Blueprint: Next-Gen Enterprise RAG & LLM
Nvidia PDFs Use Case

Vincent Granville, Ph.D.
vincentg@MLTechniques.com
www.MLTechniques.com
Version 1.0, January 2025

Contents

1 Introduction

In this paper, I explain how to build an enterprise LLM and RAG system from scratch, without training, GPU, neural network or transformer, while delivering more accurate and exhaustive results at a fraction of the cost. It includes the full code with documentation, a deep dive on each component, the advanced UI with real-time fine-tuning, relevancy scores and structured output, the efficient in-memory database architecture, as well as front-end and back-end parameters. Each with detailed description and latest updates regularly added.

To illustrate the methodology, I use a Nvidia repository consisting of financial documents stored as PDFs. Also featured here: deep PDF retrieval including tables, images, bullet lists and font types, multi-index hierarchical and contextual chunking, as well as categories or agents assignment at the backend (chunking) level. Reliance on external Python libraries is minimum and complemented by enhancements as needed, to avoid drawbacks that can cause hallucinations. You will find here the secret sauce and innovative thinking behind LLM 2.0 and Enterprise xLLM in particular, based on many years of experience and technological vision.

The focus is on the core architecture. Specialized applications such as LLM for cataloging, text clustering and predictive analytics, sub-LLMs and LLM router, DNA sequences and tabular data synthetization, API calls for problem solving and code generation, are covered in books [1, 2]. In particular, Part 1 and Chapter 10 in [1] are most relevant to the material presented here. The most recent version of this article is available here.

2 From Corpus to xLLM Format

One of the key features of this pre-processing step is to extract contextual elements when turning the corpus data into the input format required by xLLM, our proprietary model. In the case of the Wolfram corpus consisting of 15k webpages and 5k categories, the goal is to recover and use the embedded taxonomy and other contextual elements, rather than building a knowledge graph on top of the website, from scratch. For the Inbev corporate corpus, contextual elements present in the corpus include breadcrumbs, categories, tags, titles, and so on. For both examples, see details in my books [1, 2].

In the example discussed here (Nvidia PDF repository) the context includes document title, page title and number, text in big or bold font, sub-lists in bullet lists, and more. In addition, to each chunk, I assign extra contextual elements such as tag, category, or agent. For instance, the 'table' or 'image' agent indicates that the chunk contains tables or images. I describe chunking in section 2.1 and agents in section 3.4. An LLM dealing not only with text but also images and other media, is called multimodal.

2.1 Hierarchical chunking with multi-index

Chunks are text entities delimited by separators, such as sentences, paragraphs, or pages. In this version, I use two levels: paragraphs and pages. For details, see Chapter 10 in [1], also discussing table, font size, bullet lists and sublists detection. A multi-index is attached to each chunk, for easy retrieval and browsing. See the two sample chunks in this section. The index, referred to as `ID` in the Python code, is A6X2 for the first chunk and B3051X2 for the second one. The letter A indicates a parent chunk, while B stands for a sub-chunk. The strings X2, X1 indicate that the chunks are respectively in the 2nd and 1st PDF document. Images and tables are stored separately, with an index that includes the document and page number, respectively `doc_ID` and `pn` in the code.

```
A6X2~~{'title_text': '', 'description_text': '|"NVIDIA Builds Out Its|Omniverse
    Ecosystem|to Support the|Automotive Metaverse"|. |-- SiliconANGLE|. |NVIDIA
    Omniverse is an ideal tool for an|industrial world seeking to
    digitalize.|Omniverse can simulate the best|possible layouts before the first
    brick|is placed. The $3 trillion automotive|industry is modernizing all its
    processes|to take advantage of computing and|AI. BMW Group uses Omniverse
    to|build a whole factory digital twin before|constructing it physically.
    Mercedes-|Benz is using NVIDIA DRIVE IX on|Omniverse to design and simulate
    its|integrated cabins and electronics.|', 'doc_ID': 2, 'pn': 10}

B3051X1~~{'title_text': '', 'description_text': '|$|1,990|$|847|$|10,896|', 'type':
    'Data', 'doc_ID': 1, 'pn': 170, 'block_ID': 96, 'item_ID': -1, 'sub_ID': -1,
    'fs': 8.7, 'fc': 0, 'ft': 'DIN_Next_LT_Pro_Light'}
```

Besides images, the xLLM input format is plain ASCII text with a structure similar to JSON, with special or accented characters being encoded. Each row corresponds to a chunk or sub-chunk, and is stored as a hash. The hash key indicates the type of information, for instance:

- page title: `title_text`,
- full text: `description_text`,
- page number `pn`, document ID: `doc_ID`,
- type of content: `type` (the value `Data` indicates data in chunk B3051X1),
- list of tags, agents, categories, parent categories, images, tables
- font size: `fs`, font color: `fc`

and so on. Content types include data, bullet list, sub-list, note, and more. Another useful hash key is URLs, in particular, the URL pointing to the specific PDF page where the chunk is located. For instance, if doc.pdf is on example.com, then `example.com/doc.pdf#page=5` points to page 5 in the PDF. Note that the the number of hash keys in a chunk is not fixed. The full xLLM input file (the chunks stored as key-value pairs) is available here.

The backend table `Index_to_IDs` generated by the main program in section 5.1, connects each parent chunk to its sub-chunks. It has one row per parent chunk. In the example below featuring one row, the parent chunk is indexed as A18X1, while such-chunks have an index starting with letter B. The vector at the beginning indicates the document ID and page number. The numbers in black represent the size of each chunk. You can download the full table is available here. Usually, the parent chunk is larger than its sub-chunks. The exception is when the parent chunk is very small: sub-chunks have more overhead that can make them bigger than the parent chunk, but this is usually an indication of poor chunking.

```
(1, 22) {'B192X1': 205, 'B193X1': 1030, 'B194X1': 255, 'B195X1': 230, 'B197X1': 184,
    'B198X1': 201, 'B199X1': 185, 'B200X1': 201, 'B201X1': 185, 'B202X1': 202,
    'B203X1': 186, 'B204X1': 244, 'B205X1': 365, 'B206X1': 429, 'B207X1': 468,
    'B208X1': 575, 'B209X1': 174, 'A18X1': 2396}
```

2.2 Duplicate and overlapping chunks

Chunks in different PDFs may be identical. For instance, the tables of contents are identical in the 2022 and 2023 financial reports (two separate PDFs), as it based on the same template. Whether or not keeping them both is a business decision. Also, within a same PDF, some parent chunks may contain only one sub-chunk. In this case, both are identical due to poor chunking.

In the current version, sub-chunks do not overlap, and parent chunks do not overlap either. However, it is a good practice to allow for overlapping chunks. This is easy to achieve since chunks are sequentially generated. For instance, it makes sense to group sub-chunks into bigger chunks, with the last sub-chunk of a big chunk being the first sub-chunk of the next big chunk. The reason for doing this is because a token in a sub-chunk may be related to another token in the next sub-chunk. Parent chunks partially address this issue, but there is room for improvement.

Another question is why storing both sub-chunks and parent chunks, when sub-chunks alone are enough as they can be used to reconstruct the parents chunks. The reason is due to the relatively small size of enterprise corpuses. If there is enough space and memory to store both, it is more efficient to keep them both despite doubling the memory requirements. Note that the space needed in backend tables to store all the elements of a chunk grows faster than linearly with the size of the chunk: in particular, the number of token pairs consisting of tokens found in a same chunk (used to build embeddings) grows quadratically with the size of the chunk. To avoid this problem, tokens that are not close enough to each other cannot form a token pair; the maximum distance allowed is determined by the backend parameter `maxDist`, see section 3.2.

2.3 PDF parser and data

The Python code is similar to the older version documented in Chapter 10 in [1]. The main differences are:

- Dealing with multiple PDFs. See lines 292–302.
- Detection of duplicate chunks (when a parent chunk only has one sub-chunk), see lines 317–318.
- Multi-index with specific codes to distinguish chunks from sub-chunks, see lines 345 and 380.

