Algorithmes

Quels sont les enjeux autour des algorithmes ?

Pour débuter notre réflexion quelques questions :

1) Les algorithmes peuvent-ils tout résoudre ? OUI - NON

2) Si une solution à un problème est facile à vérifier alors il est facile d'en trouver un algorithme ? OUI - NON

3) Certains algorithmes sont très lents voir calculent une solution avec un temps infini mais finalement quand nous aurons à notre disposition des machines beaucoup plus rapide alors tous les algorithmes trouveront une solution avec un temps acceptable ! OUI - NON

Pour introduire ce sujet[1] :

Vos réponses aux questions ont-elles évoluées après cette vidéo ?

[1] https://www.youtube.com/watch?v=AgtOCNCejQ8

Algorithmes gloutons et k-NN

Les méthodes algorithmiques peuvent être regroupées en familles, parmi lesquelles figure celle des algorithmes dits gloutons.
Les algorithmes gloutons sont des algorithmes assez simples dans leur logique. Ainsi que leur nom le suggère, ils sont conçus pour prendre le maximum de ce qui est disponible à un moment donné.

Exemple introductif : Problème du voyageur

Supposons que l'on ait déterminé une certaine liste de villes dans lesquelles nous devons nous rendre et que l'on cherche un itinéraire minimisant la distance totale parcourue. On s'autorisera à visiter les villes dans n'importe quel ordre, mais aucune ne doit être négligée, et il faudra à la fin revenir à la vile de départ.
Voici une situation concrète. Nous partons de Nancy et devons nous rendre à Metz, à Paris, à Reims et à Troyes avant de retourner à Nancy. Le tableau des distances routières en kilomètres entre ces différentes villes est le suivant :

	Nancy	Metz	Paris	Reims	Troyes
Nancy		55	303	188	183
Metz	55		306	176	203
Paris	303	306		142	153
Reims	188	176	142		123
Troyes	183	203	153	123	

Le problème se ramène à trouver un ordre de visite des villes de Metz, Paris, Reims et Troyes pour lequel la somme des distances données par ce tableau pour chaque étape est aussi petite que possible. Une manière simple d'aborder le problème consiste à énumérer tous les ordres possibles, à calculer pour chacun la distance correspondante, pour sélectionner ensuite la plus petite. On a ici 4 ! (factoriel) = 24 itinéraires possibles, dont 12 sont détaillés dans ce tableau où le départ et l'arrivée à Nancy sont sous-entendus :

Itinéraires	Détail de la distance	Distance totale (km)
Metz – Paris – Reims – Troyes	55+306+142+123+183	809
Metz – Paris – Troyes – Reims	55+306+153+123+188	825
Metz – Troyes – Paris – Reims	55+203+153+142+188	741
Troyes – Metz – Paris – Reims	183+203+306+142+188	1022
Troyes – Metz – Reims – Paris	183+203+176+142+303	1007
Metz – Troyes – Reims – Paris	55+203+123+142+303	826
Metz – Reims – Troyes – Paris	55+176+123+153+303	810
Metz – Reims – Paris – Troyes	55+176+142+153+183	709
Reims – Metz – Paris – Troyes	188+176+306+153+183	1006
Reims – Metz – Troyes – Paris	188+176+203+153+303	1023
Reims – Troyes – Metz – Paris	188+123+203+306+303	1123
Troyes – Reims – Metz – Paris	183+123+176+306+303	1091

Les 12 autres itinéraires correspondent chacun à l'un des 12 du tableau emprunté dans le sens inverse, ce qui ne change en rien la distance. D'après le tableau et la carte de France on sait que l'itinéraire le plus court est Nancy – Metz – Reims – Paris – Troyes – Nancy avec 709 km.

Cependant cette technique ne fonctionne pas si on se fixe une liste plus longue de villes à visiter, le nombre d'itinéraires différents à analyser devient vite beaucoup trop important, et ce même pour les capacités de calcul d'un ordinateur. En effet, on a 12 itinéraires pour 4 villes (hors ville de départ), on obtient presque 2 millions pour 10 villes, et plus de 3 milliards pour 13 villes, et plus d'un milliard de milliard pour 20 villes. La valeur exacte est x!/2.

