02. 사이킷런으로 시작하는 머신러닝

정수빈(이)가 만듦
어제 16:29에에 마지막 업데이트 • 조회한 사용자 1명

05. 데이터 전처리

데이터 전처리(Data preprocessing)는 ML 알고리즘만큼 중요합니다. ML알고리즘은 데이터에 기반하고 있기 때문에 어던 데이터를 입력으로 가지느냐에 따라 결과도 크게 달라질 수 있습니다. (Garbage in, Garbage out)

결손값, Nan, Null 값은 허용되지 않습니다. 따라서 결손값들은 삭제하거나, 다른 값으로 변환되어야 합니다. 결손값들을 어떻게 처리해야 할지는 경우에 따라 다릅니다. 몇 개 되지 않는다면 피쳐의 평균값으로 대체할 수있고, 일정이상 결손값이 나오면 드롭합니다. 하지만 구체적인 기준은 없으며 데이터를 분석하면서 이러한 변형들이 결과값에 심하게 영향이 끼치지 않도록 해야합니다.

사이킷런의 머신러닝 알고리즘은 문자열을 입력값으로 허용하고 있지 않습니다. 그래서 모든 문자열은 인코딩돼서 숫자 형으로 변환해야 합니다. 또한, 불필요한 피처라고 판단되면 삭제하는 것이 좋습니다. 주민번호, 단순문자열 아이디 같은 경우 예측에 중요하지 않은 데이터라면 삭제하는게 예측 성능을 높입니다.

데이터 인코딩

머신러닝의 대표적인 인코딩 방식으로는 레이블 인코딩과 원 핫 인코딩이 있습니다.

레이블 인코딩

머신러닝의 대표적인 인코딩 방식은 레이블 인코딩입니다. 레이블 인코딩은 카테코리를 코드형 숫자로 변환하는 것입니다. 예를들어, (TV: 0), (냉장고: 1), (전자렌지: 2)과 같은 숫자형으로 변환하는 것입니다.

```
from sklearn.preprocessing import LabelEncoder

items = ['TV', '냉장고', '전자레인지', '컴퓨터', '선풍기', '선풍기', '믹서', '믹서']

encoder = LabelEncoder()
encoder.fit(items)
labels = encoder.transform(items)
print(labels) // [0 1 4 5 3 3 2 2]

print(encoder.classes_) // ['TV' '냉장고' '믹서' '선풍기' '전자레인지' '컴퓨터']
```

`encoder.fit(items)` 은 `items` 을 요소 하나와 코드를 하나씩 매칭시키는 것이고 `encoder.transform(items)` 은 `item` 을 레이블인코딩된 리스트를 리턴합니다.

```
print(encoder.inverse_transform([5, 4, 3, 1]))
// ['컴퓨터' '전자레인지' '선풍기' '냉장고']
```

`encoder.inverse_transform([5, 4, 3, 1])` 은 `encoder.transform` 의 반대로 숫자를 값으로 만들어주는 메서드입니다.

이렇게 레이블 인코딩은 간단하게 문자열을 숫자형 카테고리로 바꿉니다. 하지만 레이블 인코딩에는 치명적인(?) 단점이 있습니다. `TV` 가 0으로, `냉장고` 가 1로 변환이 되면서 0은 1보다 크기 때문에 특정알고리즘에서는 이러한 부분이 가중치로 들어가 이런 의미 없는 순서를 중요하게 인식할 수 있습니다. 이러한 단점 때문에 레이블 인코딩은 선형회귀와 같은 ML알고리즘에는 사용되지 않고 트리 계열 ML알고리즘에 주로 사용되고 있습니다.

원-핫 인코딩 (OneHot Encoding)

원-핫 인코딩은 피처 값의 유형에 따라 새로운 피처를 추가해 고유의 값에 해당하는 컬럼에만 1을 표시하고 나머지는 0으로 채우는 방식입니다.

상품 분류
TV
냉장고
전자렌지
컴퓨터
...

분류 - TV	분류 - 냉장고	분류 - 전자렌지	분류 - 컴퓨터	...
1	0	0	0	0
0	1	0	0	0
0	0	1	0	0
0	0	0	1	0
0	0	0	0	1

```python
import numpy as np
from sklearn.preprocessing import OneHotEncoder

items = ['TV', '냉장고', '전자레인지', '컴퓨터', '선풍기', '선풍기', '믹서', '믹서']

items = np.array(items).reshape(-1, 1)
print(items)

# [['TV']
#  ['냉장고']
#  ['전자레인지']
#  ['컴퓨터']
#  ['선풍기']
#  ['선풍기']
#  ['믹서']
#  ['믹서']]

oh_encoder = OneHotEncoder()
oh_encoder.fit(items)
labels = oh_encoder.transform(items)
print(labels.toarray())

# [[1. 0. 0. 0. 0. 0.]
#  [0. 1. 0. 0. 0. 0.]
# [0. 0. 0. 0. 1. 0.]
# [0. 0. 0. 0. 0. 1.]
# [0. 0. 0. 1. 0. 0.]
# [0. 0. 0. 1. 0. 0.]
# [0. 0. 1. 0. 0. 0.]
# [0. 0. 1. 0. 0. 0.]]
```

