서울시 공공 데이터 기반 서울시 생활인구 예측

김영우

프로젝트 개요 - 요약

주제	서울시 공공 데이터 기반 생활인구 예측
사전학습과목	데이터전처리, 데이터분석, 머신러닝, 딥러닝
데이터 출처	https://data.seoul.go.kr/dataVisual/seoul/seoulLivingPopulation.do
데이터 구분	Tabular
문제 유형	Regression
중점사항	▪ 시계열 데이터 전처리, 단변량/이변량분석, ML/DL 모델링

데이터 소개 – 서울 생활 인구의 정의

서울 생활 인구의 정의

서울시가 보유한 빅데이터와 KT의 통신데이터로 측정한
특정 시점에 **서울의 특정 지역**에 **존재하는 인구**

특정 시점 : 1시간 단위의 시각(時刻, time),을 의미함(00시, 01시, 02시, … 23시)
서울의 특정 지역 : 서울시(전체), 자치구(25개), 행정동(424개), 집계구역(19,153개)의 각 단위를 의미
존재하는 인구 : 서울을 커버하는 KT의 통신기지국에 존재하는 인구를 바탕으로 한 추정 인구

용어 정리

용 어	개 념
상주인구 (야간인구)	통계청에서는 인구 센서스를 통해 거주인구를 조사하는데 이를 상주인구 또는 야간인구라 함
유입인구	인구 센서스에서는 통근/통학인구를 조사하는데, 통근/통학을 통해 해당지역에 유입되는 인구를 말함
유출인구	인구 센서스에서는 통근/통학인구를 조사하는데, 통근/통학을 통해 해당지역에서 타지역으로 유출되는 인구를 말함
주간인구	상주인구에서 유입인구를 더하고 유출인구를 뺀 인구를 말함
등록인구	행정기관(주민센터)에 등록하는 주민등록인구를 말함
경제활동인구	만 15세 이상의 생산가능 연령 인구 중에서 구직활동이 가능한 취업자 및 실업자를 말함
생활인구	특정시점에 특정지역에 존재하는 모든 인구이며, 현주인구(de facto Population) 또는 현재인구, 서비스 인구라고도 함

데이터 소개

Train data set

data2017.csv
data2018.csv
data2019.csv
data2020.csv
data2021.csv

(5년치 데이터 활용)

Test data set

data2022.csv

(반년치 데이터 활용)

문제 정의

문제 정의
DEFINITION

- **목표**

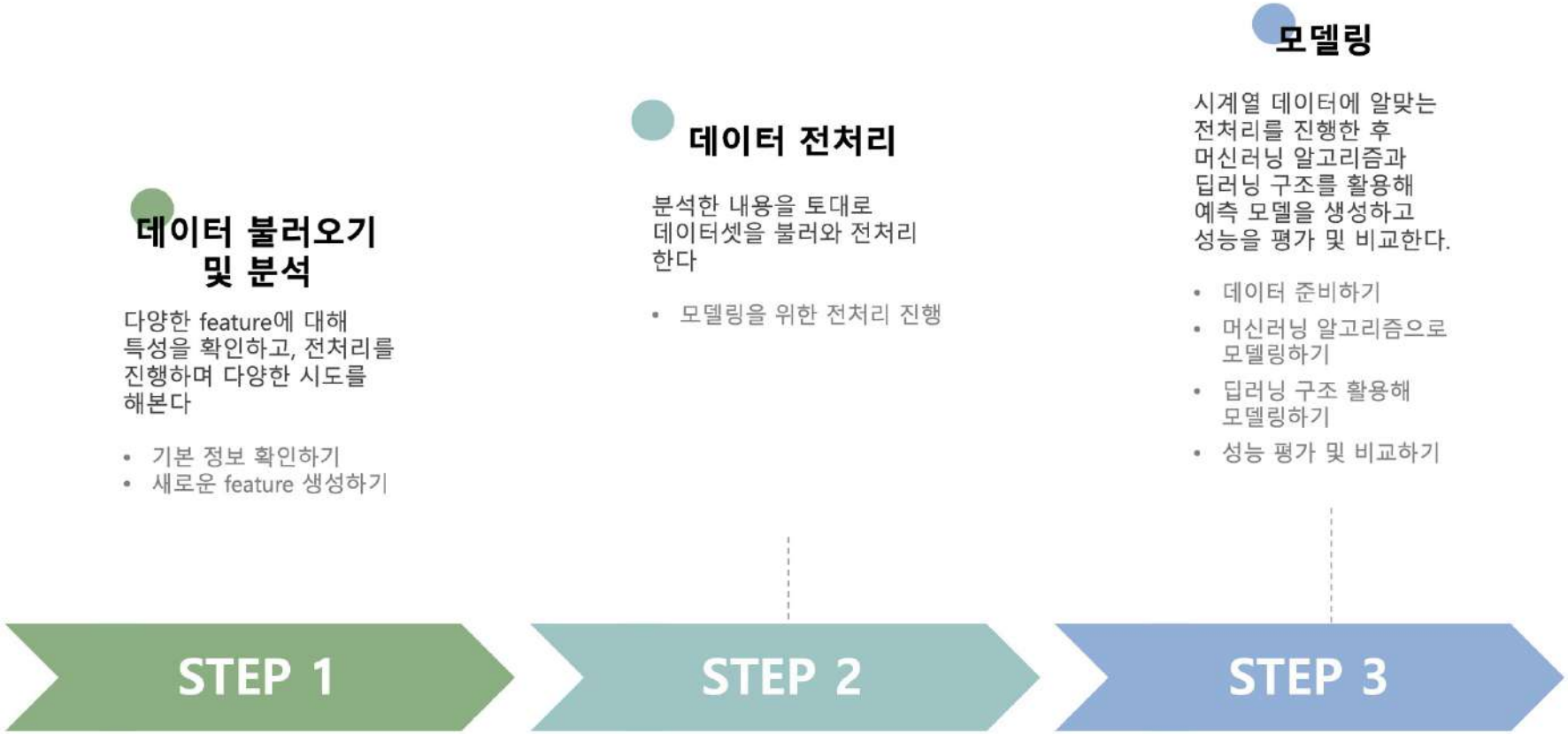
  서울의 생활인구 데이터를 분석하여, 특정 지역의 생활인구를 파악하고 해당 지역의 생활인구를 예측 한다

문제는 서울시 어느구 어느동 데이터인지 알 수 없다!!! 자신이 직접 찾아야 한다.

과제 핵심 사항

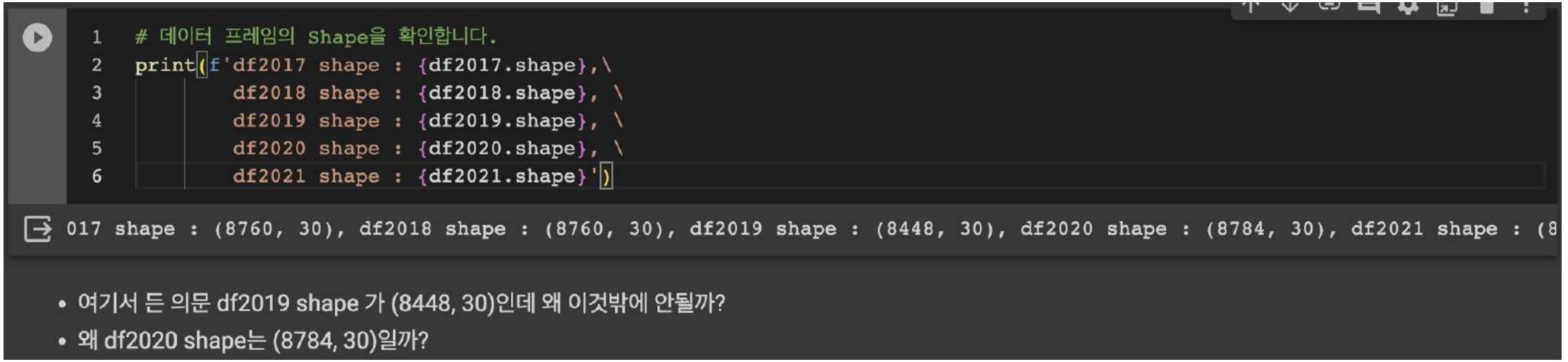

Challenge
과제 수행 전략
시계열에 맞는 데이터 구조 만들기
Target 구성
- 미래의 생활인구를 예측하기 위해 적절한 데이터 구조를 어떻게 만들까?
- 추가 데이터를 어떻게 활용할까?
Feature 구성
- 과거의 정보, 최근 추세 등을 feature로 어떻게 담아낼까?
- 각 정보들은 어떤 주기(요일, 월, 계절 등)의 패턴을 가지고 있을까?
사전확인 오류를 피하기 위한 target
✔여러 칼럼의 데이터를 합쳐서 사용
시간의 흐름을 담은 feature
✔파악된 주기에 맞는 feature 생성하기
생활인구와 관련 있는 새로운 특징 발굴
최적의 모델 선정
모델링
- 가장 성능이 좋은 모델을
- 어떻게 찾을 수 있을까?
ML, DL의 다양한 알고리즘 적용
✔알고리즘별 특성을 고려하여 모델링 수행
✔여러 구조의 딥러닝 모델링 시도
✔예측결과 분석 : 라인차트로 실제값과 예측값 비교

문제 해결 프로세스


데이터 불러오기 및 분석
다양한 feature에 대해 특성을 확인하고, 전처리를 진행하며 다양한 시도를 해본다
• 기본 정보 확인하기
• 새로운 feature 생성하기
데이터 전처리
분석한 내용을 토대로 데이터셋을 불러와 전처리 한다
• 모델링을 위한 전처리 진행
모델링
시계열 데이터에 알맞는 전처리를 진행한 후 머신러닝 알고리즘과 딥러닝 구조를 활용해 예측 모델을 생성하고 성능을 평가 및 비교한다.
• 데이터 준비하기
• 머신러닝 알고리즘으로 모델링하기
• 딥러닝 구조 활용해 모델링하기
• 성능 평가 및 비교하기
STEP 1
STEP 2
STEP 3

1. 데이터 불러오기 및 분석

주요 쟁점 : 정상적이라고 판단되는 shape가 (8760, 30)인데
2019년을 의미하는 df2019.shape는 왜 (8448, 30) 이고
2020년을 의미하는 df2020.shape는 왜 (8784, 30)인가?



1. 데이터 불러오기 및 분석

주요 쟁점 : 2019년의 경우 10월 15일 ~ 10월 27일 데이터가 없다.

2020년은 윤년이기 때문에 2월 29일까지 있다.
따라서 2월 29일 0시부터 11시까지의 총 24건의 데이터가 더 있기 때문에 (8784, 30)이다.

- 2019년의 경우 2019년 10월 15일 ~ 2019년 10월 27일 데이터가 없다.
- 2020년의 경우 2020년 2월 29일까지 있어서 24건이 추가됐다

1. 데이터 불러오기 및 분석

주요 쟁점 : 현재 없는 2019년 10월 15일 ~ 2019년 10월 27일 데이터는
2018년 10월 15일 ~ 2018년 10월 27일로 채운다.

다른 연도의 데이터로 채울 수 있지만, 2020년, 2021년은 코로나 라는 예외가 존재할 수 있겠다 싶어
그나마 생활 인구 분포가 비슷할 것 같다는 2018년을 선택했었다.

- 2020년, 2021년은 코로나 때문에 2019년 10월 15일 ~ 2019년 10월 27일 데이터는 2018년 10월 15일 ~ 2018년 10월 27일로 채운다.

```
[ ]  1  mini_2018 = df2018.loc[df2018.index.isin(range(20181015, 20181028, 1))] # 2018년 10월 15일 ~ 2018년 10월 27일 데이터
     2
     3  mini_2018.index = mini_2018.index + 10000 # 기준일ID '20181015' 같은 것을 '20191015'로 변환
     4
     5  df2019 = pd.concat([df2019, mini_2018], axis=0) # 아래 방향으로 데이터를 합친다.
     6
     7  df2019 = df2019.sort_values(by=['기준일ID', '시간대구분'], ascending=True) # '기준일 ID'을 먼저 오름차순 진행, 그 다음으로 '시간대
```

1. 데이터 불러오기 및 분석

주요 쟁점 : 2020년 2월 29일 데이터는 삭제한다.

지금 와서 생각해보면 굳이 안해줘도 되는 작업이지만
보통 2월이 28일까지 있다는 점을 생각해서 통일성을 잡아주기 위해 2020년 2월 29일
데이터를 삭제했던 것 같다.

- 2020년 2월 29일 데이터는 삭제한다.

```
[ ]   1   df2020 = df2020.loc[df2020.index != 20200229]
```

1. 데이터 불러오기 및 분석

주요 쟁점 : 2017년부터 2021년까지 데이터를 통합한다.

- df2017, df2018, df2019, df2020, df2021를 합쳐서 df_total 이라는 이름을 명명한다.

```
[ ]  1  df_total = pd.concat([df2017, df2018, df2019, df2020, df2021], axis=0) # 세로 방향으로 데이터프레임 통합
     2
     3  df_total.shape
```

```
(43800, 30)
```

1. 데이터 불러오기 및 분석

주요 쟁점 : 평가 데이터로 활용할 2022년 반년치 데이터를 가져온다. (즉 2022년 1월 ~ 2022년 6월)

```
df_test = pd.read_csv('/content/drive/MyDrive/미니프로젝트4차part1(복습)/2023.10.12_미니프로젝트4차_데이터/data2022.csv', sep=

df_test.shape
```

```
(4344, 30)
```

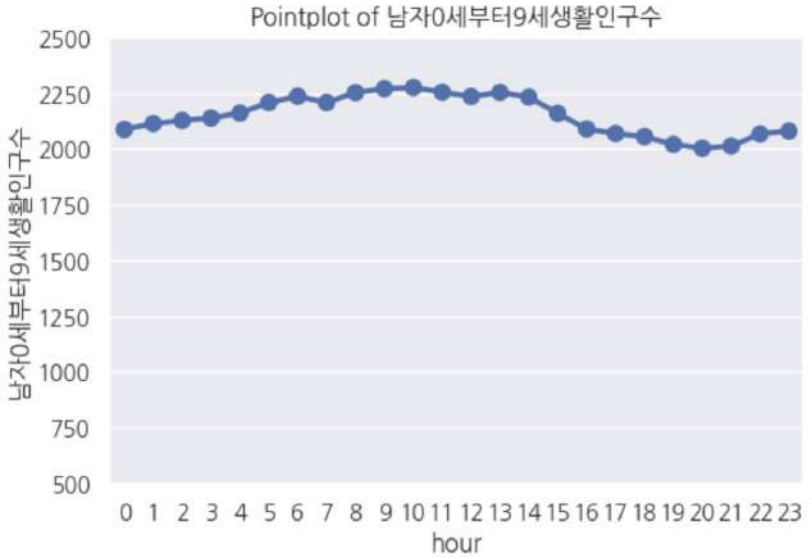
1. 데이터 불러오기 및 분석

주요 쟁점 : 시간대별로 따라 세대별 생활 인구 흐름을 파악했다.

