same conclusion: they all favor rudimentary results, with long English sentences that take time to read, returning information that you already know. By contrast, xLLM returns concise but exhaustive bullet lists grouped by section, with a relevancy score attached to each item, and deep rather than shallow content.

The Python code, input data (full crawled Wolfram corpus covering mathematics and computer science), as well as backend tables, are available on GitHub, [here](#). The sub-folder enterprise deals with the enterprise version discussed in sections 11.1 and 11.2.

The screenshot shows a ChatGPT interface. The user has asked for information about the Central Limit Theorem. The AI response is as follows:

ChatGPT

The Central Limit Theorem (CLT) is a fundamental concept in probability theory and statistics. It states that, under certain conditions, the distribution of the sum (or average) of a large number of independent, identically distributed random variables will be approximately normally distributed, regardless of the original distribution of the individual variables.

Here are the key points of the Central Limit Theorem:

1. **Independence:** The random variables should be independent of each other. This means that the occurrence of one event does not affect the occurrence of another.
2. **Identically Distributed:** The random variables should be drawn from the same probability distribution. This ensures that they have the same mean and standard deviation.
3. **Large Sample Size:** The theorem is most effective for large sample sizes. As a general rule of thumb, a sample size of 30 or more is often considered large enough for the CLT to apply, although this can vary depending on the underlying distribution.

The Central Limit Theorem has significant implications for statistical inference. Even if the original population distribution is not normal, the distribution of sample means from that population will tend to be normal for sufficiently large sample sizes. This normal distribution is often used to make inferences about population parameters using methods such as hypothesis testing and confidence intervals.

In practical terms, the Central Limit Theorem is frequently employed when working with sample means, as it allows statisticians to make assumptions about the distribution of sample means even when the original population distribution is unknown or not normally distributed.

Figure 11.4: Looking for ‘central limit theorem’, using OpenAI GPT

11.3 Python code

The Python code here pertains to the enterprise version v2 of xLLM, discussed in sections 11.1 and 11.2. It is on GitHub, in my Large-Language-Models/xllm6/enterprise/ sub-folder, [here](#). Look for

- `xllm-enterprise-v2-dev.py`, the main program listed here,
- `xllm_enterprise_util.py`, the accompanying library,
- `repository3.txt`, the anonymized augmented corpus used as input source,
- `enterprise_sample_prompts.txt`, a list of sample prompts,
- `xllm-enterprise-v2-dev-output.txt`, the output results for all sample prompts.

```

1 import xllm_enterprise_util as exllm
2
3 #--- Backend: create backend tables based on crawled corpus
4
5 tableName = (
6     'dictionary', # multitokens (key = multitoken)
7     'hash_pairs', # multitoken associations (key = pairs of multitokens)
8     'ctokens',    # not adjacent pairs in hash_pairs (key = pairs of multitokens)

```

```

9  'hash_context1', # categories (key = multitoken)
10 'hash_context2', # tags (key = multitoken)
11 'hash_context3', # titles (key = multitoken)
12 'hash_context4', # descriptions (key = multitoken)
13 'hash_context5', # meta (key = multitoken)
14 'hash_ID', # text entity ID table (key = multitoken, value is list of IDs)
15 'hash_agents', # agents (key = multitoken)
16 'full_content', # full content (key = multitoken)
17 'ID_to_content', # full content attached to text entity ID (key = text entity ID)
18 'ID_to_agents', # map text entity ID to agents list (key = text entity ID)
19 'ID_size', # content size (key = text entity ID)
20 'KW_map', # for singularization, map kw to single-token dictionary entry
21 'stopwords', # stopword list
22 )
23
24 backendTables = {}
25 for name in tableNames:
26     backendTables[name] = {}
27
28 stopwords = ('', '-', 'in', 'the', 'and', 'to', 'of', 'a', 'this', 'for', 'is', 'with', 'from',
29             'as', 'on', 'an', 'that', 'it', 'are', 'within', 'will', 'by', 'or', 'its', 'can',
30             'your', 'be', 'about', 'used', 'our', 'their', 'you', 'into', 'using', 'these',
31             'which', 'we', 'how', 'see', 'below', 'all', 'use', 'across', 'provide', 'provides',
32             'aims', 'one', '&', 'ensuring', 'crucial', 'at', 'various', 'through', 'find', 'ensure',
33             'more', 'another', 'but', 'should', 'considered', 'provided', 'must', 'whether',
34             'located', 'where', 'begins', 'any', 'what', 'some', 'under', 'does', 'belong',
35             'included', 'part', 'associated')
36 backendTables['stopwords'] = stopwords
37
38 # agent_map works, but hash structure should be improved
39 # key is word, value is agent (many-to-one). Allow for many-to-many
40 agent_map = {
41     'template':'Template',
42     'policy':'Policy',
43     'governance':'Governance',
44     'documentation':'Documentation',
45     'best practice':'Best Practices',
46     'bestpractice':'Best Practices',
47     'standard':'Standards',
48     'naming':'Naming',
49     'glossary':'Glossary',
50     'historical data':'Data',
51     'overview':'Overview',
52     'training':'Training',
53     'genai':'GenAI',
54     'gen ai':'GenAI',
55     'example':'Example',
56     'example1':'Example',
57     'example2':'Example',
58 }
59
60 KW_map = {}
61 save_KW_map = False
62 try:
63     IN = open("KW_map.txt", "r")
64 except:
65     print("KW_map.txt not found on first run: working with empty KW_map.")
66     print("KW_map.txt will be created after exiting if save = True.")
67     save_KW_map = True
68 else:
69     # plural in dictionary replaced by singular form
70     content = IN.read()
71     pairs = content.split('\n')
72     for pair in pairs:
73         pair = pair.split('\t')
74         key = pair[0]
75         if len(pair) > 1:
76             KW_map[key] = pair[1]
77     IN.close()
78
79 # manual additions (plural not in prompt but not dictionary, etc.)
80 KW_map['domains'] = 'domain'
81 KW_map['doing business as'] = 'dba'
82
83 backendTables['KW_map'] = KW_map
84

```

```

85 backendParams = {
86     'max_multitoken': 4, # max. consecutive terms per multi-token for inclusion in dictionary
87     'maxDist' : 3, # max. position delta between 2 multitokens to link them in hash_pairs
88     'maxTerms': 3, # maxTerms must be <= max_multitoken
89     'extraWeights' : # deafault weight is 1
90     {
91         'description': 0.0,
92         'category': 0.3,
93         'tag_list': 0.4,
94         'title': 0.2,
95         'meta': 0.1
96     }
97 }
98
99
100 #--- Read repository and create all backend tables
101
102 # https://raw.githubusercontent.com/VincentGranville
103 #   /Large-Language-Models/refs/heads/main/xllm6/enterprise/repository3.txt
104
105 IN = open("repository3.txt", "r")
106 data = IN.read()
107 IN.close()
108
109 entities = data.split("\n")
110 ID_size = backendTables['ID_size']
111
112 # to avoid duplicate entities (takes space, better to remove them in the corpus)
113 entity_list = ()
114
115 for entity_raw in entities:
116
117     entity = entity_raw.split("~~")
118     agent_list = ()
119
120     if len(entity) > 1 and entity[1] not in entity_list:
121
122         entity_list = (*entity_list, entity[1])
123         entity_ID = int(entity[0])
124         entity = entity[1].split("(")
125         hash_crawl = {}
126         hash_crawl['ID'] = entity_ID
127         ID_size[entity_ID] = len(entity[1])
128         hash_crawl['full_content'] = entity_raw # do not build to save space
129
130     key_value_pairs = exllm.get_key_value_pairs(entity)
131
132     for pair in key_value_pairs:
133
134         if ":" in pair:
135             key, value = pair.split(": ", 1)
136             key = key.replace("'", "")
137             if key == 'category_text':
138                 hash_crawl['category'] = value
139             elif key == 'tags_list_text':
140                 hash_crawl['tag_list'] = exllm.clean_list(value)
141             elif key == 'title_text':
142                 hash_crawl['title'] = value
143             elif key == 'description_text':
144                 hash_crawl['description'] = value # do not build to save space
145             elif key == 'tower_option_tower':
146                 hash_crawl['meta'] = value
147             if key in ('category_text', 'tags_list_text', 'title_text'):
148                 for word in agent_map:
149                     if word in value.lower():
150                         agent = agent_map[word]
151                         if agent not in agent_list:
152                             agent_list = (*agent_list, agent)
153
154             hash_crawl['agents'] = agent_list
155             exllm.update_dict(backendTables, hash_crawl, backendParams)
156
157
158 #--- Create embeddings
159
160 embeddings = {} # multitoken embeddings based on hash_pairs

```

```

161
162 hash_pairs = backendTables['hash_pairs']
163 dictionary = backendTables['dictionary']
164
165 for key in hash_pairs:
166     wordA = key[0]
167     wordB = key[1]
168     nA = dictionary[wordA]
169     nB = dictionary[wordB]
170     nAB = hash_pairs[key]
171     pmi = nAB/(nA*nB)**0.5 # try: nAB/(nA + nB - nAB)
172     # if nA + nB <= nAB:
173     #     print(key, nA, nB, nAB)
174     exlm.update_nestedHash(embeddings, wordA, wordB, pmi)
175     exlm.update_nestedHash(embeddings, wordB, wordA, pmi)
176
177
178 #-- Create sorted n-grams
179
180 sorted_ngrams = {} # to match ngram prompts with embeddings entries
181
182 for word in dictionary:
183     tokens = word.split('`')
184     tokens.sort()
185     sorted_ngram = tokens[0]
186     for token in tokens[1:len(tokens)]:
187         sorted_ngram += "~~" + token
188     exlm.update_nestedHash(sorted_ngrams, sorted_ngram, word)
189
190 # print top multitokens: useful to build agents, along with sample prompts
191 # for key in dictionary:
192 #     if dictionary[key] > 20:
193 #         print(key, dictionary[key])
194
195
196 #--- Functions used to score results ---
197
198 def rank(hash):
199     # sort hash, then replace values with their rank
200
201     hash = dict(sorted(hash.items(), key=lambda item: item[1], reverse=True))
202     rank = 0
203     old_value = 999999999999
204
205     for key in hash:
206         value = hash[key]
207         if value < old_value:
208             rank += 1
209         hash[key] = rank
210         old_value = value
211     return(hash)
212
213
214 def rank_ID(ID_score):
215     # attach weighted relevancy rank to text entity ID, with respect to prompt
216
217     ID_score0 = {}
218     ID_score1 = {}
219     ID_score2 = {}
220     ID_score3 = {}
221
222     for ID in ID_score:
223         score = ID_score[ID]
224         ID_score0[ID] = score[0]
225         ID_score1[ID] = score[1]
226         ID_score2[ID] = score[2]
227         ID_score3[ID] = score[3]
228
229     ID_score0 = rank(ID_score0)
230     ID_score1 = rank(ID_score1)
231     ID_score2 = rank(ID_score2)
232     ID_score3 = rank(ID_score3)
233
234     ID_score_ranked = {}
235     for ID in ID_score:
236         weighted_rank = 2*ID_score0[ID] + ID_score1[ID] + ID_score2[ID] + ID_score3[ID]

```

```

237     ID_score_ranked[ID] = weighted_rank
238     ID_score_ranked = dict(sorted(ID_score_ranked.items(), key=lambda item: item[1]))
239     return(ID_score_ranked)
240
241
242 #--- Main: processing prompts ---
243
244 print("\n")
245 input_ = " " # ignore, not used here
246 saved_query = ""
247 get_bin = lambda x, n: format(x, 'b').zfill(n)
248 frontendParams = exllm.default_frontendParams()
249 beta = 0.5 # overwrite 'beta' frontend param
250 ID_to_content = backendTables['ID_to_content']
251
252
253 #--- Main: Read sample prompts with correct answer ---
254
255 # https://raw.githubusercontent.com/VincentGranville
256 # /Large-Language-Models/references/main/xllm6/enterprise/enterprise_sample_prompts.txt
257
258 IN = open("enterprise_sample_prompts.txt", "r")
259 prompts = IN.read()
260 prompts = prompts.split("\n")
261
262 # --- Main: Look over all prompts ---
263
264 for query in prompts:
265
266     query = query.split("|") [0]
267     print("-----")
268     print("Prompt: ", query)
269     query = query.replace('?', ' ').replace('(', ' ').replace(')', ' ')
270     query = query.replace('"', '').replace("\s", '')
271     query = query.split(' ')
272     new_query = []
273     for k in range(len(query)):
274         token = query[k].lower()
275         if token in KW_map:
276             token = KW_map[token]
277         if token in dictionary:
278             new_query.append(token)
279     query = new_query.copy()
280     query.sort()
281     print("Cleaned:", query)
282     print("-----")
283
284 q_embeddings = {}
285 q_dictionary = {}
286
287 # --- build q_dictionary and q_embeddings based on prompt tokens ---
288
289 for k in range(1, 2*len(query)):
290
291     binary = get_bin(k, len(query))
292     sorted_word = ""
293     for k in range(0, len(binary)):
294         if binary[k] == '1':
295             if sorted_word == "":
296                 sorted_word = query[k]
297             else:
298                 sorted_word += "~" + query[k]
299
300     if sorted_word in sorted_ngrams:
301         ngrams = sorted_ngrams[sorted_word]
302         for word in ngrams:
303             if word in dictionary:
304                 q_dictionary[word] = dictionary[word]
305                 if word in embeddings:
306                     embedding = embeddings[word]
307                     for token in embedding:
308                         # pmi = embedding[token]
309                         pmi = exllm.custom_pmi(word, token, backendTables)
310                         q_embeddings[(word, token)] = pmi
311
312 # --- Scoring and selecting what to show in prompt results ---

```

```

313
314     exllm.distill_frontendTables(q_dictionary, q_embeddings, frontendParams)
315     hash_ID = backendTables['hash_ID']
316     ID_hash = {} # local, transposed of hash_ID; key = ID; value = multitoken list
317
318     for word in q_dictionary:
319         for ID in hash_ID[word]:
320             exllm.update_nestedHash(ID_hash, ID, word, 1)
321             gword = "__" + word # graph multitoken
322             if gword in hash_ID:
323                 for ID in hash_ID[gword]:
324                     exllm.update_nestedHash(ID_hash, ID, gword, 1)
325
326     ID_score = {}
327     for ID in ID_hash:
328         # score[0] is inverse weighted count
329         # score[1] is raw number of tokens found
330         score = [0, 0] # based on tokens present in the entire text entity
331         gscore = [0, 0] # based on tokens present in graph
332         for token in ID_hash[ID]:
333             if token in dictionary:
334                 score[0] += 1/(q_dictionary[token]**beta)
335                 score[1] += 1
336             else:
337                 # token must start with "__" (it's a graph token)
338                 token = token[2:len(token)]
339                 gscore[0] += 1/(q_dictionary[token]**beta)
340                 gscore[1] += 1
341         ID_score[ID] = [score[0], score[1], gscore[0], gscore[1]]
342
343     # --- Print results ---
344
345     ID_score_ranked = rank_ID(ID_score)
346     n_ID = 0
347     print("Most relevant text entities:\n")
348     print("\n  ID wRank ID_Tokens")
349     for ID in ID_score_ranked:
350         if n_ID < 10:
351             # content of text entity ID not shown, stored in ID_to_content[ID]
352             print(" %5d %3d %s" %(ID, ID_score_ranked[ID], ID_hash[ID]))
353             n_ID += 1
354
355     print("\nToken count (via dictionary):\n")
356     for key in q_dictionary:
357         print(" %4d %s" %(q_dictionary[key], key))
358
359     q_embeddings = dict(sorted(q_embeddings.items(), key=lambda item: item[1], reverse=True))
360     n_words = 0
361     print("\nTop related tokens (via embeddings):\n")
362     for word in q_embeddings:
363         pmi = q_embeddings[word]
364         if n_words < 10:
365             print(" %5.2f %s" %(pmi, word))
366         n_words += 1
367
368
369     #---- Save backend tables
370
371     def create_KW_map(dictionary):
372         # singularization
373         # map key to KW_map[key], here key is a single token
374         # need to map unseen prompt tokens to related dictionary entries
375         # example: ANOVA -> analysis~variance, ...
376
377         OUT = open("KW_map.txt", "w")
378         for key in dictionary:
379             if key.count('~') == 0:
380                 j = len(key)
381                 keyB = key[0:j-1]
382                 if keyB in dictionary and key[j-1] == 's':
383                     if dictionary[key] > dictionary[keyB]:
384                         OUT.write(keyB + "\t" + key + "\n")
385                     else:
386                         OUT.write(key + "\t" + keyB + "\n")
387
388         OUT.close()

```

```

389     return ()
390
391 if save_KW_map:
392     # save it only if it does not exist
393     create_KW_map(dictionary)
394
395 save = True
396 if save:
397     for tableName in backendTables:
398         table = backendTables[tableName]
399         OUT = open('backend_' + tableName + '.txt', "w")
400         OUT.write(str(table))
401         OUT.close()
402
403 OUT = open('backend_embeddings.txt', "w")
404 OUT.write(str(embeddings))
405 OUT.close()
406
407 OUT = open('backend_sorted_ngrams.txt', "w")
408 OUT.write(str(sorted_ngrams))
409 OUT.close()

```

11.4 Appendix: smart ranking in a nutshell

Here I present a high-level description of the algorithm introduced in section 11.1. The purpose is to decide, among dozens of candidate paragraphs selected from the corpus to answer a prompt, which ones to show in the results, and in what order. The goal is to maximize relevancy while not overwhelming the user with a long, cluttered answer. Think of it as the new PageRank for RAG/LLM, although the algorithm is radically different, and much simpler. The approach is generic and works for all RAG/LLM systems whether based on neural networks or not.

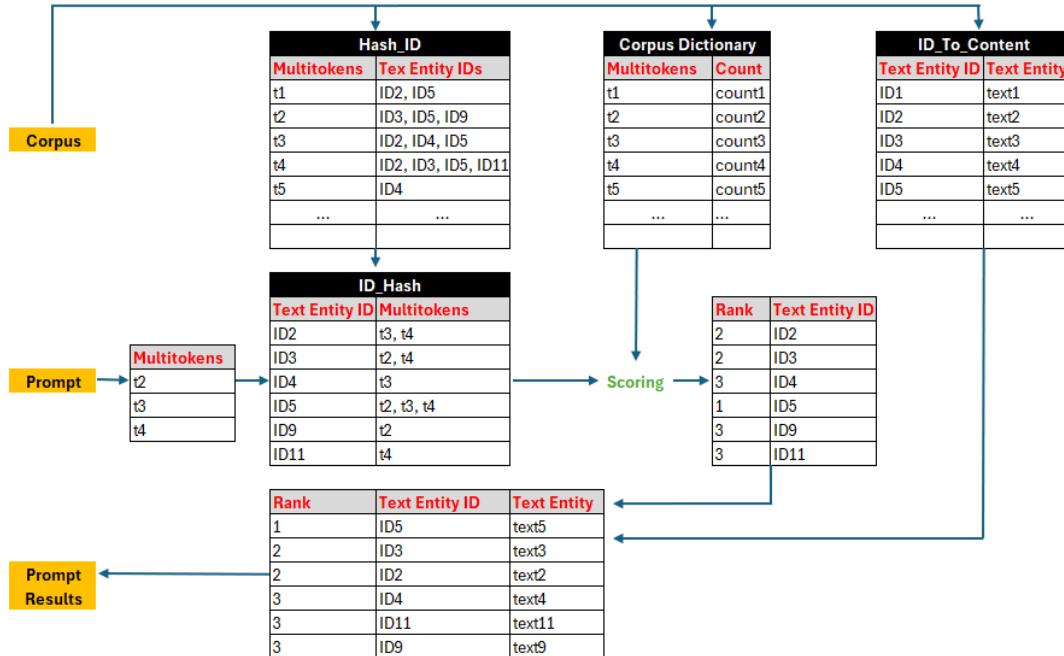


Figure 11.5: Relevancy scoring and ranking algorithm

The main steps, as illustrated in Figure 11.5, are:

1. Backend processing (linked to the corpus)

- (a) Split your corpus into text entities such as webpages, paragraphs, sections and so on. This step is similar to **chunking**. Attach an ID (called **index**) to each text entity.
- (b) Text entities have two types of fields: regular text like in all LLMs, and **knowledge graph** elements such as categories, related items, URL, tags, parent categories, title, and so on. These knowledge graphs elements are found while crawling and part of the original corpus. Or they can be added

after the full crawl. For instance, in xLLM, **agents** are assigned to text entities post-crawling, using a clustering algorithm.

- (c) You need two types of tokens: regular ones, and those found in the knowledge graph elements. The latter are called **graph tokens**. You then create a key-value table `Hash_ID`, where the key is a token, and the value is the list of text entity IDs attached to the token in question, with a token count for each one. Graph tokens start with ‘`_`’, to differentiate them from regular tokens.

2. Frontend processing (linked to the prompt)

- (a) You create a local, small key-value table `ID_Hash`, a transposed version of `Hash_ID`, where the key is a text entity ID, and the value is a list of tokens t found in the prompt, with ID in $\text{Hash_ID}[t]$.
- (b) For each ID, you compute (say) 4 **relevancy scores**: S_A, S_B corresponding to regular tokens, and S_C, S_D identical to S_A, S_B but for graph tokens. For instance, $S_B(\text{ID})$ is the number of regular tokens found both in the prompt and in text entity indexed as ID. More about S_A , below.
- (c) You sort the collected IDs according to each score, and assign 4 ranks (one per score) to each ID. The *global* rank is a weighted combinations of the 4 ranks, and subject to **fine-tuning**. In the prompt results, you display text entity IDs (their full content) with a global rank above some threshold, or the top 10 according to global rank.

Smart scoring is done as follows. Rare tokens present in the prompt (based on their occurrence in the corpus) may be boosted as they usually carry more specialized information. To achieve this, for $S_A(\text{ID})$, I use a sum of inverse powers of $n(t)$, where $n(t)$ is the number of occurrences of token t in the corpus. The sum is over all tokens t found both in the text entity indexed as ID, and in the corpus. There is an extra parameter β , representing the exponent, and subject to **fine-tuning**.

Chapter 12

Probabilistic Search: Alternative to Vector Search

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.

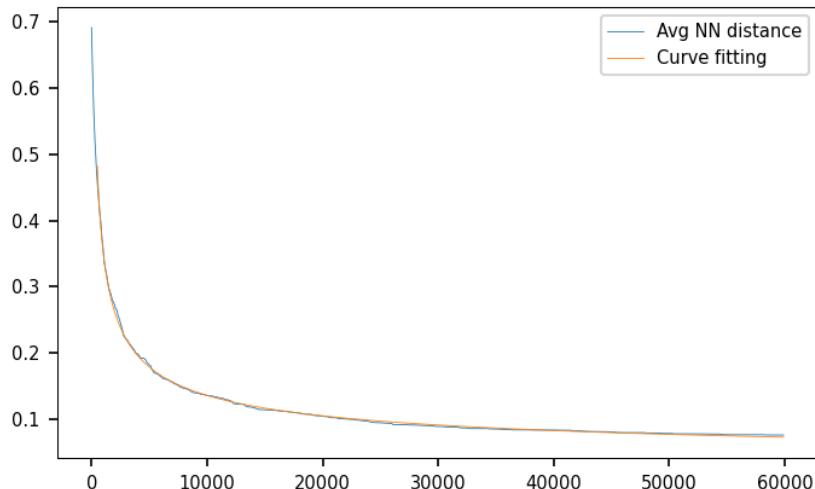


Figure 12.1: Average NN distance over time, with probabilistic ANN

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 12.1 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 [15].

12.1 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 [14]. 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:

$$\begin{aligned} S &= \{x_1, \dots, x_n\} \\ T &= \{y_1, \dots, y_m\}. \end{aligned}$$

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

$$L(\sigma) = \sum_{k=1}^n \|x_k - y_{\sigma(k)}\|. \quad (12.1)$$

The minimum is over all integer functions σ defined on $\{1, \dots, n\}$ with values in $\{1, \dots, m\}$. 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 12.1 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 15.1.

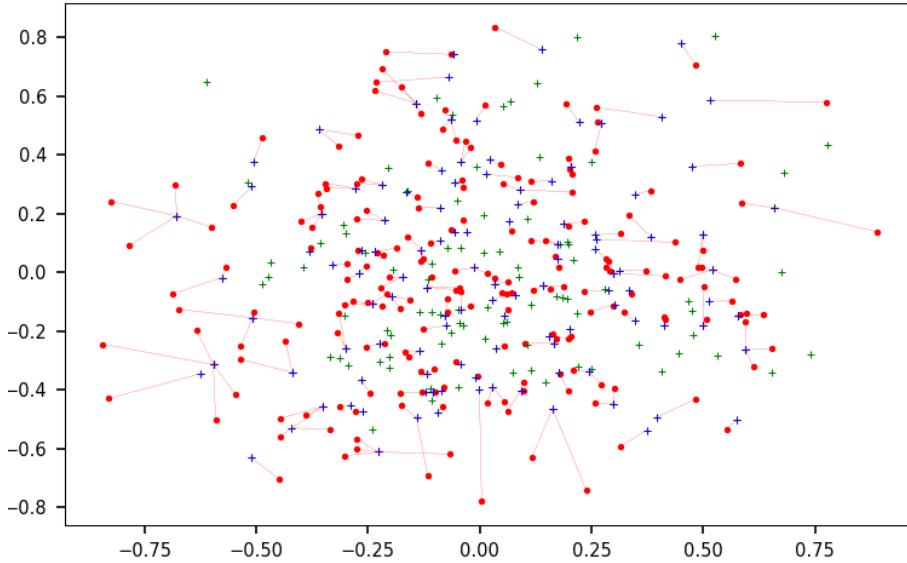


Figure 12.2: Approximate NNs from T (blue) to points in S (red) after a few thousand steps

Figure 12.2 shows the dataset used in Figure 12.1, 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.

12.2 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.

12.2.1 Embeddings and large language models

In large language models, embeddings are lists of tokens attached to a keyword. For instance, Table 12.1 is based on my specialized xLLM that answers research questions about statistics and probability: see section 7.2 in [13]. 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 12.1:

$$\begin{aligned} \text{normal} \bullet \text{Gaussian} &= 0.43903 \times 0.00858 + 0.05885 \times 0.01164 = 0.004452 \\ \text{binomial} \bullet \text{Gaussian} &= 0.11796 \times 0.00858 + 0.01117 \times 0.01164 = 0.001142 \\ \|\text{normal}\| &= 0.49678, \quad \|\text{Gaussian}\| = 0.05853, \quad \|\text{binomial}\| = 0.13998. \end{aligned}$$

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 magnitudes. It is computed as follows:

$$\begin{aligned} \rho(\text{normal}, \text{Gaussian}) &= \frac{\text{normal} \bullet \text{Gaussian}}{\|\text{normal}\| \cdot \|\text{Gaussian}\|} = 0.15311, \\ \rho(\text{binomial}, \text{Gaussian}) &= \frac{\text{binomial} \bullet \text{Gaussian}}{\|\text{normal}\| \cdot \|\text{Gaussian}\|} = 0.13940. \end{aligned}$$

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 12.1: Embeddings (one per word) with normalized PMI score attached to each token

When the goal is to compute all nearest neighbors within 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, \dots, N_q embeddings, standard k -NN over categories bundled together is $O(N^2)$ with $N = N_1 + \dots + N_q$, versus the much lower $O(N_1^2 + \dots + N_q^2)$ when the q categories are treated separately. With the ANN algorithm described in section 12, 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].

12.2.2 Generating and evaluating synthetic data

My first use of [probabilistic ANN](#) (pANN) was for synthesizing tabular data, see chapter 7 in [14]. 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 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 7. 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

$$\text{KS}(S, T) = \sup_z |F_s(z) - F_r(z)|,$$

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 in [13]. Several nearest neighbor search methods are discussed in the article “Comprehensive Guide To Approximate Nearest Neighbors Algorithms” [32].

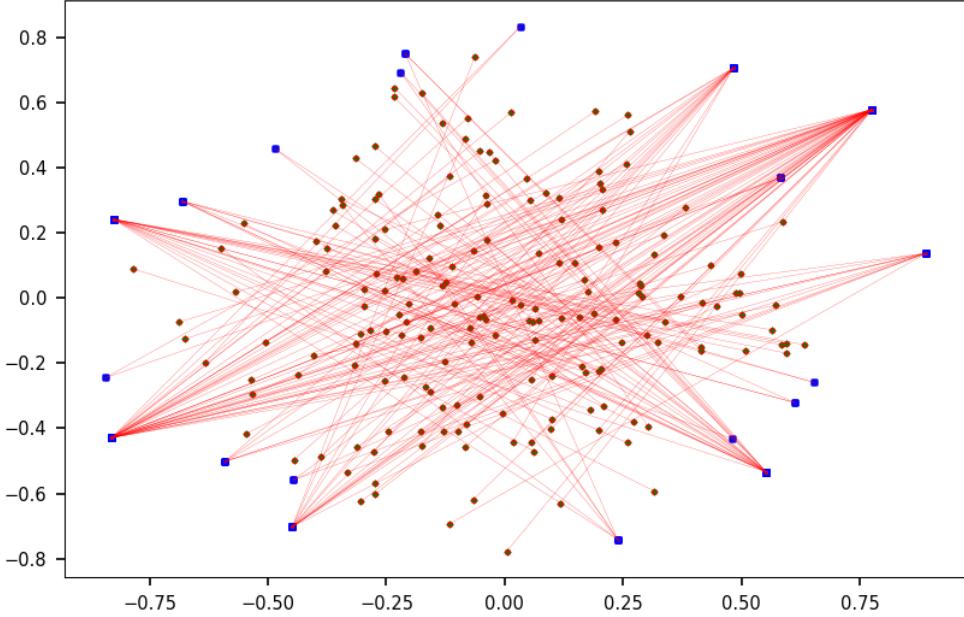


Figure 12.3: Extreme points in blue ($S = T$) obtained by maximizing the loss $L(\sigma)$

12.2.3 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[k1]` and `hash[k3]` in line 138 in the Python code in section 12.4. 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 $j = \text{arr_NN}[k_1]$. 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 has 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 12.2.2. 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 12.3.

