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 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 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’.

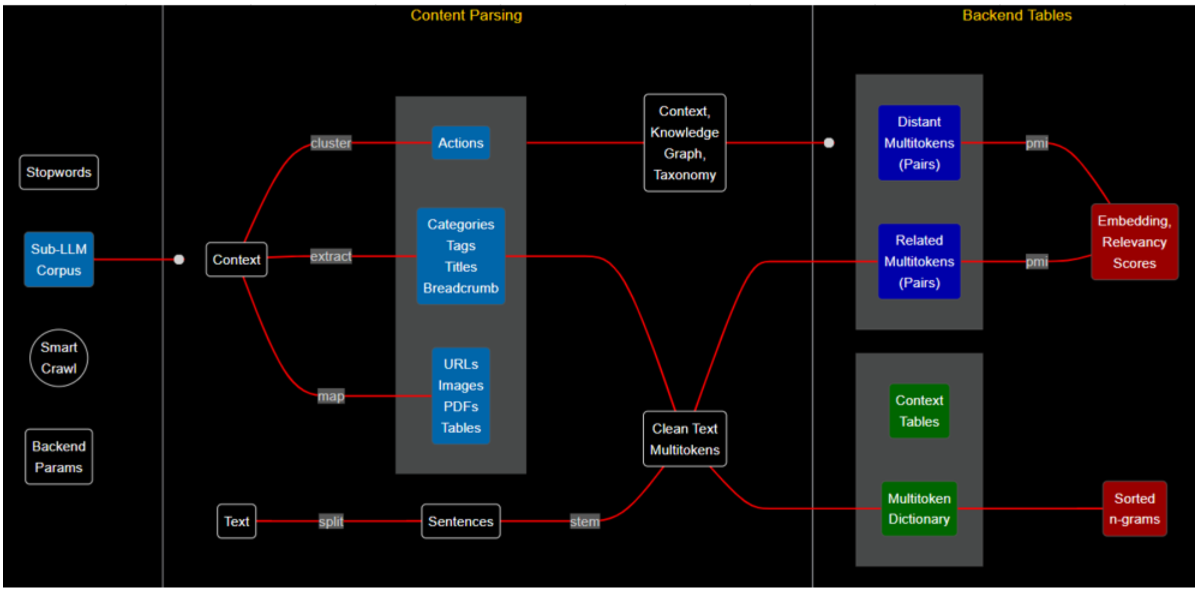

Figure 1: From crawl to backend tables (high resolution here)

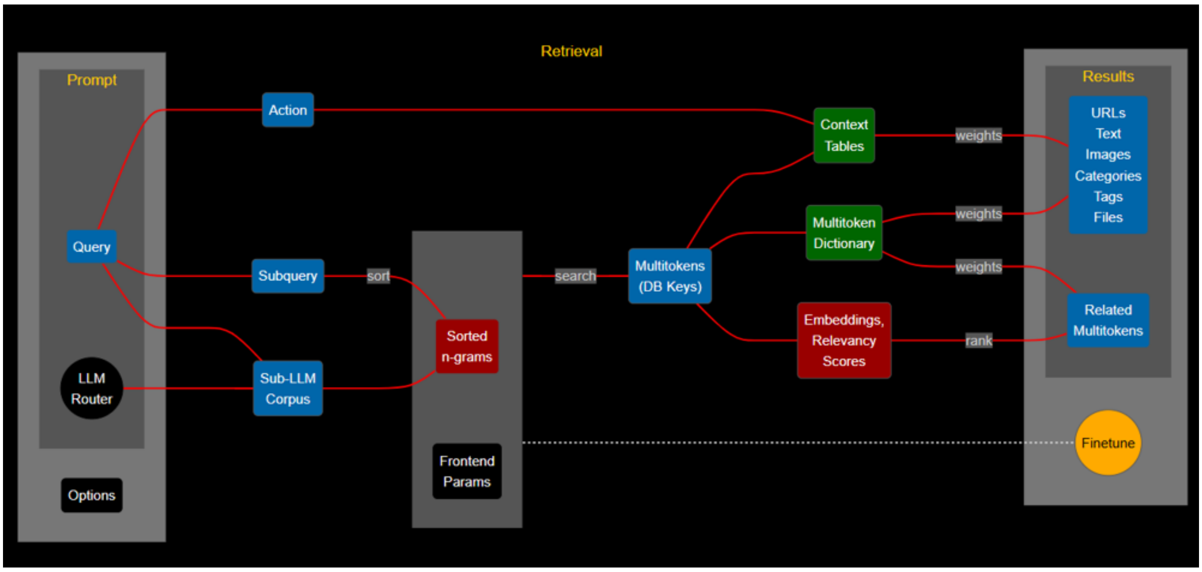

Figure 2: From prompt to query results, via backend tables (high resolution here)

