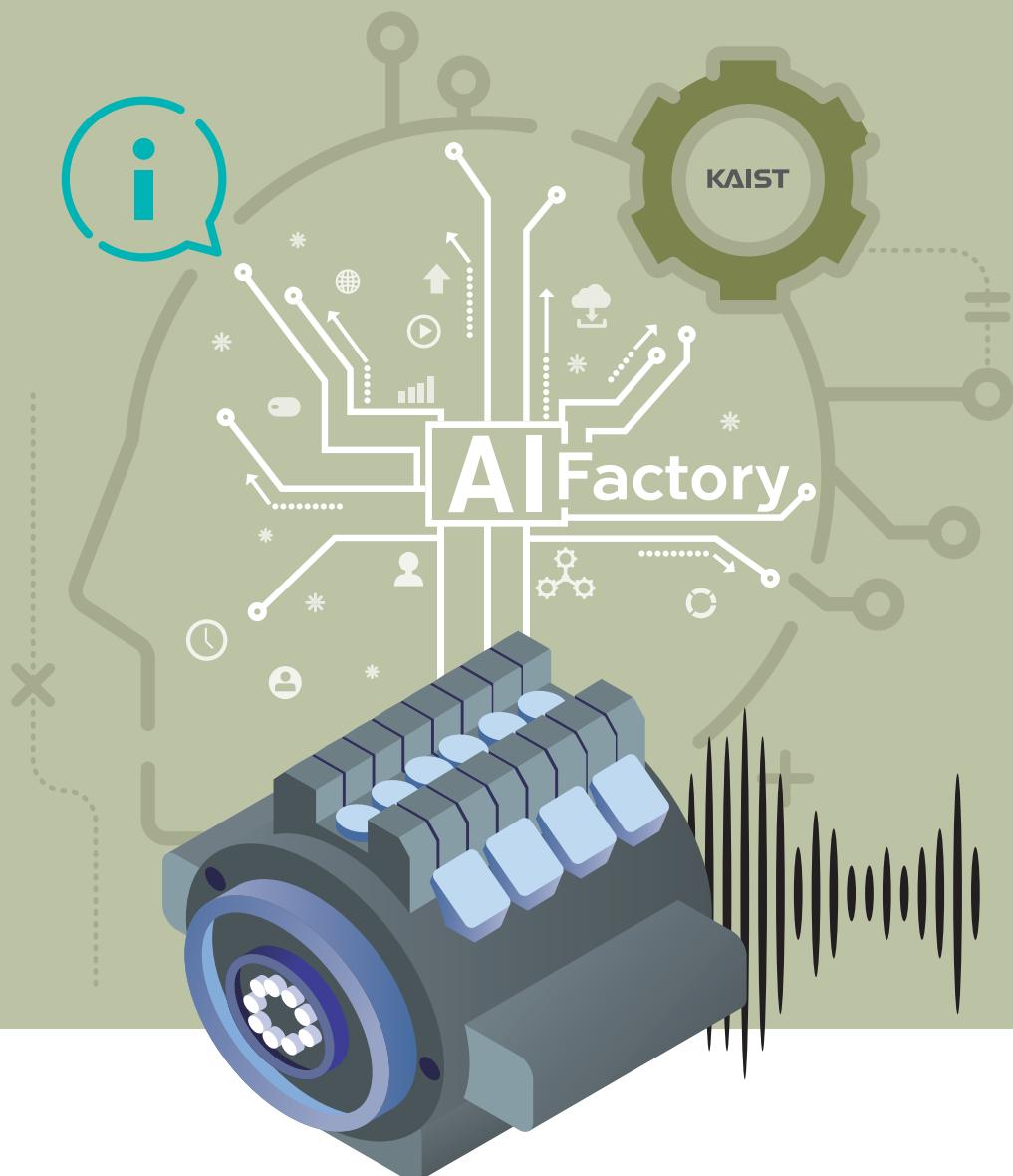
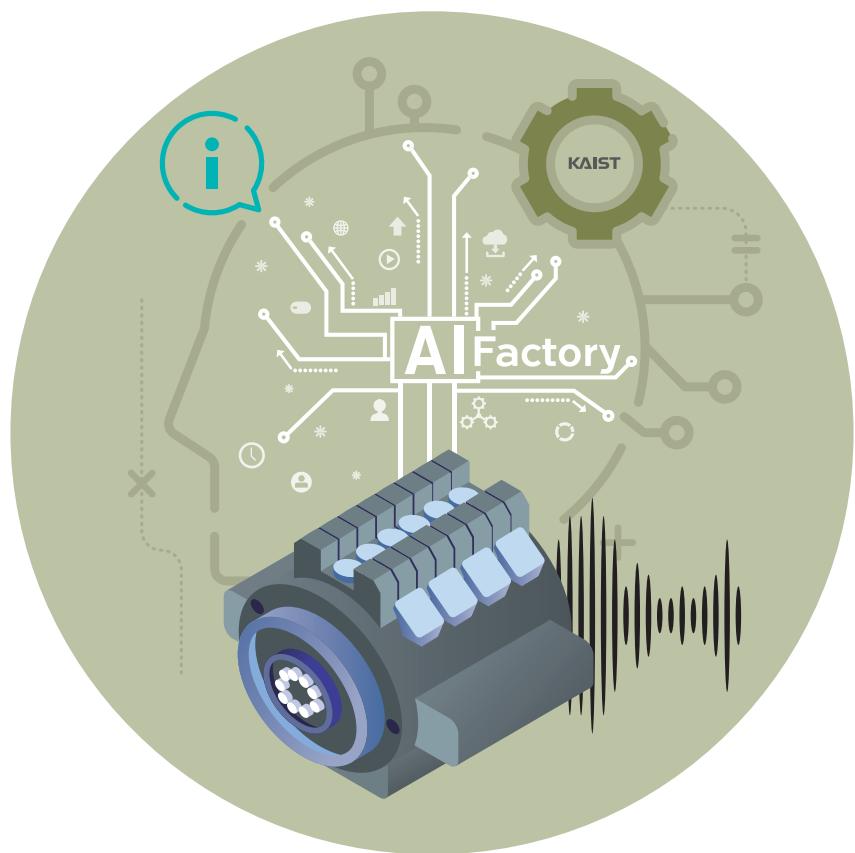


# 「Ford 엔진 진동 AI 데이터셋」 분석실습 가이드북

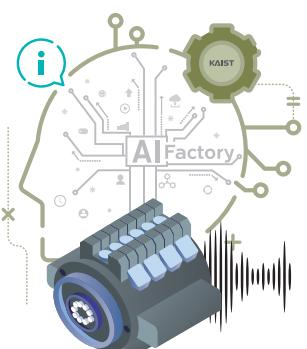


# 「Ford 엔진 진동 AI 데이터셋」 분석실습 가이드북



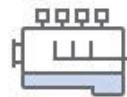
# Contents

<b>1 분석요약</b>	<b>04</b>
<b>2 분석 실습</b>	
<b>1. 분석 개요</b>	<b>05</b>
<b>1.1 분석 배경</b>	<b>05</b>
1) 공정(설비) 개요	
<b>1.2 분석 목표</b>	<b>05</b>
1) 분석목표	
2) 목표 수행을 위한 전략	
3) 제조 데이터 분석 기대효과	
4) 시사점(implication) 요약기술	
<b>2. 분석실습</b>	<b>06</b>
<b>2.1 제조데이터 소개</b>	<b>06</b>
1) 데이터 수집 방법	
2) 데이터 유형/구조	
<b>2.2 분석 모델 소개</b>	<b>09</b>
1) AI 분석모델	
<b>2.3 분석 체험</b>	<b>13</b>
1) 필요 SW, 패키지 설치 방법 및 절차 가이드	
2) 분석 실습	
[단계 ①] 필요 라이브러리 불러오기	
[단계 ②] 데이터 시각화	
[단계 ③] 데이터 특성 파악	
[단계 ④] 데이터 전처리	
[단계 ⑤] 모델 구축 및 설계	
[단계 ⑥] 모델 학습	
[단계 ⑦] 모델 평가	
[단계 ⑧] 결과 분석 및 해석	
<b>3. 유사 타 현장의 「Ford 엔진 진동 AI 데이터셋」 분석 적용</b>	<b>50</b>
<b>부 록</b>	
Appendix A. Anaconda를 활용한 가상환경(virtual env) 설정하기	<b>51</b>



# 「Ford 엔진 진동 AI 데이터셋」 분석실습 가이드북

- 필요 SW : Python
- 필요 패키지 : scikit-learn, xgboost, tensorflow, keras, pytorch
- 분석 환경 : [CPU] Intel(R) Xeon(R) Gold 6226 CPU @ 2.70GHz, [RAM] 32 GB,  
[GPU] NVIDIA Quadro RTX 6000s
- 필요 데이터 : FordA\_TRAIN.arff, FordA\_TEST.arff
- 주관 기관 : 한국과학기술원(KAIST)
- 수행 기관 : 한국과학기술원(KAIST) AI대학원 최재식교수



## 1 분석요약

No	구분	내용				
1	분석 목적 (현장 이슈, 목적)	<ul style="list-style-type: none"><li>- 미국 포드(Ford)사에서 제공한 오픈데이터셋을 사용하여 개발한 AI모델로 시스템/설비 예지 보전을 위해, 자동차 시스템 시계열 데이터셋에 대해 분류 알고리즘을 적용하여 불량 제품을 분류하고자 한다.</li><li>- 제조업에서 분석 수요가 가장 높은 시계열 데이터를 사용하여 계측 Sensor 값의 주기적인 특성 및 계측 Sensor 간의 관계 학습을 통한 분류 및 예측 문제를 해결한다.</li></ul>				
2	데이터셋 형태 및 수집방법	<ul style="list-style-type: none"><li>- 분석에 사용된 변수 : Sensor 1~500</li><li>- 데이터 수집 방법 : Ford Classification Challenge에서 제공한 오픈 데이터셋 (<a href="http://www.timeseriesclassification.com/description.php?Dataset=FordA">http://www.timeseriesclassification.com/description.php?Dataset=FordA</a>)</li><li>- 데이터셋 파일 확장자 : ARFF, text</li></ul>				
3	데이터 개수 데이터셋 총량	<ul style="list-style-type: none"><li>- 데이터 개수 : 4,921개</li><li>- 데이터셋 총량 : 30MB</li></ul>				
4	분석적용 알고리즘	<table border="1"><tr><td>알고리즘</td><td>로지스틱 회귀, XGBoost, 합성곱신경망(CNN), 순환신경망(RNN)</td></tr><tr><td>알고리즘 간략소개</td><td><ul style="list-style-type: none"><li>- 로지스틱회귀(Logistic Regression)은 데이터가 어떤 카테고리에 속할지를 0과 1사이의 연속적인 확률로 예측하는 회귀알고리즘 중 하나이다.</li><li>- XGBoost는 Gradient Boosting Algorithm(GBM) 기반의 알고리즘으로, 여러 개의 Decision Tree를 조합해서 사용하는 앙상블(ensemble) 기법으로 이 중 부스팅 계열에 속하는 알고리즘이다.</li><li>- 합성곱신경망(Convolutional Neural Network)은 시각적인 데이터를 분석하는데 사용되는 다중 순방향 신경망의 한 종류이다.</li><li>- 순환신경망(Recurrent Neural Network)은 인공신경망의 한 종류로 유닛 간의 연결이 순환적으로 구조를 갖는 특징을 가지고 있다.</li></ul></td></tr></table>	알고리즘	로지스틱 회귀, XGBoost, 합성곱신경망(CNN), 순환신경망(RNN)	알고리즘 간략소개	<ul style="list-style-type: none"><li>- 로지스틱회귀(Logistic Regression)은 데이터가 어떤 카테고리에 속할지를 0과 1사이의 연속적인 확률로 예측하는 회귀알고리즘 중 하나이다.</li><li>- XGBoost는 Gradient Boosting Algorithm(GBM) 기반의 알고리즘으로, 여러 개의 Decision Tree를 조합해서 사용하는 앙상블(ensemble) 기법으로 이 중 부스팅 계열에 속하는 알고리즘이다.</li><li>- 합성곱신경망(Convolutional Neural Network)은 시각적인 데이터를 분석하는데 사용되는 다중 순방향 신경망의 한 종류이다.</li><li>- 순환신경망(Recurrent Neural Network)은 인공신경망의 한 종류로 유닛 간의 연결이 순환적으로 구조를 갖는 특징을 가지고 있다.</li></ul>
알고리즘	로지스틱 회귀, XGBoost, 합성곱신경망(CNN), 순환신경망(RNN)					
알고리즘 간략소개	<ul style="list-style-type: none"><li>- 로지스틱회귀(Logistic Regression)은 데이터가 어떤 카테고리에 속할지를 0과 1사이의 연속적인 확률로 예측하는 회귀알고리즘 중 하나이다.</li><li>- XGBoost는 Gradient Boosting Algorithm(GBM) 기반의 알고리즘으로, 여러 개의 Decision Tree를 조합해서 사용하는 앙상블(ensemble) 기법으로 이 중 부스팅 계열에 속하는 알고리즘이다.</li><li>- 합성곱신경망(Convolutional Neural Network)은 시각적인 데이터를 분석하는데 사용되는 다중 순방향 신경망의 한 종류이다.</li><li>- 순환신경망(Recurrent Neural Network)은 인공신경망의 한 종류로 유닛 간의 연결이 순환적으로 구조를 갖는 특징을 가지고 있다.</li></ul>					
5	분석결과 및 시사점	<ul style="list-style-type: none"><li>- Sensor로부터 얻은 자동차 시스템의 주요 데이터로 시간에 따른 정상/비정상 값의 특성을 분석하고, 가공된 데이터를 학습하여 AI 모델을 개발하여 불량 제품을 분류한다.</li><li>- 개발된 AI 모델은 유사한 공정에 적용될 수 있으며, 정상/비정상 상태에 대해 정확한 Labeling 작업을 통하여 보다 정확한 분석이 가능할 것이라고 예상된다.</li></ul>				

## 2 분석 실습

### 1. 분석 개요

#### 1.1 분석 배경

##### 1) 공정(설비) 개요

###### • 공정(설비) 정의 및 특징

- 자동차 시스템이며, 자동차 하위 시스템인 엔진의 (정상 / 비정상) 상태와 연관이 있을 수 있는 주변의 500개 센서들로 여러 요소(i.e. 소음, 압력, 진동, 온도 등)들을 측정한다.

###### • AI 기반 예측 모델 개발 배경

- ① 무엇보다도 제조업 현장에서 분석 수요가 높은 시스템/설비 정상/비정상 상태 분류 그리고 예지 보전(Predictive Maintenance)을 위해, 현업의 인공지능에 대한 사전 지식이 많지 않은 초심자도 활용할 수 있는 튜토리얼을 제공하고자 한다.
- ② 제조업 현장에서 가장 쉽고 흔히 획득할 수 있는 센서 계측 값으로 이루어진 데이터셋을 활용하여 초심자들의 이해도와 접근성을 높이고, 실제로 현업에서 보유한 데이터의 형태만 간단하게 변경하여 그대로 기계학습 모델을 적용하여 예측/분류 결과물을 얻는 경험을 해볼 수 있도록 하였다.
- ③ 자동차 하위 시스템인 엔진의 이상 여부를 분류하는 문제 해결을 통해, 제조업 현장 내에서의 각종 시스템(i.e. 운영 설비, 운영 시스템 등)으로 확장/응용하여 해결하고자 하는 문제에 대한 접근 방법을 쉽게 제시하고자 한다. 이로써 제조업 현업 전문가들이 각자 보유하고 있는 데이터에 대하여 본 인공지능 튜토리얼을 쉽게 적용하여, 데이터 기반의 의사 결정을 통한 현업의 생산력 증진, 정비 능력 향상에 기여하고자 한다.

#### 1.2 분석 목표

##### 1) 분석목표

- 제조업 현장에서 흔히 접할 수 있는 데이터셋, 즉 설비 내의 여러 개의 계측 센서가 시간 순서대로 측정한 값으로 이루어진 (행: 시간, 열: 센서의 종류) 시계열 데이터(Time Series Data)에 대해, 설비의 정상/비정상을 판단하는 분류 문제(Classification Task)를 해결하고자 한다.

## 2) 목표 수행을 위한 전략

- 도메인 지식에 기반한 변수 선택 및 추출 과정(Feature Selection / Engineering) 없이, 딥러닝 모델을 활용하여 모델이 직접 모니터링 센서 간의 관계성이나 로컬한 특징과 같은 데이터에 내재된 특징을 직접 추출하고 이를 통해 분류할 수 있는 분석 프레임워크를 구축하고자 한다.

## 3) 데이터 분석 기대효과

- 현업의 실무자가 설비의 정상/비정상 상태에 직접적으로 영향을 미치는 변수에 대한 사전 도메인 지식이 없는 일반적인 상황으로 가정하였다. 즉, 도메인 지식이 없는 분석 실무자도 기계학습 기반의 정상/비정상 분류 과제를 수행할 수 있도록 한다.

## 4) 시사점(implication) 요약기술

- 제조업 현장에서 분석 수요가 높은 2가지 문제에 대해, 담당자들 각자 필드 내 보유 데이터에 분류 또는 예측 모형을 쉽게 구축하여, AI 기반으로 정상 / 비정상을 분류할 수 있을 뿐만 아니라 타겟 값 예측이 가능함. 따라서 본 튜토리얼은 분석 인프라나 역량이 다소 부족한 중소기업에서도 쉽게 AI 기반 예측/분류를 수행해볼 수 있도록 기여했다는 점에서 시사점이 크다고 판단한다.

# 2. 분석실습

## 2.1 제조데이터 소개

### 1) 데이터 수집 방법

- **제조(설비) 분야** : 자동차 운영 시스템
- **제조 공정(설비)명**: 자동차 운영 체계 내 하위 시스템(엔진)의 이상 여부를 판단하기 위해 이와 관련된 500개의 센서로부터 수집된 계측 데이터로 구성되어 있는데, 이는 일종의 시스템 운영 데이터라고 할 수 있다.
- **수집장비** : 자동차 내 계측 센서
- **데이터 출처** : 미국 포드(Ford)社에서 주최한 기계학습 경연 대회인 Ford Classification Challenge에서 제공한 오픈 데이터셋으로, 누구나 접근이 가능하며, 기존 출처는 <http://home.comcast.net/~nn-classification>이었으나, 현재는 아래의 웹페이지에서 다운로드 받을 수 있다.

<http://www.timeseriesclassification.com/description.php?Dataset=FordA>

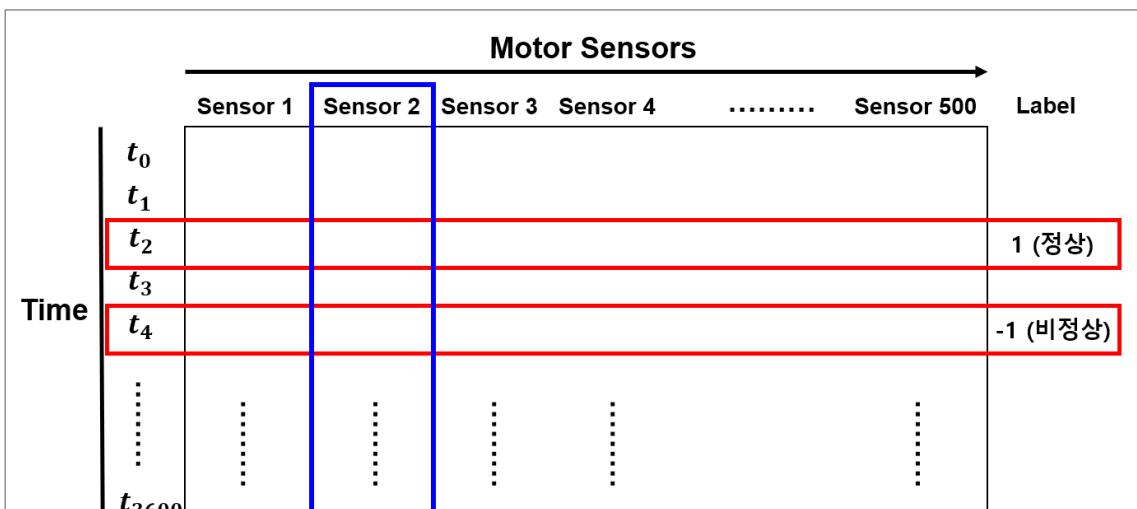
## 2) 데이터 유형/구조

### • 데이터셋 구조

- 총 4,921개의 시계열 구조로, 학습용(Training) 3,601개, 테스트용(Test) 1,320개로 구성되어 있으며, 학습용, 테스트용 데이터는 각각의 별도 파일로 제공된다. 실제 데이터 파일의 데이터 값 구성은 Figure 1과 같으며, 학습용(Training) 데이터셋은 Figure 2와 같은 구조를 지니고 있다. 테스트용(Test) 데이터셋도 학습용 데이터셋과 동일한 구조를 갖고 있으며, Figure 2과 같다.

time	sensor 1	sensor 2	sensor 3	sensor 4	sensor 5	sensor 6	sensor 7	sensor 8	sensor 9	sensor 10	sensor 11	sensor 12	sensor 13	sensor 14	sensor 15	sensor 16	sensor 17	sensor 18
t_0	-0.79717	-0.66439	-0.37301	0.040815	0.526936	0.984288	1.35312	1.578108	1.659251	1.640809	1.55229	1.437952	1.279354	1.069119	0.744547	0.277605	-0.30072	-0.93733
t_1	0.804855	0.634629	0.373474	0.038343	-0.34099	-0.74086	-1.10967	-1.39536	-1.57019	-1.61995	-1.54571	-1.36789	-1.12	-0.82855	-0.50623	-0.16259	0.18541	0.523795
t_2	0.727985	0.111284	-0.49912	-1.06863	-1.57835	-1.99053	-2.30203	-2.5034	-2.58521	-2.5506	-2.40901	-2.17932	-1.88356	-1.54374	-1.17875	-0.81691	-0.46451	-0.12124
t_3	-0.23444	-0.50216	-0.73249	-0.94613	-1.13974	-1.32334	-1.49024	-1.60708	-1.62043	-1.50693	-1.24656	-0.85266	-0.35862	0.165802	0.059512	1.07344	1.363857	1.524087
t_4	-0.17133	-0.6229	0.235829	0.710396	1.239969	1.649823	1.876321	1.865535	1.703751	1.466467	1.250755	1.056614	0.88359	0.68451	0.39371	-0.20142	-0.88803	-1.60743
t_5	-0.5409	-1.01402	-1.29823	-1.32083	-1.08225	-0.63712	-0.09037	0.462991	0.923214	1.218635	1.307273	1.191861	0.892437	0.446956	-0.10859	-0.72457	-1.32542	-1.82439
t_6	-0.33406	-1.00801	-1.55435	-1.92219	-2.08593	-2.0421	-1.8145	-1.44965	-1.00838	-0.53156	-0.0536	0.393866	0.779954	1.078232	1.264621	1.312305	1.209263	0.974747
t_7	1.04589	0.611195	0.153108	-0.27967	-0.65162	-0.92265	-1.0839	-1.13113	-1.08407	-0.96421	-0.80893	-0.65551	-0.52979	-0.44184	-0.38904	-0.3541	-0.31616	-0.24708
t_8	0.825565	0.385282	-0.06242	-0.48090	-0.83119	-1.08184	-1.2136	-1.23153	-1.1413	-0.9644	-0.73847	-0.51483	-0.34191	-0.24898	-0.24038	-0.28304	-0.33115	-0.39355
t_9	-0.28418	-0.19261	-0.03229	0.172823	0.394703	0.588146	0.704184	0.712422	0.593282	0.352499	0.019351	-0.35756	-0.73177	-0.15206	-1.28543	-1.41139	-1.41322	-1.28336
t_10	0.529562	0.695556	0.754557	0.688517	0.549053	0.391765	0.259718	0.210681	0.271933	0.405071	0.551154	0.658468	0.669671	0.573825	0.360457	0.095538	-0.16071	-0.36852
t_11	-1.07104	-1.10475	-1.0247	-0.81404	-0.5107	-0.16143	0.16804	0.466752	0.727967	0.955476	1.145068	1.279888	1.326233	1.258822	1.060805	0.711114	0.274633	-0.20777
t_12	-0.10945	-0.226	-0.30023	-0.33921	-0.36519	-0.38859	-0.45241	-0.52479	-0.59093	-0.65099	-0.65656	-0.59717	-0.45984	-0.26668	-0.0573	0.134412	0.273232	0.340043
t_13	-1.34854	-0.72549	-0.08562	0.464571	0.889827	1.163441	1.257545	1.203	1.074434	0.912903	0.718406	0.533799	0.362378	0.192934	0.04426	-0.07376	-0.17265	-0.24683
t_14	1.429452	1.079359	0.510714	-0.14623	-0.738	-1.1763	-1.36289	-1.27336	-0.96721	-0.52783	-0.06259	0.335178	0.596119	0.678625	0.607251	0.396696	0.111315	-0.18965
t_15	2.252085	2.157468	1.848938	1.397509	0.883034	0.373998	-0.05951	-0.36972	-0.54819	-0.61199	-0.60716	-0.56299	-0.54825	-0.59072	-0.70785	-0.90189	-1.11863	-1.2963
t_16	0.453874	0.424807	0.399023	0.359174	0.276194	0.136019	0.05432	-0.2545	-0.42656	-0.55548	-0.61643	-0.61643	-0.60236	-0.61174	-0.67737	-0.79457	-0.903053	-1.04305
t_17	-0.50796	-0.80718	-0.8914	-0.69334	-0.2457	0.354373	0.941005	1.349737	1.455348	1.187728	0.61337	-0.13392	-0.86692	-1.41695	-1.67133	-1.60209	-1.26786	-0.80123
t_18	1.66411	1.523809	1.318033	1.056137	0.756827	0.435069	0.073466	-0.31882	-0.73317	-1.14192	-1.47864	-1.71248	-1.78731	-1.69377	-1.44123	-1.0858	-0.66677	-0.25896
t_19	-1.87799	-1.77482	-1.49303	-1.13564	-0.79199	-0.50127	-0.28683	-0.13769	0.036191	0.268495	0.533101	0.768154	0.878121	0.83001	0.644442	0.386022	0.115917	0.08113
t_20	-0.72855	-0.77435	-0.68783	-0.50999	-0.29095	-0.07222	0.123761	0.292403	0.444331	0.655564	0.658707	0.705231	0.684755	0.5885	0.428243	0.238543	0.058321	-0.08111
t_21	1.284971	1.1534147	1.511495	1.247218	0.824374	0.420408	0.113846	-0.04427	-0.042	0.05125	0.157641	0.212762	0.215027	0.152355	0.070052	0.018707	-0.01628	-0.06624
t_22	-0.10518	0.006417	0.031866	-0.01689	-0.12747	-0.25656	-0.37252	-0.45501	-0.48212	-0.44915	-0.35232	-0.19356	0.01832	0.273621	0.527584	0.751632	0.902251	0.948191
t_23	-1.29437	-0.70589	0.075456	0.78262	1.199254	1.240025	0.944576	0.424093	-0.06289	-0.37456	-0.36961	-0.10563	0.253484	0.531651	0.576158	0.361042	-0.04137	-0.48212
t_24	-1.38308	-1.31787	-1.04129	-0.55196	0.094424	0.812345	1.493239	2.023207	2.310232	2.320263	2.00312	1.45781	0.746778	-0.03318	-0.79575	-1.46556	-1.97861	-2.29624
t_25	-0.69778	-0.62791	-0.51319	-0.42872	-0.42246	-0.48399	-0.58932	-0.69987	-0.7906	-0.83649	-0.79477	-0.68736	-0.505073	-0.3943	-0.30357	-0.26185	-0.20345	-0.12523
t_26	-1.41251	-1.224	-0.9755	-0.66874	-0.32599	0.032188	0.381795	0.708267	1.003891	1.252387	1.4409	1.560864	1.595139	1.526588	1.355212	1.08101	0.722834	0.305533
t_27	0.813045	1.171026	1.460458	1.655952	1.737196	1.68388	1.496003	1.193876	0.815583	0.399208	-0.02103	-0.40054	-0.71536	-0.97179	-1.15966	-1.29169	-1.38816	-1.4491

