

센서 데이터 기반 모션 분류 (Kaggle 개인별 경진 대회)

김영우

Overview

- AIVLE 미니 프로젝트 5차 과정
- 과제명 : 센서 데이터 기반 모션 분류
목표 : 뛰어난 분류 모델 생성

Description

- 제출 방법
 - 제출 제한은 20 번 입니다
- 1.train dataset을 활용하여 예측 모델 생성
 - 2.해당 모델로 제공한 test 데이터 결과 값 예측
 - 3.sample.csv.와 같은 형식으로 작성하여 kaggle에 제출

Evaluation

- AIVLE 미니 프로젝트 5차 과정
- 과제명 : HAR 데이터 멀티 라벨 분류
- 모델 성능 평가기준 : **Categorization Accuracy**
- 제출 양식 (csv 파일 예제)

```
ID, label
```

```
0, 1
```

```
1, 2
```

```
2, 1
```

```
...
```



Dataset Description - 라벨 설명

Y(Target)에 클래스가 11개 임을 알 수 있다. 즉 다중 분류 문제임을 알 수 있다.

- 1: 걷기
- 2: 뛰기
- 3: 천천히 걷기
- 4: 계단 오르기
- 5: 계단 내려가기
- 6: 서있기
- 7: 앉아있기
- 8: 누워있기
- 9: 자전거 타기
- 10: 서서 자전거 타기
- 11: 자전거에 앉아있기

Dataset Description - Files

- **train.csv** - the training set
- **test.csv** - the test set
- **sample_submission.csv** - a sample submission file in the correct format

Dataset Description - Columns

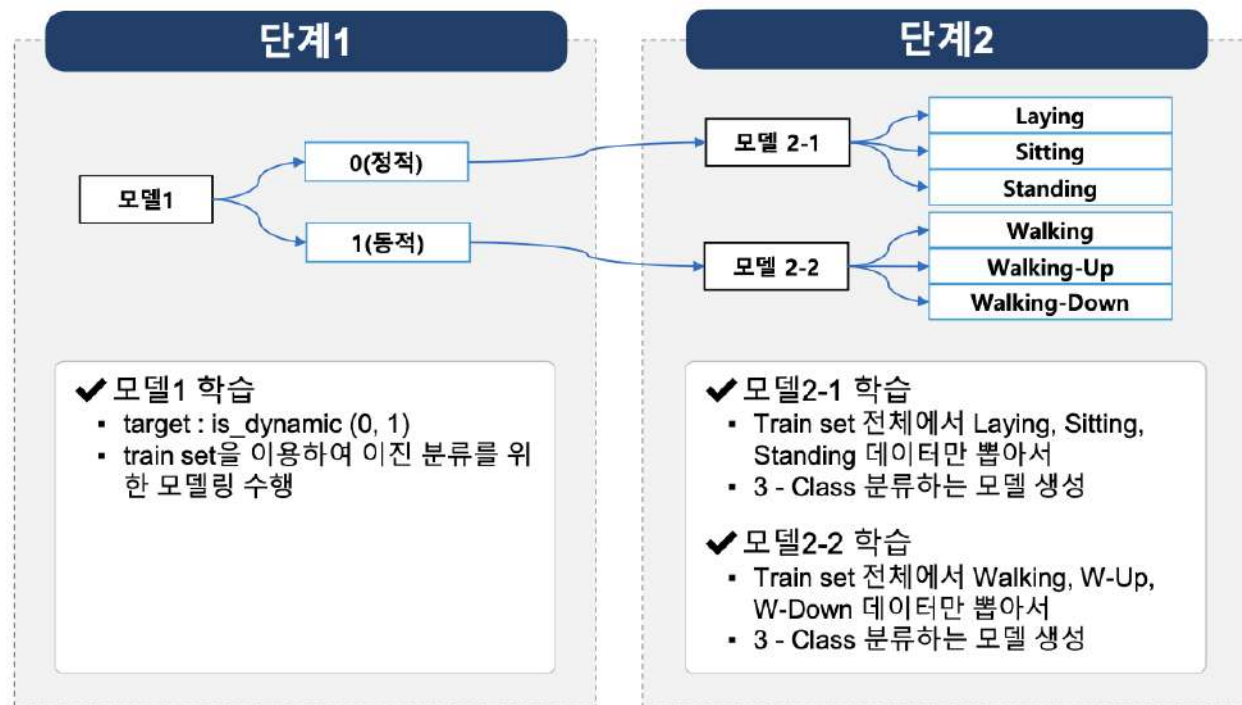
- timestamp - 측정 시간
- A_x - 신체부위 'A'에 부착된 센서의 x축 데이터
- A_y - 신체부위 'A'에 부착된 센서의 y축 데이터
- A_z - 신체부위 'A'에 부착된 센서의 z축 데이터
- B_x - 신체부위 'B'에 부착된 센서의 x축 데이터
- B_y - 신체부위 'B'에 부착된 센서의 y축 데이터
- B_z - 신체부위 'B'에 부착된 센서의 z축 데이터
- label - 라벨값(1~11) -> Y(Target)

케글 경진 대회 URL

- <https://www.kaggle.com/competitions/4-aivle-school-5th-miniproject/overview>

진행을 어떻게 했는지,
하면서 들었던 생각과 느낌에 대해서
말하고자 합니다.

