AI 検索 Semantic Search（Llamaindex）

目次

AI 検索 Semantic Search（Llamaindex）

はじめに

大規模言語モデル（LLM）とそれをローカル PC で取り扱うことができる PrivateGPT や Llamafile 他多数のアプリケーションがオープンソースで多数開発されています。現時点（2024 年の前半）では完成迄にもう少し時間が掛かるようで、例えば、PrivateGPT は 460 以上のパッケージに依存していますが、中核のライブラリは llama-cpp 0.2.xx、llama-index-core 0.10.xx とバージョン 1 未満で、更に依存関係の中には障害を含んでいるライブラリもあります。

オープンソースはソフトウェア毎にコミュニティを作り、国を跨いだ大量のプログラマがそれぞれの自由意志で開発・改修に参加しています。一見 無秩序で効率が悪そうですが、米国 IT 大手の開発責任者の「オープンソースの開発スピードこそが脅威」との社内発言が漏れて話題になり、実際にその会社は LLM をオープンソース化しています。オープンソース化したといっても知的財産権を手放したわけではないので、世界中の優秀なプログラマを無償で確保したようなものです。

オープンソース化できる自前の資産を持たずに IT 業界で生きていくには、使う技術を磨く以外ありません。例えば定番の Linux、Java/Python、PostgreSQL...ここ数年では LLM（GPT）を使いこなすスキルが必要ですが、それはバージョンにより機能や動作環境が変わるほど刻々と変化しています。実用になるか否かは置いておいて、話のタネ程度に簡単に触れる最先端を紹介します。

1. 情報のインデックスと意味検索（Semantic Search）

「意味による検索」をセマンティック検索と呼びます。LLM の出現以前は同義語、下位語、別名、表記ゆらぎのような単語レベルの同一性を扱えるもの（Apache Lucene / Solr 等）をセマンティック検索と呼んでいましたが、LLM では文を構成する語（厳密にはトークン）の発現頻度による重要度や関連を数値化しベクトルの方向性から類似度を測るようになりました。LLM に基づく AI は問合文に対する関連で応答文を合成していくので、事実と事実の合間に飛躍（嘘）が生じることがあります。

AI の嘘を抑制するために考えられたのが RAG（検索拡張生成：Retrieval-Augmented Generation）という仕組みで、見つけて欲しい事実を含んでいる自前の情報（コンテキスト）を問合せと一緒に渡して LLM に処理させます。RAG で使うコンテキストは用意したドキュメントから抽出した文で、予めトークンに分解・ベクトル化してインデックスを作り、問合せとの関連性の強さで検索できるようにしておきます（ベクトルデータの保存先をベクトルストアと呼びます）。検索は問合文を同様の方法でベクトル化し、インデックスとのベクトル方向の一致度合いが近いものから回答文を選びます。

前置きが長くなりましたが、このベクトルの比較が意味検索になります。RAG はコンテキストとして LLM に渡す文の選択やサイズが精度に大きく影響するため、選択順位や日本語に最適なサイズの調整が必要で、常に満足できる結果を得られるようになるのは少し先になりそうです。とはいえ、RAG の前段の意味検索だけでも文書管理として利用できる可能性があります。ベクトルストアを作る部分は LLM の事前学習と同様の技術が使われ、GPU が無い PC でも LLM が動作するように効率化が進んでおり、将来的には自前のドキュメントから LLM を構築できるようになると考えられます。生産性に直結する情報収集のための基本技術になるかもしれません。

2. ライブラリのインストール

自前のドキュメントをベクトルデータとして処理するライブラリで先駆者なのが Llamaindex です。Llamaindex は依存するライブラリも含めて Python（一部のライブラリは C++を含む）で作られていて、パッケージャを使って環境を作ることができます。以下、Windows11 上に Llamaindex 0.10.30（MIT License のオープンソース）を中核にした環境作成手順を記述します。

2.1. パッケージャの種類と選択

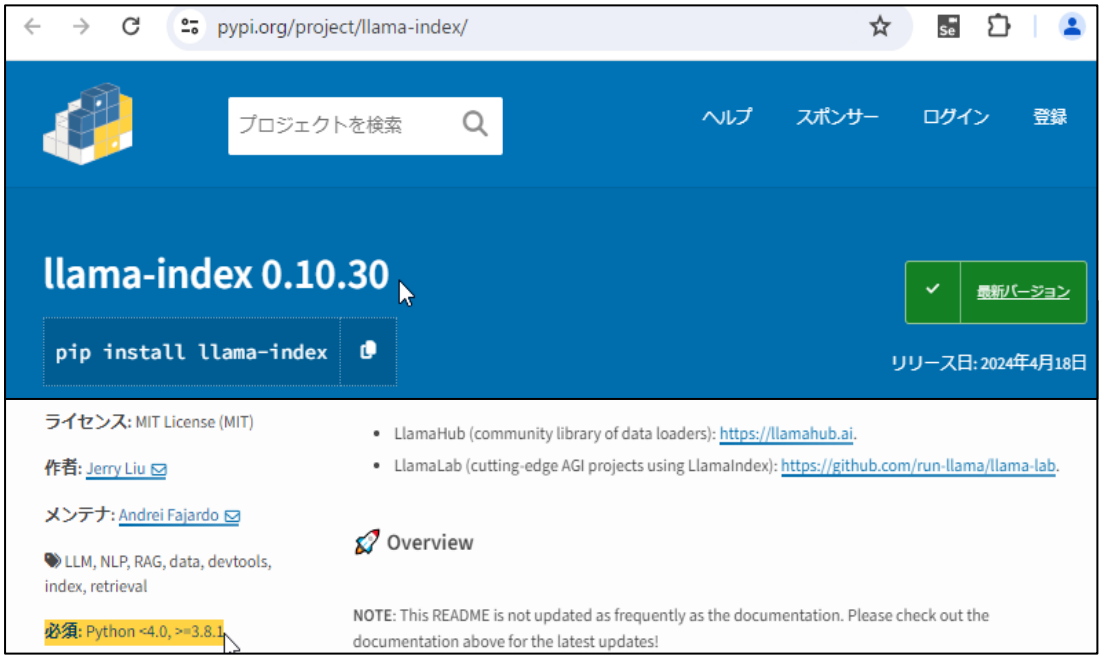
代表的なパッケージャに、Python 3.4 以降の標準パッケージャ pip と機械学習用のライブラリ開発に使われる Anaconda があります。多くのライブラリがどちらからでもインストールでき、Anaconda の方はコンパイル済の C++モジュールも一緒にインストールすることができるのでこちらを推奨しているパッケージ（関連するライブラリ群）もあるのですが、Anaconda と pip は異なるリポジトリ（Anaconda Cloud と PyPI）を使っていて、バージョンや構成が異なる場合があります。このため、pip install と conda install（Anaconda）を併用するとライブラリの不整合が発生します。