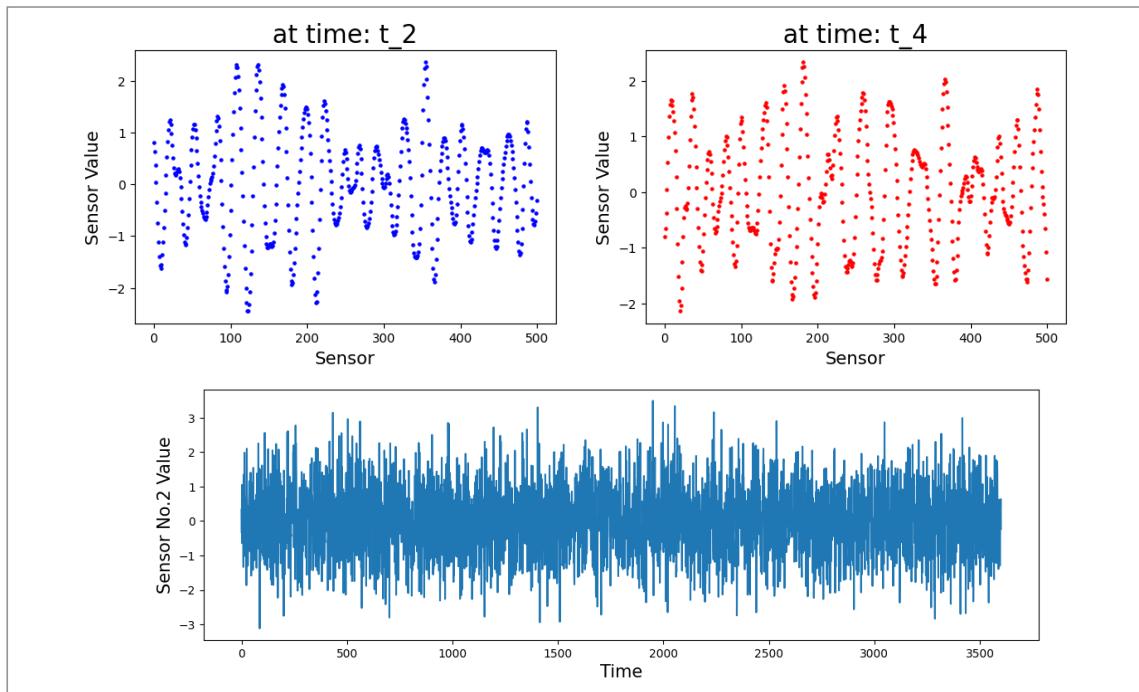
[Figure 1] 데이터 파일 형태 (스크린샷)



[Figure 2] 학습용 데이터셋 전체 구조. (테스트용 데이터셋과 동일)

- Figure 3.에서 볼 수 있듯이, 각각의 시계열 샘플(1개의 행)는 특정 시간대( $t_n$ )에서 500개의 자동차 시스템 내 센서로 측정한 계측 값들로 구성되어 있다. 또한 각 컬럼(1개의 열)은 특정 센서(500개 중 1개 센서)의 계측 값들의 시계열로 구성되어 있다.

Figure 3은  $t=2$ 일 때의 센서들의 값과  $t=4$ 일 때의 센서들의 값, 그리고 센서 2가 측정한 값들의 시계열을 시각화한 것이다. 이때  $t=2$ 일 때의 엔진의 상태 Label은 정상(1)이며,  $t=4$ 일 때의 엔진의 상태 Label은 비정상(-1)이다.



[Figure 3] 정상데이터 샘플(시간=  $t_2$ ), 비정상데이터 샘플,(시간 =  $t_4$ ) 그리고 특정 센서(Sensor No. 2)의 시계열 샘플 시각화

#### • 칼럼 수 : 501개

독립 변수/종속 변수 정의

구분	명칭
독립변수	Sensor 1~500번 (총 500개)
종속변수	정상/비정상 Class (총 1개) (Class: 정상/비정상으로 1 또는 -1)

#### • 데이터 개수 : 4,921개

분류	설명
학습용(Training) 데이터	3,601개
테스트용(Test) 데이터	1,320개

#### • 데이터 속성 정의

속성 명칭	설명	데이터 타입
Sensor 1~500	센서명 (*각 센서가 어떤 정보를 수집하는 지에 대해서는 데이터 제공처에서 따로 정보를 제공하지 않았음)	float64
라벨(Label)	정상/비정상 Class 정상 : 1 비정상 : -1	int64
Time	시간	object

독립 변수란 다른 변수에 영향을 받지 않는 변수로, 입력 값을 나타내며, 종속 변수는 독립 변수의 변화에 따른 최종 목표 값이다.

#### • 주요 변수

- 본 데이터셋의 경우 주요 변수에 대한 사전 정보가 제공되어 있지 않아, 필요한 경우 분석 담당자가 여러 가지 방법을 통해 직접 탐색해야 한다. 단, 현업에서 실제 주요 변수에 대한 사전지식이 없는 경우가 일반적이기에, 본 튜토리얼을 통해 어떠한 변수가 영향력 있는지에 대한 사전지식이 주어지지 않는 상황에서, 모델이 스스로 데이터에 내재된 주요 특징을 추출하고 예측하는 방법론을 제시하고자 한다.

#### • 데이터셋 선정 이유

- 중소기업중앙회는 빅데이터 기반 중소제조업 혁신 관련 정책 개발을 위해 지난 2020년 6월 1일부터 4일까지 전국 259개의 상생형 스마트공장 구축기업을 대상으로 실시한 「스마트공장 제조 데이터 활용 실태 및 분석 수요 조사」 결과를 발표하였는데, 조사 결과 스마트공장 내 제조 데이터 수집 비중은 ▲각종 센서 장비/시스템을 통한 자동 수집 43.3%으로 가장 높았다. 이처럼 현장에서는 센서 장비/시스템으로 수집한 데이터를 가장 흔하게 얻을 수 있으며, 이에 대한 분석 수요가 가장 높을 것으로 예상할 수 있다. 따라서, 본 가이드북은 센서 장비/시스템을 통해 수집한 데이터 분석 방법을 제공하고자 한다. 제공되는 데이터셋은 제조업에서 흔히 얻을 수 있는 센서들의 계측 값으로 구성된 시계열 데이터의 형태(X축: 센서, Y축: 시간)를 갖고 있어, 제조업 현장 담당자들에게 상당히 친숙한 데이터 형태로 볼수있다. 또한 간단한 형태를 지니고 있어, 현업의 초심자들도 데이터 자체가 의미하는 바와, 데이터 로드(Load)부터 예측까지의 과정 중 데이터의 형태 변형을 이해하기에도 가장 용이하다고 판단하였다.

## 2.2 분석 모델 소개

### 1) AI 분석모델

#### • 해당 AI 방법론(알고리즘) 선정 이유 기술

- 지도학습(Supervised Learning) 기반의 알고리즘

지도학습(Supervised Learning) 기반으로 한 “분류(Classification)” 알고리즘을 이용하여, 제조 현장의 문제를 해결하고자 한다.

“분류”를 이용하여 문제를 해결하고자 하는 경우, 입력에 대해 정상 class(1)일 확률과 비정상 class(-1)일 확률을 계산하여, 확률이 높은 class로 예측하는 과정이다. (물론 Class의 수가 꼭 2개일 필요는 없다. 정상/비정상과 같이 Class가 2개인 경우 Binary

Classification, 3개 이상인 경우 Multi-Class Classification이라고 한다.)

지도학습은 “분류”와 “회귀”가 대표적인데, “분류”와 “회귀”와의 차이점은 다음과 같다. 우선 “회귀”를 이용하여 문제를 해결하고자 하는 경우, 앞서 언급한 “분류”처럼 특정 클래스(Class)에 속할 확률을 계산하는 것이 아니라, 종속 변수의 미래의 값을 예측하는 것이다. 즉, 회귀로 나오는 출력은 연속적인 실수 값으로써, 연속성이 있고 그 연속성 중에 어디에 점을 찍을지 결정하는 문제이다. 즉, 회귀는 주어진 데이터의 특징에 따라 연속적인 종속 변수(target) 값을 추정하는 방법 (eg. 집값 추정, 주가 예측) 이었다면, 분류는 주어진 데이터의 특징에 따라 데이터를 연속성이 없는(Discrete) 특정 클래스(eg. 정상/비정상)로 분류하는 방법이다.

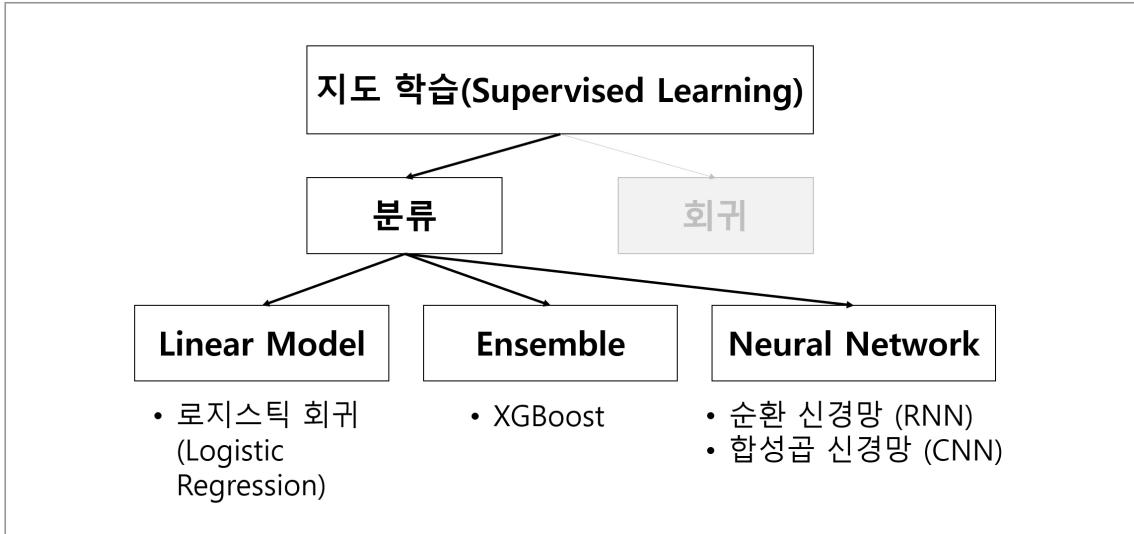
제조 현장 내 운영 시스템의 정상/비정상 상태를 판단한다든지 또는 정상/불량 물품을 분류하기 위해서는 대표적인 분류 모델을 활용하는 것이 가장 적합하기 때문이다.

#### • 적용하고자 하는 AI 분석 방법론(알고리즘)의 구체적 소개

##### - 지도학습 기반의 “분류”란?

우선 지도 학습 (Supervised Learning)은 훈련 데이터(Training Data)로부터 하나의 함수를 유추해내기 위한 기계 학습(Machine Learning)의 한 방법이다. 훈련 데이터는 일반적으로 입력 객체에 대한 속성을 벡터 형태로 포함하고 있으며 각각의 벡터에 대해 원하는 결과(Label)가 무엇인지 표시되어 있다. 이렇게 유추된 함수로 연속적인 값을 출력하는 것을 회귀분석(Regression)이라 하고, 주어진 입력 벡터가 어떤 클래스에 속하는지 표시하는 것을 분류(Classification)라 한다. 제조업 현장에서 “분류” 알고리즘을 활용하면, 공정 내 설비의 운영 상태가 정상인지 비정상인지, 혹은 최종 제작된 제품의 품질이 정상인지 아닌지를 예측할 수 있다. 만약 공정 내 설비가 정상으로 예측된 확률값이 0.5 보다 크다면, 상태를 정상으로 분류하고, 0.5보다 작으면 비정상으로 분류하는 방식이다. 여기서 정상 상태는 일반적으로 라벨(label)을 1로 설정하며, 비정상 상태는 라벨(label)을 0 또는 -1 으로 설정한다. 이렇게 두 가지 클래스 중 데이터가 어떤 클래스에 속할지 결정하는 일을 이진 분류(binary classification)라고 한다. 여기서 클래스의 종류가 3가지 이상인 경우, 예를 들어 동물의 사진을 보고 개/고양이/기타 등으로 분류하는 문제를 때에는, 다중 클래스 분류 (Multi-Class Classification)이라고 한다.

- 기계학습을 이용한 분류와 회귀 알고리즘



### (1) 로지스틱 회귀(Logistic Regression)

로지스틱 회귀(Logistic Regression)는 데이터가 어떤 카테고리에 속할지를 0과 1 사이의 연속적인 확률로 예측하는 회귀 알고리즘 중 하나이다. 그리고 예측된 확률에 기반하여 특정 데이터가 어떤 카테고리에 속할지를 결정하게 되고, 궁극적으로 목표로 하는 분류 문제를 풀게 된다. 선형 회귀(Linear Regression) 역시 마찬가지로 분류(Classification) 문제에 적용할 수 있다. 하지만 입력값이 매우 크거나 매우 작을 경우, 분류 기준 점(Classification Threshold)이 크게 바뀌어 잘못된 분류 결과를 나타내거나, 또는 출력이 음수로 나올 수 있다. 반면 로지스틱 회귀(Logistic Regression)의 출력값은 0과 1 사이의 확률 값이기 때문에, 위와 같은 선형 회귀(Linear Regression)를 통한 분류 문제 해결의 한계점을 극복할 수 있다. 이와 같은 로지스틱 회귀(Logistic Regression)은 다음과 같은 과정으로 수행된다.

- ① 모든 계수(coefficient) 값들과 절편(intercept) 값을 0으로 초기화.
- ② 각각의 속성(feature)을 이에 상응하는 계수 값(coefficient)과 곱하여 모두 더하고, 여기에 절편(intercept)도 더하여 로그 오즈(log-odds)를 계산.
- ③ 계산한 log-odds 값을 로지스틱(Logistic 또는 Sigmoid) 함수에 전달하여 0~1 사이의 확률값을 구함.
- ④ 계산한 확률값과 실제의 라벨(Label) 값을 비교하여 손실 값(Loss)를 계산하고, 경사 하강법(Gradient Descent)을 통해 최적화된 계수(coefficient)와 절편(intercept) 값을 계산.
- ⑤ 최적화된 파라미터(계수, 절편)를 구했다면, 분류 기준(Classification Threshold) 값을 조절하여, 정상과 비정상을 나눌 분류 경계(Decision Boundary)를 설정.

### (2) XGBoost

뛰어난 예측 성능, 빠른 수행 시간으로 효율적인 알고리즘으로, 결과에 영향력 있는

중요 특징(Feature)을 확인할 수 있기에 현업에 종사자들에게 유용한 인사이트를 제공 가능한 모델이다. XGBoost는 앙상블(Ensemble) 방법을 기반으로 하는데, 앙상블(Ensemble)이란 여러 가지의 모델로부터 나온 결과를 조합해서 더 나은 결과를 얻어내는 방법이다. 정확도가 높은 강한 모델을 하나 사용하는 것보다, 정확도가 낮은 약한 모델을 여러 개 조합하는 방식이 정확도가 높다는 가정하에 시작된 기법인데, 앙상블(Ensemble)은 방식에 따라서 배깅(Bagging)과 부스팅(Boosting) 방법으로 분류된다. 여기서 XGBoost는 이름에서 유추할 수 있듯이 부스팅(Boosting) 기법을 이용하여 구현한 그래디언트 부스팅(Gradient Boost) 알고리즘을 기반으로 하며, 이때 의사결정나무(Decision Tree)를 조합해서 사용한다.

### (3) 순환 신경망(Recurrent Neural Network)

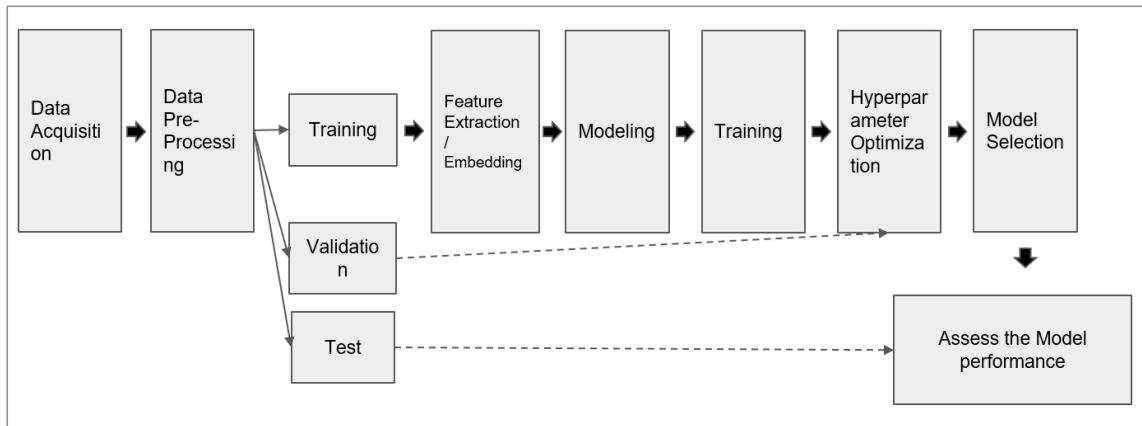
순환 신경망(Recurrent Neural Network, RNN)은 인공 신경망의 한 종류로, 유닛 간의 연결이 순환적 구조를 갖는 특징을 갖고 있다. 즉, 입력층에서 출력층까지 한 방향만으로 흐르는 순방향(Feed Forward) 신경망에 비해, 순환 신경망은 순방향 신경망과 비슷하지만, 출력 결과를 다시 입력부로 받는 기능이 있는 것이 주요 특징이다. 이러한 구조는 시간에 따라 변하는 데이터의 동적(Dynamic) 특징을 학습할 수 있도록 신경망 내부에 상태(state)를 저장할 수 있게 해주므로, 순방향 신경망과 달리 내부의 메모리를 이용해 시퀀스(sequence) 형태의 입력을 처리할 수 있다. 순환 신경망은 시퀀스(sequence) 데이터를 모델링 하기 위해 등장한 모델로써, 자연어(Natural Language)나 음성신호, 시계열 데이터와 같은 연속적인(sequential) 특징이 내재된 데이터에 아주 적합한 모델이라고 할 수 있다.

### (4) 합성곱 신경망(Convolutional Neural Network)

합성곱 신경망(Convolutional Neural Network, CNN)은 대표적으로 시각적인 데이터(e.g. 사진, 영상)을 분석하는 데 사용되는 다층의 순방향(Feed Forward) 신경망의 한 종류이다. 딥 러닝에서 심층 신경망으로 분류되며, 시각적 영상 분석에 주로 적용되는 것으로 알려져 있으나 자연어 처리 및 시공간(Spatial-Temporal) 데이터 분석에도 응용된다. 본 과제는 이미지 데이터를 활용하지 않지만, 데이터의 시간적인(Temporal Feature) 특성과 공간적인(Spatial) 특성을 스스로 추출하고 이를 기반으로 예측을 수행하는 것이 목표이기 때문에, 합성곱 신경망이 적합하다고 판단하여 활용하였다. 특히 본 과제에서 활용한 Fully Connected Network는 시멘틱 분류(Semantic Segmentation) 모델을 위해 기존에 이미지 분류에서 우수한 성능을 보인 합성곱 신경망 기반의 모델(e.g. AlexNet, VGG16, GoogLeNet)을 목적에 맞춰 변형시킨 것이다.

- AI 분석 방법론(알고리즘) 구축 절차 설명

- 아래 Figure 4와 같은 기계학습 기반의 분류/예측 프레임워크에 맞춰 진행하되, 본 가이드북에서 사용하는 모델인, 로지스틱 회귀 / XGBoost / 순환 신경망(RNN) / 합성곱 신경망(CNN)에 따라 “Modeling” 부분만 달라지게 된다. 즉 데이터 전처리 과정은 동일하며, 모델링에서 사용되는 모델만 다르다.



