

Chapter 10

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 10.3), 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 10.1.

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.

10.1 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 multitoken is in the contextual elements or in the regular text attached to a specific text entity, the length of the text entity and multitoken, the number of single tokens in the multitokens, the occurrence count attached to the multitoken across the whole corpus (stored in the multitoken `dictionary`), and whether a multitoken consists of adjacent tokens or not. From there, I build 4 scores to measure the fit between a text entity (represented by its ID), and the prompt:

- Score A: measuring the importance of the multitokens found both in the text entity, and in the prompt.
- Score B: number of multitokens found both in the text entity, and in the prompt (intersection).
- Score C: same as score A, but for multitokens also found in the contextual fields in the text entity.
- Score D: same as score 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 10.4. In particular, the formula for Score A, for a specific text entity ID, is as follows:

$$\text{Score}_A[\text{ID}] = \sum_{t \in S_P[\text{ID}]} \lambda_t w_t^{-\beta_t}, \quad (10.1)$$

where $S_P[\text{ID}]$ 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 (10.1): the fact that tokens outside $S_P[\text{ID}]$ are ignored, further speeding 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. Here, it is not an issue.

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 A, B, C, 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 B, C, D gives them more importance over A, as they are related either to multitokens found in knowledge graph elements, or to multiple tokens consisting of several tokens.

10.2 Case study

Figures 10.1 and 10.2 show two prompts, and the text entity IDs retrieved by xLLM. To see the full content of these text entities, see 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 10.1: 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 10.2: 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 A, B, C, 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 10.1 and 10.2 are only a small subset of what xLLM found, selected via the ranking mechanism. Also note the token starting with ‘_’ in Figure 10.1, identifying a `graph token`. It comes from a knowledge graph element in text entity 2701, rather than from regular text.

10.2.1 Conclusion

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**. 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 challenge is automated **model evaluation**: xLLM returns concise yet exhaustive bullet lists with a score attached to each item, see Figure 10.4. How do you assess exhaustivity? How to take into accounts 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.

10.3 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 10.3: 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 10.3 and 10.4, 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 10.1 and 10.2.

The screenshot shows a ChatGPT interface. The user has asked about the Central Limit Theorem. The AI response explains that it is a fundamental concept in probability theory and statistics, stating 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. It then lists three key points: Independence, Identically Distributed, and Large Sample Size. Finally, it discusses the implications for statistical inference, noting that 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, which is often used for hypothesis testing and confidence intervals.

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

10.4 Python code

The Python code here pertains to the enterprise version v2 of xLLM, discussed in sections 10.1 and 10.2. 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     exllm.update_nestedHash(embeddings, wordA, wordB, pmi)
175     exllm.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         exllm.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_ = " "

```

