AI Programming

데이터 마이닝 / 생성형 AI

02. 데이터 전처리와 기초통계

first coding

first coding

데이터 전처리의 개요

데이터 전처리의 중요성

데이터의 유형(정형, 비정형, 반정형 데이터)

데이터 전처리 과정 개요: 데이터 수집, 정제, 변환, 통합

데이터 수집 방법 개요

- **데이터 전처리란?**

  - **데이터를 분석**하거나 **머신러닝 모델**에 사용하기 전에 깨끗하고 유용한 상태로 준비하는 과정
  - 데이터 수집 후 비어 있는 값(누락된 값), 엉뚱한 값(오류), 필요 없는 값(중복)이 있다면, 이를 수정하고 정리하는 과정
    → 데이터 전처리는 데이터를 “사용하기 좋게” 만드는 청소 과정

  - 왜 필요한가?
    - 잘못된 데이터를 그대로 사용하면, 분석 결과나 모델이 엉뚱한 답을 내놓을 수 있기 때문
    - 데이터 전처리는 요리 전에 재료를 씻고 다듬는 것과 같습니다.깨끗한 재료로 요리를 해야 맛있는 음식을 만들 수 있듯, 깨끗한 데이터로 분석해야 좋은 결과를 얻을 수 있습니다.

데이터 수집 방법 개요

- **데이터 전처리의 중요성**

1. 데이터 품질이 결과에 미치는 영향

   - **사례**

     온라인 쇼핑몰의 추천 시스템한 온라인 쇼핑몰이 고객 맞춤형 추천 시스템을 운영

   - **문제**

     데이터에 누락된 정보와 오류가 있음 (예: 고객 나이 필드가 빈 상태이거나, 이름이 숫자로 저장된 경우.)

   - **결과**

     추천 시스템이 올바른 정보를 기반으로 학습하지 못해, 어린이에게 어른 용 제품을 추천하거나, 고객 취향과 무관한 제품을 추천 → 이는 매출 감소로 이어질 수 있음

   - **해결**

     누락된 나이는 평균값으로 채우고, 잘못된 데이터를 수정하면, 추천 정확도가 높아짐

     → 매출 증가와 고객 만족도 향상을 기대할 수 있음

데이터 수집 방법 개요

- **데이터 전처리의 중요성**

2. 모델 성능에 미치는 영향

   - **사례**

     이메일 서비스에서 스팸 이메일을 걸러내는 머신러닝 모델을 구축

   - **문제**

     이메일 텍스트에 특수문자와 HTML 태그가 포함되어 있어 모델이 주요 단어를 인식하지 못함

     데이터셋의 레이블(스팸/정상)이 잘못된 경우도 있음.

   - 결과

     모델이 제대로 학습하지 못하고 스팸 이메일을 정상 이메일로 분류하거나,

     반대로 정상 이메일을 스팸으로 분류.

   - 해결

     데이터 전처리 단계에서 특수문자와 HTML 태그를 제거하고 텍스트를 정리.

     잘못된 레이블을 확인하고 수정하면, 모델의 스팸 분류 정확도가 크게 향상됩니다.

데이터 수집 방법 개요

- **데이터 전처리의 중요성**

3. 시간과 비용 절약

   - **사례**

     한 회사가 고객 이탈을 예측하는 분석 프로젝트를 진행

   - **문제**

     동일한 고객이 이름을 다르게 입력(“홍길동” / “Hong Gil Dong”)하거나, 중복된 구매 기록이 포함됨.

     분석 중간에 데이터를 수정해야 하는 상황이 빈번히 발생.

   - **결과**

     데이터를 수정하는 데 시간이 소요되어 프로젝트가 지연되고, 추가 인력과 비용이 필요해짐.

   - 해결

     데이터 전처리 단계에서 중복을 제거하고, 고객 이름 포맷을 통일(한글 혹은 영어로 통일)하면 문제를 사전에 방지할 수 있음. → 프로젝트 기간을 단축하고 비용을 절감할 수 있음.

데이터 수집 방법 개요

- **데이터 전처리의 중요성**

4. 의사결정의 정확성 향상

   - 사례

     한 병원이 환자의 건강 상태 데이터를 분석하여 질병 위험도를 예측

   - 문제

     혈압 필드에 비정상적으로 높은 값(1000 이상)이 포함되거나, 음수 값이 입력된 경우

   - 결과

     잘못된 데이터로 분석을 진행하면, 병원이 환자의 질병 위험도를 잘못 평가할 가능성이 높음.

     → 환자의 건강을 위협하거나 병원의 신뢰도를 떨어뜨림

   - 해결

     데이터 정제 과정에서 비정상적으로 높은 값은 제거하고, 음수 값은 적절히 수정 또는 삭제.

     올바른 데이터를 기반으로 분석하면, 환자에게 정확한 진단 정보를 제공할 수 있음.

데이터 수집 방법 개요

- 데이터 전처리의 중요성

  - 데이터 전처리는 데이터의 품질을 높이고, 분석이나 모델링의 결과를 신뢰할 수 있게 해 줌
  - 데이터를 잘 정리하지 않으면 잘못된 결과를 초래하고, 시간과 비용이 낭비
  - 앞 사례에서 보듯이, **전처리는 단순한 작업이 아니라 성공적인 데이터 프로젝트의 필수적인 단계**

데이터 수집 방법 개요

- 데이터의 유형
  - 정형 데이터 (Structured Data)
  - 비정형 데이터 (Unstructured Data)
  - 반정형 데이터 (Semi-Structured Data)

데이터 수집 방법 개요

- 데이터의 유형
  - **정형 데이터 (Structured Data)**
    - 정형 데이터는 행(row)과 열(column)로 구성되어 있는, 잘 정리된 데이터

      → **데이터베이스나 엑셀 파일 (은행 기록, 월급 명세서)**
    - 특징
      - 데이터가 표처럼 정렬되어 있음. 각 열에 이름(필드명)과 데이터 유형(숫자, 문자열 등)이 정해져 있음.
    - 예시
      - 은행의 고객 정보 데이터 → 열: 이름, 나이, 계좌번호, 잔액 행: 개별 고객의 정보
      - 매출 데이터 → 열: 제품명, 판매수량, 판매가격 행: 개별 판매 기록사용
    - 사례
      - 데이터베이스에서 고객 정보를 조회. 월별 매출 통계를 계산.

데이터 수집 방법 개요

- 데이터의 유형
  - **비정형 데이터 (Unstructured Data)**
    - 비정형 데이터는 고정된 구조가 없는, 정리되지 않은 데이터 → **이미지, 동영상, 텍스트 파일**
    - 특징
      - 데이터의 모양이나 구조가 일정하지 않고 분석하려면 먼저 데이터를 가공해야 함.
    - 예시
      - 텍스트 데이터: 이메일, 채팅 기록, 문서 파일
      - 이미지 데이터: 사진, 스캔한 문서
      - 동영상 데이터: 유튜브 영상, 감시 카메라 영상
      - 오디오 데이터: 음성 녹음 파일, 음악 파일사용
    - 사례
      - 소셜 미디어 게시글을 분석해 감정(긍정/부정)을 파악
      - 이미지 데이터를 분석해 얼굴을 인식. 동영상을 분석해 특정 장면을 찾음.

