범주형 데이터 처리

- 명목형(nominal) - 순서가 없는 범주 데이터

예: 파랑, 빨강, 초록
남자, 여자
바나나, 딸기, 사과

- 순서형(ordinal) - 자연적인 순서를 가지는 데이터

예 : 낮음, 중간, 높음
청년, 노인
동의, 중립, 반대

- 벡터나 문자열로 표현되는 범주형 데이터는 머신러닝 알고리즘에서 수치값으로 입력해야 합니다.

범주형 데이터 처리

➢ 순서 없는 범주형 특성 인코딩

- 사이킷런의 LabelBinarizer를 사용하여 문자열 타깃 데이터를 원-핫 인코딩합니다.
- classes_ 속성에서 클래스를 확인 할 수 있습니다.

```
import numpy as np
from sklearn.preprocessing import LabelBinarizer, MultiLabelBinarizer

feature = np.array([["Texas"],
                    ["California"],
                    ["Texas"],
                    ["Delaware"],
                    ["Texas"]])          # 특성 데이터 생성

one_hot = LabelBinarizer()          # 원-핫 인코더 생성
one_hot.fit_transform(feature)      # 특성을 원-핫 인코딩 변환
one_hot.classes_                    #특성의 클래스를 확인
one_hot.inverse_transform(one_hot.transform(feature))   #원-핫 인코딩을 되돌립니다.
```

- 판단스를 이용하여 특성을 원-핫 인코딩할 수 있습니다.

```
import pandas as pd
pd.get_dummies(feature[:, 0] )  #특성으로 더미 변수 생성
```

범주형 데이터 처리

➢ 순서 없는 범주형 특성 인코딩

- 원-핫 인코딩 또는 더미 인코딩(dummy encoding)는 각 클래스마다 하나의 특성을 만들어냅니다.
- 클래스에 해당하는 특성은 1이 되고 나머지 특성은 0이 됩니다.
- 샘플 데이터가 여러 개의 클래스를 가지고 있는 경우에도 원-핫 인코더를 만들 수 있습니다
- 문자열 타깃 데이터를 정수 레이블로 변환할 때는 LabelEncoder를 사용합니다

```
multiclass_feature = [("Texas", "Florida"),
                      ("California", "Alabama"),
                      ("Texas", "Florida"),
                      ("Delware", "Florida"),
                      ("Texas", "Alabama")]  # 다중 클래스 특성 생성

one_hot_multiclass = MultiLabelBinarizer()              # 다중 클래스 원-핫 인코더 객체 생성
one_hot_multiclass.fit_transform(multiclass_feature)    # 다중 클래스 특성을 원-핫 인코딩 수행
one_hot_multiclass.classes_                             #특성의 클래스를 확인
```

범주형 데이터 처리

➢ 순서 없는 범주형 특성 인코딩

- 사이킷런의 OneHotEncoder 클래스는 정수형 특성을 원-핫 인코딩으로 변환
- OneHotEncoder 클래스는 v0.20부터는 문자열 데이터를 인식
- OneHotEncoder 클래스는 희소 배열을 반환이 기본값이며 sparse=False로 지정하면 밀집 배열을 반환합니다.
- categories_속성으로 클래스를 확인할 수 있습니다.
- OneHotEncoder는 입력 특성 배열을 모두 범주형으로 인식하여 변환합니다.
- 특정 열에만 적용하려면 ColumnTransformer와 함께 사용합니다.

```
from sklearn.preprocessing import OneHotEncoder

feature = np.array([["Texas", 1],
                    ["California", 1],
                    ["Texas", 3],
                    ["Delaware", 1],
                    ["Texas", 1]])   # 여러 개의 열이 있는 특성 배열 생성

one_hot_encoder = OneHotEncoder(sparse=False)
one_hot_encoder.fit_transform(feature)
one_hot_encoder.categories_
```

범주형 데이터 처리

➢ 순서 있는 범주형 특성 인코딩

- 순서가 있는 클래스는 순서 개념을 가진 수치값으로 변환해야 합니다.
- 클래스 레이블 문자열을 정수로 매핑하는 딕셔너리를 만들고 이를 필요한 특성에 적용합니다

```
#판다스 데이터프레임의 replace()를 사용하여 문자열 레이블을 수치값으로 변환합니다.
import pandas as pd

dataframe = pd.DataFrame({"Score": ["Low", "Low", "Medium", "Medium", "High"]})  # 특성 데이터 생성
scale_mapper = {"Low":1,
                "Medium":2,
                "High":3}                          # 매핑 딕셔너리 생성
dataframe["Score"].replace(scale_mapper)    # 특성을 정수로 변환

dataframe = pd.DataFrame({"Score": ["Low", "Low", "Medium", "Medium", "High", "Barely More Than Medium"]})
scale_mapper = {"Low":1,
                "Medium":2,
                "Barely More Than Medium": 3,
                "High":4}                                  # 매핑 딕셔너리 생성
dataframe["Score"].replace(scale_mapper)                   # 특성을 정수로 변환
```

