반도체 데이터 분석

1. 이상적인 모델 가정

본 프로젝트는 반도체 칩 불량 여부를 사전에 확인하는 모델을 개발하여 이윤을 극대화하는 것을 목적으로 한다. 모델 개발은 여러 변수와 샘플들로 이루어진 복잡한 반도체 데이터를 처리하기 위해 기계 학습을 이용하여 진행할 것이며 기계 학습의 각 절차에 준수하며 개발을 완료할 것이다.

먼저, 개발에 앞서 여러 가지 경우의 수를 고려하여 반도체 개발에 드는 비용을 구해보았다. 정상적으로 동작하는 반도체 칩이 정상적으로 동작함을 모델을 통해 예측에 성공한 경우를 가정해 보자. 하나의 칩을 생산하고 테스트하는데 1100 원이 든다. 이러한 칩의 판매 가격은 2000 원이므로 총 900 원의 이익을 얻을 수 있다. 다음으로 정상적으로 동작하는 반도체 칩이 정상적으로 동작함을 모델을 통해 예측하는데 실패한 경우를 가정해 보자. 이 경우 생산 비용 총 100 원의 비용이 발생하고 테스트를 진행하지 않고 미리 폐기 처분하여 테스트를 통해 발생하는 손해는 없으므로 총 100 원의 손해가 발생한다. 다음으로 불량 칩을 생산하였다고 가정하자. 모델을 통해 불량칩을 사전에 탐지하는데 성공하면 테스트 비용 1000 원을 아끼고 생산 비용 100 원의 손해만 발생한다. 반대로 불량칩을 사전에 탐지하는데 실패하면 정상칩이라고 생각하고 테스트하는데 드는 비용 1000 원과 생산 비용 100 원을 합쳐 총 1100 원의 손해가 발생한다. 반면 테스트를 통해 불량이라고 판별될 것이기 때문에 얻을 수 있는 이익은 없다. 즉, 총 1100 원의 손해가 발생한다고 볼 수 있다. 이를 표로 나타내며 다음과 같다.

	정상칩	불량칩
정상칩이라 예측	900 원	-1100 원
불량칩이라 예측	-100 원	-100 원

이제 데이터를 살펴보았다. 기계학습에서 데이터는 보통 training set 과 test set 으로 나누어 처리한다. Training set 을 통해 모델을 만들고 test set 을 통해 모델을 검증하는 과정을 거친다. 따라서 본격적으로 데이터를 다루어 모델을 만들기 전에 training set 을 바탕으로 대략적으로 정상칩과 불량칩의 비율이 어떻게 되는지 알기 위해 데이터의 샘플들을 통해 정상칩과 불량칩의 개수를 세어보았다. 다음 코드를 사용하였다.

```
sum_defect = sum(data['Label'] == 'defect')
sum_normal = sum(data['Label'] == 'normal')
```

```
print(sum_defect, sum_normal)
```

```
740 10751
```

이를 통해 총 11491 개의 칩들 중 정상칩의 개수가 10751 개 불량칩의 개수가 740 개라는 사실을 확인하였다. 이와 같은 샘플들이 주어져 있을 때 여러 모델을 가정하여 총 이득의 추이를 알아보자. 먼저, 모델을 만들지 않고 모든 경우에 대해 직접 테스트하는 경우를 고려해 보자. 이 경우 모든 정상칩에 대해 900 원의 이익이 발생할 것이고 모든 불량칩에 대해 1100 원의 손해가 보게 된다. 이를 다음 코드를 통해 계산하면 총 이익은 다음과 같다.

```
cost = 900 * sum_normal - 1100 * sum_defect
```

```
print(cost)
```

```
8861900
```

총 8861900 원의 이익이 발생하였다. 다음으로, 이상적인 모델을 가정해 보자. 이상적인 모델을 가정한 경우 모든 칩을 정확하게 예측할 수 있다. 그러면 모든 정상칩에 대하여 900 원의 이익을 얻고 불량칩의 경우에 대하여 100 원의 손해를 보게 된다. 다음 코드를 사용하여 계산한 결과 총 이득은 다음과 같다.

```
cost = 900 * sum_normal - 100 * sum_defect
```

```
print(cost)
```

```
9601900
```

즉, 총 9601900 원의 이익이 발생함을 확인할 수 있다. 다음으로 Precision 이 90%, Recall 이 90%인 경우를 생각해 보겠다. 불량칩을 판별한 경우를 Positive, 정상칩을 판별한 경우는 Negative 로 하고 계산한다. 이를 정리하여 표로 나타내면 다음과 같다.

	불량칩	정상칩
불량칩이라 예측	-100 원 (TP)	-100 원 (FP)
정상칩이라 예측	-1100 원 (FN)	900 원 (TN)

```
#마지막으로 Precision이 90%이고 Recall이 90%일 때 총 이득을 계산한다. Precision이 90%이므로 정상 칩의 90%를 정확하게 예측한다고 할 수 있다. 따라서 이를 통해 TP와 TN을 구할 수 있다.
TP = sum_defect * 0.9
FN = sum_defect * 0.1
#다음으로 Recall이 90%가 되어야 하므로 Precision을 통해 얻은 TP를 바탕으로 Recall을 계산 해 본다. Recall은 TP/(TP+FP)로 계산할 수 있으므로 FP = TP / 9 이다.
FP = TP / 9
#TN은 전체 갯수에서 TP, FN, FP를 뺀 값과 동일하다.
TN = sum_defect + sum_normal - TP - FN - FP
```

+ 코드 + 텍스트

```
#각각을 출력하면 다음과 같다.
print(TP, FN, FP, TN)
#즉, TP = 666, FN = 74, FP = 74, TN = 10677이므로 이를 통해 총 이득을 계산할 수 있다.
profit = 900 * TN - 100 * TP - 100 * FP - 1100 * FN
```

```
666.0 74.0 74.0 10677.0
```

코드를 위와 같이 작성하여 각 경우에 속하는 반도체 칩의 개수를 구하였다. 위 표를 활용하여 총 이익을 계산하면 다음과 같다.

```
print(profit)
```

```
9453900.0
```

즉, Precision 이 90%이고 Recall 이 90%인 모델을 가정하면 9453900 원의 이익을 얻을 수 있음을 확인하였다. 이를 통해 알 수 있는 점은 예측이 정확해지면 얻을 수 있는 이익이 증가한다는 점이다.

2. EDA 및 데이터 전처리

이와 같이 여러 모델을 가정하여 이익을 계산해 보았고 예측이 정확해지면 얻을 수 있는 이익이 커진다는 점을 확인하였다. 이제 본격적으로 높은 예측율을 가지는 모델을 만들 차례이다. 그 전에 앞서 지금 가지고 있는 row 데이터에 대해 EDA, 즉 탐색적 데이터 분석을 수행하고 전처리 하는 과정을 거칠 것이다. 이 과정은 데이터를 가공하여 좋은 모델을 만들기 위해 필수적인 과정이다. 탐색적 데이터 분석은 먼저 데이터를 대략적으로 확인하는 것에서 출발한다. 데이터는 다음과 같다.

```
data
```

	Label	v001	v002	v003	v004	v005	v006	v007	v008	v009	...	v581	v582	v583	v584	v585	v586	v587	v588	v589	v590
0	normal	2872.0667	2466.5526	2125.6577	989.1645	-2.7843	100.0	101.4519	0.1211	1.5088	...	NaN	NaN	0.5026	0.0152	0.0036	3.0002	0.0235	0.0214	0.0068	201.6557
1	normal	2925.8098	2541.5765	2234.0984	1281.4768	-3.0935	100.0	102.7180	0.1217	1.4794	...	NaN	NaN	0.5028	0.0148	0.0039	2.9137	0.0178	0.0086	0.0019	22.1478
2	normal	2985.1397	2434.9879	2114.9636	984.9040	-2.3782	100.0	104.8891	0.1280	1.3588	...	NaN	NaN	0.4963	0.0094	0.0032	1.8878	0.0289	0.0153	0.0053	38.0891
3	normal	3205.5703	2354.2453	2156.7551	2348.6859	-2.2280	100.0	92.2493	0.1256	1.4360	...	0.0005	NaN	0.4980	0.0180	0.0046	3.5536	0.0140	0.0150	0.0060	210.6087
4	normal	3065.9593	2543.6703	2180.5305	1190.3008	-2.2051	100.0	101.0354	0.1260	1.5015	...	0.0029	NaN	0.5024	0.0132	0.0032	2.6338	0.0259	0.0114	0.0046	45.1529
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
11486	normal	3024.2073	2424.2201	2240.4024	1566.7072	-2.6379	100.0	100.3331	0.1203	1.3916	...	NaN	NaN	0.4986	0.0235	0.0057	4.7449	-0.0031	0.0269	0.0081	442.7744
11487	normal	2949.6321	2510.9854	2181.3872	1297.5681	-2.7171	100.0	115.1750	0.1265	1.4105	...	NaN	247.1294	0.4992	0.0109	0.0026	2.2044	0.0321	0.0229	0.0084	139.2623
11488	normal	3038.8449	2456.6541	2190.1101	825.8933	-3.1172	100.0	114.8281	0.1217	1.4689	...	0.0033	142.2710	0.5025	0.0218	0.0049	4.2675	0.0313	0.0167	0.0053	74.2087
11489	normal	2993.0400	2504.6600	2229.3333	1553.3158	1.5123	100.0	102.7800	0.1235	1.4479	...	0.0038	80.4663	0.4984	0.0161	0.0039	3.2251	0.0140	0.0112	0.0038	80.4663
11490	normal	3108.1682	2269.4657	2199.5594	1489.7562	-2.0973	100.0	100.1003	0.1222	1.5727	...	0.0042	90.7147	0.4968	0.0122	0.0028	2.4340	0.0175	0.0163	0.0053	83.3342

11491 rows × 591 columns

먼저, 각 데이터의 분포를 확인하고 시각화 하는 것이 중요하다. 제일 처음 해야 하는 작업은 데이터의 요약을 통해 데이터가 수치형 변수로 이루어져 있는지 아니면 범주형 분포로 이루어져 있는지 그리고 대략적인 분포를 확인하고 시각화 하는 작업이다. 다음 코드를 통해 개략적으로 확인할 수 있다.

```
#먼저, 데이터를 요약한다.
data.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 11491 entries, 0 to 11490
Columns: 591 entries, Label to v590
dtypes: float64(590), object(1)
memory usage: 51.8+ MB
```

이를 통해 알 수 있는 점은 모든 feature 가 float64 형, 즉 수치형 변수로 이루어져 있다는 점이다. (Object(1)은 label 를 의미한다.) 물론 숫자로 써져 있어도 이산-범주형에 해당하는 feature 가 있을 가능성도 존재하지만 float64 형은 int64 형과 다르게 소수점을 포함하는 연속적 데이터를 나타내므로 이 가능성도 배제 가능하다. 따라서 모든 변수를 연속-수치형 변수라고 생각할 것이다. 이제 각 feature 의 데이터 분포를 알아보고 시각화 할 것이다. 알아보기 전에 매우 많은 feature 들이 존재하기 때문에 한 눈에 알아보기 쉽게 모두 나타내는 것은 어렵다는 점을 유의해야 한다. 그러므로 각 feature 에 해당하는 대푯값을 이용해서 나타내는 것이 더 효율적이다. 연속-수치형 변수에서 가장 많이 쓰이는 대푯값은 평균과 분산이다. 모든 변수들의 대푯값을 확인하기 위해 다음 코드를 이용하였다.

```
data.describe()
```

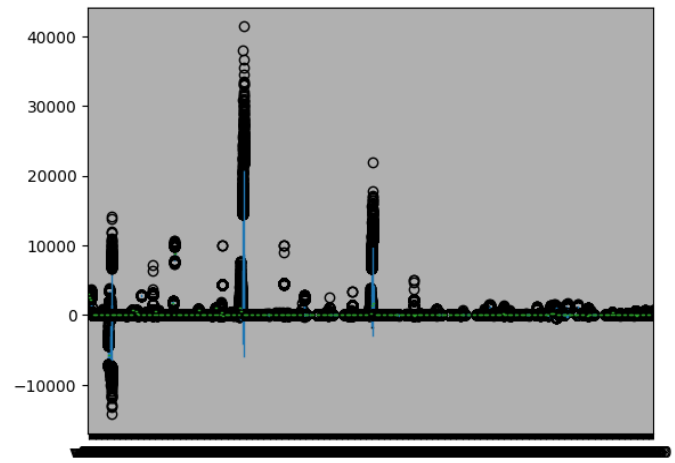
	v001	v002	v003	v004	v005	v006	v007	v008	v009	v010	...	v581	v582	v583	
count	11449.000000	11434.000000	11381.000000	11391.000000	11381.000000	11392.0	11381.000000	11419.000000	11476.000000	11473.000000	...	4548.000000	4470.000000	11482.000000	1
mean	3014.536549	2496.012299	2200.423984	1395.618317	4.378355	100.0	101.149555	0.121752	1.463317	-0.000814	...	0.005381	98.862766	0.500075	
std	73.986159	80.756122	29.337407	443.624918	57.263561	0.0	6.273182	0.009142	0.074339	0.015155	...	0.003026	88.888180	0.003381	
min	2743.240000	2162.870000	2060.660000	0.000000	-3.589900	100.0	72.388200	0.000000	1.158800	-0.050400	...	-0.001700	-52.443200	0.485300	
25%	2964.852100	2446.607350	2181.844900	1082.444400	-2.877600	100.0	97.207300	0.121800	1.413000	-0.011000	...	0.003500	46.980375	0.497900	
50%	3012.049100	2496.437850	2201.045900	1328.116000	-2.374000	100.0	101.328800	0.123100	1.463400	-0.001200	...	0.004600	70.585550	0.500100	
75%	3060.972500	2546.770425	2219.541700	1636.805550	-1.790400	100.0	105.097700	0.124500	1.514000	0.008800	...	0.006600	120.410400	0.502300	
max	3356.350000	2872.557000	2315.266700	3715.041700	1112.472800	100.0	129.252200	0.130000	1.719800	0.070700	...	0.028600	817.456300	0.512800	

8 rows × 590 columns

이와 같이 표의 형태로 각 feature 의 대푯값을 나타내었다. 그러나 이 코드를 이용하더라도 모든 데이터의 대푯값을 나타내기는 번거롭다. 어떻게 이 변수들의 분포를 나타내는 것이 깔끔할 것인가 고민해 본 결과 Boxplot 을 이용하기로 결심하였다.

```
data.boxplot()
```

```
<Axes: >
```

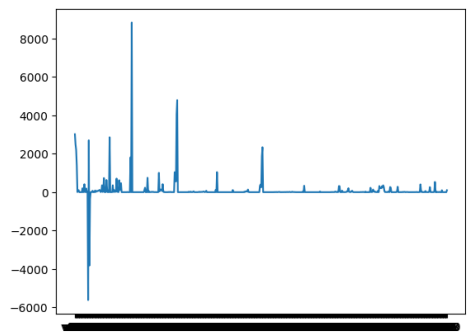


수많은 데이터의 분포를 확인하기 위해서는 Boxplot 을 이용하였지만 그림을 통해 확인할 수 있다시피 너무 많은 feature 들에 의해 그래프가 뭉개진 것을 알 수 있다. 하지만 몇몇 feature 들을 제외하면 대부분 비슷한 값의 범위를 형성하고 있다는 점을 알 수 있다.

