Hyperfast Contextual Custom LLM with Agents, Multitokens, Explainable AI, and Distillation

Vincent Granville, Ph.D.
vincentg@MLTechniques.com
www.GenAItechLab.com
Version 2.0, September 2024

Abstract

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

Contents

1 xLLM: innovative architecture

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

- No training, no neural network involved. Thus, very fast and easy to fine-tune with explainable parameters, and much fewer tokens. Yet, most tokens consist of multiple terms and are called multitokens. Also, I use variable-length embeddings. Cosine similarity and dot products are replaced by customized pmi (pointwise mutual information, [Wiki]).

- Parameters have a different meaning in my context. In standard architectures, they represent the weights connecting neurons. You have billions or even trillions of them. But there is no neural network involved here: instead, I use parametric weights governed by a few top-level parameters. The weights – explicitly specified rather than iteratively computed – are not the parameters. My architecture uses two parameter sets: frontend and backend. The former are for scoring and relevancy; they are fine-tuned in real time with no latency, by the user or with some algorithm. A relevancy score is shown to the user, for each retrieved item.

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

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

Figure 1: Nested hash database, lines 12–27 in the code

- I don't use vector or graph databases. Tables are stored as nested hashes, and fit in memory (no GPU needed). By nested hashes, I mean key-value tables, where the value may also be a key-value table. The format is similar to JSON objects, see Figures 1 and 3. In standard architectures, the central table stores the embeddings. Here, embeddings are one of many backend tables. In addition, there are many contextual tables (taxonomy, knowledge graph, URLs) built during the crawling. This is possible because input sources are well structured, and elements of structure are recovered thanks to smart crawling.

- The Python code does not use any library, nor any API call. Not even Pandas, Numpy, or NLTK. So you can run it in any environment without concern for library versioning. Yet it has fewer than 600 lines of code, including the fine-tuning part in real time. I plan to leverage some library functions in the future such as auto-correct, singularize, stem, stopwords and so on. However, home-made solutions offer more customization, such as ad-hoc stopwords lists specific to each sub-LLM, for increased performance. For instance, the one-letter word 'p' can not be eliminated if the sub-LLM deals with statistical concepts. The only exception to the "no library" rule is the Requests library, if you choose to download the test enterprise corpus from its GitHub location.

- This article focuses only on one part of an enterprise corpus: the internal documentation about how to implement or integrate AI and machine learning solutions. Other parts include marketing, IT, product, sales, legal and HR. A specific sub-LLM is built for each part, using the same architecture. The full LLM consists of these sub-LLMs, glued together with an LLM router to redirect user prompts to the specific parts, possibly spanning across multiple sub-LLMs. For instance, 'security' is found in multiple sub-LLMs.

1.1 From frontend prompts to backend tables

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

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

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

Figure 2: Primary backend tables, lines 193–210 in the code

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

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

Figure 3: Updating primary backend tables, lines 61–72 in the code

From there, each backend table is queried to retrieve the value attached to a specific n-gram found in the prompt. The value in question is also a key-value table: for instance a list of URLs where the key is an URL and the value is the number of occurrences of the n-gram in question, on the landing page. In each section (titles, URLs, descriptions and so on) results shown to the user are displayed in relevancy order, with a higher weight assigned to n-grams (that is, multitokens) consisting of many words, as opposed to multitokens consisting of one or two words. Embeddings are derived from a backend table called `hash_pairs` consisting of pairs of multitokens found in the same sub-entity in the corpus. Finally, multitokens may or may not be adjacent. Pairs with non-adjacent multitokens are called contextual pairs. Occurrences of both multitokens, as well as joint occurrence (when both are simultaneously found in a same sub-entity) are used to compute pmi, the core relevancy metric. Embeddings are stored in the `embeddings` key-value backend table, also indexed by multitokens. Again, values are key-value tables, but this time the nested values are pmi scores.

1.2 What is not covered here

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

Some features are not yet implemented in this version, but available in the previous version discussed here and in my book "State of the Art in GenAI & LLMs – Creative Projects, with Solutions", available here. The following will be available in the next release: auto-correct, stemming, singularization and other text processing techniques, both applied to the corpus (crawled data) and the prompt. I will also add the ability to use pre-computed backend tables rather than building them from the crawl each time. Backend tables produced with the default backend parameters (see code lines 193–262 in section 4.1) are on GitHub, here.

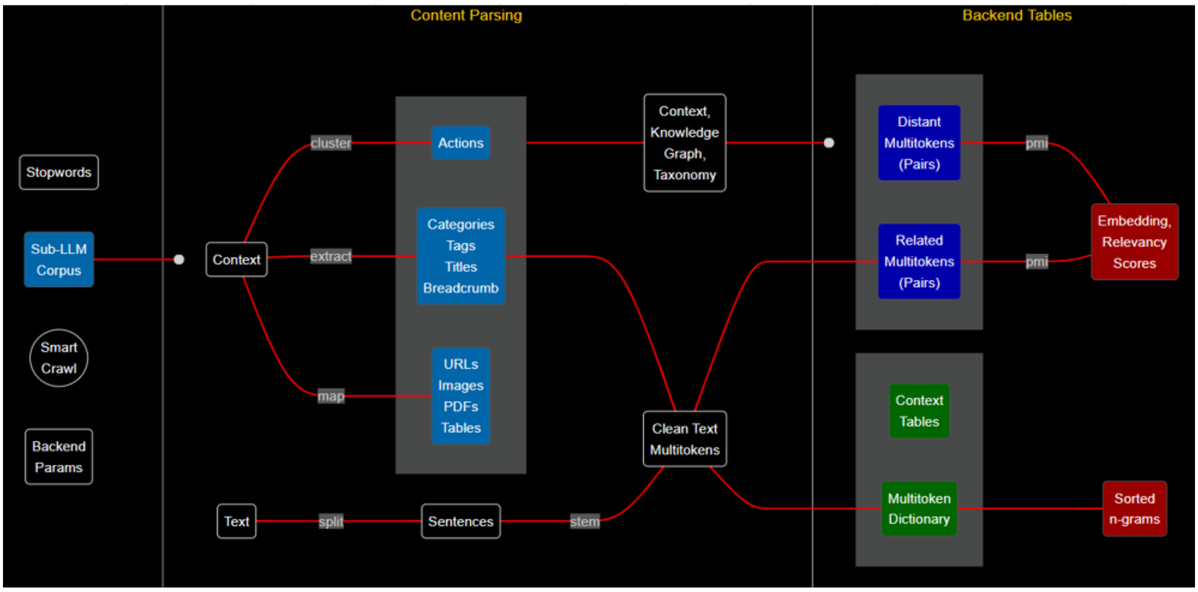


Figure 4: From crawl to backend tables (high resolution here)

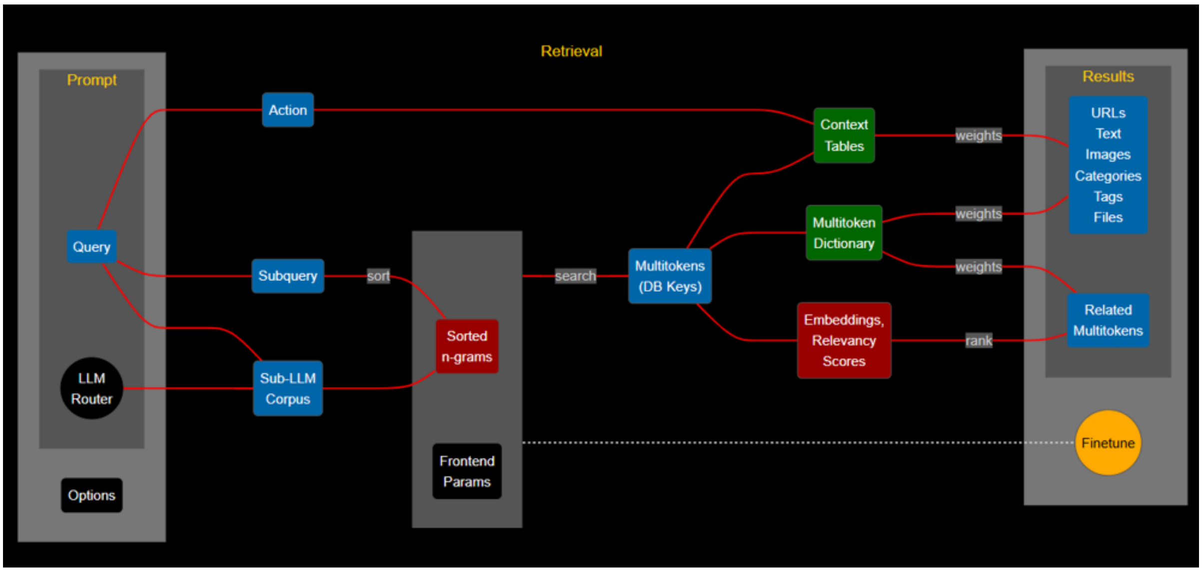


Figure 5: From prompt to query results, via backend tables (high resolution here)

Also to be included in the next release: corpus augmentation with synonyms and abbreviations dictionaries, as well as contextual multitokens. The latter is implemented in the previous version and discussed in section 8.3 in my book [4]. It consists of tokens containing non-adjacent words in the corpus. However, contextual pairs are included in the current release: it consists of pairs of non-adjacent multitokens, stored in a table called `ctokens` used to produce the embeddings. See lines 183–186 in the code. Then, words such as 'San Francisco' must be treated as single tokens.

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

2 Parameters, features, and fine-tuning

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

2.1 Backend parameters

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

Stopwords – the words to ignore when building the tables – are manually detected by looking at the most frequent tokens, both in the corpus and in prompt result: see the list in lines 216–222. Finally, when counting multitoken occurrences, appearances in categories, titles and tags get an extra boost, compared to regular text: see lines 268–275 and Figure 3. For the full list of backend parameters, see Figure 6.

```
backendParams = {
    'max_multitoken': 4, # max. consecutive terms per multi-token for inclusion in dictionary
    'maxDist' : 3,       # max. position delta between 2 multitokens to link them in hash_pairs
    'maxTerms': 3,       # maxTerms must be <= max_multitoken
    'extraWeights' :     # deafault weight is 1
       {
          'description': 0.0,
          'category':    0.3,
          'tag_list':    0.4,
          'title':       0.2,
          'meta':        0.1
       }
}
```

Figure 6: Backend parameters, lines 697–722 in the code

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

$$\text{pmi}(t_A, t_B) = \frac{n_{AB}}{\sqrt{n_A \cdot n_B}}, \tag{1}$$

where n_A, n_B, n_{AB} are the counts (computed on the full corpus) respectively for multitokens t_A, t_B, and the joint occurrence of t_A, t_B within a same sub-entity (that is, a sentence identified by separators, within a text entity). The user can choose a different formula, or different separators. Primary backend tables are listed in Figure 2.

2.2 Frontend parameters

Given the small size of the corpus and backend tables, the backend parameters can be updated in real time. Currently, the code allows the user to easily update the frontend parameters while testing various prompts. The frontend parameters are found in lines 699–721 in the code, and in Figure 13. They control the results displayed, including the choice of a customized pmi function, and top keywords to exclude such as 'data' found in almost all text entities. Adding 'data' to the `ignore` list does not eliminate results based on multitokens containing 'data', as long as the multitokens in question consist of more than one word, such as 'data asset'.

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

```python
def default_frontendParams():

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

Figure 7: Default frontend parameters, lines 699–721 in the code

2.3 Agents

Agents determine the user intent to retrieve the appropriate content. For instance:, examples, data, definitions, best practices, standards, on-boarding, and so on. In Figure 5, they are represented by the action box. One way to create an agentic LLM is to add an agent field in each text entity when crawling the corpus. See sample text entity in Table 1. You can do it using clustering techniques, applied to the corpus. Text entities are relatively small pieces of content coming straight from the corpus, usually determined by the corpus structure: in this case, a bubble database, but it could also be a repository of PDF documents or web pages.

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

Figure 8: Agent map, lines 227–245 in the code

Getting a list of top multitokens helps your build your agent backend table. In our example, see the list in question Table 1, extracted from the `dictionary` backend table. Another option consists in analyzing dozens, thousands, or millions of user prompts to identify potential actions. The ideal solution is to combine all these options to create agents that correspond not only to user intent, but also to what is actually in the corpus.

The agent map for my case study, is pictured in Figure 13. I will improve the format in the next version, and use a many-to-many rather than many-to-one table. In the key-value pairs in the picture, the value on the right is an agent, while the key on the left is a multitoken. The structure thus maps words found in the corpus, to agents. Agents are then incorporated to backend tables for retrieval. In my current implementation, there are two agent backend tables, besides `agent_map` just described:

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

These two tables are used to produce the agent section in the query results, as shown in Figure 9. For details, see lines 679–686 in the code. For instance, the fourth line in the picture tells you that the multitoken 'data assets' is associated to agent 'Governance' (among others), and that four text entity IDs match this combination: 42, 48, 199, 259, with 259 having the most content with 1153 characters.

In Figure 9, the size of each entity ID is also displayed to help the user identify IDs with more content; they might be more valuable. With the command `-i ID` in the prompt box, the user can then retrieve the full content of entity `ID`, in a format similar to Table 1. Two extra backend tables are involved in the process: `hash_size` and `ID_to_content`.

