Enefit

Predict Energy Behavior of Prosumers

Predict Prosumer Energy Patterns and Minimize Imbalance Costs.

TEAM 12
2018147011 조성윤
2019147006 박호찬
2019147013 김지현
2019147021 남기정

Contents

01 주제 선정 배경

배경 설명

ESG 경영에 대한 필요성이 대두 되는 요즘
E에 해당하는 Energy 관련 주제 선정

↓

일상에서 밀접한 전기에너지에 대한 자료 조사
수요와 공급의 일치가 중요하다는 내용에 주목
→ 해당 주제와 유사성을 띄는
Kaggle competition 참여

전기 생산에 있어 중요 사항 - 수요와 공급의 일치

수요와 공급의 일치 → 에너지 시스템의 안정성과 효율성

수요와 공급의 불일치 → 에너지 낭비와 추가 비용, 블랙 아웃

국내 신재생 규모 30GW 초과
시간대에 따라 생산 변동성이 큼

Why? 예측 불가능한 소비 행태, 에너지 생산의 불규칙성, 기후 조건의 변화

Kaggle의 Enefit Competition

Enefit

발틱 지역의 가장 큰 에너지 회사 중 하나

내부 예측 모델을 개발해 불균형 문제를 해결하고자 하나
낮은 정확도로 인한 불충분한 예측 (타겟에 영향을 미치는 다양한 변수 충분히 고려x)

Kaggle을 통해 새로운 접근 방식 활용

에너지 불균형의 큰 부분 차지 → **물류·재정적 문제 초래**

Prosumer(에너지를 생산 및 소비하는 개인 또는 단체)의 에너지 생산 및 소비 예측

목표: 예측 불가능한 에너지 생산 및 소비 특성과 관련된 불균형 최소화

02 데이터 전처리

주어진 Data

train.csv

- `county` - An ID code for the county.
- `is_business` - Boolean for whether or not the prosumer is a business.
- `product_type` - ID code with the following mapping of codes to contract types: `{0: "Combined", 1: "Fixed", 2: "General service", 3: "Spot"}`.
- `target` - The consumption or production amount for the relevant segment for the hour. The segments are defined by the `county`, `is_business`, and `product_type`.
- `is_consumption` - Boolean for whether or not this row's target is consumption or production.
- `datetime` - The Estonian time in EET (UTC+2) / EEST (UTC+3).
- `data_block_id` - All rows sharing the same `data_block_id` will be available at the same forecast time. This is a function of what information is available when forecasts are actually made, at 11 AM each morning. For example, if the forecast weather `data_block_id` for predictins made on October 31st is 100 then the historic weather `data_block_id` for October 31st will be 101 as the historic weather data is only actually available the next day.
- `row_id` - A unique identifier for the row.
- `prediction_unit_id` - A unique identifier for the `county`, `is_business`, and `product_type` combination. *New prediction units can appear or disappear in the test set.*

gas_prices.csv

- `origin_date` - The date when the day-ahead prices became available.
- `forecast_date` - The date when the forecast prices should be relevant.
- `[lowest/highest]_price_per_mwh` - The lowest/highest price of natural gas that on the day ahead market that trading day, in Euros per megawatt hour equivalent.
- `data_block_id`

client.csv

- `product_type`
- `county` - An ID code for the county. See `county_id_to_name_map.json` for the mapping of ID codes to county names.
- `eic_count` - The aggregated number of consumption points (EICs - European Identifier Code).
- `installed_capacity` - Installed photovoltaic solar panel capacity in kilowatts.
- `is_business` - Boolean for whether or not the prosumer is a business.
- `date`
- `data_block_id`

electricity_prices.csv

- `origin_date`
- `forecast_date`
- `euros_per_mwh` - The price of electricity on the day ahead markets in euros per megawatt hour.
- `data_block_id`

