Arbres de décision, forêts aléatoires, bagging et boosting

Consignes : Vous rédigerez un compte-rendu détaillé de ce travail pratique (jusqu'à la question 16), incluant des réponses littérales, numériques et graphiques, ainsi que toutes les implémentations Python réalisées. Votre compte-rendu prendra la forme soit d'un fichier `.pdf` et d'un unique code source `.py` ; soit d'un document linéaire `.ipynb`. Dans tous les cas, votre code devra être **commenté** de sorte à faire clairement apparaître les différentes parties et questions de ce travail pratique, ainsi que les opérations élémentaires réalisées dans vos implémentations. Le code Python devra être **exécutable par le correcteur** et vos résultats reproductibles. Ce compte-rendu est à envoyer par courriel à maxime.sangnier@telecom-paristech.fr au plus tard le lundi **7 juin 2015** à 23h59. Seuls les documents effectivement reçus avant ce moment seront corrigés et notés.

- Arbres de décision -

Les arbres de décision (en particulier l'algorithme *Classification And Regression Trees* – CART – que nous étudierons ici) ont été introduits par Leo Breiman et ses collaborateurs [3]. Nous rappelons ici le principe d'un arbre de décision binaire (*i.e.* un nœud possède soit deux enfants, soit aucun – c'est une feuille) et nous renvoyons le lecteur à [7, chapitre 9.2] pour une présentation plus détaillée.

Parcourir un arbre de décision consiste à réduire l'espace probable des variables expliquée en appliquant récursivement des règles simples (figure 1). En pratique, chaque nœud d'un arbre incarne une règle de décision linéaire sur l'une des variables explicatives. Étant donnée une observation $\mathbf{x}$ (on note x_j la variable observée au nœud courant) et un seuil τ, on continu le parcours de l'arbre à gauche du nœud si $x_j \leq \tau$ et à droite si $x_j > \tau$.

La construction d'un arbre de décision est réalisée à partir d'un ensemble d'apprentissage $\mathcal{D}_n = \{(\mathbf{x}_i, y_i),\ i \in [\![1, n]\!]\}$ contenant n couples d'observations explicatives $\mathbf{x}_i$ issues d'un espace vectoriel $\mathcal{X} \subset \mathbb{R}^p$ et d'observations expliquées y_i issues d'un espace $\mathcal{Y}$ discret (classification) ou continu (régression).

Un arbre est construit de manière récursive, *i.e.* en commençant par le nœud racine et en évoluant vers les feuilles. À chaque nœud est associé un sous-espace $\mathcal{N}$ de $\mathcal{X} \times \mathcal{Y}$ (déterminé par le chemin parcouru dans l'arbre – à la racine $\mathcal{N} = \mathcal{X} \times \mathcal{Y}$). La construction des enfants consiste à déterminer :

- une variable x_j à segmenter (parmi les p possibles) ;
- un seuil $\tau \in \mathbb{R}$ de décision.

Si l'on suppose ces deux paramètres connus, on peut alors partitionner l'espace courant $\mathcal{N}$ en $(\mathcal{G}(j,\tau), \mathcal{D}(j,\tau))$ suivant :

$$\mathcal{G}(j,\tau) = \{(\mathbf{x}, y) \in \mathcal{N},\ x_j \leq \tau\}$$
$$\mathcal{D}(j,\tau) = \{(\mathbf{x}, y) \in \mathcal{N},\ x_j > \tau\} = \mathcal{N} \backslash \mathcal{G}(j,\tau).$$

Par la suite, au fils gauche, $\mathcal{N}$ prend la valeur $\mathcal{G}(j,\tau)$ et réciproquement au fils droit (figure 1).

