AdaBoost (Adaptive Boosting)

AdaBoost (Adaptive Boosting) is a powerful ensemble learning technique that combines multiple weak learners to create a strong classifier. It iteratively adjusts the weights of misclassified examples to focus more on difficult cases, improving performance.

Steps in AdaBoost

1. **Initialize Weights**:
   Every sample in the training data is assigned equal weight.
   - Formula: `w_i = \frac{1}{N}` , where (N) is the total number of samples.
2. **Train Weak Learner**:
   A weak learner, like a decision stump, is trained on the weighted data, and its classification error is calculated.
   - Formula: `\epsilon = \sum_{i=1}^{N} w_i \times I(y_i \neq h(x_i))`, where (I) is the indicator function (1 for misclassified samples, 0 otherwise).
3. **Calculate Weak Learner's Weight**:
   The weight (alpha) of the weak learner is calculated based on its error (\epsilon).
   - Formula: `\alpha = \frac{1}{2} \ln \left(\frac{1 - \epsilon}{\epsilon}\right)`.
4. **Update Weights**:
   Increase the weights of misclassified samples, so that the next learner focuses on them.
   - Formula: `w_i = w_i \times \exp(\alpha \times I(y_i \neq h(x_i)))`. Normalize weights to ensure the sum is 1.
5. **Train Next Learner**:
   Repeat steps 2 to 4 for the next weak learner, which now focuses on the updated weights.
6. **Final Prediction**:
   Combine the weak learners using a weighted vote to make the final prediction.
   - Formula: `H(x) = \text{sign} \left( \sum_{m=1}^{M} \alpha_m h_m(x) \right)`,

where (\alpha_m) is the weight of the (m)-th weak learner, and (h_m(x)) is the prediction of the (m)-th weak learner.

Example

Let's assume a binary classification problem with 3 data points, initialized with equal weights:

1. A weak learner misclassifies one data point, yielding an error (\epsilon).
2. The weak learner's weight is calculated using the error.
3. The sample weights are updated, assigning higher weight to the misclassified point.
4. The next weak learner focuses on the misclassified point.
5. This process continues, and in the end, all learners are combined for the final prediction.

Real-World Applications

- **Face Detection**: AdaBoost is used in the Viola-Jones algorithm for detecting faces in images.
- **Fraud Detection**: Helps identify fraudulent transactions.
- **Customer Churn Prediction**: Classifies customers likely to churn based on historical data.

Pros

- Can **improve accuracy** by focusing on difficult cases.
- Works well with **imbalanced data**.
- **No parameter tuning** for weak learners.

Cons

- **Sensitive to noisy data** as it focuses too much on misclassified points.
- Computationally **intensive for large datasets**.

Adaboost Mathematical Implementation

To ignore all warnings:

You can use the following code at the top of your notebook to suppress all warnings:

` import warnings

warnings.filterwarnings('ignore') `

In [1]:
```
import warnings
warnings.filterwarnings('ignore')
```

In [2]:
```
import pandas as pd
import numpy as np
from mlxtend.plotting import plot_decision_regions
```

In [3]:
```
df = pd.DataFrame()
```

In [4]:
```
df['X1'] = [1,2,3,4,5,6,6,7,9,9]
df['X2'] = [5,3,6,8,1,9,5,8,9,2]
df['label'] = [1,1,0,1,0,1,0,1,0,0]
```

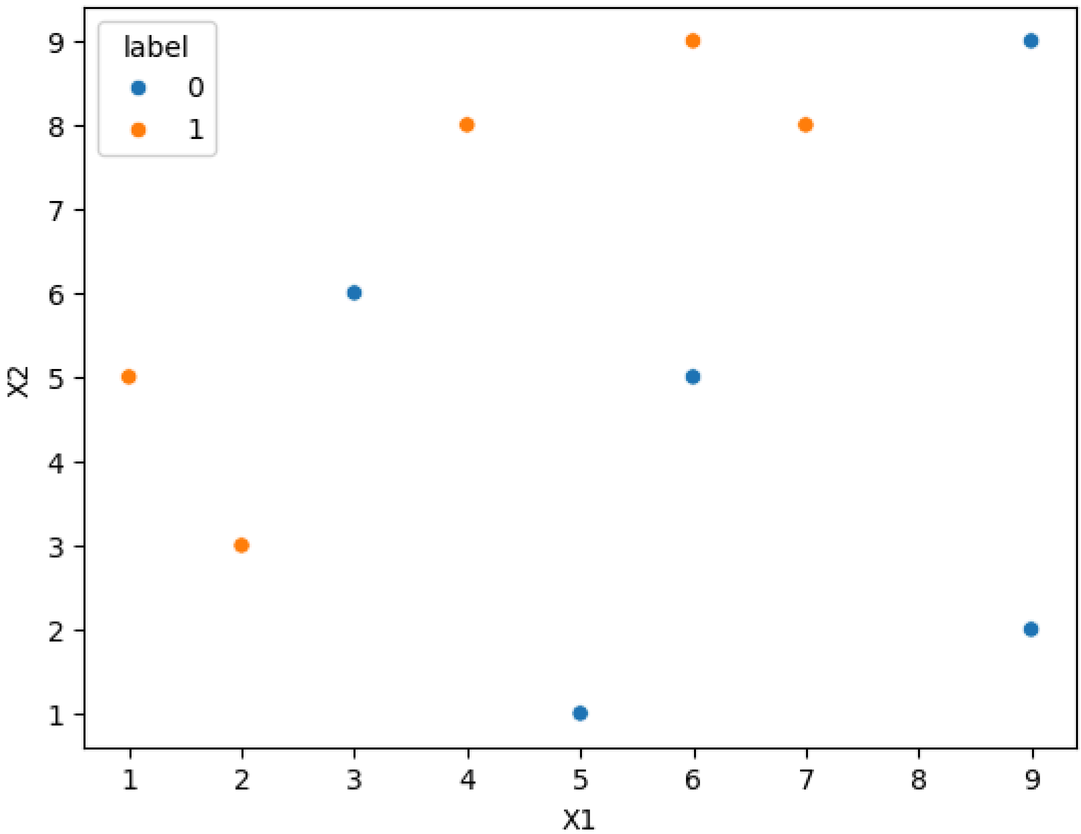
In [5]:
```
df
```

Out[5]:

