Méthodes d'ensemble

TP2 - Adaboost en pratique

LE Do Thanh Dat, YOU Borachhun

Exercice 1: Création du modèle Adaboost

1. Importer les librairies nécessaires

In [84]:
```
import matplotlib.pyplot as plt

%matplotlib inline

from visualization import plot_2d_data, plot_2d_classifier
from sklearn.tree import DecisionTreeClassifier
from sklearn.metrics import accuracy_score

import numpy as np
from sklearn.datasets import make_moons
import matplotlib.pyplot as plt
```

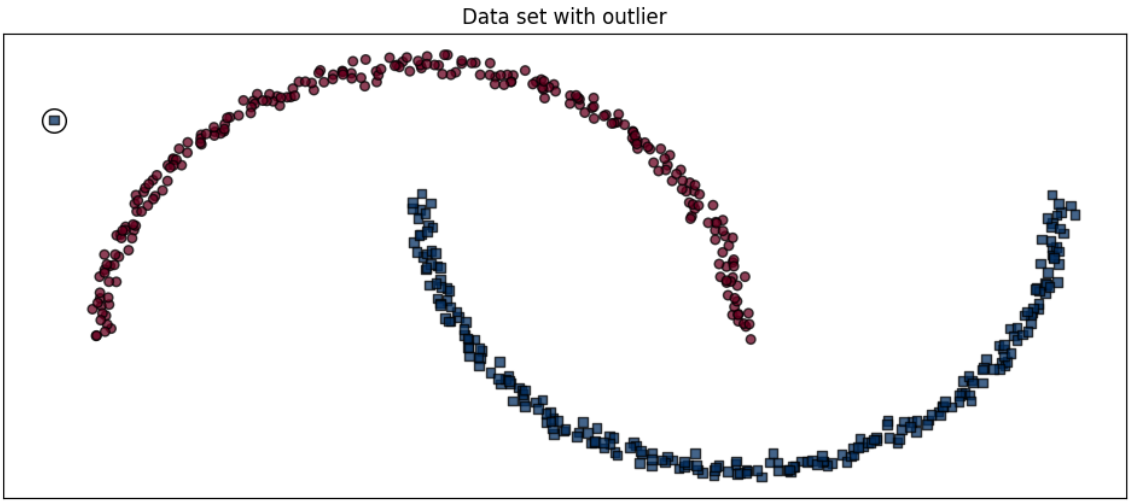
2. Importer les données avec la fonction make_moons, et ajouter un outliers

In [85]:
```
# Generate some data
X, y = make_moons(n_samples=500, noise=0.02)

# Add an outlier
X = np.concatenate((X, [[-1.15, 0.8]]))
y = np.concatenate((y, [1]))
# Convert to -1/+1 labels from 0/1 labels
y = 2*y-1
```

3. Visualiser les données

In [86]:
```
fig, ax = plt.subplots(nrows = 1, ncols = 1, figsize=(12,5))

# Plot the data set
ax.scatter([-1.15], [0.8], marker='o', s=200, c='w', edgecolors='k')
plot_2d_data(ax, X, y, alpha=0.75, s=30, title = 'Data set with outlier', colorm
ax.set_xticks([])
ax.set_yticks([])
```

