사이킷런으로 시작하는 머신러닝

01 사이킷런(scikit-learn) 소개와 특징

02 첫 번째 머신러닝 만들어 보기 - 붓꽃 품종 예측하기

```
source: [
    import pandas as pd

    # 붓꽃 데이터 세트를 로딩.
    iris = load_iris()

    # iris.data는 Iris 데이터 세트에서 피처(feature)만으로 된 데이터를 numpy로 가지고 있음.
    iris_data = iris.data

    # iris.target은 붓꽃 데이터 세트에서 레이블(결정 값) 데이터를 numpy로 가지고 있음.
    iris_label = iris.target
    print('iris target값:', iris_label)
    print('iris target명:', iris.target_names)

    # 붓꽃 데이터 세트를 자세히 보기 위해 DataFrame으로 변환.
    iris_df = pd.DataFrame(data=iris_data, columns=iris.feature_names)
    iris_df['label'] = iris.target
    iris_df.head(3)
   ]


source: [
    X_train, X_test, y_train, y_test = train_test_split(iris_data, iris_label,
                                                        test_size=0.2, random_state=11)
   ]

source: [
    # DecisionTreeClassifier 객체 생성
    dt_clf = DecisionTreeClassifier(random_state=11)

    #학습 수행
    dt_clf.fit(X_train, y_train)
   ]


source: [
    # 학습이 완료된 DecisionTreeClassifier 객체에서 테스트 데이터 세트로 예측 수행.
    pred = dt_clf.predict(X_test)
   ]

source: [
    from sklearn.metrics import accuracy_score
    print('예측 정확도: {0:.4f}'.format(accuracy_score(y_test,pred)))
   ]
```

분류 예측 프로세스

- **데이터 세트 분리**: 데이터를 학습 데이터와 테스트 데이터로 분리함.
- 모델 **학습**: 학습 데이터를 기반으로 ML 알고리즘을 적용해 모델을 학습 시킴.
- **예측** 수행: 학습된 ML 모델을 이용해 테스트 데이터의 분류를 예측함.
- **평가**: 이렇게 예측된 결괏값과 테스트 데이터의 실제 결괏값을 비교해 ML모델 성능을 평가함.

03 사이킷런의 기반 프레임워크 익히기

```
source: [
    from sklearn.datasets import load_iris
    iris_data = load_iris()
    print(type(iris_data))
   ]

source: [
    keys = iris_data.keys()
    print('붓꽃 데이터 세트의 키들:', keys)
   ]

source: [
    print('\\n feature_names 의 type:',type(iris_data.feature_names))
    print(' feature_names 의 shape:',len(iris_data.feature_names))
    print(iris_data.feature_names)

    print('\\n target_names 의 type:',type(iris_data.target_names))
    print(' feature_names 의 shape:',len(iris_data.target_names))
    print(iris_data.target_names)

    print('\\n data 의 type:',type(iris_data.data))
    print(' data 의 shape:',iris_data.data.shape)
    print(iris_data['data'])

    print('\\n target 의 type:',type(iris_data.target))
    print(' target 의 shape:',iris_data.target.shape)
    print(iris_data.target)
   ]
```