```
('Data', 'detailed') --> (511, 513)
('Example', 'data assets') --> (90,)
('Example', 'detailed') --> (90,)
('Governance', 'data assets') --> (42, 48, 199, 259)
('Governance', 'detailed') --> (101, 107)
('Governance', 'information assets') --> (223,)
('Policy', 'data assets') --> (42, 48, 199)
('Policy', 'detailed') --> (101,)
('Policy', 'information assets') --> (223,)
('Template', 'detailed') --> (107,)

  ID  Size

 511   690
 513   692
  90   772
  42   948
  48   916
 199   980
 259  1153
 101   851
 107  1242
 223   978
```

Figure 9: Example of agent section shown in query results

Currently, the agent(s) are not automatically detected from the user prompt. I will add this feature in the next version. In the meanwhile, it is possible to display the full list of agents to the user, and let him make his selection. Finally, my agents do not perform actions such as writing messages or solving math problems. Their goal is to deliver more relevant results, based on what users are looking for by analyzing prompt data. A different version of my xLLM performs clustering, build taxonomies, and make predictions based on text: see here, and Figure 16.

2.4 Reproducibility

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

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

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

2.5 Singularization, stemming, auto-correct

The `KW_map` backend table built in lines 870–888 in the code (see Figure 10), is a first attempt at adding NLP functions without using Python libraries. The table is created and saved after running the full code for the first time. Python libraries have glitches that can result in hallucinations, for instance singularizing "hypothesis" to "hypothesi". They require exception lists such as do-not-singularize as a workaround. Thus the idea to avoid them.

The code featured in Figure 10 links the singular and plural version of single-tokens found in the dictionary (when both exist), so that a user looking for (say) "tests" also gets result coming from "test". See lines 822–823 in the code when processing frontend prompts, and lines 148–149 when building backend tables.

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

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

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

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

Figure 10: Building the `KW_map` backend table

2.6 Augmentation, distillation, and frontend tables

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

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

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

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

    return(q_dictionary, q_embeddings)
```

Figure 11: Frontend token distillation before returning results

Then, I remove single tokens that are part of a multitoken when both have the same count in the dictionary. See line 862 in the code, calling the function pictured in Figure 11. It makes the output shown to the user, less cluttered. This step is called distillation. In standard LLMs, distillation is performed on backend tokens using a different mechanism, since multitokens are usually absent; it may result in hallucinations if not done properly. Also, in standard LLMs, the motivation is different: reducing a 500 billion token list, to (say) 50 billion. In xLLM, token lists are at least 1000 times smaller, so there is no real need for backend distillation.

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

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

2.7 In-memory database, latency, and scalability

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

- Pre-load the backend tables once they have been created on the first run; do not build them each time.
- Do not create the `hash_context4` and `full_content` tables; these are among the largest, and redundant with `ID_to_content`.
- Keep only one copy of identical text entities: ideally remove duplicates directly in the corpus, as opposed to using memory-consuming `entity_list` (see lines 296 and 305).
- Unless feasible, do not store `ID_to_content` that maps the entity IDs to their full content, in memory. Only store the list of IDs using small ID tables (`hash_ID`, `ID_size`, `ID_to agents`). The idea is to search for matching IDs in the backend tables when processing a prompt, and then retrieve the actual content from a database matching IDs to content.
- A distributed architecture can be useful, whereas separate sub-LLMs are stored on different clusters, if needed.

For the time being, my system is a full in-memory LLM with in-memory database. All the backend tables and text entities (see example in Table 1) are stored in memory.

Table 1: Sample text entity from corporate corpus

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

3 Case study

I now show how xLLM (the name of my LLM) works on one part of a corporate corpus (fortune 100 company), dealing with documentation on internal AI systems and policies. Here, I implemented the sub-LLM dedicated to this content. The other parts – marketing, products, finance, sales, legal, HR, and so on – require separate overlapping sub-LLMs not covered here. The anonymized corpus consists of about 300 distinct text entities, and can be found here. Table 1 features a sample text entity. The full corpus would be processed with a multi-LLM and LLM router.

In addition to the original features described in section 2, xLLM comes with a command menu, shown in Figure 12. This menu allows you to enter a standard prompt, but also to change the front-end parameters for real-time fine-tuning. Figures 4 and 5 show the main components and workflow for a single sub-LLM. Zoom in for higher resolution. For best resolution, download the original here on Google Drive for the backend diagram, and here for the frontend. Finally, the home-made LLM discussed here can be used to create a new taxonomy of the crawled corpus, based on top multitokens. These are listed, from left to right and top to bottom by order of importance, in Table 2. Note that here, I did not give a higher weight to mutlitokens consisting of multiple words. The table was produced using lines 372–375 in the Python code.

Table 2: Top multitokens found in corpus, ordered by importance

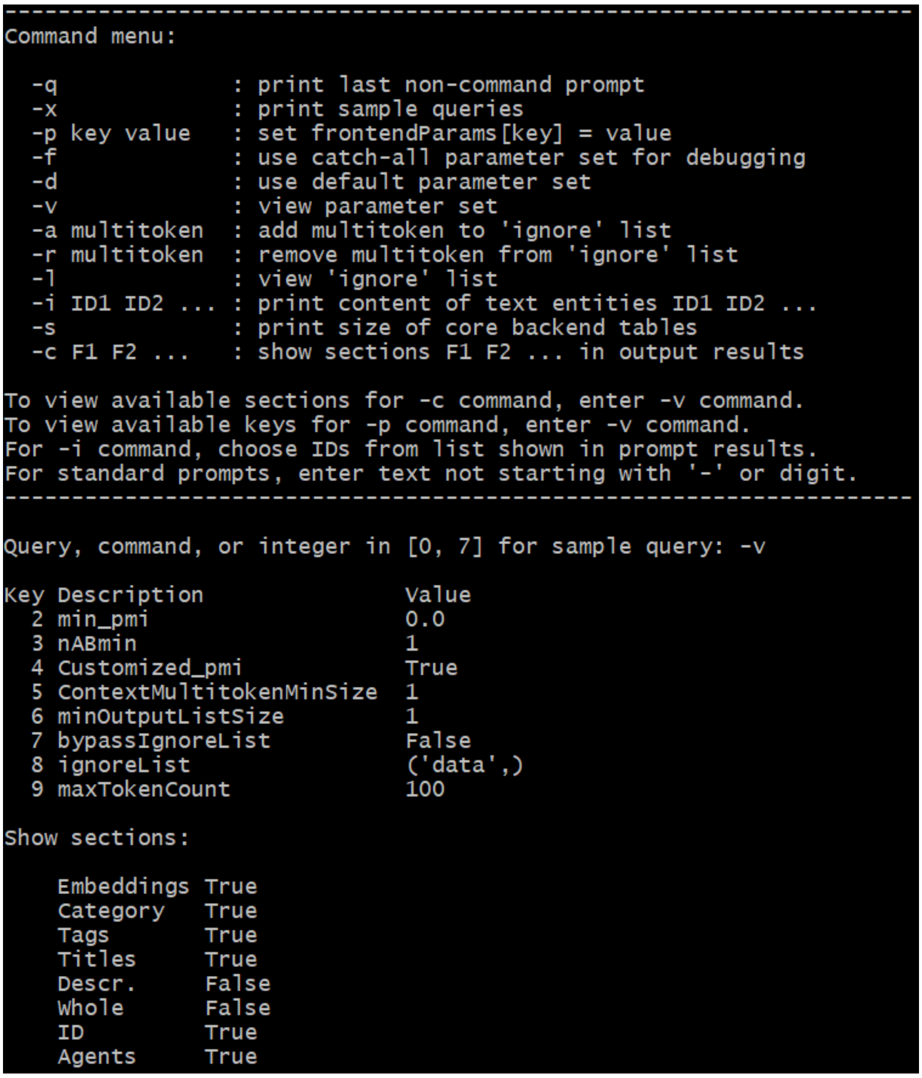
adls	storage	azure	examples	adf
csa	pipeline	development	framework	architecture
design	mltxdat	process	extract	orc
overview	quality	databricks	data quality	table
guidelines	new	guide	best practices	performance
platform	metadata	solution	business	products
project	resources	create	request	mltxhub
case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

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

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

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

Here I illustrate a full xLLM session, using a more complex sample query. It also involves fine-tuning front-end parameters in real time. The full session with commands from the command menu, and output results, is listed in section 3.2. Figure 12 shows how the command prompt looks like, as well as the result after executing the −v command.



```
-----------------------------------------------------------------------
Command menu:

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

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

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

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

Show sections:

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

Figure 12: Command options and frontend parameters

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

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

Table 3: Sample xLLM session

The detailed log with executed commands and all the output is shown in section 3.2. In particular, the nine commands in Table 3 are found at the corresponding line numbers (rightmost column in Table 3), in the log file in section 3.2. Perhaps the most useful results consist of the IDs attached to agents and multitokens related to the prompt, in lines 542–567. Also pictured in Figure 13, along with interpretation details in section 2.3. The actual content corresponding to these IDs is shown in lines 593–641. The prompt itself is shown in line 24.

I was particularly interested in finding the articles (text entities) matching my prompt, especially the titles, to check out those that interest me most. This is accomplished with the `-c Titles` command, and the results are shown in lines 988–1001. In the next code release, the corresponding text entity IDs will also be displayed along with the titles, as in the Agents section (Figure 13). This way, it is very easy to retrieve the full content corresponding the the titles in question, with the `-i` command.

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

3.2 Sample session

Here is the full log obtained by executing the commands in Table 3, including standard prompts. The executed program is called `xllm-enterprise-v2.py`, with source code in section 4.1 and on GitHub. The input data source, also on GitHub, is a fully anonymized version of one part of a corporate corpus. Keyword pairs (at the beginning) come from the embeddings backend table. Entries flagged with a star (*) mark contextual pairs. Also,

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

Below is the session log.

```
1  ______________________________________________________________________
2  Command menu:
3
4    -q           : print last non-command prompt
5    -x           : print sample queries
6    -p key value : set frontendParams[key] = value
7    -f           : use catch-all parameter set for debugging
8    -d           : use default parameter set
9    -v           : view parameter set
10   -a multitoken : add multitoken to 'ignore' list
11   -r multitoken : remove multitoken from 'ignore' list
12   -l           : view 'ignore' list
13   -i ID1 ID2 ... : print content of text entities ID1 ID2 ...
14   -s           : print size of core backend tables
15   -c F1 F2 ... : show sections F1 F2 ... in output results
16
17 To view available sections for -c command, enter -v command.
18 To view available keys for -p command, enter -v command.
19 For -i command, choose IDs from list shown in prompt results.
20 For standard prompts, enter text not starting with '-' or digit.
21 ______________________________________________________________________
22 Query, command, or integer in [0, 7] for sample query: 6
23 query: MLTxQuest Data Assets Detailed Information page
24
25   N pmi  F token [from embeddings]                      word [from prompt]
26
27   1 1.00 * confidentiality|availability                 information|assets
28   1 1.00 * availability|organization                    information|assets
29   1 1.00 * confidentiality|availability|organization    information|assets
30   1 1.00 - availability|organization|information        information|assets
31   1 1.00 * integrity|confidentiality|availability       information|assets
32   1 1.00 - organization|information                     information|assets
33   1 1.00 - organization|information|assets              information|assets
34   1 1.00 * systems|managed                              information|assets
35   1 1.00 * managed|mltxdat                              information|assets
36   1 1.00 * systems|managed|mltxdat                      information|assets
37   1 1.00 - managed|mltxdat|csa                          information|assets
38   1 1.00 - platform|against                             information|assets
39   1 1.00 * platform|against|threats                     information|assets
40   1 1.00 * threats|such                                 information|assets
41   1 1.00 * data|systems|managed                         information|assets
42   1 1.00 - csa|platform|against                         information|assets
```

```
  1 1.00 * against|threats                                 information|assets
  1 1.00 * against|threats|such                            information|assets
  1 0.71 * navigating|data                                 page|mltxquest
  1 0.71 * efficiently|navigating|data                     page|mltxquest
  1 0.71 * navigating|data|assets                          page|mltxquest
  1 0.71 - assets|page                                     page|mltxquest
  1 0.71 - data|assets|page                                page|mltxquest
  1 0.71 - page|mltxquest|while                            page|mltxquest
  1 0.71 * while|facilitating                              page|mltxquest
  1 0.71 * while|facilitating|comprehensive                page|mltxquest
  1 0.71 - assets|page|mltxquest                           page|mltxquest
  1 0.71 - mltxquest|while                                 page|mltxquest
  1 0.71 * mltxquest|while|facilitating                    page|mltxquest
  1 0.71 * facilitating|comprehensive                      page|mltxquest
  1 0.71 - assets|deta                                     page|mltxquest
  1 0.71 - information|page                                page|mltxquest
  1 0.71 - page|mltxquest|data                             page|mltxquest
  1 0.71 - information|page|mltxquest                      page|mltxquest
  1 0.71 - mltxquest|data                                  page|mltxquest
  1 0.71 * mltxquest|data|assets                           page|mltxquest
  1 0.71 * assets|users                                    page|mltxquest
  1 0.71 * data|assets|users                               page|mltxquest
  1 0.71 - mltxdat|csa|platform                            information|assets
  1 0.71 - csa|platform                                    information|assets
  2 0.67 * users|efficiently                               data|assets
  2 0.67 * efficiently|navigating                          data|assets
  2 0.67 * users|efficiently|navigating                    data|assets
  2 0.67 * aid|users|efficiently                           data|assets
  2 0.50 * global|search                                   detailed
  2 0.50 - detailed|process                                detailed
  2 0.50 * process|migrating                               detailed
  2 0.50 * detailed|process|migrating                      detailed
  2 0.50 * migrating|historical                            detailed
  2 0.50 * process|migrating|historical                    detailed
  2 0.50 - describes|detailed                              detailed
  2 0.50 - describes|detailed|process                      detailed
  2 0.47 * data|assets                                     page|mltxquest
  2 0.47 * page|mltxquest                                  data|assets
  1 0.45 - mltxdat|csa                                     information|assets
  2 0.41 - data|migration                                  detailed
  1 0.35 * guide|global                                    detailed
  1 0.35 * guide|global|search                             detailed
  1 0.35 * information|search                              detailed
  1 0.35 * search|data                                     detailed
  1 0.35 * information|search|data                         detailed

N = occurrences of (token, word) in corpus. F = * if contextual pair.
If no result, try option '-p f'.