진행 프로세스



라벨(1,2... 11)을 상식적인 기준에 따라 정적 행동(0)과 동적 행동(1)로 구분하는 모델 1를 학습하고

정적 행동(0) 중에서 어떤 정적 행동인지 구분하는 모델 2-1를 학습하고

동적 행동(1) 중에서 어떤 동적 행동인지 구분하는 모델 2-2를 학습하여

test-set를 모델 1에 넣은 다음 상황에 따라
모델 2-1 혹은 모델 2-2에 넣어
정확도를 확인하고자 한다.

라벨(label)에 따라서 클래스 불균형이 있는지 확인한다.

그 결과 클래스 불균형이 존재한다는 것을 알 수 있다.

```
✓  
0초 [6] 1 train_data['label'].value_counts()  
  
      2      12170  
      8      11497  
      1      11465  
      7      10594  
     10       9579  
      9       9282  
     11       9112  
      3       8305  
      4       7208  
      5       5411  
      6       5377  
      Name: label, dtype: int64
```

클래스 불균형 문제를 대처하기 위한 방안이 있다.

1번째는 리샘플링(Resampling)을 통한 오버샘플링(Oversampling), 언더샘플링(Undersampling)을 취하는 방식
2번째는 모델 학습시 소수 클래스에 더 높은 가중치를 부여하는 방식이 있다.



클래스 불균형 문제는 머신러닝 모델을 훈련할 때 종종 마주치는 문제 중 하나입니다. 클래스 불균형이 심할 경우, 모델이 소수 클래스를 제대로 학습하지 못하고, 다수 클래스를 너무 과도하게 학습할 가능성이 높아집니다. 이로 인해 모델의 성능이 저하될 수 있습니다.

다음은 클래스 불균형 문제를 해결하기 위해 고려할 수 있는 몇 가지 방법입니다:

1. 리샘플링 (Resampling)

- **오버샘플링 (Oversampling):** 소수 클래스의 샘플을 복제하거나, SMOTE (Synthetic Minority Over-sampling Technique)와 같은 방법을 사용하여 소수 클래스의 샘플을 인위적으로 늘립니다.
- **언더샘플링 (Undersampling):** 다수 클래스의 샘플을 제거하여 클래스 간의 균형을 맞춥니다. 하지만 이 방법은 정보 손실을 초래할 수 있습니다.

2. 가중치 조정


모델 학습 시 소수 클래스에 더 높은 가중치를 주어 학습을 균형있게 만들 수 있습니다. 많은 머신러닝 알고리즘에서는 클래스 가중치를 설정할 수 있는 옵션이 있습니다.

3. 앙상블 방법 사용

배깅(bagging)과 부스팅(boosting)과 같은 앙상블 방법을 사용하면 클래스 불균형 문제에 더 강한 모델을 만들 수 있습니다.

저는 2번째 방식, 모델 학습할 때 소수 클래스에 대한 가중치를 더 부여하는 방식을 채택했습니다. 왜냐하면 리샘플링(Resampling) 즉 오버샘플링(Oversampling)과 언더샘플링(Undersampling) 개념은 몇번 들어봤었고 강의도 들었기 때문이다.

2번째 방식은 데이터를 변화시키지 않고, 소수 클래스에 대한 가중치만 더 부여하기 때문에 1번째 방식보다 더 이점이 있을 것 같고, 그동안 시도해보지 않아서 해보고 싶은 마음도 있었다.



가중치 조정은 모델 학습 시 다양한 클래스의 샘플들에 대해 다른 중요도를 부여하는 방법입니다. 클래스가 불균형할 때, 소수 클래스의 샘플에 더 높은 가중치를 주어 모델이 이들을 더 중요하게 취급하도록 유도할 수 있습니다. 여러 머신러닝 알고리즘과 라이브러리에서 클래스 가중치를 설정하는 옵션이 있습니다.

