Fundamentos em Data Science

Aula 5 - Algoritmos de Ensemble

- Bagging
- Boosting
- Random Forests
- AdaBoost
- Stacking

Fábio Sato - fabiosato@gmail.com

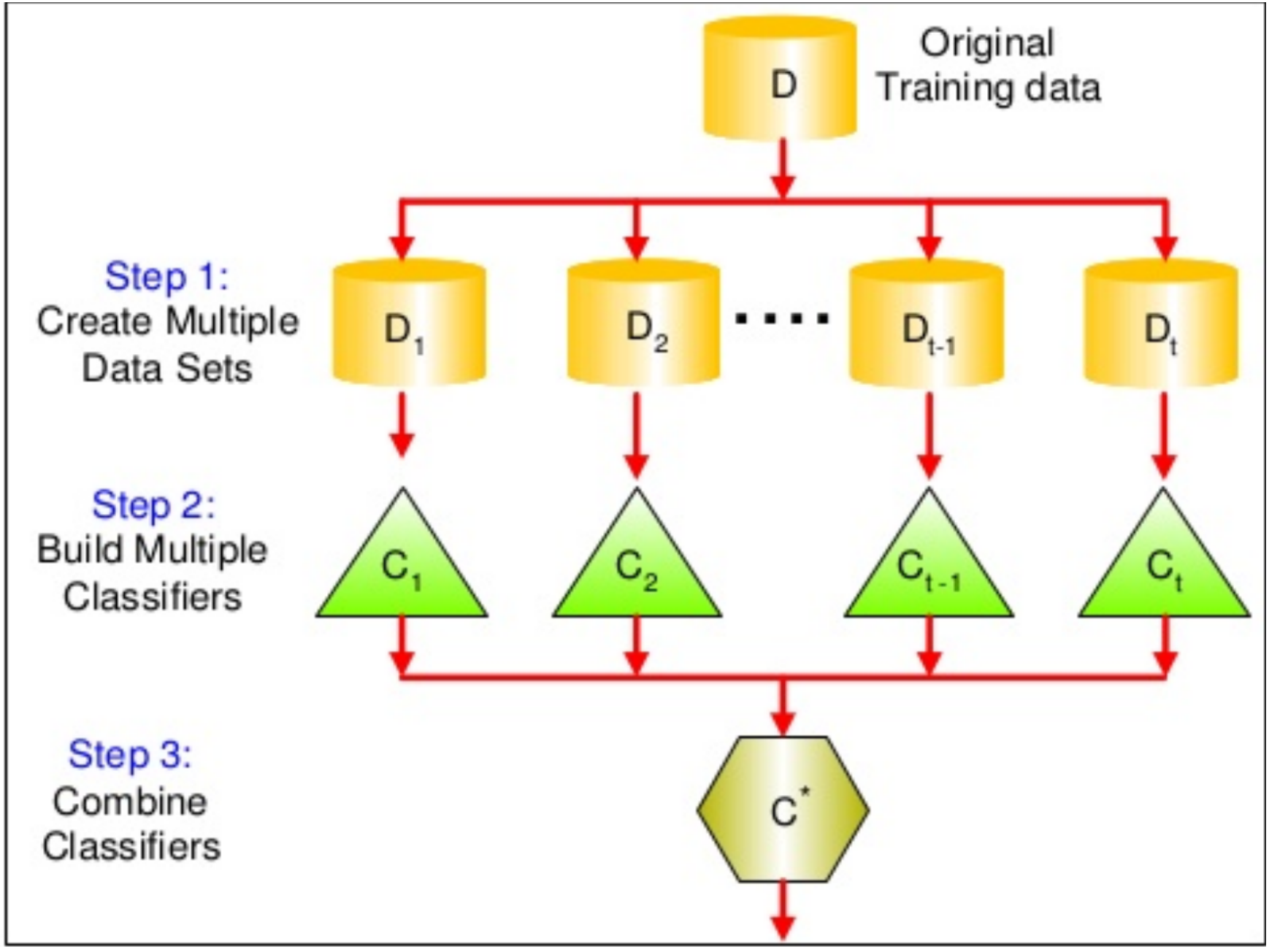
Ensemble

Meta algoritmos que combinam várias técnicas de aprendizado de máquina em um único modelo preditivo

