B.D.A

Easy Study Project

[Data-Based Bus Arrival Time Prediction]

팀명 : 3조
팀원 : 김예리, 이상준, 정제나

B.D.A

CONTENTS

01. 프로젝트 소개

- 프로젝트 주제
- 주제 선정 이유

01. 프로젝트 소개

< 프로젝트 주제 >

버스 노선별 데이터 기반 버스 운행 시간 예측

< 주제 선정 이유 >

버스 운행 시간을 실시간으로 예측하여 버스 운행 시스템 상용화

02. 프로젝트 과정

- 프로젝트 원리
- 프로젝트 코드

B.D.A

02. 프로젝트 과정

1. Feature engineering
2. Data Analyze
3. Preprocessing 1 (Perspective of ‘s/m’)
4. Preprocessing 2 (Perspective of ‘m/s’)
5. Final preprocessing
6. Data modeling

1. Feature engineering

< 불필요한 컬럼 삭제 >
- 위도, 경도 필요하지 않다고 판단하여 삭제

```
1 # 불필요한 컬럼 삭제
2 columns_to_drop = [ 'now_latitude', 'now_longitude','next_latitude','next_longitude']
3 df = df.drop(columns=columns_to_drop)
```

< 버스 운행 특성 >
- 평일과 주말의 배차 간격은 다를 것
- 주로, 출퇴근 시간에 막힐 것

B.D.A

1. Feature engineering

< 새로운 컬럼 생성 >

- 1m당 걸린 시간(s/m), 1초당 걸린 거리(m/s) 컬럼 생성 : 버스의 지연 정도를 파악하고자 함
- 요일(week), 시간(아침,점심,저녁) 컬럼 생성 : 위의 특성을 이용한 분석을 하고자 함

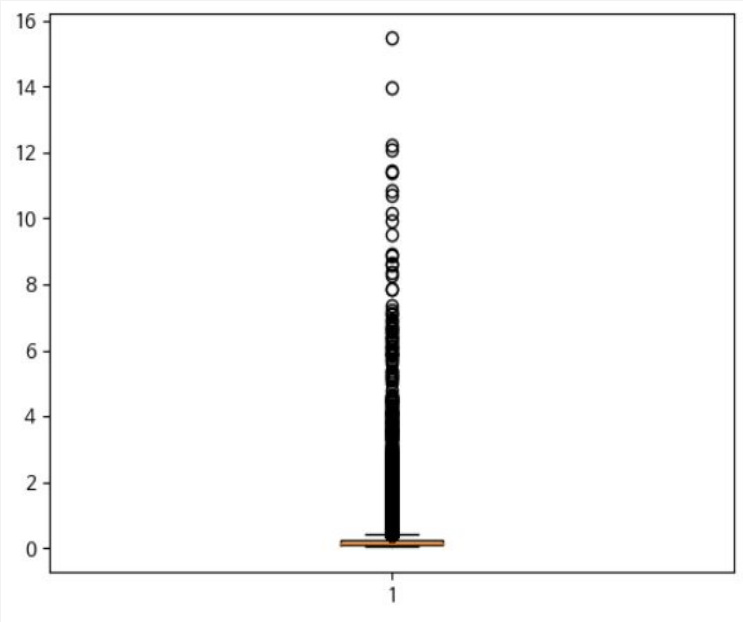
```
# 필요한 컬럼들 생성 (요일, 1미터당 걸린 시간(초), 1초당 걸린 거리(m) 계산하여 컬럼화)

df['date'] = pd.to_datetime(df['date'])
df['week'] = df['date'].dt.day_name()

df['distance'] = df['distance'].astype(int)
df['s/m'] = df['next_arrive_time'] / df['distance']
df['m/s'] = df['distance'] / df['next_arrive_time']
df['time']='time'
```

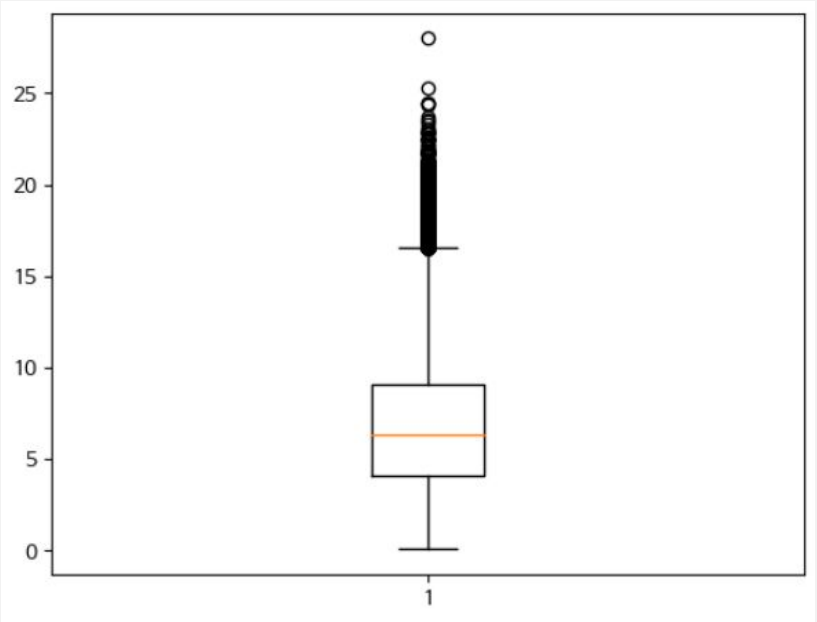
```
df.loc[ (df['now_arrive_time'] >='05시') & (df['now_arrive_time'] <'12시') ,['time'] ]= 'morning' # 05~11시
df.loc[ (df['now_arrive_time'] >='12시') & (df['now_arrive_time'] <'18시') ,['time'] ]= 'afternoon' # 12~17시
df.loc[ (df['now_arrive_time'] >='18시') | (df['now_arrive_time'] =='00시') ,['time'] ]= 'evening' # 18~0시
df
```