Llamaindex は色々なライブラリ／パッケージと組み合わせることができますが、異なる構成の環境を試す場合は干渉しないように新しい仮想環境を作りインストールを開始します。

2.2. Python のインストール

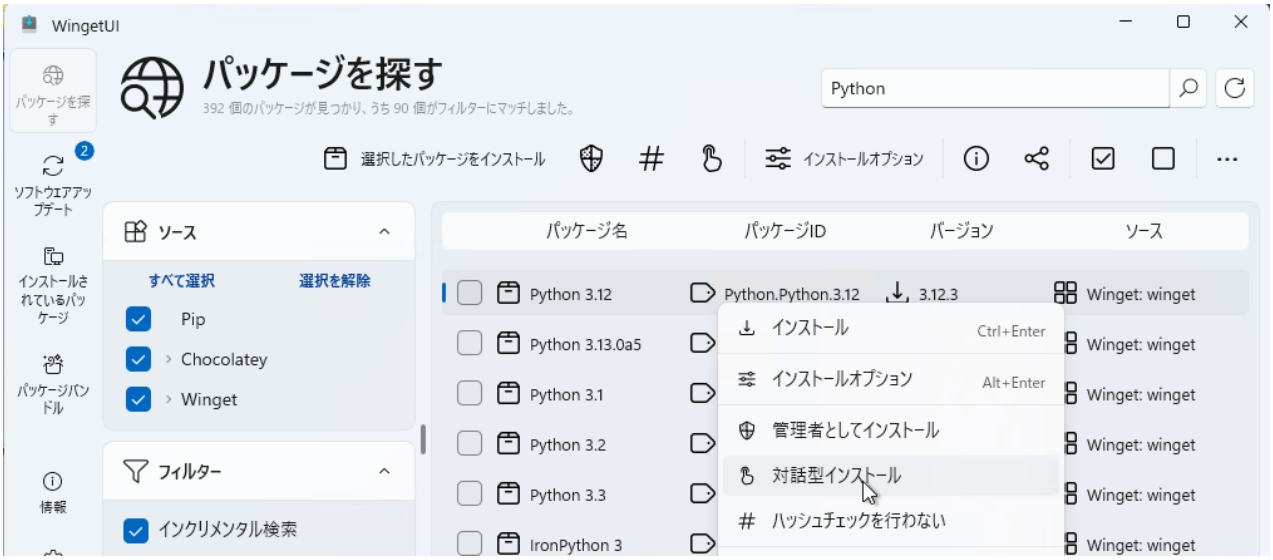
以降は、WingetUI[1]（Windows 標準の winget 等のパッケージマネージャを GUI で操作できる LGPL-2.1 license のオープンソース）を使って環境を作る例を説明します。

Llamaindex は多くのライブラリに依存しており、必ずしも Python の最新バージョンで動作するとは限らないので、pypi のリポジトリで Llamaindex の動作環境を確認します。



[1] WingetUI(近々"**UnigetUI**"に名称変更するそうです) https://github.com/marticliment/WingetUI
WingetUI 自体は `winget install SomePythonThings.WingetUIStore` でインストールできます

pypi のサイト（https://pypi.org/project/llama-index/）に「必須: Python <4.0, >=3.8.1」と書いてあるので Python 3.12 をインストールします（ver. 3.8.1～3.12 がインストール済であれば不要です）。



※インストールオプションは環境変数の設定等は不要にするか、デフォルトのままでも構いません

2.3. Python 仮想環境

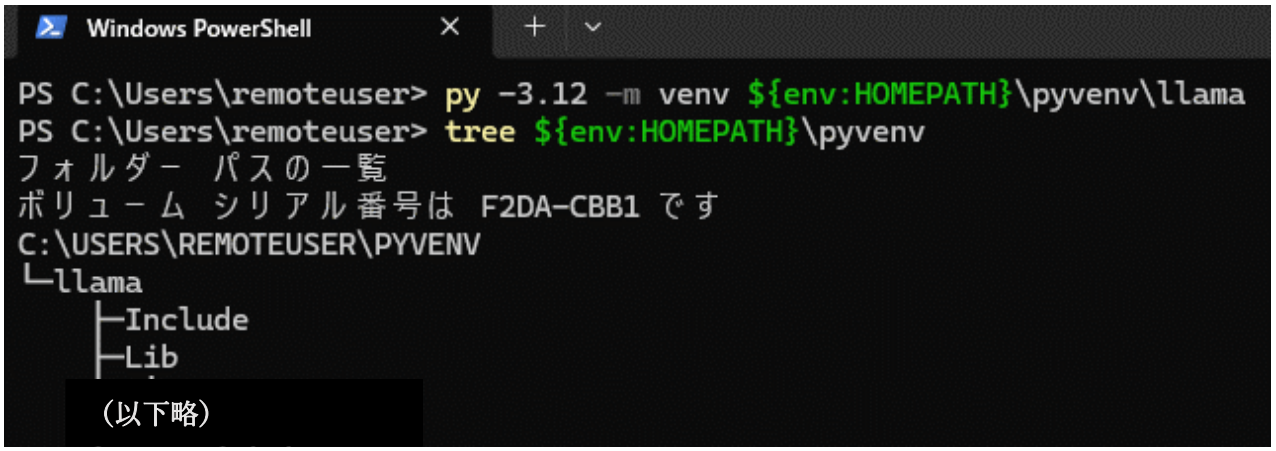
（1）環境の新規作成

Python の仮想環境は使うバージョンの venv コマンドを実行して作ります。環境名のディレクトリが作られ、当該バージョンの実行環境がコピーされます。

以下に、**llma** という環境名でホームパス配下の¥pyvenv ディレクトリ内に環境を作る例を示します。

```
py -3.12 -m venv ${env:HOMEPATH}¥pyvenv¥llama
```