데이터 수집 방법 개요

- 데이터의 유형
  - **반정형 데이터 (Semi-Structured Data)**
    - 반정형 데이터는 정형 데이터와 비정형 데이터의 중간 단계로 데이터에 구조는 있지만, 정형 데이터처럼 엄격하지 않음 → 대표적으로 JSON과 XML 같은 파일
    - 특징
      - 데이터에 구조가 있지만, 형태가 유연함.
      - 사람이 읽을 수 있으며, 컴퓨터도 처리 가능
    - 예시
      - JSON, XML, 웹 로그 데이터
    - 사용 사례
      - API 데이터를 사용해 웹 애플리케이션 개발.
      - 웹 로그 데이터를 분석해 사용자 행동 패턴 파악.

```json
{
  "이름": "홍길동",
  "나이": 30,
  "취미": ["독서", "등산"]
}
```

```xml
<고객>
  <이름>홍길동</이름>
  <나이>30</나이>
  <취미>독서</취미>
</고객>
```

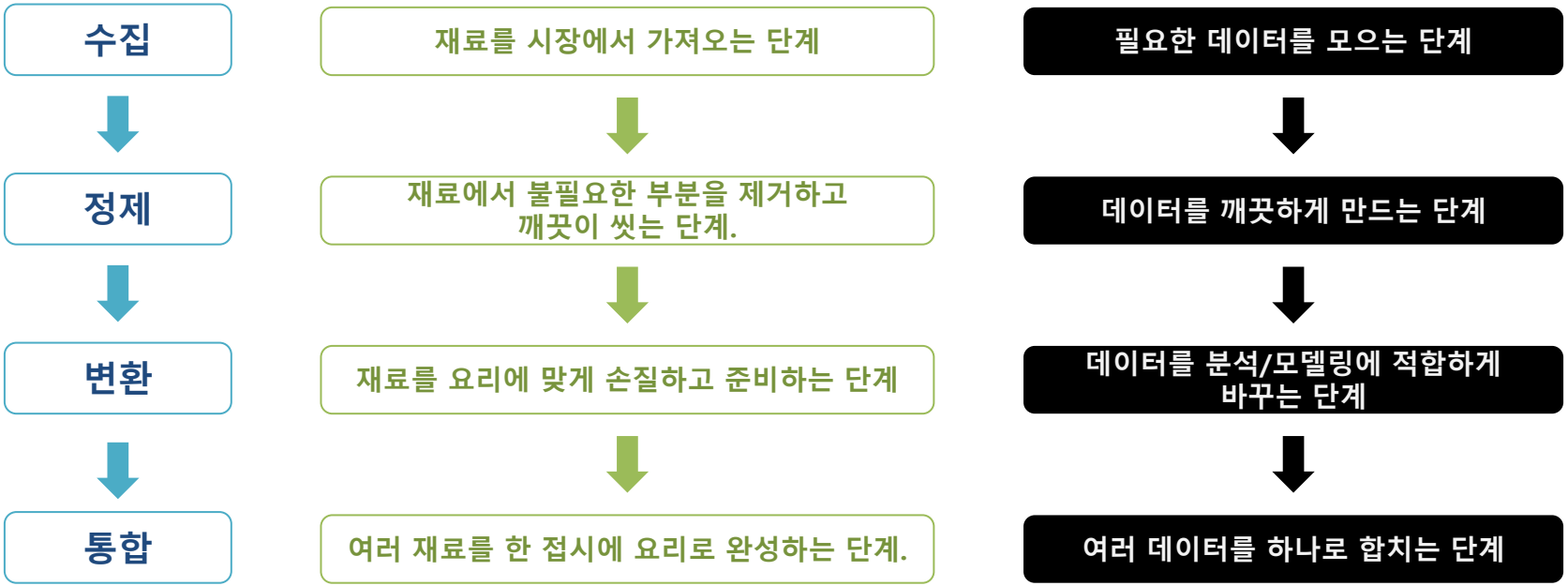
데이터 수집 방법 개요

- 데이터 전처리 과정 개요

  - 데이터 전처리는 데이터를 분석이나 모델링에 사용하기 위해 깨끗하고 유용하게 준비하는 단계
  - 데이터 수집 → 정제 → 변환 → 통합의 순서로 진행
  - 각 단계는 **데이터의 품질과 활용도를 높이는 데 초점**이 맞춰져 있음

데이터 수집 방법 개요

- 데이터 전처리 과정 개요



데이터 수집 방법 개요

- 데이터 전처리 과정

① 데이터 수집

- 데이터를 가져올 출처를 결정하고, 필요한 데이터를 수집
- 데이터가 불완전하거나 잘못된 경우가 많기 때문에, 수집한 데이터의 품질을 확인해야 함
- 주요 출처
  - 데이터베이스 (SQL, NoSQL 등)
  - API (예: 날씨 데이터, 주식 데이터)
  - 웹 크롤링 (인터넷에서 데이터 수집)
  - 센서 데이터 (IoT 기기에서 실시간 데이터 수집)
- 사례
  - 고객 행동 분석을 위해 웹사이트 방문 기록과 구매 데이터를 수집.
  - 기상 정보를 활용하기 위해 날씨 API에서 데이터를 가져옴.

데이터 수집 방법 개요

- 데이터 전처리 과정

② 데이터 정제 (Cleaning)

- 수집한 데이터는 종종 불완전하고 오류가 있을 수 있음

  → 데이터 정제는 이런 **문제를 해결하여 데이터를 깨끗하게 만드는 단계**

- 데이터를 정제하지 않으면, 이후 단계에서 잘못된 결과를 얻을 가능성이 높아짐
- 사례
  - 고객 설문 데이터를 정제하면서 이름이 중복되거나 설문에 응답하지 않은 항목을 제거.
  - IoT 데이터에서 센서 오류로 기록된 -999 값을 제거.

데이터 수집 방법 개요

- 데이터 전처리 과정
  - 주요 작업
    - **결측 값 처리 :** 데이터에 비어 있는 값(NaN, Null 등)이 있을 경우, 이를 해결.
      예: **평균값, 중앙값으로 채우거나 해당 데이터를 삭제.**
    - **오류 수정 :** 잘못된 데이터(오타, 잘못된 값 등)를 수정.
      예: **나이에 음수 값이 입력된 경우 이를 제거.**
    - **중복 제거 :** 동일한 데이터가 여러 번 나타나는 경우 삭제.
      예: **고객 정보가 중복되어 있는 경우 하나만 남김.**
    - **이상치 제거 :** 너무 크거나 작은 값 등 정상적이지 않은 값을 처리.
      예: **쇼핑몰에서 상품 가격이 1억 원으로 입력된 경우.**

데이터 수집 방법 개요

- 데이터 전처리 과정

③ 데이터 변환 (Transformation)

- 정제된 데이터를 분석과 모델링에 적합한 형태로 변환하는 단계

  → 데이터 변환은 **데이터를 해석하기 쉽게 하고, 모델링 과정에서 성능을 향상시킴**

- 사례
  - 고객 데이터를 분석하기 위해 나이 데이터를 나이대(20대, 30대 등)로 변환.
  - 주식 데이터를 로그 변환하여 모델링에 적합한 형태로 변경.

