모델 정의서

고객 음성 상담 (AICC) Project

팀명 : AIII (아르르)
소속 : LG U+ WHY NOT SW CAMP
팀장 : 박준현
팀원 : 김다란솔, 이현행, 임기빈, 신종현, 정인엽

버전 : v2

목 차

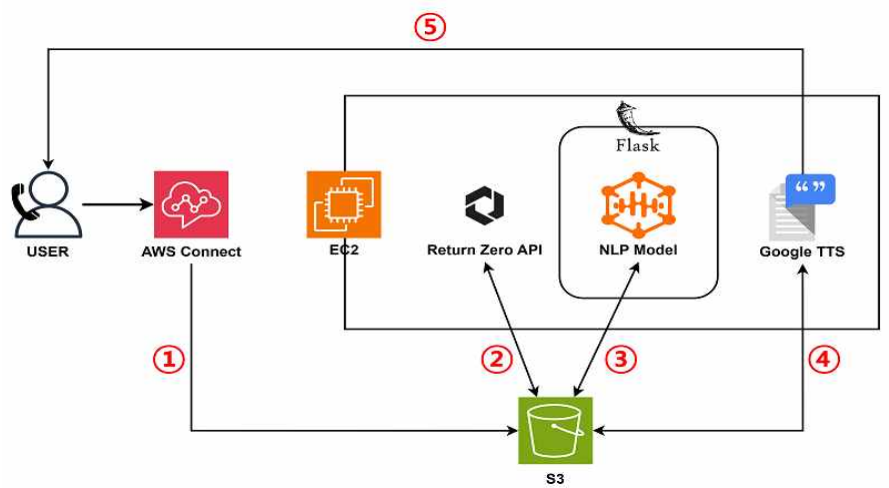
Ⅰ. 시스템 구성

1. 개요

- 주제 : 정확한 고객 요구 요약을 통한 상담의 효율성 증대 기능 개발
- 목표
  - ・불필요한 절차를 줄이고 고객의 요구를 요약하여 되물음으로써 신속 정확한 상담 운영을 목표로 함
  - ・상담 의도 파악의 정확도를 높이고 상담 시간을 단축하여 고객의 만족도 증대를 기대할 수 있음

2. 구조

(1) 아키텍처 및 전체 흐름



① 사용자 발화 수집

- 사용자가 전화로 문의 내용을 말하면 AWS Connect가 음성 데이터를 녹음하여 S3 /connect 폴더에 저장

② STT 처리 후 텍스트 생성

- EC2 내에 리턴제로 API를 호출하여 음성을 텍스트로 변환하고 결과를 S3 /stt 폴더에 저장

③ NLP 요약 처리

- EC2 서버에서 Flask화 되어있는 모델을 가져와 요약하여 S3 /nlp 폴더에 저장

④ TTS 처리 후 음성 생성

- Google TTS를 호출하여 요약 텍스트를 음성 변환하고 결과를 S3 /tts 폴더에 저장

⑤ 사용자에게 제공

(2) 주요 구성 요소

① AWS Connect

- 역할
  - 사용자가 음성 ARS를 통해 발화하는 내용을 수집하는 초기 접점
  - 통화 종료 시 수집된 음성 데이터를 후속 프로세스에서 사용
- 기술적 구성
  - AWS Connect의 전화 처리 기능을 활용하여 음성을 녹음 및 저장
  - 음성 파일(.wav)을 자동으로 S3에 업로드
  - 업로드 직후 이벤트 확인 후 EC2 서버에서 STT 작업 수행

② AWS S3 (스토리지)

- 역할
  - 서비스 전반에서 생성된 음성 파일(.wav), STT 텍스트 데이터(.txt), 요약 결과(.txt), TTS 음성 파일(.wav)을 저장
- 구성 방식
  - S3 버킷 이름 : aicc-alll
  - 폴더 구성 (폴더명)
    - /connect : 통화 종료 시 "connect인스턴스이름/CallRecordings/월/일" 폴더에 .wav 파일 자동 생성.
    - /stt : connect 폴더에 들어온 음성 파일을 텍스트 변환 후 .txt 파일로 저장 (파일명 : stt_숫자.txt)
    - /nlp : STT로 만들어진 .txt 문장을 모델 적용 후 요약 문장을 .txt 파일로 저장 (파일명 : nlp_숫자.txt)
    - /tts : 요약 .txt 문장을 Google TTS 적용 후 나온 음성을 .wav 파일로 저장 (파일명 : tts_숫자.wav)

③ EC2 Server

- 역할
  - STT, NLP, TTS 실행을 담당하는 중앙 처리 서버
- 흐름
  - S3의 /connect 폴더에 저장된 음성을 리턴제로 STT API에 전달하여 텍스트(.txt)를 생성

- S3의 /stt 폴더에 저장된 텍스트 데이터를 Flask화된 NLP 요약 모델을 포함한 코드로 가져와 요약하여 /nlp 폴더에 .txt파일로 저장
- S3의 /nlp 폴더의 요약 텍스트를 Google TTS API에 전달하여 음성(.wav)을 생성
- 생성된 음성 데이터를 S3 /tts 폴더에 저장

- 구성 및 운영 방식
  - NLP 모델 : EC2 내에서 Fine-Tuned KoBART 또는 Llama 3.2 모델을 실행
  - EC2 인스턴스는 유형은 GPU 기반 g4dn.xlarge 활용

④ 리턴제로 STT

- 역할
  - AWS Connect의 음성을 텍스트로 변환
- 특징
  - STT 결과를 S3 /stt 폴더에 저장하여 재사용 가능
- 운영 방식
  - EC2 서버에서 비동기 방식으로 호출하여 빠른 응답을 제공

⑤ Flask 내 NLP 요약 모델

- EC2 서버 내에 KoBART 및 Llama 3.2 모델 호출로 사용자의 발화를 요약
- 두 가지 모델의 성능 비교 후 적합한 결과를 제공
  - 성능 비교 : BERT-Score과 STS 지표 활용