Scikit-Learn을 사용한 가중치 조정

예를 들어, scikit-learn의 일부 분류 알고리즘에서는 `class_weight`라는 파라미터를 통해 클래스 가중치를 설정할 수 있습니다.

```
python
from sklearn.utils.class_weight import compute_class_weight
import numpy as np

# 클래스 레이블과 해당 레이블의 빈도수
labels = np.unique(train_data['label'])
class_weights = compute_class_weight(class_weight='balanced', classes=labels)

# 클래스 가중치 딕셔너리 생성
class_weight_dict = {label: weight for label, weight in zip(labels, class_weights)}

# 분류기 초기화 시 클래스 가중치 설정
classifier = SomeClassifier(class_weight=class_weight_dict)
classifier.fit(X_train, y_train)
```

위의 예시에서 `SomeClassifier`는 scikit-learn의 분류기 중 하나를 나타냅니다 (예:

Regenerate

위의 예시에서 `SomeClassifier`는 scikit-learn의 분류기 중 하나를 나타냅니다 (예: `LogisticRegression`, `SVM`, 등). `compute_class_weight` 함수는 각 클래스에 대한 가중치를 계산하여 반환합니다. 이렇게 계산된 가중치는 분류기를 초기화할 때 `class_weight` 파라미터로 전달됩니다.

TensorFlow/Keras를 사용한 가중치 조정

TensorFlow 또는 Keras를 사용하는 경우, 모델의 `fit` 메서드에 `class_weight` 파라미터를 전달하여 가중치를 조정할 수 있습니다.

```
python
model.fit(X_train, y_train, class_weight=class_weight_dict)
```

여기서 `class_weight_dict`는 클래스 레이블을 키로, 해당 클래스의 가중치를 값으로 하는 딕셔너리입니다.

각 머신러닝 라이브러리와 알고리즘마다 클래스 가중치를 설정하는 방법이 다를 수 있으므로, 사용하는 도구의 문서를 참조하여 올바른 방법을 찾는 것이 중요합니다.

시계열 데이터는 microsecond가 0.2초, 0.4초 정도 차이가 나는 것으로 파악된다.
나는 그점이 우려됐는데, 시간 간격이 갑자기 차이가 날 때 (ex. 20분, 1시간, 1일 ...)
어떻게 결측치를 처리할까 라는 고민이 있었다.

하지만 그런 고민은 하지 않고, 결측치를 비교적 쉽게 처리할 수 있겠다는 판단이 들었다.

	timestamp	A_x	A_y	A_z	B_x	B_y	B_z	label
0	2000-01-01 00:00:00.000	-0.988758	-0.144844	0.183325	-0.991851	-0.022659	-0.306466	6
1	2000-01-01 00:00:00.020	-1.008465	NaN	0.156743	-0.976224	-0.040058	-0.277258	6
2	2000-01-01 00:00:00.040	-0.991936	-0.143049	0.156618	-0.949758	-0.035796	-0.299548	6
3	2000-01-01 00:00:00.060	-1.007975	-0.140831	0.155302	-1.006806	-0.013057	-0.289020	6
4	2000-01-01 00:00:00.080	-0.991991	-0.159979	0.166965	NaN	0.015054	-0.293883	6

결측치를 어떻게 처리하는 것이 관건인데

결국 2가지 방법이 있다. 결측치가 있는 행을 제거(Remove) 하는 방식과 결측치를 어떻게 대체할 것인가?

저는 후자를 선택했습니다. 전자를 선택하면, 데이터가 소실되어서 그점이 우려웠습니다.

따라서 후자를 선택했고, 그 중에서 **이전 값과 이후 값을 더해서 평균 낸 값으로 결측치를 대체했습니다.**

```
1 train_data.info() # 변수에 대한 결측치를 확인한다.
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 100000 entries, 0 to 99999
Data columns (total 8 columns):
#   Column      Non-Null Count  Dtype  
---  -
0   timestamp   100000 non-null  datetime64[ns]
1   A_x         90000 non-null   float64
2   A_y         90000 non-null   float64
3   A_z         90000 non-null   float64
4   B_x         90000 non-null   float64
5   B_y         90000 non-null   float64
6   B_z         90000 non-null   float64
7   label       100000 non-null   int64   
dtypes: datetime64[ns](1), float64(6), int64(1)
memory usage: 6.1 MB
```


결측치를 전 값과 후 값의 평균으로 대체

```
train_data[['A_x', 'A_y', 'A_z', 'B_x',
'B_y', 'B_z']] = (train_data[['A_x', 'A_y',
'A_z', 'B_x',
'B_y', 'B_z']]).fillna(method='ffill') +
train_data[['A_x', 'A_y', 'A_z', 'B_x',
'B_y', 'B_z']].fillna(method='bfill')) / 2
```

‘Year’, ‘Month’, ‘Day’, ‘Hour’, ‘Minute’, ‘Second’, ‘Microsecond’, ‘Combined’ 변수를 추가한다.

