

공간데이터 분석

Jong-June Jeon

September 4, 2022

Department of Statistics, University of Seoul

유동인구 데이터

서울생활인구데이터

- 생활인구는 특정 지역의 유동적인 변화를 포함하기 위해 나온 지표
- 24시간동안 해당 지역에 있는 모든 인구를 포함
- 서울시와 KT가 공공빅데이터와 통신데이터를 이용하여 추계한 인구자료

서울생활인구 추계단계

1. KT 기지국 주변에 위치한 고객과 점유율을 사용하여 실제 인구를 추정하기 위해서 보정
2. 집계구별로 배분, 재추계하여 산출
3. 스마트폰 사용비율이 상대적으로 낮은 10세 미만, 80세 이상 인구 최후 보정
4. 최종적으로 산출된 생활인구 데이터를 내국인, 장기체류 외국인, 단기체류 외국인으로 나눠 발표

서울시생활인구데이터

서울열린데이터광장

서울 생활인구 현황

(2022.01.13 기준)

일일평균 생활인구

10,934천명



24시각 평균
내국인

10,477천명

+

24시각 평균
장기체류 외국인

377천명

+

24시각 평균
단기체류 외국인

79천명

Figure 1: 서울생활인구

서울시생활인구데이터

서울생활인구데이터 다운로드

<https://data.seoul.go.kr>


- 서울열린데이터광장 통계 > 서울 생활인구 탭 선택
- 필요에 따라 적절한 데이터 선택
- 예) 자치구단위 서울생활인구 일별 집계표 선택
- openAPI 또는 직접 csv, json 파일을 내려받기 가능

생활인구데이터의 수집과 전처리

자치구단위 서울생활인구(내국인) 데이터


데이터 내려받기

집계구 단위




- 서울 생활인구 (내국인)
- 서울 생활인구 (장기체류 외국인)
- 서울 생활인구 (단기체류 외국인)
- 통계지역경계 (집계구.shp, 2016)
- 서울 생활인구 데이터 설명서

행정동 단위



- 서울 생활인구 (내국인)
- 서울 생활인구 (장기체류 외국인)
- 서울 생활인구 (단기체류 외국인)
- 서울에서 생활한 서울 外지역 인구
- 서울에서 생활한 서울 內지역 인구

자치구 단위



- 서울 생활인구 (내국인)
- 서울 생활인구 (장기체류 외국인)
- 서울 생활인구 (단기체류 외국인)
- 일별 집계표(서울시, 자치구)
- 행정구역 코드정보

Figure 2: 서울생활인구 홈페이지

자치구단위 서울생활인구(내국인) 데이터

기준일ID	시간대구분	자치구코드	총생활인구수	남자0세부터9...	남자10세부터...
20220113	00	11110	202928.9692	4879.9787	2871.0295
20220113	00	11140	173923.0447	3690.7795	1500.4321
20220113	00	11170	256443.7587	7474.0119	3675.753
20220113	00	11200	324509.9998	11809.089	4905.8908
20220113	00	11215	365489.9287	10651.0678	5730.5488
20220113	00	11230	346784.6664	10598.7808	5270.3606
20220113	00	11260	378641.3774	12353.4366	5962.2769
20220113	00	11290	433388.9069	16521.1276	9263.5383

Figure 3: 자치구단위 서울생활인구(내국인) 예시

자치구단위 서울생활인구(내국인) 데이터 필드

- 기준일ID: 데이터 수집 날짜
- 시간대구분: 00 ~ 23까지 총 24시간
- 자치구코드: 행정동을 의미하는 번호
- 총생활인구수와 각 성별, 연령대별 생활인구수

생활인구데이터의 수집과 전처리

자치구코드

- 통계 > 서울 생활인구 > 자치구 단위 > 행정구역 코드정보에서 다운받을 수 있음
- 총 424개의 행정동코드 정보를 담고 있음

통계청행정동코드	행자부행정동코드	시도명	시군구명	행정동명
H_SDNG_CD	H_DNG_CD	DO_NM	CT_NM	H_DNG_NM
1101053	11110530	서울	종로구	사직동
1101054	11110540	서울	종로구	삼청동
1101055	11110550	서울	종로구	부암동
1101056	11110560	서울	종로구	평창동
1101057	11110570	서울	종로구	무악동
1101058	11110580	서울	종로구	교남동
1101060	11110600	서울	종로구	가회동
1101061	11110615	서울	종로구	종로1.2.3.4가동
1101063	11110630	서울	종로구	종로5.6가동
1101064	11110640	서울	종로구	이화동
1101067	11110670	서울	종로구	창신1동

