2026.01.28.수

생성일	@2026년 1월 28일 오전 9:17
태그	

목차

1. 데이터 전처리
2. 결측치 처리
3. 데이터 타입 변환
4. 이상치 탐지 및 제거
5. 중복 데이터 제거
6. 데이터 정규화 및 표준화

▼ 1) 데이터 전처리

- 중요성
  - 분석 정확성 보장: 결측치, 이상치, 중복 데이터는 분석 결과 왜곡
  - 모델 성능 향상
  - 자원 효율성: 중복 데이터나 불필요한 변수 제거로 컴퓨팅 리소스 절약
- 데이터 확인 및 탐색 (EDA)
  - df.info()
  - df.describe()
  - df.isnull().sum()
- 결측치 처리 (Missing Values)
- 데이터 정제 (Data Cleaning)
  - 중복 데이터 제거: df.drop_duplicates()
  - 데이터 타입 변환
  - 텍스트 정규화(포맷팅)
- 데이터 처리 3단계
  1. 데이터 수집
  2. 데이터 전처리
     - 결측치 처리
     - 이상치 제거
     - 데이터 타입 변환
     - 중복 데이터 제거
     - 데이터 정규화/표준화
  3. 데이터 가공
     - 특성 공학
     - 데이터 집계 및 그룹화
     - 파생 변수 생성
     - 데이터 변환 및 인코딩

▼ 2) 결측치 처리

- 결측치 확인 T/F

```
import pandas as pd
import numpy as np

# 결측치가 있는 데이터프레임 생성
# np.nan을 사용하여 데이터가 없는 빈 칸(결측치)을 인위적으로 만듭니다.
df = pd.DataFrame({
    'A': [1, 2, np.nan, 4, 5],
    'B': [np.nan, 2, 3, 4, 5],
    'C': [1, 2, 3, np.nan, np.nan]
})

print("원본 데이터:")
print(df)

# 결측치 확인
# isna()는 각 요소가 결측치(NaN)인지 확인하여 True(결측치) 또는 False를 반환합니다.
print("\n결측치 여부:")
print(df.isna())
```

- 결측치 제거

```
# 결측치 개수 확인
print("\n열별 결측치 개수:")
# isna()로 찾은 결측치(True)들을 sum()으로 더해 각 열의 결측치 총합을 구합니다.
print(df.isna().sum())

# 결측치 처리 방법 1: 삭제
# dropna()는 결측치가 하나라도 포함된 행(row)을 전체 데이터셋에서 삭제합니다.
df_dropped = df.dropna()  # 결측치가 있는 행 모두 삭제

print("\n결측치 행 삭제 후:")
print(df_dropped)
```

- 결측치 대체

```
# 결측치 처리 방법 2: 채우기
# fillna(값)는 데이터프레임 내의 모든 NaN을 지정한 값으로 바꿉니다.
df_filled = df.fillna(0)  # 결측치를 0으로 채우기
print("\n결측치를 0으로 채운 후:")
print(df_filled)

# 결측치 처리 방법 3: 열별 평균으로 채우기
# df.mean()을 인자로 주면 각 열의 산술 평균값을 계산하여 해당 열의 빈칸을 채웁니다.
df_mean = df.fillna(df.mean())  # 각 열의 평균값으로 채우기
print("\n결측치를 평균으로 채운 후:")
print(df_mean)
```

▼ 3) 데이터 타입 변환

```
import pandas as pd

# 잘못된 데이터 타입이 포함된 데이터프레임 생성
df = pd.DataFrame({
    'A': ['1', '2', '3', '4', '5'],              # 숫자이지만 문자열로 저장됨
    'B': [1.1, 2.2, 3.3, 4.4, 5.5],             # 이미 올바른 float 타입
    'C': ['2020-01-01', '2020-02-01', '2020-03-01', # 날짜이지만 문자열로 저장됨
          '2020-04-01', '2020-05-01'],
    'D': ['True', 'False', 'True', 'False', 'True'] # 불리언이지만 문자열로 저장됨
})
```

```
print("원본 데이터 타입:")
# dtypes 속성은 데이터프레임 내 각 열의 데이터 타입을 보여줍니다.
print(df.dtypes)

# 결과 분석:
# A, C, D가 모두 'object'(문자열) 타입으로 표시됨
# 이 상태로는 수치 연산, 날짜 연산, 논리 연산이 불가능함

# 1. 문자열 -> 정수 변환
# astype(int): 문자열 '1', '2' -> 정수 1, 2로 변환합니다.
df['A'] = df['A'].astype(int)
print(df['A'])
# 이제 수학적 연산(덧셈, 곱셈 등)이 가능해집니다.

# 2. 문자열 -> 날짜/시간 변환
# pd.to_datetime(): 문자열 '2020-01-01' -> datetime 객체로 변환합니다.
df['C'] = pd.to_datetime(df['C'])
print(df['C'])
# 날짜 연산(날짜 차이 계산, 월/년 추출 등)이 가능해집니다.

# 3. 문자열 -> 불리언 변환 (주의사항 있음)
df['D'] = df['D'].astype(bool)
print(df['D'])
# 주의: 문자열 'False'도 True로 변환됨 (비어있지 않은 문자열이므로)
```