⑥ Google TTS

- 역할
  - NLP 모델에서 생성된 요약 텍스트를 자연스러운 음성으로 변환
- 특징
  - TTS 결과를 S3 /tts 폴더에 저장하여 재사용 가능
- 운영 방식
  - EC2 서버에서 비동기 방식으로 호출하여 빠른 응답을 제공

3. AWS 환경 구성

① AWS Connect

- 클라우드 기반의 1)옴니채널 연락처 센터 서비스로 음성 데이터를 효율적으로 수집하고 관리 가능
  - 확장성
    - AWS 클라우드 인프라를 활용하여 시스템의 확장성을 극대화
    - 트래픽 급증 시 자동으로 리소스를 확장하여 대규모 사용자 대응이 가능
  - 비용 효율성
    - 종량제 과금 방식을 지원하여 초기 투자 비용이 낮음
    - 사용량에 따라 비용이 책정되기 때문에, 비효율적인 비용 낭비를 줄이고 운영 비용을 절감 가능
  - 자동화
    - 음성 데이터 녹음 및 S3 자동 저장을 통해 후속 프로세스와의 연계가 용이

② AWS S3 (스토리지)

- 저비용의 고내구성 객체 스토리지로 데이터를 안전하고 효율적으로 저장
  - 내구성
    - 99.99%의 데이터 내구성을 제공
  - 저비용
    - GB당 낮은 비용으로 비용 절감
  - 통합성
    - AWS EC2 및 다양한 서비스와 네이티브로 통합 가능

③ EC2 Server (g4dn.xlarge)

- GPU가 탑재된 고성능 EC2 인스턴스로, NLP 모델 학습 및 실행에 최적화
  - 성능
    - NVIDIA T4 GPU 및 4 vCPU로 높은 연산 능력 제공
  - 유연성
    - EC2 인스턴스는 필요에 따라 확장 및 축소가 가능하여, 트래픽이 증가하는 시간대에는 리소스를 확장하고, 불필요한 리소스를 절감 가능

1) 고객이 다양한 채널(온라인, 오프라인, 모바일 등)을 넘나들며 일관된 경험을 할 수 있도록 제공하는 접근 방식

Ⅱ. 모델

1. STT (Speech-to-Text)

(1) STT API 설명

① 사용 API

- 리턴제로 STT API

② 역할

- 사용자 음성 데이터를 실시간 또는 비동기 방식으로 텍스트로 변환

③ 기술적 구성

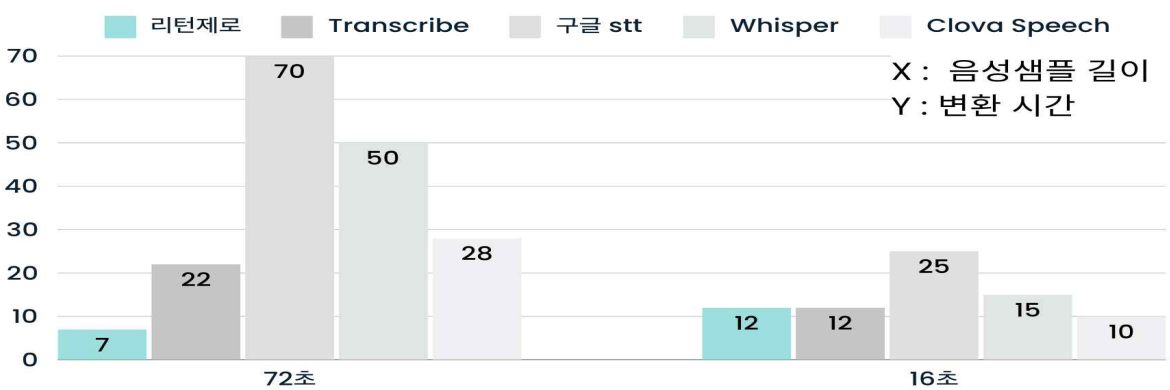
- 음성을 오디오 신호 처리 및 특징 벡터(피처) 추출한 후, RNN-T와 CTC 구조를 활용하여 음성을 텍스트로 변환
- 음성의 음향 모델(Acoustic Model)과 언어 모델(Language Model)을 결합하여 높은 인식률을 보장

(2) 특징 및 선정 이유

- .wav 파일 형식 지원
- 빠른 처리 속도
- 한국어 오인식률 최저

항목	한국어 오인식률	지원 파일 형식
리턴제로	**6.59%**	mp4, m4a, mp3, amr, flac, **wav**
Transcribe	12.34%	amr, flac, m4a, mp3, mp4, Ogg, webm wav
구글 STT	11.50%	wav, mp3, flac, Ogg, webm
Whisper	11.39%	mp3, mp4, mpeg, mpga, m4a, wav, webm
Clova Speech	9.09%	mp3, aac, ac3, Ogg, flac, wav, m4a

*** 출처 : 각 api 제공 사이트**



*** 직접 테스트 실시 결과**

(3) 활용 방안

① API 사용 흐름

- 음성 파일 업로드 (S3)
- 리턴제로 STT API 호출을 통해 음성 데이터를 텍스트로 변환
- 변환된 텍스트를 S3에 저장

② 실행 환경 및 요구사항

- Python 3.9 이상
- requests, boto3 라이브러리 설치
  - pip install requests boto3
- AWS 인증: S3 접근 권한 필요
  - IAM 역할에 S3 Access 정책 설정
- 리턴제로 STT API 인증: 클라이언트 ID와 시크릿 키 발급

