Custom Enterprise LLM/RAG with Real-Time Fine-Tuning

Vincent Granville, Ph.D.
vincentg@MLTechniques.com
www.GenAItechLab.com
Version 1.0, July 2024

1 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.

- 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. 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 are 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.

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 mt 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 200–220 in section 5) are on GitHub, here.

Also to be included in the next release: corpus augmentation with synonyms and abbreviations dictionaries, as well as contextual multitokens. The latter is implemented in the previous version and discussed in section 8.3 in my book [1]. 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 173–176 in the code. Then, words such as 'San Francisco' must be treated as single tokens.

Finally, prompts are not broken down into sub-prompts, and the concept of action is not implemented yet. 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. This topic will be discussed in my next article.

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 236–267 in the code in section 5. 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 209 in the code. The `hash_pairs` table consists of multitokens pairs, each with up to 3 terms: see parameter `maxTerms` in line 211. The maximum gap allowed between two contextual multitokens is 3 terms: see parameter `maxDist` in line 210. These limitations are set to prevent the number of pairs and tokens from exploding. In the end, there are 12,581 multitokens, stored in the `dictionary` table, after removing stopwords. The total number of multitoken pairs is 222,758, 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 200–206. Finally, when counting multitoken occurrences, appearances in categories, titles and tags get an extra boost, compared to regular text: see lines 212–219.

I did not include `embeddings` and `sorted_ngrams` in the `backendTables` structure in lines 183–198, 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.

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 492–503 in the code. 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 505-514, his/her own queries, or simple instructions to update or view the frontend parameters, using one of the options in lines 519–527. 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.

3 Case study

I tested the algorithm on one part of a corporate corpus (fortune 100 company), dealing with documentation on internal AI systems and policies. In this article, I discuss 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 500 text entities, and can be found here. Table 1 features a sample text entity.

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	

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 to decide if the results match the type of action the employee is looking for.

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'.
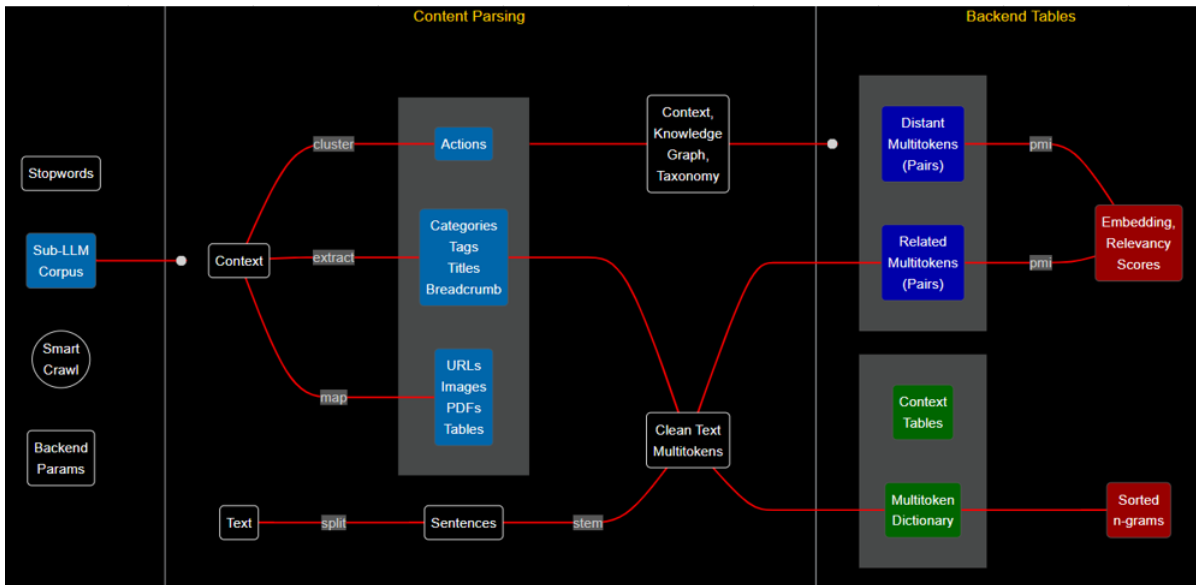

Figure 1: From crawl to backend tables (high resolution here)

