중간평가 경진대회 보고서

과목명 : 딥러닝/클라우드
학번 : 32152339
주전공 : 응용통계학과
이름 : 송준영

목차

Ⅰ. 개요

1) 분석 과정 요약

- 탐색적 자료 분석
  - 이상치 제거하지 않음.
  - var_11+var_17, var_5/var_9, var_11*var_17를 추가.
  - var_26, var_30 변수를 삭제.

- 모델링
  - base learner : 5-fold stratified cross validation으로 검증
  - stacking : 10-fold stratified cross validation으로 검증

Agorithm	Feature set	Accuracy	Parameter tuning	Stacking	Stacking accuracy
Random Forest	raw ver	91.0748%	randomized	x	93.1776%
	modified ver	91.1682%	grid	x	
XGBoost	raw ver	91.7056%	Bayesian	o	
	modified ver	92.0794%	Bayesian	x	
LightGBM	raw ver	91.5888%	randomized	x	
	modified ver	92.4065%	grid	o	
MLP	modified ver	89.1589%	grid	x	
SVM	modified ver	89.1822%	grid	x	
KNN	raw ver	91.0748%	randomized	o	
	modified ver	90.5841%	grid	x	
ExtraTrees	modified ver	91.0514%	grid	x	
HistGBM	modified ver	91.4953%	grid	x	
CatBoost	raw ver	91.8224%	randomized	x	
	modified ver	91.8458%	grid	x	

- 최종 결과
  - **cv accuracy : 93.1776%**
  - **test accuracy : 0.936681222707424**
  - **경진대회 최종 test accuracy 1위**

Ⅱ. 탐색적 자료 분석

▶ Ⅱ-1) 데이터 탐색

데이터 분석에 앞서 일단 가장 먼저 행해야하는 것은 데이터를 자세히 살펴보는 것이다.

```
In [4]: trn
```

Out[4]:

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29
0	HI	188	128	95	114	143	108	88	103	113	85	88	113	87	88	103	87	84	99	104	82	96	100	78	70	79	84	66	70	75
1	PH	174	112	88	104	119	92	74	79	88	74	67	90	68	71	73	68	71	77	90	67	71	82	65	70	75	89	73	67	71
2	GR	175	138	106	105	135	109	75	95	113	96	74	112	96	70	87	100	66	83	117	67	88	110	98	67	88	119	98	75	91
3	PH	176	111	80	106	131	96	76	99	104	85	75	89	75	79	91	75	84	103	109	82	91	96	78	78	91	96	78	82	104
4	EL	182	144	111	100	151	119	67	106	114	90	76	115	94	68	106	91	68	102	115	71	95	108	88	71	103	113	92	68	107
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
4275	EL	153	120	95	83	123	104	53	83	100	85	51	100	79	51	75	79	51	72	89	49	75	100	78	52	67	84	78	52	71
4276	PH	179	110	82	109	135	97	83	95	97	75	78	93	73	82	92	76	78	88	85	76	85	86	68	80	94	94	76	80	89
4277	EL	171	138	106	106	152	122	76	112	122	99	79	118	96	84	116	96	75	107	123	66	96	112	92	70	100	117	92	66	109
4278	PH	175	109	81	105	123	81	71	79	85	67	78	88	70	74	87	66	78	87	84	75	84	90	68	75	88	97	75	75	88
4279	EL	180	142	106	106	147	118	84	116	128	103	84	118	96	71	79	92	79	103	123	78	104	112	96	74	83	108	89	66	71

4280 rows × 32 columns

첫 번째 열이 종속변수이고, 나머지 31개의 독립변수로 데이터가 구성되어 있다. 변수명이 없기 때문에 feature engineering에 있어 domain에서의 도움을 얻긴 어려울 것으로 보인다. 모든 독립변수가 정수형이다.

```
In [16]: trn.describe()
```

Out[16]:

	1	2	3	4	5	6	7	8	9	10	11	12	13	
count	4280.0000	4280.0000	4280.0000	4280.0000	4280.0000	4280.0000	4280.0000	4280.0000	4280.0000	4280.0000	4280.0000	4280.0000	4280.0000	4280
mean	169.2586	119.2404	92.5893	99.0579	123.0671	99.1105	68.8350	82.7350	98.8636	82.4787	69.1600	99.3841	82.8103	68
std	13.6195	16.5321	18.8221	13.5730	22.8854	16.6022	13.4705	22.8976	16.6052	19.0474	13.6048	16.4668	18.9254	13
min	139.0000	73.0000	43.0000	70.0000	67.0000	50.0000	40.0000	27.0000	50.0000	29.0000	39.0000	50.0000	29.0000	40
25%	160.0000	105.0000	79.0000	90.0000	111.0000	85.0000	60.0000	71.0000	85.0000	68.0000	60.0000	86.0000	70.0000	60
50%	168.0000	121.0000	91.0000	98.0000	125.0000	101.0000	67.0000	84.0000	100.0000	81.0000	68.0000	101.0000	81.0000	68
75%	180.0000	133.0000	102.0000	110.0000	143.0000	113.0000	79.0000	102.0000	113.0000	92.0000	80.0000	113.0000	92.0000	79
max	202.0000	160.0000	164.0000	134.0000	177.0000	145.0000	102.0000	130.0000	145.0000	154.0000	104.0000	145.0000	154.0000	104

위 표에서는 각 변수의 평균, 표준편차, 최솟값, 최댓값, 사분위수 등을 관찰할 수 있는데 min과 max가 일사분위수와 삼사분위수에 근접한 것으로 보아 이상치가 없을 것으로 판단된다. 이상치에 대한 것은 다음 장 Ⅱ-2)시각화에서 자세히 살펴보도록 한다.

```
In [6]: trn.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 4280 entries, 0 to 4279
Data columns (total 32 columns):
0      4280 non-null object
1      4280 non-null int64
2      4280 non-null int64
3      4280 non-null int64
4      4280 non-null int64
5      4280 non-null int64
```

위와 같이 데이터형은 종속변수를 제외하고 정수형이므로 별다른 형변환 작업이 필요 없을 것으로 보이나 추후 분석의 용이함을 위하여 변수명을 변경해 준다.

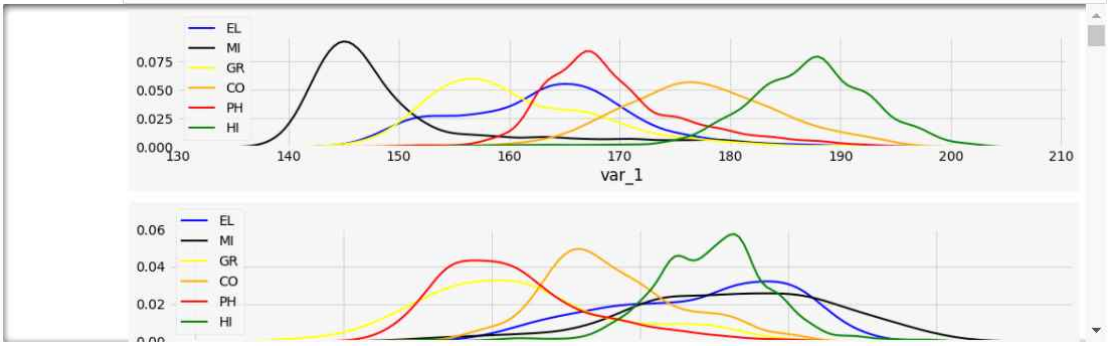
```
In [7]: trn[0].value_counts() #6개 level의 범주형 target

Out[7]: EL    1017
        PH     981
        HI     897
        MI     492
        GR     469
        CO     424
        Name: 0, dtype: int64
```

종속변수는 총 6개의 level의 범주형 변수이며, 각 수준의 빈도는 위와 같다. 수준별 빈도가 불균형하므로 k-fold보다는 **stratified k-fold cross validation**을 통해 검증함이 적절할 것으로 판단된다.