Out[86]:
```
[]
```

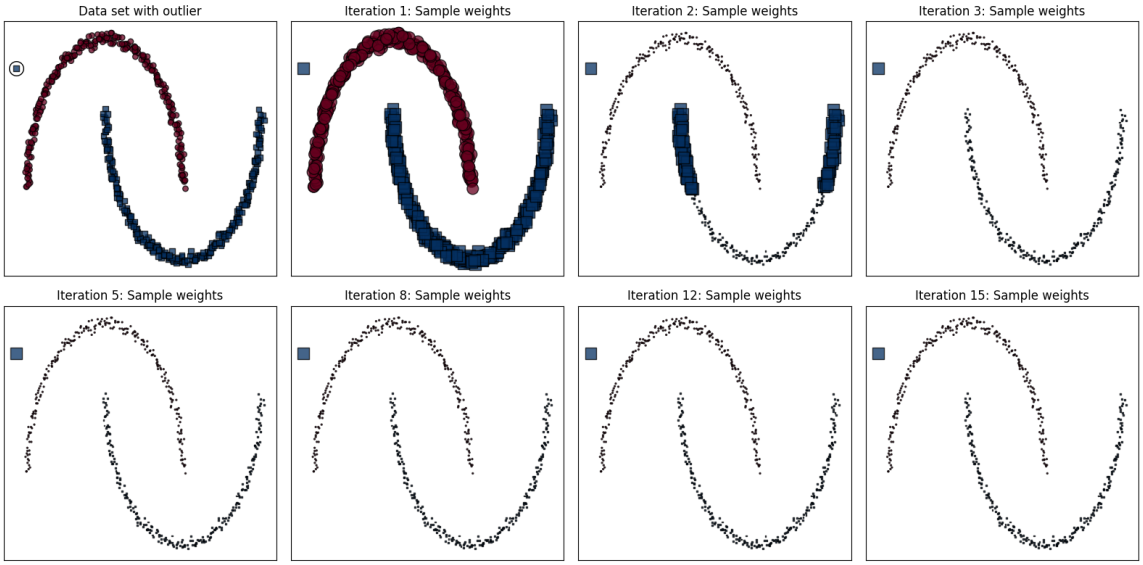


4, 5. Implémenter un Adaboost

In [87]:

```
# initialize variables
n_samples, n_features = X.shape
n_estimators = 20           # number of trees
D = np.ones((n_samples, ))  # Initialize sample weights
ensemble = []               # Initialize an empty ensemble

fig, ax = plt.subplots(nrows = 2, ncols = 4, figsize = (16, 8))

# Plot the data set
ax[0, 0].scatter([-1.15], [0.8], marker='o', s=200, c='w', edgecolors='k')
plot_2d_data(ax[0, 0], X, y, alpha=0.75, s=30, title='Data set with outlier', cc
ax[0, 0].set_xticks([])
ax[0, 0].set_yticks([])
axis_index = 0

for t in range(n_estimators):
    D = D / np.sum(D)    # Normalize the sample weights

    # Plot the training examples in different sizes proportional to their weight
    s = D / np.max(D)
    s[(0.00 <= s) & (s < 0.25)] = 2
    s[(0.25 <= s) & (s < 0.50)] = 16
    s[(0.50 <= s) & (s < 0.75)] = 64
    s[(0.75 <= s) & (s <= 1.00)] = 128

    if t in [0, 1, 2, 4, 7, 11, 14]:
        axis_index += 1
        r, c = np.divmod(axis_index, 4)
        title = 'Iteration {0}: Sample weights'.format(t + 1)
        plot_2d_data(ax[r, c], X, y, alpha=0.75, s=s, title=title, colormap='RdE
        ax[r, c].set_xticks([])
        ax[r, c].set_yticks([])

    ### Modelling
    h = DecisionTreeClassifier(max_depth=1, criterion = 'entropy')  #Initialize
    h.fit(X, y, sample_weight=D)  # Train a weak learner using sample weights
    ypred=h.predict(X)  # Predict using weak learner

    # Calculating error and new weights
    e = 1 - accuracy_score(y, ypred, sample_weight=D)  # Weighted error of the w
```

```
    a = 0.5 * np.log((1-e) / e)   # Weak learner weight

    # Update sample weights
    m = (y == ypred) * 1 + (y != ypred) * -1  # Identify correctly classified ar
    D *= np.exp(-a * m)    # Update the sample weights

    ensemble.append((a, h))   # Save the weighted weak hypothesis

fig.tight_layout()
```



Dans AdaBoost, les observations sont pondérées en fonction de l'erreur, les observations avec des erreurs plus élevées se voient attribuer des poids plus lourds à chaque itération.

Valeurs aberrantes avec des erreurs beaucoup plus élevées que les valeurs non aberrantes, donc AdaBoost augmentera leur poids. Lorsque le prochain apprenant faible est formé, il continue à mal classer les valeurs aberrantes et le poids attribué aux valeurs aberrantes continue d'augmenter à chaque itération.

Cela peut être vu sur le graphique ci-dessus, la valeur aberrante (le point carré) est de couleur plus foncée que les valeurs non aberrantes, en particulier à partir de la troisième itération. Cela signifie que la valeur aberrante est pondérée de plus en plus après chaque itération. Par conséquent, les valeurs aberrantes peuvent être mauvaises pour AdaBoost et le rendre moins robuste.