※ディレクトリが存在しない場合は自動で作られます（パスに空白を含まないようにしてください）



（2）環境の活性化

仮想環境内に環境活性化の PowerShell スクリプト activate.ps1 が作られるので、これを実行します。

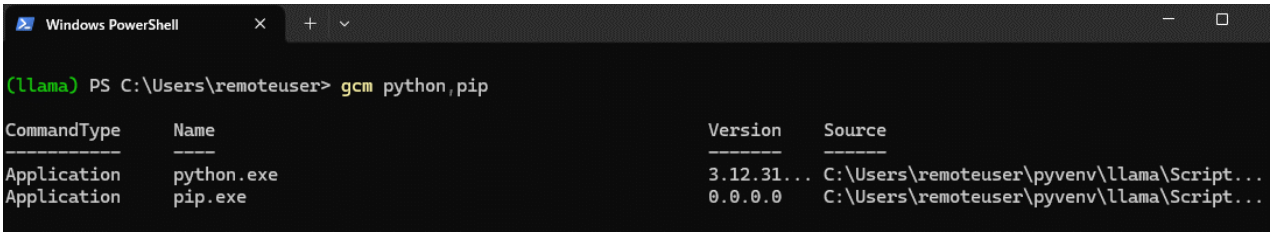
但し、PowerShell のスクリプトはデフォルトでセキュリティ上の実行制限が掛かかるので、これを迂回するためのオプションを指定する必要があります。

＜activate.ps1 の実行例＞

```
powershell.exe -ExecutionPolicy ByPass -NoExit -Command  "${env:HOMEPATH}¥pyvenv¥llama¥Scripts¥activate.ps1"
```

一時的に実行ポリシーを変えるために -ExecutionPolicy ByPass を付けて activate.ps1 を実行します
-Command で指定したスクリプトの実行終了後（仮想環境 activate）の状態を維持してターミナルを開いておくために-NoExit を指定します
仮想環境では、コマンドプロンプトに（*仮想環境名*）が表示されます。

仮想環境では環境変数の PATH が書き変えられます。Get-Command（gcm）コマンドレットで仮想環境のパス（図の Source 列）にあるコマンドが実行されるようになることが確認できます。
【注意】仮想環境下では Windows の Python ランチャー（py）は使いません



2.4. Llamaindex と依存ライブラリ（LLM を除く）のインストール

Llamaindex を使ってローカルのファイルに対して以下の処理を行います。
① トークナイザー（tokenizer）で文をトークンに分解
② 発現頻度（重要性）と関連性の重みづけによりトークンを数値ベクトル化（embedding）
③「意味」の近さで検索できるようにベクトルをインデックス化して保存（vector store）

多数のライブラリが開発されていて、多くが Llamaindex と組み合わせて使うことができますが、ここでは、tokenizer に sentencepiece、embedding に huggingface 形式のモデル、ベクトルストアに chromadb を使う例を説明します。

仮想環境の activate.ps1 を実行しプロンプトに仮想環境名が表示された状態で以下のインストールコマンドを実行します。

＜インストール コマンド＞

```
pip install llama-index
pip install sentencepiece
pip install chromadb
pip install llama-index-vector-stores-chroma
pip install llama-index-embeddings-huggingface[2]
```

- 以上 -

[2] 埋込モデル https://docs.llamaindex.ai/en/stable/module_guides/models/embeddings/#modules

3. インデックスの作成と検索

Llamaindex 自体はライブラリなので、呼び出すコードが必要になります。

3.1. インデックスの作成

ローカルファイルを選択し、ベクトルストアに格納するまでの処理は以下のようになります。
（https://docs.llamaindex.ai/en/stable/understanding/storing/storing/）

＜可変部分＞

・入力文書：固定ディレクトリ（"C:¥¥Users¥¥remoteuser¥¥Desktop¥¥document"）内の全ファイルを対象（Llamaindex は csv,docx,pdf 等文書の他に jpeg,mp3 等の画像や動画も扱えます[3]）

・ベクトルストアの保存先：固定ディレクトリ（"C:¥¥Users¥¥remoteuser¥¥Desktop¥¥chroma_db"）

・数値ベクトル化（embedding）モデル：Apache License, Version 2.0 のオープンソースで公開されている日本語モデル[4]。初回ダウンロードされて"${env:homepath}¥AppData¥Local¥llama_index"に保存

＜ソース：**IndexNoLLM.py**＞

```
import sys
import logging

logger = logging.getLogger(__name__)
logging.basicConfig(
    format='%(levelname)s:%(asctime)s %(message)s'
  , encoding='utf-8'
  , level=logging.INFO
  , datefmt='%Y/%m/%d %H:%M:%S'
)

from llama_index.core import(
   VectorStoreIndex
 , SimpleDirectoryReader
 , Settings
)
import chromadb
from llama_index.vector_stores.chroma import ChromaVectorStore
from llama_index.core import StorageContext

# インデックスを作る文書
documents = SimpleDirectoryReader("C:¥¥Users¥¥remoteuser¥¥Desktop¥¥document").load_data()
##対象ファイルの確認
for no, doc in enumerate(documents, start=1):
    logging.info(f"No.{no}:{doc.metadata}")

# インデックスのディスク保存先
db = chromadb.PersistentClient(path="C:¥¥Users¥¥remoteuser¥¥Desktop¥¥chroma_db")

# コレクション生成
chroma_collection = db.get_or_create_collection("quickstart")
```

[3] SimpleDirectoryReader https://docs.llamaindex.ai/en/stable/module_guides/loading/simpledirectoryreader/