③ 코드

- 핵심 함수 및 주요 로직
  - authenticate_stt_api() : STT API 호출을 위해 인증 토큰 발급
  - process_audio_file() : S3 파일 처리 및 변환 후 S3에 저장
  - monitor_s3_bucket() : S3 버킷 감지 및 새 파일 처리
- 보조 함수
  - get_next_file_number() : S3에 저장될 파일 번호 계산
- 예시 코드 (stt.py)

```
import os
import json
import requests
import boto3
import time

# S3 클라이언트 초기화
s3_client = boto3.client('s3')

# S3 버킷 및 폴더 설정
BUCKET_NAME = "BUCKET_NAME 입력"
INPUT_FOLDER = "INPUT_FOLDER 경로 입력"
OUTPUT_FOLDER = "OUTPUT_FOLDER 경로 입력"
```

```
# STT API 설정
STT_CONFIG = {
    "use_diarization": True,
    "diarization": {"spk_count": 2},
    "use_itn": True,
}

def authenticate_stt_api():
    """STT API 인증 토큰 발급"""
    auth_resp = requests.post(
        'https://openapi.vito.ai/v1/authenticate',
        data={'client_id': "CLIENT_ID", 'client_secret': "CLIENT_SECRET"}
    )
    auth_resp.raise_for_status()
    return auth_resp.json().get('access_token')

def process_audio_file(bucket_name, object_key, token):
    """S3에서 음성 파일 처리 후 텍스트 변환 및 저장"""
    local_audio_file = f"/tmp/{os.path.basename(object_key)}"
    local_text_file = f"/tmp/{os.path.basename(object_key).replace('.wav', '.txt')}"

    # S3에서 음성 파일 다운로드
    s3_client.download_file(bucket_name, object_key, local_audio_file)

    # STT API 호출
    with open(local_audio_file, 'rb') as audio_file:
        stt_resp = requests.post(
            'https://openapi.vito.ai/v1/transcribe',
            headers={'Authorization': f'bearer {token}'},
            data={'config': json.dumps(STT_CONFIG)},
            files={'file': audio_file}
        )
        stt_resp.raise_for_status()

    # 결과 저장
    utterances = stt_resp.json().get("results", {}).get("utterances", [])
    text_output = "\n".join([
        utterance['msg']
        for utterance in utterances
    ])
    # 예제에서 고정값 사용, 실제 구현 시 get_next_file_number 활용
    next_file_number = 1
    output_key = f"{OUTPUT_FOLDER}stt_{next_file_number}.txt"
```

```
    with open(local_text_file, 'w', encoding='utf-8') as file:
        file.write(text_output)
    s3_client.upload_file(local_text_file, bucket_name, output_key)

def monitor_s3_bucket():
    """S3 버킷 모니터링"""
    token = authenticate_stt_api()
    while True:
        response = s3_client.list_objects_v2(
        Bucket=BUCKET_NAME, Prefix=INPUT_FOLDER
    )
    if 'Contents' in response:
        for obj in response['Contents']:
            if obj['Key'].endswith(".wav"):
                process_audio_file(BUCKET_NAME, obj['Key'], token)
    time.sleep(1)

if __name__ == "__main__":
    monitor_s3_bucket()
```

④ 입력 및 출력

- 입력
  - S3의 INPUT_FOLDER에 저장된 .wav 파일
- 출력
  - 텍스트 파일이 OUTPUT_FOLDER에 stt_1.txt 형식으로 저장

2. NLP 요약 모델 (Text Summarization)

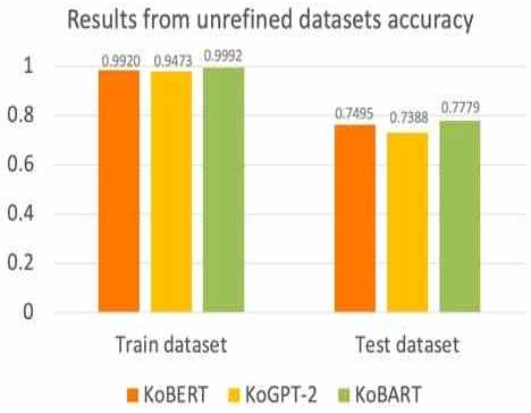
(1) 모델 설명 및 선정 이유

① 역할

- STT를 통해 얻은 텍스트 데이터를 핵심만 요약하여 간결한 내용을 도출

② 사용 모델

- KoBART
  - SK텔레콤의 AI 연구팀에서 개발한 한국어 특화 언어 모델
  - 한국어 특화된 [2]BART(Bidirectional and Auto-Regressive Transformer) 모델로, 텍스트 요약, 생성, 질의응답 등 다양한 NLP 작업에 적합
  - 구성
    - Encoder : 문맥 정보를 양방향으로 학습.
    - Decoder : 문맥 정보를 기반으로 적합한 요약 문장을 생성.
  - 주요 알고리즘
    - Transformer 구조로 [3]Self-Attention 메커니즘과 병렬 연산을 활용
    - Pretrained 모델(한국어 뉴스, SNS 데이터 기반)에 보험 관련 데이터로 Fine-Tuning
  - 특징 및 선정 이유
    - 다양한 어투(정중/화난/반어법)와 긴 문장에 대한 높은 적응력을 보임
    - 정형화된 데이터를 잘 처리하며 반말/존댓말 등을 포함한 한국어 특성에 강함



*** 출처 : KoBERT, KoGPT-2, KoBART 활용 및 하이퍼파라미터 최적화를 진행한 리뷰 감성분석 애플리케이션 구현 (논문)**

Models	Rouge-1	Rouge-2	Rouge-L
TextRank	0.542	0.39	0.276
KoBertSum - Legal	0.484	0.33	0.334
KoBertSum – News Large	0.472	0.36	0.366
KoBART - Base	0.534	0.42	0.446
KoBART – Legal	0.544	0.44	0.464
KoBART – News Large	0.57	0.47	0.508

