자전거 수요 예측 분석 보고서

Dataset : Bike Sharing Demand (Kaggle)

AI_02_전혜정

INTRO

자전거 공유 시스템은 도시에서 효율적이고 친환경적인 교통수단을 제공하며, 대여 수요 예측은 차량 관리와 사용자 경험 향상에 필수적이다. 이 프로젝트는 **과거 대여 데이터를 기반**으로 **다양한 환경적 및 시간적 요인을 고려**하여 **자전거 대여 수를 예측하는 모델을 구축**하게 되었다.

목표

목표는 **머신러닝 기법을 사용**하여 **시간대별 총 자전거 대여 수(count)를 예측**하는 것이다.

데이터셋에는 날씨 조건, 계절성, 사용자 유형 등 대여 수요에 영향을 미치는 다양한 요인이 포함되어 있다. 이 데이터를 활용하여 보이지 않는 테스트 데이터에 잘 일반화할 수 있는 견고한 모델을 구축하는 것을 목표로 한다.

데이터 설명

컬럼명	데이터 타입	설명
datetime	datetime	자전거 대여 기록의 날짜 및 시간. 예시: 2011-01-01 00:00:00
season	int	계절 (1: 봄, 2: 여름, 3: 가을, 4: 겨울)
holiday	int	공휴일 여부 (0: 평일, 1: 공휴일)
workingday	int	근무일 여부 (0: 주말/공휴일, 1: 근무일)
weather	int	날씨 상황 (1: 맑음, 2: 구름낌/안개, 3: 약간의 비/눈, 4: 폭우/폭설)
temp	float	실측 온도 (섭씨)
atemp	float	체감 온도 (섭씨)
humidity	int	습도 (%)
windspeed	float	풍속 (m/s)
casual	int	등록되지 않은 사용자의 대여 수
registered	int	등록된 사용자의 대여 수
count	int	총 대여 수 (종속 변수)

EDA

(특성들 간의 상관관계)

< hour ↔ count (0.4) >

: 시간(hour)이 증가할수록 자전거 대여량(count)이 증가하는 경향이 있음.

: 보통 출퇴근 시간(아침, 저녁)에 대여량이 증가할 가능성이 높음.

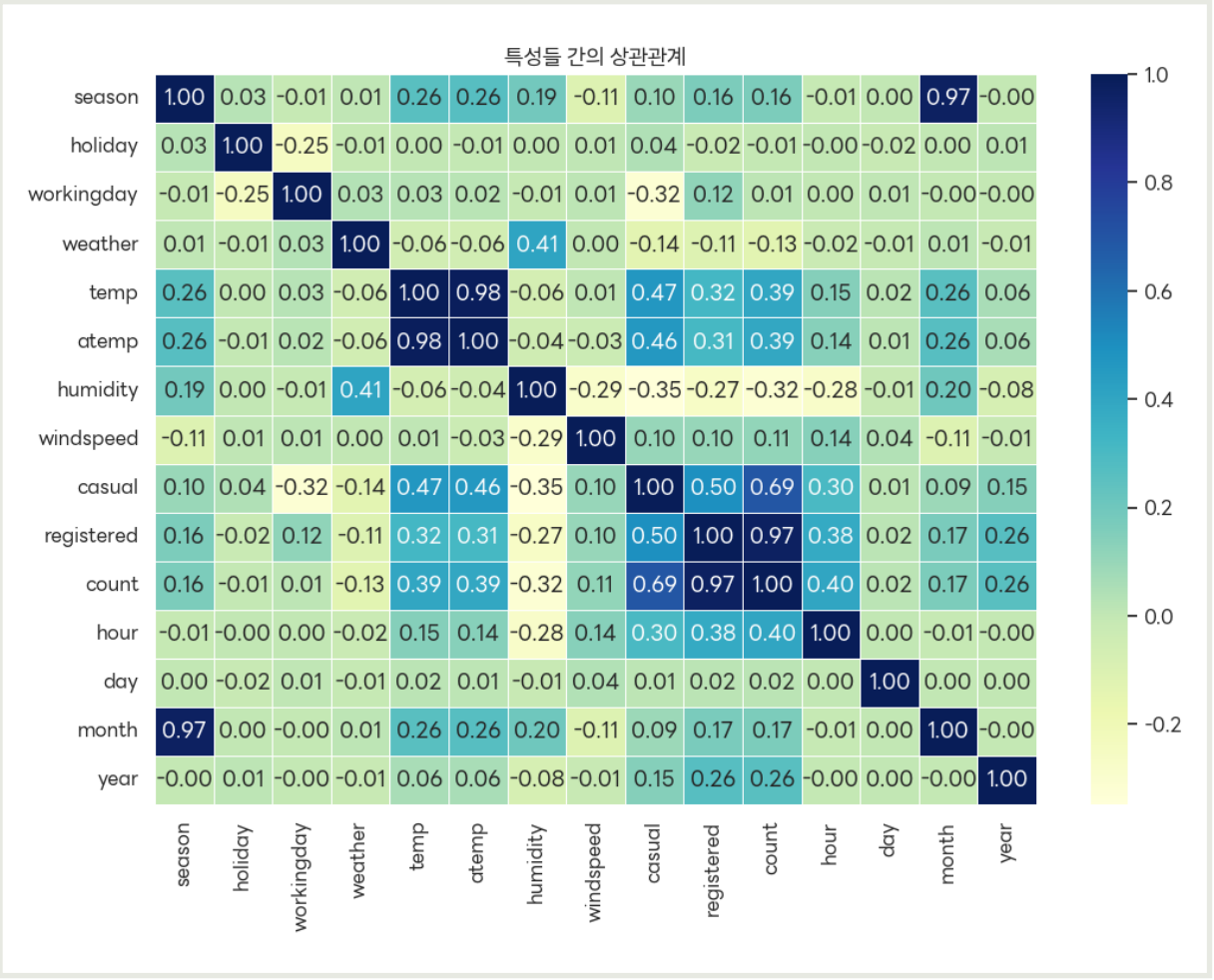
< temp ↔ count (0.39) >

: 기온이 올라갈수록 자전거 대여량이 증가하는 경향이 있음.

: 날씨가 따뜻할수록 사람들이 더 많이 자전거를 이용한다고 볼 수 있음.

< atemp ↔ temp (0.98) >

: 체감 온도와 온도는 거의 똑같다는 사실을 알 수 있음.



EDA

(count 컬럼과의 상관관계)

< count ↔ registered (0.97) >

: 회원 수 증가가 자전거 대여량 증가에 가장 큰 영향을 미침.

< count ↔ casual (0.69)) >

: 비회원 이용도 중요한 요인이지만, 회원 대여량이 더 큰 영향을 미침.

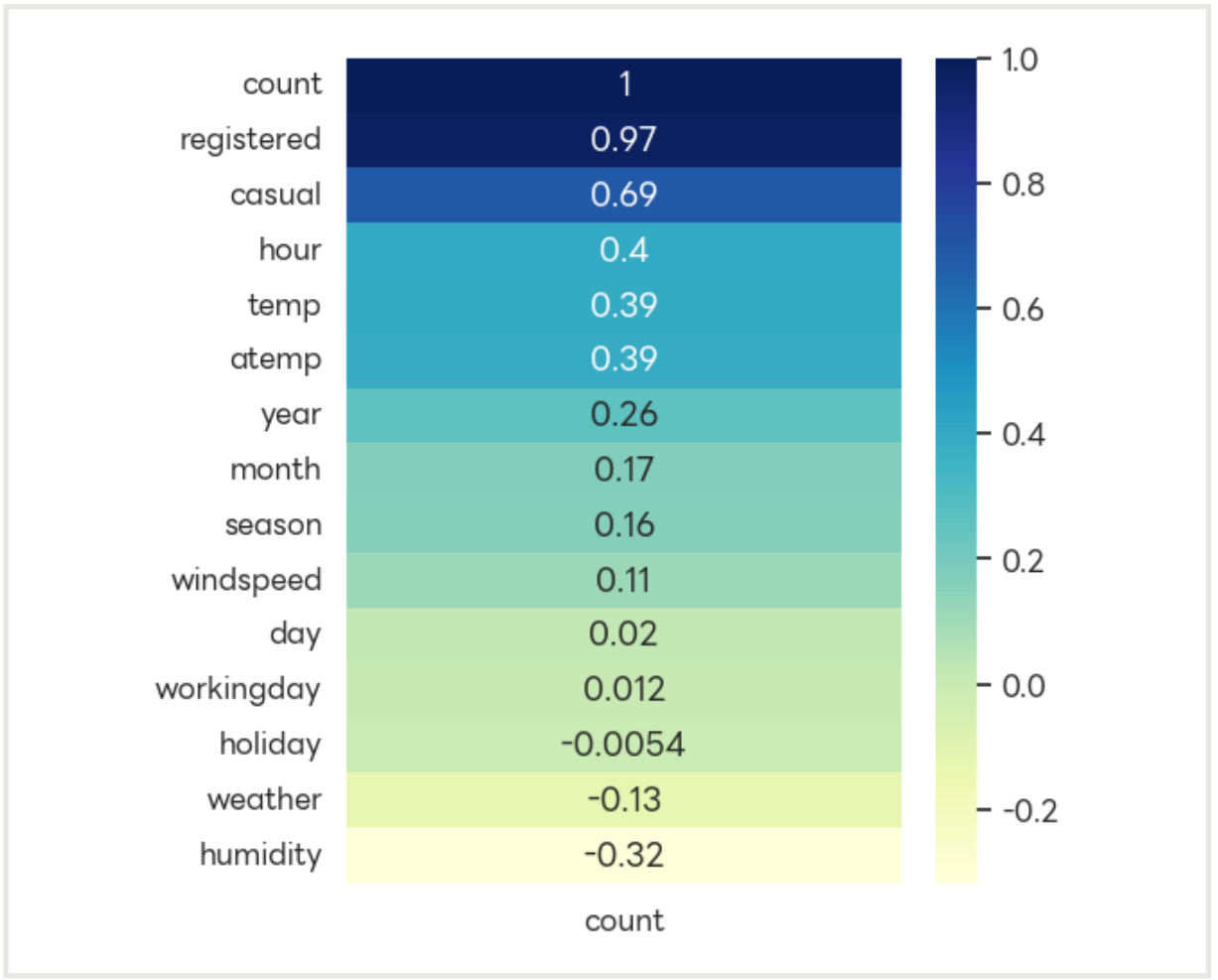
< count ↔ hour (0.4) >

: 시간대가 자전거 이용에 영향을 줌. 출퇴근 시간의 패턴을 분석하면 대여량을 예측하는 데 도움이 될 수 있음..