[Figure 4] 기계학습 기반의 분류/예측 프레임워크

## 2.3 분석 체험

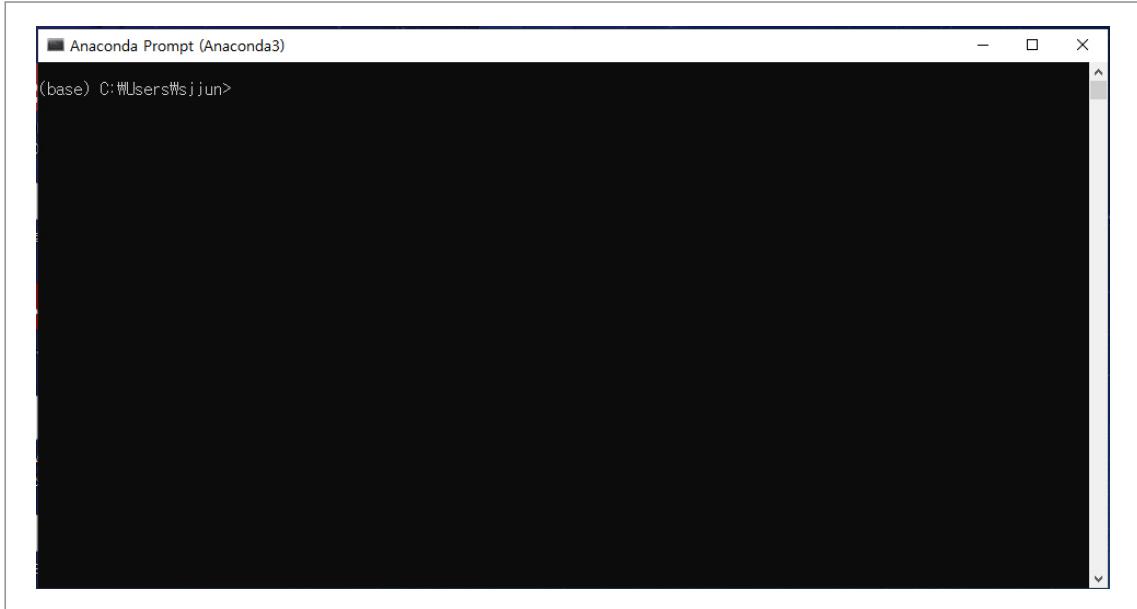
### 1) 필요 SW, 패키지 설치 방법 및 절차 가이드

- 필요 SW

- Anaconda 설치  
Appendix A 참조.

- 필요 패키지 설치 방법

① pip 명령어를 이용해서 패키지(또는 라이브러리) 설치하기: pip는 Python Package Index의 약자로 파이썬 패키지를 설치하고 관리하는 프로그램이다. 맥에서는 터미널, 윈도우에서는 명령프롬프트를 이용하면 되는데, “Anaconda”를 설치했으므로, “Anaconda Prompt”에서 설치를 진행한다. 우선 pip를 사용하기 전에 최신 버전으로 업그레이드 해주는 것이 좋다. 먼저 윈도우에서 “Anaconda Prompt”를 열기 후에 또는 아래와 같은 터미널 창에 명령어를 입력한다.



[Figure 5] Anaconda Prompt 창

```
# Requires the latest pip  
pip install --upgrade pip
```

이제 원하는 패키지를 설치하기 위해, “`pip install <패키지명>`”을 입력하면 원하는 라이브러리가 포함된 패키지를 설치할 수 있다.

\* 패키지를 설치할 수 있는 다른 방법

사용하고자 하는 패키지가 설치되어 있지 않을 경우, Jupyter Notebook Cell 상에서 “`!pip install <패키지명>`”을 실행한다. 이미 설치가 되어 있는 경우 “Requirement already satisfied”라는 문구가 뜬다.

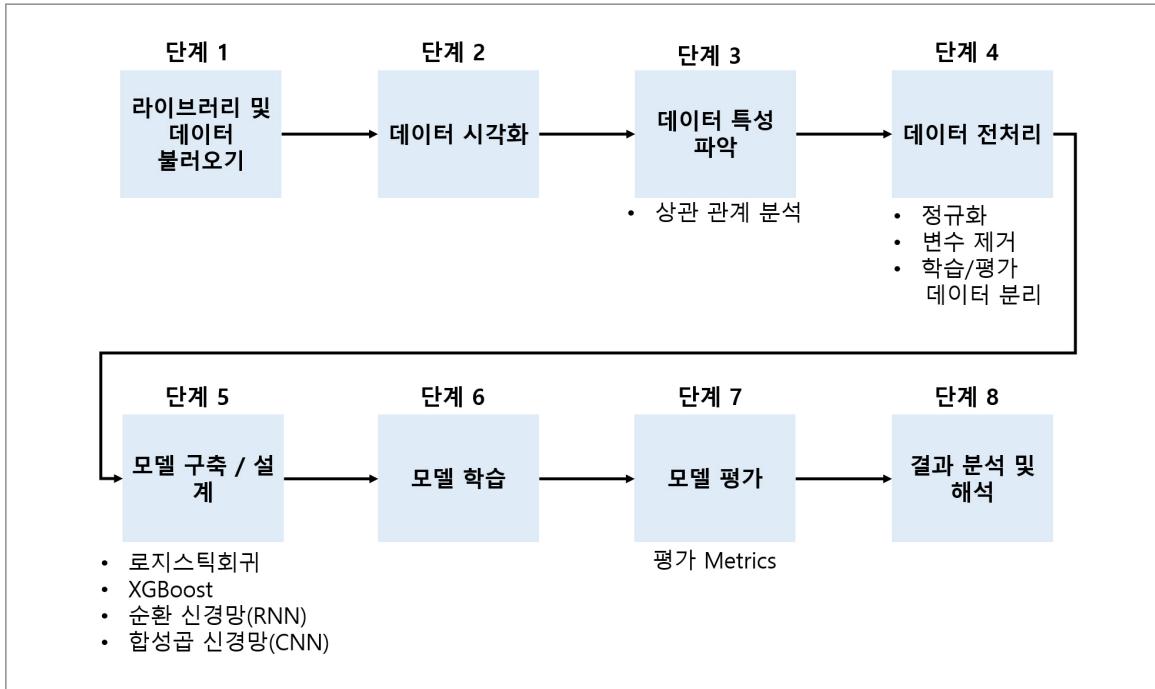
## ② 딥러닝 프레임워크 설치

딥러닝 프레임워크로 흔히 텐서플로우(TensorFlow) 또는 파이토치(PyTorch)를 사용한다. 이를 설치할 경우, 단순히 위에서 설명한 바와 같이 “`conda(pip) install tensorflow`” 명령어로 설치해도 되지만, 텐서플로우(TensorFlow) 또는 파이토치(PyTorch)를 설치하고자 하는 경우 특별히 주의를 기울일 필요가 있다. 아래 공식 웹사이트에 설치 방법이 자세히 나와 있으니 자세히 읽어보고 설치해야 향후 환경적인 문제 또는 버전 간 호환성 문제로 인한 Error를 방지할 수 있다.

텐서플로우(TensorFlow): <https://www.tensorflow.org/install?hl=ko>

파이토치(PyTorch): <https://pytorch.kr/>

## • 분석 체험 Flow Chart



[Figure 6] 분석 체험 Flow Chart

## 2) 분석 실습

### [단계 ①] 라이브러리 및 데이터 불러오기

#### ①-1. 필요 라이브러리 불러오기

```

"""
불러오고자 하는 라이브러리의 이름이 길거나, 자주 쓰는 라이브러리의 경우, 대개 약자를 쓴다.
** 흔히 통용되는 약어 **
pandas -> pd
numpy -> np
matplotlib.pyplot -> plt
Tensorflow -> tf
"""

import itertools
from time import time
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from scipy.io.arff import loadarff
from sklearn.preprocessing import StandardScaler, RobustScaler
import tensorflow as tf

```

[Figure 7] 필요 라이브러리를 불러오는 코드

불러오고자 하는 라이브러리의 이름이 길거나, 빈번하게 사용하게 될 라이브러리의 경우 약자를 쓰는 게 편하기 때문에 아래와 같이 “import <필요 라이브러리 명> as <사용자 임의의 명>” 약자로 정의 한다. 약자는 사용자 임의로 지정이 가능하나, 향후 다른 인원의 코드 재사용을 위해 가급적이면 흔히 통용되는 약자를 사용한다.

## - 데이터 다운로드 방법

본 가이드북에서 활용할 데이터는 Ford Dataset으로, 원래의 출처는 <http://home.comcast.net/~nn-classification>이었으나, 현재는 아래의 웹페이지에서 얻을 수 있다.

<http://www.timeseriesclassification.com/description.php?Dataset=FordA>

데이터를 얻어서 Load 하는 방법은 2가지로 설명하고자 한다.

### (1) 직접 다운로드하여 불러오는 방법

- ① 위 웹페이지에 접속하여, “Download this dataset” 클릭
- ② 데이터셋을 직접 다운로드하여, 로컬(Local) 내 지정 폴더에 저장한다.
- ③ 아래 코드를 실행한다.

```
file_path = './Data/' # 사용자 Local 환경 내의 다운로드 받은 데이터 파일이 위치한 경로
train_fn="FordA_TRAIN.arff" # Train 데이터 파일명
test_fn="FordA_TEST.arff" # Test 데이터 파일명

def read_ariff(path): # ariff 확장을 Load하기 위한 함수

    raw_data, meta = loadarff(path)
    cols = [x for x in meta]
    data2d = np.zeros([raw_data.shape[0],len(cols)])

    for I, col in zip(range(len(cols)),cols):
        data2d[:,i]=raw_data[col]
    return data2d

train = read_ariff(file_path + train_fn)
test = read_ariff(file_path + test_fn)
```

[Figure 8] 직접 다운로드하여 로컬 내에 폴더에 저장된 데이터를 불러오는 코드

## - 데이터를 제대로 불러왔는지 확인하기 위해, 아래의 코드를 작성하여 전체 데이터셋의 형태(shape)를 확인해보면 아래와 같다.

```
print("train_set.shape:", train.shape)
print("test_set.shape:", test.shape)
```

[Figure 9] 불러온 데이터의 형태(Shape)를 확인하는 코드

### ▶ 출력 결과

```
train_set.shape: (3601, 501)
test_set.shape: (1320, 501)
```

데이터셋이 저장되어 있는 Local 경로주소(변수 file\_path)에 파일명(“FordA\_TRAIN.arff”과 “FordA\_TEST.arff”)을 더한 데이터 경로를, 정의한 함수(read\_ariff)의 입력 값으로 넣어서 데이터를 불러온다. 불러온 학습용(Training) 데이터와 테스트용(Test) 데이터는 각각 train, test 변수에 저장되어 있다.

## (2) Data Source(웹페이지)로부터 직접 불러오는 방법

데이터를 직접 다운로드 받을 필요 없이, 데이터가 저장되어 있는 웹페이지에서 직접 데이터를 불러오는 방법으로 아래와 같은 코드를 작성하여 실행한다.

```
def read_file(file_path_url): # 입력: 파일 경로(url)
    """
    .tsv 확장자를 Load하기 위한 함수
    """
    return np.loadtxt(file_path_url, delimiter="\t")

root_url = "https://raw.githubusercontent.com/hfawaz/cd-diagram/master/FordA/" # url root

train = read_file(root_url + "FordA_TRAIN.tsv")
test = read_file(root_url + "FordA_TEST.tsv")
```

[Figure 10] 직접 다운로드하지 않고 웹 상에 저장된 데이터를 불러오는 코드

데이터를 제대로 불러왔는지 확인하기 위해, 아래의 코드를 작성하여 Dataset의 전체 Shape(형태)을 확인해보면 위의 방법 1의 데이터와 형태가 동일한 것을 확인할 수 있다.

```
print("train_set.shape:", train.shape)
print("test_set.shape:", test.shape)
```

[Figure 11] 불러온 데이터의 형태(Shape)를 확인하는 코드 (Figure 9와 동일)

### ▶ 출력 결과

```
train_set.shape: (3601, 501)
test_set.shape: (1320, 501)
```

### - 입력 변수(x), 타겟 변수(y) 분리하기

```
# 데이터 불러오기 방법 1 기준 데이터를 불러왔을 경우, 아래를 실행
x_train_temp = train[:, :-1]
y_train_temp = train[:, -1] # 마지막 column이 Label 값이 있는 column
x_test = test[:, :-1]
y_test = test[:, -1] # 마지막 column이 Label 값이 있는 column

# 데이터 불러오기 방법 2로 데이터를 불러왔을 경우, 아래의 주석을 해제
# x_train_temp = train[:, 1:]
# y_train_temp = train[:, 0] # 첫 번째 column이 Label 값이 있는 column
# x_test = test[:, 1:]
# y_test = test[:, 0] # 첫 번째 column이 Label 값이 있는 column
```

[Figure 12] 학습용, 테스트용 데이터셋을 각각 입력 변수(x)와 타겟 변수(y)로 분리하는 코드

방법 1로 데이터를 불러왔을 경우, 위의 코드를, 방법 2로 데이터를 불러왔을 경우 아래의 코드를 실행한다. 서로 다른 이유는, 방법 1로 얻은 데이터의 경우 정상/비정상 클래스 Label 속성이 있는 컬럼이 가장 마지막 열(column index = 501)임만 가장 마지막 열인 반면, 방법 2로 불러오는 데이터는 클래스 Label 속성이 있는 컬럼이 가장 첫 번째 열(column index = 1)이기 때문이다.

## - 학습용, 검증용, 테스트용 데이터셋 나누기

```
normal_x = x_train_temp[y_train_temp==1] # Train_x 데이터 중 정상 데이터
abnormal_x = x_train_temp[y_train_temp==0] # Train_x 데이터 중 비정상 데이터
normal_y = y_train_temp[y_train_temp==1] # Train_y 데이터 중 정상 데이터
abnormal_y = y_train_temp[y_train_temp==0] # Train_y 데이터 중 비정상 데이터

ind_x_normal =int(normal_x.shape[0]*0.8) # 정상 데이터를 8:2로 나누기 위한 기준 인덱스 설정
ind_y_normal =int(normal_y.shape[0]*0.8) # 정상 데이터를 8:2로 나누기 위한 기준 인덱스 설정
ind_x_abnormal =int(abnormal_x.shape[0]*0.8) # 비정상 데이터를 8:2로 나누기 위한 기준 인덱스 설정
ind_y_abnormal =int(abnormal_y.shape[0]*0.8) # 비정상 데이터를 8:2로 나누기 위한 기준 인덱스 설정

x_train = np.concatenate((normal_x[:ind_x_normal], abnormal_x[:ind_x_abnormal]), axis=0)
x_valid = np.concatenate((normal_x[ind_x_normal:], abnormal_x[ind_x_abnormal:]), axis=0)
y_train = np.concatenate((normal_y[:ind_y_normal], abnormal_y[:ind_y_abnormal]), axis=0)
y_valid = np.concatenate((normal_y[ind_y_normal:], abnormal_y[ind_y_abnormal:]), axis=0)
```

[Figure 13] 기존의 학습용 데이터셋을 학습용과 검증용으로 분리하는 코드

Test 데이터는 이미 나누어져 있어, 별도로 분리할 필요가 없다. 단, Train 데이터를 학습용(Training Data)과 검증용(Validation Data)으로 각 8:2의 비율로 나누었는데, 이때 정상과 비정상의 데이터의 비율을 동일하게 맞춰주기 위하여, 먼저 Train 데이터를 정상(normal\_x, normal\_y)과 비정상(abnormal\_x, abnormal\_y)의 데이터로 분리하고, 이들을 각각 8:2로 분리하여, 다시 통합(Concat)하는 방법으로 진행하였다.

데이터를 제대로 분리했는지 확인하기 위해, 아래의 코드를 작성하여 각 데이터셋의 형태(shape)를 확인해보면 아래와 같다.

```
print("x_train.shape:", x_train.shape)
print("x_valid.shape:", x_valid.shape)
print("y_train.shape:", y_train.shape)
print("y_valid.shape:", y_valid.shape)
print("x_test.shape:", x_test.shape)
print("y_test.shape:", y_test.shape)
```

[Figure 14] 분리된 학습용, 검증용, 테스트용 데이터셋의 형태를 출력하는 코드

## ▶ 출력 결과

```
↳ x_train.shape: (2880, 500)
    x_valid.shape: (721, 500)
    y_train.shape: (2880,)
    y_valid.shape: (721,)
    x_test.shape: (1320, 500)
    y_test.shape: (1320,)
```

## [단계 ②] 데이터 시각화

### - 시각화 I : 데이터 불균형(Data Imbalance) 확인

```
# Class의 종류 확인: 정상 1, 비정상 -1
classes = np.unique(np.concatenate((y_train, y_test), axis=0)) # classes = array([-1,  1])

x = np.arange(len(classes)) # Plot의 X축의 개수 구하기
labels = ["Abnormal", "Normal"] # Plot의 X축의 이름 구하기

values_train = [(y_train == i).sum() for i in classes] # Train 데이터의 정상/비정상 각 총 개수
values_valid = [(y_valid == i).sum() for i in classes] # Test 데이터의 정상/비정상 각 총 개수
values_test = [(y_test == i).sum() for i in classes] # Test 데이터의 정상/비정상 각 총 개수

plt.figure(figsize=(8,4)) # Plot 틀(Figure)의 Size 설정 (8X4)

plt.subplot(1,3,1) # Plot 틀(Figure) 내 3개의 subplot 중 첫 번째(왼쪽) 지정
plt.title("Training Data") # subplot 제목
plt.bar(x, values_train, width=0.6, color=["red", "blue"]) # Train 데이터의 정상/비정상 개수 BarPlot
plt.ylim([0, 1500])
plt.xticks(x, labels) # X축에 변수 기입

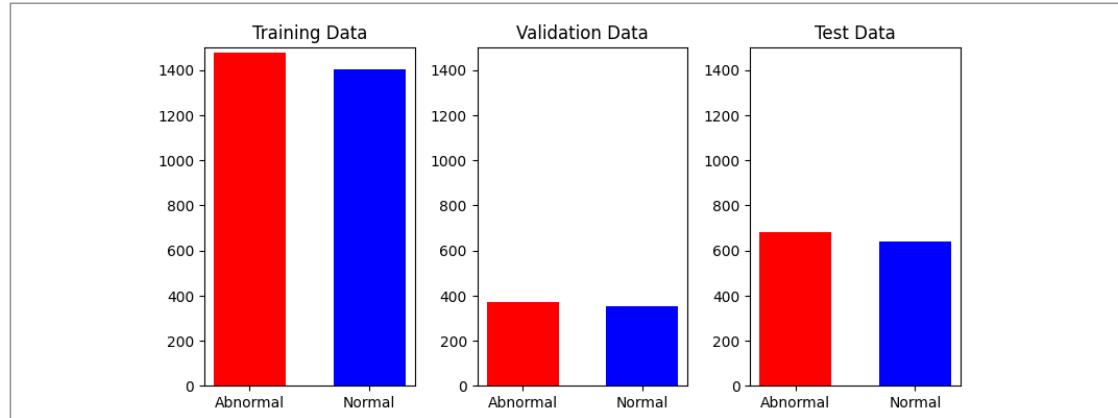
plt.subplot(1,3,2) # Plot 틀(Figure) 내 3개의 subplot 중 두 번째(가운데) 지정
plt.title("Validation Data")
plt.bar(x, values_valid, width=0.6, color=["red", "blue"]) # Test 데이터의 정상/비정상 개수 BarPlot
plt.ylim([0, 1500])
plt.xticks(x, labels)

plt.subplot(1,3,3) # Plot 틀(Figure) 내 3개의 subplot 중 세 번째(오른쪽) 지정
plt.title("Test Data")
plt.bar(x, values_test, width=0.6, color=["red", "blue"]) # Test 데이터의 정상/비정상 개수 BarPlot
plt.ylim([0, 1500])
plt.xticks(x, labels)

plt.tight_layout() # 그림 저장
plt.savefig(save_path + 'data_imbalance.png', dpi=100, bbox_inches='tight') # 그림 저장
plt.show() # 그림 출력
```

[Figure 15] 데이터 불균형 시각화 코드

### ▶ 출력 결과



[Figure 16] 데이터 불균형 시각화 결과

- 각 (정상/비정상) Class에 속한 데이터의 수의 차이가 클 경우, 분류가 제대로 이뤄지지 않을 가능성이 매우 높다. 따라서 먼저 데이터 불균형을 확인해보고, 불균형하다고 판단될 경우 추가적인 작업이 필요하다. 두 Class에 속한 데이터의 수를 맞춰주기 위한 방법은 오버샘플링(Oversampling)과 언더샘플링(Undersampling) 방법이 있다. 다만 본 데이터셋의 경우, 출력 결과 정상 Label의 데이터 수와 비정상 Label의 데이터 수가 정확히 1:1은 아니지만, 비교적 균형적인 편으로 보인다.

## - 시각화 II-1: 특정 시간에서의 시계열 샘플을 플롯

```
import random

labels = np.unique(np.concatenate((y_train, y_test), axis=0)) # labels (-1 or 1)

plt.figure(figsize = (10, 4))

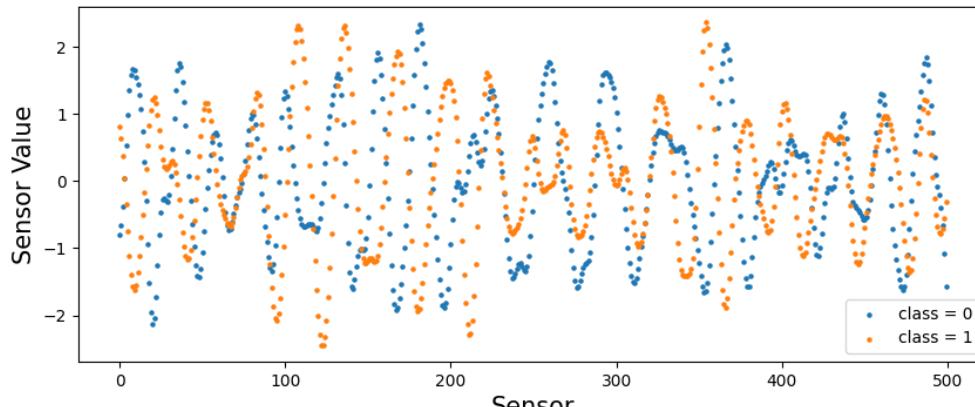
for c in labels:
    c_x_train = x_train[y_train == c]
    if c == -1: c = c +1 # 편의 상 Abnormal Class(-1)를 0으로 조정
    time_t = random.randint(0, c_x_train.shape[0]) # 0~1404 사이의 랜덤한 정수가 특정 time t가 됨
    plt.scatter(range(0, 500), c_x_train[time_t], label="class = "+str(int(c)), marker='o', s=5)

plt.legend(loc="lower right")
plt.xlabel("Sensor", fontsize=15)
plt.ylabel("Sensor Value", fontsize=15)
plt.savefig(save_path +'ford_data_ts_sample1.png', dpi=100, bbox_inches='tight')
plt.show()
```

[Figure 17] 특정 시간대( $t_n$ )에서의 500개의 센서 값 시각화 코드

- 위 “2.1 제조데이터 소개” 내 “데이터 유형/구조”에서의 설명처럼, 각 시계열 샘플(1개의 행)는 특정 시간대( $t_n$ )에서 500개의 센서로 측정한 계측 값들로 구성되어 있다. 위의 코드를 작성하여 실행해보면 정상/비정상 데이터 간의 차이를 볼 수 있다. 위 코드는 무작위로 고른 특정 시간에서의 비정상 데이터의 샘플과, 정상 데이터의 샘플을 출력해보기 위한 그림이며, 무작위의 시간대로 선택한 정상과 비정상 간의 센서 값의 분포 차이를 확인할 수 있다. (단, 본 코드 실행을 통해 전체 데이터의 정상/비정상의 차이점을 육안으로 확인하는 것은 불가능하다.)

## ▶ 출력 결과



[Figure 18] 임의의 시간대( $t_n$ )에서의 500개의 센서 값 시각화 결과

- 시각화 II-2 : 특정 시간에서의 시계열 샘플을 플롯  
(정상/비정상 샘플을 각각 출력)

```

def get_scatter_plot(c):
    time_t = random.randint(0, c_x_train.shape[0]) # 0~1404 사이의 랜덤한 정수가 특정 time t가 됨
    plt.scatter(range(0, c_x_train.shape[1]), c_x_train[time_t], marker='o', s=5, c="r" if c == -1 else "b")
    plt.title("at time: t_{}".format(time_t), fontsize=20)
    plt.xlabel("Sensor", fontsize=14)
    plt.ylabel("Sensor Value", fontsize=14)
    plt.savefig(save_path + '{}.png'.format(state="abnormal" if c == -1 else "normal"),
                dpi=100, bbox_inches='tight')
    plt.show()

labels = np.unique(np.concatenate((y_train, y_test), axis=0))

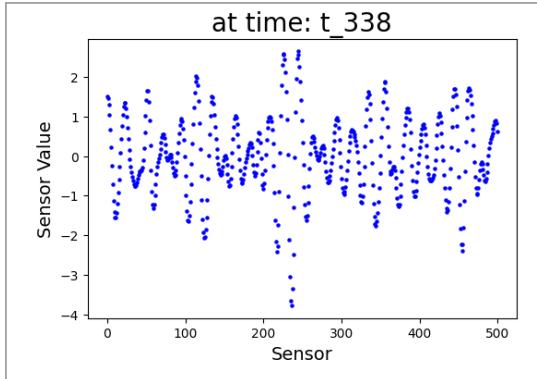
for c in labels:
    c_x_train = x_train[y_train == c]
    if c == -1:
        print("비정상 Label 데이터 수: ", len(c_x_train))
        get_scatter_plot(c)
    else:
        print("정상 Label 데이터 수: ", len(c_x_train))
        get_scatter_plot(c)

```

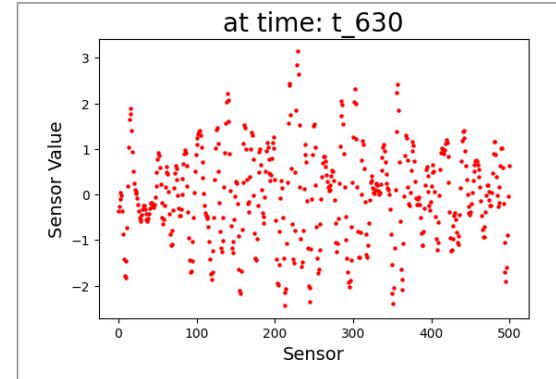
[Figure 19] 특정 시간대( $t_n$ )에서의 500개의 센서 값 시각화 코드 (정상/비정상 샘플 별도로 출력)

위 시각화 II-1과 동일하게 시계열 샘플을 출력하는 코드이나, 위 II-1의 경우 한 개의 그림(Figure)에 정상 상태의 시계열 샘플과 비정상 상태의 시계열 샘플을 함께 출력했다면, 이번에는 서로 다른 그림에 별도로 출력해보고자 한다.