The code is on Google Drive, here. The full xLLM output file is available here. The three PDF documents used as input in the code can be found respectively here, here, and here. In the Python code, lines 85–91 deal with special characters. Pages of each PDF are saved as images with the proper index in lines 396–422. Note the image detection mechanism in lines 265–274, table detection using the PymuPDF library in line 169–179, as well as home-made bullet list, sub-list and table detection for tables missed by PymuPDF. For detailed description, see Chapter 10 in [1]. PDF_Chunking_Nvidia.py

```
1  # Input PDF for this script: https://drive.google.com/file/d/1Daa9oZJm4-b6NqUsVGxK2euemcFnf8jH/
2  
3  # pymupdf.readthedocs.io/en/latest/page.html#Page.find_tables
4  # stackoverflow.com/questions/56155676/how-do-i-extract-a-table-from-a-pdf-file-using-pymupdf
5  
6  import fitz # PyMuPDF
7  
8  
9  def update_item_ID(k, entity_idx, type, table_ID):
10 
11    idx = entity_idx[k]
12    if type == 'Data':
13       # table: data row
14       flag = 'TD' #
15    elif type == 'Note':
16       # table: labels
17       flag = 'TL'
18    idx_list = list(idx)
19    idx_list[1] = flag + str(table_ID)
20    entity_idx[k] = tuple(idx_list)
21 
```

```
    return(entity_idx)


def detect_table(xLLM_entity):

    # detect and flag simple pseudo-tables

    entity_txt = xLLM_entity[0]
    entity_type = xLLM_entity[1]
    entity_idx = xLLM_entity[2]
    table_ID = -1
    table_flag = False

    for k in range(1, len(entity_type)):

        type = entity_type[k]
        text = entity_txt[k]
        old_text = entity_txt[k-1]
        old_type = entity_type[k-1]

        if ( (
                (type == 'Data' and old_type == 'Note') or
                (type == 'Note' and old_type == 'Data') or
                (type == 'Data' and old_type == 'Data')
              )
              and old_text.count('|') == text.count('|')
              and text.count('|') > 2
            ):
            print("detected table", table_ID + 1)
            if not table_flag:
                table_ID += 1
                table_flag = True
            idx = entity_idx[k]
            old_idx = entity_idx[k-1]

            # update item_ID (idx[1] and old_idx[1]) in current and previous row
            # item_ID starts with letter D if data, or N if labels

            update_item_ID(k, entity_idx, type, table_ID)
            update_item_ID(k-1, entity_idx, old_type, table_ID)

        else:
            #table_ID = -1
            table_flag = False

    return(xLLM_entity)


def cprint_page(xLLM_entity, hash_chunks, OUT):

    entity_txt = xLLM_entity[0]
    entity_type = xLLM_entity[1]
    entity_idx = xLLM_entity[2]

    for k in range(len(entity_type)):

        type = entity_type[k]
        text = entity_txt[k]
        text = text.strip()
        text = text.replace("  ", " ")
        text = text.replace(" |", "|")
        text = text.replace("| ", "|")
        text = text.replace("||","|")
        text = text.encode('unicode-escape').decode('ascii')
        text = text.replace('\\u2022', '•')
        text = text.replace('\\u2013', '--')
        text = text.replace('\\u2014', '--')
        text = text.replace('\\u2019', "'")
        text = text.replace('\\u201c', '"')
        text = text.replace('\\u201d', '"')

        idx = entity_idx[k]
        doc_ID = idx[3]
        block_ID = idx[0]
        item_ID = idx[1]
        sub_ID = idx[2]
```

```
      pn = idx[4] # page number
      fs = idx[5] # font size
      fc = idx[6] # font color
      ft = idx[7] # font typeface
      #- print(k, type, idx, text)
      OUT.write(f"{type:<8}{block_ID:>3}{item_ID:>5}{sub_ID:>3}{pn:>3}"
              f"{fs:>5}{fc:>9} {ft:<20}{text:<80}\n")
      hash_chunks[idx] = (type, text) # ignore if type == Data ??

   OUT.write("\n")
   return(hash_chunks)


def update_page(text, type, entity, idx):

   entity_txt = entity[0]
   entity_type = entity[1]
   entity_idx = entity[2]
   block_ID = idx[0]
   item_ID = idx[1]
   sub_ID = idx[2]
   k = len(entity_txt)
   if k > 0:
      old_type = entity_type[k-1]
      old_idx = entity_idx[k-1]
      old_block_ID = old_idx[0]
      old_item_ID = old_idx[1]
      old_sub_ID = old_idx[2]
   else:
      old_type = ""
      old_block_ID = ""
      old_item_ID = ""
      old_sub_ID = ""
   if type in ('Note', 'Data'):
      sep = "|"
   else:
      sep = " "

   if (type == old_type and block_ID == old_block_ID
        and item_ID == old_item_ID and sub_ID == old_sub_ID):
     new_text = entity_txt[k-1] + text + sep
     entity_txt[k-1] = new_text
   else:
      entity_txt.append(sep + text + sep)
      entity_type.append(type)
      entity_idx.append(idx)
   return(entity)


def convert_pdf_to_json(doc_ID, filename):

   pdf_path = filename + '.pdf' # input PDF
   json_path = filename + '.json' # PDF turned to Json (unused)
   text_path = filename + '.txt' # PDF turned to text (unused, for debugging)

   OUT = open(text_path, "wt", encoding="utf-8")
   hash_chunks = {}

   # Open the PDF file
   pdf_document = fitz.open(pdf_path)
   content = ""

   # Iterate through the pages
   for page_num in range(len(pdf_document)):
      OUT.write("\n-----------------------------\n")
      OUT.write("Processing page " + str(page_num) + "\n\n")
      print("Page:", page_num)
      page = pdf_document.load_page(page_num)

      text_data = page.get_text("dict") # also extract as "json" to get tokens in green font

      tabs = page.find_tables()
      for tabs_index, tab in enumerate(tabs):
         # iterate over all tables
         index = (page_num, tabs_index)
         table_data = tab.extract() # extracting tabs[i], the i-th table in this page
```

```
            if len(table_data) > 0:
               # if not, ignore this table (note the important parameter threshold here)
               OUT.write("Table " + str(index) + ":\n")
               for row in table_data:
                  OUT.write(str(row) + "\n")
               OUT.write("\n")

         itemize = False
         item_ID = -1
         sub_ID = -1
         block_ID = -1
         item = ""
         fsm = -1 # top level font size in bullet list
         fst = 64 # min title font size (top parameter)
         title = ""
         notes = ""
         old_block_number = -1
         old_font_size = -1
         entity_txt = []
         entity_idx = []
         entity_type = []
         type = ""
         old_text = ""
         old_type = ""

         for block in text_data["blocks"]:
            if block["type"] == 0: # Text block
               block_number = block["number"]
               for line in block["lines"]:
                  for span in line["spans"]:

                     text = span["text"]
                     font_name = span["font"]
                     font_size = span["size"]
                     font_size = round(font_size,1)
                     font_color = span["color"]

                     if font_size > fst:
                        type = 'Title'
                     elif ord(text[0]) == 8226:
                        itemize = True
                        if fsm == -1:
                           fsm = font_size
                        if font_size > 0.98 * fsm:
                           item_ID += 1
                           type = 'List'
                        else:
                           sub_ID +=1
                           type = 'SubList'
                     elif itemize:
                        #- itemize = ((0.99 < font_size/old_font_size < 1.01) and
                        #-          (not text[0].isupper() or ord(old_text[0]) == 8226))
                        itemize = ((0.99 < font_size/old_font_size < 1.01) or
                                 (ord(old_text[0]) == 8226))

                        if not itemize:
                           item_ID = -1
                           sub_ID = -1
                           block_ID += 1
                           type = 'Note'
                     else:
                        if not text[0].isdigit() and text[0] not in ('$', '+', '-'):
                           type = 'Note'
                        #- elif block_number != old_block_number:
                        else:
                           type = 'Data'

                     if block_ID == -1:
                        block_ID += 1
                     elif ((type not in (old_type, 'List', 'SubList')) or
                           (not (0.99 < font_size/old_font_size < 1.01))):
                        if (old_text != "" and ord(old_text[0]) != 8226 and
                              type not in ('List', 'SubList')):
                           block_ID += 1

                     idx = (block_ID, item_ID, sub_ID, doc_ID, page_num, font_size,
```

```
                       font_color, font_name, block_number)
                  entity = (entity_txt, entity_type, entity_idx)
                  update_page(text, type, entity, idx)

                  old_font_size = font_size
                  old_text = text
                  old_type = type

               old_block_number = block_number

      entity = detect_table(entity)
      cprint_page(entity, hash_chunks, OUT)


      image_list = page.get_images()
      for image_index, img in enumerate(image_list, start=1):
         xref = img[0]
         base_image = pdf_document.extract_image(xref)
         image_bytes = base_image["image"]
         size = len(image_bytes)
         ext = base_image["ext"]
         index = (page_num, image_index)
         OUT.write(f"Image {str(index):<8}{str(ext):>5} size = {str(size):>6}\n")
         #- with open(f"image_{page_num + 1}_{image_index}.{ext}", "wb") as image_file:
         #-    image_file.write(image_bytes)

      content += "__P" + str(page_num) + "\n" + page.get_text("json") # "text", "html", "json"

   OUT.close()
   with open(json_path, 'w', encoding='utf-8') as json_file:
      json_file.write(content)
   return(hash_chunks)


# --- Main ---

def get_value(hash, key):
   if key in hash:
      return(hash[key])
   else:
      return('')

filenames = (
           'nvda-f2q24-investor-presentation-final-1',
           '2022-Annual-Review',
           '2023-Annual-Report-1',
         )

outfilename = "repository_Nvidia_X.txt" # xLLM input corpus
OUT_xllm = open(outfilename,"wt")


#-- loop over PDFs

entity_list = ()

for doc_ID in range(len(filenames)):

   filename = filenames[doc_ID]
   hash_chunks = convert_pdf_to_json(doc_ID, filename)

   #-- Add small chunks to xLLM input

   hash_chunkTitle = {}
   for idx in hash_chunks:
      content = hash_chunks[idx]
      # next 2 code lines to track duplicate chunks
      if content[1] not in entity_list:
         entity_list = (*entity_list, content[1])
      if content[0] == "Title":
         hash_chunkTitle[(idx[3], idx[4])] = content[1]

   uID = 0
   hash_entities = {}

   for idx in hash_chunks:
```

```
            content = hash_chunks[idx]
            title = get_value(hash_chunkTitle, (idx[3],idx[4]))
            hash_entities[uID] = {
               'title_text': title,
               'description_text': content[1],
               'type': content[0],
               'doc_ID': idx[3],
               'pn': idx[4], # page number
               'block_ID': idx[0],
               'item_ID': idx[1],
               'sub_ID': idx[2],
               'fs': idx[5], # font size
               'fc': idx[6], # font color
               'ft': idx[7], # font typeface
              }
            uID += 1

         for ID in hash_entities:
            # print(ID, hash_entities[ID])
            findex = 'B' + str(ID) + 'X' + str(doc_ID)
            OUT_xllm.write(findex + "~~" + str(hash_entities[ID]) + "\n")

         #-- Add big chunks to xLLM input

         hash_bigEntities = {}

         for idx in hash_chunks:
            content = hash_chunks[idx]
            title = get_value(hash_chunkTitle, (idx[3],idx[4]))
            pn = idx[4], # page number
            doc_ID = idx[3]
            ID = (pn, doc_ID)
            text = content[1]
            type = content[0]
            if ID in hash_bigEntities:
               local_hash = hash_bigEntities[ID]
               local_hash['description_text'] += ". " + text
               if local_hash['title_text'] == '' and title != '':
                  local_hash['title_text'] = title
               hash_bigEntities[ID] = local_hash
            else:
               hash_bigEntities[ID] = {
                  'title_text': title,
                  'description_text': content[1],
                  'doc_ID': idx[3],
                  'pn': idx[4], # page number
               }

         vID = 0
         for ID in hash_bigEntities:
            # print(ID, hash_bigEntities[ID])
            local_hash = hash_bigEntities[ID]
            text = local_hash['description_text']
            if text not in entity_list:
               findex = 'A' + str(vID) + 'X' + str(doc_ID)
               OUT_xllm.write(findex + "~~" + str(hash_bigEntities[ID]) + "\n")
            else:
               # this big chunk is identical to a small chunk
               # maybe because small chunks are not granular enough
               # maybe due to identical big chunks found in multiple PDFs
               # ignore to avoid duplicate chunks, to save space
               doc_ID = local_hash['doc_ID']
               pn = local_hash['pn']
               print("Ignored: Doc_ID=%3d pn=%3d vID=%4d" %(doc_ID, pn,vID))
               pass
            vID += 1

      OUT_xllm.close()


      # --- PDF to Images ---

      def PDF_to_PNG(doc_ID):

         from PIL import Image