12.3 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 12.4, 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 12.4, 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.

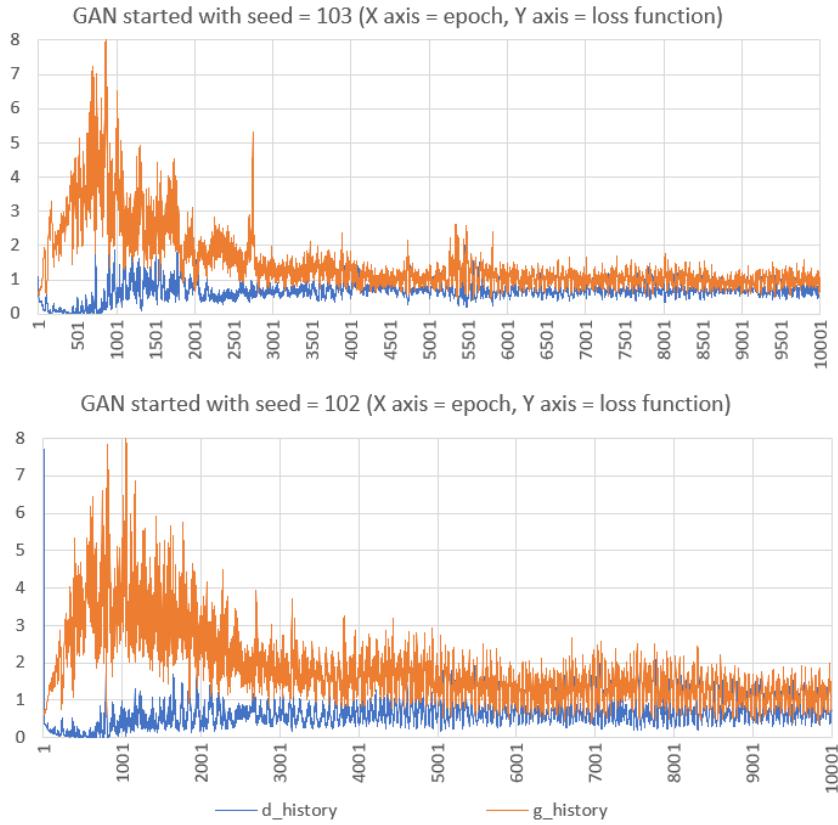


Figure 12.4: Loss function history, two versions of GAN

Step 3: Evaluation. The key metric Δ_t linked to the **loss function** (Formula 12.1) 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 $E(R)$ 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 12.4. The reason is for faster computations; the choice of the distance has very little impact. Then, the volume of the unit ball is $\nu = 2^d/d!$

12.4 Python code

See section 12.2.3 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
68     k = 0
69     for key in NNdist:
70         arr_NN[k] = arr_NN_tmp[key]
71         x[k] = x_tmp[key]
72         k += 1
73     return(x, arr_NN)
74
75 seed = 57
76 np.random.seed(seed)
77 eps = 0.0000000001
78
79 N = 200          # number of points in x[]
80 K = int(0.5 * N) # sort x[] by NN distance every K iterations
81 M = 200          # number of points in y[]
82
83 niter = 10000
84 mean = [0, 0]
85 cov = [(0.1, 0), (0, 0.1)]
86 x = np.random.multivariate_normal(mean, cov, size=N)
87 y = np.random.multivariate_normal(mean, cov, size=M)
88 # y = np.copy(x)
89 np.random.shuffle(x)
90 np.random.shuffle(y)
91
92 arr_NN = np.zeros(N)
93 arr_NN = arr_NN.astype(int)
94 hash = {}
95 sum_dist = 0
96
97 for k in range(N):
98
99     # nearest neighbor to x[k] can't be identical to x[k]
100    dist = 0
101
102    while dist < eps:
103        neighbor = int(np.random.randint(0, M))
104        dist = np.sum(abs(x[k] - y[neighbor]))
105
106    arr_NN[k] = neighbor
107    sum_dist += np.sum(abs(x[k] - y[neighbor]))
108    hash[k] = (-1,)
109
110 x, arr_NN = sort_x_by_NNdist_to_y(x, y, arr_NN)

```

```

111 low = sum_dist
112
113
114 #--- [3] Main part
115
116 mode = 'minDist' # options: 'minDist' or 'maxDist'
117 optimize = 'speed' # options: 'speed' or 'memory'
118 video = False # True if you want to produce a video
119 decay = 0.0
120
121 history_val = []
122 history_arg = []
123 flist = []
124 swaps = 0
125 steps = 0
126 frame = 0
127
128 for iter in range(niter):
129
130     k = iter % K
131     j = -1
132     while j in hash[k] and len(hash[k]) <= N:
133         # if optimized for memory, there is always only one iter in this loop
134         steps += 1
135         j = np.random.randint(0, M) # potential new neighbor y[j], to x[k]
136
137     if optimize == 'speed':
138         hash[k] = (*hash[k], j)
139
140     if len(hash[k]) <= N:
141
142         # if optimized for memory, then len(hash[k]) <= N, always
143         old_neighboor = arr_NN[k]
144         new_neighboor = j
145         old_dist = np.sum(abs(x[k] - y[old_neighboor]))
146         new_dist = np.sum(abs(x[k] - y[new_neighboor]))
147         if mode == 'minDist':
148             ratio = new_dist/(old_dist + eps)
149         else:
150             ratio = old_dist/(new_dist + eps)
151         if ratio < 1-decay/np.log(2+iter) and new_dist > eps:
152             swaps += 1
153             arr_NN[k] = new_neighboor
154             sum_dist += new_dist - old_dist
155             if sum_dist < low:
156                 low = sum_dist
157
158         if video and swaps % 4 == 0:
159
160             fname='ann_frame'+str(frame)+'.png'
161             flist.append(fname)
162             plot_frame()
163
164             # save image: width must be a multiple of 2 pixels, all with same size
165             # use save_image(fname,frame) in case of problems with plt.savefig
166             plt.savefig(fname, dpi = 200)
167             plt.close()
168             frame += 1
169
170         if iter % K == K-1:
171             x, arr_NN = sort_x_by_NNdist_to_y(x, y, arr_NN)
172
173         if iter % 100 == 0:
174             print("%6d %6d %6d %8.4f %8.4f"
175                  % (iter, swaps, steps, low/N, sum_dist/N))
176             history_val.append(sum_dist/N)
177             history_arg.append(steps) # try replacing steps by iter
178
179         history_val = np.array(history_val)
180         history_arg = np.array(history_arg)
181
182     if video:
183         clip = moviepy.video.io.ImageSequenceClip.ImageSequenceClip(flist, fps=6)
184         clip.write_videofile('ann.mp4')
185
186

```

```
187
188 #--- [4] Visualizations (other than the video)
189
190 plot_frame()
191 plt.show()
192
193 #- curve fitting for average NN distance (Y-axis) over time (X-axis)
194
195 # works only with mode == 'minDist'
196
197 def objective(x, a, b, c):
198     return(a + b*(x**c))
199
200 # ignore first offset iterations, where fitting is poor
201 offset = 5
202
203 x = history_arg[offset:]
204 y = history_val[offset:]
205
206 # param_bounds to set bounds on curve fitting parameters
207 if mode == 'minDist':
208     param_bounds=([0,0,-1],[np.inf,np.infty,0])
209 else:
210     param_bounds=([0,0,0],[np.inf,np.infty,1])
211
212 param, cov = curve_fit(objective, x, y, bounds = param_bounds)
213 a, b, c = param
214 # is c = -1/2 the theoretical value, assuming a = 0?
215 print("\n",a, b, c)
216
217 y_fit = objective(x, a, b, c)
218 ## plt.plot(x, y, linewidth=0.4)
219 plt.plot(history_arg, history_val, linewidth=0.4)
220 plt.plot(x, y_fit, linewidth=0.4)
221 plt.legend(['Avg NN distance','Curve fitting'],fontsize = 7)
222 plt.show()
```

Chapter 13

Strong Random Generators for Reproducible AI

Advances in number theory have led to the development of fast [random number generators](#) (PRNGs) with infinite period. In some instances, the formula is just one short line of code. Here, I introduce some of these PRNGs, as well as new statistical tests, better suited to detect lack of randomness. While the classic theory relies on poor test batteries and weak definition of randomness, the goal here is to produce strongly random sequences and better tools to assess their quality.

13.1 Strong Randomness and reproducibility: two key components

Modern GenAI apps rely on billions if not trillions of pseudo-random numbers. You find them in the construction of latent variables in nearly all deep neural networks and almost all applications: computer vision, synthetization, and LLMs. This component is overlooked. It is assumed that Python random functions do a good job, and very few seem to care about reproducibility, though customers do. Incidentally, all the GenAI apps that I create are fully reproducible, even those based on GANs (generative adversarial networks).

When producing so many random numbers or for strong encryption, you need top grade generators. The most popular one – adopted by Numpy and other libraries – is the [Mersenne Twister](#) [Wiki], and advertised as not fit for cryptography. It is known for its flaws: see chapter 4 in my book about chaos [9].

This material has its origins in the development of a new framework to prove the conjectured randomness of the digits of infinitely many simple math constants, such as $e^{1486/3331}$ or $\sqrt{491/3127}$. By randomness, I mean that if you don't know the name of the constant in question, you can not predict the next digit no matter how many past digits you collected, even if you collected more than there are atoms in the universe. In short, for all purposes, the sequence can not be distinguished – statistically speaking – from true random numbers. Note that in practical application, you use millions of such constants, typically using a few millions digits from each one, starting at arbitrary locations. The procedure is described in chapter 4 in [9].

The remaining focuses on three main areas. First, how to efficiently compute the digits of the mathematical constants in question to use them at scale. Then, some methodology to compare two types of random numbers: those generated by Python, versus those from the irrational numbers investigated here. Finally, proposing a new type of random digit sequence based on an incredibly simple formula leading to fast computations.

One of the benefits of my proposed random bit sequences, besides stronger randomness (infinite period) and fast implementation at scale, is to not rely on external libraries that may change over time. These libraries may change and render your results non-replicable in the long term if (say) Numpy decides to modify the internal parameters of its random generator.

Also, if your generated data is more random than the numbers used for testing its randomness, it will lead to false positives, flagging your experiment as not well randomized when actually the issue is with the test. In this chapter, I explain how to deal with this. Some of my tests involve predicting the value of a string given the values of previous strings in a sequence, a topic at the core of many large language models (LLMs).

Methods based on neural networks – mines being an exception – are notorious for hiding the seeds used in the various random generators involved. It leads to non-replicable results. It is my hope that this chapter will raise awareness about this issue, while offering better generators that do not depend on which library version you use. Last but not least, the datasets used here are infinite, giving you the opportunity to work with truly big data and infinite numerical precision.

13.2 Computing the digits of special math constants

Here, the constants must lie between 0 and 1. While I use the binary numeration system, the Python code also works with integer bases b other than 2. To introduce my method, I start with a textbook example: the exponential function. Let $\lambda_m(k) = (k \bmod m) + 1$. Define the following recursion with $k \geq 2$, assuming z and m are fixed integers with $z, m \geq 1$:

$$\begin{aligned} p_k(z, m) &= z \cdot \lambda_m(k - 1) \cdot p_{k-1}(z, m) + 1 \\ q_k(z, m) &= z \cdot \lambda_m(k - 1) \cdot q_{k-1}(z, m) \end{aligned}$$

The initial conditions are $p_1(m, z) = 0$ and $q_1(m, z) = z$. I also define the limit numbers

$$\xi(z, m) = \lim_{k \rightarrow \infty} \frac{p_k(z, m)}{q_k(z, m)}, \quad \xi(z) = \lim_{m \rightarrow \infty} \xi(z, m) = \exp\left(\frac{1}{z}\right) - \frac{z + 1}{z}. \quad (13.1)$$

If m is finite, then $\xi(z, m)$ is a rational number, albeit with a gigantic period even for rather small values of m . Finally, to make the discussion easier, I also introduce the notation

$$\xi_k(z, m) = \frac{p_k(z, m)}{q_k(z, m)} \quad (13.2)$$

The originality of my approach lies in the fact that for a fixed k , I am interested only in the first few digits of the rational number $\xi_k(z, m)$, that is, its [prefix](#), skipping the periodic part. Also, under the right conditions, the number of digits in the prefix, referred to as the prefix length, increases with k and eventually becomes infinite as $k \rightarrow \infty$. In addition, even at the limit as $k \rightarrow \infty$, the number $\xi(z, m)$ is rational if m is finite. The implications are discussed later in this chapter.

13.2.1 P-adic valuations

All rational numbers have a [period](#) – a group of digits repeating itself indefinitely – preceded by a [prefix](#). For instance, in base $b = 10$ (the ordinary decimal system), we have:

$$\frac{3011}{8325} = 0.\overline{36168168168168\dots}$$

In this example, the prefix is 36, and the period is 168. In fact, there is a simple formula to detect the length of the prefix, that is, the number of digits that it contains in a base b . Let us assume that p, q are positive integers with $p < q$. To compute the prefix $\pi(p, q)$ of p/q in base b , where b is a prime number, we proceed as follows:

- Compute the [p-adic valuations](#) $\nu_b(p), \nu_b(q)$ of p and q in base b . The p -adic valuation of p in prime base b is defined as the exponent of the highest power of b that divides p [Wiki].
- The length of the prefix $\pi(p, q)$ is given by the formula

$$L = L(\pi(p, q)) = \max \left[0, \nu_b(q) - \nu_b(p) \right] \quad (13.3)$$

It remains unchanged if you multiply p and q by an integer constant, even by a power of b . For simplicity, when there is no confusion, $L(\pi(p, q))$ is denoted as L .

- The [prefix](#) is obtained via the following [integer division](#) [Wiki]:

$$\pi(p, q) = (p \cdot b^L) // q = \frac{1}{q} \cdot \left[p \cdot b^L - (p \cdot b^L \bmod q) \right]. \quad (13.4)$$

This integer is typically represented as a string of length L in base b , with padding zeroes on the left if needed, to match the length.

When everything goes well, the iterations leading to Formula (13.1) produce integer vectors (p_k, q_k) with a prefix length that on average, increases as k increases. In particular, any k with a prefix $\pi(p_k, q_k)$ longer than the previous ones, adds new digits to $\xi(z, m)$ while preserving digits already obtained. I explain in the next section what this means. For now, it is sufficient to know that everything works nicely in the exponential case introduced earlier. Newly added digits won't be changed later as k increases.

13.2.2 Digit blocks, speed of convergence

From now on, I use $b = 2$. Also, the function $\nu_2(\cdot)$ is denoted as $\nu(\cdot)$ for convenience. When $z = 1$ and $m = \infty$ in the exponential case (13.1), each new iteration k yields $\nu(k)$ new digits. In particular, $\nu(k) = 0$ if and only if k is odd. Also p_k is always odd regardless of k , z , or m . Thus $L(\pi(p_k, q_k)) = \nu(q_k) = \nu(k!)$ in this case. These properties are peculiar to this system. Other systems discussed later also have nice properties, albeit different.

When the digits of $\xi_k(z, m)$ computed in previous iterations are preserved when overwritten as k increases, the system is said to be **digit-preserving**. A new set of digits added at a specific iteration is called a **digit block**. The goal is to find patterns (or their absence) in the non-empty blocks, and to study the asymptotic properties of the digits of $\xi_k(z, m)$ as $k \rightarrow \infty$ by focusing on the successive prefixes. The cases $m = \infty$ and $m < \infty$ are treated in the same way, even though the former results in $\xi_k(z, m)$ being rational as $k \rightarrow \infty$, while the latter leads to a known irrational number given in (13.1). To summarize, my new framework allows you to analyze the digit distribution of e by focusing on the prefix of peculiar rational numbers, some having the same proportions of zeros and ones in the binary system, and some not, with the prefix digits sequentially matching those of e when $m, k \rightarrow \infty$.

If the goal is to study the digit distribution from a theoretical point of view, speed of convergence may not be important. In applications such as pseudo-random number generators (PRNGs), speed is critical. The system presented here relies on the Taylor series of the exponential function. Binary splitting [Wiki] can accelerate convergence, along with fast multiplication [Wiki]. In general, having a numerator p_k not divisible by the base b , works best. Likewise, the faster $\nu_b(q_k)$ increases as k increases, the faster the convergence. Choosing a power of b for z , helps achieve this goal. So far, the best improvement (several orders of magnitude) was obtained using a very efficient computation of the p -adic functions. To produce random bits as fast as possible, I discuss a system with very fast convergence in section 13.2.3, though the limit number ξ is not a known constant.

Finally, if you are not interested in discovering potential patterns in the block sequence (and even if you do), you can skip the production of intermediate digits to eliminate redundancy, avoiding all the integer divisions linked to Formula (13.4), as well as the p -adic computations. That is, you can compute the (p_k, q_k) vectors iteratively and get the digits in the last iteration. In the exponential case, since block sizes are known in advance, you can slice the final digit sequence accordingly to retrieve the blocks.

In practice, for random bits generation, you use multiple sequences in parallel, for instance different values of z or a combination of sequences discussed in section 13.2.3. This further increases the speed, thanks to parallelization. Interestingly, typical congruential random number generators are similar to using $m < \infty$ in the exponential case, resulting in a finite period and less randomness. See chapter 4 in [9] for a discussion about the benefits of using irrational numbers (in this case, $m = \infty$).

13.2.3 A plethora of interesting pseudo-random sequences

The exponential case discussed at the beginning of section 13.2 is just one of many examples leading to interesting results and applications to random bits generation. There are other examples in the Python code, listed below. Here again, ξ denotes the limiting value of p_k/q_k as $k \rightarrow \infty$, while IC stands for initial conditions.

Continued fractions, with $\xi = (-1 + \sqrt{5})/2$. IC: $p_1 = 13, q_1 = 21$.

$$p_k = q_{k-1},$$

$$q_k = p_{k-1} + q_{k-1}.$$

Second order linear recursion, with $\xi = 1/(3 - \sqrt{3})$. IC: $p_0 = 0, p_1 = 1, q_0 = 0, q_1 = 2$.

$$p_k = 2p_{k-1} + 2p_{k-2} + 1,$$

$$q_k = 2q_{k-1} + 2q_{k-2}.$$

Square root, with $\xi = \sqrt{2}/4$. IC: $p_1 = 0, q_1 = 2$.

$$p_k = 2p_{k-1} + 1 \text{ if } (2p_{k-1} + 1)^2 < 2q_{k-1}^2, \text{ else } p_k = 2p_{k-1},$$

$$q_k = 2q_{k-1}.$$

Fast recursion. IC: $p_1 = 1, q_1 = 2$.

$$p_k = 2^k p_{k-1} + 3^k \bmod 2^k,$$

$$q_k = 2^k q_{k-1}.$$

Logarithm, with $\xi = \log 2$. IC: $p_0 = 1, p_1 = 5, q_0 = 2, q_1 = 8$.

$$p_k = (3k + 2)p_{k-1} - 2k^2 p_{k-2},$$

$$q_k = 2(k + 1)q_{k-1}.$$

All the above systems except the logarithm case are digit-preserving. Play with the Python code to see how block sizes evolve over the iterations, and how frequently new digits are added, in each case. The fast recursion system has this property:

$$\frac{p_k}{q_k} = \sum_{l=1}^k \frac{3^l \bmod 2^l}{2^{l(l+1)/2}} \rightarrow \xi = 0.673443360740852\dots \text{ as } k \rightarrow \infty. \quad (13.5)$$

With the exception of the first two digit blocks B_1, B_2 merged together due to initial conditions, blocks B_k ($k = 3, 4$ and so on) is k -bits long in base 2, with $B_k = 3^k \bmod 2^k$. The lower (rightmost) bit in each block is always 1. That is, $B_k \bmod 2 = 1$. For that reason, one may prefer the unbiased version, where $3^k \bmod 2^k$ is replaced by $(3^k - 1)/2 \bmod 2^k$. Finally, it is probably not difficult to prove that ξ in Formula (13.5) is irrational.

The system leading to $\sqrt{2}/4$ behaves differently. In base 2, all non-empty blocks consist of a variable number of zeros on the left, with 1 for the rightmost bit. In other words, the block sequence represents the successive runs of zeros in the binary expansion of $\sqrt{2}/4$. The non-empty blocks are 01, 01, 1, 01, 01, 000001, 001, 1, 1, 1, 001, 1, and so on. If you put them together, you get the binary expansion of $\sqrt{2}/4$.

Nice systems are the exception. Most are not digit preserving. In particular, the logarithm system defined in the above list, is not digit-preserving with the proposed initial conditions. Likewise, if you change the initial conditions to $p_1 = 1, q_1 = 2$ in the continued fraction system, p_k/q_k still converges to the same value but the digit-preserving property is lost.

The continued fraction system described here leads to the golden ratio $\xi = (-1 + \sqrt{5})/2$. It is an extreme case. Here p_k, q_k are two consecutive Fibonacci numbers. However, $\nu(q_k)$ grows extremely slowly. Even though $\nu(p_k) = 0$ when $\nu(q_k) \neq 0$ (thus boosting convergence), non-empty blocks become increasingly rare, though there are still infinitely many of them. The result: after 10^6 iterations, only 20 binary digits of ξ are computed. By contrast, the fast recursion system yields 5×10^{11} digits after 10^6 iterations.

13.3 Testing random number generators

In chapter 4 in [9], I compare traditional congruential generators with those based on irrational numbers. The goal here is different. Rather than testing the raw sequences, I focus on the digit blocks discussed earlier. These blocks may exhibit strong patterns, not compatible with randomness. This is the case for the square root system discussed in section 13.2.3. However non-randomness in the block distribution does imply non-randomness in the raw digit sequence. In the end, the binary digits of constants such as $\sqrt{2}$ better mimic randomness, compared to random bit sequences produced by Numpy with its [Mersenne Twister](#).

Indeed, the goal here is to find patterns, to better understand the digit distributions of the irrational numbers in question. The hope is that it leads to a better understanding of the underlying theoretical distributions. In turn, it could help prove that some of these digit sequences behave exactly like true random sequences. See section 13.3.1 for an example.

The methodology is as follows. I compare statistics based on the actual blocks, with those based on blocks consisting of random digits generated by Numpy. When some discrepancy is found, I use theoretical results to assess whether Python, my blocks, or both fail at mimicking randomness. Even when both pass the test, it is possible to check which one gets a better grade. Rather than sharing all the results, I present the methodology in a teaching style to help the reader learn how to use it in various contexts.

A word of caution before digging deeper into the details. The probability that a 20-bit sequence consists of zeroes only, is 0.0001%. If you look at 10^6 such sequences, the chance that at least one of them consists of 20 zeroes, is 63%. In other words, if you test a large number of independent sequences, for instance those produced by the digits of square roots of many prime numbers, you are bound to find one failing the randomness test, just by chance. Conversely, if you apply a large number of independent tests to any single sequence, chances are that it will fail some of the tests just by chance.

13.3.1 Theoretical properties of the digits of $\sqrt{2}$

In the system leading to $\sqrt{2}/4$, the blocks are anything but random: the rightmost digit is 1, preceded by a variable number of zeros. In single-digit blocks, the digit is also 1. The first block is $B_2 = 01$. It is followed by an empty block B_3 , then $B_4 = 01$. Since $q_k = 2^k$, $\nu(q_k) = k$, and $p_k < q_k$, we have $0 \leq \nu(p_k) < k$. Then, the length of the prefix satisfies $L(\pi(p_k, q_k)) = k - \nu_k(p_k) \leq k$ and block B_k is non-empty if and only if $\nu(p_k) = 0$.

Furthermore, the total number of digits of $\sqrt{2}/4$ collected over the first k blocks is equal to the sum of the lengths of these blocks. All of this allows you to make statements about the digit distribution. In particular, this distribution is fully characterized by the lengths of the successive runs of zeros (including runs of length

zero). That is, by the lengths $|B_k|$ of the non-empty blocks B_k . Each of them ends with the digit 1, terminating the current run of zeros. For non-empty blocks B_k with $k > 2$, the length (the number of digits) satisfies:

$$|B_k| = k - \operatorname{argmax}_{l < k} \left\{ \nu(p_l) = 0 \right\} = 1 + \nu(p_{k-1}) < k.$$

The argmax part represents the largest l such that $\nu(p_l) = 0$ and $l < k$. It is conjectured that $|B_k| < 1 + \log_2 k$. Also, if $\nu(p_k) \neq 0$, then $|B_k| = 0$ and $\nu(p_k) = 1 + \nu(p_{k-1})$. The number of ones in the cumulated digits collected at any given time, is equal to the number of non-empty blocks obtained.

I now check whether block lengths have a [geometric distribution](#). If they don't, it means that the randomness assumption is violated. If they do, then the expected frequency is $f_n = 2^{-n}$ for non-empty blocks of length n , with $n = 1, 2$ and so on, corresponding respectively to blocks 1, 01, 001, 0001, and so on. It would also imply that exactly 50% of the binary digits are zero, when k becomes infinite. In Figure 13.1, the Python chart bars correspond to simulated run lengths using the `random` function in Numpy with 3 different seeds and 24000 digits, while Actual correspond to the block lengths in the square root system, with the same sample size.

For lengths $n > 6$, counts are very small due to the small sample size, explaining the volatility. Thus, the randomness assumption is not violated in the [run test](#). It is worth trying with larger sample sizes to check if the good results extend beyond $n = 6$. Another test worth trying is checking the autocorrelations in subsequences of run lengths. They should all be close to zero if the data is random enough.

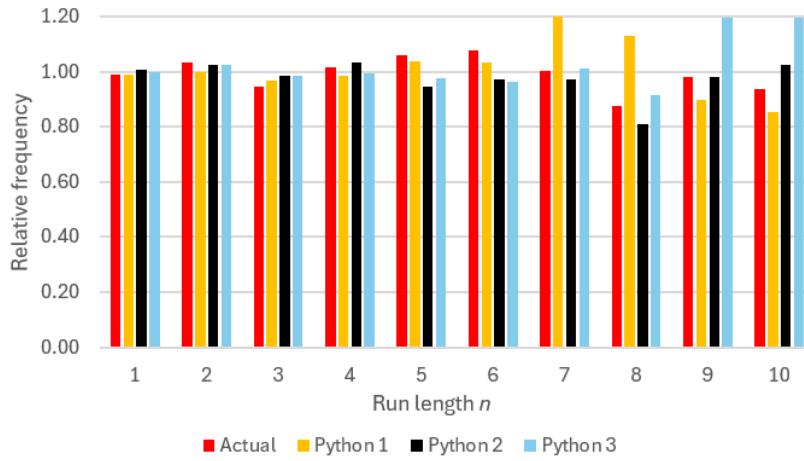


Figure 13.1: Relative frequencies of zero runs in 4 binary digits sequences

13.3.2 Fast recursion and congruent equidistribution

The square root and fast recursion systems share some properties: q_k is a power of 2, and each block B_k represents an odd integer. In the square root system, $q_k = 2^k$ and p_k is just one bit. However, in the fast recursion system, $q_k = 2^{k(k+1)/2}$ and p_k is k bits long. Thus, convergence is much faster, and you can concatenate the successive blocks to create random bit sequences growing quadratically in size as k increases. The fact that B_k is always odd, is a weakness in this case but not in the square root system. To avoid this problem, let's start by generalizing the fast recursion system, as follows:

$$p_k = 2^k p_{k-1} + A_k \bmod 2^k,$$

$$q_k = 2^k q_{k-1},$$

with the initial conditions $p_1 = A_1 \bmod 2$, and $q_1 = 2$. Here (A_k) is a sequence of integers, with A_k much larger than 2^k . In the first version, $A_k = 3^k$. Again, $B_k = A_k \bmod 2^k$, possibly with extra zeroes on the left as usual, so that B_k is k -bit long when converted to a string. If the digits are random enough, one would expect that over many blocks, $B_k \bmod m$ covers all integer values (called [residues](#)) between 0 and $m - 1$, each with about the same frequency, regardless of $m > 1$. This property is called [asymptotic congruent equidistribution](#). It is not satisfied when m is even, if $A_k = 3^k$.

One would think that choosing $A_k = (3^k - 1)/2$ would fix the issue. Indeed, it improves the situation, yet the property is not satisfied if m is a multiple of 4. What about $A_k = 3^k + k$? Now the property seems to be satisfied, but with some problem when m is a power of 2. For these m 's, the residues are evenly distributed – perfectly – which is not compatible with randomness. You need the right amount of variance: not too much, not too little. This is accomplished with $A_k = 5^k // 2^k$. Here “//” stands for the [integer division](#), also denoted as $5^k >> k$ ([bit shift](#)) since the denominator is a power of 2. It is equal to $\lfloor 5^k / 2^k \rfloor$. You cannot use $A_k = 3^k // 2^k$ because A_k must be much larger than 2^k , otherwise the property is violated for obvious reasons.

m	Exp.	Fast Recursion			Python Library		
		Model1	Model2	Model3	Seed1	Seed2	Seed3
2	5000	4998.50	0.5	92.50	1.50	102.50	63.50
3	3333	48.63	41.82	53.96	10.03	25.66	29.13
4	2500	2498.75	0.43	65.48	37.71	51.91	40.90
5	2000	37.22	54.95	40.25	59.92	38.23	30.94
6	1667	1665.84	40.27	42.13	37.75	36.85	30.98
7	1429	22.41	20.53	29.68	26.51	25.61	22.47
8	1250	2163.11	0.33	43.66	33.28	29.86	28.98
9	1111	36.10	32.00	23.59	19.64	28.39	24.03
10	1000	999.26	29.62	34.70	32.89	32.69	36.29
11	909	26.23	20.84	35.34	19.86	17.90	45.75
12	833	832.69	30.02	30.22	27.71	19.75	28.48
13	769	21.48	28.82	23.03	22.25	30.50	40.53
14	714	713.39	22.39	25.56	27.83	24.42	24.32
15	667	24.59	27.13	23.31	22.81	19.49	27.04
16	625	1080.69	0.24	27.31	26.90	18.44	19.17

Table 13.1: Stdev for congruential equidistribution test (10,000 blocks)

In the literature, congruential equidistribution is referred to as **uniform distribution** for integer sequences, a term that is misleading. Its equivalent for real numbers is called **equidistribution modulo 1**. If θ is an irrational number, then the integer sequence $A_k = \lfloor k\theta \rfloor$ satisfies the property. It also works with $A_k = \lfloor b^k \theta \rfloor$, where $b > 1$ is an integer (the **base**), assuming θ is a **normal number** in base b . Well known counter-examples: when A_k is a polynomial in k of degree 2 or higher with integer coefficients, and when $A_k = \lfloor k! e \rfloor$. Finally, the numbers $\sqrt{2}, e, \log 2$ and π are conjectured to be normal in any base. Section 13.3.1 in this chapter sets new foundations to study the normality of $\sqrt{2}$ in base $b = 2$. For more details on congruential equidistribution, see chapter 5 in [20].

String	Fast Recursion			Python Library		
	Model1	Model2	Model3	Seed1	Seed2	Seed3
0	24,986,553	24,986,515	24,993,579	24,995,827	24,998,289	24,995,167
1	25,008,446	25,008,484	25,001,420	24,999,172	24,996,710	24,999,832
00	8,327,584	8,326,983	8,330,446	8,333,231	8,335,234	8,332,216
01	12,497,442	12,497,468	12,496,467	12,495,140	12,494,518	12,498,799
10	12,497,441	12,497,469	12,496,468	12,495,141	12,494,517	12,498,799
11	8,338,874	8,338,302	8,335,781	8,334,393	8,334,637	8,333,329
000	3,566,418	3,566,701	3,570,993	3,571,832	3,572,551	3,569,057
001	6,247,215	6,245,949	6,248,383	6,249,023	6,250,880	6,248,691
010	4,996,856	4,997,481	4,996,396	4,996,259	4,995,501	4,998,112
011	6,251,062	6,249,773	6,250,134	6,250,077	6,250,783	6,249,944
100	6,247,215	6,245,949	6,248,383	6,249,023	6,250,879	6,248,691
101	5,000,084	5,000,724	4,998,010	4,996,975	4,996,353	4,999,168
110	6,251,061	6,249,774	6,250,134	6,250,078	6,250,783	6,249,945
111	3,575,452	3,575,762	3,572,484	3,573,978	3,572,508	3,572,486
χ^2	7.19	7.24	0.92	0.17	0.04	0.33

Table 13.2: String occurrences in sequences with 49,994,999 digits

Table 13.1 shows a statistical summary for each residue class modulo m , with $2 \leq m \leq 16$, for six digit sequences based on $N = 10,000$ blocks: using the random Python library with seeds 0, 1, 2 on the right, and three different versions of the fast recursion system on the left. The number of digits per sequence, called length, is $N(N-1)/2 - 1 = 49,994,999$. For each block B_k and for each modulus m , I computed the residue $B_k \bmod m$. Then I counted the occurrences of each residue over the N blocks, for each m . The expected value given m , is

N/m and featured in the second column. The six rightmost columns show the observed standard deviations for these values. In case of randomness (and thus equidistribution), the standard deviation should be small, but not too small. Also, it should decrease roughly at the same speed as $1/\sqrt{m}$ as you go down the rows in the table. The only sequence compatible with these requirements is Model3. The other ones exhibit patterns not consistent with randomness. The 3 models under “Fast Recursion” are, from left to right, $A_k = 3^k$, $A_k = 3^k + k$, and $A_k = 5^k//2^k$. See code in section 13.4.1.

Table 13.2 shows the observed frequencies for various strings, in the 6 digit sequences. The means seem correct at first glance: for instance, about 50% of zeroes and ones when you look at the first 2 rows. Model1 and Model2 exhibit more variance, while the Python library (the three rightmost columns) produce lower variance. Which ones are correct? It turns out that all but Model1 and Model2 are acceptable, with Model3 being the best and outperforming the Python library. To come to this conclusion, I computed the following statistics:

$$\chi^2(S) = \sum_{s \in S} \frac{(X_s - E[X_s])^2}{E[X_s]}, \quad (13.6)$$

where S is a set of strings for instance $S = \{00, 01, 10, 11\}$, X_s is the number of occurrences of s in the digit sequence, and $E[X_s]$ is the expected number of occurrences if the digits were random. It has a χ^2 distribution with $|S| - 1$ degrees of freedom, where $|S|$ is the number of strings in S . I computed χ^2 for $S = \{0, 1\}$. The results are displayed in the bottom row. In this case, the χ^2 expectation is 1.00. The chance that it is above 7.19 is about 0.73%, while the chance that it is below 0.04 is 15.82%. Clearly, Model3 is the best.

The computation of χ^2 for 2-bit and 3-bit strings – say, $S = \{00, 01, 10, 11\}$ – is more challenging because the strings overlap in the digit sequence. Thus, even in case of perfect randomness, these strings have uneven frequencies, and the independence requirement to apply formula (13.6) is violated. Yet, it can be addressed with more advanced probability theory.

More tests are needed to assess the quality of Model3: the run test in section 13.3.1, conditional independence (see section 13.3.3), auto-correlations and so on. The tests performed so far suggest that it outperforms Python libraries to generate random bits. Not only that, but with efficient implementation, it should also run faster. Note that if $A_k = 5^k//2^k$, then $A_k = 10^k \gg 2k$. Finally, we need to test with different values of N .

13.3.3 Exponential system: predicting the next block

Constants such as e have been thoroughly tested and passed all the randomness tests. The goal here is to focus on the blocks rather than the digits. They may exhibit patterns, at least at the beginning of the sequence. Studying them may help understand the mechanisms at play, possibly leading to theoretical results, for instance the fact that any binary string is found in the digit sequence, infinitely many times. I discuss three topics:

- Non-random behavior in one-digit blocks, with statistical analysis to back it.
- The first occurrence of any string, in the block sequence.
- Can we predict the next block given the current block?

13.3.3.1 Pattern in one-digit blocks: more ones than zeroes?

The pattern in question is not incompatible with randomness: it is balanced by more zeroes than ones in blocks consisting of multiple digits. I expect the pattern to weaken when considering a very large number of blocks. Here, I looked at the first 22,500 non-empty blocks, totaling 44,991 digits. The proportion of zeroes is 49.78%, which seems fine. However 11,250 of these blocks are one-digit, thus totaling 11,250 digits, but with a proportion of zeroes equal to only 48.07%

Now, let’s see if this could happen by chance or not, and whether there is an explanation. Using Formula (13.6) with $S = \{0, 1\}$, we have $\chi^2 = 0.88$ for the first ratio, 49.78%. This is within range for a χ^2 with 1 degree of freedom: the expected value is 1.00. But for 48.07%, we have $\chi^2 = 16.74$, and $P[\chi^2 \geq 16.74] = 0.00004$. That’s the probability that it could happen by chance, also called *p-value*.

What caused the imbalance is a sharp increase in ones at the end of the sequence in question. Big enough to cause a noticeable and long-lasting drop in the proportion of zeroes. While still within possible values in the digit sequence, it is well outside of the 99.9% range when looking at single-digit blocks. There is a lot of literature on *simple random walks* (those with equal probability of zero and one) as well as their time-continuous equivalent, *Wiener processes*. The cumulative digits of e are conjectured to follow a simple random walk. The maximum discrepancy between the cumulative sums of zeroes and ones at any iteration k is well studied, and its expectation is $\sqrt{\pi k}/2$. The *law of the iterated logarithm* offers bounds for infinite sequences. It applies to the digits, not the blocks. More on this in chapter 1, in my book [9].

13.3.3.2 Block coverage problem

In the square root system in section 13.3.1, non-empty blocks have deterministic digits: each block consists of a single 1 in the rightmost position, usually preceded by 0's on the left. But the successive block lengths appear to be unpredictable. In the exponential system, the opposite is true. Block digits are randomly distributed, but the length of block B_k is deterministic, and equal to $\nu(k)$. By definition, A block of length n represents an n -bit integer. How long does it take to cover all potential 2^n values? In other words, how many successive blocks of length n are needed until all potential combinations of n bits are covered?

Here I address this problem. Let's first look at what to expect if the digits were random. In that case, at some point we see the first block of length n . In the exponential system, it is block B_k with $k = 2^n$. What can we say about the second block of length n ? It would be block B_k with $k = 3 \cdot 2^n$ in the exponential system. What are the chances that it is different from the first one, assuming randomness?

More generally, let $T_{n,i}$ be the random variable denoting the number of blocks of length n visited until we cover i distinct ones. Also, let $\Delta_{n,i} = T_{n,i} - T_{n,i-1}$ with $i > 0$ and $T_{n,0} = 0$. We have:

$$\mathbb{E}[\Delta_{n,i}] = \sum_{j=1}^{\infty} j \cdot P(\Delta_{n,i} = j) = \sum_{j=1}^{\infty} j \cdot \left(\frac{i-1}{2^n}\right)^{j-1} \cdot \frac{2^n - (i-1)}{2^n} = \frac{2^n}{2^n - i + 1}. \quad (13.7)$$

Now, let T_n be the total number of blocks of length n visited until we cover all the 2^n distinct combinations of zeroes and ones. We have:

$$\mathbb{E}[T_n] = \sum_{i=2}^{2^n} \mathbb{E}[\Delta_{n,i}] = \sum_{i=2}^{2^n} \frac{2^n}{2^n - i + 1} = 2^n \cdot \sum_{i=1}^{2^n-1} \frac{1}{i}. \quad (13.8)$$

It follows that $\mathbb{E}[T_n]$ is asymptotically equal to $(\log 2) \cdot 2^n \cdot n$ as n tends to infinity. Now, I compare the predicted time arrivals $T_{n,1}, T_{n,2}, \dots, T_{n,2^n}$ with the observed ones.

i	$\mathbb{E}[T_{n,i}]$	$T_{n,i}^*$	$T_{n,i}$	B_{k_i}	k_i
1	1.00	1	1	0000	16
2	2.07	2	2	1011	48
3	3.21	3	3	1100	80
4	4.44	4	4	1110	112
5	5.77	6	5	1001	144
6	7.23	7	6	0110	176
7	8.83	8	9	0011	272
8	10.61	10	10	1101	304
9	12.61	12	11	0111	336
10	14.89	15	14	0001	432
11	17.56	16	16	1111	496
12	20.76	17	22	0101	688
13	24.76	20	32	0010	1008
14	30.09	30	33	1000	1040
15	38.09	39	34	0100	1072
16	54.09	45	76	1010	2416

Table 13.3: New block arrival times $T_{n,i}$ ($n = 4$)

Table 13.3 shows the 2^n arrival times $T_{n,i}$ ($i = 1, 2, \dots, 2^n$) for first occurrence of a new block of length n , here with $n = 4$. The expected value $\mathbb{E}[T_{n,i}]$ is based on Formula (13.7). For comparison purposes, I also included the values obtained by replacing the block strings by random strings using the `random` function in Numpy. These are denoted as $T_{n,i}^*$. They are based on the same block structure, and obtained by setting `random=True` in the Python code in section 13.4.2. I noticed that the last unseen block in the exponential system takes longer than expected to show up, not just for $n = 4$. This may be true only for small n . It is not incompatible with strong randomness in the full digit sequence. Note that $k_i = 2^n \cdot (2T_{n,i} - 1)$ and $T_{n,1} = 1$.

13.3.3.3 Predicting the next block

Table 13.4 shows counts for single-digit blocks 0 and 1, when the previous non-empty block is 00, 01, 10, or 11. Again, I looked at the first 22,500 non empty blocks.

Block	00	01	10	11
0	684	674	701	640
1	757	766	690	713
χ^2	3.698	5.878	0.087	3.938

Table 13.4: Counts for blocks 0 and 1 given previous block

In case of independence and uniform distribution, the eight counts should be roughly the same. But they exhibit too much variance: Formula (13.6) yields $\chi^2 = 17.534$, here with $8 - 1 = 7$ degrees of freedom. The *p-value* is $P[\chi^2 > 17.534] = 0.014$. Thus, the probability for this to happen by chance is 1.4%.

However, when treated separately in four conditional groups as in Table 13.4, the difference between the two counts in a same column is almost within normal range, except for column 01 (*p-value* = 0.015). But since we computed four χ^2 , it is not surprising that one of them is off. In any case, the probability that the current block is 1 given that the previous one is 01, is too high, at least in the first 22,500 non empty blocks. Note that for the exponential system, a block of length 2 can only be followed by a block of length 1, excluding the empty block between both. Finally, the findings in this section overlap with and corroborate those in section 13.3.3.1.

13.4 Python code

The main code is in section 13.4.2. The short code in section 13.4.1 covers the congruential equidistribution test for the fast recursion system discussed in section 13.3.2.

13.4.1 Fast recursion

This program is also on GitHub, [here](#). I used it to produce Tables 13.1 and 13.2.

