# 포팅메뉴얼

## SSAFY 광주 캠퍼스

# 팀 : C208팀

# 작성자 : 김성현

# 목차

# 인프라 및 백엔드

# 1. 개요

# 2.시스템 환경 및 사전 준비

# 3. NGINX 설치 및 기본 구성

# 4. Spring Boot 애플리케이션 Docker 화

# 5. Jenkins Pipeline 구성

# 6. MySQL 설치

# 7. 모델 서버 배포

# 8. Minio 설치

# 프론트엔드

# 1. 개요

# 2. 개발환경설정

# 3. 프로젝트 설정

# 4. 프로젝트 구조

# 5. 주요 기능 및 서비스

# 6. 환경별 설정

# 7. 빌드 및 실행

# 8. 앱 서명 및 배포

# 9. 외부 API 연동 설정

# 빅데이터 포팅 메뉴얼

# 1. 빅데이터 처리 클러스터 구축

# 2. 그룹별 소비 데이터 분석 시스템 아키텍처

# 3. 랜덤 포레스트를 활용한 맛집 랭킹 시스템

## 개요

이 매뉴얼은 다음과 같은 환경을 대상으로 합니다.

**백엔드 애플리케이션**: Spring Boot (Java 17, Spring Boot 3.4)

프론트엔드 애플리케이션 : Flutter

**CI/CD 도구**: Jenkins (Jenkins Pipeline 사용)

**컨테이너 배포**: Docker

**리버스 프록시**: NGINX

**운영 체제**: Ubuntu 22

Jenkins를 통해 소스 코드를 체크아웃, 빌드, Docker 이미지 생성 및 컨테이너 실행을 자동화하고, NGINX를 통해 Spring Boot 컨테이너에 대한 외부 접근을 처리하는 방법을 단계별로 설명합니다.

## 시스템 환경 및 사전 준비

### Ubuntu 서버

Ubuntu (22.04.4)

서버에 최신 업데이트 진행

sudo apt-get update && sudo apt-get upgrade -y

### Jenkins 설치 및 구성

**Docker** CLI를 Jenkins에서 사용할 수 있도록 아래 사항들을 점검합니다.

sudo usermod -aG docker jenkins

Jenkins에서 필요한 플러그인(예: Git, Docker Pipeline, Pipeline)이 설치되어 있어야 합니다.

### Docker 설치

Docker가 설치 및 실행 중이어야 하며, Jenkins에서 Docker 명령어를 사용할 수 있도록 설정되어 있어야 합니다.

설치 확인:

docker --version

sudo systemctl status docker

### NGINX 설치 및 기본 구성

기본 설정 파일 /etc/nginx/sites-available/default 리버스 프록시 설정

server {

listen 80;

server\_name fourlipsclover.duckdns.org;

return 301 https://$host$request\_uri;

}

server {

listen 443 ssl;

server\_name fourlipsclover.duckdns.org;

ssl\_certificate /etc/letsencrypt/live/fourlipsclover.duckdns.org/fullchain.pem;

ssl\_certificate\_key /etc/letsencrypt/live/fourlipsclover.duckdns.org/privkey.pem;

include /etc/letsencrypt/options-ssl-nginx.conf;

ssl\_dhparam /etc/letsencrypt/ssl-dhparams.pem;

location / {

root /home/ubuntu/dist;

try\_files $uri /index.html;

}

location /api/ {

proxy\_pass http://127.0.0.1:8080/;

proxy\_set\_header Host $host;

proxy\_set\_header X-Real-IP $remote\_addr;

}

}

설정 후 NGINX 재시작:

sudo systemctl reload nginx

### 설정 반영

심볼릭 링크 생성 (만약 sites-available / sites-enabled 방식 사용 시):

sudo ln -s /etc/nginx/sites-available/fourlipsclover.conf /etc/nginx/sites-enabled/

기존 default 설정이 충돌하지 않게 비활성화(삭제)하거나, 필요시 수정합니다.

Nginx 구성 테스트 & 재시작:

sudo nginx -t

sudo systemctl restart nginx

## 방화벽에서 80/443 포트 열기

sudo ufw allow 80

sudo ufw allow 443

## Spring Boot 애플리케이션 Docker화

### 프로젝트 구조

src/

├── main/

│ ├── java/

│ │ └── com/

│ │ └── patriot/

│ │ └── fourlipsclover/

│ │ ├── global/

│ │ │ ├── config/

│ │ │ ├── error/

│ │ │ ├── security/

│ │ │ └── util/

│ │ ├── member/

│ │ │ ├── controller/

│ │ │ ├── dto/

│ │ │ │ ├── request/

│ │ │ │ └── response/

│ │ │ ├── entity/

│ │ │ ├── mapper/

│ │ │ ├── repository/

│ │ │ └── service/

│ │ ├── restaurant/

│ │ │ ├── controller/

│ │ │ ├── dto/

│ │ │ ├── entity/

│ │ │ ├── mapper/

│ │ │ ├── repository/

│ │ │ └── service/

│ │ ├── locals/

│ │ │ ├── controller/

│ │ │ ├── dto/

│ │ │ ├── entity/

│ │ │ ├── mapper/

│ │ │ ├── repository/

│ │ │ └── service/

│ │ ├── search/

│ │ │ ├── controller/

│ │ │ ├── dto/

│ │ │ ├── service/

│ │ │ └── repository/

│ │ └── FourlipscloverApplication.java

│ └── resources/

│ ├── application.yml

│ ├── application-dev.yml

│ ├── application-prod.yml

│ ├── static/

│ └── templates/

└── test/

└── java/

└── com/

└── patriot/

└── fourlipsclover/

├── member/

├── restaurant/

└── locals/

### Dockerfile

FROM openjdk:17-jdk-slim

COPY build/libs/\*.jar app.jar

EXPOSE 8090

ENTRYPOINT ["java", "-jar", "/app.jar"]

## Jenkins Pipeline 구성

Jenkins Pipeline을 통해 아래 단계가 자동으로 진행되도록 구성합니다.

**Git Repository에서 소스코드 체크아웃**

**Spring Boot 애플리케이션 빌드 및 Docker 이미지 생성**

**이전 컨테이너 종료 및 삭제 (이미 실행 중인 경우)**

**새 Docker 컨테이너 실행**

### 젠킨스 파이프라인

pipeline {

agent any

environment {

// 민감한 이미지명(예: 실제 이미지명 대신 별도 관리) - 실제 값으로 변경 시 보안에 유의

IMAGE\_NAME = "REPLACE\_SPRINGBOOT\_APP\_NAME"

}

stages {

stage('Checkout') {

steps {

// Git 리포지토리 URL 등 민감 데이터 마스킹

git url: 'REPLACE\_GIT\_REPO\_URL', branch: 'release'

}

}

stage('Inject application.yml') {

steps {

// Jenkins Credential에 저장된 파일의 Credential ID를 마스킹 처리

withCredentials([file(credentialsId: 'REPLACE\_APPLICATION\_YML\_CREDENTIAL\_ID', variable: 'APP\_YML\_FILE')]) {

sh '''

cp "$APP\_YML\_FILE" backend/src/main/resources/application.yml

'''

}

}

}

stage('Build Application') {

steps {

// backend 디렉토리로 이동하여 Maven Wrapper를 통해 애플리케이션 빌드

dir('backend') {

sh './mvnw clean package'

}

}

}

stage('Build Docker Image') {

steps {

// backend 디렉토리의 Dockerfile을 사용해 Docker 이미지 빌드

dir('backend') {

sh "docker build -t ${IMAGE\_NAME} ."

}

}

}

stage('Deploy') {

steps {

// 기존 컨테이너 종료 및 제거 후, host 네트워크 모드로 새 컨테이너 실행

sh """

docker stop ${IMAGE\_NAME} || true

docker rm ${IMAGE\_NAME} || true

docker run -d \\

--network host \\

--name ${IMAGE\_NAME} \\

${IMAGE\_NAME}

"""

}

}

}

post {

success {

echo "Deployment Successful!"