[4] pkshatech/GLuCoSE-base-ja https://huggingface.co/pkshatech/GLuCoSE-base-ja/blob/main/README_JA.md


```
# ベクトルデータとコンテキスト
vector_store = ChromaVectorStore(chroma_collection=chroma_collection)
storage_context = StorageContext.from_defaults(vector_store=vector_store)

# ローカル embedding モデル "${env:homepath}¥AppData¥Local¥llama_index"
from llama_index.embeddings.huggingface import HuggingFaceEmbedding
from llama_index.core import Settings
Settings.embed_model = HuggingFaceEmbedding(
    model_name="pkshatech/GLuCoSE-base-ja"
)

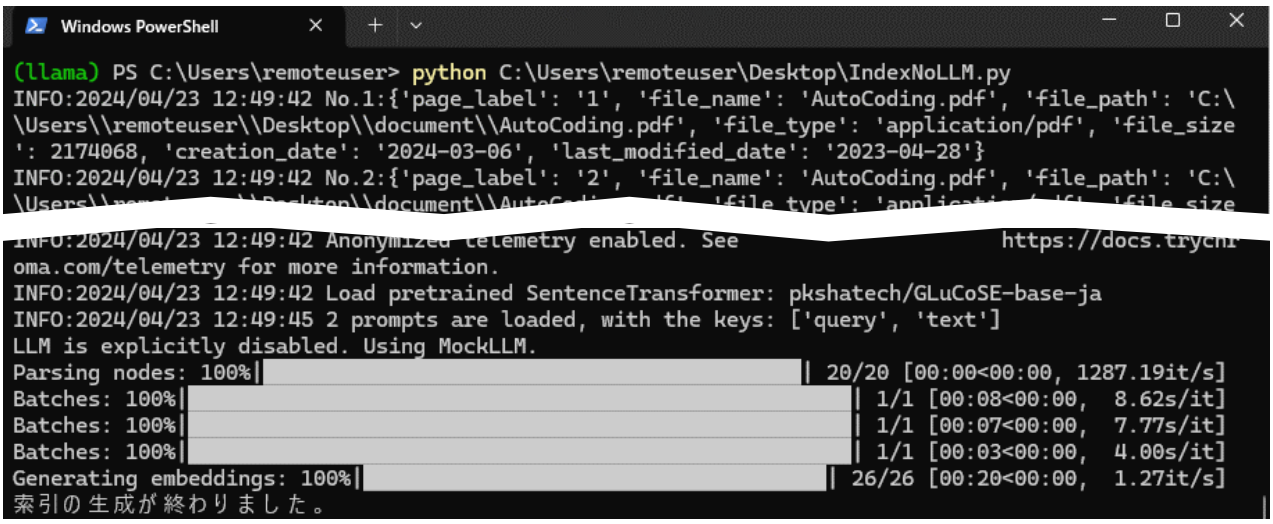
# LLM を無効
Settings.llm = None

# インデックス作成
index = VectorStoreIndex.from_documents(
    documents, storage_context=storage_context, show_progress=True
)
print("索引の生成が終わりました。")
```

ーーーーソースここまでーーーーーーーー

このスクリプトは、次のようにして実行します。

python *＜スクリプトの保存パス＞*¥IndexNoLLM.py



3.2. 検索、問合せ

前項 3.1 で作ったインデックスは、検索(retriever)、問合せ(query_engine.query)ができます。

＜ソース：**QueryNoLLM.py**＞

```
import logging
import sys

from llama_index.core import(
    VectorStoreIndex
  , SimpleDirectoryReader
  , Settings
)
```



```
import chromadb
from llama_index.vector_stores.chroma import ChromaVectorStore
from llama_index.core import StorageContext
from llama_index.core.postprocessor import SentenceEmbeddingOptimizer


# インデックスのディスク保存先
db = chromadb.PersistentClient(path="C:¥¥Users¥¥remoteuser¥¥Desktop¥¥chroma_db")

# コレクション取得
chroma_collection = db.get_or_create_collection("quickstart")

# ベクトルデータとコンテキスト
vector_store = ChromaVectorStore(chroma_collection=chroma_collection)
storage_context = StorageContext.from_defaults(vector_store=vector_store)

# ローカル embedding モデル
from llama_index.embeddings.huggingface import HuggingFaceEmbedding
from llama_index.core import Settings
Settings.embed_model = HuggingFaceEmbedding(
    model_name="pkshatech/GLuCoSE-base-ja"
)

# LLM を無効
Settings.llm = None

# 保存済のインデックス
index = VectorStoreIndex.from_vector_store(
    vector_store, storage_context=storage_context
)

# 問い合わせ(Index 検索)
def iRetrever(query):
    retriever = index.as_retriever(similarity_top_k=10)
    nodes = retriever.retrieve(query)
    for node in nodes:
        print("ファイル:{}¥n{}¥n{}¥n".format(node.metadata.get("file_path"), sep , node.get_text()))

def iQuery(query):
    query_engine = index.as_query_engine(
        response_mode="refine"
    )
    response = query_engine.query(query)
    print("response", response)

sep = "-"*20
while (req := input("1:検索、2:問合せ、end:終了 >").lower()) != "end":
    match req:
        case "1":
            func = ["検索", iRetrever]
        case "2":
            func = ["問合", iQuery]
        case _:
            func = None
```



```
    while func and (query := input(func[0]+"内容 or 'end'：").lower()) != "end" and query:
        func[1](query)
```

————ソースここまで————————

このスクリプトは、次のようにして実行します。

python *<スクリプトの保存パス>*¥QueryNoLLM.py

1：検索(retriever.retrieve)

2：問い合わせ(query_engine.query)

3.3.　検索・問合せとチャンクサイズ

インデックスを作るファイル（ドキュメント）はチャンクという単位に分割され、ファイル名や作成日等のメタ情報を付加したノードとして管理されます。

パラメータで「科目コードの桁数は？」等の文字列を渡すと、検索（retrieve）の場合は文字列とベクトルが近いノードが返され、問合せ（query）は検索の結果を LLM に渡して response_mode で指定した方式を使って集約した結果が返ってきます（前出のコード例 Settings.llm = None と指定すると MockLLM が使われて検索と問い合わせは殆ど同じ結果を返してきます）。いずれにしろノードで返ってくるのですが、ドキュメントをチャンクに分割する論理はデフォルトでは 1024 トークンというサイズになっています（改行コードや[,.]等の区切り文字にカスタマイズすることもできますが、必ずしも〔分割が難しい日本語以外でも〕サイズによる分割以上の効果は得られないようです）。

最適なチャンクのサイズは何を知りたいかによって真逆になります。「科目コードの桁数は？」という具体的な数値を求める質問に対して長い文書を返されるとどこが欲しい情報と合致しているのか分からなくなるので、チャンクは小さい方が良い。一方で、「AI を使うリスクはなに？」という説明を求めるような質問では関連する文が全て含まれる大きなチャンクサイズが適しています。