데이터 수집 방법 개요

- 데이터 전처리 과정
  - 주요 작업
    - **형태 변환** : 데이터를 사용 목적에 맞게 바꾸기.
      예: **텍스트 데이터를 숫자로 인코딩 ('남성' → 0, '여성' → 1).**
    - **정규화(Normalization)** : 데이터 범위를 일정하게 조정.
      예: **가격 데이터를 0~1 사이로 변환하여 비교 가능하게 만듦.**
    - **집계(Aggregation)** : 데이터를 요약하거나 그룹화.
      예: **일별 데이터를 월별로 집계.**
    - **차원 축소** : 불필요한 데이터를 제거하여 데이터 크기를 줄임.
      예: **머신러닝 모델에서 중요하지 않은 열 제거.**

데이터 수집 방법 개요

- 데이터 전처리 과정

④ 데이터 통합 (Integration)

- 다양한 데이터 자원에서 수집한 데이터를 하나로 합치는 과정

  → **데이터가 여러 개의 테이블, 파일, 또는 시스템에 분산되어 있을 수 있기 때문임**

  → 주의점 : 데이터의 의미와 관계를 잘 이해하지 않으면 잘못된 결합이 이루어질 수 있음

- 주요 작업

  - **데이터 병합(Merging)**

    두 개 이상의 데이터셋을 결합. 예: **고객 정보와 구매 데이터를 연결**.

  - **일관성 유지**

    서로 다른 데이터의 포맷과 구조를 맞춤. 예: **날짜 포맷을 'YYYY-MM-DD' 형식으로 통일**.

  - **중복 제거**

    통합된 데이터에서 중복된 항목 삭제.

- 사례 CRM(고객 관계 관리) 시스템의 고객 정보와 웹 로그 데이터를 통합하여 고객 행동 분석.

  병원의 환자 기록 데이터와 진료 기록 데이터를 결합해 질병 패턴 분석.

정규 표현식을 이용한 보안 로그 처리 예제

- 파이썬 re 모듈을 활용하여 보안 로그에서 IP 주소, 날짜, 접근 경로, 상태 코드 등을 추출하고,이를 CSV 파일로 저장
  - 예시 로그 데이터 (sample_logs.log)

```
192.168.1.10 - - [14/Jun/2025:10:15:32 +0900] "GET /admin/login.php HTTP/1.1" 200 532

10.20.30.40 - - [14/Jun/2025:10:16:01 +0900] "POST /upload.php HTTP/1.1" 403 189

172.16.0.2 - - [14/Jun/2025:10:17:55 +0900] "GET /index.html HTTP/1.1" 200 1024
```

정규 표현식을 이용한 보안 로그 처리 예제

- 파이썬 re 모듈을 활용하여 보안 로그에서 IP 주소, 날짜, 접근 경로, 상태 코드 등을 추출하고,이를 CSV 파일로 저장
  - 파이썬 코드: 로그 파싱 및 CSV 저장

```
import re
import csv

# 로그 파일 경로
log_file_path = 'sample_logs.txt'
output_csv_path = 'parsed_logs.csv'
```

정규 표현식을 이용한 보안 로그 처리 예제

- 파이썬 re 모듈을 활용하여 보안 로그에서 IP 주소, 날짜, 접근 경로, 상태 코드 등을 추출하고,이를 CSV 파일로 저장
  - 파이썬 코드: 로그 파싱 및 CSV 저장

```
# 정규 표현식 패턴 정의
log_pattern = re.compile(
    r'(?P<ip>\d+\.\d+\.\d+\.\d+)\s'                    # IP 주소
    r'- - \[(?P<datetime>[^\]]+)\] '                   # 날짜 및 시간
    r'"(?P<method>GET|POST|PUT|DELETE|HEAD) '          # HTTP 메서드
    r'(?P<path>[^ ]+).*?"\s'                           # 요청 경로
    r'(?P<status>\d{3})\s'                             # 상태 코드
    r'(?P<size>\d+)'                                   # 응답 크기
)

# (?P<이름>패턴) 형태는 "이름이 있는 그룹" 으로, 추출 결과를 딕셔너리처럼
groupdict()로 관리할 수 있게 해줍니다.
```

정규 표현식을 이용한 보안 로그 처리 예제

- 파이썬 re 모듈을 활용하여 보안 로그에서 IP 주소, 날짜, 접근 경로, 상태 코드 등을 추출하고,이를 CSV 파일로 저장

정규식 부분	설명	예시 추출 값
(?P<ip>₩d+₩.₩d+₩.₩d+₩.₩d+)	IP 주소 형식 (예: 192.168.0.1) (?P<ip>...)는 추출된 값을 "ip"라는 이름으로 저장	"192.168.1.10"
₩s	공백 한 칸	
- -	Apache 로그의 포맷상 사용자 식별자 자리인데 일반적으로 -로 표시	
₩[(?P<datetime>[^₩]]+)₩]	대괄호 [ ]로 감싼 시간 문자열을 "datetime"이라는 이름으로 추출	14/Jun/2025:10:15:32 +0900
`"(?PGET	POST	PUT
(?P<path>[^]+)	요청한 URL 경로 부분을 추출	"/admin/login.php"
.*?"	나머지 요청 정보(HTTP 버전 등)를 건너뜀	
(?P<status>₩d{3})	응답 상태 코드 (3자리 숫자)	200, 403 등
(?P<size>₩d+)	응답 본문의 크기(바이트 수)	예: 532

정규 표현식을 이용한 보안 로그 처리 예제

- 파이썬 re 모듈을 활용하여 보안 로그에서 IP 주소, 날짜, 접근 경로, 상태 코드 등을 추출하고,이를 CSV 파일로 저장
  - 파이썬 코드: 로그 파싱 및 CSV 저장

```
# 파싱된 결과 저장 리스트
parsed_logs = []

# 로그 파일 읽기 및 정규식 적용
with open(log_file_path, 'r') as f:
    for line in f:
        match = log_pattern.search(line)
        if match:
            parsed_logs.append(match.groupdict()) #
```