### ▶ 출력 결과



[Figure 20] 정상 샘플 시각화 결과



[Figure 21] 비정상 샘플 시각화 결과

랜덤으로 선택된 숫자 338, 630이 출력하고자 하는 시간대가 되어,  $t=338$  (정상 데이터),  $t=530$  (비정상데이터) 시점에서의 500개 센서가 측정한 계측 값들의 모습을 출력한 것이다. 모두 산점도(ScatterPlot)로 출력하였는데, 다른 형태의 그래프를 원한다면, plt.scatter가 아닌 다른 메서드(i.e. plt.plot)를 사용하면 된다.

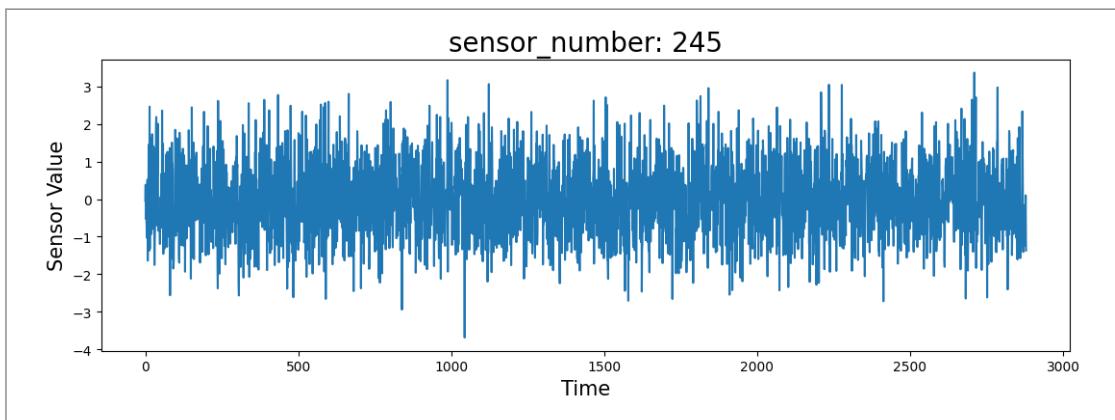
### - 시각화 III : 1개의 임의의 센서 값의 시계열을 플롯

```
sensor_number = random.randint(0, 500) # 0~500 사이의 랜덤한 정수가 Sensor 번호가 됨
plt.figure(figsize = (13, 4))
plt.title("sensor_number: {}".format(sensor_number), fontsize=20)
plt.plot(x_train[:, sensor_number])
plt.xlabel("Time", fontsize=15)
plt.ylabel("Sensor Value", fontsize=15)
plt.savefig(save_path + 'ford_a_sensor.png', dpi=100, bbox_inches='tight')
plt.show()
```

[Figure 22] 임의로 선택한 1개 센서가 측정한 시계열 값 시각화 코드

시계열 데이터일 경우, 시간에 따른 데이터의 흐름을 확인하고 싶을 것이다. 데이터의 형태를 가장 직관적으로 확인할 수 있는 방법으로, 이를 통해 직접 육안으로 이상치(Outlier)를 탐색하기도 한다. 본 코드에서는 0~500번 사이의 랜덤 정수 값을 센서 번호로 하여, 무작위로 뽑은 한 개의 센서의 시계열 데이터 값을 확인해봤다.

#### ▶ 출력 결과



[Figure 23] 임의로 선택한 1개 센서(245번)가 측정한 시계열 값 시각화 결과

랜덤으로 선택된 숫자 245가 센서 번호가 되어, 245번 센서의 시계열 플롯이다. 본 데이터에서는 이상치(Outlier)가 많이 보이지 않으나, 이상치가 많은 제조 데이터의 경우에는 확인 후 제거해주거나, 정상 값으로 조정해주는 것이 일반적이다.

## [단계 ③] 데이터 특성 파악

### - 상관관계 분석

```
import matplotlib.cm as cm
from matplotlib.collections import EllipseCollection

df = pd.DataFrame(data = x_train,
                   columns= ["sensor_{}".format(label+1) for label in range(x_train.shape[1])])
data = df.corr()

def plot_corr_ellipses(data, ax =None, **kwargs):
    M = np.array(data)
    if not M.ndim ==2:
        raise ValueError('data must be a 2D array')
    if ax is None:
        fig, ax = plt.subplots(1, 1, subplot_kw={'aspect':'equal'})
        ax.set_xlim(-0.5, M.shape[1] -0.5)
        ax.set_ylim(-0.5, M.shape[0] -0.5)

    xy = np.indices(M.shape)[::-1].reshape(2, -1).T

    w = np.ones_like(M).ravel()
    h = 1 - np.abs(M).ravel()
    a = 45 * np.sign(M).ravel()

    ec = EllipseCollection(widths=w, heights=h, angles=a, units='x', offsets=xy,
                           transOffset=ax.transData, array=M.ravel(), **kwargs)
    ax.add_collection(ec)

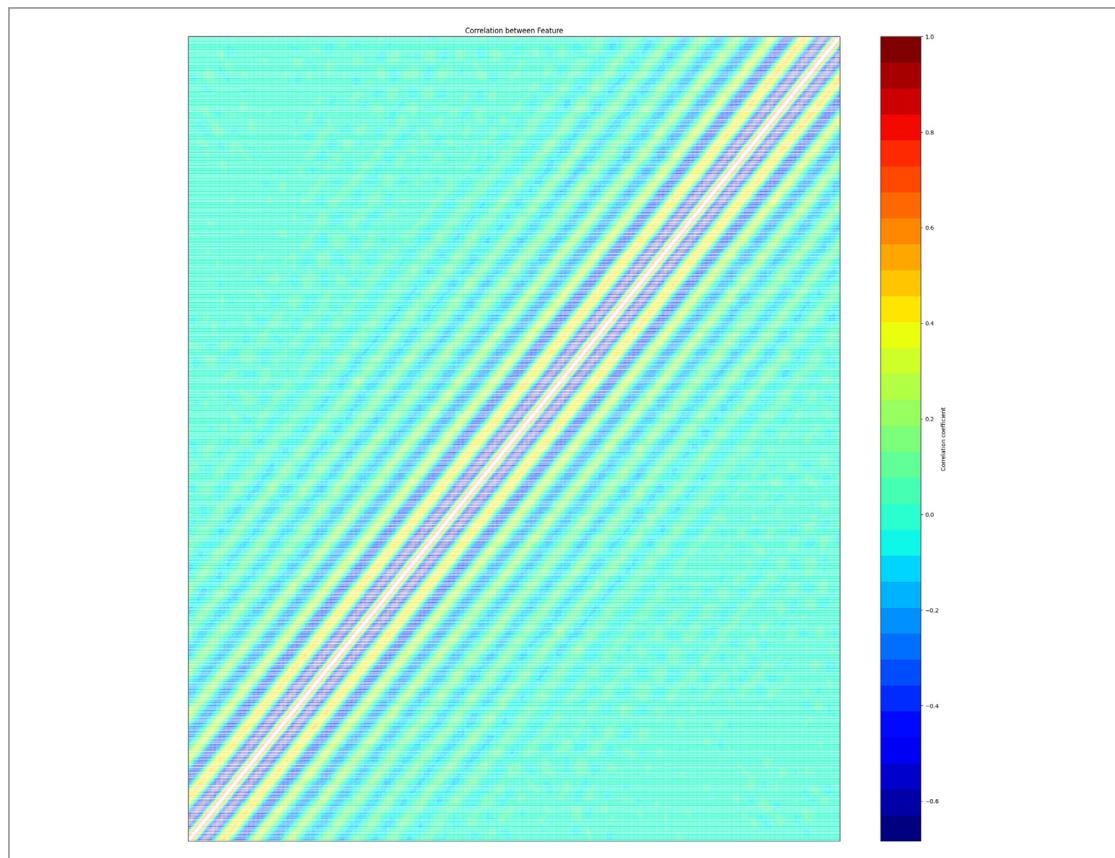
    if isinstance(data, pd.DataFrame):
        ax.set_xticks(np.arange(M.shape[1]))
        ax.set_xticklabels(data.columns, rotation=90)
        ax.set_yticks(np.arange(M.shape[0]))
        ax.set_yticklabels(data.index)
    return ec

fig, ax = plt.subplots(1, 1, figsize=(20, 20))
cmap = cm.get_cmap('jet', 31)
m = plot_corr_ellipses(data, ax=ax, cmap=cmap)
cb = fig.colorbar(m)
cb.set_label('Correlation coefficient')
plt.title('Correlation between Feature')
ax.axes.xaxis.set_visible(False)
ax.axes.yaxis.set_visible(False)
plt.tight_layout()
plt.savefig(save_path + 'corr.png', dpi=100, bbox_inches='tight') # 그림 저장
plt.show()
```

[Figure 24] 상관 관계 분석 코드

상관 관계 분석을 통해, 변수 간의 상관성을 확인해볼 수 있다. 상관성이 1에 가까울수록 양의 상관성이 높고, -1에 가까울수록 음의 상관성이 높다는 의미이다. 일반적으로 상관성이 서로 높은 변수들만 선택할 경우, 모델이 이들에 대한 의존성이 높아지는 문제(오버 피팅)가 발생할 수 있기 때문에, 입력 변수를 선택할 때 상관성이 높은 변수들을 그룹핑하고, 그룹 별로 대표적인 변수를 선택한다.

## ▶ 출력 결과



[Figure 25] 상관 관계 분석 시각화 결과

센서 번호가 가까울수록 상관성이 높은 점을 확인할 수 있다. 거리가 멀어질수록 상관성이 줄어드는 것을 확인할 수 있다. 따라서 Local한 특징(근처 센서 간의 관계)을 학습하는 모델(i.e. CNN)이 좋은 성능을 보일 수 있을 것이라 예상해볼 수 있다.

### [단계 ④] 데이터 전처리

#### - 데이터 정규화

동일 시간 길이(3,600) 내 센서 값들이 상당히 넓은 범위로 퍼져 있을 뿐만 아니라, 변수 간의 Scale이 서로 다르기 때문에, 데이터를 그대로 학습하는 것은 일반적으로 적절하지 않다. 따라서 인풋 값을 정규화(Normalization) 과정을 거치는데, 본 가이드북에서는 StandardScaler 또는 RobustScaler를 통해 진행한다.

흔히 공정 데이터에 이상치(Outlier)가 발생할 수 있는데 이에 강건한 정규화가 필요할 때가 있다. 이때 RobustScaler를 사용한다. StandardScaler는 보다 더 일반적으로 많이 사용하는 정규화 방법으로, 데이터를 단위 분산으로 조정함으로써 Outlier에 취약할 수 있는 반면, RobustScaler는 Feature 간은 스케일을 갖게 되지만 평균과 분산 대신 중간 값(median)과 사분위값(quartile)을 사용함으로써, 극단값(Outlier)에 영향을 받지 않는 특징이 있다.

```

from sklearn.preprocessing import StandardScaler
from sklearn.preprocessing import RobustScaler

# Standard Scaler를 적용하고 싶을 경우 아래 Code를 실행
stdscaler = StandardScaler()
stdscaler.fit(x_train)
x_train = stdscaler.transform(x_train)
x_valid = stdscaler.transform(x_valid)

# Robust Scaler를 적용하고 싶을 경우 아래 주석을 해제하고 Code를 실행
# rscaler = RobustScaler()
# rscaler.fit(x_train)
# x_train = rscaler.transform(x_train)
# x_valid = rscaler.transform(x_valid)

```

[Figure 26] 데이터 정규화 코드

### - 데이터 형태 변환

CNN, RNN 모델에 대해서만 해당하는 내용으로, 아래와 같이 텐서(Tensor)의 형태를 변환해준다. 즉, CNN, RNN 모델이 아니라면 아래 코드를 실행할 필요가 없다. 본 가이드북에서는 동일한 데이터에 대하여 로지스틱회귀, XGBoost, RNN, CNN을 학습 모델로 활용하므로, CNN, RNN 모델 적용 시에는 아래 코드를 실행하여 텐서의 형태를 2D에서 3D로 바꿔준 입력 인풋을 사용한다. 이는 본 시계열 데이터가 다변량(Multivariate)이며, 1개의 채널(Channel)만 갖는 점이 있기에 데이터 형태를 바꿔주고자 하는 것이다.

```

# CNN, RNN 모델의 입력 변수는 채널 축 1개 차원을 확장시킨 3D Tensor이다.
# 즉 아래 CNN, RNN 모델은 x_train, x_valid, x_test를 입력 변수로 넣지 않고,
# x_train_exp, x_valid_exp, x_test_exp을 넣는다.

x_train_exp = np.expand_dims(x_train, -1) # 채널 축 1개 차원을 확장 시킨(Expand) X_train
x_valid_exp = np.expand_dims(x_valid, -1) # 채널 축 1개 차원을 확장 시킨(Expand) X_valid
x_test_exp = np.expand_dims(x_test, -1) # 채널 축 1개 차원을 확장 시킨(Expand) X_test

# 위와 동일한 코드
# x_train_exp = x_train.reshape((x_train.shape[0], x_train.shape[1], 1))
# x_valid_exp = x_train.reshape((x_valid.shape[0], x_valid.shape[1], 1))
# x_test_exp = x_test.reshape((x_test.shape[0], x_test.shape[1], 1))

print("x_train_exp의 형태:", x_train_exp.shape)
print("x_valid_exp의 형태:", x_valid_exp.shape)
print("x_test_exp의 형태:", x_test_exp.shape)

```

[Figure 27] 데이터 형태 변환 코드

### ▶ 출력 결과

⇨ x\_train의 형태: (2880, 500, 1)  
 x\_valid의 형태: (721, 500, 1)  
 x\_test의 형태: (1320, 500, 1)

위와 같이 기존 (N, 500) 형태에서, (N, 500, 1) 형태로 1개의 차원(채널 축)을 늘려주었다. 즉, 500개의 센서가 신경망 모델의 각 노드에 매칭이 된다면, 각 노드에서 Feature(본 데이터 셋에서는 센서의 측정 값 1개)의 개수가 채널 개수와 매칭된다고 볼 수 있다.

- 종속 변수(y)를 양의 값으로 변경

모델 학습을 위해 종속 변수인 Label(비정상: -1 또는 정상: 1) 값을 양의 값으로 바꿔 준다.

```
y_train[y_train == -1] = 0  
y_valid[y_valid == -1] = 0  
y_test[y_test == -1] = 0
```

[Figure 28] 타겟 변수(y)를 양의 값으로 변경하는 코드

## [단계 ⑤] 모델 구축 및 설계

- 본 가이드북에서는 총 4개의 모델에 대한 활용 예시를 보일 것이다. 단계 1~4는 4 개 모델에 공통적으로 적용되는 데이터 탐색 및 전처리 과정이었으며, 단계 5에서는 모델의 구축 및 설계 그리고 단계 6~7에서는 모델 학습 및 평가를 진행할 예정이다. 전체 코드 내부에서 모델 별로 색깔을 구분하여 원한다면, 각 모델 별로 구축, 학습, 평가를 진행할 수 있도록 하였다.

공통 : 

① 로지스틱 회귀(Logistic Regression) : 노란 색 

② XGBoost : 회색 

③ 순환신경망 (Recurrent Neural Network) : 붉은 색 

④ 합성곱신경망 (Convolutional Neural Network) : 푸른 색 

- 로지스틱 회귀(Logistic Regression) 모델 구축

(1) Scikit-learn 라이브러리 활용

```
from sklearn.linear_model import LogisticRegression  
  
clf_lr_1 = LogisticRegression(penalty='l2',  
                               tol=0.0001,  
                               C=1,  
                               fit_intercept=True,  
                               intercept_scaling=1,  
                               random_state=2,  
                               solver='lbfgs',  
                               max_iter=1000,  
                               multi_class='auto',  
                               verbose=0)
```

[Figure 29] 로지스틱 회귀 모델 불러오기 코드

Scikit-learn 라이브러리를 활용하여, 위와 같이 로지스틱 회귀 모형을 매우 간단하게 구성하였다. 로지스틱 회귀 모형의 경우 괄호 ()안에 설정된 모수 (hyperparameter)의 경우 설정하지 않아도(즉, Default 값으로 진행해도) 성능에 큰 차이가 없으므로 사실상, 괄호 안의 파라미터의 값을 설정하지 않고 기본 설정값 (Default)으로 정의해도 된다.

## (2) numpy로 직접 구현

```
class LogisticRegression:
    def __init__(self, lr=0.01, num_iter=1000, fit_intercept=True, verbose=False):
        self.lr = lr
        self.num_iter = num_iter
        self.fit_intercept = fit_intercept
        self.verbose = verbose
        self.eps = 1e-10
        self.threshold = 0.5
        self.loss_history = list()

    def __add_intercept(self, X):
        intercept = np.ones((X.shape[0], 1))
        return np.concatenate((intercept, X), axis=1)

    def __sigmoid(self, z):
        return 1 / (1 + np.exp(-z))

    def __loss(self, h, y):
        return (-y * np.log(h+self.eps) - (1 - y) * np.log(1 - h +self.eps)).mean()

    def fit(self, X, y):
        if self.fit_intercept:
            X =self.__add_intercept(X)

        # weights initialization
        self.theta = np.zeros(X.shape[1])

        for i in range(self.num_iter):
            logit = np.dot(X, self.theta)
            hypothesis =self.__sigmoid(logit)
            gradient = np.dot(X.T, (hypothesis - y)) / y.size
            self.theta -=self.lr * gradient

            if self.verbose ==True and i % 10 ==0:
                loss =self.__loss(hypothesis, y)
                print(f'epoch: {i} Wt loss: {loss} Wt')
                self.loss_history.append(loss)
        return self.loss_history

    def predict_prob(self, X):
        if self.fit_intercept:
            X =self.__add_intercept(X)
        return self.__sigmoid(np.dot(X, self.theta))

    def predict(self, X):
        predicted_labels = np.where(self.predict_prob(X) >self.threshold, 1, 0)
        return predicted_labels

    def eval(self, x, y):
        res_y = np.round(self.predict_prob(x), 0)
        accuracy = np.sum(res_y==y) /len(y)
        return accuracy
```

[Figure 30] numpy를 활용하여 Scratch로 구현한 로지스틱 회귀

Scikit-learn 모듈을 불러와 사용할 경우 굉장히 간편하고 편리하지만, 내부에서 어떻게 작동하는지 알기 어려운 점이 있다. 다만 로지스틱 회귀는 연산 과정이 비교적 간단한 선형 모델이기 때문에 직접 구현해보는 편이 좋다. Figure 30은 numpy로 구현한 로지스틱 회귀 모델이다. numpy로 class를 하나 생성해 주었으며, 이 class 내부에는 7개의 메서드(Method)로 구성되어 있고 주요 기능은 아래와 같다.

- ① fit() : 학습데이터(Training)로 model을 학습하는 메서드
- ② predict\_prob() : 학습데이터(Training)로 학습된 model을 바탕으로 테스트(Test)데이터의 각 인스턴스의 정상일 확률을 도출하는 메서드
- ③ predict() : 학습데이터(Training)로 학습된 model을 바탕으로 테스트(Test)데이터의 Label을 확인하는 메서드

#### ④ eval() : 모델의 테스트 정확도 도출

##### - XGBoost Classifier 모델 구축

```
from xgboost import XGBClassifier  
...  
# 반드시 튜닝해야할 파라미터는 min_child_weight / max_depth / gamma  
...  
xgb = XGBClassifier(  
    learning_rate=0.1,  
    n_estimators=500,  
    max_depth=5,  
    min_child_weight=3,  
    gamma=0.2,  
    subsample=0.6,  
    colsample_bytree=1.0,  
    objective='binary:logistic',  
    nthread=4,  
    scale_pos_weight=1,  
    seed=27)
```

[Figure 31] XGBoost 모델 불러오기 코드

XGBoost 모델은 scikit-learn 외 독자적인 xgboost 모듈을 사용했으며, 설치는 아래 url을 참조한다.

<https://xgboost.readthedocs.io/en/latest/build.html>

XGBoost의 경우 초모수(hyperparameter)의 튜닝(tuning)을 권장하므로, 아래 학습 과정에서 grid search 방법을 통한 초모수 튜닝 방법을 소개할 예정이다. 모델의 초모수(hyperparameter) 별 의미를 간략히 소개하자면 다음과 같다.

##### # XGBoost의 모수 (Parameter) 소개

- ① min\_child\_weight : 값이 높아지면 under-fitting 되는 경우가 있으므로 튜닝되어야 한다.
- ② max\_depth : 루트에서 가장 긴 노드의 거리로써 트리의 최대 깊이를 정의한다. 8이면 중요 변수에서 결론까지 변수가 9개를 거친다는 의미이고, 대개 3~10을 설정한다
- ③ gamma : 노드가 split 되기 위한 loss function의 값이 감소 하는 최소값을 정의한다. gamma 값이 높아질수록 알고리즘은 보수적으로 변하고, loss function의 정의에 따라 적정 값이 달라지기 때문에 튜닝되어야 한다.
- ④ nthread : XGBoost를 실행하기 위한 병렬처리 갯수
- ⑤ colsample\_bytree : 트리를 생성 시 훈련 데이터에서 변수를 샘플링해주는 비율을 의미하며 보통 0.6~1로 설정한다
- ⑥ colsample\_bylevel: 트리의 레벨별로 훈련 데이터의 변수를 샘플링해주는 비율을 의미하며 보통 0.6~1로 설정한다.

⑦ n\_estimators : 트리의 갯수.

⑧ objective : 목적 함수를 의미하며, 모델을 사용하는 목적에 따라 다른며, 회귀에는 ‘reg’, 분류에는 ‘binary’, ‘multi’ 사용  
더 상세한 설명을 원한다면 아래 공식 웹페이지를 참조하자.  
<https://xgboost.readthedocs.io/en/latest/>

### - 순환 신경망(Recurrent Neural Network)

본 가이드북에서는 장단기 메모리 모델(Long Short Term Memory, LSTM)을 사용하였는데, LSTM은 RNN 계열의 모델 중 하나로써, 기존의 RNN 계열 모델의 약점을 보완한 모델이다. RNN의 특성상 관련 정보와 그 정보를 사용하는 지점 사이 거리가 멀 경우, 역전파(Backpropagation)시 경사도(Gradient) 값이 점차 줄어 해당 노드가 학습되지 않는 vanishing gradient problem이 발생하는데, 이를 극복하기 위해서 고안된 것이 바로 LSTM으로, LSTM은 RNN의 숨겨진 상태(hidden state)에 셀 상태(cell-state)를 추가한 구조이다.

```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import LSTM, Flatten, Dense

def make_rnn_model():
    model = Sequential()
    model.add(LSTM(units=256, return_sequences=True))
    model.add(Flatten())
    model.add(Dense(100, activation='relu'))
    model.add(Dense(2, activation='softmax'))
    return model

rnn_model = make_rnn_model()
```

[Figure 32] RNN 모델 구축 코드

케라스(keras) 프레임워크를 사용하여 구현하면 위와 같이 간단하게 구현할 수 있다. 히든 유닛(Hidden Unit) 256개로 구성된 1개의 LSTM 층과 Fully Connected Network를 추가하여 구성된 가장 간단하고 일반적인 형태이다. 사용자가 원한다면, model.add 메서드를 통해 Layer를 원하는 만큼 추가할 수 있다. 상세한 설명은 아래의 keras 공식 tutorial을 참조하는 것이 좋다.