Les algorithmiciens qui étudient ce phénomène ont de solides raisons de penser qu'il n'existe aucun algorithme donnant la solution optimale en un temps raisonnable lorsque le nombre de villes est grand.

Face à de tels problèmes d'optimisation impossibles à explorer exhaustivement, il peut être utile de connaître des algorithmes donnant rapidement une réponse qui, sans être nécessairement optimale, resterait bonne. La méthode gloutonne présentée dans ce chapitre donne une approche simple pour concevoir de tels algorithmes souvent approximatifs mais rapides.

Algorithmes gloutons et k-NN

Algorithmes gloutons

Les algorithmes gloutons sont utilisés pour répondre à des problèmes d'optimisation, c'est-à-dire des problèmes algorithmiques comme nous venons de le voir, dans lesquels l'objectif est de trouver une solution dites la « meilleure possible » selon un certain critère, parmi un ensemble de solutions également valides mais potentiellement moins avantageuses.

Ce type d'algorithme s'applique à des problèmes ayant un très grand nombre de solutions, qu'on dispose d'une fonction mathématique évaluant la qualité de chaque solution, et qu'on cherche à avoir la meilleure solution partielle.

Résolution approchée du problème du voyageur

Appliquons l'approche gloutonne à notre problème de voyageur. Le problème considéré est la visite de l'ensemble des villes, depuis une ville de départ déterminée et avec retour à cette ville. Une solution est donc n'importe quelle séquence passant par toutes les villes intermédiaires demandées, qu'on peut ramener à la succession de choix de « quelle sera ma prochaine étape ? ».

Pour appliquer la méthode gloutonne on part de la ville de départ, puis on va à la ville la plus proche, puis à la ville la plus proche de cette dernière parmi les villes non encore visistées, et ainsi de suite. De ce fait, partant de Nancy, nous irons donc en premier lieu à Metz, distante de 55 km, ensuite à Reims en 176 km, puis à Troyes en 123 km, et enfin à Paris en 153 km, avec ultime retour à Nancy en 303 km. On complète alors notre itinéraie en 810 km. Celui-ci est plus long que l'itinéraire minimal de 709 km vu en introduction, mais reste loin des nombreuses mauvaises solutions qui demandaient plus de 1000 km. Et surout, nous avons analysé qu'un unique itinéraire au lieu des 12.

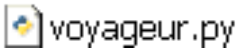
Mise en œuvre

On vous propose une réalisation en Python de l'algorithme du voyageur glouton. Les n villes à visiter sont numérotées de 0 à n-1 et les distances entre les villes sont stockées dans une liste dist à deux dimensions tel que `dist[i][j]` donne la distance de la ville numéro i à la ville numéro j.

La fonction principale voyageur prend en paramètre la liste des villes, la table des distances et le numéro de la ville de départ et renvoie le nombre de km total de l'itinéraire. Et la fonction auxiliaire plus_proche qui prend en paramètre la table des distances, le numéro de la ville actuelle et la liste des villes déjà visitées pour sélectionner la ville la plus proche de la ville actuelle parmi les villes non encore visitées, et la renvoie.

```
    def plus_proche(ville, dist, visitees):
        """Renvoie le numéro de la ville
        non encore visitées la plus proche
        de la ville actuelle, en supposant
        qu'il en existe au moins une."""
        plus_proche = ""
        for i in range(len(visitees)):
            if not visitees[i]:
                if plus_proche == "" or dist[ville][i] < dist[ville][plus_proche]:
                    plus_proche = i
        return plus_proche
def voyageur(villes, dist, depart):
    """Affiche les étapes du
    parcours glouton depuis la
    ville de départ. Et renvoie le
    nombre total de km parcouru."""
    n = len(villes)
    visitees = [False] * n
    actuelle = depart
    km = 0
    for i in range(n-1):
        visitees[actuelle] = True
        suivante = plus_proche(actuelle, dist, visitees)
        print("On va de", villes[actuelle], \
              "à", villes[suivante], \
              "en", dist[actuelle][suivante], "km")
        km = km + dist[actuelle][suivante]
        actuelle = suivante
    print("On revient de", villes[actuelle], \
          "à", villes[depart], \
          "en", dist[actuelle][depart], "km")
    km = km + dist[actuelle][depart]
    return km

#['Nancy', 'Metz', 'Paris', 'Reims', 'Troyes']
lesViles = [1, 2, 3, 4, 5]
lesDistances = [
 [0, 55, 303, 188, 183],
 [55, 0, 306, 176, 203],
 [303, 306, 0, 142, 153],
 [188, 176, 142, 0, 123],
 [183, 203, 153, 123, 0]
 ]
totalKm = voyageur(lesViles, lesDistances, 0)
print("Pour une distance totale de", totalKm, 'km.')
```

