Programming for Data Science and Artificial Intelligence

Classification - AdaBoost

Readings:

- [GERON] Ch7
- [VANDER] Ch5
- [HASTIE] Ch16
- https://scikit-learn.org/stable/modules/ensemble.html

In [136…

```
Name = "Muhammad Omer Farooq Bhatti"
Id = "122498"
```

In [5]:

```
from sklearn.model_selection import train_test_split
from sklearn.datasets import make_moons
import numpy as np
import matplotlib.pyplot as plt
from sklearn.tree import DecisionTreeClassifier
from sklearn.datasets import make_classification
```

AdaBoost

AdaBoost is a boosting algorithm that try to take a weak classifier on top of one another, **boosting** the overall performance. AdaBoost is extremely simple to use and implement, and often gives very effective results. There is tremendous flexibility in the choice of weak classifier as well. Anyhow, Decision Tree with `max_depth=1` and `max_leaf_nodes=2` are often used (also known as **stump**)

Suppose we are given training data ${(\mathbf{x_i}, y_i)}$, where $\mathbf{x_i} \in \mathbb{R}^n$ and $y_i \in \{-1, 1\}$. And we have S number of weak classifiers, denoted $h_s(x)$. For each classifier, we define α_s as the *voting power* of the classifier $h_s(x)$. Then, the hypothesis function is based on a linear combination of the weak classifier and is written as:

$$ \begin{aligned} h(x) & = \text{sign}\big(\alpha_1h_1(x) + \alpha_2h_2(x) + \cdots + \alpha_sh_s(x))\big) \\ & = \text{sign}\big(\sum_{s=1}^{S}\alpha_sh_s(x)\big) \end{aligned} $$

Our job is to find the optimal α_s, so we can know which classifier we should give more weightage (i.e., believe more) in our hypothesis function since their accuracy is relatively better compared to other classifiers. To get this alpha, we should define what is "good" classifier. This is simple, since good classifier should simply has the minimum weighted errors as:

$$\epsilon_s = \sum_{i=1}^m w_i^{s}I(h_s(x_i) \neq y_i) $$

in which the weights are initialized in the beginning as

$$w_i^{(1)} = \frac{1}{m}$$

After we perform the first classifier, we update the weights of the samples using this formula:

$$w_i^{(s+1)} = \frac{w_i^{(s)}e^{ -\alpha_sh_s(\mathbf{x_i}) y_i}}{{\displaystyle\sum_{i=1}^m w_i^{s}}} $$

where α_s is:

$$\alpha_s = \frac{1}{2}ln\frac{1-\epsilon_s}{\epsilon_s}$$

Putting everything together:

1. Loop through all features, threshold, and polarity, identify the best stump which has lowest weighted errors.

2. Calculate alpha of the first classifier

$$\alpha_s = \frac{1}{2}ln\frac{1-\epsilon_s}{\epsilon_s}$$

1. Exaggerate the incorrect samples using

$$w_i^{(s+1)} = \frac{w_i^{(s)}e^{ -\alpha_sh_s(\mathbf{x_i}) y_i}}{{\displaystyle\sum_{i=1}^m w_i^{s}}} $$

1. Repeat 1.

2. We stop 1-4 using max_iter, early stopping, or number of classifiers.

3. To predict, we use the hypothesis function:

$$ H(x) = \text{sign}\big(\sum_{s=1}^{S}\alpha_sh_s(x)\big) $$

===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

In [113…

```
class adaptiveBoostClassifier:
    def __init__(self, S=20):
        self.S=S            #Number of classifiers
        self.models=[]
        model_parameters = {'max_depth':1, 'max_leaf_nodes':2}
        for i in range(self.S):
            self.models.append( DecisionTreeClassifier(**model_parameters) )  #Declaring models using **keyw

        self.alpha = np.zeros(self.S) #alpha values for all classifiers


    def fit(self, X, y, eta=1):              # <-- eta is the learning rate used to update alpha values

        sample_weights = np.full(X.shape[0], 1/X.shape[0]) #Weight vector of size (m= number of samples),
                                                           #initialized to 1/m for each element

        for idx, model in enumerate(self.models):
            model.fit(X, y, sample_weight=sample_weights) #For each classifier, train with sample_weights
            yhat = model.predict(X)                       #Perform prediction using the training samples the

            error = sample_weights[yhat!=y].sum()  #Computing error 'Es' for the model based on summation of
                                                   #assigned to individual training samples

            #Higher 'Es' --> lower alpha (and vice versa)
            self.alpha[idx] = eta*np.log((1-error)/error+0.001)    #Use 'Es' error to get
            #print(f"alpha[{idx}]", self.alpha[idx])               #alpha (voting power for the classifier),
                                                                  #also used for re-evaluating weights for ind
                                                                #training samples, which are used to train the

            #Update weights assigned to individual training samples
            #These updated weights are used to train the next classifier
            sample_weights = sample_weights * np.exp(-self.alpha[idx]* y * yhat)
            sample_weights = sample_weights/np.sum(sample_weights)  #Normalize sample_weights
            #print(f"max weight: ", sample_weights[sample_weights.argmax()])

    def predict(self, X_test):
        #Each classifier gets a corresponding weightage as per pre-calculated alpha
        yhat=0
        for idx, model in enumerate(self.models):
            yhat = yhat + self.alpha[idx] * model.predict(X_test)
        yhat = np.sign(yhat)
        return yhat
```