<https://www.tensorflow.org/guide/keras?hl=ko>

## - 합성곱 신경망(Convolutional Neural Network)

```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import BatchNormalization, Conv1D, ReLU, GlobalAveragePooling1D,
Dense

def make_cnn_model():
    model = Sequential()
    model.add(Conv1D(filters=64, kernel_size=3, padding="same"))
    model.add(BatchNormalization())
    model.add(ReLU())
    model.add(Conv1D(filters=64, kernel_size=3, padding="same"))
    model.add(BatchNormalization())
    model.add(ReLU())
    model.add(Conv1D(filters=64, kernel_size=3, padding="same"))
    model.add(BatchNormalization())
    model.add(ReLU())
    model.add(GlobalAveragePooling1D())
    model.add(Dense(2, activation="softmax"))

    return model

cnn_model = make_cnn_model()
```

[Figure 33] CNN 모델 구축 코드

합성곱 신경망(CNN) 또한 케라스(keras) 프레임워크를 사용하여 구현하였으며, 케라스는 사용자 친화적으로, 일반적인 사용 사례에 맞춰 최적화되어, 초심자들도 모델의 구성을 이해하기 용이하다. 1D 컨볼루션층(Convolutional Layer) 3개와 끝단에 완전연결 층(Fully Connected Layer)로 구성한 가장 일반적인 형태이며, 활성화 함수는 ReLu, 폴링(Pooling) 층은 Global Average Pooling을 사용하였다. 본 문제가 분류 과제(Classification Task)이기 때문에, 마지막 활성화 함수는 마지막 뉴런의 출력 값에 대하여 Class 분류를 위하여 정규화를 해주는 소프트맥스(Softmax)를 사용하였다. 본 가이드북에서 합성곱 신경망을 구성하는 층들에 대한 소개와 이론적인 얘기를 하기엔 그 내용이 워낙 방대하기 때문에 자세한 설명은 어려우나, 웹 검색 시 굉장히 상세하게 설명된 자료들이 많으므로 이를 참조해보는 것이 좋다.

## [단계 ⑥] 모델 학습

### - 로지스틱 회귀(Logistic Regression) 학습

```
x_train_lr = np.concatenate((x_train, x_valid), axis=0) # 로지스틱 회귀 학습용 데이터
y_train_lr = np.concatenate((y_train, y_valid), axis=0) # 로지스틱 회귀 테스트용 데이터
```

[Figure 34] 로지스틱 회귀 모델 학습을 위한 학습용/테스트용 데이터 재구성

로지스틱 회귀 모델 분석 시 별도의 검증용(validation) 데이터셋이 필요가 없기에, 위에서 분리한 학습용, 검증용 데이터셋을 잠시 합쳐서 학습용으로 사용하였다. 이는 로지스틱 회귀모델에만 사용하므로 변수명(x\_train\_lr, y\_train\_lr)을 다르게 저장하였다.

## (1) Scikit-learn 로지스틱 회귀 모델 학습

```
# 단계 5의 (1) scikit-learn Logistic Regression 모델 구축에서, 모델 인스턴스 명 clf_lr_1  
clf_lr_1.fit(x_train_lr, y_train_lr)
```

[Figure 35] 로지스틱 회귀 모델 학습 코드

로지스틱 회귀는 정형화된 모델이기 때문에 일반적으로 Scikit-learn에서 제공하는 모델을 그대로 사용하는 것이 성능도 더 우수하고 사용하기도 편하다. Scikit-learn에서 제공하는 모델의 경우, 일반적으로 “fit”이라는 메서드를 사용하여 학습을 수행하며, “Predict”라는 메서드를 사용하여 예측을 수행한다.

### ▶ 출력 결과

```
LogisticRegression(C=1, class_weight=None, dual=False, fit_intercept=True,  
                   intercept_scaling=1, l1_ratio=None, max_iter=100,  
                   multi_class='auto', n_jobs=None, penalty='l2',  
                   random_state=2, solver='lbfgs', tol=0.0001, verbose=0,  
                   warm_start=False)
```

## (2) numpy로 구현한 로지스틱 회귀 모델 학습

```
# 단계 5의 (2) numpy로 직접 구현한 모델에서, 모델 인스턴스 명 clf_lr_2  
clf_lr_2 = LogisticRegression(lr=0.01, num_iter=1000, verbose=True)  
history_lr = clf_lr_2.fit(x_train_lr, y_train_lr)
```

[Figure 36] numpy로 구현한 로지스틱 회귀 모델 학습

### ▶ 출력 결과

epoch: 0	loss: 0.6931471803599453
epoch: 10	loss: 0.6904899637675816
epoch: 20	loss: 0.6889239094938908
epoch: 30	loss: 0.6877765367187595
epoch: 40	loss: 0.6868688414532079
epoch: 50	loss: 0.6861264028512316
epoch: 60	loss: 0.685506647126465
epoch: 70	loss: 0.6849811804514382
epoch: 80	loss: 0.6845297410991716
epoch: 90	loss: 0.6841373605861225
epoch: 100	loss: 0.6837927364357145
epoch: 110	loss: 0.6834871959245933
epoch: 120	loss: 0.683213997239752
epoch: 130	loss: 0.682967841058459
epoch: 140	loss: 0.6827445205184289
epoch: 150	loss: 0.6825406655388402
epoch: 160	loss: 0.6823535532062365
epoch: 170	loss: 0.6821809653950383
epoch: 180	loss: 0.6820210807339536
epoch: 190	loss: 0.6818723918967042

## - XGBoost 학습

### (1) GridSearch를 통한 초모수 튜닝

```
from sklearn.model_selection import GridSearchCV

parameters = {
    "learning_rate" : [0.1, 0.01, 0.001, 0.0001],
    # "eta" : [0.05, 0.10, 0.15, 0.20, 0.25, 0.30] ,
    "max_depth" : [ 3, 4, 5, 6, 8, 10, 12, 15],
    "min_child_weight" : [ 1, 3, 5, 7 ],
    "gamma" : [ 0.0, 0.1, 0.2 , 0.3, 0.4],
    "colsample_bytree" : [ 0.3, 0.4, 0.5 , 0.7 ]
}

grid = GridSearchCV(xgb,
                    parameters, n_jobs=4,
                    scoring="neg_log_loss",
                    cv=5)

grid.fit(
    x_train.squeeze(),
    y_train,
    eval_metric="auc",
    eval_set=[(x_train, y_train), (x_valid, y_valid)],
    verbose=True)
```

[Figure 38] XGBoost 모델 학습 코드 (with Grid Search)

XGBoost 모델의 경우 초모수(hyperparameter) Tuning을 통해 적합한 파라미터를 설정해주는 것이 추천되기에, 본 과정에 서는 grid search 방법을 소개한다. GridSearch란 우리가 직접 지정해준 몇 가지 잠재적 Parameter들의 후보군들의 모든 조합 중에서 모델의 성능이 가장 좋은 조합을 찾는 방법이다. 즉, GridSearch를 통해 “가장 우수한 성능을 보이는 모델의 하이퍼 파라미터를 찾는 하나의 방법이며, Scikit-learn에서 제공하는 GridSearchCV 모듈을 활용하면 편리하게 이용할 수 있다.

### (2) XGBoost모델 학습 (without GridSearch)

단, grid search로 진행할 경우, 설정한 각 초모수 값들의 모든 조합을 각각 모델에 대입하여 모두 학습하는 과정이기에 시간이 매우 오래 걸리므로, 다음과 같이 파라미터 값을 임의로 지정하여 진행해도 된다.

```
print('Start Training')

xgb.fit(
    x_train,
    y_train,
    eval_metric= ['auc', 'error'],
    eval_set=[(x_train, y_train), (x_valid, y_valid)],
    verbose=True
)
print('End of Training')
```

[Figure 39] XGBoost 모델 학습 코드 (without Grid Search)

## ▶ 출력 결과

```
↳ Start Training
[0] validation_0-auc:0.683769    validation_0-error:0.352083    validation_1-auc:0.556568    validation_1-error:0.445215
[1] validation_0-auc:0.730083    validation_0-error:0.33125    validation_1-auc:0.60576    validation_1-error:0.417476
[2] validation_0-auc:0.756296    validation_0-error:0.315625    validation_1-auc:0.609841    validation_1-error:0.417476
[3] validation_0-auc:0.799655    validation_0-error:0.276042    validation_1-auc:0.617937    validation_1-error:0.40638
[4] validation_0-auc:0.818645    validation_0-error:0.261806    validation_1-auc:0.644995    validation_1-error:0.382802
[5] validation_0-auc:0.833596    validation_0-error:0.249306    validation_1-auc:0.6458 validation_1-error:0.385576
[6] validation_0-auc:0.85182    validation_0-error:0.239236    validation_1-auc:0.662909    validation_1-error:0.395284
[7] validation_0-auc:0.863193    validation_0-error:0.223958    validation_1-auc:0.666359    validation_1-error:0.378641
[8] validation_0-auc:0.886401    validation_0-error:0.197917    validation_1-auc:0.683999    validation_1-error:0.377254
[9] validation_0-auc:0.891188    validation_0-error:0.195139    validation_1-auc:0.681197    validation_1-error:0.378641
[10] validation_0-auc:0.899142    validation_0-error:0.182639    validation_1-auc:0.691661    validation_1-error:0.36061
[11] validation_0-auc:0.907881    validation_0-error:0.175347    validation_1-auc:0.702264    validation_1-error:0.352288
[12] validation_0-auc:0.912596    validation_0-error:0.167014    validation_1-auc:0.704358    validation_1-error:0.367545
[13] validation_0-auc:0.921004    validation_0-error:0.15625    validation_1-auc:0.709741    validation_1-error:0.359223
[14] validation_0-auc:0.92498    validation_0-error:0.152083    validation_1-auc:0.708231    validation_1-error:0.355062
[15] validation_0-auc:0.9307 validation_0-error:0.140625    validation_1-auc:0.714699    validation_1-error:0.349515
[16] validation_0-auc:0.930729    validation_0-error:0.143403    validation_1-auc:0.713406    validation_1-error:0.349515
[17] validation_0-auc:0.933671    validation_0-error:0.138542    validation_1-auc:0.71329    validation_1-error:0.345354
[18] validation_0-auc:0.938517    validation_0-error:0.132292    validation_1-auc:0.715077    validation_1-error:0.356449
[19] validation_0-auc:0.941093    validation_0-error:0.127431    validation_1-auc:0.721583    validation_1-error:0.341193
[20] validation_0-auc:0.943606    validation_0-error:0.130556    validation_1-auc:0.720628    validation_1-error:0.345354
```

## - 순환 신경망(Recurrent Neural Network) 학습

```
from tensorflow.keras.callbacks import Callback, EarlyStopping, ModelCheckpoint,
ReduceLROnPlateau

epochs=100
batch_size =64
rnn_model.compile(loss="sparse_categorical_crossentropy",
                  optimizer='adam',
                  metrics=["sparse_categorical_accuracy"]
                 )

callbacks = [ModelCheckpoint(save_path +'rnn_best_model.h5',
                             monitor='val_loss',
                             save_best_only=True),
             ReduceLROnPlateau(
                 monitor="val_loss", factor=0.5, patience=20, min_lr=0.0001
                ),
             EarlyStopping(monitor="val_loss", patience=10, verbose=1)
            ]

history_rnn = rnn_model.fit(
    x_train_exp,
    y_train,
    batch_size=batch_size,
    epochs=epochs,
    callbacks=callbacks,
    validation_data=(x_valid_exp, y_valid),
    verbose=1
)
```

[Figure 41] 순환신경망(RNN) 모델 학습 코드

## ▶ 출력 결과

```
↳ Epoch 1/100
2/45 [=====] - ETA: 4s - loss: 1.0426 - sparse_categorical_accuracy: 0.4766WARNING:tensorflow:Method 'on_train_batch_end' is slow compared
45/45 [=====] - 9s 210ms/step - loss: 0.6901 - sparse_categorical_accuracy: 0.6271 - val_loss: 0.6131 - val_sparse_categorical_accuracy: 0.6874
Epoch 2/100
45/45 [=====] - 8s 169ms/step - loss: 0.5407 - sparse_categorical_accuracy: 0.7302 - val_loss: 0.6041 - val_sparse_categorical_accuracy: 0.6893
Epoch 3/100
45/45 [=====] - 7s 159ms/step - loss: 0.4824 - sparse_categorical_accuracy: 0.7792 - val_loss: 0.6574 - val_sparse_categorical_accuracy: 0.6602
Epoch 4/100
45/45 [=====] - 7s 154ms/step - loss: 0.4214 - sparse_categorical_accuracy: 0.8076 - val_loss: 0.6045 - val_sparse_categorical_accuracy: 0.6990
Epoch 5/100
45/45 [=====] - 7s 155ms/step - loss: 0.3407 - sparse_categorical_accuracy: 0.8604 - val_loss: 0.6374 - val_sparse_categorical_accuracy: 0.6893
Epoch 6/100
45/45 [=====] - 7s 156ms/step - loss: 0.2780 - sparse_categorical_accuracy: 0.8944 - val_loss: 0.6618 - val_sparse_categorical_accuracy: 0.7351
Epoch 7/100
45/45 [=====] - 9s 192ms/step - loss: 0.2044 - sparse_categorical_accuracy: 0.9205 - val_loss: 0.6551 - val_sparse_categorical_accuracy: 0.7878
Epoch 8/100
45/45 [=====] - 7s 155ms/step - loss: 0.1321 - sparse_categorical_accuracy: 0.9514 - val_loss: 0.7333 - val_sparse_categorical_accuracy: 0.7517
Epoch 9/100
45/45 [=====] - 8s 168ms/step - loss: 0.0911 - sparse_categorical_accuracy: 0.9681 - val_loss: 0.6430 - val_sparse_categorical_accuracy: 0.8183
Epoch 10/100
45/45 [=====] - 7s 156ms/step - loss: 0.0463 - sparse_categorical_accuracy: 0.9930 - val_loss: 0.6971 - val_sparse_categorical_accuracy: 0.8031
```

## - 합성곱 신경망(Convolutional Neural Network) 학습

```
from tensorflow.keras.callbacks import Callback, EarlyStopping, ModelCheckpoint, ReduceLROnPlateau

epochs = 300
batch_size = 64
callbacks = [
    ModelCheckpoint(
        save_path + "cnn_best_model.h5", save_best_only=True, monitor="val_loss"
    ),
    ReduceLROnPlateau(
        monitor="val_loss", factor=0.5, patience=20, min_lr=0.0001
    ),
    EarlyStopping(monitor="val_loss", patience=50, verbose=1),
]

cnn_model.compile(
    optimizer="adam",
    loss="sparse_categorical_crossentropy",
    metrics=["sparse_categorical_accuracy"],
)

history_cnn = cnn_model.fit(
    x_train_exp,
    y_train,
    batch_size=batch_size,
    epochs=epochs,
    callbacks=callbacks,
    validation_data=(x_valid_exp, y_valid),
    verbose=1,
)
```

[Figure 42] 합성곱 신경망(CNN) 모델 학습 코드

## ▶ 출력 결과

```
↳ Epoch 1/300
45/45 [=====] - 1s 28ms/step - loss: 0.5770 - sparse_categorical_accuracy: 0.6865 - val_loss: 0.7054 - val_sparse_categorical_accuracy: 0.4868
Epoch 2/300
45/45 [=====] - 1s 19ms/step - loss: 0.4783 - sparse_categorical_accuracy: 0.7698 - val_loss: 0.7382 - val_sparse_categorical_accuracy: 0.4868
Epoch 3/300
45/45 [=====] - 1s 19ms/step - loss: 0.4422 - sparse_categorical_accuracy: 0.7795 - val_loss: 0.7417 - val_sparse_categorical_accuracy: 0.4868
Epoch 4/300
45/45 [=====] - 1s 19ms/step - loss: 0.4285 - sparse_categorical_accuracy: 0.7872 - val_loss: 0.7277 - val_sparse_categorical_accuracy: 0.4868
Epoch 5/300
45/45 [=====] - 1s 21ms/step - loss: 0.4096 - sparse_categorical_accuracy: 0.7896 - val_loss: 0.6981 - val_sparse_categorical_accuracy: 0.4982
Epoch 6/300
45/45 [=====] - 1s 21ms/step - loss: 0.3991 - sparse_categorical_accuracy: 0.8007 - val_loss: 0.6871 - val_sparse_categorical_accuracy: 0.5756
Epoch 7/300
45/45 [=====] - 1s 21ms/step - loss: 0.3858 - sparse_categorical_accuracy: 0.8128 - val_loss: 0.6637 - val_sparse_categorical_accuracy: 0.6574
Epoch 8/300
45/45 [=====] - 1s 21ms/step - loss: 0.3825 - sparse_categorical_accuracy: 0.8080 - val_loss: 0.6564 - val_sparse_categorical_accuracy: 0.6755
Epoch 9/300
45/45 [=====] - 1s 19ms/step - loss: 0.3832 - sparse_categorical_accuracy: 0.8101 - val_loss: 0.6827 - val_sparse_categorical_accuracy: 0.5160
Epoch 10/300
45/45 [=====] - 1s 21ms/step - loss: 0.3660 - sparse_categorical_accuracy: 0.8309 - val_loss: 0.6274 - val_sparse_categorical_accuracy: 0.5825
Epoch 11/300
45/45 [=====] - 1s 21ms/step - loss: 0.3699 - sparse_categorical_accuracy: 0.8257 - val_loss: 0.5371 - val_sparse_categorical_accuracy: 0.6655
Epoch 12/300
45/45 [=====] - 1s 21ms/step - loss: 0.3658 - sparse_categorical_accuracy: 0.8243 - val_loss: 0.4974 - val_sparse_categorical_accuracy: 0.7184
```

## [단계 ⑦] 모델 평가

- 로지스틱 회귀(Logistic Regression) 평가

(1) Scikit-learn의 로지스틱 회귀 모델 평가

```
y_pred = clf_lr_1.predict(x_test)
score = clf_lr_1.score(x_test, y_test)
print("%s: %.2f%%" % ("Logistic Regression Prediction Rate", score*100))
```

[Figure 43] Scikit-learn의 로지스틱 회귀 모델 평가(정확도 측정) 코드

▶ 출력 결과

```
↳ Logistic Regression Prediction Rate: 48.41%
```

(2) numpy로 구현한 로지스틱 회귀 모델 평가

```
score = clf_lr_2.eval(x_test, y_test)
print("%s: %.2f%%" % ("Logistic Regression Prediction Rate", score*100))
```

[Figure 44] numpy로 구현한 로지스틱 회귀 모델 평가(정확도 측정) 코드

▶ 출력 결과

```
↳ Logistic Regression Prediction Rate: 49.17%
```

- XGBoost 평가

```
y_pred = xgb.predict(x_test)
y_pred_proba = xgb.predict_proba(x_test)[:, 1]

print("About the present model")
print("Accuracy : %.4g" % accuracy_score(y_test, y_pred))
print("AUC Score (training set): %.4f" % roc_auc_score(y_test, y_pred_proba))
print("F1 Score (training set): %.4f" % f1_score(y_test, y_pred))
```

[Figure 45] XGBoost 모델 평가(정확도, AUC, F1 Score 측정) 코드

▶ 출력 결과

```
↳
About the present model
Accuracy : 0.7803
AUC Score (training set): 0.853490
F1 Score (training set): 0.771654
```

- 순환 신경망(Recurrent Neural Network) 평가

```
from tensorflow.keras.models import load_model

rnn_model = tf.keras.models.load_model(save_path + "rnn_best_model.h5")
scores = rnn_model.evaluate(x_test_exp, y_test)

print("\n""Test accuracy", scores[1])
print("\n""Test loss", scores[0])
print("%s: %.2f%%" % (rnn_model.metrics_names[1], scores[1]*100))
```

[Figure 46] RNN 모델 평가(정확도, 손실값 측정) 코드

## ▶ 출력 결과

```
[+] 42/42 [=====] - 1s 15ms/step - loss: 0.5470 - sparse_categorical_accuracy: 0.8250
    Test accuracy 0.824999988079071
    Test loss 0.5469545722007751
    sparse_categorical_accuracy: 82.50%
```

- 합성곱 신경망(Convolutional Neural Network) 평가

```
from tensorflow.keras.models import load_model
cnn_model = tf.keras.models.load_model(save_path +"cnn_best_model.h5")
scores = cnn_model.evaluate(x_test_exp, y_test)
print("\n""Test accuracy", scores[1])
print("\n""Test loss", scores[0])
print("%s: %2.2%" % (cnn_model.metrics_names[1], scores[1]*100))
```

[Figure 47] CNN 모델 평가(정확도, 손실값 측정) 코드

## ▶ 출력 결과

```
[+] 42/42 [=====] - 0s 3ms/step - loss: 0.0976 - sparse_categorical_accuracy: 0.9705
    Test accuracy 0.9704545736312866
    Test loss 0.09757672995328903
    sparse_categorical_accuracy: 97.05%
```

## [단계 ⑧] 결과 분석 및 해석

- 공통 분석 방법 : (1) 혼동 행렬 | (2) ROC Curve | Learining Curve  
 (3) 학습 과정 중 정확도 추이, (4) 손실(Loss) 추이

### (1) 혼동 행렬(Confusion Matrix)

```
from sklearn.metrics import classification_report, confusion_matrix

def draw_confusion_matrix(model, xt, yt, model_name):
    Y_pred = model.predict(xt)

    if model_name in ["cnn", "rnn"]:
        y_pred = np.argmax(Y_pred, axis=1)
    else: y_pred = Y_pred

    plt.figure(figsize=(3,3))
    cm = confusion_matrix(yt, y_pred)
    plt.imshow(cm, interpolation='nearest', cmap=plt.cm.Blues)
    plt.title("Confusion Matrix")
    plt.colorbar()
    tick_marks = np.arange(2)
    plt.xticks(tick_marks, ['False', 'True'], rotation=45)
    plt.yticks(tick_marks, ['False', 'True'])
    thresh = cm.max()/1.2
    normalize = False

    fmt = '.2f' if normalize else 'd'
    for i,j in itertools.product(range(cm.shape[0]), range(cm.shape[1])):
        plt.text(j,i, format(cm[i,j], fmt),
                 horizontalalignment="center",
                 color="white"if cm[i,j] > thresh else "black",
                 fontsize=12)

    plt.ylabel("Actual Label")
    plt.xlabel("Predicted Label")
    plt.tight_layout()
    plt.savefig(save_path + '{}_cm.png'.format(model_name), dpi=100, bbox_inches='tight')
    plt.show()

print(classification_report(yt, y_pred))
```

[Figure 48] 혼동 행렬 (Confusion Matrix) 출력 코드

혼동 행렬(Confusion Matrix) 및 ROC 커브를 해석하기 전에 민감 도(Sensitivity)와 특이도(Specificity)의 개념을 알아야 한다. 흔히 민감 도와 특이도를 설명하기 위해 의사의 진단을 예시로 드는데, 의사는 환자를 진단해 병(e.g. 코로나 바이러스)에 걸렸느냐 안 걸렸느냐를 판단한다. 이때 발생할 수 있는 경우는 아래와 같다.

		실제	
		양성	음성
예측	양성	정상 진단	오진
	음성	오진	정상 진단

		실제	
		양성	음성
예측	양성	True Positive	False Positive
	음성	False Negative	True Negative

- Confusion Matrix를 해석하는 방법은 다음과 같다.

- ① **True Positives** : 정상인 레이블을 정상이라 하는 경우. 즉 정상을 정상이라고 정확하게 분류.
- ② **False Negatives** : 정상인 레이블을 비정상이라 하는 경우. 즉 정상을 비정상이라고 잘못 분류.
- ③ **False Positives** : 비정상 레이블을 정상이라 하는 경우. 즉, 비정상을 정상이라고 잘못 분류.
- ④ **True Negatives** : 비정상 레이블을 비정상이라 하는 경우. 즉, 비정상을 비정상이라고 정확하게 분류.

여기서 중요 용어의 개념을 다음과 같이 정리할 수 있다.

- ① **민감도(Sensitivity, True positive rate(TPR), Recall)** : 실제 병에 걸린 사람이 양성(Positive) 판정을 받는 비율.
- ② **특이도(Specificity, True Negative rate(TNR))** : 정상인이 음성(Negative) 판정을 받는 비율.