‘Combined’ 변수는 ‘Year’ + ‘Month’ + ‘Day’ + ‘Hour’ + ‘Minute’ + ‘Second’ + ‘Microsecond’를 합친 값이며

시계열 데이터에서 시간대 정보를 추가하는 것이 모델 성능을 향상시킬 수 있다는 의견이 존재한다.



시계열 데이터를 다룰 때, 시간대 관련 정보를 추가하는 것이 모델의 성능을 향상시킬 수 있습니다. 이는 추가된 시간대 정보가 데이터의 시간적 패턴을 더 잘 이해하고 예측하는 데 도움을 줄 수 있기 때문입니다. 여기에 몇 가지 이유가 있습니다:

1. 계절성(Seasonality) 및 추세(Trend) 파악:

- **계절성:** 연도, 월, 일, 시간 등의 정보를 모델에 제공하면, 모델은 데이터 내의 계절적 패턴을 파악하여 예측 정확도를 높일 수 있습니다.
- **추세:** 시간이 지남에 따라 데이터가 어떻게 변하는지를 파악하는 데 도움이 됩니다.

2. 특별한 시간대의 이벤트 파악:

- 일부 이벤트는 특정 시간대에 더 자주 발생할 수 있습니다. 예를 들어, 상점의 매출은 주말이나 휴일에 더 높을 수 있습니다.

3. 순환 패턴(Cyclical Patterns) 파악:

- 시간 데이터를 순환적인 패턴으로 변환하여, 예를 들어, 하루의 시간이나 요일을 모델이 이해할 수 있도록 할 수 있습니다.

4. 예측 변수로서의 시간:

- 시간 자체가 중요한 예측 변수가 될 수 있으며, 시간에 따라 변화하는 트렌드나 패턴을 모델이 파악할 수 있게 합니다.

5. 더 나은 성능의 모델 생성:

시간 정보를 추가함으로써, 모델이 데이터의 구조를 더 잘 이해하고, 결과적으로 더 정확한 예측을

🔄 Regenerate

정적 행동(0)과 동적 행동(1)를 구분하는 변수 'is_dynamic'를 만든다.

여기서 라벨 6, 7, 8, 11은 정적 행동(0)으로 정의

여기서 라벨 1, 2, 3, 4, 5, 9, 10은 동적 행동(1)으로 정의

- 정적행동(0), 동적행동(1)으로 나눈다.
- label을 바탕으로 저는 이렇게 동적 행동(1)과 정적 행동(0)으로 구분하였습니다.
 - 1. 걷기 -> 동적 행동(1)로 정의
 - 2. 뛰기 -> 동적 행동(1)로 정의
 - 3. 천천히 걷기 -> 동적 행동(1)로 정의
 - 4. 계단 오르기 -> 동적 행동(1)로 정의
 - 5. 계단 내려가기 -> 동적 행동(1)로 정의
 - 6. 서있기 -> 정적 행동(0)으로 정의
 - 7. 앉아있기 -> 정적 행동(0)으로 정의
 - 8. 누워있기 -> 정적 행동(0)으로 정의
 - 9. 자전거 타기 -> 동적 행동(1)로 정의
 - 10. 서서 자전거 타기 -> 동적 행동(1)로 정의
 - 11. 자전거에 앉아 있기 -> 정적 행동(0)으로 정의

train_X, train_y1, train_y2를 구분한다.

train_y1은 정적 행동에 따른 세부 분류(6, 7, 8, 11), 동적 행동에 따른 세부 분류(1, 2, 3, 4, 5, 9, 10)을 담고 있다.

train_y2는 정적 행동(0)과 동적 행동(1)을 구분하는 역할을 담당한다.

여기가 모델 1, 모델 2-1, 모델 2-2를 생성하는 출발점이다.

- 우리 train_data에서 X, y1(label), y2(is_dynamic)를 구분해요

```
1 train_X = train_data[['A_x', 'A_y', 'A_z', 'B_x', 'B_y', 'B_z', 'Year', 'Month', 'Day', 'Hour', 'Minute', 'Second',  
2 train_y1 = train_data['label']  
3 train_y2 = train_data['is_dynamic']
```

정적 행동(0)과 동적 행동(1)를 구분하는 모델 1 만들기!

0과 1이 클래스 불균형을 이룬다고 판단할 수 있다.

따라서 모델 1 훈련할 때 클래스 0에 가중치를 더 부여하는 식으로 진행하고자 한다.

- 0인지 1인지를 구분하는 곳에서 클래스 균형인지 클래스 불균형인지 확인한다.

```
1 train_y2.value_counts()
```

```
1    63420  
0    36580  
Name: is_dynamic, dtype: int64
```

Compute_class_weight 방식을 이용하여 그 결과 0에 더 가중치를 부여하는 것으로 알 수 있다.