판다스를 활용하면 더 쉽게 보실 수 있습니다 .

```python
import pandas as pd

df = pd.DataFrame({'item': ['TV', '냉장고', '전자레인지', '컴퓨터', '선풍기', '선풍기', '믹서', '
print(pd.get_dummies(df, dtype=int))
```

```
   item_TV  item_냉장고  item_믹서  item_선풍기  item_전자레인지  item_컴퓨터
0        1         0        0         0           0         0
1        0         1        0         0           0         0
2        0         0        0         0           1         0
3        0         0        0         0           0         1
4        0         0        0         1           0         0
5        0         0        0         1           0         0
6        0         0        1         0           0         0
7        0         0        1         0           0         0
```

피쳐 스케일링과 정규화

서로 다른 변수의 값 범위를 일정한 수준으로 맞추는 작업을 피처 스케일링이라고 합니다. 대표적인 방법으로는 표준화와 정규화가 있습니다. 표준화는 데이터 피처 각각이 평균이 0이고 표준편차를 1가진 값으로 변환하는 작업을 의미합니다. 정규화

는 데이터값을 특정범위 내로 조정하는 작업을 의미합니다. 주로 최대-최소 정규화를 많이 사용합니다.

표준화와 정규화를 하면 이상치(outliers)처리, 변수 간 차이 조정, 모델 성능 향상 등의 효과를 얻을 수 있습니다.

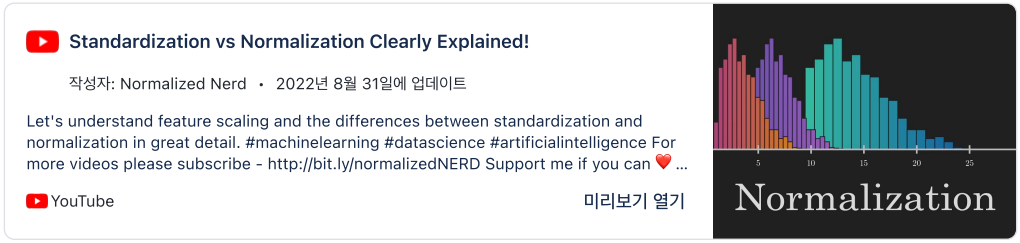
$$z = \frac{x - u}{s}$$

표준화 수식

$$x_{scaled} = \frac{x - x_{min}}{x_{max} - x_{min}}$$

최대-최소 정규화 수식

추천 영상 (한글자막o)



StandardScaler

`StandardScaler` 은 앞에서 설명한 표준화를 쉽게 지원하기 위한 클래스입니다. 개별 피처를 평균이 0이고 분산이 1인 값으로 변환합니다. 아래의 ML알고리즘은 데이터가 가우시안 분포를 가지고 있다고 가정하고 구현이 되어있습니다.

- SVM (Support Vector Machine)
- 선형 회귀 (Linear Regression)
- 로지스틱 회귀 (Logistic Regression)

`sklearn.datasets` 에서 제공하는 iris데이터를 활용하여 `StandardScaler` 이 어떻게 데이터를 변환하는지 확인해보겠습니다. 아래는 `StandardScaler` 를 적용하지 않았을 때의 코드입니다.

```
1  from sklearn.datasets import load_iris
2  import pandas as pd
3
4  iris = load_iris()
5  iris_data = iris.data
6  iris_df = pd.DataFrame(data=iris_data, columns=iris.feature_names)
7
8  print(iris_df.head(3))
```

	sepal length (cm)	sepal width (cm)	petal length (cm)	petal width (cm)
0	5.1	3.5	1.4	0.2
1	4.9	3.0	1.4	0.2
2	4.7	3.2	1.3	0.2

iris_data의 구조

```
1  print("feature 들의 평균 값")
2  print(iris_df.mean())
3  print()
4  print("feature 들의 분산 값")
5  print(iris_df.var())
```

```
feature 들의 평균 값
sepal length (cm)    5.843333
sepal width (cm)     3.057333
petal length (cm)    3.758000
petal width (cm)     1.199333
dtype: float64
```