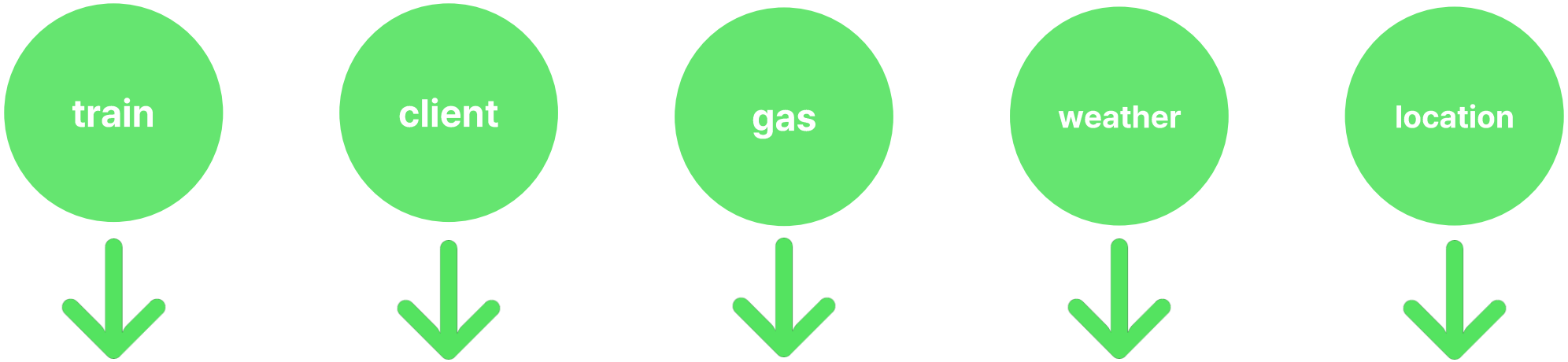
forecast_weather.csv Weather forecasts that would have been available at prediction time. Sourced from the European Centre for Medium-Range Weather Forecasts.

- `[latitude/longitude]` - The coordinates of the weather forecast.
- `origin_datetime` - The timestamp of when the forecast was generated.
- `hours_ahead` - The number of hours between the forecast generation and the forecast weather. Each forecast covers 48 hours in total.
- `temperature` - The air temperature at 2 meters above ground in degrees Celsius.
- `dewpoint` - The dew point temperature at 2 meters above ground in degrees Celsius.
- `cloudcover_[low/mid/high/total]` - The percentage of the sky covered by clouds in the following altitude bands: 0-2 km, 2-6, 6+, and total.
- `10_metre_[u/v]_wind_component` - The [eastward/northward] component of wind speed measured 10 meters above surface in meters per second.
- `data_block_id`
- `forecast_datetime` - The timestamp of the predicted weather. Generated from `origin_datetime` plus `hours_ahead`.
- `direct_solar_radiation` - The direct solar radiation reaching the surface on a plane perpendicular to the direction of the Sun accumulated during the preceding hour, in watt-hours per square meter.
- `surface_solar_radiation_downwards` - The solar radiation, both direct and diffuse, that reaches a horizontal plane at the surface of the Earth, in watt-hours per square meter.
- `snowfall` - Snowfall over the previous hour in units of meters of water equivalent.
- `total_precipitation` - The accumulated liquid, comprising rain and snow that falls on Earth's surface over the preceding hour, in units of meters.

historical_weather.csv Historic weather data.

- `datetime`
- `temperature`
- `dewpoint`
- `rain` - Different from the forecast conventions. The rain from large scale weather systems of the preceding hour in millimeters.
- `snowfall` - Different from the forecast conventions. Snowfall over the preceding hour in centimeters.
- `surface_pressure` - The air pressure at surface in hectopascals.
- `cloudcover_[low/mid/high/total]` - Different from the forecast conventions. Cloud cover at 0-3 km, 3-8, 8+, and total.
- `windspeed_10m` - Different from the forecast conventions. The wind speed at 10 meters above ground in meters per second.
- `winddirection_10m` - Different from the forecast conventions. The wind direction at 10 meters above ground in degrees.
- `shortwave_radiation` - Different from the forecast conventions. The global horizontal irradiation in watt-hours per square meter.
- `direct_solar_radiation`
- `diffuse_radiation` - Different from the forecast conventions. The diffuse solar irradiation in watt-hours per square meter.
- `[latitude/longitude]` - The coordinates of the weather station.
- `data_block_id`

Class “Feature Process Class” 제작



county	is_business	product_type	target	is_consumption	data_block_id	year	quarter	month	week	hour	...	cloudcover_high_h_mean	windspeed_10m_h_mean	winddirection_10m_h_mean	shortwave_radiation_h_mean	direct_solar_radiati
0	0	1	0.713	0	0	2021	3	9	35	0	...	0.0	5.727778	344.9	0.0	
0	0	1	96.590	1	0	2021	3	9	35	0	...	0.0	5.727778	344.9	0.0	