Figure 4: 행정동코드 예시

유동인구의 분석과 시각화

- 행정동 shp 파일을 불러오기
- crs(좌표계) 설정해주기 : 'EPSG:4326'

```
import geopandas as gpd
hang=gpd.read_file('행정구역.shp') #데이터 불러오기
hang.crs #좌표계확인
#원하는 좌표계로 재설정
hang.geometry=hang.geometry.to_crs('EPSG:4326')
#GeoDataFrame 으로 변환
hang_geo= GeoDataFrame(hang, crs='EPSG:4326', \
                        geometry='geometry')
```

유동인구의 분석과 시각화

- 행정동 별 평균 총생활인구 시각화

```
import plotly.express as px
bin_intervals= list(count_df['총생활인구수'].quantile([0,0.25, \
                                                         0.5,0.75,1]))

fig=px.choropleth_mapbox(count_df, geojson= hang_geo,
                          locations='ADM_CD', color="총생활인구수",
                          featureidkey='properties.ADM_CD',
                          center={"lat":37.586133, "lon":126.954086},
                          mapbox_style="carto-positron",
                          zoom=9, range_color=bin_intervals)

fig.show()
```

3차원 시각화

- R 패키지 'rayshader'를 활용하여 3차원 시각화
- 지도상에서 시간별 유동인구에 따라 막대의 크기가 달라지고 색깔도 변하도록 설정
- 데이터를 보간(interpolation)하여 자연스러운 애니메이션 생성

3차원 시각화

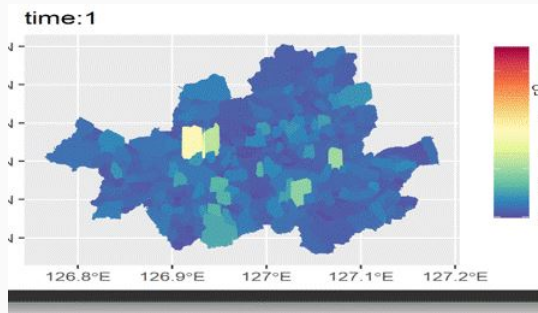


Figure 5: 3차원 시각화 결과

유동인구 예측모형 구축

생활인구는 시공간적 특성에 따라 매우 다른 패턴을 보여줌

- 도시설계에 따라 대학가, 주거밀집지역 등 행정동별로 특성이 다름
- 주중/주말 그리고 시간대별로 생활인구가 밀집되는 지역이 다름
- 연령대별로 생활인구가 밀집되는 지역이 다름