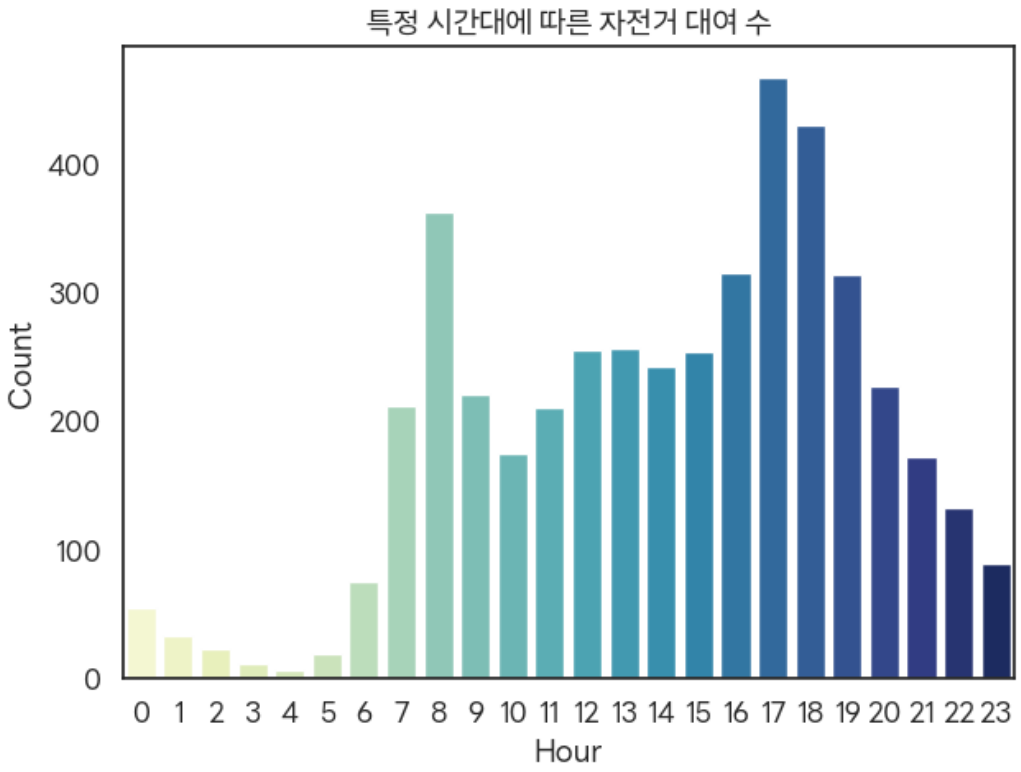
< humidity ↔ count (-0.32) >

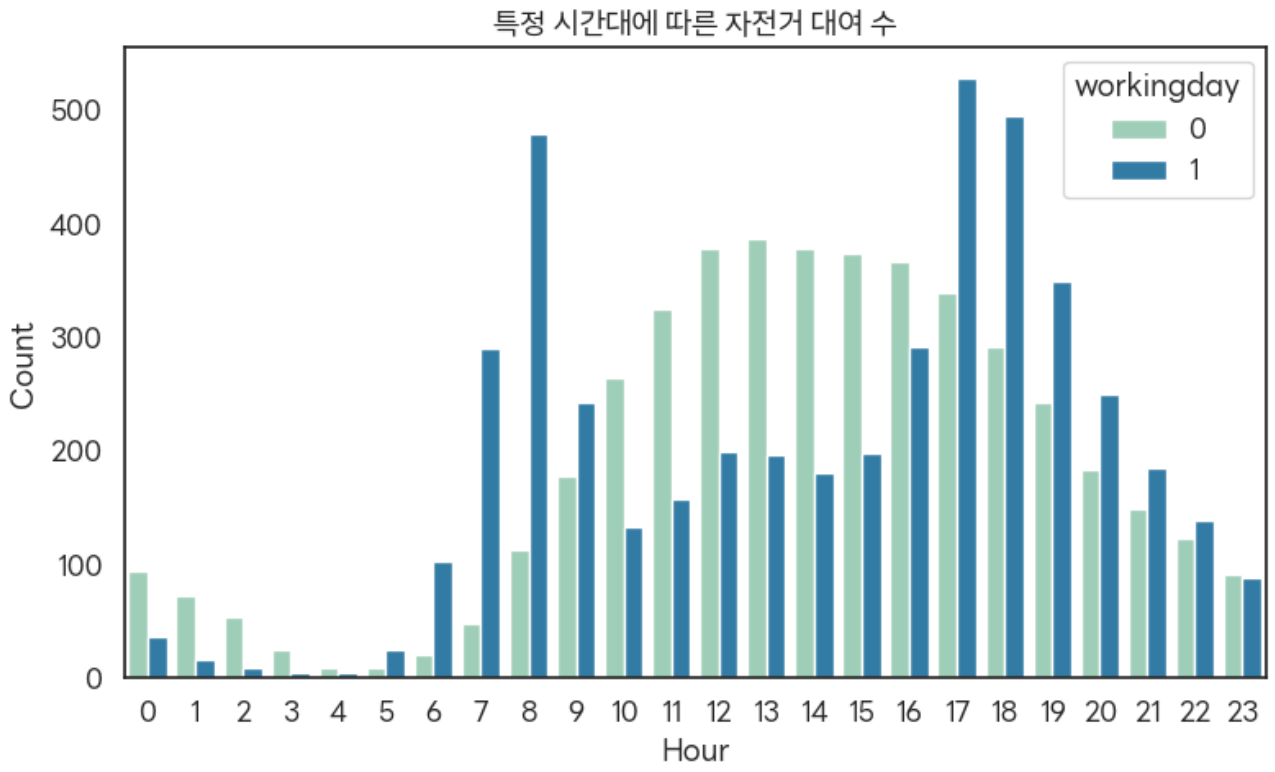
: 습도가 높아질수록 자전거 대여량이 감소하는 경향이 있음.