주어진 다양한 데이터가 하나의 df로 통일된
All in one - Feature dataframe을 생성하기 위한 class 정의

Class “Feature Process Class” 정의

```python
class FeatureProcessorClass():
    def __init__(self):
        #join 기준
        self.weather_join = ['datetime', 'county', 'data_block_id']
        self.gas_join = ['data_block_id']
        self.electricity_join = ['datetime', 'data_block_id']
        self.client_join = ['county', 'is_business', 'product_type', 'data_block_id']
        self.lat_lon_columns = ['latitude', 'longitude']
        self.agg_stats = ['mean']
```

각 data 별 join할 기준 columns 설정

추후 사용되는 lat_lon_columns, agg_stats 정의

Class “Feature Process Class” 정의

Method “create_new_column_names”&”flatten_multi_index_columns” 정의

```
def create_new_column_names(self, df, suffix, columns_no_change):
    df.columns = [col + suffix
                  if col not in columns_no_change
                  else col
                  for col in df.columns
                  ]
    return df

def flatten_multi_index_columns(self, df):
    df.columns = ['_'.join([col for col in multi_col if len(col)>0])
                  for multi_col in df.columns]
    return df
```

주어진 데이터프레임의 column이 columns_no_change에 없다면 주어진 suffix 추가, 존재한다면 추가 없이 반환

입력 받은 데이터프레임의 이름과 column의 이름을 합쳐 새로운 feature name 생성

ex)

	A		B	
	max	min	max	min
0	5	1	8	3
1	7	2	10	4

→

	A_max	A_min	B_max	B_min
0	5	1	8	3
1	7	2	10	4

Class “Feature Process Class” 정의

Method “create_data_features”& “create_client_feature” 정의

```python
def create_data_features(self, data):
    #날짜 관련 features 생성
    data['datetime'] = pd.to_datetime(data['datetime'])
    data['date'] = data['datetime'].dt.normalize()
    data['year'] = data['datetime'].dt.year
    data['quarter'] = data['datetime'].dt.quarter
    data['month'] = data['datetime'].dt.month
    data['week'] = data['datetime'].dt.isocalendar().week
    data['hour'] = data['datetime'].dt.hour
    data['day_of_year'] = data['datetime'].dt.day_of_year
    data['day_of_month']  = data['datetime'].dt.day
    data['day_of_week'] = data['datetime'].dt.day_of_week
    return data

def create_client_features(self, client):
    #client로 받은 데이터로도 신규 feature 제작
    client = self.create_new_column_names(client,
                                          suffix='_client',
                                          columns_no_change = self.client_join)
    return client
```

입력된 데이터프레임의 datetime column의 값을 시간 형식으로 변환 후

date, year, quarter, month, week, hour, day_of_year, day_of_month, day_of_week이라는 새로운 feature 생성

suffix를 추가한 새로운 column name 지정

Class “Feature Process Class” 정의

Method “create_historical_weather_features” 정의

```python
def create_historical_weather_features(self, historical_weather):

    historical_weather['datetime'] = pd.to_datetime(historical_weather['datetime'])

    historical_weather[self.lat_lon_columns] = historical_weather[self.lat_lon_columns].astype(float).round(1)
    #location 좌표를 통해 날씨 데이터와 병합
    historical_weather = historical_weather.merge(location, how = 'left', on = self.lat_lon_columns)
    historical_weather = self.create_new_column_names(historical_weather,suffix='_h',
                                                      columns_no_change = self.lat_lon_columns + self.weather_join)

    # Group by를 활용
    agg_columns = [col for col in historical_weather.columns if col not in self.lat_lon_columns + self.weather_join]
    agg_dict = {agg_col: self.agg_stats for agg_col in agg_columns}
    historical_weather = historical_weather.groupby(self.weather_join).agg(agg_dict).reset_index()

    historical_weather = self.flatten_multi_index_columns(historical_weather)

    #데이터의 특성을 고려한 전처리
    historical_weather['hour_h'] = historical_weather['datetime'].dt.hour
    historical_weather['datetime'] = (historical_weather
                                      .apply(lambda x:
                                             x['datetime'] + pd.DateOffset(1)
                                             if x['hour_h']< 11
                                             else x['datetime'] + pd.DateOffset(2),
                                             axis=1)
                                      )
    return historical_weather
```