feature들의 평균 값

```
feature 들의 분산 값
sepal length (cm)    0.685694
sepal width (cm)     0.189979
petal length (cm)    3.116278
petal width (cm)     0.581006
```

feature들의 분산 값

StandardScaler 적용된 전체 코드

```
from sklearn.datasets import load_iris
import pandas as pd
from sklearn.preprocessing import StandardScaler

iris = load_iris()
iris_data = iris.data


scaler = StandardScaler()
scaler.fit(iris_data)
iris_scaled = scaler.transform(iris_data)
iris_df_scaled = pd.DataFrame(data=iris_scaled, columns=iris.feature_names)

print("feature 들의 평균 값")
print(iris_df_scaled.mean())
print()
print("feature 들의 분산 값")
print(iris_df_scaled.var())
```

```
scaler = StandardScaler()
scaler.fit(iris_data)
iris_scaled = scaler.transform(iris_data)
iris_df_scaled = pd.DataFrame(data=iris_scaled, columns=iris.feature_names)
```

```
feature 들의 평균 값
sepal length (cm)   -1.690315e-15
sepal width (cm)    -1.842970e-15
petal length (cm)   -1.698641e-15
petal width (cm)    -1.409243e-15
```

표준화된 feature들의 평균 값

```
feature 들의 분산 값
sepal length (cm)    1.006711
sepal width (cm)     1.006711
petal length (cm)    1.006711
petal width (cm)     1.006711
```

표준화된 feature들의 분산 값

모든 칼럼 평균이 0에 아주 가까운 값으로, 분산은 1에 아주 가까운 값으로 변환된 것을 알 수 있습니다.

MinMaxScaler

`MinMaxScaler` 은 데이터값을 0과 1사이, 음수가 있다면 -1과 1사이의 값으로 변환합니다. 데이터 분포가 가우시안 분포가 아닐 경우 Min, Max Scale을 적용해볼 수 있습니다.

```
scaler = MinMaxScaler()
scaler.fit(iris_data)
iris_scaled = scaler.transform(iris_data)

iris_df_scaled = pd.DataFrame(data=iris_scaled, columns=iris.feature_names)
print("feature들의 최솟값")
print(iris_df_scaled.min())
print()
print("feature들의 최댓값")
print(iris_df_scaled.max())
```

```
feature들의 최솟값
sepal length (cm)    0.0
sepal width (cm)     0.0
petal length (cm)    0.0
petal width (cm)     0.0
```

최대-최소가 적용된 feature들의 최솟값

```
feature들의 최댓값
sepal length (cm)    1.0
sepal width (cm)     1.0
petal length (cm)    1.0
petal width (cm)     1.0
```

최대-최소가 적용된 feature들의 최댓값

```
gth (cm)  sepal width (cm)  petal length (cm)  petal w
```

```
.222222          0.625000          0.067797
.166667          0.416667          0.067797
.111111          0.500000          0.050847
```

```
iris_df_scaled.head(3)
```

학습 데이터와 테스트 데이터의 스케일링 변환 시 유의점

`StandardScaler`, `MinMaxScaler` 와 같은 Scaler 객체를 이용해 데이터의 스케일링 변환 시 `fit()`, `transform()`, `fit_transform()` 메서드를 사용합니다. `fit()` 은 데이터 변환을 위한 기준 정보 설정, `transform()` 은 이렇게 설정된 정보를 이용하여 데이터를 변환합니다. `fit_transform()` 은 `fit()`, `transform()` 을 한 번에 수행합니다.

만약에 학습 데이터와 테스트 데이터를 미리 분리한 다음에 각 데이터 셋에 `fit()`, `transform()` 적용하게 되면 주의해야할 점이 있습니다. 학습할 때, `MinMaxScaler` 를 적용한다고 했을 때 학습데이터에는 최대, 최소가 10, 0이라고 합시다. 그럼 최대는 10 → 1이 되어야 하니까 1/10이 각 피쳐마다 곱해지게 됩니다. 하지만 분리한 테스트 데이터셋의 최대, 최소는 5, 0이라고 해봅시다. 그렇다면 5 → 1이 되어야 하니까 각 피처들은 1/5가 곱해지게 됩니다. 따라서 테스트셋이 다르게 스케일이 된 것 입니다.