- 불리언 타입으로 바꿀 때 모두 True가 되어버리는 문제 해결

```
# 불리언 타입으로 변경시 모두 True가 되어버리는 문제 해결

# 원본 데이터 생성 (문자열 형태의 불리언 값들)
df_original = pd.DataFrame({
    'D': ['True', 'False', 'True', 'False', 'True']
})

# 방법 1: map() 사용
# 딕셔너리를 사용하여 특정 문자열을 실제 불리언 값으로 1:1 매칭시킵니다.
df_original['D_correct1'] = df_original['D'].map({'True': True, 'False': False})
print(df_original['D_correct1'])

# 방법 2: 조건문 사용
# 'True'라는 문자열과 같은지 비교 연산을 수행하여 그 결과를 불리언 시리즈로 만듭니다.
df_original['D_correct2'] = (df_original['D'] == 'True')
print(df_original['D_correct2'])
```

▼ 4) 이상치 탐지 및 제거

```
import pandas as pd
import numpy as np

# 이상치가 포함된 샘플 데이터 생성
# 결과의 재현성을 위해 시드(Seed)를 고정합니다.
np.random.seed(42)

# 정상적인 분포를 가진 데이터 95개 생성 (평균 50, 표준편차 10)
normal_data = np.random.normal(50, 10, 95)  # 정상 데이터 95개

# 의도적으로 넣을 극단적인 이상치 5개 정의
outliers = [120, 130, -20, -10, 150]       # 이상치 5개
```

```
# 두 데이터를 하나로 합칩니다.
data_with_outliers = np.concatenate([normal_data, outliers])

# 데이터프레임 생성
df = pd.DataFrame({
    'ID': range(1, 101),
    'Score': data_with_outliers,
    'Category': np.random.choice(['A', 'B', 'C'], 100)
})

print("원본 데이터 통계:")
# describe()를 통해 이상치가 포함된 전체 데이터의 분포 확인
print(df['Score'].describe())

def detect_outliers_iqr(data):
    """IQR 방법으로 이상치 탐지"""
    Q1 = data.quantile(0.25)    # 1사분위수 (하위 25%)
    Q3 = data.quantile(0.75)    # 3사분위수 (상위 75%)
    IQR = Q3 - Q1               # 사분위수 범위

    # 이상치 경계값 계산 (일반적으로 1.5 * IQR 사용)
    lower_bound = Q1 - 1.5 * IQR
    upper_bound = Q3 + 1.5 * IQR

    return lower_bound, upper_bound

# 이상치 식별
    # 하한선보다 작거나(|) 상한선보다 큰 데이터를 이상치로 정의합니다.
    outliers = (data < lower_bound) | (data > upper_bound)

    return outliers, lower_bound, upper_bound

# 함수 호출: 'Score' 열의 이상치 마스크와 경계값들을 받아옵니다.
outliers_mask, lower, upper = detect_outliers_iqr(df['Score'])

# 결과 출력
print(f"\nIQR 방법 - 이상치 경계: {lower:.2f} ~ {upper:.2f}")
print(f"이상치 개수: {outliers_mask.sum()}개")
print("이상치 값들:")
# 불리언 인덱싱을 사용하여 이상치에 해당하는 실제 값들만 추출하여 출력합니다.
print(df[outliers_mask]['Score'].values)

# 방법 2: Z-Score 방법으로 이상치 탐지
def detect_outliers_zscore(data, threshold=3):
    """Z-Score 방법으로 이상치 탐지 (기본 임계값: 3)"""
    # 각 데이터에서 평균을 빼고, 표준편차로 나누어 표준화(표준정규분포로 변환)
    # abs() : 절대값 반환
    z_scores = np.abs((data - data.mean()) / data.std())

    # 설정한 임계값(threshold)보다 큰 데이터를 이상치로 판단
    outliers = z_scores > threshold
    return outliers, z_scores

# 함수 호출: 'Score' 열의 이상치 여부와 Z-Score 값들을 받아옵니다.
outliers_zscore, z_scores = detect_outliers_zscore(df['Score'])

print(f"\nZ-Score 방법 - 이상치 개수: {outliers_zscore.sum()}개")
print("Z-Score가 3 이상인 값들:")
# 불리언 인덱싱을 사용하여 이상치에 해당하는 ID와 Score 값을 출력합니다.
```

```
print(df[outliers_zscore][['ID', 'Score']].values)

# 방법 2: Z-Score 방법으로 이상치 탐지
def detect_outliers_zscore(data, threshold=3):
    """Z-Score 방법으로 이상치 탐지 (기본 임계값: 3)"""
    # 각 데이터에서 평균을 빼고, 표준편차로 나누어 표준화(표준정규분포로 변환)
    # abs() : 절대값 반환
    z_scores = np.abs((data - data.mean()) / data.std())

    # 설정한 임계값(threshold)보다 큰 데이터를 이상치로 판단
    outliers = z_scores > threshold
    return outliers, z_scores

# 함수 호출: 'Score' 열의 이상치 여부와 Z-Score 값들을 받아옵니다.
outliers_zscore, z_scores = detect_outliers_zscore(df['Score'])

print(f"\nZ-Score 방법 - 이상치 개수: {outliers_zscore.sum()}개")
print("Z-Score가 3 이상인 값들:")
# 불리언 인덱싱을 사용하여 이상치에 해당하는 ID와 Score 값을 출력합니다.
print(df[outliers_zscore][['ID', 'Score']].values)

# IQR 방법으로 탐지된 이상치 제거
# ~outliers_mask : outliers_mask가 True인 행(이상치)을 제외한 나머지 행들만 선택
df_no_outliers = df[~outliers_mask].copy()

# copy() : 원본 데이터프레임을 복사하여 이상치가 제거된 새로운 데이터프레임 생성
# (SettingWithCopyWarning을 방지하고 원본 보존을 위해 권장되는 방식입니다)

print(f"\n이상치 제거 전 데이터 크기: {len(df)}")
print(f"이상치 제거 후 데이터 크기: {len(df_no_outliers)}")

print("\n이상치 제거 후 통계:")
# 정제된 데이터의 'Score' 열에 대한 요약 통계량을 출력하여 분포 변화를 확인합니다.
print(df_no_outliers['Score'].describe())
```