입력된 데이터프레임의 datetime column의 값을 시간 형식으로 변환

lat_lon_columns를 county로 치환하기 위해 location과 weather를 lat_lon_columns 기준으로 병합

column name에 _h라는 suffix 추가
lat_lon_columns와 weather_join은 제외

제외되지 않은 column들에 대해 그룹화하여 집계 통계량 계산

앞서 정의한 flatten_multi_index_columns 적용

datetime의 값을 hour_h 기준으로 조정

Class “Feature Process Class” 정의

Method “create_forecast_weather_features” 정의

```python
def create_forecast_weather_features(self, forecast_weather):
    #필요없는거 날리기
    forecast_weather = (forecast_weather.rename(columns = {'forecast_datetime': 'datetime'})
                        .drop(columns = 'origin_datetime'))
    #datetime
    forecast_weather['datetime'] = (pd.to_datetime(forecast_weather['datetime']).dt
                                    .tz_convert('Europe/Brussels') .dt.tz_localize(None))
    #location 좌표를 통해 날씨 데이터와 병합
    forecast_weather[self.lat_lon_columns] = forecast_weather[self.lat_lon_columns].astype(float).round(1)
    forecast_weather = forecast_weather.merge(location, how = 'left', on = self.lat_lon_columns)
    forecast_weather = self.create_new_column_names(forecast_weather,suffix='_f',
                                                    columns_no_change = self.lat_lon_columns + self.weather_join)

    # Group by를 활용
    agg_columns = [col for col in forecast_weather.columns if col not in self.lat_lon_columns + self.weather_join]
    agg_dict = {agg_col: self.agg_stats for agg_col in agg_columns}
    forecast_weather = forecast_weather.groupby(self.weather_join).agg(agg_dict).reset_index()

    #feature화
    forecast_weather = self.flatten_multi_index_columns(forecast_weather)
    return forecast_weather
```

forecast_datetime을 datetime으로 변경 후
origin_datetime drop

datetime column의 값을 시간 형식으로 변환
Europe/Brussels 시간대로 변환 후
시간대 정보 제거

lat_lon_columns를 county로 치환하기 위해
location과 weather를
lat_lon_columns 기준으로 병합

column name에 _f라는 suffix 추가
lat_lon_columns와 weathe_join은 제외

제외되지 않은 column들에 대해 그룹화하여
집계 통계량 계산

앞서 정의한 flatten_multi_index_columns 적용

Class “Feature Process Class” 정의

Method “create_electricity_features” 정의

```
def create_electricity_features(self, electricity):
    #datetime 정리
    electricity['forecast_date'] = pd.to_datetime(electricity['forecast_date'])
    electricity['datetime'] = electricity['forecast_date'] + pd.DateOffset(1)

    # Modify column names - specify suffix
    electricity = self.create_new_column_names(electricity,suffix='_electricity',
                                               columns_no_change = self.electricity_join)
    return electricity
```

forecast_datetime의 값을 시간 형식으로 변환

forecast_date의 값이 예측 시점인 하루 전을 의미하므로 해당 값에 1을 더해 datetime 생성

column name에 _electricity라는 suffix 추가

Class “Feature Process Class” 정의

Method “create_gas_features” 정의

```
def create_gas_features(self, gas):
    # Mean값 생성
    gas['mean_price_per_mwh'] = (gas['lowest_price_per_mwh'] + gas['highest_price_per_mwh'])/2
    gas = self.create_new_column_names(gas,suffix='_gas',
                                       columns_no_change = self.gas_join)
    return gas
```

lowest_price_per_mwh와 highest_price_per_mwh 값을 이용해 mean_price_per_mwh feature 생성

column name에 _gas라는 suffix 추가

Class “Feature Process Class” 의의

```python
def __call__(self, data, client, historical_weather, forecast_weather, electricity, gas):
    # 정의한 함수들로 feature들 전체 생성
    data = self.create_data_features(data)
    client = self.create_client_features(client)
    historical_weather = self.create_historical_weather_features(historical_weather)
    forecast_weather = self.create_forecast_weather_features(forecast_weather)
    electricity = self.create_electricity_features(electricity)
    gas = self.create_gas_features(gas)
    # 하나의 df에 모두 통합
    df = data.merge(client, how='left', on = self.client_join)
    df = df.merge(historical_weather, how='left', on = self.weather_join)
    df = df.merge(forecast_weather, how='left', on = self.weather_join)
    df = df.merge(electricity, how='left', on = self.electricity_join)
    df = df.merge(gas, how='left', on = self.gas_join)

    return df
```

주어진 데이터를 하나의 df로 통일한
All in one - Feature dataframe 생성
주어진 데이터의 mean, low, high등 통계치 feature 추가