Exercice 2: Prunning

Objectif: Déterminer le nombre d'arbres optimaux et appliquer les critères d'élagage

1. Hyperparamètres à optimiser

In [88]:
```
from sklearn.model_selection import StratifiedKFold

#hyperparameters to optimize
n_estimator_steps, n_folds = 5, 10
number_of_stumps = np.arange(5, 50, n_estimator_steps)
splitter = StratifiedKFold(n_splits=n_folds, shuffle=True)
```

```
#vector for saving results
trn_err = np.zeros((len(number_of_stumps), n_folds))
val_err = np.zeros((len(number_of_stumps), n_folds))
```

2. Définition de l'apprenant faible

In [89]:
```
#Decision stump for the modelling
stump = DecisionTreeClassifier(max_depth=1)
```

La profondeur de cet arbre est 1 (c'est ce qu'on appelle une souche). Une souche (decision stump) est un arbre de décision comptant un nœud (la racine) directement connecté à ses feuilles. Il est avantageux d'avoir cette profondeur car dans AdaBoost, les arbres sont ajustés séquentiellement, chacun apprenant des erreurs de l'arbre précédent. Une fois qu'un ensemble boosté commence à sur-apprentissage les données, il continuera à sur-apprentissage après chaque étape.

Pour cette raison, il vaut la peine que les arbres individuels soient courts pour éviter de sur-apprentissage. Par conséquent, la souche est bénéfique dans AdaBoost.

3. Entrainement avec plusieurs combinaisons des hyperparamètres à tester

On teste la performance avec un learning_rate = {0.5, 0.1} et learning_rate = 1 (par défaut):

learning_rate = 0.5:

In [90]:
```
#vector for saving results
trn_err = np.zeros((len(number_of_stumps), n_folds))
val_err = np.zeros((len(number_of_stumps), n_folds))

from sklearn.ensemble import AdaBoostClassifier

#looping on several values of number of models and spliting
for i, n_stumps in enumerate(number_of_stumps):
    for j, (trn, val) in enumerate(splitter.split(X, y)):

        #Initialize Adaboost model
        model = AdaBoostClassifier(algorithm='SAMME', estimator=stump,
                                   n_estimators=n_stumps, learning_rate=0.5)
        model.fit(X[trn, :], y[trn])

        #calculating error
        trn_err[i, j] = 1-accuracy_score(y[trn], model.predict(X[trn, :]))
        val_err[i, j] = 1-accuracy_score(y[val], model.predict(X[val, :]))

trn_err = np.mean(trn_err, axis=1)
val_err = np.mean(val_err, axis=1)
print(trn_err)
print(val_err)
```

```
[0.07873269 0.08272382 0.0499054  0.05942942 0.01840453 0.02684602
 0.01796305 0.0046573  0.00266174]
[0.10784314 0.11184314 0.07576471 0.07596078 0.03       0.03768627
 0.024      0.014      0.00996078]
```

learning_rate = 0.1:

In [91]:
```
#vector for saving results
trn_err = np.zeros((len(number_of_stumps), n_folds))
val_err = np.zeros((len(number_of_stumps), n_folds))

#looping on several values of number of models and spliting
for i, n_stumps in enumerate(number_of_stumps):
    for j, (trn, val) in enumerate(splitter.split(X, y)):

        #Initialize Adaboost model
        model = AdaBoostClassifier(algorithm='SAMME', estimator=stump,
                                   n_estimators=n_stumps, learning_rate=0.1)
        model.fit(X[trn, :], y[trn])

        #calculating error
        trn_err[i, j] = 1-accuracy_score(y[trn], model.predict(X[trn, :]))
        val_err[i, j] = 1-accuracy_score(y[val], model.predict(X[val, :]))

trn_err = np.mean(trn_err, axis=1)
val_err = np.mean(val_err, axis=1)
print(trn_err)
print(val_err)
```

```
[0.16189998 0.15857453 0.09247943 0.08516285 0.08272136 0.08316679
 0.0827253  0.08738162 0.08737965]
[0.20368627 0.21360784 0.11972549 0.10572549 0.10768627 0.10180392
 0.10388235 0.09388235 0.09576471]
```

learning_rate = 1 (default):