1. Feature engineering

< 이상치 기준 정하기 >

- 1m당 걸린 시간(s/m), 1초당 걸린 거리(m/s) 의 박스플롯 분석 : 이상치가 위로만 분포





1. Feature engineering

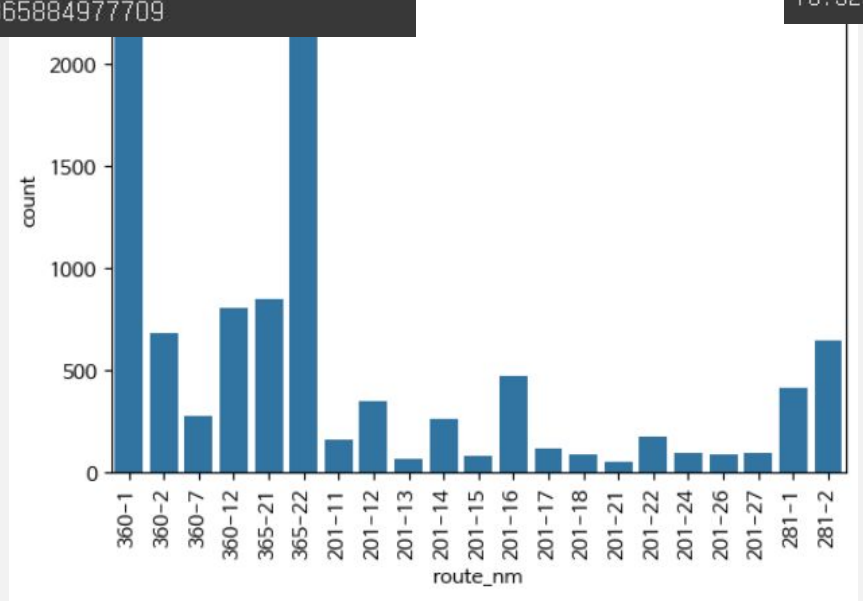
< 이상치 분석 >

- IQR 로 75% 이상의 s/m와 m/s 분석

```
2 q3 = df['s/m'].quantile(0.75)
3 q1 = df['s/m'].quantile(0.25)
4 q =  (q3-q1)*1.5+q3
```

```
1 q
```

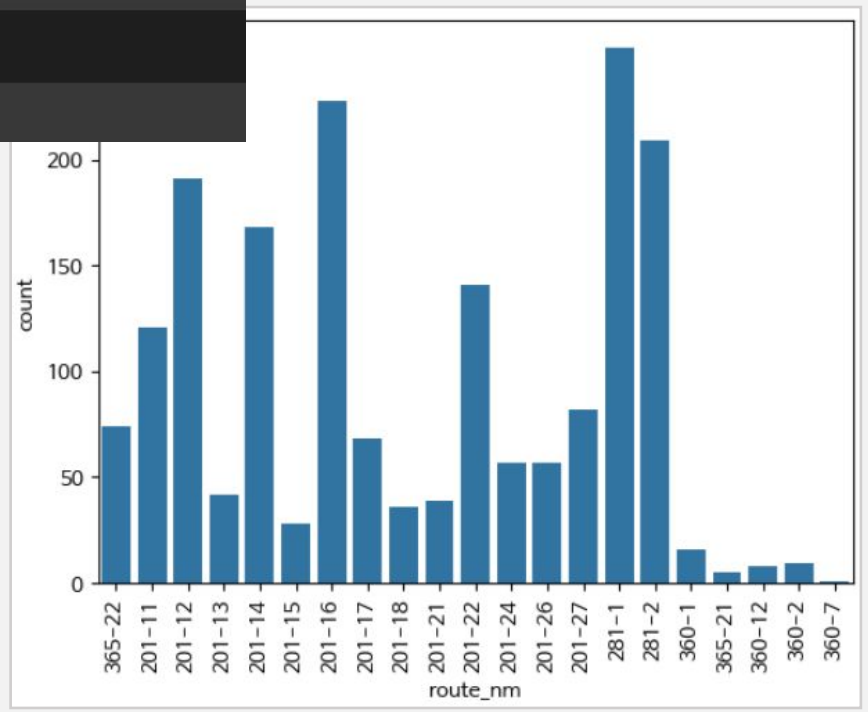
```
0.4406365884977709
```




```
1 q3 = df['m/s'].quantile(0.75)
2 q1 = df['m/s'].quantile(0.25)
3 q = (q3-q1)*1.5 +q3
```

```
1 q
```

```
16.523934659090911
```