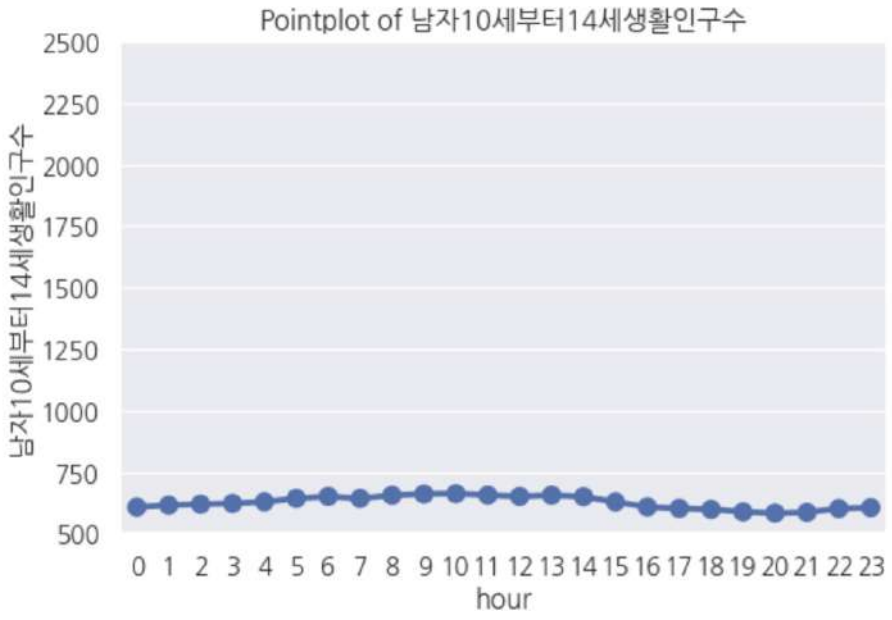
- 계절별, 연도별, 달별로도 총생활인구수를 분석할 수 있지만,
  아무래도 시간별로 총인구수를 분석하는 것이 직관적이고 가장 큰 연관이 있지 않을까 생각했다.
  - 세대별로 분석하여 시간대별 총생활인구수가 비교적 유동적인지, 고정적인지를 확인하고 싶었다.