위와 같은 오류를 피하려면,

1. 가능하다면 전체 데이터의 스케일링 변환을 적용한 다음에 학습, 테스트 데이터로 분리
2. 위의 방법이 가능하지 않다면 학습데이터의 `fit()` 을 이용하여 `transform()` 할 것

06. 사이킷런으로 수행하는 타이타닉 생존자 예측

참고링크: 사이킷런으로 수행하는 타이타닉 생존자 예측

사이킷런으로 타이타닉 생존자를 예측하는 실습입니다. 머신러닝에 앞서 데이터 분석이 선행되어야 합니다.

```
titanic_df = pd.read_csv('titanic_train.csv')
print (titanic_df.head(3))

print('\n ### 학습 데이터 정보 ### \n')
print(titanic_df.info())
```

```
   PassengerId  Survived  Pclass                                               Name
0            1         0       3                            Braund, Mr. Owen Harris
1            2         1       1  Cumings, Mrs. John Bradley (Florence Briggs Th...
2            3         1       3                             Heikkinen, Miss. Laina

RangeIndex: 891 entries, 0 to 890
Data columns (total 12 columns):
 #   Column       Non-Null Count  Dtype
---  ------       --------------  -----
 0   PassengerId  891 non-null    int64
 1   Survived     891 non-null    int64
 2   Pclass       891 non-null    int64  # 티켓의 선실 등급
 3   Name         891 non-null    object
 4   Sex          891 non-null    object
 5   Age          714 non-null    float64
 6   SibSp        891 non-null    int64  # 같이 탑승한 형제 자매 및 배우자 인원수
 7   Parch        891 non-null    int64  # 같이 탑승한 부모님 또는 어린이 인원수
 8   Ticket       891 non-null    object
 9   Fare         891 non-null    float64 # 요금
 10  Cabin        204 non-null    object # 선실 번호
 11  Embarked     889 non-null    object # 중간 정착 항구
```

총 데이터 개수는 891개이고, 컬럼은 12개임을 알 수 있습니다. 타입은 `int64`, `float64`, `object` (문자열로 봐도 무방) 입니다. `Age`, `Cabin`, `Embarked` 은 Null값이 존재하는 것을 알 수 있습니다.

우선 Null값을 없애도록 하겠습니다. `Age` 의 Null은 전체 `Age` 의 평균값, `Cabin`, `Embarked` 의 Null은 `N` 으로 바꾸겠습니다.

```
titanic_df['Age'].fillna(titanic_df['Age'].mean(), inplace=True)
titanic_df['Cabin'].fillna('N', inplace=True)
titanic_df['Embarked'].fillna('N', inplace=True)
print("데이터 세트 Null 값 개수", titanic_df.isnull().sum().sum())

# 데이터 세트 Null 값 개수 0
```

› Deprecated 경고

현재 남아있는 문자열 피처는 `Sex`, `Cabin`, `Embarked` 입니다. 먼저 이 피처들의 값을 분류해 살펴보겠습니다.

```
print("Sex 값 분포 \n", titanic_df['Sex'].value_counts())
print("\nCabin 값 분포 \n", titanic_df['Cabin'].value_counts())
print("\nEmbarked 값 분포 \n", titanic_df['Embarked'].value_counts())
```

아래가 결과입니다.

```
Sex 값 분포
 Sex
male      577
female    314
Name: count, dtype: int64

Cabin 값 분포
 Cabin
N              687
C23 C25 C27      4
G6               4
B96 B98          4
C22 C26          3
               ...
E34              1
C7               1
C54              1
E36              1
C148             1
Name: count, Length: 148, dtype: int64

Embarked 값 분포
 Embarked
S    644
C    168
Q     77
N      2
Name: count, dtype: int64
```

`Cabin` (선실)인 경우 등급별로 나눠져있다는 것을 추측할 수 있습니다. 비싼 등급의 선실일수록 살아남았을 확률이 있으니 뒤의 객실번호는 빼고 앞글자만 추출하겠습니다.

```
titanic_df['Cabin'] = titanic_df['Cabin'].str[:1]
print(titanic_df['Cabin'].head(3))
```

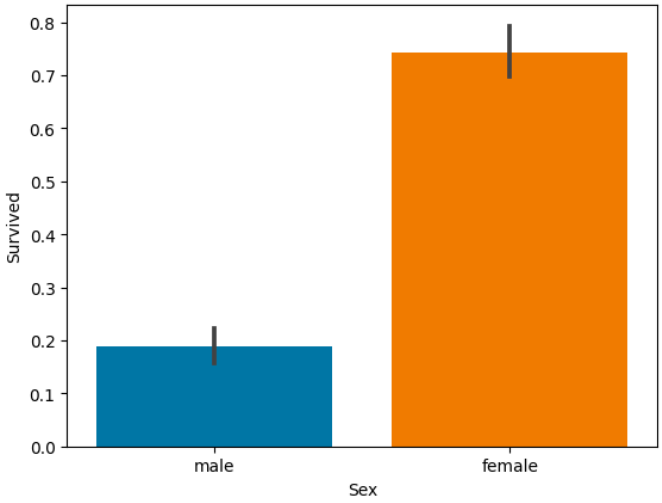
```
Name: count, dtype: int64
0    N
1    C
2    N
Name: Cabin, dtype: object

Cabin 값 분포
 Cabin
N    687
C     59
B     47
D     33
E     32
```

```
14  A     15
15  F     13
16  G      4
17  T      1
18  Name: count, dtype: int64
```

성별에 따라서도 생존률이 달라질 수 있습니다.