1. Feature engineering

< 이상치 분석 결과 : s/m 중심 이상치 처리방식 채택 >
- 이상치 분포 : 전 노선에 존재
- m/s의 이상치 처리 방식 : 이상치의 양이 매우 작아, 모두 삭제
- s/m의 이상치 처리 방식 : 노선별 이상치 count가 전체 count의 75% 이상인 것으로 정의하여, 이상치 처리

```
1 outlier_count['count'].describe()
```

```
count      21.000000
mean      505.190476
std       686.135382
min        51.000000
25%        95.000000
50%       264.000000
75%       645.000000
max      2554.000000
Name: count, dtype: float64
```

```
1 outlier_count[outlier_count['count']>=645]
```

	route_nm	count
14	281-2	645
15	360-1	2554
16	360-12	808
17	360-2	683
19	365-21	851
20	365-22	2284

2. Data Analyze

< 이상치 분석>

이상치 대체 : 주요 노선 (281-2, 360-1, 360-12, 360-2, 365-21, 365-22)에 대해선 이상치 대체 및 삭제
이상치 삭제 : 이외의 노선의 모든 이상치는 삭제

```
idx = df[((~df['route_nm'].isin(['281-2', '360-1', '360-12', '360-2', '365-21', '365-22'])) & (df['s/m'] >= q))].index
df.drop(idx, inplace =True)
```

B.D.A

2.Data Analyze

< 이상치 분석 >

- outlier_df : 주요 6개 노선의 이상치들을 데이터프레임화
- outlier_df_count : 주요 6개 노선의 정류장별 이상치들의 카운트값

```
4 outlier_df = df[(df['route_nm'].isin([ '281-2', '360-1', '360-12', '360-2', '365-21', '365-22'])) & (df['s/m']>q)]
```

```
1 outlier_count = outlier_df.groupby(['now_station','next_station'])['now_station'].count().reset_index(name = 'count')
2 outlier_count
```

- outlier_df_count의 75% 이상의 값 : 이상치 대체
- outlier_df_count의 75% 미만의 값 : 이상치 삭제

B.D.A

2. Data Analyze

< 이상치 분석 >

- 이상치의 요일별, 시간대별 분석을 위한 임의의 데이터프레임 생성

```
# outlier_df 중 노선 6개를 각각 뽑아 데이터프레임화 (변수명 : df노선번호)

df281_2 = outlier_df[outlier_df['route_nm'] == '281-2']
df360_1 = outlier_df[outlier_df['route_nm'] == '360-1']
df360_2 = outlier_df[outlier_df['route_nm'] == '360-2']
df360_12 = outlier_df[outlier_df['route_nm'] == '360-12']
df365_21 = outlier_df[outlier_df['route_nm'] == '365-21']
df365_22 = outlier_df[outlier_df['route_nm'] == '365-22']
```

```
#  6개 노선의 데이터프레임의 정거장별 카운트값 데이터프레임화 (변수명 : count_노선번호)

count_281_2 = df281_2.groupby(['now_station', 'next_station'])['route_nm'].count().reset_index(name ='count')
count_360_1 = df360_1.groupby(['now_station', 'next_station'])['route_nm'].count().reset_index(name ='count')
count_360_2 = df360_2.groupby(['now_station', 'next_station'])['route_nm'].count().reset_index(name ='count')
count_360_12 = df360_12.groupby(['now_station', 'next_station'])['route_nm'].count().reset_index(name ='count')
count_365_21 = df365_21.groupby(['now_station', 'next_station'])['route_nm'].count().reset_index(name ='count')
count_365_22 = df365_22.groupby(['now_station', 'next_station'])['route_nm'].count().reset_index(name ='count')
```

2. Data Analyze

< 이상치 분석 >

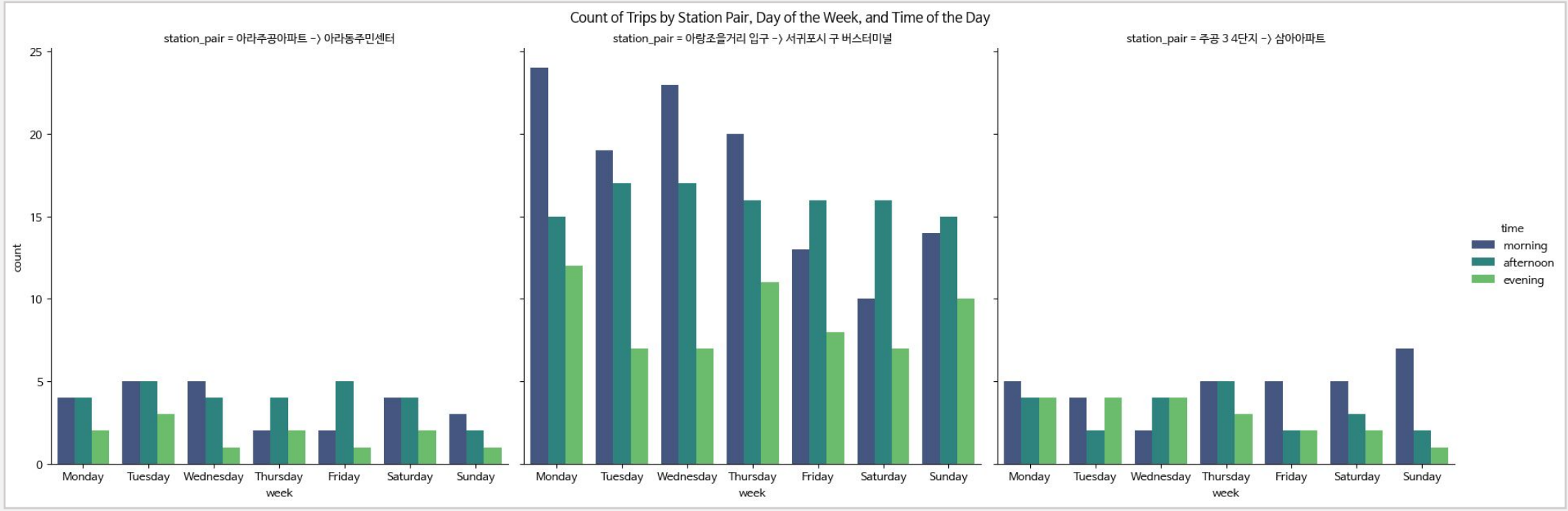
- 이상치의 요일별, 시간대별 카운트의 시각화 분석코드 (ex. 281-2번)

```python
# 281-2번의 count 64.75 이상 정류장의 요일별&시간대별 카운트의 시각화

over75_281_2 = count_281_2[count_281_2['count']>=56]
over75_281_2 = pd.merge(df281_2, over75_281_2, on = ['now_station', 'next_station'], how='inner')

# station_pair 열 추가
over75_281_2['station_pair'] = over75_281_2['now_station'] + ' -> ' + over75_281_2['next_station']

# 각 station_pair에 대한 요일별 데이터 수 계산
count_by_day_281_2 = over75_281_2.groupby(['station_pair', 'week','time'])['route_nm'].count().reset_index(name='count')

# 요일을 올바른 순서로 정렬
days_order = ['Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday', 'Saturday', 'Sunday']
times_order = ['morning','afternoon','evening']
count_by_day_281_2['week'] = pd.Categorical(count_by_day_281_2['week'], categories=days_order, ordered=True)
count_by_day_281_2['time'] = pd.Categorical(count_by_day_281_2['time'], categories=times_order, ordered=True)
count_by_day_281_2 = count_by_day_281_2.sort_values(['station_pair', 'week','time'])


plt.figure(figsize=(15, 10))
sns.catplot(x='week', y='count', hue='time', col='station_pair', data=count_by_day_281_2, kind='bar', palette='viridis', height=6, aspect=1)
plt.suptitle('Count of Trips by Station Pair, Day of the Week, and Time of the Day', y=1.02)
plt.show()
```