Estimator 이해 및 fit(), predict() 메서드

Estimator: 지도학습의 분류(classification)과 회귀(regression)의 다양한 알고리즘을 구현한 클래스

fit(): MT 모델 학습을 위해서 제공하는 메서드

predict(): 학습된 모델의 예측을 위해 제공하는 메서드

#p.94~95: 사이킷런의 주요 모듈

data, target: ndarray 타입

target_names, feature_names: ndarray, list 타입

DESCR: string 타입

04 Model Selection 모듈 소개

```
### 학습/테스트 데이터 셋 분리 – train_test_split()

source: [
    from sklearn.datasets import load_iris
    from sklearn.tree import DecisionTreeClassifier
    from sklearn.metrics import accuracy_score

    iris = load_iris()
    dt_clf = DecisionTreeClassifier()
    train_data = iris.data
    train_label = iris.target
    dt_clf.fit(train_data, train_label)
```

```
    # 학습 데이터 셋으로 예측 수행
    pred = dt_clf.predict(train_data)
    print('예측 정확도:',accuracy_score(train_label,pred))
   ]

source: [
    from sklearn.tree import DecisionTreeClassifier
    from sklearn.metrics import accuracy_score
    from sklearn.datasets import load_iris
    from sklearn.model_selection import train_test_split

    dt_clf = DecisionTreeClassifier()
    iris_data = load_iris()

   X_train, X_test, y_train, y_test = train_test_split(iris_data.data, iris_data.target
                                              test_size=0.3, random_state=121)

   ]

source: [
    dt_clf.fit(X_train, y_train)
    pred = dt_clf.predict(X_test)
    print('예측 정확도: {0:.4f}'.format(accuracy_score(y_test,pred)))
   ]

### 교차 검증 K폴드

source: [
    from sklearn.tree import DecisionTreeClassifier
    from sklearn.metrics import accuracy_score
    from sklearn.model_selection import KFold
    import numpy as np

    iris = load_iris()
    features = iris.data
    label = iris.target
    dt_clf = DecisionTreeClassifier(random_state=156)

    # 5개의 폴드 세트로 분리하는 KFold 객체와 폴드 세트별 정확도를 담을 리스트 객체 생성.
    kfold = KFold(n_splits=5)
    cv_accuracy = []
    print('붓꽃 데이터 세트 크기:',features.shape[0])
   ]


   source: [
    n_iter = 0\n",

    # KFold객체의 split( ) 호출하면 폴드 별 학습용, 검증용 테스트의 로우 인덱스를 array로 반환
    for train_index, test_index  in kfold.split(features):
    # kfold.split( )으로 반환된 인덱스를 이용하여 학습용, 검증용 테스트 데이터 추출
    X_train, X_test = features[train_index], features[test_index]
    y_train, y_test = label[train_index], label[test_index]
    #학습 및 예측
    dt_clf.fit(X_train , y_train)
    pred = dt_clf.predict(X_test)
    n_iter += 1
    # 반복 시 마다 정확도 측정
    accuracy = np.round(accuracy_score(y_test,pred), 4)
    train_size = X_train.shape[0]
    test_size = X_test.shape[0]
    print('\n#{0} 교차 검증 정확도 :{1}, 학습 데이터 크기: {2}, 검증 데이터 크기: {3}'
        .format(n_iter, accuracy, train_size, test_size))
    print('#{0} 검증 세트 인덱스:{1}'.format(n_iter,test_index))
    cv_accuracy.append(accuracy)

    # 개별 iteration별 정확도를 합하여 평균 정확도 계산
    print('\n## 평균 검증 정확도:', np.mean(cv_accuracy))
   ]
```

```
* Stratified K 폴드

source: [
    import pandas as pd

    iris = load_iris()

    iris_df = pd.DataFrame(data=iris.data, columns=iris.feature_names)
    iris_df['label']=iris.target
    iris_df['label'].value_counts()

   ]

source: [
    kfold = KFold(n_splits=3)
    # kfold.split(X)는 폴드 세트를 3번 반복할 때마다 달라지는 학습/테스트 용 데이터 로우 인덱스 번호 반환.
    n_iter =0
    for train_index, test_index  in kfold.split(iris_df):
        n_iter += 1
        label_train= iris_df['label'].iloc[train_index]
        label_test= iris_df['label'].iloc[test_index]
        print('## 교차 검증: {0}'.format(n_iter))
        print('학습 레이블 데이터 분포:\\n', label_train.value_counts())
        print('검증 레이블 데이터 분포:\\n', label_test.value_counts())
   ]

source: [