SECTION: Category

  Category: 'Products' [6 entries]                Category: 'BI Solution' [1 entries]
  Linked to: page|mltxquest (2)                   Linked to: detailed (8)
  Linked to: detailed (8)
  Linked to: information|page|mltxquest|data (1)  Category: 'Observability & Monitoring' [1 entries]
  Linked to: data|assets (9)                      Linked to: detailed (8)
  Linked to: data|assets|page|mltxquest (1)
  Linked to: page|mltxquest|data|assets (1)       Category: 'One Platform' [1 entries]
                                                  Linked to: detailed (8)
  Category: 'Governance' [3 entries]
  Linked to: detailed (8)
  Linked to: information|assets (1)
  Linked to: data|assets (9)

SECTION: Tags

  Tags: MLTxQuest [6 entries]                       Tags: metadata [2 entries]
  Linked to: page|mltxquest (2)                     Linked to: detailed (8)
  Linked to: detailed (8)                           Linked to: data|assets (9)
  Linked to: information|page|mltxquest|data (1)
  Linked to: data|assets (9)                        Tags: mltxquest [1 entries]
  Linked to: data|assets|page|mltxquest (1)         Linked to: detailed (8)
  Linked to: page|mltxquest|data|assets (1)
                                                    Tags: business [1 entries]
  Tags: Guideline [3 entries]                       Linked to: detailed (8)
  Linked to: page|mltxquest (2)
```

```
  Linked to: data|assets (9)
  Linked to: data|assets|page|mltxquest (1)

  Tags: Guidelines [5 entries]
  Linked to: page|mltxquest (2)
  Linked to: detailed (8)
  Linked to: information|page|mltxquest|data (1)
  Linked to: data|assets (9)
  Linked to: page|mltxquest|data|assets (1)

  Tags: example1 [2 entries]
  Linked to: detailed (8)
  Linked to: data|assets (9)

  Tags: example2 [2 entries]
  Linked to: detailed (8)
  Linked to: data|assets (9)

  Tags: governance [2 entries]
  Linked to: detailed (8)
  Linked to: data|assets (9)

  Tags: roles [1 entries]
  Linked to: detailed (8)

  Tags: raci [1 entries]
  Linked to: detailed (8)

  Tags: products [1 entries]
  Linked to: detailed (8)

  Tags: metrics [1 entries]
  Linked to: detailed (8)

  Tags: Historical data [1 entries]
  Linked to: detailed (8)

  Tags: Security [1 entries]
  Linked to: information|assets (1)

  Tags: privacy [1 entries]
  Linked to: data|assets (9)

  Tags: Steward [1 entries]
  Linked to: data|assets (9)

  Tags: policy [1 entries]
  Linked to: data|assets (9)

  Tags: owner [1 entries]
  Linked to: data|assets (9)

  Tags: badge [1 entries]
  Linked to: data|assets (9)

SECTION: Titles

  Titles: 'MLTxQuest - Data Assets' [3 entries]
  Linked to: page|mltxquest (2)
  Linked to: data|assets (9)
  Linked to: data|assets|page|mltxquest (1)

  Titles: 'MLTxQuest-Data Asset Deta' [5 entries]
  Linked to: page|mltxquest (2)
  Linked to: detailed (8)
  Linked to: information|page|mltxquest|data (1)
  Linked to: data|assets (9)
  Linked to: page|mltxquest|data|assets (1)

  Titles: 'MLTxQuest - Global Search' [2 entries]
  Linked to: detailed (8)
  Linked to: data|assets (9)

  Titles: 'Roles and Responsibilities Policy' [1 entries]
  Linked to: detailed (8)

  Titles: 'Business Metadata Template' [1 entries]
  Linked to: detailed (8)

  Titles: '[METRICS] Data Products' [1 entries]
  Linked to: detailed (8)

  Titles: 'Exploration - Monitoring' [1 entries]
  Linked to: detailed (8)

  Titles: 'Historical data migration' [1 entries]
  Linked to: detailed (8)

  Titles: 'Data Security Policy ' [1 entries]
  Linked to: information|assets (1)

  Titles: 'Data Privacy Policy' [1 entries]
  Linked to: data|assets (9)

  Titles: 'Data Steward Policy' [1 entries]
  Linked to: data|assets (9)

  Titles: 'Data Owner Policy' [1 entries]
  Linked to: data|assets (9)

  Titles: 'MLTxQuest - Governance Badge' [1 entries]
  Linked to: data|assets (9)
```

```
SECTION: Entity IDs

  ID: 91 [3 entries]
  Linked to: page|mltxquest (2)
  Linked to: data|assets (9)
  Linked to: data|assets|page|mltxquest (1)

  ID: 92 [5 entries]
  Linked to: page|mltxquest (2)
  Linked to: detailed (8)
  Linked to: information|page|mltxquest|data (1)
  Linked to: data|assets (9)
  Linked to: page|mltxquest|data|assets (1)

  ID: 90 [2 entries]
  Linked to: detailed (8)
  Agents: ('Example',)
  Linked to: data|assets (9)
  Agents: ('Example',)

  ID: 101 [1 entries]
  Linked to: detailed (8)
  Agents: ('Policy', 'Governance')

  ID: 107 [1 entries]
  Linked to: detailed (8)
  Agents: ('Template', 'Governance')

  ID: 139 [1 entries]
  Linked to: detailed (8)

  ID: 381 [1 entries]
  Linked to: detailed (8)

  ID: 511 [1 entries]
  Linked to: detailed (8)
  Agents: ('Data',)

  ID: 513 [1 entries]
  Linked to: detailed (8)
  Agents: ('Data',)

  ID: 223 [1 entries]
  Linked to: information|assets (1)
  Agents: ('Policy', 'Governance')

  ID: 42 [1 entries]
  Linked to: data|assets (9)
  Agents: ('Policy', 'Governance')

  ID: 48 [1 entries]
  Linked to: data|assets (9)
  Agents: ('Policy', 'Governance')

  ID: 199 [1 entries]
  Linked to: data|assets (9)
  Agents: ('Policy', 'Governance')

  ID: 259 [1 entries]
  Linked to: data|assets (9)
  Agents: ('Governance',)

SECTION: Agents

  Agents: Example [2 entries]
  Linked to: detailed (8)
  Linked to: data|assets (9)

  Agents: Policy [3 entries]
  Linked to: detailed (8)
  Linked to: information|assets (1)
  Linked to: data|assets (9)

  Agents: Governance [3 entries]
  Linked to: detailed (8)
  Linked to: information|assets (1)
  Linked to: data|assets (9)

  Agents: Template [1 entries]
  Linked to: detailed (8)

  Agents: Data [1 entries]
  Linked to: detailed (8)

Above results based on words found in prompt, matched back to backend tables.
Numbers in parentheses are occurrences of word in corpus.


SECTION: Agent and Multitoken, with ID list
   empty unless labels 'ID' and 'Agents' are in 'show'.

  Agent       Multitoken          ID list

  Data        detailed            511, 513
  Example     data|assets         90
  Example     detailed            90
  Governance  data|assets         42, 48, 199, 259
  Governance  detailed            101, 107
  Governance  information|assets  223
  Policy      data|assets         42, 48, 199
  Policy      detailed            101
  Policy      information|assets  223
  Template    detailed            107
```

```
    ID      Size (of text entity)

    511     690
    513     692
    90      772
    42      948
    48      916
    199     980
    259     1153
    101     851
    107     1242
    223     978
__________________________________________________________________
Command menu:

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

To view available sections for -c command, enter -v command.
To view available keys for -p command, enter -v command.
For -i command, choose IDs from list shown in prompt results.
For standard prompts, enter text not starting with '-' or digit.
__________________________________________________________________
Query, command, or integer in [0, 7] for sample query: -i 107 259

Entity ID 107

   Modified Date : 2024-07-02T12:51:31.993Z
   title_text : Business Metadata Template
   description_text : It outlines detailed instructions for completing the template accurately,
       covering various sections such as data dictionary, data source, sensitivity information, and
       roles. After filling out the template, users can interpret the entered data, ensuring clarity
       on sensitivity classifications, business details, and key roles. Once completed and reviewed,
       the metadata is uploaded to MLTxQuest, making it accessible through the MLTxQuest portal for
       all authorized users, thereby centralizing and simplifying access to critical information
       within the organization.
   tags_list_text : metadata, mltxquest, business
   link_list_text :
   likes_list_text : luiz.lagatosm@abc-mixa.com
   category_text : Governance

Entity ID 259

   Modified Date : 2024-06-27T11:36:39.594Z
   title_text : MLTxQuest - Governance Badge
   description_text : The Governance Badge in MLTxQuest is awarded to data assets (tables) that
       demonstrate exceptional metadata management and data quality. To earn this badge, tables must
       meet stringent criteria, including robust technical and business metadata descriptions,
       alongside maintaining a Fitness Index score above 90 consistently. This badge signifies a
       commitment to high data governance standards, providing users with confidence in data
       accuracy and transparency in its usage.
   tags_list_text : badge, governance, metadata
   link_list_text :
   likes_list_text : luiz.lagatosm@abc-mixa.com
   category_text : Governance

2 text entities found.
Completed task: -i 107 259
__________________________________________________________________
Command menu:

  -q          : print last non-command prompt
  -x          : print sample queries
  -p key value : set frontendParams[key] = value
  -f          : use catch-all parameter set for debugging
  -d          : use default parameter set
```

```
  -v          : view parameter set
  -a multitoken : add multitoken to 'ignore' list
  -r multitoken : remove multitoken from 'ignore' list
  -l          : view 'ignore' list
  -i ID1 ID2 ... : print content of text entities ID1 ID2 ...
  -s          : print size of core backend tables
  -c F1 F2 ... : show sections F1 F2 ... in output results

To view available sections for -c command, enter -v command.
To view available keys for -p command, enter -v command.
For -i command, choose IDs from list shown in prompt results.
For standard prompts, enter text not starting with '-' or digit.
--------------------------------------------------------------------
Query, command, or integer in [0, 7] for sample query: -a detailed
Completed task: -a detailed
--------------------------------------------------------------------
Command menu:

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

To view available sections for -c command, enter -v command.
To view available keys for -p command, enter -v command.
For -i command, choose IDs from list shown in prompt results.
For standard prompts, enter text not starting with '-' or digit.
--------------------------------------------------------------------
Query, command, or integer in [0, 7] for sample query: -v

Key Description       Value

  0 embeddingKeyMinSize 1
  1 embeddingValuesMinSize 2
  2 min_pmi             0.0
  3 nABmin              1
  4 Customized_pmi      True
  5 ContextMultitokenMinSize 1
  6 minOutputListSize   1
  7 bypassIgnoreList    False
  8 ignoreList          ('data', 'detailed')
  9 maxTokenCount       100

Show sections:

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

Completed task: -v
--------------------------------------------------------------------
Command menu:

  -q          : print last non-command prompt
  -x          : print sample queries
  -p key value : set frontendParams[key] = value
  -f          : use catch-all parameter set for debugging
  -d          : use default parameter set
  -v          : view parameter set
  -a multitoken : add multitoken to 'ignore' list
  -r multitoken : remove multitoken from 'ignore' list
  -l          : view 'ignore' list
  -i ID1 ID2 ... : print content of text entities ID1 ID2 ...
```

```
  -s          : print size of core backend tables
  -c F1 F2 ... : show sections F1 F2 ... in output results

To view available sections for -c command, enter -v command.
To view available keys for -p command, enter -v command.
For -i command, choose IDs from list shown in prompt results.
For standard prompts, enter text not starting with '-' or digit.
______________________________________________________________________
Query, command, or integer in [0, 7] for sample query: -p 6 2
Completed task: -p 6 2
______________________________________________________________________
Command menu:

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

To view available sections for -c command, enter -v command.
To view available keys for -p command, enter -v command.
For -i command, choose IDs from list shown in prompt results.
For standard prompts, enter text not starting with '-' or digit.
______________________________________________________________________
Query, command, or integer in [0, 7] for sample query: -p 2 0.50
Completed task: -p 2 0.50
______________________________________________________________________
Command menu:

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

To view available sections for -c command, enter -v command.
To view available keys for -p command, enter -v command.
For -i command, choose IDs from list shown in prompt results.
For standard prompts, enter text not starting with '-' or digit.
______________________________________________________________________
Query, command, or integer in [0, 7] for sample query: 6
query: MLTxQuest Data Assets Detailed Information page

  N pmi   F token [from embeddings]                     word [from prompt]

  1 1.00 * confidentiality|availability                  information|assets
  1 1.00 * availability|organization                     information|assets
  1 1.00 * confidentiality|availability|organization     information|assets
  1 1.00 - availability|organization|information         information|assets
  1 1.00 * integrity|confidentiality|availability        information|assets
  1 1.00 - organization|information                      information|assets
  1 1.00 - organization|information|assets               information|assets
  1 1.00 * systems|managed                               information|assets
  1 1.00 * managed|mltxdat                               information|assets
  1 1.00 * systems|managed|mltxdat                       information|assets
  1 1.00 - managed|mltxdat|csa                           information|assets
  1 1.00 - platform|against                              information|assets
  1 1.00 * platform|against|threats                      information|assets
  1 1.00 * threats|such                                  information|assets
  1 1.00 * data|systems|managed                          information|assets
  1 1.00 - csa|platform|against                          information|assets
  1 1.00 * against|threats                               information|assets
```

```
  1 1.00 * against|threats|such                         information|assets
  1 0.71 * navigating|data                              page|mltxquest
  1 0.71 * efficiently|navigating|data                  page|mltxquest
  1 0.71 * navigating|data|assets                       page|mltxquest
  1 0.71 - assets|page                                  page|mltxquest
  1 0.71 - data|assets|page                             page|mltxquest
  1 0.71 - page|mltxquest|while                         page|mltxquest
  1 0.71 * while|facilitating                           page|mltxquest
  1 0.71 * while|facilitating|comprehensive             page|mltxquest
  1 0.71 - assets|page|mltxquest                        page|mltxquest
  1 0.71 - mltxquest|while                              page|mltxquest
  1 0.71 * mltxquest|while|facilitating                 page|mltxquest
  1 0.71 * facilitating|comprehensive                   page|mltxquest
  1 0.71 - assets|deta                                  page|mltxquest
  1 0.71 - information|page                             page|mltxquest
  1 0.71 - page|mltxquest|data                          page|mltxquest
  1 0.71 - information|page|mltxquest                   page|mltxquest
  1 0.71 - mltxquest|data                               page|mltxquest
  1 0.71 * mltxquest|data|assets                        page|mltxquest
  1 0.71 * assets|users                                 page|mltxquest
  1 0.71 * data|assets|users                            page|mltxquest
  1 0.71 - mltxdat|csa|platform                         information|assets
  1 0.71 - csa|platform                                 information|assets
  2 0.67 * users|efficiently                            data|assets
  2 0.67 * efficiently|navigating                       data|assets
  2 0.67 * users|efficiently|navigating                 data|assets
  2 0.67 * aid|users|efficiently                        data|assets

N = occurrences of (token, word) in corpus. F = * if contextual pair.
If no result, try option '-p f'.

SECTION: Category

  Category: 'Products' [5 entries]
  Linked to: page|mltxquest (2)
  Linked to: information|page|mltxquest|data (1)
  Linked to: data|assets (9)
  Linked to: data|assets|page|mltxquest (1)
  Linked to: page|mltxquest|data|assets (1)

  Category: 'Governance' [2 entries]
  Linked to: information|assets (1)
  Linked to: data|assets (9)

SECTION: Tags

  Tags: MLTxQuest [5 entries]
  Linked to: page|mltxquest (2)
  Linked to: information|page|mltxquest|data (1)
  Linked to: data|assets (9)
  Linked to: data|assets|page|mltxquest (1)
  Linked to: page|mltxquest|data|assets (1)

  Tags: Guideline [3 entries]
  Linked to: page|mltxquest (2)
  Linked to: data|assets (9)
  Linked to: data|assets|page|mltxquest (1)

  Tags: Guidelines [4 entries]
  Linked to: page|mltxquest (2)
  Linked to: information|page|mltxquest|data (1)
  Linked to: data|assets (9)
  Linked to: page|mltxquest|data|assets (1)

SECTION: Titles

  Titles: 'MLTxQuest - Data Assets' [3 entries]
  Linked to: page|mltxquest (2)
  Linked to: data|assets (9)
  Linked to: data|assets|page|mltxquest (1)

  Titles: 'MLTxQuest-Data Asset Deta' [4 entries]
  Linked to: page|mltxquest (2)
  Linked to: information|page|mltxquest|data (1)
  Linked to: data|assets (9)
  Linked to: page|mltxquest|data|assets (1)
```

```

SECTION: Entity IDs

   ID: 91 [3 entries]
   Linked to: page|mltxquest (2)
   Linked to: data|assets (9)
   Linked to: data|assets|page|mltxquest (1)

   ID: 92 [4 entries]
   Linked to: page|mltxquest (2)
   Linked to: information|page|mltxquest|data (1)
   Linked to: data|assets (9)
   Linked to: page|mltxquest|data|assets (1)

SECTION: Agents

   Agents: Policy [2 entries]
   Linked to: information|assets (1)
   Linked to: data|assets (9)

   Agents: Governance [2 entries]
   Linked to: information|assets (1)
   Linked to: data|assets (9)

Above results based on words found in prompt, matched back to backend tables.
Numbers in parentheses are occurrences of word in corpus.

SECTION: (Agent, Multitoken) --> (ID list)

    empty unless labels 'ID' and 'Agents' are in 'show'.
______________________________________________________________________
Command menu:

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

To view available sections for -c command, enter -v command.
To view available keys for -p command, enter -v command.
For -i command, choose IDs from list shown in prompt results.
For standard prompts, enter text not starting with '-' or digit.
______________________________________________________________________
Query, command, or integer in [0, 7] for sample query: -c Titles
Completed task: -c Titles
______________________________________________________________________
Command menu:

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

To view available sections for -c command, enter -v command.
To view available keys for -p command, enter -v command.
For -i command, choose IDs from list shown in prompt results.
For standard prompts, enter text not starting with '-' or digit.
______________________________________________________________________
Query, command, or integer in [0, 7] for sample query: 6
query: MLTxQuest Data Assets Detailed Information page
```

```
SECTION: Titles

  Titles: 'MLTxQuest - Data Assets' [3 entries]
  Linked to: page|mltxquest (2)
  Linked to: data|assets (9)
  Linked to: data|assets|page|mltxquest (1)

  Titles: 'MLTxQuest-Data Asset Deta' [4 entries]
  Linked to: page|mltxquest (2)
  Linked to: information|page|mltxquest|data (1)
  Linked to: data|assets (9)
  Linked to: page|mltxquest|data|assets (1)

Above results based on words found in prompt, matched back to backend tables.
Numbers in parentheses are occurrences of word in corpus.

SECTION: (Agent, Multitoken) --> (ID list)

   empty unless labels 'ID' and 'Agents' are in 'show'.
____________________________________________________________________
Command menu:

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

To view available sections for -c command, enter -v command.
To view available keys for -p command, enter -v command.
For -i command, choose IDs from list shown in prompt results.
For standard prompts, enter text not starting with '-' or digit.
____________________________________________________________________
Query, command, or integer in [0, 7] for sample query:
____________________________________________________________________
```

3.3 Web API for enterprise xLLM

A web API is available here on xllm.GenAItechLab.com to test the application. The implementation is slightly different from the offline version. But it is based on the same anonymized corporate corpus, dealing with ML and AI policies, integration, definitions, best practices, and references, for corporate users (employees). See how it looks like in Figure 13.

3.3.1 Left panel: command menu and prompt box

The left panel allows you to fine-tune the front-end parameters in real time, and to enter your prompt at the bottom: either from pre-selected queries with the option `Seeded`, or your own prompt with the option `Custom`. The right panel shows the prompt results. The front-end parameters are the same as in the offline version (see Figure 13) except `show` options that are organized differently, customizable on the right panel.

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

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

3.3.2 Right panel: prompt results

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

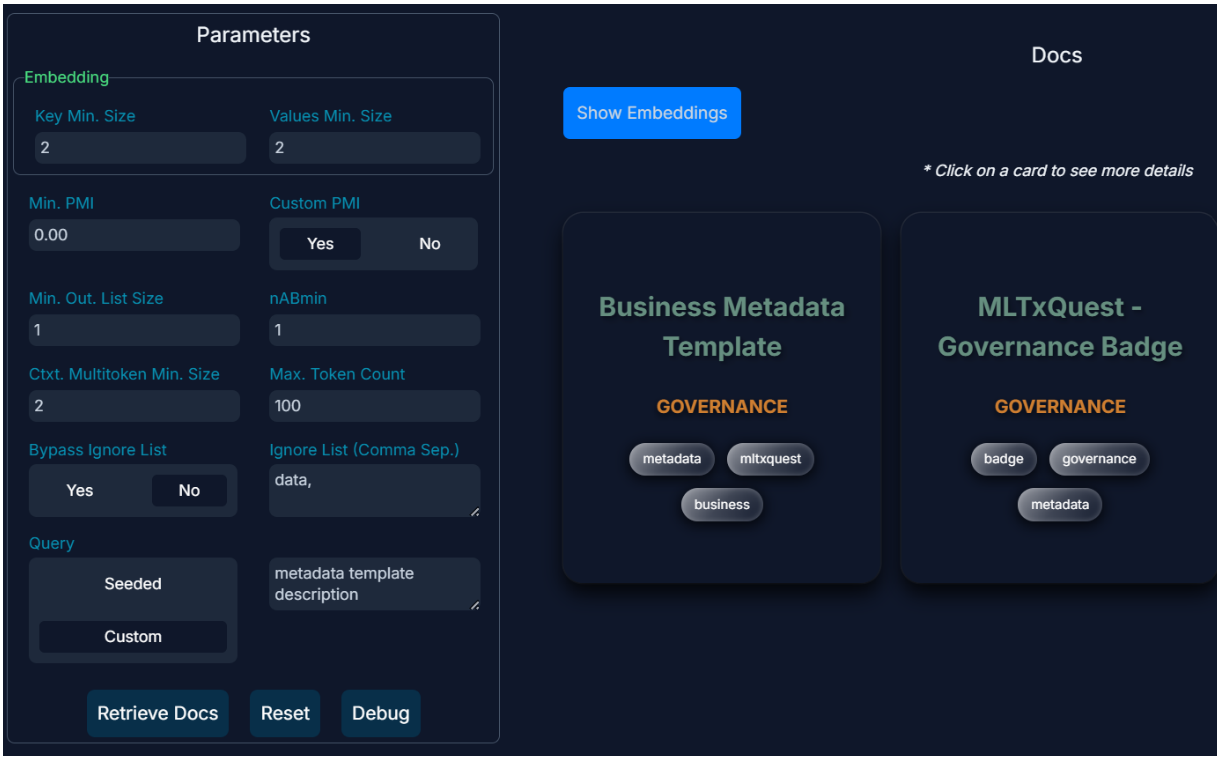


Figure 13: Web API for enterprise xLLM, with prompt results for ‘metadata template description’

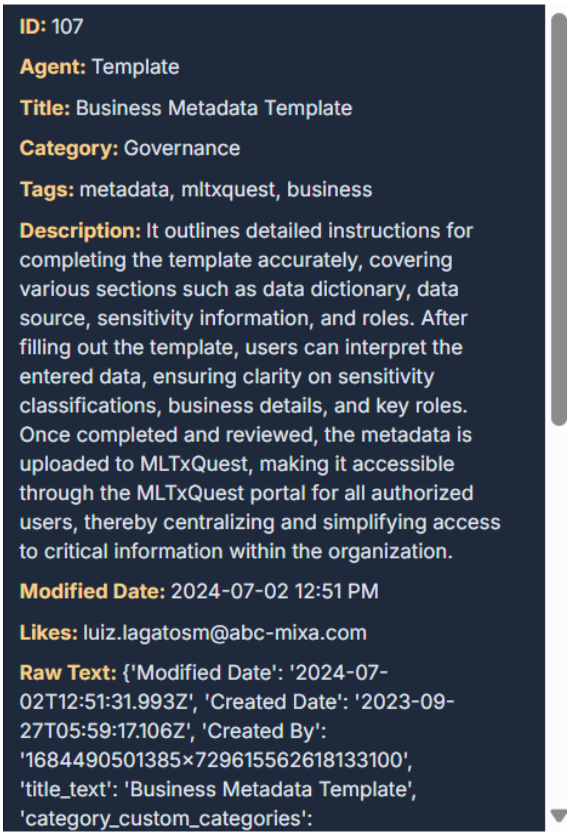
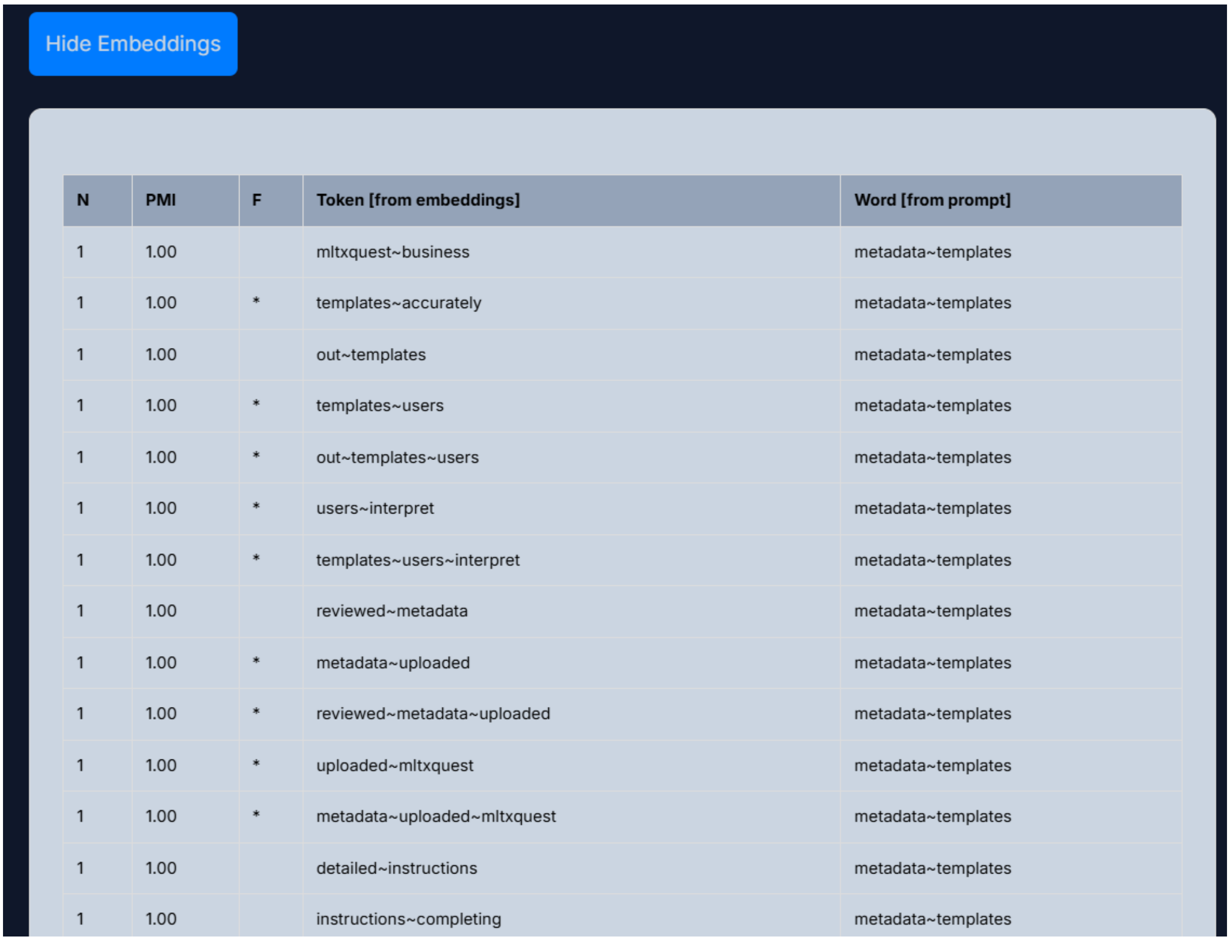


Figure 14: First item returned: details

In our example, two items were retrieved, respectively 'Business Metadata Template' and 'MLTxQuest Governance Badge'. For each item, the green, orange and white fonts represent respectively the title, category, and related tags. If you click on any item, more details show up: see Figure 14. You can expand to retrieve the full raw text: in this case, a JSON entry in the corpus (not shown by default). Also note the text entity ID to match back to the corpus, as well as triggered agents, at the top in Figure 14.



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

Figure 15: Top embedding entries for 'metadata template description'

Finally, you can check out embedding entries related to your prompt, by clicking on the `Show Embeddings` blue box visible in Figure 13. See top embedding entries in Figure 15, for 'metadata template description' using the default parameter set. The 'word' column shows multitokens extracted from the prompt, while the 'token' column represents multitokens from the backend tables, related to the 'word' in question. Multitokens flagged with a (*) are contextually related to the 'word' in question, instead of just based on immediate proximity. The PMI measures the strength of the association, while the leftmost column is another indicator of relevancy. The associations in question may come from different text entities, or from the knowledge graph itself in version 3. These embedding entries are useful to try additional prompts to refine your search, or for debugging purposes.

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

3.3.3 Next steps

The following features will be added:

- Incorporation of acronyms in the `KW_map` table, for instance to redirect 'Doing Business As' to 'DBA' if the former is found in a prompt, but not in the corpus.
- A second `dictionary` table (or alternate mechanism) for multitokens found in knowledge graph entities: categories, titles, tags, agents, and so on. The end goal is to boost these multitokens, as they have more importance and are of higher quality. In the end, to produce better relevancy scores.

- Working with contextual multitokens, consisting of non-adjacent words found together in a same text sub-entity.
- Data augmentation and more agents, with fewer text entities lacking agents.
- Breaking prompts into sub-prompts. More NLP: stemming, auto-correct, and so on.

3.4 Conclusions and references

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

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

Finally, gather a large number of user queries even before your start designing your architecture, and add prompt elements into your backend tables, as a source of data augmentation. It contributes to enhancing the quality of your system. For additional references, see [6] on mixture of experts, [3] on multitokens, [7, 8] on LLM evaluation, [9] on building your LLM from scratch, [1] on reducing LLM costs, and [2] on variable length embeddings.

4 Appendix

4.1 Python code

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

```
#--- [1] Backend: functions

def update_hash(hash, key, count=1):

   if key in hash:
      hash[key] += count
   else:
      hash[key] = count
   return(hash)


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

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


def get_value(key, hash):
   if key in hash:
```

```
        value = hash[key]
    else:
        value = ''
    return(value)


def update_tables(backendTables, word, hash_crawl, backendParams):

    category    = get_value('category', hash_crawl)
    tag_list    = get_value('tag_list', hash_crawl)
    title       = get_value('title', hash_crawl)
    description = get_value('description', hash_crawl) #
    meta        = get_value('meta', hash_crawl)
    ID          = get_value('ID', hash_crawl)
    agents      = get_value('agents', hash_crawl)
    full_content = get_value('full_content', hash_crawl) #

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

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

    return(backendTables)


def clean_list(value):

    # change string "['a', 'b', ...]" to ('a', 'b', ...)
    value = value.replace("[", "").replace("]","")
    aux = value.split("~")
    value_list = ()
    for val in aux:
       val = val.replace("'","").replace('"',"").lstrip()
       if val != '':
          value_list = (*value_list, val)
    return(value_list)


def get_key_value_pairs(entity):

    # extract key-value pairs from 'entity' (a string)
    entity = entity[1].replace("}",", '")
    flag = False
    entity2 = ""

    for idx in range(len(entity)):
        if entity[idx] == '[':
            flag = True
        elif entity[idx] == ']':
            flag = False
        if flag and entity[idx] == ",":
            entity2 += "~"
        else:
            entity2 += entity[idx]

    entity = entity2
```

```
   key_value_pairs = entity.split(", '")
   return(key_value_pairs)


def update_dict(backendTables, hash_crawl, backendParams):

   max_multitoken = backendParams['max_multitoken']
   maxDist = backendParams['maxDist']
   maxTerms = backendParams['maxTerms']

   category = get_value('category', hash_crawl)
   tag_list = get_value('tag_list', hash_crawl)
   title = get_value('title', hash_crawl)
   description = get_value('description', hash_crawl)
   meta = get_value('meta', hash_crawl)

   text = category + "." + str(tag_list) + "." + title + "." + description + "." + meta
   text = text.replace('/'," ").replace('(',' ').replace(')',' ').replace('?','')
   text = text.replace("'","").replace('"',"").replace('\\n','').replace('!','')
   text = text.replace("\\s",'').replace("\\t",'').replace(","," ").replace(":"," ")
   text = text.lower()
   sentence_separators = ('.',)
   for sep in sentence_separators:
      text = text.replace(sep, '_~')
   text = text.split('_~')

   hash_pairs = backendTables['hash_pairs']
   ctokens = backendTables['ctokens']
   KW_map = backendTables['KW_map']
   stopwords = backendTables['stopwords']
   hwords = {} # local word hash with word position, to update hash_pairs

   for sentence in text:

      words = sentence.split(" ")
      position = 0
      buffer = []

      for word in words:

         if word in KW_map:
            word = KW_map[word]

         if word not in stopwords:
            # word is single token
            buffer.append(word)
            key = (word, position)
            update_hash(hwords, key) # for word correlation table (hash_pairs)
            update_tables(backendTables, word, hash_crawl, backendParams)

            for k in range(1, max_multitoken):
               if position > k:
                  # word is now multi-token with k+1 tokens
                  word = buffer[position-k] + "~" + word
                  key = (word, position)
                  update_hash(hwords, key) # for word correlation table (hash_pairs)
                  update_tables(backendTables, word, hash_crawl, backendParams)

            position +=1

   for keyA in hwords:
      for keyB in hwords:

         wordA = keyA[0]
         positionA = keyA[1]
         n_termsA = len(wordA.split("~"))

         wordB = keyB[0]
         positionB = keyB[1]
         n_termsB = len(wordB.split("~"))

         key = (wordA, wordB)
         n_termsAB = max(n_termsA, n_termsB)
         distanceAB = abs(positionA - positionB)

         if wordA < wordB and distanceAB <= maxDist and n_termsAB <= maxTerms:
```

```
            hash_pairs = update_hash(hash_pairs, key)
            if distanceAB > 1:
               ctokens = update_hash(ctokens, key)

   return(backendTables)


#--- [2] Backend: main (create backend tables based on crawled corpus)

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

backendTables = {}
for name in tableNames:
   backendTables[name] = {}

stopwords = ('', '-', 'in', 'the', 'and', 'to', 'of', 'a', 'this', 'for', 'is', 'with', 'from',
          'as', 'on', 'an', 'that', 'it', 'are', 'within', 'will', 'by', 'or', 'its', 'can',
          'your', 'be','about', 'used', 'our', 'their', 'you', 'into', 'using', 'these',
          'which', 'we', 'how', 'see', 'below', 'all', 'use', 'across', 'provide', 'provides',
          'aims', 'one', '&', 'ensuring', 'crucial', 'at', 'various', 'through', 'find', 'ensure',
          'more', 'another', 'but', 'should', 'considered', 'provided', 'must', 'whether',
          'located', 'where', 'begins', 'any')
backendTables['stopwords'] = stopwords

# agent_map works, but hash structure should be improved
# key is word, value is agent (many-to-one). Allow for many-to-many
agent_map = {
          'template':'Template',
          'policy':'Policy',
          'governance':'Governance',
          'documentation':'Documentation',
          'best practice':'Best Practices',
          'bestpractice':'Best Practices',
          'standard':'Standards',
          'naming':'Naming',
          'glossary':'Glossary',
          'historical data':'Data',
          'overview':'Overview',
          'training':'Training',
          'genai':'GenAI',
          'gen ai':'GenAI',
          'example':'Example',
          'example1':'Example',
          'example2':'Example',
         }

KW_map = {}
try:
   IN = open("KW_map.txt","r")
except:
   print("KW_map.txt not found on first run: working with empty KW_map.")
   print("KW_map.txt will be created after exiting if save = True.")
else:
   content = IN.read()
   pairs = content.split('\n')
   for pair in pairs:
      pair = pair.split('\t')
      key = pair[0]
      if len(pair) > 1:
```

```
         KW_map[key] = pair[1]
   IN.close()
backendTables['KW_map'] = KW_map

backendParams = {
   'max_multitoken': 4, # max. consecutive terms per multi-token for inclusion in dictionary
   'maxDist' : 3,  # max. position delta between 2 multitokens to link them in hash_pairs
   'maxTerms': 3,  # maxTerms must be <= max_multitoken
   'extraWeights' : # deafault weight is 1
     {
       'description': 0.0,
       'category': 0.3,
       'tag_list': 0.4,
       'title':   0.2,
       'meta':    0.1
     }
}


local = True # first time run, set to False
if local:
   # get repository from local file
   IN = open("repository.txt","r")
   data = IN.read()
   IN.close()
else:
   # get anonymized repository from GitHub url
   import requests
   url = "https://mltblog.com/3y8MXq5"
   response = requests.get(url)
   data = response.text

entities = data.split("\n")
ID_size = backendTables['ID_size']