: 습도가 높으면 불쾌지수가 상승하고, 사람들이 야외 활동을 줄이는 것으로 추측



EDA





근무일

: **출퇴근 시간(07시~09시, 17시~19시) 수요 급증**

: 출퇴근 시간대의 수요 증가는 **근무일과 강한 연관성**을 가짐

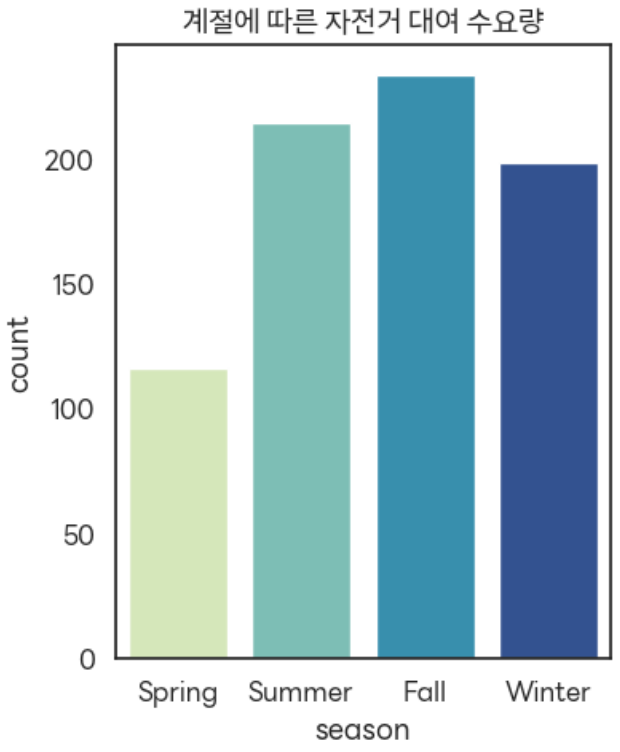
휴일

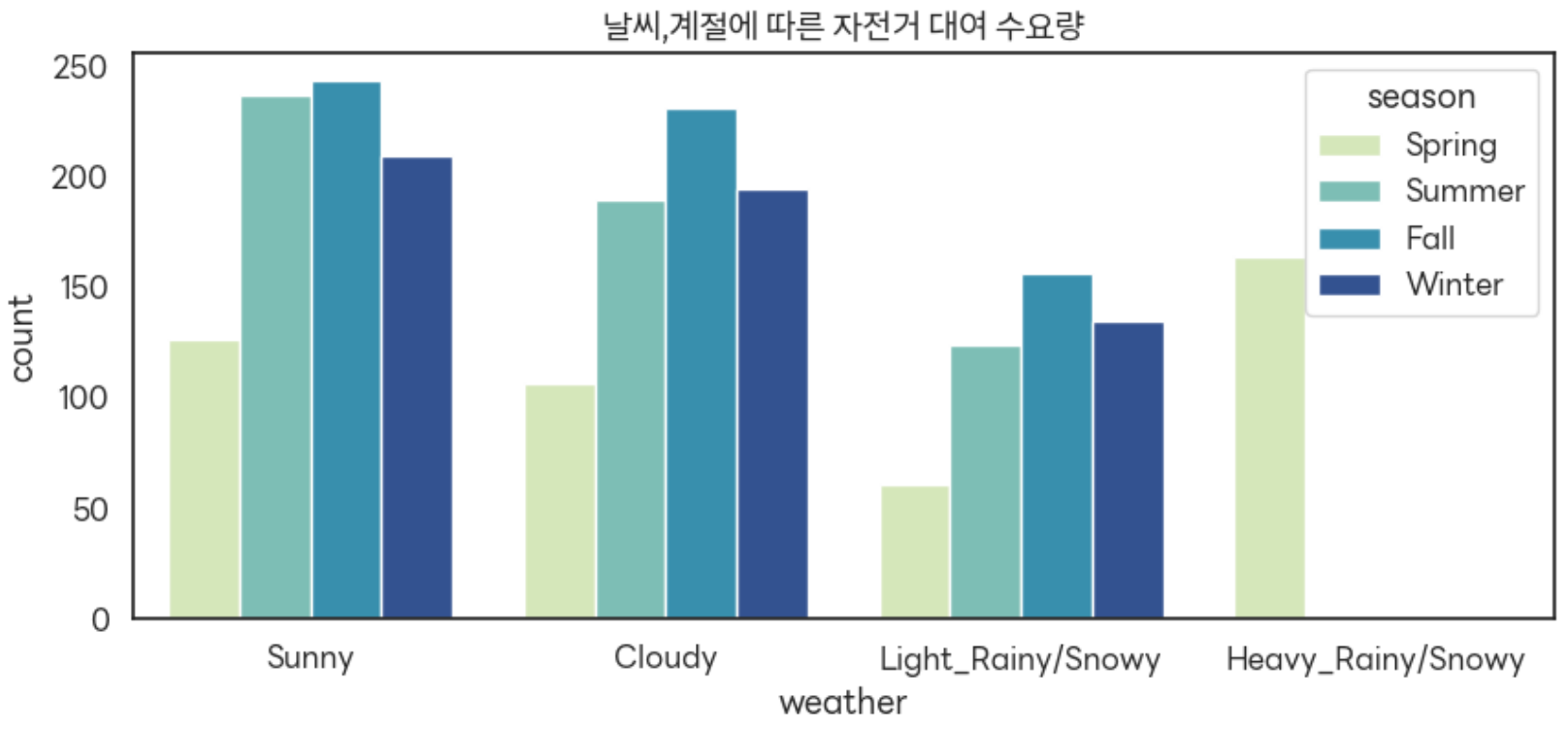
: **정오 오후(11시~18시) 수요 높음**

: 근무일과 달리 출퇴근 시간의 피크타임이 나타나지 않음

: 이는 여가 활동을 위해 자전거 대여를 이용하는 것으로 해석 가능

EDA





- ✓ **날씨가 맑을수록 자전거 대여 수요가 증가**하는 경향이 있음
- ✓ 맑은 날(Sunny)만큼은 아니지만 **흐린 날(Cloudy)에도 대여량이 많음**
- ✓ **여름(Summer)**과 **가을(Fall)**에서 대여량이 가장 높음
- ✓ **겨울(Winter)**은 다른 계절보다 **전체적으로 대여량이 적음**
- ✓ **봄(Spring)은 대체로 낮은 편**, 특히 **비나 눈이 오는 날(Spring에서 가장 낮은 막대)** 은 확연히 적음

전처리 - 결측치

```
train.isnull().sum()
```

```
season        0
holiday       0
workingday    0
weather       0
temp          0
atemp         0
humidity      0
windspeed     0
casual        0
registered    0
count         0
hour          0
day           0
month         0
year          0
dtype: int64
```

```
test.isnull().sum()
```

```
datetime      0
season        0
holiday       0
workingday    0
weather       0
temp          0
atemp         0
humidity      0
windspeed     0
hour          0
day           0
month         0
year          0
dtype: int64
```

train 데이터, test 데이터
둘 다 결측치 없다.(NULL = 0)

전처리 - 중복값

```
train.duplicated().sum()
```

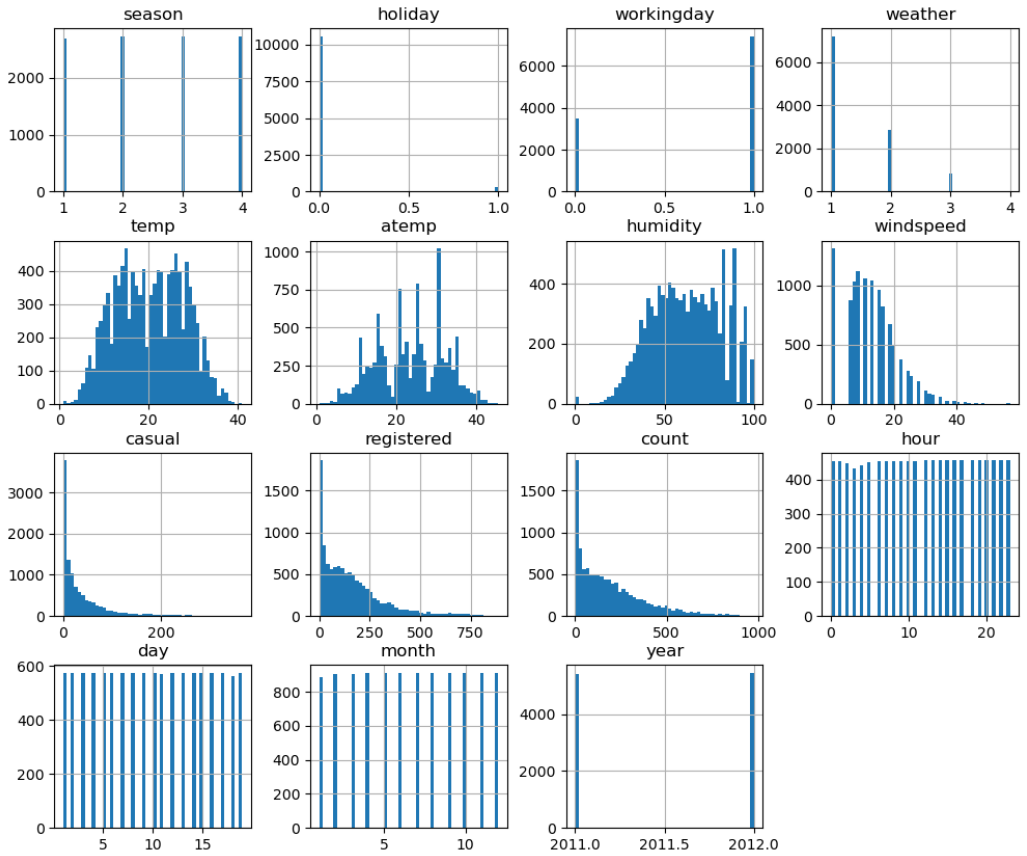
```
0
```

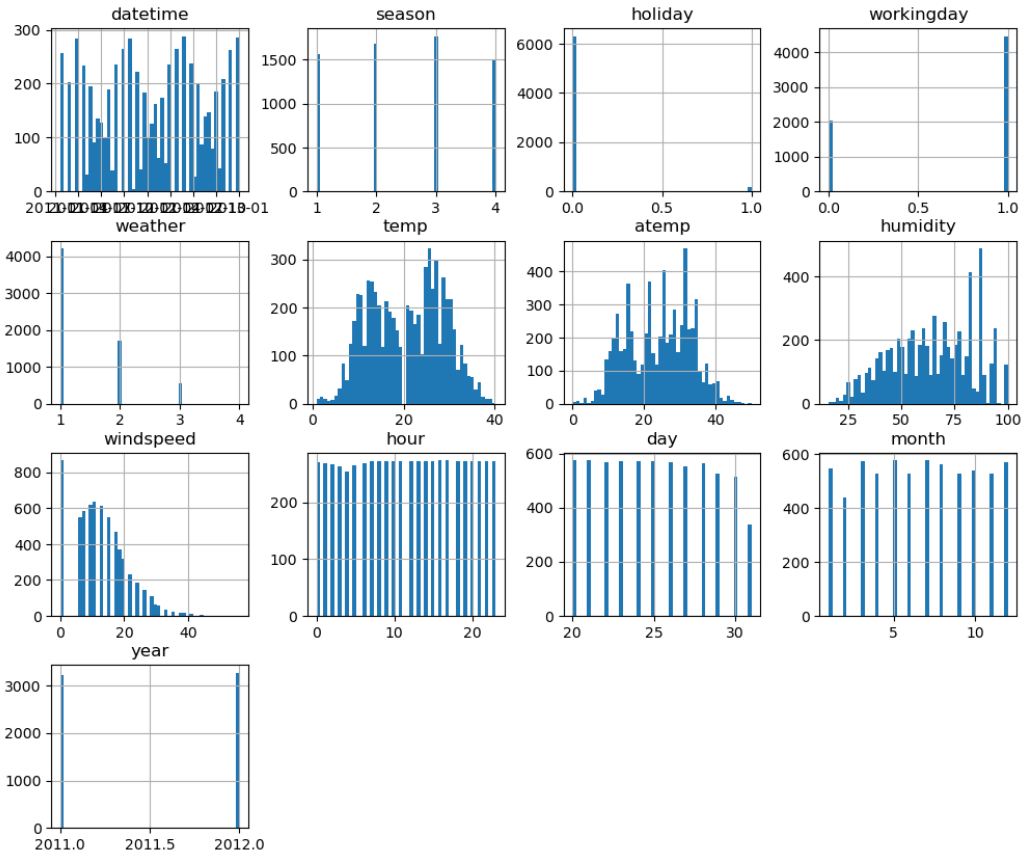
```
test.duplicated().sum()
```

```
0
```

train 데이터, test 데이터
둘 다 중복값 또한 없다.