정규 표현식을 이용한 보안 로그 처리 예제

- 파이썬 re 모듈을 활용하여 보안 로그에서 IP 주소, 날짜, 접근 경로, 상태 코드 등을 추출하고,이를 CSV 파일로 저장
  - 파이썬 코드: 로그 파싱 및 CSV 저장

```
# CSV로 저장
with open(output_csv_path, 'w', newline='') as csvfile:
    fieldnames = ['ip', 'datetime', 'method', 'path', 'status', 'size']
    writer = csv.DictWriter(csvfile, fieldnames=fieldnames)

    writer.writeheader()
    for entry in parsed_logs:
        writer.writerow(entry)

print("보안 로그가 CSV로 저장되었습니다:", output_csv_path)
```

정규 표현식을 이용한 보안 로그 처리 예제

- 파이썬 코드: 로그 파싱 및 CSV로그 파일에서 데이터 추출

  → pandas DataFrame 변환python복사편집 저장

```
import re
import pandas as pd

# 로그 파일 경로
log_file_path = 'sample_logs.log'

# 정규 표현식 패턴 정의
log_pattern = re.compile(
    r'(?P<ip>\d+\.\d+\.\d+\.\d+)\s'
    r'- - \[(?P<datetime>[^\]]+)\] '
    r'"(?P<method>GET|POST|PUT|DELETE) '
    r'(?P<path>[^ ]+).*?" '
    r'(?P<status>\d{3}) '
    r'(?P<size>\d+)'
)
```

정규 표현식을 이용한 보안 로그 처리 예제

- 파이썬 코드: 로그 파싱 및 CSV로그 파일에서 데이터 추출

  → pandas DataFrame 변환python복사편집 저장

```
# 결과 저장
log_entries = []

with open(log_file_path, 'r') as f:
    for line in f:
        match = log_pattern.search(line)
        if match:
            entry = match.groupdict()
            # datetime을 datetime 객체로 변환
            entry['datetime'] = pd.to_datetime(entry['datetime'], format='%d/%b/%Y:%H:%M:%S %z')
            entry['status'] = int(entry['status'])
            entry['size'] = int(entry['size'])
            log_entries.append(entry)

# DataFrame 생성
df = pd.DataFrame(log_entries)

# 결과 확인
print(df.head())
```

정규 표현식을 이용한 보안 로그 처리 예제

- 파이썬 코드: 로그 파싱 및 CSV로그 파일에서 데이터 추출

  → pandas DataFrame 변환python복사편집 저장

  - 403 상태 코드(Forbidden) 요청만 추출

```
forbidden_df = df[df['status'] == 403]
print(forbidden_df)
```

  - /admin 경로에 접근한 로그만 추출

```
admin_access_df = df[df['path'].str.contains(r'^/admin')]
print(admin_access_df)
```

정규 표현식을 이용한 보안 로그 처리 예제

- 파이썬 코드: 로그 파싱 및 CSV로그 파일에서 데이터 추출

  → pandas DataFrame 변환python복사편집 저장

  - 403 상태 코드(Forbidden) 요청만 추출

```
status_counts = df.groupby('status').size().reset_index(name='count')
print(status_counts)
```

  - IP별 요청 횟수 Top 5

```
top_ips = df.groupby('ip').size().reset_index(name='count').sort_values(by='count',
ascending=False).head(5)
print(top_ips)
```

  - 시간별 요청량 집계 (분 단위)python복사편집

```
df['minute'] = df['datetime'].dt.floor('min')
requests_per_minute = df.groupby('minute').size().reset_index(name='count')
print(requests_per_minute)
```

first coding

데이터 정제 (Cleaning)

결측치 처리 (제거, 대체)

이상치 처리 (IQR, Z-Score)

데이터 중복 확인 및 제거

결측치 처리 (제거, 대체)

- **결측치 처리**

  - **결측치**
    - 데이터셋에서 특정 값이 누락된 상태

      예: 설문조사에서 어떤 사람이 나이를 입력하지 않았거나, 센서가 데이터를 제대로 수집하지 못한 경우
    - **결측치는 데이터 분석 결과에 영향을 줄 수 있으므로 반드시 처리해야 함**

  - 결측치를 처리하는 이유
    - **분석 왜곡 방지** : 결측값을 그대로 두면 평균, 합계 같은 계산이 왜곡될 수 있음
    - **모델 학습 방해 방지** : 머신러닝 모델은 결측값을 포함한 데이터를 처리하지 못하는 경우가 많이 발생
    - **데이터 품질 향상** : 결측치를 처리하면 데이터가 더 신뢰할 수 있게 됨

결측치 처리 (제거, 대체)

- 결측치 처리

  - 결측치 제거
    - 사용 상황
      - 결측치가 데이터셋에서 차지하는 비율이 매우 적을 때.
      - 해당 열이나 행이 분석에 큰 영향을 주지 않을 때
    - 방법
      - 열 제거
        특정 열에 결측값이 너무 많다면, 해당 열 자체를 삭제.
      - 행 제거
        특정 행에 결측값이 있으면, 해당 행을 삭제.

```python
import pandas as pd

# 데이터프레임 생성
data = {'Name': ['Alice', 'Bob', None],
        'Age': [25, None, 30],
        'Score': [85, 90, None]}
df = pd.DataFrame(data)

# 결측치가 있는 행 제거
df_no_missing_rows = df.dropna()
print(df_no_missing_rows)

print('*'* 50)

# 결측치가 있는 열 제거
df_no_missing_cols = df.dropna(axis=1)
print(df_no_missing_cols)
```

```
    Name   Age  Score
0  Alice  25.0   85.0
**************************************************
Empty DataFrame
Columns: []
Index: [0, 1, 2]
```

결측치 처리 (제거, 대체)

- 결측치 처리

  - 결측치 대체
    - 사용 상황
      - 데이터의 양이 적거나, 결측치가 무작정 삭제되면 데이터 손실이 너무 클 때.
    - 방법
      - 고정 값으로 대체
        특정 값(예: 0, 평균, 중간값)으로 결측치를 채움.
      - 통계 기반 대체
        - » 평균값으로 대체: 수치형 데이터에서 결측값을 해당 열의 평균으로 대체.
        - » 중간값으로 대체: 평균이 극단값에 영향을 받는 경우, 중간값 사용.
        - » 최빈값으로 대체: 범주형 데이터에서 가장 많이 등장한 값으로 결측치를 대체.
      - 예측 기반 대체: 머신러닝 모델을 사용해 결측치를 예측하여 채움.

```python
import pandas as pd

# 데이터프레임 생성
data = {'Name': ['Alice', 'Bob', None],
        'Age': [25, None, 30],
        'Score': [85, 90, None]}
df = pd.DataFrame(data)

# 평균으로 대체
df['Age'] = df['Age'].fillna(df['Age'].mean())

# 중간값으로 대체
df['Score'] = df['Score'].fillna(df['Score'].median())

# 최빈값으로 대체
df['Name'] = df['Name'].fillna(df['Name'].mode()[0])

print(df)
```

```
    Name   Age  Score
0  Alice  25.0   85.0
1    Bob  27.5   90.0
2  Alice  30.0   87.5
```

결측치 처리 (제거, 대체)

- **결측치 처리**

  - 데이터 손실을 최소화하고 분석에 미치는 영향을 줄이려면
    - 결측치의 양, 데이터의 중요성, 그리고 분석 목표를 고려해야 함
    - 간단한 데이터셋에서는 제거와 평균/중간값 대체가 유용
    - 하지만, 복잡한 데이터에서는 머신러닝 기반 대체를 활용.