```

```
    pdf_path = filenames[doc_ID] + '.pdf'
    pdf_document = fitz.open(pdf_path)
    zoom = 2 # to increase the resolution
    mat = fitz.Matrix(zoom, zoom)

    for page_num in range(len(pdf_document)):
        page = pdf_document.load_page(page_num)
        pix = page.get_pixmap(matrix = mat) # or (dpi = 300)
        img = Image.frombytes("RGB", [pix.width, pix.height], pix.samples)
        filename = "PDF" + str(doc_ID) + "_" + str(page_num) + '.png'
        img.save(filename)
        print('Converting PDFs to Image ... ' + filename)

    return()

# set save_PNG = True to save PDF slides as PNG

save_PNG = False
if save_PNG:
    for doc_ID in (0, ):
        PDF_to_PNG(doc_ID)
```

3 Backend architecture

The architecture is essentially the same as described in Part 1 in [1], relying on in-memory nested hashes for backend tables, to increase speed by orders of magnitude. These tables are generated on the first run or when you update the backend parameters, and then saved as text files. It takes a few seconds to build them. On subsequent runs, they are imported for even faster processing. By backend, I mean corpus processing, while frontend deals with prompt processing.

There are several components that differentiate xLLM – our architecture – from standard LLMs, such as no transformer, no prompt engineering, no training, no weight, yet increased speed, accuracy, and exhaustivity. Also, explainable AI (intuitive parameters), much lower cost, reproducibility, and almost no reliance on external libraries or APIs, making the system more secure. The version described here is Enterprise xLLM.

Other improvements include several token types, long multi-tokens, variable length embeddings, smart relevancy scores and ranking, generic PMI (pointwise mutual information) instead of cosine similarity, and hierarchical chunking with multi-index. The concepts new to this version are described in section 2, 3.3, 3.4, and 4. Here I describe the main token types. Tokens consist of one or multiple words. In the latter case, words are separated by '∼'. A word such as 'San Francisco' is a single token, though 'San' and 'Francisco' may also be single tokens, offering some redundancy. The following token types encompass both single and multi-tokens.

- s_tokens come from the stemming process: if 'gaming' is in the prompt, the words 'game' and 'games' will also be searched for. Because they are different from the original word, they are flagged as s_tokens, and given a lower weight in the relevancy algorithm. Soon to be implemented.
- g_tokens are tokens found in the contextual fields attached to chunks, such as agent, tag, title, or category. Named `gword` in the code, the prefix 'g' indicates that they are connected to the knowledge graph. They are given a higher weight in the relevancy algorithm.
- c_tokens are multi-tokens consisting of words that are not adjacent in the corpus. I do not use them here.

3.1 Backend tables

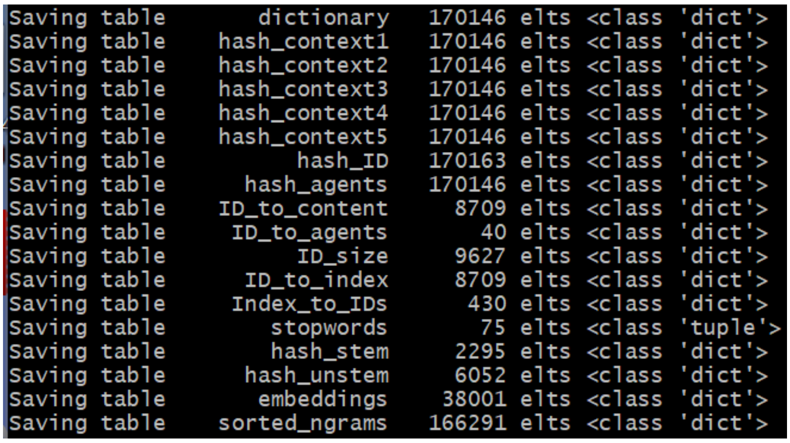
There are four types of backend tables:

- Auxiliary tables for stemming or un-stemming (`hash_stem` used in the backend, `hash_unstem` used in the frontend but built in the backend) and the `stopwords` list. Also in the same category, `Index_to_ID` maps chunk IDs to the corresponding document and page numbers; these two numbers constitute the table key.
- Indexed by a chunk. They start with 'ID' in the table name, where ID is the multi-index attached to the chunk in question. See also section 2.1.
- Indexed by a multi-token: `dictionary`, `hash_agents`, `hash_ID`, `embeddings`, `sorted_ngrams`, as well as those where the table name starts with `hash_context`. The later deals with the contextual fields such as title, tag, or category, as opposed to the actual content attached to a chunk.
- Indexed by a pair of related multi-tokens: `sorted_ngrams`. This table is used to create embeddings, to suggest related prompts to the user. It requires the large temporary `hash_pairs` table; the latter is not

saved or loaded when reading tables from text files, as it is used in the building process only, and never accessed from the frontend. The `ctokens` table, a contextual version of `hash_pairs`, is not used here.

Backend tables are loaded in memory. Many are stored as nested hashes. They are saved as text files. To read them, see the `read_backend_tables` function in the code in section 5.1, itself relying on sub-functions from our internal library. That library is imported as `exllm`, see line 4 in the same code. Backend tables are defined in lines 12–38. Also, you can find them on GitHub, here. Each row correspond to a specific key in the table.

