AdaBoost 分类器代码:

jupyter AdaBoost.py 6 minutes ago

Logout

File Edit View Language

Python

```python
#!/usr/bin/env python
# coding: utf-8

# In[1]:


# Import required packages
import pandas as pd
import numpy as np
import math
from sklearn.tree import DecisionTreeClassifier
from sklearn.base import BaseEstimator, ClassifierMixin


# In[2]:


""" Summary: The mdoel will be a AdaBoost classification.

We will build AdaBoost class to make classification. There will be 9 hyper-parameters in
the model. The model will contain fit(), predict(), get_params(), set_params(),
score(), decision_dunction(), __init__().

"""

class AdaBoostClassifier(BaseEstimator, ClassifierMixin):

    """ A class used to make classification.


    Attributes
    - - - - - -
    n_estimators : int
        The number of weak classifiers
    base_estimator : class
        The base estimator we choose to train the AdaBoost.
    estimators_ : list
        The list of base_estimators


    """
    def __init__(self, n_estimators=20, base_estimator=DecisionTreeClassifier(), estimators_=[None]):
        """


        Parameters
        - - - - - -
        n_estimators : int
            The number of weak classifiers
        base_estimator : class
            The base estimator we choose to train the AdaBoost.
        estimators_ : list
            The list of base_estimators

        """

        self.n_estimators = n_estimators
        self.base_estimator = base_estimator
        self.estimators_ = estimators_

    def fit(self, X, y, n_trees=None,
            sample_weight=None,
            sample_size=None,
            max_depth=None,
```

```python
            max_depth=None,
            max_features=None,
            min_samples_split=None,
            min_samples_leaf=None,
            min_weight_fraction_leaf=None,
            max_leaf_nodes=None,
            ):

        """Train the model.


        Parameters
        - - - - - -
        n_trees: int
            The number of bootstrapped trees
        sample_weight: list
            The weight for each samples
        sample_size: int
            The size of bootstrapped samples.
        max_depth: int
            The maximum depth of a single tree.
        max_features: int
            The maximum number of featuers used to train a single tree.
        min_samples_split: int
            The minimum number of samples required to continue splitting.
        min_samples_leaf: int
            The minimum number of samples in a leaf.
        min_weight_fraction_leaf: int
            The minimum weight fraction.
        max_leaf_nodes: int
            The maximum number of nodes.

        Returns
        - - - -
        self

        """

        # If the number of samples in a leaf is below the Min_samples_leaf, the leaf would be pruned.
        # the leaf nodes, and if it is less than this value, it will be pruned together with the sibling nodes.
        y = y.reshape(-1, 1)
        # Convert value of 0 to -1 in y.
        index = (y==0)
        y[index] = -1

        # The processes below set the initial value of each parameter.
        # if the 'if' function is true, the default value will be used,
        # but if False, it means users has input their own values.
        if self.estimators_ == [None]:
            self.estimators_ = []
        if n_trees == None:
            n_trees = 20
        if sample_weight == None:
            sample_weight = np.full(np.shape(y), 1/len(y)).reshape(-1, 1)
        if sample_size == None:
            sample_size = X.shape[0]
        if max_depth == None:
            max_depth = 5
        if max_features == None:
            max_features = None
        if min_samples_split == None:
            min_samples_split = 2
        if min_samples_leaf == None:
```

```python
        if min_samples_leaf == None:
            min_samples_leaf = 1
        if min_weight_fraction_leaf == None:
            min_weight_fraction_leaf = 0
        if max_leaf_nodes == None:
            max_leaf_nodes = None

        # Set attributes:
        # trees_ represents the list of boostrapped sample trees.
        # tree_preds records the prediction of each boostrapped sample trees.
        # estimator_errors represents saves the errors of each base estimators.
        # estimator_weights represents saves the updated voting value of each base estimator.
        trees_ = []
        tree_preds_ = []
        self.estimator_errors_ = []
        self.estimator_weights_ = []


        # The processes below will show how we train the weak classifiers.
        # diffs is the list of values of the difference of the real value and predicted
        # value for each boostrapped tree.
        # For example, If diffs[0] is [1, 1, 0], means the first two prediction is different
        # from the real value.
        diffs = []
        for n in range(n_trees):
            # Genrate m boostrap samples from a dataset
            rows = np.random.choice(X.shape[0], sample_size)
            X_sample = X[rows]
            y_sample = y[rows]

            clf = DecisionTreeClassifier(max_depth=max_depth,
                                         max_features=max_features,
                                         min_samples_split=min_samples_split,
                                         min_samples_leaf=min_samples_leaf,