	X1	X2	label
0	1	5	1
1	2	3	1
2	3	6	0
3	4	8	1
4	5	1	0
5	6	9	1
6	6	5	0
7	7	8	1
8	9	9	0
9	9	2	0

In [6]:
```
import seaborn as sns
sns.scatterplot(x=df['X1'],y=df['X2'],hue=df['label'])
```

Out[6]: <Axes: xlabel='X1', ylabel='X2'>



In [7]:
```
df.shape
```

Out[7]:
```
(10, 3)
```

In [8]:
```
df['weights'] = 1/df.shape[0]
```

In [9]:
```
df
```

Out[9]:

	X1	X2	label	weights
0	1	5	1	0.1
1	2	3	1	0.1
2	3	6	0	0.1
3	4	8	1	0.1
4	5	1	0	0.1
5	6	9	1	0.1
6	6	5	0	0.1
7	7	8	1	0.1
8	9	9	0	0.1
9	9	2	0	0.1

In [10]:
```
from sklearn.tree import DecisionTreeClassifier
```

In [11]:
```
dt1 = DecisionTreeClassifier(max_depth=1)
```

In [12]:
```python
X = df.iloc[:,0:2].values
y = df.iloc[:,2].values
```

In [13]:
```python
# Step 2 - Train 1st model
dt1.fit(X,y)
```

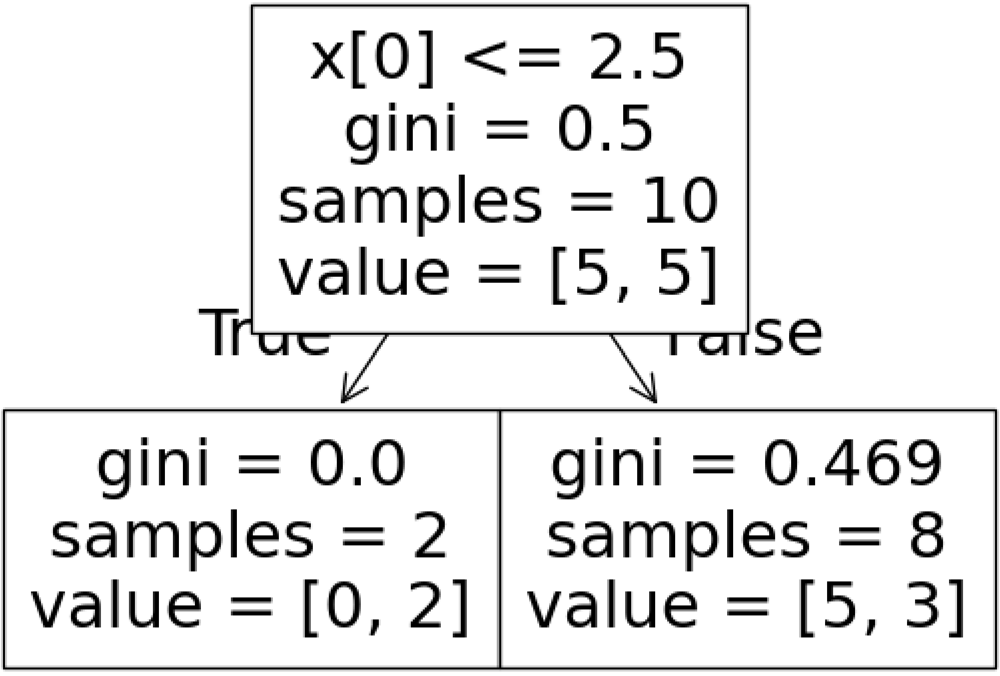
Out[13]:

▾ DecisionTreeClassifier

```
DecisionTreeClassifier(max_depth=1)
```

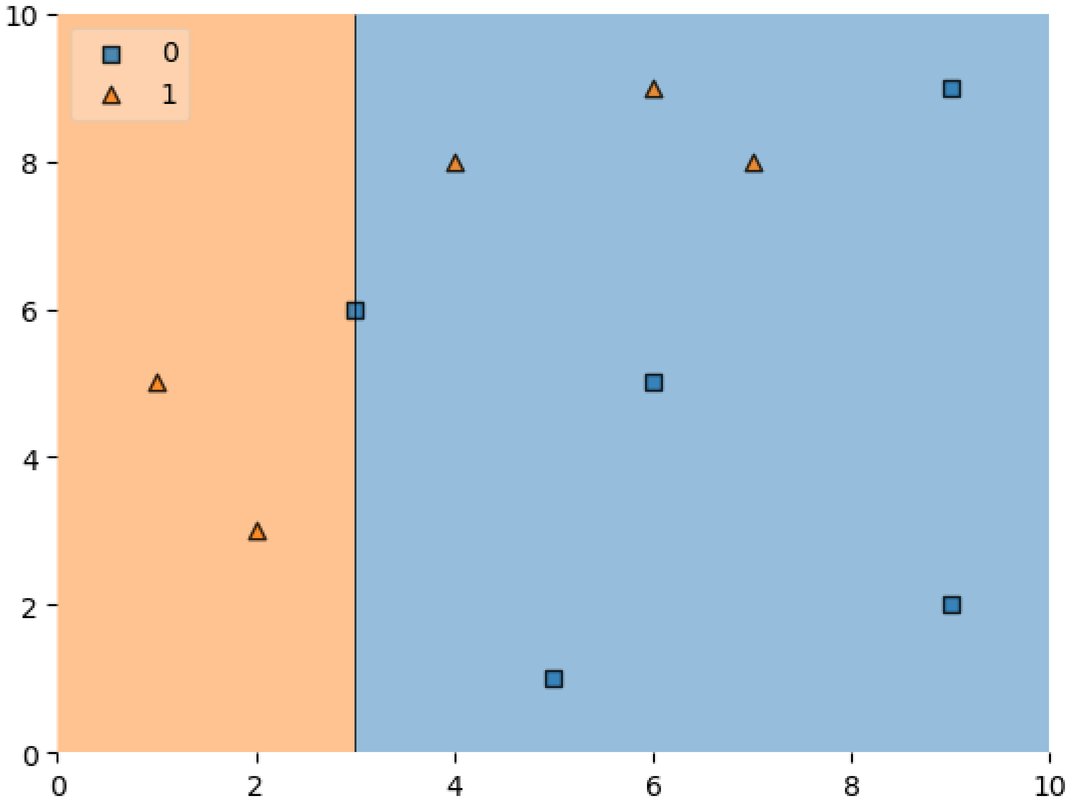
In [14]:
```python
from sklearn.tree import plot_tree
plot_tree(dt1)
```