따라서 모델 1를 훈련할 때 가중치 정보를 추가해주면 된다.

```
[ ] 1 from sklearn.utils.class_weight import compute_class_weight

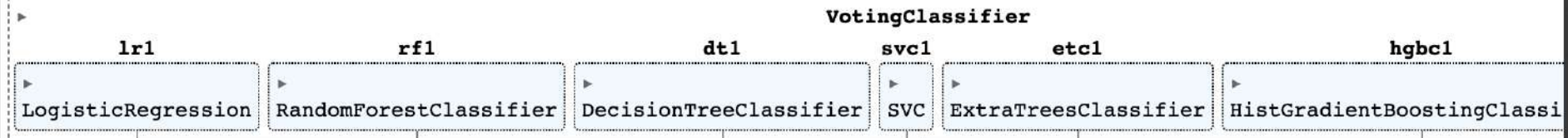
1 # 클래스 레이블과 해당 레이블의 빈도수
2 labels = np.unique(train_y2)
3 class_isdynamic_weights = compute_class_weight(class_weight='balanced', classes=labels, y=train_y2)
4
5 # 클래스 가중치 딕셔너리 생성
6 class_isdynamic_weight_dict = {label: weight for label, weight in zip(labels, class_isdynamic_weights)}
```

```
[ ] 1 class_isdynamic_weight_dict # 0를 가중치 더 부여하는 것을 알 수 있다.

{0: 1.366867140513942, 1: 0.7883948281299274}
```

여러 개의 모델을 만들어 합산하여, 결론을 내는 앙상블 모델을 만들었다.

```
1 # 다양한 분류기 생성
2 lr1 = LogisticRegression(class_weight = class_isdynamic_weight_dict, random_state=42)
3 rf1 = RandomForestClassifier(class_weight = class_isdynamic_weight_dict, random_state=42)
4 dt1 = DecisionTreeClassifier(class_weight = class_isdynamic_weight_dict, random_state=42)
5 svc1 = SVC(probability=True, class_weight = class_isdynamic_weight_dict, random_state=42)
6 etc1 = ExtraTreesClassifier(n_jobs=-1, class_weight = class_isdynamic_weight_dict, random_state=42)
7 hgbc1 = HistGradientBoostingClassifier(class_weight=class_isdynamic_weight_dict, random_state=42)
8 lgbm1 = lgbm.LGBMClassifier(class_weight=class_isdynamic_weight_dict, random_state=42)
9
10 # VotingClassifier 생성
11 modell = VotingClassifier(estimators=[('lr1', lr1), ('rf1', rf1), ('dt1', dt1), ('svc1', svc1), ('etc1', etc1), ('hgbc1', hgbc1)])
12 modell.fit(train_X, train_y2)
```



정적 행동에 따른 세부 분류(6, 7, 8, 11) 하는 모델 2-1 만들기!

정적 행동(6, 7, 8, 11)에 해당하는 static_train_X와 static_train_y를 생성했다.

```
1 # train_y1에서 정적 행동(6, 7, 8, 11)에 해당하는 행들의 인덱스를 찾기
2 indices = train_y1[train_y1.isin([6, 7, 8, 11])].index
3
4 # train_y1에서 정적 행동(6, 7, 8, 11)에 해당하는 것들만 모으기
5 static_train_y = train_y1.loc[indices]
6
7 # 해당하는 행들만 train_x에서 선택
8 static_train_x = train_x.loc[indices]
```


static_train_y에 대해서 클래스 불균형인지 확인한다.

결과를 확인해보니 클래스 불균형이여서 소수 클래스에 대해서 가중치를 더 부여하는 식으로 방향을 잡아야 한다.

- static_train_y에 대해서 클래스 균형인지 불균형인지 확인한다.

- 만약 불균형이라면 소수 클래스에 대해서 가중치를 더 부여하는 식으로 방향을 잡아야 한다.

```
1 static_train_y.value_counts() # 클래스 불균형이다.
```

```
8      11497
7      10594
11       9112
6        5377
Name: label, dtype: int64
```

정적 행동(6, 7, 8, 11)을 각각 0, 1, 2, 3으로 변환한다.

Scikit-learn의 많은 분류기들은 내부적으로 y 레이블을 0부터 시작하는 연속적인 정수로 자동으로 변환해주는데 VotingClassifier를 사용할 경우 이러한 변환을 해주지 않기 때문에 수동적으로 0부터 시작하는 연속적인 정수로 만들어줘야 한다.

그리고 소수 클래스에 대해서 가중치를 더 부여하도록 작업을 수행하고 있다.

- 정적 행동(6, 7, 8, 11)을 각각 0, 1, 2, 3으로 변환한다.

