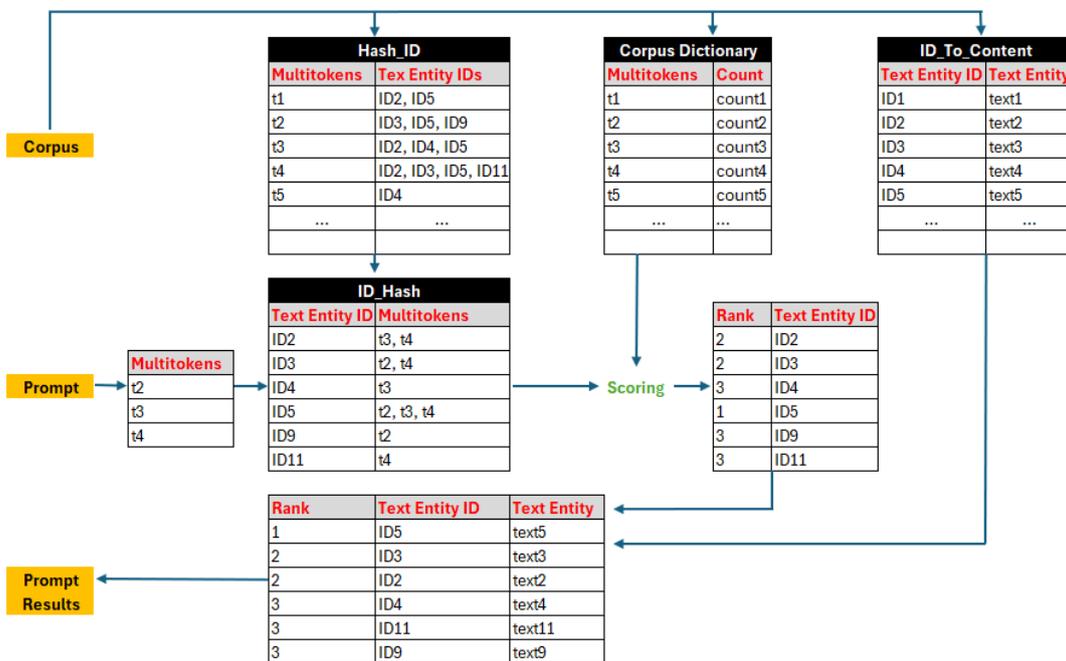
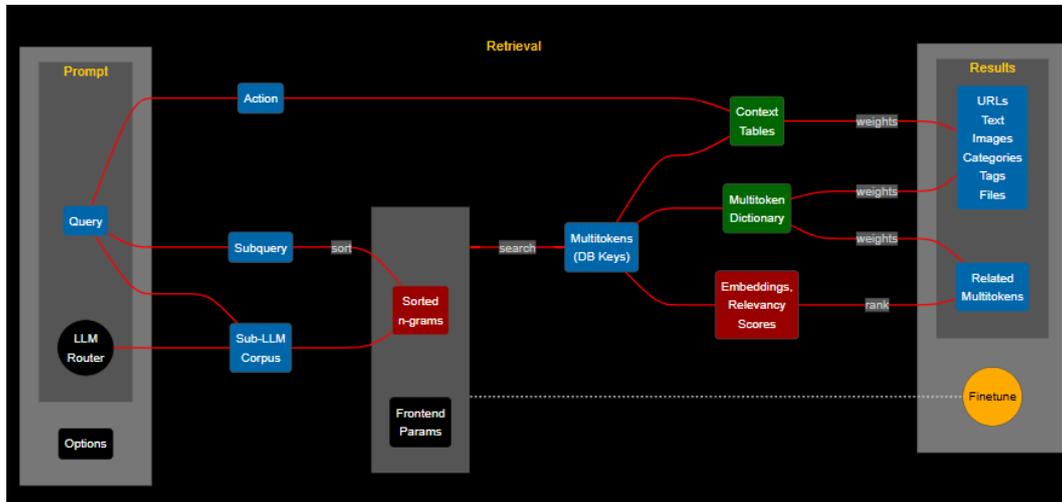