이상치 처리 (IQR, Z-Score)

- 이상치 처리

  - 이상치
    - 데이터셋에서 다른 값들과 크게 동떨어져 있는 값

      예 : 학생의 평균 시험 점수가 70~90점 사이인데, 한 학생이 150점을 받은 경우 이상치로 간주
    - 이상치는 분석 결과를 왜곡하거나 모델의 성능을 저하시킬 수 있으므로 적절히 처리해야 함

  - 이상치 처리의 필요성
    - **데이터 왜곡 방지** : 이상치는 평균, 분산, 상관관계 등의 통계값에 큰 영향을 미칠 수 있음
    - **모델 성능 향상** : 머신러닝 모델은 이상치로 인해 잘못된 패턴을 학습할 수 있음
    - **데이터 신뢰성 확보** : 이상치를 처리하면 데이터의 품질이 높아짐

이상치 처리 (IQR, Z-Score)

- 이상치 처리

  - IQR (Interquartile Range) 방법
    - IQR은 데이터의 중간 50%를 포함하는 범위를 이용해 이상치를 탐지하는 방법
      - 비대칭 분포 데이터에 적합.
      - 극단값에 영향을 덜 받음.
    - Q1 (1사분위수): 하위 25%의 값.
    - Q3 (3사분위수): 상위 75%의 값.
    - IQR: $IQR=Q3-Q1$
    - 이상치 기준
      - 아래쪽: $Q1-1.5\times IQR$Q1-1.5×IQR보다 작은 값.
      - 위쪽: $Q3+1.5\times IQR$Q3+1.5×IQR보다 큰 값.

```python
import numpy as np
import pandas as pd

# 예제 데이터
data = {'Scores': [70, 75, 80, 85, 90, 150]}
df = pd.DataFrame(data)

# IQR 계산
Q1 = df['Scores'].quantile(0.25)  # 1사분위수
Q3 = df['Scores'].quantile(0.75)  # 3사분위수
IQR = Q3 - Q1

# 이상치 기준
lower_bound = Q1 - 1.5 * IQR
upper_bound = Q3 + 1.5 * IQR

# 이상치 확인
outliers = df[(df['Scores'] < lower_bound) | (df['Scores'] >
upper_bound)]
print(outliers)
# 이상치 제거
#df = df[(df['Scores'] >= lower_bound) & (df['Scores'] <=
upper_bound)]
#이상치 변경
df.loc[df['Scores'] < lower_bound, 'Scores'] = lower_bound
df.loc[df['Scores'] > upper_bound, 'Scores'] = upper_bound
print(df)
```

```
    Scores
0     70.0
1     75.0
2     80.0
3     85.0
4     90.0
5    107.5
```

이상치 처리 (IQR, Z-Score)

- 이상치 처리

  - Z-Score 방법
    - Z-Score는 데이터의 값이 평균에서 얼마나 떨어져 있는지를 표준 편차 단위로 나타냄
      - 정규분포를 따르는 데이터에 적합.
      - 계산이 간단함.
    - Z= (X−μ)/σ
      - X: 개별 데이터 값
      - μ: 평균.
      - σ: 표준 편차.
    - 이상치 기준
      - 일반적으로 |Z|>3이면 이상치로 간주.
      - 여기서는 |Z|>2 또는 |Z|<-2

```python
import pandas as pd
from scipy.stats import zscore

# 예제 데이터
data = {'Scores': [70, 75, 80, 85, 90, 1000000]}
df = pd.DataFrame(data)

# Z-Score 계산
df['Z_Score'] = zscore(df['Scores'])

# 이상치 확인
outliers = df[(df['Z_Score'] < -2) | (df['Z_Score'] > 2)]
print(outliers)

# 이상치 제거
#df = df[(df['Z_Score'] > -2) & (df['Z_Score'] < 2)]

# 이상치를 중간값으로 대체
median = df['Scores'].median()
df.loc[(df['Z_Score'] < -2) | (df['Z_Score'] > 2), 'Scores'] =
median

print(df)
```

```
    Scores   Z_Score
0     70.0 -0.447240
1     75.0 -0.447227
2     80.0 -0.447214
3     85.0 -0.447200
4     90.0 -0.447187
5     82.5  2.236068
```

데이터 중복 확인 및 제거

- **중복 데이터**

  - 중복 데이터는 데이터셋에서 동일한 행(row)이 반복적으로 나타나는 것
  - 중복 데이터가 있으면 분석 결과가 왜곡될 수 있으며, 특히 데이터의 크기가 큰 경우 처리 속도와 메모리 사용량에 영향을 줄 수 있음

  - 중복 데이터는 다음과 같이 나타날 수 있음
    - 완전 중복: 모든 열의 값이 동일한 행.
    - 부분 중복: 특정 열의 값만 동일한 행.

```
ID   Name    Age   Score
1    Alice   25    85
2    Bob     30    90
1    Alice   25    85   <- 중복된 데이터
```

데이터 중복 확인 및 제거

- 중복 데이터

```
# pandas 중복 데이터 확인 및 제거
import pandas as pd
# 예제 데이터
data = {'Name': ['Alice', 'Bob', 'Alice', 'Charlie', 'Bob'],
        'Age': [25, 30, 25, 35, 30],
        'Score': [85, 90, 85, 95, 90]}

df = pd.DataFrame(data)
print('원본 데이터프레임:')
print(df)
```

데이터 중복 확인 및 제거

- 중복 데이터

```
# DataFrame 내 중복된 행 확인
print('중복 데이터 확인:')
print(df.duplicated())  # 각 행이 이전에 등장한 행과 중복되는지 여부를 Boolean 값으로 출력
print('*' * 50)

# 전체 중복 행 제거 (모든 열 값이 동일한 경우)
df_no_duplicates = df.drop_duplicates(keep='first')  # 기본값은 keep='first': 첫 번째 중복은 남기고 나머지는 제거
print('1. 중복 데이터 제거 후:')
print(df_no_duplicates)
print('*' * 50)

# 특정 열('Name') 기준으로 중복 행 제거
df_no_duplicates_by_name = df.drop_duplicates(subset=['Name'])  # 이름이 같은 경우, 첫 번째만 남기고 제거
print('2. 이름 기준으로 중복 데이터 제거 후:')
print(df_no_duplicates_by_name)
print('*' * 50)

# 특정 열('Name') 기준 중복 제거 + 마지막 항목을 남기고 이전 항목 제거
df_no_duplicates_by_name_last = df.drop_duplicates(subset=['Name'], keep='last')  # 마지막 중복 항목을 유지
print('3. 이름 기준으로 중복 데이터 제거 후, 마지막 항목 유지:')
print(df_no_duplicates_by_name_last)
print('*' * 50)

# 특정 열('Name') 기준으로 완전한 중복 제거 (중복된 모든 항목 삭제)
df_no_duplicates_by_name_all = df.drop_duplicates(subset=['Name'], keep=False)  # 중복된 항목 전부 제거
print('4. 이름 기준으로 중복 데이터 제거 후, 모든 중복 항목 제거:')
print(df_no_duplicates_by_name_all)
```

first coding

데이터 변환 및 스케일링

데이터 변환 (로그 변환, 스퀘어 루트 변환)