    from sklearn.model_selection import StratifiedKFold

    skf = StratifiedKFold(n_splits=3)
    n_iter=0

    for train_index, test_index in skf.split(iris_df, iris_df['label']):
      n_iter += 1
      label_train= iris_df['label'].iloc[train_index]
      label_test= iris_df['label'].iloc[test_index]
      print('## 교차 검증: {0}'.format(n_iter))
      print('학습 레이블 데이터 분포:\\n', label_train.value_counts())
      print('검증 레이블 데이터 분포:\\n', label_test.value_counts())
   ]

source: [
    dt_clf = DecisionTreeClassifier(random_state=156)\n",

    skfold = StratifiedKFold(n_splits=3)\n",
    n_iter=0\n",
    cv_accuracy=[]\n",

    # StratifiedKFold의 split( ) 호출시 반드시 레이블 데이터 셋도 추가 입력 필요
    for train_index, test_index  in skfold.split(features, label):
        # split( )으로 반환된 인덱스를 이용하여 학습용, 검증용 테스트 데이터 추출
        X_train, X_test = features[train_index], features[test_index]
        y_train, y_test = label[train_index], label[test_index]
        #학습 및 예측
        dt_clf.fit(X_train , y_train)
        pred = dt_clf.predict(X_test)

        # 반복 시 마다 정확도 측정
        n_iter += 1
        accuracy = np.round(accuracy_score(y_test,pred), 4)
        train_size = X_train.shape[0]
        test_size = X_test.shape[0]
        print('\\n#{0} 교차 검증 정확도 :{1}, 학습 데이터 크기: {2}, 검증 데이터 크기: {3}'
              .format(n_iter, accuracy, train_size, test_size))
        print('#{0} 검증 세트 인덱스:{1}'.format(n_iter,test_index))
        cv_accuracy.append(accuracy)

    # 교차 검증별 정확도 및 평균 정확도 계산
    print('\\n## 교차 검증별 정확도:', np.round(cv_accuracy, 4))
    print('## 평균 검증 정확도:', np.mean(cv_accuracy))
   ]
```

```
source: [
    from sklearn.tree import DecisionTreeClassifier
    from sklearn.model_selection import cross_val_score , cross_validate
    from sklearn.datasets import load_iris

    iris_data = load_iris()
    dt_clf = DecisionTreeClassifier(random_state=156)
    data = iris_data.data
    label = iris_data.target

    # 성능 지표는 정확도(accuracy) , 교차 검증 세트는 3개
    scores = cross_val_score(dt_clf , data , label , scoring='accuracy',cv=3)
    print('교차 검증별 정확도:',np.round(scores, 4))
    print('평균 검증 정확도:', np.round(np.mean(scores), 4))
   ]

source: [
    from sklearn.datasets import load_iris
    from sklearn.tree import DecisionTreeClassifier
    from sklearn.model_selection import GridSearchCV

    # 데이터를 로딩하고 학습데이타와 테스트 데이터 분리
    iris = load_iris()
    X_train, X_test, y_train, y_test = train_test_split(iris_data.data, iris_data.target
                                                        test_size=0.2, random_state=121)
    dtree = DecisionTreeClassifier()

    ### parameter 들을 dictionary 형태로 설정
    parameters = {'max_depth':[1,2,3], 'min_samples_split':[2,3]}
   ]

source: [
    "mport pandas as pd

   #param_grid의 하이퍼 파라미터들을 3개의 train, test set fold 로 나누어서 테스트 수행 설정.
    ### refit=True 가 default 임. True이면 가장 좋은 파라미터 설정으로 재 학습 시킴.
    grid_dtree = GridSearchCV(dtree, param_grid=parameters, cv=3, refit=True)

    # 붓꽃 Train 데이터로 param_grid의 하이퍼 파라미터들을 순차적으로 학습/평가 .
    grid_dtree.fit(X_train, y_train)

    # GridSearchCV 결과 추출하여 DataFrame으로 변환
    scores_df = pd.DataFrame(grid_dtree.cv_results_)
    scores_df[['params', 'mean_test_score', 'rank_test_score',
               'split0_test_score', 'split1_test_score', 'split2_test_score']]
   ]

source: [
    # GridSearchCV의 refit으로 이미 학습이 된 estimator 반환
    estimator = grid_dtree.best_estimator_

    # GridSearchCV의 best_estimator_는 이미 최적 하이퍼 파라미터로 학습이 됨
    pred = estimator.predict(X_test)
    print('테스트 데이터 세트 정확도: {0:.4f}'.format(accuracy_score(y_test,pred)))
   ]
```