▼ 5) 중복 데이터 제거

```
import pandas as pd
import numpy as np

# 중복 데이터가 포함된 샘플 데이터 생성
data = {
    'Name': ['Alice', 'Bob', 'Charlie', 'Alice', 'David', 'Bob', 'Eve', 'Charlie'],
    'Age': [25, 30, 35, 25, 40, 30, 28, 35],
    'City': ['Seoul', 'Busan', 'Seoul', 'Seoul', 'Daegu', 'Busan', 'Seoul', 'Seoul'],
    'Salary': [50000, 60000, 70000, 50000, 80000, 65000, 55000, 70000]
}

df = pd.DataFrame(data)

print("원본 데이터:")
print(df)
# len() 함수를 사용하여 전체 데이터의 행(row) 개수를 출력합니다.
print(f"\n원본 데이터 크기: {len(df)}행")

# 1. 완전 중복 행 확인
print("\n=== 완전 중복 행 탐지 ===")

# duplicated()는 모든 열의 값이 동일한 행을 찾아 True/False로 반환합니다.
duplicate_rows = df.duplicated()  # 모든 열이 동일한 행 탐지
```

```
print("중복 행 여부:")
print(duplicate_rows)

# True(1)의 합계를 구하여 중복된 행의 총 개수를 확인합니다.
print(f"완전 중복 행 개수: {duplicate_rows.sum()}개")

# 중복된 행이 하나라도 있다면 해당 행들의 내용을 출력합니다.
if duplicate_rows.sum() > 0:
    print("중복된 행들:")
    print(df[duplicate_rows])

# 2. 특정 열 기준 중복 확인
print("\n=== 특정 열 기준 중복 탐지 ===")

# subset 파라미터를 사용하여 'Name' 열만 기준으로 중복을 체크합니다.
# 이름이 같으면 다른 정보(급여 등)가 달라도 중복으로 간주합니다.
name_duplicates = df.duplicated(subset=['Name'])  # 이름 기준 중복
print("이름 기준 중복 행:")
print(df[name_duplicates])

# 여러 개의 열을 리스트로 전달하여 '이름'과 '나이'가 모두 같은 경우만 중복으로 체크합니다.
name_age_duplicates = df.duplicated(subset=['Name', 'Age'])  # 이름+나이 기준 중복
print("\n이름+나이 기준 중복 행:")
print(df[name_age_duplicates])

# 방법 1: 완전 중복 행 제거 (첫 번째 발생 유지)
# 모든 열의 값이 동일한 행을 찾아 삭제하며, 기본적으로 첫 번째 데이터는 남겨둡니다.
df_no_duplicates = df.drop_duplicates()
print(f"완전 중복 제거 후: {len(df_no_duplicates)}행")
# 중복된 행 중 첫번째 행만 유지
print(df_no_duplicates)

# 방법 2: 특정 열 기준 중복 제거
# subset 파라미터를 사용하여 'Name' 열이 중복되는 경우 해당 행을 삭제합니다.
df_unique_names = df.drop_duplicates(subset=['Name'])
print(f"\n이름 기준 중복 제거 후: {len(df_unique_names)}행")
# 중복된 행 중 첫번째 행만 유지
print(df_unique_names)

# 4. 조건부 중복 제거 (더 높은 급여 유지)
print("\n=== 조건부 중복 제거 (최고 급여 유지) ===")

# groupby로 이름을 묶고, 각 이름별로 'Salary'가 가장 높은 행의 인덱스(idxmax)를 추출합니다.
# 추출된 인덱스를 loc에 전달하여 해당 행들만 필터링합니다.
df_max_salary = df.loc[df.groupby('Name')['Salary'].idxmax()]

print("각 이름별 최고 급여 데이터만 유지:")
print(df_max_salary.sort_values('Name'))

# 5. 중복 데이터 통계 요약
print("\n=== 중복 데이터 요약 통계 ===")

total_rows = len(df)                                        # 전체 행 개수
unique_rows = len(df.drop_duplicates())                     # 중복 제거 후 유일한 행 개수
duplicate_count = total_rows - unique_rows                  # 중복된 행의 총 개수
duplicate_percentage = (duplicate_count / total_rows) * 100 # 중복 데이터 비율(%)
```