Vous retrouvez ce code sur le site nommé 

En changeant la ville de départ, écrivez les distances totales obtenues :

Ville de départ	Nancy	Metz	Paris	Reims	Troyes
Distance (km)	810				

Algorithmes gloutons et k-NN

Problème du rendu de monnaie

Vous êtes commerçant et devez rendre de la monnaie à vos clients de façon optimale, c'est-à-dire avec le nombre minimal de pièces et de billets.

Pour simplifier le problème, on suppose que les clients ne vous donnent que des sommes entières en euros (pas de centimes). De ce fait, les pièces et les billets à votre disposition sont 1€, 2€, 5€, 10€, 20€, 50€, 100€, 200€ et 500€ avec un nombre d'exemplaire illimité de chaque pièces et billets.

Exemple : Myriam vous achète un objet qui coûte 53 euros. Elle paye avec un billet de 200 euros. Vous devez donc lui rendre 147 euros. La meilleure façon de lui rendre la monnaie est de le faire avec un billet de 100, deux billets de 20, un billet de 5 et une pièce de 2.

Pour minimiser le nombre de pièces à rendre, il apparaît la stratégie suivante :

- On commence par rendre la pièce de la plus grande valeur possible.
- On déduit cette valeur de la somme (encore) à rendre.
- On recommence, jusqu'à obtenir une somme nulle.

En procédant ainsi, on se rend compte que l'on résout le problème étape par étape et qu'un choix optimal est fait à chaque étape (la pièce de plus grande valeur). Cette stratégie entre donc bien dans la catégorie des algorithmes gloutons.

Rendu de monnaie et système canonique

La méthode « usuelle » pour rendre la monnaie est celle de l'algorithme glouton : tant qu'il reste quelque chose à rendre, choisir la plus grosse pièce qu'on peut rendre (sans rendre trop). C'est un algorithme très simple et rapide, et on appelle canonique un système de pièces pour lequel cet algorithme donne une solution optimale quelle que soit la valeur à rendre.(Source Wikipédia)

Algorithmes gloutons et k-NN

On vous demande de compléter le programme Python renduMonnaie.pyné qui calcul le nombre de pièces ou de billets qui doivent être utilisés pour une somme à rendre donnée, en utilisant la stratégie gloutonne. Jusqu'à ce que la somme totale ait été rendue, cette fonction considère tour à tour les différentes coupures de la plus grosse à la plus petite. Dont le squelette vous est transmit sur le site, présent ci-contre :

```
def rendu_monnaie(s, pieces):
    """Renvoie la solution
    gloutonne du rendu de
    monnaie d'une somme s
    entière et positive. Le
    tableau pieces contient
    les valeurs des pièces à
    disposition dans l'ordre
    décroissant."""
    solution = []
    i = 0 # position de la première pièce à tester (la plus grande)
    while s > 0 and i < A completer: # tant qu'il reste de l'argent à rendre et que
    #toutes les pièces n'ont pas été testées
        valeur = pieces[i] # on prend la pièce d'indice i
        if valeur <= s: # s'il est possible de rendre la pièce
            solution.append(valeur) # on l'ajoute à solution
            s = A compléter # et on déduit sa valeur de la somme à rendre
        else:
            i = i + 1 # sinon on passe à la pièce immédiatement inférieure
    return A compléter

euros = [500, 200, 100, 50, 20, 10, 5, 2, 1]
somme = int(input("Entrez le montant à rendre: "))
rendu = rendu_monnaie(somme, euros)
print("Pour", somme, "€ on doit rendre les pièces et les billets suivants:", rendu)
```

Testez votre code avec différentes sommes :

Somme	Coupures rendues	
147€	1 billet de 100€ 2 billets de 20€	1 billet de 5€ 1 pièce de 2€

Ce n'est pas évident à démontrer, mais cet algorithme glouton fournit bien la solution optimale au problème de rendu de monnaie, dans le cas où le système de monnaie est l'euro.