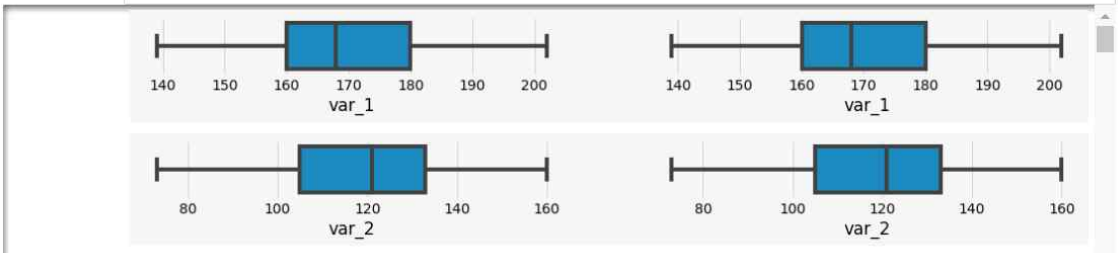
▶ Ⅱ-2) 시각화

```
In [27]: grp_col_dict = {'EL': 'blue',
                         'MI':'black',
                         'GR':'yellow',
                         'CO':'orange',
                            'PH': 'red',
                            'HI': 'green'}
for i in range(1,32):
    plt.figure(figsize=(15,2))
    for group in grp_col_dict:
        # subset of group
        subset = trn[trn['class'] == group]
        # histogram and kernel density curve
        sns.distplot(subset[f'var_{i}'],
                             hist = False, # histogram
                             kde = True,  # density curve
                              kde_kws = {'linewidth': 2},
                            color = grp_col_dict[group],
                            label = group)
    plt.show()
```



모든 변수의 종속변수 수준별 distplot을 살펴본 결과, **var_1, var_7, var_11, var_14, var_15, var_17, var_18, var_24, var_25, var_28, var_29**가 **구분 잘되는 변수**로 보인다. 다음 장인 Ⅱ-3)feature engineering에서 상기 줄에 언급하였던 변수들끼리 연산하여 feature를 추가해보기로 한다.

```
In [24]: for i in range(1,trn.shape[1]):
             plt.figure(figsize=(15,1))
             plt.subplot(1,2,1)
             sns.boxplot(x=f"var_{i}",data=trn)
             plt.subplot(1,2,2)
             sns.boxplot(x=f"var_{i}",data=trn)
             plt.show()
```



모든 변수의 boxplot을 살펴본 결과, **var_3, var_10, var_13, var_16, var_23, var_27, var_31**의 변수가 **이상치**가 존재하는 것으로 보인다. 하지만 testset에도 trainset과 유사한 이상치가 존재하므로 제거하지 않는다.

또한 모든 변수가 정규분포의 형태를 나타내므로 별다른 변수 변환하지 않는 것으로 한다.

```
In [25]: trn.corr().style.background_gradient() # 변수별 상관관계 확인
```

Out[25]:

	var_1	var_2	var_3	var_4	var_5	var_6	var_7	var_8	var_9	var_10	var_11	var_12	var_13	var_14	var_15	var_16	var_17
var_1	1	0.2119	-0.1758	0.952	0.7729	0.1794	0.8815	0.7067	0.1502	-0.185	0.9344	0.1879	-0.1823	0.9152	0.7485	-0.1956	0.8601
var_2	0.2119	1	0.8614	0.2038	0.3384	0.9332	0.1742	0.3065	0.8446	0.7331	0.1841	0.9079	0.7998	0.177	0.3171	0.7714	0.1594
var_3	-0.1758	0.8614	1	-0.1759	-0.117	0.8332	-0.1779	-0.1149	0.7591	0.857	-0.1762	0.8003	0.9187	-0.1843	-0.1206	0.8998	-0.1832
var_4	0.952	0.2038	-0.1759	1	0.8104	0.2015	0.9522	0.7723	0.178	-0.1877	0.9077	0.1841	-0.1765	0.9319	0.7672	-0.1896	0.9128
var_5	0.7729	0.3384	-0.117	0.8104	1	0.3448	0.7887	0.9581	0.3196	-0.1281	0.7271	0.3196	-0.1165	0.7589	0.9387	-0.1274	0.7522
var_6	0.1794	0.9332	0.8332	0.2015	0.3448	1	0.1925	0.3321	0.9339	0.8154	0.1618	0.8782	0.7806	0.1748	0.32	0.8018	0.173
var_7	0.8815	0.1742	-0.1779	0.9522	0.7887	0.1925	1	0.8117	0.1986	-0.1861	0.8571	0.1641	-0.1758	0.9043	0.745	-0.1842	0.9307
var_8	0.7067	0.3065	-0.1149	0.7723	0.9581	0.3321	0.8117	1	0.3416	-0.1246	0.676	0.2992	-0.1108	0.7265	0.9054	-0.1174	0.76
var_9	0.1502	0.8446	0.7591	0.178	0.3196	0.9339	0.1986	0.3416	1	0.864	0.1384	0.8241	0.7356	0.1588	0.3017	0.7802	0.1782
var_10	-0.185	0.7331	0.857	-0.1877	-0.1281	0.8154	-0.1861	-0.1246	0.864	1	-0.1793	0.7176	0.8296	-0.182	-0.1166	0.8834	-0.1779

변수별 상관관계를 확인해보니 상관계수가 높은 쌍의 변수들이 몇 확인된다.

```
In [13]: np.sum(abs(trn.corr())>0.9).sort_values(ascending=False)
```

Out[13]:

```
var_14    9
var_15    7
var_18    6
var_4     6
var_24    6
var_11    6
var_17    6
var_25    5
var_16    4
var_5     4
var_7     4
var_8     4
var_13    4
var_1     4
var_20    4
var_29    4
var_28    4
var_27    4
var_19    3
```

수치로 확인하였을 때 **var_14**의 경우 상관계수가 +- 0.9를 넘는 것이 9쌍으로 확인되었다. 제거하기엔 아직 modeling에서의 변수의 영향력을 확인하기 이전이라 보류한다.