Out[14]:
```
[Text(0.5, 0.75, 'x[0] <= 2.5\ngini = 0.5\nsamples = 10\nvalue = [5,
5]'),
 Text(0.25, 0.25, 'gini = 0.0\nsamples = 2\nvalue = [0, 2]'),
 Text(0.375, 0.5, 'True  '),
 Text(0.75, 0.25, 'gini = 0.469\nsamples = 8\nvalue = [5, 3]'),
 Text(0.625, 0.5, '  False')]
```



In [15]:
```python
plot_decision_regions(X, y, clf=dt1, legend=2)
```

Out[15]: <Axes: >



In [16]:
```
df['y_pred'] = dt1.predict(X)
```

In [17]:
```
df
```

Out[17]:

	X1	X2	label	weights	y_pred
0	1	5	1	0.1	1
1	2	3	1	0.1	1
2	3	6	0	0.1	0
3	4	8	1	0.1	0
4	5	1	0	0.1	0
5	6	9	1	0.1	0
6	6	5	0	0.1	0
7	7	8	1	0.1	0
8	9	9	0	0.1	0
9	9	2	0	0.1	0

In [18]:
```
def calculate_model_weight(error):

  return 0.5*np.log((1-error)/(error))
```

In [19]:
```
# Step 3 - calculate model weight
alpha1 = calculate_model_weight(0.3)
alpha1
```

Out[19]: 0.42364893019360184

In [20]:
```
# Step 4 - Update weights
def update_row_weights(row,alpha=0.423):
  if row['label'] == row['y_pred']:
    return row['weights'] * np.exp(-alpha)
  else:
    return row['weights'] * np.exp(alpha)
```

In [21]:
```
df['updated_weights'] = df.apply(update_row_weights,axis=1)
```

In [22]:
```
df
```

Out[22]:

	X1	X2	label	weights	y_pred	updated_weights
0	1	5	1	0.1	1	0.065508
1	2	3	1	0.1	1	0.065508
2	3	6	0	0.1	0	0.065508
3	4	8	1	0.1	0	0.152653
4	5	1	0	0.1	0	0.065508
5	6	9	1	0.1	0	0.152653
6	6	5	0	0.1	0	0.065508
7	7	8	1	0.1	0	0.152653
8	9	9	0	0.1	0	0.065508
9	9	2	0	0.1	0	0.065508

In [23]:
```
df['updated_weights'].sum()
```

Out[23]:
```
0.9165153319682015
```

In [24]:
```
df['nomalized_weights'] = df['updated_weights']/df['updated_weights'].sum
```

In [25]:
```
df
```

Out[25]:

	X1	X2	label	weights	y_pred	updated_weights	nomalized_weights
0	1	5	1	0.1	1	0.065508	0.071475
1	2	3	1	0.1	1	0.065508	0.071475
2	3	6	0	0.1	0	0.065508	0.071475
3	4	8	1	0.1	0	0.152653	0.166559
4	5	1	0	0.1	0	0.065508	0.071475
5	6	9	1	0.1	0	0.152653	0.166559
6	6	5	0	0.1	0	0.065508	0.071475
7	7	8	1	0.1	0	0.152653	0.166559
8	9	9	0	0.1	0	0.065508	0.071475
9	9	2	0	0.1	0	0.065508	0.071475

In [26]:
```
df['nomalized_weights'].sum()
```

Out[26]: 1.0

In [27]:
```
df['cumsum_upper'] = np.cumsum(df['nomalized_weights'])
```

In [28]:
```
df['cumsum_lower'] = df['cumsum_upper'] - df['nomalized_weights']
```

In [29]:
```
df[['X1','X2','label','weights','y_pred','updated_weights','cumsum_lower'
```

Out[29]:

	X1	X2	label	weights	y_pred	updated_weights	cumsum_lower	cumsum_upp
0	1	5	1	0.1	1	0.065508	0.000000	0.0714
1	2	3	1	0.1	1	0.065508	0.071475	0.1429
2	3	6	0	0.1	0	0.065508	0.142950	0.2144
3	4	8	1	0.1	0	0.152653	0.214425	0.3809
4	5	1	0	0.1	0	0.065508	0.380983	0.4524
5	6	9	1	0.1	0	0.152653	0.452458	0.6190
6	6	5	0	0.1	0	0.065508	0.619017	0.6904
7	7	8	1	0.1	0	0.152653	0.690492	0.8570
8	9	9	0	0.1	0	0.065508	0.857050	0.9285
9	9	2	0	0.1	0	0.065508	0.928525	1.0000

In [30]:
```
def create_new_dataset(df):

  indices = []

  for i in range(df.shape[0]):
    a = np.random.random()
    for index,row in df.iterrows():
      if row['cumsum_upper'] > a and a > row['cumsum_lower']:
```

```
        indices.append(index)
    return indices
```

In [31]:
```
index_values = create_new_dataset(df)

index_values
```

Out[31]: [0, 5, 5, 7, 7, 1, 3, 3, 8, 5]

In [32]:
```
second_df = df.iloc[index_values,[0,1,2,3]]
```

In [33]:
```
second_df
```

Out[33]:

	X1	X2	label	weights
0	1	5	1	0.1
5	6	9	1	0.1
5	6	9	1	0.1
7	7	8	1	0.1
7	7	8	1	0.1
1	2	3	1	0.1
3	4	8	1	0.1
3	4	8	1	0.1
8	9	9	0	0.1
5	6	9	1	0.1

In [34]:
```
dt2 = DecisionTreeClassifier(max_depth=1)
```

In [35]:
```
X = second_df.iloc[:,0:2].values
y = second_df.iloc[:,2].values
```

In [36]:
```
dt2.fit(X,y)
```

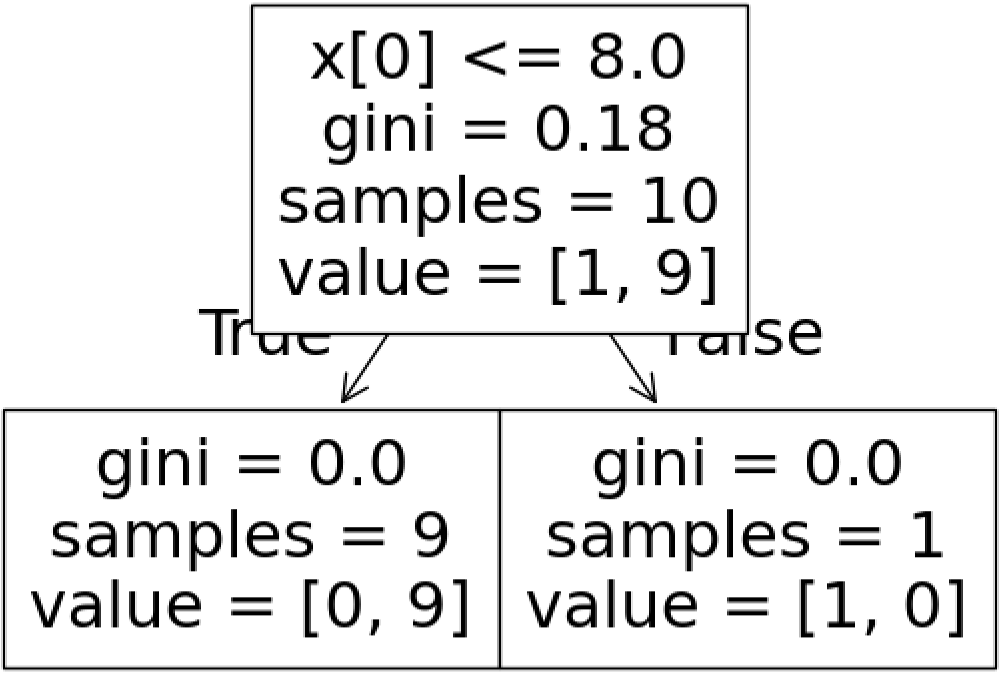
Out[36]:

▾ DecisionTreeClassifier

DecisionTreeClassifier(max_depth=1)

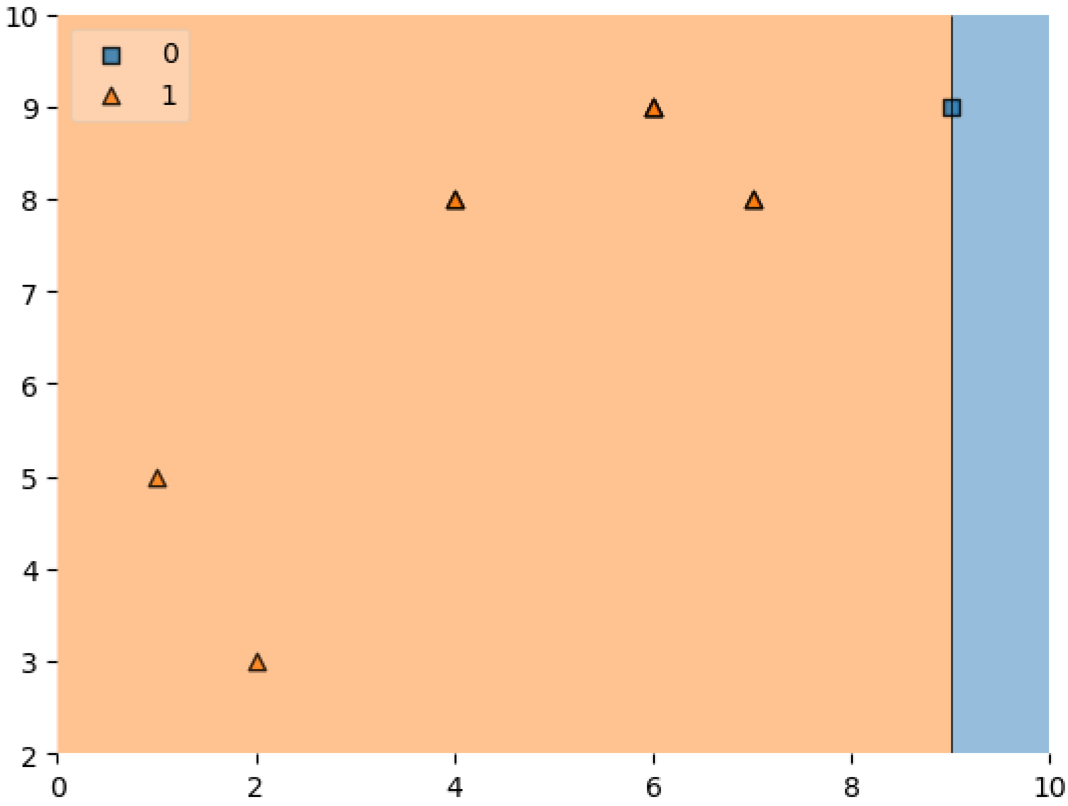
In [37]:
```
plot_tree(dt2)
```

Out[37]:
```
[Text(0.5, 0.75, 'x[0] <= 8.0\ngini = 0.18\nsamples = 10\nvalue = [1,
9]'),
 Text(0.25, 0.25, 'gini = 0.0\nsamples = 9\nvalue = [0, 9]'),
 Text(0.375, 0.5, 'True  '),
 Text(0.75, 0.25, 'gini = 0.0\nsamples = 1\nvalue = [1, 0]'),
 Text(0.625, 0.5, '  False')]
```