Possibilita obter um desempenho preditivo melhor do que o obtido por um modelo único

Objetivo: melhorar a predição de um modelo base (classificador ou regressor)

Ensemble


D
Original
Training data
Step 1:
Create Multiple
Data Sets
D1
D2
Dt-1
Dt
Step 2:
Build Multiple
Classifiers
C1
C2
Ct-1
Ct
Step 3:
Combine
Classifiers
C*

Ensemble - Tipos de Métodos

Os métodos de ensemble podem ser divididos em dois grupos principais:

- Sequencial
- Paralelo

Ensemble - Métodos Sequenciais

Modelos base são gerados sequencialmente

Explora a DEPENDÊNCIA entre os modelos base

O desempenho global pode ser melhorado atribuindo pesos maiores a exemplos classificados erroneamente pelo modelo anterior

Ensemble - Métodos Paralelos

Modelos base são gerados em paralelo

Explorar a INDEPENDÊNCIA entre os modelos base

Erros podem ser reduzidos drasticamente através da média

Ensemble - Homogêneo x Heterogêneo

Homogêneo: A maioria dos métodos de ensemble utilizam um algoritmo base de aprendizado
para produzir modelos do mesmo tipo

Heterogêneo: Existem alguns métodos que utilizam algoritmos heterogêneos (de diferentes tipos)

Para que os métodos baseados em ensemble apresentem maior acurácia do que qualquer
um de seus membros individuais, os modelos de base devem ser o mais precisos e DIVERSOS possíveis

Ensemble - Bagging

Bagging: *Boostrap Aggregation*

Redução da variância nas estimativas através da agregação (média) de múltiplas estimativas

Treina algoritmos em diferentes subconjuntos de dados

As bases de treinamento para cada modelo base são geradas através de amostragem aleatória com reposição

Para agregar as saídas dos modelos base, bagging utiliza o voto para classificação e média para regressão

Ensemble - Bagging

$$f(x) = \frac{1}{M} \sum_{m=1}^{M} f_m(x)$$

Ensemble - Bagging: Exemplo

Iremos precisar do pacote MLXtend para este exercício

```
pip install mlxtend
```

Ensemble - Bagging: Exemplo

```
import matplotlib as mpl
mpl.rcParams['figure.figsize'] = (12, 8)
import itertools
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt
import matplotlib.gridspec as gridspec
from sklearn import datasets
from sklearn.tree import DecisionTreeClassifier
from sklearn.neighbors import KNeighborsClassifier
from sklearn.ensemble import BaggingClassifier
from sklearn.model_selection import cross_val_score, train_test
from mlxtend.plotting import plot_learning_curves
from mlxtend.plotting import plot_decision_regions

np.random.seed(0)
```

Ensemble - Bagging: Exemplo

```
iris = datasets.load_iris()
X, y = iris.data[:, 0:2], iris.target

clf1 = DecisionTreeClassifier(criterion='entropy', max_depth=3)
clf2 = KNeighborsClassifier(n_neighbors=3)

bagging1 = BaggingClassifier(
        base_estimator=clf1, n_estimators=100,
    max_samples=0.3, max_features=0.8)

bagging2 = BaggingClassifier(
        base_estimator=clf2, n_estimators=100,
        max_samples=0.3, max_features=0.8)
```

Ensemble - Bagging: Exemplo

```
label = ['Decision Tree', 'k-NN',
         'Bagging Tree', 'Bagging k-NN']

clf_list = [clf1, clf2, bagging1, bagging2]

fig = plt.figure(figsize=(10, 8))
gs = gridspec.GridSpec(2, 2)
grid = itertools.product([0,1],repeat=2)

for clf, label, grd in zip(clf_list, label, grid):
    scores = cross_val_score(clf, X, y, cv=3,
        scoring='accuracy')

    print("Accuracy: %.2f (+/- %.2f) [%s]" %
        (scores.mean(), scores.std(), label))

    clf.fit(X, y)
    ax = plt.subplot(gs[grd[0], grd[1]])
    fig = plot_decision_regions(X=X, y=y, clf=clf, legend=2)
    plt.title(label)

plt.show()
```