À chaque nœud, les paramètres (j,τ) sont choisis de manière à minimiser l'*impureté* de la partition (*i.e.* à maximiser son homogénéité). Celle-ci est la moyenne pondérée de l'impureté de chaque ensemble, notée H :

$$(j,\tau) \in \operatorname*{arg\,min}_{j \in [\![1,n]\!]} \frac{\operatorname{card}(\mathcal{G}(j,\tau))}{\operatorname{card}(\mathcal{N})} H(\mathcal{G}(j,\tau)) + \frac{\operatorname{card}(\mathcal{D}(j,\tau))}{\operatorname{card}(\mathcal{N})} H(\mathcal{D}(j,\tau)).$$

Supposons que nous appliquions les arbres de décision à un problème de classification multi-classes (*i.e.* $\mathcal{Y} = [\![1, \dots c]\!]$). Étant donné un sous-ensemble $\mathcal{R}$ de $\mathcal{X} \times \mathcal{Y}$, notons $p_\ell(\mathcal{R})$ la proportion d'observations de $\mathcal{R}$ associées à la classe ℓ :

$$p_\ell(\mathcal{R}) = \frac{\operatorname{card}(\{(\mathbf{x}, y) \in \mathcal{R},\ y = \ell\})}{\operatorname{card}(\mathcal{R})}.$$

Pour la classification, on s'intéressera particulièrement aux mesures d'impureté suivantes :

— l'indice de Gini : $H(R) = \sum_{\ell=1}^{c} p_\ell(\mathcal{R})\,(1 - p_\ell(\mathcal{R}))$;

— l'entropie : $H(R) = -\sum_{\ell=1}^{c} p_\ell(\mathcal{R}) \log\left(p_\ell(\mathcal{R})\right)$.

Il subsiste un dernier point clef dans la construction d'un arbre de décision : le critère d'arrêt de la récursion. Celui-ci peut être :

— une profondeur maximale de l'arbre ;

— une limite inférieure du nombre d'observations d'apprentissage associés aux nœuds ;

— une limite inférieure du nombre d'observations d'apprentissage associées aux feuilles (cette clause peut être suffisante pour la précédente mais l'inverse n'est pas vrai) ;

— un nombre maximal de feuilles dans l'arbre.

Questions

1. Dans le cadre de la régression (*i.e.* quand on cherche à prédire une valeur continue et non discrète), proposez une mesure d'impureté. Justifiez votre choix.
2. Le fichier `utils.py` contient des fonctions de génération de données synthétiques et d'affichage en deux dimensions. Familiarisez-vous avec ces fonctions et affichez différents jeux de données avec des paramètres personnalisés.
3. Le module `tree` de `scikit-learn` permet de manipuler des arbres de décision. Par exemple, on peut construire un arbre de la manière suivante :

```
from sklearn import tree
clf = tree.DecisionTreeClassifier()
```

   Lire la page correspondante : http://scikit-learn.org/stable/modules/generated/sklearn.tree.DecisionTreeClassifier.html. Simuler avec `rand_checkers` un échantillon d'apprentissage de taille $n = 456$ et un échantillon de test de même taille (attention à bien équilibrer les classes). Créer quatre courbes qui donnent le pourcentage d'erreurs commises (en apprentissage et en test) en fonction de la profondeur maximale de l'arbre (deux courbes pour l'indice de Gini et deux autres pour l'entropie).
4. Afficher la classification résultant de l'arbre entraîné en utilisant la profondeur qui minimise l'erreur de test obtenue avec l'entropie (utiliser les fonctions `plot_2d` et `frontiere`).
5. À l'aide du code ci-dessous, exporter un graphique de l'arbre `clf` obtenu à la question précédente (nécessite l'installation de Graphviz – paquet `graphviz`).

```
import os
f = tree.export_graphviz(clf, out_file="my_tree.dot") # clf: tree classifier
os.system("dot -Tpdf my_tree.dot -o my_tree.pdf")
os.system("evince my_tree.pdf")
```

