라마인덱스(LLAMA INDEX) 활용 실습

중급 과정 (2일, 총 12시간)

강사 : 닥터윌 양진욱

Dr Will

강의 개요

과정 목표

직업훈련 강사를 위한
라마인덱스 활용 능력 배양

학습 기대효과

RAG 시스템 구축 및 교육 현장 적용

교육 대상

생성형 AI를 교육 현장에 도입하고자 하는
직업훈련 강사

이 과정에서는 RAG(Retrieval-Augmented Generation) 시스템과 LlamaIndex의 기본 개념부터 실제 구현까지 단계별로 학습합니다.
Python 환경 구축부터 다양한 문서 처리, 벡터 데이터베이스 연동까지 실습 중심으로 진행됩니다.

https://github.com/llama-index-tutorial/llama-index-tutorial

RAG & LlamaIndex 개념

RAG란?

Retrieval-Augmented Generation의 약자로,
외부 데이터를 검색하여 LLM의 응답 생성을 보강하는 기술입니다.
>> 최신 정보 반영, 환각 현상 감소, 맞춤형 응답 생성이 가능

LlamaIndex란?

다양한 데이터 소스에서 정보를 추출하고 구조화하여 LLM에 연결하는
데이터 프레임워크입니다.
>> 문서 로딩, 인덱싱, 쿼리 처리를 간편하게 구현할 수 있습니다.

LlamaIndex 활용 사례

기업 활용 사례

- 내부 문서 기반 지식 검색 시스템
- 고객 서비스 챗봇 개선
- 제품 매뉴얼 자동 질의응답

교육 현장 적용 사례

- 학습 자료 기반 맞춤형 튜터링
- 교육 콘텐츠 자동 요약 및 퀴즈 생성
- 학생 질문에 대한 맥락 인식 응답

직업훈련 강사 적용 포인트

- 교육 자료 기반 AI 보조 도구 개발
- 산업별 특화 지식베이스 구축
- 학습자 수준별 맞춤형 피드백 시스템

LlamaIndex는 다양한 산업과 교육 분야에서 활용되며, 특히 직업훈련 분야에서는 산업별 전문 지식을 AI 시스템에 효과적으로 통합하는 데 유용합니다.

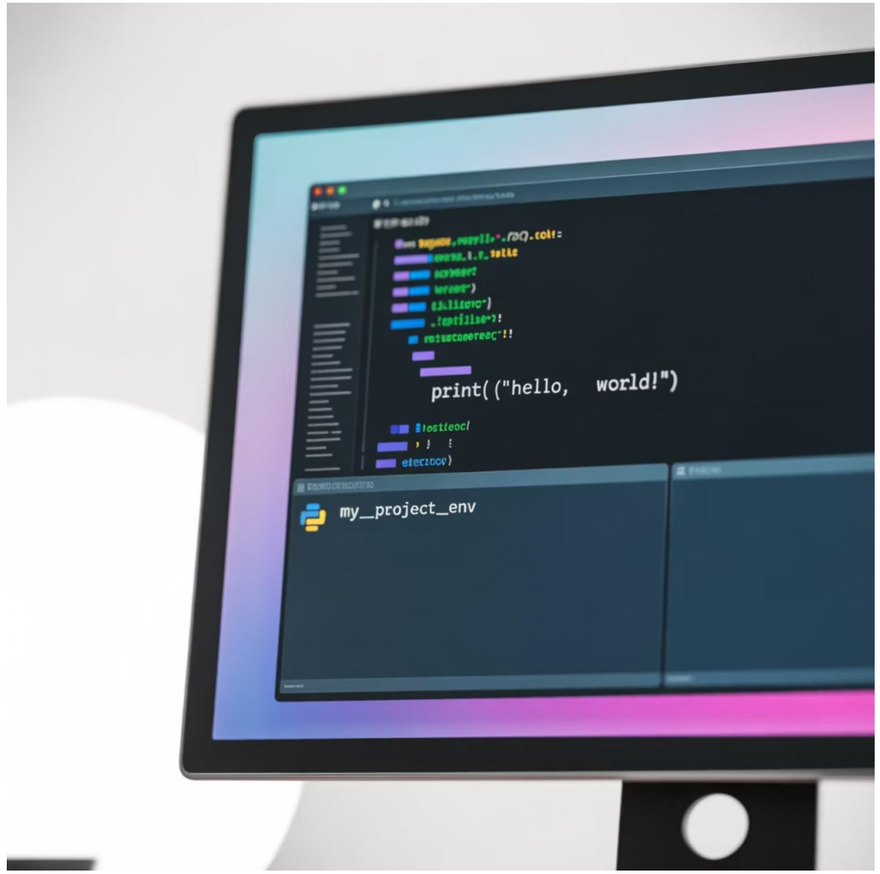
Python & VS Code 환경 구축

Python 설치 및 가상환경

- Python 3.10 이상 설치 확인
- 가상환경 생성: `python -m venv llamaindex`
- 가상환경 활성화:
  - (Windows) llamaindex\\Scripts\\activate
  - (Mac/Linux) source llamaindex/bin/activate

