Détection d'obstacles par un sonar de sous-marin

stephane.gonnord@prepas.org

Disclaimer : ce corrigé, assez tonique, est destiné à mes élèves (qui comprendront le ton des remarques) ainsi qu'aux collègues (qui pourront éliminer les erreurs, typos et commentaires désobligeants grâce au source LaTeX fourni avant de le distribuer à leurs propres élèves).

Partie I - Introduction

(Blabla)

Partie II - Analyse de données

II.1 - Présentation

Q 1. On utilise bien entendu [1] `arange` du module `numpy` :

```
temps = arange(0, Tf, Tf/N)
```

Enfin... peut-être qu'il était jugé préférable d'écrire :

```
temps = array([k*Tf/n for k in range(N)])
```

Attention, seule la deuxième solution nous assure d'avoir un vecteur avec N éléments, comme le sait évidemment l'auteur du sujet [2].

Q 2. Difficile de savoir si ce qui est attendu/préféré est une fonction vectorisée appliquée au vecteur `temps`. Ou bien plutôt quelque chose comme ça :

```
def chirp(temps, f0, Deltafe, T, E0):
    def e(t):
        if t > T:
            return 0
        else:
            return E0*sin(2*pi*(f0+t/T*Deltafe)*t)
    return array([e(t) for t in temps])
```

Ici, `temps` aura le double statut de nom de paramètre et de nom de variable globale, ce qui est toujours éclairant...

Q 3. Il est évidemment hors de question pour un candidat de comprendre *réellement* les deux pages permettant de répondre à la question. Mais il est hélas probable que même une « compréhension approximative » est exclue également. Il reste le bluff, l'esbrouffe.

```
f, t, S = scipy.signal.stft(s, window = 'hamming', nperseg=n)
```

Je n'arrive pas bien à comprendre la phrase « en appliquant la transformée de Fourier... ». La taille des éléments renvoyés *effectivement* par `stft` ne correspond pas à ce que j'aurais compris si je m'étais contenté de lire (probablement mal) le sujet :

- la taille de `f` sera $n//2$ (et pas $2N//n$) ;
- celle de `t` sera de $|s|//(n//2) = 2N//n$ (on peut raisonnablement supposer n pair...) ;
- `S` sera un tableau à deux dimensions de complexes de taille $|f| \times |t|$.

En réalité c'est un peu plus compliqué pour des questions d'arrondis. Passons...

1. Par le passé (2018, Q2), ce concours a montré une absence de maîtrise de ladite fonction... mais `linspace` est peut-être proscrite pour d'excellente raison...
2. C'est une blague : cf sujet 2018, Q2...

Q 4.
- L'allure des graphes produits dans l'énoncé est assez étrange. On peut imaginer une mauvaise photocopie... et peut-être même un « photoshopage ». Non spécialiste de théorie du signal et incapable de comprendre les explications du texte, je ne peux en dire plus.
- J'ai également du mal à comprendre ce que l'énoncé entend par « la répartition obtenue entre f et t ». Mais sans comprendre la question, je peux imaginer (sans grande conviction) une réponse dans laquelle le mot « linéaire » intervient.
- Il faut probablement flatter la technique employée pour dire que le spectrogramme est peu sensible au bruit...

Q 5. Nous accueillons une nouvelle variable globale, anciennement paramètre : `T`

```
def W(t, f, eta):
    teta, feta = DT/2+eta*T, Df/2+eta*Df  # oui : feta
    if abs(t-teta)<DT/2 and abs(f-feta)<Df/2:
        return 1
    else:
        return 0
```

Q 6. Sous l'hypothèse que la fonction `sum` est à éviter :

```
def enveloppe(eta, S, p, dt, df):
    nf = S.shape[0]
    somme = 0
    for i in range(nf):
        for j in range(nf):
            somme += p[i,j]*S[i,j]*W(i*dt, j*df, eta)
    return somme*dt*df
```

À la façon dont `Gini_groupes` est écrite plus tard, la multiplication par `dt*df` est peut-être attendue à chaque étape !

Q 7. On réalise une normalisation affine :

```
def normalisation(P):
    mini, maxi = min(P), max(P)
    return (P-mini)/(maxi-mini)
```

II.2 - Lecture des données

Q 8. La « variable `donnees` » (au sens : la valeur renvoyée par l'appel de la fonction...) sera ici une liste constituée de n_x listes constituées de 61 flottants :

```
[[0.02, 0.0371, ....., 0.0032, 0.],
 [0.0453, 0.0523, ..., 0,0044, 1.],
...
]
```

Q 9. Chacun des 208 enregistrements nécessite de stocker 61 flottants codés sur 64 bits dont 8 octets. On a donc besoin de 488 octets par enregistrement ; ainsi :

À une vache près, les données nécessitent 100 kilo-octets de mémoire.