6. Créez $n = 200$ nouvelles données avec `rand_checkers`. Pour l'arbre de décision obtenu en question 4, calculer la proportion d'erreurs faites sur ce second échantillon. Commenter.
7. Reprendre la question 3 pour le jeu de données DIGITS, disponible dans le module `sklearn.datasets`. Il peut être chargé grâce au code ci-dessous :

```
from sklearn import datasets
digits = datasets.load_digits()
```

- Agrégation de modèles -

Supposons que l'on dispose de m modèles $(f_\ell)_{\ell=1}^m$ liant les variables explicatives à celles expliquées : $f_\ell \colon \mathcal{X} \to \mathbb{R}$. Pour une donnée $\mathbf{x}$, la prédiction est obtenue par $f_\ell(\mathbf{x})$ dans le cas de la régression et par $\mathrm{sign}(f_\ell(\mathbf{x}))$ dans celui de la classification binaire.

Une agrégation F_m des modèles $(f_\ell)_{\ell=1}^m$ est obtenue par combinaison (généralement linéaire) des prédictions individuelles de chaque modèle. Étant donné un vecteur de pondération $\boldsymbol{\alpha} \in \mathbb{R}_+^m$, le modèle agrégé est défini par : $F_m(\mathbf{x}) = \sum_{\ell=1}^m \alpha_\ell f_\ell(\mathbf{x})$. On obtient de nouveau la prédiction $F_m(\mathbf{x})$ dans le cas de la régression et $\operatorname{sign}(F_m(\mathbf{x}))$ dans celui de la classification binaire. Alternativement, pour la classification multiclasse, l'agrégation peut être réalisée par vote majoritaire (la prédiction est la classe apparaissant en majorité parmi les votes individuels $(f_\ell(\mathbf{x}))_{\ell=1}^m$), moyennage de la probabilité des classes ou de la marge.

Une condition nécessaire et suffisante pour qu'une telle agrégation soit plus performante que chaque modèle pris individuellement est que ceux-ci prédisent mieux que le hasard et que leurs prédictions soient différentes lorsqu'ils ont été estimés sur des données différentes.

Le principe des méthodes d'agrégation réside donc dans l'assertion suivante : moyenner les prédictions de plusieurs modèles indépendants réduit la variance (et donc l'erreur) du résultat global. Par exemple, si l'on considère m classifieurs binaires indépendants dont la probabilité de prédire correctement est $p > 0.5$; alors la prédiction du modèle agrégé uniformément ($\alpha_l = 1/m$) suit une distribution Binomiale de paramètres p et m.

L'une des méthodes d'agrégation les plus courantes est le *bagging* (mot-valise issu de *bootstrap aggregating*) [1]. Elle consiste à calculer la moyenne des prédictions ($\alpha_l = 1/m$) des estimateurs entraînés sur des échantillons *bootstrap* des données d'apprentissage. Un tel échantillon est obtenu en tirant aléatoirement et avec remise n couples $(\mathbf{x}_i, y_i)$ de $\mathcal{D}_n$. Le *bootstrap* est une manière de générer aléatoirement de nouveaux ensembles d'apprentissage à partir de $\mathcal{D}_n$. Une autre manière consiste à tirer aléatoirement et sans remise n' données de $\mathcal{D}_n$ (avec $n' < n$). On parle alors d'échantillonnage aléatoire.

Questions

8. En reprenant l'exemple des classifieurs binaires donné ci-dessus (avec $f_\ell \colon \mathcal{X} \to \{-1, +1\}$), on suppose que l'on dispose de $m = 10$ modèles ayant une probabilité $p = 0.7$ de prédire correctement (le hasard a une probabilité 0.5 de fournir un résultat correct ; les modèles sont donc légèrement meilleurs que le hasard). Exécuter le code suivant est interpréter le graphique.

```python
from scipy.stats import binom
import numpy as np
import matplotlib.pyplot as plt

m, p = 10, 0.7 # Binomial parameters
x = np.arange(0,m+1) # Possible outputs
pmf = binom.pmf(x, m, p) # Probability mass function

plt.figure()
plt.plot(x, pmf, 'bo', ms=8)
plt.vlines(x, 0, pmf, colors='b', lw=5, alpha=0.5)
```

   Calculer (numériquement) la probabilité que le modèle agrégé uniformément ($\alpha_l = 1/m$) donne un résultat correct. On remarquera que cette probabilité est donnée par le nombre X d'issus positifs : $\frac{\mathbb{P}(X=5)}{2} + \mathbb{P}(X = 6) + \cdots + \mathbb{P}(X = 10)$.