▶ Ⅱ-3) Feature engineering

```
In [39]: xgb_p_val = np.zeros((ftr.shape[0], n_class))
         xgb_p_tst = np.zeros((tst_ar.shape[0], n_class))
         for i, (i_trn, i_val) in enumerate(cv.split(ftr, target), 1):
             print(f'training model for CV #{i}')
             xgb_clf = xgb.XGBClassifier()
             xgb_clf.fit(ftr[i_trn], target[i_trn],
                     eval_set=[(ftr[i_val], target[i_val])],verbose=False)
             xgb_p_val[i_val, :] = xgb_clf.predict_proba(ftr[i_val])
             xgb_p_tst += xgb_clf.predict_proba(tst_ar) / n_fold
             print(f'{i}fold accuarcy:',accuracy_score(target[i_val],np.argmax(xgb_p_val[i_val],axis=1)))
             print()
         print(f'{accuracy_score(target, np.argmax(xgb_p_val, axis=1)) * 100:.4f}%')
         print(ftr.shape)
         print(xgb_clf)
```

```
89.8832%
(4280, 31)
XGBClassifier(objective='multi:softprob')
```

feature engineering 하기에 앞서 **5-fold stratified cross-vaildation**으로 raw data를 **xgboost**로 학습해보니 **89.8832%** 의 accuracy를 얻었다. feature set 성능비교의 base model로 xgboost를 선택한 이유는 random forest, logistic regression, knn, svm을 모두 실험해본 결과 가장 높은 성능이 나왔기 때문이다. 이제부터는 이 성능의 유의미한 향상을 일으키는 feature set을 선택하거나 feature 중요도 분석(**eli5 모듈의 permutation importance**[1])을 통해 중요도가 높은 feature를 raw dataset에 추가하기로 한다.

먼저, 원 독립변수 31개에 대한 모든 조합의 사칙연산 feature set들을 생성한다. 예를 들어 덧셈 연산 feature set은 var1+var2, var1+var3 … var30+var31 으로 구성되어 있다.

- 모든 조합의 (+) 연산 feature set 생성

```
In [48]: trn_add = trn.drop('class', axis=1)
         tst_add = tst.copy()
         for df in [trn_add,tst_add]:
             for c1,c2 in itertools.combinations(df.columns,2):
                 col='add_{c1}_{c2}'
                 df[col]=df[c1]+df[c2]
         ftr=trn_add.values
         tst_ar=tst_add.values
```

위처럼 모든 조합에 따라 덧셈 연산을 실시한 feature을 raw dataset에 추가한다.

```
90.3271%
(4280, 496)
XGBClassifier(objective='multi:softprob')
```

위 baseline model로 496개의 덧셈 연산 feature set을 학습해보니 **90.3271%**라는 accuaracy를 얻을 수 있었다. feature의 개수가 굉장히 늘어났음에도 불구하고 accuracy의 큰 향상은 없다고 판단되므로 eli5 모듈의 permutation importance를 진행한다.

```
In [57]: perm = PermutationImportance(xgb_clf, scoring = "accuracy").fit(ftr, target)
         eli5.explain_weights_df(perm, top = ftr.shape[1], feature_names = trn_add.columns.tolist()).head()
```

Out[57]:

	feature	weight	std
0	add_var_11_var_17	0.0280	0.0021
1	add_var_7_var_24	0.0067	0.0009
2	add_var_3_var_27	0.0052	0.0014
3	add_var_4_var_28	0.0047	0.0003
4	add_var_7_var_23	0.0046	0.0008

그 결과 **var_11 + var_17** 이라는 변수가 압도적으로 높은 정확도를 출력하였다. 해당 변수는 modeling 시 추가를 시도한다.

다음은, 뺄셈 연산을 실시한 feature을 raw dataset에 추가한다.

- 모든 조합의 (-) 연산 feature set 생성

```
In [60]: trn_dif = trn.drop('class', axis=1)
         tst_dif = tst.copy()
         for df in [trn_dif,tst_dif]:
             for c1,c2 in itertools.combinations(df.columns,2):
                 col='dif_{c1}_{c2}'
                 df[col]=df[c1]-df[c2]
         ftr=trn_dif.values
         tst_ar=tst_dif.values
```

1) https://eli5.readthedocs.io/en/latest/blackbox/permutation_importance.html

```
In [64]: perm = PermutationImportance(xgb_clf, scoring = "accuracy").fit(ftr, target)
         eli5.explain_weights_df(perm, top = ftr.shape[1], feature_names = trn_dif.columns.tolist()).head()
```

Out[64]:

	feature	weight	std
0	var_14	0.0074	0.0015
1	var_15	0.0031	0.0013
2	var_9	0.0027	0.0013
3	var_17	0.0024	0.0009
4	var_19	0.0020	0.0009

역시 큰 성능 향상은 없었고, permutation importance 결과 뺄셈 연산에 대한 feature도 상위에 존재 하지 않았다.

다음은, 나눗셈 연산을 실시한 feature을 raw dataset에 추가한다.

- 모든 연산의 (/) 연산 feature set 생성

```
In [65]: trn_div = trn.drop('class', axis=1)
         tst_div = tst.copy()
         for df in [trn_div,tst_div]:
             for c1,c2 in itertools.combinations(df.columns,2):
                 col=f'div_{c1}_{c2}'
                 df[col]=df[c1]/df[c2]
         ftr=trn_div.values
         tst_ar=tst_div.values
```

```
In [67]: perm = PermutationImportance(xgb_clf, scoring = "accuracy").fit(ftr, target)
         eli5.explain_weights_df(perm, top = ftr.shape[1], feature_names = trn_div.columns.tolist()).head()
```

Out[67]:

	feature	weight	std
0	var_14	0.0060	0.0019
1	var_15	0.0024	0.0012
2	div_var_5_var_9	0.0024	0.0008
3	var_29	0.0024	0.0008
4	div_var_1_var_19	0.0018	0.0007

큰 성능 향상은 없었으나, **var_5/var_9** 변수가 중요도가 높은 변수임을 알아냈다.

다음은, 곱셈 연산을 실시한 feature을 raw dataset에 추가한다.

- 모든 연산의 (*) 연산 feature set 생성

```
In [68]: trn_mul = trn.drop('class', axis=1)
         tst_mul = tst.copy()
         for df in [trn_mul,tst_mul]:
             for c1,c2 in itertools.combinations(df.columns,2):
                 col=f'mul_{c1}_{c2}'
                 df[col]=df[c1]/df[c2]
         ftr=trn_mul.values
         tst_ar=tst_mul.values
```

```
In [94]: perm = PermutationImportance(xgb_clf, scoring = "accuracy").fit(ftr, target)
         eli5.explain_weights_df(perm, top = ftr.shape[1], feature_names = trn_mul.columns.tolist()).head()
```

Out[94]:

	feature	weight	std
0	mul_var_11_var_17	0.0276	0.0011
1	mul_var_5_var_15	0.0059	0.0007
2	mul_var_7_var_24	0.0053	0.0005
3	mul_var_3_var_27	0.0038	0.0012
4	mul_var_5_var_21	0.0035	0.0006

그 결과 **var_11 * var_17** 이라는 변수가 압도적으로 높은 정확도를 출력하였다. 해당 변수는 modeling 시 추가를 시도한다.

다음은 Ⅱ-2)시각화 part에서 찾은 변수들을 연산하여 raw data에 추가해보기로 한다.

```
In [95]: eda_vars=['var_1', 'var_7', 'var_11', 'var_14', 'var_15', 'var_17', 'var_18', 'var_24', 'var_25', 'var_28', 'var_29']

In [100]: trn_eda = trn.drop('class', axis=1)
          tst_eda = tst.copy()
          for df in [tst_eda,trn_eda]:
              for var in eda_vars:
                  col_squared=f'squared_{var}'
                  col_sqrt=f'sqrt_{var}'
                  df[col_squared]=df[var]**2
                  df[col_sqrt]=df[var]**0.5
          ftr=trn_mul.values
          tst_ar=tst_mul.values
```

```
In [111]: eli5.show_weights(perm, top = ftr.shape[1], feature_names = trn_eda.columns.tolist())
```

Out[111]:

Weight	Feature
0.0561 ± 0.0030	var_18
0.0197 ± 0.0024	var_15
0.0126 ± 0.0026	var_16
0.0118 ± 0.0021	var_10
0.0117 ± 0.0029	var_21
0.0110 ± 0.0031	var_5
0.0103 ± 0.0028	var_11

eda를 통해 얻은 연산 피쳐들이 permutation importance 결과 중요하지 않은 것으로 보여진다.