Random Forests

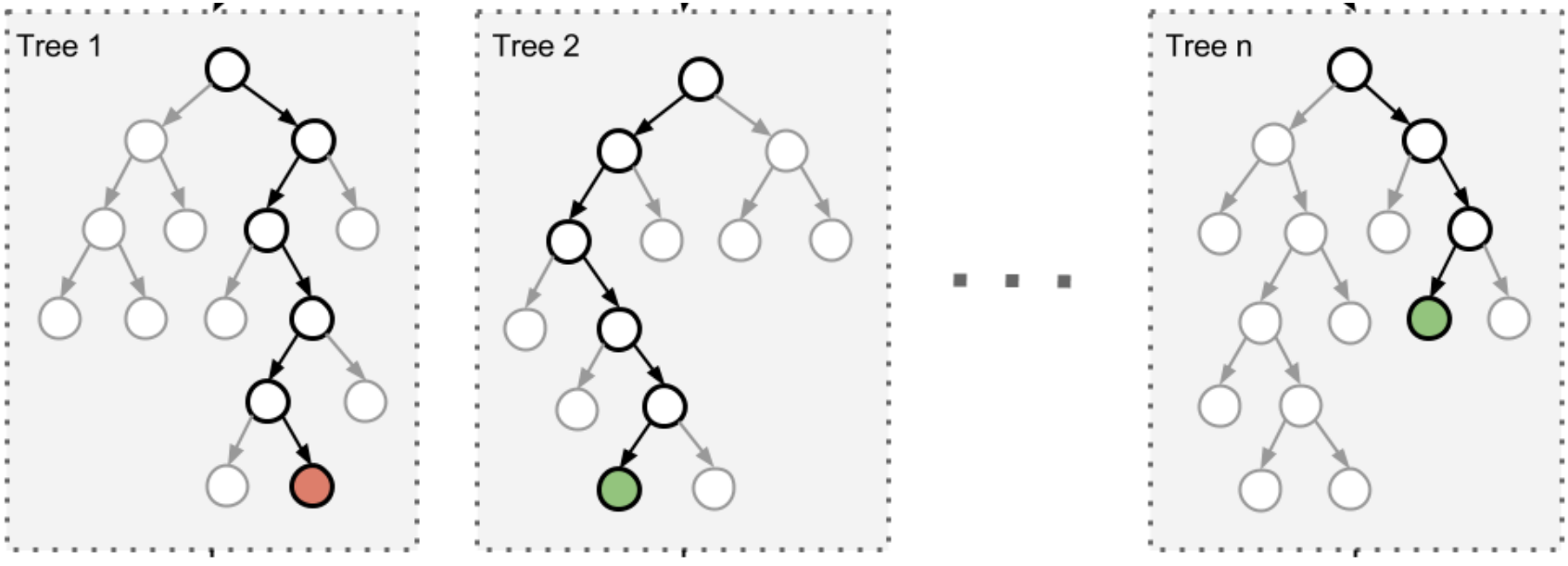
Um exemplo bem comum de classe de algoritmo de ensemble são as florestas aleatórias de árvores (Random Forests)

Em Random Forests, cada árvore é um ensemble construído a partir de uma amostra com substituição.

Além disso, ao invés de utilizar todas as características, um conjunto aleatório de características é selecionado
tornando o processo de criação de árvores ainda mais aleatório.

Como resultado, o viés da floresta cresce ligeiramente mas a variância diminui significativamente resultando em um modelo muito melhor.

Random Forests



Random Forests - Exercício

Implemente um modelo baseado em Random Forests para a base de dados 'Boston House Prices'

Ensemble - Boosting

Família de algoritmos que convertem classificadores fracos em algoritmos de aprendizado fortes.

Princípio fundamental: treinar uma sequência de classificadores fracos que são apenas um pouco melhores que adivinhar aleatoriamente (ex: pequenas árvores de decisão).

Ensemble - Boosting

Os classificadores fracos são treinados em versões com peso ponderado dos dados. Mais peso é atribuído aos exemplos que foram classificadores errôneamente pelo modelo anterior.