Algorithmes gloutons et k-NN

Rendu de monnaie glouton dans d'autres systèmes monétaires

Modifiez le code précédent en changeant les coupures avec les valeurs 10€, 6€ et 1€. Essayez le programme pour rendre la somme 12€. Que vous affiche le programme ?

Somme	Coupures rendues
12€	

Est-ce bien la solution optimale ? Expliquez pourquoi.

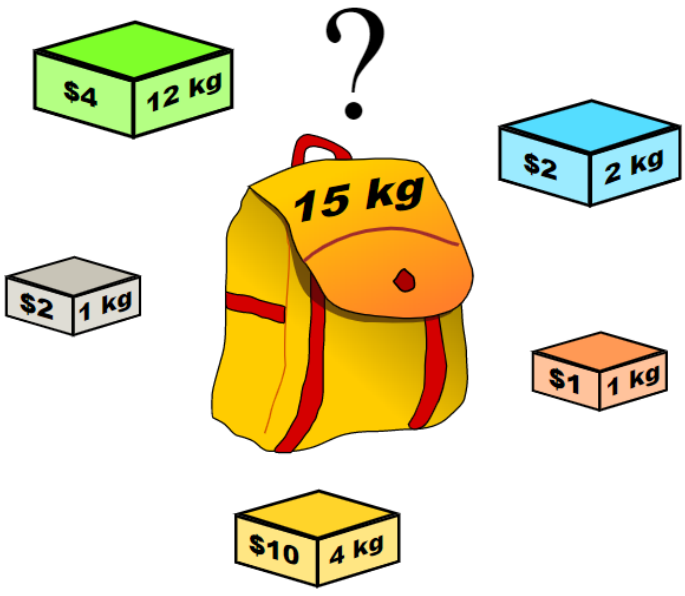
Modifiez le code précédent en changeant les coupures avec les valeurs 3€ et 2€. Essayez le programme pour rendre la somme 4€. Que vous affiche le programme ?

Somme	Coupures rendues
4€	

Est-ce bien la solution optimale ? Expliquez pourquoi.

Le problème du sac à dos

Le problème du sac à dos est l'un des 21 problèmes NP-complets de Richard Karp, exposés dans son article de 1972. Il est intensivement étudié depuis le milieu du XXe siècle et on trouve des références dès 1897, dans un article de George Ballard Mathews. La formulation du problème est fort simple, mais sa résolution est plus complexe. Les algorithmes existants peuvent résoudre des instances pratiques de taille importante. Cependant, la structure singulière du problème, et le fait qu'il soit présent en tant que sous-problème d'autres problèmes plus généraux, en font un sujet de choix pour la recherche.



Vous êtes un voleur et souhaitez emporter les objets pour maximiser la valeur totale du butin. Cependant, votre sac ne peut supporter qu'une charge maximale de 10kg. On dispose d'un sac pouvant supporter un poids maximal donné et de divers objets ayant chacun une valeur et un poids. Il s'agit de choisir les objets à emporter dans le sac afin de maximiser la valeur totale tout en respectant la contrainte du poids maximal.

Algorithmes gloutons et k-NN

On considère les objets suivants et un sac de capacité maximale de 10kg.

Objet	A	B	C	D	E	F
Masse (kg)	7	6	4	3	2	1
Valeur (€)	9100	7200	4800	2700	2600	200

Pour résoudre ce problème, il existe plusieurs stratégies possibles :

- **Stratégie 1** : On prend toujours l'objet de plus grande valeur n'excédant pas la capacité restante. Pour cela, il faut préalablement trier les objets par valeur décroissante.
- **Stratégie 2** : On prend toujours l'objet de plus faible masse. Pour cela, il faut préalablement trier les objets par masse croissante.
- **Stratégie 3** : On prend toujours l'objet de plus grand rapport valeur/masse n'excédant pas la capacité restante. Pour cela, il faut préalablement trier l'objet par rapport valeur/masse décroissant.

Appliquez les trois stratégies pour l'énoncé donné.

	Contenu du sac	Valeur (€)
Stratégie 1		
Stratégie 2		
Stratégie 3		

Y a-t-il une stratégie meilleure qu'une autre ?

