TP 1 : Apprentissage supervisé avec les Iris d'Anderson

Exercice 1 : Lecture des données

Edgar Anderson était un botaniste américain spécialisé dans la génétique et l'auteur d'un jeu de données portant sur 150 échantillons d'iris appartenant à trois espèces :

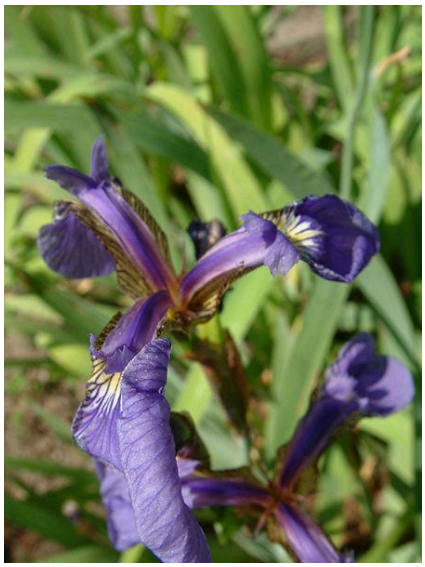

Fig. 1. – Iris setosa

Fig. 2. – Iris versicolor

Fig. 3. – Iris virginica

Pour chaque fleur (cueillies le même jour dans un même champ), il a mesuré la longueur et la largeur des sépales et des pétales en centimètres et identifié l'espèce.

Ces données ont ensuite été analysées en 1936 par le statisticien et biologiste Ronald Fisher comme exemple de sa méthode de classification, l'analyse discriminante linéaire (méthode hors programme mais très intéressante, ce sera l'exercice bonus).

Depuis, ce jeu de données est devenu un grand classique pour essayer des algorithmes d'apprentissage automatique.

J'ai séparé ce jeu de données étiquetées en un jeu de données d'entraînement de 120 fleurs qui se trouve dans le fichier `iris_jeu_entr.csv` et un jeu de données de test de 30 fleurs qui se trouve dans le fichier `iris_jeu_test.csv`.

Placez ces deux fichiers dans le même répertoire que le fichier `iris.ml`.

La fonction `lire_iris` prend en paramètre un nom de fichier `csv` et renvoie une liste de valeurs de type `fleurs` : `type fleur = {mesures : float array; etiquette : int}`

Le champ `mesures` correspond aux 4 mesures faîtes sur la fleur. Le champ `etiquette` contient la valeur 1, 2 ou 3 pour, respectivement, l'espèce « setosa », « versicolor » et « virginica ».

1) Vérifier le bon fonctionnement de `lire_iris` en affichant dans le terminal les données d'entraînement et de test.

2) À quel type de problème a-t-on affaire ici ?

Exercice 2 : Algorithme des k-plus proches voisins

3) Écrire une fonction

`k_plus_proches_voisins : int -> fleur list -> float array -> int` qui

qui prend en paramètre k, un jeu d'entraînement et les mesures d'une fleur et qui renvoie son espèce.

On utilise la distance de Manhattan, en cas d'égalité on choisit la classe du point le plus proche (parmi ceux ayant une des classes à égalité).

On prendra bien soin de factoriser le code de cette fonction (d'écrire plusieurs autres fonctions).

4) Écrire une fonction :

```
matrice_confusion : fleur list -> (float array -> int) -> int array array
```

qui prend en paramètre un jeu de test et une fonction de classification (telle que `k_plus_proche_voisin`) et qui renvoie la matrice de confusion.

Par exemple on pourra appeler cette fonction de la façon suivante (on utilise la curryfication) :

```
matrice_confusion jeu_test (k_plus_proches_voisins 5 jeu_entr)
```

5) Écrire une fonction

```
taux_erreur_global : int array array -> float
```

qui calcule le taux d'erreur global à partir d'une matrice de confusion.

6) Quelle est la meilleure valeur de k ?

Pour quelles valeurs de k observe-t-on un surapprentissage ?

7) Quelle distance fonctionne le mieux ? Euclidienne ou de Manhattan ?

8) Répondre aux questions 1 à 5 en C.

Exercice 3 : Arbres d-dimensionnels

On définit le type suivant pour représenter les arbres d-dimensionnels.

```
type arbredd = Noeud of int * int array | Feuille of int array
```

9) Écrire une fonction `creer_arbredd : int -> int array list -> arbredd` qui prend en paramètre un nombre de dimensions d est un ensemble de points et qui renvoie l'arbre d-dimensionnels de cet ensemble de points.

On choisira à chaque étape la dimensions la plus « large » (en cas d'égalité on garde la dimension dont le nombre est le plus petit).

On choisira à chaque étape le point médian de la dimension choisie (en cas d'égalité on garde le premier dans la liste).

10) Écrire une fonction `k_plus_proches_voisins2 : int -> arbredd -> float array -> int` qui implémente l'algorithme des k-plus proches voisins en utilisant un arbre d-dimensionnel.

On prendra bien soin de factoriser le code de cette fonction.

11) Vérifier qu'on obtient bien la même matrice de confusion qu'avec l'implémentation initiale et comparer le temps d'exécution.

Exercice 4 : Algorithme ID3

Comme ID3 nécessite que les attributs aient un nombre fini de valeurs possibles, on va séparer la plage de valeurs en intervalles plus petits. Pour cela, pour chaque attribut, on va prendre les valeurs minimum et maximum et séparer cet intervalle en k intervalles de même taille.

On représentera cette séparation par un tableau de $k-1$ valeurs.

Par exemple, pour $k=4$, si les valeurs d'un attribut sont toutes entre 0 et 10, le tableau représentant la séparation sera `[|2.5; 5; 7.5|]`.

Les valeurs inférieures strictement à `2.5` seront considérées comme correspondant à la valeur `0` ;
les valeurs supérieures ou égales à `2.5` et inférieures strictement à `5` comme la valeur `1` ;
les valeurs supérieures ou égales à `5` et inférieures strictement à `7.5` comme la valeur `2` ;
les valeurs supérieures ou égales `7.5` comme la valeur `3`.

> **12)** Écrire une fonction :
> `tableaux_intervalles : int -> fleur list -> float array array`
> qui prend en paramètre un entier k, un jeu d'entraînement et qui renvoie un tableau de 4 tableaux de k flottants représentant la séparation de l'intervalle de chacune des 4 mesures des fleurs.

> **13)** Écrire une fonction :
> `determiner_intervalle : float array -> float -> int`
> qui prend en paramètre un tableau représentant la séparation d'un intervalle et un flottant et qui renvoie le numéro de l'intervalle dans lequel ce flottant se trouve.

On représente les arbres de décision par le type suivant :

```
type arbre_decision = Noeud of int * arbre_decision array | Feuille of int
```

> **14)** Écrire une fonction :
> `id3 : int -> fleur list -> arbre_decision`
> qui prend en paramètre un entier k (le nombre d'intervalles de séparation) et un jeu d'entraînement et qui renvoie l'arbre de décision renvoyé par l'algorithme ID3.

> **15)** Écrire une fonction :
> `eval_arbre_decision : arbre_decision -> float array -> int`
> qui prend en paramètre un arbre de décision et les mesures d'une fleur et qui renvoie l'espèce prédite par cet arbre de décision pour cette fleur.

> **16)** Utiliser les fonctions `matrice_confusion` et `taux_erreur_global` pour déterminer la valeur de k qui donne le meilleur résultat.

Exercice 5 : L'analyse discriminante linéaire (bonus)

17) Renseignez-vous sur l'analyse discriminante linéaire et implémentez-là pour le jeu de données des Iris.