전처리 - 이상치





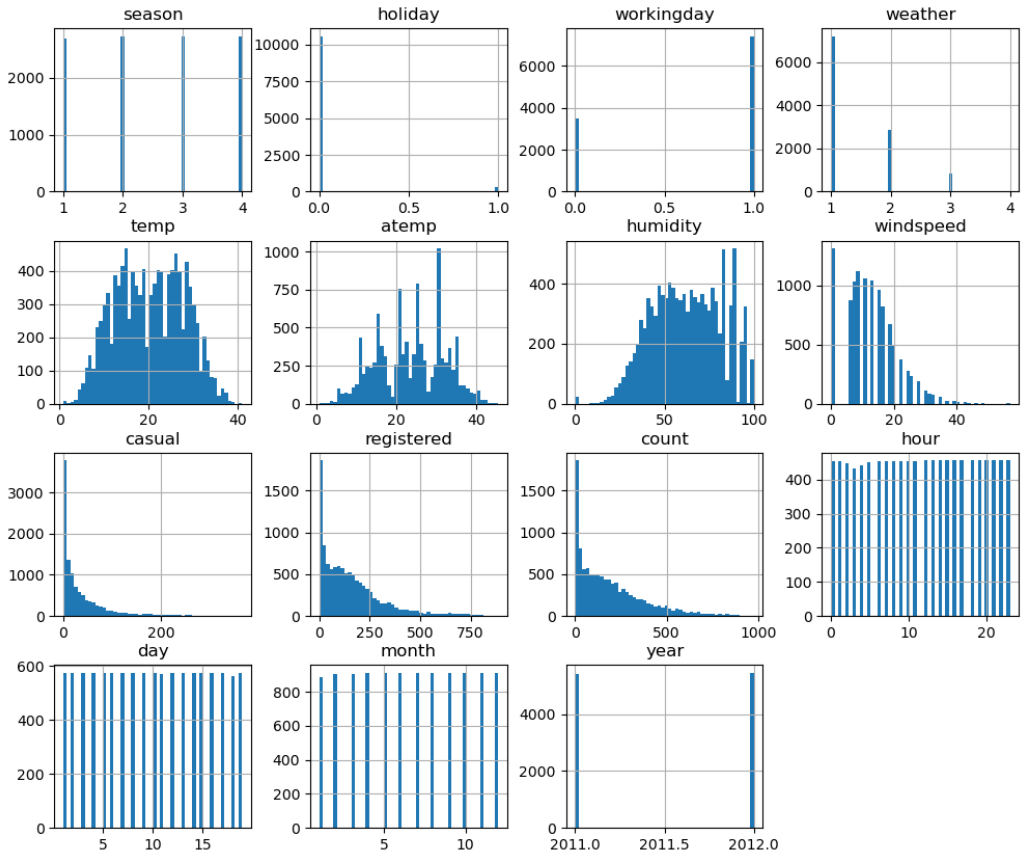
(train 데이터)

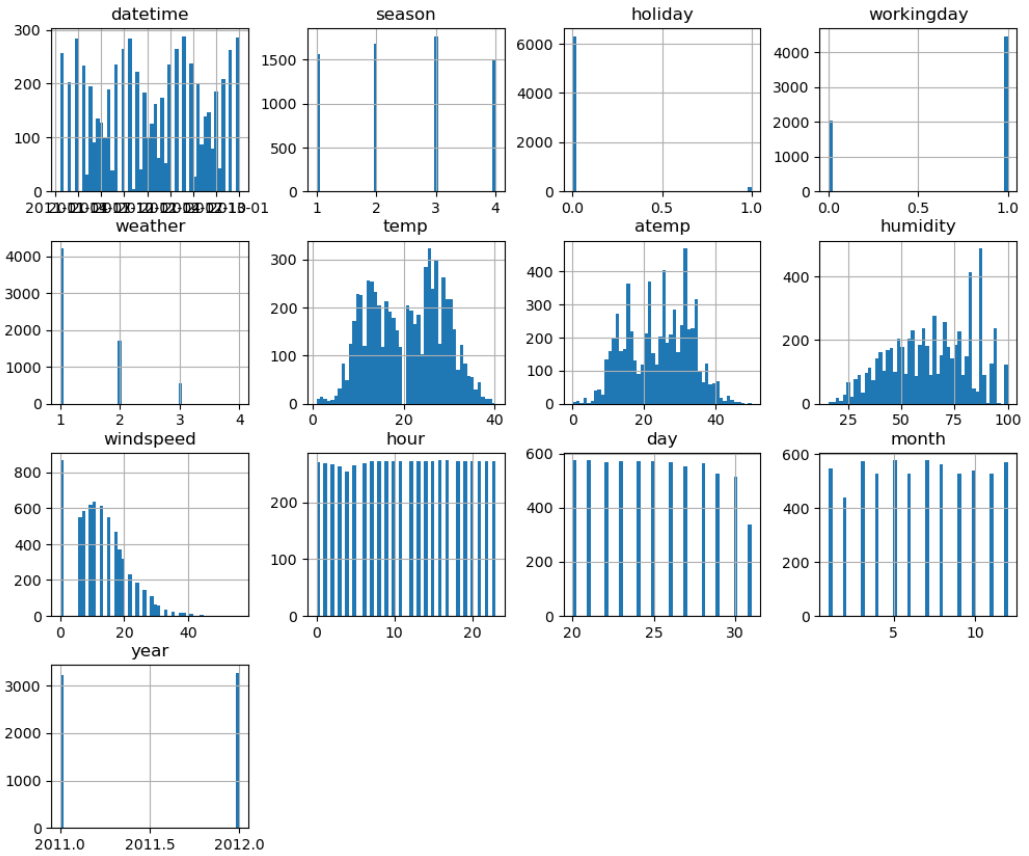
windspeed, casual, registered, count의 경우 이상치가 존재할 수 있는 그래프 개형이다.

(test 데이터)

Windspeed의 경우 이상치가 존재할 수 있는 그래프 개형이다.

전처리 - 이상치





- ✓ **습도**의 경우, **0 또는 100에 근접**한다면, 사막, 고산지대이거나 열대우림, 바다 한가운데인 경우이기에 **이상치**일 가능성이 높다.
- ✓ **풍속**의 경우, **20m/s이상**만 되어도 걸어다닐 수 없을 정도로 강한 바람이기에 20m/s 이상은 **이상치**일 가능성이 높다.

전처리 - 이상치

```python
# 이상치 개수 확인
for col in train.columns:
    q1 = train[col].quantile(0.25)
    q3 = train[col].quantile(0.75)
    iqr = q3 - q1
    lower_bound = q1 - 1.5 * iqr
    upper_bound = q3 + 1.5 * iqr
    outliers = train[(train[col] < lower_bound) | (train[col] > upper_bound)]
    print(f"{col} : {len(outliers)}")
```

```
season : 0
holiday : 311
workingday : 0
weather : 1
temp : 0
atemp : 0
humidity : 22
windspeed : 227
casual : 749
registered : 423
count : 300
hour : 0
day : 0
month : 0
year : 0
```

→ train 데이터 이상치 개수

```
datetime : 0
season : 0
holiday : 189
workingday : 0
weather : 2
temp : 0
atemp : 0
humidity : 0
windspeed : 115
hour : 0
day : 0
month : 0
year : 0
```

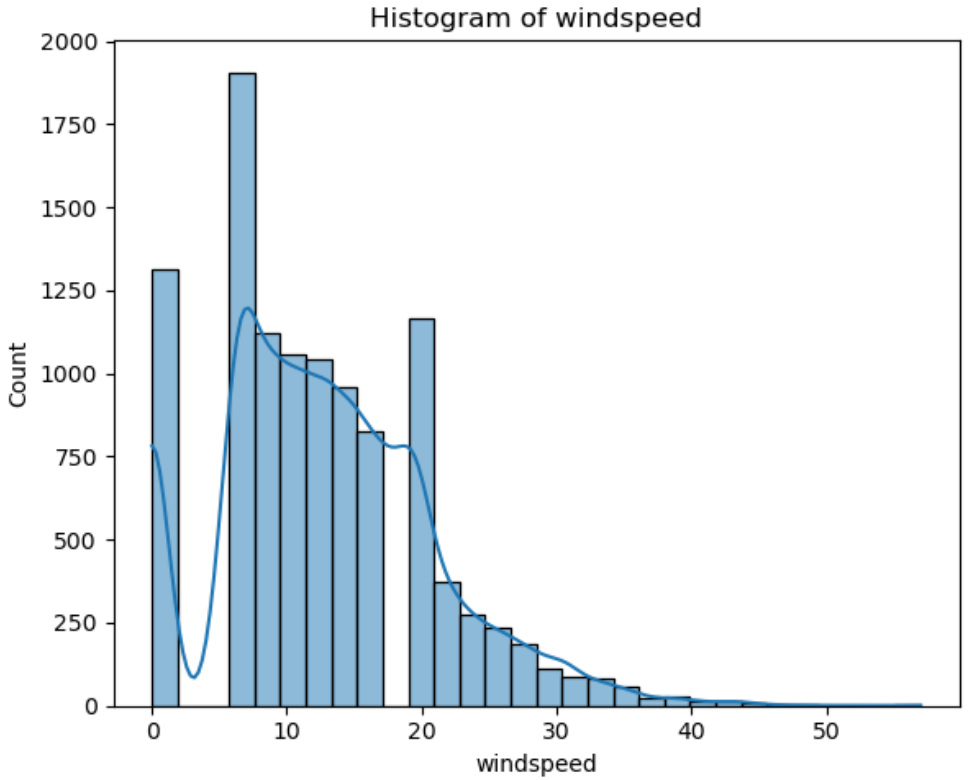
→ test 데이터 이상치 개수

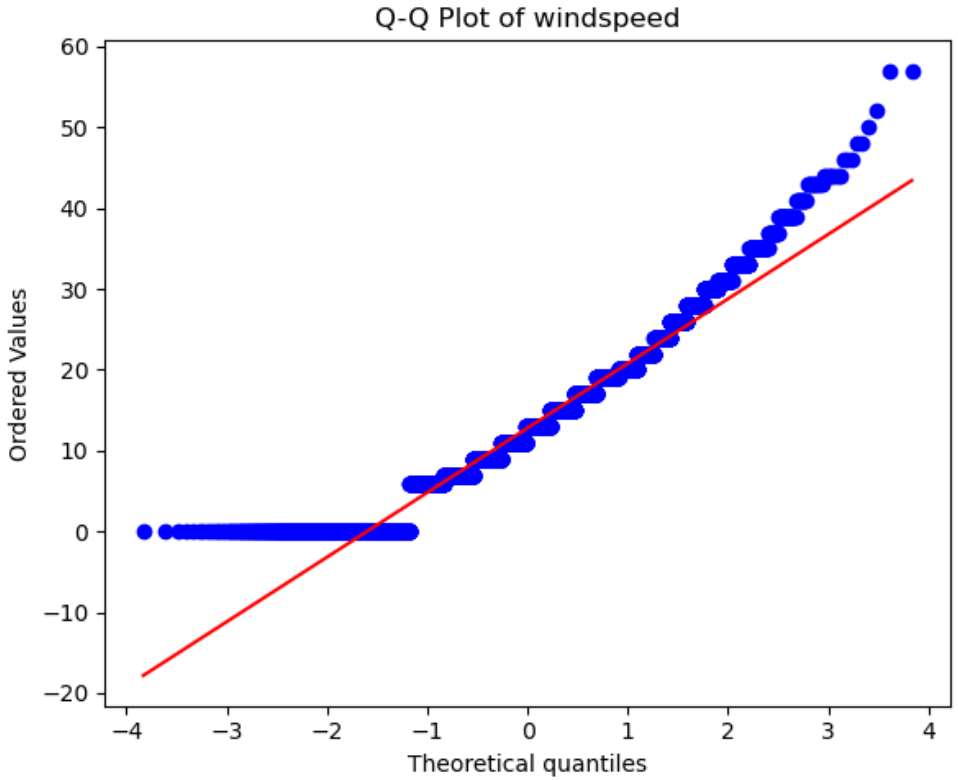
전처리 - 이상치

```python
def plot_hist_qq(data, column):
    fig, axes = plt.subplots(1, 2, figsize=(12, 5))

    # 히스토그램
    sns.histplot(data[column], bins=30, kde=True, ax=axes[0])
    axes[0].set_title(f"Histogram of {column}")

    # Q-Q 플롯
    stats.probplot(data[column], dist="norm", plot=axes[1])
    axes[1].set_title(f"Q-Q Plot of {column}")

    plt.tight_layout()
    plt.show()
```

 # 히스토그램과 Q-Q Plot 그래프를 동시에 그리기 위한 함수

전처리 - 이상치





(‘windspeed’ 컬럼 이상치 탐색)

✓ Q-Q Plot에서 양 끝이 정규분포에서 벗어난 형태 : 좌측 하단에서 점들이 직선에서 크게 벗어났다. → 풍속이 0인 데이터가 많음

: 우측 상단에서도 이상치 존재 가능성이 있다.