In [38]:
```
plot_decision_regions(X, y, clf=dt2, legend=2)
```

Out[38]: <Axes: >



In [39]:
```
second_df['y_pred'] = dt2.predict(X)
```

In [40]:
```
second_df
```

Out[40]:

	X1	X2	label	weights	y_pred
0	1	5	1	0.1	1
5	6	9	1	0.1	1
5	6	9	1	0.1	1
7	7	8	1	0.1	1
7	7	8	1	0.1	1
1	2	3	1	0.1	1
3	4	8	1	0.1	1
3	4	8	1	0.1	1
8	9	9	0	0.1	0
5	6	9	1	0.1	1

In [41]:
```
alpha2 = calculate_model_weight(0.1)
```

In [42]:
```
alpha2
```

Out[42]: 1.0986122886681098

In [43]:
```
# Step 4 - Update weights
def update_row_weights(row,alpha=1.09):
  if row['label'] == row['y_pred']:
    return row['weights'] * np.exp(-alpha)
  else:
    return row['weights'] * np.exp(alpha)
```

In [44]:
```
second_df['updated_weights'] = second_df.apply(update_row_weights,axis=1)
```

In [45]:
```
second_df
```

Out[45]:

	X1	X2	label	weights	y_pred	updated_weights
0	1	5	1	0.1	1	0.033622
5	6	9	1	0.1	1	0.033622
5	6	9	1	0.1	1	0.033622
7	7	8	1	0.1	1	0.033622
7	7	8	1	0.1	1	0.033622
1	2	3	1	0.1	1	0.033622
3	4	8	1	0.1	1	0.033622
3	4	8	1	0.1	1	0.033622
8	9	9	0	0.1	0	0.033622
5	6	9	1	0.1	1	0.033622

In [46]:
```
second_df['nomalized_weights'] = second_df['updated_weights']/second_df['
```

In [47]:
```
second_df
```

Out[47]:

	X1	X2	label	weights	y_pred	updated_weights	nomalized_weights
0	1	5	1	0.1	1	0.033622	0.1
5	6	9	1	0.1	1	0.033622	0.1
5	6	9	1	0.1	1	0.033622	0.1
7	7	8	1	0.1	1	0.033622	0.1
7	7	8	1	0.1	1	0.033622	0.1
1	2	3	1	0.1	1	0.033622	0.1
3	4	8	1	0.1	1	0.033622	0.1
3	4	8	1	0.1	1	0.033622	0.1
8	9	9	0	0.1	0	0.033622	0.1
5	6	9	1	0.1	1	0.033622	0.1

In [48]:
```
second_df['nomalized_weights'].sum()
```

Out[48]:
```
0.9999999999999999
```

In [49]:
```
second_df['cumsum_upper'] = np.cumsum(second_df['nomalized_weights'])
```

In [50]:
```
second_df['cumsum_lower'] = second_df['cumsum_upper'] - second_df['nomali
```

In [51]:
```
second_df[['X1','X2','label','weights','y_pred','nomalized_weights','cums
```

Out[51]:

	X1	X2	label	weights	y_pred	nomalized_weights	cumsum_lower	cumsum_u
0	1	5	1	0.1	1	0.1	0.0	
5	6	9	1	0.1	1	0.1	0.1	
5	6	9	1	0.1	1	0.1	0.2	
7	7	8	1	0.1	1	0.1	0.3	
7	7	8	1	0.1	1	0.1	0.4	
1	2	3	1	0.1	1	0.1	0.5	
3	4	8	1	0.1	1	0.1	0.6	
3	4	8	1	0.1	1	0.1	0.7	
8	9	9	0	0.1	0	0.1	0.8	
5	6	9	1	0.1	1	0.1	0.9	

In [52]:
```
index_values = create_new_dataset(second_df)
```

In [53]:
```
third_df = second_df.iloc[index_values,[0,1,2,3]]
```

In [54]:
```
third_df
```

Out[54]:

	X1	X2	label	weights
5	6	9	1	0.1
5	6	9	1	0.1
1	2	3	1	0.1
7	7	8	1	0.1
1	2	3	1	0.1
3	4	8	1	0.1
7	7	8	1	0.1
0	1	5	1	0.1
8	9	9	0	0.1
7	7	8	1	0.1

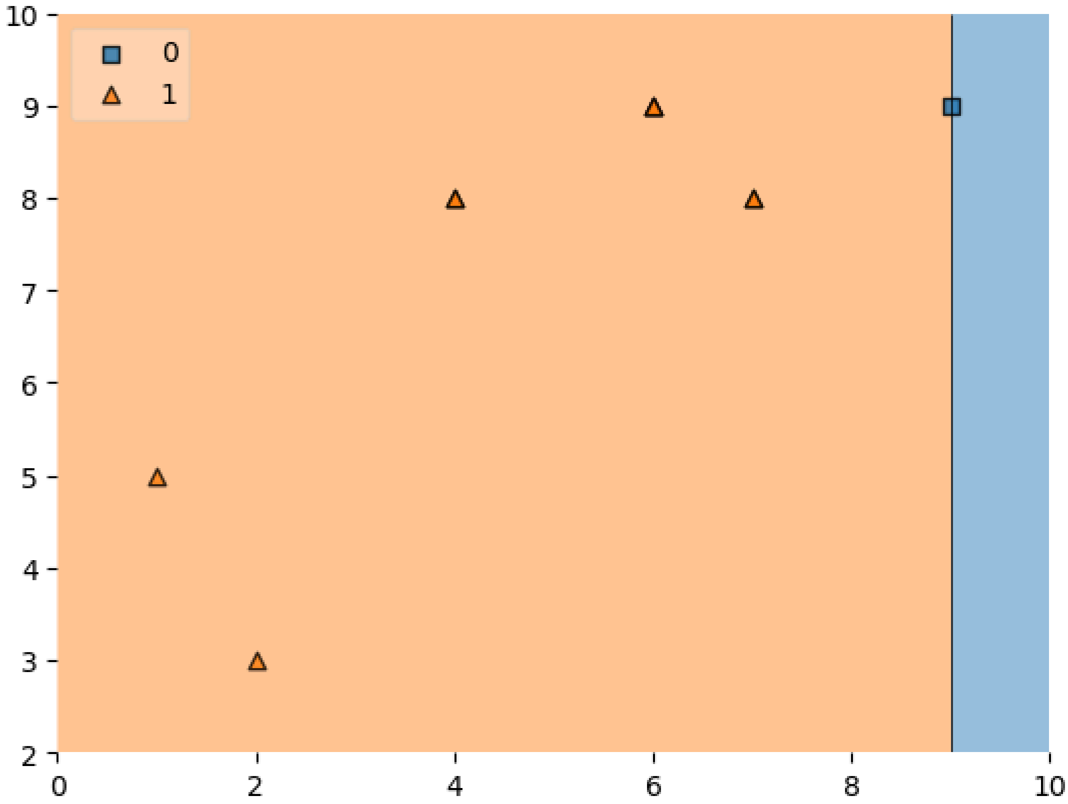
In [55]:
```
dt3 = DecisionTreeClassifier(max_depth=1)

X = second_df.iloc[:,0:2].values
y = second_df.iloc[:,2].values

dt3.fit(X,y)
```

