ChatGPT

금융 AI 해커톤 LLM 기반 시스템 개발 보고서

1. 전체 개발 방향 개요

금융보안 도메인 Q&A 성능을 높이기 위해 **한국어 사전학습 LLM**을 미세튜닝하고 최적화하는 방향으로 개발합니다. 기본 모델로는 공개된 한국어 LLM인 **Hugging Face의** `beomi/gemma-ko-7b` 등을 사용합니다 [1]. 이 모델을 금융보안 분야의 **FSKU 평가 문항** 데이터로 **LoRA**(Low-Rank Adaptation) 기법을 활용해 파인튜닝하여, **객관식**과 **주관식** 질문에 모두 정확히 답하도록 특화합니다 [2].

LoRA 기반 튜닝을 선택한 이유는 다음과 같습니다:

- **효율적 미세튜닝:** LoRA는 모델의 일부 가중치만 학습해 **학습 파라미터 수를 크게 줄여**줍니다. 이를 통해 대용량 LLM도 비교적 적은 자원으로 도메인 맞춤 훈련이 가능합니다 [3].
- **원본 지식 보존:** 전체 가중치를 업데이트하지 않으므로, 사전학습 모델의 일반적인 언어 지식을 유지하면서도 금융보안 Q&A에 필요한 **추가 지식만 학습**시킬 수 있습니다.
- **빠른 실험 반복:** 학습 파라미터가 적어 **학습 시간이 단축**되고, 여러 하이퍼파라미터 시도를 빠르게 반복할 수 있습니다.

튜닝된 모델은 **8-bit 또는 4-bit 양자화**를 적용해 경량화합니다. 양자화는 모델 가중치를 4비트 등으로 표현하여 **메모리와 디스크 사용량을 1/2 ~ 1/8 수준으로 감소**시킵니다 [4] [5]. 본 대회의 추론 환경이 **GPU 24GB VRAM, 디스크 40GB 제한**이 있으므로 [6] [7], 7억~130억 규모 모델을 양자화하여 저장해야 오프라인 서버에서도 원활히 구동됩니다. 실제 베이스라인도 `load_in_4bit=True`로 Gemma-7B를 4비트 로드합니다 [8].

또한 **프롬프트 엔지니어링**을 통해 **단일 LLM**이 두 유형의 질문에 모두 응답하도록 구조화합니다 [2]. 예컨대 객관식은 **정답 번호만 출력**하게, 주관식은 **키워드가 포함된 간결한 해설**을 작성하게 프롬프트를 설계합니다. 최종 출력은 반드시 LLM 생성에 기반해야 하며, **룰 기반이나 정답 목록 대조만으로 생성된 답변은 불가**합니다 [9]. 필요 시 **검색 증강 생성**(RAG) 기법으로 외부 지식을 참고할 수도 있지만, 검색 결과를 그대로 복붙하지 않고 **LLM으로 재가공**해야 규칙에 부합합니다 [9].

요약하면, **한국어 LLM + LoRA 미세튜닝 + 양자화 추론**이라는 틀 안에서, 금융보안 QA에 특화된 모델을 개발합니다. 데이터 전처리부터 모델 훈련, 프롬프트 설계, 추론 파이프라인, 패키징까지 일관된 프로세스로 구현하여 **오프라인 환경**에서 **단일 LLM**으로 동작하는 **완결형 시스템**을 목표로 합니다 [10].

2. GitHub 프로젝트 폴더 구조 및 README 구성

전체 프로젝트를 이해하기 쉽도록 **모듈별로 폴더 구조**를 설계합니다. 아래는 권장되는 GitHub 폴더 구조와 각 구성요소의 설명입니다:

- **README.md** – 프로젝트 개요, 사용 방법, 데이터/모델 경로, 실행 예시 등을 담은 문서.
- **requirements.txt** – 필요한 패키지 명세 (예: torch, transformers, peft 등) [11]. 대회 환경(PyTorch 2.1.0 등)에 맞춰 버전을 지정합니다.
- **setup.sh** (선택) – 의존성 설치 및 가상환경 설정 스크립트 [12].
- **data/** – 데이터 디렉토리 (예: `train.csv`, `test.csv`, `sample_submission.csv` 등). 경로는 코드에서 **상대경로**로 참조하여 재현성을 높입니다 [13].
- **models/** – 모델 가중치 파일 저장 폴더. 예를 들어 `base_model/`에 사전학습 모델(gemma-ko-7b) 파일을 두고, `finetuned_model/`에 LoRA 어댑터 또는 최종 튜닝 모델을 저장합니다.

- **src/** – 소스 코드 디렉토리 (또는 최상위에 바로 배치):
- `preprocess.py` – 데이터 전처리 스크립트 (예: 훈련용 Q&A 포맷 변환).
- `finetune_lora.py` – LoRA 미세튜닝 학습 스크립트. 데이터셋 로드, LoRA 구성 및 Trainer 실행 포함.
- `inference.py` – **추론 스크립트**. test 데이터를 불러와 전처리 → LLM 예측 → 후처리 → `submission.csv` 생성까지 **일괄 수행**합니다 [14].
- `utils.py` – 공통 함수 모음 (예: **프롬프트 생성 함수**, 출력 후처리 함수 등 재사용 코드).
- `config.json` (선택) – 하이퍼파라미터나 경로 설정을 위한 설정파일.