```

1 import numpy as np
2 import random
3 seed = 1
4 random.seed(seed) # try 0, 1, 2
5
6 def int_to_binstring(x, n):
7     # convert integer x into n-bit string
8     str = bin(x).replace("0b", '')
9     while len(str) < n:
10         str = '0' + str
11     return(str)
12
13 def hash_update(key, hash):
14     if key in hash:
15         hash[key] += 1
16     else:
17         hash[key] = 1
18     return(hash)
19
20 def summary(hash, m, N):
21     # summary stats for congruences modulo m
22     mean = 0
23     std = 0
24     for j in range(m):
25         # loop on residue classes modulo m
26         if (m, j) in hash:
27             count = hash[(m, j)]
28         else:
29             count = 0
30         mean += count
31         std += count*count
32     total = mean
33     mean /= m
34     std = np.sqrt(std/m - mean*mean)
35     # normalize
36     # mean /= N-M-1
37     # std /= np.sqrt(N-M-1)
38     return(mean, std)
39
40 M = 20 # test moduli up to M
41 N = 10000 # last block visited
42

```

```

43 hres1 = {} # residues modulo m: counts based on model
44 hres2 = {} # residues modulo m: counts based on random numbers
45 fstring = "" # digit sequence based on model
46 rstring = "" # digit sequence based on random numbers
47
48 for k in range(2, N):
49
50     # fastint = (3**k) % (2**k)
51     # fastint = ((3**k + k)) % (2**k)
52     fastint = ((5**k) >> k) % (2**k)
53     fstring = fstring + int_to_binstring(fastint, k)
54     randint = random.getrandbits(k)
55     rstring = rstring + int_to_binstring(randint, k)
56
57     for m in range(2,M):
58
59         res1 = fastint % m
60         res2 = randint % m
61         key1 = (m, res1)
62         key2 = (m, res2)
63         if k > m:
64             hres1 = hash_update(key1, hres1)
65             hres2 = hash_update(key2, hres2)
66
67     for m in range(2, M):
68         for res in range(0,m):
69             key = (m, res)
70             if key not in hres1:
71                 hres1[key] = 0
72                 print(">>> Missing:", key)
73             if key not in hres2:
74                 hres2[key] = 0
75                 print(key, hres1[key], hres2[key])
76
77 print("\nresidue classes: summary")
78 for m in range(2, M):
79     (fmean, fstd) = summary(hres1, m, N)
80     (rmean, rstd) = summary(hres2, m, N)
81     # fmean = rmean, thus nor showing rmean
82     print("%4d %8.2f %8.2f %9.2f" % (m, fmean, fstd, rstd))
83
84 print("\nSubstring counts in digit sequence")
85 substrings = ('0','1','00','01','10','11',
86               '000','001','010','011',
87               '100','101','110','111',)
88 for str in substrings:
89     print("%4s %10d %10d" % (str, fstring.count(str), rstring.count(str)))
90
91 print(fstring[0:1000])

```

13.4.2 Main code

The code is also on GitHub, [here](#). It computes p_k, q_k , the prefix $\pi(p_k, q_k)$ and the blocks B_k for the various systems described in section 13.2. In the code, B_k is named `new_digits` and it is stored as a binary string, see lines 177–170. Since full digit sequences of increasing lengths are computed at each iteration k in order to find more and more digits, you should not use this algorithm if your goal is to simply get a large number of digits, for the mathematical constants in question. In that case, you don't need to identify the various blocks, and you can remove all intermediary block and prefix computations except in the last iteration. This will dramatically increase the speed.

```

1 import gmpy2
2 import numpy as np
3 import matplotlib.pyplot as plt
4 import matplotlib as mpl
5 from matplotlib import pyplot
6
7 def update_hash(hash, key, count):
8     if key in hash:
9         hash[key] += count
10    else:
11        hash[key] = count
12    return (hash)

```

```

13
14 def p_adic(k, base):
15     # return the largest integer v such that base**v divides k
16     # https://www.geeksforgeeks.org/python-program-to-find-whether-a-no-is-power-of-two/
17     if base == 2:
18         div = k & (~ (k - 1))
19         v = len(bin(div)) - 3
20     else:
21         div = 1
22         v = 0
23         while k % div == 0:
24             div *= base
25             v += 1
26         div = div // base
27         v -= 1
28     return(v, div)
29
30 def initialize(mode, z=1, m=0):
31
32     p_buffer = []
33     q_buffer = []
34
35     if 'Exponential' in mode:
36         # p/q tends to exp(x)-1-1/x as k -> infinity
37         p_buffer.append(0)
38         p_buffer.append(0)
39         q_buffer.append(0)
40         q_buffer.append(z)
41
42     elif mode == 'Linear':
43         # p/q tends to 1/(3-SQRT(3)) as k --> infinity
44         p_buffer.append(0)
45         p_buffer.append(1)
46         q_buffer.append(0)
47         q_buffer.append(2)
48
49     elif mode == 'ContinuedFractions':
50         # p/q tends to (-1+sqrt(5))/2 as k -> infinity
51         p_buffer.append(0)
52         p_buffer.append(13)
53         q_buffer.append(0)
54         q_buffer.append(21)
55
56     elif mode == 'Special':
57         p_buffer.append(0)
58         p_buffer.append(0)
59         q_buffer.append(1)
60         q_buffer.append(2)
61
62     elif mode == 'Special2':
63         p_buffer.append(0)
64         p_buffer.append(0)
65         q_buffer.append(1)
66         q_buffer.append(2)
67
68     elif mode == 'Special3':
69         p_buffer.append(0)
70         p_buffer.append(1)
71         q_buffer.append(0)
72         q_buffer.append(2)
73
74     return(p_buffer, q_buffer)
75
76
77 #--- [1] main loop
78
79 # parameters
80 mode = 'Exponential' # options: 'Exponential', 'Exponential1', 'Exponential2',
81                      # 'Linear', 'ContinuedFractions', 'Special', 'Special2', 'Special3'
82 random = False
83 base = 2 # base must be a prime number
84 n = 45000 # number of iterations
85 m = 0    # used in Exponential mode (if m=0, p/q tends to exp(z)-1-1/z)
86 z = 1    # used in Exponential mode (integer > 0)
87 seed = 659 # used if random = True
88 np.random.seed(seed)

```

```

89
90 # local variables
91 digits = ""
92 new_digits = ""
93 hash1 = {}
94 hash2 = {}
95 hash_nu = {}
96 bprefix = ""
97 lmax = 0
98 k_old = 0
99 zeros = 0
100 ones = 0
101
102 (p_buffer, q_buffer) = initialize(mode, z, m)
103
104 for k in range(2, n+1, 1):
105
106     # p = p[k], p_buffer[1] = p[k-1], p_buffer[2] = p[k-2]
107     # q = q[k], q_buffer[1] = q[k-1], q_buffer[2] = q[k-2]
108
109     if mode == 'Exponential':
110         p = z*k*p_buffer[1] + 1
111         q = z*k*q_buffer[1]
112
113     elif mode == 'Exponential1' and m > 0:
114         p = z*min(k, m)*p_buffer[1] + 1
115         q = z*min(k, m)*q_buffer[1]
116
117     elif mode == 'Exponential2' and m > 0:
118         p = z*((k-1)%m + 1) * p_buffer[1] + 1
119         q = z*((k-1)%m + 1) * q_buffer[1]
120
121     elif mode == 'Linear':
122         p = 2*p_buffer[1] + 2*p_buffer[0] + 1
123         q = 2*q_buffer[1] + 2*q_buffer[0]
124
125     elif mode == 'ContinuedFractions':
126         p = q_buffer[1]
127         q = p_buffer[1] + q_buffer[1]
128
129     elif mode == 'Special':
130         if (2*p_buffer[1]+ 1)**2 < 2 * q_buffer[0]**2:
131             p = 2*p_buffer[1] + 1
132         else:
133             p = 2*p_buffer[1]
134             q = 2*q_buffer[1]
135
136     elif mode == 'Special2':
137         p = 2**k * p_buffer[1] + ((3**k - 1)//2) % (2**k)
138         q = 2**k * q_buffer[1]
139
140     elif mode == 'Special3':
141         p = 2**k * p_buffer[1] + (3**k) % (2**k)
142         q = 2**k * q_buffer[1]
143
144     p_buffer[0] = p_buffer[1]
145     p_buffer[1] = p
146     q_buffer[0] = q_buffer[1]
147     q_buffer[1] = q
148     nu_k, div_k = p_adic(k, base)
149     nu_p, div_p = p_adic(p, base)
150     nu_q, div_q = p_adic(q, base)
151     if p > q:
152         print("Warning: p >= q (unauthorized)")
153         exit()
154     l = max(0, nu_q - nu_p) # length of prefix
155
156     if l > lmax:
157
158         # process additional digits found
159         lmax = l
160         prefix = (base**l) * p // q
161         bprefix_old = bprefix
162         l_old = len(bprefix_old)
163         bprefix = gmpy2.mpz(prefix).digits(base)
164         while len(bprefix) < l:

```

```

165     bprefix = '0' + bprefix
166     if bprefix[0:l_old] != bprefix_old:
167         match = 'Fail'
168     else:
169         match = 'Success'
170     new_digits_old = new_digits
171
172     if random:
173         new_int = np.random.randint(base**l - l_old)
174     else:
175         new_int = prefix % (base**l - l_old)
176
177     new_digits = gmpy2.mpz(new_int).digits(base)
178     while len(new_digits) < l - l_old:
179         new_digits = '0' + new_digits
180
181     delta = k - k_old
182     zeros += new_digits.count('0')
183     ones += new_digits.count('1')
184     size = len(new_digits)
185     print("==>", k, l, size, delta, nu_k, nu_p, nu_q, match, zeros+ones,
186           zeros, ones, ">>", new_digits)
187
188     digits += new_digits
189     k_old = k
190     key = (size, k)
191     if new_digits not in hash1:
192         hash_nu[key] = new_digits
193         update_hash(hash1, new_digits, 1)
194     key = (new_digits, new_digits_old)
195     update_hash(hash2, key, 1)
196
197
198 #--- [2] Output results
199
200 #- [2.1] block counts
201
202 patterns = () # list of all potential combos of t binary digits
203 t = 5
204 for k in range(0, 2**t, 1):
205     bint = bin(k)
206     bint = bint[2:len(bint)]
207     while len(bint) < t:
208         bint = "0"+bint
209     patterns = (*patterns, bint)
210
211 print("\nblock counts\n")
212 hash1_print = {}
213
214 for key in hash1:
215     klen = len(key)
216     hash1_print[(klen, key)] = hash1[key]
217
218 for keyx in sorted(hash1_print):
219     key = keyx[1]
220     klen = len(key)
221     count = hash1[key]
222     if count > 1:
223         print(klen, key, count)
224
225 #- [2.2] first occurrence of block
226
227 print()
228 print("first occurrence of block")
229 old_size = 0
230 count = 0
231 for key in sorted(hash_nu):
232     size = key[0]
233     if size == old_size:
234         count = count+1
235     else:
236         print()
237         count = 1
238         old_size = size
239     print(count, hash_nu[key], key)
240

```