기존 데이터의 전처리 작업이 필요함

- (기존)기준일ID: 20220113 > (변경) 1월, 13일, 목요일
- (기존)자치구코드 11110530 > (변경) 종로구사직동

유동인구 예측을 위한 변량분석

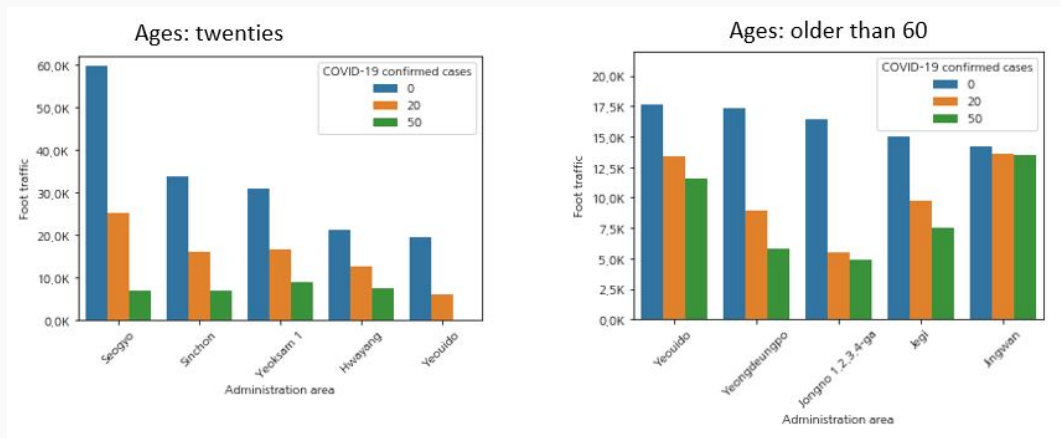


Figure 6: 연령대별 생활인구 Top5 행정동 예시

데이터 추가 작업

- 날씨 상태, 미세먼지와 같은 환경적 요인에 따라 생활인구패턴이 달라짐
- 평일도 공휴일 여부에 따라 해당 지역의 생활인구패턴이 달라짐
- 위의 변수를 추가하기 위해 공공데이터 및 openAPI 이용

기상자료수집

- 기상청 기상자료개방포털를 이용
- 종관기상관측자료(ASOS)에서 지점 > 서울특별시 선택
- 기온, 강수량, 습도와 같은 필요 변수 선택 후 csv 파일 다운로드

기상데이터의 수집과 전처리

지점	시간	기온(°C)	강수량(mm)	습도(%)
서울(108)	2022-01-11 01:00	-1.2		83
서울(108)	2022-01-11 02:00	-2.7		72
서울(108)	2022-01-11 03:00	-4.3	0.2	71
서울(108)	2022-01-11 04:00	-5.9		66
서울(108)	2022-01-11 05:00	-7.1		67
서울(108)	2022-01-11 06:00	-7.9		66
서울(108)	2022-01-11 07:00	-8.7		62
서울(108)	2022-01-11 08:00	-9.3		62

Figure 7: ASOS 자료 예시

미세먼지자료수집

- 서울시 열린데이터광장의 '서울시 일별 평균 대기오염정보' 데이터셋 사용
- 미세먼지, 초미세먼지 외 필요한 변수 사용
- 각 측정소별로 측정된 자료이지만 평균하여 서울 공통 변수로 사용

기상데이터의 수집과 전처리

측정일시	측정소명	미세먼지($\mu\text{g}/\text{m}^3$)	초미세먼지($\mu\text{g}/\text{m}^3$)
20220118	강남구	32	17
20220118	강남대로	42	18
20220118	강동구	41	22
20220118	강변북로	42	21
20220118	강북구	40	20
20220118	강서구	43	19
20220118	공항대로	44	22
20220118	관악구	30	16

Figure 8: 서울시 미세먼지 자료 예시

날짜 데이터의 처리

공휴일 데이터 받기

- 공공데이터 포털 한국천문연구원 특일 정보 API를 활용하여 공휴일 구하기
- *<https://www.data.go.kr/data/15012690/openapi.do>*의 API key 발급

필요 라이브러리 불러오기

```
import pandas as pd
import numpy as np
import requests
import json
import xmltodict
```

key = '발급받은 키를 넣어주세요'

api를 통하여 호출받은 정보를 저장할 데이터 프레임과 리스트 만들기

```
holidays = pd.DataFrame(columns=['date', 'name'])
date_list = []
name_list = []
```


날짜 데이터의 처리

```
for year_ in ['2022', '2023', '2024']:
    for month in ['01', '02', '03', '04', '05', '06', '07', '08', '09', '10', '11', '12']:
        url = "http://apis.data.go.kr/B090041/openapi/service/SpcdeInfoService/getHoliDeInfo?solYear="+year_+"&solMonth="+month+"&ServiceKey="+key
        # request 로 api data 불러오기
        content = requests.get(url).content
        #orderedDict 형태
        dict = xmltodict.parse(content)
        #xml을 json으로 변환
        jsonString = json.dumps(dict['response']['body'], ensure_ascii = False)
        #json파일을 파이썬으로
        jsonObj = json.loads(jsonString)
        if jsonObj['items'] == None:
            continue
        holi_dict = jsonObj['items']['item']
        if type(holi_dict) == list:
            for i in range(len(holi_dict)):
                date_list.append(holi_dict[i]['locdate'])
                name_list.append(holi_dict[i]['dateName'])
        else:
            date_list.append(holi_dict['locdate'])
            name_list.append(holi_dict['dateName'])

date_arr = np.array(date_list)
name_arr = np.array(name_list)
#DataFrame 형태로 저장
holidays['date'] = date_arr
holidays['name'] = name_arr
```