Conclusion

Nous avons vu que les algorithmes gloutons fournissent une stratégie pour résoudre des problèmes d'optimisation : à chaque étape, faire le meilleur choix (local). Ils donnent rapidement une solution satisfaisante à un problème mais pas nécessairement la solution optimale puisque les choix successifs ne sont jamais remis en cause.
La stratégie de force brute permettrait à coup sûr d'obtenir une solution optimale mais devient inapplicable dès que la taille des données est trop importante. C'est pourquoi une solution gloutonne est parfois privilégiée. Il existe d'autres méthodes algorithmiques pour résoudre des problèmes d'optimisation : celles-ci seront abordées en Terminale.

Algorithmes gloutons et k-NN

Problème du jardinier

Vous êtes un jardinier passionné par les iris. Vous avez alors décidé de trier plusieurs dizaines d'iris en fonction de trois caractéristiques :

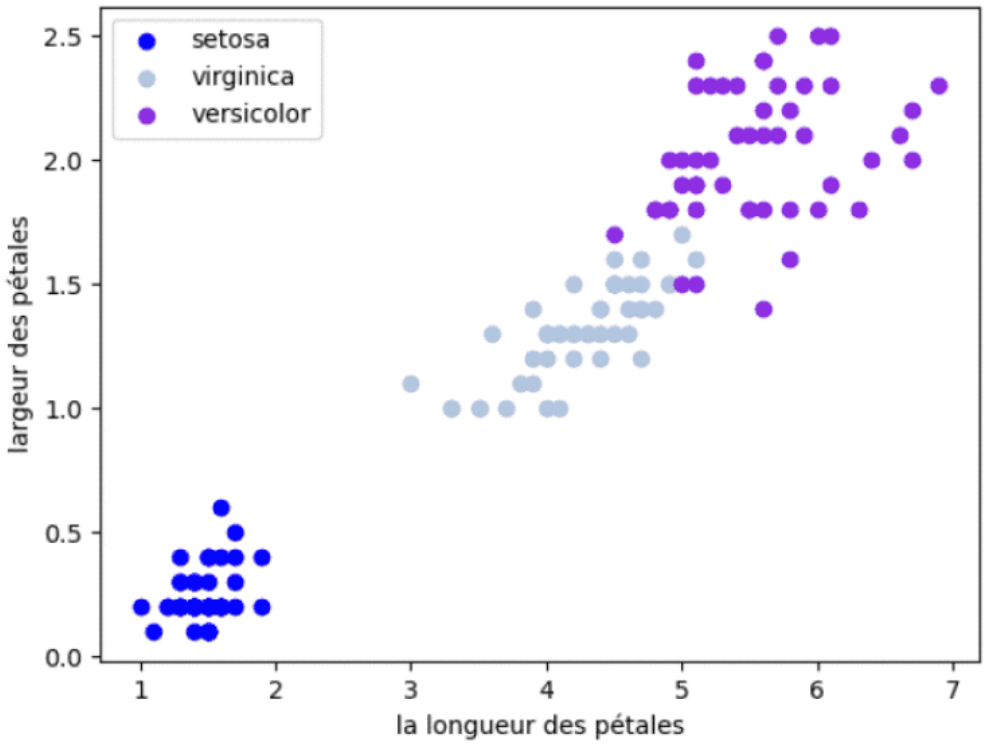
- La longueur des pétales.
- La largeur des pétales.
- L'espèce de l'iris.