}

failure {

echo "Deployment Failed!"

}

}

}

## 젠킨스 credential

## 깃랩 api 토큰 (gitlab-token)

Gitlab에서 발급 후 입력

## 깃랩 C208 프로젝트 토큰 (jenkins\_project\_access\_token)

Gitlab에서 발급 후 입력

## application.yml (application-yml-file)

server:

port: 8090

spring:

jackson:

time-zone: Asia/Seoul

data:

elasticsearch:

repositories:

enabled: true

datasource:

driver-class-name: com.mysql.cj.jdbc.Driver

url: jdbc:mysql://<DB\_URL>/fourlipsclover?useSSL=false&serverTimezone=UTC

username: REPLACE\_ME

password: REPLACE\_ME

jpa:

hibernate:

ddl-auto: update

properties:

hibernate:

dialect: org.hibernate.dialect.MySQL8Dialect

servlet:

multipart:

max-file-size: 25MB

max-request-size: 25MB

# location: C:/board/upload/

redis:

host: REPLACE\_ME

port: REPLACE\_ME

password: REPLACE\_ME

kafka:

bootstrap-servers: REPLACE\_ME:9093

consumer:

group-id: fourlipsclover-app

auto-offset-reset: earliest

key-deserializer: org.apache.kafka.common.serialization.StringDeserializer

value-deserializer: org.springframework.kafka.support.serializer.JsonDeserializer

properties:

spring.json.trusted.packages: com.patriot.fourlipsclover.\*

producer:

key-serializer: org.apache.kafka.common.serialization.StringSerializer

value-serializer: org.springframework.kafka.support.serializer.JsonSerializer

logging:

level:

org.springframework.security: DEBUG

org.springframework.web: DEBUG

com.patriot.fourlipsclover: DEBUG

# Kakao Social Login

kakao:

userinfo:

url: https://kapi.kakao.com/v2/user/me

rest-api-key: REPLACE\_ME

payment:

admin-key: REPLACE\_ME

# JWT Configuration

jwt:

secret: REPLACE\_ME

validity-in-ms: 3600000

# salt: apple-banana-couple-dice-egg-fruit-golf-house-island-jmt

# access-token:

# expiretime: 600000

# refresh-token:

# expiretime: 1209600000

# File upload paths (Windows)

# file:

# path:

# upload-images: C:/board/upload/imageUpload

# upload-files: C:/board/upload/fileUpload/

management:

endpoints:

web:

exposure:

include: health,info

endpoint:

health:

show-details: always

minio:

url: http://REPLACE\_ME

accessKey: REPLACE\_ME

secretKey: REPLACE\_ME

bucketName:

reviewImage: review-images

mypage: mypage

chatImage: chat-image

model:

server:

uri: http://localhost:

elasticsearch:

host: REPLACE\_ME

port: 9200

scheme: https

username: elastic

password: REPLACE\_ME

# MYSQL 설치

## 시스템 업데이트 및 준비

**패키지 목록 업데이트**

우선, 시스템의 패키지 목록을 최신으로 업데이트합니다.

sudo apt-get update

**업데이트 및 업그레이드**

시스템 패키지들을 최신 상태로 업그레이드합니다.

sudo apt-get upgrade -y

## MySQL 서버 설치

**MySQL 서버 설치 명령어 실행**

Ubuntu 기본 리포지토리를 사용하여 MySQL 서버 패키지를 설치합니다.

sudo apt-get install mysql-server -y

**설치 진행 상황 확인**

설치 과정 중 특별한 메시지가 나타나지 않으면, 정상적으로 설치된 것입니다.

**재부팅 시 자동 시작 설정**

서버가 재부팅되어도 MySQL 서비스가 자동으로 시작되도록 설정합니다.

sudo systemctl enable mysql

# 보안 설정

보안 스크립트 실행

sudo mysql\_secure\_installation

## MySQL 아이디 비밀번호 설정

# 모델 서버 배포(fast-api)

import torch

from transformers import BertTokenizer, BertForSequenceClassification

import time

from fastapi import FastAPI, HTTPException, Request

from fastapi.responses import JSONResponse

from pydantic import BaseModel

import uvicorn

from typing import List, Optional

# FastAPI 앱 생성

app = FastAPI(title="맛집 리뷰 감성 분석 API",

description="양자화된 BERT 모델을 활용한 맛집 리뷰 감성 분석 API")

# 모델 클래스 정의

class ReviewSentimentAnalyzer:

def \_\_init\_\_(self, model\_path="quantized\_model.pt"):

# 디바이스 설정

self.device = torch.device("cpu")

# 토크나이저 로드

self.tokenizer = BertTokenizer.from\_pretrained(

'bert-base-multilingual-cased')

# 모델 구조 로드

print("양자화된 모델 불러오는 중...")

original\_model = BertForSequenceClassification.from\_pretrained(

'bert-base-multilingual-cased',

num\_labels=2

)

# 모델 가중치 로드

empty\_quantized\_model = torch.quantization.quantize\_dynamic(

original\_model,

{torch.nn.Linear},

dtype=torch.qint8

)

# 3. 저장된 양자화 모델의 state\_dict 로드

empty\_quantized\_model.load\_state\_dict(

torch.load("quantized\_model.pt", map\_location=self.device),

strict=False # 누락된 키가 있어도 로드 허용

)

self.model = empty\_quantized\_model

self.model.eval()

print("모델 로딩 완료!")

def analyze\_review(self, review\_text):

"""리뷰 텍스트의 감성을 분석하여 긍정/부정 여부와 신뢰도를 반환합니다."""

# 텍스트 전처리 및 텐서 변환

encoded\_input = self.tokenizer(

review\_text,

return\_tensors="pt",

padding=True,

truncation=True,

max\_length=128

)

input\_tensor = {k: v.to(self.device) for k, v in encoded\_input.items()}

# 추론

start\_time = time.time()

with torch.no\_grad():

outputs = self.model(\*\*input\_tensor)

logits = outputs.logits

probabilities = torch.softmax(logits, dim=1)

prediction = torch.argmax(probabilities, dim=1).item()

inference\_time = (time.time() - start\_time) \* 1000 # 밀리초 변환

# 결과 정리

sentiment = "긍정적" if prediction == 1 else "부정적"

confidence = probabilities.tolist()[0][prediction]

return {

"text": review\_text,

"sentiment": sentiment,

"confidence": float(confidence), # float 변환하여 JSON 직렬화 가능하게 함

"inference\_time\_ms": inference\_time

}

# Pydantic 모델 - 요청 및 응답 데이터 형식 정의

class ReviewRequest(BaseModel):

text: str

class ReviewsRequest(BaseModel):

reviews: List[str]

class ReviewResponse(BaseModel):

text: str

sentiment: str

confidence: float

inference\_time\_ms: float

class ReviewsResponse(BaseModel):

results: List[ReviewResponse]

total\_count: int

processing\_time\_ms: float

# 전역 변수로 모델 로드 (앱 시작 시 한 번만 로드되도록)

analyzer = None

@app.on\_event("startup")

async def startup\_event():

"""앱 시작 시 모델 로드"""

global analyzer

analyzer = ReviewSentimentAnalyzer()

@app.get("/")

async def root():

"""API 루트 경로"""

return {"message": "맛집 리뷰 감성 분석 API에 오신 것을 환영합니다!"}

@app.post("/analyze", response\_model=ReviewResponse)

async def analyze\_single\_review(request: ReviewRequest):

"""단일 리뷰 텍스트 분석 API"""

if not request.text.strip():

raise HTTPException(status\_code=400, detail="리뷰 텍스트가 비어있습니다")

result = analyzer.analyze\_review(request.text)

return result

@app.post("/analyze-batch", response\_model=ReviewsResponse)

async def analyze\_reviews\_batch(request: ReviewsRequest):