```

241 #- [2.3] conditional block counts
242
243 print("\nconditional block counts\n")
244 hash2_print = {}
245
246 for key in hash2:
247     klenA = len(key[0])
248     klenB = len(key[1])
249     hash2_print[(klenA, klenB, key)] = hash2[key]
250
251 for keyx in sorted(hash2_print):
252     key = keyx[2]
253     old = key[0]
254     new = key[1]
255     count = hash2[key]
256     if count > 5:
257         print("(old, new):",key, hash2[key])
258
259 #- [2.4] high level summary
260
261 print()
262
263 print("Zeros vs ones:", zeros, ones)
264 print("Proportion of zeros:", zeros/(zeros + ones))
265
266 sum1 = 0
267 ndigits1 = min(80,len(digits))
268
269 for k in range(0,ndigits1,1):
270     sum1 += int(digits[k])/base** (k+1)
271
272 sum2 = 0
273 ndigits2 = min(80,len(bprefix))
274
275 for k in range(0,ndigits2,1):
276     sum2 += int(bprefix[k])/base** (k+1)
277
278 print("dCheck:", sum1)
279 print("bCheck:", sum2)
280 print("Number:", p/q)
281
282 if 'Exponential' in mode:
283     print("Target:", np.exp(1/z)-(1+1/z))
284 elif mode == 'Linear':
285     print("Target:", 1/(3-np.sqrt(3)))
286 elif mode == 'ContinuedFractions':
287     print("Target:", (-1+np.sqrt(5))/2)
288 elif mode == 'Special':
289     print("Target:", np.sqrt(2)/4)
290
291 print("Digits per n:", (zeros+ones)/n)

```

Chapter 14

Sampling Outside the Observation Range

All of the GenAI apps that I tested, including my own, have the same problem. They cannot easily generate data outside the observation range. As an example, let's focus on the insurance dataset posted on GitHub, [here](#). I use it to generate synthetic data with **GAN** (generative adversarial networks) and the NoGAN models discussed in chapters [7](#) and [8](#). In the training set, one of the features is "charges", that is, the medical expenses incurred by the policy holder, in a given year. The range is from \$1121 to \$63,770. In the synthesized data, the amount always stays within these two bounds. Worst, most models are unable to produce a synthetic maximum above \$60,000. The issue is undetected due to poor evaluation metrics, and compounded by the small size of the training set. The same is true for all the other features. The problem shows up in all the tested datasets, no matter how many observations you generate.

The consequences are persistent algorithm bias, and the inability to generate enriched or unusual data. The solution currently adopted is to work with gigantic training sets, further increasing costs linked to training, cloud and GPU time usage. What I propose here goes in the opposite direction: cost reduction, smaller training sets, high quality output based on the best evaluation metrics (see chapter [7](#)), and the ability to generate more diversified data, including meaningful outliers. All this with a fast, simple algorithm based on a clever idea.

14.1 Quantile convolution

I now discuss the concept in layman's terms, and then briefly explain the underlying theory. By comparison to **diffusion models** [\[Wiki\]](#) used in computer vision to address the issue, the technique is a lot simpler. Here I focus on the one-dimensional case here, but it generalizes to higher dimensions.

The training set consists of n observations x_1, \dots, x_n . I then create a **Gaussian mixture model** (GMM) with n components, all having the same weight $1/n$ and same variance σ_n^2 . The k -th component is a Gaussian centered at x_k , with variance σ_n^2 . I then sample N deviates from the GMM to compute its **quantile function** (the inverse of the CDF), typically with N much larger than n . At this point, we have three distributions:

- The empirical distribution H_n attached to the Gaussian mixture.
- The empirical distribution F_n attached to the training set observations.
- A generic normal distribution G_n , called **kernel**, with zero mean and variance σ_n^2 .

The setting is identical to **kernel!density estimation** [\[Wiki\]](#). The derivative of H_n plays the role of the smooth density function estimate. I denote the corresponding **probability density functions** (PDF) as h_n, f_n and g_n . We have:

$$H_n(z) = \frac{1}{n} \sum_{k=1}^n G_n(z - x_k) I(x = x_k), \quad (14.1)$$

where I is the indicator function, equal to 1 if $x = x_k$, and 0 otherwise. When n is large and the discrete PDF f_n is well approximated by a continuous density f , we have $H_n(z) \sim \int G_n(z - x) f(x) dx$. Thus, taking the derivative with respect to z , we obtain:

$$h_n(z) \sim \int g_n(z - x) f(x) dx = (g_n * f)(z). \quad (14.2)$$

The $*$ symbol denotes the **convolution product**, in this case the convolution of probability distributions [\[Wiki\]](#). If $\sigma_n \rightarrow 0$ as $n \rightarrow \infty$, then $h_n \rightarrow f$. Also, if $\sigma_n = 0$, then $H_n = F_n$ corresponds to the empirical distribution

(ECDF) computed on the training set. The ECDF F_n is known to converge to the true continuous underlying CDF F . This is another way to look at the asymptotic behavior: $\sigma_n \rightarrow 0 \Rightarrow H_n \sim F_n \rightarrow F$.

So, by choosing $\sigma_n > 0$ yet small enough, especially if n is large, we achieve the following comprise: h_n is some intermediate PDF between the discrete, chaotic f_n and the smooth, continuous theoretical but unknown limit, f . In practice, for σ_n , you can choose the standard deviation computed on the training set, multiplied by a small positive factor denoted as v_n .

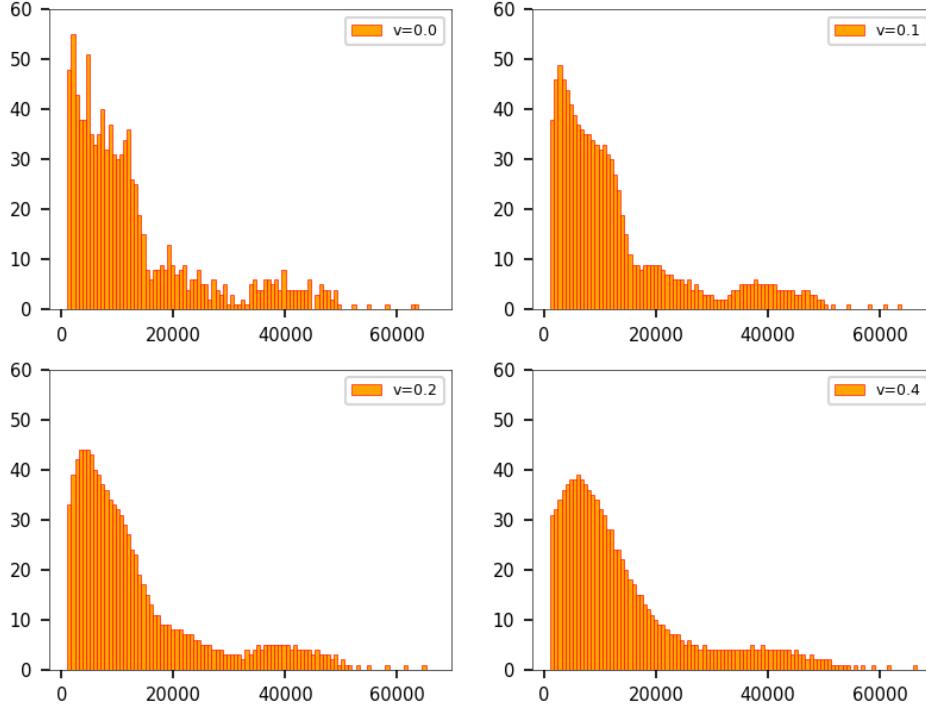


Figure 14.1: Histograms for extrapolated “charges” ($v = 0$ is the training set)

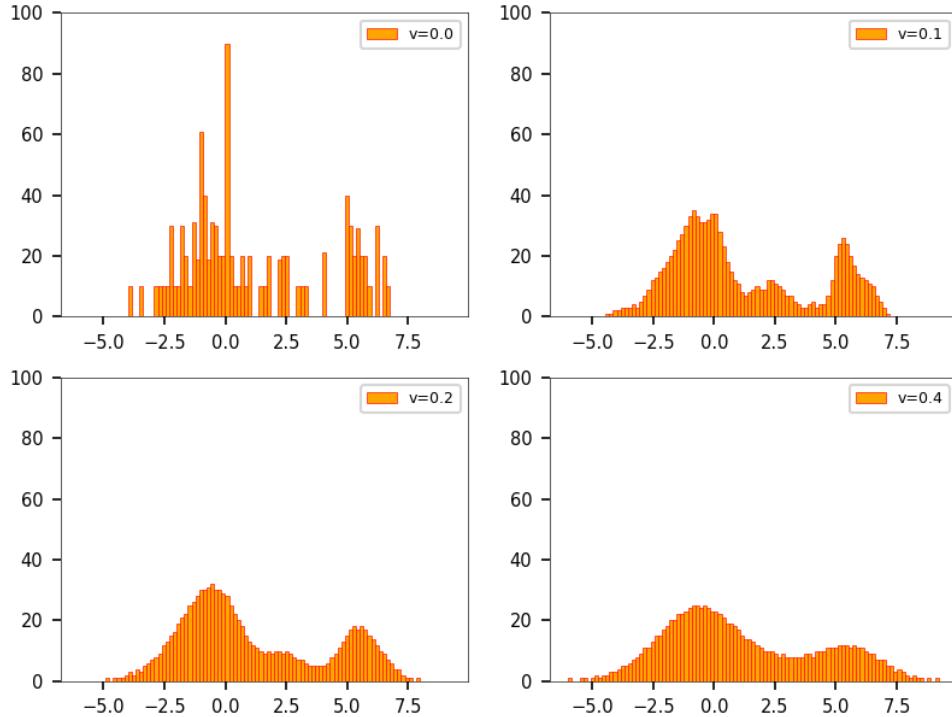


Figure 14.2: Histograms for extrapolated mixture ($v = 0$ is the training set)

With this framework, it is fast and easy to sample (say) $N = 10^6$ deviates from the H_n distribution and

sort them to compute (say) 10^3 quantiles, from 0.0005 to 0.9995 by increments of 10^{-3} , and store them in an array. More granular quantiles are obtained by interpolating the pre-computed values. If σ_n is not too small, it will generate values outside the observation range in the training set. Unlike the standard method described [here](#) and [here](#) to sample from a mixture, it does not involve inverting the CDF H_n , resulting in a much faster implementation. The NoGAN techniques to generate [synthetic data](#) (see chapters 7 and 8) heavily rely on quantiles in high dimensions, and the convoluted quantiles discussed here can easily be integrated into these algorithms.

In the end, my technique is a fully automated, data-driven version of [quantile extrapolation](#). For more on this topic, see “Nonparametric Extrapolation of Extreme Quantiles: a Comparison Study”, published in 2022 [1]. Besides smoothing empirical quantiles and outside-the-range synthetizations, another application is the generation of meaningful outliers: those not resulting from some error, but naturally occurring, albeit rarely, in the real world. This could be useful in contexts such as fraud detection.

14.2 Truncated Gaussian mixtures and bias detection

One may use kernels other than Gaussian. Discrete kernels are the best solution when dealing with a categorical or discrete feature, such as “number of children” in the insurance dataset. In this section, I discuss a different type of kernel: [truncated Gaussian](#). You use it if your data is constrained to stay within a specific domain D . For instance, in the insurance dataset, “charges” must be positive, and possibly above \$1000 due to business rules. They may be capped at \$100,000. For business reasons, “age” must be between 18 and 64 inclusive.

In the case studies in section 14.3, I use [rejection sampling](#) [Wiki] to meet these needs. The principle is simple: if a generated deviate is outside the domain D , reject it and continue sampling until you get one that lies inside D . This may result in noticeable bias in the synthetic data if σ_n is not small enough. However, one might say that the bias is in the real data, not in the synthetic data, especially if n is small. The synthetization may be a better representation of the reality. This is true especially when the truncation is one-sided, with a hard minimum (values must be above zero) but no hard maximum.

To assess whether the bias is in the real or in the synthetic data, proceed as follows. Create a training set using Monte-carlo simulations and known distribution, with $2n$ values. Then:

- Use n values (half of the training set) to compute the quantile table Q_n based on H_n .
- Compute the mean both on the half training set S_n , and on the quantile table Q_n . They are denoted respectively as $\mu(S_n)$ and $\mu(Q_n)$.
- Compute the mean $\mu(S_{2n})$ on the full training set.

If $|\mu(Q_n) - \mu(S_{2n})| < |\mu(S_n) - \mu(S_{2n})|$, then the bias is likely more pronounced in the real data (the half training set), rather than in the synthetic data! Alternatively, you can transform the generated quantiles so that the mean and variance match those measured on the training set. This makes sense if you use an unusually large σ_n with one-sided truncation, resulting in generated values far beyond the maximum or minimum observed in the training set.

14.3 Case studies

Figures 14.1, 14.2 and 14.3 show histograms associated to H_n , each with 50 bins, with 4 plots in each picture. For σ_n , I chose the standard deviation computed on the training set, multiplied by a small factor v , ranging from $v = 0.0$ (top left plot) to $v = 0.4$ (bottom right). So, the top left plot corresponds to $\sigma_n = 0$. It represents the frequency distribution in the training set. The Y-axis features bin counts, totaling $n = 1000$ across all 50 bins in each plot. The X-axis represents the observed values: the extended range, after extrapolation based on quantile convolution.

Figures 14.1 and 14.3 correspond to two of the features in the insurance dataset: “charges” (in dollar amount), and “bmi” (body mass index). I used a truncated Gaussian for the kernel. Figure 14.2 pictures an artificial dataset: the data was created using a mixture with three components, with $n = 100$ observations. Quantile convolution ($v > 0$) with a Gaussian kernel clearly generates values outside the observation range. The Python code is in section 14.4.

In my examples, $v = 0.1$ seems to be the best value, preserving the patterns in the distribution attached to the training set, while generating extreme values that are not too far from the minimum and maximum in the real data. Table 14.1 summarizes the findings. In particular, the “simulated” feature was created as a mixture with 3 components, also called clusters. With $v = 0.1$ or $v = 0.2$, the 3 components are still visible: see Figure 14.2. The technique could also be used to detect the optimum number of clusters, and generalizes to

higher dimensions. Note that with $v = 0.4$, generated values extend far beyond the observation range, allowing you to create meaningful outliers. In Table 14.1, $P_{.0005}$ and $P_{.9995}$ are extreme quantiles (convoluted if $v > 0$).

Feature	v	$P_{.0005}$	$P_{.9995}$	Median	Stdev
charges	0.0	1122	63,770	9374	12,103
charges	0.1	1076	63,709	9393	12,079
charges	0.2	1076	64,538	9485	12,090
charges	0.4	1077	66,520	10,392	12,113
bmi	0.0	15.96	53.13	30.40	6.09
bmi	0.1	15.89	53.10	30.39	6.13
bmi	0.2	15.09	53.44	30.42	6.21
bmi	0.4	12.77	54.31	30.42	6.57
simulated	0.0	-3.87	6.66	0.05	2.87
simulated	0.1	-4.35	7.20	0.13	2.89
simulated	0.2	-4.87	7.85	0.21	2.93
simulated	0.4	-5.99	9.21	0.43	3.09

Table 14.1: Extreme values as a function of v (training set: $v = 0$)

Finally, it would be interesting to see what happens when you iterate the method: starting with $H_{0,n} = F_n$ to produce $H_{1,n} = H_n$, then using $H_{1,n}$ to produce $H_{2,n}$ and so on. In short, synthesizing the synthetic data and so on. Most data synthesizers are unable to sample outside the observation range, resulting in successive iterations generating data within a shrinking range. Conversely, with a large σ_n , my method will generate data in an expanding range, over several iterations. The best solution is to choose σ_n that keeps the range stable over many iterations.

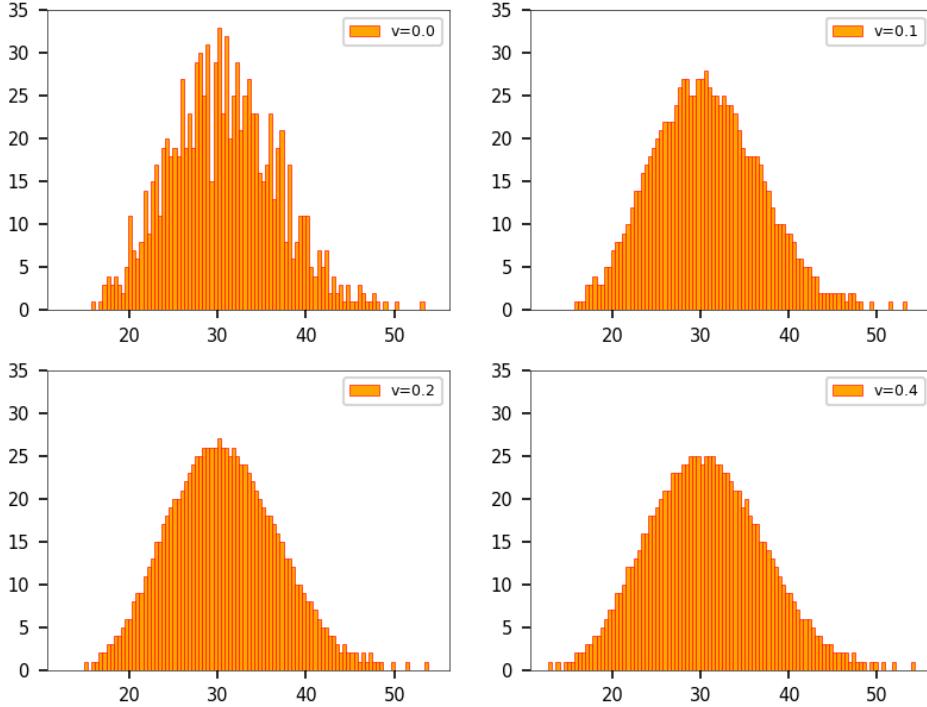


Figure 14.3: Histograms for extrapolated “bmi” ($v = 0$ is the training set)

14.3.1 Conclusion

The quantile convolution technique helps you generate data outside the observation range, thus creating truly enriched datasets, contrarily to all the tools that I tried in the context of synthetic data, whether based on deep neural networks or not, whether open-source or vendor platforms. Generalizing quantiles to higher dimensions may not seem trivial, but it has been done with NoGAN and sister methods discussed in chapters 7 and 8. The new method, akin to quantile extrapolation, blends easily with NoGAN to enhance its performance.

Current techniques to evaluate the quality of synthetic fail to capture complex feature dependencies, resulting in false negatives: generated data scored as excellent, when it is actually very poor. Deep neural networks can be very slow and volatile, requiring ad-hoc tuning for each new dataset. In this book, the focus was on new algorithms – not necessarily neural networks – that are fast and easy to train, lead to explainable AI and auto-tuning, and require less rather than more data to address the traditional challenges. For instance, in section 8.2 in [14], I illustrate how you can get better results, in addition to saving time, by randomly deleting 50% of the data in the training set. All of this using sound evaluation metrics and cross-validation.

The main goal of all this framework is cost savings while delivering better results: using less training, GPU and cloud time. It goes against the modern trend of using bigger and bigger datasets. The popularity of oversized datasets stems from the fact that it seems to be the easy solution. Yet my algorithms are simpler. Then, large companies offering cloud and GPU services have strong incentives to favor big data: the bigger, the more revenue for them, the higher the costs for the client. Since I offer free solutions, thus bearing the cost of computations, I have strong incentives to optimize for speed while maintaining high quality output. In the end, my goals are thus aligned with those of the client, not with those of cloud companies or vendor charging a premium for cloud usage, based on the volume of data.

14.4 Python code

The function `get_test_data` creates the simulated training set used in Figure 14.2, referenced as “simulated” in Table 14.1. The insurance data set is accessed from GitHub via the URL in the code, in the `get_real_data` function. Truncation is determined by the parameters `minz` and `maxz`, with no truncation if `minz>maxz`. I do not list v as a variable, but it is implicitly used in instructions such as `sigma3=0.2*np.std(data)`, where $v = 0.2$. Convolved quantiles are stored in arrays, e.g. `e1quant1`, while standard quantiles are in `pquant`. The code is also on GitHub, [here](#).

```

1 # equantile.py: extrapolated quantiles
2
3 import numpy as np
4 import matplotlib.pyplot as plt
5 import matplotlib as mpl
6 import pandas as pd
7
8 seed = 76
9 np.random.seed(seed)
10
11 def get_test_data(n=100):
12     data = []
13     for k in range(n):
14         u = np.random.uniform(0, 1)
15         if u < 0.2:
16             x = np.random.normal(-1, 1)
17         elif u < 0.7:
18             x = np.random.normal(0, 2)
19         else:
20             x = np.random.normal(5.5, 0.8)
21         data.append(x)
22     data = np.array(data)
23     return(data)
24
25 def get_real_data():
26     url = "https://raw.githubusercontent.com/VincentGranville/Main/main/insurance.csv"
27     data = pd.read_csv(url)
28     # features = ['age', 'sex', 'bmi', 'children', 'smoker', 'region', 'charges']
29     data = data[['bmi']] # choose 'bmi' or 'charges'
30     data = np.array(data)
31     return(data)
32
33 #--
34
35 def truncated_norm(mu, sigma, minz, maxz):
36     z = np.random.normal(mu, sigma)
37     if minz < maxz:
38         while z < minz or z > maxz:
39             z = np.random.normal(mu, sigma)
40     return(z)
41
42 #- sample from mixture
43
44 def mixture_deviate(N, data, f, sigma, minz, maxz, verbose=False):

```

```

45     sample = []
46     point_idx = np.random.randint(0, len(data), N)
47     mu = data[point_idx]
48     for k in range(N):
49         z = truncated_norm(mu[k], sigma, minz, maxz)
50         sample.append(z)
51         if verbose and k%10 == 0:
52             print("sampling %6d / %6d" %(k, N))
53     sample = np.array(sample)
54     sample = np.sort(sample)
55     return (sample)
56
57 #---- Main part
58
59 # data = get_test_data(100)
60 data = get_real_data()
61 N = 1000000
62 truncate = False
63
64 # minz > maxz is the same as (minz = -infinity, maxz = +infinity)
65 if truncate == True:
66     minz = 0.50 * np.min(data) # use 0.95 for 'charges', 0.50 for 'bmi'
67     maxz = 1.50 * np.max(data) # use 1.50 for 'charges', 1.50 for 'bmi'
68 else:
69     minz = 1.00
70     maxz = 0.00
71
72 sigma1 = 0.0 * np.std(data)
73 sample1 = mixture_deviate(N, data, truncated_norm, sigma1, minz, maxz)
74
75 sigma2 = 0.1 * np.std(data)
76 sample2 = mixture_deviate(N, data, truncated_norm, sigma2, minz, maxz)
77
78 sigma3 = 0.2 * np.std(data)
79 sample3 = mixture_deviate(N, data, truncated_norm, sigma3, minz, maxz)
80
81 sigma4 = 0.4 * np.std(data)
82 sample4 = mixture_deviate(N, data, truncated_norm, sigma4, minz, maxz)
83
84 arrq = []
85 equant1 = []
86 equant2 = []
87 equant3 = []
88 equant4 = []
89 pquant = []
90
91 pbins = 1000
92 step = N / pbins # N must be a multiple of pbins
93 for k in range(pbins):
94     p = (k + 0.5) / pbins
95     arrq.append(p)
96     eq_index = int(step * (k + 0.5))
97     equant1.append(sample1[eq_index])
98     equant2.append(sample2[eq_index])
99     equant3.append(sample3[eq_index])
100    equant4.append(sample4[eq_index])
101    pquant.append(np.quantile(data, p))
102
103 mpl.rcParams['axes.linewidth'] = 0.3
104 plt.rcParams['xtick.labelsize'] = 7
105 plt.rcParams['ytick.labelsize'] = 7
106
107 #---- Plot results
108
109 bins=np.linspace(np.min(equant4), np.max(equant4), num=100)
110
111 plt.subplot(2,2,1)
112 plt.hist(equant1,color='orange',edgecolor='red',bins=bins,linewidth=0.3,label='v=0.0')
113 plt.legend(loc='upper right', prop={'size': 6}, )
114 plt.ylim(0,35)
115 plt.subplot(2,2,2)
116 plt.hist(equant2,color='orange',edgecolor='red',bins=bins,linewidth=0.3,label='v=0.1')
117 plt.legend(loc='upper right', prop={'size': 6}, )
118 plt.ylim(0,35)
119 plt.subplot(2,2,3)
120 plt.hist(equant3,color='orange',edgecolor='red',bins=bins,linewidth=0.3,label='v=0.2')

```

```
121 plt.legend(loc='upper right', prop={'size': 6}, )
122 plt.ylim(0,35)
123 plt.subplot(2,2,4)
124 plt.hist(equant4,color='orange',edgecolor='red',bins=bins,linewidth=0.3,label='v=0.4')
125 plt.legend(loc='upper right', prop={'size': 6}, )
126 plt.ylim(0,35)
127 plt.show()
128
129 #--- Output some summary stats
130
131 print()
132 print("Observation range, min: %8.2f" %(np.min(data)))
133 print("Observation range, max: %8.2f" %(np.max(data)))
134 pmin = np.quantile(data, 0.5/pbins)
135 pmax = np.quantile(data, 1 - 0.5/pbins)
136 print("Python quantile %6.4f: %8.2f" %(0.5/pbins, pmin))
137 print("Python quantile %6.4f: %8.2f" %(1-0.5/pbins, pmax))
138 print("Python quantile %6.4f: %8.2f" %(0.5, np.quantile(data,0.5)))
139 print("Dataset stdev : %8.2f" %(np.std(data)))
140
141 print()
142 print("sigmal: %6.2f" %(sigmal))
143 print("Equant quantile %6.4f: %8.2f" %(0.5/pbins, equant1[0]))
144 print("Equant quantile %6.4f: %8.2f" %(1-0.5/pbins, equant1[999]))
145 print("Equant quantile %6.4f: %8.2f" %(0.5, np.median(equant1)))
146 print("Equant-based stdev : %8.2f" %(np.std(equant1)))
147
148 print()
149 print("sigma2: %6.2f" %(sigma2))
150 print("Equant quantile %6.4f: %8.2f" %(0.5/pbins, equant2[0]))
151 print("Equant quantile %6.4f: %8.2f" %(1-0.5/pbins, equant2[999]))
152 print("Equant quantile %6.4f: %8.2f" %(0.5, np.median(equant2)))
153 print("Equant-based stdev : %8.2f" %(np.std(equant2)))
154
155 print()
156 print("sigma3: %6.2f" %(sigma3))
157 print("Equant quantile %6.4f: %8.2f" %(0.5/pbins, equant3[0]))
158 print("Equant quantile %6.4f: %8.2f" %(1-0.5/pbins, equant3[999]))
159 print("Equant quantile %6.4f: %8.2f" %(0.5, np.median(equant3)))
160 print("Equant-based stdev : %8.2f" %(np.std(equant3)))
161
162 print()
163 print("sigma4: %6.2f" %(sigma4))
164 print("Equant quantile %6.4f: %8.2f" %(0.5/pbins, equant4[0]))
165 print("Equant quantile %6.4f: %8.2f" %(1-0.5/pbins, equant4[999]))
166 print("Equant quantile %6.4f: %8.2f" %(0.5, np.median(equant4)))
167 print("Equant-based stdev : %8.2f" %(np.std(equant4)))
```

Chapter 15

Additional Resources

In this chapter, I discuss various techniques to further enhance traditional AI and machine learning systems. The goal is to get a lot more from traditional tools, with much less efforts. The focus is on simplicity and abstraction, to solve many problems in many different domains, using a single but generic tool. It requires rethinking classic AI and ML algorithms, to come up with radically different approaches. The techniques described here represent a small selection from my arsenal. I provide a general overview for each of them, with references for detailed description, case studies, datasets and code.

15.1 Accelerating convergence of parameter estimates

In most cases, when you have a dataset with n observations and you try to fit a parametric model – even a deep neural network – the precision on the parameter estimates is of the order $1/\sqrt{n}$. This is a consequence of the [central limit theorem](#): statistical and approximation errors from various sources lead to this result. This law of the inverse square root is written in stone and seems unavoidable: you cannot do better. For instance, the width of a [confidence interval](#) (CI) is asymptotically equal to α/n^β as n increases, with $\beta = \frac{1}{2}$, and α depending on the problem and converging to a finite strictly positive constant as n increases. Here I explain how you can actually do better without increasing the sample size n , leading to $\beta > \frac{1}{2}$.

15.1.1 First case study

The first example is discussed in section 10.3.2 in [14]. Here I only reports the results. In Table 15.1, the “standard width” corresponds to classical methods and $\beta = \frac{1}{2}$. The “new width” is obtained with the enhanced technology discussed in the reference in question, yielding $\beta \approx 0.92$. It comes with a bigger α , thus until $n = 780$, the standard method performs better. But as n increases, the enhanced method wins. The bigger n , the bigger the improvement, with significantly shorter width compared to the standard method.

n	New width	Standard width
100	0.4742	0.2000
500	0.1078	0.0894
780	0.0716	0.0716
1000	0.0569	0.0632
2000	0.0301	0.0447
5000	0.0129	0.0283

Table 15.1: Width α/n^β of confidence interval

15.1.2 Second case study

I now discuss a second example, in more details. The full documentation is in section 4.4.3 in [13]. In this case, the deterministic data comes from number theory. However, it behaves quite chaotically, and looks like historical stock prices following a market crash: see the left plot in Figure 15.1.

I included this example because it represents spectacular empirical progress towards addressing the infamous Riemann Hypothesis. The mathematics behind the scenes are advanced and well beyond the scope of this book. In section 15.1.2.1, I mention the bare minimum in terms of theory, just enough to help you reproduce my results.

You can skip that section. I also use the standard notation attached to this problem. In particular, $s = \sigma + it$ represents a complex number. Here $t = 0$ and $s = \sigma = 0.90$ is good enough to obtain a ground-breaking result. So, I won't even use complex numbers.

I now focus on the modeling aspect. Given n observations (a_k, b_k) with $k = 1, \dots, n$, the goal is to fit the following model to data, as in a regression framework:

$$\rho_n b_k - a_k = c + \frac{\alpha}{\sqrt{n}} + \frac{\beta}{\sqrt{n} \log n} + \epsilon_k, \quad k = 1, \dots, n \quad (15.1)$$

Here, ϵ_k plays the role or the noise of residual error. In the more general case, ρ_n and a_k, b_k also depend on $s = \sigma + it$, with $0.5 < \sigma < 1$. Here, $\sigma = 0.90$ and $t = 0$ are fixed. The model parameters are c, α, β . We want the best possible parameter estimation, one that converges faster than what the central limit theorem allows for. The model components including deterministic ρ_n that depends on the observations, are described in section 15.1.2.1. Of course, the estimates of c, α, β depend on n .

The trick is to try different values of n , and performs model fitting for each of them separately. That is, on sub-datasets of increasing sizes. The results are shown in Table 15.2. Then extrapolate the estimated values c, α, β well beyond the full dataset at our disposal. To judge the quality of extrapolates, apply the same technique to a fraction of the dataset, and compare the extrapolated values (inferred from a small sample) with the corresponding estimates based on the full dataset.

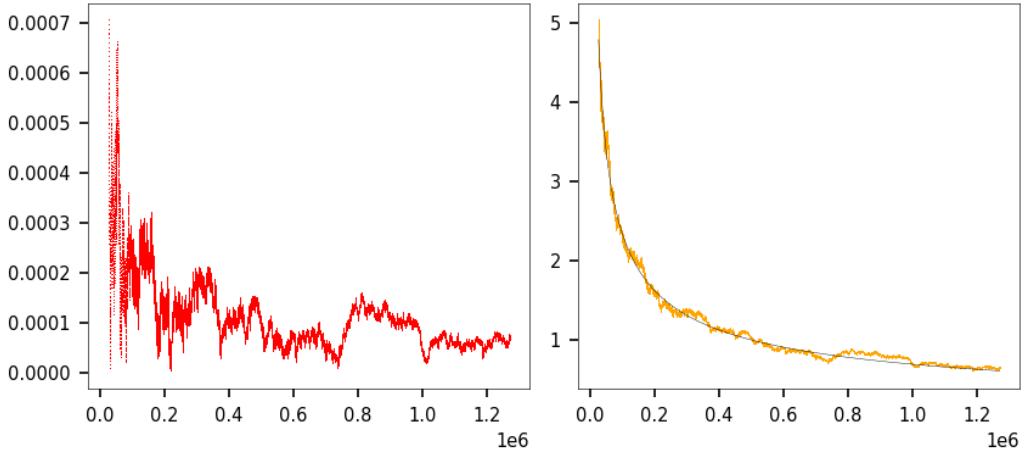


Figure 15.1: High precision extrapolation outperforming standard methods

fast converging CI section physics of numbers xxx update caption

n	γ_n	c	α	β	ρ_n	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 15.2: One curve fitting per row for model (15.1) with sample size n

Table 15.2 shows the estimated c, α, β and R-squared (residual error) for each model fitting, each one with a specific sample size n . If the Generalized Riemann Hypothesis (GRH) is true, the fit should be good and get better as n increases, with $c \rightarrow 0$ as $n \rightarrow \infty$. The quality of the fit and stability in parameter estimates as n increases is actually beyond all expectations. See right plot in Figure 15.1 where the X-axis (the error) has been multiplied by \sqrt{n} compared to the left plot featuring the raw data. Here $n \approx 1.27 \times 10^6$. Nevertheless, this does not prove GRH.

It is surprising that the fit is so good when using the true number theoretic functions, but not as great when replacing them with **synthetic** clones that mimic their behavior, such as random multiplicative **Rademacher functions** or **Beurling primes**. It is as if formula (15.1) perfectly captures and models the exact chaotic behavior of prime numbers, at least up to $n \approx 1.27 \times 10^6$. The Python code for model fitting is in section 4.4.4 in [13], and also on GitHub, [here](#). I used it to produce Figure 15.1 and Table 15.2.

15.1.2.1 Mathematical background: the Riemann Hypothesis

First, I introduce some functions and notations. The non-trivial **Dirichlet character modulo 4**, denoted as χ_4 , is a function defined over all positive integers as follows. If $k \bmod 4 = 1$, then $\chi_4(k) = 1$. If $k \bmod 4 = 3$, then $\chi_4(k) = -1$. Otherwise (that is, if k is even), then $\chi_4(k) = 0$. This function is completely multiplicative: $\chi_4(kk') = \chi_4(k)\chi_4(k')$. The corresponding **Dirichlet-L function** is denoted as $L_4(s)$, and implemented as