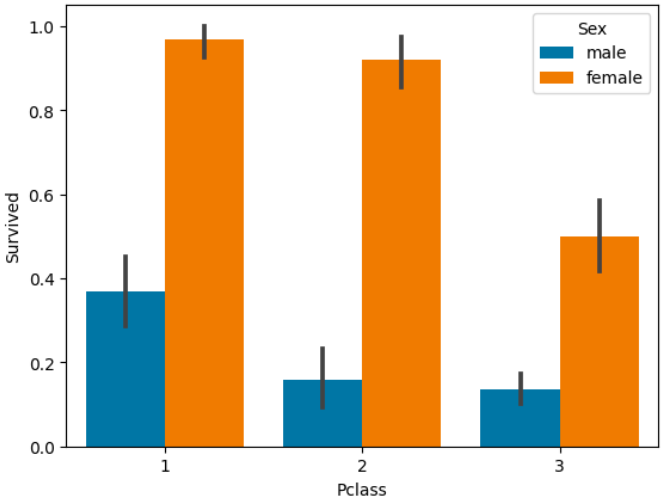
› groupby 실행한 결과

```
1  print(titanic_df.groupby(["Sex", "Survived"])['Survived'].count())
2
3  Sex     Survived
4  female  0            81
5          1           233
6  male    0           468
7          1           109
8  --
9  sns.barplot(x="Sex", y="Survived", data=titanic_df)
```



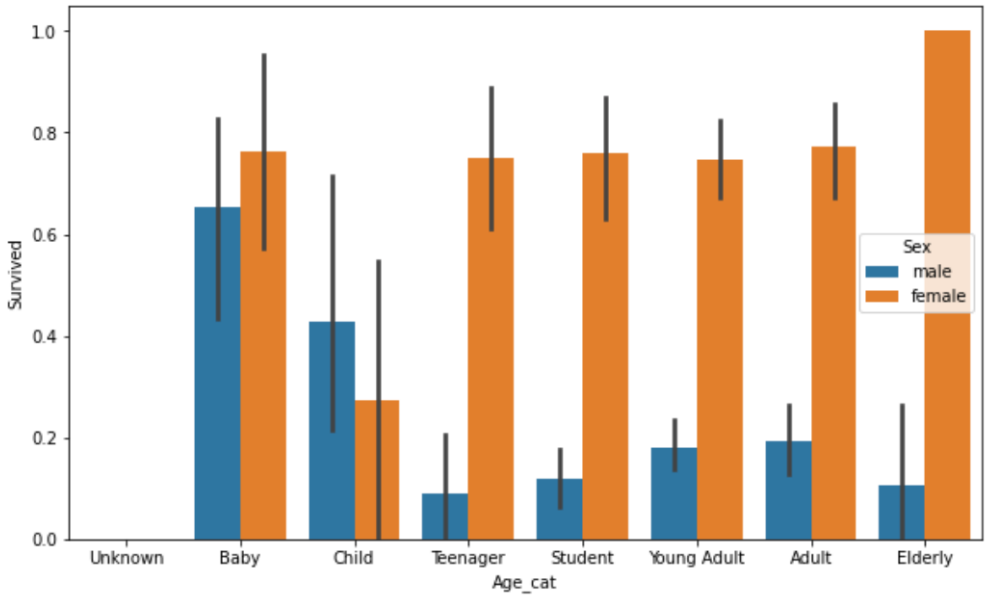
위의 그래프를 통해 여성이 생존확률이 좀 더 높다는 것을 알 수 있습니다. 아래는 선실 등급을 기준으로 생존율을 보여주는 그래프입니다.

```
1  sns.barplot(x="Pclass", y="Survived", hue="Sex", data=titanic_df)
```