False positive rate(FPR) :  $1 - \text{specificity}$

- ③ **정확도(Accuracy)** : 전체 데이터 중 제대로 분류된 데이터 비율
- ④ **에러율(Error Rate)** : 전체 데이터 중 제대로 분류되지 않은 데이터 비율
- ⑤ **정밀도(Precision)** : Positive로 예측했을 때, 실제로 Positive인 데이터 비율

## (2) ROC Curve

ROC curve를 한 마디로 이야기하자면 ROC 커브는 좌상단에 붙어있는 커브가 더 좋은 분류기를 의미한다고 생각할 수 있다.

```
from sklearn.metrics import roc_curve, auc

def draw_roc(model, xt, yt, model_name):
    Y_pred = model.predict(xt)

    if model_name in ["cnn", "rnn"]:
        y_pred = np.argmax(Y_pred, axis=1)
    else: y_pred = Y_pred

    fpr, tpr, thr = roc_curve(yt, y_pred)
    roc_auc = auc(fpr, tpr)
    plt.figure()
    lw = 2
    plt.plot(fpr, tpr, color='darkorange',
              lw=lw, label='ROC curve (area = %0.2f)' % roc_auc)
    plt.plot([0, 1], [0, 1], color='navy', lw=lw, linestyle='--')
    plt.xlim([0.0, 1.0])
    plt.ylim([0.0, 1.05])
    plt.xlabel('False Positive Rate')
    plt.ylabel('True Positive Rate')
    plt.title('Receiver operating characteristic {}'.format(model_name))
    plt.legend(loc="lower right")
    plt.ion()
    plt.tight_layout()
    plt.savefig(save_path + '{}_roc.png'.format(model_name), dpi=100, bbox_inches='tight')
    plt.show()
```

[Figure 49] ROC Curve 출력 코드

## (3) Epoch에 따른 학습 & 검증의 손실(loss) 그래프

```
def plot_loss_graph(history, pic_name):
    plt.figure()
    plt.plot(history.history['loss'])
    plt.plot(history.history['val_loss'])
    plt.title("Training & Validation Loss")
    plt.ylabel("loss", fontsize="large")
    plt.xlabel("epoch", fontsize="large")
    plt.legend(["train", "validation"], loc="best")
    plt.tight_layout()
    plt.savefig(save_path + '{}.png'.format(pic_name), dpi=100, bbox_inches='tight') # 그림 저장
    plt.show()
```

[Figure 50] 학습 과정 중, Epoch에 따른 손실 추이 그래프 출력 코드

## (4) Epoch에 따른 정확도(Accuracy Rate) 그래프

```
def plot_prediction_graph(history, pic_name):
    plt.figure()
    plt.plot(history.history["sparse_categorical_accuracy"])
    plt.plot(history.history["val_"+"sparse_categorical_accuracy"])
    plt.title("model " + "Prediction Accuracy")
    plt.ylabel("sparse_categorical_accuracy", fontsize="large")
    plt.xlabel("epoch", fontsize="large")
    plt.legend(["train", "validation"], loc="best")
    plt.tight_layout()
    plt.savefig(save_path + '{}.png'.format(pic_name), dpi=100, bbox_inches='tight') # 그림 저장
    plt.show()
```

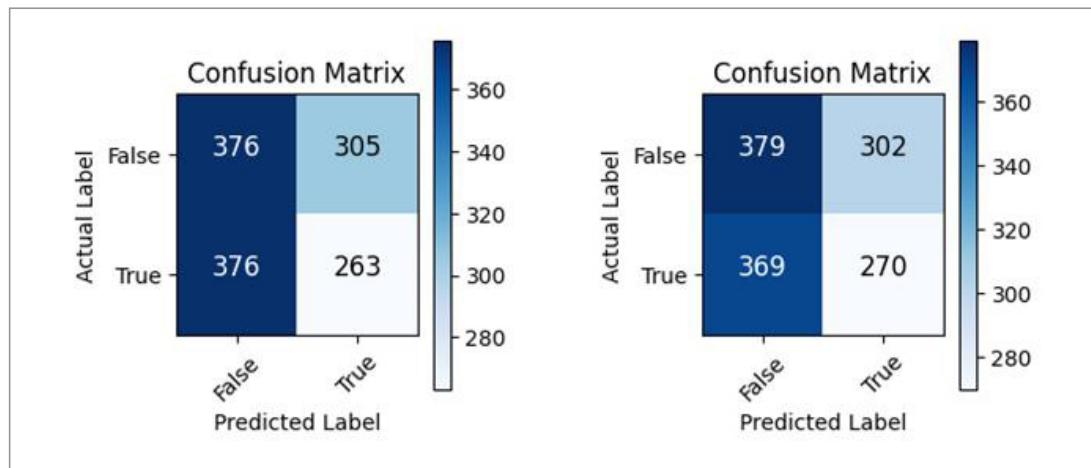
[Figure 51] 학습 과정 중, Epoch에 따른 정확도 추이 그래프 출력 코드

## - 결과 분석 및 해석 I - 로지스틱 회귀(Logistic Regression)

### (1) 혼동 행렬

```
draw_confusion_matrix(clf_lr_1, x_test, y_test, "logistic_regression_sklearn")
draw_confusion_matrix(clf_lr_2, x_test, y_test, "logistic_regression_numpy")
```

[Figure 52] 혼동 행렬 – 로지스틱 회귀 출력 코드



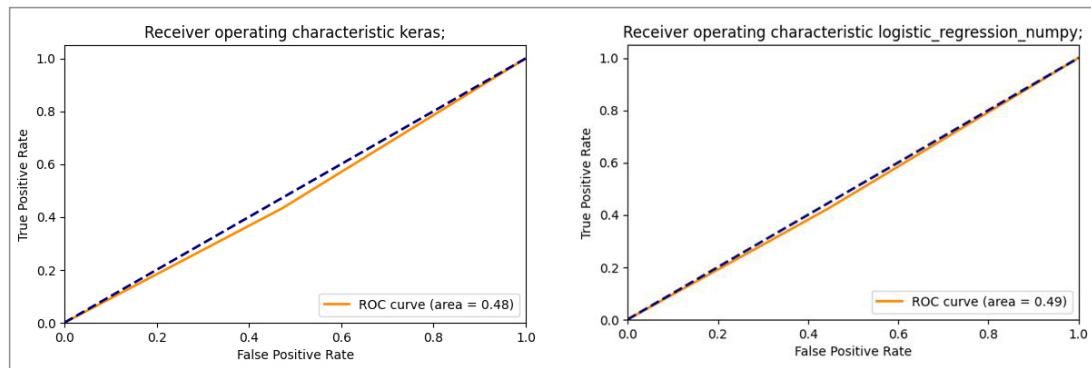
[Figure 53] 혼동 행렬 – 로지스틱 회귀(scikit-learn 모델 (좌) | numpy 구현 모델 (우))

두 로지스틱 회귀 모델의 성능은 거의 비슷하다고 볼 수 있다.

### (2) ROC Curve

```
draw_roc(clf_lr_1, x_test, y_test, "logistic_regression")
draw_roc(clf_lr_2, x_test, y_test, "logistic_regression_numpy")
```

[Figure 54] ROC Curve – 로지스틱 회귀 출력 코드



[Figure 55] ROC Curve – 로지스틱 회귀(scikit-learn 모델 (좌) | numpy 구현 모델 (우))

ROC Curve에 대해 간략이 소개하자면, 먼저 x축( $1 - \text{specificity} = \text{FPR}(\text{False Positive Rate})$ )은 가짜 중에 진짜를 찾은 비율(가짜 중에 잘못 예측한 비율)이고, y 축(Sensitivity)이 의미하는 바는 진짜 중에 진짜를 찾은 비율(진짜 중에 진짜를 잘 찾은 비율)이 된다. 진짜로 예측한 값들 중에서 실제로도 진짜일 경우가 실제로 가

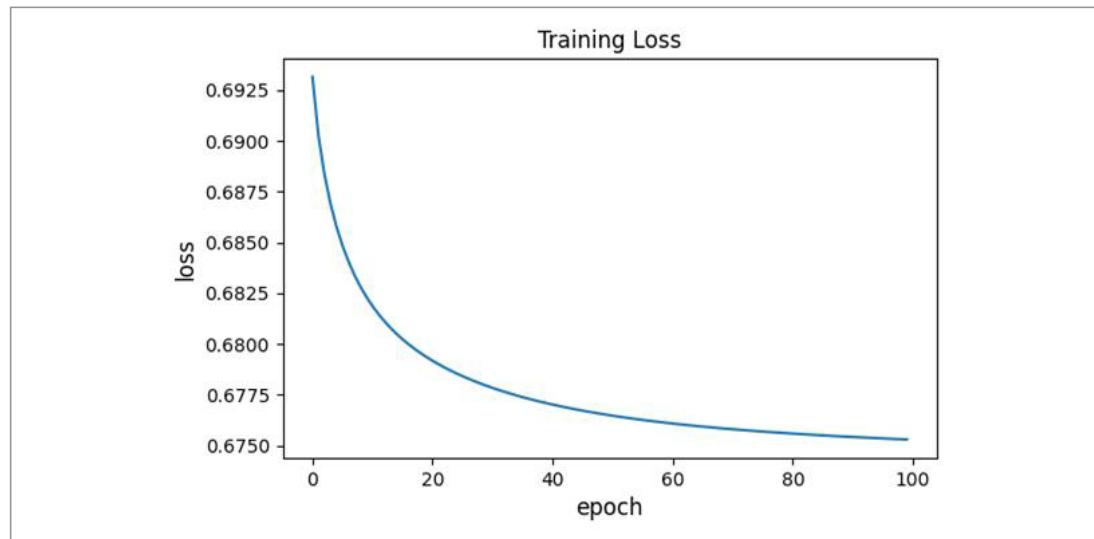
짜일 경우보다 높아야 AUC가 높게 나온다. 점선은 랜덤으로 예측을 한 것과 마찬가지로 이해하면 되므로 성능이 가장 나쁜 경우이고, 곡선이 왼쪽 위로 굽어지면 굽어질수록 AUC가 넓어지므로, 더욱 정확한 모델임을 알 수 있다. 기본적으로 Logistic Regression의 경우 ROC Curve가 점선 아래로 그려지는 것으로 보아, 본 데이터에 대해서는 좋은 모델이 아님을 알 수 있다. 다만, Figure 48.을 보면 두 로지스틱 회귀 모델의 성능 차이를 볼 수 있는데, numpy 구현 모델이 area 넓이가 조금 더 크기 때문에, 아주 조금 더 나은 모델임을 알 수 있다.

### (3) Epoch에 따른 학습 과정 중 손실(loss) 그래프

numpy로 구현한 로지스틱 회귀 모델을 통해 학습 과정 중 손실(loss) 그래프를 그려볼 수 있다. 다만, 검증용(validation) 데이터셋을 통해 학습과정 중 validation을 진행하는 절차가 없기 때문에, 학습 중 손실(Training Loss) 추이만 그려볼 수 있다.

```
plt.figure()
plt.plot(history_lr)
plt.title("Training Loss")
plt.ylabel("loss", fontsize="large")
plt.xlabel("epoch", fontsize="large")
plt.tight_layout()
plt.savefig(save_path + 'lr_learning_curve.png', dpi=100, bbox_inches='tight')
plt.show()
```

[Figure 56] Epoch에 따른 학습 과정 중 손실(loss) 그래프 구현 코드 -로지스틱 회귀



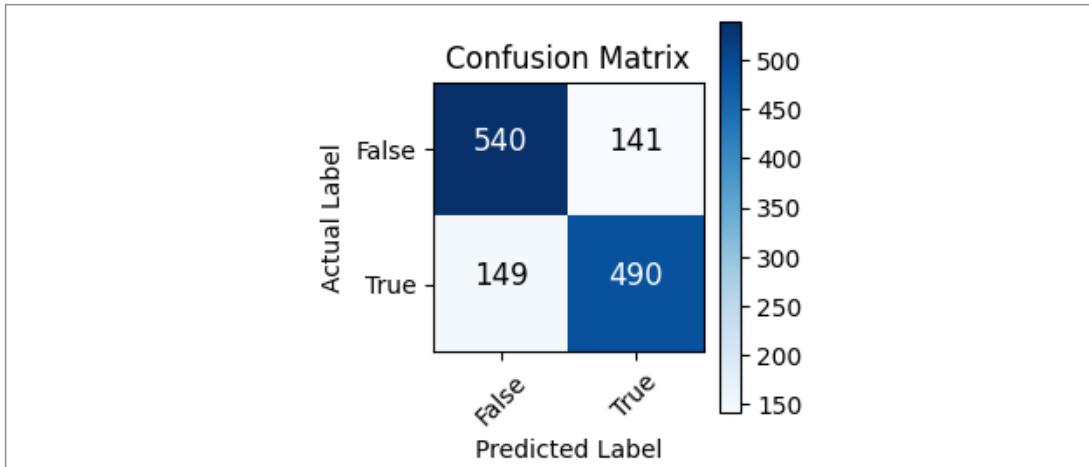
[Figure 57] Epoch에 따른 학습 과정 중 손실(loss) 그래프 - 로지스틱 회귀

## - 결과 분석 및 해석 II - XGBoost

### (1) 혼동 행렬

```
draw_confusion_matrix(xgb, x_test, y_test, "xgboost")
```

[Figure 58] 혼동 행렬 - XGBoost 출력 코드

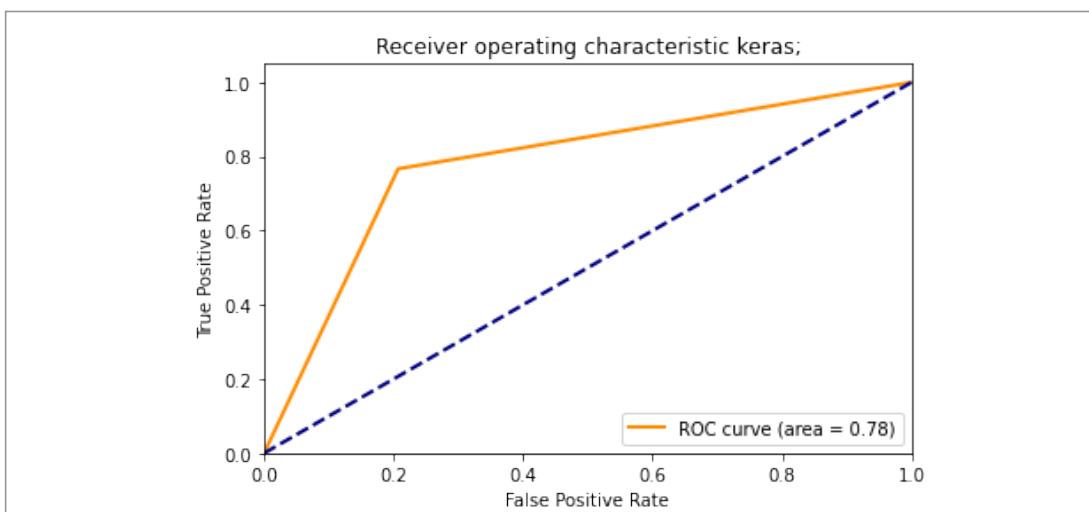


[Figure 59] 혼동 행렬 - XGBoost

### (2) ROC Curve

```
draw_roc(xgb, x_test, y_test, "xgboost")
```

[Figure 60] ROC Curve - XGBoost 출력 코드



[Figure 61] ROC Curve - XGBoost

ROC 커브를 보면, 본 데이터셋에 대해서는 XGBoost가 위의 로지스틱 회귀 모델보다 훨씬 더 좋은 성능을 갖는 모델임을 알 수 있다.

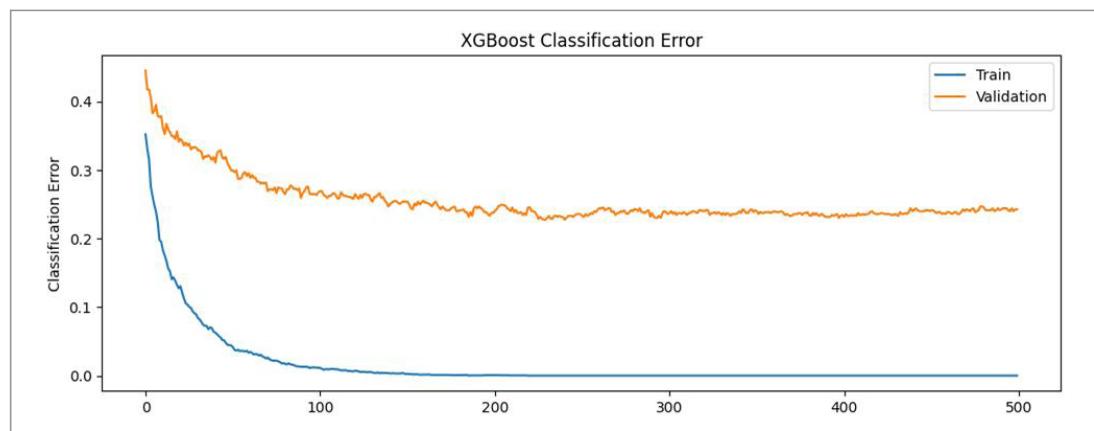
### (3) Learning Curve - XGBoost의 분류 오차 추이

```
draw_roc(xgb, x_test, y_test, "xgboost")

xgb_results = xgb.evals_result() # xgboost 모델의 평가 결과 불러오기
epochs = len(xgb_results['validation_0']['error']) # iteration 수
x_axis = range(0, epochs) # x축(epoch) 범위 설정

# plot classification error
fig, ax = plt.subplots(figsize=(10,4))
ax.plot(x_axis, xgb_results['validation_0']['error'], label='Train')
ax.plot(x_axis, xgb_results['validation_1']['error'], label='Validation')
ax.legend()
plt.ylabel('Classification Error')
plt.title('XGBoost Classification Error')
plt.tight_layout()
plt.savefig(save_path + 'learning_curve_error_xgb.png', dpi=100, bbox_inches='tight')
plt.show()
```

[Figure 62] XGBoost의 분류 오차(Classification Error) 플롯 코드

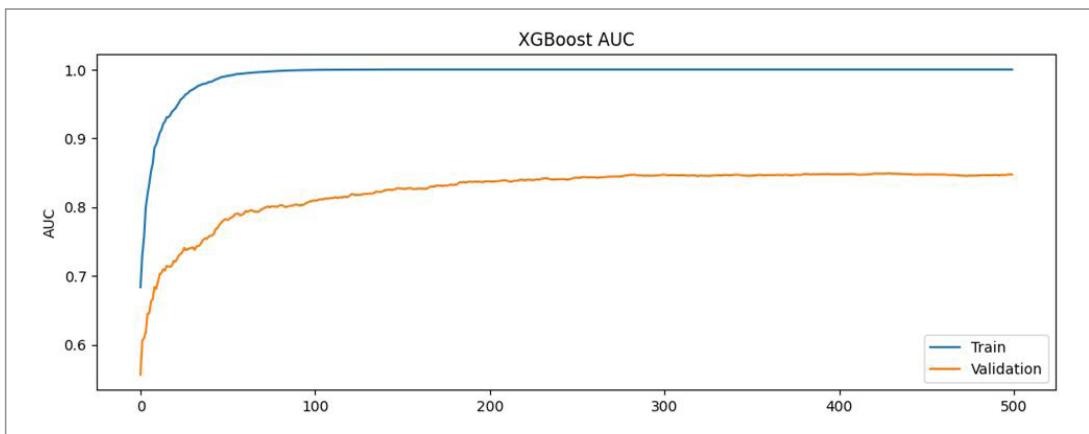


[Figure 63] XGBoost의 분류 오차(Classification Error) 플롯: 학습이 진행됨에 따라 error 값이 점차 감소함

### (4) Learning Curve - XGBoost의 학습 과정 중 AUC 추이

```
# plot auc
fig, ax = plt.subplots(figsize=(10,4))
ax.plot(x_axis, xgb_results['validation_0']['auc'], label='Train')
ax.plot(x_axis, xgb_results['validation_1']['auc'], label='Validation')
ax.legend()
plt.ylabel('AUC')
plt.title('XGBoost AUC')
plt.tight_layout()
plt.savefig(save_path + 'learning_curve_auc_xgb.png', dpi=100, bbox_inches='tight')
plt.show()
```

[Figure 64] XGBoost의 학습 과정 중 AUC 값 변화 추이 플롯 코드



[Figure 65] XGBoost의 학습 과정 중 AUC 값 변화 추이 플롯: 학습이 진행됨에 따라 AUC 값이 점점 증가함

## (5) Feature Importance 확인

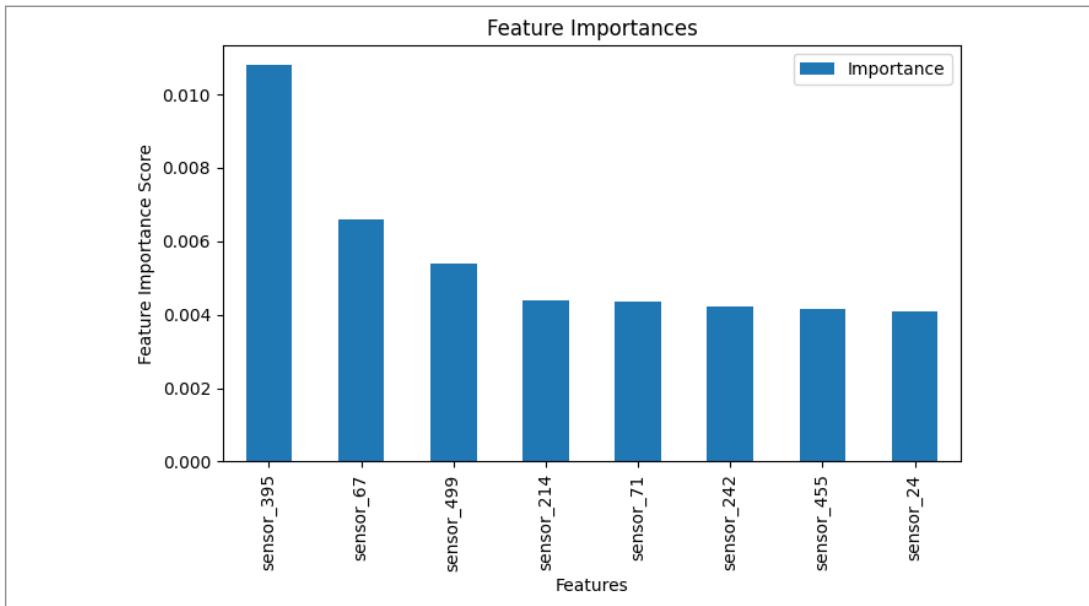
```
from sklearn.metrics import accuracy_score, roc_auc_score, f1_score
feat_imp = xgb.feature_importances_
idx = np.where(feat_imp > 0.004)
feat_imp_important = feat_imp[idx]
feat = ["sensor_{}".format(i+1) for i in idx[0]]
# clf.best_estimator_.booster().get_fscore()
res_df = pd.DataFrame(
    {'Features': feat, 'Importance': feat_imp_important}).sort_values(by='Importance', ascending=False)

res_df.plot('Features', 'Importance', kind='bar', title='Feature Importances', figsize = (7, 5))
plt.ylabel('Feature Importance Score')
plt.tight_layout()
plt.savefig(save_path + 'xgb_feature_importance.png', dpi=100, bbox_inches='tight')
plt.show()

print(res_df)
print(res_df["Features"].tolist())
```

[Figure 66] XGBoost모델을 통한 변수 중요도 확인 코드

## ▶ 출력 결과



[Figure 67] XGBoost모델을 통한 변수 중요도 시각화 결과

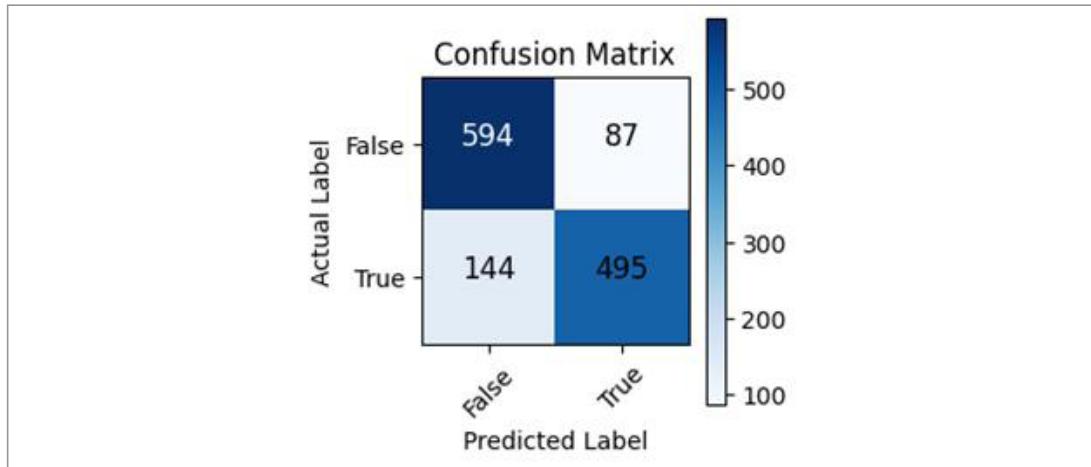
Figure 67. 과 같이 예측 결과에 영향력이 큰 변수들 순서로 출력해보면, Sensor\_395 센서가 가장 영향력 있는 변수임을 알 수 있다.

