**[프로젝트 보고서]**

|  |  |
| --- | --- |
| **프로젝트 명** | [핫플] – 기상정보 기반 서울시 생활인구 예측 제공 서비스 |
| **팀 명** | 같이 나가조 (5조) |
| **기간** | 2021/02/23 ~ 2021/03/04 |

|  |  |  |  |
| --- | --- | --- | --- |
| 팀원 | | | |
| NO. | 성명 | 연락처 | E-mail |
| 1 | 이재호 | 010-2239-5463 | edangros@gmail.com |
| 2 | 변영준 | 010-9262-2501 | byj920501@naver.com |
| 3 | 이찬 | 010-5196-2483 | ya968574@gmail.com |
| 4 | 이지민 | 010-2671-9744 | minghgh@naver.com |

**1. 팀 소개**

**역할**

다양한 경험을 하며 전반적인 프로젝트를 이해하도록 하며, 동시에 진척사항에 따라 어느 한쪽이 지나치게 늦어지는 것을 방지하기 위해 유동적으로 인력을 조절하였다. 대략적인 역할은 아래와 같다.

**이재호(프로젝트 관리)**

전체 프레임워크 설계 및 스트림 프로세스 작성

**변영준(데이터 및 머신러닝 담당)**

데이터 분석 및 ML/AUTOML 진행, 모니터링. 모델 작성

**이찬(API 관리 및 발표)**

API 로우데이터 수령 및 정제 프로세스 작성. 기술 요약 및 발표자료 작성, 발표준비

**이지민(API 탐색)**

사용가능한 API 및 주기/호출방식 확인. 전처리 프로세스 작성 및 발표자료 작성

위는 기본적인 업무이며 진행상황에 따라 업무는 유동적으로 조절하였다

**2. 프로젝트 개요**

1. **주제**

**기상정보를 활용한 서울시 생활인구 예측 및 제공 서비스**

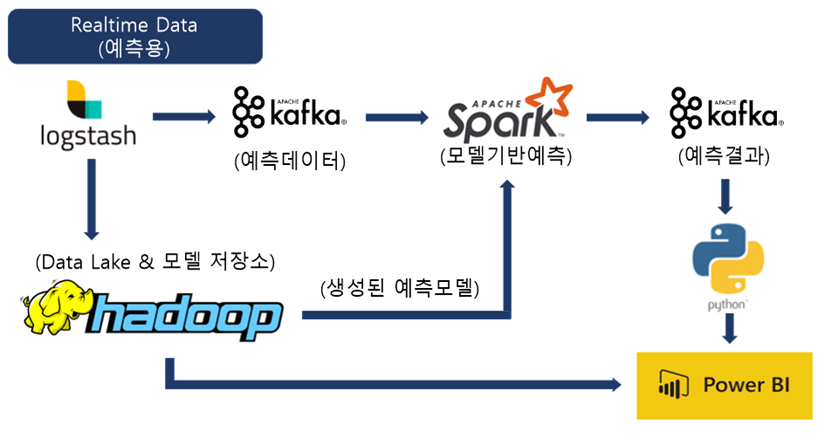
1. **이유**

서울시 생활인구 데이터는 휴대전화 보급률이 높은 한국에서 신뢰성 높은 유동인구 데이터로 이용할 수 있다. 유동인구 패턴은 개개인의 의사결정(약속, 서비스 예약 등) 뿐만 아니라 기업체/공공기관의 의사결정에서도 유용한 참고자료로 사용 가능하기에 적절한 모델로 이를 예측하는 것은 그 효용성이 크다.

시민의 생활패턴은 요일과 시간, 공휴일 여부 등의 시간적 요소뿐만 아니라 미세먼지농도, 강수량 등의 기상 패턴의 영향을 받을 것이다. 따라서 기존의 기상데이터와 생활인구 데이터를 통해 학습된 모델을 이용하여 이후의 기상정보를 실시간으로 수령, 생활인구 패턴을 예측할 수 있을 것이다.

**3.개발 내용**

**-시스템구성**



|  |  |  |
| --- | --- | --- |
| 사용기술 | 버전 | 역할 |
| Logstash | 7.11.1 | API를 수집, JSON/XML Raw Data를 해석하고 전처리하여 이후 해석이 쉽도록 필드 정리 및 규격화 |
| Apache Hadoop | 3.0.0 | Data Lake로 기본적인 이전 측정 데이터를 저장하며 ML 결과 모델 데이터를 저장, 이후 사용할 수 있도록 제공 |
| Apache Kafka | 2.2.1 | 메시지 큐로 기능하며 스트림 처리 도중의 데이터 유실을 방지하고 다중 입력, 다중 출력을 처리함 |
| Apache Zookeeper | 3.4.5 | Kafka의 동작 플랫폼으로 역할 |
| Apache Spark | 2.4.0 | Kafka로부터 출력되는 메시지 스트림을 실시간 출력, 다채널 데이터를 ML용으로 합쳐 모델 출력 결과를 다시 스트림 형태로 출력 |
| MS Power BI | -2021.02월 최신버전 | 출력된 결과물을 파이썬 스크립트로 획득, 그래픽 형태로 시각화 |
| Python | 3.6.0 | Pyspark, Pycaret, Scikit-learn 구동 |
| Oracle VirtualBox | 6.1.16 | CentOS 7 가상머신 구동 |
| CentOS 7 | 7.64-x86\_64 | 가상머신 OS |

**버전 선택**

여러 종류의 프로그램들이 맞물려 돌아가야 하는 만큼 프레임워크 구성에서 가장 중요하게 고려해야 할 사항은 버전간 호환성이었다. 이를 위해 클라우데라 하둡 배포판(CDH) 6.0.1 버전에서 제공하는 하둡/KAFKA/Spark 버전에 기초하여 프레임워크를 구성하였으며, Spark 2.4.0에 하둡 대응 버전이 존재하지 않아 상호 호환을 추가하는 아래 작업을 진행하였다.

1. Spark와 Hadoop의 연결

Spark 2.4.0은 하둡 3.0.0 전용 패키지를 제공하지 않는다. 따라서 spark 설치 후 spark\_env.yml에 다음 내용을 추가하여 하둡 설치장소로부터 직접 라이브러리를 받도록 한다. 이때 하둡과 Spark의 Scala 버전은 맞추어서 설치해야 한다. Spark 2.4.0은 Scala 2.11과 2.12를 지원하며, 본 프로젝트에서는 Scala 2.11 without-hadoop 버전을 설치하였다. 이후 Spark에서 하둡을 인식하도록 환경변수를 추가하였다.

|  |
| --- |
| export SPARK\_DIST\_CLASSPATH=$([하둡 설치경로]/bin/hadoop classpath) |

2. Spark와 Kafka의 연결

SparkSQL과 Kafka를 연결하기 위한 기본 라이브러리는 Spark에 동봉되어있지 않다. 이에 Spark 2.4.0과 Scala 2.11에 대응하는 spark-sql-kafka 라이브러리와 Kafka-Client 라이브러리를 maven 저장소에서 다운로드해 Spark의 클래스패스 경로인 jars 폴더 내부에 삽입하였다. 프로젝트에서 Elasticsearch가 제외되었지만 마찬가지로 ES-Hadoop 드라이버를 Elastic Stack 홈페이지에서 jar파일을 다운로드한 뒤 클래스패스에 추가할 수 있다.