```
dirichlet(complex(sigma, t), [0, 1, 0, -1])
```

in the Mpmath Python library. The first argument is the complex number $s = \sigma + it$. The second argument is the parameter $[0, 1, 0, -1]$, indicating that χ_4 is the underlying character. Then I define the corresponding truncated **Euler product** based on the first n prime numbers p_1, p_2 and so on ($p_1 = 2$), as

$$L_4(s, n) = \prod_{k=1}^n \left(1 - \frac{\chi_4(p_k)}{p_k^s}\right)^{-1}.$$

If $\sigma > 1$, it is well known and easy to prove that $L_4(s, n) \rightarrow L_4(s)$ as $n \rightarrow \infty$. It also implies, based on the full Euler product with $n = \infty$, that $L_4(s)$ has no root if $\sigma > 1$. But what happens if $\sigma < 1$? If it converges, it is easy to prove that it converges to $L_4(s)$. But it cannot converge if $\sigma = \frac{1}{2}$ because $L_4(s)$ has infinitely many roots ($t_1 = 6.0209\dots, t_2 = 10.2437\dots, t_3 = 12,9880\dots, t_4 = 14.1346\dots$ and so on) while the Euler product obviously has none. Some of these roots, for instance $s = \frac{1}{2} + it_4$, are also root of the **Riemann zeta function** $\zeta(s)$.

What about $\sigma = 0.90$, the case we are interested in? If the Euler product converges at $s = \sigma = 0.90$, it is again easy to prove that it converges for all $s = \sigma' + it$ with $\sigma' > \sigma$. It would constitute a seminal result in number theory, implying that $L_4(s)$ has no root if $\sigma > 0.90$. This would be a big milestone towards proving the **Generalized Riemann Hypothesis**, and the deepest result obtained so far on this 150 years old problem. Note that $L_4(s)$ and $\zeta(s)$ are sisters belonging to a same class of Dirichlet-L functions. The standard Riemann Hypothesis deals with $\zeta(s)$, the most basic in that family. Yet, $L_4(s)$ is easier to handle.

Now I introduce additional notations to establish the link with the model fitting problem. Again, p_1, p_2 and so on denote the prime numbers, with $p_1 = 2$.

$$\begin{aligned} \Lambda_n &= \frac{1}{n} \sum_{k=1}^n \chi_4(p_k), \\ \delta_n(s) &= L_4(s, n) - L_4(s) \\ \rho_n(s) &= \frac{\text{Stdev}[\Lambda_1, \dots, \Lambda_n]}{\text{Stdev}[\delta_1(s), \dots, \delta_n(s)]}. \end{aligned}$$

The link to Formula (15.1) is as follows: $a_k = \Lambda_k, b_k = \delta_k(s), \rho_n = \rho_n(s)$, with $s = \sigma + it$ and $\sigma = 0.90, t = 0$. In Table 15.2, $\gamma_n = \sqrt{n} \cdot \Lambda_n$, while R^2 is the R-squared attached to each fit (one per row), also depending on the sample size n . Here n ranges from about 127,000 to 1.27×10^6 and Stdev stands for the standard deviation.

To make progress on proving GRH, the next step consists of checking if both $\gamma_n = \sqrt{n} \cdot \Lambda_n$ and ρ_n converge to some finite, strictly positive constants. If this is the case, it could facilitate the proof of GRH or a weaker yet spectacular result, at least for $L_4(s)$. However, even proving this intermediary step may be very hard if not impossible. More details can be found in section 4.4 in [13], and in chapter 17 – fully dedicated to GRH and synthetic number theoretic functions – in [15].

15.2 Generic, all-in-one curve fitting, regression and clustering

The technique described here was first introduced as “cloud regression” in chapter 2 in [14], to emphasize the fact that the observations are like a cloud: there may or may not be an independent variable or response. The approach covers supervised and unsupervised learning in a same algorithm, including unsupervised regression. Here I focus on one particular case: curve fitting. I start by introducing the concept of data (called point cloud), parameter, and shape.

The **dataset** is denoted as W , and consists of $m + 1$ variables and n observations. Thus W is a $n \times (m + 1)$ matrix as usual. The k -th row corresponds to the k -th observation $W_k = (W_{k,0}, W_{k,1}, \dots, W_{k,m+1})$. For backward compatibility with traditional models, I use the notation $W_{k,0} = Y_k$ for the dependent variable or response (if there is one), and $(X_{k,1}, \dots, X_{k,m}) = (W_{k,1}, \dots, W_{k,m+1})$ for the independent variables of features. The column vector corresponding to the response is denoted as Y , and the $n \times m$ matrix representing the independent variables is denoted as X . The whole data set W is referred to as the point cloud.

The **parameter** is a multivariate column vector denoted as $\theta = (\theta_0, \theta_1, \dots, \theta_d)$, with $d + 1$ components. Typically, $d = m$ and θ satisfies some constraint, specified by $\eta(\theta) = 0$ for some function η . The most common functions are $\eta(\theta) = \theta^T \theta - 1$, $\eta(\theta) = \theta_0 + 1$, and $\eta(\theta) = (\theta_0 + \dots + \theta_d) - 1$. Here T denotes the matrix/vector transposition operator.

The purpose is to fit a **shape** to the point cloud. The most typical shapes, after proper mapping, are hyperplanes or quadratic forms (ellipsoids). The former is a particular case of the latter. The shape belongs to a parametric family of equations driven by the multivariate parameter θ . The equation of the shape is $g(w, \theta) = 0$, for some function g . Typical examples include $g(w, \theta) = w\theta$ and $g(w, \theta) = w\theta - 1$, with $d = m$. The former usually involves an intercept: $X_{k,1} = 1$ for all $k = 1, \dots, n$. Keep in mind that w and θ are vectors, but $g(w, \theta)$ is a real number, not a vector. Thus $w\theta$ represents a **dot product** [Wiki].

15.2.1 Solution, R-squared and backward compatibility

The shape that best fits the data corresponds to $\theta = \theta^*$, obtained by minimizing the squares:

$$\theta^* = \arg \min_{\theta} \sum_{k=1}^n g^2(W_k, \theta). \quad (15.2)$$

The solution may not be unique. Uniqueness and **numerical stability** will be addressed in a future article, but the basics are covered in this document. The constraint $\eta(\theta) = 0$ guarantees that the solution requires solving a (sometimes non-linear) system of $d + 2$ equations with $d + 2$ unknowns. In some cases, $d \leq m$ to avoid **model identifiability** issues [Wiki]. Also, a large d may result in **overfitting** [Wiki]. Then, you want $n > d$ otherwise the solution may not be unique unless you add more constraints on θ . The solution θ^* is obtained by solving the system

$$\begin{cases} \sum_{k=1}^n \nabla_{\theta}[g^2(W_k, \theta)] = \lambda \nabla_{\theta}[\eta(\theta)], \\ \eta(\theta) = 0 \end{cases} \quad (15.3)$$

where ∇_{θ} is the **gradient** operator with respect to θ [Wiki], and λ is called the **Lagrange multiplier** [Wiki]. This is a classic constrained convex optimization problem. The top part of (15.3) consists of a system of $d + 1$ equations with $d + 2$ unknowns $\theta_0, \dots, \theta_d$ and λ . The bottom part is a single equation with $d + 1$ unknowns $\theta_0, \dots, \theta_d$. Combined together, it constitutes a system of $d + 2$ equations with $d + 2$ unknowns. Note the analogy with **Lasso regression** [Wiki] when $\eta(\theta) = \theta^T \theta - 1$, that is, when $\theta^T \theta = 1$.

The **mean squared error** (MSE) relative to a particular θ is defined as

$$\text{MSE}(\theta) = \frac{1}{n} \sum_{k=1}^n g^2(W_k, \theta) \geq \text{MSE}(\theta^*). \quad (15.4)$$

The inequality in (15.4) is an immediate consequence of (15.2). Now define the **R-squared** with respect to θ as

$$R^2(\theta) = 1 - \frac{\text{MSE}(\theta^*)}{\text{MSE}(\theta)}. \quad (15.5)$$

It follows immediately that $0 \leq R^2(\theta) \leq 1$. A perfect fit corresponds to $\text{MSE}(\theta^*) = 0$ (the whole cloud residing on the shape). In that case, if $\theta \neq \theta^*$ and the optimum θ^* is unique, then $R^2(\theta) = 1$.

In traditional linear regression, the R-squared is defined as $R^2(\theta_*)$ where θ_* is the optimum θ for the base model. The base model corresponds to all the coefficients θ_i attached to the independent variables set to zero, except the one attached to the intercept. In other words, in the base model, the predicted Y is constant, equal to the empirical mean of Y . As a result, $\text{MSE}(\theta_*) = \text{Var}[Y]$, the empirical variance of Y . A consequence is that $R^2(\theta_*)$ is the square of the correlation between the observed response Y , and the predicted response of the full model.

Backward compatibility with traditional linear regression works as follows. The standard univariate regression corresponds to $g(w, \theta) = w\theta = \theta_0 y + \theta_1 x + \theta_2$, with the constraint $\theta_0 = -1$. Thus $g(w, \theta) = 0$ if and only if $y = \theta_1 x + \theta_2$. This generalizes to multivariate regression as well. A more elegant formulation in the new methodology is to replace the constraint $\theta_0 = -1$ by the symmetric constraint $\theta_0^2 + \theta_1^2 + \theta_2^2 = 1$. Note that w is a row vector and θ is a column vector.

15.2.2 Model upgrades

By model, I mean the general setting of the method: there is no probabilistic model involved in this discussion. [Prediction intervals](#) [Wiki] for the individual error $g(W_k, \theta^*)$ at each data point W_k (or for the estimated response attached to Y_k , if there is an independent variable) and [confidence regions](#) [Wiki] for θ^* can be obtained via re-sampling and [bootstrapping](#) [Wiki]. This is also true for points outside the training set.

Also, the squares can be replaced by absolute values, as in [quantile regression](#) [Wiki], to minimize the impact of outliers and for scale preservation: if a variable is measured in years, then squares are expressed in squared years, a metric that is meaningless. This leads to a modified, better metric to assess the quality of the fit, replacing the R-squared. The [goodness-of-fit](#) (say, the R-squared) should be measured on the [validation set](#) [Wiki] even though θ^* is computed on a subset of the [training set](#): this is a standard practice, called [cross-validation](#) [Wiki].

Now, let's get back to the [R-squared](#). In standard linear regression, the R-squared is defined as $R^2(\theta_*)$ via Formula (15.5), where θ_* is the optimum θ for the base model (the predicted response is constant, equal the mean of Y for the base model). In the new methodology, there may be no response. Still, the definition of R^2 extends to that situation, and is compatible with the traditional version. What's more, it leads to many possible R^2 , one for each sub-model (not just the base model), and this is true too for the standard regression. A sub-model corresponds to adding constraints on the parameter vector θ , or in other words, working with a subset of the parameter space. Let θ_* be the optimum for a specific sub-model, while θ^* is the optimum for the full model. Then the definition of R^2 , depending on θ_* , is unchanged. It could not be any simpler!

Now you can use R^2 for model comparison purposes and even for [feature selection](#) [Wiki]. You can test the improvement obtained by using the full model over a sub-model, with the metric $S(\theta_*) = R^2(\theta^*) - R^2(\theta_*)$. Here θ_* is the optimum θ attached to the sub-model. Obviously, $0 \leq S(\theta_*) \leq 1$. The larger $S(\theta_*)$, the bigger the improvement. Conversely, the smaller, the better the performance of the sub-model. Examples include fitting an ellipse (full model) versus fitting a circle (sub-model) or using all the features (full model) versus using a subset (sub-model). You can compare sub-models and rank them according to $S(\theta_*)$. This allows you to identify the smallest set of features that achieve a good enough $S(\theta_*)$, for [dimensionality reduction](#) purposes [Wiki].

Finally, another update consists of using positive weights $\psi_k(\theta)$ in Formula (15.2). This amounts to performing [weighted regression](#) [Wiki]. For instance, data points far away from the optimum shape, that is observations with a large $g^2(W_k, \theta^*)$, may be discarded to reduce the impact of outliers. Or the weights can be used to balance the coefficients θ_i , in an effort to achieve scale-invariance in the expression $w\theta$. Then the top system in (15.3) becomes

$$\sum_{k=1}^n \psi_k(\theta) \nabla_\theta [g^2(W_k, \theta)] + \sum_{k=1}^n g^2(W_k, \theta) \nabla_\theta [\psi_k(\theta)] = \lambda \nabla_\theta [\eta(\theta)]. \quad (15.6)$$

15.2.3 Case studies

In section 15.2.3.1, I show how to solve the logistic regression. The first version is standard least squares, to further illustrate backward compatibility with the traditional method. The second one illustrates how it could be done if you want to follow the spirit of the new methodology. Then I discuss two fundamental examples based on synthetic data.

15.2.3.1 Logistic regression, two ways

In the traditional setting, $w = (y, x)$ where y is the response, and x the features. For the [logistic regression](#) [Wiki], we have

$$g(w, \theta) = g(y, x) = y - F(x\theta), \quad \text{with } x\theta = \theta_1 x_1 + \dots + \theta_m x_m.$$

Here $x_1 = 1$ corresponds to the intercept, thus we have $m - 1$ actual features x_2, x_3, \dots, x_m . There is no constraint on the parameter θ , thus there is no function $\eta(\theta)$. In Formula (15.3), $\eta(\theta) = 0$ should be ignored, and $\lambda = 0$. The function F is a cumulative distribution function with a symmetric density around the origin. In this case, $F(x\theta) = 1/(1 + \exp[-x\theta])$ is the standard [logistic distribution](#) [Wiki].

In the new methodology, one would proceed as follows. First, the original data is denoted as $z = (v, u)$. The logistic regression applies to the original data. Here v is the response, and u the feature vector. The parameter θ is unchanged (not subject to a mapping), and still denoted as θ . This regression can be stated as

$$g(z, \theta) = g(v, u) = v - F(u\theta), \quad \text{with } u\theta = \theta_1 u_1 + \dots + \theta_m u_m.$$

The first step is to map $z = (v, u)$ onto $w = (y, x)$, with the hope of simplifying the problem. This is done via the link function $y = F^{-1}(v) = \log[v/(1-v)]$ and $u = x$. Now we are back to

$$g(z, \theta) = g(w, \theta) = g(y, x; \theta) = y - x\theta, \quad \text{with } x\theta = \theta_1 x_1 + \dots + \theta_m x_m.$$

This is how standard linear regression is expressed in the new framework. But it is still the traditional linear regression, with nothing new. The final step consists in extending θ , adding one component θ_0 to $\theta_1, \dots, \theta_m$. With the new θ (still denoted as θ) we have $g(w, \theta) = w\theta = \theta_0 w_0 + \dots + \theta_m w_m$. You need to add one constraint on θ . The constraint $\theta_0 = -1$, that is $\eta(\theta) = \theta_0 + 1$, yields the exact same solution as traditional linear regression. But $\theta^T \theta = 1$, that is $\eta(\theta) = \theta^T \theta - 1$, makes the problem somehow symmetric, and more elegant.

However, in many applications, the response v in the original space is either 0 or 1, such as cancer versus non-cancer, or fraud versus non-fraud. In this case, the link function is undefined. The mapping with the link function works if the response is a proportion, strictly between zero and one. Otherwise, the standard logistic regression is the best approach. A possible workaround is to use for F a distribution with a finite support, such as uniform on $[a, b]$. Afterall, the observed values (the features) are always bounded anyway. Then, intuitively, given θ , estimates of a and b are proportional respectively to the minimum and maximum of $U_k \theta$, over $k = 1, \dots, n$.

This suggests a new approach to logistic regression. First, use the model $v = F_\theta(u\theta)$ in the (v, u) -space, where $0 \leq v \leq 1$ and F_θ is the empirical distribution [Wiki] of $u\theta$ given θ . Then choose θ^* that minimizes the sum of squared residuals:

$$\theta^* = \arg \min_{\theta} \sum_{k=1}^n g^2(V_k, U_k; \theta) = \arg \min_{\theta} \sum_{k=1}^n (V_k - F_\theta(U_k \theta))^2.$$

Remember, U_k is a row vector, and θ is a column vector; the dot product $U_k \theta$ is a real number. Also, V_k is the binary response attached to the k -th observation, while U_k is the corresponding m -dimensional feature vector, both in the original (v, u) -space. The empirical distribution F_θ is computed as follows: $F_\theta(t)$ is the proportion of observed feature vectors, among U_1, \dots, U_n , satisfying $U_k \theta \leq t$. Such a method could be called CDF regression. You can use the methodology presented here to solve it, but it would be very computer-intensive, because F_θ depends on θ in a non-obvious way. The predicted value for V_k , is $F_{\theta^*}(U_k \theta^*)$ in this case.

15.2.3.2 Ellipsoid and hyperplane fitting

This is a fundamental example, with hyperplanes being a particular case of ellipsoids. I illustrate the methodology with an example based on synthetic data, in a small dimension. The idea is to represent the shape with a quadratic form. In two dimensions, the equations is

$$\theta_0 x^2 + \theta_1 xy + \theta_2 y^2 + \theta_3 x + \theta_4 y + \theta_5 = 0.$$

The trick is to re-write it with artificial variables $w_0 = x^2, w_1 = xy, w_2 = y^2, w_3 = x, w_4 = y, w_5 = 1$, so that we can use the general framework with $g(w, \theta) = w\theta$. Again, $w\theta$ is the dot product. To avoid the trivial solution $\theta^* = 0$, let's add the constraint $\theta^T \theta = 1$, that is, $\eta(\theta) = \theta^T \theta - 1$. Then, θ^* is solution of the system

$$\begin{cases} (W^T W - \lambda I)\theta = 0, \\ \theta^T \theta = 1. \end{cases} \quad (15.7)$$

The above solution is correct in any dimension. It is a direct application of (15.3). Here W is the $n \times 6$ matrix containing the n observations. Thus, $W_{k0} = X_k^2, W_{k1} = X_k Y_k, W_{k2} = Y_k^2, W_{k3} = X_k, W_{k4} = Y_k, W_{k5} = 1$. The Python code and additional details, for a slightly different version with a slightly different $\eta(\theta)$, can be found [here](#). I use it in my own code, available on my GitHub repository, [here](#), under the name `fittingEllipse.py`. It is based on Halir's article about fitting ellipses [17].

The Python code checks if the fitted shape is actually an ellipse. However, in the spirit of my methodology, it does not matter if it is an ellipse, a parabola, an hyperbola or even a line. The uniqueness of the solution is unimportant: indeed, if two very different solutions (say an ellipse and a parabola) yield the same minimum mean squared error and are thus both optimal, it says something about the data set, something interesting to know. However, it would be interesting to compute $R^2(\theta_*)$ using Formula (15.5), where θ_* corresponds to a circle. It would tell whether the full model (ellipse) over a significant improvement over the circle sub-model.

Ellipsoid fitting shares some similarities with multivariate polynomial regression [34]. The differences are:

- Ellipse fitting is a “full” model; in the polynomial regression $y = \theta_1 + \theta_2 x + \theta_3 x^2$, the terms y^2 and xy are always missing.

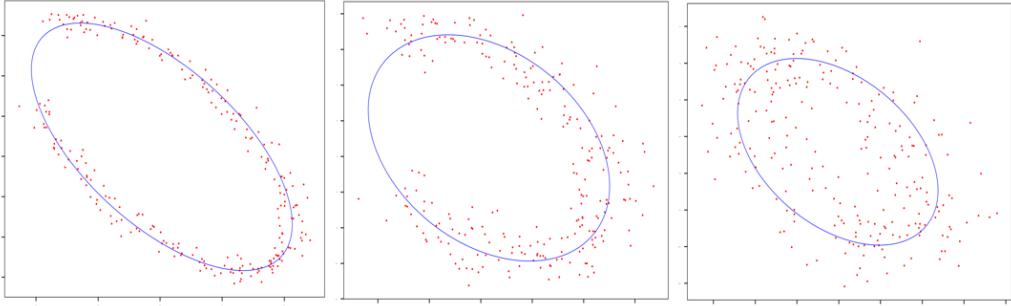


Figure 15.2: Fitted ellipse (blue), given the training set (red) distributed around a partial arc

- Polynomial regression fits a curve that is unbounded such as $y = x^2$, resulting in poor fitting; to the contrary in ellipse fitting (if the solution is actually an ellipse) the solution is bounded.
- To get as many terms in polynomial regression as in ellipse fitting, the only way is to increase the degree of the polynomial, which further increases the instability of the solution.

Finally, Umbach [33] proposes a different approach to ellipse fitting. It is significantly more complicated, and indeed, they stopped at the circle. In short, their method directly estimates the center, semi-axis lengths and rotation angle via least squares, as opposed to estimating the coefficients in the quadratic form that represents the ellipse.

15.2.3.3 Curve fitting: 250 examples in one video

Ellipse fitting is performed by setting `mode='CurveFitting'` in the Python code. The program automatically creates a number of ellipses, specified by their parameters (center, lengths of semi-axes, and rotation angle), then generates a different training set for each ellipse, and outputs the result of the fitting procedure as an image. The images are then bundled together to produce a video, and an animated gif. Each image features a particular ellipse and training set, as well as the fitted ellipse based on the training set. The ellipses parameters are set by the variable `params` in the code: it is an array with five components. The number of ellipses is set by the parameter `nframes`, which also determines the number of frames in the output video.

Actually, the program does a little more: it works with ellipse arcs. Using the centroid of the training set to estimate the center of the ellipse does not work in this case. So the program retrieves the original, unknown ellipse even if the training set consists of points randomly distributed around a portion of that ellipse. The arc in question is determined by a lower and upper angle in polar coordinates, denoted respectively as `tmin` and `tmax` in the code, with `tmin=0` and `tmax=2π` corresponding to the full ellipse.

The training set consists of n observations generated as follows. First sample n points on the ellipse (or the arc you are interested in). Then perturb these points by adding some noise. You have two options: `noise_CDF='Uniform'` and `noise_CDF='Normal'`. The amount of noise is specified by the parameter `noise` in the code. For the uniform distribution on the square $[-a, a] \times [-a, a]$, `noise` represents a . For the bivariate normal distribution with covariance matrix $\sigma^2 I$ where I is the identity matrix, it represents σ^2 . There are various ways of sampling points on an ellipse. Three options are offered here, set by the parameter `sampling`. They are described in section 15.2.4, in the paragraph “Sampling points on an ellipse arc”. The option ‘Enhanced’ is the only one performing stochastic sampling (points randomly picked up on the ellipse), and used in Figure 15.3.

In Figure 15.3, the size of the training set is $n = 30$ while in Figure 15.2, $n = 250$. In the code, n is represented by the variable `npts`. The training set is colored in red, the fitted ellipse in blue, and if present on the image as in Figure 15.3, the true ellipse is in black. The latter appears as a polygon rather than an ellipse because the sampled points on the true ellipse are joined by segments, and n is small. Typically, the true and fitted ellipses are very close to each other, although there is a systematic bias too small to be noticed to the naked eye unless the ellipse eccentricity is high. More on this soon.

Table 15.3 compares the exact parameter values (set by the user) of the true ellipse in Figure 15.3, to a few sets of estimated values obtained by least squares. Each set of estimates is computed using a different training set. All training sets are produced with same amount and type of noise, to give an idea of the variance of the parameter estimates at a specific level of noise. The five parameters are the ellipse center (x_0, y_0) , the lengths of the semi axes (a_p, b_p) , and the ellipse orientation (the rotation angle ϕ).

In some cases, the solution may not be unique, or could be an hyperbola or parabola rather than an ellipse. For instance, if the ellipse is reduced to a circle, any value for the rotation angle is de facto correct, though

the estimated curve is still unique and correctly identified. Also, if the true ellipse has a high eccentricity, the generated white (unbiased) noise forces the training set points inside the ellipse more often than they should, as opposed to outside the boundary. This is because inside the ellipse, the noise from the North side strongly overlaps with the noise from the South side, assuming the long axis is the horizontal one. The result is biased estimates for a_p and b_p , smaller than the actual ones. In the end, the fitted curve has a higher eccentricity than the true one. The effect is more pronounced the higher the eccentricity. If the variance of the noise is small enough, there is almost no bias.

I posted a video featuring 250 fitted ellipses with the associated training sets, [here](#) on YouTube. It is also on GitHub, [here](#). The accompanying animated gif is also on GitHub, [here](#). All were produced with the Python code. In the video, the transition from one ellipse to the next one is very smooth. While I use 250 different combinations of arcs, rotation angles, eccentricities and noises to feature a large collection of very different cases, these configurations slowly progress from one frame to the next one in the video. But the 250 frames eventually cover a large spectrum of situations. The last one shows a perfect fit, where the training set points are all on the true ellipse.

15.2.3.4 Confidence region for the fitted ellipse: application to meteorite shapes

The computation of [confidence regions](#) is performed by setting mode='ConfidenceRegion' in the Python code. This time the program automatically creates a number of training sets (determined by the parameter nframes), for the same ellipse specified by its parameters params: center, lengths of semi-axes, and rotation angle. Then it estimates the ellipse parameters, and thus the true ellipse, uniquely determined by the parameters in question. Figure 15.3 shows the confidence region for the example outlined in Table 15.3.

	x_0	x_1	a_p	b_p	ϕ
Exact values	3.00000	-2.50000	7.00000	4.00000	0.78540
Training set 1	2.61951	-2.41818	6.44421	3.82838	0.72768
Training set 2	2.77270	-2.32346	6.59185	4.24624	0.59971
Training set 3	3.29900	-2.60532	6.71834	4.15181	0.87760
Training set 4	2.71936	-2.42349	7.15562	4.52900	0.80404

Table 15.3: Estimated ellipse parameters vs true values ($n = 30$), for shape in Figure 15.3

Actually I decided to display a polygon instead of the fitted ellipse, by selecting the option sampling='Enhanced'. The polygon consists of the predicted locations of the $n = 30$ training set points on the fitted ellipse. These locations are obtained in the exact same way that predicted values are obtained in a linear regression problem and then shown on the fitted line. After all, ellipse fitting as presented in this section is a particular case of the general cloud regression technique. I then joined these points using segments, resulting in one polygon per training set. The superimposition of these polygons is the confidence region.