더 확실하게 범주의 분포를 확인하는 방법은 평균과 분산을 표로 나타내는 것이다. 두 값이 연속-수치형 변수의 대표적인 대푯값들이기 때문이다.

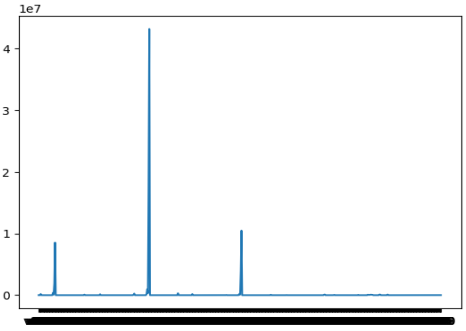
```
plt.plot(data_var.mean())
```

```
[<matplotlib.lines.Line2D at 0x7f149a3b0be0>]
```




```
plt.plot(data_var.var())
```

```
[<matplotlib.lines.Line2D at 0x7f1499b34bb0>]
```



평균과 분산을 이용해 대략적인 분포를 알아본 결과 몇몇 변수들의 평균과 분포가 아주 크고 나머지 변수들은 분산이 0 에 가깝거나 평균이 0 에 가까운 것을 알 수 있다. 분산의 크기를 통해 몇몇 변수들이 기계학습에 큰 영향을 끼칠 것이라는 점을 미리 알 수 있다. 평균의 크기를 통해 데이터를 scaling 을 통해 조정해 줄 필요가 있다는 사실을 인지할 수 있다.

이제 다음으로 이상치를 탐색하여 처리할 것이다. 이상치의 종류에는 일반적인 데이터의 범위를 벗어나는 데이터들과 알 수 없는 이유로 측정이 제대로 되지 않아 측정되지 않은 결측치가 존재한다. 먼저, 결측치를 다루어 보도록 하자.

데이터를 보면 중간 중간에 Nan 이 존재함을 알 수 있다. Nan 으로 쓰여진 데이터들이 결측치이다. 먼저, 결측치가 어떻게 분포되어 있는지 확인한다.

```
#각 feature에 대하여 결측치를 조사하면 다음과 같다.
data.isna().sum(axis=0)
```

```
Label      0
v001      42
v002      57
v003     110
v004     100
        ...
v586       7
v587       8
v588      11
v589       5
v590       6
Length: 591, dtype: int64
```

```
data.isna().sum(axis=1)
```

```
0         29
1         30
2         32
3         28
4         27
          ..
11486     30
11487     18
11488     25
11489     24
11490     25
Length: 11491, dtype: int64
```

위 결과는 각 feature 별 그리고 각 sample 별로 나타낸 결측치의 개수이다.

```
print('# of variables with missing:', (data.isna().sum(axis=0)>0).sum() )
print('# of samples with missing:', (data.isna().sum(axis=1)>0).sum() )
print('# of total missing:',data.isna().sum().sum() )
```

```
# of variables with missing: 538
# of samples with missing: 11491
# of total missing: 307010
```

총 11491 * 591 = 6791181 개의 sample 들 중 307010 개의 결측치가 존재함을 확인할 수 있다. 비율도 따지면 대략 2%정도의 매우 많은 결측치가 존재하는 것이다. 결측치를 처리하는 방법에는 sample 혹은 feature 를 제거하거나 아니면 평균치 혹은 중간치로 대치하는 방법이 존재한다. 두 방법을 어떻게 쓸 것인지 알아보기 위해서는 결측치들의 분포를 확인해 볼 필요가 있다. 일단 결측치가 가장 큰 sample 과 feature 의 결측치 개수를 다음과 같이 알아보았다.

```
[17] max(data.isna().sum(axis=0))

     10489
```

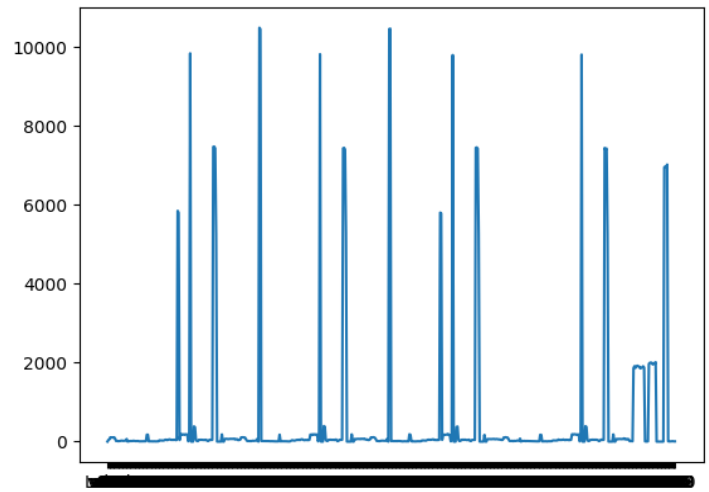
```
[20] max(data.isna().sum(axis=1))

     148
```

Sample 들의 관점에서 보았을 때 결측치의 수가 많은 경우가 존재하지만 과반수를 넘어서서 데이터 해석에 영향을 끼칠 정도로 많지는 않다고 볼 수도 있다. 반면 feature 들의 관점에서 보았을 때 feature 들을 제거하는 방법은 반드시 필요해 보인다. 위에서 얻은 결과에 따르면 특정 feature 는 결측치가 대략 90% 존재한다는 점을 확인할 수 있다. 이 경우 너무 많은 결측치로 인해 이 feature 에서 유의미한 정보를 뽑아 낼 수 있다고 보기 어렵다. 따라서 결측치 제거를 통해 이와 같이 유의미하게 결측치가 많은 경우는 제거해 주어야 한다. 보다 자세하게 결측치의 분포를 알기 위해 다음과 같이 분포를 그래프로 나타냈다.

```
plt.plot(data.isna().sum(axis=0))
```

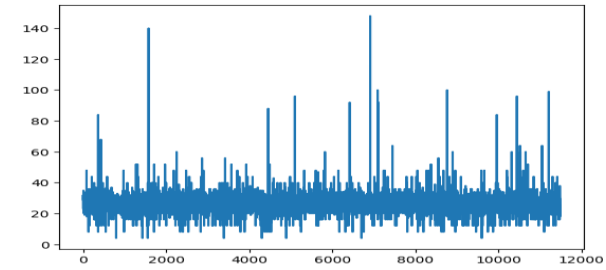
```
[<matplotlib.lines.Line2D at 0x78ce333559c0>]
```



그림을 통해 각 feature 에 대해 결측치의 수를 분포로 나타내면 다음과 같다.

```
plt.plot(data.isna().sum(axis=1))
```

```
[<matplotlib.lines.Line2D at 0x78ce32b94550>]
```

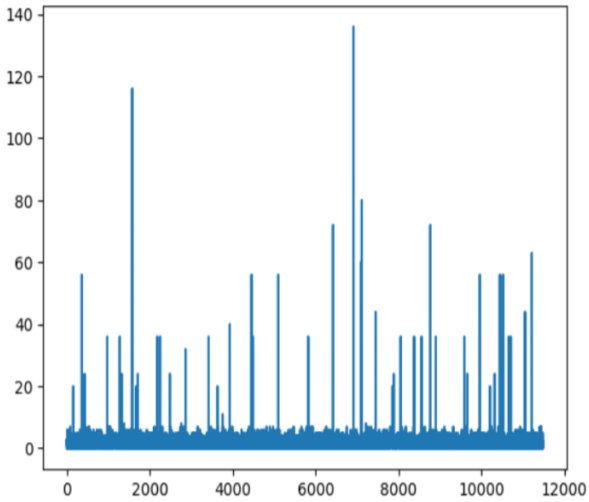


각 sample 들의 경우에 대해 나타냈을 경우는 위와 같다. 두 경우 모두 이상치가 outlier 에 속한다고 볼 수 있을 만큼 많은 경우가 존재한다는 것을 쉽게 확인할 수 있다. 이러한 경우에 대해 결측치를 제거해 주어야 한다는 결론을 내릴 수 있다. Feature 의 경우 그래프를 통해 대략 300 개 이상의 결측치가 존재하면 outlier 에 해당될 정도라고 할 수 있고 sample 의 경우는 대략 40 개 이상부터 결측치가 많이 존재한다고 생각할 수 있고 이를 바탕으로 결측치를 제거해 줄 것이다. 먼저, 300 개 이상의 결측치를 가진 경우 feature 를 제거해 주었다.

```
data_clean = data.dropna(axis = 1, thresh = 11491 - 300)
print( data_clean.shape )
print('# of total missing:',data_clean.isna().sum().sum() )
```

```
(11491, 531)
# of total missing: 22946
```

그 결과 남은 feature 의 수는 531 개이다. 590 개 중 59 개의 feature 가 제거되었다. 많은 feature 들이 제거되었지만 추가적으로 sample 를 몇 개 제거해 줄 것이다. 결측치를 새면 여전히 outlier 로 보일정도로 많은 경우가 존재하기 때문이다. 아래 그림을 통해 알 수 있다.



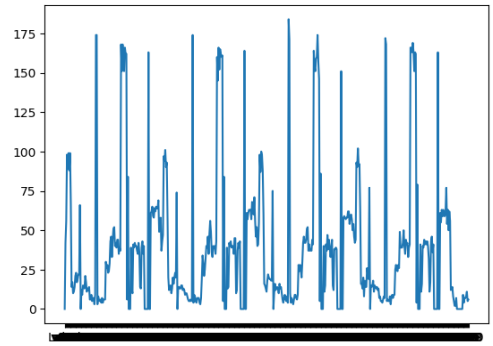
그래프를 통해 결측치의 숫자가 대부분 약 7 인 것으로 보이므로 추가적으로 결측치가 7 개 이상의 경우 결측치를 제거해 줄 것이다. 최종적으로 결측치를 제거한 결과는 다음과 같이 같다.

```
data_clean = data_clean.dropna(axis = 0, thresh = 531 - 7)
print( data_clean.shape )
print('# of total missing:',data_clean.isna().sum().sum() )
```

```
(11441, 531)
# of total missing: 20952
```

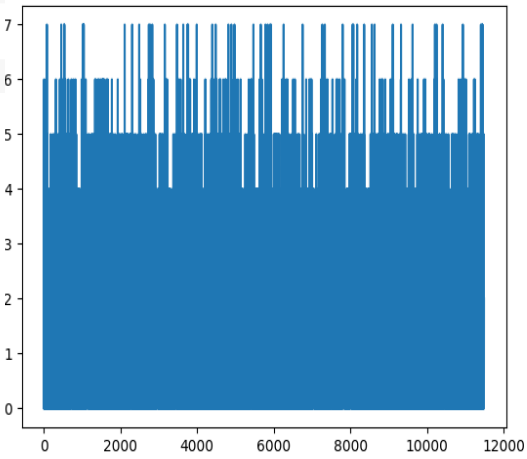
```
plt.plot(data_clean.isna().sum(axis=0))
```

```
[<matplotlib.lines.Line2D at 0x78ce21a17c70>]
```



```
plt.plot(data_clean.isna().sum(axis=1))
```

```
<matplotlib.lines.Line2D at 0x78ce22d2aa40>]
```



결측치의 개수가 매우 많이 균등해졌음을 확인할 수 있다. 이제 남은 결측치들은 제거해 주기 보다는 대치해 주는 것이 좋다. 결측치가 많이 존재하지 않고 대치할 값을 찾는데 필요한 데이터의 양이 충분하기 때문이다.

```
#결측치 대치
data_clean = data_clean.fillna( data_clean.mean() )
data = data_clean
print( data.shape )
print('# of total missing:',data.isna().sum().sum() )
```

```
<ipython-input-55-a23d686e38ca>:2: FutureWarning: The de
  data_clean = data_clean.fillna( data_clean.mean() )
(11441, 531)
# of total missing: 0
```