As previsões são combinadas usando voto ponderado (classificação) ou soma ponderada (regressão) para produzir a estimativa final.

Boosting - AdaBoost

Os classificadores fracos no AdaBoost são árvores de decisão com profundida igual a 1, chamados de *stumps* (tocos).

O AdaBoost pondera as observações, colocando mais peso na dificuldade de classificar exemplos e menos naqueles que já são classificados corretamente.

Novos classificadores fracos são adicionados sequencialmente e focam seu treinamento nos padrões mais difíceis.

Exemplos que são difíceis de serem classificados recebem cada vez mais peso até que o algoritmo os classifiquem corretamente.

AdaBoost - Algoritmo

1. Atribua pesos uniformes ao vetor $\mathbf{w}$ onde $\sum_i w_i = 1$
2. Para cada j em m rodadas de boosting, faça:
3. Treine um classificador fraco: $c_j = train(\mathbf{X}, \mathbf{y}, \mathbf{w})$
4. Estime as novas classes: $\hat{y} = predict(C_j, \mathbf{X})$
5. Calcule a taxa de erro ponderado: $\epsilon = \mathbf{w} \cdot (\hat{\mathbf{y}} == \mathbf{y})$
6. Calcule o coeficiente: $\alpha_j = 0.5log\frac{1-\epsilon}{\epsilon}$
7. Atualize os pesos: $\mathbf{w} = \mathbf{w} \times exp(-\alpha_j \times \hat{\mathbf{y}} \times \mathbf{y})$
8. Normalize os pesos para que a soma seja 1: $\mathbf{w} := \mathbf{w} / \sum_i w_i$
9. Realize as estimativas finais:
   $\hat{\mathbf{y}} = (\sum_{j=1}^{m} (\alpha_j \times predict(C_j, \mathbf{X})) > 0)$

Boosting - Exemplo

```
import itertools
import numpy as np

import seaborn as sns
import matplotlib.pyplot as plt
import matplotlib.gridspec as gridspec

from sklearn import datasets

from sklearn.ensemble import AdaBoostClassifier
from sklearn.model_selection import cross_val_score, train_test

from mlxtend.plotting import plot_learning_curves
from mlxtend.plotting import plot_decision_regions
```

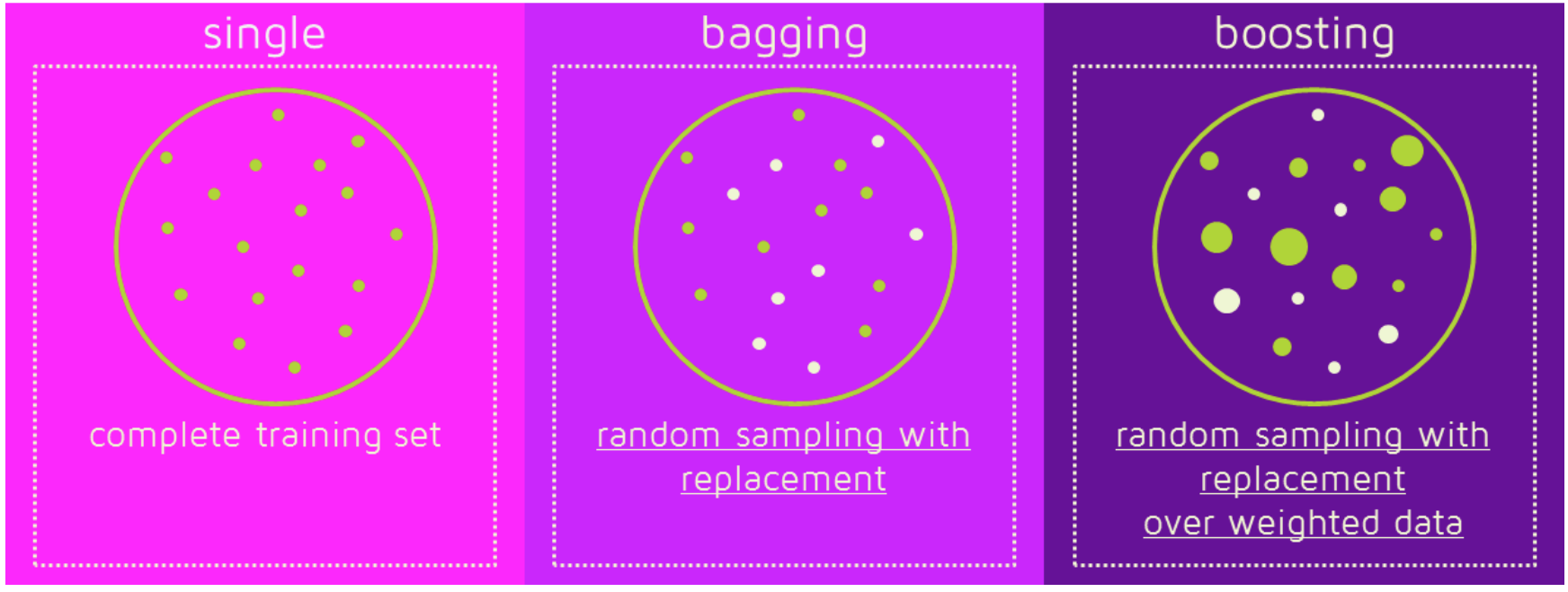
Boosting - Exemplo

```
iris = datasets.load_iris()
X, y = iris.data[:, 0:2], iris.target

clf = DecisionTreeClassifier(criterion='entropy', max_depth=1)

num_est = [1, 2, 3, 6]
label = ['AdaBoost (n=1)', 'AdaBoost (n=2)', 'AdaBoost (n=3)',
         'AdaBoost (n=10)']
```

