

□ 연구 개요

○ 배경

- 신약개발 과정에서 ADMET(Absorption, Distribution, Metabolism, Excretion, Toxicity) 특성은 후보물질의 약효와 안전성을 결정짓는 핵심 요인임
- 그러나 실제 실험 기반의 ADMET 평가에는 많은 비용과 시간이 소요되며, 화학적 구조·생물학적 컨텍스트·실험 환경 등 다양한 요인을 통합적으로 해석하기 어려움 존재.
- 이에 따라 데이터 기반 ADMET 예측이 활발히 연구되고 있으며, 특히 그래프 신경망(GNN)을 활용한 분자 구조 표현과, BioBERT·LLM을 이용한 문헌 임베딩을 결합하는 방식이 새로운 대안으로 제시.
- 이러한 접근은 구조적·언어적 정보를 함께 고려함으로써 약물의 흡수·분포·대사·배설·독성 등 다중 생리학적 특성을 동시에 추론할 수 있는 가능성을 마련.
- 최근에는 단일 엔드포인트(예: 독성, 대사 안정성) 예측을 넘어 다중 엔드포인트(Multi-Task) ADMET 모델을 구축해 파라미터 공유를 통한 성능 향상과 일반화 능력을 확보하려는 시도가 증가

○ 관련 연구

- Wang, X. et al. (2023). 멀티태스크 GNN로 여러 ADMET 엔드포인트 동시 예측
- Cremer et al. (2023). 3D 구조 기반 GNN 적용 언어정보·문헌지식 통합 부족
- Goh et al. (2024) SMILES fragment와 언어모델 결합 구조·언어 통합 방식이 아직 초기 단계

○ 기존 연구의 한계점 및 개선 사항

- ADMET 관련 데이터는 실험, 문헌, Organoid 소스, 이미지·스펙트럼 등 다양한 형태로 분산되어 있으며 이를 하나의 스키마로 통합할 표준화된 데이터 파이프라인이 부재
- 각 데이터 간 식별자, 버전, provenance 관리가 미흡해 확장성과 재현성이 낮음
- 기존 GNN 기반 모델은 구조적 정보에는 강하지만, 생물학적 컨텍스트나 문헌 지식을 반영이 어려우며 LLM/BioBERT 기반 모델은 언어적 지식은 풍부하나, 실제 화학 구조적 특징을 반영하지 못해 예측 정확도가 낮음

- 흡수(A), 분포(D), 대사(M), 배설(E), 독성(T) 모델을 각각 독립적으로 학습함으로써 계산 자원이 낭비되고, 서로 연관된 생리학적 특성을 공유 제한
- 엔드포인트 간 공통 표현을 활용한 멀티태스크(head sharing) 접근 부족

○ 연구 목표

- 본 연구는 기존의 단일 모델 단일 데이터 한계를 극복하고 화학구조, 문헌, 오가노이드, 멀티모달 데이터를 통합하는 데이터 스키마를 설계하며 GNN과 LLM을 융합한 멀티테스트 ADMET 예측 베이스라인 모델을 구축하는 것을 목표로 함

1. 외부 데이터 파이프라인 연동 규격 정의

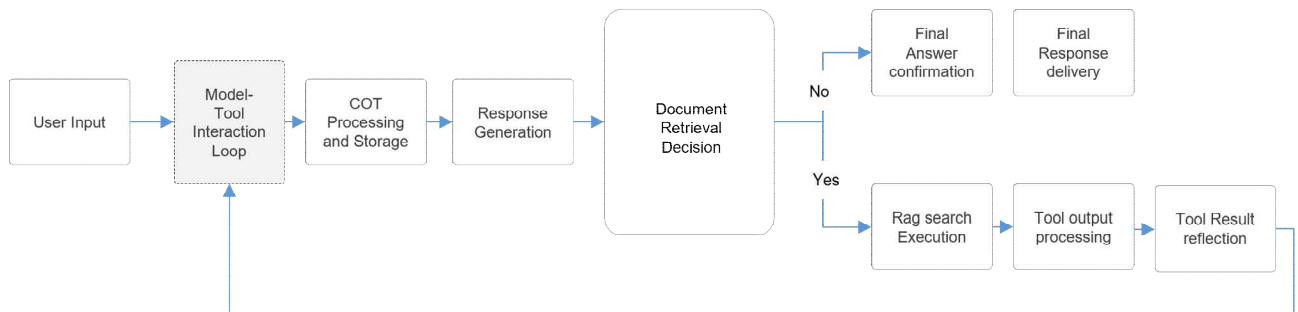
- Toxicity Knowledge CoT 외에도 Organoid Source Data, Source Knowledge Database, Multi-modal Data 등을 참조할 수 있는 스키마 명세서