전처리 - 이상치

```
train["windspeed"].loc[train["windspeed"] > 30 ].value_counts()
```

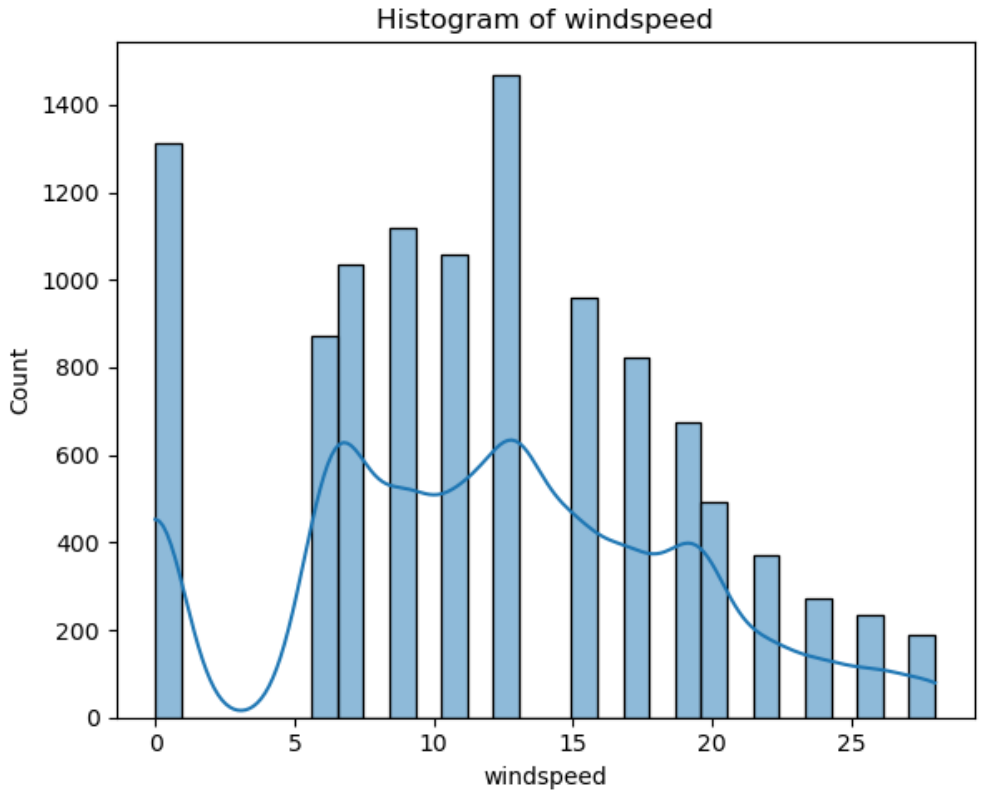
```
windspeed
30.0026    111
31.0009     89
32.9975     80
35.0008     58
39.0007     27
36.9974     22
43.0006     12
40.9973     11
43.9989      8
46.0022      3
56.9969      2
47.9988      2
51.9987      1
50.0021      1
Name: count, dtype: int64
```

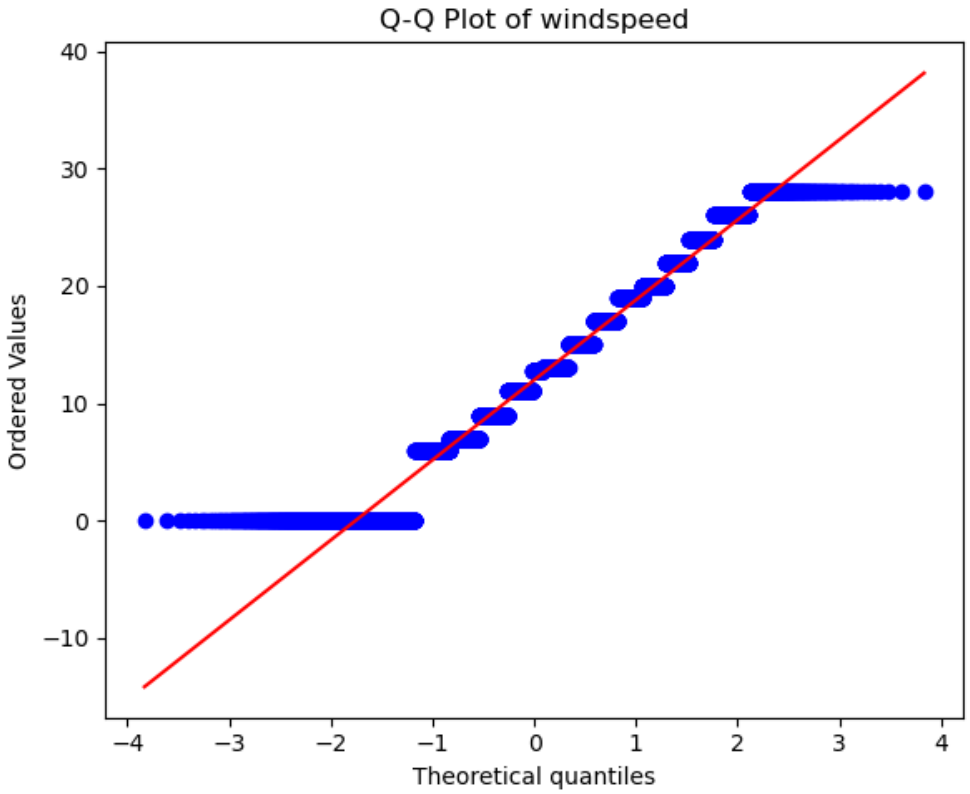
```
train["windspeed"] = np.where(train["windspeed"] >= 30, train["windspeed"].mean() , train["windspeed"])
```

(이상치 처리 방법)

: windspeed가 20m/s 이상인 것부터 처리를 하면 데이터 손실이 있을 것 같아 30m/s이상인 값들을 windspeed 컬럼들 값의 평균값으로 대체.