|  |
| --- |
| *<[Spark 설치경로]/jars 폴더 아래에 jar파일로 라이브러리를 추가하였음>* |
|  |

3. Spark와 Jupyter notebook의 연결

Jupyter notebook이 Python3 위에서 동작하므로 Spark 드라이버를 Jupyter로 맞추면 worker와의 버전 충돌이 발생한다. 따라서 Worker의 실행환경은 Python 3.7에, 드라이버의 실행환경은 Jupyter notebook에 연결하여 Pyspark 실행시 파이썬 쉘이 아닌 주피터 서버가 구동되도록 하였다.

|  |
| --- |
| *<spark\_env.yml 추가 내용>* |
| export PYSPARK\_PYTHON=/usr/bin/python3  export PYSPARK\_DRIVER\_PYTHON=jupyter |

-데이터 소스

|  |  |  |
| --- | --- | --- |
| 데이터 종류 | 출처 | 데이터 형식 |
| 종관기상관측자료  (2017-2020) | 기상청 | Excel |
| 미세먼지측정 확정자료  (2017-2020) | 한국환경공단 | CSV |
| 서울시 생활인구자료  (2017-2021) | 서울시 | CSV |
| 기상청 동네예보자료  (1일) | 기상청 | XML (RSS) |
| 미세먼지 실시간 측정자료 | 공공데이터포탈 | JSON (리스트 형태) |

**데이터의 수집 및 적재**

데이터는 Logstash의 Http-poller 플러그인을 이용하여 웹상의 API와 RSS 피드로부터 데이터를 받은 뒤 가공하여 원하는 필드를 Kafka로 전송하고, 하둡에 적재하였다.

Logstash conf파일 작성시 Human Error를 줄이기 위해 VSCode의 *Logstash Configuration Syntax/Language* 플러그인을 이용하였으며 Stdout 및 file output 플러그인으로 데이터 가공 여부를 점검하며 진행하였다.

|  |
| --- |
| *home/dataplatform/logstash-7.11.1/conf/contaminationAPI.logstash.conf* |
| input {  http\_poller {  urls => {  myurl => "http://openapi.airkorea.or.kr/openapi/services/rest/ArpltnInforInqireSvc/getCtprvnRltmMesureDnsty?serviceKey=[API키]&numOfRows=40&pageNo=1&sidoName=%EC%84%9C%EC%9A%B8&ver=1.3&\_returnType=json"  }  schedule => { cron => "0 \* \* \* \* UTC"}  codec => "json"  }  }  filter {  split { field => "list" }  mutate {  copy => {  "[list][dataTime]" => "Time"  "[list][stationName]" => "location"  "[list][pm10Value]" => "pm10"  "[list][pm25Value]" => "pm25"  }  remove\_field => ["ArpltnInforInqireSvcVo","parm","list","totalCount"]  #add\_field => ["Date","time"]  }  dissect {  mapping => {  "Time" => "%{date} %{+hour}"  }  }  }  output {  #file{  # path => "/home/tdata/weather/dust\_1.csv"  # codec => json  #}  kafka {  codec => json  bootstrap\_servers => "localhost:9092"  topic\_id => "dust\_test"  }  webhdfs {  host => "127.0.0.1"  port => 9870  user => "root"  path => "/user/logs/%{+YYYY}-%{+MM}-%{+dd}/contamination-%{+HH}.log"  flush\_size => 100  idle\_flush\_time => 10  compression => "gzip"  retry\_interval => 3  codec => 'json' # write The hdfs file is in json format, otherwise the content is %{message}  }  } |

|  |
| --- |
| home/dataplatform/logstash-7.11.1/conf/weatherAPI.logstash.conf |
| input {  http\_poller {  schedule => {  #매일 3시간마다 실행시키는 cron  #cron => "0 0 0/3 \* \* UTC"  cron => "\* \* \* \* \* UTC"  }  urls => {  gangnam => "http://www.kma.go.kr/wid/queryDFSRSS.jsp?zone=1168066000"  seocho => "http://www.kma.go.kr/wid/queryDFSRSS.jsp?zone=1165066000"  songpa => "http://www.kma.go.kr/wid/queryDFSRSS.jsp?zone=1171063100"  }  }  }  filter {  #날아오는 소스가 xml이므로 json으로 변경  xml {  #message 필드가 xml형식  source => "message"  #message 필드 내부에서 xpath 쿼리 진행, 모든 data 어트리뷰트를 'data'필드에 추가  xpath => ["//data","data"]  #위와 마찬가지로 goo를 추가(주소)  xpath => ["/rss/channel/item/category","goo"]  #그 외 json으로 변한 xml 데이터는 여기로 전송  target => "\_message"  }  xml {  #xpath로 쿼리된 내용도 xml 형식 => json형식으로 바꿈  source => "goo"  target => "loc"  }  #json으로 바뀌면 필드명이 "서울특별시 xxx구 xxx동" => split해 두번째를 최종 획득  mutate {  split => {  "loc" => " "  }  copy => {  "[loc][1]" => "location"  }  }  #data 필드에는 모든 data가 리스트로 묶여있음 => 쪼개서 각자 이벤트로 만듬  split {  field => "data"  target => "\_data"  }  #안쓰는 열 다 날림  mutate {  remove\_field => ["goo","message","loc","data","chennel","\_message"]  }  #data도 xpath로 쿼리했던거라 xml형식 => json으로 형식 변경  xml {  source => "\_data"  target => "data"  }  #지금은 data : {hour : ["24"], day : [1]....}식으로 들어있음. 이걸 뜯어 필드 복구  mutate {  copy => {  "[data][seq]" => "index"  "[data][hour][0]" => "hour"  "[data][day][0]" => "dayoffset"  "[data][temp][0]" => "airtemp"  "[data][ws][0]" => "windspeed"  "[data][reh][0]" => "humidity"  "[data][r06][0]" => "rainfall"  "[data][s06][0]" => "snowfall"  }  #타입 변경  convert => {  "index" => "integer"  "hour" => "integer"  "dayoffset" => "integer"  "airtemp" => "float"  "windspeed" => "float"  "humidity" => "float"  "rainfall" => "float"  "snowfall" => "float"  }  #나머지는 필요없으니 삭제  remove\_field => ["data","\_data"]  }  }  output {  kafka {  bootstrap\_servers => ["localhost:9092"]  topic\_id => "climate\_test"  codec => json  }  stdout {    }  webhdfs {  host => "127.0.0.1"  port => 9870  user => "root"  path => "/user/logs/%{+YYYY}-%{+MM}-%{+dd}/weather-%{+HH}.log"  flush\_size => 100  idle\_flush\_time => 10  compression => "gzip"  retry\_interval => 3  codec => 'json' # write The hdfs file is in json format, otherwise the content is %{message}  }  } |

해당 명령어 실행 뒤, 윈도우 환경에서 가상머신에 접속해 KAFKA 내용을 확인하는 툴을 제작하여 Kafka의 정상 실행, 네트워크를 통한 접속기능 및 메시지의 로드 여부를 점검하였다.

|  |
| --- |
| 파이썬 Kafka Consumer |
| from kafka import KafkaConsumer  from json import loads  # topic, broker list  consumer = KafkaConsumer(  'preds',  bootstrap\_servers=['192.168.0.30:9092'],  auto\_offset\_reset='earliest',  enable\_auto\_commit=True,  group\_id='agegde',  value\_deserializer=lambda x: x.decode('utf-8'),  consumer\_timeout\_ms=1000  )  # consumer list를 가져온다  print('[begin] get consumer list')  for message in consumer:  print(message.value)  print('[end] get consumer list') |

종종 HDFS 출력이 정지하는 경우가 발생했는데, 이는 HDFS Datanode의 이상으로 인해 Datanode가 정지하면서 발생하는 문제였다. 마땅한 원인을 찾지는 못하였지만 시스템 사양을 문제로 가정하고, 가상머신을 6코어, RAM 32GB로 강화하여 처리한 결과 안정적으로 작업이 이루어졌다.