"""여러 리뷰 텍스트를 배치로 분석하는 API"""

if not request.reviews:

raise HTTPException(status\_code=400, detail="리뷰 목록이 비어있습니다")

start\_time = time.time()

results = []

for review in request.reviews:

if review.strip(): # 빈 텍스트가 아닌 경우만 분석

result = analyzer.analyze\_review(review)

results.append(result)

total\_time = (time.time() - start\_time) \* 1000 # 밀리초 변환

return {

"results": results,

"total\_count": len(results),

"processing\_time\_ms": total\_time

}

@app.exception\_handler(Exception)

async def global\_exception\_handler(request: Request, exc: Exception):

"""전역 예외 처리기"""

return JSONResponse(

status\_code=500,

content={"message": f"서버 오류가 발생했습니다: {str(exc)}"}

)

if \_\_name\_\_ == "\_\_main\_\_":

# 서버 실행

uvicorn.run("app:app", host="0.0.0.0", port=8000, reload=False)

## 프로젝트 파일 준비

# 프로젝트 디렉토리 생성

mkdir -p ~/sentiment-api

cd ~/sentiment-api

# 필요한 파일 생성

# app.py 파일 (기존 코드 사용)

# quantized\_model.pt 모델 파일 (필수)

## Dockerfile 생성

cat > Dockerfile << EOF

FROM python:3.9-slim

WORKDIR /app

# 필수 패키지 설치

RUN apt-get update && apt-get install -y \

build-essential \

&& rm -rf /var/lib/apt/lists/\*

# 필요한 Python 패키지 설치

COPY requirements.txt .

RUN pip install --no-cache-dir -r requirements.txt

# 애플리케이션 코드 복사

COPY app.py .

COPY quantized\_model.pt .

# 포트 설정

EXPOSE 8000

# 실행 명령어

CMD ["python", "app.py"]

EOF

# requirements.txt 생성

cat > requirements.txt << EOF

torch>=2.0.0

transformers>=4.30.0

fastapi>=0.95.1

uvicorn>=0.22.0

pydantic>=2.0.0

sentence-transformers>=2.2.0

python-multipart>=0.0.6

EOF

# Docker 이미지 빌드 및 실행

# Docker 이미지 빌드

docker build -t sentiment-api:latest .

# Docker 컨테이너 실행

docker run -d --name sentiment-analysis -p 8000:8000 sentiment-api:latest

# 컨테이너 상태 확인

docker ps

# API 테스트

# curl로 API 테스트

curl -X POST http://localhost:8000/analyze \

-H "Content-Type: application/json" \

-d '{"text":"이 음식점은 정말 맛있었어요!"}'

# curl로 API 테스트

curl -X POST http://localhost:8000/extract-tags \

-H "Content-Type: application/json" \

-d '{"text":"이 카페에는 콘센트가 많아서 모각코 하기 아주 좋습니다. ㅎ!!!!"}'

# 8. Minio 설치

### 데이터 디렉토리 준비

mkdir -p /home/ubuntu/minio-data

### 컨테이너 실행

docker run -d \

--name minio \

-p 9000:9000 \

-p 9001:9001 \

-e MINIO\_ROOT\_USER= \

-e MINIO\_ROOT\_PASSWORD= \

-v /home/ubuntu/minio-data:/data \

minio/minio server /data --console-address ":9001"

### 실행 확인

브라우저에서 http://domain-name:9001 접속

MINIO\_ROOT\_USER / MINIO\_ROOT\_PASSWORD로 로그인

MinIO 웹 콘솔이 뜨면 성공!

### 버킷 생성

웹 콘솔 좌측 상단 ‘Buckets’ → ‘+ Create Bucket’ 클릭

버킷 이름: chat-images, mypage, review-images

이 버킷에 이미지나 파일을 업로드할 수 있음.

# 프론트엔드 포팅 메뉴얼

## 개요

프로젝트명: 네잎클로버

프레임워크: Flutter

주요 기능: 소셜 로그인, 지도 연동, 그룹 관리, 일정 관리, 정산 기능, 리뷰/평가

## 개발 환경 설정

### 2.1 필수 설치 항목

Flutter SDK 3.7.0 이상

Dart SDK (Flutter SDK 내장)

Android Studio / VS Code

Git

Android 에뮬레이터 또는 실제 디바이스

iOS 개발 환경 (Xcode, Mac OS 필요)

### 2.2 Flutter 설치 및 설정

[Flutter SDK 설치 페이지](https://docs.flutter.dev/get-started/install) 참고

환경 변수 설정: [Flutter SDK 경로]/bin

flutter doctor

## 3. 프로젝트 설정

### 3.1 프로젝트 클론

git clone https://github.com/your-username/S12P21C208.git

cd S12P21C208/frontend

### 3.2 의존성 설치

flutter pub get

### 3.3 환경 변수 설정

루트 디렉토리에 .env 파일 생성:

KAKAO\_NATIVE\_APP\_KEY=[카카오 네이티브 앱 키]

KAKAO\_MOBILITY\_APP\_KEY=[카카오 모빌리티 앱 키]

API\_BASE\_URL=https://fourlipsclover.duckdns.org

## 프로젝트 구조

### 주요 디렉토리 구조

lib/

├── config/ 앱 설정 (routes.dart, theme.dart)

├── constants/ 상수 정의

├── models/ 데이터 모델

├── providers/ Provider 상태 관리

├── screens/ UI 화면

├── services/ API 및 외부 서비스

├── utils/ 유틸리티 함수

└── widgets/ 공통 위젯

assets/

├── images/

├── icons/

└── fonts/

## 주요 기능 및 서비스

카카오 로그인: kakao\_service.dart

그룹 관리: group\_api.dart

계획 관리: plan\_api.dart

정산 기능: settlement\_api.dart

위치 기반 서비스: kakao\_map\_service.dart

리뷰 관리: review\_service.dart

초대/딥링크: deep\_link\_service.dart

채팅 서비스: chat\_service.dart

결제 서비스: payment/

## 환경별 설정

개발 서버:

API\_BASE\_URL=https://fourlipsclover.duckdns.org

테스트 및 프로덕션 서버는 별도 지정

## 빌드 및 실행

디버그 실행:

flutter run

Android APK 빌드:

flutter build apk --release

Android App Bundle 빌드:

flutter build appbundle

iOS 빌드:

flutter build ios –release

## 앱 서명 및 배포

### Android

키스토어 생성 및 설정:

keytool -genkey -v -keystore upload-keystore.jks -keyalg RSA -keysize 2048 -validity 10000 -alias upload

key.properties 생성 및 설정:

storePassword=키스토어 비밀번호

keyPassword=키 비밀번호

keyAlias=upload

storeFile=../app/upload-keystore.jks

android/app/build.gradle에 서명 설정 추가

### 카카오 키 해시 등록

디버그 및 릴리즈 키 해시 생성 및 등록 ([카카오 개발자 콘솔](https://developers.kakao.com/))

### 앱 패키지명 변경

flutter pub run change\_app\_package\_name:main com.your.package.name

### 보안 주의사항

키 파일(.jks, key.properties) 버전 관리 제외(.gitignore 설정 필수)