Reading these tables is greatly facilitated using the `eval` Python function that turns strings into nested hashes and can detect integers and floats. Thanks to it, I was able to remove and simplify a number of reading functions in our internal library; the price to pay is a slight reduction in speed. To build the tables (as opposed to reading them), set `build_tables` to True. When running the code, you will see the tables – whether built or read – displayed on the screen as in Figure 1, with the number of rows per table, and the table type.



```
Saving table      dictionary   170146 elts <class 'dict'>
Saving table   hash_context1   170146 elts <class 'dict'>
Saving table   hash_context2   170146 elts <class 'dict'>
Saving table   hash_context3   170146 elts <class 'dict'>
Saving table   hash_context4   170146 elts <class 'dict'>
Saving table   hash_context5   170146 elts <class 'dict'>
Saving table         hash_ID   170163 elts <class 'dict'>
Saving table     hash_agents   170146 elts <class 'dict'>
Saving table   ID_to_content     8709 elts <class 'dict'>
Saving table    ID_to_agents       40 elts <class 'dict'>
Saving table         ID_size     9627 elts <class 'dict'>
Saving table     ID_to_index     8709 elts <class 'dict'>
Saving table    Index_to_IDs      430 elts <class 'dict'>
Saving table       stopwords       75 elts <class 'tuple'>
Saving table       hash_stem     2295 elts <class 'dict'>
Saving table     hash_unstem     6052 elts <class 'dict'>
Saving table      embeddings    38001 elts <class 'dict'>
Saving table   sorted_ngrams   166291 elts <class 'dict'>
```

Figure 1: Backend tables, with type and number of rows

3.2 Backend parameters

Backend parameter values are set in the `build_backend_tables` function. See code in section 5.1, also shown below for convenience.

```
backendParams = {
   'max_multitoken': 4, # max. consecutive terms per multi-token
   'maxDist' : 3,    # max. distance between 2 multi-tokens to link them in hash_pairs
   'create_hpairs': True,    # required for embeddings, takes time and memory
   'create_ctokens': False,  # needs create_hpairs set to TRUE for activation
   'use_stem': False,        # backend stemming
   'extraWeights':           # default weight is 1
     {
       'description': 0.0,
       'category': 0.0, # try 0.5
       'tag_list': 0.0, # try 0.5
       'title':   0.0,  # try 0.5
       'meta':    0.0   # try 0.5
     }
}
```

If you set `create_hpairs` to False, embeddings won't be created. It saves some time and space, but prevents the user from seeing related keywords that could be used for extra prompting. The `extraWeights` values allow you to boost the importance of multi-tokens found in contextual elements, compared to those in the standard text. Finally, a higher `maxDist` increases the width of the contextual window for embeddings, but also leads to more memory usage, and bigger `hash_pairs` and `embeddings` tables especially for large chunks. It leads to more results potentially shown to the user, though distillation and the scoring engine, controlled by the frontend parameters, will only pick up only the most relevant ones.

3.3 Backend stemming

With `'use_stem':True` in `backendParams`, words in the corpus are replaced by their most common alternative that is also present in the corpus. Table 1 shows the original word in column $\texttt{token}_1$, with its substitution in column $\texttt{token}_2$. Columns n_1, n_2 are the number of occurrences in the corpus, respectively for the original word, and its substitution. It allows you to easily retrieve and group the different versions of a corpus word. It also results in smaller backend tables and slightly faster processing. But it sometimes produce inaccurate answers when the stemming is too aggressive. The default is `'use_stem':False`.

n_1	n_2	token_1	token_2
8	42	region	regions
12	42	regional	regions
8	188	register	registered
26	70	registrants	registrant
12	70	registration	registrant
85	182	regulation	regulations
2	182	regulator	regulations
16	182	regulators	regulations
8	14	reimburse	reimbursement
4	14	reimbursed	reimbursement
4	14	reimbursements	reimbursement
2	12	reincorporation	reincorporated
2	6	reinforce	reinforced
4	6	reinvent	reinvented
2	6	reinventing	reinvented
2	6	reinvents	reinvented
4	12	reinvest	reinvested
8	12	reinvestment	reinvested

Table 1: Backend table `hash_stem` (extract)

The code below, in the `update_dict` function in the `xllm_enterprise_nvidia_util` library, does the substitution. The algorithm behind the scenes is explained in section 4.3. Note that n_1, n_2 in Table 1 are usually even integers due to double counting: each token appears both in a sub-chunk and in its chunking, unless the parent has only one child.

```
for word in words:

   if use_stem and word in hash_stem:
      word = hash_stem[word]
```

3.4 Adding tags, categories, or agents

There are two flavors of agents. In both cases, the goal is to assess the user intent and serve results accordingly. Retrieval agents focus on detecting the type of content the user is looking for: best practices, definitions, templates, datasets, code, spreadsheets, images, tables, and so on. Action agents perform executive functions, such as solving a math problem, generating code, generating synthetic data, auto-tagging, auto-indexing, cataloging, summarizing, predictive analytics, or text clustering.

Action agents rely on API calling external apps such as Wolfram to solve math problems. However, xLLM has high quality internal apps for a number of tasks, such as cataloging, text clustering, predictive analytics on text data, and tabular data synthetization. In contrast to standard LLMs that try to assign agents to prompts, xLLM offers the user the option to select retrieval agents listed on the command menu. Also, agents are assigned to chunks along with other contextual elements such as titles, tags, or categories. Unlike titles, agents are assigned post-crawling based on business needs and corpus content. In the Nvidia use case, images and tables are two possible values for the agent. To create a meaningful list of agents, you need to look at top multi-tokens found in the `dictionary` backend table. Tags and categories are built using the same principle.

In the current version of xLLM, agents are detected and assigned to chunks in the `build_backend_tables` function in `xllm-enterprise-nvidia-dev.py`. Look for the keyword 'agent' in the code in question. The plan is to have them assigned earlier when building the xLLM file with the PDF parser. Either way, they end up in the `ID_to_agents` backend table. In this table, the key is a chunk represented by its ID (nulti-index), and the value is the list of associated agents. Chunks with no agent are not included in that table. This allows for easy agent retrieval in the frontend. See `agent_map` for the current list of agents.

```
     ID      ID_agents
   B2X0      ('income', 'tax', 'cash', 'products', 'data', 'accelerated computing', 'data center', 'non-gaap')
 A154X1      ('stock', 'income', 'tax', 'cash', 'revenue', 'common stock', 'financial statements')
 A175X1      ('stock', 'cash', 'data', 'assests', 'financial statements', 'restricted stock', 'data center')
 A179X1      ('cash', 'securities', 'assests', 'financial statements')
 A182X1      ('income', 'cash', 'equity', 'securities', 'assests', 'financial statements')
 A186X1      ('products', 'assests', 'financial statements')
B2250X2      ('stock', 'financial statements')
B2749X2      ('stock', 'income', 'tax', 'cash', 'directors', 'common stock', 'securities', 'financial statements')
```

Figure 2: Sample prompt results, agents view corresponding to summary view in Figure 3

Figure 2 shows the agents view, that is, the agent list attached to each chunk relevant to the prompt. In this example, the prompt is 'financial statements 2024', and chunks are represented by their ID (a multi-index). I also discuss this prompt in section 4.5, featuring the summary and embeddings views. These three views are part of the prompt results. Here, we are dealing with retrieval agents. All of them are soft agents, defined as keywords found in the chunk content, as opposed to hard agents, defined as agents attached to the contextual fields of a chunk such as tags, categories, or agents. Hard agents have names that start with a capital letter, by contrast to soft agents. Note that soft retrieval agents play the same role as pre-assigned tags. They are useful to decide whether or not you want to see the full content of a specific chunk, before clicking on it to get the detailed view.

4 Frontend architecture

The frontend deals with how to connect prompt tokens to backend tables, what results to display to the user, in what order, and how: structured output with concise but exhaustive, original results not reworded differently, by contrast to standard LLMs. The main components are unstemming, sorted n-grams, frontend distillation, and the relevancy scores. The UI allows for real-time fine-tuning. It also offers agents or categories to choose from, as well as entering multi-tokens or negative keywords in the prompt box. Finally, it displays embeddings to suggest related prompts, and the ability to retrieve full chunks from the corpus, including tables, URLs, and images, in just one click. It also displays relevancy scores attached to each chunk featured in the prompt results.

4.1 Frontend tables

The two main tables are `q_dictionary` and `q_embeddings`. They are local versions of the `dictionary` and `embeddings` backend tables. The prefix 'q' indicates that they are restricted to the data detected in the user query (the prompt). The `sorted_ngrams` (built in the backend) allows you to very efficiently check all 2^n combinations of frontend keywords $A_1, \dots, A_n$ in the prompt, to identify the few ones found in the corpus, and stored as multi-tokens in the `dictionary`. If frontend stemming is activated, the backend table `hash_unstem` is also used.

Also, `ID_hash` is a local (prompt-related), transposed version of the backend table `hash_ID`. It is a multi-level key-value table, also called nested hash. The key is a corpus chunk represented by its multi-index. The value is a hash table (sub-hash) containing the multi-tokens found simultaneously in the corresponding chunk and in the prompt. The sub-key is a multi-token, and the sub-value is the multi-token multiplicity: the number of occurrences of the multi-token in question, in the parent chunk.