```

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/refs/heads/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("\n-----")
268     print("Prompt: ", query)
269     query = query.replace('?', ' ').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             # context 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     #---- Save backend tables
369
370     def create_KW_map(dictionary):
371         # singularization
372         # map key to KW_map[key], here key is a single token
373         # need to map unseen prompt tokens to related dictionary entries
374         #   example: ANOVA -> analysis~variance, ...
375
376         OUT = open("KW_map.txt", "w")
377         for key in dictionary:
378             if key.count('~') == 0:
379                 j = len(key)
380                 keyB = key[0:j-1]
381                 if keyB in dictionary and key[j-1] == 's':
382                     if dictionary[key] > dictionary[keyB]:
383                         OUT.write(keyB + "\t" + key + "\n")
384                     else:
385                         OUT.write(key + "\t" + keyB + "\n")
386
387         OUT.close()
388         return()
389
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()
```

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. [\[Link\]](#). 94, 161
- [2] Marc G. Bellemare et al. The Cramer distance as a solution to biased Wasserstein gradients. *Preprint*, pages 1–20, 2017. arXiv:1705.10743 [\[Link\]](#). 98
- [3] Iulia Brezeanu. How to cut RAG costs by 80% using prompt compression. *Blog post*, 2024. TowardsData-Science [\[Link\]](#). 31, 137
- [4] Wei Chen and Mark Fuge. Synthesizing designs with interpart dependencies using hierarchical generative adversarial networks. *Journal of Mechanical Design*, 141:1–11, 2019. [\[Link\]](#). 93
- [5] Johnathan Chiu, Andi Gu, and Matt Zhou. Variable length embeddings. *Preprint*, pages 1–12, 2023. arXiv:2305.09967 [\[Link\]](#). 31, 136
- [6] Fabian Gloeckle et al. Better & faster large language models via multi-token prediction. *Preprint*, pages 1–29, 2024. arXiv:2404.19737 [\[Link\]](#). 31, 67
- [7] Vincent Granville. Generative AI: Synthetic data vendor comparison and benchmarking best practices. *Preprint*, pages 1–13, 2023. MLTechniques.com [\[Link\]](#). 90, 98
- [8] Vincent Granville. Generative AI technology break-through: Spectacular performance of new synthesizer. *Preprint*, pages 1–16, 2023. MLTechniques.com [\[Link\]](#). 80, 83
- [9] Vincent Granville. *Gentle Introduction To Chaotic Dynamical Systems*. MLTechniques.com, 2023. [\[Link\]](#). 144, 146, 147, 150
- [10] Vincent Granville. How to fix a failing generative adversarial network. *Preprint*, pages 1–10, 2023. MLTechniques.com [\[Link\]](#). 82
- [11] Vincent Granville. Massively speed-up your learning algorithm, with stochastic thinning. *Preprint*, pages 1–13, 2023. MLTechniques.com [\[Link\]](#). 81
- [12] Vincent Granville. Smart grid search for faster hyperparameter tuning. *Preprint*, pages 1–8, 2023. MLTechniques.com [\[Link\]](#). 47, 81, 115
- [13] Vincent Granville. *State of the Art in GenAI & LLMs – Creative Projects, with Solutions*. MLTechniques.com, 2024. [\[Link\]](#). 126, 166, 168
- [14] Vincent Granville. *Statistical Optimization for AI and Machine Learning*. MLTechniques.com, 2024. [\[Link\]](#). 54, 112, 114, 115, 135, 137, 166
- [15] Vincent Granville. *Synthetic Data and Generative AI*. Elsevier, 2024. [\[Link\]](#). 135, 168
- [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. 100
- [17] Albert Jiang et al. Mixtral of experts. *Preprint*, pages 1–13, 2024. arXiv:2401.04088 [\[Link\]](#). 31
- [18] L. Kuipers and H. Niederreiter. *Uniform Distribution of Sequences*. Dover, 2012. [\[Link\]](#). 149
- [19] 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 [\[Link\]](#). 93
- [20] 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. [\[Link\]](#). 99
- [21] Tengyuan Liang. Estimating certain integral probability metric (IPM) is as hard as estimating under the IPM. *Preprint*, pages 1–15, 2019. arXiv:1911.00730 [\[Link\]](#). 99
- [22] Andrei Lopatenko. Evaluating LLMs and LLM systems: Pragmatic approach. *Blog post*, 2024. [\[Link\]](#). 31