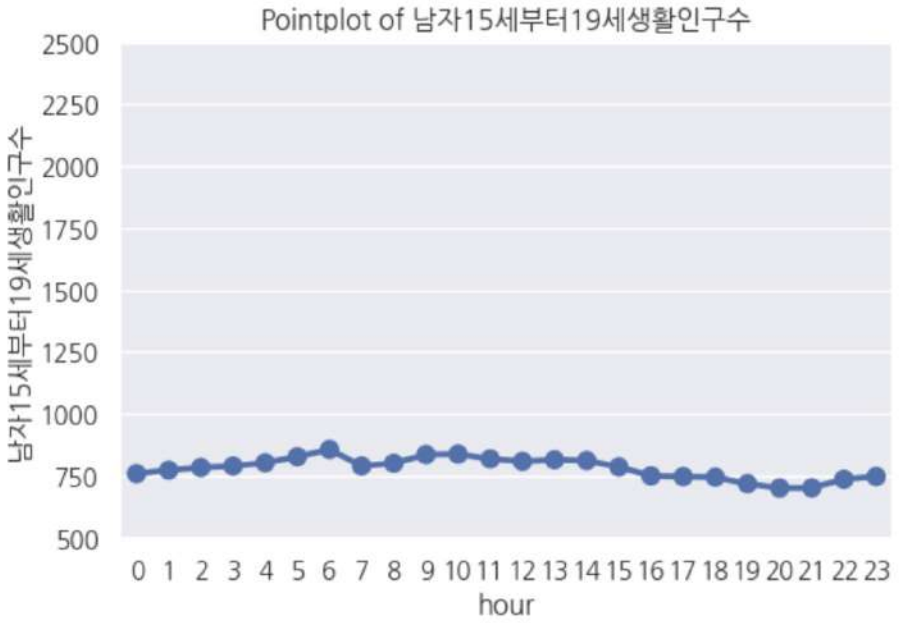
```
#시간대(hour)별 '총생활인구수','남자0세부터9세생활인구수' ~ '여자70세이상생활인구수' 분석
n = len(cols_to_plot)
fig, axes = plt.subplots(n, 1, figsize=(6, 4*n))

for i, col in enumerate(cols_to_plot):
    sns.pointplot(x=df_total['시간대구분'], y=df_total[col], ax=axes[i])
    axes[i].set_title(f'Pointplot of {col}')
    axes[i].set_xlabel('hour')
    axes[i].set_ylabel(col)

    if col != '총생활인구수':
        axes[i].set_ylim([500, 2500])

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

1. 데이터 불러오기 및 분석

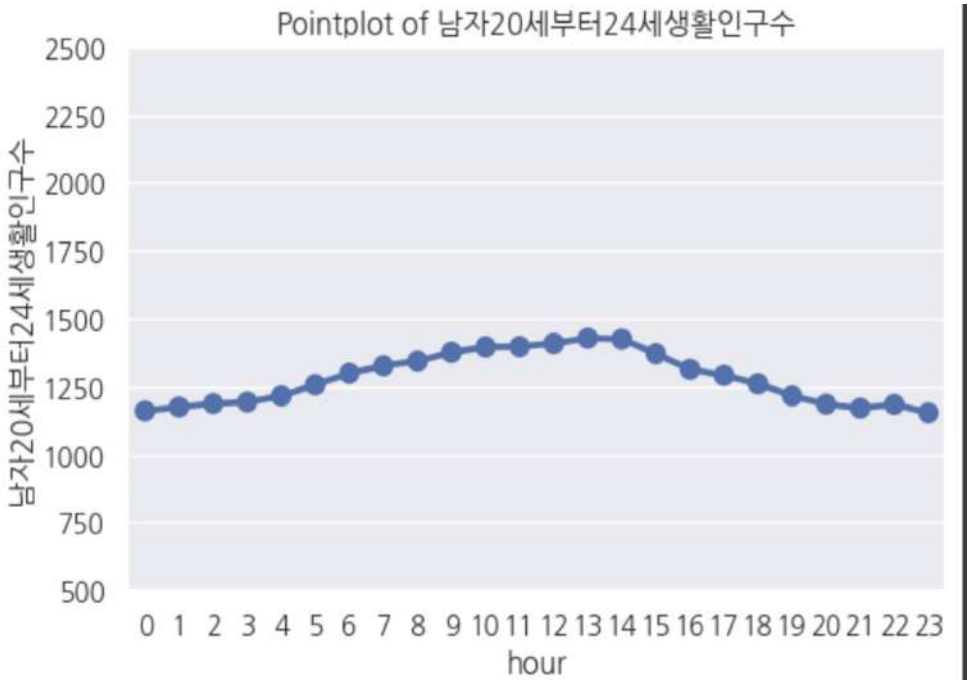


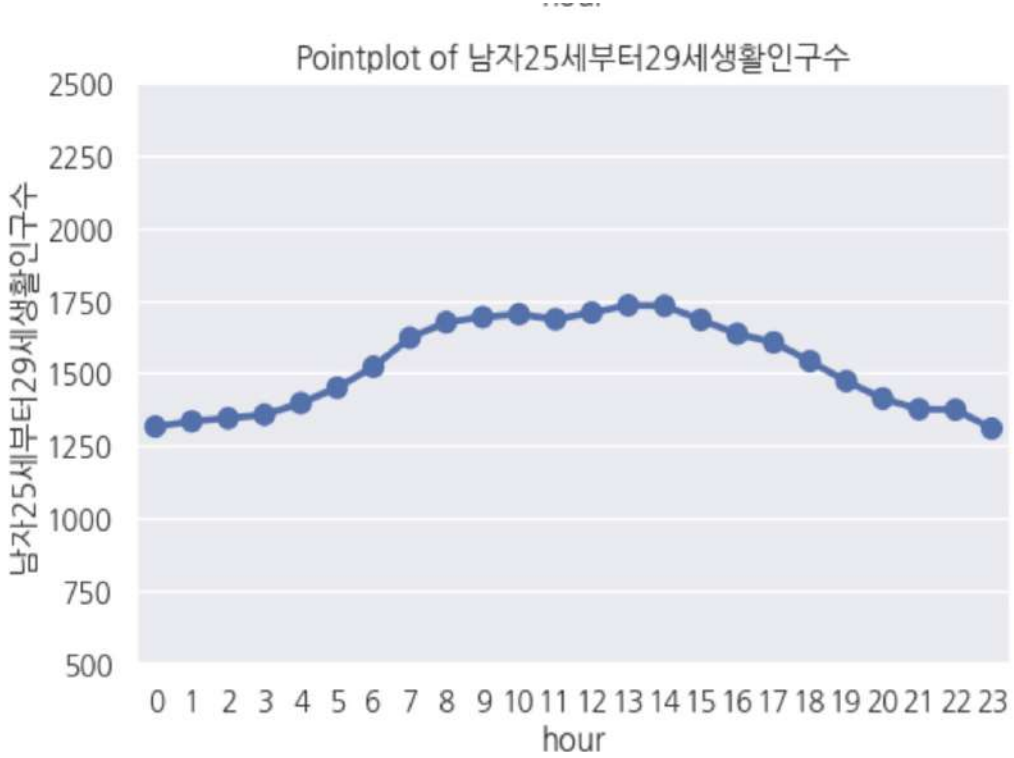
1. 데이터 불러오기 및 분석



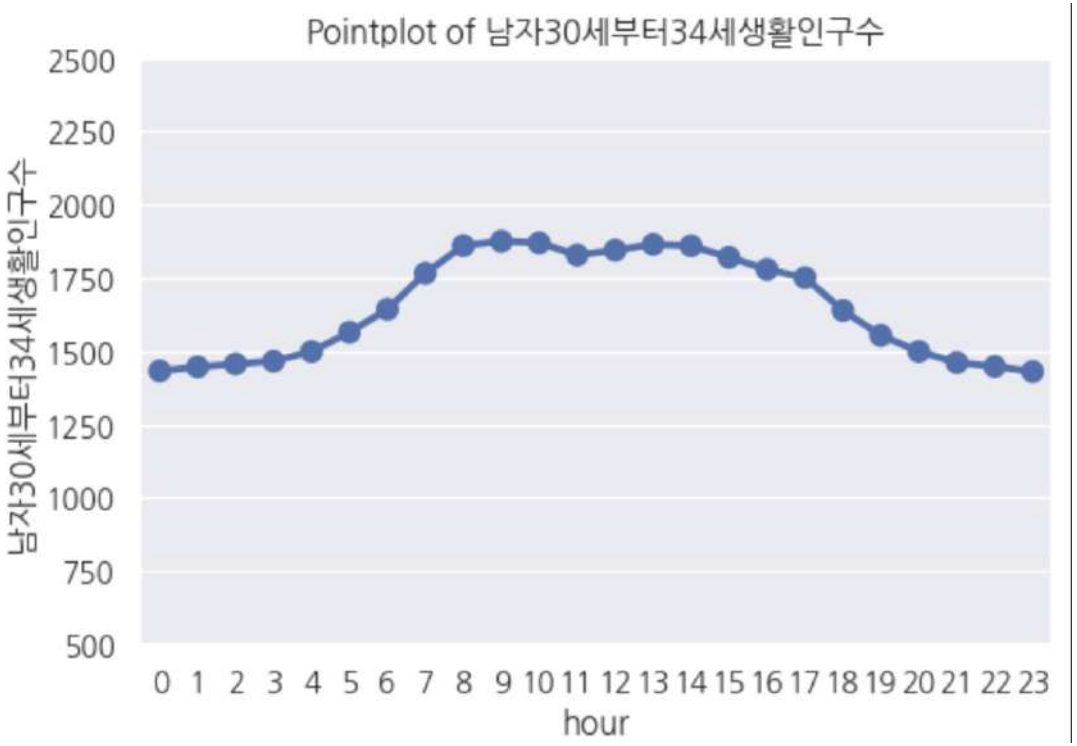


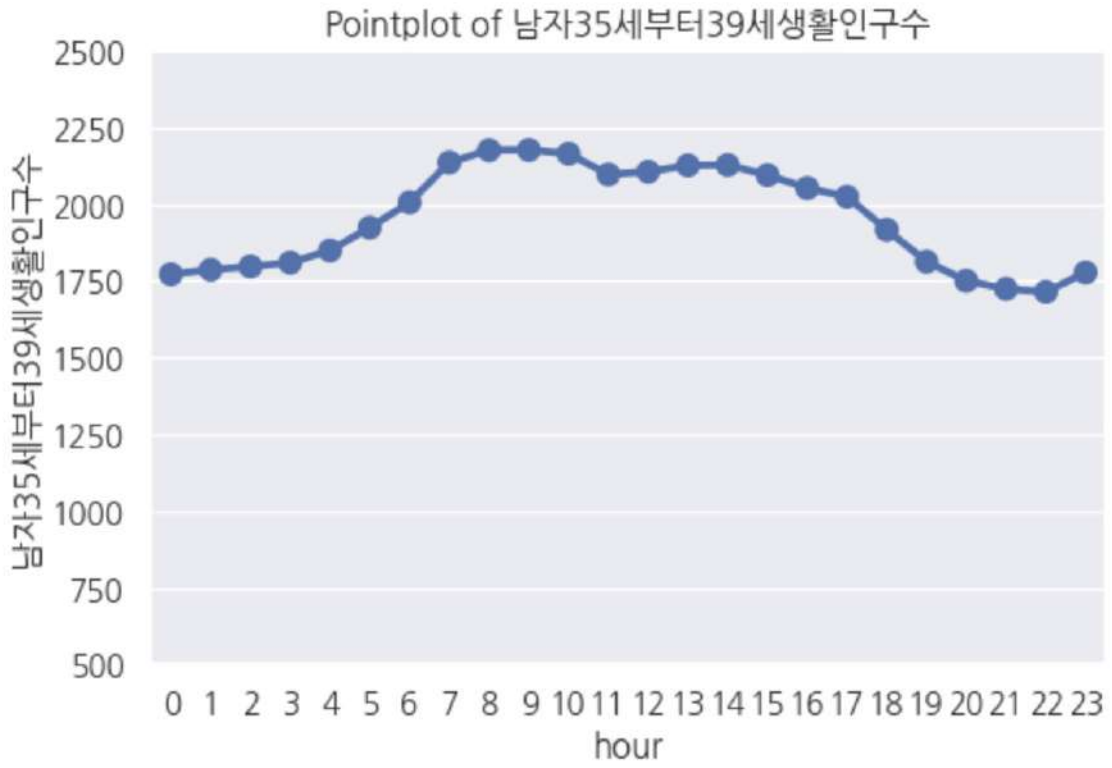
1. 데이터 불러오기 및 분석



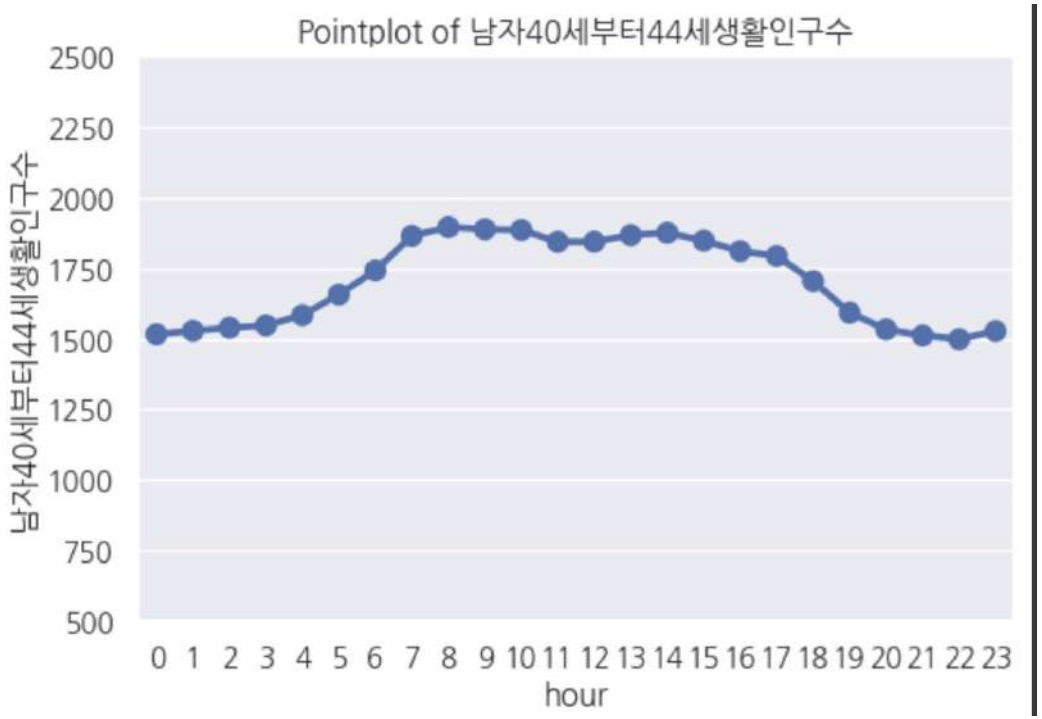


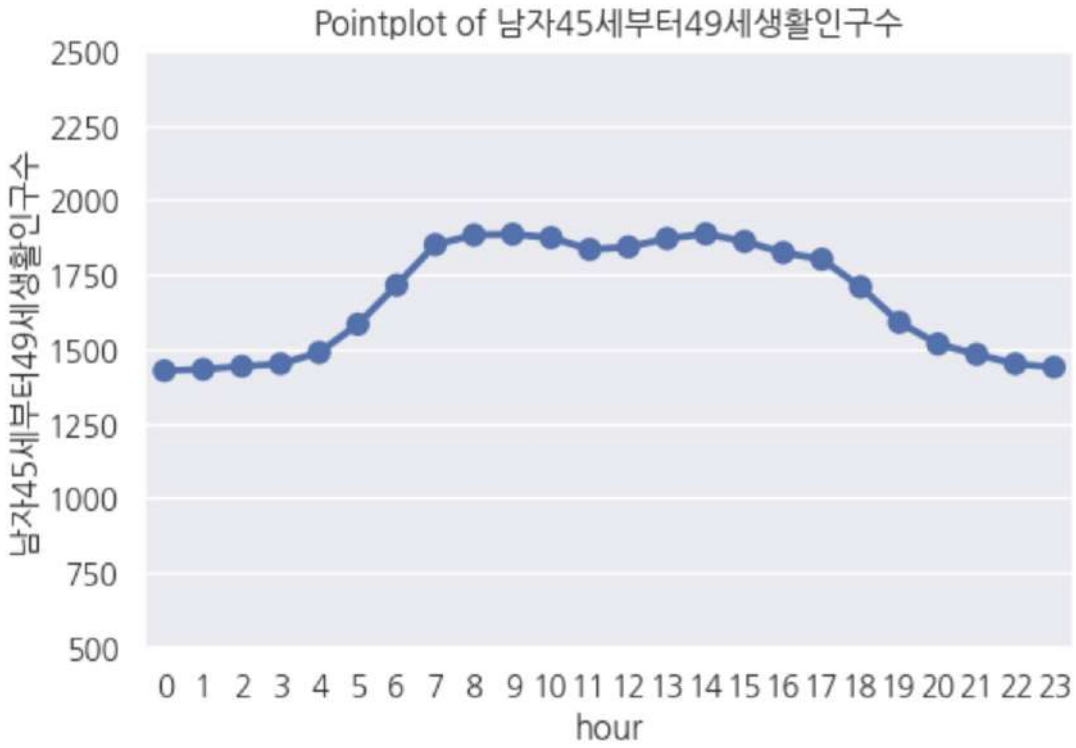
1. 데이터 불러오기 및 분석



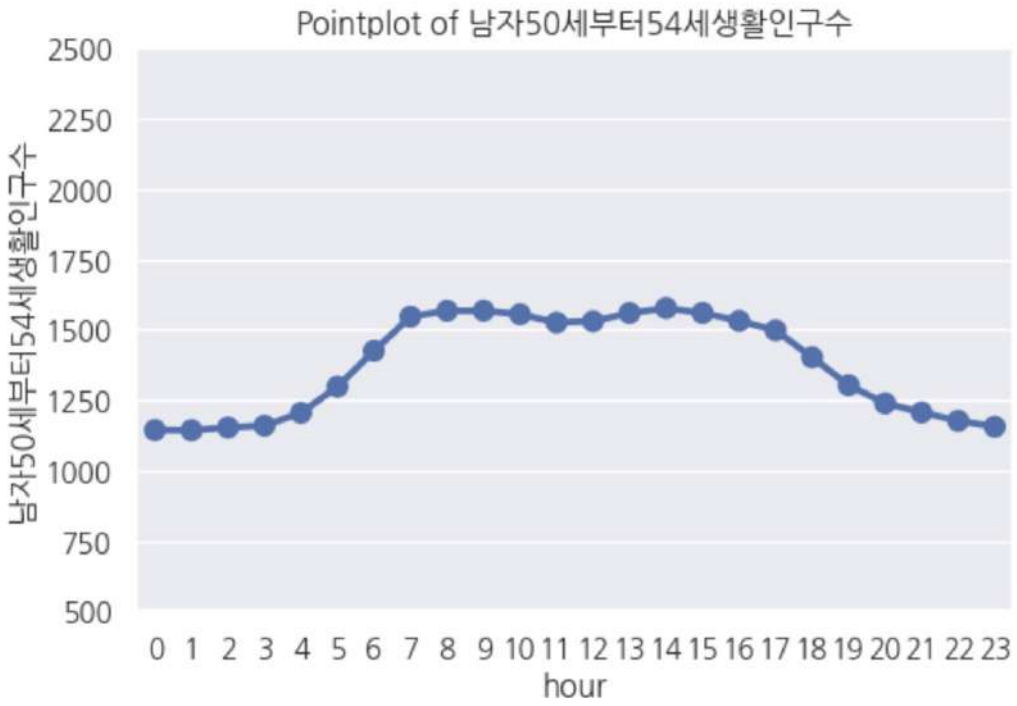


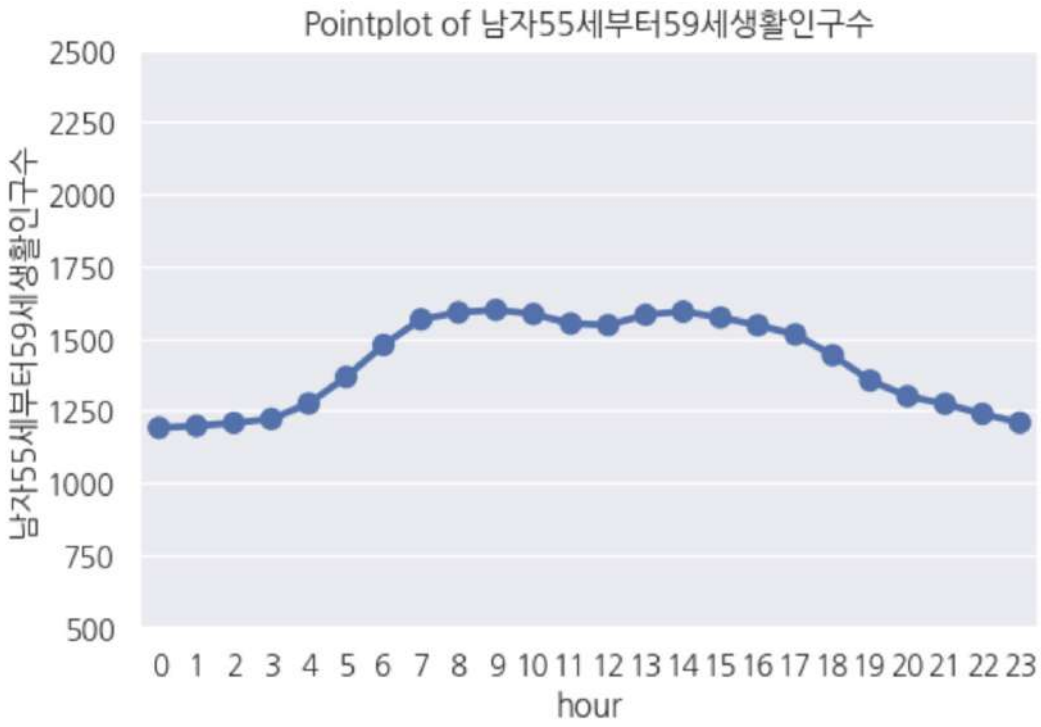
1. 데이터 불러오기 및 분석



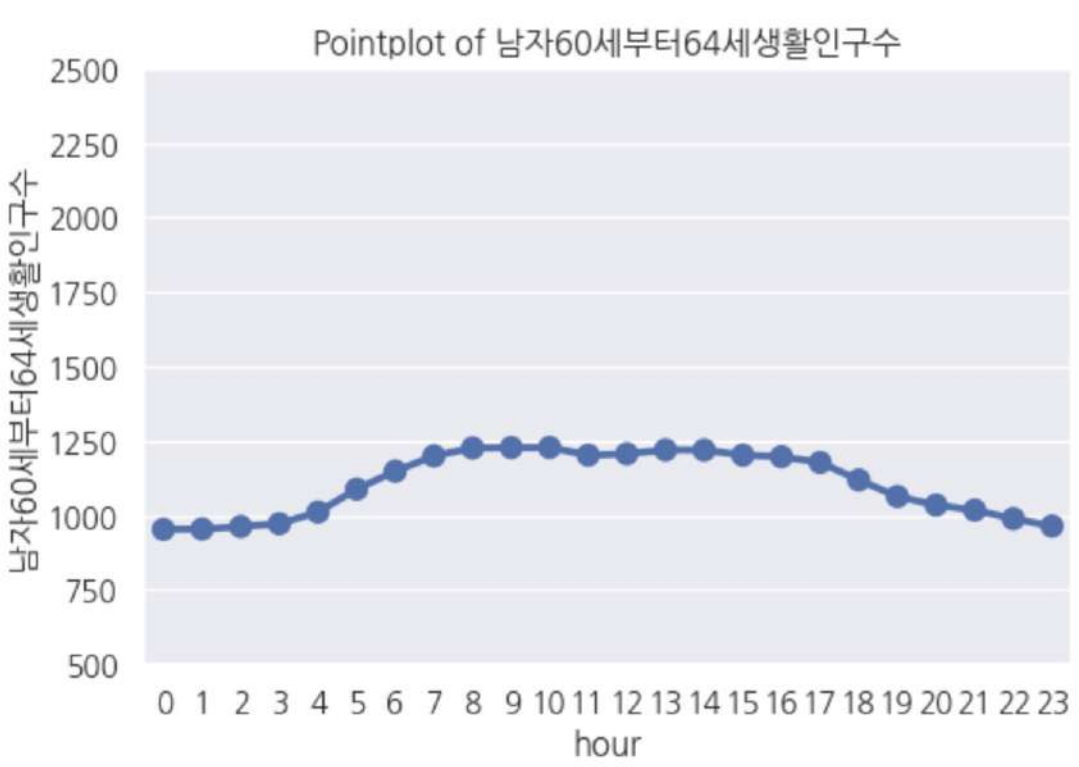


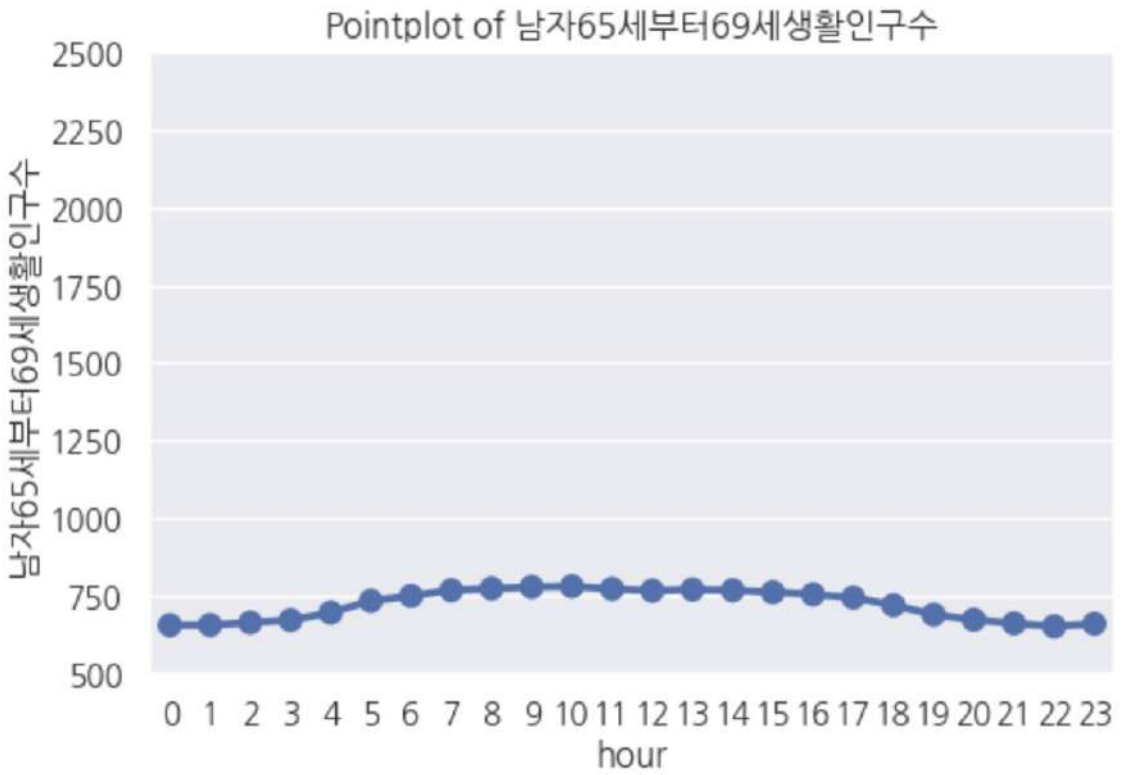
1. 데이터 불러오기 및 분석



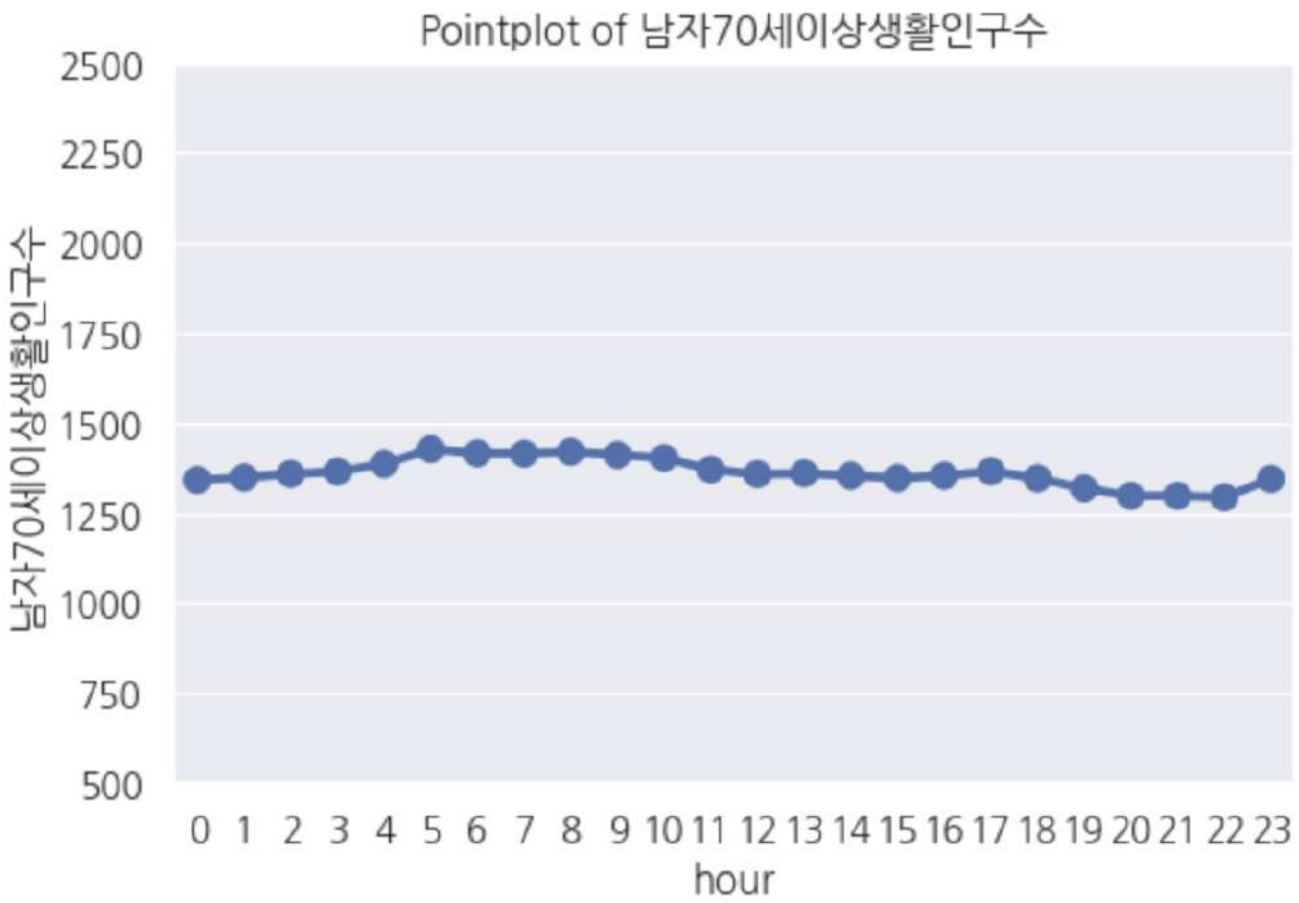


