01 - Supervised Learning - Classification - AdaBoost(Solution)_st122097_thantham

September 20, 2021

1 Programming for Data Science and Artificial Intelligence

1.1 Classification - AdaBoost

1.2 Name: Thantham Khamyai

1.3 Student ID: 122097

1.3.1 ===Task===

Your work: Let's modify the above scratch code: - Notice that if err $= 0$, then α will be undefined, thus attempt to fix this by adding some very small value to the lower term - Notice that sklearn version of AdaBoost has a parameter learning_rate. This is in fact the $\frac{1}{2}$ in front of the α calculation. Attempt to change this $\frac{1}{2}$ into a parameter called eta, and try different values of it and see whether accuracy is improved. Note that sklearn default this value to 1. - Observe that we are actually using sklearn DecisionTreeClassifier. If we take a look at it closely, it is actually using weighted gini index, instead of weighted errors that we learn above. Attempt to write your own class of class Stump that actually uses weighted errors, instead of weighted gini index - Put everything into a class

[1]:
```
from sklearn.datasets import make_classification
from sklearn.model_selection import train_test_split
import numpy as np

X, y = make_classification(n_samples=500, random_state=48)
y = np.where(y==0,-1,1)  #change our y to be -1 if it is 0, otherwise 1

X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.3, random_state=42)
```

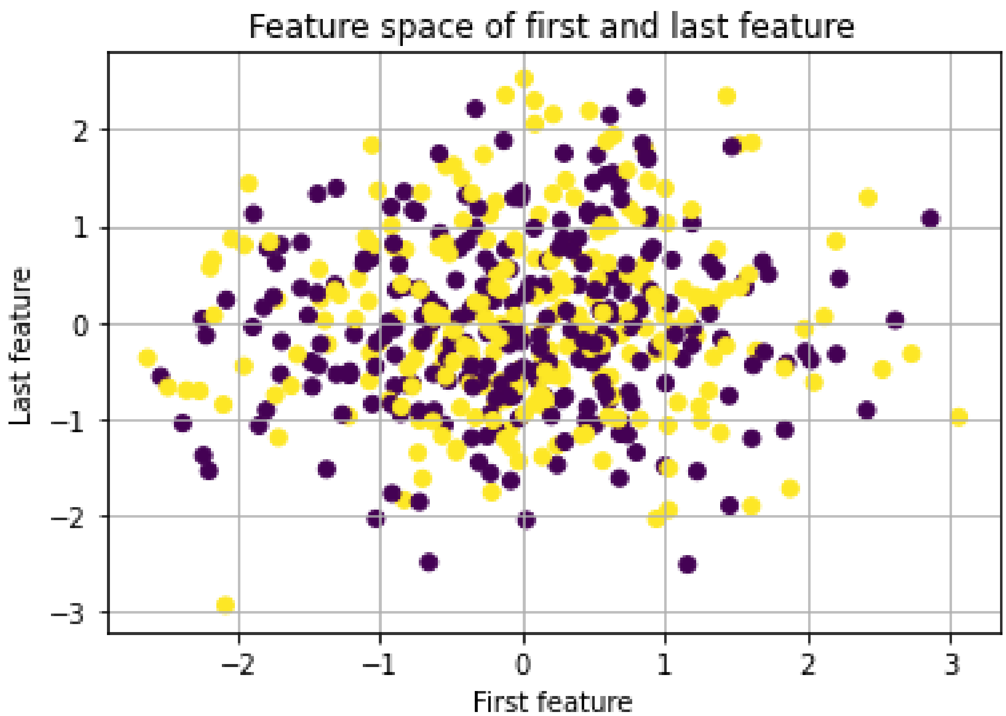
Show the scatter of dataset before for imagining the dataset looks like. However, the dataset was 20 dimension features, we therefore just show it simply first and last feature combination

[2]:
```
import matplotlib.pyplot as plt

plt.scatter(X[:,0], X[:, -1], c=y)
plt.title('Feature space of first and last feature')
plt.xlabel('First feature')
```

```
plt.ylabel('Last feature')
plt.grid()
plt.show()


print('Shape of dataset: ',X.shape)
```




```
Shape of dataset:  (500, 20)
```

1.3.2 TASK 3: Stump class

Actually, we have to put stump class may be into inner class, but we personally have separated class called **StumpClassifier** to further use separately for different appraoch.

This is also make the **Adaboost** class being more feasible to be used for different kinds of classifier

[3]:
```
class StumpClassifier:

    def __init__(self):
        self.polarity = 1
        self.feature_index = None
        self.threshold = None

    def fit(self, X, y):
```

```python
        self.min_err = np.inf
        m,n = X.shape
        W = np.full(m, 1/m) # init equal weight

        for feature in range(n): # looping for all features

                feature_vals = np.sort(np.unique(X[:, feature])) # get sorted features
                thresholds = (feature_vals[:-1] + feature_vals[1:])/2 # get all of thresholds in features

                for threshold in thresholds: # looping for each threshold

                    for polarity in [1, -1]: # check for polarity twice

                        yhat = np.ones(len(y)) # init all answers as 1

                        # change the answer into -1 if the feature value that less than threshold in current polarity
                        yhat[polarity * X[:, feature] < polarity * threshold] = -1

                        # evaluate error
                        err = W[(yhat != y)].sum()

                        # if error is less than before -> define current feature_idx and threshold of splitting
                        if err < self.min_err:
                            self.polarity = polarity
                            self.threshold = threshold
                            self.feature_index = feature
                            self.min_err = err

                # if further feature splitting gives more less of error -> define that feature for feature split


    def predict(self, X):
        m, n = X.shape

        pred = np.ones(m) # init all answer as 1

        # change the answer into -1 if the feature value that less than threshold in current polarity
```

```
        pred[self.polarity * X[:,self.feature_index] < self.polarity * self.
↪threshold] = -1

        return np.sign(pred) # return sign of the answer
```

1.3.3 TASK 4: Put every thing into a class

I made this class similar to the way of sklearn do. we can put any classifier into the this class.