In [131…

```
#Make classification data
X, y = make_classification(n_samples=500, random_state = 100)

#Replace y=0 with y=-1 (same as with SVM)
y[y==0] = -1

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.3, random_state = 50)

model = adaptiveBoostClassifier(S=10)
model.fit(X_train, y_train, eta=0.5)
yhat = model.predict(X_test)
print("Accuracy = ", np.sum(yhat==y_test)/y_test.shape[0])
```

```
Accuracy =  0.8866666666666667
```

In [121…

```
#Permuting different values of eta to see model's response
learning_rates = np.linspace(0.01, 1.0, 100)     #range() sequence only works for integer values
max_accuracy=0
best_eta = 0
for eta in learning_rates:              #loop through 100 values to find best Eta
    model.fit(X_train, y_train, eta)
    yhat = model.predict(X_test)
    accuracy = np.sum(yhat==y_test)/y_test.shape[0]
    if accuracy>max_accuracy:
        max_accuracy = accuracy
        best_eta = eta

print(f"Best Eta: {best_eta}, Accuracy = ", max_accuracy)
```

```
Best Eta: 0.92, Accuracy =  0.9133333333333333
```

1. Reference: https://engineering.purdue.edu/kak/Tutorials/AdaBoost.pdf
2. https://scikit-learn.org/stable/modules/generated/sklearn.datasets.make_classification.html

In [124…

```
class stump:
    def __init__(self):
        self.polarity = 1
        self.feature_index = None
        self.threshold = None
        self.alpha = None

class adaptiveBoostwithStump:
    def __init__(self, S=20):
        self.S=S            #Number of classifiers
        self.models=[]
        for i in range(self.S):
            self.models.append( stump() )  #Declaring models

        self.alpha = np.zeros(self.S) #alpha values for all classifiers


    def fit(self, X, y, eta=0.5):              # <-- eta is the learning rate used to update alpha values

        sample_weights = np.full(X.shape[0], 1/X.shape[0]) #Weight vector of size (m= number of samples),
                                                           #initialized to 1/m for each element

        for idx, model in enumerate(self.models):

            minimum_error= np.inf  #setting Minimum error to infinity for later evaluation

            subsample_idx = sample_weights.argsort()       #Get sorted weight indexes lowest to highest
            subsample_idx = subsample_idx[:-int(len(subsample_idx)/4)]   # sampling for 1/4th of total sampl
            xt = X[subsample_idx]       #subsampling for training model based on large sample_weights

            for feature in range(X.shape[1]):    #Repeat for every feature in training set

                #Ordered list of samples for each feature
                unique_feature_values_sorted = np.sort(np.unique(xt[:, feature])) #Choosing only samples wit

                #  ( [1, 2, 3, 4] + [2, 3, 4, 5])/2 = [1.5, 2.5, 3.5, 4.5 ] <-- example to explain code line
                thresholds = (unique_feature_values_sorted[:-1] + unique_feature_values_sorted[1:])/2

                for threshold in thresholds:        #Stepping through feature threshold values of our subsam
                    for polarity in [1, -1]:           #checking for error for both polarities
                        yhat = np.ones(X.shape[0]) #initialize prediction for all training samples to 1

                        #X[features]<threshold --> y = -1 --> classify as -1
                        #-X[features] < -threshold --> y = -1 --> classify as -1
                        yhat[ polarity * X[:, feature] < polarity * threshold ] = -1 #checking over all trai

                        #Misclassification Rate (Type 1 for polarity = 1 and Type 2 for polarity = -1)
                        error = sample_weights[yhat!=y].sum()   #Adding weighted contribution made by each s

                        if error < minimum_error:
                            minimum_error = error           #Defining model with minimum misclassification ra
                            model.polarity = polarity       #Recording polarity for minimum error for later p
                            model.threshold = threshold     #Recording threshold for minimum error for later 
                            model.feature_index = feature   #Recording feature index for minimum error for la

            #Set alpha = voting power for model
            model.alpha = eta * np.log((1-minimum_error)/(minimum_error+0.00000001))
            self.alpha[idx] = model.alpha

            #Update weights assigned to individual training samples
            #These updated weights are used to train the next classifier
            sample_weights = sample_weights * np.exp(-self.alpha[idx]* y * yhat)
            sample_weights = sample_weights/np.sum(sample_weights)  #Normalize sample_weights
            #print(f"max weight: ", sample_weights[sample_weights.argmax()])

    def predict(self, X_test):
        #Each classifier gets a corresponding weightage as per pre-calculated alpha
        yhat=0
        for model in self.models:
            predictions = np.ones(X_test.shape[0])  #initialize to 1
            #Predicting using our logic previously employed for calculating misclassification error
            predictions[model.polarity * X_test[:, model.feature_index] < model.polarity * model.threshold] 
            yhat = yhat + model.alpha * predictions
        yhat = np.sign(yhat)
        return yhat
```