1. 데이터 불러오기 및 분석

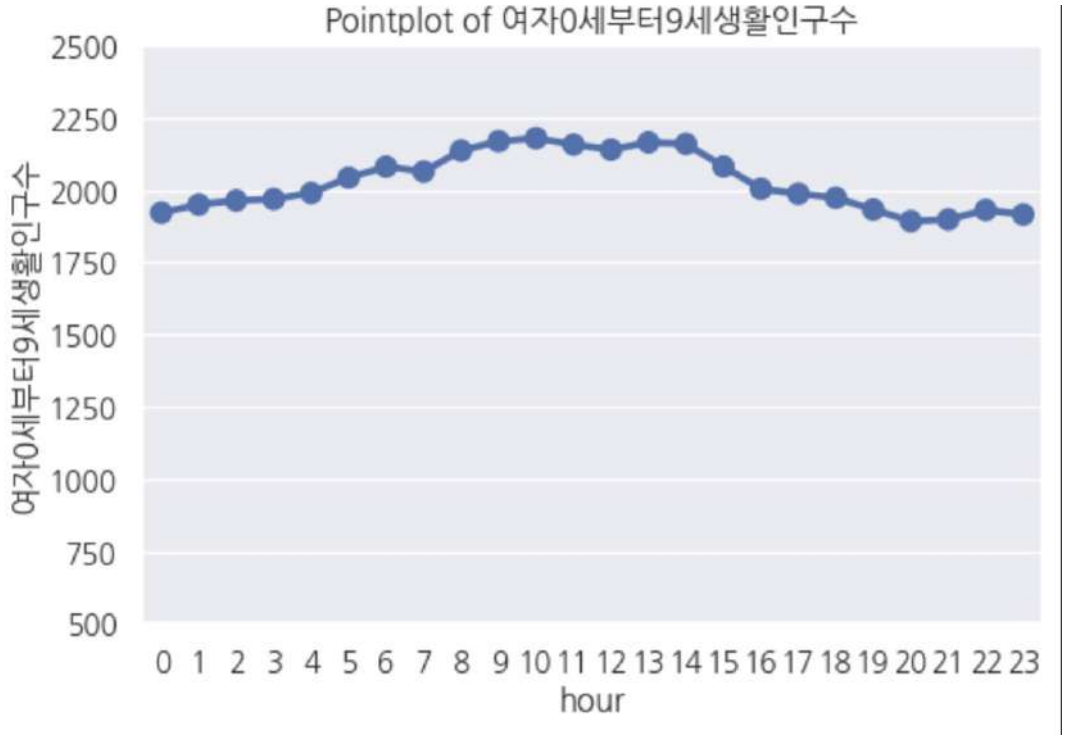


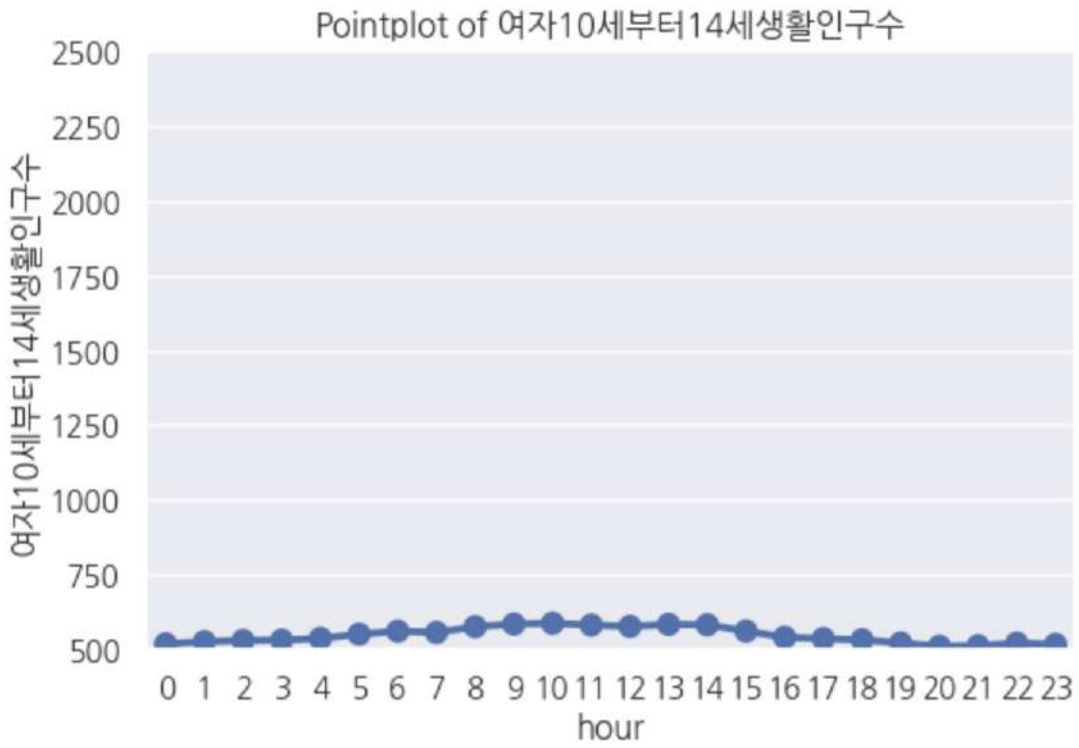


1. 데이터 불러오기 및 분석

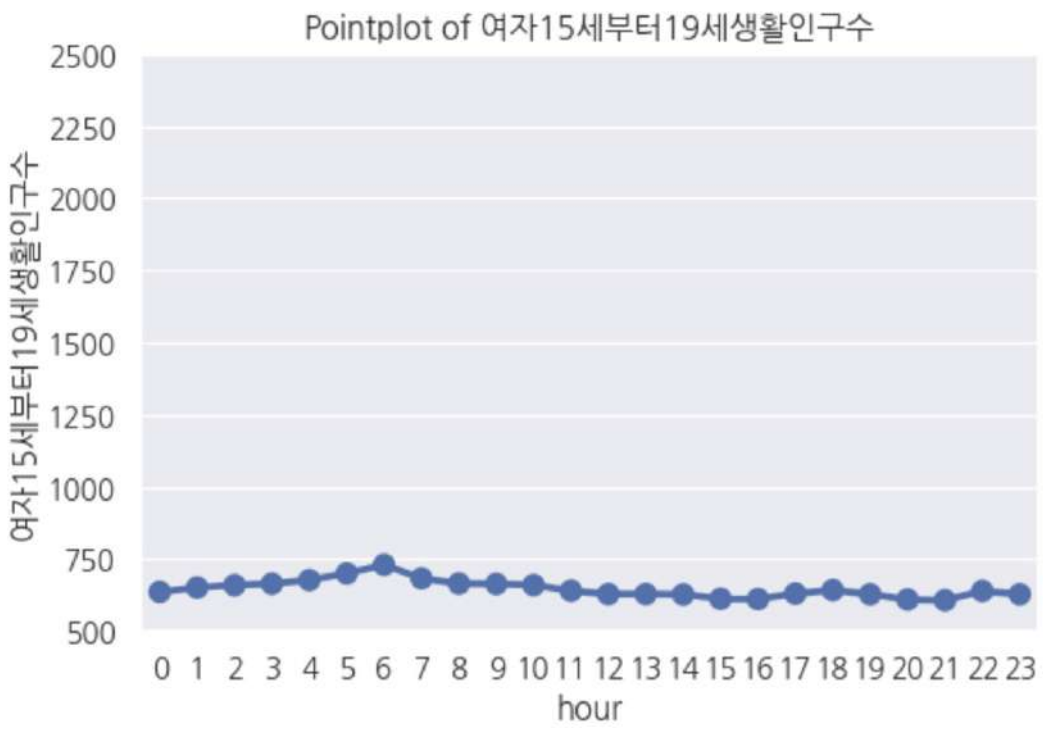


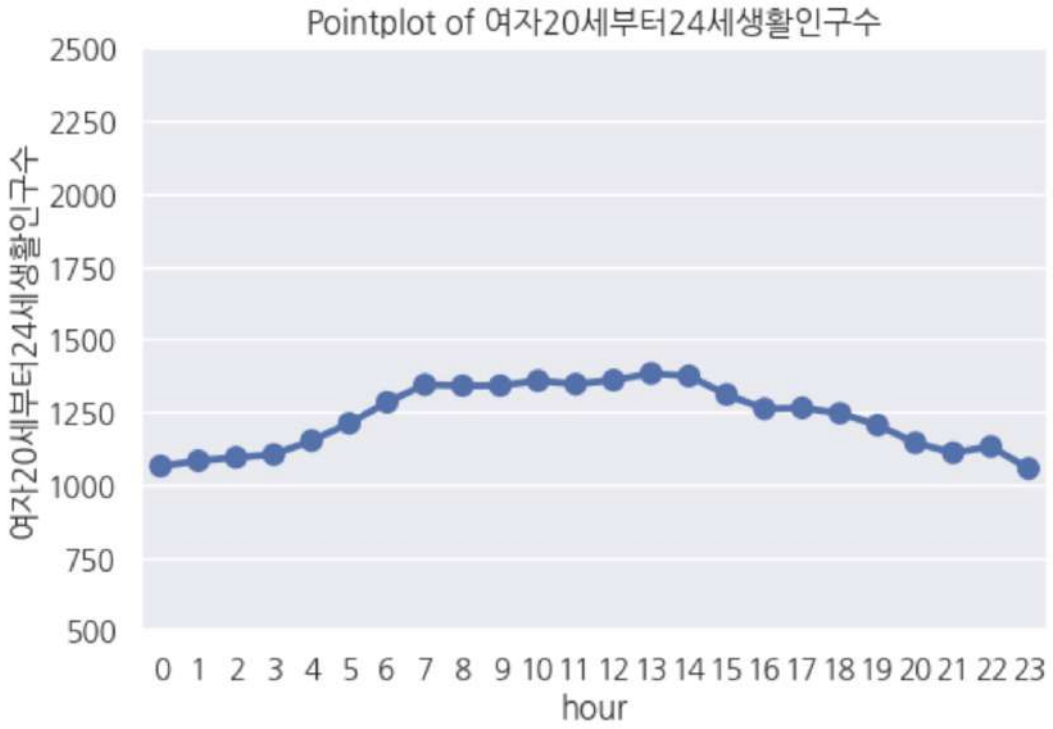
1. 데이터 불러오기 및 분석



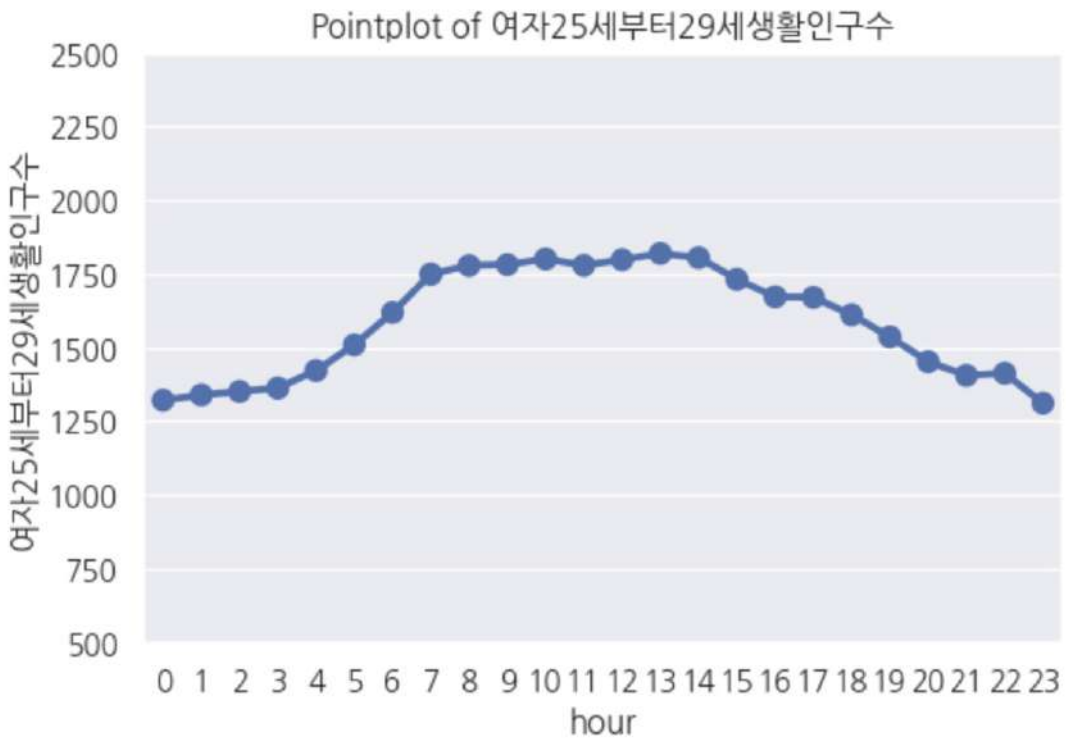


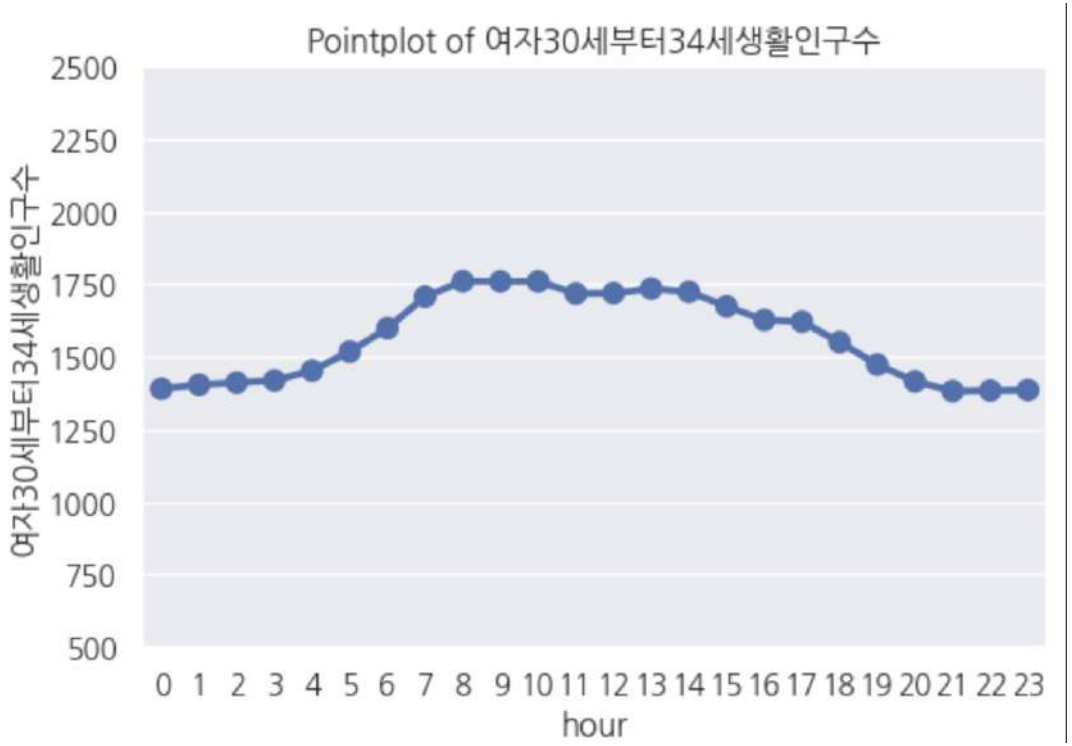
1. 데이터 불러오기 및 분석



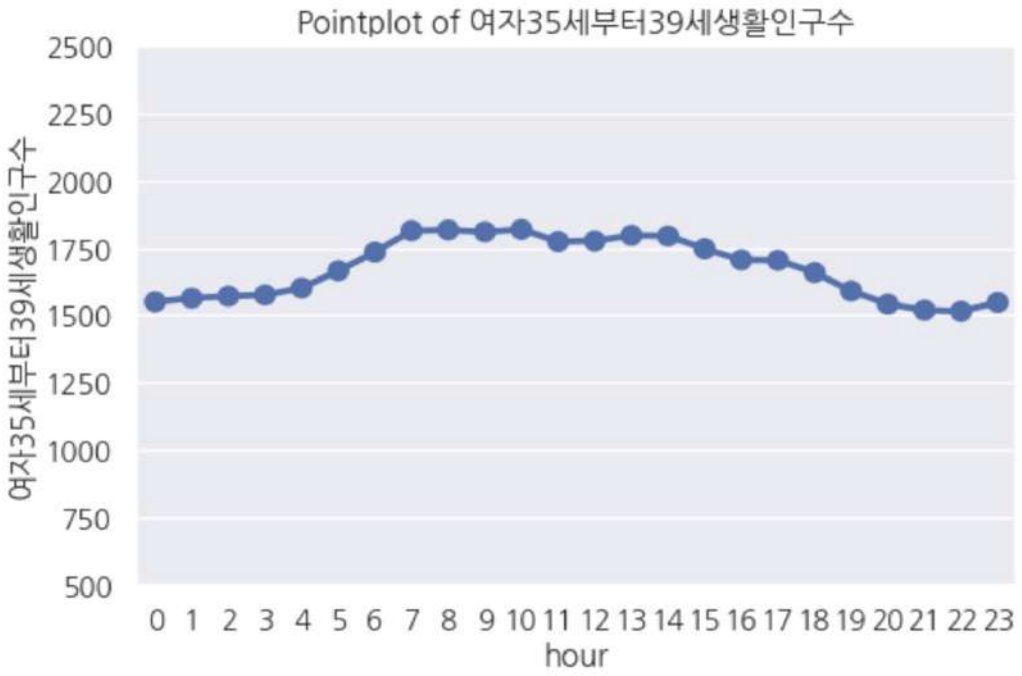


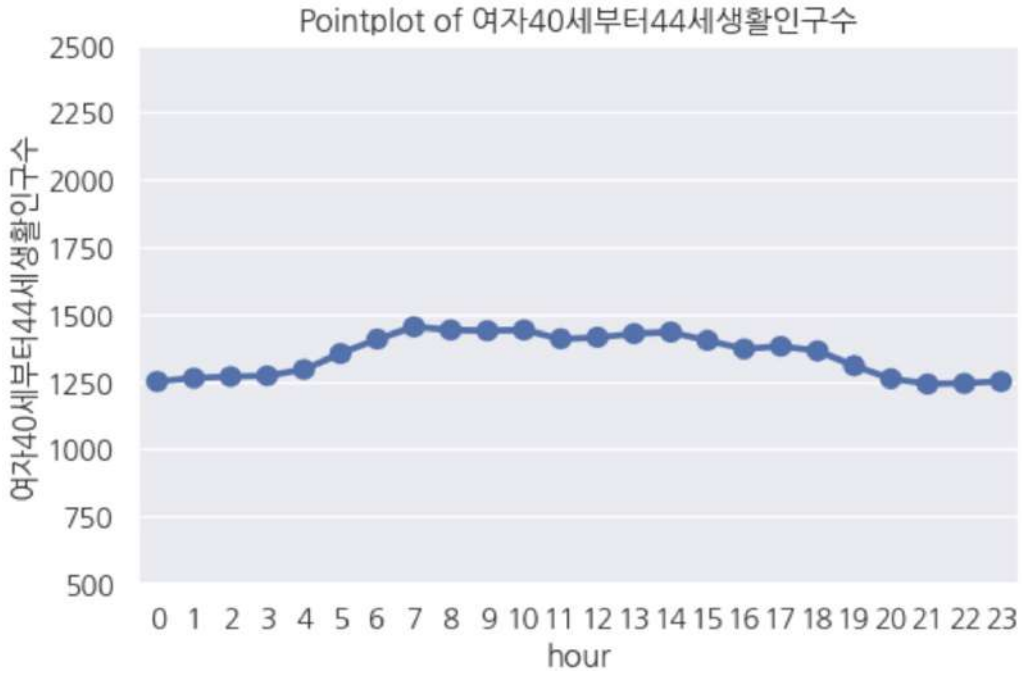
1. 데이터 불러오기 및 분석



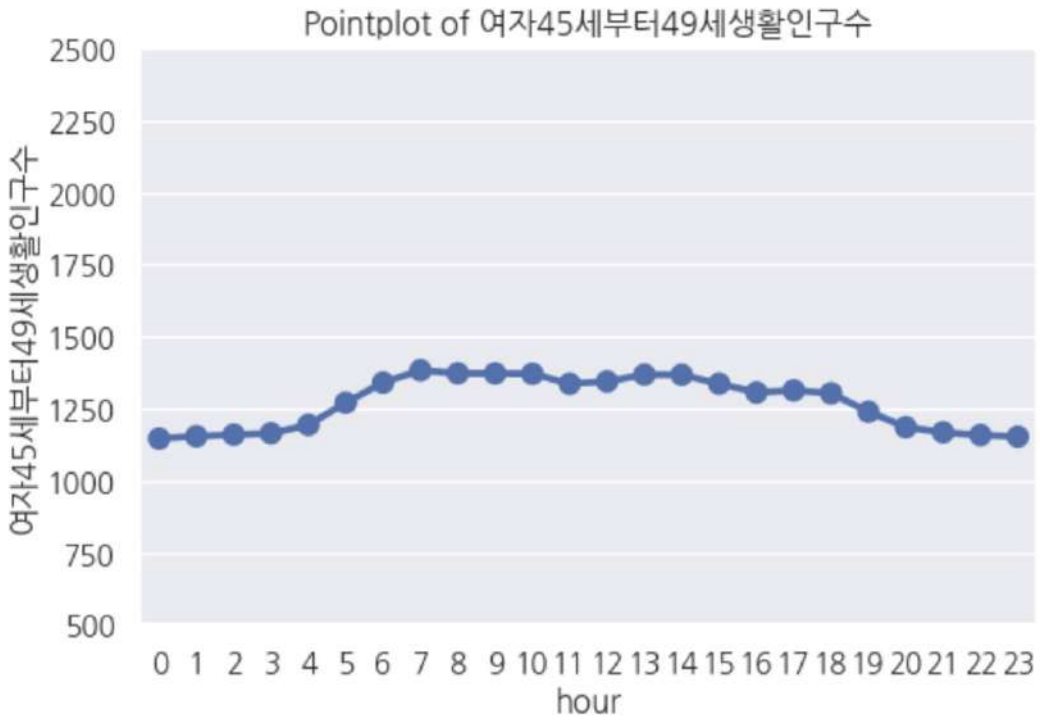


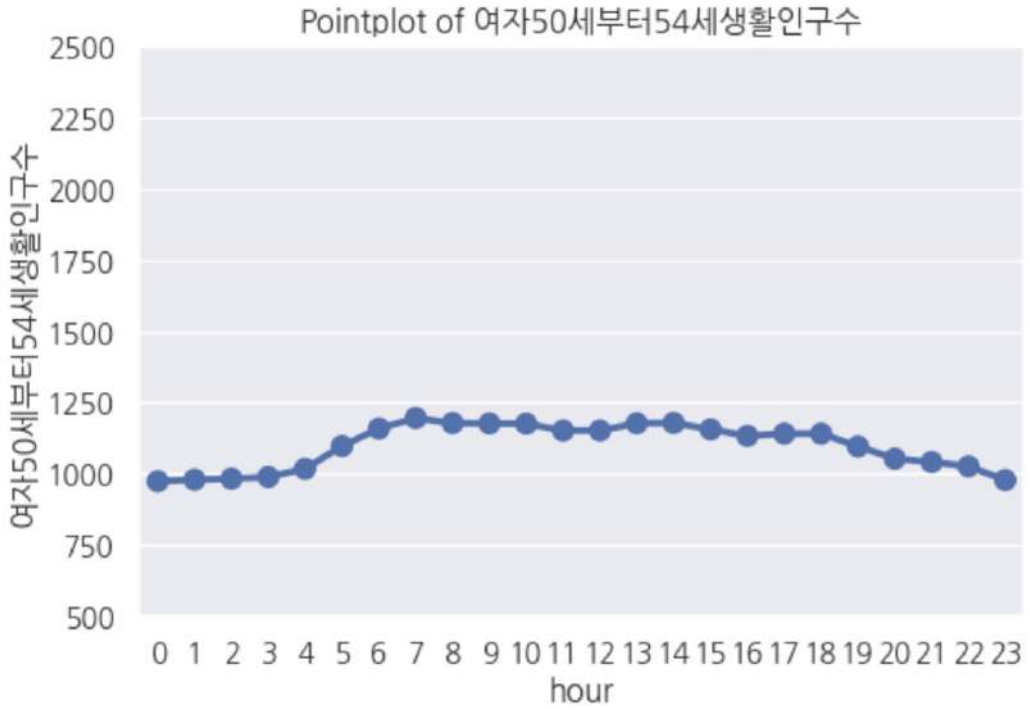
1. 데이터 불러오기 및 분석



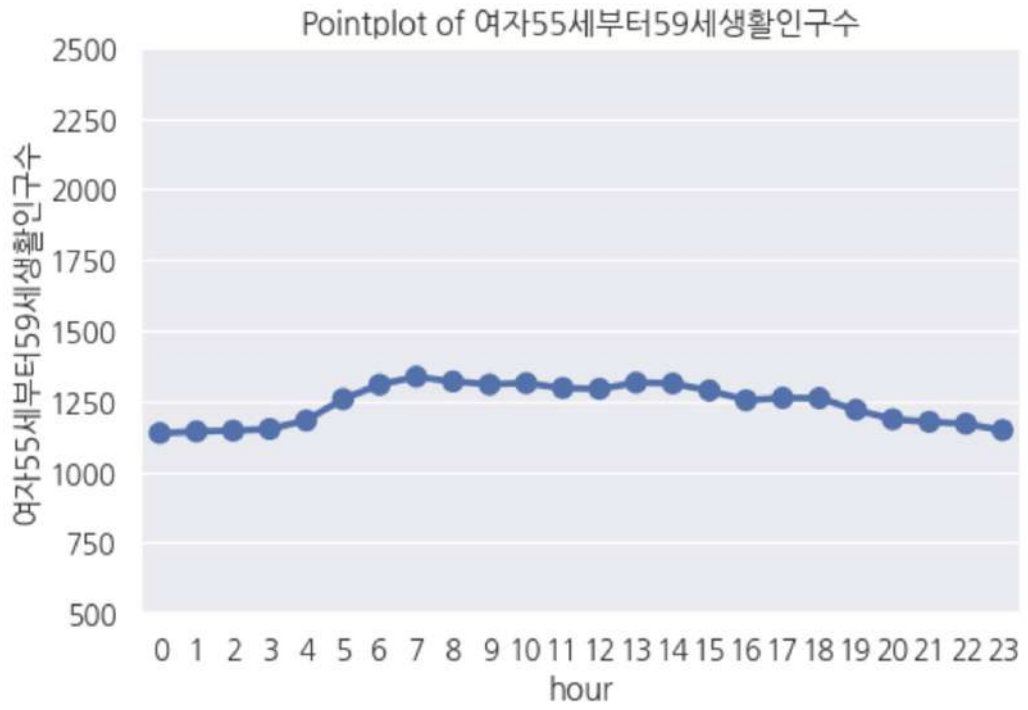


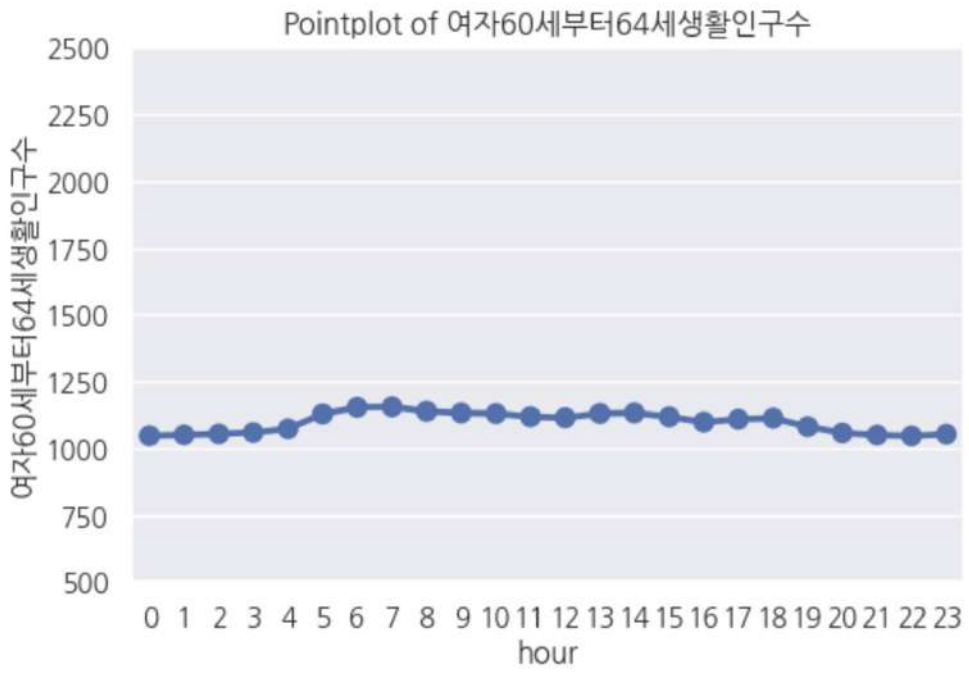
1. 데이터 불러오기 및 분석



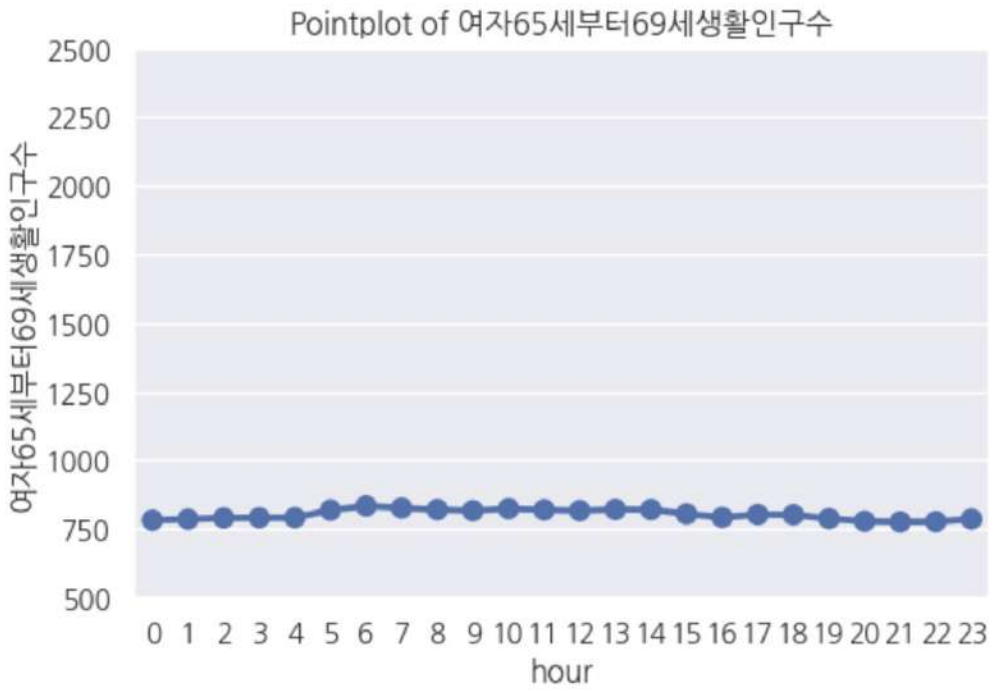


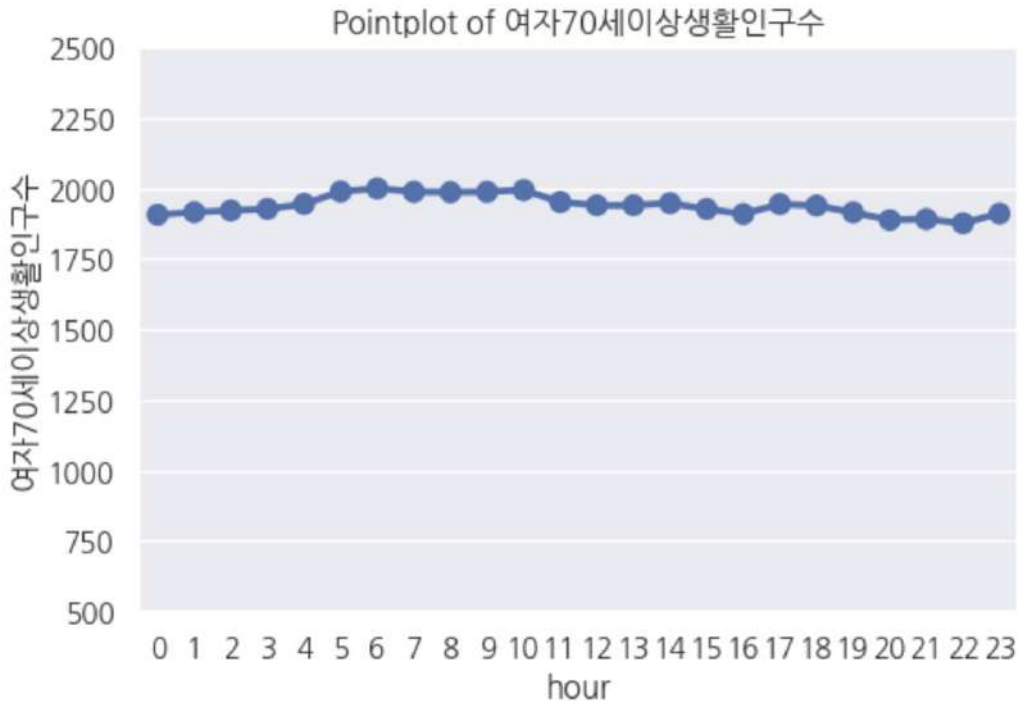
1. 데이터 불러오기 및 분석





1. 데이터 불러오기 및 분석





1. 데이터 불러오기 및 분석

- 남자0세부터9세생활인구수 ~ 여자70세이상생활인구수 변수 중에서
- 시간대에 따른 변화가 큰 변수를 '유동인구', 그렇지 않은 변수를 '비유동인구'로 정의
- '남성 유동인구' : 남자 20~ 24세 ~ 남자 55 ~ 59세
- '남성 비유동인구' : 나머지 남자 세대
- '여성 유동인구' : 여자 0세~ 9세, 여자 20세 ~ 24세 ~ 여자 55세 ~ 59세
- '여성 비유동인구' : 나머지 여자 세대

2. 데이터 전처리

- '유동인구', '비유동인구' 변수를 추가한다.
  - broadcasting 적용