날짜 데이터의 처리

pandas DatetimeIndex를 사용하자

```
# DatetimeIndex에서 월 추출
```

```
pandas.DatetimeIndex.month
```

```
# DatetimeIndex에서 날 추출
```

```
pandas.DatetimeIndex.day
```

```
# DatetimeIndex에서 요일 추출
```

```
pandas.DatetimeIndex.weekday
```

날짜 데이터의 처리

```
pollution_df = pd.read_csv("서울시 일별 평균 대기오염도 정보.csv", \
                             header=0, index_col=0, encoding='CP949')
# int index를 datetime index로 변환
pollution_df.index = pd.to_datetime(pollution_df.index, \
                                     format='%Y%m%d')

pollution_df['월'] = pollution_df.index.month
pollution_df['날짜'] = pollution_df.index.day
pollution_df['요일'] = pollution_df.index.weekday
```

유동인구 예측을 위한 신경망 모형의 구성

여러개의 output에 대해서 서로 공유하는 input layer를 정의

```
shared_input_layer = [layers.Input(shared_x_input[0].shape[1])  
                       for _ in range(M)]  
  
shared_dense1 = layers.Dense(d1,  
                              kernel_regularizer=tf.keras.regularizers.l2(0.01),  
                              activation='relu')  
shared_dense2 = layers.Dense(d2,  
                              kernel_regularizer=tf.keras.regularizers.l2(0.01),  
                              activation='relu')  
  
shared_h = [shared_dense2(shared_dense1(x))  
            for x in shared_input_layer]
```

유동인구 예측을 위한 신경망 모형의 구성

shared input, input 사이의 인접정보와 각 output 계산에 필요한 input을 결합

```
input_layer = [layers.Input(x_input[0].shape[1]) for _ in range(M)]
```

```
# adjacency matrix embedding
```

```
w_loc_dense = layers.Dense(d2,  
                             kernel_regularizer=tf.keras.regularizers.l2(0.01),  
                             activation='relu')
```

```
w_loc = w_loc_dense(adj_mat)
```

```
w_loc = tf.split(w_loc, num_or_size_splits=M, axis=0)
```

```
w_loc = [tf.tile(w_loc[i], (batch_size, 1)) for i in range(M)]
```

```
concat_h = [tf.concat([x, h, w], axis=-1)  
             for x, h, w in zip(input_layer, shared_h, w_loc)]
```

유동인구 예측을 위한 신경망 모형의 구성

for loop을 이용해 결합된 input에 대해 반복적으로 신경망 모형을 계산

```
dense_layers1 = [layers.Dense(d2,  
                                kernel_regularizer=tf.keras.regularizers.l2(0.01),  
                                activation='relu') for _ in range(M)]  
drop_out1 = [layers.Dropout(0.1) for _ in range(M)]  
  
hs1 = [d(h) for d, h in zip(dense_layers1, concat_h)]  
do1 = [d(h) for d, h in zip(drop_out1, hs1)]
```

유동인구 예측을 위한 신경망 모형의 구성

여러개의 output에 대해서 각각 loss를 계산하고, 하나로 합치는 custom loss 정의, *GradientTape*에 이를 적용

```
@tf.function
def loss_fun(y, y_pred):
    loss_ = 0
    for i in range(M):
        loss_ = loss_ + tf.math.reduce_mean(tf.math.square(
            tf.cast(tf.squeeze(y[i]), tf.float32) \
                - tf.cast(y_pred[i], tf.float32)))
    return loss_

with tf.GradientTape() as tape:
    y_pred = model([x_batch, shared_x_batch])
    loss = loss_fun(y_batch, y_pred)
grads = tape.gradient(loss, model.trainable_weights)
optimizer.apply_gradients(zip(grads, model.trainable_weights))
```