학습/테스트 데이터 세트 분리 - train_test_split() #return 값~튜블형태

train_test_split()

- 첫 번째 파라미터 → 피치 데이터 세트
- 두 번째 파라미터 → 레이블 데이터 세트
- 선택적 옵션 → test_size(전체 데이터에서 테스트 데이터 세트,의 비율

  train_size(전체 데이터에서 학습용 데이터 세트의 비율

shuffle(데이터를 분산시키는 용도, 데이터를 분리하기 전 섞을지 여부 판단),
random_state(동일한 학해 주어지는 난수 값. 학습/테스트용 데이터 세트를 생성하기 위)

교차검증

:본고사를 치르기 전에 모의고사를 여러 번 보는 것과 같은 것으로 특정 데이터에만 과적합되는 학습 모델이 만들어져 다른 테스트용 데이터가 들어올 경우 생기는 성능 저하의 문제점을 개선하기 위해 이용하는 학습/평가 방식

K폴드 교차 검증 #보편적인 사용

:K개의 데이터 폴드 세트를 만들어서 K번 만큼 각 폴트 세트에 학습과 검증평가를 반복적으로 수행하는 방법

Stratified K 폴드

:불균형한 분포도(_분포도가 한쪽으로 치우침)를 가진 레이블(결정 클래스) 데이터 세트를 위한 K 폴드 방식

- 왜곡된 레이블 데이터 세트에서는 반드시 이용
- 분류에서의 교차 검증은 K폴드가 아닌 **Stratified K 폴드로 분할 되어야**
- 회귀의 결정값은 연속된 숫자값이기 때문임

교차 검증을 보다 간편하게 - cross_val_score()

K 폴드

- 폴드 세트를 설정
- for 루프에서 반복으로 학습 및 테스트 데이터의 인덱스를 추출
- 반복작인 학습과 예측 수행
- 예측 성능 반환

위 과정을 한꺼번에 수행해주는 API ⇒ cross_val_score()