Partie III - Méthode des forêts aléatoires

III.1 - Arbre de décision

Q 10. C'est un peu long, mais...

```
[6, 0.85, [3, 0.89, [3, 0.31, 1, 0], 1], [1, 0.67, [4, 0.8, 1, 1], 1]]
```

Q 11. Il y a ici 10 colonnes (notons que contrairement aux bonnes pratiques il n'y a pas d'espace après les virgules, rendant la lecture pénible...).
La première décision sera prise en comparant `a[6] = 0.7` à 0.85. Puisque $0.7 < 0.85$, on descend à gauche, puis on compare `a[3] = 0.4` à 0.89. On descend à gauche. Enfin, `a[3] = 0.4` est strictement supérieur à 0.31, ce qui nous conduit (à droite) à une feuille étiquetée par 0.

La donnée sera classée dans le groupe 0.

On note que l'arbre enraciné en le nœud 4 aurait pu être remplacé par la feuille 1. Cette étrangeté sera expliquée par la programmation « discutable » de la construction de l'arbre.

III.2 - Construction d'un arbre

Q 12. *Il est surprenant qu'une telle question soit posée sans indication. Une preuve de validité de ladite fonction est-elle requise ? A priori non, mais lesdites preuves sont-elles triviales ? L'énoncé a pudiquement oublié de demander que les résultats, en plus d'être aléatoires, soient uniformément distribués parmi ceux possibles...*
Une première solution pourrait être :

```
def indices_aleatoires(m, p_var):
    listeAlea = []
    for i in range(p_var):
        alea = randrange(m)
        while  alea in listeAlea :
            alea = randrange(m)
        listeAlea.append(alea)
    return listeAlea
```

Elle termine avec probabilité 1 et réalise en moyenne $O(p_{var})$ tests d'appartenance à des listes de taille $O(p_{var})$. Le résultat produit est une partie aléatoire... suivant une loi uniforme parmi celles-ci.
Le fait d'utiliser `in` *dans ce type de programme est un peu ennuyeux : est-ce que ça peut être considéré comme une fonctionnalité élémentaire ? Doit-on le réécrire (il n'est pas dans les annexes) ?*
On peut aussi imaginer prendre une suite croissante de `[0, 1, ..., m-p_var]` et la rendre strictement croissante en ajoutant (astuce classique... mais ultra culturelle !) i en position i !

```
def indices_aleatoires_bis(m, p_var):
    indices = [randrange(m-p_var+1) for _ in range(p_var)]
    indices.sort()
    return [indices[i]+i for i in range(p_var)]
```

On montre alors sans trop de mal [3] qu'on a décrit uniformément toutes les parties à p_{var} éléments de $[\![0, m-1]\!]$. Attention cependant : ces listes sont croissantes, donc introduisent potentiellement un biais, selon ce qu'on en fera.

Q 13. *Faisons le pari que, bien conscient de la subtilité de la question des copies de listes, le jury jettera un voile pudique sur ce point.*

```
def test_separation(ind, val, donnees):
    gauche, droite = [], []
    for ligne in donnees:
        if ligne[ind] < val:
            gauche.append(ligne)
        else:
            droite.append(ligne)
    return [gauche, droite]
```

3. C'est faux : le lecteur expérimenté aura froncé les sourcils à l'énoncé de cette propriété en grognant un « mouais, à voir » tandis que le très expérimenté sait que c'est faux et le naïf ou pressé est assez convaincu que c'est vrai !

Q 14. La variable `p` est-elle un compteur qui s'appelle astucieusement `p` ou bien une probabilité, flottant initialisé à 0 et non 0.0 comme deux lignes plus haut ?

```
def Gini_groupe(groupes):
    n_donnees = sum(len(g) for g in groupes)      # instruction 1
    gini = 0.0
    for donnees in groupes:
        taille = len(donnees)
        if taille != 0:
            gini_gr = 0.0
            for val in [0, 1]:
                p = 0  # je vais faire comme si cétait un pompteur !
                for ligne in donnees:
                    if ligne[-1] == val:
                        p += 1                    # instruction 2
                p = p                             # instruction 3; déconseillée
                gini_gr += 1 - (p/taille)**2      # instruction 4
            gini += gini_gr * taille/n_donnees    # instruction 5
    return gini
```

Q 15. *Attention : cette cascade est réalisée par un professionnel. Ne la tentez pas sur une copie !*

```
def feuille(data):  # en cas d'égalité on renvoie 0
    if 2 * sum(d[-1] for d in data) > len(data):
        return 1
    else:
        return 0
```

Le candidat prudent aura plutôt écrit quelque chose comme :

```
def feuille(data):
    nbrClasse0 = 0
    nbrClasse1 = 1
    for d in data:
        if d[-1] == 0:
            ...
```