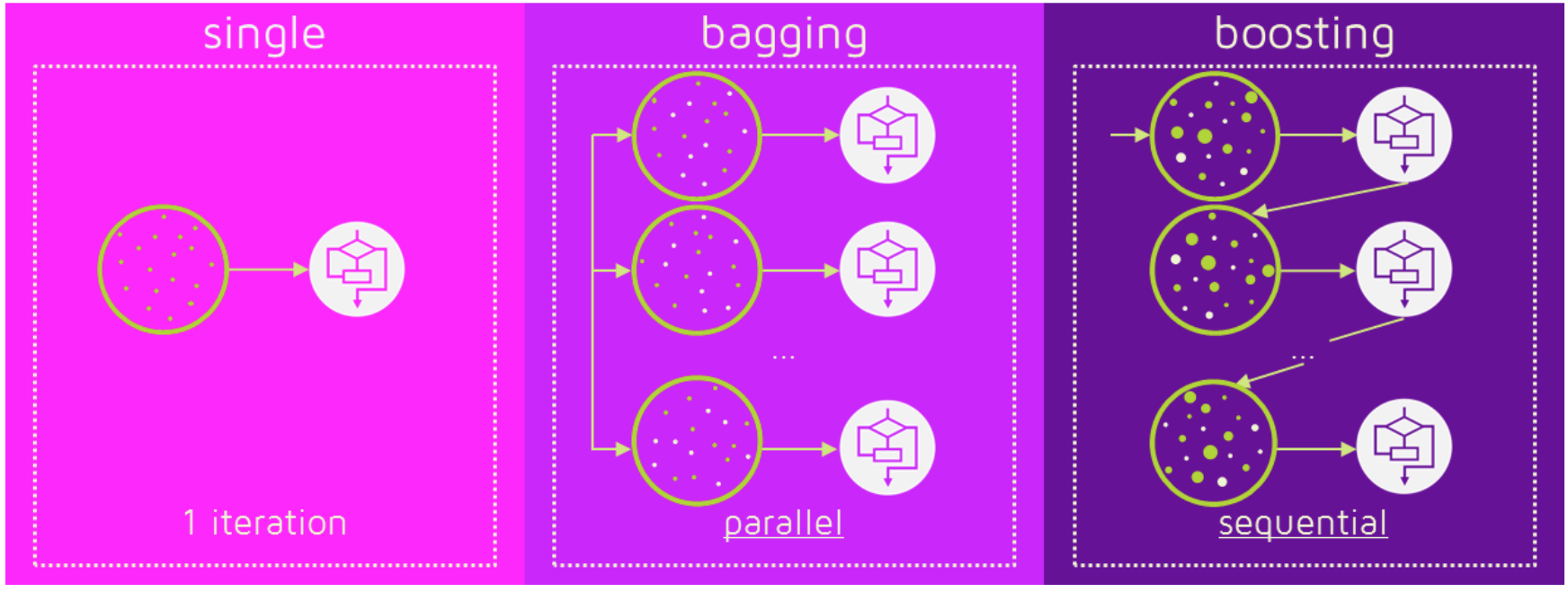
Boosting - Exemplo

```
fig = plt.figure(figsize=(10, 8))
gs = gridspec.GridSpec(2, 2)
grid = itertools.product([0,1],repeat=2)

for n_est, label, grd in zip(num_est, label, grid):
    boosting = AdaBoostClassifier(base_estimator=clf,
                                  n_estimators=n_est)
    boosting.fit(X, y)
    ax = plt.subplot(gs[grd[0], grd[1]])
    fig = plot_decision_regions(X=X, y=y, clf=boosting, legend=
    plt.title(label)

plt.show()
```