2. Data Analyze

< 이상치 분석>

- 이상치의 요일별, 시간대별 카운트의 시각화 자료 (ex. 281-2번)
- 결과 : 요일별로 비슷한 움직임을 보이나, 시간대별(아침,점심,저녁)로는 차이를 보임
- ***처리 방향 : 요일별 가중치는 고려하지 않고, 시간대별 가중치를 고려하여 이상치 처리***



B.D.A

3. Preprocessing1

이상치 처리 단계

1. outlier가 아닌 데이터들에서 해당 6개 노선들의 시간대별 최댓값 구하기
2. 데이터 추출(6개 노선 outlier의 count>=56인 데이터 각각 노선별로 추출)
3. 추출한 데이터 시간대별 최댓값 대체
4. 대체한 6개 노선의 이상치 데이터들을 이상치가 존재하지 않는 데이터프레임과 병합

3. Preprocessing1

<281 -2 번 예시> - 1단계

morning/ afternoon / evening 별 s/m의 최댓값 계산한 데이터프레임
count_by_time_281_2 정의

```
# 281-2번

df_without_outliers281_2= df_without_outliers6[df_without_outliers6['route_nm'] == '281-2']


count_by_time_281_2 = df_without_outliers281_2.groupby(['time','s/m'])['route_nm'].count().reset_index(name='count')


times_order = ['morning','afternoon','evening']
count_by_time_281_2['time'] = pd.Categorical(count_by_time_281_2['time'], categories=times_order, ordered=True)
count_by_time_281_2 = count_by_time_281_2.sort_values(['time','s/m'])

count_by_time_281_2= count_by_time_281_2.groupby('time')['s/m'].max()

count_by_time_281_2= count_by_time_281_2.reset_index()
count_by_time_281_2.columns = ['time', 's/m']
```

count_by_time_281_2

```
count_by_time_281_2
```

[12]:

	time	s/m
0	morning	0.439024
1	afternoon	0.439024
2	evening	0.440191

B.D.A

3. Preprocessing1

<281 -2 번 예시> - 2단계

count가 56이상인 정류장만 모아둔 over75_281_2를 추출.

이로써 count가 56을 넘지 않는 데이터들은 삭제됨

```
over75_281_2
```

	id	date	route_id	vh_id	route_nm	now_station	now_arrive_time	distance	next_station	next_arrive_time	week	s/m	m/s	time	co
0	14336	2019-10-15	405328102	7983400	281-2	아랑조을거리 입구	11시	114.0	서귀포시 구 버스터미널	54	Tuesday	0.473684	2.111111	morning	
1	14384	2019-10-15	405328102	7983400	281-2	아랑조을거리 입구	15시	114.0	서귀포시 구 버스터미널	68	Tuesday	0.596491	1.676471	afternoon	
2	14471	2019-10-15	405328102	7983406	281-2	아랑조을거리 입구	06시	114.0	서귀포시 구 버스터미널	110	Tuesday	0.964912	1.036364	morning	
3	14511	2019-10-15	405328102	7983406	281-2	아랑조을거리 입구	10시	114.0	서귀포시 구 버스터미널	110	Tuesday	0.964912	1.036364	morning	
4	14642	2019-10-15	405328102	7983409	281-2	아랑조을거리 입구	08시	114.0	서귀포시 구 버스터미널	58	Tuesday	0.508772	1.965517	morning	
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	
432	209247	2019-10-28	405328102	7983415	281-2	아라주공아파트	07시	328.0	아라동주민센터	148	Monday	0.451220	2.216216	morning	
433	209328	2019-10-28	405328102	7983415	281-2	아라주공아파트	16시	328.0	아라동주민센터	148	Monday	0.451220	2.216216	afternoon	
434	209374	2019-10-28	405328102	7983415	281-2	아라주공아파트	21시	328.0	아라동주민센터	148	Monday	0.451220	2.216216	evening	
435	209464	2019-10-28	405328102	7983430	281-2	아라주공아파트	13시	328.0	아라동주민센터	150	Monday	0.457317	2.186667	afternoon	
436	210339	2019-10-28	405328102	7983486	281-2	아라주공아파트	09시	328.0	아라동주민센터	150	Monday	0.457317	2.186667	morning	