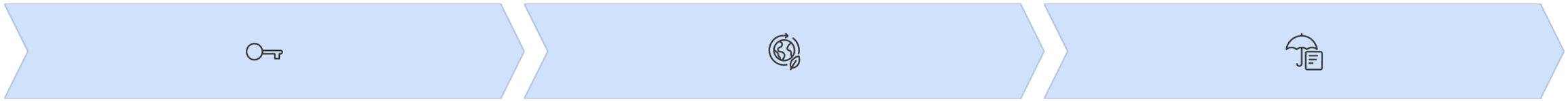
VS Code 확장 기능

- Python 확장 설치
- Jupyter 확장 설치
- Python Indent 확장 설치



효율적인 개발을 위해 적절한 개발 환경 구축이 중요합니다. 가상환경을 사용하면 프로젝트별로 독립된 패키지 관리가 가능합니다.

API Key 발급 및 환경변수 설정



API 키 발급

- OpenAI API 키: platform.openai.com 접속 후 가입 및 발급
- Gemini API 키: ai.google.dev 접속 후 가입 및 발급

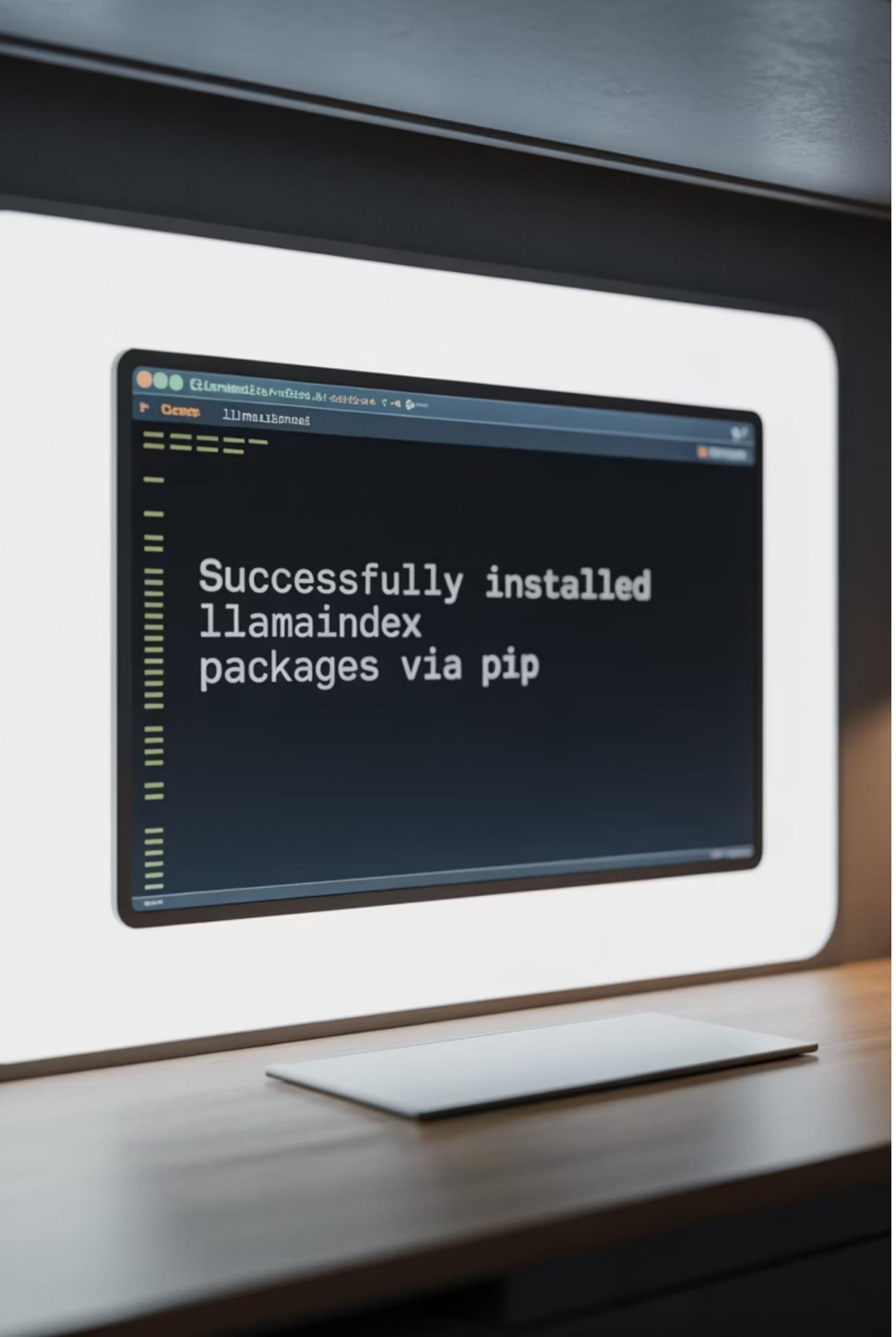
환경변수 설정

- Mac/Linux: `export OPENAI_API_KEY="OUR_OPENAI_KEY"`
- Windows PowerShell: `$env:OPENAI_APKEY="YOUR_OPENAI_KEY"`

보안 관리

- .env 파일 사용 (gitignore에 추가)
- python-dotenv 패키지 활용

API 키는 외부에 노출되지 않도록 환경변수로 관리하는 것이 중요합니다. 특히 공개 저장소에 키가 노출되지 않도록 주의해야 합니다.



LlamaIndex 설치하기

```
pip install llama-index==0.11.0 llama-index-core llama-index-embeddings-openai
```

주요 패키지 구성

- `llama-index-core`: 핵심 기능 제공
- `llama-index-embeddings-openai`: OpenAI 임베딩 모델 연동
- 기타 필요한 커넥터 패키지 (선택적 설치)

환경변수 설정 확인

Python 코드로 환경변수 확인:

```
import osprint(os.environ.get("OPENAI_API_KEY"))
```

환경변수가 제대로 설정되었는지 확인하는 과정이 중요합니다.

DocumentReader & Connector

PDF 문서

기본 PDF 리더 또는 PyPDF2, PDFMiner 등 활용