## 외부 API 연동 설정

### 카카오 API 설정

카카오 개발자 콘솔에서 앱 생성 및 플랫폼 설정

Android 및 iOS 환경별 설정 필수

### Android Manifest

android/app/src/main/AndroidManifest.xml에 인터넷, 위치 권한, 카카오 로그인 및 딥링크 설정

### iOS Info.plist 설정

ios/Runner/Info.plist에 권한 및 URL 스킴 설정 추가

## 주요 의존성 패키지

상태 관리: provider

API 요청: http

보안 저장소: flutter\_secure\_storage

환경변수 관리: flutter\_dotenv

카카오 SDK: kakao\_flutter\_sdk

위치 서비스: geolocator

권한 관리: permission\_handler

결제 서비스: portone\_flutter

## 문제 해결

Gradle 빌드 오류 시: flutter clean, 의존성 업데이트 확인

카카오 로그인 실패: 키 해시 등록, 환경별 설정 확인

## 배포 체크리스트

### Android

버전 업데이트

아이콘 업데이트

App Bundle 빌드

Google Play 콘솔 업로드

### iOS

버전 업데이트

아이콘 업데이트

Archive 생성 및 App Store Connect 업로드

## 참고 자료

[Flutter 공식 문서](https://docs.flutter.dev/)

[카카오 개발자 문서](https://developers.kakao.com/docs/)

## 프로젝트 특화 정보

API 서버: [https://fourlipsclover.duckdns.org](https://fourlipsclover.duckdns.org/)

API 문서: 서버팀 요청

**네입클로버 빅데이터 포팅 메뉴얼**

**목차**

1. [빅데이터 처리 클러스터 구축](https://claude.ai/chat/5bf449ce-8b76-4bc6-b220-e9b13bb36788#%EB%B9%85%EB%8D%B0%EC%9D%B4%ED%84%B0-%EC%B2%98%EB%A6%AC-%ED%81%B4%EB%9F%AC%EC%8A%A4%ED%84%B0-%EA%B5%AC%EC%B6%95)
2. [그룹별 소비 데이터 분석 시스템 아키텍처](https://claude.ai/chat/5bf449ce-8b76-4bc6-b220-e9b13bb36788#%EA%B7%B8%EB%A3%B9%EB%B3%84-%EC%86%8C%EB%B9%84-%EB%8D%B0%EC%9D%B4%ED%84%B0-%EB%B6%84%EC%84%9D-%EC%8B%9C%EC%8A%A4%ED%85%9C-%EC%95%84%ED%82%A4%ED%85%8D%EC%B2%98)
3. [랜덤 포레스트를 활용한 맛집 랭킹 시스템](https://claude.ai/chat/5bf449ce-8b76-4bc6-b220-e9b13bb36788#%EB%9E%9C%EB%8D%A4-%ED%8F%AC%EB%A0%88%EC%8A%A4%ED%8A%B8%EB%A5%BC-%ED%99%9C%EC%9A%A9%ED%95%9C-%EB%A7%9B%EC%A7%91-%EB%9E%AD%ED%82%B9-%EC%8B%9C%EC%8A%A4%ED%85%9C)

**빅데이터 처리 클러스터 구축**

**EC2**

* **운영체제**: Ubuntu 24.04 LTS
* **인스턴스 타입**: t2.xlarge
* **스토리지**: 20GiB

**버전 정보**

* **OpenJDK ver**: 1.8.0
* **Python**: 3.8.2
* **Apache Hadoop**: 3.2.3
* **Apache Spark**: 3.2.1
* **Apache Zookeeper**: 3.8.0
* **Zeppelin**: 0.10.1

**클러스터 구축 방법**

**1. AWS EC2 인스턴스 배포 및 연결(MobaXterm활용)**

1. AWS 관리 콘솔에 로그인
2. EC2 대시보드로 이동하여 "인스턴스 시작" 버튼 클릭
3. 인스턴스 설정:
   * **이름**: node1
   * **AMI**: Ubuntu 24.04 LTS
   * **인스턴스 유형**: t2.xlarge
   * **키 페어**: 기존 키 페어 선택 또는 새 키 페어 생성
   * **네트워크 설정**: 기본 VPC 및 서브넷
   * **스토리지**: 20GiB gp3
4. "인스턴스 시작" 버튼 클릭

**MobaXterm을 사용한 EC2 연결**

1. MobaXterm 실행
2. "Session" 버튼 클릭 후 "SSH" 탭 선택
3. 다음 정보 입력:
   * **Remote host**: EC2 인스턴스의 퍼블릭 IP 주소
   * **Username**: ubuntu
   * **Port**: 22
4. "Advanced SSH settings" 섹션 확장
5. "Use private key" 옵션 선택 후 AWS에서 다운로드한 키 페어 파일(.pem) 지정
6. "OK" 버튼 클릭하여 연결

**기본 시스템 업데이트**

sudo apt update

sudo apt upgrade -y

**2. Java 설치 및 환경설정**

**OpenJDK 설치**

sudo apt install -y openjdk-8-jdk

**Java 환경변수 설정**

sudo nano /etc/profile.d/java.sh

다음 내용 추가:

export JAVA\_HOME=/usr/lib/jvm/java-8-openjdk-amd64

export PATH=$PATH:$JAVA\_HOME/bin

변경사항 적용:

source /etc/profile.d/java.sh

**Java 설치 확인**

java -version

echo $JAVA\_HOME

**3. Hadoop 설치 및 환경설정**

**Hadoop 다운로드 및 설치**

cd /opt

sudo wget https://archive.apache.org/dist/hadoop/common/hadoop-3.2.3/hadoop-3.2.3.tar.gz

sudo tar -xzf hadoop-3.2.3.tar.gz

sudo mv hadoop-3.2.3 hadoop

sudo chown -R ubuntu:ubuntu /opt/hadoop

**Hadoop 환경변수 설정**

sudo nano /etc/profile.d/hadoop.sh

다음 내용 추가:

export HADOOP\_HOME=/opt/hadoop

export HADOOP\_CONF\_DIR=$HADOOP\_HOME/etc/hadoop

export HADOOP\_MAPRED\_HOME=$HADOOP\_HOME

export HADOOP\_COMMON\_HOME=$HADOOP\_HOME

export HADOOP\_HDFS\_HOME=$HADOOP\_HOME

export YARN\_HOME=$HADOOP\_HOME

export PATH=$PATH:$HADOOP\_HOME/bin:$HADOOP\_HOME/sbin

변경사항 적용:

source /etc/profile.d/hadoop.sh

**Hadoop 설정 파일 구성**

**core-site.xml**:

sudo nano $HADOOP\_HOME/etc/hadoop/core-site.xml

다음 내용으로 수정:

<configuration>

<property>

<name>fs.defaultFS</name>

<value>hdfs://node1:9000</value>

</property>

</configuration>

**hdfs-site.xml**:

sudo nano $HADOOP\_HOME/etc/hadoop/hdfs-site.xml

다음 내용으로 수정:

<configuration>

<property>

<name>dfs.replication</name>

<value>3</value>

</property>

<property>

<name>dfs.namenode.name.dir</name>

<value>/opt/hadoop/data/namenode</value>

</property>

<property>

<name>dfs.datanode.data.dir</name>

<value>/opt/hadoop/data/datanode</value>

</property>

</configuration>

**mapred-site.xml**:

sudo nano $HADOOP\_HOME/etc/hadoop/mapred-site.xml

다음 내용으로 수정:

<configuration>

<property>

<name>mapreduce.framework.name</name>

<value>yarn</value>

</property>

<property>

<name>mapreduce.application.classpath</name>

<value>$HADOOP\_HOME/share/hadoop/mapreduce/\*:$HADOOP\_HOME/share/hadoop/mapreduce/lib/\*</value>

</property>

</configuration>

**yarn-site.xml**:

sudo nano $HADOOP\_HOME/etc/hadoop/yarn-site.xml

다음 내용으로 수정:

<configuration>

<property>

<name>yarn.nodemanager.aux-services</name>

<value>mapreduce\_shuffle</value>

</property>

<property>

<name>yarn.resourcemanager.hostname</name>

<value>node1</value>

</property>

</configuration>

**데이터 디렉토리 생성**

mkdir -p /opt/hadoop/data/namenode

mkdir -p /opt/hadoop/data/datanode

**workers 파일 설정**

sudo nano $HADOOP\_HOME/etc/hadoop/workers

다음 내용으로 수정:

node1

node2

node3

**4. Spark 설치 및 환경설정**

**Spark 다운로드 및 설치**

cd /opt

sudo wget https://archive.apache.org/dist/spark/spark-3.2.1/spark-3.2.1-bin-hadoop3.2.tgz

sudo tar -xzf spark-3.2.1-bin-hadoop3.2.tgz

sudo mv spark-3.2.1-bin-hadoop3.2 spark

sudo chown -R ubuntu:ubuntu /opt/spark

**Spark 환경변수 설정**

sudo nano /etc/profile.d/spark.sh

다음 내용 추가:

export SPARK\_HOME=/opt/spark

export PATH=$PATH:$SPARK\_HOME/bin:$SPARK\_HOME/sbin

export PYSPARK\_PYTHON=/usr/bin/python3

변경사항 적용:

source /etc/profile.d/spark.sh

**Spark 설정 파일 구성**

**spark-env.sh**:

cp $SPARK\_HOME/conf/spark-env.sh.template $SPARK\_HOME/conf/spark-env.sh

nano $SPARK\_HOME/conf/spark-env.sh

다음 내용 추가:

export JAVA\_HOME=/usr/lib/jvm/java-8-openjdk-amd64

export SPARK\_MASTER\_HOST=node1

export HADOOP\_CONF\_DIR=$HADOOP\_HOME/etc/hadoop

export SPARK\_EXECUTOR\_MEMORY=2g

export SPARK\_DRIVER\_MEMORY=1g

**spark-defaults.conf**:

cp $SPARK\_HOME/conf/spark-defaults.conf.template $SPARK\_HOME/conf/spark-defaults.conf

nano $SPARK\_HOME/conf/spark-defaults.conf

다음 내용 추가:

spark.master spark://node1:7077

spark.eventLog.enabled true

spark.eventLog.dir hdfs://node1:9000/spark-logs

spark.history.fs.logDirectory hdfs://node1:9000/spark-logs

**workers 파일 설정**

cp $SPARK\_HOME/conf/workers.template $SPARK\_HOME/conf/workers

nano $SPARK\_HOME/conf/workers

다음 내용으로 수정:

node1

node2

node3

**5. Zookeeper 설치 및 환경설정**

**Zookeeper 다운로드 및 설치**

cd /opt

sudo wget https://archive.apache.org/dist/zookeeper/zookeeper-3.8.0/apache-zookeeper-3.8.0-bin.tar.gz

sudo tar -xzf apache-zookeeper-3.8.0-bin.tar.gz

sudo mv apache-zookeeper-3.8.0-bin zookeeper

sudo chown -R ubuntu:ubuntu /opt/zookeeper

**Zookeeper 환경변수 설정**

sudo nano /etc/profile.d/zookeeper.sh

다음 내용 추가:

export ZOOKEEPER\_HOME=/opt/zookeeper

export PATH=$PATH:$ZOOKEEPER\_HOME/bin

변경사항 적용:

source /etc/profile.d/zookeeper.sh

**Zookeeper 설정 파일 구성**

cp $ZOOKEEPER\_HOME/conf/zoo\_sample.cfg $ZOOKEEPER\_HOME/conf/zoo.cfg

nano $ZOOKEEPER\_HOME/conf/zoo.cfg

다음 내용으로 수정:

tickTime=2000

initLimit=10

syncLimit=5

dataDir=/opt/zookeeper/data

clientPort=2181

maxClientCnxns=60

admin.enableServer=true

admin.serverPort=9876

server.1=node1:2888:3888

server.2=node2:2888:3888

server.3=node3:2888:3888

**데이터 디렉토리 생성 및 myid 설정**

mkdir -p /opt/zookeeper/data

echo "1" > /opt/zookeeper/data/myid

**6. AMI 생성 및 인스턴스 복제**

**AMI 생성**

1. AWS 관리 콘솔에서 EC2 인스턴스(node1) 선택
2. "작업" > "이미지 및 템플릿" > "이미지 생성" 클릭
3. 다음 정보 입력:
   * **이미지 이름**: BigData-Cluster-Base
   * **이미지 설명**: Base image for BigData Cluster with Hadoop, Spark and Zookeeper
4. "이미지 생성" 버튼 클릭
5. AMI 생성 완료까지 대기

**인스턴스 복제**

1. AMI 생성 완료 후 "AMI" 섹션으로 이동
2. 생성한 AMI 선택 후 "시작하기" 버튼 클릭
3. 다음 구성으로 인스턴스 설정:
   * **인스턴스 수**: 2 (node2, node3 생성)
   * **이름**: node2, node3
   * 기타 설정은 원본 인스턴스와 동일하게 유지
4. "인스턴스 시작" 버튼 클릭

**7. 탄력적IP 및 호스트이름 설정**

**탄력적 IP 할당**

1. AWS 관리 콘솔에서 "탄력적 IP" 섹션으로 이동
2. "탄력적 IP 주소 할당" 버튼 클릭
3. 기본 설정 유지하고 "할당" 버튼 클릭
4. 세 개의 인스턴스(node1, node2, node3)에 각각 탄력적 IP 할당:
   * 새로 할당된 탄력적 IP 선택 후 "작업" > "탄력적 IP 주소 연결" 클릭
   * 인스턴스 선택 후 "연결" 버튼 클릭

**/etc/hosts 파일 설정**

모든 노드(node1, node2, node3)에서 다음 명령 실행:

sudo nano /etc/hosts

다음 내용 추가(IP 주소는 각 노드의 탄력적 IP로 변경):

127.0.0.1 localhost

<node1\_ip> node1

<node2\_ip> node2

<node3\_ip> node3

**호스트이름 설정**

각 노드에서 다음 명령 실행:

**Node1**:

sudo hostnamectl set-hostname node1

**Node2**:

sudo hostnamectl set-hostname node2

**Node3**:

sudo hostnamectl set-hostname node3

**myid 파일 설정**

각 노드의 Zookeeper myid 파일 설정:

**Node1**:

echo "1" > /opt/zookeeper/data/myid

**Node2**:

echo "2" > /opt/zookeeper/data/myid

**Node3**:

echo "3" > /opt/zookeeper/data/myid

**8. Zookeeper 클러스터 실행**

**각 노드에서 Zookeeper 서비스 시작**

모든 노드(node1, node2, node3)에서 다음 명령 실행:

$ZOOKEEPER\_HOME/bin/zkServer.sh start

**Zookeeper 서비스 상태 확인**

모든 노드에서 다음 명령 실행:

$ZOOKEEPER\_HOME/bin/zkServer.sh status

정상적으로 설정된 경우, 한 노드는 "leader"로, 나머지 두 노드는 "follower"로 표시됩니다.

**9. Hadoop 클러스터 실행**

**HDFS 포맷 (node1에서만 실행)**

hdfs namenode -format

**Hadoop 클러스터 시작 (node1에서만 실행)**

start-dfs.sh

start-yarn.sh

**Hadoop 클러스터 상태 확인**

hdfs dfsadmin -report

yarn node -list

**HDFS에 디렉토리 생성**

hdfs dfs -mkdir -p /spark-logs

hdfs dfs -chmod -R 777 /spark-logs

**10. Spark 클러스터 실행**

**Spark 클러스터 시작 (node1에서만 실행)**

$SPARK\_HOME/sbin/start-all.sh

**Spark 클러스터 상태 확인**

$SPARK\_HOME/bin/spark-submit --version

**11. 보안그룹 편집 및 WebUI 확인**

**보안그룹 편집**

1. AWS 관리 콘솔에서 "보안 그룹" 섹션으로 이동
2. 클러스터 인스턴스에 연결된 보안 그룹 선택
3. "인바운드 규칙 편집" 버튼 클릭
4. 다음 포트 추가:
   * HDFS NameNode WebUI: 9870
   * YARN ResourceManager WebUI: 8088
   * Spark Master WebUI: 8080
   * Spark History Server: 18080
   * Zeppelin: 8890
5. "규칙 저장" 버튼 클릭

**웹 인터페이스 접속 확인**

웹 브라우저에서 다음 URL을 통해 각 서비스의 웹 인터페이스에 접속:

* HDFS NameNode: http://node1-public-ip:9870
* YARN ResourceManager: http://node1-public-ip:8088
* Spark Master: http://node1-public-ip:8080
* Spark History Server: http://node1-public-ip:18080

**12. 클러스터 실행 스크립트 생성 및 권한설정**

**클러스터 시작 스크립트 생성**

nano ~/start-cluster.sh

다음 내용 추가:

#!/bin/bash

echo "Starting ZooKeeper cluster..."