9. Écrire un script mettant en œuvre le *bagging* avec des arbres souches (*i.e.* de profondeur 1, aussi appelés *tree stumps*), puis avec des arbres plus profonds. On partira du code ci-dessous. On pourra utiliser la fonction `np.random.randint` pour générer les échantillons *Bootstrap*.

```python
import numpy as np
import matplotlib.pyplot as plt
from sklearn.tree import DecisionTreeRegressor

# Create a random dataset
rng = np.random.RandomState(1)
X = np.sort(5 * rng.rand(80, 1), axis=0)
y = np.sin(X).ravel()
y[::5] += 1 * (0.5 - rng.rand(16))
```

```
# Fit regression model
max_depth = 1
clf_1 = DecisionTreeRegressor(max_depth=max_depth)
clf_1.fit(X, y)

# Predict
X_test = np.arange(0.0, 5.0, 0.01)[:, np.newaxis]

y_1 = clf_1.predict(X_test)

# Plot the results
import pylab as pl
plt.close('all')
plt.figure()
plt.scatter(X, y, c="k", label="data")
plt.plot(X_test, y_1, c="g", label="Tree (depth: %d)" % max_depth)
plt.xlabel("data")
plt.ylabel("target")
plt.title("Decision Tree Regression")
plt.legend()
plt.show()
```

10. Observer le rôle de m ainsi que de la profondeur des arbres (max_depth).
11. A quoi reconnaît-on que les estimateurs construits par les arbres sont biaisés et que le *bagging* réduit leur variance ?
12. En jouant sur le niveau de bruit, mettre en évidence le sur-apprentissage.
13. Observer qu'on peut réduire ce phénomène en échantillonnant aléatoirement et sans remise au lieu de prendre des échantillons *bootstrap*.

- Forêts aléatoires -

Les forêts aléatoires (*random forests*) sont une méthode d'agrégation de modèles (*bagging*) appliquée aux arbres de décision [2]. Leur principe est de combiner uniformément ($\alpha_l = 1/m$) des arbres entraînés sur des échantillons *bootstrap* de l'ensemble d'apprentissage. Pour induire un comportement différents de ces arbres (nécessaire au bon fonctionnement de l'agrégation), les variables x_j à segmenter sont choisies aléatoirement et seuls les seuils de décision τ sont déterminés par minimisation de l'impureté. Notons que l'agrégation est réalisée par vote majoritaire dans le cas de la classification multiclasse.

Questions

14. Compléter le script ci-dessous afin d'évaluer le score des forêts aléatoires (`RandomForestRegressor` et `RandomForestClassifier`) et des arbres agrégés (`BaggingRegressor` et `BaggingClassifier`) sur les jeux de données boston, diabetes (régression), iris et digits (classification). Pour chaque problème et chaque méthode, afficher la moyenne et l'écart type du score évalué par une procédure de validation croisée en 5 étapes (`scores = cross_val_score(clf, X, y, cv=5)`). Rappeler la différence entre les deux approches et en déduire une explication de leurs comportements lorsque le nombre d'estimateurs varie.