- [23] Sebastián Maldonado et al. An adaptive loss function for deep learning using OWA operators. *Preprint*, pages 1–15, 2023. arXiv:2305.19443 [Link]. 111
- [24] Hussein Mozannar et al. The RealHumanEval: Evaluating Large Language Models’ abilities to support programmers. *Preprint*, pages 1–34, 2024. arXiv:2404.02806 [Link]. 31
- [25] Michael Naaman. On the tight constant in the multivariate Dvoretzky–Kiefer–Wolfowitz inequality. *Statistics & Probability Letters*, 173:1–8, 2021. [Link]. 99
- [26] Sergey Shchegrikovich. How do you create your own LLM and win The Open LLM Leaderboard with one Yaml file? *Blog post*, 2024. [Link]. 31
- [27] Bharath Sriperumbudur et al. On the empirical estimation of integral probability metrics. *Electronic Journal of Statistics*, pages 1550–1599, 2012. [Link]. 99
- [28] 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]. 93
- [29] Eyal Trabelsi. Comprehensive guide to approximate nearest neighbors algorithms. *Blog post*, 2020. TowardsDataScience [Link]. 138
- [30] Jinsung Yoon et al. GAIN: Missing data imputation using generative adversarial nets. *Preprint*, pages 1–10, 2018. arXiv:1806.02920 [Link]. 93

Index

- χ^2 distribution, 150
- k -NN, 52, 135
- k -means clustering, 68
- k -medoids clustering, 68
- n -gram, 8
 - sorted, 50
- p -adic valuation, 145
- p -value, 150, 152
- abbreviation dictionary, 14
- acronym, 52
- acronyms, 46
- action, 8, 12, 51
- activation function, 100
- Adam (stochastic gradient descent), 52
- agent, 12, 18, 29, 46, 53, 123
 - multi-agent, 51
- ANN (approximate nearest neighbors search), 44, 54
 - probabilistic ANN (pANN), 52
- ANN (approximate nearest neighbors), 137
 - probabilistic ANN (pANN), 137
- approximated nearest neighbors, 135
- augmentation, 15, 48, 52
 - augmented knowledge graph, 15
 - taxonomy augmentation, 15, 126
- auto-correct, 14
- auto-encoder, 52
- auto-regressive model, 53
- auto-tuning, 92, 98, 100
- backend, 29, 52, 53, 123
- base, 149
- batch, 92, 94, 100
- Bayesian hierarchical models, 93
- benchmarking, 52
- Beurling primes, 168
- binary search
 - interpolated, 138
- binning, 112
- bit shift, 148
- bootstrapping, 70
- caching, 44, 54
- categorical feature, 92
- central limit theorem, 166
- confidence interval, 166
- confidence intervals, 70
 - model-free, 83
- congruential equidistribution
 - asymptotic, 148
- connected components, 68
- context, 123
 - contextual pairs, 45
 - contextual tables, 45
 - contextual tokens, 45
- convolution product, 159
- copula, 93, 101
- cosine similarity, 54, 137
- covariance matrix, 114
- Cramér's V, 92
- crawling
 - smart crawling, 8, 45
- cross-validation, 48, 69, 90
- customization (LLM), 49
- data distillation, 81
- database
 - graph database, 53
 - JSON, 48
 - key-value pairs, 48
 - nested hash, 48
 - vector database, 48
- debugging, 49
- deep neural network, 100
- deep neural network (DNN), 52, 111
- dendrogram, 70
- diffusion, 52, 82
- diffusion model, 159
- digit block, 146
- digit-preserving, 146
- Dirichlet character, 168
- Dirichlet- L function, 168
- distance matrix, 67
- distillation, 14, 46, 52
- distributed computing, 46
- DNN (deep neural network), 45
- dot product, 136
- dummy variable, 92
- ECDF (empirical distribution), 54, 80, 111
- ECDF empirical distribution, 91
- embedding, 19, 45, 52
 - quantized embedding, 48
 - variable length embedding, 7, 48
- embeddings
 - variable length, 136
- empirical density function, 99
- empirical distribution, 68
 - multivariate, 54, 80, 90, 93, 99, 111, 138
- encoding
 - smart encoding, 69
- EPDF (empirical probability density function), 111

- epoch (neural networks), 135, 139
equidistribution modulo 1, 149
Euler product, 168
evaluation (GenAI models), 111
evaluation metric, 52
evaluation metrics, 48, 53
exhaustivity (LLM/RAG output), 123
explainable AI, 50, 52, 80, 90, 100

feature encoding, 54
feature engineering, 53
fine-tuning, 52
 exhaustive results, 47, 123
 in real time, 17
 LLM parameters, 46
 self-tuning, 49
frontend, 52

GAN (generative adversarial network), 13, 80, 111, 159
 hierarchical GAN, 93
 NoGAN, 14
Gaussian mixture model, 114, 159
generative adversarial network, 52, 82, 137, 159
generative adversarial networks, 90