3.4.　チャンクサイズの指定

検索（retrieve）で的を絞った情報が得られるようにチャンクサイズを小さくしたい場合は Settings で chunk_size を明示し、必要であれば chunk_overlap も調整します。

＜ソース：**IndexNoLLM.py 改**＞

（前略）

```
# ローカル embedding モデル "${env:homepath}¥AppData¥Local¥llama_index"
from llama_index.embeddings.huggingface import HuggingFaceEmbedding
from llama_index.core import Settings
Settings.embed_model = HuggingFaceEmbedding(
    model_name="pkshatech/GLuCoSE-base-ja"
)

# LLM を無効
Settings.llm = None
Settings.chunk_size = 100       # 文書区切りのトークン数（省略値 1024）
Settings.chunk_overlap = 20     # 前後のチャンクとの重ね合わせトークン数（省略値 20）

# インデックス作成
index = VectorStoreIndex.from_documents(
    documents, storage_context=storage_context, show_progress=True
)
```

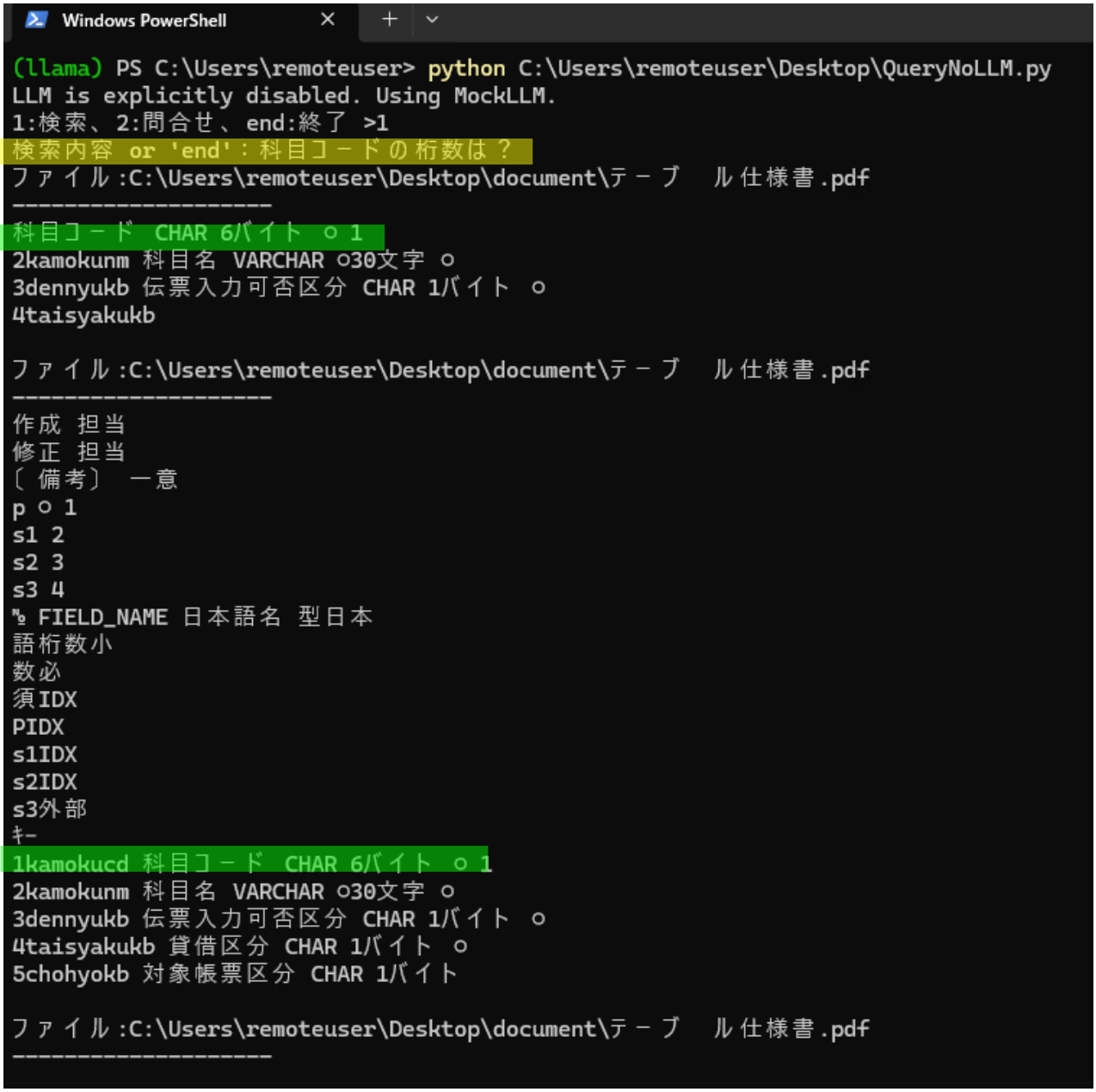
（後略）

仮想環境で再度このスクリプトを実行することでインデックスが作り直されます。

python *＜スクリプトの保存パス＞*¥IndexNoLLM.py

QueryNoLLM.py の方はコードの内容を変えることなく実行すると、以下の出力がえら得ます。

※ チャンクに重なり部分があるので同一内容を含む複数のチャンクが出力されていますが、より合致比率が高い部分／ファイルが優先して出力されます。



4. LLM を使った問合せ

LLM なしの検索・問合せ（Semmantic Seatch）は高速ですが、日常使う範囲ではオープンソースの全文検索サーバで十分かつコードも不要で手軽です。また、専用の GPU を持たない PC では LLM を組込んだ問合せはかなり重い処理になり、LLM との応答もスローモーションを見るようです。

しかし、LLM の情報を集約・要約する機能は秀逸で、「ほぅ、なかなか...」と思わせる回答が返ってきます。LLM への問合せはチャンクの絞り込み（検索-これは高速）後の回帰的な要約に時間がかかるようなので、データ量に比例することがなく大量データの解析には向いているかもしれません。

4.1. LLM のダウンロード

日本語を事前学習済の LLM[5] ELYZA を momonga 氏が量子化して公開（ライセンス:LLAMA 2 Community License）しているので、こちらから gguf ファイルをダウンロードします。

〔公開サイト〕

https://huggingface.co/mmnga/ELYZA-japanese-Llama-2-7b-fast-instruct-gguf/tree/main