- Scikit-learn의 많은 분류기들은 내부적으로 y 레이블을 0부터 시작하는 연속적인 정수로 자동으로 변환해주는데 하지만 VotingClassifier는 이러한 변환을 해주지 않기 때문에 수동적으로 우리가 y 레이블을 0부터 시작하는 연속적인 정수로 만들어줘야 한다.

```
[ ] 1 from sklearn.utils.class_weight import compute_class_weight
```

```
1 클래스 레이블과 해당 레이블의 빈도수
2 abels = np.unique(static_train_y)
3 lass_staticClassification_weights = compute_class_weight(class_weight='balanced', classes=labels, y=static_train_y)
4
5 클래스 가중치 딕셔너리 생성
6 lass_staticClassification_weight_dict = {label: weight for label, weight in zip(labels, class_staticClassification_we
```

```
[ ] 1 class_staticClassification_weight_dict # 6를 가중치 더 부여하는 것을 알 수 있다.
```

```
{6: 1.7007625069741492,
7: 0.8632244666792525,
8: 0.7954248934504653,
11: 1.0036215978928884}
```

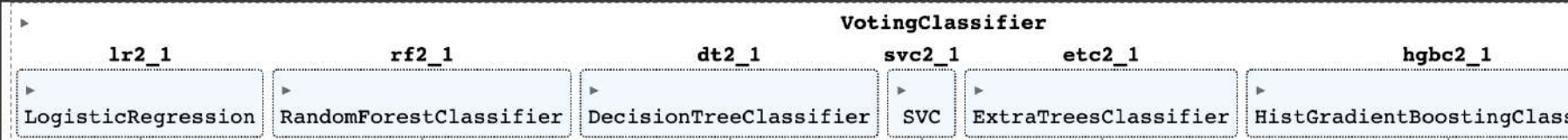
```
1 from sklearn.preprocessing import LabelEncoder
2
3 # le.classes_하면 6, 7, 8, 11이 print 된다.
4 le = LabelEncoder()
5 static_train_y_encoded = pd.Series(le.fit_transform(static_train_y)) # 1차원 넘파이 배열보다는 Series 형태로 사용하는 것을 선호해
6 class_staticClassification_weight_encoded_dict= {i: class_staticClassification_weight_dict[class_label] for i, class_
```

```
[ ] 1 class_staticClassification_weight_encoded_dict
```

```
{0: 1.7007625069741492,
1: 0.8632244666792525,
2: 0.7954248934504653,
3: 1.0036215978928884}
```

여러 개의 모델을 만들어 합산하여, 결론을 내는 앙상블 모델을 만들었다.

```
1 # 다양한 분류기 생성
2 lr2_1 = LogisticRegression(class_weight = class_staticClassification_weight_encoded_dict, random_state=42)
3 rf2_1 = RandomForestClassifier(class_weight = class_staticClassification_weight_encoded_dict, random_state=42)
4 dt2_1 = DecisionTreeClassifier(class_weight = class_staticClassification_weight_encoded_dict, random_state=42)
5 svc2_1 = SVC(probability=True, class_weight = class_staticClassification_weight_encoded_dict, random_state=42)
6 etc2_1 = ExtraTreesClassifier(n_jobs=-1, class_weight = class_staticClassification_weight_encoded_dict, random_state=42)
7 hgbc2_1 = HistGradientBoostingClassifier(class_weight = class_staticClassification_weight_encoded_dict, random_state=42)
8 lgbm2_1 = lgbm.LGBMClassifier(class_weight=class_staticClassification_weight_encoded_dict, random_state=42)
9
10 # VotingClassifier 생성
11 model2_1 = VotingClassifier(estimators=[('lr2_1', lr2_1), ('rf2_1', rf2_1), ('dt2_1', dt2_1), ('svc2_1', svc2_1), ('etc2_1', etc2_1), ('hgbc2_1', hgbc2_1)])
12 model2_1.fit(static_train_X, static_train_y_encoded)
```



동적 행동에 따른 세부 분류(1, 2, 3, 4, 5, 9, 10) 하는 모델 2-2 만들기!

동적 행동(1, 2, 3, 4, 5, 9, 10)에 해당하는 dynamic_train_X와 dynamic_train_y를 생성했다.

```
1 # train_y1에서 동적 행동(1, 2, 3, 4, 5, 9, 10)에 해당하는 행들의 인덱스를 찾기
2 indices = train_y1[train_y1.isin([1, 2, 3, 4, 5, 9, 10])].index
3
4 # train_y1에서 동적 행동(1, 2, 3, 4, 5, 9, 10)에 해당하는 것들만 모으기
5 dynamic_train_y = train_y1.loc[indices]
6
7 # 해당하는 행들만 train_x에서 선택
8 dynamic_train_x = train_x.loc[indices]
```

dynamic_train_y에 대해서 클래스 불균형인지 확인한다.

결과를 확인해보니 클래스 불균형이어서 소수 클래스에 대해서 가중치를 더 부여하는 식으로 방향을 잡아야 한다.