# to avoid duplicate entities (takes space, better to remove them in the corpus)
entity_list = ()

for entity_raw in entities:

   entity = entity_raw.split("~~")
   agent_list = ()

   if len(entity) > 1 and entity[1] not in entity_list:

      entity_list = (*entity_list, entity[1])
      entity_ID = int(entity[0])
      entity = entity[1].split("{")
      hash_crawl = {}
      hash_crawl['ID'] = entity_ID
      ID_size[entity_ID] = len(entity[1])
      hash_crawl['full_content'] = entity_raw # do not build to save space

      key_value_pairs = get_key_value_pairs(entity)

      for pair in key_value_pairs:

         if ": " in pair:
            key, value = pair.split(": ", 1)
            key = key.replace("'","")
            if key == 'category_text':
               hash_crawl['category'] = value
            elif key == 'tags_list_text':
               hash_crawl['tag_list'] = clean_list(value)
            elif key == 'title_text':
               hash_crawl['title'] = value
            elif key == 'description_text':
               hash_crawl['description'] = value # do not build to save space
            elif key == 'tower_option_tower':
               hash_crawl['meta'] = value
            if key in ('category_text','tags_list_text','title_text'):
               for word in agent_map:
                  if word in value.lower():
                     agent = agent_map[word]
                     if agent not in agent_list:
                        agent_list =(*agent_list, agent)
```

```
      hash_crawl['agents'] = agent_list
      update_dict(backendTables, hash_crawl, backendParams)

# [2.1] Create embeddings

embeddings = {} # multitoken embeddings based on hash_pairs

hash_pairs = backendTables['hash_pairs']
dictionary = backendTables['dictionary']

for key in hash_pairs:
   wordA = key[0]
   wordB = key[1]
   nA = dictionary[wordA]
   nB = dictionary[wordB]
   nAB = hash_pairs[key]
   pmi = nAB/(nA*nB)**0.5 # try: nAB/(nA + nB - nAB)
   # if nA + nB <= nAB:
   #   print(key, nA, nB, nAB)
   update_nestedHash(embeddings, wordA, wordB, pmi)
   update_nestedHash(embeddings, wordB, wordA, pmi)


# [2.2] Create sorted n-grams

sorted_ngrams = {} # to match ngram prompts with embeddings entries

for word in dictionary:
   tokens = word.split('~')
   tokens.sort()
   sorted_ngram = tokens[0]
   for token in tokens[1:len(tokens)]:
      sorted_ngram += "~" + token
   update_nestedHash(sorted_ngrams, sorted_ngram, word)

# print top multitokens: useful to build agents, along with sample prompts
# for key in dictionary:
#    if dictionary[key] > 20:
#       print(key, dictionary[key])


#--- [3] Frontend: functions

# [3.1] custom pmi

def custom_pmi(word, token, backendTables):

   dictionary = backendTables['dictionary']
   hash_pairs = backendTables['hash_pairs']

   nAB = 0
   pmi = 0.00
   keyAB = (word, token)
   if word > token:
      keyAB = (token, word)
   if keyAB in hash_pairs:
      nAB = hash_pairs[keyAB]
      nA = dictionary[word]
      nB = dictionary[token]
      pmi = nAB/(nA*nB)**0.5
   return(pmi)

# [3.2] update frontend params

def cprint(ID, entity):
   # print text_entity (a JSON text string) nicely

   print("--- Entity %d ---\n" %(ID))
   keys = (
          'title_text',
          'description_text',
          'tags_list_text',
          'category_text',
          'likes_list_text',
          'link_list_text',
```

```
          'Modified Date',
         )
   entity = str(entity).split("~~")
   entity = entity[1].split("{")
   key_value_pairs = get_key_value_pairs(entity)

   for pair in key_value_pairs:
      if ": " in pair:
         key, value = pair.split(": ", 1)
         key = key.replace("'","")
         if key in keys:
            print("> ",key,":")
            value = value.replace("'",'').split("~")
            for item in value:
               item = item.lstrip().replace("[","").replace("]","")
               print(item)
            print()
   return()

def update_params(option, saved_query, sample_queries, frontendParams, backendTables):

   arr = []
   ID_to_content = backendTables['ID_to_content']
   for param in frontendParams:
      arr.append(param)
   task = option
   print()

   if option == '-l':
      print("Multitoken ignore list:\n", frontendParams['ignoreList'])

   elif option == '-v':
      print("%3s %s %s\n" %('Key', 'Description'.ljust(25), 'Value'))
      for key in range(len(arr)):
         param = arr[key]
         value = frontendParams[param]
         if param != 'show':
            print("%3d %s %s" %(key, param.ljust(25), value))
         else:
            print("\nShow sections:\n")
            for section in value:
               print("  %s %s" %(section.ljust(10),value[section]))

   elif option == '-f':
      # use parameter set to show as much as possible
      for param in frontendParams:
         if param == 'ignoreList':
            frontendParams[param] = ()
         elif param == 'Customized_pmi':
            # use customized pmi
            frontendParams[param] = True
         elif param == 'show':
            showHash = frontendParams[param]
            for section in showHash:
               # show all sections in output results
               showHash[section] = True
         elif param == 'maxTokenCount':
            frontendParams[param] = 999999999
         else:
            frontendParams[param] = 0

   elif option == '-d':
      frontendParams = default_frontendParams()

   elif '-p' in option:
      option = option.split(' ')
      if len(option) == 3:
         paramID = int(option[1])
         if paramID < len(arr):
            param = arr[paramID]
            value = option[2]
            if value == 'True':
               value = True
            elif value == 'False':
               value = False
            else:
```

```
               value = float(option[2])
            frontendParams[param] = value
         else:
            print("Error 101: key outside range")
      else:
         print("Error 102: wrong number of arguments")

   elif '-a' in option:
      option = option.split(' ')
      if len(option) == 2:
         ignore = frontendParams['ignoreList']
         ignore =(*ignore, option[1])
         frontendParams['ignoreList'] = ignore
      else:
         print("Error 103: wrong number of arguments")

   elif '-r' in option:
      option = option.split(' ')
      if len(option) == 2:
         ignore2 = ()
         ignore = frontendParams['ignoreList']
         for item in ignore:
            if item != option[1]:
               ignore2 = (*ignore2, item)
         frontendParams['ignoreList'] = ignore2
      else:
         print("Error 104: wrong number of arguments")

   elif '-i' in option:
      option = option.split(' ')
      nIDs = 0
      for ID in option:
         if ID.isdigit():
            ID = int(ID)
            # print content of text entity ID
            if ID in ID_to_content:
               cprint(ID, ID_to_content[ID])
               nIDs += 1
      print("\n %d text entities found." % (nIDs))

   elif option == '-s':
      print("Size of some backend tables:")
      print(" dictionary:", len(backendTables['dictionary']))
      print(" pairs    :", len(backendTables['hash_pairs']))
      print(" ctokens :", len(backendTables['ctokens']))
      print(" ID_size :", len(backendTables['ID_size']))

   elif '-c' in option:
      show = frontendParams['show']
      option = option.split(' ')
      for section in show:
         if section in option or '*' in option:
            show[section] = True
         else:
            show[section] = False

   elif option == '-q':
      print("Saved query:", saved_query)

   elif option == '-x':
      print("Index Query\n")
      for k in range(len(sample_queries)):
         print(" %3d %s" %(k, sample_queries[k]))

   print("\nCompleted task: %s" %(task))
   return(frontendParams)

# [3.3] retrieve info and print results

def print_results(q_dictionary, q_embeddings, backendTables, frontendParams):

   dictionary = backendTables['dictionary']
   hash_pairs = backendTables['hash_pairs']
   ctokens   = backendTables['ctokens']
   ID_to_agents = backendTables['ID_to_agents']
   ID_size   = backendTables['ID_size']
```

```
   show      = frontendParams['show']

   if frontendParams['bypassIgnoreList'] == True:
      # bypass 'ignore' list
      ignore = ()
   else:
      # ignore multitokens specified in 'ignoreList'
      ignore = frontendParams['ignoreList']

   if show['Embeddings']:
      # show results from embedding table

      local_hash = {} # used to not show same token 2x (linked to 2 different words)
      q_embeddings = dict(sorted(q_embeddings.items(),key=lambda item: item[1],reverse=True))
      print()
      print("%3s %s %1s %s %s"
          %('N','pmi'.ljust(4),'F','token [from embeddings]'.ljust(35),
           'word [from prompt]'.ljust(35)))
      print()

      for key in q_embeddings:

         word = key[0]
         token = key[1]
         pmi = q_embeddings[key]
         ntk1 = len(word.split('~'))
         ntk2 = len(token.split('~'))
         flag = " "
         nAB = 0
         keyAB = (word, token)

         if word > token:
            keyAB = (token, word)
         if keyAB in hash_pairs:
            nAB = hash_pairs[keyAB]
         if keyAB in ctokens:
            flag = '*'
         if ( ntk1 >= frontendParams['embeddingKeyMinSize'] and
              ntk2 >= frontendParams['embeddingValuesMinSize'] and
              pmi >= frontendParams['min_pmi'] and
              nAB >= frontendParams['nABmin'] and
              token not in local_hash and word not in ignore
            ):
            print("%3d %4.2f %1s %s %s"
                %(nAB,pmi,flag,token.ljust(35),word.ljust(35)))
            local_hash[token] = 1 # token marked as displayed, won't be shown again

      print()
      print("N = occurrences of (token, word) in corpus. F = * if contextual pair.")
      print("If no result, try option '-p f'.")
      print()

   sectionLabels = {
     # map section label to corresponding backend table name
     'Dict'  :'dictionary',
     'Pairs':'hash_pairs',
     'Category':'hash_context1',
     'Tags'  :'hash_context2',
     'Titles':'hash_context3',
     'Descr.':'hash_context4',
     'Meta'  :'hash_context5',
     'ID'   :'hash_ID',
     'Agents':'hash_agents',
     'Whole' :'full_content',
   }
   local_hash = {}
   agentAndWord_to_IDs = {}

   for label in show:
      # labels: 'Category','Tags','Titles','Descr.','ID','Whole','Agents','Embeddings'

      if show[label] and label in sectionLabels:
         # show results for section corresponding to label

         tableName = sectionLabels[label]
         table = backendTables[tableName]
```

```
         local_hash = {}
         print(">>> RESULTS - SECTION: %s\n" % (label))

         for word in q_dictionary:

            ntk3 = len(word.split('~'))
            if word not in ignore and ntk3 >= frontendParams['ContextMultitokenMinSize']:
               content = table[word] # content is a hash
               count = int(dictionary[word])
               for item in content:
                  update_nestedHash(local_hash, item, word, count)

         for item in local_hash:

            hash2 = local_hash[item]
            if len(hash2) >= frontendParams['minOutputListSize']:
               print(" %s: %s [%d entries]" % (label, item, len(hash2)))
               for key in hash2:
                  print(" Linked to: %s (%s)" %(key, hash2[key]))
                  if label == 'ID' and item in ID_to_agents:
                     # here item is a text entity ID
                     LocalAgentHash = ID_to_agents[item]
                     local_ID_list = ()
                     for ID in LocalAgentHash:
                        local_ID_list = (*local_ID_list, ID)
                     print(" Agents:", local_ID_list)
                     for agent in local_ID_list:
                        key3 = (agent, key) # key is a multitoken
                        update_nestedHash(agentAndWord_to_IDs, key3, item)

               print()
         print()

   print("Above results based on words found in prompt, matched back to backend tables.")
   print("Numbers in parentheses are occurrences of word in corpus.\n")

   print("-------------------------------------")
   print(">>> RESULTS - SECTION: (Agent, Multitoken) --> (ID list)")
   print(" empty unless labels 'ID' and 'Agents' are in 'show'.\n")
   hash_size = {}
   for key in sorted(agentAndWord_to_IDs):
      ID_list = ()
      for ID in agentAndWord_to_IDs[key]:
         ID_list = (*ID_list, ID)
         hash_size[ID] = ID_size[ID]
      print(key,"-->",ID_list)
   print("\n ID Size\n")
   for ID in hash_size:
      print("%4d %5d" %(ID, hash_size[ID]))

   return()


#--- [4] Frontend: main (process prompt)

# [4.1] Set default parameters

def default_frontendParams():

   frontendParams = {
                 'embeddingKeyMinSize': 1, # try 2
                 'embeddingValuesMinSize': 2,
                 'min_pmi': 0.00,
                 'nABmin': 1,
                 'Customized_pmi': True,
                 'ContextMultitokenMinSize': 1, # try 2
                 'minOutputListSize': 1,
                 'bypassIgnoreList': False,
                 'ignoreList': ('data',),
                 'maxTokenCount': 100, # ignore generic tokens if large enough
                 'show': {
                         # names of sections to display in output results
                         'Embeddings': True,
                         'Category'  : True,
                         'Tags'    : True,
                         'Titles'  : True,
```

```
                         'Descr.' : False, # do not built to save space
                         'Whole'  : False, # do not build to save space
                         'ID'     : True,
                         'Agents' : True,
                        }
                 }
   return(frontendParams)

# [4.2] Purge function

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

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

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

   return(q_dictionary, q_embeddings)