2. 다중 엔드포인트 베이스라인 모델 설계

- 구조 기반 GNN과 BioBERT / LLM 등의 문헌 임베딩을 결합한 아키텍처 선정을 통해 다양한 입력에 대해 대응할 수 있도록 사전학습 수행
- ADMET 엔드포인트에 대해 Multi-Task 학습 헤드 구성해 파라미터 공유 이점 검증

□ 주요 설계

○ Architecture 구성도



○ 수도 코드 (Pesudo code)

Input: user query Q, dialogue history H

Output: Final Answer A

functionRAG_Response_Pipeline(Q, H):

 Compose message and update history ($H \leftarrow Q$)

 repeat:

 Process reasoning phase (<think>…</think> capture & store)

 Generate partial answer (streaming allowed)

 if external knowledge is required then

$R \leftarrow \text{rag_search}(\text{enhanced query from } Q \text{ and } H)$

 Summarize/structure R and extract source metadata

 Append R to history (for subsequent inference)

 continue

 end if

 if the answer is sufficiently complete then

 break

 end if

 end repeat

 return final answer A

□ 기술 개발 결과

○ 주요 성능 지표(Performance Metrics)

- 개발 목표: MMLU(Massive Multitask Language Understanding) 기준 정답 정확도 90% 이상 달성
- 검증방식: MMLU 형식으로 재구성한 독성 도메인 QA 데이터셋을 기반으로 수행
- 본 연구의 Generalized ADMET inference baseline 또한 동일한 Base LLM을 사용하므로, Base LLM의 일반추론 성능은 두 시스템 간 공통된 지표를 사용
- MMLU-style 독성 벤치마크는 Base LLM의 general reasoning capability를 도메인에 맞게 확장한 평가이며, Toxicity-MMLU의 결과(Accuracy 95%)를 두 항목의 공통 MMLU 성능으로 적용 함
- 이에 따라 제안서에서 요구한 MMLU $\geq 90\%$ 목표치 충족

○ 소스코드 및 주속

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
단일 파일: OpenSearch 기반 단순 RAG + OpenAI Chat Completions(function calling)
- single_turn(사용자 입력 1회)이지만, 모델은 도구를 여러 번 호출할 수 있도록 반복 추론 지원
- RAG 도구: rag_search (semantic-only, OpenSearch KNN 사용)
- rag_search 입력: query (string) 하나만
- rag_search 출력: OpenSearch 검색 응답 전체(JSON)에서 각 hit._source의 'embedding'만 제거
"""

import os
import json
from typing import SimpleNamespace
from typing import Any, Dict, List, Optional
import requests
import openai

try:
    from opensearchpy import OpenSearch
except ImportError:
    raise SystemExit("opensearch-py가 설치되어 있지 않습니다. `pip install opensearch-py` 후 실행하세요.") from e

# =====
# 0) 환경 설정 / 기본값
# =====

OPENAI_MODEL=os.getenv("OPENAI_MODEL","gpt-4o-mini")
OPENAI_BASE_URL=os.getenv("OPENAI_BASE_URL","none")
EMBEDDING_URL=os.getenv("EMBEDDING_URL","text-embedding-3-large")
OPENAI_API_KEY=os.getenv("OPENAI_API_KEY","none")

OS_HOST=os.getenv("OS_HOST","localhost")
OS_PORT=int(os.getenv("OS_PORT","9200"))
OS_INDEX=os.getenv("OS_INDEX","rag-index")
```

```
# 인덱스 매핑에 맞게 필드명 맞춰주세요.
OS_EMBED_FIELD=os.getenv("OS_EMBED_FIELD","embedding") # 벡터 필드 (k-NN)

# k-NN 검색 상수
DEFAULT_TOP_K=int(os.getenv("RAG_TOP_K","5"))

# 도구 반복 호출 상한 (무한루프 방지)
MAX_TOOL_CALLS=int(os.getenv("MAX_TOOL_CALLS","6"))

# =====
# 1) Tool Spec 빌더 (요구 스펙 유지)
# =====

def build_fn_spec_from_raw(raw_tool):
    """MCP raw_tool → OpenAI function spec (요청된 스펙 그대로)"""
    schema=getattr(raw_tool,"inputSchema",{}).or{}
    props={}
    for prop_name,prop_schema in schema.get("properties",{}).items():
        desc=prop_schema.get("description",prop_schema.get("title","")).strip()
        props[prop_name]={
            "type":prop_schema.get("type","string"),
            "description":desc
        }
    required=schema.get("required",[])
    return{
        "type":"function",
        "function":{
            "name":raw_tool.name,
            "description":(raw_tool.description or "").strip(),
            "parameters":{
                "type":"object",
                "properties":props,
                "required":required
            }
        }
    }