**머신러닝**

모델에는 기온/습도/미세먼지농도 등의 수치 데이터와 요일, 시간대, 공휴일여부 등의 범주형 데이터가 종합되어있다. 따라서 최종적으로 랜덤 포레스트 모델이 가장 적합한 결론을 도출할 거라고 예상하였다. 일단 다양한 모델을 참고해야 하는 시점에서 수백MB에 달하는 전체 데이터세트를 다루기에는 컴퓨팅 자원에 한계가 있다고 판단, 머신러닝 탐색은 강남구의 20-24세 남성에 대해서만 시행하였다.

ML 학습 데이터는 아래 다섯 종류의 테이블 파일로 나누어져있었다.

-Calander.csv : 2016 - 2030년의 모든 날짜에 대한 요일/공휴일/샌드위치데이 등의 날짜정보가 담겨있는 파일

-Population.csv : 2017-2021년의 자치구별 매 시간별 서울시 생활인구 기록 데이터

-Climate.csv : 2017-2020년의 ASOS 종관기상관측 데이터

-Contamination.csv : 2017-2020년의 미세먼지 관측치 확정값

-dong\_code.csv : 행정동코드와 자치구, 동 이름이 연동된 파일  
해당 데이터에서 필요한 행만 취하고, 최종적으로 한 테이블로 만들기 위해 아래와 같은 전처리과정을 거쳐 total\_data.csv파일을 획득하였고, 해당 파일로 머신러닝을 진행하였다.

|  |
| --- |
| 데이터 전처리 Jupyter Notebook 파일 |
| #모듈 임포트  import pandas as pd  import numpy as np  import json  import datetime as dt  from datetime import datetime,timedelta |
| #달력 데이터 준비  calendar = pd.read\_csv("/content/drive/MyDrive/mnt/Calendar.csv")  refined\_calendar = pd.DataFrame()  refined\_calendar['date'] = pd.to\_datetime(calendar.YYYYMMDD,format='%Y%m%d')  refined\_calendar['holiday']=calendar.HOLIDAY  refined\_calendar['n\_holiday']=calendar.NATIONAL\_HOLIDAY  refined\_calendar['big\_holiday']=calendar.BIG\_HOLIDAY  refined\_calendar['sandwich']=calendar.SANDWICH\_DAY  refined\_calendar['event']=calendar.Event  refined\_calendar['covid']=calendar['COVID-19'] |
| #기온 데이터 준비  climate = pd.read\_csv("/content/drive/MyDrive/mnt/climate.csv")  climate.head()  refined\_climate = climate[['일시','기온(°C)','강수량(mm)','풍속(m/s)','습도(%)','일조(hr)','일사(MJ/m2)','적설(cm)','전운량(10분위)','지면온도(°C)']]  column\_transform = {  '일시':'date',  '기온(°C)':'airtemp',  '강수량(mm)':'rainfall',  '풍속(m/s)':'windspeed',  '습도(%)':'humidity',  '일조(hr)':'suntime',  '일사(MJ/m2)':'sunstrength',  '적설(cm)':'snowfall',  '전운량(10분위)':'cloudamount',  '지면온도(°C)':'landtemp'  }  refined\_climate.rename(columns=column\_transform,inplace=True)  #시간을 쪼갬  refined\_climate['datetime']=pd.to\_datetime(refined\_climate.date,format="%Y-%m-%d %H:%M")  refined\_climate["hour"]=refined\_climate["datetime"].dt.hour  refined\_climate["date"]=pd.to\_datetime(refined\_climate.datetime.dt.date,format="%Y-%m-%d")  refined\_climate.fillna(value=0,inplace=True)  refined\_climate.head() |
| #1-24시간 포맷을 0-23시간으로 바꾸는 함수  #2020022024 -> 2020022100으로 바꾸는 애를 만들어야함  def pdtimecorrector(row):  #날짜부분과 시간부분을 분리  hour = row % 100  date = datetime.strptime(str(int(row / 100)),"%Y%m%d")  #24시에 대해서 처리  if hour == 24:  hour = 0#0시로 바꾸고  date = date + timedelta(days=1)#하루 뒤로 미룸  #다시 포맷 맞춰서 리턴  return(int(date.strftime("%Y%m%d"))\*100+hour) |
| #대기오염 데이터 준비  data\_contamination = pd.read\_csv("/content/drive/MyDrive/mnt/contamination.csv",encoding="euc-kr")  data\_contamination.datetime.astype(int).head()  #일부 행만 취득  contamination\_refined = data\_contamination[["location","datetime","PM10","PM25"]]  #먼저 pdtimecorrector를 통해 시간 원상복귀  contamination\_refined['datetime']=contamination\_refined.datetime.apply(lambda x : pdtimecorrector(x))  #시간과 날짜 분리  contamination\_refined['hour']=(contamination\_refined.datetime.astype(int)%100).astype(int)  #날짜는 재구축  contamination\_refined['date']=pd.to\_datetime((contamination\_refined.datetime/100).astype(int).astype(str),format='%Y%m%d')  contamination\_refined.hour.unique() |
| #인구 데이터 준비  data\_population = pd.read\_csv("/content/drive/MyDrive/mnt/population.csv",encoding='euc-kr')  data\_population.head()  refined\_population = pd.DataFrame()  tempdtindex = pd.DatetimeIndex(pd.to\_datetime(data\_population.기준일ID, format='%Y%m%d'))  #날짜를 분할 추출  refined\_population['date']=pd.to\_datetime(data\_population.기준일ID, format='%Y%m%d')  refined\_population['year']=tempdtindex.year  refined\_population['month']=tempdtindex.month  refined\_population['day']=tempdtindex.day  refined\_population['dayofweek']=tempdtindex.dayofweek  refined\_population['loccode']=data\_population['자치구코드']  refined\_population['hour']=data\_population['시간대구분']  refined\_population['pop2024m']=data\_population['남자20세부터24세생활인구수']  refined\_population['pop2024f']=data\_population['여자20세부터24세생활인구수']  refined\_population['pop4549m']=data\_population['남자45세부터49세생활인구수']  refined\_population['pop4549f']=data\_population['여자45세부터49세생활인구수']  refined\_population['pop5559m']=data\_population['남자55세부터59세생활인구수']  refined\_population['pop5559f']=data\_population['여자55세부터59세생활인구수']  refined\_population['pop70m']=data\_population['남자70세이상생활인구수']  refined\_population['pop70f']=data\_population['여자70세이상생활인구수']  refined\_population.tail(50) |
| #동 코드 데이터 정리  dong\_code = pd.read\_csv("/content/drive/MyDrive/mnt/dong\_code.csv")  dong\_code["loccode"]=(dong\_code["행정동코드"].astype(int)/100000).astype(int)  dong\_code["location"]=dong\_code["시군구명"]  dong\_code = dong\_code[["location","loccode"]].drop\_duplicates()  dong\_code.head() |
| #테이블 조인  #생활인구를 기준으로 해당 날짜 및 시간의 날씨, 미세먼지 정보, 구 이름을 조인함  total\_data = pd.merge(refined\_population, refined\_calendar, how='left', on='date')#날짜는 일단위로 제공됨  total\_data = pd.merge(total\_data, refined\_climate, how='left', on=['date','hour'])#기온 데이터는 시간단위로 제공됨  total\_data = pd.merge(total\_data, dong\_code, how='left', on='loccode')#동별로 코드가 다름  total\_data = pd.merge(total\_data, contamination\_refined, how='left', on=['date','hour','location'])#미세먼지는 날짜,시간,동까지 검사  total\_data.to\_csv("/content/drive/MyDrive/mnt/total\_data.csv") |