데이터 스케일링 (표준화, 정규화)

범주형 데이터 처리 (레이블 인코딩, 원-핫 인코딩)

데이터 변환 (로그 변환, 스퀘어 루트 변환)

- 데이터 변환

  - 데이터의 분포를 변화시켜 분석이나 모델링에 유리한 형태로 만드는 과정
  - **로그 변환**과 **스퀘어 루트 변환**은 데이터를 처리할 때 자주 사용되는 기법
  - 특히 데이터의 **왜도(Skewness)**를 줄이고 더 정규 분포에 가깝게 만들기 위해 활용

데이터 변환 (로그 변환, 스퀘어 루트 변환)

- **로그 변환 (Log Transformation)**

  - 데이터 값의 로그(log)를 취하는 방법
  - 데이터 값이 매우 크거나, 몇 개의 값이 지나치게 크고 나머지는 작은 경우
    → (오른쪽으로 치우친 분포)를 다룰 때 효과적

  - **로그 변환을 하면**
    - 큰 값은 축소되고,
    - 작은 값은 상대적으로 더 커져
    - 데이터 분포를 평탄하게 만들어 줌

데이터 변환 (로그 변환, 스퀘어 루트 변환)

- 로그 변환 (Log Transformation)

  - 예시 : **회사의 매출 데이터**
    - 회사의 매출 데이터

      [1000, 2000, 5000, 10000, 100000]

      → 이 데이터는 몇 개의 큰 값(100000) 때문에 매우 오른쪽으로 치우쳐 있음

    - **이 데이터들을 로그 변환하면**

      → 로그 변환: log(1000), log(2000), log(5000), log(10000), log(100000)

      로그 값(기본 자연 로그, ln 사용 시)

      → [6.91, 7.60, 8.52, 9.21, 11.51]

데이터 변환 (로그 변환, 스퀘어 루트 변환)

- **로그 변환** (Log Transformation)

  - 예시 : 회사의 매출 데이터 (로그 변환)
    - 장점
      - 큰 값의 영향을 줄이고 작은 값의 상대적 차이를 부각할 수 있음
      - 분포의 왜도를 줄이고, 모델링 시 더 나은 예측 성능을 얻을 수 있음

    - 주의점
      - 데이터에 0이나 음수가 있으면 로그를 계산할 수 없기 때문에,

        → 변환 전에 **값에 작은 상수(예: 1)** 를 더해야 함

        예: $\log(x+1)$

데이터 변환 (로그 변환, 스퀘어 루트 변환)

- 스퀘어 루트 변환 (Square Root Transformation)

  - 데이터의 제곱근을 구하는 방법
  - 로그 변환보다는 덜 극단적인 방법
  - 데이터의 크기를 축소하거나 오른쪽 치우침을 완화하는 데 사용

데이터 변환 (로그 변환, 스퀘어 루트 변환)

- 스퀘어 루트 변환 (Square Root Transformation)

  - 예제: 학생들의 시험 점수

    - 데이터: [1, 4, 16, 36, 100]
      → 이 데이터는 몇몇 값이 너무 커서 데이터가 불 균형하게 보일 수 있음

    - 스퀘어 루트 변환
      → 변환 값 = [1, 2, 4, 6, 10]

      결과
      [1, 2, 4, 6, 10]

데이터 변환 (로그 변환, 스퀘어 루트 변환)

- **스퀘어 루트 변환 (Square Root Transformation)**

  - **예제: 학생들의 시험 점수**

    - 장점
      - 데이터의 스케일을 줄이면서도 값 간의 상대적 관계를 유지
      - 비선형적 관계를 일부 선형화 할 수 있음

    - 주의점
      - 음수 값에 사용할 수 없습니다. (**음수에는 제곱근이 실수로 정의되지 않음**)

데이터 변환 (로그 변환, 스퀘어 루트 변환)

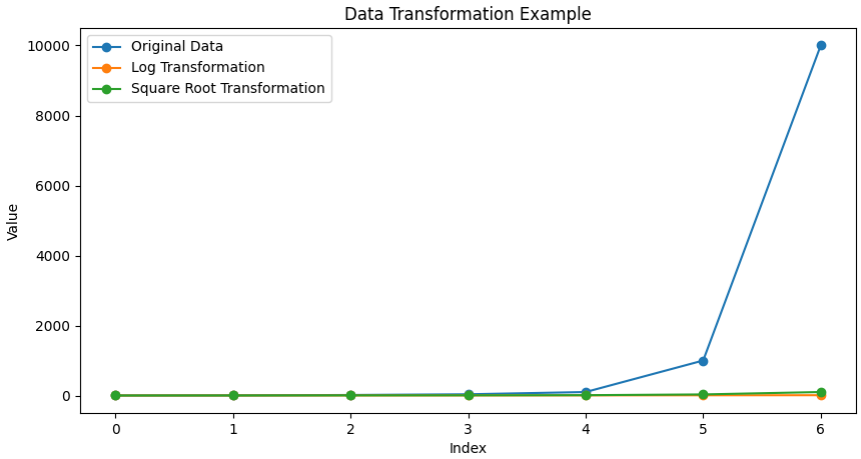
- 데이터 변환

  - 로그 변환과 스퀘어 루트 변환 비교

특성	로그 변환	스퀘어 루트 변환
적용 대상	매우 큰 값, 오른쪽 치우친 분포	작은 값과 큰 값이 공존하는 분포
효과	큰 값을 더 크게 줄임	큰 값을 약간 줄임
수식	y=log(x)	y= x
제약 조건	음수/0에는 사용 불가	음수에는 사용 불가

데이터 변환 (로그 변환, 스퀘어 루트 변환)

- 데이터 변환 예제




```python
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

# 예제 데이터
data = pd.DataFrame({'Original': [1, 4, 16, 36, 100, 1000, 10000]})

# 로그 변환
data['Log Transformation'] = np.log1p(data['Original'])  # log(x + 1)

# 스퀘어 루트 변환
data['Square Root Transformation'] = np.sqrt(data['Original'])

# 데이터 시각화
plt.figure(figsize=(10, 5))
plt.plot(data['Original'], label='Original Data', marker='o')
plt.plot(data['Log Transformation'], label='Log Transformation',
marker='o')
plt.plot(data['Square Root Transformation'], label='Square Root
Transformation', marker='o')
plt.legend()
plt.title('Data Transformation Example')
plt.xlabel('Index')
plt.ylabel('Value')
plt.show()
```