#p112 cross_val_score()의 주요 파라미터 설명 !!자료 보충

GridSearchCV - 교차 검증과 최적 하이퍼 파라미터 튜닝을 한번에

하이퍼 파라미터

- 머신러닝 알고리즘 구성 요소 → 알고리즘 예측 성능과 관련

GridSearchCV

- 하이퍼 파라미터의 최적 값 찾음
- 클래스의 생성자로 들어가는 주요 파라미터 설명 #p114
-

05 데이터 전처리

```
### 데이터 인코딩
* 레이블 인코딩(Label encoding)

source: [
    from sklearn.preprocessing import LabelEncoder

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

    # LabelEncoder를 객체로 생성한 후 , fit( ) 과 transform( ) 으로 label 인코딩 수행.
    encoder = LabelEncoder()
    encoder.fit(items)
    labels = encoder.transform(items)
    print('인코딩 변환값:',labels)
   ]

source: [
    print('인코딩 클래스:',encoder.classes_)
   ]

source: [
    print('디코딩 원본 값:',encoder.inverse_transform([4, 5, 2, 0, 1, 1, 3, 3]))
   ]

* 원-핫 인코딩(One-Hot encoding)

source: [
    from sklearn.preprocessing import OneHotEncoder
    import numpy as np

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

    # 먼저 숫자값으로 변환을 위해 LabelEncoder로 변환합니다.
    encoder = LabelEncoder()
    encoder.fit(items)
    labels = encoder.transform(items)
    # 2차원 데이터로 변환합니다.
    labels = labels.reshape(-1,1)

    # 원-핫 인코딩을 적용합니다.
    oh_encoder = OneHotEncoder()
    oh_encoder.fit(labels)
    oh_labels = oh_encoder.transform(labels)
    print('원-핫 인코딩 데이터')
    print(oh_labels.toarray())
    print('원-핫 인코딩 데이터 차원')
    print(oh_labels.shape)
   ]

source: [
    import pandas as pd

    df = pd.DataFrame({'item':['TV','냉장고','전자렌지','컴퓨터','선풍기','선풍기','믹서','믹서'] })
    pd.get_dummies(df)
   ]

### 피처 스케일링과 정규화
* StandardScaler

source: [
    from sklearn.datasets import load_iris
    import pandas as pd
    # 붓꽃 데이터 셋을 로딩하고 DataFrame으로 변환합니다.
    iris = load_iris()
    iris_data = iris.data
    iris_df = pd.DataFrame(data=iris_data, columns=iris.feature_names)

    print('feature 들의 평균 값')
    print(iris_df.mean())
    print('\\nfeature 들의 분산 값')
    print(iris_df.var())
```

```
   ]

* MinMaxScaler

source: [
    from sklearn.preprocessing import StandardScaler

    # StandardScaler객체 생성
    scaler = StandardScaler()
    # StandardScaler 로 데이터 셋 변환. fit( ) 과 transform( ) 호출.
    scaler.fit(iris_df)
    iris_scaled = scaler.transform(iris_df)

    #transform( )시 scale 변환된 데이터 셋이 numpy ndarry로 반환되어 이를 DataFrame으로 변환
    iris_df_scaled = pd.DataFrame(data=iris_scaled, columns=iris.feature_names)
    print('feature 들의 평균 값')
    print(iris_df_scaled.mean())
    print('\\nfeature 들의 분산 값')
    print(iris_df_scaled.var())
   ]


* Scaler를 이용하여 학습 데이터와 테스트 데이터에 fit(), transform(), fit_transform() 적용 시 유의사항
source: [
    from sklearn.preprocessing import MinMaxScaler
    import numpy as np

    # 학습 데이터는 0 부터 10까지, 테스트 데이터는 0 부터 5까지 값을 가지는 데이터 세트로 생성
    # Scaler클래스의 fit(), transform()은 2차원 이상 데이터만 가능하므로 reshape(-1, 1)로 차원 변경
    train_array = np.arange(0, 11).reshape(-1, 1)
    test_array =  np.arange(0, 6).reshape(-1, 1)
   ]

source: [
    # 최소값 0, 최대값 1로 변환하는 MinMaxScaler객체 생성
    scaler = MinMaxScaler()
    # fit()하게 되면 train_array 데이터의 최소값이 0, 최대값이 10으로 설정.
    scaler.fit(train_array)
    # 1/10 scale로 train_array 데이터 변환함. 원본 10-> 1로 변환됨.
    train_scaled = scaler.transform(train_array)

    print('원본 train_array 데이터:', np.round(train_array.reshape(-1), 2))
    print('Scale된 train_array 데이터:', np.round(train_scaled.reshape(-1), 2))
   ]

source: [
    # 앞에서 생성한 MinMaxScaler에 test_array를 fit()하게 되면 원본 데이터의 최소값이 0, 최대값이 5으로 설정됨
    scaler.fit(test_array)
    # 1/5 scale로 test_array 데이터 변환함. 원본 5->1로 변환.
    test_scaled = scaler.transform(test_array)
    # train_array 변환 출력
    print('원본 test_array 데이터:', np.round(test_array.reshape(-1), 2))
    print('Scale된 test_array 데이터:', np.round(test_scaled.reshape(-1), 2))
   ]

source: [
    scaler = MinMaxScaler()
    scaler.fit(train_array)
    train_scaled = scaler.transform(train_array)
    print('원본 train_array 데이터:', np.round(train_array.reshape(-1), 2))
    print('Scale된 train_array 데이터:', np.round(train_scaled.reshape(-1), 2))

    # test_array에 Scale 변환을 할 때는 반드시 fit()을 호출하지 않고 transform() 만으로 변환해야 함.
    test_scaled = scaler.transform(test_array)
    print('\\n원본 test_array 데이터:', np.round(test_array.reshape(-1), 2))
    print('Scale된 test_array 데이터:', np.round(test_scaled.reshape(-1), 2))
   ]
```