여성의 경우 일, 이등실에 따른 생존 확률의 차이는 크지 않으나, 삼등실의 경우 생존 확률이 상대적으로 많이 떨어짐을 알 수 있습니다. 남성의 경우 일등실의 경우가 2,3등실보다 생존율이 높다는 것을 알 수 있습니다. 이번에는 Age에 따른 생존률을 알아보겠습니다. Age인 경우 값이 많기 때문에 나이별로 카테고리로 분류해서 그래프로 나타내겠습니다.

```
def get_category(age):
    cat = ''
    if age <= -1: cat = 'Unknown'
    elif age <= 5: cat = "Baby"
    elif age <= 12: cat = 'Child'
    elif age <= 18: cat = "Teenager"
    elif age <= 25: cat = "Student"
    elif age <= 35: cat = "Young Adult"
    elif age <= 60: cat = "Adult"
    else: cat = 'Elderly'

    return cat

plt.figure(figsize=(10, 6))

group_names = ['Unknown', 'Baby', 'Child', 'Teenager', 'Student', 'Young Adult', 'Adu

titanic_df['Age_cat'] = titanic_df['Age'].apply(lambda x : get_category(x))
sns.barplot('Age_cat', 'Survived', hue='Sex', data=titanic_df, order=group_names)
titanic_df.drop('Age_cat', axis=1, inplace=True)
```



여자 Baby의 경우 생존 확률이 높고 여자 Child의 경우 생존률이 낮은 것으로 나타났다. 그리고 여자 Elderly의 생존률이 매우 높은 것을 알 수 있다. 이제까지 분석한 결과 Sex, Age, Pclass 등이 중요하게 생존을 좌우하는 피처임을 확인할 수 있었다.

레이블 인코딩

남아있는 문자열 카테고리 피처를 숫자형 카테고리 피처로 변환하겠습니다. 사이킷런의 LabelEncoder을 이용하여 레이블 인코딩을 하겠습니다.

```
from sklearn import preprocessing

def encode_features(dataDF):
    features = ['Cabin', 'Sex', 'Embarked']
    for feature in features:
        le = preprocessing.LabelEncoder()
        le = le.fit(dataDF[feature])
        dataDF[feature] = le.transform(dataDF[feature])

    return dataDF

titanic_df = encode_features(titanic_df)
titanic_df.head()
```

`Cabin`, `Sex`, `Embarked` 이 숫자로 바뀌었음을 알 수 있습니다.

	PassengerId	Survived	Pclass	Name	Sex	Age	SibSp	Parch	Ticket	Fare	Cabin	Embarked
0	1	0	3	Braund, Mr. Owen Harris	1	22.0	1	0	A/5 21171	7.2500	7	3
1	2	1	1	Cumings, Mrs. John Bradley (Florence Briggs Th...	0	38.0	1	0	PC 17599	71.2833	2	0
2	3	1	3	Heikkinen, Miss. Laina	0	26.0	0	0	STON/O2. 3101282	7.9250	7	3
3	4	1	1	Futrelle, Mrs. Jacques Heath (Lily May Peel)	0	35.0	1	0	113803	53.1000	2	3
4	5	0	3	Allen, Mr. William Henry	1	35.0	0	0	373450	8.0500	7	3

전처리 함수

위에 나온 함수들을 아래와 같이 정리할 수 있습니다.

```
# Null 처리 함수
def fill_na(df):
    df['Age'].fillna(df['Age'].mean(), inplace=True)
    df['Cabin'].fillna('N', inplace=True)
    df['Embarked'].fillna('S', inplace=True)
    df['Fare'].fillna(0, inplace=True)

    return df

# 불필요한 칼럼 제거
def drop_features(df):
    df.drop(['PassengerId', 'Name', 'Ticket'], axis=1, inplace=True)

    return df

# 레이블 인코딩 수행
def format_features(df):
    df['Cabin'] = df['Cabin'].str[:1]
    features = ['Cabin', 'Sex', 'Embarked']
    for feature in features:
        le = preprocessing.LabelEncoder()
        le = le.fit(df[feature])
        df[feature] = le.transform(df[feature])

    return df

def transform_features(df):
    df = fill_na(df)
    df = drop_features(df)
    df = format_features(df)

    return df
```

데이터 셋 분리

우선 `train_test_split` 을 이용하여 학습 데이터와 테스트 데이터를 분리합니다. 이 데이터들 기반으로 머신러닝 모델을 학습(fit)하고 예측(predict)할 것입니다. (`random_state` 이건 없어도 되는 값입니다.)

```
from sklearn.model_selection import train_test_split

X_train, X_test, y_train, y_test = train_test_split(X_titanic_df, y_titanic_df, test_s
```

`transform_features` 는 최종 데이터 전처리함수입니다. 이제 이 함수를 이용하여 다시 원본데이터를 가공해야합니다. ML알고리즘인 결정 트리, 랜덤 포레스트, 로지스틱 회귀를 이용해 타이타닉 생존자를 예측해봅시다. 알고리즘에 대해서는 뒤에서 더 자세하게 설명합니다.