지금까지 다양한 연산을 통하여 feature을 추가하여 중요한 변수들을 알아보았다. 그 결과, **var_11+var_17, var_5/var_9, var_11*var_17**라는 변수를 얻을 수 있었다. 이를 추가하여 중요도 분석을 실시한다.

```
In [116]: perm = PermutationImportance(xgb_clf, scoring = "accuracy").fit(ftr, target)
          eli5.show_weights(perm, top = ftr.shape[1], feature_names = trn_imp.columns.tolist())
```

Out[116]:

Weight	Feature
0.0241 ± 0.0052	mul_var_11_var_17
0.0211 ± 0.0048	var_18
0.0171 ± 0.0036	div_var_5_var_9
0.0124 ± 0.0041	var_15
0.0102 ± 0.0035	var_16
0.0098 ± 0.0019	var_21
0.0074 ± 0.0048	var_10
0.0069 ± 0.0025	var_24
0.0069 ± 0.0026	add_var_11_var_17
0.0068 ± 0.0021	var_3
0.0067 ± 0.0024	var_13

추가한 피쳐들이 위와 같이 높은 중요도를 가진 것으로 나타난다. 이렇게 생성한 feature set을 사용하기에 앞서 반대로 중요하지 않은 변수도 삭제해준다. 고차원에서 오는 과적합을 일부 방지하기 위해서이다.

0.0008 ± 0.0022	var_2
0.0001 ± 0.0002	var_25
0.0000 ± 0.0013	var_22
-0.0005 ± 0.0021	var_30
-0.0005 ± 0.0012	var_26

위처럼 **var_26, var_30 변수를 삭제**한 후 본격적인 modeling을 시작한다.

Ⅲ. 모델링

모델링을 기술하기에 앞서 개략적인 모델링의 과정을 설명한다. 해당 경진대회를 통해 다양한 알고리즘 및 파라미터 튜닝법을 익히기 위하여 최대한 다양하고 많은 알고리즘과 튜닝법을 사용할 예정이다. 다양한 모델을 위 데이터셋으로 학습시킨 후 마지막에는 그 개별 모델들을 ensemble하여 성능을 끌어올릴 것이다. ensemble의 **base leaner로는 random forest, xgboost, lightGBM, MLPclassifier, SVM, KNN, extraTREE, catboost, histGBM** 이렇게 총 9개를 시도하였다. [2]scikit-learn 홈페이지에 보면 classification 알고리즘을 보기 좋게 정리해놓아 이를 참고하여 다양한 알고리즘을 시도 할 수 있었다. 또한 모든 모델은 **Ⅱ-1) 데이터 탐색**에서 기술하였듯이 **stratified 5-fold cross validation**을 사용한다.

파라미터 튜닝으로는 **grid search, randomized search, Bayesian(hyperopt)**[3]를 사용할 것이다.

```
In [131]: n_fold = 5
          n_class = 6
          seed = 32152339
          cv = StratifiedKFold(n_splits=n_fold, shuffle=True, random_state=seed)

In [132]: print(trn_imp.shape)
          print(tst_imp.shape)

          (4280, 32)
          (1833, 32)
```

우선 모델링 환경을 구축한다. seed, cv 등을 정의하고, dataset을 확인한다.

▶ 1) Random Forest

grid search를 통해 최적의 파라미터를 찾는다. random forest는 정수형이나 캐릭터형 파라미터 옵션이 중요하기 때문에 **grid search**를 사용하였다.

```
In [136]: parameters = {'n_estimators': [1000,2000],
                        'criterion': ['gini','entropy'],
                        'max_depth': [15,16,17,18]
              }
          clf = GridSearchCV(RandomForestClassifier(n_jobs=-1,random_state=seed),
                             parameters,
                             verbose=True,
                             cv=cv,
                             n_jobs=-1)
          clf.fit(ftr, target)

          Fitting 5 folds for each of 16 candidates, totalling 80 fits

          [Parallel(n_jobs=-1)]: Using backend LokyBackend with 8 concurrent workers.
          [Parallel(n_jobs=-1)]: Done  34 tasks      | elapsed:  1.9min
          [Parallel(n_jobs=-1)]: Done  80 out of  80 | elapsed:  4.7min finished

Out[136]: GridSearchCV(cv=StratifiedKFold(n_splits=5, random_state=32152339, shuffle=True),
                       estimator=RandomForestClassifier(n_jobs=-1, random_state=32152339),
                       n_jobs=-1,
                       param_grid={'criterion': ['gini', 'entropy'],
                                   'max_depth': [15, 16, 17, 18],
                                   'n_estimators': [1000, 2000]},
                       verbose=True)

In [139]: print(clf.best_params_)

          {'criterion': 'gini', 'max_depth': 17, 'n_estimators': 2000}
```

2) https://scikit-learn.org/stable/supervised_learning.html#supervised-learning
3) http://hyperopt.github.io/hyperopt/

max_depth 같은 경우에는 직접 대략적인 깊이를 탐색하고 자세한 깊이는 grid search를 통해 얻었다. 그 결과 gini 함수를 사용하며, 17의 깊이가 성능이 가장 좋은 것으로 나타났다.

```
In [ ]: rf_p_val = np.zeros((ftr.shape[0], n_class))
rf_p_tst = np.zeros((tst_ar.shape[0], n_class))
for i, (i_trn, i_val) in enumerate(cv.split(ftr, target), 1):
    print(f'training model for CV #{i}')
    rf_clf = RandomForestClassifier(n_estimators = 2000,
                                    random_state=seed,
                                        criterion='gini',
#                                       criterion='entropy',
                                    verbose=False,
                                    oob_score=True,
                                    n_jobs=-1,
                                    max_depth=17)
    rf_clf.fit(ftr[i_trn], target[i_trn])
    rf_p_val[i_val, :] = rf_clf.predict_proba(ftr[i_val])
    rf_p_tst += rf_clf.predict_proba(tst_ar) / n_fold
    print(f'{i}fold accuarcy:',accuracy_score(target[i_val],np.argmax(rf_p_val[i_val],axis=1)))
```

```
In [143]: print(rf_clf)
print('valid accuracy : ',f'{accuracy_score(target, np.argmax(rf_p_val, axis=1)) * 100:.4f}%')
```

```
RandomForestClassifier(max_depth=17, n_estimators=2000, n_jobs=-1,
                       oob_score=True, random_state=32152339, verbose=False)
valid accuracy :  91.1682%
```

이에 따라 얻은 valid cross validation accuracy는 **91. 1682%**이다.

```
In [ ]: algo_name = 'rf'
feature_name = '91.1682'
model_name = f'{algo_name}_{feature_name}'

p_val_file = val_dir / f'{model_name}.val.csv'
p_tst_file = tst_dir / f'{model_name}.tst.csv'

np.savetxt(p_val_file, rf_p_val, fmt='%.6f', delimiter=',')
np.savetxt(p_tst_file, rf_p_tst, fmt='%.6f', delimiter=',')
```

ensemble에 사용하기 위하여 클래스별 확률 예측값을 local환경에 저장한다.