Out[55]:

```
▾ DecisionTreeClassifier
DecisionTreeClassifier(max_depth=1)
```

In [56]:
```
plot_decision_regions(X, y, clf=dt3, legend=2)
```

Out[56]: <Axes: >



In [57]:
```
third_df['y_pred'] = dt3.predict(X)
```

In [58]:
```
third_df
```

Out[58]:

	X1	X2	label	weights	y_pred
5	6	9	1	0.1	1
5	6	9	1	0.1	1
1	2	3	1	0.1	1
7	7	8	1	0.1	1
1	2	3	1	0.1	1
3	4	8	1	0.1	1
7	7	8	1	0.1	1
0	1	5	1	0.1	1
8	9	9	0	0.1	0
7	7	8	1	0.1	1

In [59]:
```
alpha3 = calculate_model_weight(0.7)
alpha3
```

Out[59]: -0.4236489301936017

In [60]:
```
print(alpha1,alpha2,alpha3)
```

0.42364893019360184 1.0986122886681098 -0.4236489301936017

Prediction

In [61]:
```
query = np.array([1,5]).reshape(1,2)
dt1.predict(query)
```
Out[61]:
```
array([1])
```
In [62]:
```
dt2.predict(query)
```
Out[62]:
```
array([1])
```
In [63]:
```
dt3.predict(query)
```
Out[63]:
```
array([1])
```
In [64]:
```
alpha1*1 + alpha2*(1) + alpha3*(1)
```
Out[64]:
```
1.09861228866811
```
In [65]:
```
np.sign(1.09)
```
Out[65]:
```
1.0
```
In [66]:
```
query = np.array([9,9]).reshape(1,2)
dt1.predict(query)
```
Out[66]:
```
array([0])
```
In [67]:
```
dt2.predict(query)
```
Out[67]:
```
array([0])
```
In [68]:
```
dt3.predict(query)
```
Out[68]:
```
array([0])
```
In [69]:
```
alpha1*(1) + alpha2*(-1) + alpha3*(-1)
```
Out[69]:
```
-0.2513144282809062
```
In [70]:
```
np.sign(-0.25)
```
Out[70]:
```
-1.0
```

Adaboost Hyperparameter Tuning

Import all the required frameworks

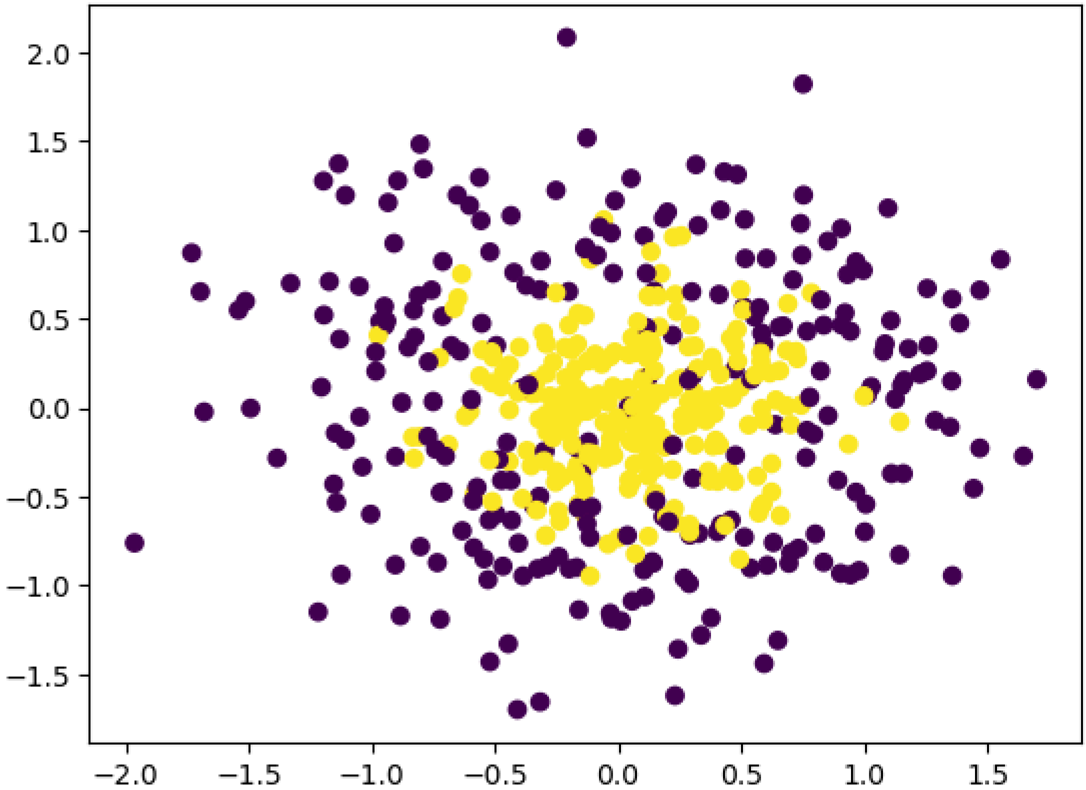
In [71]:
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
from mlxtend.plotting import plot_decision_regions
```