Figure 1 - Iris setosa

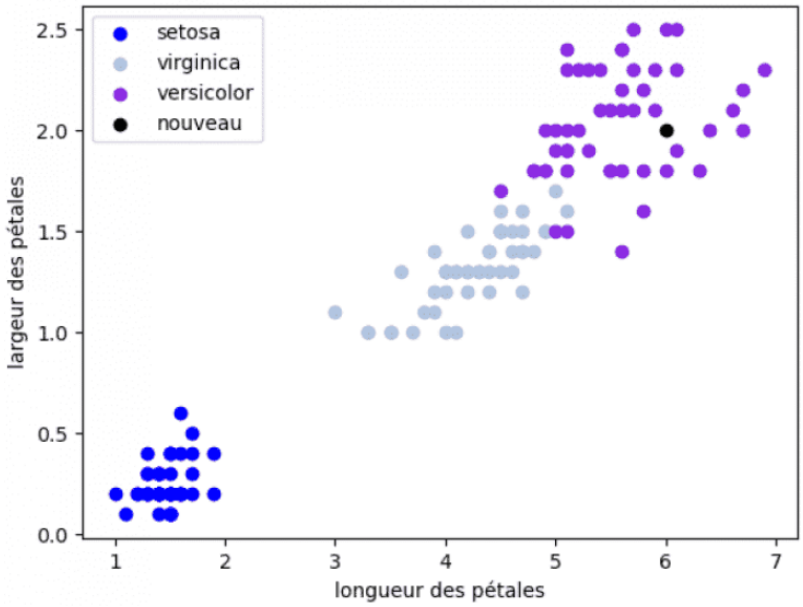
Figure 2 - Iris versicolor

Figure 3 - Iris virginica

Vous obtenez donc le graphique suivant :

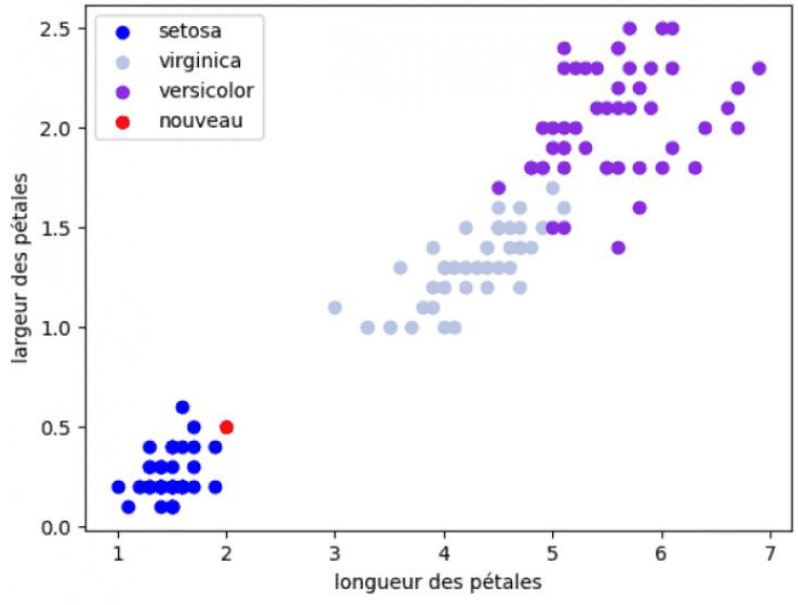


Cependant, un(e) ami(e) à vous qui ne connais rien aux iris, vous donne un iris avec sa longueur et sa largeur. On le place alors sur le graphique et nous obtenons :



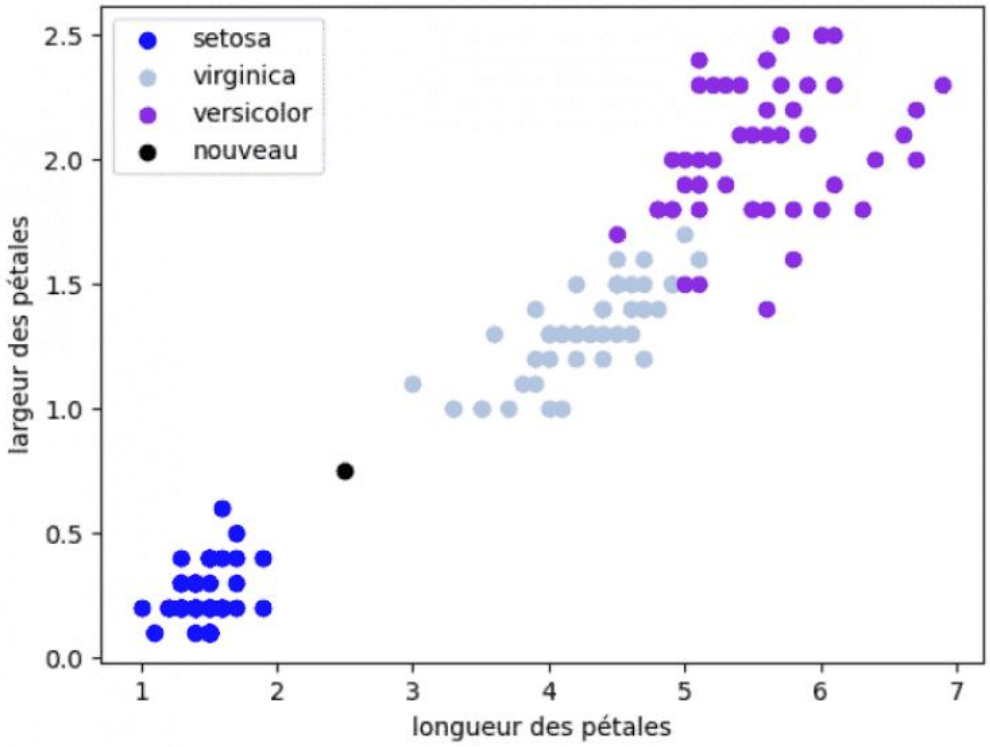
D'après vous, en vous appuyant du graphique, en déduire l'espèce de l'iris de votre ami(e) :