범주형 데이터 처리

➢ 순서 있는 범주형 특성 인코딩

- 사이킷런 v0.20에서 범주형 데이터를 정수로 인코딩하는 OrdinalEncoder가 추가되었습니다.
- OrdinalEncoder는 클래스 범주를 순서대로 변환합니다.
- 특정 열만 범주형으로 변환하려면 ColumnTransformer와 함께 사용합니다.

```
import pandas as pd

features = np.array(
                    [ [ "Low", 10],
                      ["High", 50 ],
                      ["Medium", 3] ] )

ordinal_encoder = OrdinalEncoder()
ordinal_encoder.fit_transform(features)
ordinal_encoder.categories_
```

범주형 데이터 처리

➢ 특성 딕셔너리를 인코딩

- 딕셔너리를 특성 행렬로 변환
- DictVectorizer클래스는 0이 아닌 값의 원소만 저장하는 희소 행렬을 반환
- 자연어 처리 분야와 같은 매우 큰 행렬을 다룰때 메모리 사용량을 최소화해야하기 때문에 유용합니다.
- Sparse=False로 지정하면 밀집 벡터를 출력할 수 있습니다
- get_feature_names()를 사용하여 생성된 특성의 이름을 얻을 수 있습니다.

```
from sklearn.feature_extraction import DictVectorizer
data_dict = [{"Red": 2, "Blue": 4},
             {"Red": 4, "Blue": 3},
             {"Red": 1, "Yellow": 2},
             {"Red": 2, "Yellow": 2}]      # 딕셔너리 생성

dictvectorizer = DictVectorizer(sparse=False)   # DictVectorizer 객체 생성
features = dictvectorizer.fit_transform(data_dict)  # 딕셔너리를 특성 행렬로 변환
features   # 특성 행렬을 확인

feature_names = dictvectorizer.get_feature_names()   # 특성 이름을 얻습니다.
feature_names  # 특성 이름을 확인

import pandas as pd
pd.DataFrame(features, columns=feature_names)
```

	Blue	Red	Yellow
0	4.0	2.0	0.0
1	3.0	4.0	0.0
2	0.0	1.0	2.0
3	0.0	2.0	2.0

범주형 데이터 처리

➢ 특성 딕셔너리를 인코딩

- 머신러닝 알고리즘은 행렬 형태의 데이터를 기대합니다.
- 예] 문서 데이터를 가지고 있을 때 각 문서에 등장한 모든 단어의 횟수를 담은 딕셔너리를 사이킷런의 DictVectorizer를 사용하여 행렬 형태의 데이터를 만들 수 있습니다.

```
from sklearn.feature_extraction import DictVectorizer

# 네 개의 문서에 대한 단어 카운트 딕셔너리를 만듭니다.
doc_1_word_count = {"Red": 2, "Blue": 4}
doc_2_word_count = {"Red": 4, "Blue": 3}
doc_3_word_count = {"Red": 1, "Yellow": 2}
doc_4_word_count = {"Red": 2, "Yellow": 2}

doc_word_counts = [doc_1_word_count,
                   doc_2_word_count,
                   doc_3_word_count,
                   doc_4_word_count]          # 리스트로 생성

dictvectorizer.fit_transform(doc_word_counts) # 단어 카운트 딕셔너리를 특성 행렬로 변환합니다.
```

범주형 데이터 처리

➢ 누락된 클래스 값 대체하기 1

- 머신러닝 분류 알고리즘을 훈련하여 누락된 값을 예측하는 것입니다 (k-최근접 이웃KNN) 분류기를 사용, 이상적인 방법)

```
import numpy as np
from sklearn.neighbors import KNeighborsClassifier

X = np.array([[0, 2.10, 1.45],
              [1, 1.18, 1.33],
              [0, 1.22, 1.27],
              [1, -0.21, -1.19]])  # 범주형 특성을 가진 특성 행렬 생성

# 범주형 특성에 누락된 값이 있는 특성 행렬을 만듭니다.
X_with_nan = np.array([[np.nan, 0.87, 1.31],    [np.nan, -0.67, -0.22]])
clf = KNeighborsClassifier(3, weights='distance')  # KNN  객체 생성
trained_model = clf.fit(X[:,1:], X[:,0])   #훈련

imputed_values = trained_model.predict(X_with_nan[:,1:])   # 누락된 값의 클래스를 예측
# 예측된 클래스와 원본 특성을 열로 합칩니다.
X_with_imputed = np.hstack((imputed_values.reshape(-1,1), X_with_nan[:,1:]))
np.vstack((X_with_imputed, X)) # 두 특성 행렬을 연결
```