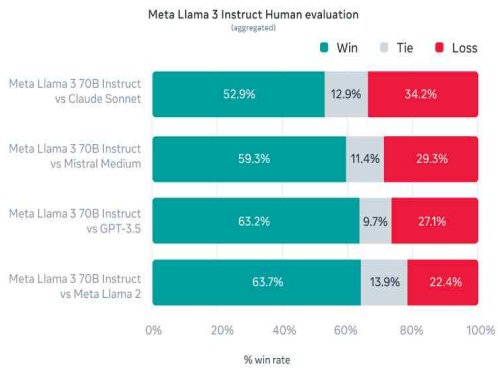
*** 출처 : 사전학습 기반의 법률문서 요약 방법 비교연구 (논문)**

2) 양방향 및 자기 회귀 변환 : 출력 텍스트를 양방향으로 생성할 수 있으며, 왼쪽에서 오른쪽으로, 오른쪽에서 왼쪽으로의 맥락을 모두 통합할 수 있음
3) 문장의 각 단어가 다른 단어들과의 유사도(연관성)를 계산해 전체 문맥을 고려한 표현을 학습하는 방식

- Llama-3.2-Korean-GGACHI-1B-Instruct-v1
  - LLaMA (Large Language Model Meta AI)의 한국어 변형 모델
  - BERT 기반의 Encoder-Decoder 아키텍처와 GPT 기반의 생성 능력을 결합한 하이브리드 구조
  - 구성
    - Large-scale Transformer 아키텍처로 10억 개의 매개변수(Parameter) 사용
    - 한국어 데이터를 대규모로 학습하여 방언, 비속어 등 다양한 텍스트를 처리 가능
  - 주요 알고리즘
    - 4)Attention Mechanism과 MLM(Masked Language Modeling) 기법을 통해 입력 데이터를 효율적으로 요약
    - 5)Seq2Seq (Sequence-to-Sequence) 학습 방식 사용
  - 특징 및 선정 이유
    - 반어법, 전문가적 어투와 같은 비일상적 텍스트 처리에서 강점
    - 프롬포트를 활용한 요약 결과를 생성 가능

Category / Benchmark	Llama 3.1 70B	Llama 3.3 70B	Amazon Nova Pro	Llama 3.1 405B	Gemini Pro 1.5	GPT-4o	Claude 3.5 Sonnet
General MMLU Chat (0-shot, CoT)	86.0	86.0	85.9	88.6	87.1	87.5	88.9
MMLU PRO (5-shot, CoT)	66.4	68.9	-	73.4	76.1	73.8	77.8
Instruction Following IFEval	87.5	92.1	92.1	88.6	91.9	84.6	89.3
Code HumanEval (0-shot)	80.5	88.4	89.0	89.0	89.0	86.0	93.7
MBPP EvalPlus (base) (0-shot)	86.0	87.6	-	88.6	87.8	83.9	86.8
Math MATH (0-shot, CoT)	67.8	77.0	76.6	73.9	82.9	76.9	78.3
Reasoning GPQA Diamond (0-shot, CoT)	48.0	50.5	-	49.0	53.5	47.5	65.0
Tool use BFCL v2 (0-shot)	77.5	77.3	-	81.1	80.3	74.0	79.3
Long context NIH/Multi-needle	97.5	97.5	-	98.1	94.7	-	99.4
Multilingual Multilingual MGSM (0-shot)	86.9	91.1	-	91.6	89.6	90.6	92.8
Pricing* 1M Input tokens (Cheapest among providers)*	$0.1	$0.1	$0.80	$1.0	$1.30	2.5$	$3.0
1M Output tokens (Cheapest among providers)*	$0.4	$0.4	$3.20	$1.8	$5.0	10.0$	$15.0

* 출처 : 라마 공식 홈페이지



* 출처 : 라마 공식 홈페이지

③ 최적화

- KoBART와 Llama 모델을 각각 가지고 있는 데이터로 Fine-Tuning 후 BERT-Score와 Semantic Textual Similarity (STS)로 성능 평가

4) 문장 속 모든 단어에 대해 집중해야 할 정도를 계산하고 0과 1 사이의 숫자로 표현
5) 한 문장(시퀀스)을 다른 문장(시퀀스)으로 변환하는 모델을 의미, 기계 번역, 챗봇, 요약 등 시퀀스 데이터를 입력으로 받아 다른 시퀀스 데이터를 출력하는 데 사용

(2) 모델 학습 데이터 설명

① 사용된 데이터셋

- 데이터셋 규모
  - 총 데이터셋 수 : 약 12,000개
- 데이터셋 유형
  - 직접 생성한 보험 관련 문의 (GPT 사용하여 생성)

② 데이터셋 구성

- 각 데이터는 2개의 칼럼으로 구성
  - Input : 사용자의 문의 텍스트
  - Summary : 문의를 요약한 텍스트

③ 데이터 특징

- 언어 스타일
  - '반말/존댓말', '평서문/의문문'과 같은 다양한 어투를 포함하여 실제 고객 발화를 반영
- 발화 길이
  - 짧은 발화 (1~2문장)와 긴 발화 (3~5문장)가 혼합.
- 토큰화 시 길이

KoBART		Llama 3.2	
Input	Summary	Input	Summary
제한 없음		1024	1024

- 감정 표현
  - '화난/정중한'과 같은 감정 상태를 반영한 데이터
- 특이 case
  - 반어법 (부정적이지만 긍정적인 척하는) 시나리오
  - 전문가인 척 아는 척하는 시나리오

④ 데이터 형태

- 파일 확장자 : .csv
- 파일 이름 : data_개수.csv
- 2개의 컬럼 (input, summary)으로, 데이터는 기본적으로 다음과 같은 형태로 구성
  - input : 상대 차가 뒤에서 박는 사고가 났어. 내 보험 적용 여부를 알고 싶어.
  - summary : 자동차 사고 보험 여부 확인 요청