- dynamic_train_y에 대해서 클래스 균형인지 불균형인지 확인한다.
 - 만약 불균형이라면 소수 클래스에 대해서 가중치를 더 부여하는 식으로 방향을 잡아야 한다.

```
1 dynamic_train_y.value_counts() # 클래스 불균형이다.
```

```
2      12170  
1      11465  
10      9579  
9       9282  
3       8305  
4       7208  
5       5411  
Name: label, dtype: int64
```

동적 행동(1, 2, 3, 4, 5, 9, 10)을 각각 0, 1, 2, 3, 4, 5, 6으로 변환한다.

Scikit-learn의 많은 분류기들은 내부적으로 y 레이블을 0부터 시작하는 연속적인 정수로 자동으로 변환해주는데

VotingClassifier를 사용할 경우 이러한 변환을 해주지 않기 때문에 수동적으로 0부터 시작하는 연속적인 정수로 만들어줘야 한다.

그리고 소수 클래스에 대해서 가중치를 더 부여하도록 작업을 수행하고 있다.

• 동적 행동(1, 2, 3, 4, 5, 9, 10)을 각각 0, 1, 2, 3, 4, 5, 6으로 변환한다.

◦ Scikit-learn의 많은 분류기들은 내부적으로 y 레이블을 0부터 시작하는 연속적인 정수로 자동으로 변환해주는데

하지만 VotingClassifier는 이러한 변환을 해주지 않기 때문에 수동적으로 우리가 y 레이블을 0부터 시작하는 연속적인 정수로 만들어줘야 한다.

```
1 from sklearn.utils.class_weight import compute_class_weight
2
3 # 클래스 레이블과 해당 레이블의 빈도수
4 labels = np.unique(dynamic_train_y)
5 class_dynamicClassification_weights = compute_class_weight(class_weight='balanced', classes=labels, y=dynamic_train_y)
6
7 # 클래스 가중치 딕셔너리 생성
8 class_dynamicClassification_weight_dict = {label: weight for label, weight in zip(labels, class_dynamicClassification_weights)}
9
10 class_dynamicClassification_weight_dict # 5를 더 가중치 부여하는 것으로 알 수 있다.
{1: 0.7902311382468382,
 2: 0.7444535743631882,
 3: 1.0909090909090908,
 4: 1.2569367369589346,
 5: 1.6743670301238218,
 9: 0.9760827407886231,
10: 0.9458189790165988}
```


```
[ ] 1 from sklearn.preprocessing import LabelEncoder
2
3 # ld.classes_하면 1, 2, 3, 4, 5, 9, 10이 print 된다.
4 ld = LabelEncoder()
5 dynamic_train_y_encoded = pd.Series(ld.fit_transform(dynamic_train_y)) # 1차원 넘파이 배열보다는 Series 형태로 사용하는 것을 선택
6 class_dynamicClassification_weight_encoded_dict = {i: class_dynamicClassification_weight_dict[class_label] for i, class_label in enumerate(ld.classes_)}
```

```
[ ] 1 class_dynamicClassification_weight_encoded_dict
```

```
{0: 0.7902311382468382,
 1: 0.7444535743631882,
 2: 1.0909090909090908,
 3: 1.2569367369589346,
 4: 1.6743670301238218,
 5: 0.9760827407886231,
 6: 0.9458189790165988}
```


여러 개의 모델을 만들어 합산하여, 결론을 내는 앙상블 모델을 만들었다.

```
1 # 다양한 분류기 생성
2 lr2_2 = LogisticRegression(class_weight = class_dynamicClassification_weight_encoded_dict, random_state=42)
3 rf2_2 = RandomForestClassifier(class_weight = class_dynamicClassification_weight_encoded_dict, random_state=42)
4 dt2_2 = DecisionTreeClassifier(class_weight = class_dynamicClassification_weight_encoded_dict, random_state=42)
5 svc2_2 = SVC(probability=True, class_weight = class_dynamicClassification_weight_encoded_dict, random_state=42)
6 etc2_2 = ExtraTreesClassifier(n_jobs=-1, class_weight = class_dynamicClassification_weight_encoded_dict, random_state=42)
7 hgbc2_2 = HistGradientBoostingClassifier(class_weight=class_dynamicClassification_weight_encoded_dict, random_state=42)
8 lgbm2_2 = lgbm.LGBMClassifier(class_weight=class_dynamicClassification_weight_encoded_dict, random_state=42)
9
10 # VotingClassifier 생성
11 model2_2 = VotingClassifier(estimators=[('lr2_2', lr2_2), ('rf2_2', rf2_2), ('dt2_2', dt2_2), ('svc2_2', svc2_2), ('etc2_2', etc2_2), ('hgbc2_2', hgbc2_2)])
12 model2_2.fit(dynamic_train_X, dynamic_train_y_encoded)
```



테스트 데이터를 가지고 분류하기

학습 데이터처럼 전처리를 진행해서 진행했습니다.
이 과정은 생략하고 본문만 설명드리겠습니다.