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)
```

```
    update_nestedHash(backendTables['hash_context2'], word, tag_list)
    update_nestedHash(backendTables['hash_context3'], word, title)
    update_nestedHash(backendTables['hash_context4'], word, description)
    update_nestedHash(backendTables['hash_context5'], word, meta)
    update_nestedHash(backendTables['hash_ID'], word, ID)
    update_nestedHash(backendTables['full_content'], word, 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(",", " ")
    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']
   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
```

```
)

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')

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

entities = data.split("\n")

for entity_raw in entities:

   entity = entity_raw.split("~~")

   if len(entity) > 1:

      entity_ID = int(entity[0])
      entity = entity[1].split("{")
      hash_crawl = {}
      hash_crawl['ID'] = entity_ID
      hash_crawl['full_content'] = entity_raw

      key_value_pairs = get_key_value_pairs(entity)

      for pair in key_value_pairs:
         if ": " in pair:
```

```python
            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
            elif key == 'tower_option_tower':
               hash_crawl['meta'] = value

      backendTables = update_dict(backendTables, hash_crawl, backendParams)


print()
print(len(backendTables['dictionary']))
print(len(backendTables['hash_pairs']))
print(len(backendTables['ctokens']))


# [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
# 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 update_params(option, frontendParams):

    arr = []
    for param in frontendParams:
        arr.append(param)
    print()

    if option == '-l':
        print("Multitoken ignore list:\n", frontendParams['ignoreList'])
    elif option == '-v':
        print("%3s %s %s" %('Key', 'Description'.ljust(25), 'Value'))
        for key in range(len(arr)):
            param = arr[key]
            value = frontendParams[param]
            print("%3d %s %s" %(key, param.ljust(25), value))
    elif option == '-f':
        for param in frontendParams:
            if param == 'ignoreList':
                frontendParams[param] = ()
            else:
                frontendParams[param] = 0
    elif '-p' in option:
        option = option.split(' ')
        if len(option) == 3:
            paramID = int(option[1])
            if paramID < len(arr):
                param = arr[paramID]
                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")
   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']

   if frontendParams['bypassIgnoreList'] == 1:
      # ignore multitokens specified in 'ignoreList'
      ignore = frontendParams['ignoreList']
   else:
      # bypass 'ignore' list
      ignore = ()

   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 showed 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 (in output) 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',
      'whole' :'full_content'
   }
   local_hash = {}

   for label in ('category','tags','titles','descr.','ID','whole'):
      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]))
            print()
      print()

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

   return()


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

print("\n")
input_ = " "
get_bin = lambda x, n: format(x, 'b').zfill(n)

ignore = ('data',)
frontendParams = {
               'embeddingKeyMinSize': 2,
               'embeddingValuesMinSize': 2,
               'min_pmi': 0.00,
               'nABmin': 1,
               'Customized_pmi': 1,
               'ContextMultitokenMinSize': 2,
               'minOutputListSize': 1,
               'bypassIgnoreList': 0,
               'ignoreList': ignore
              }

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'
               'stellar', 'table',
             )

while len(input_) > 0:
```

```
print()
options = ('-p', '-f', '-v', '-a', '-r', '-l')
print("---")
print("Query options:")
print(" -p key value : set frontendParams['key'] = value")
print(" -f        : use catch-all 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()

input_ = input("Query, query option, or integer in [0, %d] for sample query: "
            %(len(sample_queries)-1))

flag = True # False --> query to change params, True --> real query
for option in options:
   if option in input_:
      update_params(input_, frontendParams)
      input_ = " "
      flag = False

if input_.isdigit():
   if int(input_) < len(sample_queries):
     query = sample_queries[int(input_)]
     print("query:",query)
   else:
     print("Value must be <", len(sample_queries))
     query = ""
else:
   query = input_

query = query.split(' ')
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 frontendParams['Customized_pmi'] == 0:
                     pmi = embedding[token]
                  else:
                     # customized pmi
                     pmi = custom_pmi(word, token, backendTables)
                  q_embeddings[(word, token)] = pmi

if len(query) == 1:
   # single-token query
```

```python
        frontendParams['embeddingKeyMinSize'] = 1
        frontendParams['ContextMultitokenMinSize'] = 1

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


#--- [5] Save backend tables

save = False
if save:
    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()
```

References

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