```
1  df_total['유동인구'] = df_total['남자20세부터24세생활인구수'] + df_total['남자25세부터29세생활인구수'] + df_total['남자30세부터34
2  df_total['비유동인구'] = df_total['남자0세부터9세생활인구수'] + df_total['남자10세부터14세생활인구수'] + df_total['남자15세부터19
```

+ 코드 + 텍스트

```
1  df_test['유동인구'] = df_test['남자20세부터24세생활인구수'] + df_test['남자25세부터29세생활인구수'] + df_test['남자30세부터34세생활
2  df_test['비유동인구'] = df_test['남자0세부터9세생활인구수'] + df_test['남자10세부터14세생활인구수'] + df_test['남자15세부터19세생활
```

2. 데이터 전처리

주요 쟁점 : 어떤 변수를 추가할 것이며, Target이 무슨 변수인가?

- 하루 뒤(1일 후, 같은 시간대)를 예측할 것이다.
    - '하루 전 총생활인구수', '하루 전 유동인구', '하루 전 비유동인구' 변수를 추가한다.
    - '하루 뒤 총생활인구수' 변수를 추가한다. (Target)

2. 데이터 전처리

주요 쟁점 : 하루 전 같은 시간대의 '총생활인구수', '유동인구', '비유동인구' 변수를 추가한다.
다만 2017년 1월 1일 데이터는 하루 전 같은 시간대 데이터가 없으므로 NaN이 된다.
추후 전처리가 필요하다.