Quelques jours plus tard, votre ami(e) trouve encore un iris. Vous l'ajoutez au graphique :



D'après vous, en vous appuyant du graphique, en déduire l'espèce de l'iris de votre ami(e) :

Dès le lendemain, il trouve encore un nouvel iris. Comme les fois précédentes, vous placez le point :



D'après vous, en vous appuyant du graphique, en déduire l'espèce de l'iris de votre ami(e). Est-ce évident ?

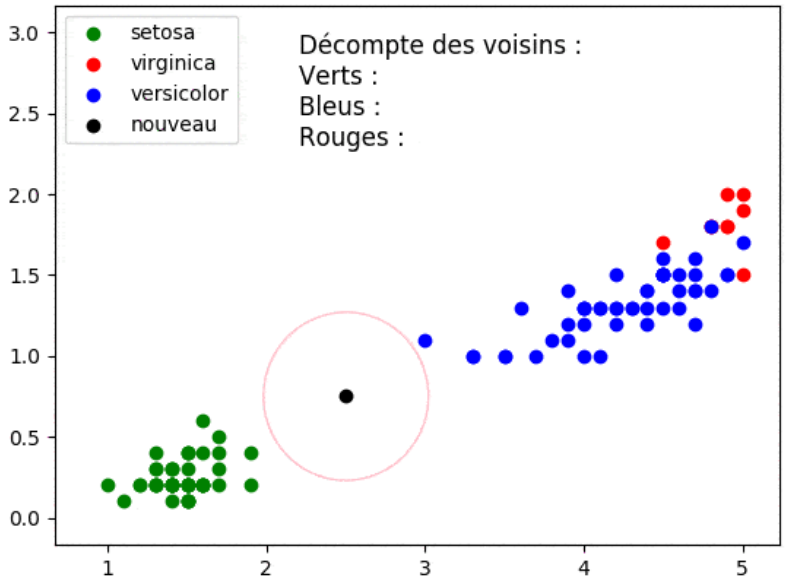
Le problème ici, c'est qu'il n'y a rien qui nous permet de décider véritablement quelle est l'espèce de ce nouvel iris. Pour décider, il nous faudrait un critère de décision :

- Moins subjectif qu'un nuage de point ou qu'un très proche.
- Algorithme pour qu'une machine puisse décider.

Algorithmes gloutons et k-NN

L'algorithme « k-NN » des k plus proches voisins

« k-NN » car en anglais l'algorithme s'appelle « k-Nearest Neighbors ».



On voit bien dans le décompte des voisins que le choix du nombre k est important. Cela fait partie des leviers de tous les spécialistes du « deep learning ». Qui est une technique de l'Intelligence Artificielle.

Avec k = 1, de quelle espèce est le nouvel iris ?

Avec k = 2, de quelle espèce est le nouvel iris ?

Avec k = 3, de quelle espèce est le nouvel iris ?

Que pouvez-vous conclure ?

Algorithmes gloutons et k-NN

Sources

https://codesturm.eu/2020/04/05/tp-knn/
http://www.info-mounier.fr/1nsi/seq10/algorithmes_gloutons.php
https://info.blaisepascal.fr/nsi-algorithmes-gloutons
https://fr.wikipedia.org/wiki/Probl%C3%A8me_du_sac_%C3%A0_dos
https://fr.wikipedia.org/wiki/21_probl%C3%A8mes_NP-complets_de_Karp