전처리 - 이상치



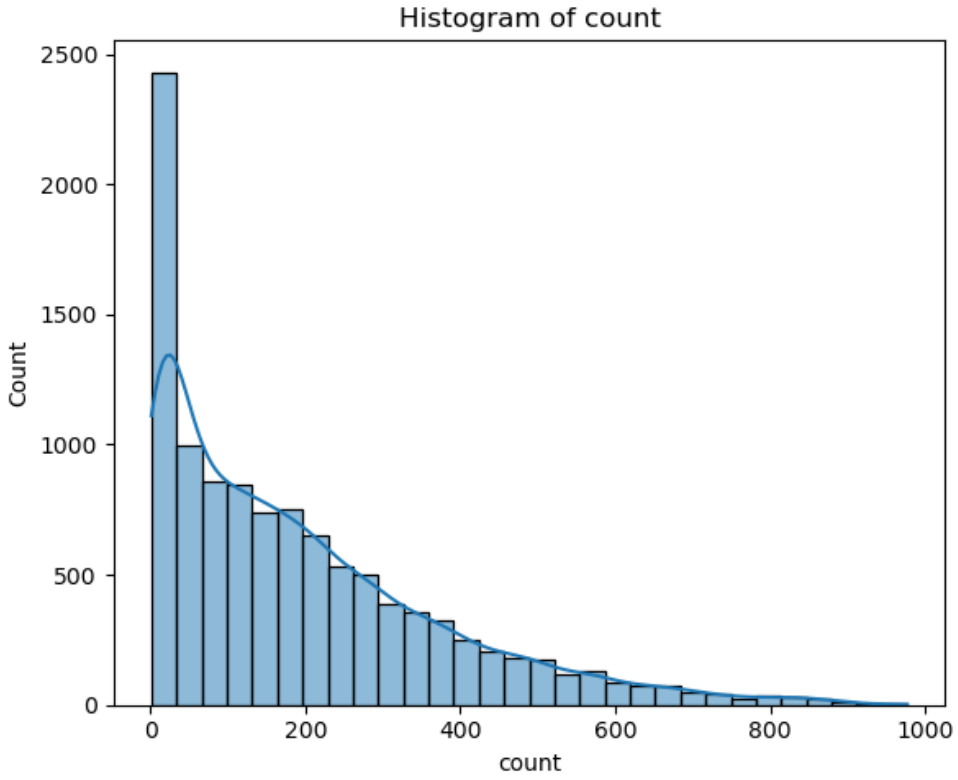


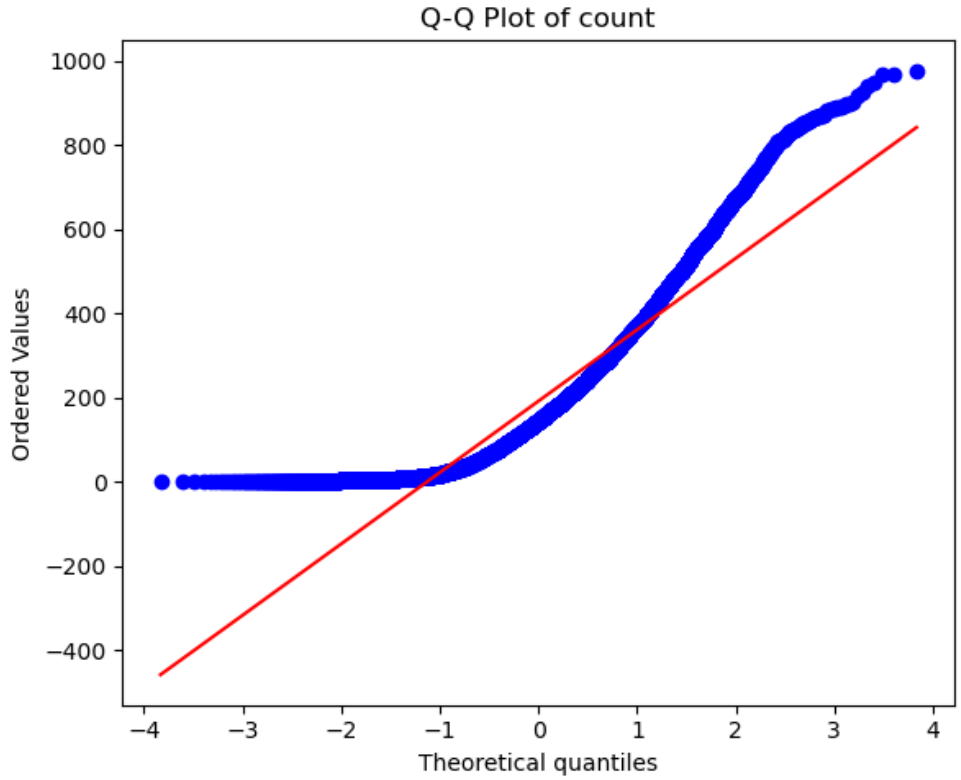
(이상치 처리 후 'windspeed' 컬럼)

✓ 이전 그래프보다 정규 분포에 가까워졌다.

✓ 이상치 처리를 통해 풍속 데이터의 왜곡이 줄어듦

전처리 - 이상치





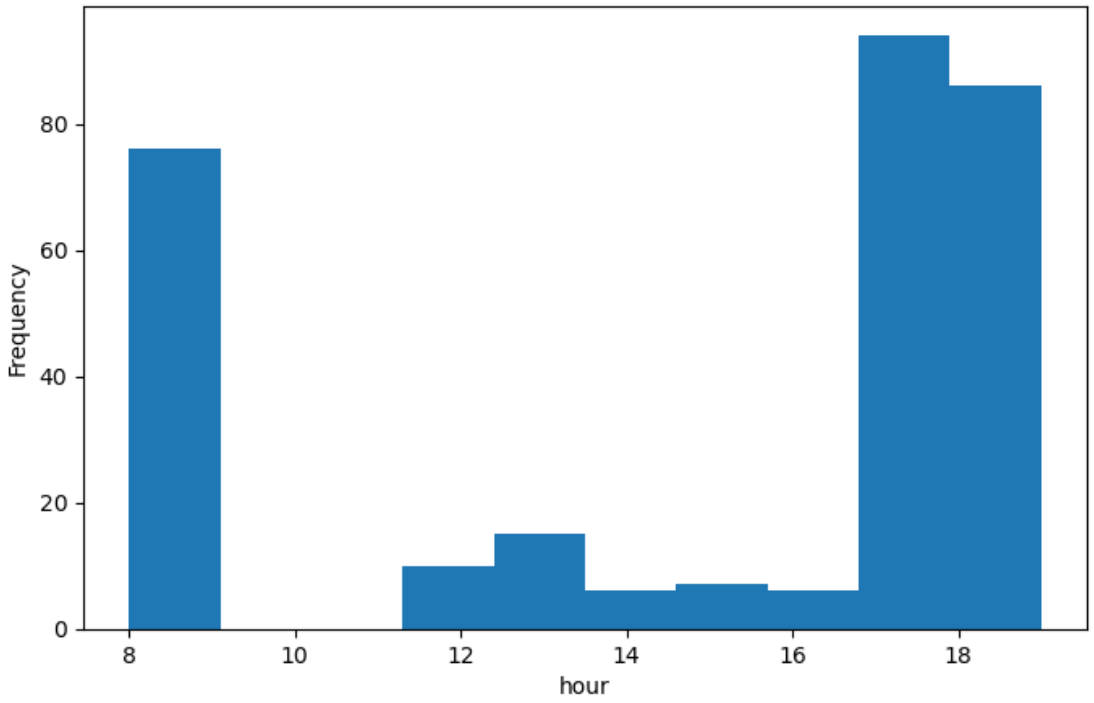
(‘count’ 컬럼 이상치 탐색)

✓ 히스토그램에서 자전거 대여량 데이터는 정규성을 따르지 않으며, 오른쪽으로 긴 꼬리를 가진 분포 (Right-Skewed Distribution) 분포를 보인다.

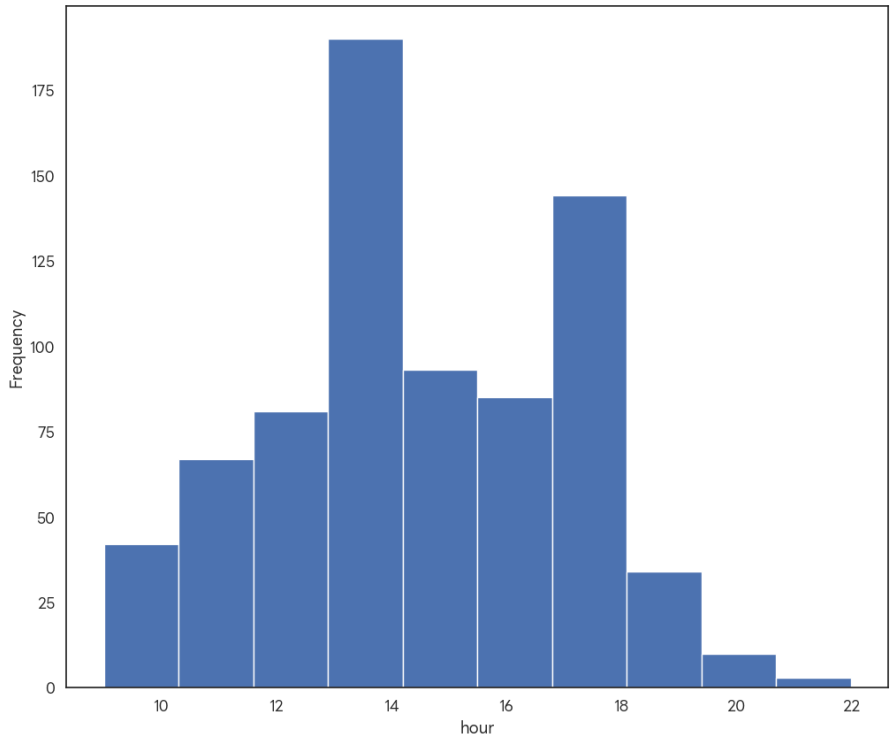
✓ QQ Plot 좌측 하단과 우측 상단에 몇 개의 극단적인 이상치가 존재.

전처리 - 이상치

(train 데이터)



(train 데이터)



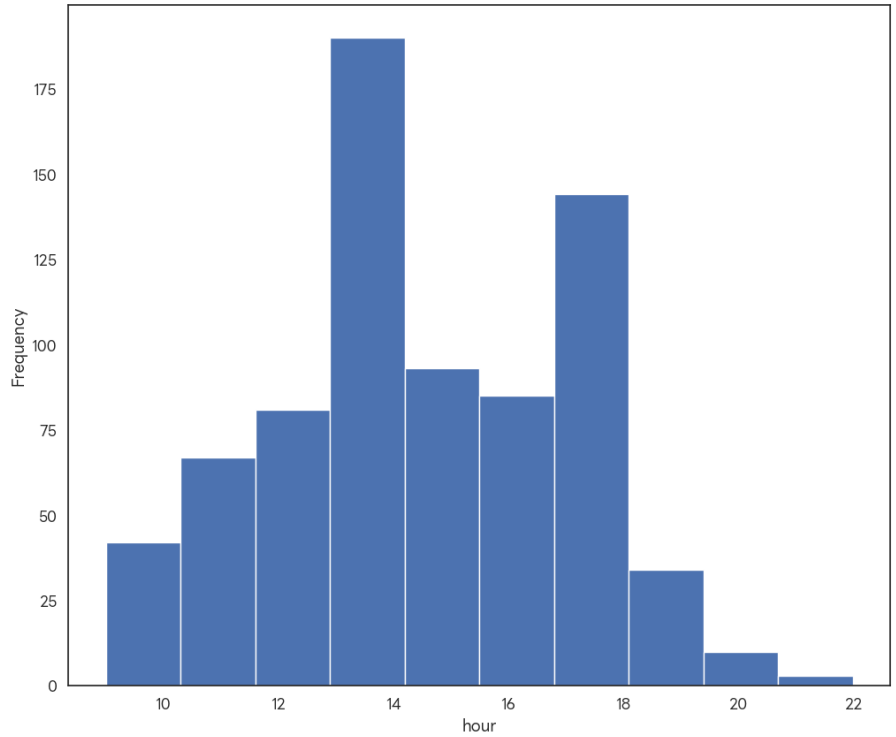
(train 데이터)

windspeed, casual, registered, count의 경우 이상치가 존재할 수 있는 그래프 개형이다.