# [4.3] Main

print("\n") #
input_ = " "
saved_query = ""
get_bin = lambda x, n: format(x, 'b').zfill(n)
frontendParams = default_frontendParams()
sample_queries = (
               'parameterized datasets map tables sql server',
               'data load templates importing data database data warehouse',
               'pipeline extract data eventhub files',
               'blob storage single parquet file adls gen2',
               'eventhub files blob storage single parquet',
               'parquet blob eventhub more files less storage single table',
               'MLTxQuest Data Assets Detailed Information page',
               'table asset',
             )

while len(input_) > 0:

   print()
   print("-------------------------------------")
   print("Command menu:\n")
   print(" -q          : print last non-command prompt")
   print(" -x          : print sample queries")
   print(" -p key value : set frontendParams[key] = value")
   print(" -f          : use catch-all parameter set for debugging")
   print(" -d          : use default parameter set")
   print(" -v          : view parameter set")
   print(" -a multitoken : add multitoken to 'ignore' list")
   print(" -r multitoken : remove multitoken from 'ignore' list")
   print(" -l          : view 'ignore' list")
   print(" -i ID1 ID2 ... : print content of text entities ID1 ID2 ...")
   print(" -s          : print size of core backend tables")
   print(" -c F1 F2 ... : show sections F1 F2 ... in output results")
   print("\nTo view available sections for -c command, enter -v command.")
   print("To view available keys for -p command, enter -v command.")
   print("For -i command, choose IDs from list shown in prompt results.")
   print("For standard prompts, enter text not starting with '-' or digit.")
```

```
print("-------------------------------------\n")

input_ = input("Query, command, or integer in [0, %d] for sample query: "
            %(len(sample_queries)-1))
flag = True # False --> query to change params, True --> real query
if input_ != "" and input_[0] == '-':
      # query to modify options
      frontendParams = update_params(input_, saved_query,
                             sample_queries, frontendParams,
                             backendTables)
      query = ""
      flag = False
elif input_.isdigit():
   # actual query (prompt)
   if int(input_) < len(sample_queries):
     query = sample_queries[int(input_)]
     saved_query = query
     print("query:",query)
   else:
     print("Value must be <", len(sample_queries))
     query = ""
else:
   # actual query (prompt)
   query = input_
   saved_query = query

query = query.split(' ')
new_query = []
for k in range(len(query)):
   token = query[k].lower()
   if token in KW_map:
      token = KW_map[token]
   if token in dictionary:
      new_query.append(token)
query = new_query.copy()
query.sort()
q_embeddings = {}
q_dictionary = {}

for k in range(1, 2**len(query)):

   binary = get_bin(k, len(query))
   sorted_word = ""
   for k in range(0, len(binary)):
      if binary[k] == '1':
         if sorted_word == "":
            sorted_word = query[k]
         else:
            sorted_word += "~" + query[k]

   if sorted_word in sorted_ngrams:
      ngrams = sorted_ngrams[sorted_word]
      for word in ngrams:
         if word in dictionary:
            q_dictionary[word] = dictionary[word]
            if word in embeddings:
               embedding = embeddings[word]
               for token in embedding:
                  if not frontendParams['Customized_pmi']:
                     pmi = embedding[token]
                  else:
                     # customized pmi
                     pmi = custom_pmi(word, token, backendTables)
                  q_embeddings[(word, token)] = pmi

# if len(query) == 1:
#   # single-token query
#   frontendParams['embeddingKeyMinSize'] = 1
#   frontendParams['ContextMultitokenMinSize'] = 1

distill_frontendTables(q_dictionary,q_embeddings,frontendParams)

if len(input_) > 0 and flag:
   print_results(q_dictionary, q_embeddings, backendTables, frontendParams)
```

```python
#--- [5] Save backend tables

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

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

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


save = True
if save:
   create_KW_map(dictionary)
   for tableName in backendTables:
      table = backendTables[tableName]
      OUT = open('backend_' + tableName + '.txt', "w")
      OUT.write(str(table))
      OUT.close()

   OUT = open('backend_embeddings.txt', "w")
   OUT.write(str(embeddings))
   OUT.close()