README.md 핵심 항목:

- **프로젝트 개요:** 해결하려는 문제 설명 (금융보안 QA 생성)과 솔루션 요약 (LLM+LoRA 접근).
- **데이터:** 제공된 FSKU 문항 데이터 설명, 전처리 방식, 외부 데이터 사용 여부와 출처 (사용했다면 라이선스 명시).
- **모델:** 사용한 사전학습 모델 정보 (예: gemma-ko-7b, 공개일 및 라이선스 [15])와 왜 선택했는지, 미세튜닝 방법(LoRA) 설명.
- **환경 세팅:** 요구되는 패키지와 버전 (requirements.txt 참고), 실행 환경(Python 3.10, CUDA 11.8 등) 안내.
- **훈련 방법:** LoRA 파인튜닝 절차 요약 – 주요 하이퍼파라미터 (learning rate, epoch 등)와 훈련에 소요된 자원 (GPU, 시간).
- **프롬프트 설계:** 객관식/주관식에 대해 어떻게 프롬프트를 구성했는지 예시 포함 설명 (아래 4번 항목에서 상세 기술).
- **추론 사용법:** `inference.py` 실행 방법과 입력/출력 형식. 예를 들어 `python inference.py --model_dir models/finetuned_model/ --test data/test.csv` 등의 사용 예와, 결과로 `submission.csv`가 생성됨을 기술.
- **파일 구조:** 위에서 설계한 디렉토리 구조와 각 파일 역할을 표나 리스트로 정리.
- **결과 및 평가:** 모델의 성능 간략 요약 (리더보드 점수 등)과 한계, 추후 개선 방향. 특히 주관식 답변에 대한 품질 논의 등을 포함하면 좋습니다.
- **대회 규칙 준수 사항:** 모델/데이터 라이선스 적합성, 외부 API 미사용(완전 로컬 추론) 명시 [16], 단일 LLM 사용 강조 등 규칙 준수를 확인합니다.

이러한 구조와 README 작성을 통해, **프로젝트 전반**을 일목요연하게 정리하고 사용자(평가자)가 쉽게 재현할 수 있도록 합니다. 특히 **코드 재현성**(requirements 명시, 경로 일관성)과 **설계 의도**(왜 그런 선택을 했는지)에 대한 서술이 중요합니다.

3. LoRA 기반 파인튜닝 흐름 및 핵심 코드 설명

파인튜닝 데이터 준비: 우선 제공된 금융보안 Q&A 데이터를 모델이 학습할 수 있는 형식으로 가공합니다. 각 문항은 질문과 정답으로 이루어진 **지도학습** 형태로 사용됩니다. 객관식 문항의 경우 질문 본문과 보기 선택지, 정답 번호가 주어지며, 주관식은 질문과 모범 답안 텍스트가 주어질 것입니다. 이를 통합하여 **Instruction-Answer 포맷**의 텍스트로 변환합니다. 예를 들어, 하나의 훈련 샘플을 아래와 같이 구성합니다:

- **객관식**: `"질문: ...\n1. 보기1\n2. 보기2\n3. 보기3\n...\n정답: 2"` (정답 번호만 기재)
- **주관식**: `"질문: ...\n정답: [모범 답안 텍스트]"`

이처럼 프롬프트에 질문 및 보기, 그리고 기대하는 출력 형식을 넣고, **모델 출력이 정답만 생성**하도록 정답 부분을 레이블로 삼아 학습합니다. (훈련 시에는 `"정답: ..."~` 부분을 모델의 타깃 시퀀스로 사용). 데이터가 충분하지 않다면 **데이터 증강**도 고려할 수 있습니다 (예: 유사 질문 생성 또는 비슷한 공개 문항 추가) [17]. 단, 증강 시 사용한 외부 데이터/모델은 규칙상 공개 라이선스여야 합니다 [18].

모델 및 LoRA 설정: HuggingFace `transformers`와 `peft` 라이브러리를 이용해 모델과 LoRA 어댑터를 초기화합니다. 핵심 흐름은 다음과 같습니다:

1. **사전학습 모델 로드:** `AutoModelForCausalLM.from_pretrained`를 사용해 base 모델을 로드합니다. 메모리 효율을 위해 **4-bit 양자화** 설정을 켜서 로드합니다 (예: `BitsAndBytesConfig(load_in_4bit=True, bnb_4bit_quant_type="nf4", ...)` 전달). 베이스라인 코드에서도 4bit로 모델을 불러오고 있습니다 [8]. 모델 로드시 `device_map="auto"`나 `.to(device)`를 지정해 GPU 메모리에 올라가도록 합니다.

2. **LoRA 구성 생성:** LoRA 어댑터의 하이퍼파라미터를 설정합니다. HuggingFace `peft.LoraConfig` 객체를 생성하며, **주요 파라미터**:

3. `r` (랭크): 새로 학습되는 저랭크 행렬의 차원. 작을수록 학습 파라미터가 적지만 표현력은 낮아집니다. 일반적으로 **r=8 또는 16** 등을 많이 사용합니다 (모델이 클 경우 성능 향상을 위해 16↑도 고려).
4. `lora_alpha`: LoRA의 스케일링 계수로, 최종 LoRA 적용값의 크기를 조절합니다. 흔히 **α를 랭크의 2배로 설정**(예: r=8이면 α=16)하여 안정적인 학습을 도모합니다 [19]. α가 클수록 LoRA 보정의 영향력이 커지고, 너무 크면 불안정할 수 있어 적절한 값이 필요합니다.
5. `lora_dropout`: LoRA 적용에 dropout을 줄 비율. **0.05 ~ 0.1** 정도를 줘서 과적합을 완화할 수 있습니다 [19]. 데이터량이 적을수록 약간의 dropout이 도움이 됩니다.
6. `target_modules`: LoRA를 적용할 모델 내 레이어 이름들을 지정합니다. 일반적인 LLM(LLaMA 계열 등)은 `q_proj`, `v_proj`, `k_proj`, `o_proj` (Attention의 Query/Value/Key/Output 프로젝션) 및 MLP의 일부 등에 LoRA를 적용합니다. 최신 `peft`에서는 지원 모델의 경우 자동으로 대상 모듈을 선택하지만, 새로운 아키텍처일 경우 오류가 날 수 있어 해당 레이어명을 명시해야 합니다 [20] [21]. Gemma-7B 모델이 LLaMA 유사 구조라면 기본값으로 충분할 가능성이 높습니다.