▼ 6) 데이터 정규화 및 표준화

- 표준화
  - 평균 0, 표준편차를 1로 변환
  - 데이터가 정규분포를 따를 때
  - 선형 회귀, 로지스틱 회귀, SVM 등에 적합
  - 이상치가 있어도 사용가능하지만 영향을 받음
  - StandardScaler
- 정규화
  - 0-1 범위로 변환
  - 데이터의 분포가 균등할 때
  - 신경망, KNN 등에 적합
  - 이상치에 매우 민감
  - MinMax Scaler
- 로버스트 스케일링
  - 이상치가 많은 데이터에 적합
  - 중앙값과 IQR 사용으로 이상치에 덜 민감
  - Robust Scaler
- 로그 변환
  - 왜도가 큰 데이터 (한 쪽으로 치우친 분포)
  - 지수적 증가 패턴의 데이터
  - 양수 데이터만 적용 가능
- 온라인 쇼핑몰 고객 데이터 전처리

```
import pandas as pd
import numpy as np
from sklearn.preprocessing import StandardScaler

# 기본 고객 데이터 생성
n_customers = 1000
customer_data = {
    'customer_id': range(1, n_customers + 1),
    'name': [f'Customer_{i}' for i in range(1, n_customers + 1)],
    'age': np.random.normal(35, 12, n_customers).astype(int),
    'gender': np.random.choice(['M', 'F', 'Male', 'Female', 'm', 'f', ''], n_customers),
    'city': np.random.choice(['Seoul', 'Busan', 'Daegu', 'Incheon', 'Gwangju', ''], n_customers),
    'total_purchase': np.random.exponential(50000, n_customers),
    'purchase_count': np.random.poisson(5, n_customers),
    'last_purchase_days': np.random.randint(1, 365, n_customers),
    'membership_level': np.random.choice(['Bronze', 'Silver', 'Gold', 'Platinum', ''], n_customers)
}

# 데이터프레임 생성
# 이전 단계에서 만든 customer_data 딕셔너리를 데이터프레임으로 변환합니다.
df = pd.DataFrame(customer_data)
df_original = df.copy()  # 원본 데이터 보존 (최종 리포트 비교용)

print("=== 1단계: 원본 데이터 탐색 ===")
# df.shape : 데이터의 행(row)과 열(column) 개수를 튜플 형태로 반환합니다.
print(f"데이터 크기: {df.shape}")

print(f"\n데이터 타입:")
# df.dtypes : 각 열이 어떤 데이터 타입(int, float, object 등)인지 보여줍니다.
```

```
print(df.dtypes)

print(f"\n처음 5행:")
# df.head() : 데이터프레임의 상단 5개 데이터를 출력하여 실제 모습을 확인합니다.
print(df.head())

print("1. 결측치 현황:")
# isnull() : 데이터프레임 내의 결측치(NaN)를 확인합니다.
# sum() : 결측치 개수의 합계를 열별로 구합니다.
missing_data = df.isnull().sum()

# 주의: 빈 문자열('')은 결측치로 처리되지 않으므로 사전에 변환이 필요할 수 있습니다.
#

# 전체 행 개수 대비 결측치 비율(%) 계산
missing_percentage = (missing_data / len(df)) * 100

# 결측치 정보 요약 데이터프레임 생성
missing_summary = pd.DataFrame({
    '결측치_개수': missing_data,
    '결측치_비율(%)': missing_percentage
})

# 결측치가 하나라도 있는 열만 필터링하여 출력
print(missing_summary[missing_summary['결측치_개수'] > 0])

# 2. 데이터 타입 문제 확인
print(f"\n2. 데이터 타입 문제:")

# dtype 속성을 통해 'age' 열이 수치형(int 등)으로 잘 설정되었는지 확인합니다.
print(f"나이 데이터 타입: {df['age'].dtype}")

# 데이터의 최솟값과 최댓값을 확인하여 상식적인 범위 안에 있는지 점검합니다.
print(f"나이 범위: {df['age'].min()} ~ {df['age'].max()}")

# 비정상적인 값 추출: 나이가 0세 미만이거나 100세를 초과하는 경우를 찾습니다.
# tolist()를 사용하여 결과값들을 리스트 형태로 깔끔하게 출력합니다.
print(f"비정상적인 나이 값: {df[(df['age'] < 0) | (df['age'] > 100)]['age'].tolist()}")

# 3. 범주형 데이터 일관성 문제
print(f"\n3. 성별 데이터 일관성 문제:")