$ZOOKEEPER\_HOME/bin/zkServer.sh start

echo "Starting Hadoop cluster..."

$HADOOP\_HOME/sbin/start-dfs.sh

$HADOOP\_HOME/sbin/start-yarn.sh

echo "Starting Spark cluster..."

$SPARK\_HOME/sbin/start-all.sh

echo "Starting Zeppelin server..."

$ZEPPELIN\_HOME/bin/zeppelin-daemon.sh start

echo "All services started. Checking status..."

$ZOOKEEPER\_HOME/bin/zkServer.sh status

hdfs dfsadmin -report | grep "Live datanodes"

yarn node -list | grep "Total Nodes"

$SPARK\_HOME/bin/spark-submit --version

**클러스터 중지 스크립트 생성**

nano ~/stop-cluster.sh

다음 내용 추가:

#!/bin/bash

echo "Stopping Zeppelin server..."

$ZEPPELIN\_HOME/bin/zeppelin-daemon.sh stop

echo "Stopping Spark cluster..."

$SPARK\_HOME/sbin/stop-all.sh

echo "Stopping Hadoop cluster..."

$HADOOP\_HOME/sbin/stop-yarn.sh

$HADOOP\_HOME/sbin/stop-dfs.sh

echo "Stopping ZooKeeper cluster..."

$ZOOKEEPER\_HOME/bin/zkServer.sh stop

echo "All services stopped."

**스크립트 실행 권한 설정**

chmod +x ~/start-cluster.sh

chmod +x ~/stop-cluster.sh

**13. Zeppelin 설치 및 PySpark 연동**

**Zeppelin 다운로드 및 설치**

cd /opt

sudo wget https://archive.apache.org/dist/zeppelin/zeppelin-0.10.1/zeppelin-0.10.1-bin-all.tgz

sudo tar -xzf zeppelin-0.10.1-bin-all.tgz

sudo mv zeppelin-0.10.1-bin-all zeppelin

sudo chown -R ubuntu:ubuntu /opt/zeppelin

**Zeppelin 환경변수 설정**

sudo nano /etc/profile.d/zeppelin.sh

다음 내용 추가:

export ZEPPELIN\_HOME=/opt/zeppelin

export PATH=$PATH:$ZEPPELIN\_HOME/bin

변경사항 적용:

source /etc/profile.d/zeppelin.sh

**Zeppelin 설정 파일 구성**

**zeppelin-env.sh**:

cp $ZEPPELIN\_HOME/conf/zeppelin-env.sh.template $ZEPPELIN\_HOME/conf/zeppelin-env.sh

nano $ZEPPELIN\_HOME/conf/zeppelin-env.sh

다음 내용 추가:

export JAVA\_HOME=/usr/lib/jvm/java-8-openjdk-amd64

export SPARK\_HOME=/opt/spark

export HADOOP\_CONF\_DIR=/opt/hadoop/etc/hadoop

export PYSPARK\_PYTHON=/usr/bin/python3

export ZEPPELIN\_PORT=8890

**zeppelin-site.xml**:

cp $ZEPPELIN\_HOME/conf/zeppelin-site.xml.template $ZEPPELIN\_HOME/conf/zeppelin-site.xml

nano $ZEPPELIN\_HOME/conf/zeppelin-site.xml

필요한 경우 설정 수정

**Zeppelin 서비스 시작**

$ZEPPELIN\_HOME/bin/zeppelin-daemon.sh start

**Zeppelin 웹 인터페이스 접속 확인**

웹 브라우저에서 다음 URL을 통해 Zeppelin 웹 인터페이스에 접속:

http://node1-public-ip:8890

**Zeppelin에서 PySpark 인터프리터 설정**

1. Zeppelin 웹 인터페이스에 접속
2. 우측 상단의 "anonymous" > "Interpreter" 클릭
3. "spark" 인터프리터를 찾아 "edit" 버튼 클릭
4. 다음 속성 설정:
   * **spark.master**: spark://node1:7077
   * **SPARK\_HOME**: /opt/spark
   * **spark.submit.deployMode**: cluster
5. "Save" 버튼 클릭
6. 인터프리터 재시작

이제 Zeppelin 노트북에서 PySpark를 사용할 수 있습니다.

**그룹별 소비 데이터 분석 시스템 아키텍처**

**시스템 아키텍처**

**1. 데이터 수집 및 저장 계층**

* **MySQL 데이터베이스**: 사용자 그룹과 소비 데이터를 저장
  + 주요 테이블: member, group, restaurant, visit\_payment
* **Debezium 커넥터**: MySQL 변경 데이터 캡처(CDC)
  + 소비 내역 변경 사항을 실시간으로 감지하여 Kafka로 전송
* **Apache Kafka**: 메시지 큐잉 및 이벤트 스트리밍
  + 데이터 토픽: mysql-server.fourlipsclover.visit\_payment
  + 분석 결과 토픽: group-analysis-results
* **HDFS**: 대용량 데이터 저장소
  + 소비 데이터 저장: /user/fourlipsclover/visit\_payment
  + AIHUB 데이터 저장: /user/aihub/consumption.csv

**2. 데이터 처리 및 분석 계층**

* **Apache Spark**: 분산 데이터 처리 엔진
  + 소비 데이터와 AIHUB 데이터 병합 및 분석
  + 다양한 분석 유형 실행: 기본 비교, 카테고리별, 방문 인원별, 시간대별, 요일별, 지역별
* **Apache Zeppelin**: 데이터 분석 노트북
  + 분석 스크립트 작성 및 실행
  + 동적 파라미터 처리를 통한 그룹별 분석
  + 분석 결과 시각화

**3. 애플리케이션 계층**

* **Spring Boot 백엔드**:
  + REST API를 통한 분석 결과 제공
  + Kafka Consumer를 통한 분석 결과 수신
  + 메모리 캐시를 활용한 분석 결과 관리
  + Zeppelin API 연동을 통한 분석 작업 요청
* **Flutter 프론트엔드**:
  + 사용자 인터페이스 제공
  + 분석 결과 시각화

**데이터 흐름**

1. **데이터 수집**
   * MySQL의 소비 데이터 변경 사항이 Debezium을 통해 감지됨
   * 변경 데이터가 Kafka 토픽으로 전송됨
2. **데이터 적재**
   * Spark가 Kafka 토픽의 데이터를 읽어 HDFS에 Parquet 형식으로 저장
   * AIHUB 데이터는 CSV 형식으로 HDFS에 저장됨
3. **데이터 분석**
   * 사용자가 특정 그룹의 분석을 요청 (API 호출)
   * Spring Boot 서버가 Zeppelin API를 통해 분석 파라그래프 실행 요청
   * Zeppelin이 Spark를 이용하여 HDFS에 저장된 데이터 분석 수행
   * 분석 결과가 Kafka 토픽으로 전송됨
4. **결과 제공**
   * Spring Boot 서버가 Kafka에서 분석 결과를 수신하여 메모리 캐시에 저장
   * 클라이언트 요청에 따라 분석 결과를 REST API로 제공
   * 클라이언트가 분석 결과를 시각화하여 표시