437 rows × 15 columns

B.D.A

3. Preprocessing1

<281 -2 번 예시> - 2단계

count가 56이상인 정류장만 모아둔 over75_281_2 데이터프레임의 ‘s/m’값을
count_by_time_281_2에서 ‘time’열이 일치하는 ‘s/m’의 값을 가져와 업데이트

```
for i in range(0, 3):
    mask = (over75_281_2['time'] == count_by_time_281_2.iloc[i, 0])
    over75_281_2.loc[mask, 's/m'] = count_by_time_281_2.iloc[i, 1]
```

count_by_time_281_2

[12]:

```
count_by_time_281_2
```

[12]:

	time	s/m
0	morning	0.439024
1	afternoon	0.439024
2	evening	0.440191

B.D.A

3. Preprocessing1

<281 -2 번 예시> - 3단계

업데이트된 over75_281_2 데이터프레임.

s/m의 이상치들이 최댓값으로 대체되었음.

이렇게 모든 6개 노선에 대해 개수가 56개를 넘는 이상치 대체

	id	date	route_id	vh_id	route_nm	now_station	now_arrive_time	distance	next_station	next_arrive_time	week	s/m	m/s	time
0	14336	2019-10-15	405328102	7983400	281-2	아랑조을거리 입구	11시	114.0	서귀포시 구 버스터미널	54	Tuesday	0.439024	2.111111	morning
1	14384	2019-10-15	405328102	7983400	281-2	아랑조을거리 입구	15시	114.0	서귀포시 구 버스터미널	68	Tuesday	0.439024	1.676471	afternoon
2	14471	2019-10-15	405328102	7983406	281-2	아랑조을거리 입구	06시	114.0	서귀포시 구 버스터미널	110	Tuesday	0.439024	1.036364	morning
3	14511	2019-10-15	405328102	7983406	281-2	아랑조을거리 입구	10시	114.0	서귀포시 구 버스터미널	110	Tuesday	0.439024	1.036364	morning
4	14642	2019-10-15	405328102	7983409	281-2	아랑조을거리 입구	08시	114.0	서귀포시 구 버스터미널	58	Tuesday	0.439024	1.965517	morning
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
432	209247	2019-10-28	405328102	7983415	281-2	아라주공아파트	07시	328.0	아라동주민센터	148	Monday	0.439024	2.216216	morning
433	209328	2019-10-28	405328102	7983415	281-2	아라주공아파트	16시	328.0	아라동주민센터	148	Monday	0.439024	2.216216	afternoon
434	209374	2019-10-28	405328102	7983415	281-2	아라주공아파트	21시	328.0	아라동주민센터	148	Monday	0.440191	2.216216	evening
435	209464	2019-10-28	405328102	7983430	281-2	아라주공아파트	13시	328.0	아라동주민센터	150	Monday	0.439024	2.186667	afternoon
436	210339	2019-10-28	405328102	7983486	281-2	아라주공아파트	09시	328.0	아라동주민센터	150	Monday	0.439024	2.186667	morning

437 rows × 15 columns

3. Preprocessing1

- 4단계

(1) s/m의 이상치가 존재하지 않는 데이터프레임인 df_without_outliers 생성

```
df_without_outliers = df[df['s/m']<q]
```

(2) 이상치 처리 된 6개 노선 데이터프레임 모두 병합해 new 데이터프레임 생성

```
#new: 이상치 처리 된 6개 노선 데이터프레임을 모두 병합
new = pd.concat([over75_281_2, over75_360_1, over75_360_2, over75_360_12, over75_365_21, over75_365_22])
```

3. Preprocessing1

(3) new 프레임과 df_without_outliers 병합하여 [‘s/m’]열에 대해 전처리를 마친 df_ver1 데이터프레임 생성후, id로 내림차순 정리하여 원본 데이터와 순서를 같게 만듦.

```
[ ]  df_ver1 = pd.concat([df_without_outliers, new], axis = 0)
     df_ver1 = df_ver1.sort_values('id')
```

3. Preprocessing1

<table>
<tr><td>전체 행

210457</td><td>나머지
노선</td><td colspan="2">주요 6개
노선</td></tr>
<tr><td>이상치</td><td>2784(drop)</td><td>Under 75