예시 코드:

```
from peft import LoraConfig, get_peft_model, prepare_model_for_kbit_training
model = prepare_model_for_kbit_training(model)  # 4bit 모델을 미세튜닝 준비 (LayerNorm 등 정규화)
lora_config = LoraConfig(
    r=8, lora_alpha=16, target_modules=["q_proj","v_proj","k_proj","o_proj"],
    lora_dropout=0.05, bias="none", task_type="CAUSAL_LM"
)
model = get_peft_model(model, lora_config)
print(model.print_trainable_parameters())  # 훈련가능 파라미터 수 출력
```

위와 같이 하면 모델 내 필요한 부분에 **LoRA 레이어가 주입**되고, 전체 파라미터 대비 매우 적은 수(%로 1% 미만)가 학습 대상이 됨을 확인할 수 있습니다 [3].

1. **훈련 루프/Trainer 설정:** HuggingFace `Trainer` 또는 `trl` 라이브러리의 `SFTTrainer`를 활용하여 Supervised Fine-Tuning을 수행합니다 [22].
2. **데이터셋:** `datasets` 라이브러리로 전처리한 학습 데이터를 불러오고 (`train_test_split` 등으로 일부를 validation에 활용 가능), 데이터 포맷에 따라 collate_fn을 정의합니다. `SFTTrainer`를 사용하면 대화/지시 형식에 맞게 자동으로 input을 구성해줄 수 있습니다.
3. **학습 하이퍼파라미터**: learning rate, batch size, epoch/step 수 등을 설정합니다. LoRA 튜닝 시 **학습률**은 대체로 **1e-4 ~ 2e-5 수준**에서 실험하며, 데이터 크기와 epoch 수에 따라 조절합니다. (예: 데이터가 수만 건 이상이면 2~3 epoch에 2e-4도 시도, 데이터가 적으면 1e-5로 낮춰 장기간 학습으로 미세하게 조정). 참고로 공개된 Gemma-7B Instruction 튜닝 예에서는 **AdamW**, **lr=1.5e-5**, **cosine 스케줄러** 등을 사용했습니다 [23].

4. **기타**: gradient accumulation, fp16 혼합정밀도 사용 (`Trainer`에서 `fp16=True` 설정) 등을 통해 메모리 효율을 높입니다. LoRA 외 파라미터는 freeze되므로 weight decay는 0 또는 아주 작은 값으로 둡니다. 또한 `save_steps`나 `epoch`마다 checkpoint를 저장하고, `early_stopping`으로 과적합을 방지할 수 있습니다.

`Trainer` 실행 시, `model`은 LoRA가 적용된 PeftModel이고, `train_dataset`과 `eval_dataset`을 주고 `trainer.train()`을 호출합니다. 학습 로그를 보며 loss 감소 추이를 확인하고 적절한 시점에 종료하거나 validation loss가 상승하면 조기종료합니다.

1. **튜닝 결과 및 저장:** 훈련 완료 후 `trainer.model`(혹은 LoRA 적용 모델)을 저장합니다. `peft`를 이용했다면 **어댑터만 별도로 저장**할 수도 있고 (`model.save_pretrained("adapter")` 시 수 MB~수백 MB의 LoRA 가중치만 저장), 또는 **원본 모델에 합쳐진 전체 가중치**를 저장할 수도 있습니다. 합칠 경우 `model.merge_and_unload()`를 호출하여 LoRA 가중치를 base 모델에 반영한 후 `model.save_pretrained("finetuned_model")`로 저장하면, 추론 시 추가 세팅 없이 곧바로 사용 가능한 완전한 모델이 얻어집니다. 다만 4bit로 로드한 모델은 merge 시 내부적으로 fp16으로 바꿔야 하므로 메모리 고려가 필요합니다. 어댑터만 저장할 경우, 추론 때 base 모델 로드 후 `PeftModel.from_pretrained`으로 적용하여 사용하게 됩니다.

요약하면, **LoRA 파인튜닝**은 **(데이터 준비) → (4bit 모델 로드) → (LoRA 구성 및 적용) → (학습 실행) → (결과 저장)** 흐름으로 이루어집니다. 핵심 코드는 모델 로드와 LoRA 설정 부분으로, 위 예시처럼 몇 줄로 구현 가능합니다. 이러한 접근으로 **학습 가능한 파라미터를 0.3% 수준**으로 줄여도 [3] 금융 문항에 특화된 성능 향상을 얻을 수 있습니다.

사용자 튜닝 가능 핵심 요소: 사용자 입장에서 조절할 수 있는 중요한 하이퍼파라미터/전략은 다음과 같습니다: - LoRA **랭크 r** 및 **알파 α**: 모델 용량 대비 충분한지, 과소적합/과적합 여부에 따라 조절. - LoRA **드롭아웃율**: 데이터 적을수록 소폭 늘려 일반화 향상. - **학습률** 스케줄: 초기에는 높게 후반에 낮추는 **cosine**이나 **linear decay** 사용으로 안정적 수렴. - **문항 포맷**: 객관식과 주관식 데이터를 한 모델에 섞어 학습하므로, 한 배치내 두 형식이 섞이면 학습이 불안정할 수 있습니다. 이를 완화하기 위해 **batch를 유형별로 묶거나**, 프롬프트에 명시적으로 형식을 표시하는 전략을 실험할 수 있습니다. - **손실 함수 가중치 조정**: 객관식과 주관식의 손실 비중을 조절하고자 한다면 (예컨대 객관식 정확도 향상을 더 원하면) 데이터 증량이나 손실 가중 조정도 고려 가능합니다.

이런 튜닝 과정을 통해 최종적으로 **금융보안 QA에 최적화된 LLM 가중치**를 얻고, 이를 다음 단계인 추론 및 배포에 활용합니다.

4. 프롬프트 설계와 LangChain 활용 검토

프롬프트 엔지니어링은 이 프로젝트의 성패를 좌우할 중요한 요소입니다. 단일 LLM으로 두 가지 유형의 질문에 모두 답해야 하므로, **프롬프트 구조를 유형별로 명확히 구분**하여 모델의 응답 형식을 제어합니다 [2].