▶ 2) XGBoost

XGBoost[4]의 경우, parameter의 수가 많고 실수형 parameter가 꽤 존재하므로 **hyperopt** 모듈을 활용하여 Bayesian hyper parameter optimization을 진행한다.

```
In [304]: params = {"objective": "multiclass",
          "random_state": seed,
          "n_jobs": -1,
            'tree_method':'exact'}

space = {
    "max_depth":hp.choice('max_depth', np.arange(15,25,1)),
    "n_estimators": hp.choice('n_estimators', np.arange(100,500,5)),
    "learning_rate": hp.loguniform("learning_rate", np.log(0.01), np.log(0.15)),
    "colsample_bytree": hp.quniform("colsample_bytree", .8, 1.0, 0.1),
    "subsample": hp.quniform("subsample", .8, 1.0, 0.1),
    "min_child_weight": hp.choice('min_child_weight', np.arange(1,20,1,dtype='int'))}

# fit_params={'early_stopping_rounds': 50}
```

```
In [305]: def objective(hyperparams):
    model = xgb.XGBClassifier(**params,**hyperparams)
    best_score=cross_val_score(model,ftr,target,
                               scoring='accuracy',
#                                  fit_params = fit_params,
                               cv=cv).mean()
    return {'loss' : 1-best_score, 'status' : STATUS_OK, 'model': model}

trials = Trials()
best = fmin(fn=objective,space=space,trials=trials,
            algo=tpe.suggest,max_evals=300)

hyperparams = space_eval(space,best)
# n_best = trials.best_trial['result']['model'].best_iteration_
params.update(hyperparams)
print(params)
```

최적의 파라미터를 위와 같이 찾았으며, 해당 파라미터로 학습을 진행한다.

4) https://xgboost.readthedocs.io/en/latest/

```
100%|██████████| 300/300 [3:12:11<00:00, 38.44s/trial, best loss: 0.0829439252336449]
{'objective': 'multiclass', 'random_state': 32152339, 'n_jobs': -1, 'tree_method': 'exact', 'colsample_bytree': 0.8, 'learning_rate': 0.07
074858258554872, 'max_depth': 18, 'min_child_weight': 1, 'n_estimators': 395, 'subsample': 1.0}
```

```
In [224]: xgb_p_val = np.zeros((ftr.shape[0], n_class))
          xgb_p_tst = np.zeros((tst_ar.shape[0], n_class))
          for i, (i_trn, i_val) in enumerate(cv.split(ftr, target), 1):
              print(f'training model for CV #{i}')
              xgb_clf = xgb.XGBClassifier(**params)
              xgb_clf.fit(ftr[i_trn], target[i_trn],
                          eval_set=[(ftr[i_val], target[i_val])]
                               ,early_stopping_rounds=100
                          )

              xgb_p_val[i_val, :] = xgb_clf.predict_proba(ftr[i_val])
              xgb_p_tst += xgb_clf.predict_proba(tst_ar) / n_fold
              print(f'{i}fold accuarcy:',accuracy_score(target[i_val],np.argmax(xgb_p_val[i_val],axis=1)))
              print()
          print(xgb_clf)
          print(f'{accuracy_score(target, np.argmax(xgb_p_val, axis=1)) * 100:.4f}%')
          print(ftr.shape,tst_ar.shape)
```

```
92.0794%
(4280, 6) (1833, 6)
```

92.0794%의 accuracy를 보인다.

마찬가지로 클래스별 확률 예측값을 local환경에 저장한다.

▶ 3) LightGBM

grid search를 활용하여 parameter tuning을 진행하였다.

```
In [220]: lgb_p_val = np.zeros((ftr.shape[0], n_class))
          lgb_p_tst = np.zeros((tst_ar.shape[0], n_class))
          for i, (i_trn, i_val) in enumerate(cv.split(ftr, target), 1):
              print(f'training model for CV #{i}')
              lgb_clf = lgb.LGBMClassifier(
          #                            objective_type='xentropy',
                                       n_estimators=1000,
                                         boosting_type ='gbdt',
                                        num_leaves=100,
                                       learning_rate=0.1,
                                          min_data_in_leaf=30,
                                      feature_fraction_bynode = 0.8,
                                         feature_fraction=0.5,
                                 early_stopping_rounds =100,
                                          metric='multi_error',
                                       random_state=seed,
                                       n_jobs=-1)
              lgb_clf.fit(ftr[i_trn], target[i_trn],
                          eval_set=[(ftr[i_val], target[i_val])],
                             )

              lgb_p_val[i_val, :] = lgb_clf.predict_proba(ftr[i_val])
              lgb_p_tst += lgb_clf.predict_proba(tst_ar) / n_fold
              print(f'{i}fold accuarcy:',accuracy_score(target[i_val],np.argmax(lgb_p_val[i_val],axis=1)))

          print()
          print(lgb_clf)
          print(ftr.shape)
          print(f'{accuracy_score(target, np.argmax(lgb_p_val, axis=1)) * 100:.4f}%')
```

```
LGBMClassifier(early_stopping_rounds=100, feature_fraction=0.5,
               feature_fraction_bynode=0.8, metric='multi_error',
               min_data_in_leaf=30, n_estimators=1000, num_leaves=100,
               random_state=32152339)
(4280, 32)
92.4065%
```

단일 모델 중 가장 높은 accracy인 **92.4065**를 기록하였다. lightGBM의 경우 parameter[5]의 종류가 다른 알고리즘보다 상대적으로 많아 **grid search**에 많은 시간비용이 투입되었다. 특이한 점은 lightGBM 알고리즘 특성상 해당 데이터처럼 행의 수가 많지 않으면 과적합을 방지하기 어렵다고 알려져 있는데, 높은 성능이 나온 것이다. 더불어 위 lightGBM의 파라미터에서 max_depth를 지정해주지 않았는데, 이는 default인 -1로 각 나무를 생성한 것이다. 즉,

5) https://lightgbm.readthedocs.io/en/latest/Parameters.html

나무 깊이의 제한을 두지 않은 것이다.

▶ 4) Multi-Layer Perceptron

Multi-Layer Perceptron(이하 MLP)의 경우 위 알고리즘들과는 다르게 독립변수들 간의 종속성, 차원의 저주, 표준화에 특히 민감하다. 그리하여 아래와 같이 PCA와 scaling을 진행한 후 알고리즘을 학습한다.

```
In [232]: trn_mlp=trn.copy()
          tst_mlp=tst.copy()
          ftr_mlp=trn_mlp.values
          tst_ar_mlp=tst_mlp.values

In [233]: scaler = StandardScaler()
          scaler.fit(ftr_mlp)
          ftr_mlp = scaler.transform(ftr_mlp)
          tst_ar_mlp = scaler.transform(tst_ar_mlp)

In [234]: pca = PCA(n_components=12)
          pca_12 = pca.fit_transform(ftr_mlp)
          pca_12_tst= pca.fit_transform(tst_ar_mlp)

In [245]: sum(pca.explained_variance_ratio_)
Out[245]: 0.9857448654370654
```