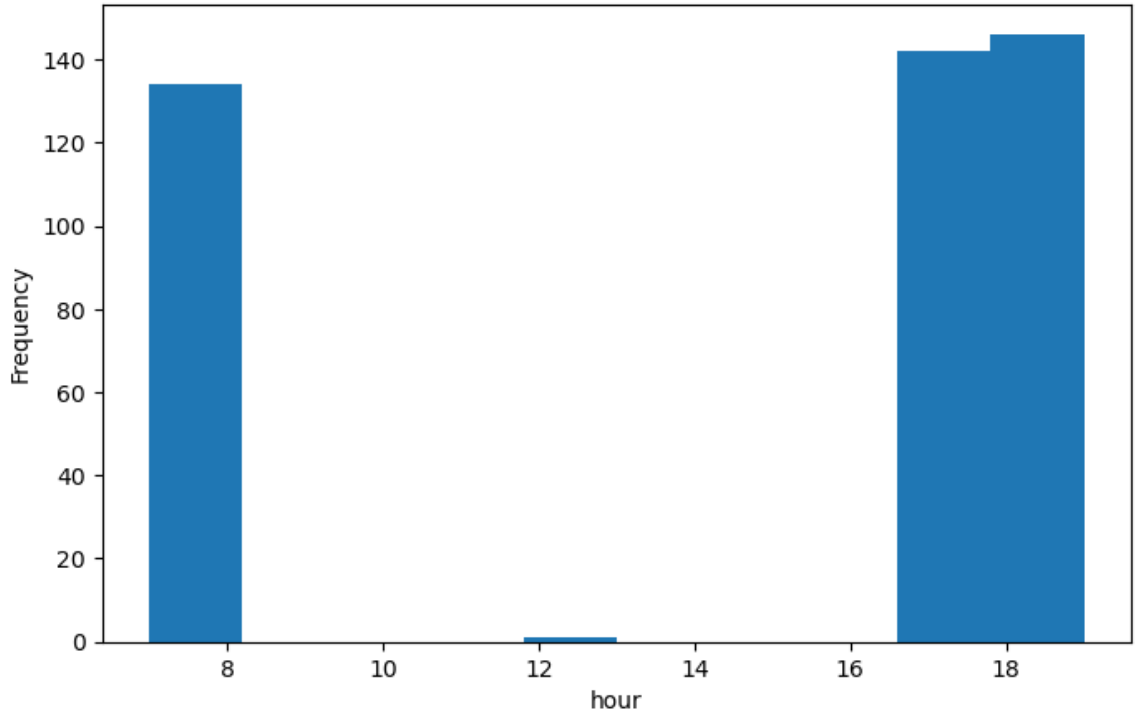
(test 데이터)

Windspeed의 경우 이상치가 존재할 수 있는 그래프 개형이다.

전처리 - 이상치



(등록되지 않은 사용자(casual)의 대여 수에 따른 이상치)



(등록된 사용자(registered)의 대여 수에 따른 이상치)

✓ casual 컬럼에 대한 이상치 경우, 점심시간대와 출퇴근 시간 이외의 시간대에서의 이상치가 의심된다.

✓ registered 컬럼에 대한 이상치 경우, 출퇴근 시간대라는 근거가 명확해 보인다.

전처리 - 데이터 변환

```
# 문자열 -> datetime
train.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 10886 entries, 0 to 10885
Data columns (total 12 columns):
 #   Column      Non-Null Count  Dtype
---  ------      --------------  -----
 0   datetime    10886 non-null  object
 1   season      10886 non-null  int64
 2   holiday     10886 non-null  int64
 3   workingday  10886 non-null  int64
 4   weather     10886 non-null  int64
 5   temp        10886 non-null  float64
 6   atemp       10886 non-null  float64
 7   humidity    10886 non-null  int64
 8   windspeed   10886 non-null  float64
 9   casual      10886 non-null  int64
 10  registered  10886 non-null  int64
 11  count       10886 non-null  int64
dtypes: float64(3), int64(8), object(1)
memory usage: 1020.7+ KB
```

```
train['datetime'] = pd.to_datetime(train['datetime'])
train.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 10886 entries, 0 to 10885
Data columns (total 12 columns):
 #   Column      Non-Null Count  Dtype
---  ------      --------------  -----
 0   datetime    10886 non-null  datetime64[ns]
 1   season      10886 non-null  int64
 2   holiday     10886 non-null  int64
 3   workingday  10886 non-null  int64
 4   weather     10886 non-null  int64
 5   temp        10886 non-null  float64
 6   atemp       10886 non-null  float64
 7   humidity    10886 non-null  int64
 8   windspeed   10886 non-null  float64
 9   casual      10886 non-null  int64
 10  registered  10886 non-null  int64
 11  count       10886 non-null  int64
dtypes: datetime64[ns](1), float64(3), int64(8)
memory usage: 1020.7 KB
```

```
# 연도, 월, 일, 시간 추출
train['hour'] = train['datetime'].dt.hour
train['day'] = train['datetime'].dt.day
train['month'] = train['datetime'].dt.month
train['year'] = train['datetime'].dt.year
```

```
train.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 10886 entries, 0 to 10885
Data columns (total 16 columns):
 #   Column      Non-Null Count  Dtype
---  ------      --------------  -----
 0   datetime    10886 non-null  datetime64[ns]
 1   season      10886 non-null  int64
 2   holiday     10886 non-null  int64
 3   workingday  10886 non-null  int64
 4   weather     10886 non-null  int64
 5   temp        10886 non-null  float64
 6   atemp       10886 non-null  float64
 7   humidity    10886 non-null  int64
 8   windspeed   10886 non-null  float64
 9   casual      10886 non-null  int64
 10  registered  10886 non-null  int64
 11  count       10886 non-null  int64
 12  hour        10886 non-null  int32
 13  day         10886 non-null  int32
 14  month       10886 non-null  int32
 15  year        10886 non-null  int32
dtypes: datetime64[ns](1), float64(3), int32(4), int64(8)
memory usage: 1.2 MB
```

#object(문자열)을 datetime 타입으로 변환

연도, 월, 일, 시간 추출해서 각각의 컬럼 생성

모델링

```
df = df[['season', 'holiday', 'workingday', 'weather', 'temp','humidity',
         'windspeed','casual', 'registered','count', 'hour','day', 'month','year']]
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 10886 entries, 0 to 10885
Data columns (total 14 columns):
 #   Column      Non-Null Count  Dtype
---  ------      --------------  -----
 0   season      10886 non-null  object
 1   holiday     10886 non-null  int64
 2   workingday  10886 non-null  int64
 3   weather     10886 non-null  object
 4   temp        10886 non-null  float64
 5   humidity    10886 non-null  int64
 6   windspeed   10886 non-null  float64
 7   casual      10886 non-null  int64
 8   registered  10886 non-null  int64
 9   count       10886 non-null  int64
 10  hour        10886 non-null  int32
 11  day         10886 non-null  int32
 12  month       10886 non-null  int32
 13  year        10886 non-null  int32
dtypes: float64(2), int32(4), int64(6), object(2)
memory usage: 1020.7+ KB
```

머신러닝 알고리즘은 숫자 데이터만 처리할 수 있기에 원-핫 인코딩(One-Hot Encoding)을 해야 한다. 또한 원-핫 인코딩은 범주형 데이터를 효과적으로 변환하는 방법이기도 하다.

이 과정은 머신러닝 모델의 정확도를 높이고 학습 효율성을 향상시키는 중요한 전처리 과정이다.

예측 모델링에 사용할 컬럼들을 설정하여 새로운 데이터프레임 생성

모델링

```
X = df.drop(columns=['count'])
y = df['count']
X.shape, y.shape
```

```
((10886, 47), (10886,))
```

모델을 학습하기 위해

독립변수(X)와 종속 변수(y)를 분리함.

X : 10886개의 샘플 개수 / 47개의 특성(2D

y : 10886개의 샘플 개수(1D)

```
from sklearn.preprocessing import StandardScaler
```

```
ss = StandardScaler()
```

```
X_scaled_ss = ss.fit_transform(X)
```

```
X_scaled_ss
```

```
array([[-0.17149048, -1.46067232, -1.33366069, ..., -0.30220576,
        -0.30220576, -0.30238674],
       [-0.17149048, -1.46067232, -1.43890721, ..., -0.30220576,
        -0.30220576, -0.30238674],
       [-0.17149048, -1.46067232, -1.43890721, ..., -0.30220576,
        -0.30220576, -0.30238674],
       ...,
       [-0.17149048,  0.68461625, -0.80742813, ..., -0.30220576,
        -0.30220576,  3.30702336],
       [-0.17149048,  0.68461625, -0.80742813, ..., -0.30220576,
        -0.30220576,  3.30702336],
       [-0.17149048,  0.68461625, -0.91267464, ..., -0.30220576,
        -0.30220576,  3.30702336]])
```

✓ 표준화는 sklearn.preprocessing의 StandardScaler를 사용하여 수행함. → 표준화 시, 동일한 범위를 가지게 되어 모델이 **특정 변수에 의존하는 현상을 방지** 가능.

✓ **MinMaxScaler 대신 StandardScaler를 선택한 이유**

→ 사용할 데이터는 이상치가 존재하는 변수도 포함되어 있기에 **이상치에 덜 민감한** StandardScaler을 선택하였다.

→ StandardScaler : 평균=0, 표준편차=1 로 변환

평가 지표

```python
def evaluate_regression_model(test, prediction):
    print("MAE : ", mean_absolute_error(test, prediction))
    print("MSE : ", mean_squared_error(test, prediction))
    print("RMSE : ", root_mean_squared_error(test, prediction))
```

→ **MAE, MSE, RMSE**

```python
def rmsle(y_true, y_pred, convertExp=True):
    # 지수변환
    if convertExp:
        y_true = np.exp(y_true)
        y_pred = np.exp(y_pred)

    # 로그변환 후 결측값을 0으로 변환
    log_true = np.nan_to_num(np.log(y_true+1))
    log_pred = np.nan_to_num(np.log(y_pred+1))

    # RMSLE 계산
    output = np.sqrt(np.mean((log_true - log_pred)**2))
    return output
```

→ **RMSLE**

모델링&평가 - Linear Regression

```
lr_model_ss.fit(X_train, log_y_train)
```

LinearRegression

LinearRegression()

```
lr_model_ss.coef_, lr_model_ss.intercept_
```

```
(array([-4.41616247e-03, -3.74129239e-02,  2.13173366e-01, -5.66787967e-02,
        -2.80797430e-02,  2.38382932e-02,  2.43964683e-01,  8.97827852e+11,
         2.45910646e+12,  1.30418374e+12, -8.29833432e-04, -1.39918691e-01,
         2.30624192e-02, -1.38395819e-01, -2.32991890e-01, -3.43554574e-01,
        -4.04930220e-01, -1.95108444e-01,  5.76913930e-02,  2.49447018e-01,
         3.81723297e-01,  3.12378995e-01,  2.47087958e-01,  2.70364605e-01,
         3.06303395e-01,  3.05292120e-01,  2.90211415e-01,  2.97921783e-01,
         3.53161231e-01,  4.32005264e-01,  4.18344702e-01,  3.63381375e-01,
         2.98750947e-01,  2.47822734e-01,  2.00699274e-01,  1.22125381e-01,
         5.35453940e-02,  7.46638293e-02, -9.92724661e+11, -9.94211961e+11,
        -9.94211961e+11,  5.76985959e+11,  5.76985959e+11,  5.76122811e+11,
        -2.56067734e+11, -2.56067734e+11, -2.56195395e+11]),
 4.55551579740905)
```

✓ 선형 회귀 모델 훈련

- LinearRegression() 모델을 X_train(훈련 데이터)과 log_y_train(로그 변환된 타깃 변수)으로 학습시킴.