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn import datasets
from sklearn.ensemble import (RandomForestClassifier, RandomForestRegressor)
from sklearn.ensemble import (BaggingClassifier, BaggingRegressor)
from sklearn.tree import (DecisionTreeClassifier, DecisionTreeRegressor)
from sklearn.utils import shuffle
from sklearn.cross_validation import cross_val_score
```

```
%matplotlib inline

# Parameters for the tree and the random forest
n_estimators = 10 # Up to 500
max_depth = 4
min_samples_split = 1
params = {'n_estimators': n_estimators, 'max_depth': max_depth,
                      'min_samples_split': min_samples_split}

# For each database
for name, dataset in (('Boston', datasets.load_boston()),
                      ('Diabetes', datasets.load_diabetes()),
                      ('Iris', datasets.load_iris()),
                      ('Digits', datasets.load_digits())):
    # Suffle data
    X, y = shuffle(dataset.data, dataset.target, random_state=0)
    # Normalize data
    mean, std = X.mean(axis=0), X.std(axis=0)
    X = (X - mean) / std
    # Get rid of Nan values
    X[np.isnan(X)] = 0.
```

15. En tant que méthode d'agrégation, les forêts aléatoires peuvent retourner des probabilités associées à chaque prédiction. La probabilité d'appartenance d'une donnée à une classe est la proportion d'arbres ayant prédit cette classe. En utilisant le jeu de données IRIS restreint aux deux premières variables, afficher la probabilité des classes prédites pour chaque observation. On partira du script suivant et on augmentera le nombre d'estimateurs (*i.e.* d'arbres aléatoires).

```
# Parameters
n_estimators = 1
plot_colors = "bry"
plot_step = 0.02

# Load data
iris = datasets.load_iris()
X, y = iris.data[:, :2], iris.target
# Shuffle
X, y = shuffle(X, y, random_state=42)
# Normalize
mean, std = X.mean(axis=0), X.std(axis=0)
X = (X - mean) / std

# Train the model
model = RandomForestClassifier(n_estimators=n_estimators)
clf = model.fit(X, y)

# Plot the decision boundary
x_min, x_max = X[:, 0].min() - 1, X[:, 0].max() + 1
y_min, y_max = X[:, 1].min() - 1, X[:, 1].max() + 1
xx, yy = np.meshgrid(np.arange(x_min, x_max, plot_step),
                     np.arange(y_min, y_max, plot_step))

plt.close('all')
for tree in model.estimators_:
    Z = tree.predict(np.c_[xx.ravel(), yy.ravel()])
    Z = Z.reshape(xx.shape)
```

```
            cs = plt.contourf(xx, yy, Z, alpha=1. / n_estimators, cmap=plt.cm.Paired)
        plt.axis("tight")