```
from sklearn.model_selection import train_test_split
from sklearn.datasets import make_circles
```

In [72]:
```
np.random.seed(42)
X, y = make_circles(n_samples=500, factor=0.1, noise=0.35, random_state=4
```

In [73]:
```
plt.scatter(X[:,0],X[:,1],c=y)
```

Out[73]: <matplotlib.collections.PathCollection at 0x157c2add0>



In [74]:
```
from sklearn.ensemble import AdaBoostClassifier
from sklearn.model_selection import cross_val_score

model = AdaBoostClassifier()

np.mean(cross_val_score(model,X,y,scoring='accuracy',cv=10))
```

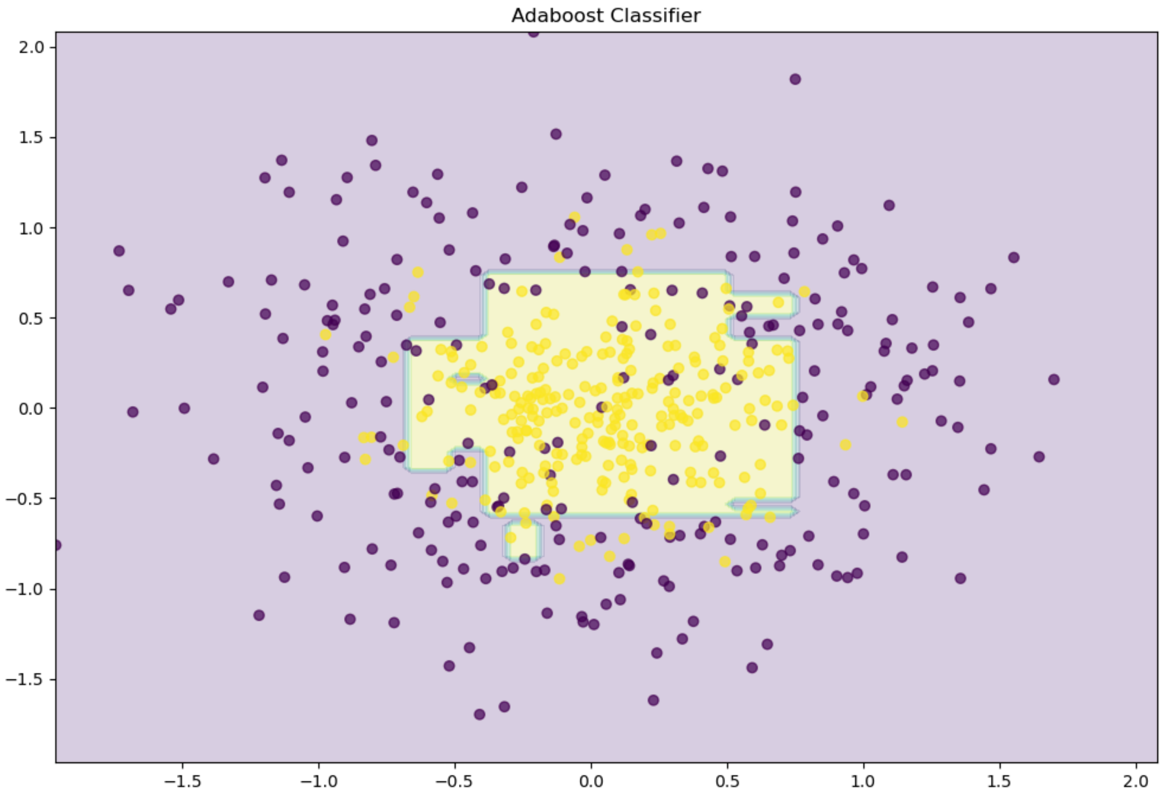
Out[74]: 0.786

In [75]:
```
model.fit(X,y)
```

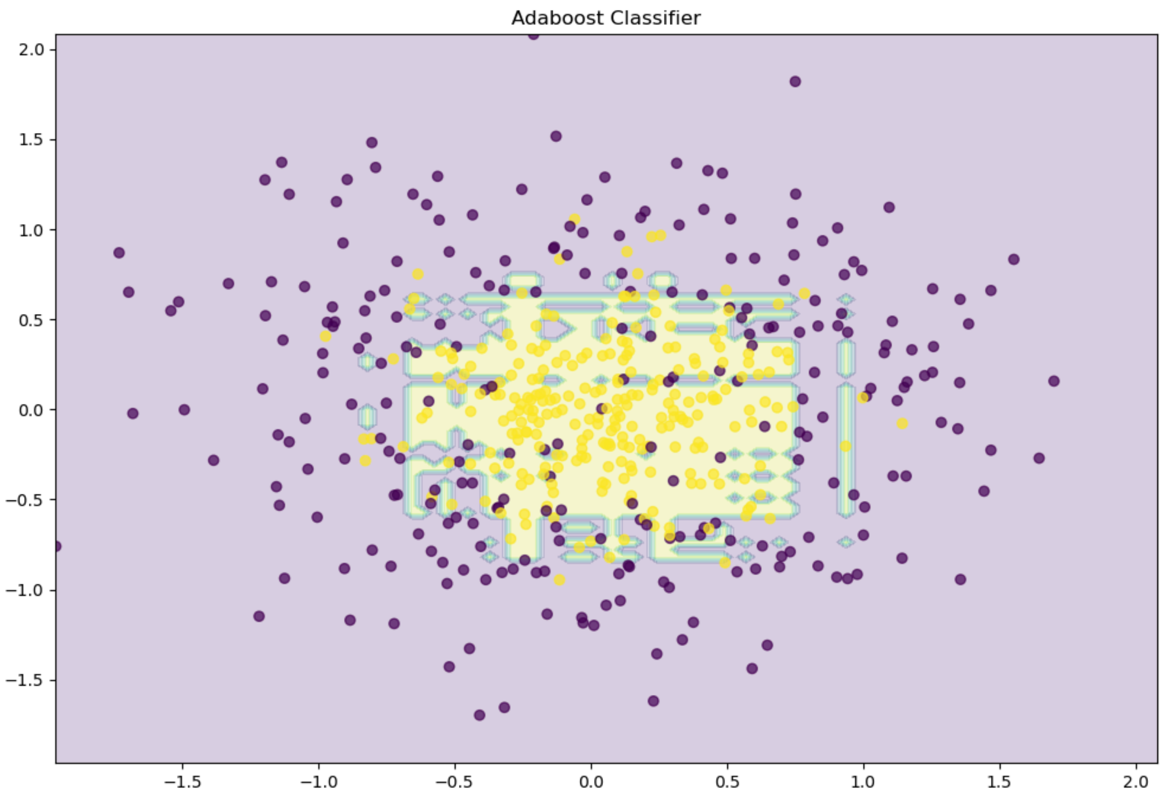
Out[75]:
```
▾ AdaBoostClassifier
AdaBoostClassifier()
```

In [76]:
```
def plot_decision_boundary(clf):
    plt.figure(figsize=(12, 8))
    x_range = np.linspace(X.min(), X.max(), 100)
    xx1, xx2 = np.meshgrid(x_range, x_range)
    y_hat = clf.predict(np.c_[xx1.ravel(), xx2.ravel()])
    y_hat = y_hat.reshape(xx1.shape)
    plt.contourf(xx1, xx2, y_hat, alpha=0.2)
    plt.scatter(X[:,0], X[:,1], c=y, cmap='viridis', alpha=.7)
```