TXT 파일

기본 텍스트 리더로 간편하게 로드 가능

CSV 데이터

표 형식 데이터 처리를 위한 특수 리더 제공

HWP 문서

한글 문서는 확장 리더 설치 필요 (pyhwp 등)

LlamaIndex는 다양한 문서 형식을 지원하며, SimpleDirectoryReader를 통해 폴더 내 여러 문서를 한 번에 로드할 수 있습니다.
한글 문서(HWP)의 경우 별도 확장 리더가 필요하며, 이미지가 포함된 문서는 OCR 기능을 활용할 수 있습니다.

노드와 분할 전략

분할 전략의 중요성

문서를 어떻게 나누느냐에 따라 검색 정확도와 응답 품질이 크게 달라집니다.
적절한 분할은 관련 정보를 정확히 검색하고 맥락을 유지하는 데 필수적입니다.

주요 분할 전략

- 토큰 단위: 고정 토큰 수로 분할 (예: 512 토큰)
- 문장 단위: 문장 경계로 분할
- 의미 단위: 의미적 연관성에 따라 분할

분할 전략 비교

전략	장점	단점
토큰 단위	구현 간단	맥락 손실 가능
문장 단위	자연스러운 분할	문장 길이 불균형
의미 단위	맥락 유지 우수	구현 복잡

실습: 데이터 로딩

01_quickstart_annotated.ipynb

```
from llama_index.core import SimpleDirectoryReader, VectorStoreInd
ex

# data/txt 폴더에 있는 문서를 모두 로드합니다.
docs = SimpleDirectoryReader('./data/txt').load_data()

# 문서 개수를 출력합니다.
print('docs:', len(docs))
```

SimpleDirectoryReader는 지정된 디렉토리에서 지원되는 모든 문서 형식을 자동으로 로드합니다. recursive=True 옵션을 사용하면 하위 폴더의 문서까지 모두 로드합니다.

문서 메타데이터 확인

로드된 문서는 텍스트 내용뿐만 아니라 파일명, 파일 경로, 생성 날짜 등의 메타데이터도 함께 저장됩니다. 이 메타데이터는 검색 결과에 출처 정보를 제공하는 데 유용합니다.