Finally, `ID_score` is built using `ID_hash`, and assigns a multivariate score measuring the relevancy between a specific chunk, and the prompt. The score vector is made of sub-scores, each measuring a particular aspect of relevancy: for instance, the presence of contextual multi-tokens matching those in the prompt, the number of tokens found simultaneously in a chunk and in the prompt, and so on. Each sub-score is sorted separately to produce sub-ranks, see `ID_score_ranked`. This simple hash table is indexed by a chunk ID; the value is a score vector. Then the final rank attached to a chunk, relative to a prompt, is a weighted sum of the sub-ranks.

4.2 Frontend parameters

The frontend parameter values are set in the `default_frontendParams` function in our internal library, see code in section 5.2. The list is much shorter compared to the previous version discussed in Part 1 in [1]. However, additional parameters will be added to better fine-tune the relevancy scores and distillation function. The PMI function is now moved to the backend, thus eliminating frontend PMI parameters.

To use frontend stemming, set `use_stem` to True and make sure any glitch from the Nltk Python library are addressed with the workaround discussed in section 4.3, to avoid low relevancy on occasions (or hallucinations in the case of standard LLMs). Note that `maxTokenCount` is attached to the frontent distillation function, activated when `distill` is set to True. The smaller the value, the stronger the impact. The parameter `beta` is attached to the relevancy score. Several relevancy parameters will soon be added: what they will control is discussed in see section 4.4. The current frontend parameter list in the code below.

```
frontendParams = {
                 'distill': False,
                 'maxTokenCount': 1000, # ignore generic tokens if large enough
                 'beta': 1.0, # used in text entity relevancy score, try 0.5
                 'nresults': 20, # max number of chunks to show in results
                 'use_stem': False,
                }
```

In the end, the goal is to offer a small number of intuitive parameters based on explainable AI, to make real-time fine-tuning easy for non-experts. As previously, there will be a catch-all parameter set for the expert user who wants to see the full output instead of the selection displayed on the screen. This helps understand the reason why some results may be missing depending on the parameter values. It is a valuable feature for LLM debugging. Currently, for debugging, set `distill` and `use_stem` to False, and set `nresults` (the maximum number of chunks returned) to a very large value, say 1000.

4.3 Frontend stemming

The goal is to find in the corpus keywords related to those in the prompt. A simple example is 'projection' not found in the corpus, even though 'projections' is there. Also, if 'gaming' is a prompt token also present in the corpus, it makes sense to retrieve 'games' and 'game', not found in the prompt but present in the corpus. Yet a more complex example is 'projection' related to 'prediction' and 'forecast'.

For stemming, I use the PorterStemmer available in the NLTK Python library. See the `stem` function in our `xllm_enterprise_nvidia_util` library. It produces the `hash_unstem` backend table featured in Table 2, addressing a complex problem known as unstemming. Despite being one of the least aggressive stemmers, it is still too aggressive for a system like ours that does not rely on transformers. Some entries in `hash_unstem` must be broken down into smaller pieces to significantly improve performance. As an example, 'project' and 'projection' cannot have the same stem. The full `hash_unstem` table is in a zip archive on GitHub, here.

Frontend stemming works as follows. If the word 'games' is in the prompt, xLLM looks for all alternate versions found in the corpus, such as 'gaming', yet giving a higher weight to the original word in the prompt. First, 'games' needs to be stemmed to 'game' using the PorterStemmer, then unstemmed to all variants using `hash_unstem['game']`. This feature is activated if the frontend parameter `use_stem` is set to True.

stem	related corpus tokens
enterpris	enterprises, enterprise
race	racing, race, races
adopt	adopt, adoption, adopted, adopts, adopting
belief	belief, beliefs
global	global, globally
end	end, ends, ended, ending
game	gaming, games, game, gamings
return	returned, returns, return, returning
grow	growing, grows, grow
number	number, numbers
gpu	gpu, gpus
cloud	cloud, clouds
provid	providers, providing, provider, providence
deploy	deploying, deploy, deployed, deployment

Table 2: Backend table `hash_unstem` (extract)

If in addition, the backend parameter with same name `use_stem` is also set to True, then backend tables are smaller and retrieval is faster. It may result in lower accuracy and works with minimum stemming only. Note that stemming can be done semi-manually, by first sorting all single tokens (consisting of one word) in the `dictionary` backend table, and then group close neighbors while browsing the dictionary, avoiding false positives (grouping unrelated tokens) and false negatives (failure to identify two tokens as related).

4.4 Scoring engine

See high-level discussion in section 4.1, explaining the score vector, sub-scores, sorted scores and sub-ranks, and how the final rank attached to a corpus chunk is computed, relative to the prompt. This component is being enhanced, to better take into account s_tokens (synonyms to tokens in the prompt), g_tokens (found in contextual fields attached to chunks), single versus multi-tokens, token multiplicity, chunk size, token rarity, whether a token is first in the prompt or not, favoring recent content, and so on.

4.5 Advanced user interface with-real time fine-tuning

While most LLM and RAG systems offer nothing more than a basic search box to enter prompts, our platform comes with a UI offering several options. Prompt results are concise, well structured and allow the user to get a general overview before digging deeper via subsequent actions or prompts. See description in Chaper 3 in [1]. The main features include

- Fine-tuning frontend parameters in real time,
- The user can choose specific tags, categories (sub-LLMs) or agents on the menu when playing with the web API, or on the command line in the offline version,
- New to this version are the options to enter negative keywords and multi-tokens in the prompt query. For instance, a search that includes the multi-token 'public∼conference∼call' will only return chunks that contain all words glued together. If adding the negative keyword '!publication' (with the exclamation point to indicate that the keyword is negative), it will ignore chunks that also include the word 'publication'.
- Processing prompts in bulk with results saved in text file. Ability to re-use a previous prompt and modify them, or re-use a previous set of parameters, or the catch-all parameter to return everything found in the corpus and related to the prompt (useful for debugging purposes).
- Relevancy scores attached to each chunk displayed in the prompt results. The most relevant chunks are displayed each with a short summary, including associated contextual elements such as tags, categories, title, agents, chunk size, and relevant tokens matching those in the prompt. The user can click on specific chunks to see the full details (tables, images, full text and so on).

Figure 3 shows the summary view for the prompt 'financial statements 2024'. It displays eight retrieved chunks: 5 sub-chunks with ID starting with A, and 3 parent chunks with ID starting with B. The columns 'PDF' and 'pn' respectively show the PDF identifier and page number within that document. Column 'wRank' shows the weighted rank, an indicator of relevancy to the prompt. Chunk A175X1 contains 'financial∼ statements' glued together rather than separate; also its size is smaller than the top 2 chunks, making it more concentrated, and thus, potentially more interesting. By choosing specific chunks, you can dig deeper and retrieve the full content in the detailed view (not shown here). The context view shows the chunks with tags, titles, tables, images, and agents attached to them (not shown here) while the embeddings view in Figure 4 shows related keywords to use in subsequent prompts. The left column in that view shows the PMI, an indicator of relevancy. Note that 'reconciliation' is detected as a corpus word related to the prompt token 'financial'. For the agents view, see Figure 2 in section 3.4.

```
------------------
Prompt:  financial statements 2024
Cleaned: ['2024', 'financial', 'statements']
------------------
Most relevant text entities:


       ID wRank   size PDF   pn ID_Tokens
     B2X0     5   6836   0    1 {'statements': 12, 'financial': 12, 'statements~financial': 1, '2024': 1}
   A154X1     7   4083   1  158 {'statements': 4, 'financial': 4, 'financial~statements': 4, '2024': 1}
   A175X1     7   2400   1  179 {'statements': 1, 'financial': 2, 'financial~statements': 1, '2024': 1}
   A179X1     7   2687   1  183 {'statements': 1, 'financial': 1, 'financial~statements': 1, '2024': 1}
   A182X1     7   1722   1  186 {'statements': 1, 'financial': 1, 'financial~statements': 1, '2024': 1}
   A186X1     7   2319   1  190 {'statements': 1, 'financial': 1, 'financial~statements': 1, '2024': 2}
  B2250X2     7   1992   2   85 {'statements': 1, 'financial': 1, 'financial~statements': 1, '2024': 3}
  B2749X2     7   2662   2  127 {'statements': 2, 'financial': 4, 'financial~statements': 2, '2024': 1}
```

Figure 3: Sample prompt results, summary view

By allowing the user to choose frontend parameters in real-time, the system can collect the most popular parameter sets to automatically build a selection of default or template parameter sets. This is known as self-tuning. Finally, in the next version, it will be possible to give a higher weight to the first token in the prompt, and to favor the most recent results when timestamps are included in the corpus. Also, in the next version, the user will have the ability to browse next and previous chunks, starting at a chunk displayed in prompt results: this feature is easy to implement since chunks are sequentially indexed during the crawl. In short, the UI is an LLM browser of its own, rather than a search box. Its scoring engine is described as the new PageRank for LLMs.

```
Top related tokens (via embeddings):

     0.30 ('financial', 'non-gaap~gaap')
     0.30 ('financial', 'gaap~financial')
     0.30 ('financial', 'non-gaap~gaap~financial')
     0.30 ('financial', 'gaap~financial~measures')
     0.30 ('financial', 'reconciliation')
     0.29 ('financial', 'financial~measures')
```

Figure 4: Sample prompt results, embeddings view

4.6 Testing and Evaluation

Evaluating LLMs is tricky for a number of reasons. Standard evaluation and benchmarking metrics fail to assess important qualities, such as exhaustivity, conciseness, depth and structuredness of prompt results. Part of the problem is because standard LLMs are trained to predict missing or next tokens, an important task in word guessing and earlier versions such as Bert, but now largely irrelevant to the problems it aims to solve.