(3) 학습 과정

① 데이터 분할

- 분할 비율 (전체 데이터 중 ~%)
  - Training: 90%
  - Test: 10%
- Validation
  - Training 데이터 中 90% + 새로운 데이터 10%
    - 새로운 데이터는 보험사 유형 제외 데이터

② 분할 방식

- 파이썬 코드 내 랜덤 샘플링 사용

③ 사용 모델 및 하이퍼파타미터 설정

항목	KoBART	Llama 3.2
토크나이저	PreTrainedTokenizerFast	AutoTokenizer
학습률 (Learning Rate)	3e-4, 5e-5	3e-4, 5e-5
Batch Size	2	2
Gradient Accumulation	-	2
최대 입력 길이	-	1024
최대 출력 길이	-	1024
Optimizer	AdamW	AdamW
Epoch	5, 10, 20	5, 10, 20
Weight Decay	0.01	0.01
평가 주기	500	500

*** 하이퍼파라미터 설정**

④ 학습 방식

- 지도 학습 (Supervised Learning)
  - Transformer 기반의 Seq2Seq (Sequence-to-Sequence) 모델
    - 입력 데이터 : 원본 텍스트(문장 또는 문단)
    - 목표 데이터 : 모델이 입력 문장을 바탕으로 정확한 요약 문장을 생성하도록 학습
- QLoRA (Quantized Low-Rank Adaptation)
  - [6]4-bit 양자화와 LoRA를 결합하여 대형 모델의 메모리 사용량과 학습 비용을 획기적으로 줄이는 효율적 미세조정 기법
  - QLoRA 설정
    - r(가중치 행렬의 rank) : 8, 16, 32

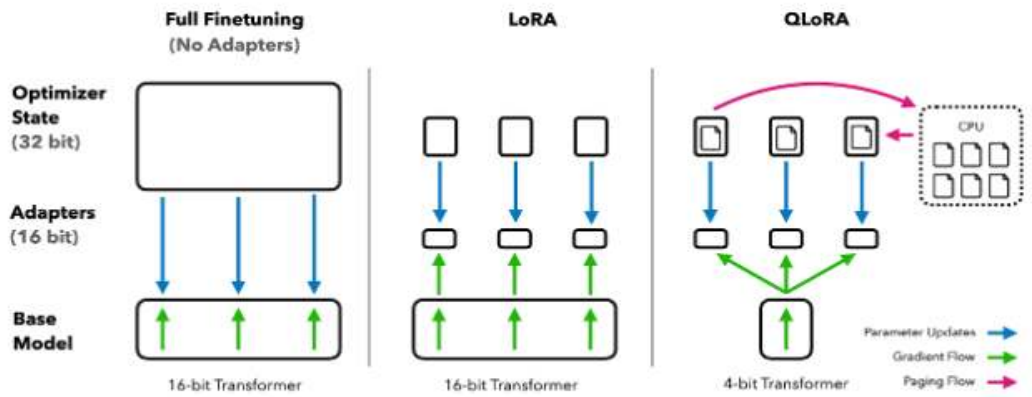


Figure 1: Different finetuning methods and their memory requirements. QLoRA improves over LoRA by quantizing the transformer model to 4-bit precision and using paged optimizers to handle memory spikes.

*** 출처 : QLoRA: Efficient Finetuning of Quantized LLM (논문)**

- 손실 함수
  - 순전파와 역전파를 활용한 [7]CrossEntropyLoss
    - 순전파 : 입력 데이터를 토크나이징하여 입력 시퀀스로 변환한 후 모델에 전달
      모델은 입력 시퀀스를 기반으로 예측 logits (단어 분포)을 생성하고,
      디코더를 통해 다음 토큰을 예측
      생성된 예측 logits과 정답 labels를 비교하여 CrossEntropyLoss를 계산
    - 역전파 : [8]Loss.backward()를 통해 손실 값을 기준으로 기울기(gradient)를 계산
      옵티마이저를 사용해 모델의 가중치를 업데이트
  - 모델의 예측과 실제 정답 간의 차이를 측정하고 이를 줄여나가도록 학습

6) 모델의 가중치나 활성화값을 4비트로 압축하여 메모리 사용량을 줄이고 계산 효율성을 향상
7) 분류 문제에서 사용되는 손실 함수, 함수는 모델의 예측 확률과 실제 레이블 간의 차이를 측정
8) 손실 함수의 기울기를 계산하여, 각 파라미터에 대한 손실의 기여도를 구함, 역전파 알고리즘을 기반으로 체인 룰을 이용해 신경망의 모든 가중치에 대한 기울기를 계산

- 검증
  - 500 step마다 검증 데이터를 사용해 모델 성능 점검
  - 손실값과 유사도 점수를 통해 모델의 요약 품질 평가
  - 검증 성능이 개선되면 해당 가중치를 기록, 최종모델로 활용
  - x축 step, y축 점수로 시각화
    - ▸ Training Loss and Validation Loss : 학습과 검증 손실 값의 변화 추이 확인
    - ▸ BERT-Score and STS Score : 모델의 요약 성능을 정량적으로 평가 후 학습 개선 여부 확인

하드웨어	Window 10 Pro 64bit	**CPU**	13th Gen Intel(R) Core(TM) i7-13620H (16 CPUs), ~2.4GHz
		GPU	RTX 4060 Laptop GPU(8GB)
소프트웨어	라이브러리: Huggingface Transformers, PyTorch		
	AWS CLI 및 Boto3를 활용하여 S3와의 연동 자동화		

*** 학습 환경**

(4) 평가 지표

① BERT-Score

- 요약 텍스트와 9)Ground Truth 간의 의미적 유사도를 측정

② Semantic Textual Similarity (STS)

- 요약 텍스트와 Ground Truth 간의 문장 의미 유사도를 점수화

(5) 성능 검증 및 모델 선택

① 두 모델 (KoBART와 Llama-3.2)의 하이퍼마라미터를 조정하며 BERT-Score과 STS 상위 4개를 후보 모델로 선정 후, 직접 테스트를 통해 최종 모델 선정