위와 같이 scaling 및 PCA를 진행하였으며 12개의 요인으로 차원축소 하였다. 12개의 요인은 32개 변수의 분산 중 약 98.5%를 설명한다고 볼 수 있다. 이 12개의 요인으로 알고리즘 학습을 진행한다. **grid search**를 이용하여 parameter tuning 하였다.

```
In [235]: mlp_p_val = np.zeros((pca_12.shape[0], n_class))
          mlp_p_tst = np.zeros((pca_12_tst.shape[0], n_class))
          for i, (i_trn, i_val) in enumerate(cv.split(pca_12, target), 1):
              print(f'training model for CV #{i}')
              mlp_clf = MLPClassifier(
                  hidden_layer_sizes=(10),
                                  max_iter=10000,
                                  learning_rate_init=0.001,
                                  activation='relu',
                                  verbose=10,
                                  alpha= 0.0001,
                                  n_iter_no_change=20,
                                   solver= 'adam',
                                  random_state=seed
                                 )
              mlp_clf.fit(pca_12[i_trn], target[i_trn])
              mlp_p_val[i_val, :] = mlp_clf.predict_proba(pca_12[i_val])
              mlp_p_tst += mlp_clf.predict_proba(pca_12_tst) / n_fold
              print(f'{i}fold accuarcy:',accuracy_score(target[i_val],np.argmax(mlp_p_val[i_val],axis=1)))
              print()
          print(mlp_clf)
          print(f'{accuracy_score(target, np.argmax(mlp_p_val, axis=1)) * 100:.4f}%')
          print(pca_12.shape)

          MLPClassifier(hidden_layer_sizes=10, max_iter=10000, n_iter_no_change=20,
                        random_state=32152339, verbose=10)
          88.9486%
          (4280, 12)
```

88.9486%의 accuracy를 보인다. 비록 위 tree계열 모델들보다 성능은 낮지만 ensemble에서 다양한 모델이 투입되면 성능 향상에 유리하므로 이를 기대해본다. 마찬가지로 local 환경에 확률 예측값을 저장한다.

▶ 5) Support Vector Machine + Bagging

SVM 알고리즘 역시 scaling을 한 뒤 학습하였다. 성능이나 시간 비용 면에서 상대적으로 뒤쳐지는 모습을 보였으나 parameter 많지 않아 tuning에 있어 시간 소모가 비교적 덜하였다. 딥러닝/클라우드 과목에서 학습한 SVM의 hyper parameter 중 kernel은 'rbf'의 성능이 가장 뛰어났다.

해당 모델에서의 특징은 SVM의 variance를 최소화하기 위하여 [6]BaggingClassifier와 함께 사용하였다. variance를 줄임과 동시에 model의 성능을 약 0.8% 향상시키는 효과까지 얻을 수 있었다. **grid search**를 이용하여 parameter tuning 하였다.

```
In [251]: svm_p_val = np.zeros((ftr_mlp.shape[0], n_class))
          svm_p_tst = np.zeros((tst_ar_mlp.shape[0], n_class))
          for i, (i_trn, i_val) in enumerate(cv.split(ftr_mlp, target), 1):
              print(f'training model for CV #{i}')
              estimator = SVC(verbose=True,kernel='rbf',random_state=seed)
              svm_clf = BaggingClassifier(base_estimator=estimator,
                                          n_estimators=200,
                                                random_state=seed,
                                          n_jobs=-1)
              svm_clf.fit(ftr_mlp[i_trn], target[i_trn])

              svm_p_val[i_val, :] = svm_clf.predict_proba(ftr_mlp[i_val])
              svm_p_tst += svm_clf.predict_proba(tst_ar_mlp) / n_fold
              print(f'{i}fold accuarcy:',accuracy_score(target[i_val],np.argmax(svm_p_val[i_val],axis=1)))

          print(f'{accuracy_score(target, np.argmax(svm_p_val, axis=1)) * 100:.4f}%')
          print(svm_clf)

          training model for CV #1
          1fold accuarcy: 0.8936915887850467
          training model for CV #2
          2fold accuarcy: 0.8878504672897196
          training model for CV #3
          3fold accuarcy: 0.889018691588785
          training model for CV #4
          4fold accuarcy: 0.8925233644859814
          training model for CV #5
          5fold accuarcy: 0.8960280373831776
          89.1822%
          BaggingClassifier(base_estimator=SVC(random_state=32152339, verbose=True),
                            n_estimators=200, n_jobs=-1, random_state=32152339)
```

accuracy는 **89.1822%**이다. 마찬가지로 local환경에 확률 예측값을 저장한다.

▶ 6) K-Nearest Neighbor + Bagging

KNN도 마찬가지로 scaling을 진행한다. 더불어 BaggingClassifier와 함께 사용하여 0.9%의 성능을 향상시켰다. **grid search**를 이용하여 parameter tuning 하였다.

```
In [253]: from sklearn.neighbors import KNeighborsClassifier


          knn_p_val = np.zeros((ftr_mlp.shape[0], n_class))
          knn_p_tst = np.zeros((tst_ar_mlp.shape[0], n_class))
          for i, (i_trn, i_val) in enumerate(cv.split(ftr_mlp, target), 1):
              print(f'training model for CV #{i}')
              estimator = KNeighborsClassifier(n_neighbors=3)
              knn_clf = BaggingClassifier(base_estimator=estimator,
                                          n_estimators=500,
                                               verbose=False,
                                               random_state=seed,
                                          n_jobs=-1)
              knn_clf.fit(ftr_mlp[i_trn], target[i_trn])

              knn_p_val[i_val, :] = knn_clf.predict_proba(ftr_mlp[i_val])
              knn_p_tst += knn_clf.predict_proba(tst_ar_mlp) / n_fold
              print(f'{i}fold accuarcy:',accuracy_score(target[i_val],np.argmax(knn_p_val[i_val],axis=1)))

          print()
          print(knn_clf)
          print(ftr.shape)
          print(f'{accuracy_score(target, np.argmax(knn_p_val, axis=1)) * 100:.4f}%')
          # print(confusion_matrix(target, np.argmax(rn_p_val, axis=1)))

          BaggingClassifier(base_estimator=KNeighborsClassifier(n_neighbors=3),
                            n_estimators=500, n_jobs=-1, random_state=32152339,
                            verbose=False)
          (4280, 32)
          90.5841%
```

90.5841%의 정확도를 얻었으며, local 환경에 확률 예측값을 저장한다.

6) https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.BaggingClassifier.html

▶ 7) ExtraTrees

7)ExtraTreesclassifier는 Randomforest와 유사하나, 그 차이점은 사용하는 결정트리가 다르다는 점에 있다. Randomforest는 DecisionTree를 결정트리로 사용하지만 ExtraTrees는 ExtraTree를 결정트리로 사용한다. 성능 면에서도 randomforest와 유사하다. random forest와는 다르게 entropy를 불순도 지표로 사용하였다.

grid search로 parameter tuning 하였다.

```
In [254]: from sklearn.ensemble import ExtraTreesClassifier

ext_p_val = np.zeros((ftr.shape[0], n_class))
ext_p_tst = np.zeros((tst_ar.shape[0], n_class))
for i, (i_trn, i_val) in enumerate(cv.split(ftr, target), 1):
    print(f'training model for CV #{i}')
    ext_clf = ExtraTreesClassifier(n_estimators=1000,criterion='entropy', random_state=seed)
    ext_clf.fit(ftr[i_trn], target[i_trn])
    ext_p_val[i_val, :] = ext_clf.predict_proba(ftr[i_val])
    ext_p_tst += ext_clf.predict_proba(tst_ar) / 5
    y_pred = ext_clf.predict(ftr[i_val])
    y_pred = pd.Series(y_pred)
    print(accuracy_score(pd.Series(target[i_val]), y_pred))
    print()
print(f'{accuracy_score(target, np.argmax(ext_p_val, axis=1)) * 100:.4f}%')
# print(confusion_matrix(target, np.argmax(ext_p_val, axis=1)))

training model for CV #1
0.9158878504672897

training model for CV #2
0.897196261682243

training model for CV #3
0.9065420560747663

training model for CV #4
0.9170560747663551

training model for CV #5
0.9158878504672897

91.0514%
```

accuracy는 **91.0514%**이며 local환경에 확률 예측값을 저장하였다.