대치한 결과 더 이상 결측치들은 존재하지 않는다. 이제 다음으로 일반적인 범위를 벗어나는 이상치들을 탐색할 것이다. 이를 위해 정규분포에서 쓰이는 zscore 로 데이터를 나타내어 표현할 것이다. (iloc() 함수를 통해 label 을 없애 주었는데 그 이유는 label 이 있을 경우 오류가 발생하기 때문이다.)

```
#다음으로 이상치를 탐색한다.
data = data.iloc[:,1:]
sp.stats.zscore(data)
```

	v002	v003	v004	v005	v006	v007	v008	v009	v010	v011	...	v577	v578	v583	v584	v585	v586	v587	v588	v589	v590
0	-0.364374	-2.557205	-0.920873	-0.125614	NaN	0.048790	-0.077958	0.610534	-0.586106	-0.425009	...	-0.533300	-0.400723	0.747130	0.003484	-0.060743	-0.009736	0.166574	0.571029	0.540021	1.077433
1	0.566433	1.152198	-0.258006	-0.131028	NaN	0.252221	-0.010111	0.214633	-1.186880	0.310525	...	-0.520989	-0.409856	0.806304	-0.021514	0.025973	-0.035710	-0.294230	-0.892110	-1.178260	-0.818750
2	-0.755992	-2.923016	-0.930535	-0.118503	NaN	0.601063	0.702288	-1.409369	1.559514	-0.584908	...	-0.526174	-0.526137	-1.116848	-0.358985	-0.176365	-0.343768	0.603125	-0.126249	0.014016	-0.650359
3	-1.757750	-1.493465	2.162068	-0.115873	NaN	-1.429842	0.430898	-0.369792	-0.493679	0.321185	...	-0.489831	-0.140458	-0.613870	0.178469	0.228311	0.156439	-0.601433	-0.160541	0.259485	1.172005
4	0.592411	-0.680185	-0.464763	-0.115472	NaN	-0.018132	0.476130	0.512232	-1.503771	-0.329070	...	-0.533849	-0.557487	0.687956	-0.121505	-0.176365	-0.119759	0.360596	-0.572049	-0.231452	-0.575742
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
11486	-0.889586	1.367837	0.388801	-0.123051	NaN	-0.130974	-0.168422	-0.967684	0.404180	-1.128563	...	-0.521520	-0.524626	-0.436348	0.522189	0.546270	0.514164	-1.983844	1.199721	0.995891	3.624425
11487	0.186895	-0.650881	-0.221517	-0.124437	NaN	2.253755	0.532669	-0.713176	0.080687	0.076007	...	-0.503169	-0.435104	-0.258826	-0.265243	-0.349797	-0.248699	0.861822	0.742490	1.101092	0.418357
11488	-0.487183	-0.352499	-1.291118	-0.131443	NaN	2.198017	-0.010111	0.073240	0.199521	0.022707	...	-0.478836	-0.396844	0.717543	0.415948	0.315027	0.370810	0.797147	0.033782	0.014016	-0.268819
11489	0.108417	0.989199	0.358434	-0.050381	NaN	0.262183	0.193432	-0.209547	0.536218	0.864841	...	-0.235213	-0.332226	-0.495522	0.059729	0.025973	0.057797	-0.601433	-0.594910	-0.511988	-0.202719
11490	-2.809595	-0.029269	0.214302	-0.113585	NaN	-0.168379	0.046429	1.471013	-1.866876	-0.062572	...	-0.496969	0.214840	-0.968913	-0.184000	-0.291987	-0.179755	-0.318483	-0.011941	0.014016	-0.172424

11441 rows × 529 columns

Zscore 를 통해 정규분포를 가정하였을 때 각 feature 에 해당하는 sample 들의 데이터가 평균으로부터 얼마나 떨어져 있는지 알 수 있다. 여기서 Nan 을 확인할 수 있는데 Nan 은 평균과 완전히 같기 때문에 생기는 것이다. V006 과 같이 모든 sample 에 대해 동일한 변수들은 이와 같이 데이터로서 효력이 없기 때문에 나중에 변수 선택에서 처리해 줄 것이다. 이제 이상치로 분류하기 위한 기준을 정해야 한다. 보통 3 sigma 바깥, 즉 퍼센트로 따지면 99.73% 바깥에 있는 샘플들을 이상치로 정한다. 따라서 이러한 기준을 바탕으로 이상치의 수를 샌다. 이와 같은 방식으로 구한 이상치의 수는 다음과 같다.

```
# 3sigma (99.73%) 바깥 쪽의 것을 이상치로 한다
outlier_idx = np.abs( sp.stats.zscore(data) ) > 3
print('# of outliers:', outlier_idx.sum().sum() )
```

```
# of outliers: 48175
```

즉, 48175 개의 outlier 라고 할 수 있는 이상치가 존재함을 확인하였다. 전체 표본의 개수가 대략 600 만개임을 생각하면 0.27%보다는 더 많은 대략 0.67% 정도의 이상치가 존재한다고 생각할 수 있다. 즉, 이상치가 많이 존재하는 데이터 셋이라고 할 수 있다. 하지만 무작정 이상치를 제거하거나 대치시키면 데이터의 특성을 지워버려 나중에 모델을 통해 훈련시킬 때 어려워질 수 있다. 대신, 사용할 모델을 선택할 때 이상치가 많다는 점을 고려하여 좀더 굳건한 모델을 사용하는 것이 좋다.

이제 데이터의 변환을 수행할 차례이다. 앞서 각 feature 들의 분포를 관찰하여 확인할 수 있었던 사실은 feature 마다 분포가 천차만별 다르다는 점이다. 그래서 데이터를 변환하여 scaling 해 주어야 데이터 학습을 더 용이하게 할 수 있다. Scaling 에는 크게 min-max scaling 과 standard scaling 두 가지가 있다. 본 프로젝트에서는 min-max scaling 보다는 standard scaling 을 사용할 것이다. Min-max scaling 은 0~1 사이로 mapping 시키는 것인데 그렇게 하기에는 데이터의 범주가 매우 방대하기 때문이다.

```
data_org = data
from sklearn.preprocessing import StandardScaler
scaler = StandardScaler()
scaler.fit(data)
data_scaled = scaler.transform(data)
```

```
data = pd.DataFrame(data_scaled, columns=[data_org.columns])
```

Data scaling 은 다음과 같이 StandardScaler 라이브러리를 불러와서 할 수 있다. 불러온 라이브러리를 통해 scaled 된 data 를 얻을 수 있고, 이 data 이차원 배열로 주어지므로 위와 같이 DataFrame 형식으로 바꾸어야 한다. Scaled 된 데이터를 나타내면 다음과 같다.

data

	v001	v002	v003	v004	v005	v006	v007	v008	v009	v010	...	v577	v578	v583	v584	v585	v586	v587	v588	v589
0	-1.930479	-0.364374	-2.557205	-0.920873	-0.125614	0.0	0.048790	-0.077958	0.610534	-0.586106	...	-0.533300	-0.400723	0.747130	0.003484	-0.060743	-0.009736	0.166574	0.571029	0.5400
1	-1.202614	0.566433	1.152198	-0.258006	-0.131028	0.0	0.252221	-0.010111	0.214633	-1.186880	...	-0.520989	-0.409856	0.806304	-0.021514	0.025973	-0.035710	-0.294230	-0.892110	-1.1782
2	-0.399085	-0.755992	-2.923016	-0.930535	-0.118503	0.0	0.601063	0.702288	-1.409369	1.559514	...	-0.526174	-0.526137	-1.116848	-0.358985	-0.176365	-0.343768	0.603125	-0.126249	0.0140
3	2.586297	-1.757750	-1.493465	2.162068	-0.115873	0.0	-1.429842	0.430898	-0.369792	-0.493679	...	-0.489831	-0.140458	-0.613870	0.178469	0.228311	0.156439	-0.601433	-0.160541	0.2594
4	0.695488	0.592411	-0.680185	-0.464763	-0.115472	0.0	-0.018132	0.476130	0.512232	-1.503771	...	-0.533849	-0.557487	0.687956	-0.121505	-0.176365	-0.119759	0.360596	-0.572049	-0.2314
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
11436	0.130023	-0.889586	1.367837	0.388801	-0.123051	0.0	-0.130974	-0.168422	-0.967684	0.404180	...	-0.521520	-0.524626	-0.436348	0.522189	0.546270	0.514164	-1.983844	1.199721	0.9958
11437	-0.879979	0.186895	-0.650881	-0.221517	-0.124437	0.0	2.253755	0.532669	-0.713176	0.080687	...	-0.503169	-0.435104	-0.258826	-0.265243	-0.349797	-0.248699	0.861822	0.742490	1.1010
11438	0.328266	-0.487183	-0.352499	-1.291118	-0.131443	0.0	2.198017	-0.010111	0.073240	0.199521	...	-0.478836	-0.396844	0.717543	0.415948	0.315027	0.370810	0.797147	0.033782	0.0140
11439	-0.292088	0.108417	0.989199	0.358434	-0.050381	0.0	0.262183	0.193432	-0.209547	0.536218	...	-0.235213	-0.332226	-0.495522	0.059729	0.025973	0.057797	-0.601433	-0.594910	-0.5119
11440	1.267140	-2.809595	-0.029269	0.214302	-0.113585	0.0	-0.168379	0.046429	1.471013	-1.866876	...	-0.496969	0.214840	-0.968913	-0.184000	-0.291987	-0.179755	-0.318483	-0.011941	0.0140

11441 rows × 530 columns

Scaling 이 잘 되었다는 사실을 확인할 수 있다. 마지막으로 통계 분석에 사용할 feature 를 선별해야 한다. 선별하는 방법에는 여러 가지가 존재한다. 첫 번째 방법은 PCA 를 사용하는 것이다. PCA 는 Principal Component Analysis 의 약자로 선형 대수에서 쓰이는 convariance matrix 를 구한 후 eiegenvalue decomposition 을 적용시켜 variance 에 영향을 끼치지 않는 feature 들을 제거하는 방법이다. 보통 PCA 분석은 데이터를 간단화 시켜 overfitting 문제를 해결하는데 많이 쓰인다. 여기선 PCA 분석을 한번 적용시켜 볼 것이다. 왜냐하면 v006 처럼 한눈에 보았을 때 variance 에 영향을 끼치지 않아 학습시킬 때 쓸모없는 feature 들이 보이기 때문이다. 하지만 PCA 로 인해 모델이 단순화되어서 성능이 떨어지는 경우가 존재하기 때문에 다음 단계에서 알맞은 모델을 선택할 때 사용하여 PCA 를 적용했을 때와 적용하지 않았을 때를 비교하여 PCA 적용할지 여부를 알아볼 것이다. 두 번째 방법은 feature selection 을 이용하는 방법이다. 이 방법은 PCA 와 비슷하지만 조금 더 직접적으로 여러 feature 들을 선택해 가며 모델을 돌려서 어떤 feature 들을 선택했을 때 모델이 가장 좋을지 알아보는 방법이다. 이 방법은 확실하게 어떤 feature 들을 선택하면 좋을지 알아볼 수 있다는 장점이 있지만 문제점은 똑 같은 모델을 여러 번 훈련해야 한다는 점이다. 특히, 본 프로젝트에는 몇 백개의 feature 이 존재하므로 대략적으로 같은 모델을 몇 백번씩 훈련시켜 보아야 한다는 점을 알 수 있다. 그러므로 본 프로젝트에서 사용하는 광범위한 데이터와는 맞지 않는다고 생각하므로 사용하지 않을 것이다. 마지막 방법은 p-value 를 계산해 보는 방법이다. P-value 는 유효성을 나타내며 어떤 귀무 가설을 세웠을 때 귀무 가설에서 예측한 값보다 더 극단적인 값이 나올 확률을 나타낸다. P-value 가 작으면 더 극단적인 값이 나올 확률이 적으므로 귀무가설을 기각할 수 있고 반대로 p-value 가 크면 더 극단적인 값이 나올 확률이 크므로 귀무가설을 채택해야 한다. 더 극단적인 값이 나올 확률이 크다는 것은 데이터로부터 얻을 수 있는 정보가 부족하기 때문에 가설이 옳다고 할 수 있는 근거가 부족함을 뜻하기 때문이다. 데이터 처리를 할 때 p-value 를 구하는 방법은 여러 가지가 있지만 어떤 변수를 사용하는가에 따라 달라진다. 본 프로젝트에서는 수치형 변수를 통해 이중 분류를 하는 것이 목적이므로 t-

test 를 이용할 것이다. T-test 는 흔히 스튜어트 t 검정이라고 불린다. T-test 는 귀무가설 전제하에 데이터가 t 분포를 따른다고 가정하고 가설 검정하여 p-value 를 구하는 방법이다. T 검저에서 p-value 를 구하기 위해 t 값이라는 통계량을 정의하는데 t 값은 두 집단의 평균 차이를 표준오차로 나눈 값으로 정의된다. 이 값은 두 변수 사이의 독립성을 측정하는 것이라고 생각할 수 있으므로 두 변수 사이의 관계를 t 분포에 넣어 p-value 를 구한다고 생각할 수 있다. T-test 를 적용하여 모든 feature 에 대해 p-value 를 다음 코드를 통해 구했다. (data_clean 은 데이터 변환을 시키기 전에 데이터를 가지고 있는 Dataframe 이다. 이 데이터를 사용한 이유는 scale 에 p-value 가 영향을 받기 때문이다.)

```
#p-value 구하기
#data_label와 data_feature를 이용한다.
Y = data_clean.iloc[:,1:]
out = pd.DataFrame({"Var": Y.columns})
slist = np.zeros(out.shape[0])
plist = np.zeros(out.shape[0])
for i in range(out.shape[0]):
    x = data_clean["Label"]
    y = Y.iloc[:,i]
    n = x.unique()
    ylist = []
    for j in range(len(n)):
        ylist.append(y[x==n[j]])
    r = sp.stats.ttest_ind(*ylist)
    s = r.statistic
    p = r.pvalue
    slist[i] = s
    plist[i] = p
out['stat'] = slist
out['pvalue'] = plist
```