② 성능 비교

- BERT-Score와 STS의 평균
- 직접 테스트를 통한 팀원 전체의 투표
  - 1~2문장의 짧은 문장, 3문장 이상인 긴 문장을 직접 입력
  - 팀원 전체가 평가하여 1점부터 5점 만점으로 점수를 매김
  - 가장 높은 점수를 받은 문장을 최종적으로 선택

9) AI 모델 출력값을 훈련 및 테스트하는 데 사용되는 실제 환경의 데이터

(6) AWS EC2에서 Flask를 이용한 NLP 모델 배포

① 배포 환경 및 요구사항

- 플랫폼
  - AWS EC2 (Amazon Linux 2023 AMI)
- Python 3.9 이상
- flask, boto3, transformers 라이브러리 설치
  - pip install flask boto3 transformers
- AWS 인증: CLI 및 S3 접근 권한 필요
  - aws configure
  - IAM 역할에 S3 Access 정책 설정
- 모델 업로드
  - Fine-Tuning된 KoBART 또는 Llama 모델을 EC2에 업로드
- Flask 서버 코드 배포 및 실행
  - Flask 코드를 EC2에 업로드하거나 직접 작성
  - Flask 서버 실행
    - python app.py

② 코드

- KoBART 학습 코드
  - 핵심 함수 및 주요 로직
    - KoBART 모델 로드 및 QLoRA 적용
    - train_and_validate() : 모델을 미세 조정하며, 정기적으로 검증 데이터를 사용해 성능 평가
    - save_model() : 미세 조정된 모델과 토크나이저를 저장
  - 보조 함수
    - preprocess_data() : 데이터셋 로드 및 전처리
    - validate_model() : 모델 검증 및 평가

· 모델 학습 코드

```
from transformers import BartForConditionalGeneration
from transformers import PreTrainedTokenizerFast
from peft import LoraConfig, get_peft_model
from transformers import AdamW
from torch.utils.data import DataLoader, random_split
import torch
import pandas as pd
from bert_score import score
from sentence_transformers import SentenceTransformer, util

# KoBART 모델과 토크나이저 로드
model_name = "hyunwoongko/kobart"
model = BartForConditionalGeneration.from_pretrained(model_name)
tokenizer = PreTrainedTokenizerFast.from_pretrained(model_name)

# QLoRA 설정 적용
lora_config = LoraConfig(
    r=8,
    lora_alpha=32,
    target_modules=["q_proj", "v_proj"],
    lora_dropout=0.1,
    bias="none",
    task_type="SEQ_2_SEQ_LM",
)
lora_model = get_peft_model(model, lora_config)
lora_model = lora_model.to("cuda")

# 데이터 전처리 함수
def preprocess_data(data_df, tokenizer):
    inputs = tokenizer(
        list(data_df["input"]),
        truncation=True,
        padding=True,
        return_tensors="pt")
    labels = tokenizer(
        list(data_df["summary"]),
        truncation=True,
        padding=True,
        return_tensors="pt“
    )["input_ids"]
    return list(zip(inputs["input_ids"], labels))

# 데이터셋 로드 및 조합
file_path = '데이터셋 파일.csv'
validation_file_path = '추가 검증 데이터셋 파일.csv'

data = pd.read_csv(file_path, encoding='euc-kr')
```

```
validation_data = pd.read_csv(validation_file_path, encoding='euc-kr')

# 데이터셋 분할
train_size = int(len(data) * 0.9)
test_size = len(data) - train_size

train_data_indices = random_split(range(len(data)), [train_size, test_size])[0]
test_data_indices = random_split(range(len(data)), [train_size, test_size])[1]

# 검증 데이터 조합
validation_data_limit = 1200
validation_data_size = int(validation_data_limit * 0.1)
train_data_size_for_validation = validation_data_limit - validation_data_size

validation_data_part = validation_data[:validation_data_size]
train_data_part_for_validation_indices = list(
    train_data_indices
)[:train_data_size_for_validation]
remaining_train_indices = list(train_data_indices)[train_data_size_for_validation:]

train_data_part_for_validation = data.iloc[train_data_part_for_validation_indices]
remaining_train_data = data.iloc[remaining_train_indices]
validation_set = pd.concat([validation_data_part, train_data_part_for_validation])

# 데이터 전처리
train_dataset = preprocess_data(remaining_train_data, tokenizer)
test_dataset = preprocess_data(data.iloc[list(test_data_indices)], tokenizer)
validation_dataset = preprocess_data(validation_set, tokenizer)

# DataLoader 생성
dataloader_train = DataLoader(train_dataset, batch_size=2, shuffle=True)
dataloader_test = DataLoader(test_dataset, batch_size=2)
dataloader_validation = DataLoader(validation_dataset, batch_size=2)

# 학습 설정
optimizer = AdamW(lora_model.parameters(), lr=3e-4)

# 학습 루프
lora_model.train()
epochs = 10
for epoch in range(epochs):
    for batch in dataloader_train:
        input_ids, labels = batch
        input_ids, labels = input_ids.cuda(), labels.cuda()

        # 순전파 및 손실 계산
        outputs = lora_model(input_ids=input_ids, labels=labels)
        loss = outputs.loss
```

```
        # 역전파 및 가중치 업데이트
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

# 검증 함수
def validate_model(lora_model, dataloader_validation, tokenizer):
    lora_model.eval()
    preds, refs = [], []
    with torch.no_grad():
        for batch in dataloader_validation:
            input_ids, labels = batch
            input_ids, labels = input_ids.cuda(), labels.cuda()

            outputs = lora_model.generate(input_ids=input_ids)
            decoded_preds = tokenizer.batch_decode(
                outputs, skip_special_tokens=True
            )
            decoded_labels = tokenizer.batch_decode(
                labels, skip_special_tokens=True
            )
            preds.extend(decoded_preds)
            refs.extend(decoded_labels)

    # 성능 평가
    P, R, F1 = score(preds, refs, lang="ko", verbose=True)
    print(f"BERTScore - F1: {F1.mean().item():.4f}")
    sts_model = SentenceTransformer(
        'sentence-transformers/paraphrase-multilingual-MiniLM-L12-v2'
    )
    pred_embeddings = sts_model.encode(preds, convert_to_tensor=True)
    ref_embeddings = sts_model.encode(refs, convert_to_tensor=True)
    cosine_scores = util.pytorch_cos_sim(pred_embeddings, ref_embeddings)
    avg_sts_score = torch.mean(torch.diag(cosine_scores)).item()
    print(f"STS Score: {avg_sts_score:.4f}")

# 검증 수행
validate_model(lora_model, dataloader_validation, tokenizer)

# 모델 저장
save_path = "kobart_qlora_finetuned"
lora_model.save_pretrained(save_path)
tokenizer.save_pretrained(save_path)
print(f"Model saved to {save_path}")
```