▶ 8) CatBoost

catboost[8]는 xgboost, lightGBM과 더불어 성능 좋은 boosting model로 평가받고 있으며, xgboost와 마찬가지로 level-wise로 트리를 만들어 나간다. (lightGBM은 Leaf-wise) 또한, xgboost, lightGBM과 더불어 아직 scikit-learn(ver.0.23.2 기준)에 함수가 없으므로 따로 module을 install해야 한다. **grid search**로 parameter tuning 하였다.

```
In [9]: from catboost import CatBoostClassifier, Pool

cb_p_val = np.zeros((ftr.shape[0], n_class))
cb_p_tst = np.zeros((tst_ar.shape[0], n_class))
for i, (i_trn, i_val) in enumerate(cv.split(ftr, target), 1):
    print(f'training model for CV #{i}')
    cb_clf = CatBoostClassifier(iterations=3000,
                                loss_function='MultiClass',
                                random_seed = seed,
                                task_type = "GPU" ,
                                eval_metric='Accuracy')
    cb_clf.fit(ftr[i_trn], target[i_trn],
            eval_set=[(ftr[i_val], target[i_val])])

    cb_p_val[i_val, :] = cb_clf.predict_proba(ftr[i_val])
    cb_p_tst += cb_clf.predict_proba(tst_ar) / n_fold
print(f'{accuracy_score(target, np.argmax(cb_p_val, axis=1)) * 100:.4f}%')
print(confusion_matrix(target, np.argmax(cb_p_val, axis=1)))
```

특이한 점은 해당 모델은 gpu를 사용하였는데, gpu/cpu 여부에 따라 나머지 parameter가

7)https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.ExtraTreesClassifier.html
8) https://catboost.ai/docs/concepts/python-reference_catboostclassifier.html

동일하여도 예측값이 다르니 이 점에 주의해야 한다.

```
91.8458%
[[283   3   3  72   2  61]
 [  2 993   9  13   0   0]
 [  3  11 429   1   3  22]
 [ 19   5   0 857   0  16]
 [  2   0   9   1 477   3]
 [ 47   0  20  21   1 892]]
```

accuracy 는 **91.8458%**로 lightGBM,xgboost에 이어 세 번째로 높은 성능을 자랑하였다. 마찬가지로 local 환경에 확률 예측값을 저장한다.

▶ 9) HistGBM

HistGBM[9]의 경우 비교적 최근에 공개된 알고리즘으로써, Gradient Boosting Classification Tree가 histogram 기반이다. 일반 GBM보다 큰 dataset에서 훨씬 빠르게 작동한다는 장점을 가지고 있다. (보통 n_samples >=10000) 본 데이터셋은 해당되지 않으나 다양한 알고리즘의 적용을 위하여 시도하였다. max_iter과 learning rate는 **grid search**를 통하여 탐색하였다.

```
In [260]:
from sklearn.experimental import enable_hist_gradient_boosting
from sklearn.ensemble import HistGradientBoostingClassifier

hgb_p_val = np.zeros((ftr.shape[0], n_class))
hgb_p_tst = np.zeros((tst_ar.shape[0], n_class))
# tree_model = DecisionTreeClassifier(max_depth = 10)

for i, (i_trn, i_val) in enumerate(cv.split(ftr, target), 1):
    print(f'training model for CV #{i}')
    hgb_clf = HistGradientBoostingClassifier(max_iter=2000,
                                     # validation_fraction=0.1,
                                     # n_iter_no_change=15,
                                     verbose=True,
                                     random_state=seed,
                                         learning_rate = 0.01
                                       )

    hgb_clf.fit(ftr[i_trn], target[i_trn])
    hgb_p_val[i_val, :] = hgb_clf.predict_proba(ftr[i_val])
    hgb_p_tst += hgb_clf.predict_proba(tst_ar) / 5
    print(f'{i}fold accuarcy:',accuracy_score(target[i_val],np.argmax(hgb_p_val[i_val],axis=1)))

    print()
print(f'{accuracy_score(target, np.argmax(hgb_p_val, axis=1)) * 100:.4f}%')
print(confusion_matrix(target, np.argmax(hgb_p_val, axis=1)))
```

```
In [263]:
print(f'{accuracy_score(target, np.argmax(hgb_p_val, axis=1)) * 100:.4f}%')
```

```
91.4953%
```

91.4953%의 정확도를 얻었으며, local환경에 확률 예측값을 저장한다.

▶ 10) Ensemble

ensemble은 모델 성능 향상과 안정성 강화에 효과적이다. 그렇기 때문에 경진대회에서 몹시 자주 사용되는 기법이다. 크게 bagging, boosting, stacking, voting 등이 있다. bagging과 boosting은 위 9가지 model에서 이미 유용하게 사용한 바 있다. 이제부터는 위 9가지 모델을 기반으로 voting과 stacking에 적용해볼 것이다.

----------Voting

voting은 크게 두가지 방식으로 나눌 수 있다. VotingClassifier의 parameter 중 'voting'에서 soft는 개별 분류기의 예측을 평균 내어 확률이 가장 높은 클래스를 예측할 수 있다. 이

9) https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.HistGradientBoostingClassifier.html

를 간접 투표라고 한다. 이 방식은 확률이 높은 투표에 비중을 더 두기 때문에 예측된 확률 값을 직접 투표하는 직접 투표 방식인 hard보다 일반적으로 더 높은 성능을 보인다. 그렇기에 아래에서 나타나듯, **간접 투표** 방식을 사용하였다.

In [324]:
```
from sklearn.ensemble import VotingClassifier

p_val = np.zeros((ftr.shape[0], n_class))
p_tst = np.zeros((tst_ar.shape[0], n_class))
for i, (i_trn, i_val) in enumerate(cv.split(ftr, target), 1):
    print(f'training model for CV #{i}')
    clf = VotingClassifier(estimators=[
         ('lgb',lgb_clf),
         ('rf',rf_clf),
        ('xgb',xgb_clf),
         ('mlp',mlp_clf),
        ('svm',svm_clf),
         ('ext',ext_clf),
        ('cb',cb_clf),
        ('hgb',hgb_clf),
        ('knn',knn_clf)
    ],voting='soft',verbose=True,n_jobs=-1)
    clf.fit(ftr[i_trn], target[i_trn])
    p_val[i_val, :] = clf.predict_proba(ftr[i_val])
    p_tst += clf.predict_proba(tst_ar) / n_fold
    print(f'{i}fold accuarcy:',accuracy_score(target[i_val],np.argmax(p_val[i_val],axis=1)))

print()
print('',ftr.shape)
print(clf)
print(f'valid cv accuracy : {accuracy_score(target, np.argmax(p_val, axis=1)) :.6f}')
```

valid cv accuracy : 0.915421

만족스러운 성능이 나오지 않아 Stacking을 해보았다. 보통 성능을 올리기 위해선 voting보다는 stacking의 성능이 더 좋다고 알려져 있다.