```
1  # 24시간 전의 데이터 가져오기
2  df_total['하루 전 총생활인구수']= df_total['총생활인구수'].shift(24)
3  df_total['하루 전 유동인구'] = df_total['유동인구'].shift(24)
4  df_total['하루 전 비유동인구'] = df_total['비유동인구'].shift(24)
```

```
[ ]  1  df_total.head(24)
```

	기준일ID	시간대구분	총생활인구수	유동인구	비유동인구	하루 전 총생활인구수	하루 전 유동인구	하루 전 비유동인구
0	2017-01-01 00:00:00	0	31535.2200	20275.1660	11260.0552	NaN	NaN	NaN
1	2017-01-01 01:00:00	1	31188.9174	20067.9000	11121.0182	NaN	NaN	NaN
2	2017-01-01 02:00:00	2	31240.4974	20183.7367	11056.7602	NaN	NaN	NaN
3	2017-01-01 03:00:00	3	31442.4314	20276.3294	11166.1016	NaN	NaN	NaN
4	2017-01-01 04:00:00	4	31922.7751	20736.3245	11186.4478	NaN	NaN	NaN
5	2017-01-01 05:00:00	5	33633.7304	21907.0407	11726.6897	NaN	NaN	NaN
6	2017-01-01 06:00:00	6	34876.8006	22854.9545	12021.8469	NaN	NaN	NaN
7	2017-01-01 07:00:00	7	35358.9775	23234.0442	12124.9306	NaN	NaN	NaN
8	2017-01-01 08:00:00	8	36038.7688	23537.1377	12501.6329	NaN	NaN	NaN
9	2017-01-01 09:00:00	9	37353.1794	24712.8368	12640.3409	NaN	NaN	NaN
10	2017-01-01 10:00:00	10	37534.7596	24543.8806	12990.8798	NaN	NaN	NaN
11	2017-01-01 11:00:00	11	38257.1671	25256.2066	13000.9663	NaN	NaN	NaN
12	2017-01-01 12:00:00	12	38423.5288	25495.8027	12927.7267	NaN	NaN	NaN
13	2017-01-01 13:00:00	13	37666.9073	25119.2627	12547.6464	NaN	NaN	NaN

2. 데이터 전처리

주요 쟁점 : 하루 전 같은 시간대의 '총생활인구수', '유동인구', '비유동인구' 변수를 추가한다.
다만 2022년 1월 1일 데이터는 하루 전 같은 시간대 데이터가 없으므로 NaN이 된다.
추후 전처리가 필요하다.

```
1  # 24시간 전의 데이터 가져오기
2  df_test['하루 전 총생활인구수']= df_test['총생활인구수'].shift(24)
3  df_test['하루 전 유동인구'] = df_test['유동인구'].shift(24)
4  df_test['하루 전 비유동인구'] = df_test['비유동인구'].shift(24)
```

```
[ ]  1  df_test.head(5)
```

	기준일ID	시간대구분	총생활인구수	유동인구	비유동인구	하루 전 총생활인구수	하루 전 유동인구	하루 전 비유동인구
0	2022-01-01 00:00:00	0	30509.7386	20263.2335	10246.5077	NaN	NaN	NaN
1	2022-01-01 01:00:00	1	30759.5067	20451.1147	10308.3933	NaN	NaN	NaN
2	2022-01-01 02:00:00	2	31048.8787	20479.7378	10569.1413	NaN	NaN	NaN
3	2022-01-01 03:00:00	3	31076.1092	20654.0596	10422.0490	NaN	NaN	NaN
4	2022-01-01 04:00:00	4	31714.9309	21006.0597	10708.8706	NaN	NaN	NaN

2. 데이터 전처리

주요 쟁점 : 하루 뒤 같은 시간대의 '총생활인구수', '유동인구', '비유동인구' 변수를 추가한다.
다만 2021년 12월 31일 데이터는 하루 뒤 같은 시간대 데이터가 없으므로 NaN이 된다.
추후 전처리가 필요하다.

```
# 하루 뒤(1일 후, 같은 시간대) 총생활인구수를 가져온다. (Target이 될 예정)
df_total['하루 뒤 총생활인구수'] = df_total['총생활인구수'].shift(-24)
```

2. 데이터 전처리

주요 쟁점 : 하루 뒤 같은 시간대의 '총생활인구수', '유동인구', '비유동인구' 변수를 추가한다.
다만 2022년 6월 30일 데이터는 하루 뒤 같은 시간대 데이터가 없으므로 NaN이 된다.
추후 전처리가 필요하다.

```
# 하루 뒤(1일 후, 같은 시간대) 총생활인구수를 가져온다. (Target이 될 예정)
df_test['하루 뒤 총생활인구수'] = df_test['총생활인구수'].shift(-24)
```

2. 데이터 전처리

- train) 현재 2017년 1월 1일 데이터는 '하루 전 총생활인구수', '하루 전 유동인구', '하루 전 비유동인구'가 NaN이다.
  결국 2016년 12월 31일 데이터가 필요한데 이게 무슨 동의 데이터인지 모르는 이상 이를 구할 수 있는 방법이 일단은 없다. 그래서 그냥 2017년 1월 1일 데이터는 삭제하려고 한다.

```
# '기준일ID' 열이 '2017-01-01'인 행을 필터링하여 데이터프레임 업데이트
df_total = df_total.iloc[24:]
```

2. 데이터 전처리

- test) 현재 2022년 1월 1일 데이터는 '하루 전 총생활인구수', '하루 전 유동인구', '하루 전 비유동인구'가 NaN이다.
  우리는 2021년 12월 31일 데이터를 찾을 수 있으므로 그 점을 이용해서 NaN을 채운다.