The reason for using polygons rather than ellipses is for a particular application: estimating the shape of a small, far away celestial body based on a low resolution image. This is particularly useful when creating a taxonomy of these bodies: the shape parameters are used to classify them and understand their history as well as gravity interactions, and can be used as features in a machine learning algorithm. Then, for a small meteorite, people expect to see it as a polyhedron (the 3D version of a polygon) rather than an ellipsoid. Of course, if the number n of points in the training set is large, then the polyhedron is indistinguishable from the fitted ellipsoid. But in practice, with low resolution images, n is usually pretty small.

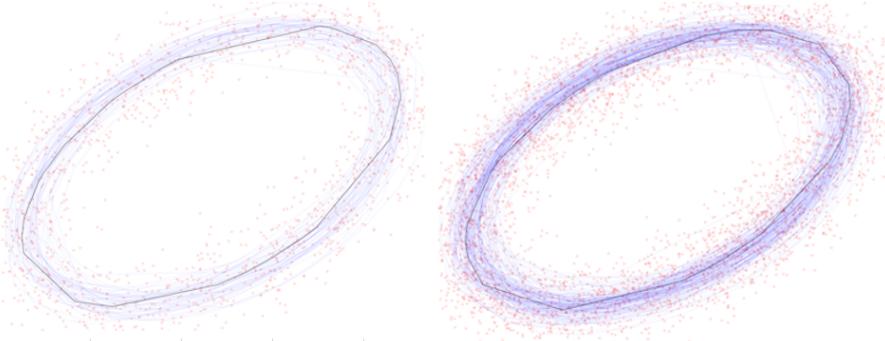


Figure 15.3: Confidence region in blue, $n = 30$ training set points; 50 training sets (left) vs 150 (right)

15.2.4 Python code

The main parameters in the code are highlighted in red in this high level summary. The code is on GitHub [here](#), under the name `fittingEllipse.py`.

The least square optimization is performed using an implementation of the Halir and Flusser algorithm [17], adapted from a version posted [here](#) by Christian Hill, the author of the book “Learning Scientific Programming with Python” [18]. The optimization – minimizing the sum of squared errors between the observed points and the fitted ellipse – is performed on the coefficients of the quadratic equation representing the ellipse. This is the easiest way to do it, and it is also the approach that I use elsewhere in this chapter. The function `fit_ellipse` does the job, while `cart_to_pol` converts these coefficients into meaningful features: the center, rotation angle, eccentricity and the major and minor semi-axes of the ellipse [[Wiki](#)].

15.2.4.1 Sampling points on an ellipse arc

The Python code also integrates other components written by various authors. First, it offers three options to sample points on an ellipse or a partial arc of an ellipse, via the parameter `sampling` in the main section of the code:

- Evenly spaced on the perimeter, via the function `sample_from_ellipse_even`. The code is adapted from an anonymous version posted [here](#). It requires the evaluation of [elliptic integrals](#) [[Wiki](#)].
- Randomly chosen on the perimeter, in such a way that on average, the distance between two consecutive sampled points on the ellipse is constant. It involves sampling from a multinormal distribution, rescaling the points and then sorting the sampled points so that they are ordered on the perimeter. This also requires sorting an array according to another array. It is done in the function `sample_from_ellipse`.
- The standard, easiest but notoriously skewed sampling. It consists of choosing equally spaced angles in the polar representation of the ellipse. For curve fitting, it is good enough with very little differences compared to the two other methods.

For sampling on a partial arc rather than the full ellipse, set the parameters `tmin` and `tmax` to the appropriate values, in the main loop. These are angles in the polar coordinate system, and should lie between 0 and 2π . The full ellipse corresponds to `tmin` set to zero, and `tmax` set to 2π .

15.2.4.2 Training set and ellipse parameters

Then, to create the training set, perturb the sampled points on the ellipse via uniform or Gaussian noise. The choice is set by the parameter `noise_CDF` in the main section of the code. The parameter `noise` determines the amount of noise, or in other words, the noise variance. Points, be it on the ellipse or in the training set, are arrays with names `x` and `y` (respectively for the X and Y coordinates). The number of points in the training set is determined by the parameter `npts`.

The shape of the ellipse is set by the 5-dimensional parameter vector `params`. Its components, denoted as `x0`, `y0`, `ap`, `bp`, `phi` throughout the code, are respectively the center of the ellipse, the length of the semi-axes, and the orientation of the ellipse (the rotation angle).

15.2.4.3 Confidence regions versus curve fitting

The program creates `nframes` ellipses, one at a time in the main loop. At each iteration, the created ellipse and training set is saved as a PNG image, for inclusion in a video or animated gif (see next paragraph). This is why the variable controlling the main loop is called `frame`. At each iteration the true parameters of the ellipse (the ones you chose), and their least squares estimates are displayed on the screen.

If the parameter `mode` is set to ‘ConfidenceRegion’, then the amount of noise and all ellipse parameters are kept constant throughout the iterations. The fitted shapes varies from one iteration to the next depending on the training set (itself depending on the noise), creating a [confidence region](#) for a specific ellipse, given a specific amount of noise. New fitted ellipses keep being added to the image without erasing older ones, to display the confidence region under construction. Highly eccentric ellipses result in biased confidence regions. The method used to build the confidence region is known as [parametric bootstrap](#) [[Wiki](#)].

To the contrary, if `mode` is set to ‘FittingCurves’, a different ellipse with different parameters and different amount of noise is generated at each iteration, erasing the previous one in the new image. The purpose in this case is to assess the quality of the fit depending on the amount of noise and the shape of the ellipse (the eccentricity and whether you use a full or partial arc for training, in particular).

15.3 A simple geospatial interpolation method

This technique is described in details in chapter 4 in [14]. In this section I focus on one application and case study: geospatial interpolation. In the process, I also show how the methodology can be used to generate [synthetic data](#), be it time series or geospatial data.

Depending on the parameters, the interpolation is either close to nearest-neighbor methods, [kriging](#) [Wiki] (also known as Gaussian process regression), or a truly original and hybrid mix of additive and multiplicative techniques. There is an option not to interpolate at locations far away from the training set, where regression or interpolation results may be meaningless, regardless of the technique used.

The general principle is simple. We want to interpolate a function $g(t)$ at certain points $t = \rho_1, \rho_2, \dots$ belonging to a set R called the root set. These points are the roots of some function ψ . We create a function $w(t, \rho)$ which is equal to zero only if $t = \rho$ and $\rho \in R$. The functions ψ and w are chosen so that when $t \rightarrow \rho \in R$, the limit $\psi(t)/w(t, \rho)$ – a quotient where both the numerator and denominator are zero – exists and is different from zero. The limit in question is denoted as $\lambda(\rho)$. The interpolated function, denoted as $f(t)$ and defined by (15.8), is by construction identical to $g(t)$ when $t \in R$. This leads to the formulation

$$f(t) = \psi(t) \cdot \sum_{\rho \in R} \frac{f(\rho)}{\lambda(\rho)} \cdot \frac{1}{w(t, \rho)}, \quad \text{with } \lambda(\rho) = \lim_{t \rightarrow \rho} \frac{\psi(t)}{w(t, \rho)}. \quad (15.8)$$

Here $w(t, \rho) = 0$ if and only if $t = \rho$. The functions ψ and w must be chosen so that the limit in Formula (15.8) always exists and is different from zero. Typically, $w(t, \rho)$ measures how close t and ρ are to each other. If the summation is infinite and the series is [conditionally convergent](#) [Wiki] – as opposed to [absolutely convergent](#) – then the roots ρ need to be properly ordered. Convergence of the series may also require that $w(t, \rho) \rightarrow \infty$ fast enough as $|\rho| \rightarrow \infty$ and t is fixed.

In one dimension, the limit can be computed using l'Hôpital's rule [Wiki]:

$$\lambda(\rho) = \frac{\psi'(\rho)}{w'(\rho, \rho)}, \quad \text{with } \psi'(t) = \frac{\partial \psi(t)}{\partial t} \text{ and } w'(t, \rho) = \frac{\partial w(t, \rho)}{\partial t}.$$

Multiple applications of l'Hôpital's rule may be required for [roots with multiplicity](#) [Wiki]. The symbol ∂ stands for the [partial derivative](#) [Wiki], here with respect to t . In higher dimensions, the limit usually does not exist except under certain circumstances, see [23] and section 15.3.1.

15.3.1 Problem in two dimensions

In this section I discuss a basic example with a finite summation, where everything works nicely. However, it is a fundamental and very important case, as it applies to all regression problems. Also, it easily generalizes to higher dimensions. I use the notation $t = (x, y)$ and $z = f(t) = f(x, y)$. Let us assume that $\psi(x, y)$ has n roots $\rho_k = (x_k, y_k)$ with $k = 1, \dots, n$. The setting is as follows: we have a dataset with n observations (z_k, x_k, y_k) for $k = 1, \dots, n$. Here z_k is the response or dependent variable, and x_k, y_k are the two features, also called independent variables or predictors. I use

$$\psi(t) = \psi(x, y) = \prod_{k=1}^n w_k(x, y), \quad \lambda(\rho_k) = \lambda(x_k, y_k) = \prod_{i \neq k} w_i(x, y),$$

with the notation $w_k(x, y) = w(t, \rho_k) = w(x, y; x_k, y_k)$. I provide a specific example in formula (15.10). For now, let us keep in mind that by construction, $w(x, y; x', y') = 0$ if and only if $(x, y) = (x', y')$. It follows that

$$z = f(x, y) = \sum_{k=1}^n \gamma_k f(x_k, y_k), \quad \text{with } \gamma_k = \prod_{i \neq k} \frac{w_i(x, y)}{w_i(x_k, y_k)} = \prod_{i \neq k} \frac{w(x, y; x_i, y_i)}{w(x_k, y_k; x_i, y_i)}. \quad (15.9)$$

Thus, $z_k = f(x_k, y_k)$, for $k = 1, \dots, n$. Given a new observation (x, y) , the predicted response z , based on the n data points in the training set, is provided by formula (15.9). If (x, y) is already in the training set, then the predicted z will be exact.

Unlike in traditional kernel-based methods, here the choice of the “distance” or “kernel” function w is critical. Some adaptations preserve the fact that $z_k = f(x_k, y_k)$ for $k = 1, \dots, n$, while providing significantly better predictions and smoothness for observations outside the training set. This makes the method a suitable alternative to regression techniques. In particular, I implemented the following upgrades:

- Replacing γ_k by $\gamma'_k = \gamma_k / (1 + \gamma_k)$. It guarantees that these coefficients lie between 0 and 1.

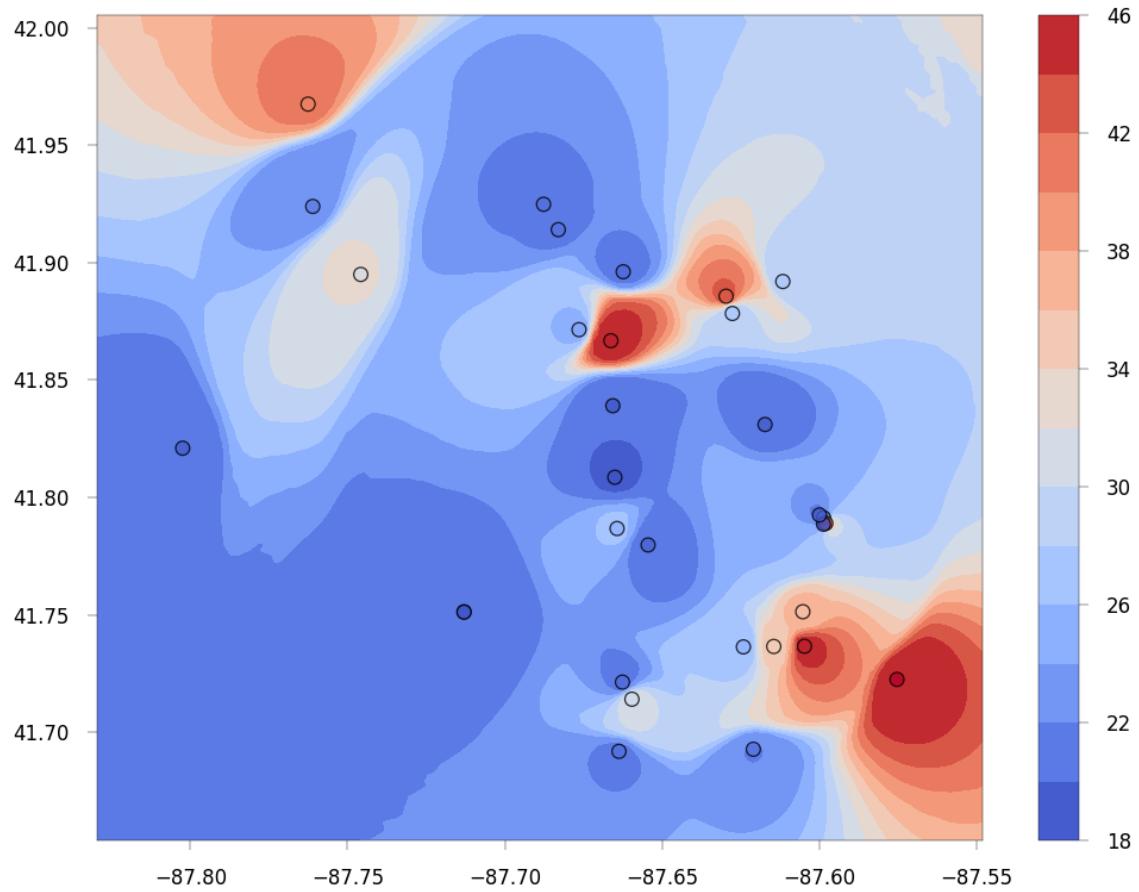


Figure 15.4: Temperature data: interpolation with my method (observed values at dots)

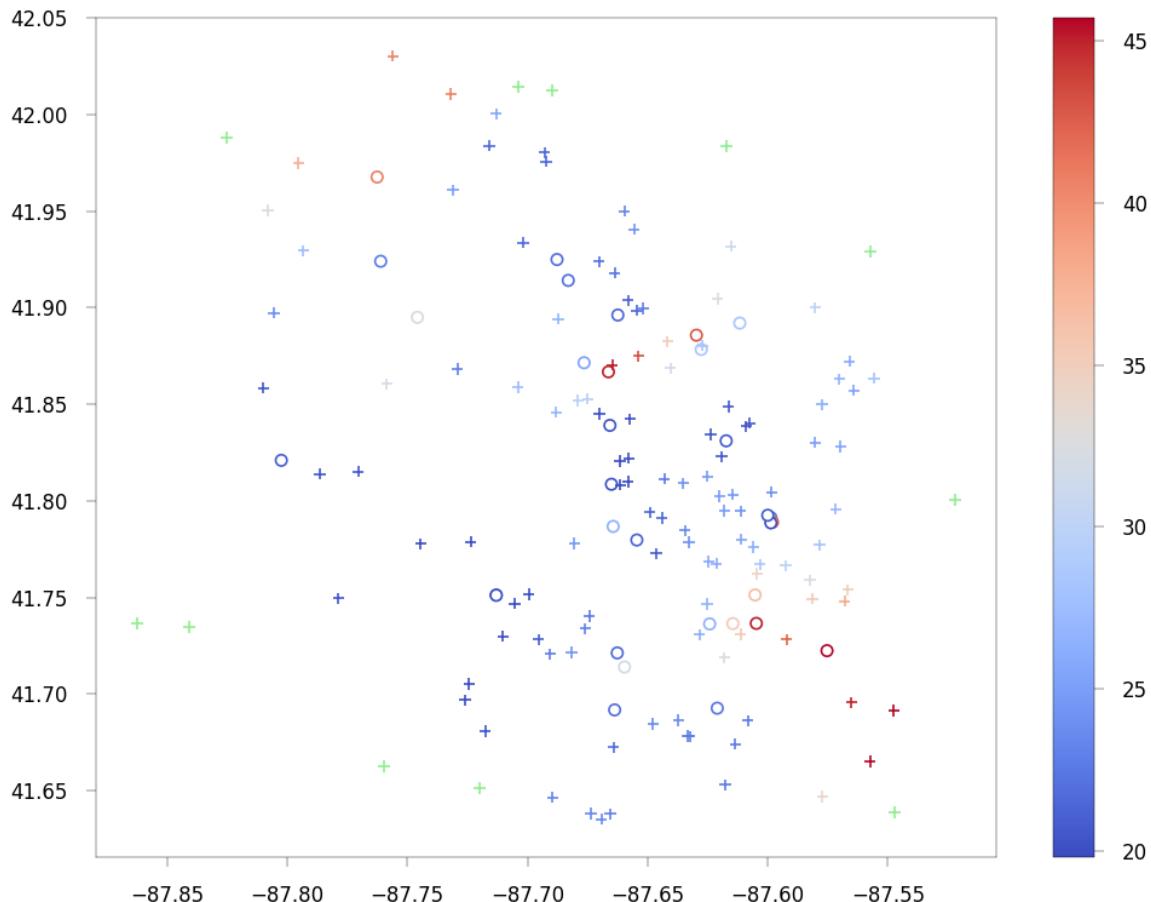


Figure 15.5: My method: round dots represent observed values, “+” are interpolated

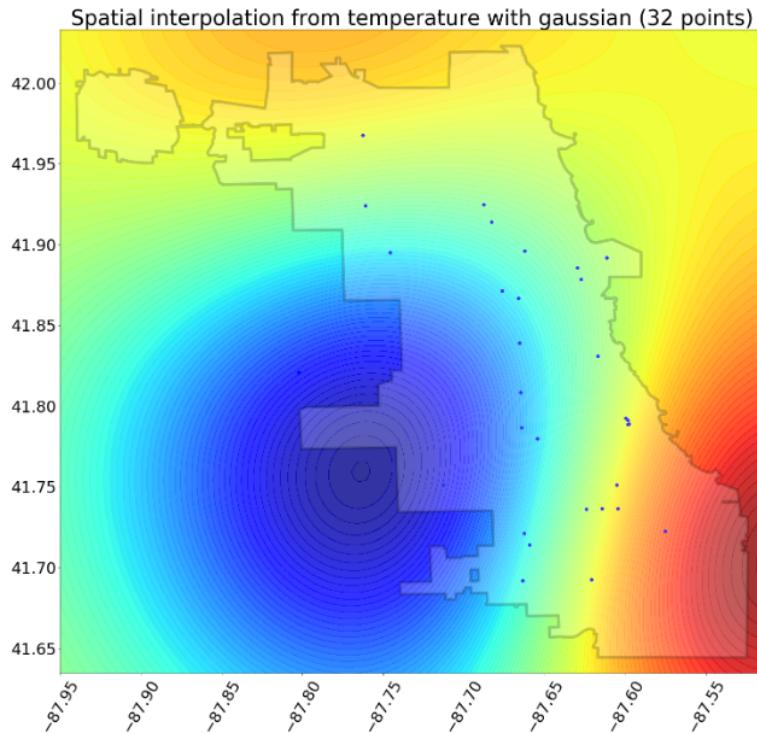


Figure 15.6: Temperature dataset: interpolation using ordinary kriging

- Replacing γ'_k by $\gamma_k^* = \gamma'_k / w_k^\kappa(x, y)$ where $\kappa \geq 0$ is an hyperparameter. This reduces the impact of the point (x_k, y_k) if it is too far away from (x, y) .
- Normalizing γ_k^* so that their sum is equal to 1. This eliminates additive bias outside the training set.

These transformations make the technique somewhat hybrid: a combination of multiplicative, additive, and nearest neighbor methods. Further improvement is obtained by completely ignoring a point (x_k, y_k) when interpolating $f(x, y)$, if $w_k(x, y) > \delta$. Here $\delta > 0$ is an hyperparameter. It may result in the inability to make a prediction for a point (x, y) far away from all training set points: this is actually a desirable feature, not a defect.

15.3.2 Spatial interpolation of the temperature dataset

To test the method presented in section 15.3.1, I used the streaming high frequency temperature data in Chicago, retrieved from [Array of Things](#). The data was analyzed [here](#) in 2019 using CyberGISX, a set of [GIS](#) tools [\[Wiki\]](#) developed in Python by the University of Illinois. They used ordinary kriging. The dataset has 3 fields: latitude (shown on the vertical axis), longitude (horizontal axis) and temperature (the color).

Figures 15.4 and 15.6 show the results: my method versus ordinary kriging. The picture corresponding to kriging covers a larger area, with vast regions without training locations. This is extrapolation rather than interpolation, and the deep blue well and strong red dome are artifacts of the method. They are much less pronounced in my picture (Figure 15.4). Indeed, I had to force my technique to cover an area away from the training set, beyond what is reasonable, to avoid blank (non-interpolated) zones in my image.

Figure 15.5 used a smaller interpolation window: the green “+” are non-interpolated locations due to their distance to the training set. Interpolating so far away from the training set, without additional information on how the real data behaves (heat domes, cold atmospheric depressions) is meaningless and does not generalize to other fields. The 32 training set locations are represented by circular dots in all three pictures. In Figure 15.4, the color of these dots (exact temperature) does not match the color of the background (interpolated value on nearby location) despite the appearance. There are tiny differences, not visible to the naked eye: it proves that the method works! One location in the South East corner has four training set points in very close proximity, with vastly different temperatures (the red dot is almost hidden): there you can see that the interpolation averaged out the four temperatures in question.

For $w_k(x, y)$, I used the function defined by (15.10), with $\alpha = 1, \beta = 2$. I did not make any efforts to find ideal parameter values, as this would defeat the purpose of designing a generic algorithm that works in many settings with as little fine-tuning as possible. For the same reason, the parameter κ in section 15.3.1 is

set to 2, and δ is automatically computed as the smallest value that guarantees all the tested locations can be interpolated.

$$w_k(x, y) = \left(|x - x_k|^\beta + |y - y_k|^\beta \right)^\alpha, \text{ with } \alpha, \beta > 0. \quad (15.10)$$

The parameters α, β control the smoothness of the interpolated function. Choosing a small value for δ amounts to using a nearest neighbor type of interpolation. Choosing a high value for κ amounts to performing kriging. Thus the method is eclectic and encompasses various types of interpolation. The Python implementation follows best practices: the data is first normalized before interpolation, divisions by zero are properly handled, and you can choose not to interpolate at locations too far away from the training set by adjusting δ .

As seen in Figure 15.5, the Python code also generates 4 copies of the training set; the number of copies is specified by the variable `ppo` in the code. In each copy, the location of each point is uniformly distributed in a circle around the original training set location that it represents. The radius of that circle is determined by the variable `radius` in the code. This `synthetic data` is used to test the performance of the algorithm. It allows you to play with different values of the radius. The final line of code computes the average distance (temperature discrepancy) between exact values in the training set and the associated values in the synthetic data, interpolated at sampled locations.

15.3.3 Python code

The Python code `interp.py` is also on my GitHub repository, [here](#). The functions and parameters are described in sections 15.3.1 and 15.3.2. The main function, performing interpolation on a 2-dimensional grid applied to temperatures in the Chicago area, is rather simple. The data is stored into the `data` array, mapped to the `npdata` Numpy array. It is then mapped onto a grid, represented by the `zgrid` array. The grid is used only to produce contour plots.

Four copies of the training set are generated (using `ppo=4`). They can be viewed as four synthetized versions of the training set, with locations and temperatures distributed just like in the original training set. The synthetized locations are stored in the arrays `xa` and `ya` (latitude and longitude); the synthetized temperatures obtained by interpolation are stored in the array `za`.

Interpolated values computed on locations identical to a training set location are exact, by design. Note that before interpolating, the data is transformed: it is normalized to have zero mean and unit variance, a standard practice. It is de-normalized at the end to produce the contour plots. Each interpolated value is computed using a variable number of nodes. That number depends on how many nodes are close enough to the target location. A node is a location in the training set with known temperature.

The number of nodes, for each synthetized location, is stored in the `npt` array. Using the default parameter value for `alpha` guarantees that there is always at least one node (the nearest neighbor) to compute the interpolated value. This can lead to meaningless interpolated values for locations far away from the training set. Reducing the default `alpha` results in some non-interpolated values marked as `Nan`, and it is actually recommended. The un-computed values show up as a green “+” in Figure 15.5.

Finally, the `interpolate` function accepts locations `x, y` that are either a single location or an array of locations. Accordingly, the returned value `z` – the temperature – can be a single value or an array. The `audit` parameter is used internally for testing and monitoring purposes.

15.4 Math-free, parameter-free gradient descent

I discuss techniques related to the gradient descent method in 2D. The goal is to find the minima of a target function, called the cost function. The values of the function are computed at evenly spaced locations on a grid and stored in memory. Because of this, the approach is not directly based on derivatives, and there is no calculus involved. It implicitly uses discrete derivatives, but foremost, it is a simple geometric algorithm. The learning parameter typically attached to gradient descend is explicitly specified here: it is equal to the granularity of the mesh and does not need fine-tuning. In addition to gradient descent and ascent, I also show how to build contour lines and orthogonal trajectories, with the exact same algorithm.

The method generalizes to any dimension if you don't store the whole grid in memory. In higher dimensions, use a small moving window in the feature space rather than the static, full grid. The window is a small neighborhood around the current location in the gradient descent. It is updated at each iteration. The methods also work on raw datasets, without any math function to minimize. In the end, math functions are discretized; the ones investigated in this section are as chaotic as business datasets.

Here, I apply the method to investigate the Riemann Hypothesis. The related functions are defined on the complex plane. However, no advanced knowledge of complex calculus is required as I use the standard 2D space in my illustrations. I focus on a particular type of functions that mimic the Riemann zeta function $\zeta(\sigma, t)$ in the complex plane. For the argument of ζ , I may use the notation $\sigma + it$ or (σ, t) interchangeably. In particular, I am interested in the modulus (also called radius) $|\zeta(\sigma, t)|$.

Orthogonal trajectories (the red curves in Figure 15.7) are more difficult to obtain than contour lines: you typically need to solve differential equations to find them. Yet again, I show how to do it without math. The Python code relies on the Mpmath library, to handle special functions such as ζ in the complex plane.

15.4.1 Introduction

The easiest way to understand the methodology is to look at Figure 15.8. You have a real-valued bivariate function $|\zeta(\sigma + it)|$ with two arguments σ, t , with $0.25 \leq \sigma \leq 1.60$ and $201.0 \leq t \leq 203.3$ respectively on the vertical and horizontal axes. The argument is actually the complex number $\sigma + it$, but here we treat it as a vector (σ, t) . The function ζ is the complex-valued [Riemann zeta function](#) [Wiki], and $|\cdot|$ stands for the [modulus](#) [Wiki]. The modulus of a complex number $\sigma + it$ is defined as $\sqrt{\sigma^2 + t^2}$. In particular, $|\zeta(\sigma, t)|$ is always positive and equal to zero (the minimum potential value) if and only if $(\sigma, t) = \sigma + it$ is a root of ζ .

The dashed line corresponding to $\sigma = \frac{1}{2}$ is called the [critical line](#) [Wiki] in number theory: RH states that all the infinitely many zeros of ζ stand on that line: they must have $\sigma = \frac{1}{2}$. In Figure 15.8, two of them are visible and represented by the green dots. In the rectangular area shown in the picture, $|\zeta(\sigma, t)|$ takes on values between zero (dark blue) and a little over 2.0 (red). For a 3D version, see Figure 15.9. An animated 3D version, with views from different angles, can be found [here](#).

Two types of curves are shown in Figure 15.8:

- [Contour lines](#) [Wiki]: their color ranges from blue to red depending on the value of $|\zeta(\sigma, t)|$ that they represent. In fluid dynamics and electromagnetism, they are called equipotential curves. The color represents the [contour level](#) attached to a specific contour line.
- [Orthogonal trajectories](#) [Wiki]: these are perpendicular to the contour lines. They are called streamlines in fluid dynamics. If the image represented an elevation map, each raindrop hitting a contour line would follow the streamline it is attached to, and move downstream until it hits a minimum: a green dot in our example.

The yellow dots in Figure 15.8 are sampled on a specific but arbitrary contour line. If you pick up a yellow dot, and follow downstream the orthogonal trajectory (in red) that it is attached to, you will end up at one of the two minima of $|\zeta(\sigma + it)|$ in the picture. That's how the [gradient descent method](#) [Wiki] works. The difference with traditional approaches is that here, a finite number of values of $|\zeta(\sigma + it)|$ are computed on a rectangular grid. Thus, the gradient descent is performed on a discretized version of the target function. It does not even require the function to be differentiable, and derivatives are not explicitly computed. This is why I call the method math-free.

If you look closely at Figure 15.8, you will notice that there is small cluster of nearby green dots attached to each of the two minima (the roots of $|\zeta(\sigma + it)|$ in this case). The reason is because each yellow dot – a starting point in the gradient descent – may lead to a different end-point on the grid: the same root maybe, but up to the first few digits. For instance, in my example the precision is about 0.005. Increasing the granularity of the grid fixes the issue. But the price to pay is a lot more memory and time requirements, proportional to the number of cells in the grid. In practice, one can start with a rather coarse grid covering a large rectangular area to get a rough approximation. Then work with a finer grid covering a small area centered around the previously obtained approximation to get a better one, and so on.

My method is particularly useful when you want to see the convergence paths (reaching a green dot) for a large number of initial points (the yellow dots) at once. The bottleneck is computing the values of $|\zeta(\sigma, t)|$ for over 50,000 vectors (σ, t) to populate the grid. It is slow because the series to compute ζ converges slowly especially when $\sigma < 1$ and t is large. Using a product of sine functions even in the complex plane, instead of ζ , is at least 10 times faster. In all cases though, finding all the paths at once is fast. Of course, you can compute just one convergence path and not use the large grid: instead, you would compute the limited data you need about $|\zeta(\sigma, t)|$ on the fly, rather than access it from the 2D arrays `za`, `dx`, and `dy`. These arrays contain a lot more than what you need, and are a waste of time and memory if you are interested in just one path.

15.4.2 Implementation details

Using the parameter names in the Python code, the real-valued function $|\zeta(\sigma, t)|$ is first tabulated on a regular rectangular grid with t between `min_t` and `max_t`, and σ between `min_sigma` and `max_sigma`. Increments for

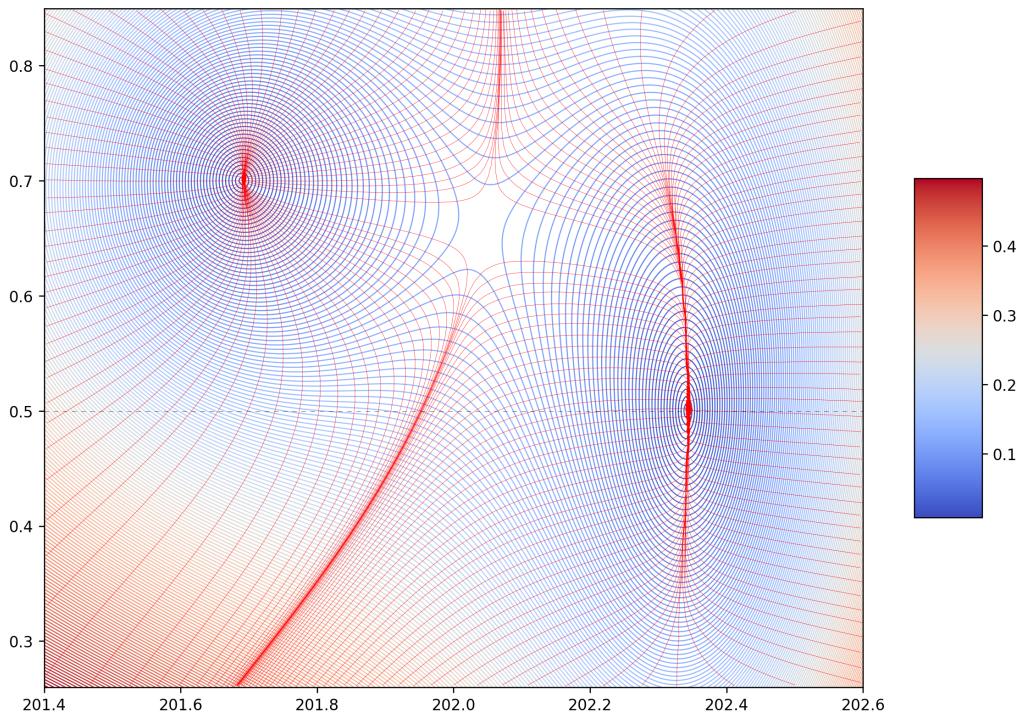
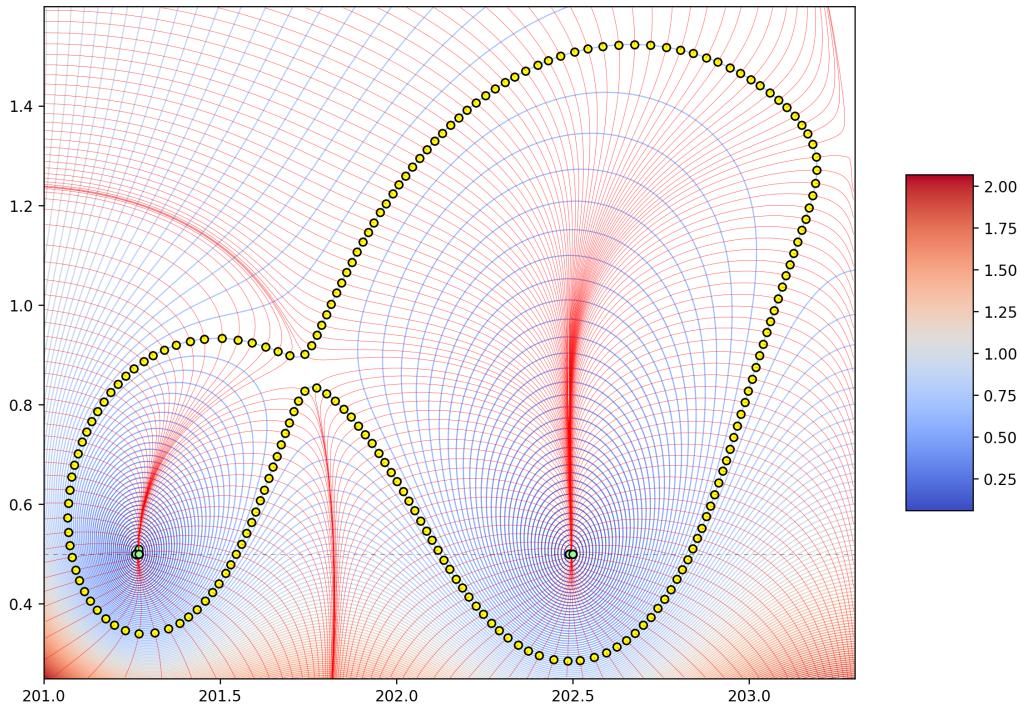


Figure 15.7: Orthogonal trajectories (red) and contour lines for the Sinh2 function

Figure 15.8: Orthogonal trajectories (red) and contour lines for $|\zeta(\sigma, t)|$

t and σ are respectively `incr_t` and `incr_sigma`. The 2D grid is named `za` in the Python code; it maps the evenly distributed vertices (points where the function is tabulated) onto 2D integer coordinates, for easy use in contour plots with Matplotlib. The mapping is performed using the `meshgrid` Numpy function. Also, `za` is a global variable (array).

This step does not involve mathematics. From there, there will be no mathematics involved either. All operations are performed using simple manipulations of the values stored in the grid `za`, without calling the function ζ ever again. Given a location represented by a 2D index $[h, k]$ in `za`, the gradient is the following vector: the first component `dx[h, k]` is the difference between the next and current value on the grid in the horizontal direction; the second component `dy[h, k]` is computed in the vertical direction. This grid-based gradient must be adjusted to represent a real gradient in the actual space: divide `dx[h, k]` by `incr_t`, and `dy[h, k]` by `incr_sigma`. Note that `za[h, k]` is equal to $|\zeta(\sigma, t)|$ with $[h, k]$ representing the location of (σ, t) on the grid.

The grid-based gradient for all grid locations is obtained with the Numpy function `gradient`, with one line of code: `dy, dx=np.gradient(za)`. It is used to plot the orthogonal trajectories. To produce the contour lines (which are orthogonal to the orthogonal trajectories) swap `dx`, `dy` and use `-dx` instead of `dx`. The very core of the Python program resides in the following code snippet