Creat_revealed_target_train 함수 정의&Target 결측치 대체

```python
def create_revealed_targets_train(data):
    # day_lag 만큼의 과거 target값을 feature로 생성
    original_datetime = data['datetime']
    revealed_targets = data[['datetime', 'prediction_unit_id', 'is_consumption', 'target']].copy()
    day_lag = 2 #kaggle test data에 과거 데이터는 이틀치만 제공
    revealed_targets['datetime'] = original_datetime + pd.DateOffset(day_lag)
    data = data.merge(revealed_targets,how='left',on = ['datetime', 'prediction_unit_id', 'is_consumption'],
                      suffixes = ('', f'_{day_lag}_days_ago'))
    return data
```

앞서 정의한 method를 사용해
이틀 전 target 값을 불러와 feature 생성

생성된 feature를 포함한 데이터프레임을
하나의 데이터프레임으로 병합 및 반환

```python
print(train.isnull().sum())
#target 결측치 모두 특정 날짜의 새벽 3am 값 --> 전날 target값으로 대체
missing_data = train[train.isna().any(axis=1)]
dt = missing_data['datetime'].unique()
for i in dt:
    idx_to_process = train[train['datetime']== i].index
    ii = i-pd.Timedelta(days=1)
    train.loc[idx_to_process,'target'] = train[train['datetime']== ii]['target'].values

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

train 데이터에 수요량과 생산량을 나타내는
'target' column에
결측치가 존재한다는 사실을 확인

결측치가 특정 datetime에서만 발생
→ 24시간 이전의 target 값으로 대체

최종 dataframe 결합 이후 결측치 보충

```python
def data_preprocessing(train, client, historical_weather, forecast_weather, electricity_prices, gas_prices):
    data = train.copy()
    client = client.copy()
    historical_weather = historical_weather.copy()
    forecast_weather = forecast_weather.copy()
    electricity = electricity_prices.copy()
    gas = gas_prices.copy()

    FeatureProcessor = FeatureProcessorClass()

    data = FeatureProcessor(data,client,historical_weather,forecast_weather,electricity,gas)
    df = create_revealed_targets_train(data)

    # Remove columns for features
    no_features = ['date','latitude','longitude','hours_ahead','hour_h',
                   'prediction_unit_id','data_block_id','currently_scored','row_id']

    remove_columns = [col for col in df.columns for no_feature in no_features if no_feature in col]
    features = [col for col in df.columns if col not in remove_columns]

    df = df[features]
    df = df.bfill()
```

FeatureProcessor를 거치고 난 후 발생하는 결측치를 bfill을 통해 보충

Standard scale, One-hot encoding 적용

```
## Standard Scaling
features_not_to_scale = ['county','is_business','product_type',
                         'is_consumption','year','quarter',
                         'month','week','hour',
                         'day_of_year','day_of_month',
                         'day_of_week','target']

features_to_scale = [col for col in df.columns if col not in features_not_to_scale]

Scaler = StandardScaler().fit(df[features_to_scale])
print(Scaler.mean_.tolist())
with open('train_scaler.pkl', 'wb') as file:
    pickle.dump(Scaler, file)

df[features_to_scale] = Scaler.transform(df[features_to_scale])
```

Numerical 한 데이터에
standard scale 적용

```
## One-hot Encoding
df = df.reset_index(drop=True)

columns_to_onehot_encode = df[['county', 'product_type']]

Encoder = OneHotEncoder(sparse_output=False)
encoded = Encoder.fit_transform(columns_to_onehot_encode)
encoded_df = pd.DataFrame(encoded, columns=Encoder.get_feature_names_out(['county', 'product_type']))

df = pd.concat([df, encoded_df], axis=1)

df.drop(['county', 'product_type'], axis=1, inplace=True)
```

categorical 데이터에
one-hot encoding 적용

Train_test_split

```
#학습용, 검증용, 시험용으로 분리
X_train, X_test = train_test_split(df, test_size=0.2)
X_train, X_val = train_test_split(X_train, test_size=0.2)

y_train = X_train.pop('target')
y_val = X_val.pop('target')
y_test = X_test.pop('target')
```

전체 데이터 기준 8:2로 train과 test 분할

분할한 train data를 다시 한번 8:2로 train과 valid로 분할

03 모델링 및 학습

모델 선정 배경

LSTM

주어진 데이터가 시계열 데이터이므로
LSTM 접근 시도

전처리 된 데이터프레임을 data_block_id 별로 구분
➔ 하나의 data_block_id 마다 6144개 행
(county 16*is_business 2*product_type 4*is_consumption 2*datetime 24)

Input: 6144*# of feature shape의 2차원 데이터
Output: 6144*1(target)
형태를 가진 모델 학습

Input shape을 일정하게 만들어 주기 위한 base df 구축