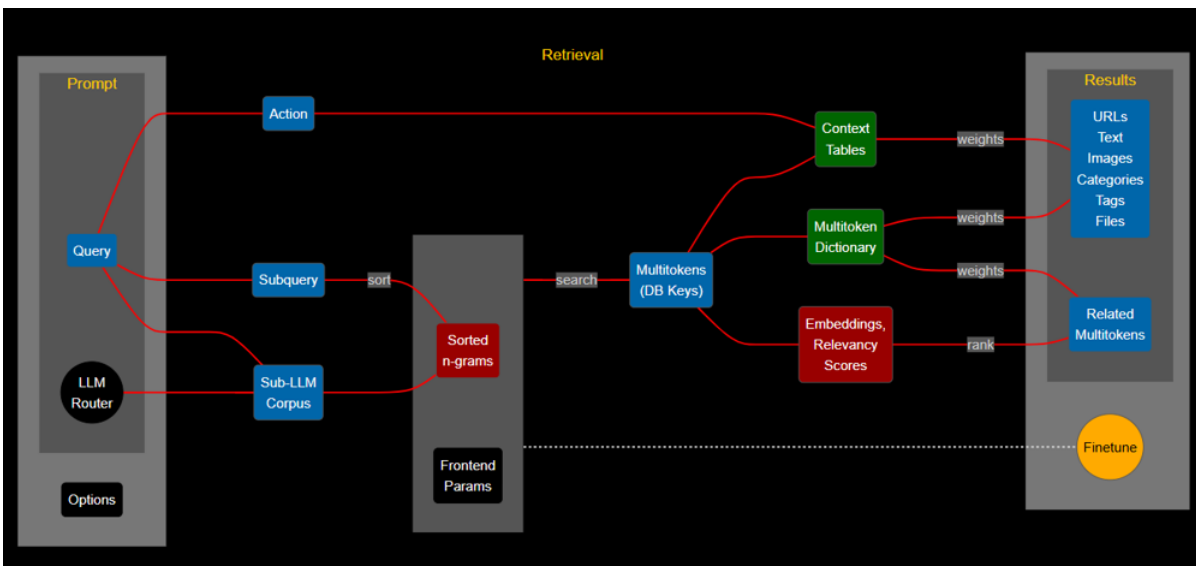

Figure 2: From prompt to query results, via backend tables (high resolution here)

Figures 1 and 2 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. They 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 308–311 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		

4 Conclusions

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.

5 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.

```python
#--- [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)
    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)
```