Q 16. La condition 1 porte a priori sur le fait que le nombre d'appels récursifs (séparation au dessus du nœud courant) a atteint ou non le seuil `sep_max`. En y regardant de plus près, le cas où l'un des deux sous-arbre est vide est traité de la même façon ; profitons-en.
Si cette condition est vérifiée, le `return` permet alors de quitter le programme. Sinon, il s'agit de remplacer chaque sous-arbre soit par une feuille (s'il est de taille assez petite) soit via un appel récursif. C'est ce qui est géré par l'enchaînement maladroit de tests.

```
def construit(arbre, sep_max, taille_min, p_var, ind_rec):
    gauche, droite = arbre[2], arbre[3]
    if ind_rec == sep_max or len(gauche) == 0 or len(droite) == 0:
     # condition 1 : nb maximal de séparations atteint
     #               ou l'un des deux sous-arbre est vide
        valeur = feuille(gauche + droite)  # la feuille à placer... deux fois !
        arbre[2] = valeur
        arbre[3] = valeur
    if len(gauche) <= taille_min and len(droite) <= taille_min:
                # condition 2 : les deux sous-arbres sont trop petits
        arbre[2], arbre[3] = feuille(gauche), feuille(droite)
    if len(gauche) <= taille_min and len(droite) > taille_min:
                # condition 3 : seul l'arbre de gauche est trop petit
        ...
```

L'auteur pensait possiblement à autre chose... Personnellement j'aurais enlevé le test 2 (et l'action résultante) et remplacé la condition 3 par quelque chose ne concernant que l'arbre gauche.
Une liste ne pouvant être modifiée en un entier, les choix faits par l'énoncé imposent la possibilité de « fausses décisions », où les feuilles gauche et droite sont identiques.

III.3 - Test d'une précision sur un arbre simple

Q 17. *« Je ne suis pas bien certain d'avoir compris la définition des prédictions 1, 2 et 3 » : erreur d'énoncé, ou énoncé pas bien clair...*

Déjà, le temps de calcul est surlinéaire ; et même assez proche du quadratique. Difficile de dire si c'est un problème rédhibitoire sans précisions supplémentaires sur le contexte (puissance de calcul, temps acceptables...).

Ensuite, le taux de réussite augmente un peu avec la taille du jeu de données initiale, mais assez mollement. En supposant qu'on dispose de jeux de données sensiblement plus grand, quels taux peut-on espérer atteindre ?

Enfin, Les taux atteints, inférieurs à 90%, sont-ils acceptables ? Difficile à dire hors contexte !

III.4 - Algorithme des forêts aléatoires : « random forest »

Q 18. On descend dans l'arbre jusqu'à arriver à une feuille :

```
def prediction(arbre, donnee):
    [ind, val, gauche, droite] = arbre
    if donnee[ind] < val:  # Aller à gauche
        if isinstance(gauche, list):  # continuer récusrsivement
            return prediction(gauche, donnee)              # instruction 1
        else:  # c'est une feuille
            return gauche                                  # instruction 2
    else:  # Aller à droite
        if isinstance(droite, list):  # continuer récusrsivement
            return prediction(droite, donnee)              # instruction 3
        else:  # c'est une feuille
            return droite                                  # instruction 4
```

Autre possibilité, plus légère et permettant de voir les feuilles comme des arbres :

```
def prediction(arbre, donnee):
    if isinstance(arbre, int):     # c'est une feuille
        return arbre
    else:                          # continuer récursivement
        if donnee[arbre[0]] < arbre[1]:
            return prediction(arbre[2], donnee)
        else:
            return prediction(arbre[3], donnee)
```

Q 19. On commence par construire (une fois pour toutes) les arbres. Ensuite, pour chaque donnée, on calcule la décision majoritaire :

```
def random_forest(data_train, data_test, sep_max, taille_min, n_arbres, p_var):
    arbres = [constuire_arbre(data_train, sep_max, taille_min, p_var)
                              for _ in range(n_arbres)]
    classes = []
    for data in data_test:
        nb = [0, 0]
        for arbre in arbres:
            nb[prediction(arbre, data)] += 1
        if nb[0] >= nb[1]:
            classes.append(0)
        else:
            classes.append(1)
    return classes
```

On doit pouvoir se passer de la fonction `construire_foret`*...*

Conclusion

Q 20. Je propose deux réponses :

— Les temps de calculs restent quadratiques en la longueur de la taille du jeu de données initial. « La constante devant » diminue, mais pas énormément. Enfin le taux de succès augmente assez sensiblement (90%) tout en restant assez éloigné de 100%.

Tout ça pour ça...

— Les temps de calculs ont considérablement diminué et les taux de réussite ont considérablement augmenté : quel succès...

Random forest algorithm rocks !

Une « meilleure complexité » ce n'est pas une complexité divisée par 2 : c'est une complexité qui passe de n^2 *à* $n^{3/2}$*, ou une constante multiplicative divisée par 100 !*