1712
(drop)</td><td>Over 75

6113
(최댓값 대체)</td></tr>
<tr><td>이상치X</td><td>100608</td><td colspan="2">99240</td></tr>
</table>

df_ver1

최종 df_ver1
100608+99240
+6113
= 205961

4496개 drop, 6113개 처리 -> 205961행

B.D.A

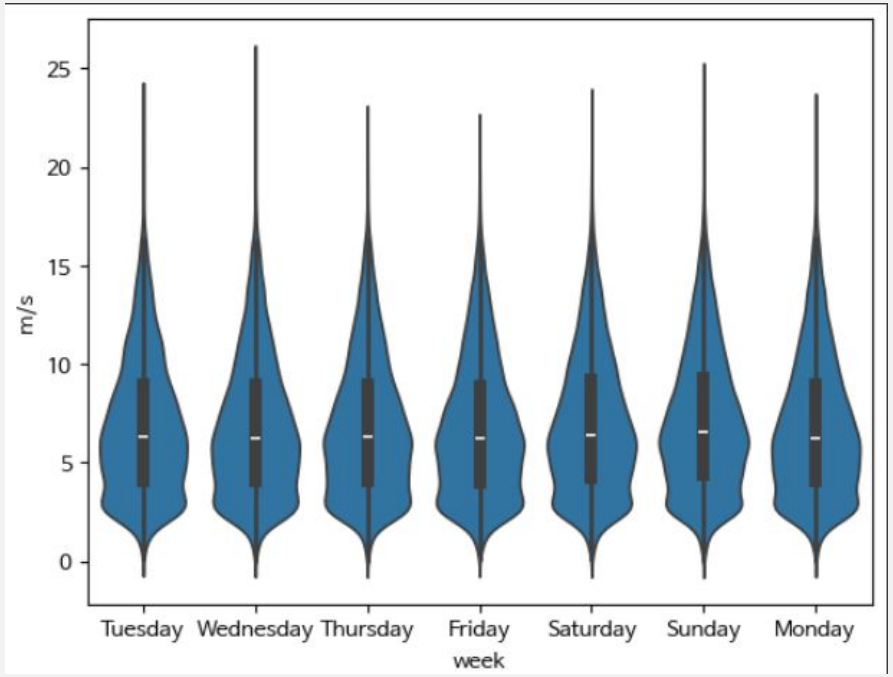
4. Preprocessing2

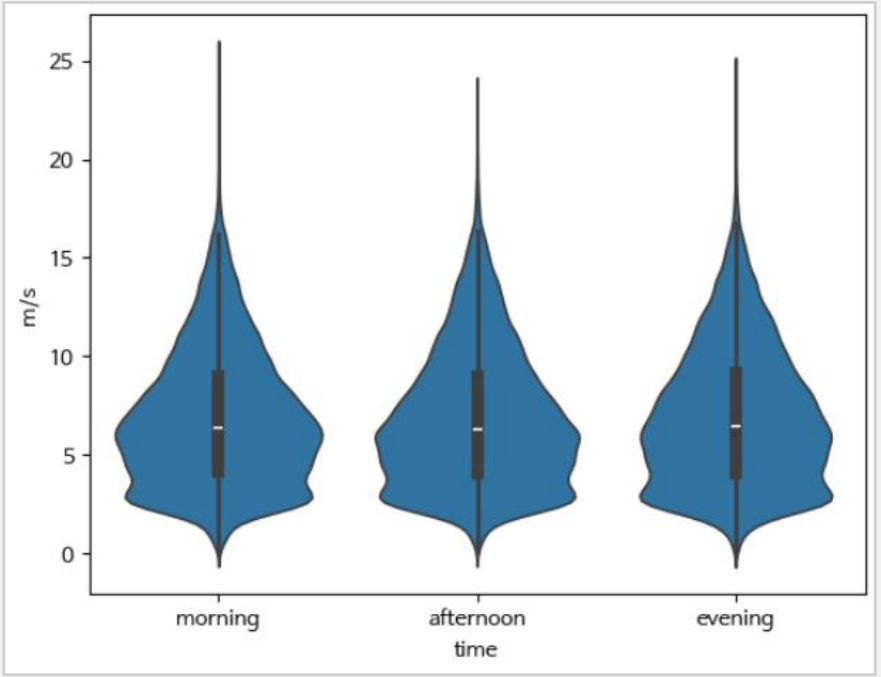
이상치 처리 단계

1. outlier가 아닌 데이터들에서 해당 6개 노선들의 시간대별 최댓값 구하기
2. 데이터 추출(6개 노선 outlier의 count>=56인 데이터 각각 노선별로 추출)
3. 추출한 데이터 시간대별 최댓값 대체
4. 대체한 6개 노선의 이상치 데이터들을 이상치가 존재하지 않는 데이터프레임과 병합

4. Preprocessing2

<이상치 분석>
- m/s 로 잡힌 전체 이상치 개수 : 1827개 (존재유무에 따라 크게 영향 받지 않을 크기)
- 전체 df의 m/s 분포 확인 : 두드러지는 이상치가 없다고 판단





4. Preprocessing2

< 이상치 제거>

- 전체 df에서 m/s가 q값을 넘어가는 값(이상치)를 삭제

```
1 q3 = df['m/s'].quantile(0.75)
2 q1 = df['m/s'].quantile(0.25)
3 q = (q3-q1)*1.5 +q3
```

```
1 q
```

```
16.52393465909091
```

```
1 df = df[df['m/s']<q]
```

5. Final Preprocessing

[준비 된 데이터]

-s/m 전처리한 df_ver1 (지연된 버스 처리)
-m/s 전처리한 df_ver2 (너무 빨랐던 버스 처리)