----------Stacking

stacking을 사용하기 위해선 위에서 local환경에 저장해 놓은 확률 예측값을 불러온다.

In [320]:
```
tst_dir = Path('submission/tst')
val_dir = Path('submission/val')
model_names = [
#         'cb_91.8224',
#         'xgb_91.9626',
#         'xgb_92.0561',
#         'xgb_32_92.0794',
        'xgb_91.7056',
#         'svm_89.1355',
         'svm_89.1822',
        'rf_91.0748',
         'ext_91.0514',
         'hgb_91.4953',
#         'mlp_89.3692',
         'mlp_89.1589',
#         'lgb_91.5888',
        'cb_91.8458',
        'lgb_92.4065',
#         'knn_90.9579',
        'knn_91.0748'
]
stk_trn = []
stk_tst = []
feature_names = []
for model in model_names:
    stk_trn.append(np.loadtxt(val_dir / f'{model}.val.csv', delimiter=','))
    stk_tst.append(np.loadtxt(tst_dir / f'{model}.tst.csv', delimiter=','))
    feature_names += [f'{model}_class0', f'{model}_class1', f'{model}_class2',
                      f'{model}_class3', f'{model}_class4', f'{model}_class5']
stk_trn = np.hstack(stk_trn)
stk_tst = np.hstack(stk_tst)
```

주석된 확률 예측값은 raw data로 예측했던 것이다. 이렇게 스태킹을 위한 데이터셋을 구성하고, 아래와 같이 lightGBM으로 스태킹을 진행한다.

In [321]:
```
p_val = np.zeros((stk_trn.shape[0], n_class))
p_tst = np.zeros((stk_tst.shape[0], n_class))
for i, (i_trn, i_val) in enumerate(cv.split(stk_trn, target), 1):
    print(f'training model for CV #{i}')
    clf = lgb.LGBMClassifier(objective='multiclass',
                             n_estimators=1000,
                             learning_rate=0.1,
                               max_depth=5,
                             metric='multi_error',
                             early_stopping_rounds=300,
                               feature_fraction=0.3,
                             feature_fraction_bynode = 0.7,
                             random_state=seed,
                             n_jobs=-1)
    clf.fit(stk_trn[i_trn], target[i_trn],
            eval_set=[(stk_trn[i_val], target[i_val])],
            eval_metric='multi_error',
            verbose=False)
    p_val[i_val, :] = clf.predict_proba(stk_trn[i_val])
    p_tst += clf.predict_proba(stk_tst) / n_fold
    print(f'{i}fold accuarcy:',accuracy_score(target[i_val],np.argmax(p_val[i_val],axis=1)))
print()
print('seed:',seed)
print('models:',model_names)
print(clf)
now = datetime.datetime.now()
nowDatetime = now.strftime('%Y-%m-%d_%H%M%S')
print(nowDatetime)
print(f'cv accuracy : {accuracy_score(target, np.argmax(p_val, axis=1)) :.6f}')
```

```
seed: 19960829
models: ['xgb_91.7056', 'svm_89.1822', 'rf_91.0748', 'ext_91.0514', 'hgb_91.4953', 'mlp_89.1589', 'cb_91.8458', 'lgb_92.4065', 'knn_91.074
8']
LGBMClassifier(early_stopping_rounds=300, feature_fraction=0.3,
               feature_fraction_bynode=0.7, max_depth=5, metric='multi_error',
               n_estimators=1000, objective='multiclass',
               random_state=19960829)
2020-10-24_150846
cv accuracy : 0.929439
```

모든 모델로 10fold stratified cv로 검증하니 0.929439의 정확도를 얻을 수 있었다. 이 모델 중 stacking 성능에 오히려 방해가 되는 모델이 있을 수도 있다고 판단하여 많은 시도 끝에 최종적으로 3개의 모델을 stacking 하였다.

```
seed: 19960829
models: ['xgb_91.7056', 'lgb_92.4065', 'knn_91.0748']
LGBMClassifier(early_stopping_rounds=300, feature_fraction=0.3,
               feature_fraction_bynode=0.7, max_depth=5, metric='multi_error',
               n_estimators=1000, objective='multiclass',
               random_state=19960829)
2020-10-24_151359
cv accuracy : 0.931542
```

최종적으로 위와 같이 xgboost, lightGBM, KNN을 선택하였고, 최종 CV accuracy는 **93.1542%**가 나왔다.

Ⅳ. 결언

▶ 1) 결과

- xgboost, lightGBM, KNN을 stacking
- cv accuracy : 93.1776%
- test accuracy : **93.66812227074 24%**
- 최종 순위 : 1위

▶ 2) 느낀 점

- XGBoost, LigthGBM, Catboost의 경우 powerful한 boosting 알고리즘답게 높은 성능을 자랑함을 느낌.
- ensemble을 이용하더라도 너무 많은 모델이 투입되면 과적합에 빠져 오히려 성능이 심하게 저하될 수 있음.
- bagging을 사용하면 과적합을 제어하는데 효과적이라는 것을 느낌. 몇몇 weak learner와 bagging을 함께 이용했을 때 과적합 방지와 성능향상에 효과적이었음.
- 모델별로 학습시키기 전에 선제되어야 할 과정을 반드시 알고 있어야 하는 것이 중요하다는 것을 느낌. 일례로 MLP의 경우 scaling 유무에 따라 5%까지 성능이 차이나는 것을 확인.
- 개별 base learner의 성능이 우수하면 반드시 stacking model 성능이 향상되는 것이 아니라, 모델이나 feature set이 얼마나 다양한가(독립적인가)가 중요한 부분이라는 것을 느낌.
- 경진대회라는 경쟁을 통해 많은 실력향상을 느꼈고, DACON, Kaggle 등 데이터 분석 경진대회에 활발히 참여하는 목표를 세울 예정.

▶ 3) 한계점

- dataset이 비교적 크지 않아 seed의 영향을 많이 받았음.
- python에 익숙하지 않아 대부분 hard coding으로 이루어져 실수가 잦고, 시간이 오래 걸렸음.
- EDA를 통해 얻은 변수가 성능 향상에 도움되지 않는 것이 많았음.
- 경험이 부족하여 EDA를 통하여 얻는 인사이트가 많지 않았음.
- 변수의 의미를 알 수 없어 feature engineering 시 domain을 통해 작업이 불가능하였음.

▶ 4) 참고

- 딥러닝/클라우드 수업 교재 및 code
- 핸즈온 머신러닝 : 사이킷런, 케라스, 텐서플로 2를 활용한 머신러닝, 딥러닝 완벽 실무
- 데이터 과학을 위한 통계
- 사이킷런 홈페이지 : https://scikit-learn.org/stable/index.html
- LightGBM 홈페이지 : https://lightgbm.readthedocs.io/en/latest/Parameters.html
- XGBoost 홈페이지 : https://xgboost.readthedocs.io/en/latest/
- CatBoost 홈페이지 : https://catboost.ai/docs/concepts/python-reference_catboostclassifier.html