# =====
# 2) LLM 파이프라인 (단순화 버전)
# - OpenSearch 클라이언트/임베딩 호출/툴 호출을 한 클래스에 통합
# =====

class LLMPipeline:
    def __init__(self,model_url:str,model_name:str):
        # OpenAI
        openai.api_key=OPENAI_API_KEY
        openai.base_url=model_url
        self.model_name=model_name

        # OpenSearch (로그인/SSL 없이 고정)
        self.os_client=OpenSearch(
            hosts=[{"host":OS_HOST,"port":OS_PORT}],
            use_ssl=False,
            http_compress=True,
        )
        # 인덱스 확인
        try:
            if not self.os_client.indices.exists(index=OS_INDEX):
                raise RuntimeError(f"OpenSearch 인덱스가 존재하지 않습니다: {OS_INDEX}")
        except Exception as e:
            raise SystemExit(f"OpenSearch 연결/인덱스 확인 실패: {e}")

    # 간단 RawTool → tool spec
```

```

raw_tool=SimpleNamespace(
    name="rag_search",
    description=(
        "간단한 RAG 검색 도구입니다. "
        "'임베딩+k-NN(Vector)로 OpenSearch 인덱스를 조회하고, "
        "'검색 응답에서 각 문서의 'embedding' 필드를 제거한 원본 JSON을 반환합니다."
    ),
    inputSchema={
        "type":"object",
        "properties":{
            "query":{
                "type":"string",
                "description":(
                    "검색 질의(필수). 3-8 토큰의 핵심 키워드를 추천합니다. "
                    "예: \"2024 성과계획서\" \"프로그램목표 1-1\""
                ),
            },
        },
        "required":["query"]
    }
)
self.tool_spec=_build_fn_spec_from_raw(raw_tool)
self._tool_impl_map={"rag_search":self._call_rag_search}

# OpenAI Chat 호출
def _chat_once(self,messages:list,tools:list):
    returnopenai.chat.completions.create(
        model=self.model_name,
        messages=messages,
        temperature=0.6,
        top_p=0.95,
        tools=tools,
        tool_choice="auto",
        stream=False
    )

def _embed_query(self,query:str)->List[float]:
    body={"texts":[query],"batch_size":64,"normalize":True}
    response=requests.post(url=EMBEDDING_URL,json=body)
    ifresponse.status_code==200:
        result=response.json()
        returnresult.get("embeddings",[[0.0]*768])[0]
    return[0.0]*768 # 안전 가드

# rag_search 구현: query → embed → kNN → embedding 필드 제거 → JSON 문자열 반환
def _call_rag_search(self,args:Dict[str,Any])->str:
    query=(argsor{}).get("query")
    ifnotqueryornotstr(query).strip():
        returnjson.dumps({"error":"query is required"},ensure_ascii=False)

    # 1) 임베딩
    try:
        emb=self._embed_query(query)
    exceptExceptionas e:
        returnjson.dumps({"error":f"embedding failed: {e}"},ensure_ascii=False)

    # 2) OpenSearch k-NN (BM25 없음)
    body={
        "size":DEFAULT_TOP_K,
        "query":{
            "knn":{
                OS_EMBED_FIELD:{ # 예: "embedding"
                    "vector":emb, # 리스트[float] 그대로
                    "k":DEFAULT_TOP_K
                }
            }
        }
    }

```

```
        }
    }
try:
    resp=self.os_client.search(index=OS_INDEX,body=body)
exceptExceptionas e:
    returnjson.dumps({"error":f"opensearch knn search failed: {e}"},ensure_ascii=False)

# 3) embedding 필드 제거
print(f'opensearch_resp: {resp}')
try:
    filtered=json.loads(json.dumps(resp,ensure_ascii=False)) # deep copy
    hits=filtered.get("hits",{}).get("hits",[])
    forinhits:
        src=h.get("_source",{})
        ifisinstance(src,dict)andOS_EMBED_FIELDinsrc:
            src.pop(OS_EMBED_FIELD,None)
exceptException:
    filtered={"error":"unexpected opensearch response structure"}

returnjson.dumps(filtered,ensure_ascii=False)

@staticmethod
def_as_dict_message(msg_obj)->Dict[str,Any]:
    ifisinstance(msg_obj,dict):
        returnmsg_obj
    d={
        "role":getattr(msg_obj,"role","assistant"),
        "content":getattr(msg_obj,"content",None),
    }
    ifgetattr(msg_obj,"tool_calls",None):
        d["tool_calls"]=[]
        fortcimsg_obj.tool_calls:
            d["tool_calls"].append({
                "id":tc.id,
                "type":tc.type,
                "function":{
                    "name":tc.function.name,
                    "arguments":tc.function.arguments
                }
            })
    returnd