우선 **역할 지시**를 통해 모델에게 금융보안 분야 전문가로서 답하도록 맥락을 부여합니다. 베이스라인에서는 프롬프트 맨 앞에 "당신은 금융보안 전문가입니다." 라는 문장을 넣어, 모델이 전문적인 어투로 답변하게 유도했습니다 [24]. 다음으로 **문항 유형별 지시사항**을 추가합니다:

- **객관식 문항**: 모델이 오직 정답 번호만 출력하도록 지시해야 합니다. 불필요한 설명이나 논리를 답변에 덧붙이면 자동 채점 정확도가 떨어지므로, 다음과 같은 템플릿을 사용합니다:

프롬프트 예시 (객관식) [24]:

```
당신은 금융보안 전문가입니다.
아래 질문에 대해 적절한 **정답 선택지 번호만 출력**하세요.
```

```
질문: {질문 내용}
선택지:
1. ...
2. ...
3. ... (필요한 만큼 선택지 나열)

답변:
```

위와 같은 구조로, **"정답 선택지 번호만 출력"**하라는 지시를 굵게 강조하여 모델이 답변란에 숫자만 내놓도록 합니다 [25]. 실제 베이스라인 구현에서도 이 프롬프트를 생성하여 모델에 입력하고 있습니다. 이런 방식으로 질문-보기 목록-답변 포맷을 확립하면, 모델은 답변을 생성할 때 선택지 내용까지 고려한 뒤 **해당 번호만 생성**하게 됩니다. 예를 들어 질문이 "다음 중 개인정보 암호화 알고리즘이 아닌 것은?"이고 보기 1~4가 있다면, 모델은 `답변:` 다음에 `3` 과 같이 한 글자만 출력하게 되는 식입니다.

- **주관식 문항**: 모델에게 해당 질문에 대한 간략하고 정확한 서술형 답변을 작성하도록 유도합니다. 객관식과는 달리 텍스트로 된 설명형 답이 필요하므로, 다음과 같은 템플릿을 사용합니다:

프롬프트 예시 (주관식) [26]:

```
당신은 금융보안 전문가입니다.
아래 주관식 질문에 대해 정확하고 간략한 설명을 작성하세요.

질문: {질문 내용}

답변:
```

여기서는 **"정확하고 간략한 설명"**을 요청하여, 너무 장황하지 않으면서 핵심을 찌르는 답변을 하도록 유도합니다 [27]. 또한 주관식 답변의 채점 기준에 **키워드 재현율**과 **의미 유사도**가 있으므로 [28], 모델이 답변 생성 시 질문의 핵심 키워드를 활용하도록 하는 것이 중요합니다. 이를 위해 질문 내용에 포함된 전문용어를 답변에도 자연스럽게 포함하거나, 금융보안 맥락의 용어를 쓰도록 학습 단계에서 유도합니다. 역할 지시를 "금융보안 전문가"로 한 것도 도메인 용어 사용을 늘리려는 의도입니다.

이러한 프롬프트 설계는 **훈련 단계**에서도 그대로 활용됩니다. 모델이 미세튜닝 시 이미 동일한 형식의 입력을 접하고 정답을 생성하도록 학습하므로, 추론 시에도 일관된 템플릿을 쓰는 것이 효과적입니다. 특히 객관식의 경우 **모델 출력이 불필요한 문장 없이 숫자만 나오도록** 훈련부터 제약을 거는 것이 성능에 중요합니다. 만약 모델이 훈련 중에도 숫자 대신 이유를 설명하려 한다면, 프롬프트에 **"번호만 답하라"**는 지시를 반복 강조하거나, 그런 실패 사례를 추가 학습시켜 교정합니다.

LangChain 사용 여부 검토: LangChain은 LLM 활용을 위한 프레임워크로, 프롬프트 체인, 검색 결합, 여러 툴 연계 등의 고급 기능을 제공합니다. 이번 해커톤 시나리오에서 **LangChain을 사용할 수는 있지만 필수는 아닙니다.** 고려할 수 있는 LangChain 활용 방안과 유의사항은 다음과 같습니다:

- **프롬프트 관리 측면:** LangChain의 `PromptTemplate` 등을 사용하면 위에서 설계한 프롬프트 양식을 코드에서 깔끔하게 관리할 수 있습니다. 하지만 우리 사례처럼 **단순한 포맷**에서는 Python f-string만으로도 충분하며, LangChain 도입이 큰 이점은 아닙니다. 오히려 종속 라이브러리가 하나 늘어나는 셈이므로, 코드 단순성을 원한다면 **프롬프트 빌드는 직접 구현**해도 문제 없습니다.

- **검색 증강 (RAG) 측면:** 만약 금융보안 관련 외부 지식(예: 규제 문서, 가이드라인)을 활용해 **추론 정확도를 높이고자 한다면**, LangChain을 활용해 **질문 → 벡터DB 검색 → LLM 응답 생성** 체인을 구성할 수 있습니다. 예를

들어, 미리 금융보안 지식을 담은 문서를 임베딩하여 로컬 벡터스토어(FAISS 등)에 저장하고, 질문 입력 시 관련 문서를 검색해 **컨텍스트**로 제공한 후 답변을 생성하게 할 수 있습니다. 이러한 **Retrieval QA 체인**은 LangChain으로 쉽게 구현할 수 있습니다. **다만** 고려해야 할 점:

- 대회 규칙에 따라 외부 데이터 사용 시 **라이선스 요건**을 만족해야 하며 [18], 사용한 데이터의 출처를 제출해야 합니다. 허가된 데이터만 활용해야 함을 명심합니다.
- 검색에 사용하는 임베딩 모델이나 QA 체인은 **인터넷 없이 로컬**에서 작동해야 합니다 [29] [16]. 즉 OpenAI API 등을 쓰면 탈락이므로, HuggingFace의 `SentenceTransformer` 등의 오픈 임베딩 모델을 requirements에 포함해 로컬로 돌리는 방식이어야 합니다.
- RAG를 쓰더라도 최종 답변은 **LLM이 생성한 텍스트**여야 합니다 [9]. LangChain으로 검색한 문서를 그대로 내보내면 안 되고, 반드시 **모델이 읽고 재구성**하도록 프롬프트에 넣어야 합니다.
- **멀티스텝 체인/툴 사용:** LangChain을 활용하면 예컨대 수학 계산이 필요한 경우 계산기 툴을 호출하거나, 질문 유형을 분류하는 Chain을 만들 수도 있습니다. 그러나 이번 과제에서는 **질문 유형 분류**정도만 간단히 필요하며, 이는 이미 코드로 구현한 `is_multiple_choice` 함수로 처리 가능합니다 [30]. LLM 에이전트 기능 등은 과하면 오히려 복잡성만 높입니다. 따라서 **필요 최소한의 논리만 코드로 직접 구현**하고, LangChain의 에이전트 기능은 사용하지 않는 편이 낫습니다.

정리하면: LangChain은 선택 사항이며, **Prompt Template 관리나 RAG 구현에 도움이 될 수 있지만 필수 아님**입니다. 대회 규칙상 LangChain 자체 사용을 제한하진 않으므로, 사용하더라도 감점 요인은 없습니다. 다만 **Offline 동작**과 **외부 API 미사용** 원칙만 지킨다면 자유롭게 활용 가능하겠습니다. 프로젝트 난이도와 남은 시간 등을 고려해, **기본적인 프롬프트 전략이 효과적이라면 굳이 LangChain에 의존하지 않아도 무방**합니다. 반대로, 주관식 답변의 정확도를 높이기 위해 **백과사전적 지식이 필요**하고 이를 위해 RAG를 도입해야 한다고 판단되면, LangChain으로 구현하되 상기한 요건들을 충족시키면 됩니다.

5. Inference.py 구성안 및 개발 체크리스트

해커톤 제출용 `inference.py` 스크립트는 **단일 실행**으로 최종 제출 파일(`submission.csv`)을 생성할 수 있어야 합니다 [31] [32]. 이를 위해 `inference.py`는 다음과 같은 **구성 요소**와 **절차**를 갖추도록 설계합니다:

1. **모델 및 토크나이저 로드:**
2. `transformers`에서 `AutoTokenizer.from_pretrained`와 `AutoModelForCausalLM.from_pretrained`를 사용해 **튜닝된 모델**을 로드합니다. LoRA 어댑터를 별도로 저장했다면 먼저 base 모델을 로드한 후 `peft.PeftModel.from_pretrained`로 어댑터를 적용합니다.
3. 로드 시 `device_map="auto"`로 GPU에 로드하고, `load_in_4bit=True` (또는 `load_in_8bit=True`) 옵션으로 **양자화된 추론**을 수행합니다 [8].
4. `torch.cuda.is_available()` 등을 확인해 디바이스 할당을 점검하고, 메모리 여유를 고려해 `torch.inference_mode(True)` 컨텍스트나 `.eval()` 모드로 설정하여 불필요한 gradient 연산을 끕니다.
5. 로드한 모델과 토크나이저 객체는 **전역 변수**로 유지하여, 여러 샘플 추론 시 매번 재로드하지 않도록 합니다.
6. **입력 데이터 불러오기:**
7. 대회에서 제공한 `test.csv` (혹은 지정된 입력 경로의 파일)을 읽어옵니다. 예를 들어 Pandas `pd.read_csv("data/test.csv")`로 DataFrame을 만들고, 필요한 컬럼(예: `question` 또는 `내용` 등)을 가져옵니다 [33].
8. 테스트 데이터에 식별자(ID)가 있다면 함께 가져와 나중에 제출파일에 포함시킵니다.

9. 데이터 크기에 따라 **batch 처리**는 선택사항입니다. 질문당 모델 생성 호출이 1회씩 이뤄지므로, 보통 루프로 순차 처리하되, `tqdm`으로 진행률을 표시하면 편리합니다 [34].

10. **질문 전처리 & 프롬프트 구성:**

11. 각 질문별로, **객관식 여부 판별** 함수를 적용합니다. 베이스라인에서는 정규식으로 `^\s*[1-9]` 패턴을 찾아 **선택지 번호가 2개 이상**이면 객관식으로 간주했습니다 [30]. 우리 코드도 동일한 로직의 `is_multiple_choice(question_text: str) -> bool` 함수를 사용합니다.
12. 객관식이면 **질문 본문과 보기 리스트**를 분리합니다. 베이스라인의 `extract_question_and_choices` 함수는 줄을 나누어 숫자로 시작하는 줄은 보기로, 나머지는 질문으로 수집했습니다 [35] [36]. 이 함수를 이용해 `question_text` -> `(question, options_list)`를 얻습니다.
13. 이제 **프롬프트 문자열**을 만듭니다. 4번 섹션에서 설계한 템플릿을 그대로 적용합니다. 의사코드:

```
if is_multiple_choice(q_text):
    q, options = extract_question_and_choices(q_text)
    options_text = "\n".join(options)
    prompt = (
        "당신은 금융보안 전문가입니다.\n"
        "아래 질문에 대해 적절한 **정답 선택지 번호만 출력**하세요.\n\n"
        f"질문: {q}\n"
        "선택지:\n"
        f"{options_text}\n\n"
        "답변:"
    )
else:
    prompt = (
        "당신은 금융보안 전문가입니다.\n"
        "아래 주관식 질문에 대해 정확하고 간략한 설명을 작성하세요.\n\n"
        f"질문: {q_text}\n\n"
        "답변:"
    )
```

    이처럼 문자열 포맷팅으로 프롬프트를 생성합니다 [37] [26]. **주의:** 프롬프트 끝에 `답변:`까지 넣고, 그 다음 자리를 모델이 채우게 합니다.