It is compounded by the fact that wordy English prose is valued by many users, over concise, yet deep and well structured answers. Also, professional users and laymen will rate the results to a same prompt very differently, though a well designed LLM could deliver different results depending on the type of user. In our case, the user can choose agents or categories on the command menu when doing a search, and even negative keywords, to get more relevant results. Finally, our LLM shows relevancy scores attached to each item in prompt results. None of these features are taken into account in standard evaluation metrics.

To compare two different versions of our LLM, to check whether upgrades or different parameters actually lead to better performance, you want to separately test the different components impacted before a new release. More specifically:

- **Chunking**. Look for chunks that are too large or too small. Check the variance in chunk size, and also for the distribution of sub-chunk sizes within each parent chunk.
- **Stemming**. Try with and without stemming on the backend, with the **backend** parameter `use_stem` set to True or False. In the `hash_unstem` table, look for entries with 3 or more tokens attached to a stem, especially for stems with high occurrence; see whether they should be broken down into smaller pieces to make the stemming less agressive and less ambiguous. Test with the **frontend** parameter `use_stem` set to True or False.
- **Duplicates**. Eliminate duplicate chunks, unless business reasons tell otherwise: if the same parent chunk is in two different PDFs, it might make sense to keep both copies. For sub-chunks, do not dedupe.
- **Distillation**. Done on the frontend. Test with and without distillation, by setting the frontend parameter `distill` to True or False.
- **Scoring**. Look at the rank and score attached to each chunk returned in the prompt results, for test prompts. Play with the related frontend parameters to fine-tune the scores. Are g_tokens and s_tokens given the proper weights? Are g_tokens correctly detected? Are multi-tokens favored over single tokens? Is token multiplicity and specialized tokens given the proper weights? Multiplicities are found in the nested hash `ID_hash[ID]`, where `ID` identifies a chunk, and `ID_hash[ID]` is the daughter hash storing its multi-tokens (the keys) that are also in the prompt with their multiplicities (the values).
- **Context**. Are there many chunks missing contextual fields, such as title or agent? Are tables correctly detected and indexed with the multi-index system? Are agents correctly assigned to chunks?
- **Speed**. Large chunks can significantly increase the size of backend tables `hash_pairs` and `embeddings`. These two tables also take more time to build, yet are not critical components of the system. I use them to suggest related prompts. Removing duplicate chunks requires `entity_list` which may take a lot of memory. Do we want to load all the chunks in memory (faster), especially the parent chunks which can be built using the sub-chunks? It is OK to have slower backend processing, as long as frontend is very fast. To improve speed and reduce memory usage, use lower values for the backend parameters `max_multitoken` and `maxDist`. Also, you can set `create_hpairs` to False: then no embedding is generated.

You may also try a different PMI function. PMIs are used in our variable length embeddings, as an alternative to cosine similarity and dot product. Also, you want to test with and without building the backend tables. Not building them means that they are pre-loaded in memory with `build_tables` set to False. You may want to verify that this step (reading the tables from text files and retrieving the correct architecture) works correctly.

As for actual evaluation, a possible test consists of extracting various pieces of text from the PDFs and check if they are retrieved depending on the prompts. In this case, the prompts are shorter, scrambled, cleaned versions of the original extract, possibly using different words. For instance, in the future, we will have a synonyms dictionary matching (say) 'forecast' to 'projection' and 'prediction', in case the word 'forecast' is in a prompt but not in the corpus. And even if in the corpus, to retrieve text containing not just 'forecast', but also its synonyms. Same with 'gen ai', 'genai' and 'generative ai'.

5 Python code

The Python code is back-compatible with the previous version published in [1]. It still relies on fast in-memory databases stored as nested hashes, for backend tables. However, it has been significantly optimized for speed without increasing memory usage or table sizes, especially to deal with very large chunks. Finally, it has been significantly simplified, with some parameters no longer needed, and new ones added.

Now, you can read the backend tables from text files rather than creating them each time. Also, a number of functions have been moved to an internal library (see section 5.2) to make the code more readable. It also features new NLP functions related to stemming and unstemming.