1. df_ver1과 df_ver2에 공통으로 존재하는 데이터 추출 (이상치가 겹치지 않음)

```
df_final = df1.merge(df2, on = ['id','date', 'route_id', 'vh_id', 'route_nm', 'now_station',
                'now_arrive_time', 'next_station', 'next_arrive_time','week', 'time'], how = 'inner')

df_final.drop(columns = ['s/m_y', 'distance_y', 'm/s_x'], inplace = True)

df_final.rename(columns = {'s/m_x' : 's/m', 'm/s_y' : 'm/s', 'distance_x' :'distance'}, inplace = True)
```

5. Final Preprocessing

2. df_final의 ‘s/m’를 이용해 next_arrive_time 업데이트

(‘m/s’의 이상치는 모두 지워졌기 때문에 고려대상이 아님)
[‘s/m’] *[distance] = next_arrive_time임을 이용해 지연된 버스의 next_arrive_time 이상치 처리

과정)
-원본 데이터를 불러와 이상치들의 id인 outlier_index list 를 저장
-df_final의 id가 이상치 id와 겹치는 경우에만 next_arrive_time 값 업데이트

```
[ ]  outlier_index = df_out['id']
     outlier_index = outlier_index.to_list()
```

```
[ ]  for idx in outlier_index:
         # 각 행에 대한 조건을 검사하여 불리언 시리즈 생성
         condition = df_final['id'] == idx

         # 조건을 만족하는 행에 대해서 'next_arrive_time' 값을 계산
         df_final.loc[condition, 'next_arrive_time'] = df_final.loc[condition, 's/m'] * df_final.loc[condition, 'distance']
```

5. Final Preprocessing

처리된 결과물]

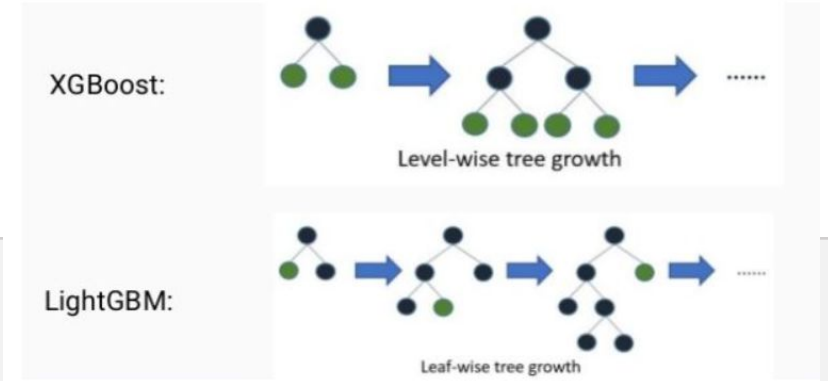
예시) 원본 데이터의 id 34225
['s/m']가 약 4.185로 distance에 비해 비정상적으로 긴 arrive_time이 걸렸음을 알 수 있음

	id	date	route_id	vh_id	route_nm	now_station	now_arrive_time	distance	next_station	next_arrive_time	week	s/m	m/s	time
1972	34225	2019-10-17	405136012	7997033	360-12	탐라도서관	20시	518.0	제주고등학교/중흥S클래스	2168	Thursday	4.185328	0.23893	evening

최종 df_final의 id 34225
['s/m']는 360-12 버스 저녁 시간대의 정상 값들의 최댓값인 0.439024로 대체됨
따라서 next_arrive_time 역시 distance * s/m = 227.414로 대체됨

	id	date	route_id	vh_id	route_nm	now_station	now_arrive_time	distance	next_station	next_arrive_time	week	s/m	time	m/s
33197	34225	2019-10-17	405136012	7997033	360-12	탐라도서관	20시	518.0	제주고등학교/중흥S클래스	227.414634	Thursday	0.439024	evening	0.23893

6. Data modeling



출처 : https://www.slideshare.net/GabrielCyprianoSaca/xgboost-lightgbm

1. GridsearchCV
최적의 모델, 즉 좋은 성능의 모델을 찾기 위해 쓰이는 기법.

사용자가 직접 모델의 하이퍼 파라미터의 값을 리스트로 입력하면 값에 대한 경우의 수마다
예측 성능을 측정 평가하여 비교하면서 최적의 하이퍼 파라미터 값을 찾는 과정을 진행

2. XGboost : Extreme gradient Boosting
Gradient Descent 기법을 Boosting(여러 개의 Decision Tree 모형으로 조합하여 사용하는 Ensemble 기법) 기법을 이용하여 병렬 학습으로 구현된 Level-wise tree 모델

3. LightGBM
XGboost 의 단점을 보완한 모델로 메모리 사용량과 학습에 걸리는 시간을 줄인 Leaf-wise tree 모델

B.D.A

6. Data modeling

1. df_final(train data) 의 범주형 데이터들의 수치화

- now_station, next_station : Label encoding 수행

```
from sklearn.preprocessing import LabelEncoder # 라벨 인코더

station_encoder = LabelEncoder() # 인코더 생성

_station = list(df_final['now_station'].values) + list(df_final['next_station'].values) # train_data 의 모든 정류장 이름
station_set = set(_station)
#print(len(station_set))

station_encoder.fit(list(station_set)) # 인코딩

# 모든 학습, 시험 데이터의 정류장 정보 치환
df_final['now_station'] = station_encoder.transform(df_final['now_station']) #train data
df_final['next_station'] = station_encoder.transform(df_final['next_station'])
```

```
df_final = pd.get_dummies(df_final,columns=['time']) # 원 핫 인코딩을 수행
df_final.head()
```