# Building Disruptive AI & LLM Technology From Scratch



# Contents

<b>I</b>	<b>Hallucination-Free LLM with Real-Time Fine-Tuning</b>	<b>7</b>
<b>1</b>	<b>Enterprise xLLM for Search &amp; Retrieval</b>	<b>8</b>
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
<b>2</b>	<b>Parameters, features, and fine-tuning</b>	<b>12</b>
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
<b>3</b>	<b>Case study</b>	<b>18</b>
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
<b>4</b>	<b>Appendix</b>	<b>34</b>
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
<b>II</b>	<b>Outperforming Neural Nets and Classic AI</b>	<b>57</b>
<b>5</b>	<b>Building and evaluating a taxonomy-enriched LLM</b>	<b>58</b>
5.1	Project and solution . . . . .	60
5.2	Python code . . . . .	61
<b>6</b>	<b>LLM for Clustering and Predictions</b>	<b>67</b>
6.1	Project and solution . . . . .	68
6.2	Visualizations and discussion . . . . .	71
6.3	Python code . . . . .	71
<b>7</b>	<b>Fast, High-Quality NoGAN for Data Synthetization</b>	<b>79</b>
7.1	Project description . . . . .	79
7.2	Solution . . . . .	81
7.3	Python implementation . . . . .	82
<b>8</b>	<b>Hierarchical Bayesian NoGAN for Data Synthetization</b>	<b>88</b>
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
<b>9</b>	<b>Boosting Model Evaluation with Smart Adaptive Loss Functions</b>	<b>108</b>
9.1	Project and solution . . . . .	109
9.2	Python code . . . . .	113
<b>10</b>	<b>Deep Retrieval and Multi-Index Chunking for PDFs</b>	<b>118</b>
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

<b>III Innovations in Statistical AI</b>	<b>129</b>
<b>11 Building a Ranking System to Enhance Prompt Results</b>	<b>130</b>
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
<b>12 Probabilistic Search: Alternative to Vector Search</b>	<b>142</b>
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
<b>13 Strong Random Generators for Reproducible AI</b>	<b>152</b>
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
<b>14 Sampling Outside the Observation Range</b>	<b>166</b>
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
<b>15 Additional Resources</b>	<b>173</b>
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
<b>16 Trading the S&amp;P 500 Index</b>	<b>192</b>
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](https://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](https://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.

# 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.

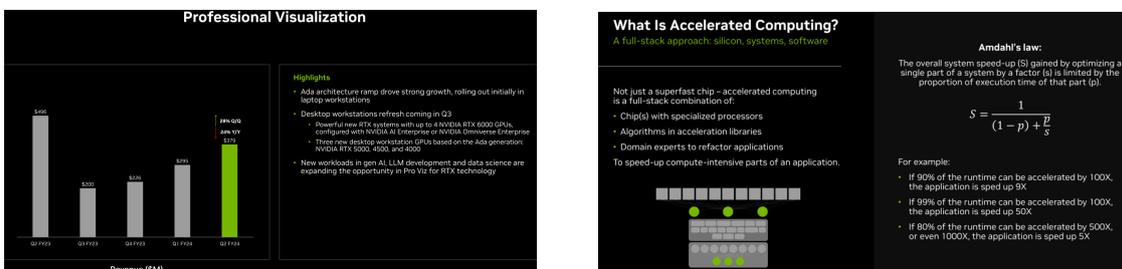


Figure 10.1: Nvidia PDF: slide with sublist, hidden table (left); with 2 lists and formula (right)

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 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     return(entity)
144
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
273 # --- Main ---
274
275 doc_ID = 0 # to identify the PDF doc
276 filename = 'nvda-f2q24-investor-presentation-final-1'
277 pdf_path = filename + '.pdf'
278 json_path = filename + '.json'
279 text_path = filename + '.txt'

```

```

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.

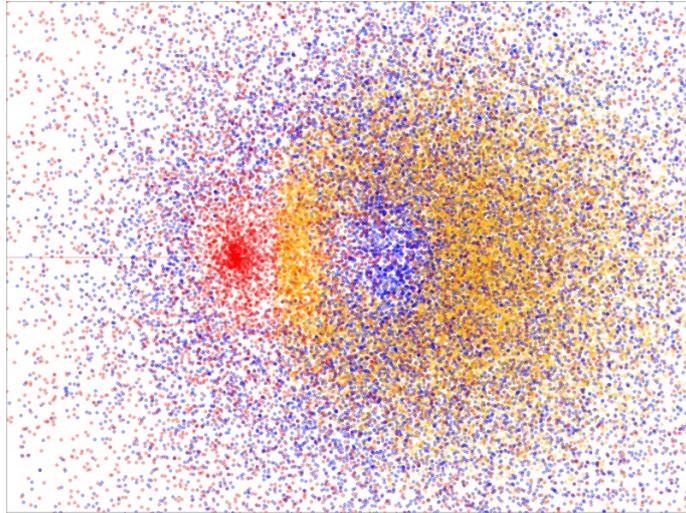


Figure 10.2: Training an LLM is as hard as training a clustering algorithm

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.

---

<p>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.</p>	<p>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.</p>
<p>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.</p>	<p>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.</p>

---

# Index

- $\chi^2$  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