5.1 Main program

The most recent version of the code is on a shared Google Drive, here. It imports the internal library listed in section 5.2. The prompts are read from a text files available on GitHub, here. A number of components need improvements, in particular the scoring engine, the stemmer, and g_tokens (`gword` in the code). Some need to be fully built, such as showing tables and images in the prompt results, overlapping chunks, more or better contextual fields, improved table detection, and showing absolute rather than relative scores in prompt results (the latter are just chunk ranks).

```
# xllm-enterprise-nvidia-dev.py
# text entity / sub-entity have same meaning as chunk / sub-chunk

import xllm_enterprise_nvidia_util as exllm

#--- Backend: create backend tables based on crawled corpus

# hash_pairs, ctokens are intermediate tables used when building backends
# no need to save them, or to load them when build_tables = False
# hash_context4 now contains IDs (short) rather than associated text (long)

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', # chunk IDs attached to multitoken (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)
  '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)
  'ID_to_index', # map ID to multi-index (key = text entity ID)
  'Index_to_IDs', # subIDs attached to ID, with size
  'stopwords', # stopword list
  'hash_stem', # stemmed words in dictionar
  'hash_unstem', # match prompt words to multitokens (key = stemmed word)
  'embeddings', # to show related keywords in prompt results (key = multitoken)
  'sorted_ngrams', # to build embeddings (key = multitoken),
  # 'ID_to_tags', # map ID to tag list (key = text entity ID) --- to be added
)

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', 'what', 'some', 'under', 'does', 'belong',
          'included', 'part', 'associated')
backendTables['stopwords'] = stopwords

# agent_map key is corpus word, value is agent (many-to-one)
agent_map = { 'stock': 'Stock',
           'fiscal 2022': 'Year 2022',
           'fiscal 2023': 'Year 2023',
           'income': 'Income',
           'tax': 'Tax',
           'cash': 'Cash',
           'products': 'Products',
           'revenue': 'Revenue',
           'directors': 'Directors',
           'data': 'Data',
           'equity': 'Equity',
           'management': 'Management',
           'common stock': 'Common Stock',
           'securities': 'Securities',
           'assets': 'Assests',
           'financial statements': 'Financial Statements',
           'restricted stock': 'Restricted Stock',
           'accelerated computing': 'Accelerated Computing',
           'data center': 'Data Center',
           'non-gaap': 'Non-GAAP',
          }


#--- Read repository and create all backend tables

def read_backend_tables(backendTables):

   for tableName in backendTables:

      print("reading %16s" %(tableName), end = " ")
      filename = "backend_" + tableName + ".txt"
      if tableName == 'stopwords':
         backendTables[tableName] = exllm.read_list(filename)
      elif tableName in ('dictionary', 'ID_size'):
         backendTables[tableName] = exllm.read_pairs(filename)
      elif tableName in ('hash_stem', ):
         backendTables[tableName] = exllm.read_table(filename, format = "str")
      elif tableName not in ('hash_pairs', 'ctokens'):
         backendTables[tableName] = exllm.read_table(filename)
      print("(size: %8d)" %(len(backendTables[tableName])))

   return(backendTables)


def build_backend_tables(repository, backendTables, save=True):

   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
      'create_hpairs': True, # required for embeddings, takes time and memory
      'create_ctokens': False, # needs create_hpairs set to TRUE for activation
      'use_stem': False, # backend stemming
      'extraWeights' : # default weight is 1
         {
           'description': 0.0,
           'category': 0.0, # 0.5
           'tag_list': 0.0, # 0.5
           'title':    0.0, # 0.5
           'meta':     0.0  # 0.5
         }
   }

   IN = open(repository, "r")
   data = IN.read()
   IN.close()

   entities = data.split("\n")
   count = 0
   context_fields = ('category_text','tags_list_text','title_text')
```

```
   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:

         n1 = len(entities)
         n2 = len(backendTables['dictionary'])
         n3 = len(backendTables['hash_pairs'])
         if count % 50 == 0:
            print("Processing chunk %4d out of %5d [%6d %6d]" %(count,n1, n2, n3))
         count += 1

         # entity_list = (*entity_list, entity[1]) # avoid duplicates [use lots memory]
         hash_chunk = eval(entity[1])

         # entity_ID can be a string (multi-index)
         entity_ID = entity[0]
         hash_crawl = {}
         hash_crawl['ID'] = entity_ID
         ID_size[entity_ID] = len(entity[1])

         for key in hash_chunk:

               value = hash_chunk[key]
               if key == 'category_text':
                  hash_crawl['category'] = value
               elif key == 'tags_list_text':
                  hash_crawl['tag_list'] = exllm.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 == 'type':
                  hash_crawl['meta'] = value
               if key in context_fields or key in 'description_text':
                  # remove <in 'description_text'>, except in demo
                  for word in agent_map:
                     agent = agent_map[word]
                     if key == 'description_text':
                        agent = agent.lower()
                     if word in value.lower() and agent not in agent_list:
                        agent_list =(*agent_list, agent)

         # hash_crawl['full_content'] = hash_chunk
         hash_crawl['mindex'] = (hash_chunk['doc_ID'], hash_chunk['pn'])
         hash_crawl['agents'] = agent_list
         exllm.update_dict(backendTables, hash_crawl, backendParams)

   #-- 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)
      exllm.update_nestedHash(embeddings, wordA, wordB, pmi)
      exllm.update_nestedHash(embeddings, wordB, wordA, pmi)

   backendTables['embeddings'] = embeddings
```

```
   #-- 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
      exllm.update_listHash(sorted_ngrams, sorted_ngram, word)

   backendTables['sorted_ngrams'] = sorted_ngrams

   #-- Create stem table

   (hash_stem, hash_unstem) = exllm.stem(dictionary)
   for word in hash_stem:
      # need to manually delete some entries [not done yet]
      lead_word = hash_stem[word]
      cnt1 = int(dictionary[word])
      cnt2 = int(dictionary[lead_word])
      print("%3d\t%2d\t%s\t%s" %(cnt1, cnt2, word, lead_word))
   backendTables['hash_stem'] = hash_stem
   backendTables['hash_unstem'] = hash_unstem

   #-- save backend tables

   if save:

      # save backend tables
      for tableName in backendTables:
         if tableName not in ('hash_pairs','ctokens'):
            table = backendTables[tableName]
            print("Saving table %16s %8d elts %s" %(tableName, len(table),type(table)))
            OUT = open('backend_' + tableName + '.txt', "w")
            if type(table) is tuple:
               OUT.write(str(table))
            else:
               for key in table:
                  value = table[key]
                  OUT.write(str(key) + "\t" + str(value) + "\n")
            OUT.close()

      # save backend parameters
      OUT = open('backendParams.txt', "w")
      OUT.write(str(backendParams))
      OUT.close()

   return(backendTables)


#--- Main for backend (reading or creating backend tables)

build_tables = True

if build_tables:
   repository_file = "repository_Nvidia_X.txt"
   build_backend_tables(repository_file, backendTables)
   embeddings = backendTables['embeddings']
   sorted_ngrams = backendTables['sorted_ngrams']
else:
   read_backend_tables(backendTables)

dictionary = backendTables['dictionary']
DICT = open("dict.txt","wt")

for key in dictionary:
   count = dictionary[key]
   ntk = 1 + key.count("~") # number of single tokens in multitokens
   DICT.write(str(count) + "\t" + str(ntk) + "\t" + key + "\n")

DICT.close()


#--- Functions used to score results ---
```

```
def rank(hash):
   # sort hash, then replace values with their rank

   hash = dict(sorted(hash.items(), key=lambda item: item[1], reverse=True))
   rank = 0
   old_value = 999999999999

   for key in hash:
      value = hash[key]
      if value < old_value:
         rank += 1
      hash[key] = rank
      old_value = value
   return(hash)


def rank_ID(ID_score):
   # attach weighted relevancy rank to text entity ID, with respect to prompt

   ID_score0 = {}
   ID_score1 = {}
   ID_score2 = {}
   ID_score3 = {}

   for ID in ID_score:
      score = ID_score[ID]
      ID_score0[ID] = score[0]
      ID_score1[ID] = score[1]
      ID_score2[ID] = score[2]
      ID_score3[ID] = score[3]

   ID_score0 = rank(ID_score0)
   ID_score1 = rank(ID_score1)
   ID_score2 = rank(ID_score2)
   ID_score3 = rank(ID_score3)

   ID_score_ranked = {}
   for ID in ID_score:
      weighted_rank = 2*ID_score0[ID] + ID_score1[ID] + ID_score2[ID] + ID_score3[ID]
      ID_score_ranked[ID] = weighted_rank
   ID_score_ranked = dict(sorted(ID_score_ranked.items(), key=lambda item: item[1]))
   return(ID_score_ranked)


#--- Main for frontend (prompt processing prompts)

print("\n")
input_ = " "
saved_query = ""
get_bin = lambda x, n: format(x, 'b').zfill(n)
frontendParams = exllm.default_frontendParams()
use_stem = frontendParams['use_stem']
beta = frontendParams['beta']
ID_to_content = backendTables['ID_to_content']


#--- Main: Read sample prompts ---

from nltk.stem import PorterStemmer
from collections import defaultdict
stemmer = PorterStemmer()

sorted_ngrams = backendTables['sorted_ngrams']
embeddings = backendTables['embeddings']
ID_to_content = backendTables['ID_to_content']
hash_unstem = backendTables['hash_unstem']

IN = open("enterprise_nvidia_prompts.txt","r")
prompts = IN.read()
prompts = prompts.split("\n")

# --- Main: Look over all prompts ---

for query in prompts:
```

```
    query = query.split("|")[0]
    print("\n------------------")
    print("Prompt: ", query)
    query = query.replace('?',' ').replace('(',' ').replace(')',' ').replace('.',' ')
    query = query.replace("'",'').replace("\\s",'')
    query = query.split(' ')
    for k in range(len(query)):
       query[k] = query[k].lower() # need to eliminate this down the line
    new_query = []
    neg_query = [] # keywords to exclude
    altTokens = ()

    for k in range(len(query)):
       token = query[k]
       if token[0] == '!':
          token = token[1:len(token)]
          neg_query.append(token)
       if use_stem:
          tstem = stemmer.stem(token)
          if tstem in hash_unstem:
             tlist = hash_unstem[tstem]
             for altToken in tlist:
                if token != altToken and altToken not in altTokens :
                   altTokens = (*altTokens, altToken)
       if token in dictionary:
          new_query.append(token)

    q_altTokens = ()
    for altToken in altTokens:
       if altToken not in new_query:
          new_query.append(altToken)
          q_altTokens = (*q_altTokens, altTokens)

    query = new_query.copy()
    query.sort()
    print("Cleaned:", query)
    print("------------------")

    q_embeddings = {}
    q_dictionary = {}

    # --- build q_dictionary and q_embeddings based on prompt tokens ---

    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]
                embedding = exllm.get_value(word, embeddings)
                exllm.add_embedding(q_embeddings, word, embedding)

    # deal with prompt multitokens, if there are any [need to add multitoken stemming]
    for word in query:
       if '~' in word and word in dictionary and word not in q_dictionary:
          q_dictionary[word] = dictionary[word]
          embedding = exllm.get_value(word, embeddings)
          exllm.add_embedding(q_embeddings, word, embedding)

    # --- Scoring and selecting what to show in prompt results ---

    if frontendParams['distill']:
       # gow is this working with negative keywords?
       exllm.distill_frontendTables(q_dictionary,q_embeddings,frontendParams)
    hash_ID = backendTables['hash_ID']
    ID_hash = {} # local, transposed of hash_ID; key = ID; value = multitoken list
```

```python
   for word in q_dictionary:
      for ID in hash_ID[word]:
         local_hash = hash_ID[word]
         if word not in neg_query:
            exllm.update_nestedHash(ID_hash, ID, word, local_hash[ID])
      gword = "__" + word # graph multitoken
      if gword in hash_ID and word not in neg_query:
         for ID in hash_ID[gword]:
            exllm.update_nestedHash(ID_hash, ID, gword, 1)

   ID_score = {}
   for ID in ID_hash:
      # score[0] is inverse weighted count
      # score[1] is raw number of tokens found
      score = [0, 0] # based on tokens present in the entire text entity
      gscore = [0, 0] # based on tokens present in graph (context elements)
      for token in ID_hash[ID]:
         if token in dictionary:
            score[0] += 1/(q_dictionary[token]**beta)
            score[1] += 1
         else:
            # token must start with "__" (it's a graph token)
            token = token[2:len(token)]
            gscore[0] += 1/(q_dictionary[token]**beta)
            gscore[1] += 1
      ID_score[ID] = [score[0], score[1], gscore[0], gscore[1]]

   # --- Print results ---

   ID_score_ranked = rank_ID(ID_score)
   nresults = frontendParams['nresults']

   print("Most relevant chunks with multitokens, doc_ID, pn, rank, size:\n")
   # also show absolute score as opposed to rank
   n_ID = 0
   print("\n     ID wRank size PDF pn ID_Tokens")
   ID_index = backendTables['ID_to_index']
   ID_size = backendTables['ID_size']
   for ID in ID_score_ranked:
      if n_ID < nresults:
         # content of text entity ID not shown, stored in ID_to_content[ID]
         # add tags
         mindex = ID_index[ID] # multi-index
         doc_ID = mindex[0] # PDF number
         pn = mindex[1]   # page number within PDF
         rankx = ID_score_ranked[ID] # need to also create ID_score[ID] (absolute score)
         size = ID_size[ID]
         print(" %8s  %3d %6d %3d %4d %s"
               %(ID, rankx, size, doc_ID, pn, ID_hash[ID]))
      n_ID += 1

   print("Most relevant chunks with agents:\n")
   n_ID = 0
   print("\n     ID   ID_agents")
   ID_to_agents = backendTables['ID_to_agents']
   for ID in ID_score_ranked:
      if n_ID < nresults:
         agents = exllm.get_value(ID, ID_to_agents)
         if len(agents) > 0:
            print(" %8s   %s" %(ID, agents))
      n_ID += 1

   print("\nToken count (via dictionary):\n")
   for key in q_dictionary:
      print(" %4d   %s" %(q_dictionary[key], key))

   print("\nTop related tokens (via embeddings):\n")
   q_embeddings = dict(sorted(q_embeddings.items(), key=lambda item: item[1], reverse=True))
   n_words = 0
   for word in q_embeddings:
      pmi = q_embeddings[word]
      if n_words < 10:
         print(" %5.2f %s" %(pmi, word))
      n_words += 1
```

```
    full_content = False
    if full_content:
       print("\nFull content sorted by relevancy\n")
       # ID starts with A for big chunks, with B for small chunks
       n_ID = 0
       for ID in ID_score_ranked:
          content = ID_to_content[ID]
          rankx = ID_score_ranked[ID]
          size = ID_size[ID]
          if n_ID < nresults and size < 60000:
             print("%8s %3d %5d %s %s\n" %(ID, rankx, size, ID_hash[ID], content))
          n_ID += 1
```