Out 이라는 Data Frame 를 생산하여 p-value 와 stats, 즉 통계치를 삽입한다. 이를 통해 p-value 를 확인할 수 있다. 보다 보기 편하게 하기 위해 p-value 를 기준으로 내림차순으로 sorting 을 해주었다.

```
out.sort_values(by = 'pvalue', ascending = False)
```

	Var	stat	pvalue
201	v213	0.016159	0.987108
271	v290	-0.031577	0.974810
411	v441	0.037114	0.970395
403	v433	0.055193	0.955985
328	v351	-0.071015	0.943387
...	...	...	...
498	v535	NaN	NaN
499	v536	NaN	NaN
500	v537	NaN	NaN
501	v538	NaN	NaN
502	v539	NaN	NaN

데이터를 확인하니 NaN 으로 뜨는 데이터들이 있다. 이 데이터들은 보통 p-value 가 계산을 통해 나타낼 수 없을 때 나타난다. 이러한 경우는 보통 데이터에서 feature 에 해당하는 모든 sample 의 값이 거의 같을 때 발생한다. 데이터에서 sample 에 관계없이 모든

feature 가 같으면 이진 분류를 하는데 거의 도움이 안 된다. 대표적으로 위에서 설명했던 val006 의 경우 모든 sample 에서 동일한 값을 가져서 variance 가 0 이 되어 분류는 하는데 큰 의미가 없었다. 그러므로 이러한 값들은 제거해 주어야 한다.

```
data_pvalue = data.iloc[:,1:]
num = 0
for i in data_pvalue:
  if (out['pvalue'].isna())[num] == 1:
    data_pvalue = data_pvalue.drop([i], axis = 1)
    data = data.drop([i], axis = 1)
  num = num + 1
```

(Label 을 포함하지 않는 data_pvalue 라는 Dataframe 을 만들었다. Data_pvalue 에서 해당하는 feature 를 지우면서 동시에 data 에서 해당하는 feature 를 지웠다.)

위 코드를 이용하여 pvalue 가 NaN 이 뜨는 feature 들을 제거할 수 있었다. 다음으로 pvalue 가 아주 높은 경우를 살펴본다. P-value 가 높은 경우 귀무가설을 채택해야 하므로 분류를 하는데 의미가 퇴색된다. 그러므로 p-value 가 높은 값들을 제거해 줄 것이다. P-value = 0.05 일때를 보통 기준으로 하고 이 데이터토 feature 의 숫자가 매우 많으므로 평균적인 경우로 회귀한다고 생각하고 평균적인 기준을 적용시켜 p-value>0.05 인 경우 데이터에서 feature 를 제거할 것이다. 남은 feature 들이 데이터 전치를 통해 선택하는 feature 가 된다.

```
#p-value가 큰 경우 지워준다.
num = 0
out = out.reindex(range(0, out.shape[0]))
for i in data_pvalue:
    if out['pvalue'][num] >= 0.05:
      data_pvalue = data_pvalue.drop([i], axis = 1)
      data = data.drop([i], axis = 1)
    num = num + 1
```

지운 결과는 다음과 같다.

	Label	v001	v004	v005	v007	v008	v009	v010	v011	v012	...	v575	v577	v578	v583	v584	v585	v586	v587	v589	v590
0	normal	-1.930479	-0.920873	-0.125614	0.048790	-0.077958	0.610534	-0.586106	-0.425009	0.008740	...	-0.517639	-0.533300	-0.400723	0.747130	0.003484	-0.060743	-0.009736	0.166574	0.540021	1.077433
1	normal	-1.202614	-0.258006	-0.131028	0.252221	-0.010111	0.214633	-1.186880	0.310525	-0.608130	...	-0.528261	-0.520989	-0.409856	0.806304	-0.021514	0.025973	-0.035710	-0.294230	-1.178260	-0.818750
2	normal	-0.399085	-0.930535	-0.118503	0.601063	0.702288	-1.409369	1.559514	-0.584908	1.226458	...	-0.527477	-0.526174	-0.526137	-1.116848	-0.358985	-0.176365	-0.343768	0.603125	0.014016	-0.650359
3	normal	2.586297	2.162068	-0.115873	-1.429842	0.430898	-0.369792	-0.493679	0.321185	0.865950	...	-0.505194	-0.489831	-0.140458	-0.613870	0.178469	0.228311	0.156439	-0.601433	0.259485	1.172005
4	normal	0.695488	-0.464763	-0.115472	-0.018132	0.476130	0.512232	-1.503771	-0.329070	0.922029	...	-0.525837	-0.533849	-0.557487	0.687956	-0.121505	-0.176365	-0.119759	0.360596	-0.231452	-0.575742
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
11436	normal	0.130023	0.388801	-0.123051	-0.130974	-0.168422	-0.967684	0.404180	-1.128563	-0.960627	...	-0.498061	-0.521520	-0.524626	-0.436348	0.522189	0.546270	0.514164	-1.983844	0.995891	3.624425
11437	normal	-0.879979	-0.221517	-0.124437	2.253755	0.532669	-0.713176	0.080687	0.076007	1.146345	...	-0.490416	-0.503169	-0.435104	-0.258826	-0.265243	-0.349797	-0.248699	0.861822	1.101092	0.418357
11438	normal	0.328266	-1.291118	-0.131443	2.198017	-0.010111	0.073240	0.199521	0.022707	1.739182	...	-0.462017	-0.478836	-0.396844	0.717543	0.415948	0.315027	0.370810	0.797147	0.014016	-0.268819
11439	normal	-0.292088	0.358434	-0.050381	0.262183	0.193432	-0.209547	0.536218	0.864841	-0.584096	...	-0.234110	-0.235213	-0.332226	-0.495522	0.059729	0.025973	0.057797	-0.601433	-0.511988	-0.202719
11440	normal	1.267140	0.214302	-0.113585	-0.168379	0.046429	1.471013	-1.866876	-0.062572	-0.423870	...	-0.501921	-0.496969	0.214840	-0.968913	-0.184000	-0.291987	-0.179755	-0.318483	0.014016	-0.172424

앞서 언급한 V006 과 같은 feature 들이 잘 지워진 것을 확인할 수 있다. 데이터의 feature 개수는 340 개이고 sample 의 개수는 11441 개이다. 이를 통해 변수 선택을 마무리 지었고 최종적으로 EDA 와 데이터 전처리과정을 마무리 지었다고 할 수 있다.

3. 예측 모델 만들기

이제 본격적으로 전처리한 데이터를 바탕으로 예측 모델을 만들 것이다. 예측 모델은 training set 을 통해 만드므로 지금까지 전처리한 데이터를 xtrain 으로 놓고 레이블을 ytrain 으로 할 것이다.

다양한 예측 모델이 존재하는데 어떤 모델을 사용할지 고르기 위해서는 위에서 작성한 데이터를 살펴보아야 한다. 먼저, 위에서 살펴본 데이터를 보았을 때 특정 feature 몇 개의 variance 가 아주 큼을 알 수 있다. 이를 통해 트리 모델 같은 경우 sub-spacing 이 필수적으로 들어가야 한다고 생각할 수 있다. 또한, KNN 과 같은 불안정한 모델은 boosting 과 같이 예측력을 높여줄 수 있는 방법을 함께 적용시켜 주어야 한다. 다음으로 검증 방법을 무엇을 쓸 것인지도 데이터를 바탕으로 생각해 볼 수 있다. 일단 cross validation 을 사용해 볼 수 있다. Cross validation 은 전체 데이터를 n 개로 나눈 후 하나의 set 을 validation set 나머지를 training set 으로 지정하여 검증하는 과정이다. 보통 n 개이 set 을 각각 validation set 으로 하여 측정을 n 번 반복하여 모델을 훈련시키고 각 검증집합들의 성능을 평균하여 성능을 측정한다. Cross validation 의 parameter 인 n 이 너무 커지면 모델을 훈련해야 하는 횟수가 많아지므로 적절한 n 을 선택해야 한다. 본 프로젝트에서 사용하는 데이터는 방대하므로 n=3 정도로 작은 n 값을 선택할 것이다. 다음으로, 모델을 선정할 때 사용할 지표에 대해서도 생각해야 한다. 보통 모델을 평가할 때 accuracy, 즉 정확도를 많이 활용한다. 하지만 이번에는 다른 평가 방법도 고려해야 한다. 본 프로젝트에 사용할 training set 을 살펴보자. 이 training set 에 존재하는 두 개의 class 에 해당하는 샘플들의 수가 거의 20:1 정도로 굉장히 한 쪽으로 치우쳐 있는 것을 알 수 있다. 이렇게 되면 모델을 training 을 했을 때 과하게 한 쪽 class 쪽으로 training 되어서 deterministic 하게 한 쪽 class 쪽을 출력하는 모델이 생성될 수 있다. 그러면 test set 에서의 성능이 떨어지게 된다. 이를 방지하기 위해선 두 가지 방법이 있다. 첫 번째 방법은 좀 더 균형이 잡혀 있는 training set 을 사용하는 것이다. 하지만 현재 이렇게 하는 것은 불가능하다. 다른 방법은 training set 에서 성능 평가 방법을 f1 score 로 바꾸는 것이다. F1 score 는 precision 과 recall 의 조화 평균으로 나타나기 때문에 모델이 한 쪽 class 만 예측하는 것을 잘한다면 값이 낮아지게 되어있다. 또한, 본 프로젝트에서 중요한 것은 이윤을 최대화하는 것이다. 앞서 설명한 이윤표를 통해 알 수 있다시피 잘못 예측하더라도 정상칩이라고 예측했을 때 손해가 커진다. 그러므로 정확도로 예측하여 모델이 deterministic 하게 되는 이슈를 피하기 위해 f1 score 를 평가 지표로 사용하는 것이 더 바람직하다. 추가적으로 이윤 극대화를 하기 위해서는 threshold 를 알맞게 조종하는 것이 필요하다. 이중 분류 모델에서 threshold 는 이중 분류 모델의 분류 기준이 된다. 보통 0.5 로 설정하지만 이 데이터는 class 가 한 쪽으로 치우쳐 있으므로 다른 분류 기준이 필요하다.

이 기준은 이윤표에서 도출해 낼 수 있다. 이득표를 보면 이익을 내기 위해서는 무조건 정상칩이라고 예측하는 것이 유리하다. 정상칩의 개수 및 불량칩의 개수와 이득표를 통해 계산할 수 있는 기대 수익은 무조건 정상칩으로 예측할 때 커진다. 극단적으로 모든 칩을 정상칩이라고 예측하면 최소한 이익은 얻게 된다. 하지만 모든 칩을 불량칩이라고 예측하면 손해만 생기게 된다. 이러한 점을 바탕으로 threshold 를 올려 정상칩, 즉 음성을 최대한 많이 예측하는 것이 이득이라는 결론을 내릴 수 있다. Threshold 를 몇으로 하는 것인지 최적인지는 각 모델마다 다르므로 여러 값들을 넣어 찾아보아야 한다. Recall 이 커지면 precision 은 작아진다. Recall 이 커지고 precision 이 작아지게 하기 위해서는 마지막으로 모델이 너무 복잡하여 overfitting 이 되는 이슈를 사전에 방지하기 위해 PCA 를 사용할 것이다. 정확하게는 PCA 를 적용시켜 얻은 모델과 아닌 모델을 비교하여 더 좋은 모델을 선택할 것이다.

첫 번째로 사용할 모델은 로지스틱 회귀이다. 로지스틱 회귀는 대표적인 선형 분류 모델이다. 이 모델은 강건한 모델이고 이상치에 잘 흔들리지 않는다. 이 모델은 보통 규제화 기법을 적용시켜 같이 쓰이기도 한다. 로지스틱 회귀 모델에는 여러 파라미터들이 존재한다. 먼저, penalty 파라미터는 사용할 규제화를 나타낸다. L1, L2, 그리고 둘을 섞은 elastic net 세 가지가 존재한다. 보통 logistic regression 에서는 L2 규제화를 많이 쓰지만 feature 가 0 이되는 성질도 가지고 있는 L1 규제화를 함께 써서 overfitting 문제를 방지하기 위해 elastic net 을 사용하는 것이 좋다. Elastic net 에는 2 가지 튜닝 parameter 가 존재한다. 첫 번째는 C parameter 이다. C parameter 는 L1 규제화나 L2 규제화에서 사용하는 alpha 파라메터의 역이다. 즉, regularization strength 의 역을 나타낸다고 할 수 있다. 다음으로 l1_ratio 라는 parameter 가 존재한다. 이 parameter 는 elastic net 에서 L1 규제화와 L2 규제화 사이의 비율을 어느 정도로 조절할지 나타낸다. 하지만 본 프로젝트에서는 Elastic net 을 사용하기 힘들다. 왜냐하면 데이터에 0 이 존재하기 때문이다. 데이터에 0 이 존재하면 L1 규제화에서 loss function 을 미분하는 것이 불가능하다. 따라서 L1 규제화를 적용시킬 수 없다. 대신 L2 규제화를 그대로 적용시킬 것이다. 다음으로 solver 의 종류를 정해야 한다. Solver 는 머신 러닝에서 최적화 문제를 풀기 위해 사용하는 기법의 종류이다. 여러 가지가 있지만 대용량 데이터를 다루기 위해 solver = 'saga'를 사용했다. Class_weight = 'balanced'로 지정하여 한 쪽으로 class 가 치우쳐 있는 문제를 최대한 해결하였다. 이와 같이 로지스틱 모델을 구성하였고 GridSearchCV 로 튜닝하여 최적의 모델을 찾았다.