# unique() : 해당 열에 어떤 종류의 값들이 들어있는지 고유값 목록을 보여줍니다.
print(f"성별 고유값: {df['gender'].unique()}")

# value_counts() : 각 고유값이 몇 번씩 등장하는지 빈도를 계산합니다.
# 이를 통해 오타나 잘못된 표기가 얼마나 섞여 있는지 한눈에 알 수 있습니다.
print(f"성별 값 개수: {df['gender'].value_counts()}")

# 4. 이상치 확인 (구매 금액)
print(f"\n4. 구매 금액 이상치 확인:")

# 1사분위수(Q1)와 3사분위수(Q3)를 구하여 IQR 범위를 계산합니다.
Q1 = df['total_purchase'].quantile(0.25)
Q3 = df['total_purchase'].quantile(0.75)
IQR = Q3 - Q1

# 1.5 * IQR 법칙을 사용하여 이상치 판단을 위한 상한선과 하한선을 정합니다.
outlier_threshold_low = Q1 - 1.5 * IQR
outlier_threshold_high = Q3 + 1.5 * IQR
```

```
# 계산된 경계값을 기준으로 이상치에 해당하는 행들만 필터링합니다.
outliers = df[(df['total_purchase'] < outlier_threshold_low) |
               (df['total_purchase'] > outlier_threshold_high)]

# 결과 출력: 이상치의 개수, 전체 데이터 대비 비중(%), 그리고 경계값 수치 확인
print(f"이상치 개수: {len(outliers)}개 ({len(outliers)/len(df)*100:.1f}%)")
print(f"이상치 범위: {outlier_threshold_low:.0f} 미만 또는 {outlier_threshold_high:.0f} 초과")

# 5. 중복 데이터 확인
print(f"\n5. 중복 데이터 확인:")

# duplicated() : 모든 열의 값이 완벽히 일치하는 완전 중복 행을 찾습니다.
duplicates = df.duplicated()
print(f"완전 중복 행: {duplicates.sum()}개")

# subset=['name'] : 다른 정보가 달라도 'name' 열의 값만 같으면 중복으로 간주합니다.
name_duplicates = df.duplicated(subset=['name'])
print(f"이름 중복: {name_duplicates.sum()}개")

# 비정상적인 나이 값을 중앙값으로 대체
# 1. 정상 범위(0세~100세)에 있는 데이터들만의 중앙값을 계산합니다.
median_age = df[(df['age'] >= 0) & (df['age'] <= 100)]['age'].median()

# 2. .loc를 사용하여 범위를 벗어난 행들을 찾아 계산한 중앙값으로 덮어씁니다.
df.loc[(df['age'] < 0) | (df['age'] > 100), 'age'] = median_age

print(f"정제 후 나이 범위: {df['age'].min()} ~ {df['age'].max()}")
print(f"나이 중앙값으로 대체: {median_age}세")

print("\n2-2. 성별 데이터 표준화")
# 표준화 전, 현재 어떤 지저분한 값들이 섞여 있는지 확인합니다.
print(f"표준화 전 성별 값: {df['gender'].unique()}")

# 성별 데이터 표준화 매핑 딕셔너리 정의
# '원래 값': '바꿀 값' 형태로 짝을 지어줍니다.
gender_mapping = {
    'M': 'Male', 'm': 'Male', 'Male': 'Male',
    'F': 'Female', 'f': 'Female', 'Female': 'Female',
    '': 'Unknown'
}

# map() 함수를 사용하여 딕셔너리 기준에 따라 값을 일괄 변경합니다.
# fillna('Unknown')은 매핑되지 않은 예상치 못한 값이 있을 경우 'Unknown'으로 채워줍니다.
df['gender'] = df['gender'].map(gender_mapping).fillna('Unknown')

# # map(gender_mapping) : gender_mapping 딕셔너리에 따라 값 매핑
# # fillna('Unknown') : 매핑된 값이 None인 경우 'Unknown'으로 대체

print(f"표준화 후 성별 값: {df['gender'].unique()}")
# 최종적으로 정제된 성별 데이터의 분포를 확인합니다.
print(f"성별 분포:\n{df['gender'].value_counts()}")

# 2-3. 도시 데이터 결측치 처리
print("\n2-3. 도시 데이터 결측치 처리")
# 처리 전, 빈 문자열('')이 포함된 현재 도시 분포를 확인합니다.
print(f"결측치 처리 전 도시 분포:\n{df['city'].value_counts()}")

# 빈 문자열을 NaN으로 변환 후 최빈값으로 대체
# 1. Pandas가 결측치로 인식할 수 있도록 빈 문자열('')을 np.nan으로 바꿉니다.
```

```
df['city'] = df['city'].replace('', np.nan)