**분석 유형**

1. **기본 비교** (basic\_comparison)
   * 선택 그룹 vs 다른 그룹의 평균/총 지출 비교
2. **방문 인원별 비교** (personnel\_comparison)
   * 방문 인원수에 따른 소비 패턴 비교
3. **시간대별 비교** (time\_comparison)
   * 하루 중 시간대별 소비 패턴 비교
4. **요일별 비교** (day\_of\_week\_comparison)
   * 요일별 소비 패턴 비교

**구현된 API 엔드포인트**

* `GET /api/analysis/group/{groupId}

**제플린 분석 코드**

**데이터 로드 및 전처리**

%spark

import org.apache.spark.sql.functions.\_

import org.apache.spark.sql.types.\_

import java.sql.Timestamp

import org.apache.hadoop.fs.{FileSystem, Path}

// 스파크 설정 변경

spark.conf.set("spark.sql.parquet.int96RebaseModeInWrite", "LEGACY")

// 1. Debezium 형식의 카프카 데이터를 읽기

val debeziumSchema = new StructType()

.add("schema", StringType)

.add("payload", new StructType()

.add("after", new StructType()

.add("visit\_payment\_id", IntegerType)

.add("amount", IntegerType)

.add("restaurant\_id", IntegerType)

.add("user\_id", LongType)

.add("visited\_personnel", IntegerType)

.add("created\_at", TimestampType)

.add("paid\_at", TimestampType)

.add("data\_source", StringType)

)

.add("op", StringType)

)

println("카프카에서 방문 결제 데이터 읽기 시작...")

// 카프카에서 visit\_payment 데이터 읽기

val rawDF = spark

.read

.format("kafka")

.option("kafka.bootstrap.servers", "43.203.49.76:9093")

.option("subscribe", "mysql-server.fourlipsclover.visit\_payment")

.option("startingOffsets", "earliest")

.option("endingOffsets", "latest") // 현재 최신 오프셋까지 읽기

.load()

println(s"카프카에서 ${rawDF.count()} 개의 레코드를 읽었습니다.")

// 데이터 파싱

val parsedDF = rawDF

.selectExpr("CAST(value AS STRING)")

.select(from\_json(col("value"), debeziumSchema).as("data"))

.select("data.payload.after.\*", "data.payload.op")

println("데이터 파싱 완료. 유효한 데이터 필터링 중...")

// 날짜 필드 정제

val validDF = parsedDF

.filter(col("visit\_payment\_id").isNotNull)

.filter(col("op") === "c") // 생성 작업만 필터링

.drop("op")

.withColumn("created\_at", when(col("created\_at").isNull || year(col("created\_at")) < 1900, lit(current\_timestamp())).otherwise(col("created\_at")))

.withColumn("paid\_at", when(col("paid\_at").isNull || year(col("paid\_at")) < 1900, lit(current\_timestamp())).otherwise(col("paid\_at")))

println(s"유효한 데이터 ${validDF.count()} 개를 발견했습니다.")

// HDFS에 저장할 경로

val hdfsPath = "/user/fourlipsclover/visit\_payment"

// HDFS에서 기존 데이터 삭제

try {

val fs = FileSystem.get(spark.sparkContext.hadoopConfiguration)

val path = new Path(hdfsPath)

if (fs.exists(path)) {

println(s"기존 데이터 경로 ${hdfsPath}를 삭제합니다.")

fs.delete(path, true)

println("기존 데이터 삭제 완료.")

} else {

println(s"경로 ${hdfsPath}에 기존 데이터가 없습니다.")

}

} catch {

case e: Exception =>

println(s"데이터 삭제 중 오류 발생: ${e.getMessage}")

e.printStackTrace()

}

// 데이터 샘플 확인

println("저장할 데이터 샘플:")

validDF.show(5)

// HDFS에 저장

try {

validDF

.write

.mode("overwrite") // 혹시 남아있는 데이터가 있을 경우를 대비해 overwrite 유지

.parquet(hdfsPath)

println(s"방문 결제 데이터 ${validDF.count()}개가 HDFS에 성공적으로 저장되었습니다.")

} catch {

case e: Exception =>

println(s"데이터 저장 중 오류 발생: ${e.getMessage}")

e.printStackTrace()

}

// 저장된 데이터 확인

val savedDF = spark.read.parquet(hdfsPath)

println(s"HDFS에 저장된 데이터 수: ${savedDF.count()}")

println("저장된 데이터 샘플:")

savedDF.show(5)

**그룹 소비 분석 함수**

%spark

import org.apache.spark.sql.functions.\_

import org.apache.spark.sql.DataFrame

import scala.collection.mutable.Map

// 그룹 ID를 받는 파라미터화된 함수

def analyzeGroupSpending(groupId: Long): Map[String, DataFrame] = {

try {

println(s"그룹 ID ${groupId}에 대한 분석 시작...")

val visitPaymentPath = "/user/fourlipsclover/visit\_payment"

val aihubPath = "/user/aihub/consumption.csv"

// 캐시 무효화

spark.sqlContext.clearCache()

// visit\_payment 데이터 로드

println("HDFS에서 방문 결제 데이터 로드 중...")

val visitPaymentDF = spark.read

.parquet(visitPaymentPath)

.filter(col("amount").isNotNull)

println(s"방문 결제 데이터 ${visitPaymentDF.count()}개 로드 완료.")

println("방문 결제 데이터 샘플:")

visitPaymentDF.show(3)

// AIHUB CSV 데이터 로드

println("AIHUB 데이터 로드 중...")

val aihubDF = spark.read

.option("header", "true")

.option("inferSchema", "true")

.csv(aihubPath)

.select("group\_id", "personnel", "amount", "paid\_at")

println(s"AIHUB 데이터 ${aihubDF.count()}개 로드 완료.")

println("AIHUB 데이터 샘플:")

aihubDF.show(3)

// 두 데이터프레임 통합

println("데이터 통합 중...")

val combinedDF = visitPaymentDF

.select(

col("user\_id").as("group\_id"),

col("visited\_personnel").as("personnel"),

col("amount"),

col("paid\_at")

)

.union(aihubDF)

.filter(col("personnel").between(0, 20))

// 그룹 ID 체크

println("데이터에 존재하는 그룹 ID 확인:")

combinedDF.select("group\_id").distinct().orderBy("group\_id").show(10)

// 분석 대상 그룹의 데이터 수 확인

val ourGroupCount = combinedDF.filter(col("group\_id") === groupId).count()

println(s"우리 그룹(ID: ${groupId})의 데이터 수: ${ourGroupCount}")

if (ourGroupCount == 0) {

println(s"경고: 그룹 ID ${groupId}에 해당하는 데이터가 없습니다!")