```
base = []

county = range(16)
is_business = range(2)
product_type = [1, 2, 3, 0]
is_consumption = range(2)

for i in county:
    for j in is_business:
        for k in product_type:
            for l in is_consumption:
                base.append([i, j, k, l])

base = pd.DataFrame(base, columns=['county', 'is_business',
                                   'product_type', 'is_consumption'])
```

LSTM

1) t-window size 시점부터 t 시점 까지 데이터로
t 시점 target을 예측하는 LSTM 모델의 개념과 충돌

2) 6144개 행 안에 같은 시간, 다른 장소에 대한 데이터 존재
➔
다른 시간, 같은 장소에 대한 데이터를 요하는 LSTM과 맞지 않음

county, is_business, product_type, is_consumption 별로 구분
전체 데이터에서 15312(datetime 24*data_block_id 638)개씩 활용해
개별 모델링

window_size를 24로 지정, 24시간 동안의 데이터를 사용해 target 예측
➔ test loss mean: 약 138

개별 모델 생성 및 학습

```
def train_LSTM(county, is_business, product_type, is_consumption):
    global df
    data = df[(df['is_business'] == is_business)
              & (df['is_consumption'] == is_consumption)
              & (df['county_' + str(county)] == 1)
              & (df['product_type_' + str(product_type)] == 1)]
```

⋮

```
# Model Design
global models_LSTM, rlr, ely
idx = str(county) + '_' + str(is_business) + '_' + str(product_type) + '_' + str(is_consumption)
models_LSTM[idx] = Sequential([
    LSTM(50, input_shape=(X_train.shape[1], X_train.shape[2]), return_sequences=True),
    LSTM(20, return_sequences=False),
    Dense(1, name='output')
])
models_LSTM[idx].compile(optimizer='adam', loss='mae')
```

county, is_business, product_type, is_consumption
기준으로 개별 모델을 학습하고, test를 진행하는 함수 정의

개별 모델 생성 및 학습

```
for county in range(16):
    for is_business in range(2):
        for product_type in range(4):
            for is_consumption in range(2):
                train_LSTM(county, is_business, product_type, is_consumption)
```

For 구문을 통해 개별 모델 학습 및 test 진행

```
0_0_0_0
96/96 [==============================] - 0s 5ms/step - loss: 4.6383e-05
4.6383123844861984e-05

0_0_0_1
96/96 [==============================] - 1s 6ms/step - loss: 9.0631e-09
9.063130690378784e-09

0_0_1_0
96/96 [==============================] - 1s 6ms/step - loss: 400.9623
400.9623107910156

0_0_1_1
96/96 [==============================] - 1s 6ms/step - loss: 443.1091
443.109130859375
```

```
6_0_2_1
96/96 [==============================] - 1s 6ms/step - loss: 0.0030
```

```
11_1_3_1
96/96 [==============================] - 2s 8ms/step - loss: 5215.7754
```

•
•
•