# 2. mode() 함수를 사용하여 가장 많이 등장하는 도시 이름을 찾습니다.
# [0]을 붙이는 이유는 mode()가 시리즈 형태를 반환하기 때문에 첫 번째 값을 가져오기 위함입니다.
most_common_city = df['city'].mode()[0]

# 3. fillna()를 사용하여 NaN 값들을 찾은 최빈값으로 채워줍니다.
df['city'] = df['city'].fillna(most_common_city)

print(f"결측치 처리 후 도시 분포:\n{df['city'].value_counts()}")
print(f"최빈값 '{most_common_city}'로 결측치 대체")

# 2-4. 멤버십 레벨 결측치 처리
print("\n2-4. 멤버십 레벨 결측치 처리")

# 빈 문자열('')을 'Bronze'로 대체합니다.
# 이는 데이터가 없는 고객을 신규 가입 고객으로 간주하고 기본 등급을 부여하는 방식입니다.
df['membership_level'] = df['membership_level'].replace('', 'Bronze')  # 신규 고객은 Bronze로 설정

# 정제 후 각 등급별 고객 수가 어떻게 변했는지 확인합니다.
print(f"멤버십 레벨 분포:\n{df['membership_level'].value_counts()}")

# 3-1. 구매 금액 이상치 처리
print("\n3-1. 구매 금액 이상치 처리")
print(f"이상치 처리 전 구매 금액 통계:")
# 처리 전 데이터의 분포(평균, 표준편차, 최대값 등)를 먼저 확인합니다.
print(df['total_purchase'].describe())

# IQR 방법으로 이상치 탐지 및 처리
# 데이터의 25% 지점(Q1)과 75% 지점(Q3)을 구합니다.
Q1 = df['total_purchase'].quantile(0.25)
Q3 = df['total_purchase'].quantile(0.75)
IQR = Q3 - Q1

# 이상치를 판단할 하한선(lower_bound)과 상한선(upper_bound)을 설정합니다.
lower_bound = Q1 - 1.5 * IQR
upper_bound = Q3 + 1.5 * IQR

# 이상치를 경계값으로 대체 (Winsorization)
# 하한선보다 작은 값은 하한선으로, 상한선보다 큰 값은 상한선으로 덮어씁니다.
df.loc[df['total_purchase'] < lower_bound, 'total_purchase'] = lower_bound
df.loc[df['total_purchase'] > upper_bound, 'total_purchase'] = upper_bound

print(f"이상치 처리 후 구매 금액 통계:")
# 처리 후 최대값(max)이 upper_bound와 일치하게 변했는지 확인합니다.
print(df['total_purchase'].describe())

# 3-2. 파생 변수 생성
print("\n3-2. 파생 변수 생성")

# 평균 구매 금액 계산
# 총 구매 금액을 구매 횟수로 나누어 '건당 평균 결제액' 열을 만듭니다.
df['avg_purchase_amount'] = df['total_purchase'] / df['purchase_count']

# 구매 횟수가 0인 경우 발생할 수 있는 결측치(NaN)를 0으로 채워줍니다.
df['avg_purchase_amount'] = df['avg_purchase_amount'].fillna(0) # 구매 횟수가 0인 경우

# 고객 세그먼트 생성 (RFM 분석 기반)
# 1. 모든 고객의 등급을 기본적으로 'Regular'로 설정합니다.
df['customer_segment'] = 'Regular'
```

```
# 2. VIP 등급: 상위 20%의 매출액을 기록하고, 최근 30일 이내에 방문한 핵심 고객
df.loc[(df['total_purchase'] > df['total_purchase'].quantile(0.8)) &
       (df['last_purchase_days'] < 30), 'customer_segment'] = 'VIP'

# 3. At_Risk 등급: 하위 20%의 매출액을 기록하거나, 마지막 방문 후 180일이 지난 이탈 위기 고객
df.loc[(df['total_purchase'] < df['total_purchase'].quantile(0.2)) |
       (df['last_purchase_days'] > 180), 'customer_segment'] = 'At_Risk'

# 최종 분류된 고객 세그먼트별 인원수를 출력합니다.
print(f"고객 세그먼트 분포:\n{df['customer_segment'].value_counts()}")

# 1. 순서가 있는 범주형 데이터 (멤버십 레벨) - 라벨 인코딩
# 등급 간의 우열이 있으므로 수치로 매핑합니다.
membership_order = {'Bronze': 1, 'Silver': 2, 'Gold': 3, 'Platinum': 4}
df['membership_level_encoded'] = df['membership_level'].map(membership_order)

# map(membership_order) : membership_order 딕셔너리에 따라 값 매핑
print(df[['membership_level', 'membership_level_encoded']].head())