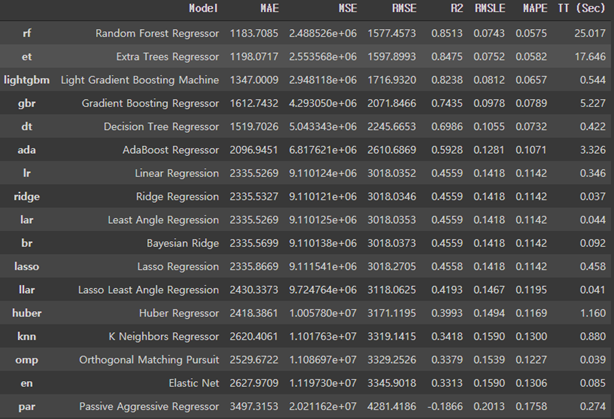
이를 통해 머신러닝을 진행한 결과는 아래와 같다

|  |
| --- |
|  |
|  |
|  |
|  |
|  |

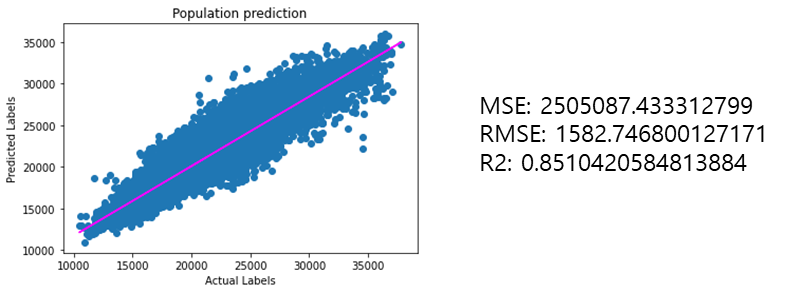
최종적으로 예상과 마찬가지로 랜덤포레스트 모델이 가장 좋은 R2값을 보여줌을 확인했다. 다만 Train Set이 아닌 Test Set에 적용시 R2값이 0 근방까지 급격히 떨어지는 오버핏 현상이 나타났다.

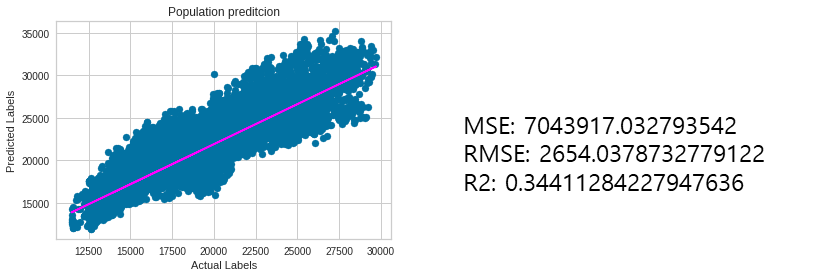
**Pycaret을 통한 Auto ML**

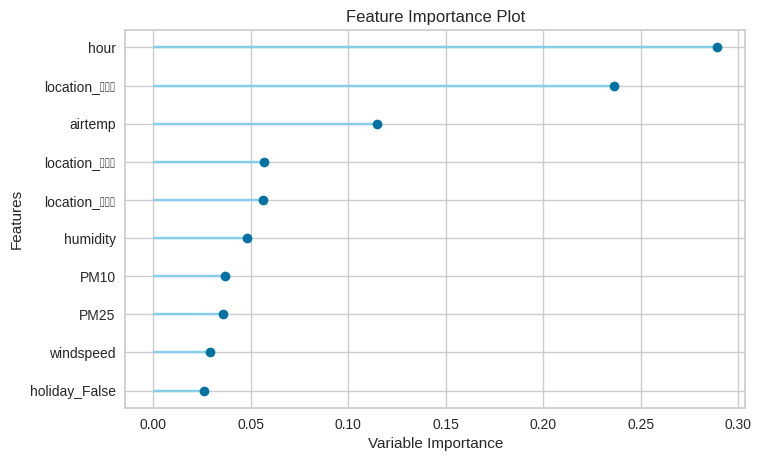
Pycaret을 사용하면 주어진 데이터세트에 맞는 최상의 모델 및 오버피팅을 방지할 수 있는 튜닝까지 머신러닝을 통해 획득할 수 있다. 따라서 모델 개선 및 오버피팅 극복을 위해 Pycaret을 사용하여 모델 탐색을 진행하였다.



탐색 결과, Scikit-learn의 모델 탐색 결과와 마찬가지로 랜덤포레스트 모델의 R2값이 가장 뛰어나게 나타남을 확인할 수 있었다. 다만, 실행시간에 있어서 Gradient Boosting Regressor 계통의 모델들이 그에 준하는 R2값과 매우 짧은 실행시간을 보여주므로 목적에 맞게 선택할 수 있을 것이다. 본 서비스에서 대부분의 API는 1시간의 긴 갱신 텀을 가지고 있기에 모델 실행 시간이 매우 중요한 요소는 아니므로, 랜덤포레스트 모델로 진행하기로 하였고, 데이터셋을 강남, 서초, 송파 3개 자치구의 20-24세 남성으로 확대하여 분석을 진행하였다.

Train Set에 대해서는 0.85의 R2값을 얻을 수 있었으며, 이는 Scikit-learn의 랜덤포레스트 모델보다 높은 값이다. 이는 Pycaret에서 자동으로 one-hot encoding 등의 항목 전처리를 실행하였기 때문으로 보인다.

Test Set에 대해서는 여전히 0.34라는 낮은 R2값을 보이나, 필드 조정의 효과로 0에 수렴했던 Scikit-learn에서의 모델보다는 월등히 높은 R2값을 가지게 되었다.

각 Feature의 영향도는 위와 같이 나타났다. 사실상 시간과 장소, 즉 시간대와 자치구 이름이 지배적인 영향을 끼치고 있었고, 기온을 제외한 나머지 기상요인의 영향은 상당히 미미했다. 기온이 시간과 서로 관련성이 높은 기상요소임을 감안한다면, 기상요소들의 영향이 상당히 제한적임을 시사한다고 볼 수 있다.

**실시간 생활인구 예측시스템**

예측시스템의 모델은 Pycaret을 통해 얻은 랜덤포레스트 모델을 사용하였다. 위에서 구성하였던 Pyspark 연동된 Jupyter notebook상에서 Kafka 스트림의 스트림-스트림 조인연산과 ML 예측을 실시간 데이터에 대해 수행할 수 있도록 하였다.