}

// 1. 기본 그룹 비교

println("1. 기본 비교 분석 수행 중...")

val basicComparison = combinedDF

.groupBy(

when(col("group\_id") === groupId, "우리 그룹")

.otherwise("다른 그룹").as("group\_type")

)

.agg(

avg("amount").as("avg\_spending"),

sum("amount").as("total\_spending"),

count("\*").as("transaction\_count")

)

// 2. 방문 인원별 소비 패턴 비교

println("2. 방문 인원별 분석 수행 중...")

val personnelComparison = combinedDF

.groupBy(

coalesce(col("personnel"), lit(0)).as("visited\_personnel"),

when(col("group\_id") === groupId, "우리 그룹")

.otherwise("다른 그룹").as("group\_type")

)

.agg(

avg("amount").as("avg\_spending"),

count("\*").as("visit\_count")

)

.orderBy("visited\_personnel", "group\_type")

// 3. 시간대별 비교

println("3. 시간대별 분석 수행 중...")

val timeComparison = combinedDF

.filter(col("paid\_at").isNotNull)

.groupBy(

hour(col("paid\_at")).as("hour\_of\_day"),

when(col("group\_id") === groupId, "우리 그룹")

.otherwise("다른 그룹").as("group\_type")

)

.agg(

avg("amount").as("avg\_spending"),

count("\*").as("visit\_count")

)

.orderBy("hour\_of\_day", "group\_type")

// 4. 요일별 비교

println("4. 요일별 분석 수행 중...")

val dayOfWeekComparison = combinedDF

.filter(col("paid\_at").isNotNull)

.groupBy(

dayofweek(col("paid\_at")).as("day\_of\_week"),

when(col("group\_id") === groupId, "우리 그룹")

.otherwise("다른 그룹").as("group\_type")

)

.agg(

avg("amount").as("avg\_spending"),

count("\*").as("visit\_count")

)

.orderBy("day\_of\_week", "group\_type")

// 모든 분석 결과 확인

println("\n분석 결과:")

println("\n기본 비교 분석 결과:")

basicComparison.show()

println("\n방문 인원별 소비 패턴 비교:")

personnelComparison.show()

println("\n시간대별 소비 패턴 비교:")

timeComparison.show(24) // 모든 시간대 표시

println("\n요일별 소비 패턴 비교:")

dayOfWeekComparison.show()

// 모든 분석 결과를 맵으로 반환

val resultMap = Map[String, DataFrame]()

resultMap("basic\_comparison") = basicComparison

resultMap("personnel\_comparison") = personnelComparison

resultMap("time\_comparison") = timeComparison

resultMap("day\_of\_week\_comparison") = dayOfWeekComparison

println(s"그룹 ID ${groupId}에 대한 모든 분석이 완료되었습니다.")

resultMap

} catch {

case e: Exception =>

println(s"분석 과정에서 오류 발생: ${e.getMessage}")

e.printStackTrace()

Map[String, DataFrame]()

}

}

**분석 결과 Kafka 전송 함수**

%spark

import org.apache.spark.sql.DataFrame

import scala.collection.mutable.Map

// 분석 결과를 카프카로 전송하는 함수

def sendAnalysisToKafka(groupId: Long): Boolean = {

try {

// 그룹 분석 수행

val analysisResults = analyzeGroupSpending(groupId)

if (analysisResults.isEmpty) {

println("분석 결과가 없습니다.")

return false

}

// 결과에 그룹 ID와 타임스탬프 추가

val timestamp = current\_timestamp()

val enrichedResults = scala.collection.mutable.Map[String, DataFrame]()

// 각 분석 결과에 그룹 ID와 타임스탬프 추가

analysisResults.foreach { case (key, df) =>

if (df != null) {

val enrichedDF = df.withColumn("group\_id", lit(groupId))

.withColumn("timestamp", lit(timestamp))

.withColumn("analysis\_type", lit(key))

enrichedResults(key) = enrichedDF

}

}

// 모든 분석 결과를 하나의 JSON 객체로 변환

val completeResult = Map[String, String]()

enrichedResults.foreach { case (key, df) =>

val jsonRows = df.toJSON.collect()

completeResult(key) = jsonRows.mkString("[", ",", "]")

}

// 최종 JSON 생성

import org.apache.spark.sql.types.\_

import org.apache.spark.sql.Row

val resultSchema = StructType(Seq(

StructField("analysis\_id", StringType, false),

StructField("group\_id", LongType, false),

StructField("timestamp", TimestampType, false),

StructField("analysis\_data", StringType, false)

))

val resultRows = completeResult.map { case (key, jsonData) =>

Row(key, groupId, new Timestamp(System.currentTimeMillis()), jsonData)

}.toSeq

val resultDF = spark.createDataFrame(

spark.sparkContext.parallelize(resultRows),

resultSchema

)

// 결과 확인

println("카프카로 전송할 데이터 스키마:")

resultDF.printSchema()

println("카프카로 전송할 데이터 샘플:")

resultDF.show(truncate = false)

// 카프카로 전송

resultDF.selectExpr("to\_json(struct(\*)) AS value")

.write

.format("kafka")

.option("kafka.bootstrap.servers", "43.203.49.76:9093")

.option("topic", "group-analysis-results")

.save()

println(s"그룹 ID ${groupId}의 다양한 분석 결과가 카프카로 전송되었습니다.")

true

} catch {

case e: Exception =>

println(s"카프카 전송 오류: ${e.getMessage}")

e.printStackTrace()

false

}

}

// 분석할 그룹 ID 설정 (이 값이 동적으로 변경됨)

val groupId = 95L // 이 줄을 Spring Boot에서 변경합니다

println(s"분석할 그룹 ID: $groupId")

// 그룹 ID를 사용하여 분석 실행

val success = sendAnalysisToKafka(groupId)

if (success) {

println(s"그룹 ID ${groupId}의 카프카 전송 성공!")

} else {

println(s"그룹 ID ${groupId}의 카프카 전송 실패. 로그를 확인하세요.")

}

**랜덤 포레스트를 활용한 맛집 랭킹 시스템**

**1. 랜덤 포레스트란?**

랜덤 포레스트는 여러 개의 의사결정나무를 결합하여 더 강력한 예측 모델을 만드는 머신러닝 알고리즘입니다. 마치 숲(forest)에서 여러 나무들이 함께 의견을 모으는 것처럼, 다수의 의사결정나무가 투표하여 최종 결과를 도출합니다.

**의사결정나무의 한계와 랜덤 포레스트의 해결책**

* **단일 의사결정나무**: 하나의 길만 따라 예측하므로 과적합(overfitting)되기 쉽고, 특정 데이터에 지나치게 영향을 받습니다.
* **랜덤 포레스트**: 여러 나무가 각기 다른 데이터와 특성을 보고 판단하므로 전체적인 시각을 가지고 균형 잡힌 결정을 내립니다.

**2. 랜덤 포레스트 작동 원리**

**랜덤성을 통한 다양한 관점 확보**

1. **부트스트랩 샘플링**: 원본 데이터에서 무작위로 샘플을 여러 번 추출 (중복 허용)
2. **특성 무작위 선택**: 각 분기점에서 전체 특성 중 일부만 무작위로 선택하여 고려
3. **다수결 투표**: 모든 트리의 예측 결과를 합쳐 최종 결정

이런 랜덤성 덕분에 각 트리는 데이터의 서로 다른 패턴과 관계를 학습하게 됩니다.

**3. 맛집 랭킹에 랜덤 포레스트 적용 이유**

**복잡한 사용자 행동 패턴 이해**

맛집 평가는 다양한 요소의 복합적인 상호작용입니다. 랜덤 포레스트는 이런 복잡한 패턴을 잘 포착합니다:

* 리뷰 내용의 긍정/부정 감정
* 다른 사용자들의 리뷰에 대한 반응(좋아요/싫어요)
* 방문 횟수와 결제 금액
* 사용자의 신뢰도 점수

**공정한 랭킹 생성**

단일 지표만으로는 맛집을 평가하기 어렵습니다. 랜덤 포레스트는 여러 지표를 종합적으로 고려하여 더 공정한 랭킹을 만들어냅니다.

**이상치와 노이즈에 강건함**

일부 사용자의 극단적인 평가나 비정상적인 행동에 영향을 덜 받아, 더 안정적인 랭킹을 제공합니다.

**4. 맛집 랭킹 시스템 구현 과정**

**1. 데이터 수집 및 전처리**

맛집 랭킹에 영향을 미치는 여러 데이터를 수집합니다:

* **방문 정보**: 실제 방문 및 결제 기록
* **리뷰 정보**: 사용자가 작성한 리뷰 및 감정 분석 결과
* \*\*반응`: 특정 그룹의 모든 분석 결과 조회
* GET /api/analysis/group/{groupId}/{analysisType}: 특정 그룹의 특정 유형 분석 결과 조회