테스트 데이터로 정적 행동(0)인지 동적 행동(1)인지 구분하는 모델 1에 넣어봅니다.

- 모델 1를 넣어봅시다.

```
[ ] 1  modell_predict = list(modell.predict(test_X))  
    2  
    3  print('0으로 예측한 개수 : ', modell_predict.count(0))  
    4  print('1로 예측한 개수 : ', modell_predict.count(1))
```

0으로 예측한 개수 : 4904

1로 예측한 개수 : 8330

정적 행동(0)인지 동적 행동(1)으로 어떻게 구분했는지에 따라
정적 행동 세부 분류하는 메서드, 동적 행동 세부 분류하는 메서드를 수행합니다.

```
1  # model1_predict 된 것을 하나씩 for문으로 돌린다.
2  for idx, mlp in enumerate(model1_predict):
3      if mlp == 0:
4          result = static_predict(idx)
5          if result == 0:
6              model2_predict.append(6)
7          elif result == 1:
8              model2_predict.append(7)
9          elif result == 2:
10             model2_predict.append(8)
11         else:
12             model2_predict.append(11)
13     else:
14         result = dynamic_predict(idx)
15         if result == 0:
16             model2_predict.append(1)
17         elif result == 1:
18             model2_predict.append(2)
19         elif result == 2:
20             model2_predict.append(3)
21         elif result == 3:
22             model2_predict.append(4)
23         elif result == 4:
24             model2_predict.append(5)
25         elif result == 5:
26             model2_predict.append(9)
27         else:
28             model2_predict.append(10)
```

정적 행동 세부 분류하는 메서드, 동적 행동 세부 분류하는 메서드는 아래와 같이 구성됩니다.

- model1_predict를 가지고 0이냐 1이냐에 따라서 모델 2-1, 모델 2-2를 통해 최종 예측을 시도한다.

```
1 # 모델 2-1를 통해 최종 예측 하는 함수
2 def static_predict(idx):
3     result = model2_1.predict(test_X.iloc[idx].values.reshape(1, -1)) # Series를 2차원 넘파이 배열로 변환한다. 왜냐하면 머신러닝
4     return result[0]
```

```
[ ] 1 # 모델 2-2를 통해 최종 예측 하는 함수
2 def dynamic_predict(idx):
3     result = model2_2.predict(test_X.iloc[idx].values.reshape(1, -1)) # Series를 2차원 넘파이 배열로 변환한다. 왜냐하면 머신러닝
4     return result[0]
```

```
[ ] 1 # 최종 예측된 것을 담는 리스트
2 model2_predict = []
```

Kaggle 제출 형태로 만든다.

Kaggle 제출 형태로 만든다

```
1 submit = pd.DataFrame({  
2     'id' : list(range(0, 13234, 1)),  
3     'label' : model2_predict  
4 })
```

```
[ ] 1 submit['label'].value_counts()
```

```
2      1612  
8      1539  
1      1477  
7      1406  
10     1359  
11     1269  
9      1192  
3      1028  
4       961  
5       701  
6       690
```

```
Name: label, dtype: int64
```

```
[ ] 1 # csv 파일로 저장  
2 submit.to_csv('/content/drive/MyDrive/미니프로젝트5차part2_Kaggle/predict.csv', index=False)
```

Kaggle 경진 대회에서 199명이 참가한 것으로 알고 있는데

Public Score가 0.98942 였으면 30등 ~ 40등 한 것으로 기억하고 있다.

Submission and Description

Public Score ⓘ

Selected



predict.csv

Complete (after deadline) · 6h ago

0.98942



느낀점

1. '클래스 불균형' 개념 인식을 확실히 깨우친 프로젝트였다.

예전에 프로젝트를 진행할 때 클래스 불균형이라는 개념에 대해서 확실히 깨우치지도 못했고 이를 해소하고자 하는 노력이 부족했는데 이번 프로젝트 계기로 분류 모델일 때 클래스 불균형인지 아닌지 확인하여, 만약 불균형이면 이를 해소하기 위한 방법을 수행해야 되겠다고 생각이 들었다.

2. 이번 프로젝트는 단계적 모델링을 수행했다. 모델 1을 거쳐 모델 2-1을 수행하기도 하고 모델 2-2를 수행하기도 했다.

본래 그냥 11개 라벨에 대해서 다중 분류를 해도 되지만, 단계적 모델링이라는 개념을 도입해서 프로젝트를 수행했다. 이렇게 프로젝트가 진행될 수 있구나를 느끼게 되었다.