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

```python
    hash_pairs = backendTables['hash_pairs']
    ctokens = backendTables['ctokens']
    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 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
  'hash_pairs', # multitoken associations
  'hash_context1', # categories
  'hash_context2', # tags
  'hash_context3', # titles
  'hash_context4', # descriptions
  'hash_context5', # meta
  'ctokens',    # not adjacent pairs in hash_pairs
  'hash_ID',    # ID, such as document ID or url ID
  'full_content' # full content
```

```python
194  )
195
196  backendTables = {}
197  for name in tableNames:
198      backendTables[name] = {}
199
200  stopwords = ('', '-', 'in', 'the', 'and', 'to', 'of', 'a', 'this', 'for', 'is', 'with',
         'from',
201              'as', 'on', 'an', 'that', 'it', 'are', 'within', 'will', 'by', 'or', 'its',
                   'can',
202              'your', 'be','about', 'used', 'our', 'their', 'you', 'into', 'using', 'these',
203              'which', 'we', 'how', 'see', 'below', 'all', 'use', 'across', 'provide',
                   'provides',
204              'aims', 'one', '&', 'ensuring', 'crucial', 'at', 'various', 'through', 'find',
                   'ensure',
205              'more', 'another', 'but', 'should', 'considered', 'provided', 'must',
                   'whether',
206              'located', 'where', 'begins', 'any')
207
208  backendParams = {
209      'max_multitoken': 4, # max. consecutive terms per multi-token for inclusion in
           dictionary
210      'maxDist' : 3,  # max. position delta between 2 multitokens to link them in hash_pairs
211      'maxTerms': 3,  # maxTerms must be <= max_multitoken
212      'extraWeights' : # deafault weight is 1
213        {
214          'description': 0.0,
215          'category': 0.3,
216          'tag_list': 0.4,
217          'title':   0.2,
218          'meta':    0.1
219        }
220  }
221
222
223  local = True
224  if local:
225      # get repository from local file
226      IN = open("repository.txt","r")
227      data = IN.read()
228      IN.close()
229  else:
230      # get repository from GitHub url
231      import requests
232      url = "https://mltblog.com/3y8MXq5"
233      response = requests.get(url)
234      data = response.text
235
236  entities = data.split("\n")
237
238  for entity_raw in entities:
239
240      entity = entity_raw.split("~~")
241
242      if len(entity) > 1:
243
244          entity_ID = int(entity[0])
245          entity = entity[1].split("{")
246          hash_crawl = {}
247          hash_crawl['ID'] = entity_ID
248          hash_crawl['full_content'] = entity_raw
249
250          key_value_pairs = get_key_value_pairs(entity)
251
252          for pair in key_value_pairs:
253              if ": " in pair:
```

```python
254                  key, value = pair.split(": ", 1)
255                  key = key.replace("'","")
256                  if key == 'category_text':
257                      hash_crawl['category'] = value
258                  elif key == 'tags_list_text':
259                      hash_crawl['tag_list'] = clean_list(value)
260                  elif key == 'title_text':
261                      hash_crawl['title'] = value
262                  elif key == 'description_text':
263                      hash_crawl['description'] = value
264                  elif key == 'tower_option_tower':
265                      hash_crawl['meta'] = value
266
267          backendTables = update_dict(backendTables, hash_crawl, backendParams)
268
269
270  print()
271  print(len(backendTables['dictionary']))
272  print(len(backendTables['hash_pairs']))
273  print(len(backendTables['ctokens']))
274
275
276  # [2.1] Create embeddings
277
278  embeddings = {} # multitoken embeddings based on hash_pairs
279
280  hash_pairs = backendTables['hash_pairs']
281  dictionary = backendTables['dictionary']
282
283  for key in hash_pairs:
284      wordA = key[0]
285      wordB = key[1]
286      nA = dictionary[wordA]
287      nB = dictionary[wordB]
288      nAB = hash_pairs[key]
289      pmi = nAB/(nA*nB)**0.5 # try: nAB/(nA + nB - nAB)
290      # if nA + nB <= nAB:
291      #   print(key, nA, nB, nAB)
292      update_nestedHash(embeddings, wordA, wordB, pmi)
293      update_nestedHash(embeddings, wordB, wordA, pmi)
294
295
296  # [2.2] Create sorted n-grams
297
298  sorted_ngrams = {} # to match ngram prompts with embeddings entries
299
300  for word in dictionary:
301      tokens = word.split('~')
302      tokens.sort()
303      sorted_ngram = tokens[0]
304      for token in tokens[1:len(tokens)]:
305          sorted_ngram += "~" + token
306      update_nestedHash(sorted_ngrams, sorted_ngram, word)
307
308  # print top multitokens
309  # for key in dictionary:
310  #   if dictionary[key] > 20:
311  #      print(key, dictionary[key])
312
313
314  #--- [3] Frontend: functions
315
316  # [3.1] custom pmi
317
318  def custom_pmi(word, token, backendTables):
319
```

```python
320     dictionary = backendTables['dictionary']
321     hash_pairs = backendTables['hash_pairs']
322
323     nAB = 0
324     pmi = 0.00
325     keyAB = (word, token)
326     if word > token:
327         keyAB = (token, word)
328     if keyAB in hash_pairs:
329         nAB = hash_pairs[keyAB]
330         nA = dictionary[word]
331         nB = dictionary[token]
332         pmi = nAB/(nA*nB)**0.5
333     return(pmi)

334
335 # [3.2] update frontend params
336
337 def update_params(option, frontendParams):
338
339     arr = []
340     for param in frontendParams:
341         arr.append(param)
342     print()
343
344     if option == '-l':
345         print("Multitoken ignore list:\n", frontendParams['ignoreList'])
346     elif option == '-v':
347         print("%3s %s %s" %('Key', 'Description'.ljust(25), 'Value'))
348         for key in range(len(arr)):
349             param = arr[key]
350             value = frontendParams[param]
351             print("%3d %s %s" %(key, param.ljust(25), value))
352     elif option == '-f':
353         for param in frontendParams:
354             if param == 'ignoreList':
355                 frontendParams[param] = ()
356             else:
357                 frontendParams[param] = 0
358     elif '-p' in option:
359         option = option.split(' ')
360         if len(option) == 3:
361             paramID = int(option[1])
362             if paramID < len(arr):
363                 param = arr[paramID]
364                 value = float(option[2])
365                 frontendParams[param] = value
366             else:
367                 print("Error 101: key outside range")
368         else:
369             print("Error 102: wrong number of arguments")
370     elif '-a' in option:
371         option = option.split(' ')
372         if len(option) == 2:
373             ignore = frontendParams['ignoreList']
374             ignore =(*ignore, option[1])
375             frontendParams['ignoreList'] = ignore
376         else:
377             print("Error 103: wrong number of arguments")
378     elif '-r' in option:
379         option = option.split(' ')
380         if len(option) == 2:
381             ignore2 = ()
382             ignore = frontendParams['ignoreList']
383             for item in ignore:
384                 if item != option[1]:
385                     ignore2 = (*ignore2, item)
```

```
386             frontendParams['ignoreList'] = ignore2
387         else:
388             print("Error 104: wrong number of arguments")
389     return(frontendParams)
390
391 # [3.3] retrieve info and print results
392
393 def print_results(q_dictionary, q_embeddings, backendTables, frontendParams):
394
395     dictionary = backendTables['dictionary']
396     hash_pairs = backendTables['hash_pairs']
397     ctokens = backendTables['ctokens']
398
399     if frontendParams['bypassIgnoreList'] == 1:
400         # ignore multitokens specified in 'ignoreList'
401         ignore = frontendParams['ignoreList']
402     else:
403         # bypass 'ignore' list
404         ignore = ()
405
406     local_hash = {} # used to not show same token 2x (linked to 2 different words)
407     q_embeddings = dict(sorted(q_embeddings.items(),key=lambda item:
408             item[1],reverse=True))
408     print()
409     print("%3s %s %1s %s %s"
410             %('N','pmi'.ljust(4),'F','token [from embeddings]'.ljust(35),
411              'word [from prompt]'.ljust(35)))
412     print()
413
414     for key in q_embeddings:
415         word = key[0]
416         token = key[1]
417         pmi = q_embeddings[key]
418         ntk1 = len(word.split('~'))
419         ntk2 = len(token.split('~'))
420         flag = " "
421         nAB = 0
422         keyAB = (word, token)
423         if word > token:
424             keyAB = (token, word)
425         if keyAB in hash_pairs:
426             nAB = hash_pairs[keyAB]
427         if keyAB in ctokens:
428             flag = '*'
429         if ( ntk1 >= frontendParams['embeddingKeyMinSize'] and
430              ntk2 >= frontendParams['embeddingValuesMinSize'] and
431              pmi >= frontendParams['min_pmi'] and
432              nAB >= frontendParams['nABmin'] and
433              token not in local_hash and word not in ignore
434             ):
435             print("%3d %4.2f %1s %s %s"
436                     %(nAB,pmi,flag,token.ljust(35),word.ljust(35)))
437             local_hash[token] = 1 # token marked as displayed, won't be showed again
438
439     print()
440     print("N = occurrences of (token, word) in corpus. F = * if contextual pair.")
441     print("If no result, try option '-p f'.")
442     print()
443
444     sectionLabels = {
445         # map section label (in output) to corresponding backend table name
446         'dict' :'dictionary',
447         'pairs':'hash_pairs',
448         'category':'hash_context1',
449         'tags' :'hash_context2',
450         'titles':'hash_context3',
```

```python
451         'descr.':'hash_context4',
452         'meta' :'hash_context5',
453         'ID'   :'hash_ID',
454         'whole' :'full_content'
455     }
456     local_hash = {}
457
458     for label in ('category','tags','titles','descr.','ID','whole'):
459         tableName = sectionLabels[label]
460         table = backendTables[tableName]
461         local_hash = {}
462         print(">>> RESULTS - SECTION: %s\n" % (label))
463         for word in q_dictionary:
464             ntk3 = len(word.split('~'))
465             if word not in ignore and ntk3 >= frontendParams['ContextMultitokenMinSize']:
466                 content = table[word] # content is a hash
467                 count = int(dictionary[word])
468                 for item in content:
469                     update_nestedHash(local_hash, item, word, count)
470         for item in local_hash:
471             hash2 = local_hash[item]
472             if len(hash2) >= frontendParams['minOutputListSize']:
473                 print(" %s: %s [%d entries]" % (label, item, len(hash2)))
474                 for key in hash2:
475                     print(" Linked to: %s (%s)" %(key, hash2[key]))
476                 print()
477         print()
478
479     print()
480     print("Results based on words found in prompt, matched back to backend tables.")
481     print("Numbers in parentheses are occurrences of word in corpus.")
482
483     return()
484
485
486 #--- [4] Frontend: main (process prompt)
487
488 print("\n")
489 input_ = " "
490 get_bin = lambda x, n: format(x, 'b').zfill(n)
491
492 ignore = ('data',)
493 frontendParams = {
494                 'embeddingKeyMinSize': 2,
495                 'embeddingValuesMinSize': 2,
496                 'min_pmi': 0.00,
497                 'nABmin': 1,
498                 'Customized_pmi': 1,
499                 'ContextMultitokenMinSize': 2,
500                 'minOutputListSize': 1,
501                 'bypassIgnoreList': 0,
502                 'ignoreList': ignore
503              }
504
505 sample_queries = (
506                 'parameterized datasets map tables sql server',
507                 'data load templates importing data database data warehouse',
508                 'pipeline extract data eventhub files',
509                 'blob storage single parquet file adls gen2',
510                 'eventhub files blob storage single parquet',
511                 'parquet blob eventhub more files less storage single table',
512                 'MLTxQuest Data Assets Detailed Information page'
513                 'stellar', 'table',
514              )
515
516 while len(input_) > 0:
```