```

1 if mode == 'Descent':
2     sign = +1
3 elif mode == 'Ascent':
4     sign = -1
5 iter = 0
6 while iter < n_iter:
7     if type == 'Gradient':
8         t = t - learn_t * sign * dx[h,k]/incr_t
9         sigma = sigma - learn_sigma * sign * dy[h,k]/incr_sigma
10    elif type == 'Contour':
11        t = t - learn_t * sign * dy[h,k]/incr_sigma
12        sigma = sigma + learn_sigma * sign * dx[h,k]/incr_t
13    if t>min_t and t<max_t and sigma>min_sigma and sigma<max_sigma:
14        x.append(t)
15        y.append(sigma)
16    old_z = za[h, k]
17    h = int(0.5+(sigma - min_sigma)/incr_sigma)
18    k = int(0.5+(t - min_t)/incr_t)
19    if h<h_steps-2 and k<k_steps-2 and h>0 and k>0:
20        z = za[h, k]
21    else:
22        iter = 9999999999
23        showEnd = False
24    iter = iter + 1

```

This piece of code produces the various curves that you see for instance in Figure 15.7. The red and blue curves are obtained respectively with `type='Gradient'` and `type='Contour'`. The former is associated to orthogonal trajectories. Points of the curve under construction are added to the arrays `x` and `y` for plotting purposes. The parameter `mode` specifies in which direction a curve is drawn, given a starting point. For the gradient, Descent means going down towards a minimum, while Ascent means going up. Note that $[h, k]$ represents the location (2D index) of (t, σ) on the grid `za`.

Figure 15.8 shows a number of points – the yellow dots – on a same contour line. Each of these points is attached to one orthogonal trajectory by construction. Given one of these starting points, if you follow the associated orthogonal trajectory downward, you end up to a minimum of the function in question. If you follow it upward, you go up and depending on the starting point, you eventually comes close to a saddle point [Wiki] (the equivalent of a mountain pass), then continue upward following a ridge separating two valleys. You can use my method to find yellow dots all located on a same contour line. However the successive iterations very slightly deviate from the initial contour line over time. Instead, I used a contour line generated by the Matplotlib function `contourf` using the `collections` attribute of the contour object `CS`.

A core parameter in the algorithm is the learning rate [Wiki]. Here I use two of them: one for the horizontal, and one for the vertical direction (`learn_t` and `learn_sigma` in the code). However, unlike in traditional implementations, you don't need to fine-tune them: they are set to the increments in each direction (respectively `incr_t` and `incr_sigma`). This makes my methodology more suitable for black-box implementations.

Finally, another important parameter is the number of iterations `n_iter` in the main algorithm implemented in the `gradient_descent` function. If too low, it will generate the beginning of the path towards the minimum, but not the full path. If too high, the last iterations do not bring additional value and are a waste of time. The smaller `incr_t` and `incr_sigma`, the more accurate the solution. But reducing the increments

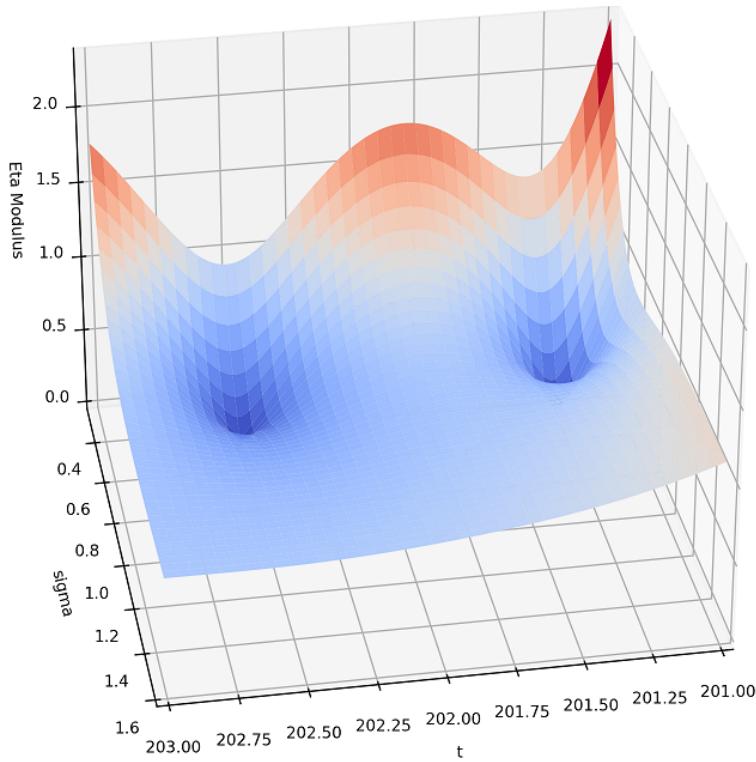


Figure 15.9: Now a 3D view of $|\zeta(\sigma, t)|$, showing the two roots

by a factor 2 increases memory usage (size of arrays `za`, `dx`, `dy`) and time by a factor 4. It also means more iterations needed in the gradient descent. A good compromise is to not be too granular: then the curves won't be perfectly smooth, but you can smooth them out with the parameter `smooth` in the main part of the program.

Further speed improvement is obtained by using a stopping rule. Instead of running a fixed, large number of iterations (the parameter `n_iter`), stop after the maximum number of iterations is reached, or when the error becomes low enough and can not be improved – whichever comes first. Due to the finite size and limited resolution of the grid, you will always reach a point where no further improvement is possible, unless you increase the granularity.

15.4.3 General comments about the methodology and parameters

Despite using the standard notation $(\sigma, t) = \sigma + it$ for the Riemann zeta function ζ , all the figures show σ on the vertical axis, and t on the horizontal axis.

Most of the Python code deals with visualization aspects: the gradient function represents a small portion of the code. In particular, I also generate 3D data animations of the target functions, seen from various angles. You can download the videos [here](#) for $|\zeta(\sigma, t)|$, and [here](#) for the function referred to as `Sinh2` in the code. There is also a video showing how the gradient descent progresses and converges, starting with 100 random locations, again for the `Sinh2` function. You can see it [here](#) on YouTube, or you can download it on GitHub, [here](#). It is based on the code featured in section 15.4.5.

A number of functions, beside $|\zeta(\sigma, t)|$, are available in the code. It includes functions that have a similar behavior, to study the impact of a root, how it propagates vertically when σ increases, and to see if roots can be detected far away from their actual locations. The parameter `Function` lets you choose which function you want to test. For that purpose, you can obtain a broader view of the function, covering a much larger interval for t , by choosing the option `View='Global'`.

It is interesting to note that Python plots or libraries featuring orthogonal trajectories are hard to find, possibly because the mathematical way to compute them is not straightforward: it involves solving a partial differential equation, see section 15.4.4. The closest I found is quiver and stream plots, that indicate the direction of a field with arrows (magnetic, fluid dynamics, atmospheric gradients and so on), computed locally at a number of locations, with the length of the arrow representing the local velocity or intensity. See for instance [here](#).

I now discuss some useful parameters of the method, that I haven't covered yet. The parameter names correspond to those in the Python code featured in section 15.4.5.

- In Figure 15.8, the number of possible starting points (the yellow dots) is automatically computed and denoted as `n`. To make the starting points visible, set `showStart=True`. Likewise, to make the end points (blue dots) visible, set `showEnd=False`. To show the orthogonal trajectories (the red curves), set `showPath=True`.
- The yellow dots are picked up on a contour line generated within the `contour` Matplotlib environment. The contour in question is specified by the parameter `level`. Its value can be anywhere between 1 (the contour with coldest color) and `nlevels` (the contour with the warmest color). The parameter `nlevels` is set by the user. You can choose not to include all the `n` yellow dots found on the contour line, but instead a fraction of them: this is controlled by the parameter `step`.
- If instead you want to start with `n` random points, control `n` and the locations of the yellow dots, use the code in section 15.4.5.
- The option `View='Global'` is not used here, but discussed in chapter 1 in [14].

15.4.4 Mathematical version of gradient descent and orthogonal trajectories

The purpose of this section is to explain the traditional mathematical approach to [gradient descent](#) optimization [[Wiki](#)], also known as [steepest descent](#). In the end, the method described in section 15.4.2 is a discretized version of this technique, although it does not require the target function – in this case $|\zeta(\sigma, t)|$ – to be differentiable. In addition, I also explain how the orthogonal trajectories are typically obtained with a mathematical formulation. Finally, I show how the two methods – no math versus math – are related.

In the traditional math formulation, one starts with a rough estimate or guess for the location of the optimum. In other words, one of the yellow dots in Figure 15.8. Then after successive iterations following downward the red curve attached to the yellow point, you end up at the location of a minimum in case of convergence, here a blue dot. As in Figure 15.8, which location you end up with may depend on your starting location, unless the function is concave and has only one minimum.

One major difference is that you can have an infinite number of iterations and indefinitely increase the accuracy of the solution. In the discrete version, you are limited by the granularity of the grid: to increase accuracy, you need a finer grid. This is why you see a few overlapping blue dots very close to each other in Figure 15.8: we are getting very close to the optimum regardless of where we start from, but to get further improvements (so that the distinct blue dots are indistinguishable to the naked eye), you need a finer grid. In my example, the precision is about 0.005 and determined by the parameters `incr_t` and `incr_sigma`. After getting a good approximation, it is possible to start over again, but this time with a much narrower grid around the optimum, and increased precision.

Now, here is how iteration $n+1$ looks like when trying to solve the problem in mathematical terms, starting with (σ_0, t_0) . To be as general as possible, I replaced $|\zeta(\sigma, t)|$ by an arbitrary function $f(\sigma, t)$.

$$\begin{aligned} t_{n+1} &= t_n - \lambda \frac{\partial f}{\partial t}(\sigma_n, t_n) \\ \sigma_{n+1} &= \sigma_n - \lambda' \frac{\partial f}{\partial \sigma}(\sigma_n, t_n) \end{aligned}$$

where λ, λ' are the learning rate parameters corresponding to `learn_t`, `learn_sigma` in section 15.4.2. The iterated (σ_n, t_n) may or may not converge depending on the learning rates: this is an issue you don't have to face with my discretized version.

One of the features of my method is the computation of contour lines and orthogonal trajectories at once, allowing you to plot the convergence paths for a large number of points, in little time. The time-consuming step is creating the `za` array that contains the values of the function computed at a large number of evenly spaced locations. There is little literature about how to obtain the equations of the orthogonal trajectories, yet the problem is rather elementary. It requires solving a differential equation. See [here](#) for a simple explanation. This tutorial explains why the orthogonal trajectories to a family of concentric circles are straight lines going through the common center. It then shows that for a family of ellipses, the orthogonal trajectories are confocal hyperbolas satisfying a specific parametric equation.

The principle is simple. Each contour line satisfies an equation $f(\sigma, t) = \gamma$ where γ is a parameter called the [contour level](#). Let $g(\sigma, t) = \gamma$ define an orthogonal trajectory. The orthogonality between the two means that $\nabla^T f(\sigma, t) \cdot \nabla g(\sigma, t) = 0$, where ∇ is the gradient operator, T is the transpose operator turning a column into a row vector, and the product is a dot product [[Wiki](#)] between vectors. Thus, you need to solve the following differential equation, where g is the unknown:

$$\frac{\partial f}{\partial t} \frac{\partial g}{\partial t} + \frac{\partial f}{\partial \sigma} \frac{\partial g}{\partial \sigma} = 0.$$

The solution depends on a parameter, in the same way that a contour line depends on the contour level. The way it is implemented in the discretized version is as follows. The vector $(dx, dy)^T$ is the grid-based gradient of f , used to produce the orthogonal trajectories. Then $(dy, -dx)^T$ is the grid-base gradient of g , used to produce the contour lines. The dot product is $dx \cdot dy - dy \cdot dx = 0$. For the actual gradient, divide dx and dy respectively by dt and $d\sigma$. In the Python code, dt and $d\sigma$ are represented by the variables `incr_t` and by `incr_sigma`. Also, dx and dy are represented by `dx` and `dy`. This leaves the dot product unchanged and equal to zero.

A benefit of my method is that it is easy to implement even if you don't have a function f to begin with, but instead data points with a measurement at each location: in short, a real-life dataset.

15.4.5 Python code

The main code (on GitHub, not listed here) generates the figures and videos discussed here. The starting points to illustrate convergence paths are all sampled on a contour line. If instead you are interested in starting points at arbitrary, random locations, you need to replace the main section of the program by the code snippet listed below. The parameters are described in section 15.4.2 and 15.4.3. The main Python code `gradient.py` is on my GitHub repository, [here](#).

I also produced a data animation, featuring gradient descent in progress starting with 100 random points. You can view the video on YouTube, [here](#). For the code, use the main program and replace the section "Steepest descent: main part" by the code snippet below. It is also available on GitHub [here](#).

```

1  #--- Steepest descent: main part
2
3  # import moviepy.video.io.ImageSequenceClip
4
5  dy, dx = np.gradient(za) # matrices with same dim as za
6  smooth = 0              # integer, to smooth trajectories (0 = no smoothing)
7  learn_t = incr_t        # learning parameter in gradient method
8  learn_sigma = incr_sigma # learning parameter in gradient method
9
10 n = 100
11 np.random.seed(101)
12 xx = np.random.uniform(min_t,max_t,n)
13 yy = np.random.uniform(min_sigma,max_sigma,n)
14 showEnd = False
15 showPath = True
16 n_iter = 1
17 flist = []
18 fps = 4
19
20 for frame in range(200):
21
22     print("frame",frame)
23     image='RH4_ortho'+str(frame)+'.png'
24     if frame == 0:
25         showStart = True
26     else:
27         showStart = False
28
29     for i in range(0, n):
30         t = xx[i]
31         sigma = yy[i]
32         (x, y) = gradient_descent(t, sigma, showStart, showEnd, showPath, 'Descent', n_iter,
33             learn_t, learn_sigma, 'Gradient')
34         m = len(x)
35         xx[i] = x[m-1] # new t attached to starting point i
36         yy[i] = y[m-1] # new sigma attached to starting point i
37         plt.savefig(image,bbox_inches='tight')
38         flist.append(image)
39
40 # output video
41 clip = moviepy.video.io.ImageSequenceClip.ImageSequenceClip(flist, fps=fps)
42 clip.write_videofile('RH4_ortho.mp4')
```

Chapter 16

Trading the S&P 500 Index

Before telling what this chapter is about, I want to discuss Figure 16.1. It features problems found in almost all AI systems. The solutions are relevant to many contexts well beyond Fintech. Each dot in the scatterplots represents the performance of an algorithm based on two parameters: β (X-axis) and σ (Y-axis). There is an extra parameter ω , with two values tested here: $\omega = 40$ on the left, and $\omega = 60$ on the right. The framework is highly non-linear, and indeed quite chaotic.

The color indicates the performance level: blue for great, green for good, gray for similar to baseline, orange for poor, and red for bad. The goal is to find decent parameter vectors (β, σ, ω) in the parameter space, based on the 128 tested configurations: 64 on the left, and another 64 on the right. Each test consists of running an algorithm on 12 different large portions of the input dataset, itself consisting of 40 years' worth of daily stock market data. In short, each performance measurement (color) is averaged over 12 different input datasets. The approach is similar to using [grid search](#) to optimize hyperparameters in a neural network. But there is more to it, applicable to the core of almost all neural networks and machine learning techniques: [gradient descent](#).

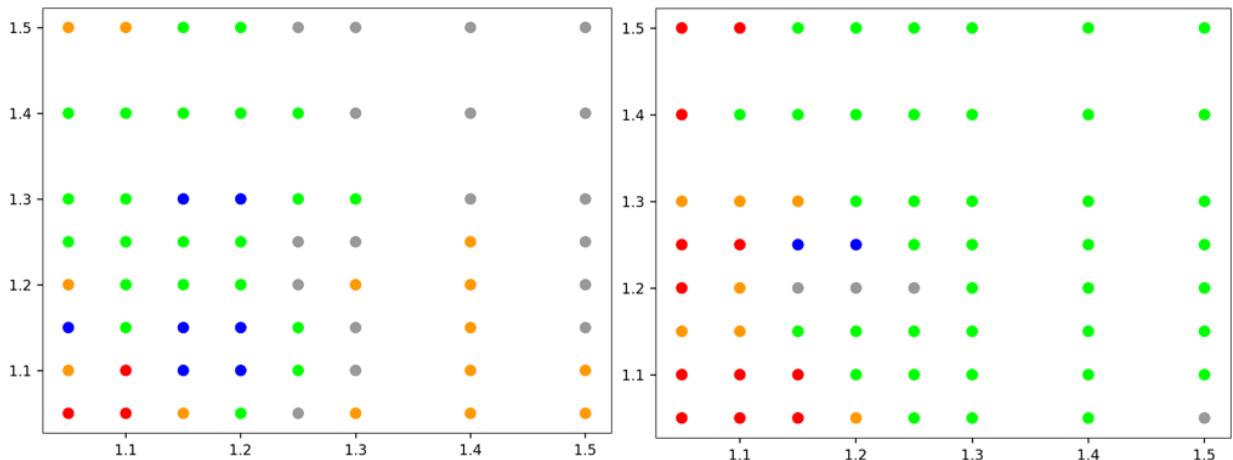


Figure 16.1: Blue/green outperform baseline, orange/red underperform (left: $\omega = 40$, right: $\omega = 60$)

Interesting conclusions:

- There are areas with relatively stable and good performance (green) in the parameter space. There may be many of them, separated by rather large regions of medium or low performance. For gradient descent, as long as it lands in one of them, the results will be good. However, progress within a green zone can be very slow, as the gradient approaches zero. You can stop the descent once deep enough inside a green zone. See how gradient descent works on datasets like this one, without [loss function](#), in section 15.4. An alternative method to find the optimum is [smart grid search](#).
- You want to find a large green region with good or medium performance, rather than a small one with good or great performance. Also, the boundary of these regions can be very unstable, thus the reason to avoid small ones. Note that large green regions may have red zones inside. Do not get too close, these red zones are like black holes. Some of the best parameters (the blue dots) are very close to some of the worst (red dots). Avoid these blue dots to reduce [overfitting](#). This is typical in many problems, where

green zones are **basins of attraction** in the underlying chaotic **dynamical system** acting behind the scenes, surrounded by basins of repulsion. For more details, see my book [9] on this topic.

- The somewhat chaotic boundaries of the various regions can be detected using the **interpolation** algorithm described in section 15.3. This algorithm generalizes to higher dimensions.

Now I focus on the problem addressed in this chapter: designing S&P 500 trading strategies that outperform the baseline (staying long the entire period). The idea consists of exiting (selling) and re-entering (buying) at the right times, including for the first entry, over a long time period, ranging from 3 to 10 years. Depending on the parameters β, σ, ω to be explained later, the number of exits ranges from one or two, to a dozen. At the end of the pre-specified time period, two scenarios are possible: either you have already exited, or you still have an open position. The latter is called *holding*. The right times to exit or re-enter are determined by price variations in the index, your accumulated ROI, the price at your last exit, the time elapsed since your last exit, and the new price. It is well-known that this index, while going up on average, experiences sharp drops – sometimes prolonged – and rebounds that can be leveraged.

16.1 Non-standard strategies

It is difficult to successfully arbitrage the stock market due to the large number of participants competing against you. Staying long on the S&P 500 index is one of the most effective strategies, even outperforming many if not most professional traders, in the long run. In order to do better while minimizing competition, you need to use strategies that Wall Street professionals must avoid. For instance, keeping cash for extended periods of time (sometimes for years in a row) to be able to jump in at the right time with no advance notice – after a massive crash – then buy and sell during a short window following the crash to leverage the resulting volatility, to finally have a stable long position acquired at a steeply discounted price. You then sell the position in question years or months later when its value has massively increased following the slow or fast rebound. Then you repeat the cycle.

The strategies discussed here are inspired by the above principle. Each strategy has a specific parameter vector (β, σ, ω) and its own time frame, starting at t_0 and ending at t_f . The first buy takes place at a time t'_0 in that window, typically with $t'_0 > t_0$, and the last sell at a time t'_f either within that window, or later on if you must still hold your position at time t_f . The annualized compound return is compared to the baseline: buying at t_0 , selling at t_f . If you still hold your position at t_f , the return on your strategy is based on the value of your position at t_f . The starting point to compute your return is t_0 , not t'_0 .

If you are all cash, you buy when the price reaches $\beta \cdot p_{\min}(\omega)$, where $p_{\min}(\omega)$ is the bottom price observed in the last w days preceding the purchase. If holding (active position), you sell when the price hits the $\sigma \cdot p_{\text{buy}}$ threshold, where p_{buy} is the price you paid per unit in your last purchase. Much of the work consists of testing various (β, σ, ω) that works well across various time frames $[t_0, t_f]$, and that are stable. That is, not sensitive to small variations. Now that everything is in place, I make the link to the names used in the Python code in section 16.3.

- In the code, $t_0, t_f, \beta, \sigma, \omega$ are respectively named `init`, `end`, `buy_param`, `sell_param`, `windowLow`, and stored in the `params` dictionary. See lines 121–125 in the code. One extra parameter `rnd` is discussed later. For t'_0 and t'_f , the corresponding notations are `entry_idate` and `exit_idate`. The prices p_{buy} and p_{\min} are respectively denoted as `buy_price` and `bottom[params]`. Finally, `params` is the key to many key-value tables.
- I tested 4 values for t_0 and 3 for t_f . Thus, for each strategy (β, σ, ω) , the summary statistics are based on $4 \times 3 = 12$ observations. An observation is the result of trading the entire time period $[t_0, t_f]$. See lines 47–48 in the code. Note that instead of specifying t_f , I use `duration`, with $t_f = t_0 + \text{duration}$.
- I tested 8 values for β , 8 for σ , and 2 for ω , leading to $8 \times 8 \times 2 = 128$ strategies, each with 12 observations based on the 12 combinations of t_0, t_f . See lines 49–51 in the code. Thus, the output dataset (see [here](#)) has 128 rows: one per strategy, each featuring averages based on 12 measurements.
- Each daily price is a value between the low and the high of the day, specified by the fixed weight `rnd=0.50`. See line 128 in the code. You should try with different values of `rnd`, to rule out strategies too sensitive to little variations.

The code is somewhat complicated because of the pre-processing step to accelerate many computations. The bottleneck is calculating the minimum price in the moving window consisting of the last w days. It is done once in lines 58–61, rather than for each (β, σ, ω) , speeding up the computations by a factor 128.

Other important metrics attached to a strategy are the number `nbuys` of “buy” trades, and among those trades, the `wins`, that is, when buying at a lower price than the previous sell. See lines 81 and 84 in the code. The hold rate is the chance to still be in a holding position at the end of the time period. In the results, all these numbers are averaged per strategy (β, σ, ω) with the strategy average computed over the 12 time series of trading activity (one for each $[t_0, t_f]$). For a strategy, the success rate is defined as the chance to outperform the “buy and hold” baseline.

16.2 Selecting a strategy based on its features

Figure 16.2 shows the summary results for the top 30 strategies, out of 128. The annualized ROI gain over the baseline ranges from 0.82% to -1.71%. Here the baseline is a 4.97% ROI. The average number of “buys” per strategy ranges from barely above 1 to nearly 11. Among those buys, usually fewer than 50% are “wins”, meaning buying at a lower price than the previous sell. Despite this seemingly low performance, the overall success rate (outperforming baseline) per strategy may be over 50% due to the fact that a few of the wins are spectacular, taking place after a massive price drop.

β	σ	ω	buys	wins	delta return	base return	strategy return	success rate	hold rate
1.20	1.10	40	6.25	3.00	0.82%	4.97%	5.79%	67%	83%
1.05	1.15	40	4.17	2.17	0.71%	4.97%	5.68%	42%	67%
1.15	1.25	60	2.75	1.67	0.62%	4.97%	5.59%	58%	83%
1.15	1.10	40	6.00	2.75	0.58%	4.97%	5.55%	50%	83%
1.15	1.30	40	2.50	1.33	0.57%	4.97%	5.53%	75%	83%
1.20	1.25	60	2.75	1.75	0.56%	4.97%	5.53%	58%	83%
1.15	1.15	40	4.42	2.00	0.55%	4.97%	5.52%	67%	83%
1.20	1.15	40	4.42	1.92	0.51%	4.97%	5.48%	67%	92%
1.20	1.30	40	2.58	1.00	0.51%	4.97%	5.48%	58%	100%
1.25	1.25	60	2.83	1.25	0.49%	4.97%	5.46%	67%	92%
1.30	1.10	60	6.08	2.42	0.48%	4.97%	5.45%	50%	83%
1.25	1.15	60	4.33	1.83	0.47%	4.97%	5.44%	58%	83%
1.30	1.30	60	2.33	0.83	0.47%	4.97%	5.44%	50%	75%
1.30	1.15	60	4.33	1.42	0.45%	4.97%	5.42%	50%	92%
1.20	1.25	40	2.83	1.00	0.42%	4.97%	5.39%	42%	100%
1.20	1.10	60	5.75	3.17	0.42%	4.97%	5.39%	58%	50%
1.20	1.30	60	2.33	1.08	0.42%	4.97%	5.39%	67%	75%
1.20	1.05	40	11.08	6.50	0.38%	4.97%	5.35%	42%	83%
1.10	1.30	40	2.42	1.08	0.38%	4.97%	5.35%	58%	83%
1.25	1.30	60	2.33	0.92	0.38%	4.97%	5.35%	50%	75%
1.10	1.15	40	4.17	2.08	0.38%	4.97%	5.35%	25%	75%
1.15	1.25	40	2.67	0.92	0.37%	4.97%	5.34%	33%	83%
1.25	1.05	60	10.58	4.25	0.34%	4.97%	5.31%	42%	50%
1.15	1.50	40	1.92	0.83	0.33%	4.97%	5.30%	50%	100%
1.40	1.25	60	2.75	1.17	0.31%	4.97%	5.28%	58%	92%
1.30	1.25	60	2.75	1.17	0.31%	4.97%	5.28%	58%	92%
1.15	1.40	40	2.17	0.83	0.29%	4.97%	5.26%	67%	92%
1.20	1.40	40	2.17	0.83	0.29%	4.97%	5.26%	67%	92%
1.50	1.25	60	2.75	1.00	0.27%	4.97%	5.24%	58%	92%
1.25	1.30	40	2.50	0.92	0.27%	4.97%	5.24%	42%	100%
1.30	1.05	60	10.83	4.42	0.23%	4.97%	5.20%	50%	67%
1.20	1.50	40	1.92	1.00	0.22%	4.97%	5.19%	50%	100%
1.40	1.40	60	2.17	0.83	0.22%	4.97%	5.19%	58%	92%

Figure 16.2: Average stats for each strategy (β, σ, ω)

Not surprisingly, the chances to finish with an active position (holding) is rather high due to the small volume of trading activity, favoring long over short-term holding positions. See rightmost column in Figure 16.2. It would be interesting to see what proportion of time is spent on holding a position, versus being all cash.

How to choose a strategy, that is, β, σ and ω ? A large number of “buys” with 40% or more that are “wins” is an indication of robustness. Another benefit is that you are more frequently in a cash position. This can be useful if you face some sudden emergencies. However, frequent trades may be subject to short term capital gains. You also want to avoid good strategies (green or blue dot) with parameters β, σ, ω too close to a red dot in Figure 16.1.

Finally, most of the tests were performed on the last 20 years of historical data. Earlier patterns are slightly different and more erratic. Also, you can play with multiple strategies in parallel. The strategies can be used

to identify entry and exit points in the S&P 500 index. It would be interesting the break down performance based on the length $t_f - t_0$ of the time window

16.3 Python code and dataset: 40 years' worth of historical data

The Python code spx500.py is on GitHub in my “Statistical Optimization” repository: click [here](#) to access it. The input dataset spx500.txt and output results spx500-results.xlsx are in the same repository, respectively [here](#) and [here](#). The input data comes from Yahoo Finance: you can also download it from [here](#), and also access data for individual stocks.