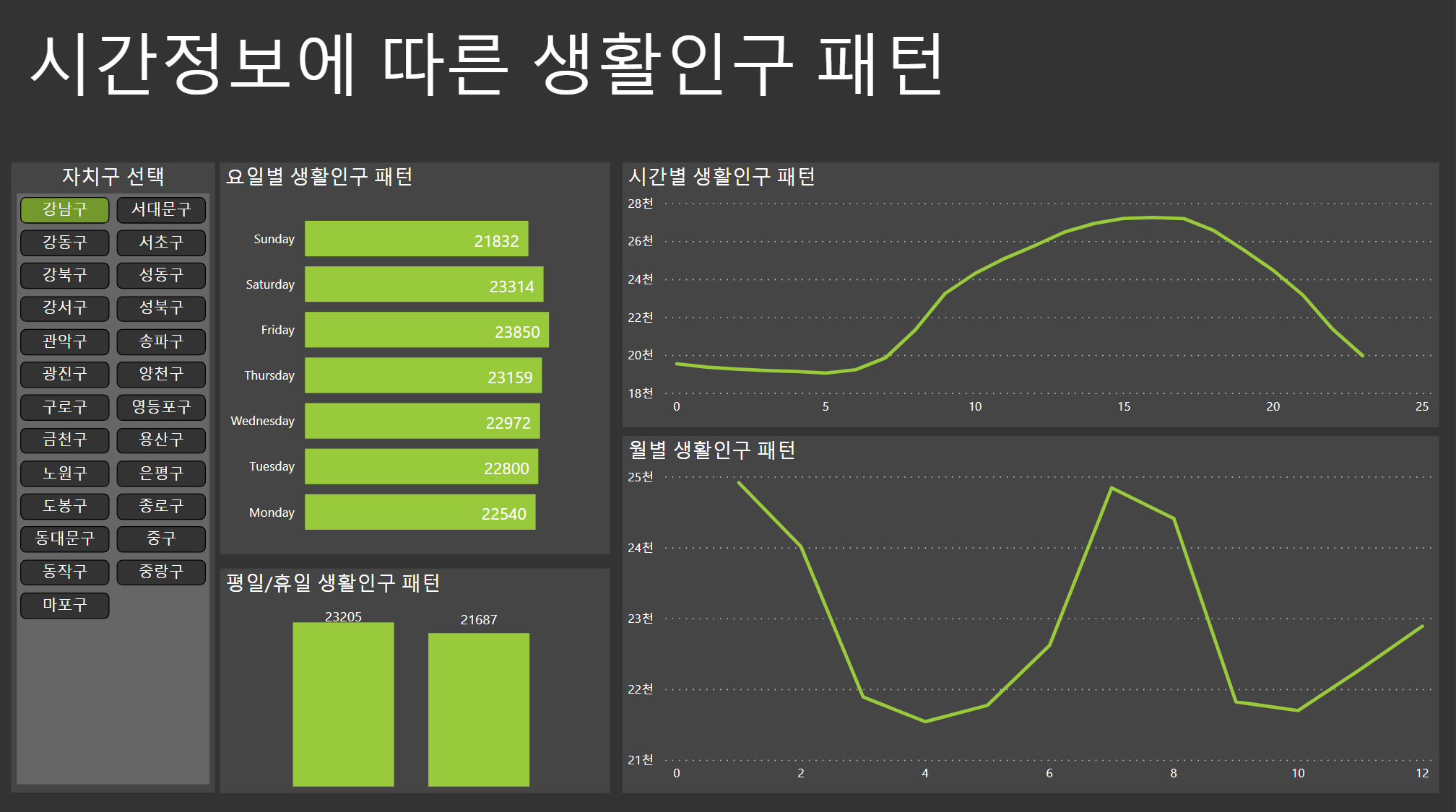
|  |
| --- |
| #모듈 임포트  from pyspark.sql import SparkSession  import pandas as pd  from pycaret.regression import \*  from datetime import datetime, timedelta |
| #스파크 세션 생성  # Spark session & context  spark = (SparkSession\  .builder\  .master('local')\  .appName('kafka\_t2')\  .config("spark.jars.packages", "org.apache.spark:spark-sql-kafka-0-10\_2.11:2.4.0")  .getOrCreate())  sc = spark.sparkContext |
| #kafka세션 1 - 기온  df\_climate\_raw = (spark  .readStream  .format("kafka")  .option("kafka.bootstrap.servers", "127.0.0.1:9092") # kafka server  .option("subscribe", "climate\_test") # topic  .option("startingOffsets", "earliest") # start from beginning  .load()) |
| #kafka세션 2 - 미세먼지  df\_dust\_raw = (spark  .readStream  .format("kafka")  .option("kafka.bootstrap.servers", "localhost:9092") # kafka server  .option("subscribe", "dust\_test") # topic  .option("startingOffsets", "earliest") # start from beginning  .load()) |
| #캘린더 로드  calendar = pd.read\_csv('/home/dataplatform/py/Calendar.csv')  refined\_calendar = pd.DataFrame()  refined\_calendar['date'] = calendar.YYYYMMDD.astype(str)  refined\_calendar['holiday']=calendar.HOLIDAY.astype(bool)  refined\_calendar['n\_holiday']=calendar.NATIONAL\_HOLIDAY.astype(bool)  refined\_calendar['big\_holiday']=calendar.BIG\_HOLIDAY.astype(bool)  refined\_calendar['sandwich']=calendar.SANDWICH\_DAY.astype(bool)  refined\_calendar['dayofweek']=pd.to\_datetime(calendar.YYYYMMDD,format='%Y%m%d').dt.dayofweek  refined\_calendar['date'] = pd.to\_datetime(calendar.YYYYMMDD,format='%Y%m%d').astype(str) |
| #스키마 복구 - 날씨  from pyspark.sql.functions import from\_json, explode  from pyspark.sql.types import StructType, StructField, BooleanType, LongType, IntegerType, FloatType, StringType  # 바이너리방식이니 복구해야지  df1 = (df\_climate\_raw  .withColumn("key", df\_climate\_raw["key"].cast(StringType()))  .withColumn("value", df\_climate\_raw["value"].cast(StringType())))  # 스키마 정의  schema\_climate = StructType(  [StructField("location",StringType(),True),  StructField("hour",StringType(),True),  StructField("dayoffset",StringType(),True),  StructField("airtemp",StringType(),True),  StructField("windspeed",StringType(),True),  StructField("humidity",StringType(),True),  StructField("rainfall",StringType(),True),  StructField("snowfall",StringType(),True),  StructField("@timestamp",StringType(),True)  ]  )  # 스키마를 통해 데이터 복구  df\_climate = (df1  # value 안에 데이터 복구  .withColumn("value", from\_json("value", schema\_climate))  #평탄화  .select("timestamp","value.\*")  #워터마크  .withWatermark("timestamp", "3 hours ")  ) |
| # 스키마 복구  df2 = (df\_dust\_raw  .withColumn("key", df\_dust\_raw["key"].cast(StringType()))  .withColumn("value", df\_dust\_raw["value"].cast(StringType())))  # 스키마 정의  schema\_dust = StructType(  [StructField("location",StringType(),True),  StructField("pm25",StringType(),True),  StructField("pm10",StringType(),True),  StructField("date",StringType(),True),  StructField("hour",StringType(),True)  ]  )  # 스키마를 통해 데이터 복구  df\_dust = (df2  # value 안에 데이터 복구  .withColumn("value", from\_json("value", schema\_dust))  .select("timestamp","value.\*")  #워터마크  .withWatermark("timestamp", "3 hours ")  ) |
| from pyspark.sql.functions import expr  #그다음은 조인  #https://docs.databricks.com/\_static/notebooks/stream-stream-joins-python.html  df\_total = df\_climate.alias("cl").join(  df\_dust.alias("du"),  expr("""  cl.location = du.location  """)).select("cl.timestamp",  "cl.location",  "cl.hour",  "cl.dayoffset",  "cl.airtemp",  "cl.windspeed",  "cl.humidity",  "cl.rainfall",  "cl.snowfall",  "du.pm25",  "du.pm10",  "du.date") |
| #모델 로드  pop\_model = load\_model('/home/dataplatform/py/popmode\_pycaret\_0304')  #사용자 정의 함수 (UDF)를 생성함.  # https://chioni.github.io/posts/pandasudf/  #데이터베이스에서의 그것과 똑같음. 매 줄마다 실행할거임  from pyspark.sql.functions import udf  #1. 딕셔너리 구성 -> pandas dataframe으로 바꿔서  #2. calender와 join해서 휴일 등 정보를 획득하고  #3. pycaret 모델에 넣어 predict된 값을 리턴  #데이터로 받아야 하는 필드 : location, hour, airtemp, windspeed, humidity, rainfall, snowfall, pm25, pm10, date,dayoffset  def predict\_pop(location, hour, airtemp, windspeed, humidity, rainfall, snowfall, pm25, pm10, date, dayoffset):  #1. 필드 복구  temp\_dict = {"location":location,  "hour":int(hour),  "airtemp":float(airtemp),  "windspeed":float(windspeed),  "humidity":float(humidity),  "n\_rainfall":float(rainfall)>0,#내림/안내림으로 값 조정했음  "n\_snowfall":float(snowfall)>0,#내림/안내림으로 값 조정했음  "PM25":float(pm25),  "PM10":float(pm10),  "date":datetime.strptime(date,"%Y-%m-%d") + timedelta(days=int(dayoffset))#dayoffset만큼 내일로 밀어야 함  }  ddf = pd.DataFrame.from\_dict(temp\_dict, orient = "index").transpose()  ddf['date'] = ddf.date.astype(str)  #ddf = ddf.join(refined\_calendar[['dayofweek','date','holiday','n\_holiday','big\_holiday','sandwich']],on='date', how='left')  ddf = ddf.merge(refined\_calendar[['dayofweek','date','holiday','n\_holiday','big\_holiday','sandwich']],on='date', how='left')  #만들어진거 기반 모델 예측  pred = predict\_model(pop\_model, data = ddf)  return float(pred["Label"])  score\_udf = udf(predict\_pop, FloatType()) #udf 등록 |
| result\_df = df\_total.withColumn("prediction", score\_udf("location","hour","airtemp","windspeed","humidity","rainfall","snowfall","pm25","pm10","date","dayoffset"))    #콘솔에서 보기  result\_df.writeStream.format("console").start() |
| #카프카 쓰기 스트림 개설  from pyspark.sql.functions import to\_json, struct  query = (result\_df.select(to\_json(struct([result\_df[x] for x in result\_df.columns])).alias("value")).writeStream.format("kafka")  .option("kafka.bootstrap.servers", "192.168.0.30:9092")  .option("topic", "preds")  .option("checkpointLocation", "/home/dataplatform/py/temp").start()  ) |