5.2 Internal library

This library is on Google Drive, here. The backend tables are also in the same GitHub folder. In this version, I removed the code for command-line processing (real-time fine-tuning and so on) as prompts are read from a text file. However, it is included in the previous version featured in Part 1 in [1] and the corresponding features will be accessible from the UI in the new Web API under construction.

```
# xllm_enterprise_nvidia_util.py

#--- Stemming

def stem(dictionary):

   from nltk.stem import PorterStemmer
   # from nltk.stem import WordNetLemmatizer
   stemmer = PorterStemmer()
   # lemmatizer = WordNetLemmatizer()

   hash_unstem = {}
   for word in dictionary:
      if word.count('~') == 0:
         key = stemmer.stem(word)
         # key = lemmatizer.lemmatize(word)
         hash_unstem = update_listHash(hash_unstem, key, word)

   hash_stem = {}
   for key in hash_unstem:
      list = hash_unstem[key]
      if len(list) > 1:
         # a few stems with 3+ words need breaking down [not done yet]
         # print(key, hash_unstem[key])
         max_cnt = 0
         for word in list:
            cnt = dictionary[word]
            if cnt > max_cnt:
               max_cnt = cnt
               lead_word = word
         for word in list:
            if word != lead_word:
               hash_stem[word] = lead_word

   hash_stem = dict(sorted(hash_stem.items()))
   return(hash_stem, hash_unstem)


#--- Read backend-tables

def get_data(filename, path):
   if 'http' in path:
      response = requests.get(path + filename)
      data = (response.text).replace('\r','').split("\n")
   else:
      file = open(filename, "r")
      data = [line.rstrip() for line in file.readlines()]
      file.close()
   return(data)


def read_table(filename, format = "float", path = ''):
   table = {}
```

```
    data = get_data(filename, path)
    for line in data:
       line = line.split('\t')
       if len(line) > 1:
         value = line[1]
         if format == 'str':
            table[line[0]] = line[1]
         else:
            table[line[0]] = eval(line[1])
    return(table)


def read_list(filename, path = ''):
    data = get_data(filename, path)
    stopwords = eval(data[0])
    return(stopwords)


def read_pairs(filename, path = ''):
    dictionary = {}
    data = get_data(filename, path)
    for line in data:
       line = line.split('\t')
       if len(line) > 1:
          if type(line[1]) is not float:
             dictionary[line[0]] = float(line[1])
          else:
             dictionary[line[0]] = line[1]
    return(dictionary)


#--- Hash functions

def update_listHash(hash, key, value):

    if key in hash:
       list = hash[key]
       if value not in list:
          list = (*list, value)
    else:
       list = (value,)
    hash[key] = list
    return(hash)


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)


#--- Build back-end tables

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) #
   mindex    = get_value('mindex', hash_crawl)

   ID_size = backendTables['ID_size']

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

   if flag != '':
      gword = flag + word
      update_nestedHash(backendTables['hash_ID'], gword, ID)

   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, ID) # used to be 'description'
   update_nestedHash(backendTables['hash_context5'], word, meta)
   update_nestedHash(backendTables['hash_ID'], word, ID)
   update_nestedHash(backendTables['hash_agents'], word, agents)
   # update_nestedHash(backendTables['full_content'], word, full_content) # takes space, don't
       nuild?

   if ID not in backendTables['ID_to_content']:
      for agent in agents:
         # new format: listHash; old format: nestedHash
         # update_nestedHash(backendTables['ID_to_agents'], ID, agent)
         update_listHash(backendTables['ID_to_agents'], ID, agent)
      update_hash(backendTables['ID_to_content'], ID, full_content)
      update_hash(backendTables['ID_to_index'], ID, mindex)
      update_nestedHash(backendTables['Index_to_IDs'], mindex, ID, ID_size[ID])

   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']
   create_hpairs = backendParams['create_hpairs']
   create_ctokens = backendParams['create_ctokens']
   use_stem = backendParams['use_stem']


   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 + "."
   text = text.replace('/'," ").replace('(',' ').replace(')',' ').replace('?','')
   text = text.replace("'","").replace('"',"").replace('\\n','').replace('!',' ')
   text = text.replace("\\s",'').replace("\\t",'').replace(","," ").replace(":"," ")
   text = text.replace(";"," ").replace("|"," ").replace("--"," ").replace("  "," ").lower()
   sentence_separators = ('.',)
   for sep in sentence_separators:
      text = text.replace(sep, '_~')
   text = text.split('_~')

   hash_pairs = backendTables['hash_pairs']
   ctokens = backendTables['ctokens']
   hash_stem = backendTables['hash_stem']
   stopwords = backendTables['stopwords']
   buffer2 = []

   for sentence in text:

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

      for word in words:

         if use_stem and word in hash_stem:
            word = hash_stem[word]

         if word not in stopwords:
            # word is single token
            buffer.append(word)
            buffer2.append(word)
            update_tables(backendTables, word, hash_crawl, backendParams)

            for k in range(1, max_multitoken):
               if offset > 0:
                  # word is now multi-token with k+1 tokens
                  word = buffer[offset-k] + "~" + word
                  buffer2.append(word)
```

```
                  update_tables(backendTables, word, hash_crawl, backendParams)

            offset += 1

    if create_hpairs:

        for k in range(len(buffer2)):

            wordA = buffer2[k]
            lbound = max(0, k-maxDist) # try bigger value for maxDist
            ubound = min(len(buffer2), k+maxDist+1)

            for l in range(lbound, ubound):
                wordB = buffer2[l]
                key = (wordA, wordB)
                if wordA < wordB:
                    hash_pairs = update_hash(hash_pairs, key)
                    if create_ctokens and k != l:
                        ctokens = update_hash(ctokens, key)

    return(backendTables)


def default_frontendParams():

    frontendParams = {
                    'distill': False,
                    'maxTokenCount': 1000, # ignore generic tokens if large enough
                    'beta': 1.0, # used in text entity relevancy score, try 0.5
                    'nresults': 20, # max number of chunks to show in results
                    'use_stem': False,
                   }
    return(frontendParams)


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)


def add_embedding(q_embeddings, word, embedding):
    for token in embedding:
        pmi = embedding[token]
        q_embeddings[(word, token)] = float(pmi)
    return(q_embeddings)
```

References

[1] Vincent Granville. *Building Disruptive AI & LLM Technology from Scratch*. MLTechniques.com, 2024. [Link]. 1, 2, 3, 9, 12, 14, 16, 23

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

Index