   OUT = open('backend_sorted_ngrams.txt', "w")
   OUT.write(str(sorted_ngrams))
   OUT.close()
```

4.2 Thirty features to boost LLM performance

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

4.2.1 Fast search and caching

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

4.2.2 Leveraging sparse databases

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

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

4.2.3 Contextual tokens

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

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

4.2.4 Adaptive loss function

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

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

4.2.5 Contextual tables

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

4.2.6 Smart crawling

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

4.2.7 LLM router, sub-LLMs, and distributed architecture

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

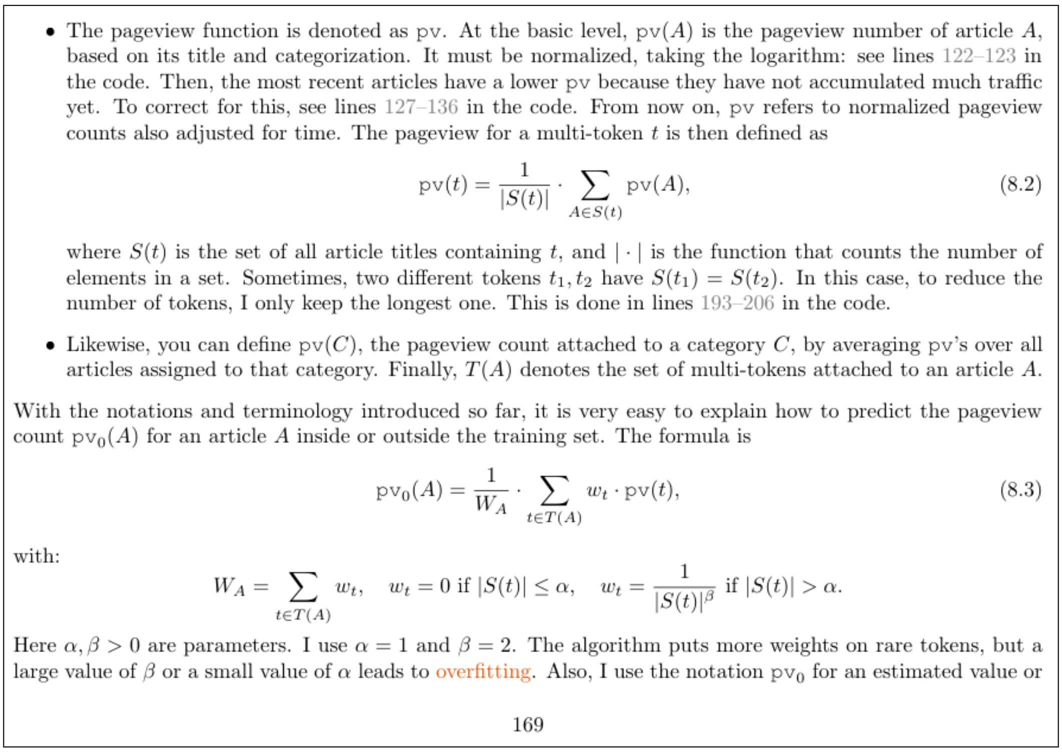
To achieve this, you need an LLM router. It identifies prompt elements and retrieve the relevant information in the most appropriate sub-LLMs. Each one hast its set of backend tables, hyperparameters, stopword list, and so on. There may be overlap between different sub-LLMs. Fine-tuning can be done locally,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

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

4.2.8 From one trillion parameters down to two

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

Blackbox parameters — the weights themselves — over and over. I illustrate this in Figure 16, featuring material from my coursebook, available here.



- The pageview function is denoted as pv. At the basic level, $\text{pv}(A)$ is the pageview number of article A, based on its title and categorization. It must be normalized, taking the logarithm: see lines 122–123 in the code. Then, the most recent articles have a lower pv because they have not accumulated much traffic yet. To correct for this, see lines 127–136 in the code. From now on, pv refers to normalized pageview counts also adjusted for time. The pageview for a multi-token t is then defined as

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

where $S(t)$ is the set of all article titles containing t, and $|\cdot|$ is the function that counts the number of elements in a set. Sometimes, two different tokens t_1, t_2 have $S(t_1) = S(t_2)$. In this case, to reduce the number of tokens, I only keep the longest one. This is done in lines 193–206 in the code.

- Likewise, you can define $\text{pv}(C)$, the pageview count attached to a category C, by averaging pv's over all articles assigned to that category. Finally, $T(A)$ denotes the set of multi-tokens attached to an article A.

With the notations and terminology introduced so far, it is very easy to explain how to predict the pageview count $\text{pv}_0(A)$ for an article A inside or outside the training set. The formula is

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

with:

$$W_A = \sum_{t \in T(A)} w_t, \quad w_t = 0 \text{ if } |S(t)| \leq \alpha, \quad w_t = \frac{1}{|S(t)|^\beta} \text{ if } |S(t)| > \alpha.$$

Here $\alpha, \beta > 0$ are parameters. I use $\alpha = 1$ and $\beta = 2$. The algorithm puts more weights on rare tokens, but a large value of β or a small value of α leads to overfitting. Also, I use the notation pv_0 for an estimated value or

169

Figure 16: LLM for classification, with only 2 parameters

4.2.9 Agentic LLMs

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

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

4.2.10 Data augmentation via dictionaries

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

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

4.2.11 Distillation done smartly

In xLLM, I build two frontend tables `q_dictionary` and `q_embeddings` each time a user generates a new prompt, in order to retrieve the relevant content from the corpus. These tables are similar and linked to the dictionary and embeddings backend tables, but far smaller and serving a single prompt. Then, I remove single tokens that are part of a multi-token when both have the same count in the dictionary. See Figure 11. It makes the output results more concise.

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

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

4.2.12 Reproducibility

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

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

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

4.2.13 Explainable AI

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

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

4.2.14 No training, in-memory LLM

With zero parameter, there is no need for training, though fine-tuning is still critical. Without the big neural network machinery, you or the user (thanks to explainable parameters) can fine-tune with in-memory database (the backend tables structured as nested hashes in my case), and in real time, with predictable outcome resulting from any change. There is no risk of overfitting.

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

4.2.15 No neural network

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

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

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

4.2.16 Show URLs and references

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

4.2.17 Taxonomy-based evaluation

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

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

4.2.18 Augmentation via prompt data

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

4.2.19 Variable-length embeddings, indexing, and database optimization

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

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

Figure 12 shows the command prompt in xLLM. Note the `-p` option for real-time fine-tuning, and also the `-i` option to retrieve full content from a list of text entity IDs. To further optimize search, you can use quantized embeddings or probabilistic nearest neighbor search. The latter is discussed here. The word retrieval is sometimes used instead of search.

4.2.20 Favor backend over frontend engineering

The need for prompt engineering is due in part to faulty backend implementation. Too many tokens (most of them being noise), the choice of poor input sources (for instance, Reddit), too much reliance on embeddings only, and failure to detect and retrieve the embedded structure (knowledge graph, taxonomy) when crawling the corpus. Instead, knowledge graphs are built on top rather than from the ground up. Prompt engineering is the fix to handle the numerous glitches. Some glitches come the Python libraries themselves, see section 4.2.21.

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

4.2.21 Use NLP and Python libraries with caution

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

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

4.2.22 Self-tuning and customization

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

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

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

4.2.23 Local, global parameters, and debugging

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

- Global parameters are identical for all sub-LLMs. They may not perform as well as local parameters, but they are easier to maintain. Also, they can be more difficult to fine-tune. However, you can fine-tune them first on select sub-LLMs, before choosing the parameter set that on average, performs best across multiple high-usage sub-LLMs.

- Local parameters boost performance but require more time to fine-tune, as each sub-LLM has a different set. At the very least, you should consider using ad-hoc stopwords lists for each sub-LLM. These are built by looking at top tokens prior to filtering or distillation, and letting an expert determine which tokens are worth ignoring, for the topic covered by the sub-LLM in question.

You can have a mix of global and local parameters. In xLLM, there is a catch-all parameter set that returns the maximum output you can possibly get from any prompt. It is the same for all sub-LLMs. See option `-f` on the command prompt menu in Figure 12. You can use this parameter set as starting point, and modify values until the output is concise enough and shows the most relevant items at the top. The `-f` option is also used for debugging.

4.2.24 Displaying relevancy scores, and customizing scores

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

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

4.2.25 Intuitive hyperparameters

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

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

4.2.26 Sorted n-grams and token order preservation

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

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

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

4.2.27 Blending standard tokens with tokens in the knowledge graph

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

4.2.28 Boosted weights for knowledge-graph tokens

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

4.2.29 Versatile command prompt

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

With in-memory LLMs such as xLLM, there is no training. Fine-tuning is a lot easier and faster, thanks to explainable AI: see section 4.2.25. In addition to standard prompts, the user can enter command options in the prompt box, for instance `-p key value` to assign `value` to parameter `key`. See Figure 12. The next prompt will be based on the new parameters, without any delay to return results based on the new configuration.

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

4.2.30 Boost long multitokens and rare single tokens

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

Converserly, some single tokens, even consisting of one or two letters depending on the sub-LLM, may be quite rare, indicating that they have high informative value. There is an option in xLLM not to ignore single tokens with fewer than a specified number of occurrences.

Note that in many LLMs on the market, tokens are very short. They consist of parts of a word, not even a full word, and multitokens are absent. In xLLM, very long tokens are favored, while tokens that are less than a word, don't exist. Yet, digits like 1 or 2, single letters, IDs, symbols, codes, and even special characters can be token if they are found *as is* in the corpus.

4.3 LLM glossary

Here, I included the terms that are most relevant to xLLM. In particular, some terms listed in in this glossary are unique to xLLM, others such as decoder or transformer are only found in standard LLMs. Many are found in both. Here, DNN stands for deep neural network. In DNNs, that is, in traditional LLMs, the parameters are the weights connecting neurons. In xLLM, there is usually zero or very few weights; parameters play the role of hyperparameters.

Table 4: LLM glossary

agent	A mechanism to detect user intent in a prompt, to retrieve the most appropriate content. An agent determines what the user is looking for: definition, examples, search results, data, best practices, references, URLs, and so on. Different from an action, which consists of running a separate app for instance to write an email, do some data analysis, or perform some computations. LLMs that can handle various agents are called multi-agent.
ANN	Approximate nearest neighbor search. Similar to the K-NN algorithm used in supervised classification, but faster and applied to retrieving information in vector databases, such as LLM embeddings stored as vectors. I designed a probabilistic version called pANN, especially useful for model evaluation and improvement, with applications to GenAI, synthetic data, and LLMs. See section 8.1 in [4].
backend	The xLLM architecture is split into backend (Figure 4) and frontend (Figure 5). The backend (parameters, tables) deals with the corpus, knowledge graph, and augmentation. It does not see what is in the prompt. The frontend deals with the prompt. It does not see what is in the backend. The LLM is an interface that connects frontend content, to backend tables.
contextual token	A multitoken consisting of multiple single tokens, say (A, B, C), where the tokens A and B, or B and C, are not adjacent to each other in the corpus. Still, A, B and C are found in a same text sub-entity, for instance a paragraph in a larger text entity (web page or JSON entity).
diffusion	Diffusion models use a Markov chain with diffusion steps to slowly add random noise to data and then learn to reverse the diffusion process to construct desired data samples from the noise. The output is usually a dataset or image similar but different from the original ones. Unlike variational auto-encoders, diffusion models have high dimensionality in the latent space (latent variables): the same dimension as the original data. Very popular in computer vision and image generation.
distillation	Data distillation is a technique used to reduce the size of your dataset with minimum loss of information, sometimes even improving predictive algorithms, even via random deletions. It is also used in the context of LLMs, to reduce the size of token lists, and to remove noise or garbage.
embedding	In LLMs, embeddings are typically attached to a keyword, paragraph, or element of text; they consist of tokens. The concept has been extended to computer vision, where images are summarized in small dimensions by a number of numerical features (far smaller than the number of pixels). Likewise, in LLMs, tokens are treated as the features in your dataset, especially when embeddings are represented by fixed-size vectors. The dimension is the number of tokens per embedding. See token entry.
encoder	An auto-encoder is (typically) a neural network to compress and reconstruct unlabeled data. It has two parts: an encoder that compacts the input, and a decoder that reverses the transformation. The original transformer model was an auto-encoder with both encoder and decoder. However, OpenAI (GPT) uses only a decoder. Variational auto-encoders (VAE) are very popular.
exhaustivity	One of the most overlooked evaluation metrics when assessing LLM performance or in benchmarking tests. It depends on the corpus and the knowledge of the expert judging the prompt results. To achieve exhaustivity, consider corpus augmentation and using acronyms to identify all possible name variations for words found in the prompt, to map them to what is in the corpus.
explainable AI	LLMs powered by deep neural networks are Blackboxes governed by hyperparameters. To the contrary, xLLM does not need training and does not use weights estimated via DNNs. Instead, it is fine-tuned using intuitive parameters, a feature known as explainable AI.
fine-tuning	Fine-tuning consists in testing various parameter values (hyperparameters when neural networks are involved) to increase performance (speed, exhaustivity or relevancy in prompt results). Different from training. In xLLM, there is no training, except if used as a classifier, taxonomy builder, or for predictions. Instead, the user can fine-tune frontend parameters in real time. You can also automatically determine the optimum parameters based on user preferences. This is called self-tuning.
frontend	See backend.

Continued on next page

Table 4: LLM glossary (Continued)

GAN	Generative adversarial network. One of the many types of DNN (deep neural network) architecture. It consists of two DNNs: the generator and the discriminator, competing against each other until reaching an equilibrium. Good at generating synthetic images similar to those in your training set (computer vision). Key components include a loss function, a stochastic gradient descent algorithm such as Adam to find a local minimum to the loss function, and hyperparameters to fine-tune the results. Not good at synthesizing tabular data, thus the reason I created NoGAN: see section 2.1 in [4].
GPT	In case you did not know, GPT stands for Generative Pre-trained Transformer. The main application is LLMs. See transformer.
graph database	My LLMs rely on taxonomies attached to the crawled content. Taxonomies consist of categories, subcategories and so on. When each subcategory has exactly one parent category, you use a tree to represent the structure. Otherwise, you use a graph database.
hash database	See also key-value database. The most sophisticated is a nested hash, where the key can be any structure (typically a t-uple or a list) and the value is itself a nested hash. They can be updated very quickly in memory, and the structure is similar to JSON databases. It is extensively used in xLLM, see Figure 1.
hyperparameter	In LLMs powered by deep neural networks (DNN), the hyperparameters are those of the DNN: number of epochs, seed, number of layers, batch size, type a gradient descent, learning rate, parameters attached to the loss function, and so on. In xLLM, hyperparameters – actually called parameters – govern the type of results returned to a user prompt. There are two types: backend and frontend parameters. The former are linked to retrieval in the corpus when crawling. The latter are linked to prompt processing and you can fine-tune them in real time, from the command prompt.
in-memory LLM	When backend tables, weights, and token lists fit in memory, it makes sense to load everything in memory to boost speed. This is the case with xLLM. It can be done with large LLMs, especially those not relying on neural networks and billions of weights.
key-value database	Also known as hash table or dictionary in Python. In xLLM, embeddings have variable size. I store them as short key-value tables rather than long vectors. Keys are tokens, and a value is the association between a token, and the word attached to the parent embedding.
knowledge graph	A structure connecting high-level text elements such as categories and subcategories, or tags. A typical example is a taxonomy. It can be retrieved from the corpus itself when crawling, thanks to breadcrumbs or categorization embedded in the corpus. Or built on top of it, or augmented via external input sources. Categories can have multiple parent categories, and multiple subcategories.
LangChain	Available as a Python library or API, it helps you build applications that read data from internal documents and summarize them. It allows you to build customized GPTs, and blend results to user queries or prompts with local information retrieved from your environment, such as internal documentation or PDFs.
LLaMA	An LLM model that predicts the next word in a word sequence, given previous words. See how I use them to predict the next DNA subsequence in DNA sequencing, here. Typically associated to auto-regressive models or Markov chains.
LLM	Large language model. Modern version of NLP (natural language processing) and NLG (natural language generation). Applications include chatbots, sentiment analysis, text summarization, search, and translation.
multi-agent system	LLM architecture with multiple specialized LLMs. The input data (a corpus or vast repository) is broken down into top categories. Each one has its own LLM, that is, its own embeddings, dictionary, and related tables. Each specialized LLM is sometimes called a simple LLM. Sometimes, the word agent applies to a single or sub-LLM. It represents a mechanism to detect user intent and to serve results matching the intent. For instance, showing definitions, examples, references, tables, best practices and so on.

Continued on next page

Table 4: LLM glossary (Continued)

multimodal	Any architecture that blends multiple data types: text, videos, sound files, and images. The emphasis is on processing user queries in real-time, to return blended text, images, and so on. For instance, turning text into streaming videos.
normalization	Many evaluation metrics take values between 0 and 1 after proper scaling. Likewise, weights attached to tokens in LLM embeddings have a value between -1 and +1. In many algorithms and feature engineering, the input data is usually transformed first (so that each feature has same variance and zero mean), then processed, and finally you apply the inverse transform to the output. These transforms or scaling operations are known as normalization.
parameter	This word is mostly used to represent the weights attached to neuron connections in DNNs. Different from hyperparameters. The latter are knobs to fine-tune models. Also different from the concept of parameter in statistical models despite the same spelling.
RAG	Retrieval-augmentation-generation. In LLMs, retrieving data from summary tables (embeddings) to answer a prompt, using additional sources to augment your training set and the summary tables, and then generating output. Generation focuses on answering a user query (prompt), on summarizing a document, or producing some content such as synthesized videos.
regularization	Turning a standard optimization problem or DNN into constrained optimization, by adding constraints and corresponding Lagrange multipliers to the loss function. Potential goals: to obtain more robust results, or to deal with over-parameterized statistical models and ill-conditioned problems. Example: Lasso regression. Different from normalization.
reinforcement learning	A semi-supervised machine learning technique to refine predictive or classification algorithms by rewarding good decisions and penalizing bad ones. Good decisions improve future predictions; you achieve this goal by adding new data to your training set, with labels that work best in cross-validation testing. In my LLMs, I let the user choose the parameters that best suit his needs. This technique leads to self-tuning and/or customized models: the default parameters come from usage.
synthetic data	Artificial tabular data with statistical properties (correlations, joint empirical distribution) that mimic those of a real dataset. You use it to augment, balance or anonymize data. Few methods can synthesize outside the range observed in the real data (your training set). I describe how to do it in section 10.4 in [5]. A good metric to assess the quality of synthetic data is the full, multivariate Kolmogorov-Smirnov distance, based on the joint empirical distribution (ECDF) computed both on the real and generated observations. It works both with categorical and numerical features. The word synthetic data is also used for generated (artificial) time series, graphs, images, videos and soundtracks in multimodal applications.
text entity	The main text unit in xLLM, for instance a JSON entry such as pictured in Table 1, with raw text, various fields, and contextual information such as category. Sometimes augmented with agent labels. It could also be (say) a Wikipedia web page. Sub-entities are shorter and delimited by pre-specified separators such as period or semi-colon. For two multitokens to be connected (other than via the knowledge graph), they must reside within a same sub-entity.
token	In LLMs or NLP, a token is a single word; embeddings are vectors, with each component being a token. A word such as "San Francisco" is a single token, not two. In my LLMs, I use double tokens, such as "Gaussian distribution" for terms that are frequently found together. I treat them as ordinary (single) tokens. Also, the value attached to a token is its "correlation" (pointwise mutual information) to the word representing its parent embedding, see Table 8.1 in [4]. But in traditional LLMs, the value is simply the normalized token frequency computed on some text repository.
transformer	A transformer model is an algorithm that looks for relationships in sequential data, for instance, words in LLM applications. Sometimes the words are not close to each other, allowing you to detect long-range correlations. It transforms original text into a more compact form and relationships, to facilitate further processing. Embeddings and transformers go together.

Continued on next page

Table 4: LLM glossary (Continued)

vector search	A technique combined with feature encoding to quickly retrieve embeddings in LLM summary tables, most similar to prompt-derived embeddings attached to a user query in GPT-like applications. Similar to multivariate "vlookup" in Excel. A popular metric to measure the proximity between two embeddings is the cosine similarity. To accelerate vector search, especially in real-time, you can cache popular embeddings and/or use approximate search such as ANN.
weight	Also called parameters in deep neural networks (DNN). They are the weights attached to connections between neurons. Thus, LLMs based on DNNs may have billions or even trillions of them, while xLLM has zero, except in the version that performs clustering and predictions (less than 5 weights). Another way to look at it is that weights are implicit in xLLM (not estimated) and governed by a few high-level parameters. See section 4.2.8.

References

[1] Iulia Brezeanu. How to cut RAG costs by 80% using prompt compression. *Blog post*, 2024. TowardsDataScience [Link]. 24

[2] Johnathan Chiu, Andi Gu, and Matt Zhou. Variable length embeddings. *Preprint*, pages 1–12, 2023. arXiv:2305.09967 [Link]. 24

[3] Fabian Gloeckle et al. Better & faster large language models via multi-token prediction. *Preprint*, pages 1–29, 2024. arXiv:2404.19737 [Link]. 24

[4] Vincent Granville. *State of the Art in GenAI & LLMs – Creative Projects, with Solutions*. MLTechniques.com, 2024. [Link]. 4, 44, 45, 46

[5] Vincent Granville. *Statistical Optimization for AI and Machine Learning*. MLTechniques.com, 2024. [Link]. 46

[6] Albert Jiang et al. Mixtral of experts. *Preprint*, pages 1–13, 2024. arXiv:2401.04088 [Link]. 24

[7] Andrei Lopatenko. Evaluating LLMs and LLM systems: Pragmatic approach. *Blog post*, 2024. [Link]. 24

[8] Hussein Mozannar et al. The RealHumanEval: Evaluating Large Language Models' abilities to support programmers. *Preprint*, pages 1–34, 2024. arXiv:2404.02806 [Link]. 24

[9] Sergey Shchegrikovich. How do you create your own LLM and win The Open LLM Leaderboard with one Yaml file? *Blog post*, 2024. [Link]. 24

Index