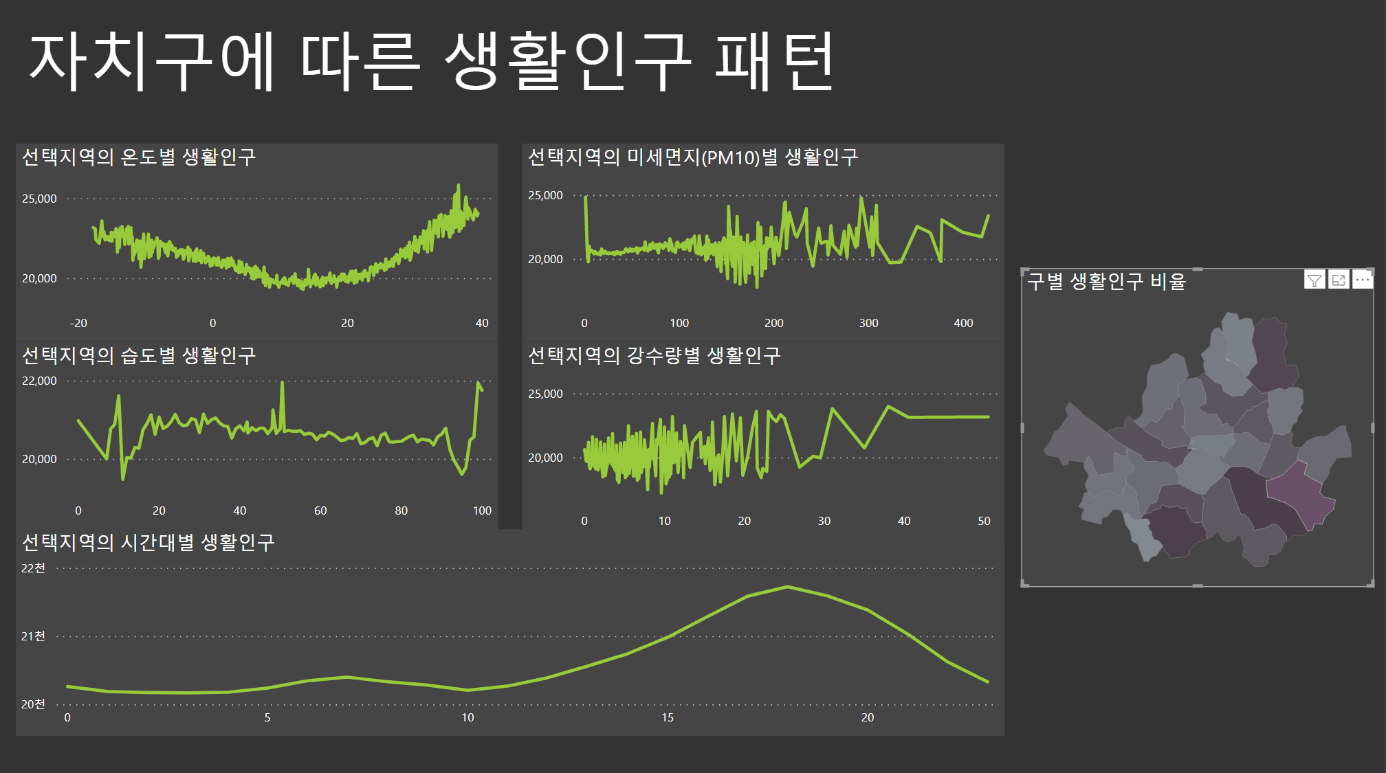
이로부터 KAFKA에 적히는 데이터는 Logstash를 통해 수집되었던 현 시점의 미세먼지와 동네 기상 예보, 그리고 그로부터 예측된 각 지역별 생활인구이다.