데이터 변환 (로그 변환, 스퀘어 루트 변환)

- 데이터 변환

  - 정리
    - 로그 변환은 매우 큰 값을 다룰 때 효과적
    - 스퀘어 루트 변환은 큰 값과 작은 값의 균형을 맞추는 데 유용
    - 두 방법 모두 데이터의 분포를 조정하고, 모델 학습 성능을 개선하는 데 중요한 역할

데이터 스케일링 (표준화, 정규화)

- **데이터 스케일링: 표준화와 정규화**

  - 데이터 스케일링은 데이터의 값 범위를 조정하는 과정
  - 머신러닝 모델이 데이터의 크기 차이에 영향을 덜 받게 하기 위해 중요함
  - 특히 변수의 단위나 크기가 다를 때, 스케일링은 모델 학습에 있어 필수적인 과정

데이터 스케일링 (표준화, 정규화)

- 데이터 스케일링

  - 표준화 (Standardization)

    - **표준화는 데이터의 값을 평균이 0, 표준편차가 1이 되도록 변환하는 방법**
      - 데이터를 똑같은 기준선에 맞춰서 공정하게 비교하게 해주는 과정
      - **평균** : 데이터들의 중심값
      - **표준편차** : 데이터가 평균에서 얼마나 떨어져 있는지 측정한 값
    - 데이터의 분포 모양을 바꾸지는 않고, 단위 차이를 없애 주는 것
    - 수식
      - x: 원래 데이터 값
      - μ: 평균값
      - σ: 표준편차

$$z = \frac{x - \mu}{\sigma}$$

데이터 스케일링 (표준화, 정규화)

- 데이터 스케일링

수식 $$z = \frac{x - \mu}{\sigma}$$

- x: 원래 데이터 값
- μ: 평균값
- σ: 표준편차

  - **표준화 (Standardization)**
    - 예제
      - 학생들의 시험 점수 : [50, 80, 90, 100, 60]
        - » 평균

          (50+80+90+100+60) / 5 = 76

        - » 표준편차

          약 18.26

        - » 표준화 결과

          $$z = \frac{x - 76}{18.26}$$

          결과: [-1.42, 0.22, 0.77, 1.32, -0.88]

데이터 스케일링 (표준화, 정규화)

- 데이터 스케일링

  - **표준화 (Standardization)**
    - 장점
      - 데이터가 서로 다른 범위와 단위를 가지더라도, 동일한 기준으로 변환
      - 특히 거리 기반 알고리즘(예: KNN, SVM)에서 효과적
    - 주의점
      - 데이터가 정규 분포를 따르지 않더라도 표준화를 적용할 수 있지만,
      - 분포의 왜도는 남아 있을 수 있음

데이터 스케일링 (표준화, 정규화)

- 데이터 스케일링

  - 정규화 (Normalization)
    - 정규화는 데이터를 0과 1 사이로 변환하는 방법
    - 값의 크기를 줄이고, 데이터를 일정한 스케일로 맞춤

    - 수식 (Min-Max Scaling)

$$x' = \frac{x - \min(x)}{\max(x) - \min(x)}$$

      - x: 원래 데이터 값
      - $\min(x)$: 데이터의 최소값
      - $\max(x)$: 데이터의 최대값

데이터 스케일링 (표준화, 정규화)

- 데이터 스케일링

  - 정규화 (Normalization)
    - 예제 :직원의 연봉 데이터
      [3000, 4000, 10000, 15000, 20000]

      - 최소값: 3000, 최대값: 20000
      - 정규화 결과

$$x' = \frac{x - 3000}{20000 - 3000}$$

결과: [0, 0.0588, 0.4118, 0.7059, 1]

수식 (Min-Max Scaling)

$$x' = \frac{x - \min(x)}{\max(x) - \min(x)}$$

- x: 원래 데이터 값
- $\min(x)$: 데이터의 최소값
- $\max(x)$: 데이터의 최대값

데이터 스케일링 (표준화, 정규화)

- 데이터 스케일링

  - 정규화 (Normalization)
    - 장점
      - 데이터가 일정 범위 내에 있어, 스케일에 민감한 알고리즘(예: **신경망, 회귀**)에서 적합
      - 이상치(outlier)의 영향을 줄이는 데도 유용함
    - 주의점
      - 데이터에 이상치가 있을 경우, 최대값과 최소값이 크게 왜곡될 수 있음

        → 데이터 스케일일 전 이상치에 대한 처리가 선행되어야 함!

데이터 스케일링 (표준화, 정규화)

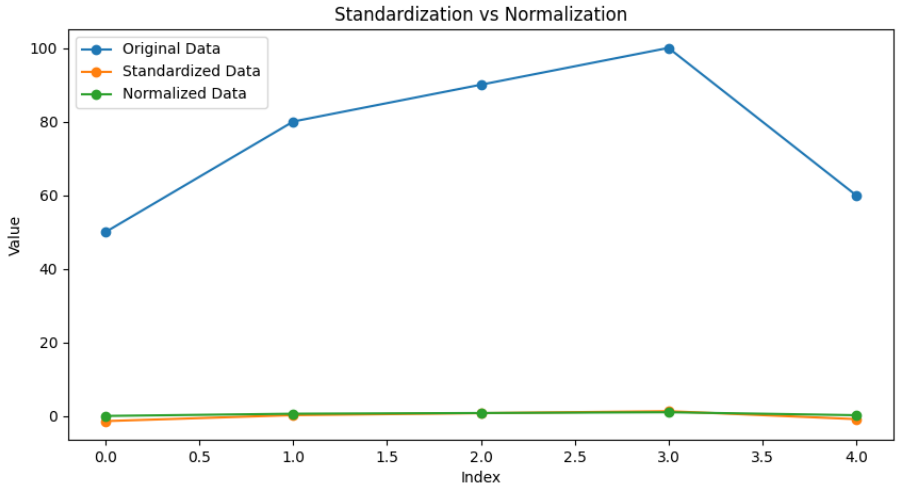
- 데이터 스케일링

  - 표준화 vs. 정규화 비교

특성	표준화 (Standardization)	정규화 (Normalization)
적용 후 범위	평균 0, 표준편차 1	0에서 1 사이
사용 시기	정규 분포에 가까운 데이터	값의 범위가 클 때
알고리즘 적용	거리 기반 알고리즘(KNN, SVM 등)	신경망, 회귀 등
이상치에 대한 민감도	덜 민감	더 민감

데이터 스케일링 (표준화, 정규화)

- 데이터 스케일링 예제




```python
import numpy as np
import pandas as pd
from sklearn.preprocessing import StandardScaler, MinMaxScaler
import matplotlib.pyplot as plt

# 데이터 생성
data = pd.DataFrame({'Original': [50, 80, 90, 100, 60]})

# 표준화
scaler_standard = StandardScaler()
data['Standardized'] =
scaler_standard.fit_transform(data[['Original']])

# 정규화
scaler_minmax = MinMaxScaler()
data['Normalized'] = scaler_minmax.fit_transform(data[['Original']])

# 데이터 확인
print(data)
# 데이터 시각화
plt.figure(figsize=(10, 5))
plt.plot(data['Original'], label='Original Data', marker='o')
plt.plot(data['Standardized'], label='Standardized Data', marker='o')
plt.plot(data['Normalized'], label='Normalized Data', marker='o')
plt.legend()
plt.title('Standardization vs Normalization')
plt.xlabel('Index')
plt.ylabel('Value')
plt.show()
```

데이터 스케일링 (표준화, 정규화)

- 데이터 스케일링

  - **표준화 사례 :** 병원 데이터 분석
    - 환자의 체온, 심박수, 혈압 등이 모두 다른 단위를 가집니다.