인덱싱(Indexing)

```
# VectorStoreIndex를 사용해 문서 인덱스를 생성합니다.
index = VectorStoreIndex.from_documents(docs)

# 인덱스에서 쿼리엔진(QueryEngine)을 생성합니다.
qe = index.as_query_engine(similarity_top_k=5)

# 쿼리 실행: 문서 요약 요청
print(qe.query('핵심 정책을 요약해줘'))
```

인덱싱 과정

1. 문서를 노드(청크)로 분할
2. 각 노드를 임베딩 벡터로 변환
3. 벡터를 인덱스에 저장

쿼리 엔진 설정

- `similarity_top_k`: 검색할 최상위 유사 노드 수
- `node_postprocessors`: 검색 후 필터링 처리

쿼리(Query) 처리 과정

1. 쿼리 변환

사용자 질문을 임베딩 벡터로 변환

2. 유사도 검색

벡터 인덱스에서 가장 유사한 노드 검색

3. 후처리

검색된 노드 필터링 및 재정렬

4. 응답 생성

검색된 컨텍스트와 쿼리를 LLM에 전달하여 응답 생성

실습: 질의응답 시스템

```
from llama_index.core.postprocessor import SimilarityPostprocessor

# 유사도 기반 후처리기 설정 (유사도 0.35 이하 결과 필터링)
postprocessor = SimilarityPostprocessor(similarity_cutoff=0.35)

# 쿼리 엔진 생성
query_engine = index.as_query_engine(
    similarity_top_k=8,  # 상위 8개 유사 노드 검색
    node_postprocessors=[postprocessor]  # 후처리기 적용
)

# 쿼리 실행
query = "이 문서는 어떻게 구성되어 있는지 알려줘“
response = query_engine.query(query)

# 결과 출력
print(response)
```

실습: 질의응답 시스템

후처리기(Postprocessor) 활용

SimilarityPostprocessor는 유사도가 특정 임계값(similarity_cutoff) 미만인 노드를 필터링합니다.
이를 통해 관련성이 낮은 정보를 제외하고 더 정확한 응답을 생성할 수 있습니다.

쿼리 엔진 최적화 팁

- similarity_top_k 값을 조정하여 검색 범위 조절
- 다양한 후처리기를 조합하여 검색 품질 향상
- 쿼리 템플릿을 사용하여 질문 형식 표준화

Chroma 실습 & 다중모달 RAG

Chroma 벡터 DB 연동

```
!pip install llama-index-vector-stores-chroma
!pip install llama-index-embeddings-openai
!pip install chromadb
```

다중모달 RAG

다중모달 RAG는 텍스트뿐만 아니라 이미지, 오디오 등 다양한 형태의
데이터를 함께 처리할 수 있는 시스템입니다.
이미지 기반 RAG는 이미지에서 특징을 추출하여 벡터화한 후
유사 이미지를 검색하고, 이를 문서 컨텍스트와 결합하여 응답을 생성합니다.

RAG 에이전트 및 고급 검색 기법

LLM 기반 검색 증강 생성(RAG) 시스템의 구현과 최적화 방법론

목차

1

RAG 에이전트 개념

검색 증강 생성의 기본 원리와 구조

2

간단한 RAG 에이전트 구현

문서 검색 및 요약 도구 결합 실습

3

고급 검색 기법

Re-ranking과 HYDE 기법을 통한 검색 성능 향상

4

Function Calling 및 Text-to-SQL

외부 API 연동과 데이터베이스 쿼리 자동화

5

MCP 및 최종 프로젝트

모듈형 컨텍스트 프로세싱과 종합 프로젝트 안내

RAG 에이전트 개념

RAG(Retrieval-Augmented Generation) 에이전트는 대규모 언어 모델(LLM)의 생성 능력과 외부 지식 검색을 결합한 시스템입니다.

- 최신 정보 접근 가능
- 환각(hallucination) 감소
- 도메인 특화 지식 활용
- 소스 인용 및 추적 가능

RAG 에이전트는 사용자 질의를 분석하고, 관련 문서를 검색한 후, 검색된 컨텍스트를 바탕으로 응답을 생성합니다.

간단한 RAG 에이전트 구현

문서 검색 도구와 요약 도구를 결합한 기본 RAG 에이전트를 구현해 보겠습니다.

```
# (의사 코드) 문서 검색 Tool + 요약 Tool 결합 에이전트
tools = [retrieval tool, summarizer tool]
state = {}
def agent(query):
   if '요약' in query:
      ctx = retrieval tool(query)
      return summarizer tool(ctx)
   else:
      return retrieval_tool(query)

print(agent("이 보고서를 3줄로 요약해줘"))
```

이 간단한 에이전트는 사용자 쿼리에 '요약'이라는 키워드가 포함되어 있는지 확인하고, 그에 따라 적절한 도구를 선택하여 실행합니다.