13

```
517
518     print()
519     options = ('-p', '-f', '-v', '-a', '-r', '-l')
520     print("---")
521     print("Query options:")
522     print(" -p key value : set frontendParams['key'] = value")
523     print(" -f        : use catch-all parameter set")
524     print(" -v        : view parameter set")
525     print(" -a multitoken : add multitoken to 'ignore' list")
526     print(" -r multitoken : remove multitoken from 'ignore' list")
527     print(" -l        : view 'ignore' list")
528     print()
529
530     input_ = input("Query, query option, or integer in [0, %d] for sample query: "
531                 %(len(sample_queries)-1))
532
533     flag = True # False --> query to change params, True --> real query
534     for option in options:
535         if option in input_:
536             update_params(input_, frontendParams)
537             input_ = " "
538             flag = False
539
540     if input_.isdigit():
541         if int(input_) < len(sample_queries):
542             query = sample_queries[int(input_)]
543             print("query:",query)
544         else:
545             print("Value must be <", len(sample_queries))
546             query = ""
547     else:
548         query = input_
549
550     query = query.split(' ')
551     query.sort()
552     q_embeddings = {}
553     q_dictionary = {}
554
555     for k in range(1, 2**len(query)):
556
557         binary = get_bin(k, len(query))
558         sorted_word = ""
559         for k in range(0, len(binary)):
560             if binary[k] == '1':
561                 if sorted_word == "":
562                     sorted_word = query[k]
563                 else:
564                     sorted_word += "~" + query[k]
565
566         if sorted_word in sorted_ngrams:
567             ngrams = sorted_ngrams[sorted_word]
568             for word in ngrams:
569                 if word in dictionary:
570                     q_dictionary[word] = dictionary[word]
571                     if word in embeddings:
572                         embedding = embeddings[word]
573                         for token in embedding:
574                             if frontendParams['Customized_pmi'] == 0:
575                                 pmi = embedding[token]
576                             else:
577                                 # customized pmi
578                                 pmi = custom_pmi(word, token, backendTables)
579                             q_embeddings[(word, token)] = pmi
580
581     if len(query) == 1:
582         # single-token query
```

```
583        frontendParams['embeddingKeyMinSize'] = 1
584        frontendParams['ContextMultitokenMinSize'] = 1
585
586    if len(input_) > 0 and flag:
587        print_results(q_dictionary, q_embeddings, backendTables, frontendParams)
588
589
590 #--- [5] Save backend tables
591
592 save = False
593 if save:
594    for tableName in backendTables:
595        table = backendTables[tableName]
596        OUT = open('backend_' + tableName + '.txt', "w")
597        OUT.write(str(table))
598        OUT.close()
599
600    OUT = open('backend_embeddings.txt', "w")
601    OUT.write(str(embeddings))
602    OUT.close()
603
604    OUT = open('backend_sorted_ngrams.txt', "w")
605    OUT.write(str(sorted_ngrams))
606    OUT.close()
```

References

[1] Vincent Granville. *State of the Art in GenAI & LLMs – Creative Projects, with Solutions.* MLTechniques.com, 2024. [Link]. 2