In [132…

```
#Make classification data
X, y = make_classification(n_samples=500, random_state = 100)

#Replace y=0 with y=-1 (same as with SVM)
y[y==0] = -1

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.3, random_state = 50)

model = adaptiveBoostwithStump(S=10)
model.fit(X_train, y_train, eta=0.5)
yhat = model.predict(X_test)
print("Accuracy = ", np.sum(yhat==y_test)/y_test.shape[0])
```

```
Accuracy =  0.68
```

In [133…

```
#Permuting different values of S & eta to see model's response
learning_rates = np.linspace(0.1, 1.0, 5)      #range() sequence only works for integer values
max_accuracy=0
best_eta = 0
for s in [1, 3, 5, 6, 8, 10, 13, 15, 17, 20]:
    model = adaptiveBoostwithStump(S=s)
    for eta in learning_rates:                 #loop through 100 values to find best Eta
        model.fit(X_train, y_train, eta)
        yhat = model.predict(X_test)
        accuracy = np.sum(yhat==y_test)/y_test.shape[0]
        print(f"S: {s}, Eta: {eta}, Accuracy = ", accuracy)

        ## HIGHEST ACCURACY ACHIEVED AT S=5, eta=0.1, accuracy = 0.9066666
```

```
S: 1, Eta: 0.1, Accuracy =  0.8933333333333333
S: 1, Eta: 0.325, Accuracy =  0.8933333333333333
S: 1, Eta: 0.55, Accuracy =  0.8933333333333333
S: 1, Eta: 0.775, Accuracy =  0.8933333333333333
S: 1, Eta: 1.0, Accuracy =  0.8933333333333333
S: 3, Eta: 0.1, Accuracy =  0.9
S: 3, Eta: 0.325, Accuracy =  0.8733333333333333
S: 3, Eta: 0.55, Accuracy =  0.68
S: 3, Eta: 0.775, Accuracy =  0.68
S: 3, Eta: 1.0, Accuracy =  0.68
S: 5, Eta: 0.1, Accuracy =  0.9066666666666666
S: 5, Eta: 0.325, Accuracy =  0.68
S: 5, Eta: 0.55, Accuracy =  0.68
S: 5, Eta: 0.775, Accuracy =  0.68
S: 5, Eta: 1.0, Accuracy =  0.68
S: 6, Eta: 0.1, Accuracy =  0.8866666666666667
S: 6, Eta: 0.325, Accuracy =  0.68
S: 6, Eta: 0.55, Accuracy =  0.68
S: 6, Eta: 0.775, Accuracy =  0.68
S: 6, Eta: 1.0, Accuracy =  0.68
S: 8, Eta: 0.1, Accuracy =  0.8733333333333333
S: 8, Eta: 0.325, Accuracy =  0.68
S: 8, Eta: 0.55, Accuracy =  0.68
S: 8, Eta: 0.775, Accuracy =  0.68
S: 8, Eta: 1.0, Accuracy =  0.68
S: 10, Eta: 0.1, Accuracy =  0.68
S: 10, Eta: 0.325, Accuracy =  0.68
S: 10, Eta: 0.55, Accuracy =  0.68
S: 10, Eta: 0.775, Accuracy =  0.68
S: 10, Eta: 1.0, Accuracy =  0.68
S: 13, Eta: 0.1, Accuracy =  0.68
S: 13, Eta: 0.325, Accuracy =  0.68
S: 13, Eta: 0.55, Accuracy =  0.68
S: 13, Eta: 0.775, Accuracy =  0.68
S: 13, Eta: 1.0, Accuracy =  0.68
S: 15, Eta: 0.1, Accuracy =  0.68
S: 15, Eta: 0.325, Accuracy =  0.68
S: 15, Eta: 0.55, Accuracy =  0.68
S: 15, Eta: 0.775, Accuracy =  0.68
S: 15, Eta: 1.0, Accuracy =  0.68
S: 17, Eta: 0.1, Accuracy =  0.68
S: 17, Eta: 0.325, Accuracy =  0.68
S: 17, Eta: 0.55, Accuracy =  0.68
S: 17, Eta: 0.775, Accuracy =  0.68
S: 17, Eta: 1.0, Accuracy =  0.68
S: 20, Eta: 0.1, Accuracy =  0.68
S: 20, Eta: 0.325, Accuracy =  0.68
S: 20, Eta: 0.55, Accuracy =  0.68
S: 20, Eta: 0.775, Accuracy =  0.68
S: 20, Eta: 1.0, Accuracy =  0.68
```