데이터 인코딩

대표적인 인코딩 방식

- 레이블 인코딩
- 원-핫 인코딩

레이블 인코딩

- LabelEncoder클래스로 구현
- LabelEncoder객체 생성 후 fit()과 transform() 호출
- 문자열 값을 숫자형 카테고리 값으로 변환 *숫자의 특성으로 인한(대소비교) 중요도/순서로 인식되는 문제 주의

원-핫 인코딩

- 피처 값의 유형에 따라 새로운 피처를 추가해 고유 값에 해당하는 칼럼에만 1을 표시
- 나머지 칼럼에는 0을 표시
- 해당 고유 값에 매칭되는 피처만 1이 되고 나머지 피처는 0을 입력함
- 사이킷런에서 OneHotEncoder 클래스로 쉽게 변환 가능

  *변환하기 전에 모든 문자열 값이 숫자형 값으로 변환되어 있어야 함

  *입력 값으로 2차원 데이터가 필요함

피처 스케일링 정규화

- 서로 다은 변수의 값 범위를 일정한 수준으로 맞추는 작업
- 표준화 & 정규화가 이에 대한 대표적인 예

*표준화: 데이터 피처 각각이 평균이 0이고 분산이 1인 가우시안 정규 분포를 가진 값으로 변환하는 것

*정규화: 서로 다른 피처의 크기를 통일하기 위해 크기를 변환해주는 개념

StandardScaler

- 표준화를 쉽게 지원하기 위한 클래스
- 특히 사이킷런에서 구현한 RBF 커널을 이용하는 <서포트 벡터 머신>, <선형 회귀>, <로지스틱 회귀> 는 데이터가 가우시안 분포를 가지고 있다고 가정하고 구현되었기에 사전에 표준화를 적용

MinMaxScaler

- 데이터값을 0과 1사이의 범위 값으로 변환 *음수 존재시 (1-~+1)값으로 변환
- 데이터의 분포가 가우시안 분포가 아닐 경우 Mln, Max 적용

학습 데이터와 테스트 데이터의 스케일링 변환 시 유의점 #p128~

- 전체 데이터의 스케일링 변환을 적용한 뒤 학습과 테스트 데이터로 분리
- 1이 여의치 않다면 테스트 데이터 변환 시에는 fit()이나 fit_transform()을 적용하지 않고 학습 데이터로 이미 fit()된 Scaler 객체를 이용해 transform()으로 변환