먼저, 전처리한 데이터는 다음 코드를 통해 xtrain 과 ytrain 으로 변경하였다.

```
#전처리한 데이터를 train_set으로 변형
#train_test_split 함수가 자동적으로 해주는 작업을 직접 해주어야 함
#데이터 type을 혹시 모르니 모두 float로 변환
xtrain = data.iloc[:,1:]
xtrain = xtrain.astype(float)
#Label도 마찬가지로 training을 원할하게 하기 위해 1차원 array 형식으로 펴준다
ytrain = pd.DataFrame({'Label': data['Label']})
ytrain = ytrain.values.ravel()
```

그 다음 ytrain 은 Label_encoder 를 통해서 0 과 1 로 변환해 주었다. 그래야 f1 score 를 구할 수 있기 때문이다.

```
#추가로 f1 score를 구할 때 편리하기 위해 label의 class를 0,1로 바꾼다
from sklearn.preprocessing import LabelEncoder
label_encoder = LabelEncoder()
ytrain = label_encoder.fit_transform(ytrain)
```

이제 Logistic regression 을 훈련할 차례이다. Logistic regression 은 solver 로 lbfgs 를 사용할 것이다. Saga 가 더 좋을 수 있지만 saga 는 random shuffling 을 포함하고 있어서 feature 가 많은 데이터에서는 잘 동작 안 할 수 있다. L2 penalty 를 C 를 통해 튜닝해 줄 것이다. Class weight 는 balanced 로 설정하여 앞서 언급한 class 가 한 쪽으로 치우쳐 져서 발생하는 문제를 해결할 것이고 max_iter = 5000 으로 설정하여 max_iter 에 도달하면 프로그램이 converging 을 끝내게 할 것이다.

```
#Logistic regression 튜닝, cross validation는 3으로 하여 훈련
#C는 Ridge 규제화의 alpha의 역수 값
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import GridSearchCV
params = {'C': (0.001, 0.01, 0.1, 1.0, 10.0, 100.0)}
f = GridSearchCV( LogisticRegression(max_iter = 5000, penalty = 'l2', solver = 'lbfgs', class_weight= 'balanced' ), params, cv = 3 )
f.fit(xtrain, ytrain)
f.best_estimator_
```

```
LogisticRegression(C=100.0, class_weight='balanced', max_iter=5000)
```

즉, C = 100.0 일 때 최적의 모델을 튜닝할 수 있다는 사실을 알 수 있다. 이 모델을 바탕으로 threshold probabilities 를 바꿔가며 최적의 모델을 찾아보자.

```
# Threshold 설정 (모든 모델에 대해 동일 ) 및 f1_score 성능 측정
# 각 class의 갯수에 가중치를 줘서 평균을 구했다. (average = 'weighted')
from sklearn.metrics import f1_score
probabilities = f.predict_proba(xtrain)[:,1]
threshold = [0.1,0.2,0.3,0.4,0.5]
f1_list = []
for i in range(len(threshold)):
  predictions = (probabilities > threshold[i]).astype(int)
  f1 = f1_score(ytrain, predictions, average = 'weighted')
  f1_list.append(f1)
print(f1_list)
print(max(f1_list))
```

```
[0.8914156859168355, 0.8837878673792602, 0.8601795455698088, 0.8089600320950963, 0.7052626401380339]
0.8914156859168355
```

즉, threshold 가 0.1 일 때 f1 score 이 최대가 되는 것을 확인할 수 있다. 이 모델이 logistic regression 에서 가장 최적화된 모델이다.

다음으로 PCA 를 적용시킨다. PCA 는 Principal Components Analysis 의 약자이고 covariance matrix 을 활용하여 변수가 어느 축들로 분포되어 있는지 알려준다. PCA 는 데이터를 간단화

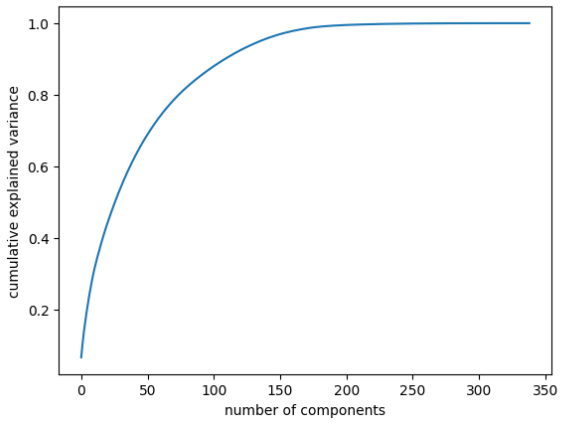
시켜서 overfitting 문제를 해결하기 위해 사용되며 기본적으로 데이터의 covariance matrix 을 eigenvalue decomposition 하여 principal components 들을 얻은 다음 90%이상의 variance 를 유지할 수 있을 때 필요한 component 만을 선택하여 데이터를 재구성한다. 위 과정을 sklearn library 로 구현하면 다음과 같다.

```
#PCA해보기
from sklearn.decomposition import PCA
pca = PCA()
pca.fit(xtrain)
```

```
▾ PCA
PCA()
```

그 다음 PCA 한 결과를 바탕으로 component 의 개수당 얼마나 variance 가 설명되었는지 그래프로 나타내었다.

```
#전체 데이터에서 설명이 되는 variance를 그래프로 나타내기
plt.plot(np.cumsum(pca.explained_variance_ratio_))
plt.xlabel('number of components')
plt.ylabel('cumulative explained variance');
```



몇몇 변수들이 많은 variance 를 설명하고 있는 것을 확인할 수 있다.

```
#n_component 변수를 지정하여 위 그래프에서 0.90이상이 될 때 components의 수를 측정
n_component = 0
for i in range(xtrain.shape[1]):
  if np.cumsum(pca.explained_variance_ratio_)[i] > 0.9:
    print('number of principal components :', i)
    n_component = i
    break
```

```
number of principal components : 109
```

이제 다음으로 90% variance 를 설명할 수 있는 변수들을 고른다. 고르면 다음과 같이 109 개를 고를 수 있다. 이제 PCA 를 적용시켜 모델을 얻는다. 얻은 모델은 다음과 같다.

```
#Logistic regression 튜닝, cross validation는 3으로 하여 훈련
#C는 Ridge 규제화의 alpha의 역수 값
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import GridSearchCV
params = {'C': (0.001, 0.01, 0.1, 1.0, 10.0, 100.0, 1000.0)}
f = GridSearchCV( LogisticRegression(max_iter = 10000, penalty = 'l2', solver = 'lbfgs', class_weight= 'balanced' ), params,  error_score='raise', cv = 3 )
f.fit(ztrain, ytrain)
f.best_estimator_
```

```
▾                    LogisticRegression
LogisticRegression(C=10.0, class_weight='balanced', max_iter=10000)
```

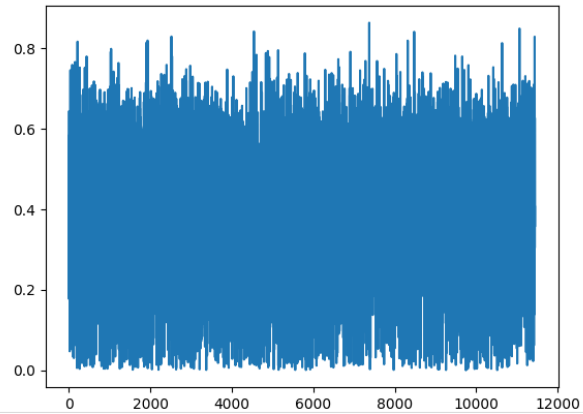
모델의 f1 score 를 구하면 다음과 같다.

```
#f1값 구하기
from sklearn.metrics import f1_score
probabilities = f.predict_proba(ztrain)[:,1]
threshold = [0.1,0.2,0.3,0.4,0.7,0.9,0.999]
f1_list = []
for i in range(len(threshold)):
  predictions = (probabilities > threshold[i]).astype(int)
  f1 = f1_score(ytrain, predictions, average = 'weighted')
  f1_list.append(f1)
print(f1_list)
print(max(f1_list))
```

```
[0.8757840251045367, 0.8490153331909692, 0.8042623237102642, 0.6963554162796306, 0.032607575593069774, 0.007715084053418995 4, 0.007715084053418995 4]
0.8757840251045367
```

```
plt.plot(probabilities)
```

```
[<matplotlib.lines.Line2D at 0x7e3c54871f30>]
```



다음으로 KNN 모델을 훈련시킬 것이다. KNN 모델은 거리를 기반으로 한다. 근처에 있는 n 개의 neighbors 들의 평균을 구해 decision boundary 를 그리는 모델이다. 이 모델에서는 n_neighbors 라는 파라미터를 튜닝해 줄 수 있다. N-neighbors 는 평균을 구할 때 선택할 neighbors 의 개수다. N_neighbors 가 1 이면 overfitting 이 걸리게 되고 샘플의 수와 같으면 underfitting 이 생긴다. 따라서 적절하게 정해줘야 한다. 그리고 가중치도 parameter 로 선택할 수 있다. 가중치에는 총 두 가지가 존재하다. 첫 번째는 가중치를 항상 1 로 하는 것이다. 이 방식을 보통 쓰지만 이미 데이터 셋이 scaling 된 상태이기 때문에 sample 간의 거리가 매우 작고 그러므로 본 프로젝트에서 사용하면 오류가 생긴다. 그러므로 다른 방식을 사용한다. 다른 방식은 바로 distance 이며 거리가 멀수록 가중치를 작게 거리가 가까우면 가중치를 크게 하는 방식이다. 이 방법을 이용하면 scaling 으로 sample 들 간의 거리가 줄어든 문제를 어느 정도 해결할 수 있다. 또한 거리 측정 방식도 선택할 수 있다. 거리 선택 방식에는 manhattan distance 와 Euclidean distance 2 개가 존재한다. Manhattan distance 는 거리의 x 축과 y 측의 합을 구하는 것이고 Euclidean distance 는 우리가 흔히 사용하는 피타고라스 정리를 이용한 거리 측정 방식이다. 이를 일반화하면 minkowski distance 가 되지만 minkowski distance 나 manhattan distance 보다는 Euclidean distance 를 이용할 것이다. 고차원 데이터에서는 아무래도 Euclidean distance 가 더 정확하기 때문이다. 즉, default 로 설정한다. KNN 알고리즘에는 ball_tree, kd_tree 와 같은 것들도 존재하는데

default = 'auto'로 설정하면 알아서 최적의 알고리즘을 찾아주기 때문에 default 로 놓으면 된다. 이를 통해 KNN 모델을 학습시킨 결과는 다음과 같다.

```
#KNN 모델을 만들 것이다. n_neighbors를 튜닝해 줄 것이고 weights는 distance로 할 것이다.
from sklearn.neighbors import KNeighborsClassifier
params = {'n_neighbors': np.arange(20,100)}
from sklearn.model_selection import GridSearchCV
f = GridSearchCV(KNeighborsClassifier(weights = 'distance'),params, cv = 3)
f.fit(xtrain,ytrain)
f.best_params_
```

```
{'n_neighbors': 20}
```

```
f.best_estimator_
```

```
KNeighborsClassifier
KNeighborsClassifier(n_neighbors=20, weights='distance')
```

파라미터를 튜닝한 결과 n_neighbors 가 20 일 때가 최적의 모델이라는 결론을 얻을 수 있었다. 이렇게 얻은 parameter 를 바탕으로 f1 score 을 구하면 다음과 같다.

```
#f1값 구하기
from sklearn.metrics import f1_score
probabilities = f.predict_proba(xtrain)[:,1]
threshold = [0.1,0.2,0.3,0.4,0.5]
f1_list = []
for i in range(len(threshold)):
  predictions = (probabilities > threshold[i]).astype(int)
  f1 = f1_score(ytrain, predictions, average = 'weighted')
  f1_list.append(f1)
print(f1_list)
print(max(f1_list))
```

```
[0.9941620512272686, 0.9941620512272686, 0.9941620512272686, 0.9941620512272686, 0.9941620512272686]
0.9941620512272686
```

즉, 0.9941620512272686 다고 장담은 하지 못한다. 왜냐하면 overfitting 문제가 있을 수 있기 때문이다. 또한 threshold 가 변하는데 계속 같은 f1 score 가 나온다는 사실은 defect 를 예측할 확률이 아주 낮게 설정되었음을 시사하고 이는 앞서 설명한 class imbalance 으로 인해 한 쪽 class 를 무조건 답하는 식으로 모델이 훈련되는 문제점과 일맥상통한다. 따라서 현재 결과만 보았을 때 KNN 을 통해 얻은 모델을 선택하는 것은 좋지 않을 거라 생각할 수 있다.

다음으로 PCA 를 적용시킨 데이터를 바탕으로 KNN 모델을 튜닝하여 최선의 모델을 구해본다. (PCA 를 하는 부분은 앞부분과 동일하기 때문에 생략한다.)

```
#f1값 구하기
from sklearn.metrics import f1_score
probabilities = f.predict_proba(ztrain)[:,1]
threshold = [0.1,0.2,0.3,0.4,0.999]
f1_list = []
for i in range(len(threshold)):
  predictions = (probabilities > threshold[i]).astype(int)
  f1 = f1_score(ytrain, predictions, average = 'weighted')
  f1_list.append(f1)
print(f1_list)
print(max(f1_list))
```

```
[0.9935166976888887, 0.9935166976888887, 0.9935166976888887, 0.9935166976888887, 0.9935166976888887]
0.9935166976888887
```

얻을 수 있는 결과값은 PCA 를 안 했을 경우와 대동소이 하다. 즉, 모델 복잡도의 문제보다는 class imbalance 문제의 영향이 있음을 추측할 수 있다.