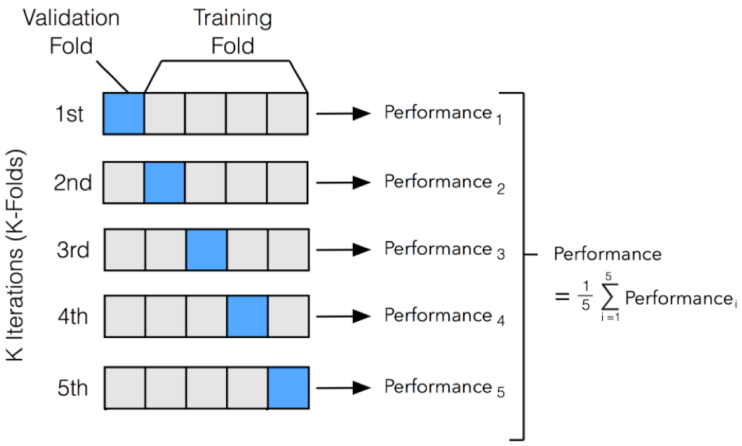
학습 및 평가

```
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import RandomForestClassifier
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score

dt_clf = DecisionTreeClassifier(random_state=11)
rf_clf = RandomForestClassifier(random_state=11)
lr_clf = LogisticRegression()

# DecisionTreeClassifier 학습/예측/평가
dt_clf.fit(X_train, y_train)
dt_pred = dt_clf.predict(X_test)
print('DecisionTreeClassifier 정확도: {0:.4f}'.format(accuracy_score(y_test, dt_pred)))

# RandomForestClassifier 학습/예측/평가
rf_clf.fit(X_train, y_train)
rf_pred = rf_clf.predict(X_test)
print('RandomForestClassifier 정확도: {0:.4f}'.format(accuracy_score(y_test, rf_pred)))

# LogisticRegression 학습/예측/평가
lr_clf.fit(X_train, y_train)
lr_pred = lr_clf.predict(X_test)
print('LogisticRegression 정확도: {0:.4f}'.format(accuracy_score(y_test, lr_pred)))
```

```
DecisionTreeClassifier 정확도: 0.7877
RandomForestClassifier 정확도: 0.8547
LogisticRegression 정확도: 0.8492
```

책에서는 `LogisticRegression` 이 0.86으로 정확도가 비교적 높게 나왔지만, 제가 테스트할 때는 `RandomForestClassifier` 이 로지스틱 회귀보다 정확도가 근소하게 높게 나왔습니다. 하지만 아직 최적화 작업을 수행하지 않았고, 데이터양도 충분하지 않기 때문에 어떤 알고리즘이 가장 좋다고 판단할 수 없습니다. 다음으로는 교차 검증으로 모델들을 좀 더 평가해보겠습니다.

KFold 평가



kFold 교차 검증

```
from sklearn.model_selection import KFold

def exec_kfold(clf, folds=5):
    # 폴드 세트를 5개인 KFold 객체를 생성, 폴드 수만큼 예측결과 저장을 위한 리스트 객체 생성
    kfold = KFold(n_splits=folds)
    scores = []

    # KFold 교차 검증 수행
    for iter_count, (train_index, test_index) in enumerate(kfold.split(X_titanic_df)):
        # X_titanic_df 데이터에서 교차 검증별로 학습과 검증 데이터를 가리키는 index 생성
```

```
        X_train, X_test = X_titanic_df.values[train_index], X_titanic_df.values[test_
        y_train, y_test = y_titanic_df.values[train_index], y_titanic_df.values[test_
        # Classifier 학습, 예측, 정확도 계산
        clf.fit(X_train, y_train)
        predictions = clf.predict(X_test)
        accuracy = accuracy_score(y_test, predictions)
        scores.append(accuracy)
        print('교차 검증 {0} 정확도: {1:.4f}'.format(iter_count, accuracy))

    # 5개 fold에서의 평균 정확도 계산
    mean_score = np.mean(scores)
    print('평균 정확도: {0:.4f}'.format(mean_score))
```

`DecisionTreeClassifier` 를 `kfold` 로 교차검증 해보겠습니다.

```
exec_kfold(dt_clf, folds=5)
```

결과값은 아래와 같습니다.

```
교차 검증 0 정확도: 0.7542
교차 검증 1 정확도: 0.7809
교차 검증 2 정확도: 0.7865
교차 검증 3 정확도: 0.7697
교차 검증 4 정확도: 0.8202
평균 정확도: 0.7823
```

책에서는 `DecisionTreeClassifier` 만 있지만 다른 분류 알고리즘도 평가를 해보겠습니다.
`RandomForestClassifier` 를 `kfold` 로 교차검증 해보겠습니다.

```
exec_kfold(rf_clf, folds=5)
```