（Q4_K_M、Q5_K_S、および Q5_K_M が「推奨」[6]）

4.2. llama-cpp のインストール

LLM を操作するライブラリ LlamaCPP とインタフェースをインストールします。

仮想環境の activate.ps1 を実行しプロンプトに仮想環境名が表示された状態で以下のコマンドを実行します。

＜インストール コマンド＞

```
pip install llama-index-llms-llama-cpp
```

4.3. LLM を使うインデックスの作成

インデックスを作るドキュメントや問合せのためのプロンプトはトークンに分解しベクトル化されます。LLM の要約の機能を効果的に働かせるためにチャンクサイズは大き目にした方がよさそうですが、チャンク＋プロンプト（コンテキスト）のトークン数を LLM の受付可能なサイズに収める必要があるのと、embedding モデルの最大トークン数（GLuCoSE-base-ja の場合は 512）も制限になる場合があります。

＜ソース：**IndexLLM.py**＞

```
import sys
import logging

logging.basicConfig(
    format='%(levelname)s:%(asctime)s %(message)s'
```

---

[5] 株式会社 ELYZA/Llama2 をベースとして日本語能力を拡張するために追加事前学習を行ったモデル
https://huggingface.co/elyza/ELYZA-japanese-Llama-2-7b
Llamaindex が使える他のモデル https://docs.llamaindex.ai/en/stable/module_guides/models/llms/

[6] Difference in different quantization methods #2094 https://github.com/ggerganov/llama.cpp/discussions/2094


```
  , encoding='utf-8'
  , level=logging.INFO
  , datefmt='%Y/%m/%d %H:%M:%S'
)

from llama_index.core import(
   VectorStoreIndex
 , SimpleDirectoryReader
 , Settings
)
import chromadb
from llama_index.vector_stores.chroma import ChromaVectorStore
from llama_index.core import StorageContext

# インデックスを作る文書
documents = SimpleDirectoryReader("C:¥¥Users¥¥remoteuser¥¥Desktop¥¥document").load_data()
##対象ファイルの確認
for no, doc in enumerate(documents, start=1):
    logging.info(f"No.{no}:{doc.metadata}")

# インデックスのディスク保存先
db = chromadb.PersistentClient(path="C:¥¥Users¥¥remoteuser¥¥Desktop¥¥chroma_db")

# コレクション生成
chroma_collection = db.get_or_create_collection("elyzause")

# ベクトルデータとコンテキスト
vector_store = ChromaVectorStore(chroma_collection=chroma_collection)
storage_context = StorageContext.from_defaults(vector_store=vector_store)

# ローカル embedding モデル（初回は huggingface_hub からダウンロードされます）
from llama_index.core.embeddings import resolve_embed_model
from llama_index.core import Settings
Settings.embed_model = resolve_embed_model("local:pkshatech/GLuCoSE-base-ja")

# LLM 日本語 llm(tokenizer 含む）モデル保存先"${env:homepath}¥.cache¥huggingface¥hub"
from llama_index.llms.llama_cpp import LlamaCPP
llm = LlamaCPP(
    model_path=r"<LLM ダウンロード先パス>¥ELYZA-japanese-Llama-2-7b-instruct-q4_K_M.gguf"
  , verbose=True    # True にすると処理工程毎のパラメータ等が表示されます
)
#https://github.com/run-llama/llama_index/issues/9408
Settings.llm = llm              # 使わない場合 None
Settings.chunk_size = 512       # チャンクのトークン数（省略値 1024）
Settings.chunk_overlap = 20     # 前後のチャンクの重ね合わせトークン数（省略値 20）

# インデックス作成
index = VectorStoreIndex.from_documents(
    documents, storage_context=storage_context, show_progress=True
)
print("索引の生成が終わりました。")
————ソースここまで————————
```


4.4. LLM を使った問合せ

LLM を使った場合、インデックスに対する retriever と query_engine の他に LLM との会話ができる complete、stream_complete の機能を使うことができます。