In [134…

```
#Make classification data with 3 informative features
X, y = make_classification(n_samples=500, n_informative=3, random_state = 100)

#Replace y=0 with y=-1 (same as with SVM)
y[y==0] = -1

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.3, random_state = 50)

model = adaptiveBoostwithStump(S=10)
model.fit(X_train, y_train, eta=0.5)
yhat = model.predict(X_test)
print("Accuracy = ", np.sum(yhat==y_test)/y_test.shape[0])
```

```
Accuracy =  0.74
```

In [135…

```
#Permuting different values of S & eta to see model's response
learning_rates = np.linspace(0.1, 1.0, 5)      #range() sequence only works for integer values
max_accuracy=0
best_eta = 0
for s in [1, 3, 5, 6, 8, 10, 13, 15, 17, 20]:
    model = adaptiveBoostwithStump(S=s)
    for eta in learning_rates:                 #loop through 100 values to find best Eta
        model.fit(X_train, y_train, eta)
        yhat = model.predict(X_test)
        accuracy = np.sum(yhat==y_test)/y_test.shape[0]
        print(f"S: {s}, Eta: {eta}, Accuracy = ", accuracy)
```

```
S: 1, Eta: 0.1, Accuracy =  0.7933333333333333
S: 1, Eta: 0.325, Accuracy =  0.7933333333333333
S: 1, Eta: 0.55, Accuracy =  0.7933333333333333
S: 1, Eta: 0.775, Accuracy =  0.7933333333333333
S: 1, Eta: 1.0, Accuracy =  0.7933333333333333
S: 3, Eta: 0.1, Accuracy =  0.7933333333333333
S: 3, Eta: 0.325, Accuracy =  0.7933333333333333
S: 3, Eta: 0.55, Accuracy =  0.74
S: 3, Eta: 0.775, Accuracy =  0.74
S: 3, Eta: 1.0, Accuracy =  0.74
S: 5, Eta: 0.1, Accuracy =  0.7933333333333333
S: 5, Eta: 0.325, Accuracy =  0.74
S: 5, Eta: 0.55, Accuracy =  0.74
S: 5, Eta: 0.775, Accuracy =  0.74
S: 5, Eta: 1.0, Accuracy =  0.74
S: 6, Eta: 0.1, Accuracy =  0.7933333333333333
S: 6, Eta: 0.325, Accuracy =  0.74
S: 6, Eta: 0.55, Accuracy =  0.74
S: 6, Eta: 0.775, Accuracy =  0.74
S: 6, Eta: 1.0, Accuracy =  0.74
S: 8, Eta: 0.1, Accuracy =  0.7866666666666666
S: 8, Eta: 0.325, Accuracy =  0.74
S: 8, Eta: 0.55, Accuracy =  0.74
S: 8, Eta: 0.775, Accuracy =  0.74
S: 8, Eta: 1.0, Accuracy =  0.74
S: 10, Eta: 0.1, Accuracy =  0.78
S: 10, Eta: 0.325, Accuracy =  0.74
S: 10, Eta: 0.55, Accuracy =  0.74
S: 10, Eta: 0.775, Accuracy =  0.74
S: 10, Eta: 1.0, Accuracy =  0.74
S: 13, Eta: 0.1, Accuracy =  0.74
S: 13, Eta: 0.325, Accuracy =  0.74
S: 13, Eta: 0.55, Accuracy =  0.74
S: 13, Eta: 0.775, Accuracy =  0.74
S: 13, Eta: 1.0, Accuracy =  0.74
S: 15, Eta: 0.1, Accuracy =  0.74
S: 15, Eta: 0.325, Accuracy =  0.74
S: 15, Eta: 0.55, Accuracy =  0.74
S: 15, Eta: 0.775, Accuracy =  0.74
S: 15, Eta: 1.0, Accuracy =  0.74
S: 17, Eta: 0.1, Accuracy =  0.74
S: 17, Eta: 0.325, Accuracy =  0.74
S: 17, Eta: 0.55, Accuracy =  0.74
S: 17, Eta: 0.775, Accuracy =  0.74
S: 17, Eta: 1.0, Accuracy =  0.74
S: 20, Eta: 0.1, Accuracy =  0.74
S: 20, Eta: 0.325, Accuracy =  0.74
S: 20, Eta: 0.55, Accuracy =  0.74
S: 20, Eta: 0.775, Accuracy =  0.74
S: 20, Eta: 1.0, Accuracy =  0.74
```