GPT, 52
GPU, 47
gradient descent, 70, 91, 100, 114, 135
 steepest, 139
 stochastic, 111, 139
graph
 graph database, 53
 knowledge graph, 48
GRH (Generalized Riemann Hypothesis), 168
grid search, 81
 smart grid search, 47, 115

Hadamard product, 91
hallucination, 48
hash table
 inversion, 67
 nested hash, 8, 45, 46, 48, 53, 67, 123, 126
Hellinger distance, 99, 111
 multivariate, 45
hierarchical Bayesian model, 101
hierarchical clustering, 68
hierarchical deep resampling, 90, 93
holdout, 93
Hungarian algorithm, 111, 135
hyperparameter, 47, 50, 52, 81, 92
hyperrectangles, 80

identifiability (statistics), 134
imputation (missing values), 93
in-memory database, 16
in-memory LLM, 16, 47
indexing
 text entity index, 48, 123
integer division, 148
integral probability metrics, 99

JSON, 8, 45
 database, 48, 53

kernel, 159
kernel density estimation, 159
key-value database, 8, 53
knowledge graph, 123
knowledege graph, 48
Kolmogorov-Smirnov distance, 68, 80, 92, 93, 99, 113, 138
multivariate, 45, 54, 111

label feature, 98
LangChain, 53
large language model
 evaluation, 126
large language models (LLM), 57
latency, 15
law of the iterated logarithm, 150
LLaMA, 45, 53
LLM (large language model)
 agentic LLM, 12
 debugging, 49
 for clustering, 13
 for predictive analytics, 13
 in-memory, 16
 LLM router, 8, 46, 123
 multi-LLM, 17
 self-tuned, 49
 sub-LLM, 17, 123
 xLLM, 17
loss function, 68, 70, 91, 92, 100, 111, 135, 139
 adaptive loss, 45, 111

matrix
 positive semidefinite, 114
Mersenne twister, 144, 147
mixture model, 99
mixture of experts, 46, 49, 123
multi-agent system, 51, 53
multimodal system, 53
multinomial distribution, 80
multitoken (see token), 52

nearest neighbors
 K-NN, 134
 approximate (ANN), 134
 probabilistic (pANN), 134
NLG (natural language generation), 53
NLP (natural language processing), 14, 53
node (interpolation), 99
NoGAN, 52, 80, 112, 137
 constrained, 112
 probabilistic, 112
normal number, 149
normalization, 53, 54

OpenAI, 114
overfitting, 47, 68, 80

parameter
 in neural networks, 53, 54
 in xLLM, 53
period, 145

PMI (pointwise mutual information), 7, 9, 19, 31, 49, 50
pointwise mutual information (PMI), 54, 136
Poisson process, 139
positive semidefinite (matrix), 114
prefix, 145
principal component analysis, 93
PRNG (pseudo-random number generator), 14, 47, 144
probability density function, 159
prompt
 command prompt options, 48
 prompt engineering, 48
 synthesizer, 125
prompt compression, 137
Python library
 GenAI-evaluation, 113
 Sklearn, 68
 Sklearn_extra, 68

quantile, 81, 113
 extrapolated, 161
quantile function, 94, 159
quantization, 48

Rademacher function, 168
radix search, 138
RAG (retrieval augmentation generation), 15, 53, 123
random walk
 simple, 150
rank
 relevancy rank, 124
regularization, 54
reinforcement learning, 49, 54, 100
rejection sampling, 161
relevancy
 relevancy score, 47, 50, 123
replicability, 47
reproducibility, 14, 47
resampling, 112
residue, 148
retrieval, 48
 retrieval augmentation generation (RAG), 53
Riemann hypothesis
 generalized, 168
Riemann zeta function, 168
run test, 148

scalability, 15
score
 relevancy rank, 124
 relevancy score, 29, 47
search
 probabilistic search, 48
 vector search, 48
seed (random number generator), 13, 92
self-tuning, 52, 54
separator (text), 54
similarity metric, 68
simulated annealing, 100
singularization, 14
stemming, 14
stopwords, 8, 49
swap, 91, 100
synonyms dictionary, 14
synthetic data, 54, 111, 161
 constrained, 112, 115
synthetic function, 168

taxicab distance, 139
taxonomy, 53
 augmentation, 126
 taxonomy creation, 57, 126
tensor, 92
TensorFlow, 92
text entity, 12, 29, 46, 48, 52, 123
 sub-entity, 18
 text entity ID, 29
time complexity, 81
token, 52, 54
 contextual, 45, 67
 graph token, 123, 125
 multi-token, 67
 multitoken, 7, 19, 52, 123
training, 47, 52
transformer, 13, 45, 47, 52, 54, 100
truncated Gaussian, 161

unit ball, 139

validation set, 80, 93
vanishing gradient, 92
variational auto-encoder, 52
vector database, 48, 54
vector search, 48, 54, 134
vectorization, 81

Wasserstein GAN, 100
Wasserstein GAN (WGAN), 111
Weibull distribution, 139
Wiener process, 150

XGboost, 80
xLLM, 17, 57
 e-xLLM (for enterprise corpus), 123