다음으로 다루어 볼 모델은 Bayesian 모델이다. Bayesian 모델은 확률론적으로 기계학습에 접근하는 모델들이다. Bayesian 모델에는 분류 모델 회귀 모델이 모두 존재하고, 분류 모델에는 크게 3 가지가 존재한다. 첫 번째 모델은 Bernoulli NB 이다. Bernoulli NB 는 strong independence assumption 을 바탕으로 분류하는 Naïve Bayesian Classifier 의 한 종류이다. 이 모델은 multivariate Bernoulli model 에 대하여 사용할 수 있는 모델이다. Multivariate Bernoulli model 은 이중 분류를 하기 위해 디자인되었으므로 본 데이터를 학습시키는데 적합한 모델이다. 기본적으로 Naïve Bayesian Classifier 의 문제점인 예측 값이 0 으로 떨어지는 문제점을 가지고 있지만 한번 튜닝해보도록 한다.

모든 NB 모델들이 그렇듯 Bernoulli NB 에 존재하는 튜닝 파라미터는 alpha 이다. 나머지는 모델 파라메터들이고 주로 prior 를 조정해주는 것이기 때문에 prior 에 대한 정보가 없는 이 데이터에서는 스스로 학습해야 하므로 의미 없다. 또한 categories 의 개수도 2 개로 정해져 있기 때문에 binarize 와 같이 binary 로 바꿔주는 기능도 필요 없다. 다시 alpha 파라메터로 돌아가서 alpha 파라메터는 앞서 언급한 0 이 되는 문제를 해결하기 위해 사용한다. Naïve Bayesian classifier 구한 각 conditional probability 들의 분모와 분자에 alpha 를 더해주는 방법이다. Laplacian smoothing 이라고도 불리며 현실에서 거의 안 쓰이는 단어가 모델에서 0 으로 예측하는 것을 방지하기 위한 방법이라 생각할 수 있다. 이 데이터에서도 비슷한 일이 일어날 수 있으므로 alpha 파라메터를 튜닝해 줄 것이다.

```
#GridSearchCV로 a값 지정하고 LDA, QDA는 교차검증
from sklearn.naive_bayes import BernoulliNB
from sklearn.model_selection import GridSearchCV
params = {'alpha': 10 ** np.linspace(-3,3,21)}
f = GridSearchCV(BernoulliNB(), params, cv = 3)
f.fit(xtrain,ytrain)
```

GridSearchCV
estimator: BernoulliNB
BernoulliNB

```
f.best_params_
```

```
{'alpha': 1000.0}
```

Alpha 파라메터를 튜닝할 범위는 보통 사용되는 범위를 선택하였다. GridSearchCV 로 튜닝한 결과는 위와 같이 alpha = 1000 일때가 제일 좋다는 결론을 얻을 수 있다.

```
#f1값 구하기
from sklearn.metrics import f1_score
probabilities = f.predict_proba(xtrain)[:,1]
threshold = [0.1,0.2,0.3,0.4,0.999]
f1_list = []
for i in range(len(threshold)):
  predictions = (probabilities > threshold[i]).astype(int)
  f1 = f1_score(ytrain, predictions, average = 'weighted')
  f1_list.append(f1)
print(f1_list)
print(max(f1_list))
```

```
[0.8960321322888648, 0.8957437831598406, 0.8952906255004074, 0.8949733529251255, 0.8652995937583321]
0.8960321322888648
```

튜닝한 결과를 바탕으로 f1 score 를 출력하면 위와 같다. 결과를 통해 알 수 있다시피 threshold 가 바뀌더라도 심지어 0.999 까지 올라가도 f1 score 가 거의 constant 하게 유지되는 것을 알 수 있다. 이를 통해 Bernoulli NB 모델은 class imbalance 문제를 매우 심하게 겪고 있다는 결론을 내릴 수 있다.

PCA 를 적용했을 때의 결과도 한번 살펴본다.

```
#f1값 구하기
from sklearn.metrics import f1_score
probabilities = f.predict_proba(ztrain)[:,1]
threshold = [0.1,0.2,0.3,0.4,0.7,0.9,0.999]
f1_list = []
for i in range(len(threshold)):
  predictions = (probabilities > threshold[i]).astype(int)
  f1 = f1_score(ytrain, predictions, average = 'weighted')
  f1_list.append(f1)
print(f1_list)
print(max(f1_list))
```

```
[0.8985553202972857, 0.8985553202972857, 0.8985553202972857, 0.8985553202972857, 0.8985116707121418, 0.8599173112681979, 0.007715084063418995]
0.8985553202972857
```

PCA 를 적용시켰을 때도 비슷하게 높은 threshold 를 적용시켜야 f1 score 이 감소함을 알 수 있다. 그러나 overfitting 문제가 해결되었는지 감소하기 위한 threshold 가 0.9 정도 많이 감소한 것을 알 수 있다. 하지만 여전히 class imbalance 문제를 크게 겪고 있다는 사실을 확인할 수 있다.

다음으로 LDA 모델을 다룰 것이다. LDA 는 Linear Discriminant Analysis 의 약자로 variance 를 maximize 하고 평균 사이의 거리를 최대화시키는 mapping 을 찾는 방법이다. 이 모델은 흔히 discriminant analysis 라고 불리며 판단 경계를 선형으로 잡고 선 위에 있는 점들을 class A, 밑에 점들을 class B, 이런 식으로 분류한다. LDA 는 다음과 같이 implement 할 수 있다.

LDA 는 따로 튜닝해줄 파라미터가 없다. N_components 라는 파라메터가 존재하긴 하는데 PCA 에서 하는 것과 유사하기 때문에 해줄 필요가 없다. 나머지 parameter 들은 특정 solver 를 사용한다는 조건이 있을 때 사용 가능하기 때문에 default 로 한다. Solver 는 기본 solver 인 svd 를 사용할 것이다. 그러므로 store_covariance 를 True 로 하여 각 class 에 해당하는 covariance matrix 를 계산하여 저장하도록 해야 모델이 잘 훈련된다. LDA 를 훈련한 결과는 다음과 같다.

```
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis
f = LinearDiscriminantAnalysis(store_covariance=True, solver = 'svd')
f.fit(xtrain,ytrain)
```

```
LinearDiscriminantAnalysis
LinearDiscriminantAnalysis(store_covariance=True)
```

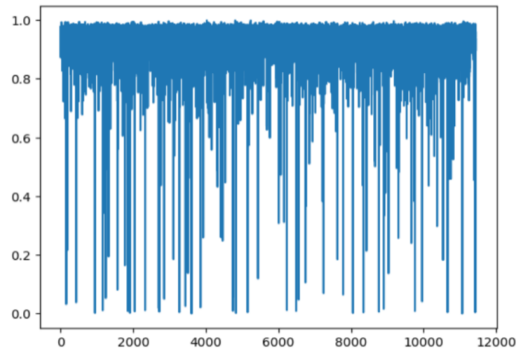
```
#f1값 구하기
from sklearn.metrics import f1_score
probabilities = f.predict_proba(xtrain)[:,1]
threshold = [0.1,0.2,0.3,0.4,0.999]
f1_list = []
for i in range(len(threshold)):
  predictions = (probabilities > threshold[i]).astype(int)
  f1 = f1_score(ytrain, predictions, average = 'weighted')
  f1_list.append(f1)
print(f1_list)
print(max(f1_list))
```

```
[0.8981568864584427, 0.898544073395126, 0.8985547492199937, 0.8983225174357622, 0.008241267834076676]
0.8985547492199937
```

F1 score 를 확인하면 Naïve Bayesian 모델보다는 확실히 성능이 향상되었음을 확인할 수 있다. Threshold 가 바뀌더라도 같은 값에 머물러 있지 않으므로 class imbalance 문제에서 꽤 자유롭기 때문이다. Probabilities 를 plot 하여도 이를 확인할 수 있다.

```
plt.plot(probabilities)
```

```
[<matplotlib.lines.Line2D at 0x7889cd02fd60>]
```



Probabilities 가 낮은 경우가 상당히 존재한다는 점에서 defect 로 분류하는 케이스가 생각보다 많다는 사실을 알 수 있다. 그러므로 class imbalance 문제에서 좀더 자유롭다고 할 수 있다.

이제 PCA 를 적용시켜 볼 것이다.

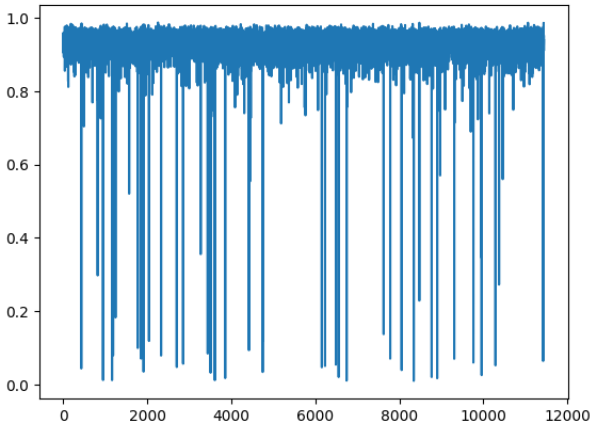
```
#f1값 구하기
from sklearn.metrics import f1_score
probabilities = f.predict_proba(ztrain)[:,1]
threshold = [0.1,0.2,0.3,0.4,0.7,0.9,0.999]
f1_list = []
for i in range(len(threshold)):
  predictions = (probabilities > threshold[i]).astype(int)
  f1 = f1_score(ytrain, predictions, average = 'weighted')
  f1_list.append(f1)
print(f1_list)
print(max(f1_list))
```

```
[0.8976653166351386, 0.897696684411012, 0.897564055501257, 0.8977258470247577, 0.8974592238903115, 0.8659620725150744, 0.007715084053418995]
0.8977258470247577
```

PCA 를 적용시켜 LDA 를 해보면 threshold probabilities 가 바뀜에 따라 f1 score 가 줄어들었음을 확인할 수 있다. 이를 통해 PCA 를 하여 데이터가 간단해지면서 데이터의 예측율이 줄어들었다고 생각할 수 있다.

```
plt.plot(probabilities)
```

```
[<matplotlib.lines.Line2D at 0x7889bb623730>]
```



실제로 probabilities 들을 plot 해 보면 defect 라고 표현하는 경우가 줄어들었음을 통해 이를 확인할 수 있다.

이제 LDA 이어 QDA 를 분석해 보겠다. QDA 는 GaussianNB 와 동일한 모델이다. 즉, 연속형 변수를 다룰 때 사용하는 Naïve Bayesian Classifier 라고 생각할 수 있다. QDA 는 나머지는 모두 같지만 판단 경계가 선형이 아니라 이차함수인 discriminant model 이다. QDA 의 파라미터들은 모두 LDA 와 동일하게 생각하면 된다. QDA 를 통해 예측을 해 보겠다.

```
from sklearn.discriminant_analysis import QuadraticDiscriminantAnalysis
f = QuadraticDiscriminantAnalysis(store_covariance=True)
f.fit(xtrain,ytrain)
```

```
/usr/local/lib/python3.10/dist-packages/sklearn/discriminant_analysis.py:926: UserWarning: Variables are collinear
  warnings.warn("Variables are collinear")
```

```
QuadraticDiscriminantAnalysis(store_covariance=True)
```

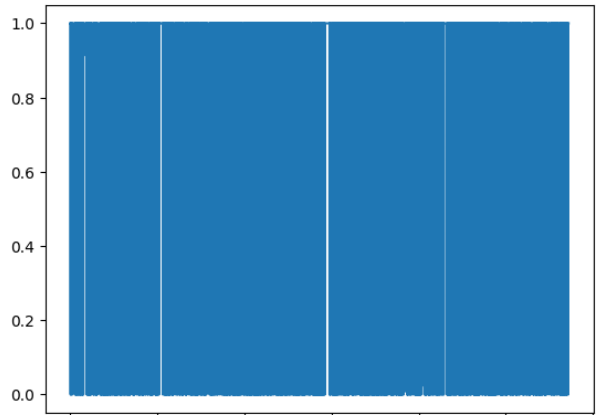
(동일한 선형 분포를 가지는 feature 가 있음을 나타냄)

```
#f1값 구하기
from sklearn.metrics import f1_score
probabilities = f.predict_proba(xtrain)[:,1]
threshold = [0.1,0.2,0.3,0.4,0.5,0.6,0.999]
f1_list = []
for i in range(len(threshold)):
  predictions = (probabilities > threshold[i]).astype(int)
  f1 = f1_score(ytrain, predictions, average = 'weighted')
  f1_list.append(f1)
print(f1_list)
print(max(f1_list))
```

```
[0.9813459481832508, 0.9810233072847964, 0.9807013250522432, 0.9804602673480473, 0.9800593148468556, 0.9798992155141326, 0.9742856012270066]
0.9813459481832508
```

```
plt.plot(probabilities)
```

```
[<matplotlib.lines.Line2D at 0x7889bb66d000>]
```



이 결과를 통해 알 수 있다시피 QDA 는 너무 과하게 overfitting 되어 있다. Threshold 를 거의 1 에 가깝게 바꾸어도 1 에 육박할 정도이다. 그리고 class imbalance 문제도 거의 해결하지 못해 항상 normal 를 예측한다. 즉, QDA 는 전혀 사용하면 안 되는 과하게 복잡한 모델이라고 할 수 있다.