```
1  np.array(list(results.values())).mean()
```

```
138.5842235658203
```

모델 별 성능의 편차가 크지만 LSTM 성능이 좋은 조합이 존재

모델 선정 배경

LSTM 반려 & DNN 선정

학습 진행 시 모델 별 test loss의 큰 차이
&
Kaggle에 제출 시 주어진 test data는
한 data_block_id 당 24개 행
주어진 test data로
24개의 target을 예측해야 함
But
현재 모델은 연속적인 n개의 data로
n-23개의 target 예측

데이터의 column 중
시간을 나타내는 데이터 존재
&
그 외 feature들도 계절성을
반영한다 판단

DNN으로 접근해도
충분한 성능이 나올 것이라 판단

모델 선정 배경

LSTM 반려 & DNN 선정

실습 코드의 가장 기본적인 모델 구조(100_30_10) 사용

Kaggle submission 기준 1위 MAE: 62
train data 기준 test loss: 48
→ 좋은 성능 기대

But
submit score: 181
local의 test loss와 큰 차이 존재

격차를 줄이기 위한 여러 모델 구조 및 hyper parameter 시도

모델 발전 과정

hyperparameter - shuffle

train_test_split의 shuffle 여부

처음엔, 시계열 데이터이므로 shuffle이 되면 안된다고 판단

하지만, RNN이나 LSTM 등 시계열 데이터를 다루는 모델을 활용하지 않기 때문에 shuffle이 문제 되지 않음

오히려 shuffle을 통해 더 오랜 기간 동안의 정보를 모델에 학습시키는 것이 일반화 성능 향상을 위해 바람직할 것이라 가정

동일한 모델 기준 비교

shuffle =True: submit_score = 79.79
shuffle = False: submit_score = 101.80

shuffle을 통해 전체 기간 데이터를 활용하는 것이 submit_score를 올리는데 더 유리하다 판단

모델 발전 과정

hyperparameter _ elr, rlr patience, epoch 수

초기
early stop patience = learning rate patience
→ learning rate 조절이 진행 되지 않음

개선 사항
learning rate p.= 5 , early stop p. =10으로 진행 시
local minimum에 나오지 못하는 경우가
존재한다 판단
↓
rlr,elr : 5,10 → 10,20 조절

초기
epoch = 100
→ 학습이 더 필요한데 epoch가 적어 멈춘다 판단

개선 사항
epoch를 1000으로 늘리고 학습을 진행
→ 하나의 모델을 학습하는데, 너무 많은 시간 필요
→ epoch = 200이 적절하다 판단

모델 발전 과정

Ver.1	
Shuffle	False
learning rate patience	10
early stop patience	10
batch / epoch	200 / 100

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 dense (Dense)               (None, 100)               6900

 dense_1 (Dense)             (None, 30)                3030

 dense_2 (Dense)             (None, 1)                 31

=================================================================
```

Ver.2	
Shuffle	False
learning rate patience	5
early stop patience	10
batch / epoch	200 / 100

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 dense (Dense)               (None, 100)               6900

 dense_1 (Dense)             (None, 30)                3030

 dense_2 (Dense)             (None, 1)                 31

=================================================================
```

모델 발전 과정

Ver.3	
Shuffle	False
learning rate patience	10
early stop patience	20
batch / epoch	200 / 200
submit score (MAE)	101.80

Layer (type)	Output Shape	Param #
dense (Dense)	(None, 100)	6400
dense_1 (Dense)	(None, 60)	6060
dense_2 (Dense)	(None, 30)	1830
dense_3 (Dense)	(None, 30)	930
dense_4 (Dense)	(None, 10)	310
dense_5 (Dense)	(None, 10)	110
output (Dense)	(None, 1)	11

Ver.4	
Shuffle	True
learning rate patience	10
early stop patience	20
batch / epoch	200 / 200
submit score (MAE)	79.79

Layer (type)	Output Shape	Param #
dense (Dense)	(None, 100)	6400
dense_1 (Dense)	(None, 60)	6060
dense_2 (Dense)	(None, 30)	1830
dense_3 (Dense)	(None, 30)	930
dense_4 (Dense)	(None, 10)	310
dense_5 (Dense)	(None, 10)	110
output (Dense)	(None, 1)	11

모델 발전 과정

Ver.5	
Shuffle	True
learning rate patience	10
early stop patience	20
batch / epoch	200 / 200
submit score (MAE)	79.29

Layer (type)	Output Shape	Param #
dense (Dense)	(None, 100)	6400
dense_1 (Dense)	(None, 300)	30300
dense_2 (Dense)	(None, 600)	180600
dense_3 (Dense)	(None, 300)	180300
dense_4 (Dense)	(None, 100)	30100
dense_5 (Dense)	(None, 60)	6060
dense_6 (Dense)	(None, 30)	1830
dense_7 (Dense)	(None, 10)	310
output (Dense)	(None, 1)	11

Ver.6 결측치 처리 방식 변경	
Shuffle	True
learning rate patience	10
early stop patience	20
batch / epoch	200 / 200
submit score (MAE)	81.35

Layer (type)	Output Shape	Param #
dense (Dense)	(None, 100)	6400
dense_1 (Dense)	(None, 300)	30300
dense_2 (Dense)	(None, 600)	180600
dense_3 (Dense)	(None, 300)	180300
dense_4 (Dense)	(None, 100)	30100
dense_5 (Dense)	(None, 60)	6060
dense_6 (Dense)	(None, 30)	1830
dense_7 (Dense)	(None, 10)	310
output (Dense)	(None, 1)	11

모델 발전 과정

Ver.7 결측치 처리 방식 변경2	
Shuffle	True
learning rate patience	10
early stop patience	20
batch / epoch	200 / 200
submit score (MAE)	80.25

Layer (type)	Output Shape	Param #
dense (Dense)	(None, 100)	6400
dense_1 (Dense)	(None, 300)	30300
dense_2 (Dense)	(None, 600)	180600
dense_3 (Dense)	(None, 300)	180300
dense_4 (Dense)	(None, 100)	30100
dense_5 (Dense)	(None, 60)	6060
dense_6 (Dense)	(None, 30)	1830
dense_7 (Dense)	(None, 10)	310
output (Dense)	(None, 1)	11

Ver.8 feature 추가	
Shuffle	True
learning rate patience	10
early stop patience	20
batch / epoch	200 / 200
submit score (MAE)	83.73

Layer (type)	Output Shape	Param #
dense (Dense)	(None, 100)	6400
dense_1 (Dense)	(None, 300)	30300
dense_2 (Dense)	(None, 600)	180600
dense_3 (Dense)	(None, 300)	180300
dense_4 (Dense)	(None, 100)	30100
dense_5 (Dense)	(None, 60)	6060
dense_6 (Dense)	(None, 30)	1830
dense_7 (Dense)	(None, 10)	310
output (Dense)	(None, 1)	11

DNN model 구조