신경망 모형적합

텐서플로우 커스텀 레이어의 구성

`tf.Variable`을 통해 선형 레이어의 가중치 정의하기

```
class Linear(layers.Layer): # subclass
    def __init__(self, units=32, input_dim=32):
        super(Linear, self).__init__()
        w_init = tf.random_normal_initializer()
        self.w = tf.Variable(initial_value=w_init(shape=(input_dim,
                                                         units),
                                                         dtype='float32'),
                             trainable=True)

        b_init = tf.zeros_initializer()
        self.b = tf.Variable(initial_value=b_init(shape=(units,),
                                                         dtype='float32'),
                             trainable=True)

    def call(self, inputs):
        return tf.matmul(inputs, self.w) + self.b
```

텐서플로우 커스텀 레이어의 구성

`add_weight`을 통해 선형 레이어의 가중치 정의하기

```
class Linear(layers.Layer):
    def __init__(self, units=32, input_dim=32):
        super(Linear, self).__init__()
        self.w = self.add_weight(shape=(input_dim, units),
                                initializer='random_normal',
                                trainable=True)
        self.b = self.add_weight(shape=(units, ),
                                initializer='zeros',
                                trainable=True)

    def call(self, inputs):
        return tf.matmul(inputs, self.w) + self.b
```

텐서플로우 커스텀 레이어의 구성

학습이 불가능한 weight 정의하기

```
class Compute_Sum(layers.Layer):
    def __init__(self, input_dim):
        super(Compute_Sum, self).__init__()
        self.total = tf.Variable(initial_value=tf.zeros((input_dim, )),
                                  trainable=False)

    def call(self, inputs):
        self.total.assign_add(tf.reduce_sum(inputs, axis=0))
        # 지금까지 입력된 값들을 total에 상수처럼 누적하여 저장
        return self.total
```

weights: 1

non-trainable weights: 1

trainable_weights: []

텐서플로우 커스텀 레이어의 구성

- 많은 경우에, input의 크기를 미리 알 수 없는 경우가 있고, layer를 만든 이후에 이러한 input 값이 알려지면 weights를 생성하고 싶을 수 있다.
- Keras API에서는, `build(input_shape)` method를 이용해 다음과 같이 weights를 이후에 생성할 수 있다.
- `__call__` method는 첫 번째 호출이 되는 시점에 자동으로 `build`를 실행시킨다.

```
def build(self, input_shape):  
    self.w = self.add_weight(shape=(input_shape[-1], self.units),  
                             initializer='random_normal',  
                             trainable=True)  
    self.b = self.add_weight(shape=(self.units, ),  
                             initializer='random_normal',  
                             trainable=True)
```

텐서플로우 커스텀 손실함수

- layer는 재귀적으로 전진 방향 전파 학습을 하는 도중 손실함수 값을 수집한다!
- layer에서 `call` method는 손실 값을 저장하는 tensor를 생성할 수 있도록 해주어, 후에 training loop을 작성할 때 사용가능하도록 해준다.
- `self.add_loss(value)`를 사용!

```
class ActivityRegularizationLayer(layers.Layer):  
    def __init__(self, rate=1e-2):  
        super(ActivityRegularizationLayer, self).__init__()  
        self.rate = rate  
  
    def call(self, inputs):  
        self.add_loss(self.rate * tf.reduce_sum(inputs))  
        return inputs
```

텐서플로우 커스텀 손실함수

- 이렇게 생성된 손실 값은(임의의 내부 layer의 손실 함수 값을 포함하여) `layer.losses`를 이용해 불러올 수 있다.
- 이러한 특성은 top-level layer에서의 모든 `call`의 시작에 초기화 된다.
- 이는 `layer.losses`가 항상 마지막 전진 방향 전파 학습의 손실 값만을 저장하기 위함이다.

```
class OuterLayer(layers.Layer):  
    def __init__(self):  
        super(OuterLayer, self).__init__()  
        self.activitiy_reg = ActivityRegularizationLayer(1e-2)  
  
    def call(self, inputs):  
        return self.activitiy_reg(inputs)
```