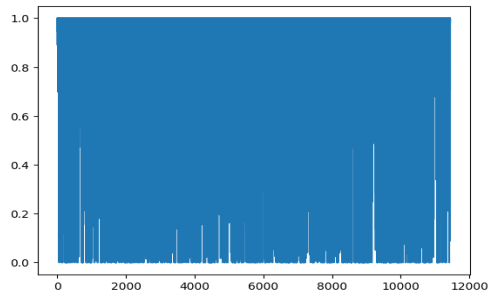
한번 PCA 를 사용해보자.

```
#f1값 구하기
from sklearn.metrics import f1_score
probabilities = f.predict_proba(ztrain)[:,1]
threshold = [0.1,0.2,0.3,0.4,0.7,0.9,0.999]
f1_list = []
for i in range(len(threshold)):
  predictions = (probabilities > threshold[i]).astype(int)
  f1 = f1_score(ytrain, predictions, average = 'weighted')
  f1_list.append(f1)
print(f1_list)
print(max(f1_list))
```

```
[0.9248192957839076, 0.9280479883338335, 0.9292744267853423, 0.9289188181437709, 0.9258985340740624, 0.9072433807158263, 0.6828345192774891]
0.9292744267853423
```

```
plt.plot(probabilities)
```

```
[<matplotlib.lines.Line2D at 0x7889ccd29f60>]
```



Threshold probabilities 와 probabilities 를 그린 곡선에서 알 수 있다시피 PCA 를 적용하기 전보다는 나아졌다. Overfitting 문제도 약간이나마 해결되었고 threshold 가 1 에 몰려 있는 현상도 어느 정도 해결되었다. 하지만 여전히 overfitting 이 되어 있음을 다른 모델들과 비교하여 알 수 있다. 만약 PCA 를 더 강하게 한다면 더욱 overfitting 문제가 해결될 수 있지만 이 경우 데이터에 존재하는 중요한 특성을 잃어버릴 수 있기 때문에 위험하기 때문에 PCA 는 이 모델에 대해서도 다른 모델들과 동일하게 적용할 것이다.

이와 같이 LDA 를 제외한 나머지 Bayesian 모델들은 효과적이지 못했음을 확인하였다. 이제 다음으로 SVM 모델을 다루어 본다. SVM 모델은 supporting vector machine 이라고 불리며 선형 경계와 feature 들 사이의 최소 거리를 최대화하는 것을 목표로 한다. 물론 SVM 모델에서 이러한 제약을 낮추는 경우도 있고 이를 soft-margin SVM 이라고 하고 조건을 유지하면 hard-margin SVM 이라고 한다. SVM 은 수학적으로 hinge loss 에 L2 regularization 을 적용한 것과 동일한 형태라는 것을 증명할 수 있고 kernal 함수를 적용시켜 선형 경계가 아니라 더 높은 차원의 경계로 mapping 시키는 것도 가능하다. SVM 는 이처럼 다양한 parameter 를 가지고 있는 복잡한 모델이다.

SVM 에 존재하는 parameter 들은 다음과 같다. 먼저, SVM 의 classifier 모델은 SVC 라고 불리며 여러 가지 다양한 parameter 들이 존재한다. 먼저, logistic regression 과 마찬가지로 C 라는 parameter 가 존재한다. 이 파라미터는 regularization parameter 이고 squared l2 parameter 로 주어진다. 나머지 기능은 logistic regression 에서 설명한 것과 동일하며 hard SVM 을 soft-margin SVM 으로 바꾸는 parameter 이므로 꼭 튜닝해 주어야 한다. 다음으로 kernal 함수들이 존재한다. Kernal 에는 대표적으로 linear, poly, rbf, sigmoid 가 존재한다. 앞서 사용한 모델들을 통해 알 수 있는 점은 복잡한 모델을 사용할수록 overfitting 문제가

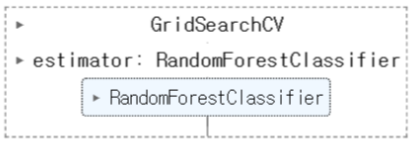
심각해저서 문제가 생긴다는 점이다. 그러므로 linear kernal 나 polynomial kernal 를 사용한다. 보통 polynomial kernal 많이 쓰므로 polynomial kernal 에서 polynomial parameter 를 2 로 지정할 것이다. 그 외에 shrinking parameter 가 존재한다. 이 parameter 는 현재의 decision boundary 를 만드는데 필요 없는 support vector 를 지울지 말지를 결정하는 parameter 이다. 이러한 방식을 쓰면 훈련 속도는 빨라지겠지만 오차가 커질 수 있다. 그러므로 본 프로젝트에서는 쓰지 않는다. Probability 는 fitting 을 위해서 항상 True 로 설정해야 한다. Logistic regression 과 마찬가지로 class_weight 라는 parameter 가 존재한다. 이는 logistic regression 에서와 동일한 역할을 하고 class imbalance 가 있는 현재 데이터에 필수적으로 적용시켜야 한다. SVM 이 다른 모델들 보다 이 데이터를 예측하는데 유리할 수 밖에 없는 이유이기도 하다. Logistic regression 과 마찬가지로 class_weight = 'balanced'로 설정해준다. 나머지 parameter 들은 default 도 설정해 줘도 무방하다. 예를 들어, decision function shape 와 같은 함수들은 다중 분류에서 사용된다. 이를 바탕으로 SVM 모델을 튜닝하려고 했다. 그러나 SVM 은 아주 큰 문제점이 있다. SVM 은 O(n^3)이기 때문에 아주 느리다는 점이다. 본 프로젝트에서 사용하는 방대한 데이터를 training 하는데 몇 시간이 넘게 걸린다. 모델 복잡도도 아주 크고 SVM 에서 Probability = True 로 설정하는 순간 확률을 구하기 위해 대량의 계산을 하게 되어 더 느려지게 되어 있다. 즉, sample 이 많은 경우에 유용하지 않은 모델이다. 따라서 SVM 을 다루어 보지는 않을 것이다.

다음으로 랜덤 포레스트 분류기를 이용한 모델을 만들어 보겠다. 랜덤 포레스트 분류기는 흔히 트리 모델에 sub-spacing 과 bagging 을 함께 사용한 모델로 ensemble 모델로 분류된다. 트리 모델은 다루지 않고 바로 랜덤 포레스트 분류기는 사용하는 이유는 트리 모델의 부족한 성능을 랜덤 포레스트 분류기가 채워주기 때문이다. 트리 모델은 우선 불안정하다. 트리 모델에서 분류를 할 때 데이터에 따르기 때문에 input 데이터에 따라 쉽게 바뀐다. 그래서 예측력이 떨어지는데 이를 재표집을 포함한 반복학습과 특정 변수의 영향력이 커지는 것을 막아서 미연에 방지한 것이 랜덤 포레스트 모델이다. 특히 이 데이터를 예측하는 모델을 만드는데 sub-spacing 이 꼭 필요한 이유는 위 전처리에서 알 수 있는 것처럼 특정 몇몇 변수이 예측에 끼치는 영향력이 크기 때문이다. 이러한 점을 해소하기 위해서는 sub-spacing 이 꼭 필요하다. 사실 sub-spacing 과 bagging 은 KNN 과 같이 예측력이 낮은 다른 모델에 적용해도 된다. 물론 본 프로젝트에서는 이미 KNN 으로 만든 모델의 complexity 가 매우 큰 관계로 KNN 에 bagging 을 적용한 모델을 다룰 필요가 없다. 이처럼 bagging 가 sub-spacing 을 다룬 대표적인 tree 모델이 랜덤 포레스트 분류기를 활용 및 튜닝하여 모델을 생성해 볼 것이다. 이 밖에도 Boosting 을 활용한 GBM machine 과 같은 방법들이 존재한다. 그러나 Boosting 을 활용한 GBM machine 은 overfitting 이 일어나기

매우 쉬우므로 따로 다루지 않는다. 위의 결과들을 통해 데이터가 복잡해서 overfitting 이 쉽게 일어난다는 사실은 명확하게 확인할 수 있다. 그러므로 GBM machine 과 같은 Boosting 을 활용한 ensemble model 은 다루지 않을 것이다.

Random Forest 모델에서 사용할 수 있는 parameter 들은 매우 다양하다. 일단 random forest 모델은 n_estimator 라는 parameter 를 가지고 있다. 이 parameter 는 random forest 모델에서 사용할 tree 의 개수를 정한다. Criterion 은 결정 계수를 gini 혹은 entropy 로 할 것인지 정할 수 있게 해준다. Max_depth 는 트리의 최대 깊이를 지정한다. 너무 깊어지면 overfitting 이 발생할 수 있으므로 적절하게 설정해 준다. Min_sample_split 는 트리에서 노드를 분할하기 위한 최소 샘플의 개수를 나타낸다. 불균형 데이터에서 특정 class 만 선택하는 일이 발생할 수 있으므로 1 로 설정해 준다. Min_samples_leaf 는 leaf 노드가 가져야 하는 최소 샘플의 개수를 뜻하고 bootstrap set 은 bagging 을 통해 재표집을 허용할지 여부를 결정한다. 이러한 parameter 들이 존재하고 튜닝할 수 있다. 모두 다 튜닝하면 좋겠지만 이는 너무 연산 양이 많으므로 한 가지 혹은 두 가지 parameter 를 튜닝하면 된다. 여기선 overfitting 을 잡는 것이 중요함으로 overfitting 을 제어할 수 있는 max_depth 와 max_leaf_nodes 를 튜닝하고 min_sample_split 를 4 로 기본값보다 크게 설정해서 overfitting 을 방지한다. Max_leaf 는 고차원 데이터의 다양성을 고려하여 작게 설정하는 것도 좋지만 overfitting 문제를 방지하기 위해 1 로 설정하겠다. 이와 같이 설정하여 random forest classifier 모델은 구성한 결과는 다음과 같다.

```
tree_size = np.arange(2,50,2)
nlist = np.arange(10,50,2)
from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import GridSearchCV
params = {
          'max_leaf_nodes' : tree_size,
          'max_depth' : nlist
        }
f = GridSearchCV(
     RandomForestClassifier(
     n_estimators = 100,
     max_features = 1,
     max_depth = nlist,
     min_samples_split = 4,
     min_samples_leaf = 1,
     max_leaf_nodes = None,
    ), params )
f.fit(xtrain,ytrain)
```



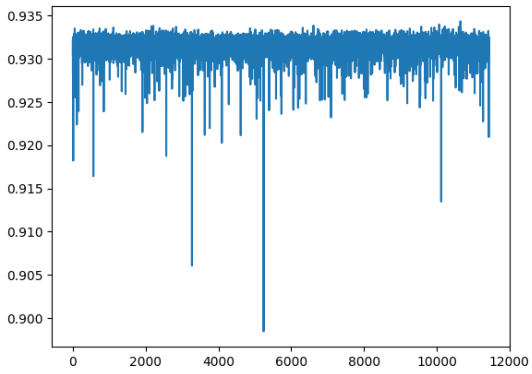
이 코드를 통해 얻은 결과는 다음과 같다.

```
#f1값 구하기
from sklearn.metrics import f1_score
probabilities = f.predict_proba(xtrain)[:,1]
threshold = [0.1,0.2,0.3,0.4,0.5,0.6,0.999]
f1_list = []
for i in range(len(threshold)):
  predictions = (probabilities > threshold[i]).astype(int)
  f1 = f1_score(ytrain, predictions, average = 'weighted')
  f1_list.append(f1)
print(f1_list)
print(max(f1_list))
```

```
[0.8985553202972857, 0.8985553202972857, 0.8985553202972857, 0.8985553202972857, 0.8985553202972857, 0.8985553202972857, 0.007715084053418995]
0.8985553202972857
```

```
plt.plot(probabilities)
```

```
[<matplotlib.lines.Line2D at 0x7f125eb88eb0>]
```



과하지 않지만 어느 정도 overfitting 되어 있는 것을 발견할 수 있다.

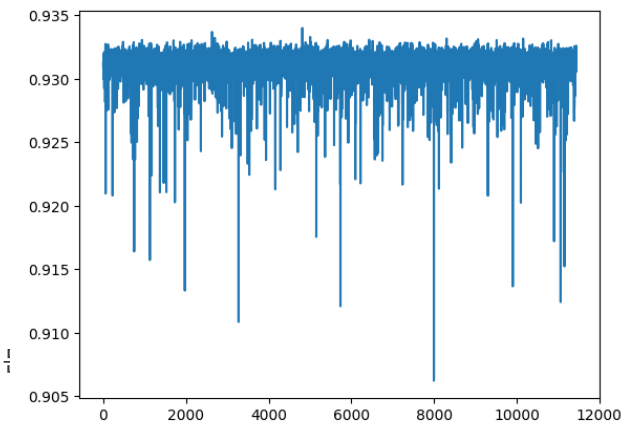
다음으로 PCA 를 적용시켜 보았다. PCA 를 적용시킨 훈련 데이터를 통해서 random forest classifier 모델을 구성한 결과는 다음과 같다.

```
#f1값 구하기
from sklearn.metrics import f1_score
probabilities = f.predict_proba(ztrain)[:,1]
threshold = [0.1,0.2,0.3,0.4,0.7,0.9,0.999]
f1_list = []
for i in range(len(threshold)):
  predictions = (probabilities > threshold[i]).astype(int)
  f1 = f1_score(ytrain, predictions, average = 'weighted')
  f1_list.append(f1)
print(f1_list)
print(max(f1_list))
```

```
[0.8985553202972857, 0.8985553202972857, 0.8985553202972857, 0.8985553202972857, 0.8985553202972857, 0.8985553202972857, 0.007715084053418995]
0.8985553202972857
```

```
plt.plot(probabilities)
```

```
[<matplotlib.lines.Line2D at 0x7f125e5db5b0>]
```



마지막으로 인공신경망 모델을 사용하겠다. 인공신경망 모델은 일반화된 선형 회귀 모델이라고 할 수 있다. 이 모델은 여러 가지 계층 구조로 이루어져 있다. 각 계층에는 activation 함수가 있고 입력값을 받아 activation 함수를 계산한 후 output 이 일정 이상이면 다음 output 을 출력하는 방식이다. 이런 식으로 계층 구조가 이어지다고 최종적으로 sigmoid 로 이루어진 output layer 를 통해 이진 분류를 하게 된다. 이를 이진 분류 perceptron 이라고 한다. 각 perceptron 은 여러 parameter 들과 input 들의 linear combination 으로 이루어진 input 을 받고 훈련을 하여 parameter 를 업데이트 하게 된다. Parameter 는 loss function 을 최적화시키는 function 을 찾아 업데이트 하게 된다. 만약 parameter 들이 모두 수렴하게 되면 훈련은 끝난다. 즉, 인공신경망은 아주 많은 parameter 들로 구성되어 있다. 이러한 parameter 은 모델에서 자동적으로 학습되는 parameter 들이고 learning rate, activation 함수, batch 사이즈, loss function 의 종류, 최적화 기법, layer 의 숫자, 각 layer 에 존재하는 perceptron 들의 input 숫자 등등을 튜닝 파라메터들로 선택할 수 있다. 물론, 이 parameter 들을 다 튜닝 하는 것은 불가능하다. 모델의 훈련 시간이 아주 길기 때문에 어느 파라메터를 튜닝하기 보다는 훈련 데이터를 보고 적절한 튜닝 파라미터를 선택해야 한다.