Re-ranking을 통한 검색 최적화

검색 결과의 품질을 향상시키기 위해 Re-ranking 기법을 적용할 수 있습니다.
초기 검색 결과를 더 정교한 모델로 재평가하는 과정입니다.

Re-ranking의 개념과 필요성
전통적인 검색 시스템은 키워드 매칭이나 TF-IDF 같은 기본적인 점수 계산을 통해 결과를 반환합니다.
하지만 이런 방식은 사용자의 실제 의도나 문맥을 충분히 반영하지 못할 수 있습니다.
Re-ranking은 이런 한계를 보완하여 사용자에게 더 유용한 결과를 제공합니다.

구현 전략
Two-stage 검색 아키텍처

1. First Stage (Retrieval): 빠른 검색으로 후보군 선별 (예: BM25, dense retrieval)
2. Second Stage (Re-ranking): 정교한 모델로 상위 결과 재정렬

HYDE 기법 소개

HYDE란?

Hypothetical Document Embeddings의 약자로,
검색 성능을 향상시키는 고급 기법입니다.

작동 원리:

1. LLM이 쿼리에 대한 가상의 이상적인 문서를 생성
2. 생성된 가상 문서를 임베딩
3. 이 임베딩을 사용하여 실제 문서 검색

이 방식은 쿼리와 문서 간의 의미적 간극을
을 줄여 검색 정확도를 높입니다.

HYDE 적용 실습

HYDE 기법을 실제 검색 시스템에 적용하여 검색 정확도를 개선해 보겠습니다.

1단계: 쿼리 분석

사용자 쿼리의 의도와 주제를 파악합니다.

2단계: 가상 문서 생성

LLM을 사용하여 쿼리에 대한 이상적인 응답 문서를 생성합니다.

3단계: 임베딩 및 검색

생성된 가상 문서를 임베딩하여 실제 문서 코퍼스에서 유사한 문서를 검색합니다.

4단계: 결과 재정렬

검색된 문서를 관련성에 따라 재정렬하여 최종 응답을 생성합니다.

이 과정을 통해 단순 키워드 매칭이나 기본 임베딩 검색보다 훨씬 정확한 검색 결과를 얻을 수 있습니다.

Function Calling 개요

Function Calling은 LLM이 외부 API나 도구를 호출하는 능력을 말합니다. 이를 통해:

- 실시간 데이터 접근 (날씨, 주가 등)
- 계산 및 데이터 처리 작업 수행
- 외부 시스템과의 상호작용
- 복잡한 워크플로우 자동화

LLM은 사용자 의도를 파악하고, 적절한 함수를 선택하여 필요한 매개변수와 함께 호출합니다.

Text-to-SQL 개념

Text-to-SQL은 자연어 질문을 SQL 쿼리로 변환하는 기술입니다.
이를 통해 데이터베이스 지식이 없는 사용자도 복잡한 데이터 분석이 가능해집니다.

주요 구성 요소:

- 스키마 이해: 테이블과 컬럼 구조 파악
- 의도 분석: 사용자 질문의 의도 해석
- SQL 생성: 적절한 쿼리 구문 작성
- 실행 및 결과 해석: 쿼리 실행 및 결과 설명

Text-to-SQL 실습

장점

- 데이터베이스 전문 지식 없이도 복잡한 쿼리 가능
- 자연어로 데이터 분석 접근성 향상
- 반복적인 쿼리 작업 자동화

도전 과제

- 복잡한 스키마 이해의 어려움
- 모호한 질문 해석
- 최적화된 쿼리 생성

MCP(Modular Context Processing) 개요

MCP는 LLM 애플리케이션의 컨텍스트 처리를 모듈화하는 아키텍처 패턴입니다.

주요 특징:

- **모듈성**: 독립적인 컨텍스트 처리 모듈
- **확장성**: 새로운 기능을 쉽게 추가 가능
- **재사용성**: 공통 컴포넌트 공유
- **유연성**: 다양한 사용 사례에 적응

MCP는 서버-클라이언트 구조로 구현되어 다양한 도구와 기능을 효율적으로 관리합니다.

문서 검색 MCP 실습

클라이언트

사용자 쿼리 수신 및 MCP 서버에 요청 전송

MCP 서버

등록된 도구 실행 및 컨텍스트 처리

벡터 데이터베이스

문서 임베딩 저장 및 유사도 검색

응답 생성

검색 결과와 LLM을 활용한 최종 응답 생성


Thank You