```
[ ]  1  # 2021년 12월 31일 '총생활인구수', '유동인구', '비유동인구' 데이터를 가져옴
     2  tt = df_total[['총생활인구수', '유동인구', '비유동인구']].tail(24)
```

```
[ ]  1  # NaN 값을 채우기 위한 인덱스 범위 설정
     2  start_idx = df_test.loc[df_test['하루 전 총생활인구수'].isna()].index[0]
     3  end_idx = df_test.loc[df_test['하루 전 총생활인구수'].isna()].index[-1] + 1
     4
     5  # '하루 전 총생활인구수' 채우기
     6  df_test.iloc[start_idx:end_idx:1, df_test.columns.get_loc('하루 전 총생활인구수')] = tt['총생활인구수'].values
     7
     8  # '하루 전 유동인구' 채우기
     9  df_test.iloc[start_idx:end_idx:1, df_test.columns.get_loc('하루 전 유동인구')] = tt['유동인구'].values
    10
    11  #'하루 전 비유동인구' 채우기
    12  df_test.iloc[start_idx:end_idx:1, df_test.columns.get_loc('하루 전 비유동인구')] = tt['비유동인구'].values
```

2. 데이터 전처리

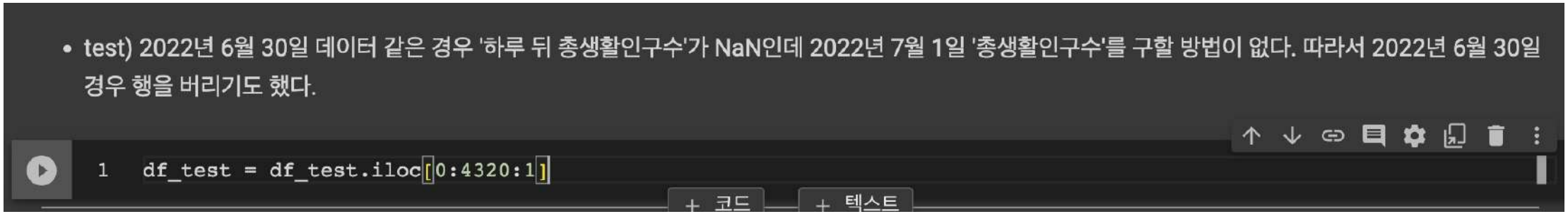
- train) 2021년 12월 31일 '하루 뒤 총생활인구수'(Target) 결측치를 채운다.
  - 2022년 1월 1일 '총생활인구수' 데이터가 있으니 그것을 이용해서 NaN를 대체한다.

```
1  # 2022년 1월 1일 '총생활인구수' 데이터를 가져옴
2  gg = df_test['총생활인구수'].values[0:24:1]
3  gg
```

```
array([30509.7386, 30759.5067, 31048.8787, 31076.1092, 31714.9309,
       33368.3456, 34962.7828, 34948.9644, 34311.2156, 33966.982 ,
       34039.4022, 33570.1052, 33307.6517, 33125.8385, 32584.0065,
       31298.8122, 30426.6285, 30445.2773, 30258.8998, 30071.1298,
       30140.0597, 30183.2726, 30694.2288, 31496.5201])
```

```
1  # 데이터프레임의 '하루 뒤 총생활인구수' 열의 마지막 24개 행에 new_values 값을 할당
2  df_total['하루 뒤 총생활인구수'].iloc[-24:] = gg
```

2. 데이터 전처리


• test) 2022년 6월 30일 데이터 같은 경우 '하루 뒤 총생활인구수'가 NaN인데 2022년 7월 1일 '총생활인구수'를 구할 방법이 없다. 따라서 2022년 6월 30일 경우 행을 버리기도 했다.
1 df_test = df_test.iloc[0:4320:1]
+ 코드
+ 텍스트

2. 데이터 전처리

주요 쟁점 : 상관 계수를 확인한다.

2. 데이터 전처리

주요 쟁점 : 다중 공선성을 확인한다. 다중 공선성 문제가 발생하는 것으로 확인할 수 있다.
하지만 어떤 변수를 만든간에 다중 공선성 문제를 피해갈 수 없었다.

```python
# 다중 공선성 체크
from statsmodels.stats.outliers_influence import variance_inflation_factor

vif_total  = pd.DataFrame()
vif_total['Vif Factor'] = [variance_inflation_factor(train_x.values, i) for i in range(train_x.shape[1])]
vif_total['features'] = train_x.columns
vif_total
```

	Vif Factor	features
0	6.004800e+14	총생활인구수
1	2.905548e+14	유동인구
2	6.045100e+13	비유동인구
3	6.004800e+14	하루 전 총생활인구수
4	2.905548e+14	하루 전 유동인구
5	6.045100e+13	하루 전 비유동인구

- 이번 프로젝트는 어떤 변수를 사용하던간에 다중 공선성을 피해갈 수가 없다.

3. 모델링

Linear Regression – 0.72

```
[ ]  1  # 아래에 실습코드를 작성하세요.
     2  model1 = LinearRegression()
     3  model1.fit(train_x, train_y)
```

```
▾ LinearRegression
LinearRegression()
```

```
[ ]  1  print('훈련 세트 점수 : ', model1.score(train_x, train_y))
     2  print('평가 세트 점수 : ', model1.score(test_x, test_y))
```

```
훈련 세트 점수 :  0.7918200253355155
평가 세트 점수 :  0.7200775423986912
```

```
[ ]  1  pred_y = model1.predict(test_x)
```

```
[ ]  1  print('RMSE : ', mean_squared_error(test_y, pred_y) ** 0.5) # RMSE는 오차로써 작을수록 좋다.
     2  print('r2 score : ', r2_score(test_y, pred_y))
```

```
RMSE :  1877.142923630923
r2 score :  0.7200775423986912
```

3. 모델링

Random Forest(n_estimators=100, max_depth=4) – 0.74

```
[8]  1  model2 = RandomForestRegressor(n_estimators=100,  max_depth=4, n_jobs=-1)
     2  model2.fit(train_x, train_y)
```

```
<ipython-input-8-eeb1b6229638>:2: DataConversionWarning: A column-vector y was passed when a 1d array was expected. Pleas
  model2.fit(train_x, train_y)
```

RandomForestRegressor

RandomForestRegressor(max_depth=4, n_jobs=-1)

```
1  print('훈련 세트 점수 : ', model2.score(train_x, train_y))
2  print('평가 세트 점수 : ', model2.score(test_x, test_y))
```

```
훈련 세트 점수 :  0.7965539558590361
평가 세트 점수 :  0.7408953292562348
```

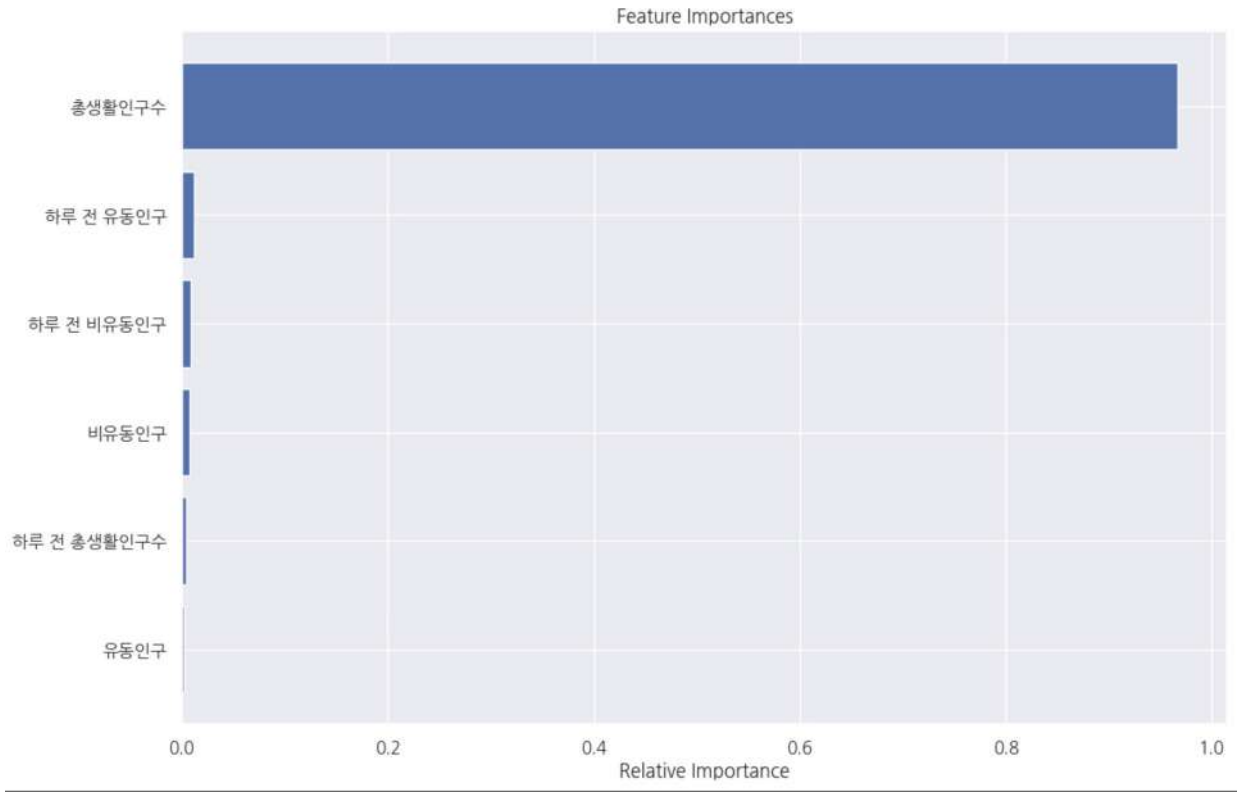
```
[10]  1  pred_y = model2.predict(test_x)
```

```
1  print('RMSE : ', mean_squared_error(test_y, pred_y) ** 0.5) # RMSE는 오차로써 작을수록 좋다.
2  print('r2 score : ', r2_score(test_y, pred_y))
```

```
RMSE :  1805.9931149818076
r2 score :  0.7408953292562346
```

3. 모델링

Random Forest Feature Importances



3. 모델링

GradientBoosting(n_estimators=500, max_depth=4, learning_rate=0.01) – 약 0.75

```
[ ]  1  # 아래에 실습코드를 작성하세요.
     2  model3 = GradientBoostingRegressor(n_estimators=500, max_depth=4, learning_rate=0.01, random_state=42)
     3  model3.fit(train_x, train_y)
```

```
/usr/local/lib/python3.10/dist-packages/sklearn/ensemble/_gb.py:437: DataConversionWarning: A column-vector y was passed
  y = column_or_1d(y, warn=True)
```

```
GradientBoostingRegressor(learning_rate=0.01, max_depth=4, n_estimators=500,
                          random_state=42)
```

```
     1  # 아래에 실습코드를 작성하세요.
     2  print('훈련 세트 점수 : ', model3.score(train_x, train_y))
     3  print('평가 세트 점수 : ', model3.score(test_x, test_y))
```

```
훈련 세트 점수 :  0.8153847089937892
평가 세트 점수 :  0.7491824209904054
```

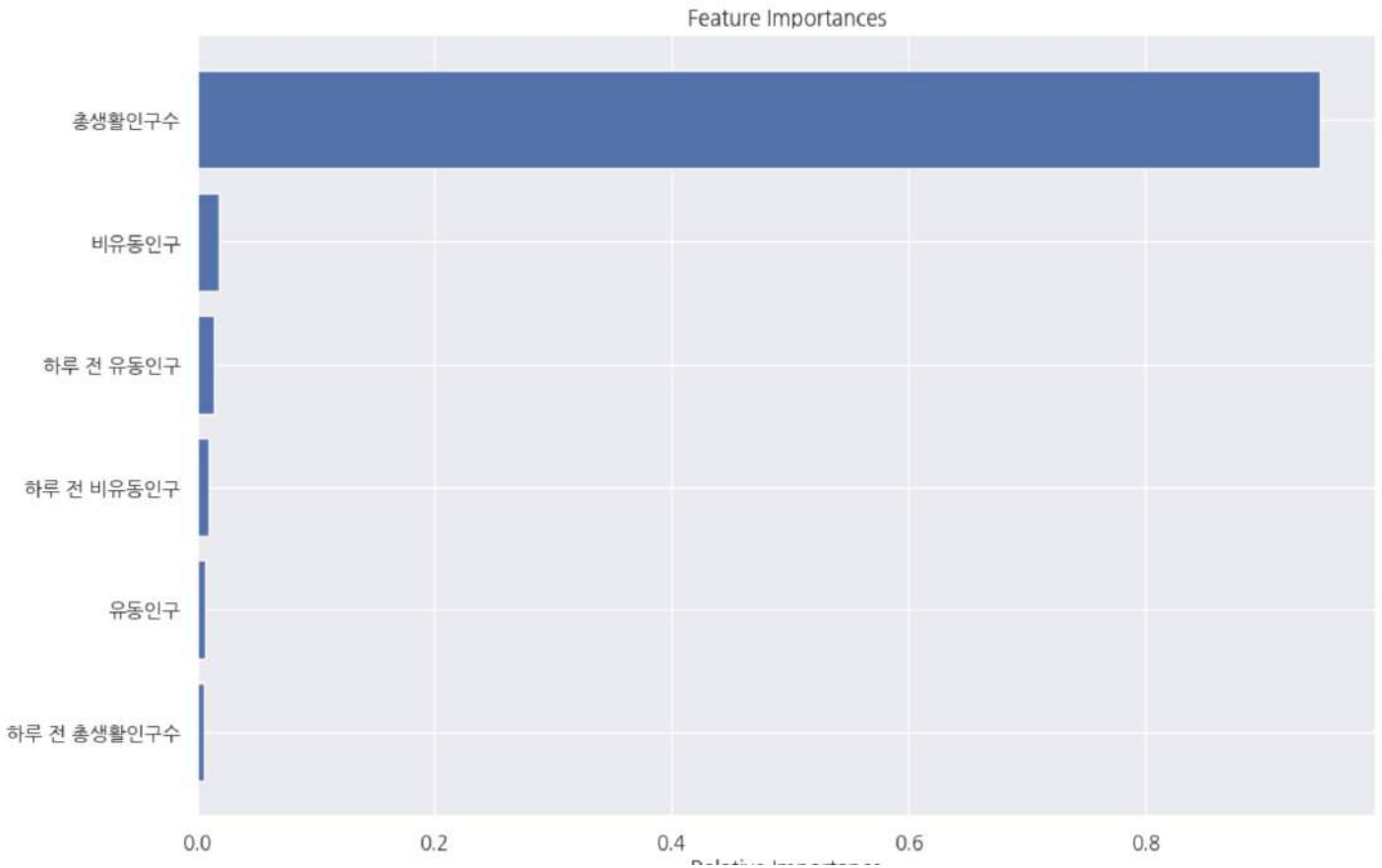
```
[ ]  1  pred_y = model3.predict(test_x)
```

```
[ ]  1  # 아래에 실습코드를 작성하세요.
     2  print('RMSE : ', mean_squared_error(test_y, pred_y) ** 0.5) # RMSE는 오차로써 작을수록 좋다.
     3  print('r2 score : ', r2_score(test_y, pred_y))
```

```
RMSE :  1776.8773653474057
r2 score :  0.7491824209904054
```

3. 모델링

GradientBoosting Feature Importances



3. 모델링

XGBoost(n_estimators=200, learning_rate=0.1, max_depth=5) – 0.74

```
1  model5 = XGBRegressor(n_estimators=200, learning_rate=0.1, max_depth=5, )
2  model5.fit(train_x, train_y)
```