- 결과 분석 및 해석 III - 순환 신경망(RNN)

(1) 혼동 행렬

```
draw_confusion_matrix(rnn_model, x_test_exp, y_test, "rnn")
```

Figure 67. 혼동 행렬 - RNN 출력코드

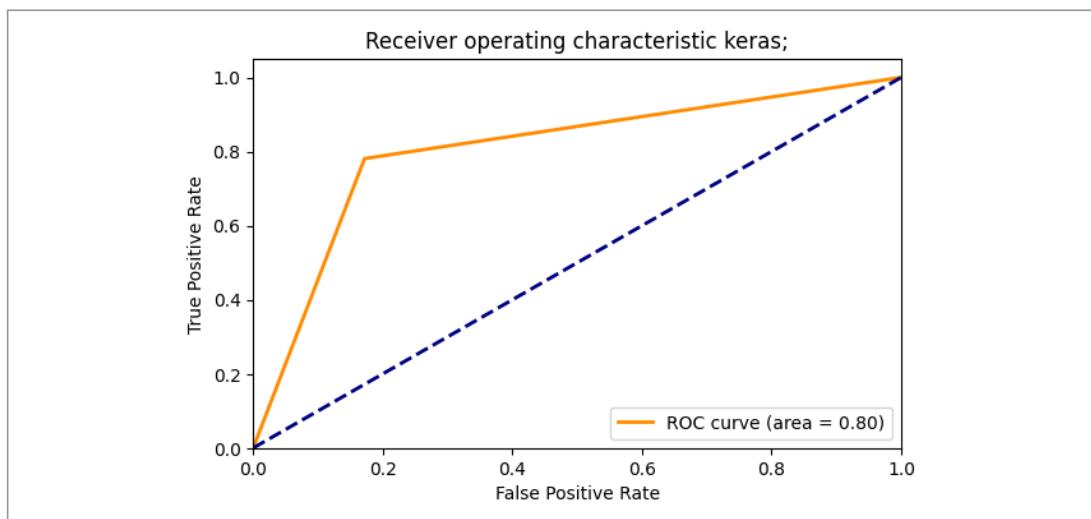


[Figure 68] 혼동 행렬 - RNN

(2) ROC Curve

```
draw_roc(rnn_model, x_test_exp, y_test, "rnn")
```

Figure 69. ROC Curve - RNN 출력 코드



[Figure 70] ROC Curve - RN

### (3) Learning Curve: Epoch에 따른 학습, 검증 손실(Loss) 추이

```
plot_loss_graph(history_rnn, "rnn")
```

[Figure 71] 학습 과정 중, Epoch에 따른 손실값 추이 그래프 - RNN 출력 코드

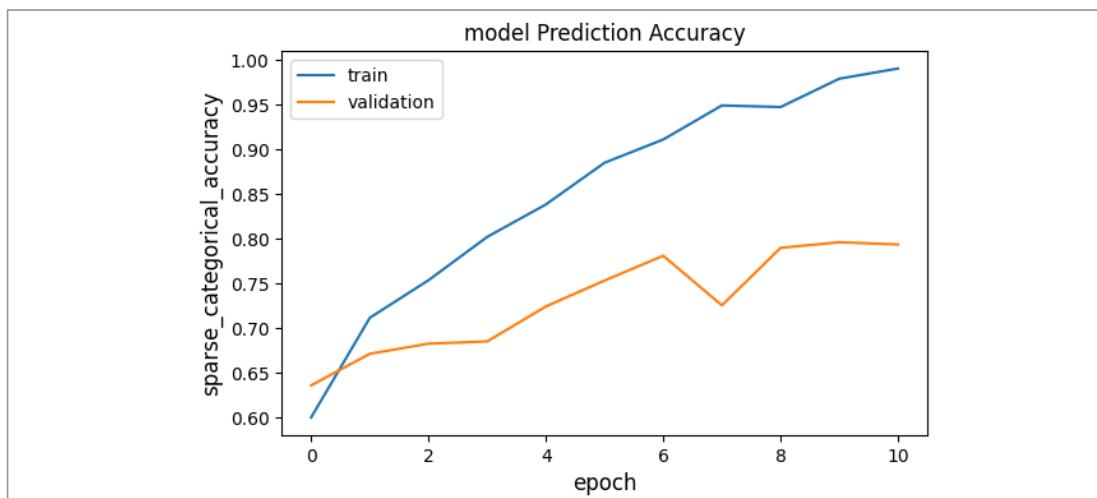


[Figure 72] 학습 과정 중, Epoch에 따른 손실값 추이 그래프 - RNN

### (4) Learning Curve : Epoch에 따른 학습, 검증 정확도 추이

```
plot_prediction_graph(history_rnn, "rnn")
```

[Figure 73] 학습 과정 중, Epoch에 따른 손실값 추이 그래프 - RNN 출력 코드



[Figure 74] 학습 과정 중, Epoch에 따른 정확도 추이 그래프- RNN

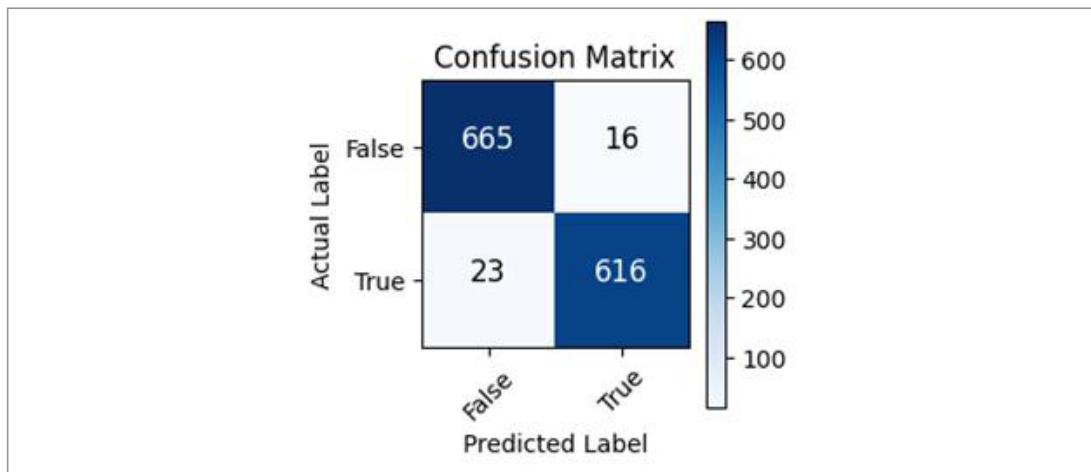
Validation의 Prediction Accuracy와 Loss 값의 추이를 보면 상당히 빠르게 (Epoch 5 이내에) 과적합되는 점을 알 수 있다. 즉 Epoch를 5~10 이내로 가져가는 것이 좋다. 단, 지나치게 빠르게 과적합될 경우 학습이 충분히 되지 않기 때문에 좋은 모델이 나오기 어렵다. 이 경우 다시 원점으로 돌아가서 RNN 모델을 재설계(i.e. Layer 수 변경 등)하는 것을 추천한다.

## - 결과 분석 및 해석 IV - CNN

### (1) 혼동 행렬

```
draw_confusion_matrix(cnn_model, x_test_exp, y_test, "cnn")
```

[Figure 75] 혼동 행렬 - CNN 출력 코드



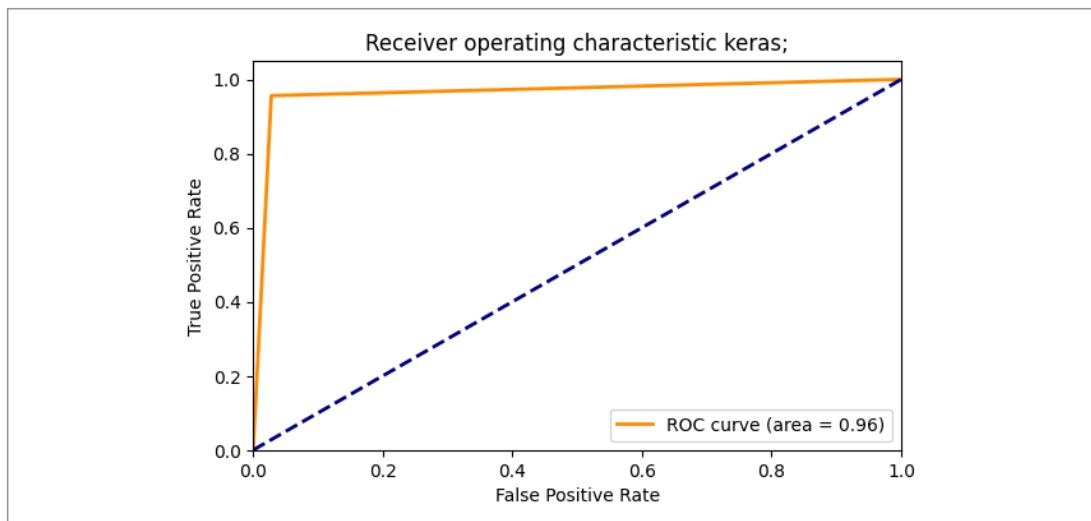
[Figure 76] 혼동 행렬 - CNN

이전의 모델들에 비해 상당히 분류를 잘하는 것을 확인할 수 있다. 정확도가 높을 뿐만 아니라, True Positive(정상을 정상이라고 판단함), False Positive(비정상을 비정상으로 판단함) 값이 매우 높으므로 우수한 분류기라고 평가할 수 있다.

### (2) ROC Curve

```
draw_roc(cnn_model, x_test_exp, y_test, "cnn")
```

[Figure 77] ROC Curve - CNN 출력 코드



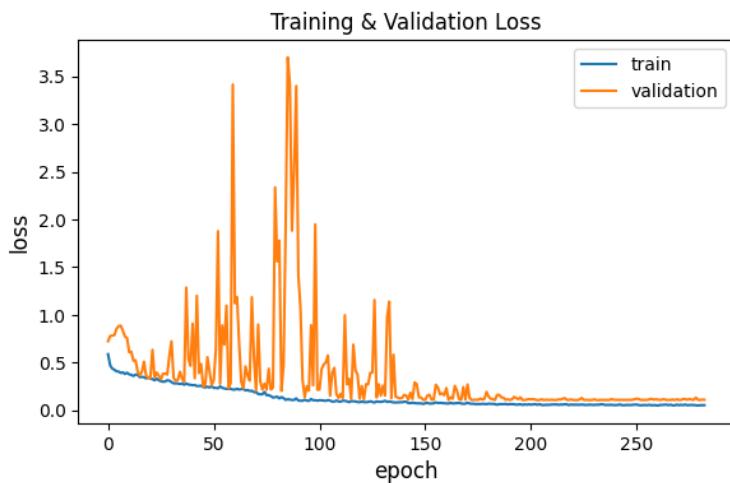
[Figure 78] ROC Curve - CNN

이전의 모델들에 비해 ROC 커브가 좌측 상단으로 가장 많이 꺾여 있어 Area가 가장 넓다. 가장 우수한 분류기로 판단할 수 있다. 곡선이 굽어지면 굽어질수록 AUC가 넓어지므로, 더욱 정확한 모델이기에, CNN이 타 모델에 비해 더 나은 모델임을 확인할 수 있다.

### (3) Epoch에 따른 학습, 검증 손실(Loss) 추이와 정확도

```
plot_loss_graph(history_cnn, "cnn")
```

[Figure 79] 학습 과정 중, Epoch에 따른 손실값 추이 그래프 - CNN 출력 코드

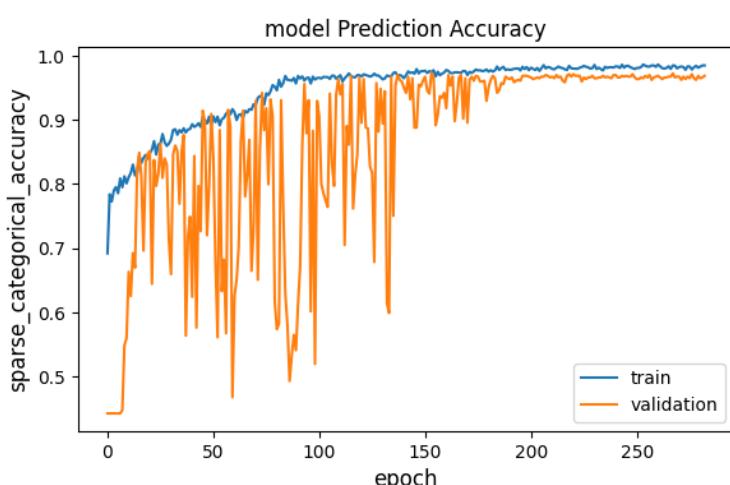


[Figure 80] 학습 과정 중, Epoch에 따른 손실값 추이 그래프 - CNN

### (4) Learning Curve: Epoch에 따른 학습, 검증 정확도 추이

```
plot_loss_graph(history_cnn, "cnn")
```

[Figure 81] 학습 과정 중, Epoch에 따른 정확도 추이 그래프 - CNN 출력 코드



[Figure 82] 학습 과정 중, Epoch에 따른 정확도 추이 그래프 - CNN

Validation의 Prediction Accuracy와 Loss 값의 추이를 보면 130 Epoch 까지는 급격하게 흔들리다가, 이후부터는 안정화되어가는 것을 확인할 수 있다. 200 Epoch부터는 성능 개선이 거의 없으며, 계속 학습을 진행할 경우 Validation Accuracy가 조금씩 떨어지는 것을 확인할 수 있는데 이는 모델이 과적합(Overfitting) 되고 있음을 보여준다. 따라서 그림을 보고 Epoch 수(본 실험에서는 epoch=200~300)를 조절해주는 것이 바람직하다.

결과적으로 본 Ford Dataset에 대해서는 4개의 모델 가운데, CNN모델이 가장 우수한 성능을 보임을 확인하였다. 정리하면 아래와 같다.

- Table: 모델 성능 비교 -

	정확도	Recall	F1 Score	Remarks
로지스틱 회귀 (1)	0.48	0.48	0.48	scikit-learn
로지스틱 회귀 (2)	0.49	0.49	0.49	scratch (numpy)
XGBoost	0.78	0.78	0.78	
RNN(LSTM)	0.82	0.82	0.82	
CNN	0.97	0.98	0.97	

### 3. 유사 타현장의 「Ford 엔진 진동 AI 데이터셋」 분석 적용

#### 3.1 본 분석이 적용 가능한 제조현장 소개

- 공장을 운영하는 제조 모든 분야에 적용이 가능하며, 특히 설비 운영 및 제품의 불량품 판정 등에 유용하게 적용될 수 있다.

#### 3.2 본 「Ford 엔진 진동 AI 데이터셋」 분석을 원용하여 타 제조현장 적용시, 주요 고려사항

- 정상/비정상 상태에 대한 Labeling이 필요하며, Labeling 작업은 정상/비정상 상태를 판단할 수 있는 현업 전문가 직접 Labeling을 하는 것을 추천 한다.

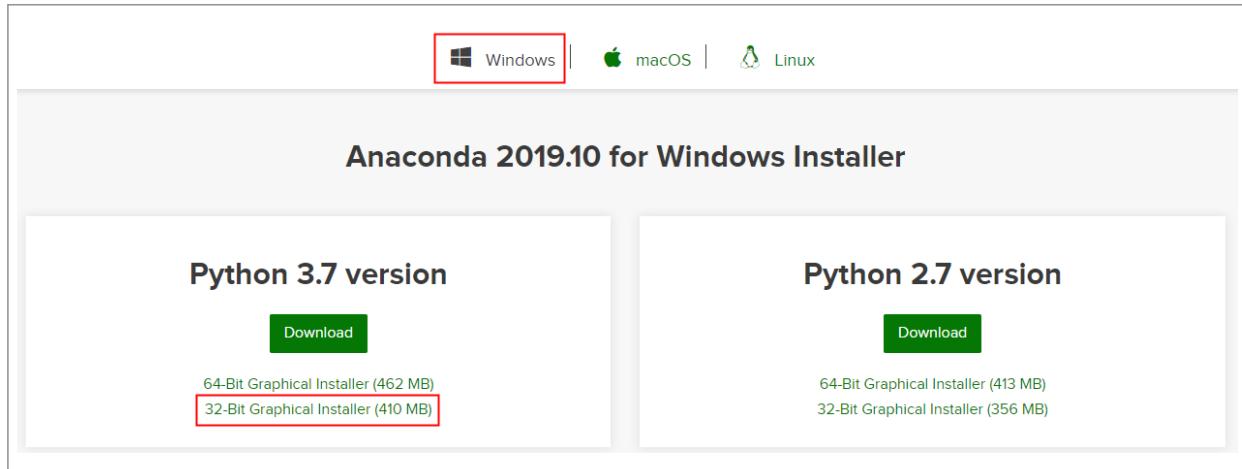
## ◎ 기본 권장 설치 방법

- Python을 활용하여 개발할 때 권장하는 가상환경 (Virtual Environment)을 설정하는 방법을 소개한다. 가상환경(Virtual Environment)이란? 가상환경은 말 그대로 가상의 독립적인 개발 환경을 만들어 주는 것으로, 중요한 이유는 Python 버전 관리와 패키지간의 충돌 방지가 있다. 프로젝트별로 필요한 Python 패키지만 설치해서 사용하면 되는데, 가상환경이 아닌 Base에 패키지를 모두 설치해버리면 불필요한 패키지까지 설치된 환경이 될 것이고, 가끔 dependency가 꼬여서 Error를 야기한다. 이러한 이유들로 프로젝트별로 각각의 가상환경을 만들고 이 환경에서 개발하는 것이 바람직하다.
- python은 현재 2.x 버전과 3.x 버전이 혼용되는 과도기에 있고 (물론, 요즘엔 대다수의 프로젝트들이 3.x로 많이 업그레이드를 하고 이를 support하고 있다), 때론, 2.x 버전의 python 환경에서 프로젝트를 개발해야 할 때도 있고, 3.x 버전의 python 환경에서 개발해야 할 때도 있다. 이럴 때마다, uninstall과 install하면서 python 버전을 바꿀 수는 없을 것이다.

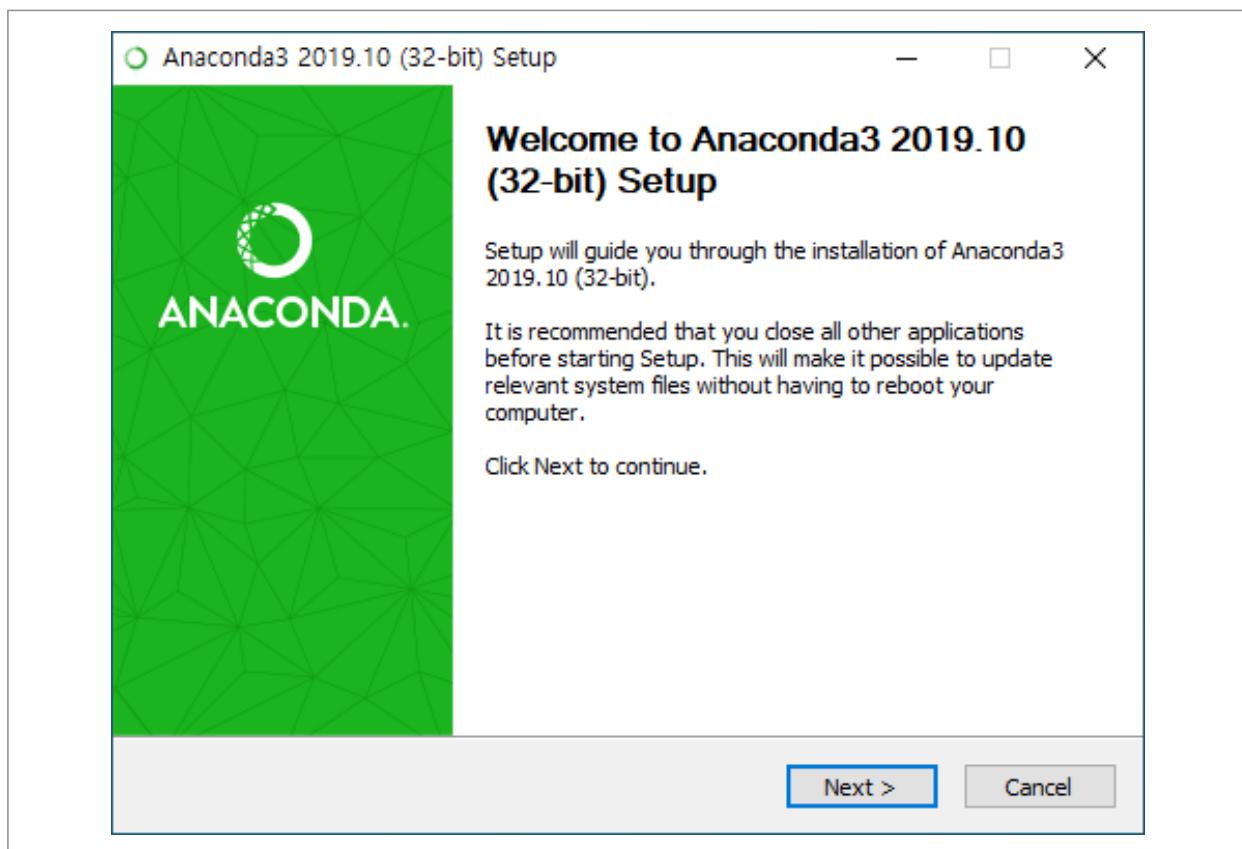
## ◎ 가상환경 설치하기

- Anaconda를 이용해 가상환경을 만들고 이를 관리하는 방법은 다음과 같다. Anaconda는 수백 개의 파이썬 패키지를 포함하고 있는 모듈로써, 먼저 아나콘다 배포판을 설치하기 위해 다음 URL로 이동한다. 파이썬은 현재 2 버전과 3 버전이 있는데, 파이썬 3.xx 버전을 사용한다.  
<https://www.anaconda.com/distribution>
- 파이썬 3.7 버전에도 다시 윈도우 64비트용 설치 파일과 32비트용 설치 파일이 있다. 보통은 사용 중인 PC에 64비트 윈도우가 설치돼 있으면 64비트용 설치 파일을 내려받고, 32비트 윈도우가 설치돼 있다면 32비트용 설치 파일을 내려받으면 된다.