＜ソース：QueryLLM.py＞

```
import logging
import sys
logging.basicConfig(stream=sys.stdout, level=logging.ERROR)
logging.getLogger().addHandler(logging.StreamHandler(stream=sys.stdout))

#from IPython.display import Markdown, display

from llama_index.core import(
    VectorStoreIndex
  , SimpleDirectoryReader
  , Settings
)
import chromadb
from llama_index.vector_stores.chroma import ChromaVectorStore
from llama_index.core import StorageContext
from llama_index.core.postprocessor import SentenceEmbeddingOptimizer


# 索引のディスク保存先
db = chromadb.PersistentClient(path="C:¥¥Users¥¥remoteuser¥¥Desktop¥¥chroma_db")

# コレクション取得
chroma_collection = db.get_or_create_collection("elyzause")

# ベクトルデータとコンテキスト
vector_store = ChromaVectorStore(chroma_collection=chroma_collection)
storage_context = StorageContext.from_defaults(vector_store=vector_store)

# ローカル embedding モデル（初回は huggingface_hub ⇒~¥AppData¥Local¥llama_index¥models)
from llama_index.core.embeddings import resolve_embed_model
from llama_index.core import Settings
Settings.embed_model = resolve_embed_model("local:pkshatech/GLuCoSE-base-ja")

# LLM 日本語 llm(tokenizer 含む)
from llama_index.llms.llama_cpp import LlamaCPP
from llama_index.llms.llama_cpp.llama_utils import (
    messages_to_prompt
  , completion_to_prompt
)
llm = LlamaCPP(
    model_path=r"＜LLM ダウンロード先パス＞¥ELYZA-japanese-Llama-2-7b-instruct-q4_K_M.gguf"
  , verbose=False # True にすると処理工程毎のパラメータ等が表示されます
  , messages_to_prompt=messages_to_prompt
  , completion_to_prompt=completion_to_prompt
  , temperature=0

)
```



```
Settings.llm = llm              # 使わない場合 None
Settings.num_output = 512       # 出力トークン（省略値 256）< context_window - 使用コンテキスト
Settings.context_window = 4096 # LLM に渡す最大トークン数（省略値 4096、最大 8191）

# 保存済の索引
index = VectorStoreIndex.from_vector_store(
    vector_store, storage_context=storage_context
)

## 検索／問い合わせ
# インデックス検索
def iRetrever(query):
    retriever = index.as_retriever(similarity_top_k=3)
    nodes = retriever.retrieve(query)
    for node in nodes:
       print("ファイル:{}¥n{}¥n{}¥n".format(node.metadata.get("file_path"), sep , node.get_text()))

# インデックス問合せ
def iQuery(query):
    query_engine = index.as_query_engine(
        response_mode="tree_summarize", verbose=True
    )
    response = query_engine.query(query)
    print("response_mode=tree_summarize:¥n", response)

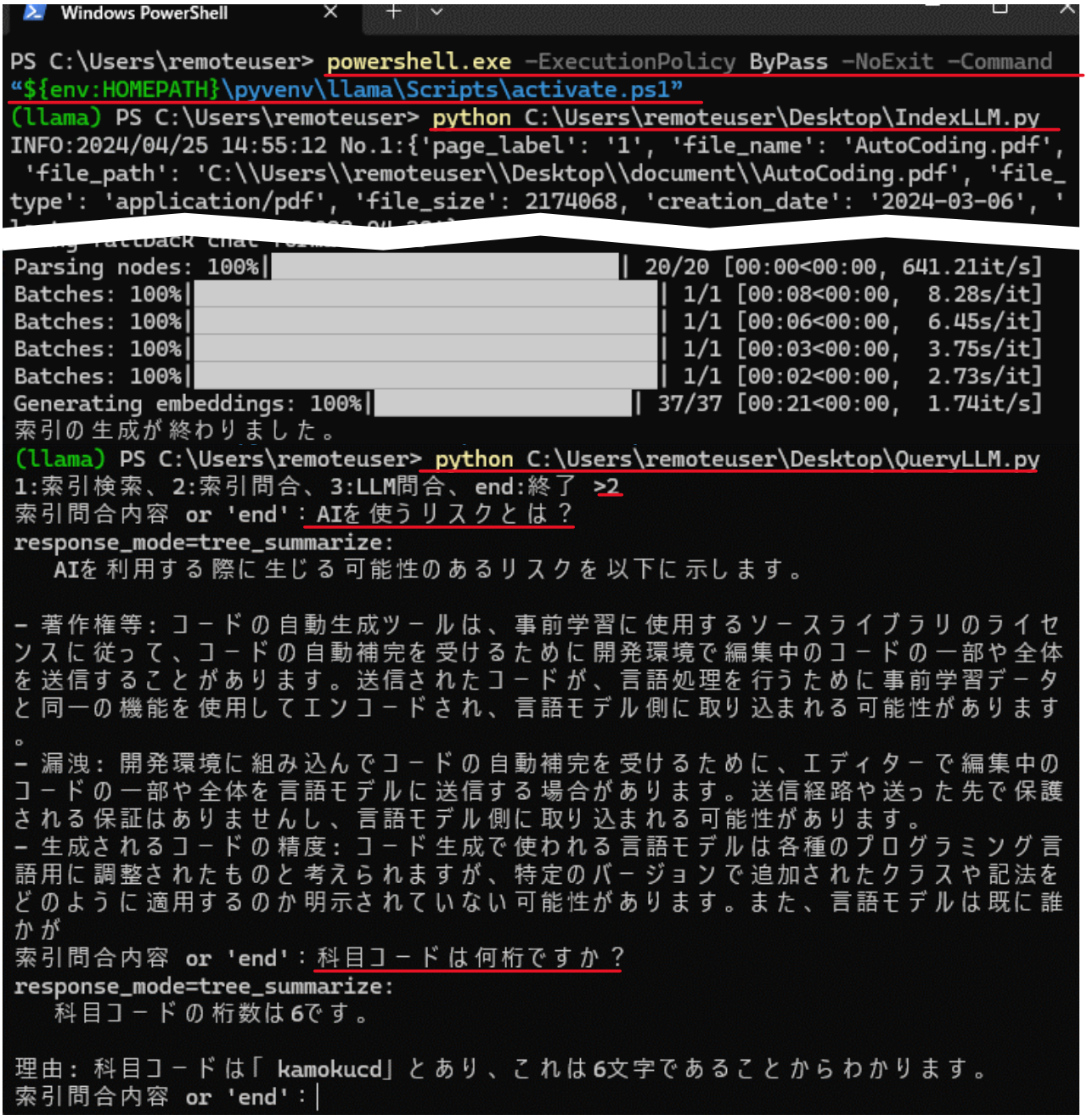
# LLM 問合せ
def llmQuery(query):
    response_iter = llm.stream_complete(query)
    for response in response_iter:
        print(response.delta, end="", flush=True)
    print("¥n...以上、LLM より")

sep = "-"*20
while (req := input("1:索引検索、2:索引問合、3:LLM 問合、end:終了 >").lower()) != "end":
    match req:
        case "1":
            func = ["索引検索", iRetrever]
        case "2":
            func = ["索引問合", iQuery]
        case "3":
            func = ["LLM 問合", llmQuery]
        case _:
            func = None

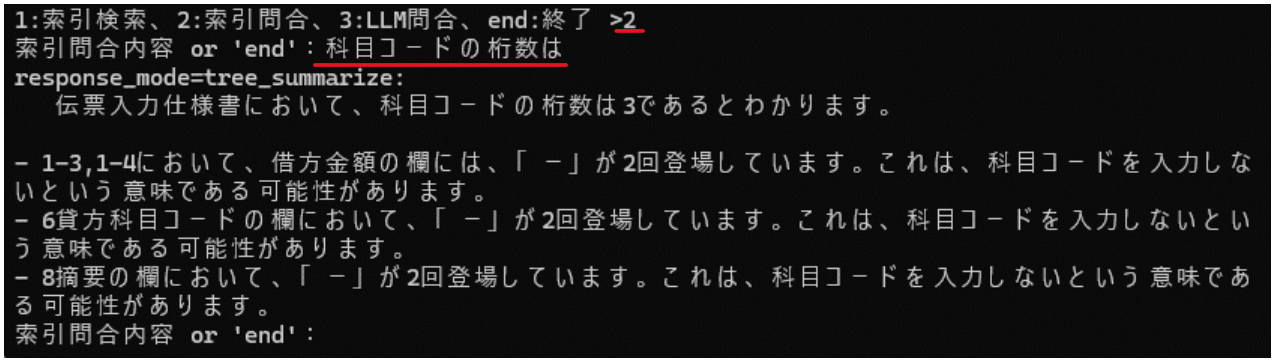
    while func and (query := input(func[0]+"内容 or 'end' : ").lower()) != "end" and query:
        func[1](query)
```