- Flask 코드

▪ 핵심 함수 및 주요 로직

▸ process_file() : S3 파일 요약 후 S3에 저장

▸ monitor_s3() : S3 버킷 감지 및 새 파일 처리

▪ 예시 코드 (app.py)

```
from flask import Flask, jsonify
import boto3
from transformers import BartForConditionalGeneration
from transformers import PreTrainedTokenizerFast
import threading
import time
import os

app = Flask(__name__)

# S3 설정
s3 = boto3.client('s3')
bucket_name = "버킷 이름 입력"
input_folder = "INPUT_FOLDER 경로 입력"
summary_folder = "OUTPUT_FOLDER 경로 입력"

# 모델 로드
model_dir = "모델 폴더 이름"
model = BartForConditionalGeneration.from_pretrained(model_dir)
tokenizer = PreTrainedTokenizerFast.from_pretrained(model_dir)

# 이미 처리된 파일 리스트를 저장할 Set
processed_files = set()
def process_file(file_key):
    """
    S3에서 파일 다운로드, 요약 처리, 결과 업로드
    """
    try:
        # 파일 다운로드
        original_filename = file_key.split('/')[-1]  # 원래 파일 이름
        input_path = f"/tmp/{original_filename}"
        s3.download_file(bucket_name, file_key, input_path)

        # 모델 처리
        with open(input_path, 'r', encoding='utf-8') as f:
            input_text = f.read()

        inputs = tokenizer(
            input_text,
            return_tensors="pt",
```

```
            truncation=True,
            padding=True,
            max_length=1024
        )
        summary_ids = model.generate(
            inputs["input_ids"],
            max_length=128,
            num_beams=4,
            early_stopping=Tru
        )
        summary_text = tokenizer.decode(
            summary_ids[0],
            skip_special_tokens=True
        )

        # 결과를 summary 폴더에 저장할 새로운 파일명 생성
        new_filename = original_filename.replace("stt_", "nlp_")  # 파일명 변경
        # 지정된 파일명을 summary_folder에 저장
        output_key = f"{summary_folder}{new_filename}"
        output_path = f"/tmp/{new_filename}"

        with open(output_path, 'w', encoding='utf-8') as f:
            f.write(summary_text)

        # S3에 업로드
        s3.upload_file(output_path, bucket_name, output_key)
    except Exception as e:
        pass

def monitor_s3():
    """
    S3 폴더를 주기적으로 확인하여 새로운 파일을 처리
    """
    while True:
        try:
            response = s3.list_objects_v2(
                Bucket=bucket_name,
                Prefix=input_folder
            )
            if 'Contents' in response:
                for obj in response['Contents']:
                    file_key = obj['Key']
                    if file_key not in processed_files and file_key.endswith('.txt'):
                        process_file(file_key)
                        processed_files.add(file_key)
        except Exception as e:
            pass
```

```
            time.sleep(1)  # 1초마다 폴더 확인

@app.route('/status', methods=['GET'])
def status():
    """
    현재 처리된 파일 목록을 반환
    """
    return jsonify({"processed_files": list(processed_files)})

if __name__ == "__main__":
    # 백그라운드 스레드로 S3 모니터링 시작
    monitor_thread = threading.Thread(target=monitor_s3, daemon=True)
    monitor_thread.start()

    # Flask 앱 실행
    app.run(host='0.0.0.0', port=5000)
```

④ 입력 및 출력

- 입력
  - S3의 INPUT_FOLDER에 stt_1.txt 형식으로 저장된 파일
- 출력
  - 텍스트 파일이 OUTPUT_FOLDER에 nlp_1.txt 형식으로 저장

3. TTS (Text-to-Speech)

(1) TTS API 설명

① 사용 API

- Google TTS API

② 역할

- 요약된 텍스트 데이터를 자연스러운 음성으로 변환하여 사용자에게 응답

③ 기술적 구성

- Tacotron 2와 WaveNet 기반의 음성 합성 엔진을 사용하여 자연스러운 발음과 억양을 생성

④ 주요 알고리즘

- Tacotron 2 : Seq2Seq 모델로 텍스트를 음향 10)스펙트로그램으로 변환
- WaveNet : 생성된 스펙트로그램 데이터를 음성으로 변환하는 샘플 기반의 신경망

10) 소리나 파동을 시각화하여 파악하기 위한 도구로, 파형(waveform)과 스펙트럼(spectrum)의 특징이 조합되어 있음

⑵ 특징 및 선정 이유

- WaveNet과 같은 딥러닝 기술 활용 인간과 유사한 자연스러운 음성 제공
- .wav 파일 형식 지원
- 속도, 음량 등의 커스터마이징 옵션 지원
- TTS 결과는 S3에 저장하여 재사용 가능