        # Plot the training points
        for i, c in zip(xrange(3), plot_colors):
            idx = np.where(y == i)
            plt.scatter(X[idx, 0], X[idx, 1], c=c, label=iris.target_names[i],
                        cmap=plt.cm.Paired)
        plt.show()
```

16. Sur le même jeu de données, comparer les scores des forêts aléatoires et des arbres de décisions seuls (`DecisionTreeClassifier`) en fonction de la profondeur maximale des arbres. Pour cela, vous utilisez une procédure de validation croisée en 5 étapes et ferez varier la profondeur maximale de 1 à 30. Mettre ainsi en évidence la capacité des forêts aléatoires (comparées aux arbres de décision) à réduire le sur-apprentissage (y compris lorsque l'on utilise des arbres profonds, particulièrement sujets à ce phénomène).

- Boosting -

Le *boosting* est une généralisation du *bagging* qui pondère de manière non-uniforme les contributions des sous-modèles ($\alpha_\ell \neq \frac{1}{m}$ a priori). Historiquement, l'un des premiers algorithmes de *boosting* à rencontrer un réel succès s'appelle « AdaBoost.M1 » (par la suite, nous ferons majoritairement référence à cet algorithme lorsque nous mentionnerons la technique du *boosting*). Il a été proposé par Freund et Schapire en 1997 [4]. Pour cette partie, on pourra se référer aux articles [5, 6]. Des informations supplémentaires en lien avec [5], sont aussi disponibles sur la page de l'auteur http://www-stat.stanford.edu/~jhf/R-MART.html et dans [7, Chapitre 10].

Une approche d'agrégation par *boosting* construit itérativement les sous-modèles à agréger en insistant sur les données mal classées. C'est une technique gloutonne qui ne modifie par les modèles estimés aux itérations précédentes. À l'étape $m-1$, le modèle agrégé courant est donné par : $F_{m-1}(\mathbf{x}) = \sum_{\ell=1}^{m-1} \alpha_\ell f_\ell(\mathbf{x})$. Étant donnée une nouvelle contribution f_m, le principe du *boosting* est de déterminer une pondération α_m pour le nouvel estimateur, de sorte à obtenir un modèle agrégé $F_m = F_{m-1} + \alpha_m f_m$ plus performant que F_m. En pratique, α_m est obtenue par minimisation du risque empirique exponentiel : $\alpha_m \propto \beta^*$, avec

$$\beta^* = \arg\min_\beta \frac{1}{n}\sum_{i=1}^n \exp\Big(- y_i\big(F_{m-1}(\mathbf{x}_i) + \beta f_m(\mathbf{x}_i)\big)\Big).$$

L'algorithme AdaBoost est donné ci-après. On remarque que cet algorithme augmente les poids des données qui sont mal classées (*i.e.* $y_i \neq f_\ell(\mathbf{x}_i)$) par le dernier modèle estimé. Ceci a pour but de prêter plus attention à eux à l'itération suivante. De plus, on peut noter que le *boosting* est une technique généralement très efficace lorsqu'elle est combinée à des arbres de décisions. En revanche et contrairement aux forêts aléatoires, le *boosting* se prête difficilement à des estimations en parallèle des sous-modèles.

Questions

17. En gardant à l'esprit que les sous-modèles f_ℓ sont binaires (*i.e.* les prédictions sont 0 ou 1), montrer (par différentiation) que la solution du dernier problème d'optimisation est $\beta^* = \frac{1}{2}\log\left(\frac{\mathbb{P}_{\hat{w}}(Y=f_m(X))}{\mathbb{P}_{\hat{w}}(Y\neq f_m(X))}\right)$, où les probabilités empiriques sont définies comme dans l'algorithme AdaBoost et $\hat{w}_i = \exp(-y_i F_{m-1}(\mathbf{x}_i))$.
18. Montrer que les poids $u_i^\ell \propto u_i^{\ell-1}\cdot\exp(-\beta^* y_i f_\ell(\mathbf{x}_i))$ (où β^* est défini ci-dessus) et les poids $w_i^\ell \propto w_i^{\ell-1}\cdot\exp(\alpha_\ell\cdot\mathbb{1}_{\{y_i\neq f_\ell(\mathbf{x}_i)\}})$ (où α_ℓ est défini dans l'algorithme AdaBoost) sont identiques (avec la convention $\hat{F}_0 = 0$ et $w^0 = u^0 = (\frac{1}{n},\dots,\frac{1}{n})$). Dans ce qui précède, $\propto$ comprend l'opération de normalisation.
19. Écrire une routine mettant en oeuvre AdaBoost avec des arbres de profondeur 1, puis 2, puis 10, sur le jeu de données digits. Afficher les fonctions de décision (question 15). On utilisera pour cela le sous-algorithme `SAMME` et la commande d'import :

Algorithme 1 : AdaBoost

Data : Un ensemble d'apprentissage $\mathcal{D}_n$, un nombre d' étapes m, un vecteur poids $w^0 \in \mathbb{R}^n$ (généralement on choisit $w_i^0 = \frac{1}{n}$ pour tout $i \in [\![1, n]\!]$).

Result : Un modèle agrégé F_m

for $\ell = 1$ *to* m **do**

- Estimer un classifieur binaire (faible) f_ℓ minimisant l'erreur pondérée par le vecteur de poids $w^{\ell-1} = (w_1^{\ell-1}, \dots, w_n^{\ell-1})$;
- Calculer l'erreur associée $\mathrm{err}_\ell = \mathbb{P}_{w^{\ell-1}}(Y \neq f_\ell(X)) = \sum_{i=1}^n w_i^{\ell-1} \mathbb{1}_{\{y_i \neq f_\ell(\mathbf{x}_i)\}}$; et le poids associé à l'étape ℓ, $\alpha_\ell = \log[(1 - \mathrm{err}_\ell)/\mathrm{err}_\ell)]$;
- Mise à jour des poids : $w_i^{int} = w_i^{\ell-1} \exp(c_\ell \cdot \mathbb{1}_{\{y_i \neq f_\ell(\mathbf{x}_i)\}})$ et normalisation : $w_i^\ell = w_i^{int} / \sum_{j=1}^n w_j^{int}$.

$F_m = \mathrm{sign}\left(\sum_{\ell=1}^m \alpha_\ell f_\ell\right)$