14. **모델 추론 실행:**

15. 준비된 프롬프트를 모델에 입력하여 답변을 생성합니다. `transformers`의 `pipeline("text-generation", ...)`을 사용할 수 있고, 또는 `model.generate()`를 직접 호출해도 됩니다. 베이스라인은 pipeline을 활용하여 한 줄로 생성했습니다 [38]. Pipeline 사용 시 `device=0` (GPU)로 하고, `max_new_tokens=...`, `temperature=...`, `top_p=...` 등의 **생성 파라미터**를 지정합니다 [38].
16. **생성 파라미터 설정:** 객관식 정답은 한두 글자이므로 `max_new_tokens=5` 정도로 충분하지만, 주관식은 몇 문장까지 나올 수 있어 너무 작게 두면 답이 짤릴 수 있습니다. 베이스라인에서는 둘을 구분하지 않고 일괄적으로 `max_new_tokens=128`을 사용했습니다 [38]. 우리도 안전하게 100~200 사이로 지정하고, 혹시 너무 긴 답변이 나오면 후처리에서 자를 수 있습니다.
    - `temperature`는 **생성의 무작위성**을 조절하는데, 객관식에서는 낮게(모델이 가장 확률 높은 답변을 선택하도록) 설정하는 편입니다. 베이스라인은 `temperature=0.2`로 매우 확정적으로 생성하게 했

습니다 [38]. 주관식의 경우 약간 올려 다양성을 줄 수도 있으나, 평가는 고정 정답 기반이라 높은 창의성은 불필요합니다. 0.2~0.5 사이로 낮게 유지합니다.
    - `top_p=0.8~0.9`로 설정하여 불필요하게 확률 낮은 토큰이 선택되지 않게 제한합니다 [38].
    - `do_sample`는 temperature>0인 경우 기본적으로 True입니다. 객관식의 경우도 temperature 0.2면 약간 샘플링이 있지만 거의 argmax에 가깝습니다. 혹시 완전히 결정적 출력을 원하면 `temperature=0, do_sample=False`로 하면 되지만, 0은 지원 안 할 수도 있어 1e-9 등으로 넣는 편법이 필요합니다. 굳이 그정도는 아니어도 됩니다.

17. 생성 결과로 모델이 완성한 텍스트(예: `"답변: 2"` 혹은 `"답변: ...설명..."`)를 얻습니다. pipeline을 쓰면 리스트[{"generated_text": "..."}]를 반환하므로 그 중 텍스트를 꺼냅니다 [38]. `model.generate`를 쓰면 토큰 시퀀스를 받고 `tokenizer.decode()`로 문자열을 복원합니다.

18. **모델 출력 후처리:**

19. 모델이 출력한 텍스트에서 **실제 답변 부분만 추출**합니다. 우리의 프롬프트 설계상 `답변:` 이후가 모델 생성부인데, 경우에 따라 모델이 `답변:`을 다시 반복하거나 이상한 형식이 섞일 수 있습니다. 이를 정제해야 합니다. 베이스라인의 `extract_answer_only` 함수는 이러한 후처리를 전담합니다 [39].
20. 우선 `generated_text.split("답변:")`로 나눠, **"답변:" 이후의 부분**만 남깁니다 [40]. 없으면 원문 전체를 사용합니다.
21. 결과 문자열을 `.strip()`하여 양끝 공백을 제거하고, 비어있으면 `"미응답"` 등의 기본값을 넣습니다 [41]. (대회 평가에서 무응답은 오답 처리되겠지만, 그래도 공백보다는 명시하는 편이 로그 등에 좋습니다.)
22. **객관식**의 경우, 모델이 숫자만 내놓았는지 확인합니다. 혹시나 모델이 `"정답은 2입니다"`처럼 출력하면 숫자 이외 텍스트를 제거해야 합니다. 정규식으로 첫 번째 등장하는 숫자 패턴을 추출하면 됩니다 [42]. 예: `re.match(r"\D*([1-9][0-9]?)", text)`를 적용해 숫자만 뽑습니다. 찾았으면 해당 숫자 문자열로 답을 대체하고, 못 찾았으면 (모델이 엉뚱하게 문장을 낸 경우) 차라리 원 텍스트 전체나 `"미응답"`을 넣는 전략을 취합니다 [43].
23. **주관식**의 경우엔 특별한 정제는 필요 없습니다. 다만 너무 장황하면 적당히 요약할 수 있으나, 평가가 임베딩 유사도 기반이므로 함부로 줄이는 건 역효과일 수 있습니다. 그래도 문장이 아예 끝나지 않고 잘린 경우 등을 대비해 문장 단위로 끊어줄 수는 있습니다.

24. 후처리 결과 최종 얻은 답변을 변수에 저장합니다 (예: `pred_answer`).

25. **결과 저장:**

26. 모든 테스트 질문에 대해 위 과정을 거쳐 **예측 답변 목록**을 얻으면, 이를 제출 양식에 맞게 저장합니다. 보통 `sample_submission.csv`를 제공하므로 그 포맷에 맞춰 DataFrame을 만들고 to_csv로 출력합니다. 예를 들어:

```
submission = pd.DataFrame({
    "index": test_df["index"],      # 원본 질문 ID
    "answer": predictions_list      # 모델 예측 답변 (번호 또는 텍스트)
})
submission.to_csv("submission.csv", index=False, encoding="utf-8-sig")
```

    UTF-8로 인코딩하고 BOM(Byte Order Mark)을 넣는 것은 Excel 호환 등을 위해 `utf-8-sig` 옵션으로 저장하면 됩니다 [44]. 인덱스 컬럼은 포함하지 않도록 `index=False`를 지정합니다.

27. 출력 CSV의 컬럼명과 형식은 대회 규정에 맞춰야 합니다. 예를 들어 index 없이 오로지 답만 제출하라는 경우도 있으니, 주어진 sample을 참고해야 합니다.