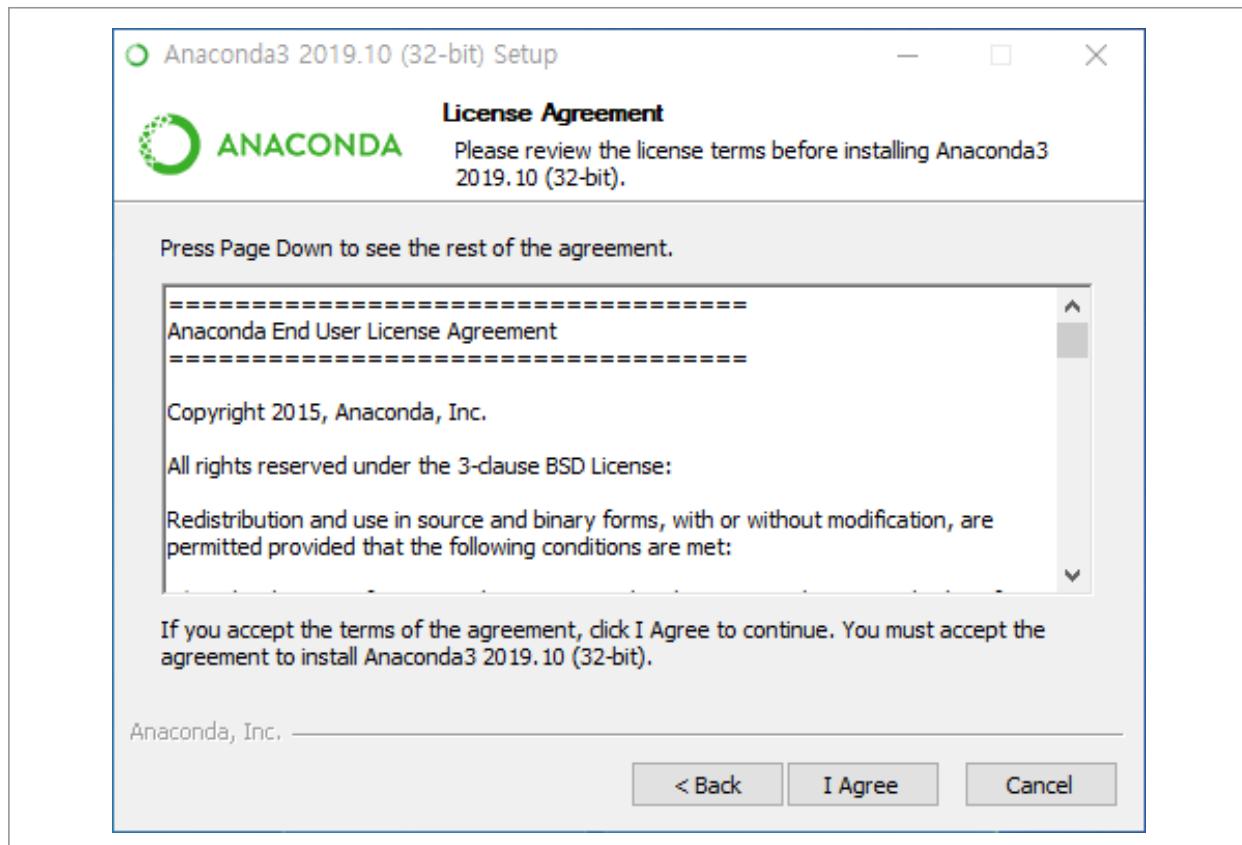
① Anaconda 설치 파일 다운로드



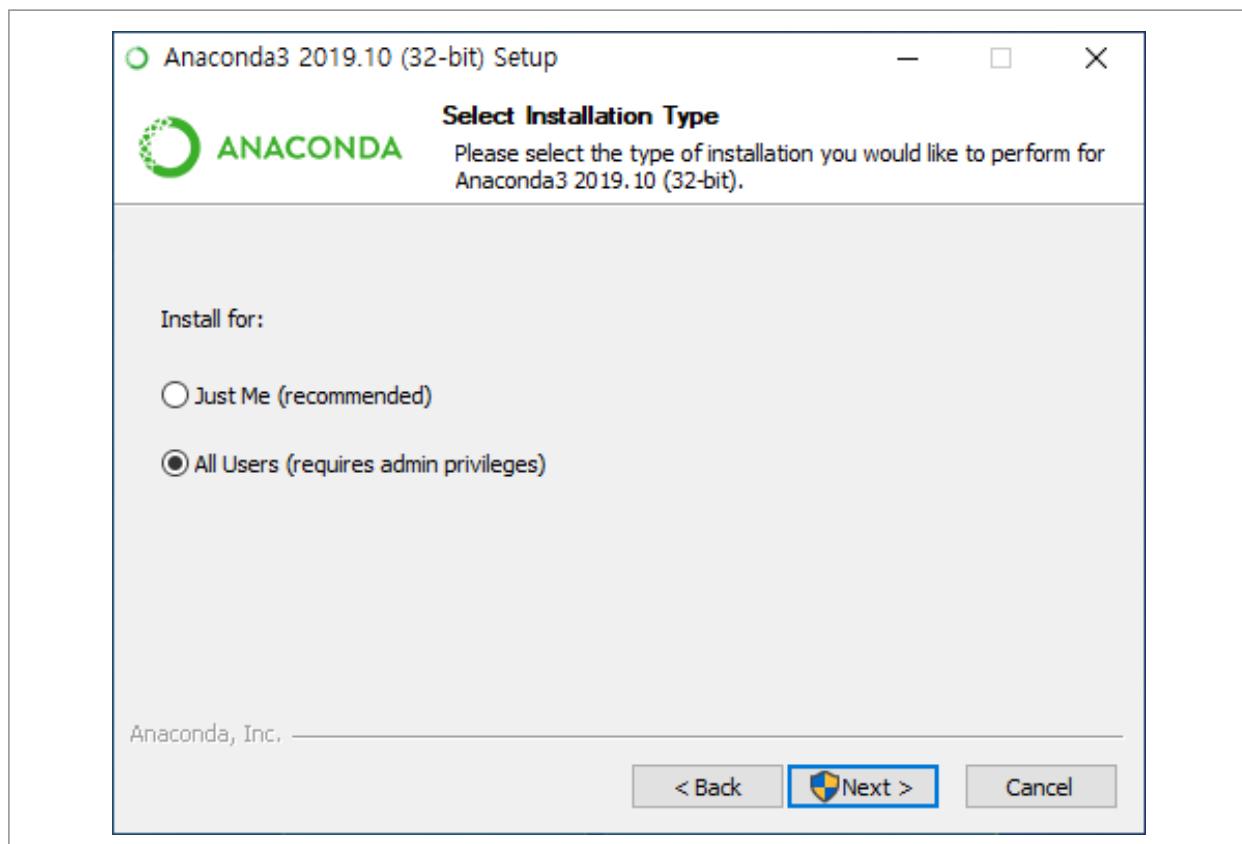
- ② 설치 파일을 정상적으로 내려받았다면 해당 파일(.exe)에 마우스 오른쪽 버튼을 클릭한 후 [관리자 권한으로 실행] 메뉴를 선택해서 설치를 시작한다. 그림 1.4와 같이 첫 번째 설치 화면이 나오면 [Next] 버튼을 클릭해 다음 단계로 진행한다.



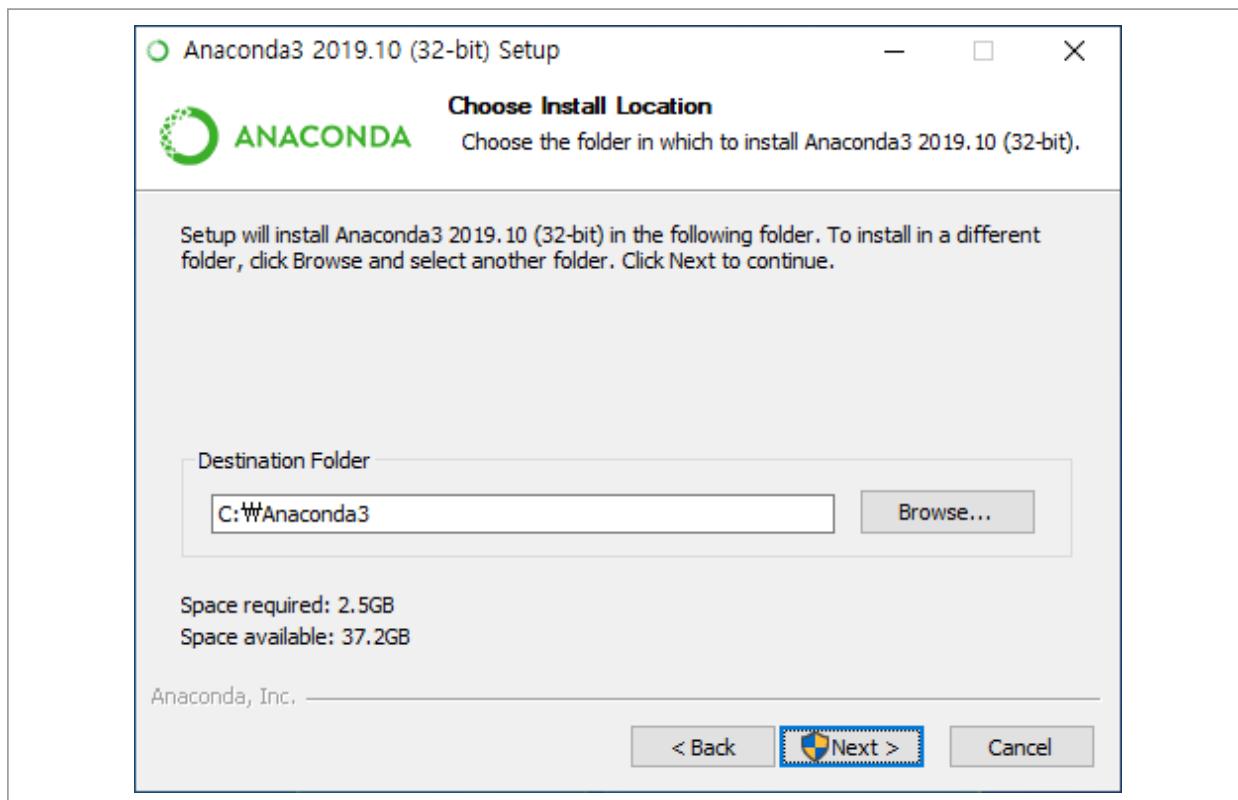
③ 라이선스 동의에 관한 내용이다. [I Agree] 버튼을 클릭해 다음 단계로 이동한다.



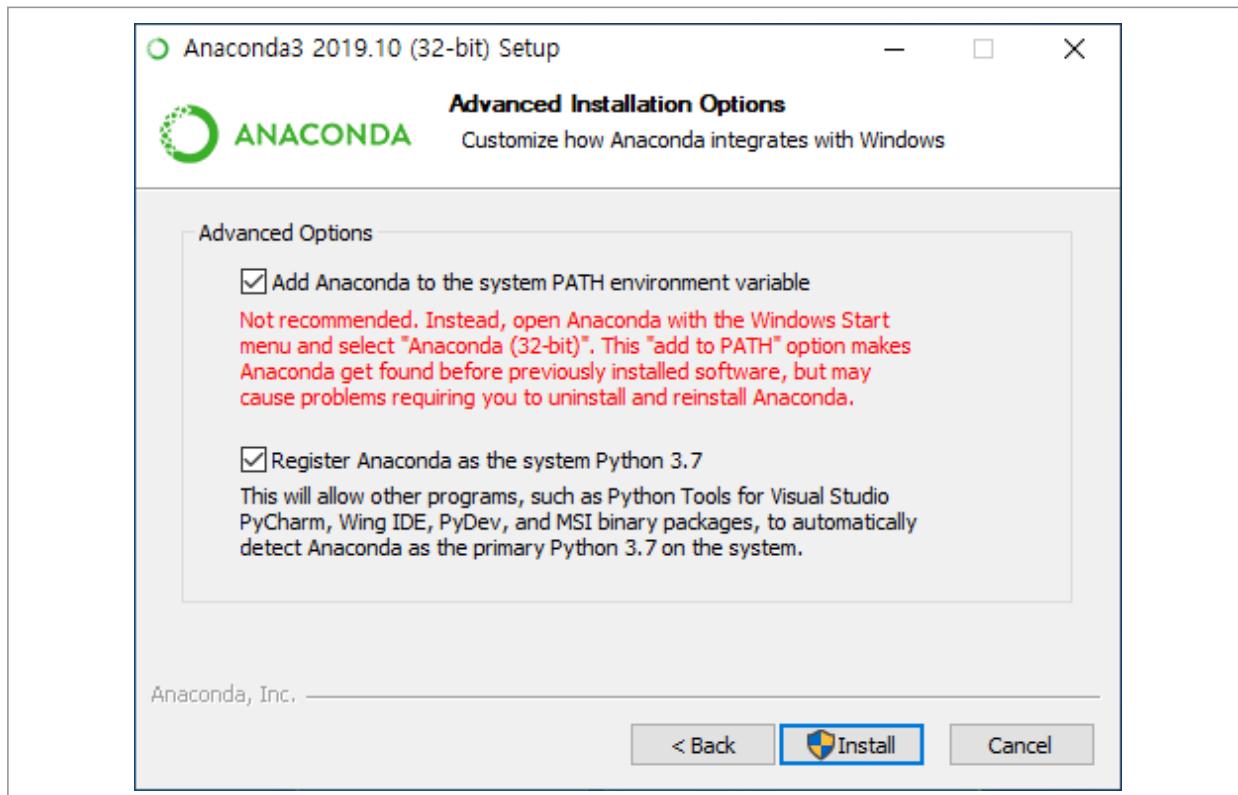
④ 설치 유형을 선택하는 단계에서는 [All Users]를 선택한 후 [Next] 버튼을 클릭한다.



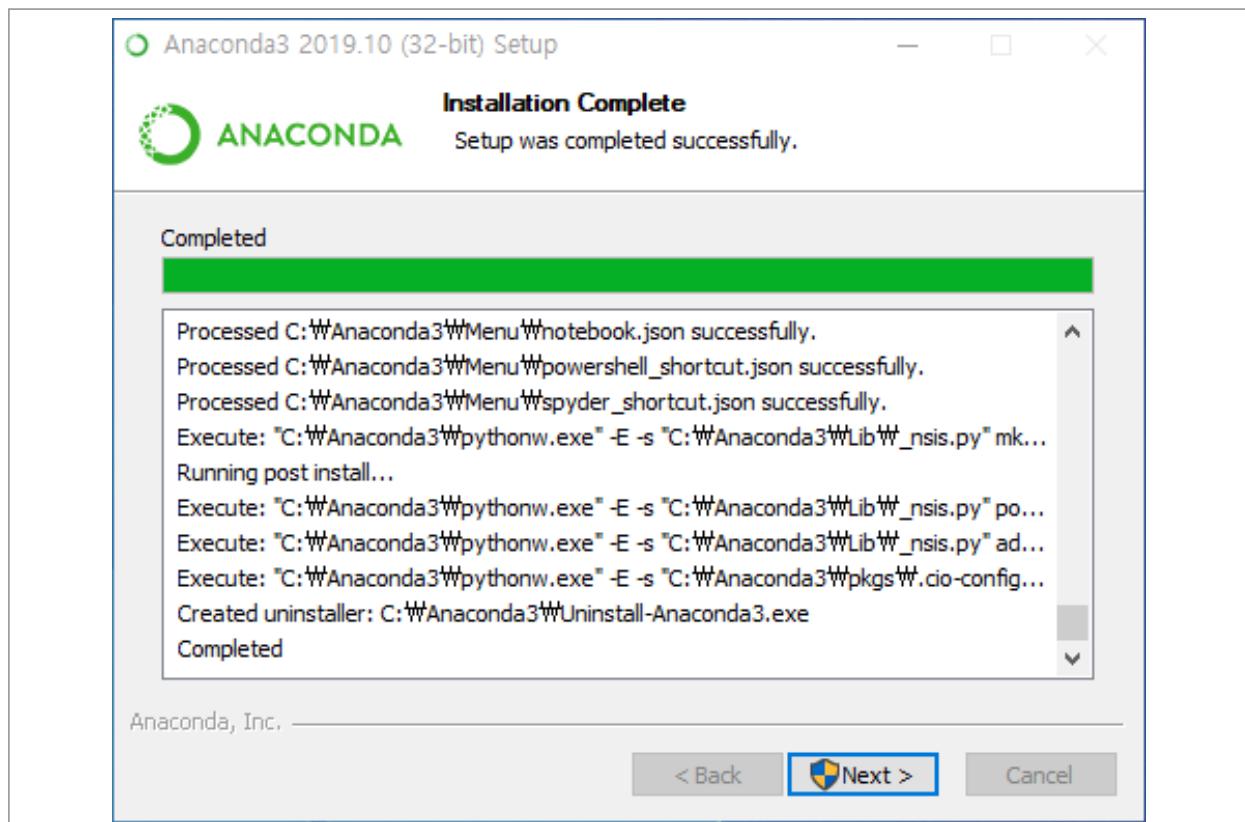
- ⑤ 아나콘다 배포판이 설치될 디렉터리를 선택하는 화면에서는 [Destination Folder]를 ‘C:\Anaconda3’으로 변경한 후 [Next] 버튼을 클릭한다.



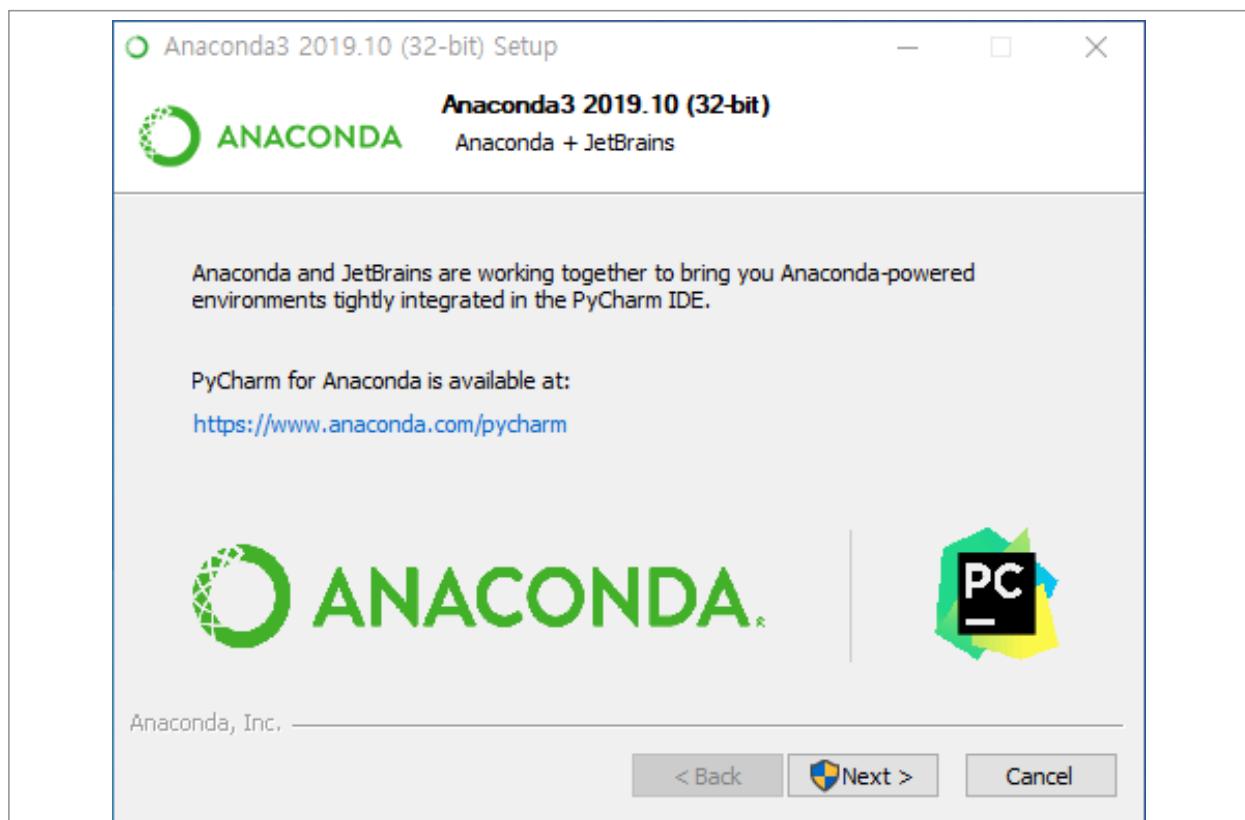
- ⑥ 추가옵션 선택화면에서 그림과 같이 두 가지 옵션을 모두 선택한 후 [Install] 버튼을 클릭해 다음 단계로 이동한다.



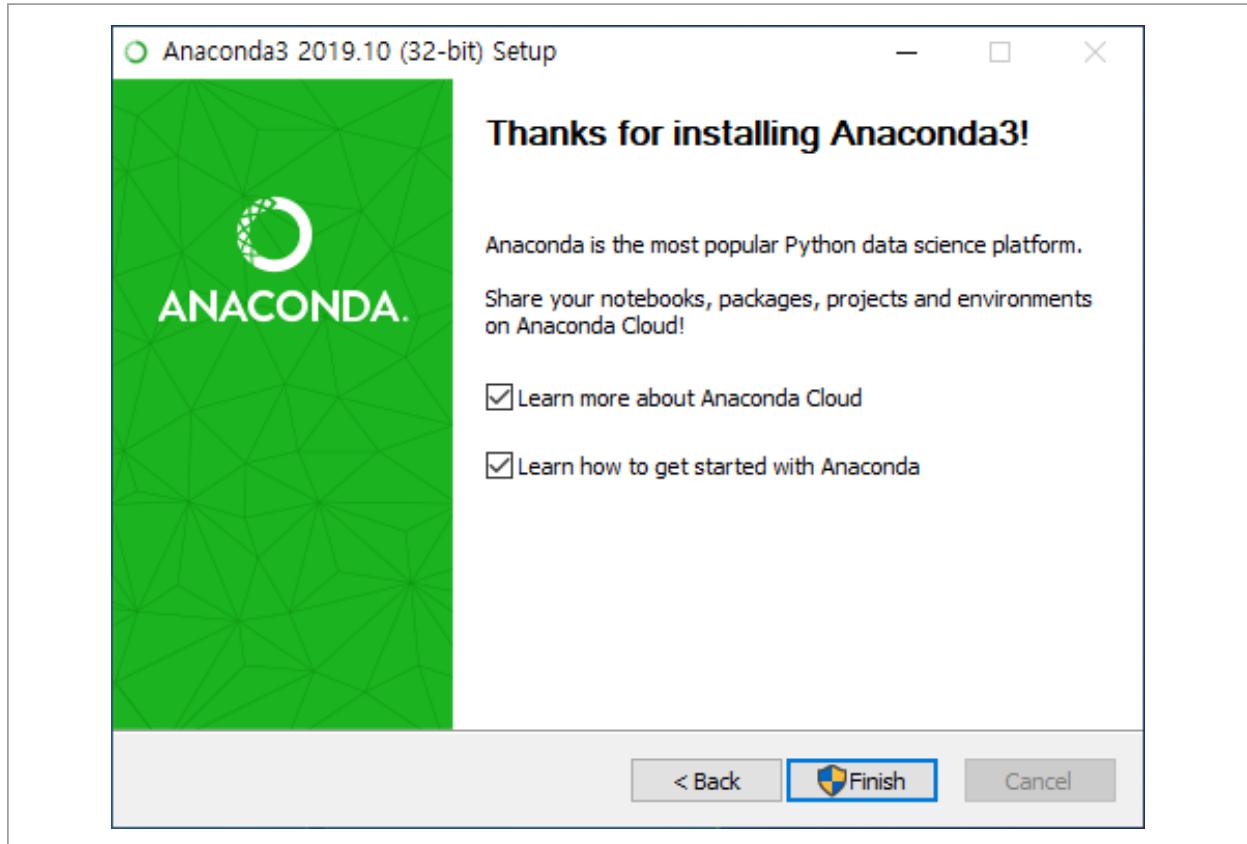
⑦ 설치가 완료되면 화면에 'Completed'라는 메시지가 출력됩니다. 이때 [Next] 버튼을 눌러 다음 단계로 이동한다.



⑧ 그림 상의 [Next] 버튼을 클릭해 다음 단계로 넘어간다.

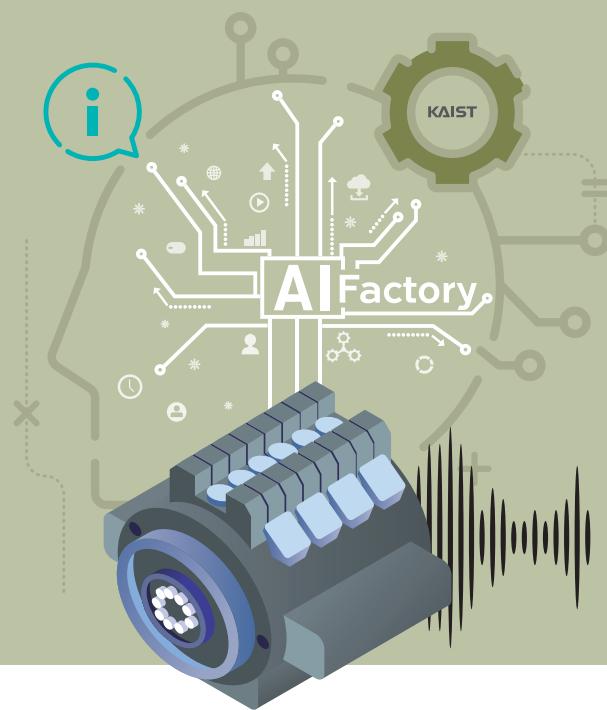


- ⑨ 이로써 아나콘다 배포판의 모든 설치 과정이 완료되었으며, 그림과 같이 [Finish] 버튼을 클릭해 설치 과정을 종료한다.



- ⑩ Anaconda를 설치하셨으면, 터미널에서 conda라는 명령어를 통해 가상환경을 만들고, 패키지 관리를 할 수 있다.

# 「Ford 엔진 진동 AI 데이터셋」 분석실습 가이드북



중소벤처기업부



스마트제조혁신추진단



34141 대전광역시 유성구 대학로 291 한국과학기술원(KAIST)  
T. (042)350-2114 F. (042)350-2210(2220)