```

1 import numpy as np
2 import pandas as pd
3 from datetime import datetime
4 import matplotlib.pyplot as plt
5 import matplotlib as mpl
6
7
8 #--- [1] Read data
9
10 data = pd.read_csv("spx500.txt")
11
12 def date_to_int(date, start_date):
13     xstart_date = datetime.strptime(start_date, '%b %d %Y')
14     xdate = datetime.strptime(date, '%b %d %Y')
15     date_int = str(xdate - xstart_date)
16     date_int = date_int.split(' ')[0]
17     if date_int == '0:00:00':
18         date_int = 0
19     else:
20         date_int = int(date_int)
21     return(date_int)
22
23 arr_date = data['Date']
24 arr_low = data['Low']
25 arr_high = data['High']
26 arr_open = data['Open']
27 arr_idate = []
28 arr_tdate = []
29 nobs = len(arr_date)
30
31 for k in range(nobs):
32     arr_idate.append(date_to_int(arr_date[k], arr_date[0]))
33     tdate = datetime.strptime(arr_date[k], '%b %d %Y')
34     arr_tdate.append(tdate)
35
36 data.insert(1, 'iDate', arr_idate, True)
37
38 print(data.head())
39 print(data.columns.values)
40 print(data.shape)
41 # features: 'Date','iDate','Open','High','Low','Close','Adj_Close','Volume'
42
43 #--- [2] Initializations for main loop
44
45 h_params = {}
46
47 l_init = (4000, 5000, 6000, 7000)
48 l_duration = (1000, 2000, 3000)
49 l_windowLow = (40, 60)
50 l_buy_param = (1.05, 1.1, 1.15, 1.2, 1.25, 1.3, 1.4, 1.5)
51 l_sell_param = (1.05, 1.1, 1.15, 1.2, 1.25, 1.3, 1.4, 1.5)
52
53 h_max = {}
54 h_min = {}
55
56 #- [2.1] preprocessing to speed up computations
57
58 for k in range(nobs):
59     for windowLow in l_windowLow:
60         if k >= windowLow:
61             h_min[(k, windowLow)] = min(arr_low[k-windowLow:k-1])
62
63 #- [2.2] create table of parameter sets (stored as key in h_params)
```

```

64
65     for buy_param in l_buy_param:
66         for sell_param in l_sell_param:
67             for windowLow in l_windowLow:
68                 for init in l_init:
69                     for duration in l_duration:
70                         end = init + duration
71                         key = (init, end, buy_param, sell_param, windowLow)
72                         h_params[key] = 1
73
74 #-- [2.3] Create main hash tables
75 #      The index (same in all tables) is the parameter set
76
77 bottom    = {}
78 hold      = {} # true if we have open position
79 value     = {}
80 max_hold  = {}
81 nbuys     = {} # number buys done during whole period
82 entry_price = {} # buy price on first buy
83 sell_price = {} # sell price on last sell, before current buy
84 wins      = {} # a win is buying a lower price than last sell
85
86 init_price = {} # price when trading period starts
87 buy_price  = {} # buy price
88 end_price  = {} # price when trading period ends
89 entry_idate = {} # date of first buy
90 exit_idate = {} # date of last sell if out, otherwise end_idate
91 init_idate = {} # date when trading period starts
92 end_idate = {} # date when trading period ends
93 start     = {}
94
95 for params in h_params:
96
97     bottom[params] = arr_high[0]
98     hold[params]   = False
99     value[params]  = 0
100    max_hold[params] = 0
101    nbuys[params] = 0
102    entry_price[params] = -1
103    sell_price[params] = -1
104    wins[params] = 0
105
106
107 #--- [3] Main loop
108
109 for k in range(nobs):
110     # loop over all trading days
111
112     idate = arr_idate[k]
113     price_high = arr_high[k]
114     price_low = arr_low[k]
115     if k % 100 == 0:
116         print(arr_date[k])
117
118     for params in h_params:
119         # update trading stats for each param set in parallel
120
121         init    = params[0]
122         end     = params[1]
123         buy_param = params[2]
124         sell_param = params[3]
125         windowLow = params[4]
126         rnd     = 0.5
127
128         price = rnd * price_high + (1-rnd) * price_low
129
130         if k == init:
131             init_price[params] = price
132             init_idate[params] = idate
133         elif k == end:
134             end_price[params] = price
135             end_idate[params] = idate
136
137         if k >= windowLow and price < h_min[(k, windowLow)]:
138             bottom[params] = price
139

```

```

140     if k >= init and not hold[params] and k < end:
141         if price < buy_param * bottom[params]:
142             # buy
143             buy_price[params] = price
144             if value[params] == 0:
145                 # first purchase
146                 value[params] = price
147                 entry_price[params] = price
148                 entry_idate[params] = idate
149                 if price < init_price[params]:
150                     # first buy at lower price than init price
151                     wins[params] += 1
152                 elif buy_price[params] < sell_price[params]:
153                     wins[params] += 1
154                 start[params] = idate
155                 hold[params] = True
156                 nbuys[params] += 1
157
158             elif hold[params] and k < end:
159                 span = idate-start[params]
160                 if span > max_hold[params]:
161                     max_hold[params] = span
162                 if price > sell_param * buy_price[params]:
163                     # sell
164                     sell_price[params] = price
165                     value[params] *= price/buy_price[params]
166                     hold[params]= False
167                     exit_idate[params] = idate
168
169
170 #--- [4] Summary stats for each param set
171
172 # group params in h_params by (init, end)
173 # stored average stats by grouped params, in arr_local
174
175 hash_performance = {}
176 hash_count = {}
177
178 for params in h_params:
179     if hold[params]:
180         value[params] *= end_price[params]/buy_price[params]
181         exit_idate[params] = end_idate[params]
182     else:
183         if end_price[params] < sell_price[params]:
184             wins[params] += 1
185         duration1 = end_idate[params] - init_idate[params] + 1
186         duration2 = exit_idate[params] - entry_idate[params] + 1
187         R_market = end_price[params] / init_price[params]
188         R_strategy = value[params] / entry_price[params]
189         adj_R_market = 100*(R_market**((365/duration1) - 1))
190         adj_R_strategy = 100*(R_strategy**((365/duration1) - 1))
191         ratio = R_strategy / R_market # reinvest all to compound return
192         performance = adj_R_strategy - adj_R_market
193         if performance > 0:
194             success = 1
195         else:
196             success = 0
197
198     key = (params[2], params[3], params[4])
199     arr_local = [nbuys[params], wins[params],
200                  performance, adj_R_market, adj_R_strategy,
201                  duration1, duration2, success, hold[params]]
202
203     if key in hash_performance:
204         arr_local2 = hash_performance[key]
205         for idx in range(len(arr_local2)):
206             arr_local2[idx] += arr_local[idx]
207         hash_performance[key] = arr_local2
208         hash_count[key] += 1
209     else:
210         hash_performance[key] = arr_local
211         hash_count[key] = 1
212
213 #- [4.1] to get averages for each param set, divide sums by sample size cnt
214
215 cnt = hash_count[key]

```

```

216
217     for key in hash_performance:
218         arr_avg = hash_performance[key]
219         for idx in range(len(arr_avg)):
220             arr_avg[idx] /= cnt
221
222
223 #--- [5] Print results
224
225 OUT = open("spx500-results2.txt", "w")
226
227 labels="beta\tsigma\tomega\tsample size\tbuys\tbuys below last sell\tDelta return"
228 labels+="base return\ttrading return\tperiod (days)\tactive peiod\tsuccess rate\thold rate"
229 OUT.write(labels + "\n")
230
231 for key in hash_count:
232     strout = ""
233     for param in key:
234         strout += str(param) + "\t"
235     cnt = hash_count[key]
236     strout += str(cnt) + "\t"
237     arr_avg = hash_performance[key]
238     for idx in range(len(arr_avg)):
239         strout += str(arr_avg[idx]) + "\t"
240     strout += "\n"
241     OUT.write(strout)
242
243 OUT.close()
244
245
246 #--- [6] Visualizations
247
248 hash_xy = {}
249 hash_color = {}
250 for value in l_windowLow:
251     hash_xy[value] = []
252     hash_color[value] = []
253
254 for key in hash_count:
255     cnt = hash_count[key]
256     arr_avg = hash_performance[key]
257     for params in key:
258         buy_param = params[0]
259         sell_param = params[1]
260         window_low = params[2]
261         performance = arr_avg[2] # performance
262         if performance > 0.5:
263             color = [0, 0, 1] # blue
264         elif performance > 0.1:
265             color = [0, 1, 0] # lightgreen
266         elif performance > -0.1:
267             color = [0.6, 0.6, 0.6] # lightgray
268         elif performance > -0.5:
269             color = [1, 0.6, 0] # orange
270         else:
271             color = [1, 0, 0] # red
272         local_arr = hash_xy[window_low]
273         local_arr.append([buy_param, sell_param])
274         local_arr = hash_color[window_low]
275         local_arr.append(color)
276
277 mpl.rcParams['axes.linewidth'] = 0.5
278 plt.rcParams['xtick.labelsize'] = 9
279 plt.rcParams['ytick.labelsize'] = 9
280
281 for window_low in hash_xy:
282     z = np.array(hash_xy[window_low])
283     z = np.transpose(z)
284     color = hash_color[window_low]
285     plt.scatter(z[0], z[1], c = color)
286     plt.show()

```

Bibliography

- [1] Fabiola Banfi, Greta Cazzaniga, and Carlo De Michele. Nonparametric extrapolation of extreme quantiles: a comparison study. *Stochastic Environmental Research and Risk Assessment*, 36:1579–1596, 2022. [\[Link\]](#). 92, 168
- [2] Marc G. Bellemare et al. The Cramer distance as a solution to biased Wasserstein gradients. *Preprint*, pages 1–20, 2017. arXiv:1705.10743 [\[Link\]](#). 96
- [3] Iulia Brezeanu. How to cut RAG costs by 80% using prompt compression. *Blog post*, 2024. TowardsData-Science [\[Link\]](#). 33, 145
- [4] Wei Chen and Mark Fuge. Synthesizing designs with interpart dependencies using hierarchical generative adversarial networks. *Journal of Mechanical Design*, 141:1–11, 2019. [\[Link\]](#). 91
- [5] Johnathan Chiu, Andi Gu, and Matt Zhou. Variable length embeddings. *Preprint*, pages 1–12, 2023. arXiv:2305.09967 [\[Link\]](#). 33, 144
- [6] Fabian Gloeckle et al. Better & faster large language models via multi-token prediction. *Preprint*, pages 1–29, 2024. arXiv:2404.19737 [\[Link\]](#). 33, 67
- [7] Vincent Granville. Generative AI: Synthetic data vendor comparison and benchmarking best practices. *Preprint*, pages 1–13, 2023. MLTechniques.com [\[Link\]](#). 88, 96
- [8] Vincent Granville. Generative AI technology break-through: Spectacular performance of new synthesizer. *Preprint*, pages 1–16, 2023. MLTechniques.com [\[Link\]](#). 79, 82
- [9] Vincent Granville. *Gentle Introduction To Chaotic Dynamical Systems*. MLTechniques.com, 2023. [\[Link\]](#). 152, 154, 155, 158, 193
- [10] Vincent Granville. How to fix a failing generative adversarial network. *Preprint*, pages 1–10, 2023. ML-Techniques.com [\[Link\]](#). 81
- [11] Vincent Granville. Massively speed-up your learning algorithm, with stochastic thinning. *Preprint*, pages 1–13, 2023. MLTechniques.com [\[Link\]](#). 80
- [12] Vincent Granville. Smart grid search for faster hyperparameter tuning. *Preprint*, pages 1–8, 2023. ML-Techniques.com [\[Link\]](#). 49, 80, 112
- [13] Vincent Granville. *State of the Art in GenAI & LLMs – Creative Projects, with Solutions*. MLTechniques.com, 2024. [\[Link\]](#). 68, 70, 108, 111, 125, 133, 144, 146, 173, 175
- [14] Vincent Granville. *Statistical Optimization for AI and Machine Learning*. MLTechniques.com, 2024. [\[Link\]](#). 56, 91, 94, 109, 111, 112, 143, 145, 170, 173, 175, 182, 190
- [15] Vincent Granville. *Synthetic Data and Generative AI*. Elsevier, 2024. [\[Link\]](#). 91, 143, 175
- [16] Vincent Granville, Mirko Krivanek, and Jean-Paul Rasson. Simulated annealing: A proof of convergence. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 16:652–656, 1996. 98
- [17] Radim Halíř and Jan Flusser. Numerically stable direct least squares fitting of ellipses. *Preprint*, pages 1–8, 1998. [\[Link\]](#). 178, 181
- [18] Christian Hill. *Learning Scientific Programming with Python*. Cambridge University Press, 2016. [\[Link\]](#). 181
- [19] Albert Jiang et al. Mixtral of experts. *Preprint*, pages 1–13, 2024. arXiv:2401.04088 [\[Link\]](#). 33
- [20] L. Kuipers and H. Niederreiter. *Uniform Distribution of Sequences*. Dover, 2012. [\[Link\]](#). 157
- [21] Jogendra Nath Kundu et al. GAN-Tree: An incrementally learned hierarchical generative framework for multi-modal data distributions. *IEEE/CVF International Conference on Computer Vision*, pages 8190–8199, 2019. arXiv:1908.03919 [\[Link\]](#). 91
- [22] Nicolas Langrené and Xavier Warin. Fast multivariate empirical cumulative distribution function with connection to kernel density estimation. *Computational Statistics & Data Analysis*, 162:1–16, 2021. [\[Link\]](#). 97

- [23] Gary R. Lawlor. A l'Hospital's rule for multivariable functions. *Preprint*, pages 1–13, 2013. arXiv:1209.0363 [Link]. 182
- [24] Tengyuan Liang. Estimating certain integral probability metric (IPM) is as hard as estimating under the IPM. *Preprint*, pages 1–15, 2019. arXiv:1911.00730 [Link]. 97
- [25] Andrei Lopatenko. Evaluating LLMs and LLM systems: Pragmatic approach. *Blog post*, 2024. [Link]. 33
- [26] Sebastián Maldonado et al. An adaptive loss function for deep learning using OWA operators. *Preprint*, pages 1–15, 2023. arXiv:2305.19443 [Link]. 108
- [27] Hussein Mozannar et al. The RealHumanEval: Evaluating Large Language Models' abilities to support programmers. *Preprint*, pages 1–34, 2024. arXiv:2404.02806 [Link]. 33
- [28] Michael Naaman. On the tight constant in the multivariate Dvoretzky–Kiefer–Wolfowitz inequality. *Statistics & Probability Letters*, 173:1–8, 2021. [Link]. 97
- [29] Sergey Shchegrikovich. How do you create your own LLM and win The Open LLM Leaderboard with one Yaml file? *Blog post*, 2024. [Link]. 33
- [30] Bharath Sriperumbudur et al. On the empirical estimation of integral probability metrics. *Electronic Journal of Statistics*, pages 1550–1599, 2012. [Link]. 97
- [31] 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]. 91
- [32] Eyal Trabelsi. Comprehensive guide to approximate nearest neighbors algorithms. *Blog post*, 2020. TowardsDataScience [Link]. 146
- [33] D. Umbach and K.N. Jones. A few methods for fitting circles to data. *IEEE Transactions on Instrumentation and Measurement*, 52(6):1881–1885, 2003. [Link]. 179
- [34] D. A. Vaccari and H. K. Wang. Multivariate polynomial regression for identification of chaotic time series. *Mathematical and Computer Modelling of Dynamical Systems*, 13(4):1–19, 2007. [Link]. 178
- [35] Jinsung Yoon et al. GAIN: Missing data imputation using generative adversarial nets. *Preprint*, pages 1–10, 2018. arXiv:1806.02920 [Link]. 91

Index

- χ^2 distribution, 158
 k -NN, 54, 143
 k -means clustering, 68
 k -medoids clustering, 68
 n -gram, 9
 sorted, 52
 p -adic valuation, 153
 p -value, 158, 160
- abbreviation dictionary, 15
acronyms, 48, 54
action, 9, 13, 54
activation function, 98
Adam (stochastic gradient descent), 55
agent, 13, 19, 31, 48, 55, 121, 130, 141
 multi-agent, 54
ANN (approximate nearest neighbors search), 46, 56, 145
 probabilistic ANN (pANN), 54, 145
augmentation, 16, 50, 54
 augmented knowledge graph, 16
 taxonomy augmentation, 16, 133
auto-correct, 15
auto-encoder, 54
auto-regressive model, 55
auto-tuning, 90, 96, 98
- backend, 30, 54, 55, 130
base, 157
basin of attraction, 193
batch, 90, 92, 98
Bayesian hierarchical models, 91
benchmarking, 54
Beurling primes, 175
binary search
 interpolated, 146
binning, 109
bit shift, 156
bootstrapping, 70, 177
- caching, 46, 56
cataloging, 133
categorical feature, 90
CDF regression, 178
central limit theorem, 173
chunking, 118, 120, 132, 140
confidence interval, 70, 173
 model-free, 82
confidence region, 177
congruential equidistribution
 asymptotic, 156
- connected components, 68
context, 130
 contextual pairs, 47
 contextual tables, 47
 contextual tokens, 47
contour level, 186, 190
contour lines, 186
convergence
 absolute, 182
 conditional, 182
convolution product, 166
copula, 91, 99
cosine similarity, 56, 145
covariance matrix, 111
Cramér's V, 90
crawling
 smart crawling, 9, 47
critical line (number theory), 186
cross-validation, 50, 69, 88, 177
customization (LLM), 51
- data distillation, 80
database
 graph database, 55
 JSON, 50
 key-value pairs, 50
 nested hash, 50
 vector database, 50
debugging, 51
deep neural network (DNN), 55, 98, 108
dendrogram, 70
diffusion, 54, 81
diffusion model, 166
digit block, 154
digit-preserving, 154
dimensionality reduction, 177
Dirichlet character, 175
Dirichlet- L function, 175
disambiguation, 126
disembiguation, 53
distance matrix, 67
distillation, 15, 48, 54
distributed computing, 48
distribution
 logistic, 177
DNN (deep neural network), 47
dot product, 144, 176
dummy variable, 90
dynamical system, 193
- ECDF (empirical distribution), 56, 79, 89, 108

- embedding, 20, 47, 54
 - quantized embedding, 50
 - variable length embedding, 8, 50, 144
- empirical density function, 97
- empirical distribution, 68, 178
 - multivariate, 56, 79, 88, 91, 97, 108, 146
- encoding
 - smart encoding, 69
- EPDF (empirical probability density function), 108
- epoch (neural networks), 143, 148
- equidistribution modulo 1, 157
- Euler product, 175
- evaluation, 108
 - universal, 126
- evaluation metrics, 50, 54, 56
- exhaustivity (LLM/RAG output), 126, 130
- explainable AI, 52, 55, 79, 88, 98
- feature engineering, 56
- feature selection, 177
- fine-tuning, 55, 141
 - exhaustive results, 49, 130
 - in real time, 18
 - LLM parameters, 48
 - self-tuning, 51
- frontend, 54
- GAN (generative adversarial network), 14, 79, 108, 166
 - hierarchical GAN, 91
 - NoGAN, 15
- Gaussian mixture model, 111, 166
- generative adversarial network, 55, 81, 88, 145, 166
- GIS, 184
- GPT, 55
- GPU, 49
- gradient descent, 70, 89, 98, 111, 143, 190, 192
 - steepest, 147
 - stochastic, 108, 147, 148
- gradient operator, 176
- graph
 - graph database, 55
 - knowledge graph, 50
- GRH (Generalized Riemann Hypothesis), 175
- grid search, 80, 192
 - smart grid search, 49, 112, 192
- Hadamard product, 89
- hallucination, 50
- hash table
 - inversion, 67
 - nested hash, 9, 47, 48, 50, 55, 67, 130, 133
- Hellinger distance, 97, 108
 - multivariate, 47
- hierarchical Bayesian model, 99
- hierarchical clustering, 68
- hierarchical deep resampling, 88, 91
- holdout, 91
- Hungarian algorithm, 108, 143
- hyperparameter, 49, 52, 55, 80, 90
- hyperrectangles, 79
- identifiability (statistics), 142
- imputation (missing values), 91
- in-memory database, 17
- in-memory LLM, 17, 49
- indexing, 140
 - auto-indexing, 133
 - glossary generation, 133
 - multi-index, 118, 120
 - text entity index, 50, 130
- integer division, 156
- integral probability metrics, 97
- interpolation, 193
- JSON, 9, 47, 118
 - database, 50, 55
- kernel, 166
- kernel density estimation, 166
- key-value database, 9, 55
- knowledge graph, 50, 120, 130, 140
- Kolmogorov-Smirnov distance, 68, 79, 90, 91, 97, 110, 146
 - multivariate, 47, 56, 108
- kriging, 182
- label, 96
- Lagrange multiplier, 176
- LangChain, 55
- large language model (LLM)
 - evaluation, 133
- large language models (LLM), 58
- Lasso regression, 176
- latency, 16
- law of the iterated logarithm, 158
- learning rate, 188
- link function, 178
- LLaMA, 47, 55
- LLM (large language model)
 - agentic LLM, 13
 - debugging, 51
 - for auto-indexing, 133
 - for cataloging, 133
 - for clustering, 14
 - for glossary generation, 133
 - for predictive analytics, 14
 - in-memory, 17
 - LLM router, 9, 48, 130
 - multi-LLM, 18
 - self-tuned, 51
 - sub-LLM, 18, 130
 - xLLM, 18
- logistic distribution, 177
- logistic regression, 177
- loss function, 68, 70, 89, 90, 98, 108, 143, 147, 192
 - adaptive loss, 47, 108
- matrix
 - positive semidefinite, 111
- mean squared error, 176
- Mersenne twister, 152, 155
- mixture model, 97
- mixture of experts, 48, 51, 130
- model identifiability, 176

- modulus (complex number), 186
multi-agent system, 54, 55
multimodal system, 56, 118
multinomial distribution, 79
multiple root, 182
multitoken (see token), 54
- nearest neighbors
 K-NN, 142
 approximate (ANN), 142
 probabilistic (pANN), 142
 search, 143
- NLG (natural language generation), 55
NLP (natural language processing), 15, 55
node (interpolation), 97
NoGAN, 55, 79, 109, 145
 constrained, 109
 probabilistic, 109
- normal number, 157
normalization, 56
notebookML, 133
- OpenAI, 111
orthogonal trajectory, 186
overfitting, 49, 68, 79, 176, 192
- parameter
 in neural networks, 56
 in xLLM, 55
- parametric bootstrap, 181
partial derivative, 182
period, 153
PMI (pointwise mutual information), 8, 10, 20, 32, 51, 52
pointwise mutual information (PMI), 56, 144
Poisson process, 148
positive semidefinite (matrix), 111
prediction interval, 177
prefix, 153
principal component analysis, 91
PRNG (pseudo-random number generator), 15, 49, 152
probability density function, 166
prompt
 command prompt options, 50
 prompt compression, 145
 prompt engineering, 50
 prompt synthesizer, 133
- Python library
 GenAI-evaluation, 110
 Sklearn, 68
 Sklearn_extra, 68
- quantile, 80, 110
 extrapolated, 168
quantile function, 92, 166
quantile regression, 177
quantization, 50
- R-squared, 176, 177
Rademacher function, 175
radix search, 146
RAG (retrieval augmentation generation), 16, 56, 130
- random walk
 simple, 158
- rank
 relevancy rank, 131
- recency, 53
regularization, 56
reinforcement learning, 51, 56, 98, 127
rejection sampling, 168
relevancy
 relevancy score, 49, 52, 127, 130, 141
- replicability, 49
reproducibility, 15, 49
resampling, 109
residue, 156
retrieval, 50
 retrieval augmentation generation (RAG), 56
- Riemann hypothesis
 generalized, 175
- Riemann zeta function, 175, 186
run test, 156
- saddle point, 188
scalability, 16
score
 relevancy rank, 131
 relevancy score, 30, 49
 smart scoring, 141
- search
 probabilistic search, 50
 vector search, 50
- seed (random number generator), 14, 90
self-tuning, 55, 56, 127
separator (text), 56
similarity metric, 68
simulated annealing, 98
singularization, 15
steepest descent, 190
stemming, 15
stopwords, 9, 51
supervised learning, 125
swap, 89, 98
synonyms dictionary, 15
synthetic data, 56, 108, 168, 182, 185
 constrained, 109, 112
- synthetic function, 175
- tagging
 auto-tagging, 121
- taxicab distance, 148
- taxonomy, 55
 augmentation, 133
 taxonomy creation, 58, 133
- tensor, 90
TensorFlow, 90
text entity, 13, 30, 48, 50, 54, 120, 130
 sub-entity, 19
 text entity ID, 31
- time complexity, 80
- token, 54, 56
 business token, 53
 contextual, 47, 67

graph token, 53, 120, 130, 132, 141
multitoken, 8, 20, 54, 67, 120, 130
negative token, 54
next-token prediction, 70
training, 49, 55
transformer, 14, 47, 49, 55, 56, 98
truncated Gaussian, 168

unit ball, 148
unsupervised learning, 125

validation set, 79, 91, 177
vanishing gradient, 90
variational auto-encoder, 54
vector database, 50
vector search, 50, 56, 142
vectorization, 80

Wasserstein GAN (WGAN), 98, 108
Weibull distribution, 148
weighted regression, 177
Wiener process, 158

XGboost, 79
xLLM, 18, 58
 e-xLLM (for enterprise corpus), 130