28. **검증 및 종료:**

29. 스크립트 실행 전에, 준비된 `inference.py`를 **로컬 오프라인 환경**에서 테스트해봅니다. 임의로 테스트 질문 몇 개를 넣어 동작을 확인하고, 모델 로드부터 CSV 저장까지 **오류 없이 수행**되는지 검증합니다.
30. 추론 시간도 확인해야 합니다. 4090 GPU 기준으로 샘플당 수 초~수십 초 예상하므로, 테스트셋 크기시간을 계산해 제한 시간 내 끝나는지 봅니다 (규정상 약 샘플당 30초 이내* 권장 [45]). 필요시 `generate`의 `max_new_tokens`나 `num_beams`(beam search시) 등을 조정해 속도를 튜닝합니다. 일반적으로 greedy/샘플링 생성은 빠릅니다.
31. 메모리 사용도 모니터링하여 24GB VRAM 한도를 넘지 않도록 합니다. 7B 모델 4bit + LoRA는 5GB 내외일 것이므로 여유가 있지만, 혹시 모를 메모리 누수를 방지하기 위해 루프 내에서 거대한 변수를 생성하지 않도록 주의합니다.

체크리스트 요약: - [x] **경로**: 코드 내 경로들은 모두 상대경로로 지정했는가? (데이터, 모델 파일 등) - [x] **모델 로드**: 사전학습+LoRA 가중치 로드 코드가 올바른가? 양자화 옵션 및 device 설정 확인. - [x] **프롬프트**: 객관식/주관식 여부 판단 로직과 프롬프트 포맷이 요구사항에 맞는가? (번호만 출력 등) - [x] **생성 파라미터**: max_new_tokens, temperature 등 적절히 설정됐는가? (객관식의 확정적 출력 보장 등) - [x] **후처리**: 모델 출력에서 정답만 떼어내는 처리가 완벽한가? 공백/잘못된 출력 처리, 숫자 추출 등. - [x] **출력 형식**: 제출 파일 컬럼명, 인코딩(UTF-8) 등 규칙 준수 여부 [46] . - [x] **성능 점검**: 로컬에서 샘플 테스트 시 예상 출력이 제대로 나오는가? (예: 객관식 질문 넣어보면 "X" 한 글자만 출력되는지) - [x] **시간/메모리**: 추론 시간은 허용 범위 내인가? VRAM/RAM 사용은 적절한가? (필요시 torch.cuda.empty_cache 등 호출 고려) - [x] **외부의존**: 인터넷 사용이나 허용되지 않은 API 호출이 전혀 없는가? (LangChain 사용 시 내부에서라도 인터넷 호출 요소 없는지 확인)

이상의 사항을 모두 만족하면, `inference.py`는 **클릭 한 번으로 결과 산출**이 가능한 상태가 됩니다. 코드 내에 충분한 **주석**을 달아 평가자가 로직을 쉽게 이해하도록 하는 것도 권장됩니다.

6. 모델 패키징 및 40GB 환경 대응 전략

마지막으로, 완성된 모델과 코드를 대회 제출 요건에 맞게 **패키징**합니다. 오프라인 서버에서의 실행 및 용량 한도를 고려하여 아래와 같은 전략을 적용합니다:

- **모델 가중치 파일 포함:** 추론 코드와 함께 **미세튜닝된 모델 가중치(체크포인트)**를 반드시 제공합니다 [32] . 인터넷이 차단되므로 모델을 사전에 다운로드해 두거나 Hugging Face Hub에서 불러올 수 없습니다. LoRA 사용 시 두 가지 선택이 있습니다:
- **(a) 통합 모델 제공:** 사전학습 모델 + LoRA가 합쳐진 최종 모델의 바이너리 (fp16 또는 bf16) 파일들을 제공. 이 경우 용량이 base 모델과 비슷하게 큽니다 (7B 모델 fp16 약 13GB). 8비트로 저장해도 파일 크기는 fp16과 동일하므로, **디스크 절약 효과는 없습니다**. 다만 로드 후 메모리 효율만 좋아집니다.
- **(b) Base + LoRA 별도 제공:** 용량 문제를 줄이려면 **Base 모델은 대회측이 제공**하는 형태가 이상적이지만, 명시적으로 제공된다는 언급은 없습니다. 따라서 Base 모델도 포함해야 합니다. 대신 **LoRA 어댑터는 용량이 수백 MB 이하**로 작으므로 같이 포함해도 부담이 적습니다. 이 접근의 장점은, Base를 만약 4bit가 아닌 8bit로 양자화해 저장하면 디스크 용량을 절반으로 줄일 수 있습니다. 예컨대 13GB fp16 모델을 `bitsandbytes`로 8bit 양자화하여 `model.bin`을 생성하면 약 6~7GB 정도로 줄일 수 있습니다 (이때 `state_dict`를 quantize하여 저장하는 방법이 필요하며, Hugging Face에서는 4bit 저장은 지원 안하지만 8bit는 가능할 수도 있습니다).

- **(c) GPTQ 등 활용:** 만약 지원된다면, **GPTQ** 방식으로 4비트 정적 양자화한 모델 파일을 생성할 수 있습니다. 이는 파일 크기를 약 4GB대로 낮춰주고, 추론시 추가 라이브러리(gptq-for-llama 등)가 필요하지만, 허용된다면 좋은 방법입니다. 다만 대회 환경에 해당 라이브러리를 설치해야 하고, 검증 리스크가 있으므로 확실치 않다면 무리하지 않는 게 좋습니다.

- **용량 관리:** 전체 제출 패키지 (코드 + 모델)가 40GB 내에 들어와야 합니다 [47] . 7B 모델(fp16 13GB) + LoRA(<<1GB) + 코드/데이터(<1GB)는 충분하지만, 만약 13B 모델을 썼다면 fp16으로 26GB라서 40GB에 여

유가 적습니다. 이 경우 **8bit 양자화 저장** 또는 **모델 체크포인트 파일을 분할**하여 용량을 맞춰야 합니다. HuggingFace Transformers는 보통 큰 모델을 여러 shard로 나눠 저장할 수 있는데 (`save_pretrained`시 `max_shard_size` 옵션), 이를 이용해 각 shard를 10GB 이하로 쪼개둘 수 있습니다. 이는 FAT32 등의 파일 시스템 한계에도 대비됩니다.