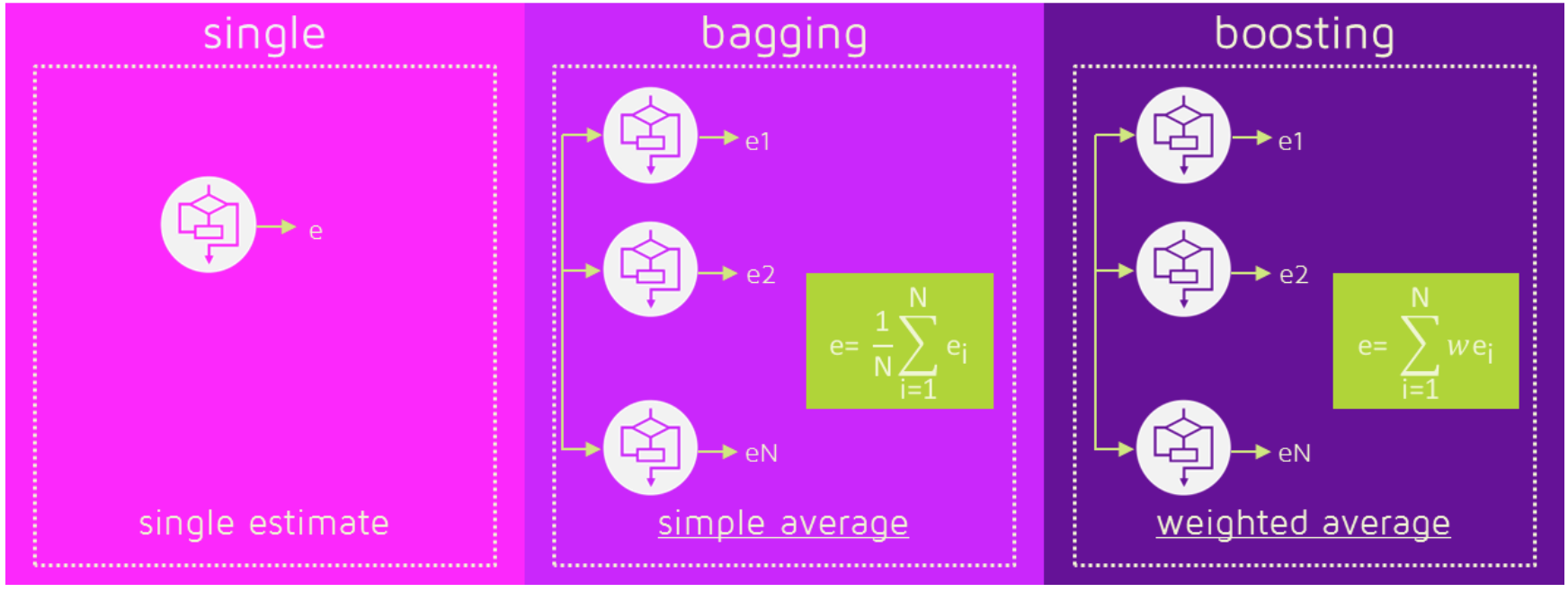
Bagging vs Boosting: Bases



Bagging vs Boosting: Sequencial vs Paralelo



Bagging vs Boosting: Treinamento



Bagging vs Boosting: qual é o melhor?