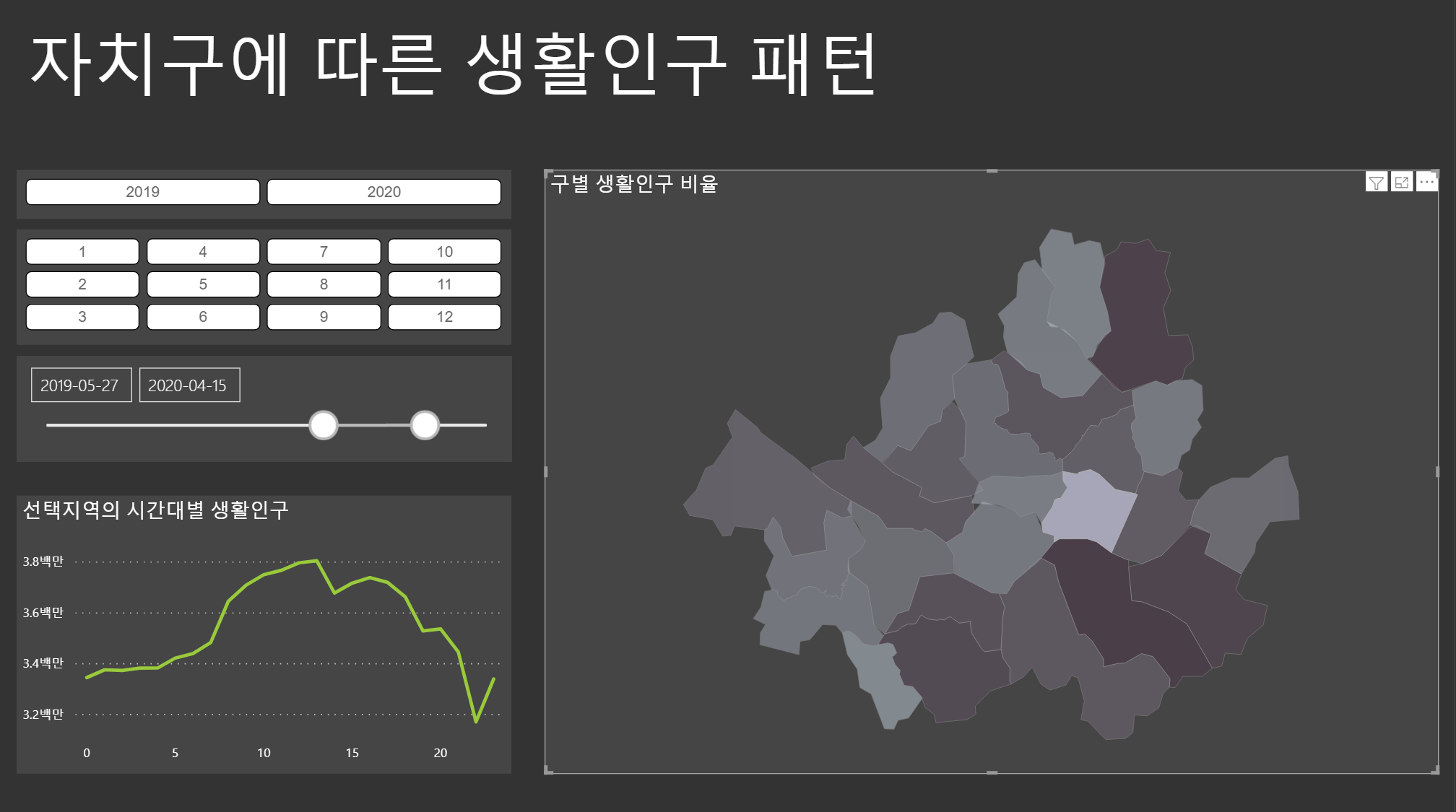
**4. 결과물**

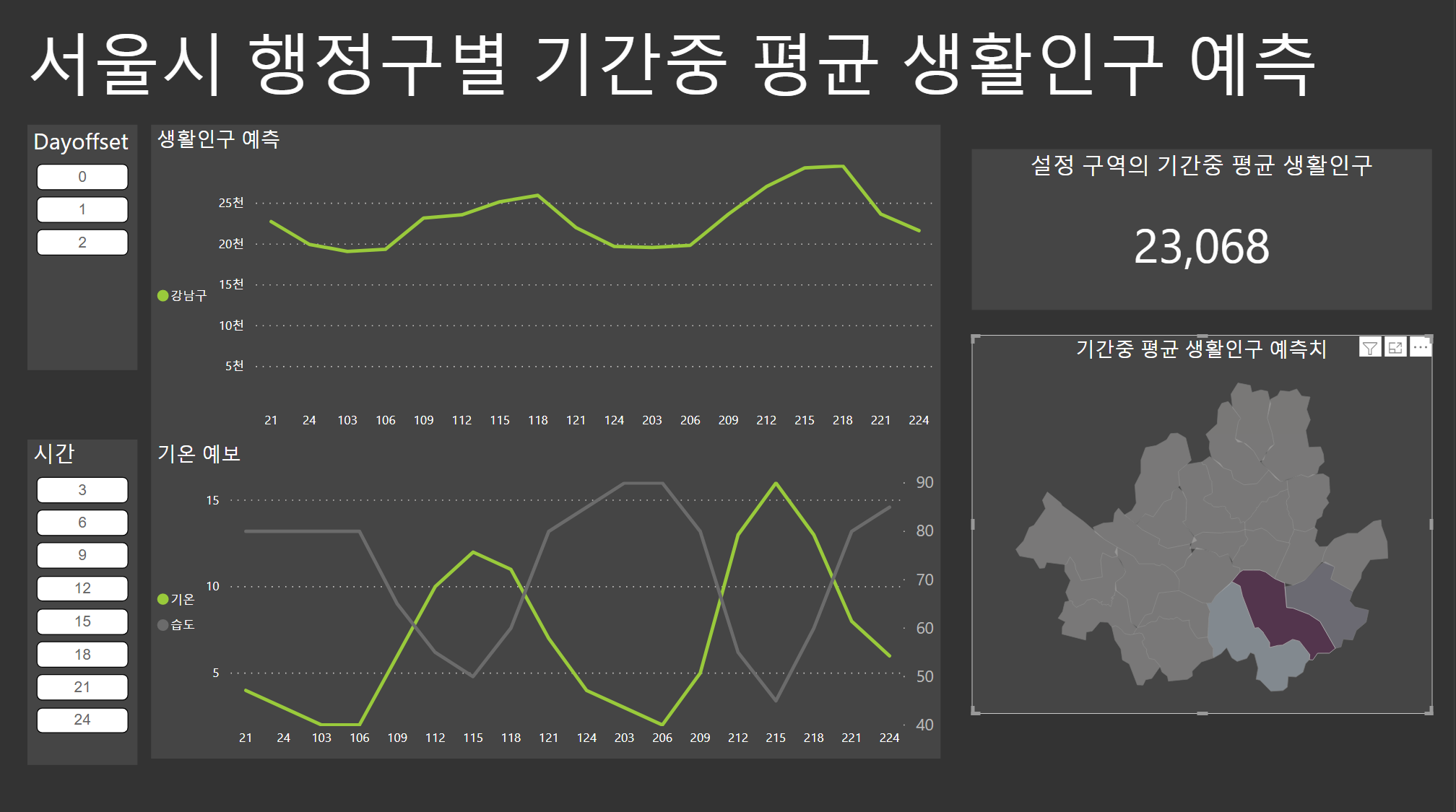
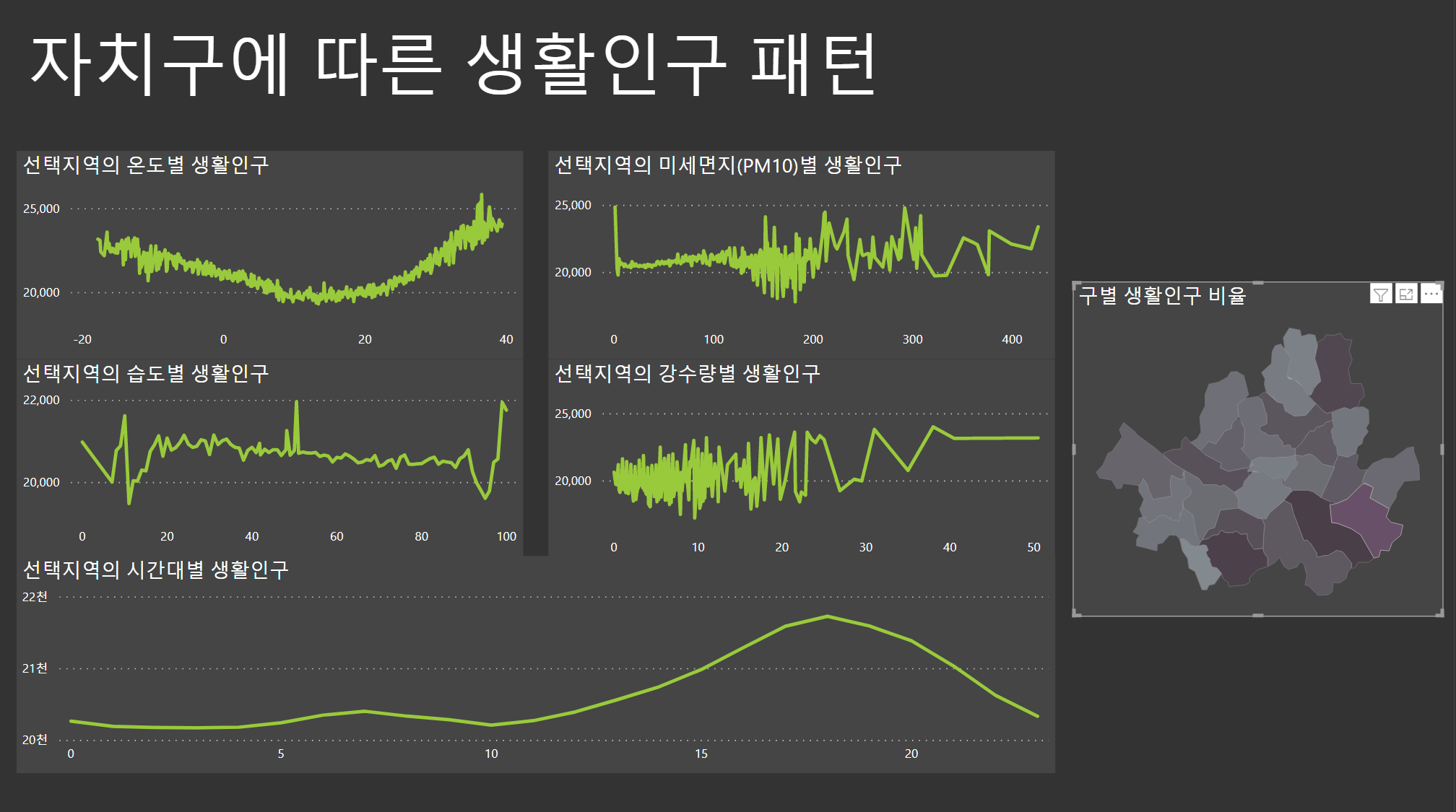
시각화를 위해서 Microsoft Power BI를 사용하였으며, 아래 스크립트를 사용하여 바로 로드하려 하였으나, Power BI상의 에러로 인해 직접 로드할 수 없어 CSV파일을 통해 로드하였다. 여러 차례의 개선이 이루지고 있지만 현 시점에서 Power BI의 스크립트를 통한 데이터 로드 기능은 불안정한 상태이다.

|  |
| --- |
| kafkaconsumer.py |
| from kafka import KafkaConsumer  from json import loads  # topic, broker list  consumer = KafkaConsumer(  'preds',  bootstrap\_servers=['192.168.0.30:9092'],  auto\_offset\_reset='earliest',  enable\_auto\_commit=True,  group\_id='power\_bi',  value\_deserializer=lambda x: x.decode('utf-8'),  consumer\_timeout\_ms=1000  )  messages = []  # consumer list를 가져온다  print('[begin] get consumer list')  for message in consumer:  messages.append(message)  print('[end] get consumer list')  print('KAFKA로부터 %d 개의 데이터가 입력되었습니다.'%len(messages))  # 데이터테이블 구축  import pandas as pd  temparray = []  for m in messages:  tempdf = pd.DataFrame.from\_dict(loads(m.value),orient="index").transpose()  temparray.append(tempdf)  datas = pd.concat(temparray)  temparray = ""  tempdf = ""  #달력 로드  calendar = pd.read\_csv("C:\\Users\\User\\Desktop\\Minipj\\Calendar.csv")  #날짜데이터 만들기  calendar["date"]=pd.to\_datetime(calendar.YYYYMMDD, format="%Y%m%d")  datas["date"]=pd.to\_datetime(datas.date,format="%Y-%m-%d")  joined\_data = pd.merge(datas,calendar,how="left",on="date")  joined\_data.to\_csv("C:\\Users\\User\\Desktop\\Minipj\\preicted.csv") |







프로젝트 기간상 시각화에 큰 시간을 쏟지는 못하였지만 기존 데이터에 대한 분석을 시각화하였으며, 최종적으로는 처리된 실시간 데이터 기반의 생활인구 예측치 또한 표시시키는 데에 성공하였다. 강남구의 경우 일반적인 생활 패턴, 즉 낮 시간대에 생활인구가 증가하는 패턴이 계속 예측되었으며, 1일 후보다 2일 후에 다소 증가한 기온의 영향으로 생활인구가 더 높을 것으로 추측되었다.

**5. 문제점 및 개선사항**

**문제점**

모델의 정확도가 개선되지 못한 점이 현 시점에서 가장 큰 문제이다. Pycaret의 Hyperparameter Tuning 과정을 진행하여 하였으나, 메모리 부족으로 인해 해당 과정을 진행하지 못했다. 또다른 방법으로는 범주형 데이터인 시간과 지역의 영향력이 압도적으로 지배적인 요소인 만큼, 데이터를 시간과 지역 단위로 쪼갠 뒤 각 지역의 각 시간대별로 별개의 모델을 구축하는 방법이 있겠지만 마찬가지로 연산 시간의 부족으로 인해 수행할 수 없었다.

시각화 또한 PowerBI는 실시간 시각화에 적합한 툴이 아니므로 웹 기반의 Grafana 등 다른 시각화 툴로의 시스템 이전이 필요하다.

**개선사항**

-모델에 대해 Hyperparameter Tuning을 진행해 오버피팅을 통제한다.

-데이터를 시각과 지역 단위로 쪼개 별개의 모델을 구축하고 스트림 입력값에 따라 적절한 모델이 선택되어 예측에 사용되도록 스트림 처리를 개선한다.

-실시간 시각화가 가능하도록 Elasticsearch를 엔드포인트에 추가, 예측 결과를 저장한 뒤 Grafana등으로 시각화한다.