# 2. 순서가 없는 범주형 데이터 (성별, 도시) - 원-핫 인코딩
# 각 범주를 독립된 열로 만들고 0과 1로 표시합니다.
df_encoded = pd.get_dummies(df, columns=['gender', 'city'], prefix=['gender', 'city'])

print(f"인코딩 후 열 개수: {len(df_encoded.columns)}개")
# 원래 데이터프레임에는 없었던, 새로 생성된 인코딩 열 목록을 확인합니다.
print(f"새로 생성된 열: {[col for col in df_encoded.columns if col not in df.columns]}")
print(df_encoded.head())

# gender가 Male이면, gender_male이 1, 나머지는 0
# city가 서울이면, city_seoul이 1, 나머지는 0

# 4-1. 수치형 데이터 정규화
print("\n4-1. 수치형 데이터 정규화")

# 정규화할 수치형 열 선택
numeric_columns = ['age', 'total_purchase', 'purchase_count', 'last_purchase_days', 'avg_purchase_amou
nt']

# 정규화 전 데이터 검증 및 정제
print("정규화 전 데이터 검증:")
for col in numeric_columns:
    # 무한대 값 확인: np.isinf()를 사용해 수학적으로 계산 불가능한 값을 찾습니다.
    inf_count = np.isinf(df[col]).sum()

    # NaN 값 확인: 정제 과정에서 누락된 값이 있는지 다시 점검합니다.
    nan_count = df[col].isnull().sum()

    # 극값 확인: 10의 10승(100억) 이상의 비정상적으로 큰 값이나 작은 값을 탐지합니다.
    extreme_values = df[col][(df[col] > 1e10) | (df[col] < -1e10)]

    print(f"{col}: 무한대 값 {inf_count}개, NaN 값 {nan_count}개, 극값 {len(extreme_values)}개")

    # 무한대 값을 NaN으로 변환: 이후 fillna() 등으로 일괄 처리하기 위해 표준화합니다.
    df[col] = df[col].replace([np.inf, -np.inf], np.nan)

    # 정규화할 수치형 열 선택
numeric_columns = ['age', 'total_purchase', 'purchase_count', 'last_purchase_days', 'avg_purchase_amou
nt']
```

```
# 정규화 전 데이터 검증 및 정제
print("정규화 전 데이터 검증:")
for col in numeric_columns:
    # 무한대 값 확인: 수학적 오류(0으로 나누기 등)로 발생한 무한대 값을 집계합니다.
    inf_count = np.isinf(df[col]).sum()

    # NaN 값 확인: 데이터 누락 여부를 다시 한번 점검합니다.
    nan_count = df[col].isnull().sum()

    # 극값 확인: 상식적인 범위를 벗어난 거대 수치(100억 이상 등)를 탐지합니다.
    extreme_values = df[col][(df[col] > 1e10) | (df[col] < -1e10)]

    # 참고:
    # 1e10 : 10의 10승
    # 1e-10 : 10의 -10승

    # 각 열별 검증 결과를 출력합니다.
    print(f"{col}: 무한대 값 {inf_count}개, NaN 값 {nan_count}개, 극값 {len(extreme_values)}개")

    # 무한대 값을 NaN으로 변환
# np.isinf() 등으로 탐지된 무한대(inf, -inf)를 판다스가 처리하기 쉬운 NaN으로 바꿉니다.
df[col] = df[col].replace([np.inf, -np.inf], np.nan)

# NaN 값을 중앙값으로 대체
# 해당 열에 결측치가 하나라도 존재하는지 확인합니다.
if df[col].isnull().sum() > 0:
    # 데이터의 분포를 왜곡하지 않는 중앙값을 계산합니다.
    median_val = df[col].median()

    # fillna()를 사용하여 결측치를 계산된 중앙값으로 채웁니다.
    df[col] = df[col].fillna(median_val)

    # 정제 결과를 콘솔에 출력하여 가독성을 높입니다.
    print(f" -> {col}의 결측치를 중앙값 {median_val:.2f}로 대체")

    # 극값 처리 (99.9% 분위수로 제한)
# 데이터의 상위 0.1% 지점(0.999)과 하위 0.1% 지점(0.001)을 경계로 설정합니다.
upper_limit = df[col].quantile(0.999)
lower_limit = df[col].quantile(0.001)

# 설정한 경계 범위를 벗어나는 데이터를 마스킹합니다.
extreme_mask = (df[col] > upper_limit) | (df[col] < lower_limit)