6. Data Modeling

1. df_final(train data) 의 범주형 데이터들의 수치화
   - time : one-hot encoding 수행

```
df_final = pd.get_dummies(df_final,columns=['time']) # 원 핫 인코딩을 수행
df_final.head()
```

   - week : 수치화 및 간략화 작업 수행 (평일은 0, 주말은 1)

```
## train data

df_final['date'] = pd.to_datetime(df_final['date']) # date 값을 datetime으로
df_final['week'] = df_final['date'].dt.weekday  # Monday 0, Sunday 6
df_final['week'] = df_final['week'].apply(lambda x: 0 if x <= 5 else 1)
# 0 ~ 5 는 월요일 ~ 금요일이므로 평일이면 0, 주말이면 1을 설정하였다

train_data = pd.get_dummies(df_final, columns=['week']) # 평일/주말에 대해 One-hot Encoding

train_data = train_data.drop('date', axis=1) # 필요없는 date 칼럼을 drop

## train data

train_data = train_data.drop(['id', 'now_arrive_time', 'route_nm'], axis=1)

train_data.head()
```

03. 프로젝트 결과

- 코드 실행 결과

03. 프로젝트 결과

```
input_var = list(train_data.columns)
input_var.remove('next_arrive_time')
input_var.remove('m/s')
input_var.remove('s/m')

Xtrain = train_data[input_var] # 학습 데이터 선택
Ytrain = train_data['next_arrive_time'] # target 값인 Y 데이터 선택

Xtest = test_data[input_var] # 시험 데이터도 선택
```

```
[ ]  from sklearn.model_selection import train_test_split

     X_train, X_valid, Y_train, Y_valid = train_test_split(Xtrain, Ytrain, test_size=0.2, random_state=156)
```

train 데이터에서 train test split을 이용하여 모델을 만드는 데에 활용할 데이터 분리

03. 프로젝트 결과

```
[ ]  from sklearn.model_selection import GridSearchCV

     def get_best_params(model, params):
         grid_model = GridSearchCV(model, param_grid=params, scoring='neg_mean_squared_error', cv=5)
         grid_model.fit(X_train, Y_train)
         rmse = np.sqrt(-1* grid_model.best_score_)
         print('최적 평균 RMSE 값:', np.round(rmse, 4))
         print('최적 파라미터:', grid_model.best_params_)

         return grid_model.best_estimator_
```

GridSearchCV를 사용하여 최적의 파라미터를 찾는 함수 get_best_params 함수 정의

03. 프로젝트 결과

```
[ ]  import xgboost as xgb

     # xgb_params 값을 바꾸어주며 learning_rate, max_depth 등 파라미터 성능 비교 시도했었음.
     xgb_params = {'max_depth':[3,5,7,9], 'min_child_weight':[1,3], 'colsample_bytree':[0.5,0.75], 'n_estimators':[100, 200, 980, 1000, 2500, 3300]}

     xgb_reg = xgb.XGBRegressor(learning_rate= 0.1, nthread = -1, n_jobs=-1)

     best_xgb = get_best_params(xgb_reg, xgb_params)
```

```
최적 평균 RMSE 값: 25.6401
최적 파라미터: {'colsample_bytree': 0.5, 'max_depth': 5, 'min_child_weight': 1, 'n_estimators': 3300}
```

```
from lightgbm import LGBMRegressor

# lgbm_params 값을 바꾸어주며 learning_rate, max_depth 등 파라미터 성능 비교 시도했었음.
lgbm_params = {'n_estimators':[100, 200, 450, 980, 1000, 2500, 3300]}

lgbm_reg = LGBMRegressor(learning_rate=0.01)

best_lgbm = get_best_params(lgbm_reg, lgbm_params)
```

```
최적 평균 RMSE 값: 25.6838
최적 파라미터: {'n_estimators': 3300}
```

```
최적 평균 RMSE 값: 25.4875
최적 파라미터: {'n_estimators': 1000}
```

XGBoost 모델과 LightGBM 모델을 이용하여 하이퍼파라미터 튜닝해 본 결과

XGBoost RMSE : 25.7 (n_estimators : 3300)
LightGBM RMSE : 25.4875 (n_estimators : 1000)

04. 느낀점

- 프로젝트, 스터디 하면서 느낀 점

B.D.A

04. 느낀 점

[이상준]

열정적인 팀원분들과 함께 많은 것을 배울 수 있어 재밌는 경험이 되었다.

혼자서 할 수 없었던 데이터를 분석하는 과정부터 모델링하여 성능을 검증하는 과정까지 모든 과정에서 배울 점이 많아서 많은 것을 얻어가는 기회가 된 것 같다.

[정제나]

이번 프로젝트를 통해 배웠던 모델들을 실제로 적용 및 활용할 수 있었어서,

매우 효과적인 학습 활동으로 이어질 수 있었다

또한 실제로 하나의 프로젝트가 어떻게 진행되는지 그 단계와 함수들 활용법에 대해 익힐 수 있었던 좋았다.

[김예리]

이번 프로젝트를 통해 데이터 분석과 전처리, 모델링까지 모든 과정을 해보면서 데이터를 어떻게 바라보아야 하는지에 대한 시각이 길러진 것 같았다. 또한 이 프로젝트는 데이콘을 참고하여 진행하였는데 실제 데이콘 수상 팀보다 오차가 더 낮게 나와서 매우 뿌듯했다.

B.D.A

THANK YOU