결과값은 아래와 같습니다.

```
교차 검증 0 정확도: 0.7933
교차 검증 1 정확도: 0.8090
교차 검증 2 정확도: 0.8427
교차 검증 3 정확도: 0.7697
교차 검증 4 정확도: 0.8596
평균 정확도: 0.8148
```

`LogisticRegression` 를 `kfold` 로 교차검증 해보겠습니다.

```
exec_kfold(lr_clf, folds=5)
```

결과값은 아래와 같습니다.

```
교차 검증 0 정확도: 0.8045
교차 검증 1 정확도: 0.7809
교차 검증 2 정확도: 0.7753
교차 검증 3 정확도: 0.7528
교차 검증 4 정확도: 0.8202
평균 정확도: 0.7867
```

`kfold` 으로 검증한 결과 `RandomForestClassifier` 가 가장 정확도가 높다는 것을 알 수 있습니다.

cross_val_score 평가

`kfold` 말고 `cross_val_score` 로 `RandomForestClassifier` 를 평가해보겠습니다.

```
from sklearn.model_selection import cross_val_score

scores = cross_val_score(rf_clf, X_titanic_df, y_titanic_df, cv=5)
for iter_count, accuracy in enumerate(scores):
    print('교차 검증 {0} 정확도: {1:.4}'.format(iter_count, accuracy))

print('평균 정확도: {:.4f}'.format(np.mean(scores)))
```

```
교차 검증 0 정확도: 0.7933
교차 검증 1 정확도: 0.7921
교차 검증 2 정확도: 0.8539
교차 검증 3 정확도: 0.764
교차 검증 4 정확도: 0.8708
평균 정확도: 0.8148
```

RandomForestClassifier 평가에서는 kfold , cross_val_score 로 평가했을 때 정확도가 같게 나왔지만, 다르게 나오는 경우도 있습니다. 다르게 나온 것은 cross_val_score 내부 구현 중 StratifiedKFold 를 이용해 폴드 세트를 분할하기 때문입니다.

GridSearchCV 평가

마지막으로 GridSearchCV 를 이용해 DecisionTreeClassifier 를 평가해보겠습니다. CV는 5개의 폴드 세트를 지정하고 하이퍼 파라미터 max_depth , min_samples_split , min_samples_leaf 를 변경하면서 성능을 측정합니다. 최적의 하이퍼 파라미터의 예측을 출력하고, 최적 하이퍼 파라미터로 학습된 estimator를 이용해 데이터셋 예측을 수행하고 그 정확도를 출력합니다.

```
from sklearn.model_selection import GridSearchCV

parameters = {'max_depth': [2, 3, 5, 10],
              'min_samples_split': [2, 3, 5],
              'min_samples_leaf': [1, 5, 8]
              }

grid_dclf = GridSearchCV(dt_clf, param_grid=parameters, scoring='accuracy', cv=5)
grid_dclf.fit(X_train, y_train)

print(f'GridSearchCV 최적 하이퍼 파라미터: {grid_dclf.best_params_}')
print(f'GridSearchCV 최고 정확도: {round(grid_dclf.best_score_, 4)}')
best_dclf = grid_dclf.best_estimator_

# GridSearchCV의 최적 하이퍼 파라미터로 학습된 Estimator로 예측 및 평가 수행
dpredictions = best_dclf.predict(X_test)
accuracy = accuracy_score(y_test, dpredictions)
print(f'테스트 세트에서의 DecisionTreeClassifier 정확도: {round(accuracy, 4)}')
```

```
GridSearchCV 최적 하이퍼 파라미터: {'max_depth': 3, 'min_samples_leaf': 5, 'min_samples_spl
GridSearchCV 최고 정확도: 0.7992
테스트 세트에서의 DecisionTreeClassifier 정확도: 0.8715
```

RandomForestClassifier 를 GridSearchCV 로 평가했을 때는 아래와 같은 결과가 나왔습니다.

```
GridSearchCV 최적 하이퍼 파라미터: {'max_depth': 10, 'min_samples_leaf': 1, 'min_samples_sp
GridSearchCV 최고 정확도: 0.8132
테스트 세트에서의 RandomForestClassifier 정확도: 0.8603
```

하이퍼 파라미터 변경으로 DecisionTreeClassifier 처럼 정확도가 많이 올라갈 수도 있지만 (8%향상) 일반적으로 이 정도 수준으로 증가시키기는 매우 어렵습니다. 테스트용 데이터 세트가 작기 때문에 수치상으로 예측 성능이 많이 증가한 것처럼 보입니다.

+ 레이블 추가

첫 번째로 반응을 추가해 보세요