[4]:
```
from sklearn.tree import DecisionTreeClassifier
from sklearn.neighbors import KNeighborsClassifier
from sklearn.svm import SVC

class AdaBoost:

    def __init__(self, estimator = DecisionTreeClassifier, n_estimators=20,␣
↪model_params={}, eta=0.5):

        self.n_estimators = n_estimators
        self.model_params = model_params
        self.eps = 1e-10 # < ======== TASK 1: Epsilon
        self.eta = eta # < ====================== TASK 2: Adding learning rate␣
↪(ETA)

        # init estimator classifiers set following the predefined params
        self.models = [estimator(**self.model_params) for _ in range(self.
↪n_estimators)]


    def fit(self, X_train, y_train):

        m, n = X_train.shape

        self.W = np.full(m, 1/m) # init weak learner weights of samples

        self.alpha = np.zeros(self.n_estimators) # init alpha of each classifier

        for i, model in enumerate(self.models): # looping each model in modelset

            model.fit(X_train, y_train) # simply fit
            yhat = model.predict(X_train) #simply predict


        err = self.W[(y_train != yhat)].sum() # evaluate error of current model

        # calculate alpha (voting power) of current model
```

```
        self.alpha = self.eta * np.log((1-err)/(err + self.eps)) # < == Eps␣
↪implementation

        # calculate Weaker learner weights of each samples
        self.W = (self.W * np.exp(-self.alpha*yhat*y_train) ) / np.sum(self.W)


    def predict(self, X_test):

        H_X = 0

        for i, model in enumerate(self.models):  # looping for each model in␣
↪modelset

            yhat = model.predict(X_test) # simply predict
            H_X += self.alpha * yhat # weight the answers

        return np.sign(H_X) # just give the sign of answers
```

To show up the model class, I put **StumpClassifier** for estimator and keep it default number of estimators

[5]:
```
model = AdaBoost(estimator=StumpClassifier, n_estimators=20, model_params={},␣
 ↪eta=0.5)
model.fit(X_train, y_train)
```

As we known, lets predict the result of training

[6]:
```
y_pred = model.predict(X_test)
```

The result is also being the same as set of {-1,+1} following therotical

[7]:
```
y_pred[:20]
```

[7]:
```
array([ 1.,  1.,  1., -1.,  1., -1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,
        1., -1.,  1.,  1., -1., -1.,  1.])
```

Finally, examine the classification report

[8]:
```
from sklearn.metrics import classification_report

print(classification_report(y_test, y_pred))
```

```
              precision    recall  f1-score   support

          -1       0.80      0.80      0.80        70
           1       0.82      0.82      0.82        80

    accuracy                           0.81       150
```

```
   macro avg       0.81      0.81      0.81       150
weighted avg       0.81      0.81      0.81       150
```

1.3.4 Adaboost Alternative Usage

Lets try implementing different way of **Adaboost**: KNN, SVM, DecisionTree

[9]:
```
ada_knn = AdaBoost(estimator=KNeighborsClassifier, n_estimators=10,␣
 ↪model_params={'n_neighbors': 5}, eta=0.5)
ada_knn.fit(X_train, y_train)
print(classification_report(y_test, ada_knn.predict(X_test)))
```

```
              precision    recall  f1-score   support

          -1       0.77      0.80      0.78        70
           1       0.82      0.79      0.80        80

    accuracy                           0.79       150
   macro avg       0.79      0.79      0.79       150
weighted avg       0.79      0.79      0.79       150
```

[10]:
```
ada_svm = AdaBoost(estimator=SVC, n_estimators=5, model_params={}, eta=0.5)
ada_svm.fit(X_train, y_train)
print(classification_report(y_test, ada_svm.predict(X_test)))
```

```
              precision    recall  f1-score   support

          -1       0.76      0.81      0.79        70
           1       0.83      0.78      0.80        80

    accuracy                           0.79       150
   macro avg       0.79      0.79      0.79       150
weighted avg       0.80      0.79      0.79       150
```

[11]:
```
ada_dt = AdaBoost(estimator=DecisionTreeClassifier, n_estimators=5,␣
 ↪model_params={'max_depth':3}, eta=0.5)
ada_dt.fit(X_train, y_train)
print(classification_report(y_test, ada_dt.predict(X_test)))
```

```
              precision    recall  f1-score   support

          -1       0.73      0.87      0.80        70
           1       0.87      0.72      0.79        80

    accuracy                           0.79       150
   macro avg       0.80      0.80      0.79       150
```

```
weighted avg       0.80      0.79      0.79       150
```

StumpClassifier Usage For show up the usage of Stump classifier

[12]:
```
stump = StumpClassifier()
stump.fit(X_train, y_train)
print(classification_report(y_test, stump.predict(X_test)))

print('Feature to split: ', stump.feature_index)
print('Feature threshold: ', stump.threshold)
print('Min error: ', stump.min_err)
```

```
              precision    recall  f1-score   support

          -1       0.80      0.80      0.80        70
           1       0.82      0.82      0.82        80

    accuracy                           0.81       150
   macro avg       0.81      0.81      0.81       150
weighted avg       0.81      0.81      0.81       150

Feature to split:  13
Feature threshold:  -0.24586126765035565
Min error:  0.14
```

[ ]:
```

```