```
from sklearn.ensemble import AdaBoostClassifier
```

20. Appliquer AdaBoost sur les données digits découpées en deux échantillons : apprentissage et test. Tracer les erreurs d'apprentissage et de test en fonction du nombre d'itérations du *boosting* (fixer `random_state=1`).
21. Que remarquez vous ? Que se passe-t-il si la profondeur des arbres de classification est grande ?

- Liens utiles -

⋆⋆⋆ http://scikit-learn.org/stable/modules/tree.html

⋆⋆⋆ http://scikit-learn.org/stable/modules/generated/sklearn.tree.DecisionTreeClassifier.html

⋆⋆ IPython Notebook : http://nbviewer.ipython.org/github/ipython/ipython/tree/1.x/examples/notebooks/

⋆⋆ Plus de détails sur les arbres (notamment pour le langage `R`) : http://www.stat.cmu.edu/~cshalizi/350/lectures/22/lecture-22.pdf

Références

[1] L. Breiman. Stacked regressions. *Mach. Learn.*, 24(1) :49–64, 1996. 3

[2] L. Breiman. Random Forests. *Mach. Learn.*, 45(1) :5–32, 2001. 4

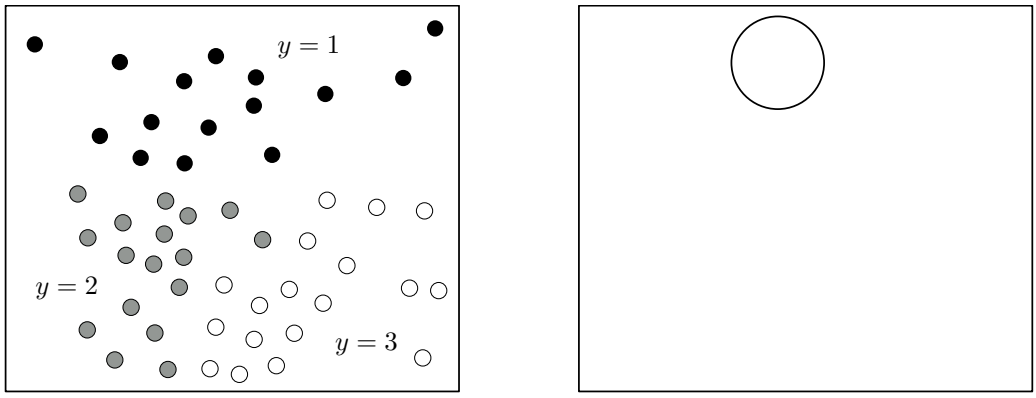
[3] L. Breiman, J. H. Friedman, R. A. Olshen, and C. J. Stone. *Classification and regression trees*. Wadsworth Statistics/Probability Series. Wadsworth Advanced Books and Software, Belmont, CA, 1984. 1