- Zip으로 압축할 경우 용량을 줄일 순 있지만, 추론 시 **압축 풀 시간과 용량**을 고려해야 합니다. 대회 안내에 압축에 대한 언급은 없으나, 사전에 압축을 풀고 평가할 가능성이 있습니다. **가급적 압축되지 않은 상태**로 제출하거나, README에 압축 해제 방법을 안내합니다.
- 불필요한 파일은 제거합니다. 예를 들어 학습시 생성된 옵티마이저 상태 (`optimizer.pt`)나 로거 파일 등은 제외합니다. 오직 **모델의 weight 파일과 필요 중간산출물**만 남깁니다.

- **Safetensors 활용:** PyTorch의 `.bin` 대신 **safetensors** 형식으로 모델을 저장하면 로드 속도가 빠르고 보안성이 높습니다. 다만 safetensors는 fp16 기준 사이즈 이점은 크지 않으나, 로드가 메모리 맵 방식이라 40GB 디스크 제한에서 메모리 맵 사용이 문제되지 않는지 확인 필요. 대부분 문제없지만, paranoid하게는 .bin 대비 안전하게 사용할 수 있습니다.

- **환경 요구사항 충족:** 제출 패키지에는 requirements.txt에 명시된 패키지들이 모두 포함되거나, 오프라인 설치가 가능해야 합니다 [48]. 기본 환경(Python 3.10, CUDA 11.8, PyTorch 2.1)은 지원되므로, 추가로 **transformers, peft, accelerate, bitsandbytes, pandas** 등을 요구사항에 넣습니다 [11]. 이때 패키지 버전은 사전에 호환성 테스트한 것으로 고정합니다. 예를 들어:

- transformers == 4.40.1 (베이스라인 기준) [11], 또는 최신 안정버전 4.46 etc.
- peft == 0.4.x (사용한 버전에 따라)
- accelerate == 0.20+ (모델 병렬 로드에 필요)
- bitsandbytes == 0.42.0 (혹은 0.45.2, 실험 시 문제 없었던 버전) [11].

- 기타: numpy, scipy (만약 후처리에 쓰이면), sentencepiece (모델 토크나이저에 필요할 수 있음), safetensors 등.

- **오프라인 동작 검증:** 제출 전에 **인터넷 연결 없이** 가상 머신 등에서 패키지를 풀어 같은 환경에서 돌려봅니다. 이때 requirements 설치도 미리 해봐서, 파이썬 패키지 간 충돌이나 누락이 없는지 검사합니다. 인터넷 없이 pip 설치가 가능하려면 **파이휀휀휀** (아마 내부 미러나 캐시)로 설치될 것이므로, 버전 매칭이 중요합니다. 혹은 `whl` 파일을 함께 제공하는 방법도 있지만, 일반적으로 요구사항만으로 해결될 것입니다.

- **결과보고서 및 기타 파일:** 코드 외에 결과보고서(PDF)도 제출 요구사항이므로, README에 준하는 내용을 별도 문서로도 정리합니다. 코드와 모델이 잘 작동해도 문서 미비로 감점되지 않도록, **보고서에 사용 모델, 데이터, 기법, 결과를 상세히 기술**합니다.

종합하면, **모델 경량화 + 패키지 구성 최적화**를 통해 40GB 제한 내에서 원활히 추론되도록 합니다. 7B 모델 기준으로는 큰 어려움이 없겠지만, 그래도 디스크 용량과 메모리를 한번 더 점검하고 제출하는 것이 안전합니다. 이러한 준비를 마치면, 오프라인 평가 환경에서 곧바로 코드를 실행해 **Private 테스트 데이터를 복원**할 수 있을 것입니다 [49].

본 보고서는 **한국어 금융 AI 해커톤** 참가를 위한 LLM 기반 시스템 개발의 전 과정을 다루었습니다. 요약하면, 공개 한국어 LLM을 LoRA로 도메인 특화 미세튜닝하고, 구조적인 프롬프트 설계 및 효율적 추론 파이프라인을 구축하여, **단일 LLM으로 금융보안 객관식·주관식 문항 모두에 답변**하는 모델을 완성하였습니다 [2]. 제안된 폴더 구조와 코드 설계, 하이퍼파라미터 선택 근거를 참고하여 구현하면, 대회 요구사항을 충족하는 결과를 얻을 수 있을 것으로 기대합니다. 시스템 개발 후에는 충분한 테스트와 규칙 준수 점검을 거쳐, **안정적이고 재현 가능한 최종 패키지**를 제출하시기 바랍니다. 성공적인 해커톤 완주를 기원합니다!

---

1 beomi/gemma-ko-7b · Hugging Face
https://huggingface.co/beomi/gemma-ko-7b

2 6 7 9 10 13 14 15 16 17 18 28 29 31 32 45 46 47 48 49 2025_금융_AI_Challenge__금융_AI_모델_경쟁.pdf
file://file-3wLQyTiTWDMDNUrwqaNEFM

3 4 5 19 20 21 22 Fine-Tuning Your First Large Language Model (LLM) with PyTorch and Hugging Face
https://huggingface.co/blog/dvgodoy/fine-tuning-llm-hugging-face

8 24 25 26 27 30 33 34 35 36 37 38 39 40 41 42 43 44 [Baseline]_gemma-ko-7b 기반 금융보안 텍스트 생성 AI 모델.ipynb
file://file-EPzrNczUXHbLSN1C2T76Hw

11 requirements.txt
file://file-TCGDBqBraH9NTeVYT8hn4P

12 setup.sh
file://file-9dTnauYp5mT22tysWB74gW

23 lemon-mint/gemma-ko-7b-instruct-v0.50 · Hugging Face
https://huggingface.co/lemon-mint/gemma-ko-7b-instruct-v0.50