06 사이킷런으로 수행하는 타이타닉 생존자 예측

```
source: [
    import numpy as np
    import pandas as pd
    import matplotlib.pyplot as plt
    import seaborn as sns
    %matplotlib inline

    titanic_df = pd.read_csv('./titanic_train.csv')
    titanic_df.head(3)
   ]

source: [
    print('\\n ### train 데이터 정보 ###  \\n')
    print(titanic_df.info())
   ]

source: [
    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 갯수 ',titanic_df.isnull().sum().sum())
   ]

source: [
    print(' Sex 값 분포 :\\n',titanic_df['Sex'].value_counts())
    print('\\n Cabin 값 분포 :\\n',titanic_df['Cabin'].value_counts())
    print('\\n Embarked 값 분포 :\\n',titanic_df['Embarked'].value_counts())
   ]

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

source: [
    titanic_df.groupby(['Sex','Survived'])['Survived'].count()
   ]

source: [
    sns.barplot(x='Sex', y = 'Survived', data=titanic_df)
   ]

source: [
    sns.barplot(x='Pclass', y='Survived', hue='Sex', data=titanic_df)
   ]

source: [
    # 입력 age에 따라 구분값을 반환하는 함수 설정. DataFrame의 apply lambda식에 사용.
    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

    # 막대그래프의 크기 figure를 더 크게 설정
    plt.figure(figsize=(10,6))

    #X축의 값을 순차적으로 표시하기 위한 설정
    group_names = ['Unknown', 'Baby', 'Child', 'Teenager', 'Student', 'Young Adult', 'Adult', 'Elderly']

    # lambda 식에 위에서 생성한 get_category( ) 함수를 반환값으로 지정.
    # get_category(X)는 입력값으로 'Age' 컬럼값을 받아서 해당하는 cat 반환
    titanic_df['Age_cat'] = titanic_df['Age'].apply(lambda x : get_category(x))
    sns.barplot(x='Age_cat', y = 'Survived', hue='Sex', data=titanic_df, order=group_names)
    titanic_df.drop('Age_cat', axis=1, inplace=True)
   ]


source: [
    from sklearn import preprocessing

    def encode_features(dataDF)\n",
        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()
   ]

source: [
    from sklearn.preprocessing import LabelEncoder

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

    # 머신러닝 알고리즘에 불필요한 속성 제거
    def drop_features(df):
        df.drop(['PassengerId','Name','Ticket'],axis=1,inplace=True)
        return df\n",

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

    # 앞에서 설정한 Data Preprocessing 함수 호출
    def transform_features(df):
        df = fillna(df)
        df = drop_features(df)
        df = format_features(df)
        return df
   ]

source: [
    # 원본 데이터를 재로딩 하고, feature데이터 셋과 Label 데이터 셋 추출.
    titanic_df = pd.read_csv('./titanic_train.csv')
    y_titanic_df = titanic_df['Survived']
    X_titanic_df= titanic_df.drop('Survived',axis=1)
```

```
    X_titanic_df = transform_features(X_titanic_df)
   ]

source: [
    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_size=0.2, random_state=11)
   ]

source: [
    from sklearn.tree import DecisionTreeClassifier
    from sklearn.ensemble import RandomForestClassifier
    from sklearn.linear_model import LogisticRegression
    from sklearn.metrics import accuracy_score

    # 결정트리, Random Forest, 로지스틱 회귀를 위한 사이킷런 Classifier 클래스 생성
    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)))
   ]

source: [
    from sklearn.model_selection import KFold

    def exec_kfold(clf, folds=5):
        # 폴드 세트를 5개인 KFold객체를 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_index]
            y_train, y_test = y_titanic_df.values[train_index], y_titanic_df.values[test_index]

            # 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))
    # exec_kfold 호출
    exec_kfold(dt_clf , folds=5)
   ]

source: [
    from sklearn.model_selection import cross_val_score

    scores = cross_val_score(dt_clf, X_titanic_df , y_titanic_df , cv=5)
    for iter_count,accuracy in enumerate(scores):
        print(\"교차 검증 {0} 정확도: {1:.4f}\".format(iter_count, accuracy))
```

```
    print(\"평균 정확도: {0:.4f}\".format(np.mean(scores)))
   ]

source: [
    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('GridSearchCV 최적 하이퍼 파라미터 :',grid_dclf.best_params_)
    print('GridSearchCV 최고 정확도: {0:.4f}'.format(grid_dclf.best_score_))
    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('테스트 세트에서의 DecisionTreeClassifier 정확도 : {0:.4f}'.format(accuracy))
   ]
```