                                         min_weight_fraction_leaf=min_weight_fraction_leaf,
                                         max_leaf_nodes=max_leaf_nodes
                                         )
            clf.fit(X_sample, y_sample)

            tree_pred = clf.predict(X)
            tree_pred = np.array(tree_pred).reshape(-1,1)

            tree_preds_.append(tree_pred)
            diff = ((tree_pred-y)/2)**2
            diffs.append(diff)
            trees_.append(clf)
        # The iteration process.
        for i in range(self.n_estimators):
            # Calculate misclassification rate for each classifier
            # The choose the minimum misclassification rate.
            errors = []
            for t in range(len(tree_preds_)):
                error = sum(sample_weight*diffs[t])[0]
                errors.append(error)
            min_error = min(errors)

            # Stop criteria 1: if the minimum error is larger than 0.5,
            # this means the trees can not get better classification than random guess so we'd better stop.
            if min_error>=0.5:
                return self
            # We get the index of the selected base estimator using minerror_index.
            # And pick the classifier with the lowest error.
```

```
            # Then take out the diff with lowest error.
            minerror_index = errors.index(min_error)
            base_estimator_= trees_[minerror_index]
            self.min_error_diff = diffs[minerror_index]

            # Calculate voting power for the best classifier
            min_error_2 = np.clip(min_error, 1e-15, 1-1e-15).copy()
            a = 0.5 * (math.log((1-min_error_2)/min_error_2, 10))
            # Update the base estimator.
            # Update the voting power.
            self.estimators_.append(base_estimator_)
            self.estimator_weights_.append(a)
            self.estimator_errors_.append(min_error)

            #Stop point 2: if we got perfect prediction, then we stop.
            if min_error==0:
                return self
            # Update the data sample weights using e of the most recent classifier.
            # If we make a mistake in the prediction,
            # then the weight of the sample will get larger then before,
            # if we predict right, then the weight will get smaller.
            sample_weight_new = []
            for i in range(len(self.min_error_diff)):
                if self.min_error_diff[i]==0:
                    new = (1/(2*(1-min_error)))*sample_weight[i]
                    sample_weight_new.append(new)
                if self.min_error_diff[i]==1:
                    new = (1/(2*min_error))*sample_weight[i]
                    sample_weight_new.append(new)
            sample_weight_new = np.array(sample_weight_new)
            sample_weight = sample_weight_new
```

```

        return self

    # We are supposed to return the information of parameters set at the begining.
    # get_params returns a dict.
    def get_params(self, deep=True):
        """ Print the parameter information of the class.

        Returns
        - - - -
        { 'n_estimators': self.n_estimators,
          'base_estimator': self.base_estimator, 'estimators_':self.estimators_
        }

        """

        return { 'n_estimators': self.n_estimators,
                 'base_estimator': self.base_estimator, 'estimators_':self.estimators_
               }

    # We make prediction in this process.
    # We just input X.
    def predict(self, X):
        """ Prdict the y based on the input X


        Returns
        - - - -
        y

        """
```

```python

        preds = []
        for i in range(len(self.estimators_)):
            u = self.estimators_[i].predict(X)
            u = u.reshape(-1, 1)
            pred = u * self.estimator_weights_[i]
            preds.append(pred)

        final_preds = np.array(preds).sum(axis=0)
        y = self.decision_funtion(final_preds)

        return y  #predict returns an ndarray

    # score function will give the accuracy of a prediction.
    def score(self, X, y):
        """  Calulate the accuracy of the prediction to X and y

             Returns
             - - - -
             score

        """

        pred = self.predict(X)
        pred = np.array(pred).reshape(-1,1)

        y = y.reshape(-1, 1)

        index = (y==0)
        y[index]=-1

        diff = ((pred-y)/2)**2
        score = 1 - diff.sum()/len(diff)
        return score    #score fucntion returns a float
    # We manually set value of parameters in this step.
    # set_params has single arg called params, which is a **kwargs.
    # set_params returns self.
    def set_params(self, **params):

        """ Set the parameters for the class.

            Returns
            - - - -
            self


        """

        for parameter, value in params.items():
            setattr(self, parameter, value)
        return self
        #set_params has single arg called params, which is a **kwargs
        #set_params returns self)
     # We refer to a sign() function in decision_function()
    def decision_funtion(self, X):
        """ Assign 1 and -1 to the prediction

            Returns
            - - - -
            y

        """

        y = np.sign(X)
        return y


# In[ ]:
```