범주형 데이터 처리

➢ **누락된 클래스 값 대체하기 2**

- 누락된 값을 특성에서 가장 자주 등장하는 값으로 채우기

```
from sklearn.impute import SimpleImputer

# 두 개의 특성 행렬을 합칩니다.
X_complete = np.vstack((X_with_nan, X))

imputer = SimpleImputer(strategy='most_frequent')
imputer.fit_transform(X_complete)
```

범주형 데이터 처리

➢ 불균형한 특성 클래스 처리

- 타깃 벡터가 매우 불균형한 클래스로 이루어져 있는 경우 더 많은 데이터를 수집하는 것이 어렵다면 모델 평가 지표를 변경합니다.
- 모델에 내장된 클래스 가중치 매개변수를 사용하거나 다운샘플링이나 업샘플링을 고려합니다.
- RandomForestClassifier는 불균형한 영향을 줄일 수 있도록 클래스에 가중치를 부여할 수 있는 class_weight 매개변수를 제공합니다.
- class_weight값을 balanced로 지정하여 클래스 빈도에 반비례하게 자동으로 가중치를 만들 수 있습니다.

```
import numpy as np
from sklearn.ensemble import RandomForestClassifier
from sklearn.datasets import load_iris

iris = load_iris()  # 붓꽃 데이터 로드
features = iris.data   # 특성 행렬
target = iris.target  # 타깃 벡터

features = features[40:,:]   # 처음 40개 샘플을 삭제
target = target[40:]

# 클래스 0을 음성 클래스로 하는 이진 타깃 벡터를 만듭니다.
target = np.where((target == 0), 0, 1)
target  # 불균형한 타깃 벡터를 확인
```

범주형 데이터 처리

- **불균형한 특성 클래스 처리**
  -

```
weights = {0: .9, 1: 0.1}  # 가중치 생성
# 가중치를 부여한 랜덤 포레스트 분류기 객체 생성
RandomForestClassifier(class_weight=weights)
# 균형잡힌 클래스 가중치로 랜덤 포레스트 모델을 훈련
RandomForestClassifier(class_weight="balanced")
```

범주형 데이터 처리

➢ 불균형한 특성 클래스 처리

- 다수 클래스의 샘플을 줄이거나(다운샘플링) 소수 클래스의 샘플을 늘릴 수 있습니다.(업샘플링)
- 다운 샘플링에서는 다수 클래스에서 중복을 허용하지 않고 랜덤하게 샘플을 선택하여 소수 클래스와 같은 크기의 샘플 부분집합을 만듭니다.
- 예) 소수 클래스에 10개의 샘플이 있다면 다수 클래스에서 10개의 샘플을 랜덤하게 선택하여 20개의 샘플을 데이터로 사용합니다.

```
# 각 클래스의 샘플 인덱스를 추출합니다.
i_class0 = np.where(target == 0)[0]
i_class1 = np.where(target == 1)[0]

# 각 클래스의 샘플 개수
n_class0 = len(i_class0)
n_class1 = len(i_class1)

# 클래스 0의 샘플만큼 클래스 1에서 중복을 허용하지 않고 랜덤하게 샘플을 뽑습니다.
# from class 1 without replacement
i_class1_downsampled = np.random.choice(i_class1, size=n_class0, replace=False)

# 클래스 0의 타깃 벡터와 다운샘플링된 클래스 1의 타깃 벡터를 합칩니다.
np.hstack((target[i_class0], target[i_class1_downsampled]))
# 클래스 0의 특성 행렬과 다운샘플링된 클래스 1의 특성 행렬을 합칩니다.
np.vstack((features[i_class0,:], features[i_class1_downsampled,:]))[0:5]
```

범주형 데이터 처리

➢ 불균형한 특성 클래스 처리

- 업샘플링에서는 다수 클래스의 샘플만큼 소수 클래스에서 중복을 허용하여 랜덤하게 샘플을 선택합니다.

```
# 클래스 1의 샘플 개수만큼 클래스 0에서 중복을 허용하여 랜덤하게 샘플을 선택합니다.
i_class0_upsampled = np.random.choice(i_class0, size=n_class1, replace=True)

# 클래스 0의 업샘플링된 타깃 벡터와 클래스 1의 타깃 벡터를 합칩니다.
np.concatenate((target[i_class0_upsampled], target[i_class1]))

# 클래스 0의 업샘플링된 특성 행렬과 클래스 1의 특성 행렬을 합칩니다.
np.vstack((features[i_class0_upsampled,:], features[i_class1,:]))[0:5]
```