      예: 체온(35-40°C), 심박수(60-100 bpm), 혈압(80-120 mmHg).

      이러한 데이터는 표준화를 통해 단위를 없애고 공통 스케일로 변환해야 함

  - **정규화 사례 :** 이미지 데이터 전처리
    - 이미지 픽셀 값은 0~255 범위

      → 0~1 사이로 정규화 하면 더 안정적이고 빠른 학습 가능

범주형 데이터 처리 (레이블 인코딩, 원-핫 인코딩)

- **범주형 데이터 처리**

  - **범주형 데이터**

    → 수치가 아닌 텍스트 데이터(예: "사과", "배", "포도")처럼 특정 범주를 나타내는 데이터

  - 머신러닝 알고리즘은 숫자 데이터를 처리하기 때문에, 범주형 데이터를 숫자로 변환해야 함
  - 여기서 두 가지 대표적인 방법이 **레이블 인코딩과 원-핫 인코딩**

범주형 데이터 처리 (레이블 인코딩, 원-핫 인코딩)

- 범주형 데이터 처리

  - 레이블 인코딩 (Label Encoding)
    - 범주형 데이터를 정수로 매핑하여 각 범주를 고유한 숫자로 변환하는 방법

    - 방법
      데이터의 범주마다 고유한 번호를 부여

      예: "사과" → 0,
      "배" → 1,
      "포도" → 2

범주형 데이터 처리 (레이블 인코딩, 원-핫 인코딩)

- 범주형 데이터 처리

  - 레이블 인코딩 (Label Encoding)
    - 예제 : 과일 데이터: ["사과", "배", "포도", "사과", "포도"]
    - 레이블 인코딩
      - "사과" → 0
      - “배" → 1
      - “포도“ → 2
    - 결과: [0, 1, 2, 0, 2]

범주형 데이터 처리 (레이블 인코딩, 원-핫 인코딩)

- 범주형 데이터 처리

  - 레이블 인코딩 (Label Encoding)
    - 장점
      - 간단하고 직관적이며, 메모리를 적게 사용
    - 단점
      - **범주의 순서가 없는데도 숫자가 순서를 암시할 수 있습니다.**

        예: "사과"(0)가 "배"(1)보다 더 작거나 낮다는 의미를 부여할 위험이 있음
    - 사용 시기
      - 범주가 순서형 데이터인 경우(예: "낮음", "보통", "높음") 적합함

범주형 데이터 처리 (레이블 인코딩, 원-핫 인코딩)

- 범주형 데이터 처리

  - 원-핫 인코딩 (One-Hot Encoding)
    - 범주형 데이터를 이진 벡터로 변환
    - 각 범주마다 별도의 열(column)을 만들고, 해당 범주에만 1을 표시하고 나머지는 0으로 표시

    - 방법
      - 각 범주를 새로운 열로 추가하고,
      - 데이터가 해당 범주에 속하면 1, 아니면 0으로 표시

범주형 데이터 처리 (레이블 인코딩, 원-핫 인코딩)

- 범주형 데이터 처리

  - 원-핫 인코딩 (One-Hot Encoding)
    - 예제: 과일 데이터: ["사과", "배", "포도", "사과", "포도"]
    - 원-핫 인코딩
      - "사과" → [1, 0, 0]
      - "배" → [0, 1, 0]
      - "포도" → [0, 0, 1]
    - 결과

사과	배	포도
1	0	0
0	1	0
0	0	1
1	0	0
0	0	1

범주형 데이터 처리 (레이블 인코딩, 원-핫 인코딩)

- 범주형 데이터 처리

  - 원-핫 인코딩 (One-Hot Encoding)
    - 장점
      - 숫자가 범주의 순서를 암시하지 않으므로, 레이블 인코딩의 단점을 극복
    - 단점
      - 고차원 문제: 범주의 수가 많아질수록 많은 열이 생성되어 메모리 소모가 큼
        예: 1000개의 범주가 있다면 1000개의 열이 추가됨
    - 사용 시기
      - 범주가 순서가 없는 데이터(예: "사과", "배", "포도")인 경우 적합함

범주형 데이터 처리 (레이블 인코딩, 원-핫 인코딩)

- 범주형 데이터 처리

  - 레이블 인코딩 vs 원-핫 인코딩 비교

특성	레이블 인코딩	원-핫 인코딩
출력 형식	정수 (0, 1, 2, ...)	이진 벡터 ([1, 0, 0], ...)
순서 정보 암시	있음	없음
고차원 문제	없음	범주가 많으면 열이 많이 생성됨
사용 적합성	순서형 데이터	비순서형 데이터

범주형 데이터 처리 (레이블 인코딩, 원-핫 인코딩)

- 범주형 데이터 처리 예제

```
import pandas as pd
from sklearn.preprocessing import LabelEncoder, OneHotEncoder

# 데이터 생성
data = pd.DataFrame({'Fruits': ['사과', '배', '포도', '사과', '포도']})

# Step 1: 레이블 인코딩
label_encoder = LabelEncoder()
data['Label Encoding'] = label_encoder.fit_transform(data['Fruits'])

# Step 2: 원-핫 인코딩 (OneHotEncoder 사용, sparse_output=False로 개선)
one_hot_encoder = OneHotEncoder(sparse_output=False)
one_hot_encoded = one_hot_encoder.fit_transform(data[['Label Encoding']])

# Step 3: 결과를 DataFrame으로 변환
one_hot_columns = one_hot_encoder.get_feature_names_out()
one_hot_df = pd.DataFrame(one_hot_encoded, columns=one_hot_columns)

# Step 4: 원래 데이터에 추가
data = pd.concat([data, one_hot_df], axis=1)
# 출력
print(data)
```

	Fruits	Label Encoding	Label Encoding_0	Label Encoding_1	Label Encoding_2
0	사과	1	0.0	1.0	0.0
1	배	0	1.0	0.0	0.0
2	포도	2	0.0	0.0	1.0
3	사과	1	0.0	1.0	0.0
4	포도	2	0.0	0.0	1.0