# === public: single-turn 처리 메서드 (요청에 따라 이름 변경) ===
defchat_process(self,user_input:str):
    system_prompt=(
        "당신은 필요한 경우 rag_search 도구를 여러 번 호출해 적절한 컨텍스트를 수집한 뒤 "
        "도구 응답(JSON) 내 정보를 근거로 한국어로 정확하고 간결하게 답변하세요. "
        "rag_search는 'query' 하나만 입력받고, OpenSearch 응답에서 'embedding' 필드를 제거한"
    )
    원본 JSON을 반환합니다."
    "최종적으로 사용자가 이해하기 쉬운 답변을 제시하세요."
)
    messages:List[Dict[str,Any]]=[
        {"role":"system","content":system_prompt},
        {"role":"user","content":user_input}
    ]
    tools=[self.tool_spec]

    tool_calls_used=0
    last_assistant_content:Optional[str]=None

# 반복 루프: 모델이 도구를 0~N번 호출 가능
whiletool_calls_used<=MAX_TOOL_CALLS:
    resp=self._chat_once(messages=messages,tools=tools)
    msg=resp.choices[0].message
    print("\nAssistant:")
    print(msg.contentor"")



---


```

```
# 대화 로그에 assistant 메시지(도구 호출 의도 포함)를 추가
messages.append(self._as_dict_message(msg))

# 도구 호출 유무 확인
tool_calls= getattr(msg,"tool_calls",None)
ifnottool_calls:
    # 도구 호출이 없다면 이것이 최종 답변
    last_assistant_content=msg.get("content")ifisinstance(msg,dict)elsemsg.content
    print("\nAssistant:")
    print(last_assistant_contentor(""))
    break

# 도구 호출 수행
for tc in tool_calls:
    if tc.type!="function":
        continue
    fn_name=tc.function.name
    raw_args=tc.function.arguments or "{}"
    try:
        args=json.loads(raw_args)
    except Exception:
        args={}
    print(f'tool_call args :{args}')

    impl=self._tool_impl_map.get(fn_name)
    ifimplisNone:
        tool_result=json.dumps({"error":f"no implementation for tool '{fn_name}'"},ensure_ascii=False)
    else:
        tool_result=impl(args)

    print(f'tool_result :{tool_result}')

    messages.append({
        "role":"tool",
        "tool_call_id":tc.id,
        "name":fn_name,
        "content":tool_result
    })

    tool_calls_used+=1
    if tool_calls_used>=MAX_TOOL_CALLS:
        break

    if tool_calls_used>=MAX_TOOL_CALLS:
        # 상한 도달 시 마지막으로 'tool_choice=none'으로 강제 종료 답변 유도
        final=openai.chat.completions.create(
            model=self.model_name,
            messages=messages,
            temperature=0.6,
            top_p=0.95,
            tools=tools,
            tool_choice="none",
            stream=False
        )
        last_assistant_content=final.choices[0].message.get("content")
        break

print("\nAssistant:")
print(last_assistant_contentor(""))
```

```
# =====
# 3) main
# =====

def main():
    # OpenAI 키 설정 알림
    openai.api_key="none"

    # 파이프라인 준비
    pipe=LLMPipeline(model_url=OPENAI_BASE_URL,model_name=OPENAI_MODEL)

    # 사용자 입력 (single turn)
    user_input=input("User: ").strip()
    if not user_input:
        print("입력이 비어 있습니다. 프로그램을 종료합니다.")
        return

    # 실행 (chat_process)
    pipe.chat_process(user_input)

if __name__=="__main__":
    main()
```

□ 소스코드 공개

○ Git-Hub 공개

<https://github.com/KwangSun-Ryu/ADMET-AGI-Toxicity-AI-Prototype-and-Baseline-->

□ 참고문헌

1. Du, Bing-Xue, et al. "ADMET property prediction via multi-task graph learning under adaptive auxiliary task selection." *Iscience* 26.11 (2023).
2. Cremer, J., Medrano Sandonas, L., Tkatchenko, A., Clevert, D. A., & De Fabritiis, G. (2023). Equivariant graph neural networks for toxicity prediction. *Chemical Research in Toxicology*, 36(10), 1561-1573.
3. Aksamit, N., Tchagang, A., Li, Y., & Ombuki-Berman, B. (2024). Hybrid fragment-SMILES tokenization for ADMET prediction in drug discovery. *BMC bioinformatics*, 25(1), 255.