```python
from tensorflow.keras.layers import Dense, Input, Dropout
# perceptron, flatten 해주는 라이브러리 불러오기
from tensorflow.keras.models import Sequential
# Sequential 하게 모델을 연결해주는 라이브러리 불러오기
from tensorflow.keras.optimizers.legacy import Adam
model = Sequential([
    Input(63),
    Dense(100, activation='relu'),
    Dense(300, activation='relu'),
    Dense(600, activation='relu'),
    Dense(300, activation='relu'),
    Dense(100, activation='relu'),
    Dense(60, activation='relu'),
    Dense(30, activation='relu'),
    Dense(10, activation='relu'),
    Dense(1, name='output')
])
model.compile(optimizer=Adam(0.001), loss='mae')
model.summary()
```

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 dense (Dense)               (None, 100)               6400

 dense_1 (Dense)             (None, 300)               30300

 dense_2 (Dense)             (None, 600)               180600

 dense_3 (Dense)             (None, 300)               180300

 dense_4 (Dense)             (None, 100)               30100

 dense_5 (Dense)             (None, 60)                6060

 dense_6 (Dense)             (None, 30)                1830

 dense_7 (Dense)             (None, 10)                310

 output (Dense)              (None, 1)                 11

=================================================================
Total params: 435911 (1.66 MB)
Trainable params: 435911 (1.66 MB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
```

Ver.5	
Shuffle	True
learning rate patience	10
early stop patience	20
batch / epoch	200 / 200
submit score (MAE)	79.29 (369등)
kaggle 1st score	62.63 (1등)

다양한 모델구조를
테스트 해보았을 때
다음과 같은 구조에서
현재로선 가장 좋은 성능 발휘

Ensemble

LSTM + DNN

LSTM 모델
Kaggle 제출 불가능
+
특정 모델의 큰 오차

▼

해당 모델 DNN 대체

→

loss > 70 모델 대상으로
DNN 대체

```
print("Test MAE: ", np.array(list(test_evaluation_result.values())).mean())
✓ 0.0s
Test MAE:  9.326986274845062
```

Test Score
48 → 9.3

04 평가

XGBoost와의 비교

DNN	XGBoost
최고 성적 (MAE) : 79.29	최고 성적 (MAE) : 89.19

XGBoost에서도 hyper parameter 개선 시도
But 여전히 DNN 모델이 더 뛰어난 성능 기록

개선사항

LSTM 모델의 활용	LSTM 모델 구현은 했지만 Kaggle 제출 불가능한 문제 존재 해당 문제 해결 및 LSTM과 DNN의 앙상블 활용 예정
새로운 모델 구조 제작	기존의 DNN 모델 구조에서 일부를 변경하는 방법이 아닌 전혀 다른 구조로 층을 쌓는 시도 예정
데이터 증강	Train 데이터의 기간이 2년이 조금 안 되는 기간 -> 6, 7, 8월은 한 번씩만 포함되어 있음 -> 기간의 Imbalance를 해결하기 위해 해당 기간의 데이터를 Oversampling 등의 방식으로 데이터 불균형 해소

역할분담

조성윤

-DNN 구조 설계, 결측치 해결
-ppt 제작

박호찬

-class 오류 제거
-RNN+LSTM 모델 구축

공통

- 데이터 전처리
- class 함수 제작

김지현

- RNN+LSTM input shape 문제 시도
- 모델 학습 진행

남기정

-EDA 진행
-ppt 제작

E.O.D.