```
    plt.title("Adaboost Classifier")
    plt.show()

plot_decision_boundary(model)
```

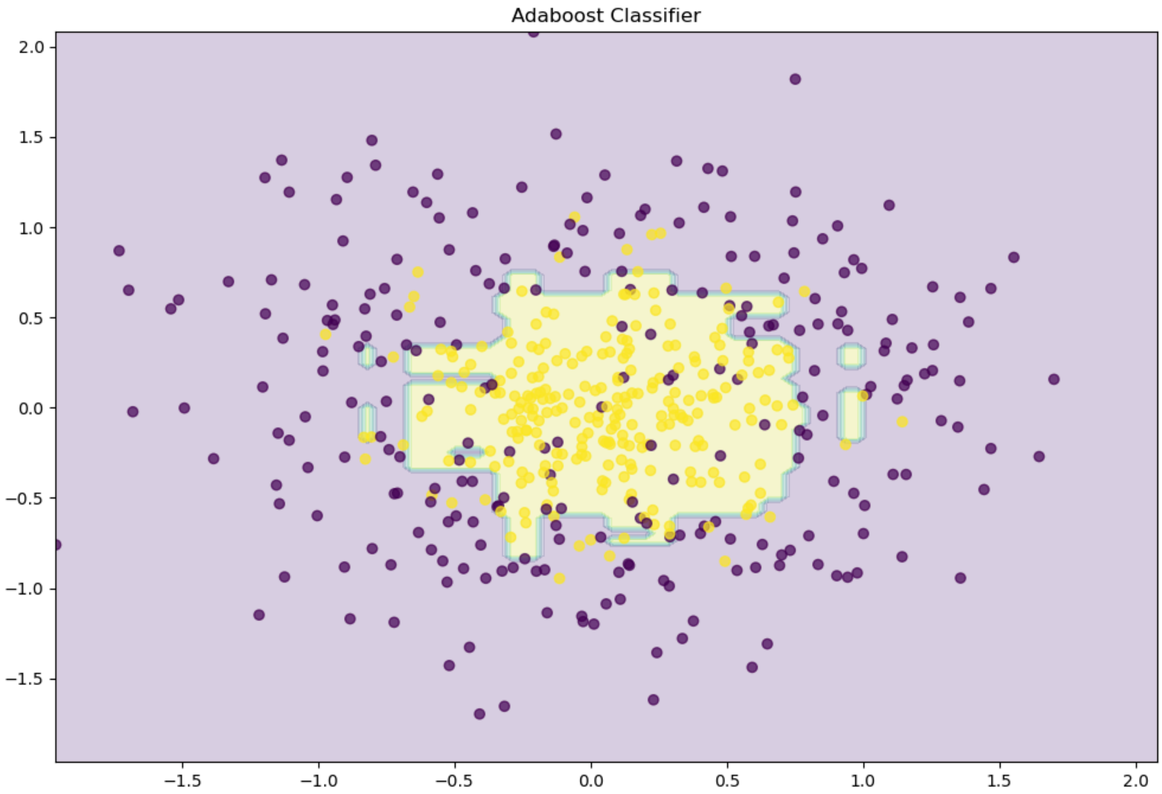


In [77]:
```
model_1 = AdaBoostClassifier(n_estimators=1500)
model_1.fit(X,y)
plot_decision_boundary(model_1)
```



In [78]:
```
model_1 = AdaBoostClassifier(n_estimators=1500, learning_rate=0.1)
model_1.fit(X,y)
```

```
plot_decision_boundary(model_1)
```



Optimize the parameters in AdaboostClassifier: GridSearchCV

In [ ]:
```
from sklearn.model_selection import GridSearchCV

grid = dict()
grid['n_estimators'] = [10, 50, 100, 500]
grid['learning_rate'] = [0.0001, 0.001, 0.01, 0.1, 1.0]
grid['algorithm'] = ['SAMME', 'SAMME.R']


grid_search = GridSearchCV(estimator=AdaBoostClassifier(), param_grid=gri
# execute the grid search
grid_result = grid_search.fit(X, y)
# summarize the best score and configuration
print("Best: %f using %s" % (grid_result.best_score_, grid_result.best_pa
```

Best: 0.832000 using {'algorithm': 'SAMME', 'learning_rate': 0.1, 'n_estimators': 500}