⑶ 활용 방안

① API 사용 흐름

- 텍스트 파일 업로드 (S3)
- Google TTS API 호출을 통해 텍스트 데이터를 음성 파일로 변환
- FFmpeg로 AWS Connect 포맷에 맞게 변환
  - 포맷 형식은 8kHz, Mono, PCM μ-law
- 변환된 음성 파일을 S3에 저장

② 실행 환경 및 요구사항

- Python 3.9 이상
- Google TTS, boto3 라이브러리 설치
  - pip google-cloud-texttospeech boto3
- AWS 인증: S3 접근 권한 필요
  - IAM 역할에 S3 Access 정책 설정
- Google TTS API 활성화: 계정 키(JSON) 다운로드 후 환경 변수 설정
  - export GOOGLE_APPLICATION_CREDENTIALS = "계정 키.json"

③ 코드

- 핵심 함수 및 주요 로직
  - read_text_from_s3() : S3에서 텍스트 파일 읽기
  - synthesize_text() : Google TTS API 호출하여 텍스트를 음성으로 변환
  - convert_to_aws_connect_format() : FFmpeg로 음성을 AWS Connect 포맷으로 변환
  - process_files() : 텍스트 파일을 처리하고 음성으로 변환
  - upload_to_s3() : 변환된 음성 파일을 S3에 업로드
- 보조 함수
  - get_next_tts_file_name() : S3에 저장될 파일 번호 계산

- 예시 코드 (tts.py)

```
import boto3
import os
from google.cloud import texttospeech
import subprocess

# S3 클라이언트 초기화
s3 = boto3.client('s3')

# 처리한 파일 목록을 저장할 집합
processed_files = set()

def read_text_from_s3(bucket_name, file_key):
    """S3에서 텍스트 파일 읽기"""
    response = s3.get_object(Bucket=bucket_name, Key=file_key)
    return response['Body'].read().decode('utf-8')

def synthesize_text(text):
    """Google TTS API 호출로 텍스트를 음성으로 변환"""
    client = texttospeech.TextToSpeechClient()

    # SSML 태그로 발음 교정
    ssml_text = "<speak>{}</speak>".format(
        text.replace(
            "문의", '<phoneme alphabet="ipa" ph="muni">문의</phoneme>'
            )
    )

    input_text = texttospeech.SynthesisInput(ssml=ssml_text)
    voice = texttospeech.VoiceSelectionParams(
        language_code='ko-KR',
        name='ko-KR-Wavenet-A',
        ssml_gender=texttospeech.SsmlVoiceGender.NEUTRAL
    )
    audio_config = texttospeech.AudioConfig(
        audio_encoding=texttospeech.AudioEncoding.LINEAR16
    )
    response = client.synthesize_speech(
        input=input_text, voice=voice, audio_config=audio_config
    )
    return response.audio_content

def convert_to_aws_connect_format(input_path, output_path):
    """FFmpeg를 사용하여 AWS Connect 요구사항에 맞게 파일 변환"""
    command = [
        "ffmpeg",
        "-i", input_path,
```

```
        "-ar", "8000",  # 샘플링 레이트 8kHz
        "-ac", "1",  # Mono
        "-c:a", "pcm_mulaw",  # PCM μ-law 코덱
        "-f", "wav",  # 파일 형식 명시
        output_path
    ]
    subprocess.run(command, check=True)

def upload_to_s3(audio_content, bucket_name, object_name):
    """변환된 음성 파일을 S3에 업로드"""
    temp_file = "/tmp/temp_audio_file.wav"
    converted_file = "/tmp/converted_audio_file.wav"

    with open(temp_file, "wb") as f:
        f.write(audio_content)

    # 파일 변환
    convert_to_aws_connect_format(temp_file, converted_file)

    # S3에 업로드
    s3.upload_file(converted_file, bucket_name, object_name)

    # 임시 파일 삭제
    os.remove(temp_file)
    os.remove(converted_file)

def get_next_tts_file_name(bucket_name, prefix='tts/'):
    """S3에서 tts/ 경로의 다음 파일 이름 생성"""
    response = s3.list_objects_v2(Bucket=bucket_name, Prefix=prefix)
    max_index = 0

    if 'Contents' in response:
        for obj in response['Contents']:
            file_key = obj['Key']
            base_name = os.path.basename(file_key)

            # 파일 이름에서 번호 추출
            if base_name.startswith('tts_') and base_name.endswith('.wav'):
                try:
                    index = int(base_name[4:-4])
                    max_index = max(max_index, index)
                except ValueError:
                    continue

    return f"tts/tts_{max_index + 1}.wav"

def process_files(bucket_name):
    """S3의 텍스트 파일을 읽어 TTS로 변환하고 S3에 저장"""
```

```
    response = s3.list_objects_v2(Bucket=bucket_name, Prefix='nlp/')
    if 'Contents' in response:
        for obj in response['Contents']:
            file_key = obj['Key']
            # 텍스트 파일만 처리
            if file_key.endswith('.txt') and file_key not in processed_files:
                # 텍스트 읽기
                text = read_text_from_s3(bucket_name, file_key)
                # 텍스트 변환 전 "문의하신 내용이" 추가
                text = f"문의하신 내용이 {text}"
                # 텍스트를 음성으로 변환
                audio_content = synthesize_text(text)
                # 파일 이름 생성
                wav_file_key = get_next_tts_file_name(bucket_name)
                # 음성을 S3에 업로드
                upload_to_s3(audio_content, bucket_name, wav_file_key)
                # 처리된 파일 추가
                processed_files.add(file_key)

def main():
    bucket_name = 'Bucket Name 입력'

    while True:
        process_files(bucket_name)

        # 주기적으로 S3를 확인 (예: 1초마다 확인)
        time.sleep(1)

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

④ 입력 및 출력

- 입력
  - S3의 /nlp 폴더에 저장된 .txt 파일
- 출력
  - .wav 파일이 /tts 폴더에 tts_1.wav 형식으로 저장