# 극값을 벗어나는 값을 찾아서 극값(경계값)으로 제한합니다.
if extreme_mask.sum() > 0:
    # 상한선 초과값은 상한선으로, 하한선 미달값은 하한선으로 대체합니다.
    df.loc[df[col] > upper_limit, col] = upper_limit
    df.loc[df[col] < lower_limit, col] = lower_limit

    # 처리된 결과(개수 및 조정된 범위)를 출력하여 전처리 이력을 남깁니다.
    print(f" -> {col}의 극값 {extreme_mask.sum()}개를 {lower_limit:.2f}~{upper_limit:.2f} 범위로 제한")

    # 정규화 전 통계
print("\n정규화 전 통계:")
# numeric_columns에 포함된 주요 수치형 데이터들의 분포(평균, 최소, 최대 등)를 요약합니다.
print(df[numeric_columns].describe())

# 데이터 유효성 최종 확인
print("\n데이터 유효성 최종 확인:")
for col in numeric_columns:
```

```
    # np.isinf().any() : 해당 열에 무한대 값(inf)이 하나라도 있으면 True를 반환합니다.
    has_inf = np.isinf(df[col]).any()

    # isnull().any() : 해당 열에 결측치(NaN)가 하나라도 있으면 True를 반환합니다.
    has_nan = df[col].isnull().any()

    # 각 열별로 정규화 연산에 방해가 되는 요소가 없는지 최종 결과를 출력합니다.
    print(f"{col}: 무한대값 {has_inf}, NaN값 {has_nan}")

    # StandardScaler를 사용한 표준화
scaler = StandardScaler()
df_scaled = df_encoded.copy()

try:
    # fit_transform을 사용하여 선택한 수치형 열들을 한 번에 표준화합니다.
    df_scaled[numeric_columns] = scaler.fit_transform(df[numeric_columns])
    print("\n정규화 성공!")

except Exception as e:
    # 오류 발생 시 실패 원인과 함께 디버깅을 위한 추가 정보를 출력합니다.
    print(f"\n정규화 실패: {e}")

    # 추가 디버깅 정보
    for col in numeric_columns:
        # 각 열의 범위와 통계량을 확인하여 정규화를 방해하는 요소(Inf, NaN 등)를 찾습니다.
        print(f"{col} 범위: {df[col].min():.2f} ~ {df[col].max():.2f}")
        print(f"{col} 평균: {df[col].mean():.2f}, 표준편차: {df[col].std():.2f}")

print("\n정규화 후 통계:")
# 표준화가 완료된 데이터의 요약 통계량을 확인합니다. (평균은 0, 표준편차는 1에 수렴해야 합니다)
print(df_scaled[numeric_columns].describe())

print("\n4-2. 최종 데이터 품질 검증")

# 최종적으로 정제되고 스케일링 된 데이터프레임의 크기(행, 열)를 확인합니다.
print(f"최종 데이터 크기: {df_scaled.shape}")

# isnull().sum().sum() : 데이터프레임 전체에서 단 하나의 결측치라도 남아있는지 전수 조사합니다.
print(f"결측치 확인: {df_scaled.isnull().sum().sum()}개")

# duplicated().sum() : 데이터 정제 과정 이후 새롭게 발생했거나 남아있는 중복 행이 있는지 확인합니다.
print(f"중복 행 확인: {df_scaled.duplicated().sum()}개")

print("\n=== 데이터 전처리 완료 리포트 ===")
# 원본 데이터와 최종 데이터의 크기(행, 열)를 비교하여 출력합니다.
print(f"원본 데이터: {df_original.shape[0]}행 {df_original.shape[1]}열")
print(f"최종 데이터: {df_scaled.shape[0]}행 {df_scaled.shape[1]}열")

print(f"처리된 문제들:")
# 전처리 과정에서 해결된 주요 이슈들을 정량적으로 요약합니다.
print(f"  - 나이 이상치 {len(df_original[(df_original['age'] < 0) | (df_original['age'] > 100)])}개 수
정")
print(f"  - 성별 데이터 표준화 완료")
print(f"  - 도시 결측치 {df_original['city'].isnull().sum()}개 처리")
print(f"  - 구매 금액 이상치 처리 완료")
print(f"  - 파생 변수 2개 생성 (평균 구매 금액, 고객 세그먼트)")
print(f"  - 범주형 데이터 인코딩 완료")
print(f"  - 수치형 데이터 표준화 완료")
```

```
print(f"\n데이터 전처리가 성공적으로 완료되었습니다!")
print(f"이제 머신러닝 모델 학습이나 데이터 분석에 사용할 수 있습니다.")
```