Não há um vencedor: depende dos dados, da complexidade do problema e do algoritmo base

Bagging e Boosting reduzem a variância de uma estimativa única. Modelo com maior estabilidade.

Bagging raramente irá melhorar o viés de um modelo único com baixo desempenho. Entretanto pode gerar um modelo combinado com menores erros.

Se o problema do modelo único é o overfitting, então Bagging é a melhor opção. Boosting não ajuda a evitar/melhorar o overfitting.

Stacking

Stacking é uma técnica de aprendizado por ensemble que combina múltiplos modelos de classificação
ou regressão através de um meta classificador ou meta regressor.

Os modelos de base são treinados em um conjunto completo da base de treinamento, e então o meta modelo utiliza as
saídas dos modelos de base como características de entrada.

A base geralmente utiliza diferentes algoritmos de aprendizado e portanto geralmente stacking se refere a
um modelo heterogêneos de ensemble

Stacking - Algoritmo

Algorithm Stacking

1: Input: training data $D = \{x_i, y_i\}_{i=1}^{m}$
2: Ouput: ensemble classifier H
3: *Step 1: learn base-level classifiers*
4: **for** $t = 1$ to T **do**
5: learn h_t based on D
6: **end for**
7: *Step 2: construct new data set of predictions*
8: **for** $i = 1$ to m **do**
9: $D_h = \{x'_i, y_i\}$, where $x'_i = \{h_1(x_i), ..., h_T(x_i)\}$
10: **end for**
11: *Step 3: learn a meta-classifier*
12: learn H based on D_h
13: return H

Stacking: Exemplo

```
import itertools
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt
import matplotlib.gridspec as gridspec

from sklearn import datasets

from sklearn.linear_model import LogisticRegression
from sklearn.naive_bayes import GaussianNB
from sklearn.ensemble import RandomForestClassifier
from mlxtend.classifier import StackingClassifier
```

Stacking: Exemplo

```
iris = datasets.load_iris()
X, y = iris.data[:, 1:3], iris.target

clf1 = KNeighborsClassifier(n_neighbors=1)
clf2 = RandomForestClassifier(random_state=1)
clf3 = GaussianNB()
lr = LogisticRegression()
sclf = StackingClassifier(classifiers=[clf1, clf2, clf3],
                          meta_classifier=lr)
```

Stacking: Exemplo

```
label = ['KNN', 'Random Forest', 'Naive Bayes',
         'Stacking Classifier']
clf_list = [clf1, clf2, clf3, sclf]
fig = plt.figure(figsize=(10,8))
gs = gridspec.GridSpec(2, 2)
grid = itertools.product([0,1],repeat=2)

clf_cv_mean = []
clf_cv_std = []
for clf, label, grd in zip(clf_list, label, grid):

    scores = cross_val_score(clf, X, y, cv=3,
        scoring='accuracy')
    print("Accuracy: %.2f (+/- %.2f) [%s]" %
        (scores.mean(), scores.std(), label))
    clf_cv_mean.append(scores.mean())
    clf_cv_std.append(scores.std())

    clf.fit(X, y)
    ax = plt.subplot(gs[grd[0], grd[1]])
    fig = plot_decision_regions(X=X, y=y, clf=clf)
    plt.title(label)
plt.show()
```

Stacking: Exemplo

```
plt.figure()
(_, caps, _) = plt.errorbar(range(4), clf_cv_mean, yerr=clf_cv_
for cap in caps:
    cap.set_markeredgewidth(1)
plt.xticks(range(4), ['KNN', 'RF', 'NB', 'Stacking'])
plt.ylabel('Accuracy'); plt.xlabel('Classifier'); plt.title('St
plt.show()
```

Stacking: Exemplo

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

plt.figure()
plot_learning_curves(X_train, y_train, X_test, y_test, sclf, pr
plt.show()
```