In [92]:
```
#vector for saving results
trn_err = np.zeros((len(number_of_stumps), n_folds))
val_err = np.zeros((len(number_of_stumps), n_folds))

#looping on several values of number of models and spliting
for i, n_stumps in enumerate(number_of_stumps):
    for j, (trn, val) in enumerate(splitter.split(X, y)):

        #Initialize Adaboost model
        model = AdaBoostClassifier(algorithm='SAMME', estimator=stump,
                                   n_estimators=n_stumps, learning_rate=1)
        model.fit(X[trn, :], y[trn])

        #calculating error
        trn_err[i, j] = 1-accuracy_score(y[trn], model.predict(X[trn, :]))
        val_err[i, j] = 1-accuracy_score(y[val], model.predict(X[val, :]))

trn_err = np.mean(trn_err, axis=1)
val_err = np.mean(val_err, axis=1)
print(trn_err)
print(val_err)
```

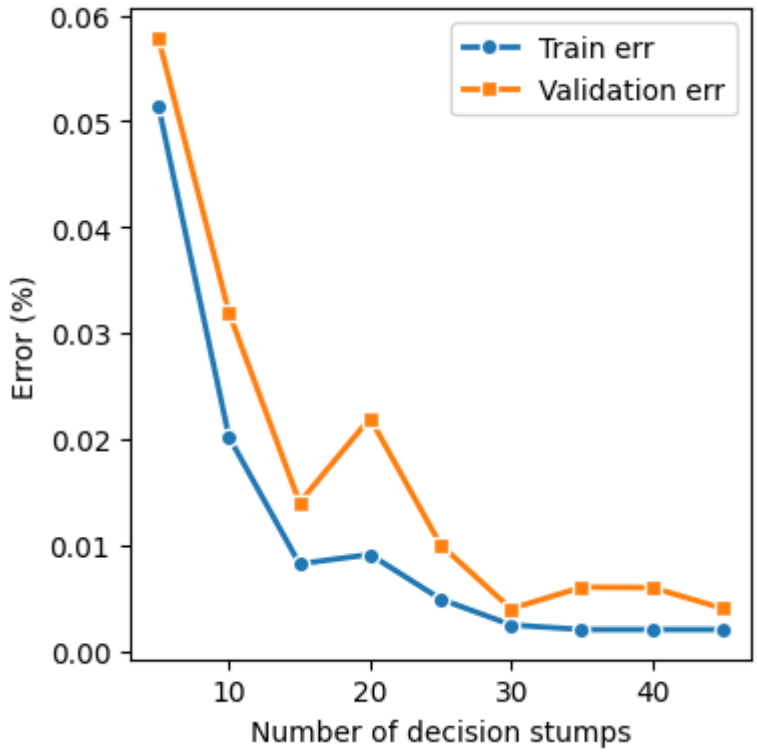
```
[0.05145849 0.02017985 0.00820695 0.0090914  0.00487854 0.00243902
 0.00199606 0.00199557 0.00199606]
[0.05784314 0.03192157 0.014      0.02196078 0.01       0.00396078
 0.006      0.00596078 0.004     ]
```

Concernant tous les résultats ci-dessus, les erreurs du modèle dans l'ensemble d'entraînement (training set) et l'ensemble de validation (validation set) augmentent lorsque on baisse **learning_rate**. On fait donc choisir **learning rate = 1.0** (par défaut)

4. Visualisation des résultats

On continue à déterminer le nombre d'arbres optimaux avec **learning_rate = 1.0**

In [93]:
```
fig, ax = plt.subplots(nrows=1, ncols=1, figsize=(4, 4))

ax.plot(number_of_stumps, trn_err, marker='o', markeredgecolor='w', linewidth=2)
ax.plot(number_of_stumps, val_err, marker='s', markeredgecolor='w', linewidth=2)
ax.legend(['Train err', 'Validation err'])
ax.set_xlabel('Number of decision stumps')
ax.set_ylabel('Error (%)')

fig.tight_layout()
```



Sur le graphique ci-dessus, on peut voir que le nombre de souches de décision optimaux est **45 souches**, car 45 souches présentent l'erreur la plus faible dansl'ensemble d'entraînement et l'ensemble de validation.