✓ 회귀 계수(coef_) 및 절편(intercept_) 출력

- **coef_** : 학습된 모델의 특성별 회귀 계수(가중치)를 나타냄
  : coef_ 배열의 각 숫자는 각 특성이 타깃 값에 미치는 영향력을 의미.
- **intercept_** : 모든 특성이 0일 때 예측되는 값 (절편값)

모델링&평가 - GridSearchCV

```
from sklearn.model_selection import GridSearchCV

param_grid_ridge = {'하이퍼파라미터_이름': ['튜닝할 값 리스트']}
'객체명' = GridSearchCV('모델이름()', param_grid_ridge, cv='교차검증 횟수', scoring='r2', n_jobs=-1)
'객체명'.fit(X_train, log_y_train)

print("최적의 하이퍼파라미터:", '객체명'.best_params_)
'변수명' = '객체명'.best_estimator_
print('변수명')
```

(GridSearchCV 사용방법)

GridSearchCV는 머신러닝 모델의 **하이퍼파라미터를 최적화**하는 방법 중 하나이다.

여러 개의 하이퍼파라미터를 조합해보고, 가장 좋은 성능을 내는 조합을 선택 (단, 연산 속도가 느림.)

모델링&평가 - Ridge Regression

```python
from sklearn.model_selection import GridSearchCV

param_grid_ridge = {'alpha': [0.0001, 0.001, 0.01, 0.1, 1]}
ridge = GridSearchCV(Ridge(), param_grid_ridge, cv=5, scoring='r2', n_jobs=-1)
ridge.fit(X_train, log_y_train)

print("최적의 하이퍼파라미터:", ridge.best_params_)
best_ridge = ridge.best_estimator_
print(best_ridge)
```

```
최적의 하이퍼파라미터: {'alpha': 0.1}
Ridge(alpha=0.1)
```

```python
# 최적의 Ridge 모델 가져오기
print("Ridge 회귀 계수 (Weights):", best_ridge.coef_)
print("Ridge 절편 (Intercept):", best_ridge.intercept_)
```

```
Ridge 회귀 계수 (Weights): [-6.24406809e-03 -3.81395288e-02  2.14060419e-01 -5.59708349e-02
 -2.76453164e-02  2.34165238e-02  2.43938172e-01 -1.99534149e-01
  3.96425761e-02  7.41855510e-02 -3.09513714e-04 -1.39956630e-01
  2.32651759e-02 -1.38583623e-01 -2.33157908e-01 -3.43396097e-01
 -4.04412353e-01 -1.94975202e-01  5.77518252e-02  2.49036736e-01
  3.81380652e-01  3.12328039e-01  2.47785792e-01  2.69949286e-01
  3.05703498e-01  3.05332454e-01  2.90336844e-01  2.98525300e-01
  3.52671952e-01  4.31999113e-01  4.18040234e-01  3.63361319e-01
  2.98378867e-01  2.47548603e-01  2.00066056e-01  1.21764362e-01
  5.32813687e-02  7.48338652e-02 -2.09414353e-02  4.70537886e-02
  3.59015355e-02  2.88999818e-02  4.32546512e-02  6.02553779e-02
  4.98758747e-02  3.89071783e-02  2.73842723e-02]
Ridge 절편 (Intercept): 4.552990076444588
```

ridge.best_params_: 최적의 alpha 값을 출력
best_ridge.coef_: 최적 모델의 가중치 (Feature Weights)
best_ridge.intercept_: 최적 모델의 절편 (Intercept)

모델링&평가 - Lasso Regression

```python
# 최적의 파라미터 찾기 : GridSearchCV
from sklearn.model_selection import GridSearchCV

param_grid_lasso = {'alpha': [0.0001, 0.001, 0.01, 0.1, 1]}

lasso = GridSearchCV(Lasso(), param_grid_lasso, cv=5, scoring='r2', n_jobs=-1)
lasso.fit(X_train, log_y_train)

print("최적의 하이퍼파라미터:", lasso.best_params_)
best_lasso = lasso.best_estimator_
print(best_lasso)
```

```
최적의 하이퍼파라미터: {'alpha': 0.0001}
Lasso(alpha=0.0001)
```

```python
best_lasso = lasso.best_estimator_

print("Lasso 회귀 계수 (Weights):", best_lasso.coef_)
print("Lasso 절편 (Intercept):", best_lasso.intercept_)
```

```
Lasso 회귀 계수 (Weights): [-6.14995773e-03 -3.80088119e-02  2.14478784e-01 -5.60872659e-02
 -2.74480827e-02  2.33103981e-02  2.43802083e-01 -2.66003013e-01
  0.00000000e+00  4.96638129e-02 -2.31245334e-04 -1.39845881e-01
  2.31516516e-02 -1.39674397e-01 -2.34235844e-01 -3.44447712e-01
 -4.05476072e-01 -1.96046429e-01  5.64852735e-02  2.47763233e-01
  3.80102321e-01  3.11024093e-01  2.46460332e-01  2.68606063e-01
  3.04349332e-01  3.03961386e-01  2.88957514e-01  2.97143758e-01
  3.51291352e-01  4.30631794e-01  4.16678005e-01  3.62018548e-01
  2.97052265e-01  2.46233318e-01  1.98756134e-01  1.20470147e-01
  5.28266803e-02  7.43110269e-02 -3.87106641e-02  2.90607763e-02
  1.78422433e-02 -1.43404928e-02 -0.00000000e+00  1.69598137e-02
  2.22542264e-02  1.13778790e-02 -0.00000000e+00]
Lasso 절편 (Intercept): 4.552986569071433
```

lasso.best_params_: 최적의 alpha 값을 출력
best_lasso.coef_: 최적 모델의 가중치 (Feature Weights)
best_lasso.intercept_: 최적 모델의 절편 (Intercept)

모델링&평가 - Lasso Regression

```python
# 최적의 파라미터 찾기 : GridSearchCV
from sklearn.model_selection import GridSearchCV

param_grid_sgd = {
    'alpha': [0.0001, 0.001, 0.01], # 정규화 강도 (L1, L2 규제의 세기)
    'penalty': ['l1', 'l2'],    # 정규화 방식 선택 (Lasso vs Ridge)
    'learning_rate': ['constant', 'optimal', 'invscaling'] # 학습률 스케줄링 방식
}

sgd = GridSearchCV(SGDRegressor(max_iter=1000, tol=1e-3), param_grid_sgd, cv=5, scoring='r2', n_jobs=-1)
sgd.fit(X_train, log_y_train)
```

```python
best_sgd = sgd.best_estimator_
print("sgd 회귀 계수 (Weights):", best_sgd.coef_)
print("sgd 절편 (Intercept):", best_sgd.intercept_)
```

```
sgd 회귀 계수 (Weights): [ 0.         -0.03029063  0.26916463 -0.0671943  -0.00484829  0.01362529
  0.23158587 -0.19359781  0.          0.0825838   0.         -0.13059263
  0.         -0.21359456 -0.33030014 -0.45456897 -0.48763389 -0.30590366
 -0.02262305  0.12824374  0.25679884  0.20093466  0.11563806  0.13395438
  0.17411732  0.17212341  0.17206925  0.16642651  0.22898554  0.30645269
  0.29469176  0.24294444  0.17418818  0.13215107  0.08608146  0.
  0.          0.01493578 -0.01658545  0.02746585  0.         -0.01167918
  0.          0.01472026  0.          0.          0.        ]
sgd 절편 (Intercept): [4.55977744]
```

sgd.best_params_: 최적의 alpha 값을 출력
best_sgd.coef_: 최적 모델의 가중치 (Feature Weights)
best_sgd.intercept_: 최적 모델의 절편 (Intercept)

모델링&평가

모델 / 평가지표	R^2	MAE	MSE	RMSE	RMSLE
Linear Regression	0.8265	0.4572	0.3853	0.6208	0.5833
Ridge Regression	0.826534	0.45751	0.38522	0.62066	0.58327
Lasso Regressioin	0.826533	0.45754	0.38255	0.62069	0.58328
SGD Regression	0.8164	0.4712	0.4085	0.6391	0.6008

R^2 (결정계수) : 모델의 성능을 나타내는 지표로 1에 가까울수록 좋음.

MAE : 오차의 평균 절댓값을 나타내며, 낮을수록 좋음

MSE : 오차를 제곱하여 평균을 낸 값 → 큰 오차가 있을 경우 더 큰 영향을 받음.

RMSE : MSE의 제곱근으로, 실제 데이터 단위와 같아 해석이 용이함.

RMSLE : 로그 변환을 사용한 RMSE로, 상대적 오차를 평가하는 데 유용.

결론

<성능>

- Linear, Ridge, Lasso 모델은 성능이 거의 동일하게 보여짐

  : 데이터 자체가 선형 관계를 충분히 따르고 있음.

- SGD Regression은 다른 회귀 모델에 성능이 낮게 나타남.

<최적 모델 선택>

- Linear, Ridge, Lasso 모두 성능 차이가 거의 없으므로, Linear Regression이 가장 간단하고 효율적

- 하지만 과적합이 의심되기에, Ridge Regression 또는 Lasso Regression을 선택하는 것이 바람직하다고 생각함.

운영 전략

<시간대별 패턴 반영>

- 대여 수요가 적은 시간대에는 추가 할인을 제공하여 많은 이용을 유도
- 출퇴근 시간대에 수요가 높은 지역에 자전거를 추가 배치

<요일별 패턴 반영>

- 평일 : 회사 밀집 지역에 추가 배치
- 주말 : 관광지나 공원 주변에 추가 배치