텐서플로우 커스텀 손실함수

```
layer = OuterLayer()
'''어떠한 layer도 call되지 않았으므로 손실 값이 없다'''
assert len(layer.losses) == 0
_ = layer(tf.random.normal((1, 1)))
'''layer가 1번 call되었으므로 손실 값은 1개'''
print(layer.losses)
assert len(layer.losses) == 1

[<tf.Tensor: shape=(), dtype=float32, numpy=0.007476533>]
```

```
'''layer.losses는 각각의 __call__의 시작에서 초기화'''  
_ = layer(tf.random.normal((1, 1)))  
'''마지막으로 생성된 손실 값'''  
print(layer.losses)  
assert len(layer.losses) == 1
```

```
[<tf.Tensor: shape=(), dtype=float32, numpy=0.00865575>]
```


텐서플로우 커스텀 모형 적합

MCP penalty를 계산할 custom layer 정의와 그 사용법

```
class MCP(layers.Layer):
    def __init__(self, lambda_, r):
        super(MCP, self).__init__()
        self.lambda_ = lambda_
        self.r = r

    def call(self, weight):
        penalty1 = self.lambda_ * tf.abs(weight) \
            - tf.math.square(weight) / (2. * self.r)
        penalty2 = tf.math.square(self.lambda_) * self.r / 2
        return tf.reduce_sum(
            penalty1 * tf.cast(tf.abs(weight) <= self.r * self.lambda_, tf.float32) +
            penalty2 * tf.cast(tf.abs(weight) > self.r * self.lambda_, tf.float32))

# MCP penalty 추가
mcp = MCP(lambda_, r)
model.add_loss(lambda: mcp(custom_layer.weights[0]))
```

텐서플로우 커스텀 모형 적합

custom 모형 정의 (가중치가 반복적으로 사용되는 모형)

```
class CustomLayer(layers.Layer):
    def __init__(self, output_dim, **kwargs):
        super(CustomLayer, self).__init__(**kwargs)
        self.output_dim = output_dim
        self.lambda_ = lambda_
        self.r = r
    def build(self, input_shape):
        self.w = self.add_weight(shape=(input_shape[-1], 1),
                                initializer="random_normal",
                                trainable=True)
        self.b = self.add_weight(shape=(),
                                initializer="random_normal",
                                trainable=True)
    def call(self, x):
        w_repeated = tf.repeat(self.w, self.output_dim, axis=-1)
        b_repeated = tf.repeat(self.b, self.output_dim)
        return tf.matmul(x, w_repeated) + b_repeated
```

텐서플로우 커스텀 모형 적합

*GradientTape*을 이용해 MCP penalty를 custom model 학습에 적용

```
optimizer = K.optimizers.SGD(0.01)
```

```
for i in range(100):  
    with tf.GradientTape() as tape:  
        yhat = model(X)  
        loss = tf.reduce_mean(tf.losses.mean_squared_error(y, yhat))  
        loss += sum(model.losses) # MCP penalty 적용  
  
    grad = tape.gradient(loss, model.trainable_weights)  
    optimizer.apply_gradients(zip(grad, model.trainable_weights))  
  
    if i % 10:  
        print(i, loss)
```

```
withpenalty = custom_layer.weights[0]
```

텐서플로우 커스텀 모형 적합

MCP penalty의 적용 결과 확인 (모형의 가중치 시각화)

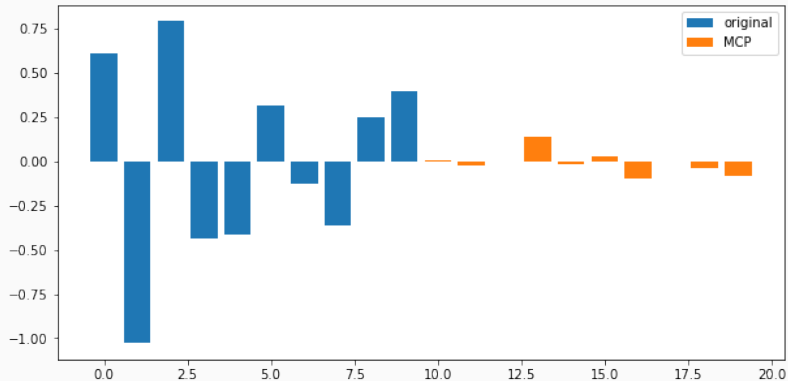


Figure 9: custom 모형의 가중치 bar plot; left: no penalty, right: MCP penalty