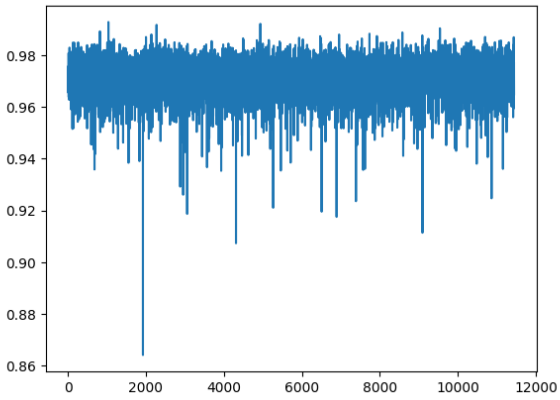
인공신경망 모델은 sklearn 라이브러리로 구현하게 되면 시간이 오래 걸리므로 tensorflow 를 이용하여 구현할 것이다. 그러므로 f1 score 를 얻는 방법도 달라진다. 지금까지 했던 것처럼 f.predict_prob 가 아니라 tensorflow 의 model.predict 를 이용하여 각 sample 의 probabilities 를 구할 수 있다. 그리고 불균형을 문제를 해결하기 위한 조치도 취할 수 있다. 바로 class_weight 를 설정하는 것이다. Layer 는 overfitting 을 해결하기 위해서 너무 복잡하지 않게 input 과 output layer 2 개만 설정했다. 그리고 activation 함수로는 각각 sigmoid, sigmoid, 그리고 softmax 를 사용하였다. 모델을 전체 훈련시키는 epoch 은 10 번만 돌려준다. 모델이 overfitting 되는 것을 방지하기 위해서다. 이와 같이 인공신경망 모델을 구성한 결과는 다음과 같다.

```
probabilities = model.predict(xtrain)[:,0]
threshold = [0.1,0.2,0.3,0.4,0.5,0.6,0.999]
f1_list = []
from sklearn.metrics import f1_score
for i in range(len(threshold)):
  predictions = (probabilities > threshold[i]).astype(int)
  f1 = f1_score(ytrain, predictions, average = 'weighted')
  f1_list.append(f1)
print(f1_list)
print(max(f1_list))
```

```
358/358 [==============================] - 1s 3ms/step
[0.008767508446475454, 0.008767508446475454, 0.008767508446475454, 0.008767508446475454, 0.008767508446475454, 0.008767508446475454, 0.8985553202972857]
0.8985553202972857
```

```
plt.plot(probabilities)
```

```
[<matplotlib.lines.Line2D at 0x7f0f422d1c90>]
```




```
[204] print(class_weight)
```

```
{1: 0.5734659410771252, 0: 7.8058748403576}
```

Class_weight 는 직접 계산해 주었고 overfitting 을 방지하기 위해 중간에 layer 는 하나도 삽입하지 않았다. 그리고 neural net 의 판단 기준을 따르기 위해 원래는 normal = 1, defect = 0 으로 설정하였지만 이를 바꾸어 주었다.

수행 결과는 그림과 같다. 앞서 본 RandomForest 모델처럼 굉장히 심하게 overfitting 이 되어 있고 threshold 가 높아야지 정확도가 높아진다. 그러므로 대부분의 모델을 normal 로 분류하는 경향이 있으므로 class imbalance 문제를 해결하는 것을 실패하였다.

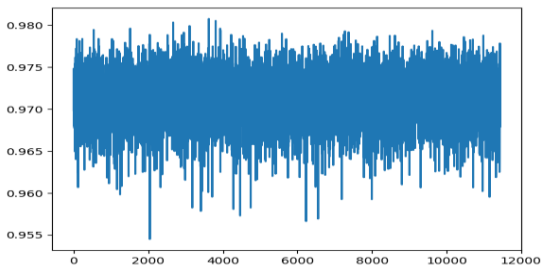
이제 PCA 를 적용하여 모델을 만들어 볼 것이다.

```
probabilities = model.predict(ztrain)[:,0]
threshold = [0.1,0.2,0.3,0.4,0.5,0.6,0.999]
f1_list = []
from sklearn.metrics import f1_score
for i in range(len(threshold)):
  predictions = (probabilities > threshold[i]).astype(int)
  f1 = f1_score(ytrain, predictions, average = 'weighted')
  f1_list.append(f1)
print(f1_list)
print(max(f1_list))
```

```
358/358 [==============================] - 1s 2ms/step
[0.008767508446475454, 0.008767508446475454, 0.008767508446475454, 0.008767508446475454, 0.008767508446475454, 0.008767508446475454, 0.8985553202972857]
0.8985553202972857
```

```
plt.plot(probabilities)
```

```
[<matplotlib.lines.Line2D at 0x7f1057949f00>]
```



큰 차이 없이 비슷한 모델이다. 즉 앞서 본 것과 같이 과하게 overfitting 되어 있다.

지금까지 여러 모델들을 구성해 보았다. 모델들을 바탕으로 최종 모델들을 선택하기 위해 두 가지 요소를 고려할 것이다. 첫 번째는 overfitting 문제에서 얼마나 자유로운가이다. 두 번째는 class imbalance 문제를 어느 정도 해결하고 있는지다. 이러한 기준을 보았을 때 가장 적절한 모델은 Logistic Regression 모델이다. 유일하게 overfitting 문제를 어느 정도 해소하고 class imbalance 문제에서 자유롭기 때문이다. 그 중에서도 PCA 를 적용시키지 않은 모델이 가장 성능이 좋은 것을 f1 score 를 비교하여 알 수 있다. 모델의 선택 기준은 이처럼 threshold 를 변화시켜 가며 모델의 f1 score 를 확인하여 overfitting 이슈에서 얼마나 자유로운지 보았다. 즉, 이를 통해 logistic regression 에 C = 100.0 이고 penalty = l2, solver = lbfgs, class_weight = 'balanced' 모델을 선택할 수 있다. Threshold = 0.1 이다.

4. 평가데이터 분류

찾은 모델을 통해 평가데이터를 분류하였다. 평가데이터는 훈련데이터에서 했던 방식과 동일하게 전처리 하였다. 전처리 방법은 훈련데이터와 동일한 feature 들을 삭제해 주고 대치도 훈련데이터의 평균값으로 대치하는 방식으로 진행하였다. 그 결과는 txt 파일에 저장하였다.

5. 총 이득 산정

다음 코드를 통해 총 이득은 쉽게 산정할 수 있다. 총 이득을 산정할 때도 데이터 전처리를 해 주어야 하고 대신 샘플을 삭제하면 안 된다. 훈련은 샘플까지 전처리 한 데이터로 하고 예측은 샘플은 삭제하지 않은 데이터로 한다.

```
pd.crosstab(ytrain_new, predictions)
```

col_0	0	1
row_0		
0	40	700
1	243	10508

```
#위에서 사용한 모델을 쓰면 결과는 다음과 같다. (0은 defect이고 1은 normal이다.)
TP = pd.crosstab(predictions, ytrain_new)[0][0]
FN = pd.crosstab(predictions, ytrain_new)[0][1]
FP = pd.crosstab(predictions, ytrain_new)[1][0]
TN = pd.crosstab(predictions, ytrain_new)[1][1]
profit = 900 * TN - 100 * TP - 1100 * FP - 100 * FN
print(profit)
```

```
9115900
```

총 이득은 9115900 원으로 이상적인 모델보다는 적고 모델을 쓰지 않은 경우 보다는 많은 이익을 얻었음을 알 수 있다. 사실 이 모델은 예측하기 위해서 overfitting 이 되어 있지 않은 모델을 선정했다. 그래서 모델의 precision 과 recall 이 낮다. 이득도 낮아졌다.

6. 이상적인 모델 가정 2

만약 반도체 판매가격이 줄어들었을 때를 가정한다. 다음 코드를 통해 이상적인 경우와 모델을 사용하지 않을 경우 각각에 대해 이익을 알 수 있다.

```
[586] #Q6
      data = pd.read_csv('data98_semi_train.csv')
      sum_defect = sum(data['Label'] == 'defect')
      sum_normal = sum(data['Label'] == 'normal')
      #정상적인 칩을 생산하여 얻을 수 있는 이익은 900원에서 100원으로 줄어들었고 불량인 칩을 생산하면 보게 되는 손해는 1100원이다. 따라서 훈련 데이터 셋을 통해 계산할 수 있는 총 이득은 다음과 같다.
      cost = 100 * sum_normal - 1100 * sum_defect

[100] print(cost)

      261100

[102] #다음으로 데이터 분석을 통하여 불량을 완벽히 판별하는 이상적인 모델을 만들었을 경우 총 이득을 구해보자. 이 경우 정상적인 칩을 정상적으로 판별하였을 때 900원의 이득을 얻을 수 있고 불량인 칩을 판별
      #계산할 수 있는 총 이득은 다음과 같다.
      cost = 100 * sum_normal - 100 * sum_defect

      print(cost)

      1001100

[538] #위에서 사용한 모델을 쓰면 결과는 다음과 같다.
      TP = 40
      FN = 700
      FP = 243
      TN = 10508
      profit = 100 * TN - 100 * TP - 100 * FP - 1100 * FN

[539] print(profit)

      252500
```

이상적인 모델 가정 3

훈련 데이터 셋과 지금까지 개발한 모델 중에 이러한 경우 현재 모델보다 향상시킬 수 있는 모델이 존재하지 않는다. 왜냐하면 다른 모델들은 너무 과하게 overfitting 이 되고 class imbalance 문제를 겪고 있기 때문에 항상 normal 을 예측할 것이고 그러면 normal 을 예측했을 때 기대값이 -1000 원으로 defect 를 예측했을 때 보다 크므로 이윤이 적어질 수밖에 없다. 트리 모델 같이 단순한 모델을 쓴다면 defect 를 더 많이 예측할 것이므로 이윤이 많아질 수 있다. 아니면 이 모델에서 threshold 를 높이면 된다.

7. 이상적인 모델 가정 3

```
] #Q6
  data = pd.read_csv('data98_semi_train.csv')
  sum_defect = sum(data['Label'] == 'defect')
  sum_normal = sum(data['Label'] == 'normal')
  #정상적인 칩을 생산하여 얻을 수 있는 이익은 900원에서 3900원으로 늘었고 불량인 칩
  cost = 3900 * sum_normal - 1100 * sum_defect

] print(cost)

  41114900

] #다음으로 데이터 분석을 통하여 불량을 완벽히 판별하는 이상적인 모델을 만들었을 경
  #계산할 수 있는 총 이득은 다음과 같다.
  cost = 3900 * sum_normal - 100 * sum_defect

] print(cost)

  41854900

] #위에서 사용한 모델을 쓰면 결과는 다음과 같다.
  TP = pd.crosstab(predictions, ytrain_new)[0][0]
  FN = pd.crosstab(predictions, ytrain_new)[0][1]
  FP = pd.crosstab(predictions, ytrain_new)[1][0]
  TN = pd.crosstab(predictions, ytrain_new)[1][1]
  profit = 3900 * TN - 100 * TP - 1100 * FP - 100 * FN
  print(profit)

  40639900
```

이 경우는 이익을 향상시킬 수 있는 모델이 존재한다. 모든 반도체를 normal 하다고 가정해도 이익이 더 커질 수 있기 때문이다. 위의 모델을 예시로 들면 neural net 이나 KNN 모델은 class imbalance 문제와 overfitting 문제로 인해 100%에 가깝게 normal 을 예측한다. 이런 모델들은 설사 예측이 틀리더라도 기대 이익이 1900 원으로 defect 를 예측했을 때의 -200 원 보다 훨씬 높기 때문에 더 높은 이익을 얻을 수 있다. 따라서 이익을 향상시킬 수 있는 모델은 매우 많이 존재한다. 두 경우에 대해서 알 수 있는 점은 좋은 모델을 만들기 어렵다는 점이다. 이렇게 defect 가 적은 경우 class imbalance 문제에 걸려 좋은 모델을 생산하기 특히 어려워진다.