```
XGBRegressor(base_score=None, booster=None, callbacks=None,
             colsample_bylevel=None, colsample_bynode=None,
             colsample_bytree=None, device=None, early_stopping_rounds=None,
             enable_categorical=False, eval_metric=None, feature_types=None,
             gamma=None, grow_policy=None, importance_type=None,
             interaction_constraints=None, learning_rate=0.1, max_bin=None,
             max_cat_threshold=None, max_cat_to_onehot=None,
             max_delta_step=None, max_depth=5, max_leaves=None,
             min_child_weight=None, missing=nan, monotone_constraints=None,
             multi_strategy=None, n_estimators=200, n_jobs=None,
             num_parallel_tree=None, random_state=None, ...)
```

```
[ ]  1  # 아래에 실습코드를 작성하세요.
     2  print('훈련 세트 점수 : ', model5.score(train_x, train_y))
     3  print('평가 세트 점수 : ', model5.score(test_x, test_y))
```

```
훈련 세트 점수 :  0.8393539661683178
평가 세트 점수 :  0.7455155437139623
```

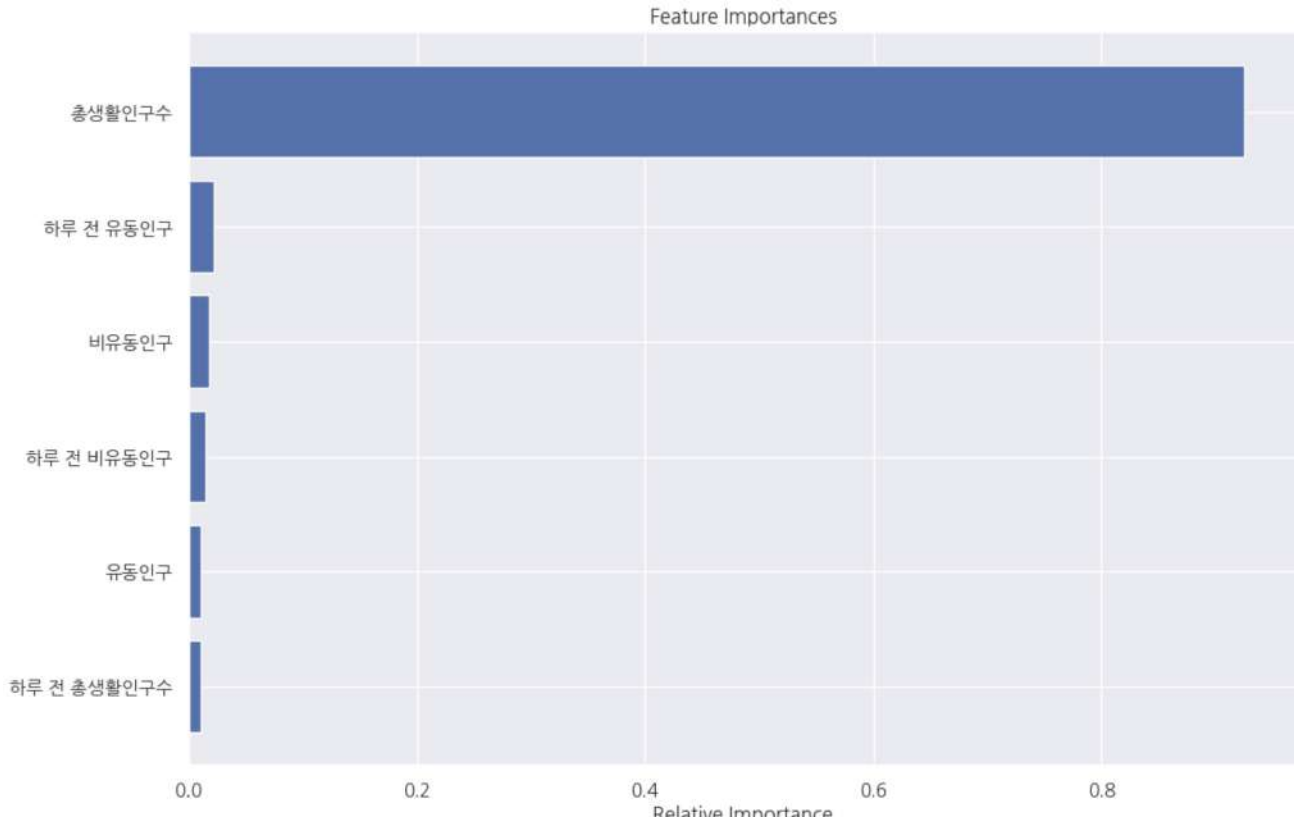
```
[ ]  1  pred_y = model5.predict(test_x)
```

```
[ ]  1  # 아래에 실습코드를 작성하세요.
     2  print('RMSE : ', mean_squared_error(test_y, pred_y) ** 0.5) # RMSE는 오차로써 작을수록 좋다.
     3  print('r2 score : ', r2_score(test_y, pred_y))
```

```
RMSE :  1789.8189417865697
r2 score :  0.7455155437139623
```

3. 모델링

XGBoost(n_estimators=200, learning_rate=0.1, max_depth=5) – Feature Impotrances



3. 모델링

GridSearch를 적용한 XGBoost – 0.75 (약 3시간 40분 소요)

• 그리드 서치

```
params = {
    'max_depth' : range(1, 31, 1),
    'n_estimators' : [50, 100, 150, 200, 250, 300],
    'learning_rate' : [0.05, 0.1, 0.2],
}
```

```
grid_xgboost = GridSearchCV(XGBRegressor(), params, cv = 5, n_jobs = -1, verbose=True)

grid_xgboost.fit(train_x, train_y)
```

```
Fitting 5 folds for each of 540 candidates, totalling 2700 fits
/usr/local/lib/python3.10/dist-packages/joblib/externals/loky/process_executor.py:752: UserWarni
  warnings.warn(
```

GridSearchCV
estimator: XGBRegressor
XGBRegressor

```
grid_xgboost.best_params_
```

```
{'learning_rate': 0.05, 'max_depth': 4, 'n_estimators': 100}
```

```
# 아래에 실습코드를 작성하세요.
print('훈련 세트 점수 : ', grid_xgboost.score(train_x, train_y))
print('평가 세트 점수 : ', grid_xgboost.score(test_x, test_y))
```

```
훈련 세트 점수 :  0.8145117325691043
평가 세트 점수 :  0.7503181055669121
```

```
pred_y = grid_xgboost.predict(test_x)
```

```
# 아래에 실습코드를 작성하세요.
print('RMSE : ', mean_squared_error(test_y, pred_y) ** 0.5) # RMSE는 오차로써 작을수록 좋다.
print('r2 score : ', r2_score(test_y, pred_y))
```

```
RMSE :  1772.850012635655
r2 score :  0.7503181055669121
```

3. 모델링

인공신경망 – 0.73

```
[ ]
# 아래에 실습코드를 작성하세요.
model6 = Sequential()

model6.add(Dense(256, activation='swish', input_shape=(train_scaled_x.shape[1],), name='hidden1'))

model6.add(Dense(128, activation='swish', name='hidden2'))

model6.add(Dense(64, activation='swish', name='hidden3'))

model6.add(Dense(1)) # 출력층

model6.compile(optimizer=Adam(learning_rate=0.1), loss='mse', metrics=[root_mean_squared_error, r_squared]) # metri
```

3. 모델링

인공신경망 – 0.73

```
early_stopping = EarlyStopping(monitor='val_loss', patience=8,  restore_best_weights=True) # 과대 적합을 막기 위함
history = model6.fit(train_scaled_x, train_y, epochs=15, batch_size=128, validation_split=0.2, callbacks=early_stop
```

```
r: 2022.5896 - r_squared: 0.7785 - val_loss: 3558343.5000 - val_root_mean_squared_error: 1862.4211 - val_r_squared: 0.3421
r: 2042.5071 - r_squared: 0.7730 - val_loss: 2135236.2500 - val_root_mean_squared_error: 1434.7898 - val_r_squared: 0.6157
r: 2049.5894 - r_squared: 0.7722 - val_loss: 2050446.3750 - val_root_mean_squared_error: 1407.9730 - val_r_squared: 0.6307
r: 2027.9370 - r_squared: 0.7768 - val_loss: 2074631.5000 - val_root_mean_squared_error: 1422.3781 - val_r_squared: 0.6272
r: 2017.2329 - r_squared: 0.7785 - val_loss: 2566168.2500 - val_root_mean_squared_error: 1573.6752 - val_r_squared: 0.5328
r: 2030.3892 - r_squared: 0.7761 - val_loss: 1952112.3750 - val_root_mean_squared_error: 1373.3375 - val_r_squared: 0.6518
r: 2020.4708 - r_squared: 0.7780 - val_loss: 2655650.2500 - val_root_mean_squared_error: 1605.1746 - val_r_squared: 0.5072
r: 2017.1849 - r_squared: 0.7789 - val_loss: 2497151.2500 - val_root_mean_squared_error: 1554.3776 - val_r_squared: 0.5449
r: 2038.8915 - r_squared: 0.7745 - val_loss: 2080370.3750 - val_root_mean_squared_error: 1418.8346 - val_r_squared: 0.6279
r: 2014.5265 - r_squared: 0.7794 - val_loss: 2234599.0000 - val_root_mean_squared_error: 1469.0927 - val_r_squared: 0.5969
r: 2033.3593 - r_squared: 0.7749 - val_loss: 2200898.5000 - val_root_mean_squared_error: 1458.7048 - val_r_squared: 0.6027
r: 2041.6562 - r_squared: 0.7735 - val_loss: 2788442.7500 - val_root_mean_squared_error: 1646.8137 - val_r_squared: 0.4947
r: 2013.1743 - r_squared: 0.7805 - val_loss: 2713198.0000 - val_root_mean_squared_error: 1621.8699 - val_r_squared: 0.5085
r: 2040.8942 - r_squared: 0.7737 - val_loss: 2892901.0000 - val_root_mean_squared_error: 1675.5431 - val_r_squared: 0.4694
```

3. 모델링

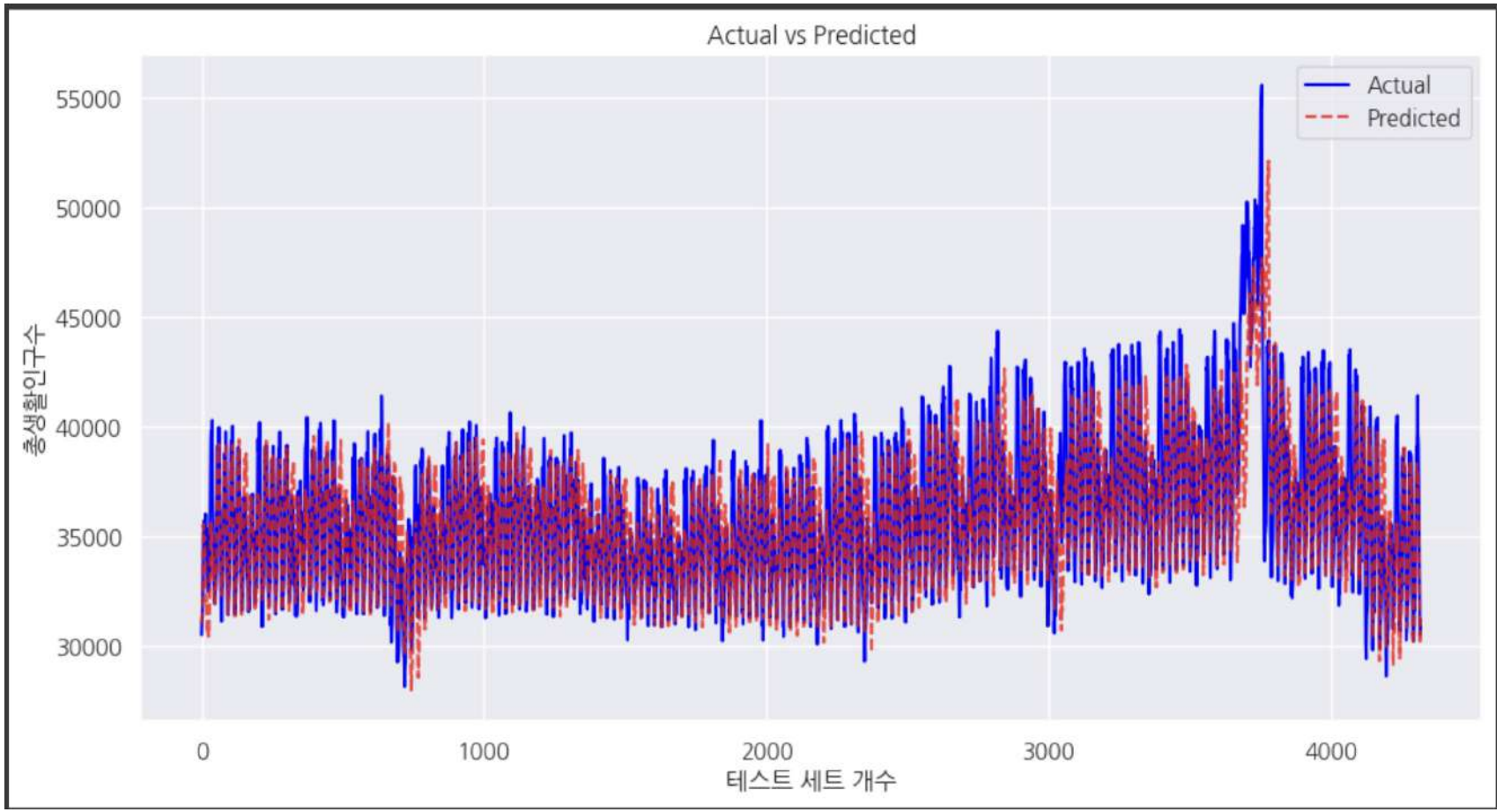
인공신경망 – 0.73

```
[ ]  1  # 아래에 실습코드를 작성하세요.
     2  predictions = model6.predict(test_scaled_x)
     3
     4  print('r2 : ', r2_score(test_y, predictions))
```

```
135/135 [==============================] - 0s 1ms/step
r2 :  0.7385214993886177
```

3. 모델링

인공신경망을 통한 예측값과 실제값 비교



4. 느낀점

느낀점

- 역시 모델은 만든다는 것은 불확실성의 영역인 것 같다. 우리가 중요하다고 생각했던 변수를 이용해서 무조건 모델의 성능을 향상시킨다고 보장하는 것도 아니며, 우리가 중요하지 않다고 생각하는 변수를 버림으로써 무조건 모델의 성능을 향상시킨다고 보장하는 것도 아니다.
- 즉 모델의 성능을 향상시키기 위해 어떤 길을 가더라도 불확실한 그런 찜찜한 마음을 한 켠에 느낀다.(개인적으로)
  진짜 일일히 모든 경우의 수를 찾아 노가다가 이루어질 수 있구나 하는 생각이 들었다.
- 이 프로젝트에서는 어떤 변수를 갖고 해도 결국은 다중 공선성 문제를 피해갈 수 없었다.
- 대체로 머신러닝, 딥러닝 모두 성능이 좋은편은 아니겠지만 일반적으로 추세를 잘 따라간다고 할 수 있다.