[4] Y. Freund and R. E. Schapire. A decision-theoretic generalization of on-line learning and an application to boosting. *Journal of computer and system sciences*, 55(1) :119–139, 1997. 6

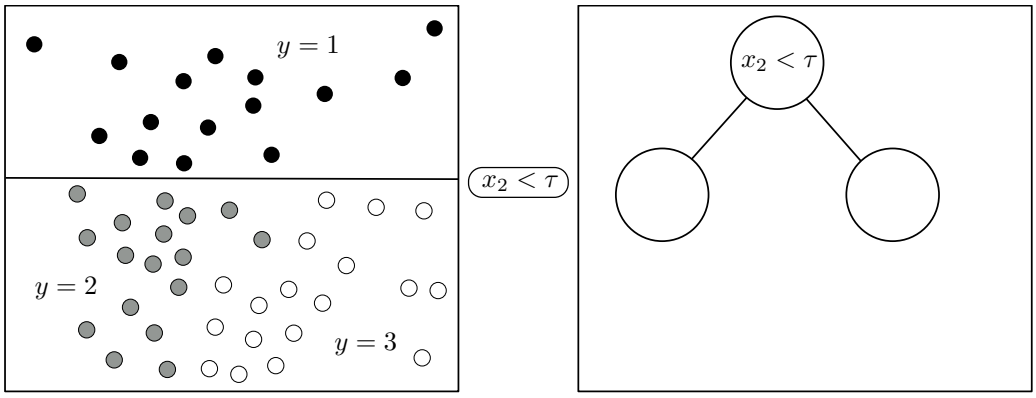
[5] J. Friedman. Greedy function approximation : a gradient boosting machine. *Ann. Statist*, 29(5) :1189–1232, 2001. http://www-stat.stanford.edu/~jhf/ftp/trebst.pdf. 6

[6] J. Friedman, T. Hastie, and Robert R. Tibshirani. Additive logistic regression : a statistical view of boosting. *Ann. Statist.*, 28(2) :337–407, 2000. http://www-stat.stanford.edu/~jhf/ftp/boost.pdf. 6

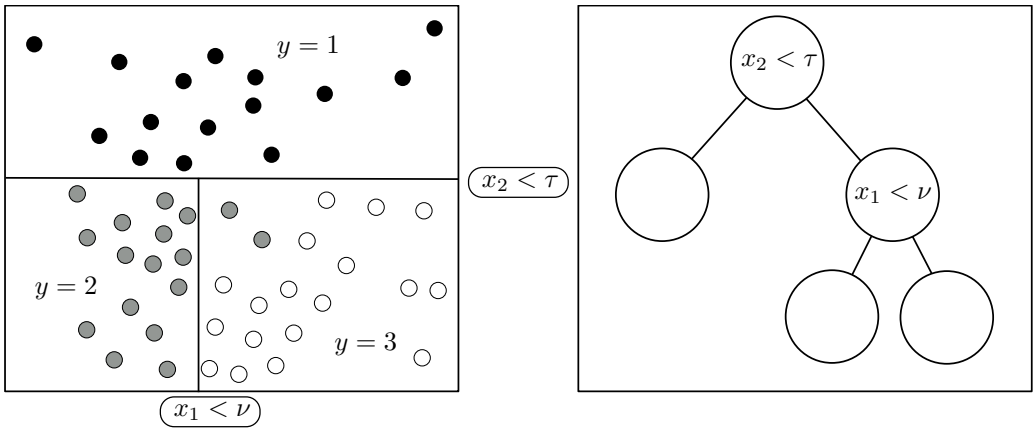
[7] T. Hastie, R. Tibshirani, and J. Friedman. *The elements of statistical learning*. Springer Series in Statistics. Springer, New York, second edition, 2009. http://www-stat.stanford.edu/~tibs/ElemStatLearn/. 1, 6



(a) Aucune découpe.



(b) Une découpe.



(c) Deux découpes.

Figure 1 – Exemple de parcours (et alternativement de création) d'un arbre de décision.