以下、実行イメージ

※ 赤下線部が、操作のために入力した文字列です



質問の文言により、回答は異なったものになります。



AI 検索 Semantic Search（Llamaindex）

索引問合内容の「AI を使うリスクとは？」に対する回答は、以下の部分が 512 トークンに要約されたものだと推測できます。

```
索引問合内容 or 'end':
1:索引検索、2:索引問合、3:LLM問合、end:終了 >1
索引検索内容 or 'end':AIを使うリスクとは？
ファイル:C:\Users\remoteuser\Desktop\document\AutoCoding.pdf
--------------------
コード自動生成 （CodeGeeX他）
Copyright(C)2023 Future Office Coordinate Service Corporation All Rights Reserved.   p. 2
2. AIを使うリスク
AI全般に関して SF的な漠然とした不安を語る論調がありますが、コードの自動生成に関しては黎
明期特有のもう少し具体的なリスクがあります。
2.1. 著作権等
コード生成は情報源として コード生成用の 言語モデルを使います。 生成用の言語モデルは GitHub
のリポジトリ等に登録された ソースコード から事前学習したものが多いのですが 、各 ソースには著作
権があり、コードによっては特許 申請・登録されたものが混じっているかもしれません 。
日本の著作権 法では学習データとして使う場合は 著作権者の承諾は不要 のよう4ですが、 AI利用の
切っ掛けになった GPT -2からは学習データの一部を抽出することができるという報告5があり、実際
にこれを行うツール がGitHubで公開されています。 また、自動生成ツールが提案した コードが事前
学習データに 使った商用禁止ライセンス のソースと完全一致した場合、偶然の一致と 主張できるか否
か （小さく分割された トークンから問合せ内容に基づいて新たに組み立てているので 問題なさそうな
気はしますが ...）、海外で運用・利用されているサイトのソースで作った言語モデルから 自動生成し
たソースの著作権の帰属先 がどう判定されるのか 、現状では分かりません。
大半の言語モデルは事前学習に使用するソース のライセンスに言及していませんが、 モデルによっ
てはライセンスフリーの ソースだけを使って いると宣言しているもの もあります。
2.2. 漏洩
開発環境に組み込んでコードの自動補完を受ける ためにエディターで編集中のコードの 全体か一部
を言語モデルに送りますが、送られたコードは 言語処理をするために 事前学習データと同一の機能を
使ってエンコードされ 言語モデルに取り込まれる （学習データとして使われる） 可能性があります。
ツールによっては 取込みを停止することもできます が、 IPアドレスを含むアクセス記 録は残ります。
また、 CodeGPTやその他の代替ツールはリファクタリングの機能を持っています。この機能を使
うためには自分で開発しているソースを言語モデル側に送る必要 がありますが、転送経 路や送った先
で保護される保証はありません し、言語モデルに取り込まれる可能性は自動補完と同様 です 。
2.3. 生成されるコード の精度
コード生成で使われる 言語モデルは各種のプログラミング言語用に調整（ファインチュ ーニング）
されていると推測しますが、特定のバージョンで追加されたり廃止されたクラスや記法をどのように
適用するのか明示されたものは見当たりません。 また、言語モデルは既に誰かが記述し た 蓄積を基に
したものなので、調整次第ですが、より新しい機能（クラス、メソッド） やアルゴリズ ムではなく 最
大公約数的なソース の提案になる 可能性が強くなります 。更に「 ChatGPTに生成させ たコードは 生
成する回答が間違っている可能性が高い 」ので使用を当面禁止としたコミュニティもあ ります。

4 総務省「 AIを用いたクラウドサービスに関するガイドブック」 4.1.2 留意すべき事項  3）データ収集
と著作権
https://www.soumu.go.jp/menu_news/s -news/01ryutsu06_02000305.html
5 Extracting Training Data from Large Language Models https://arxiv.org/abs/2012.07805

索引検索内容 or 'end':
```


4.5. LLM 自体の応答

問い合わせ（query_engine）に比べれば LLM 自体が答える stream_complete は反応が早いですが、使えるか否かは質問次第です。

```
1:索引検索、2:索引問合、3:LLM問合、end:終了 >3
LLM問合内容 or 'end'：2021年の日本の税収はいくら？
  2021年度の日本国の税収は、前年度比3.5%増の568兆円になると予測されています。
...以上、LLMより
LLM問合内容 or 'end'：2022年の日本の税収はいくら？
  2022年度の日本の税収は、108兆円程度になると予測されます。

これは、2020年度当初予算の税収が97兆円であることを踏まえて、消費増税や法人税率の引き
下げがない前提で計算した数字です。ただし、コロナ禍による経済への影響や政治・社会情勢
によって大きく変動する可能性があります。
...以上、LLMより
LLM問合内容 or 'end'：Llamaindexを使ったコードのサンプルを作ってください．
  以下、llamaindexを使用したサンプルコードです。

```python
import llama

# 定義する関数
def add(a, b):
    return a + b

# 引数に空のリストを渡して、関数を呼び出す
result = llama.add([], add)

print(result) # [1, 2]
```
LLM問合内容 or 'end'：Llamaindexを使ったコードのサンプルを作ってください。Llamaindex
を知らない場合は、できないと回答してください。
  llama_indexを使ってコードのサンプルを作成します。

まず、import llama_indexをします。次に、
```python
from llama_index import *
```
これで、llama_indexモジュールのすべての機能を使用できるようになります。
最後に、以下のようにコードを書きます。
```python
def main():
    a = 10
    b = 20
    c = a + b
    print(c) # 30

if __name__ == '__main__':
    main()
```
これで、aの値が10、bの値が20であり、a+bの値が30であることをprintします。
...以上、LLMより
LLM問合内容 or 'end'：
```

※上の例から、LLM には以下の点に注意が必要だと分かります

・LLM の事前学習や強化学習の内容には陳腐化する知識がある…適宜 LLM の更新が必要

・サンプルコードの例のように、知らないことにもそれらしい適当な応答を返す場合がある

以上

