MELEPE

Zeste de savoir

Un peu de Machine Learning avec les SVM

21 août 2020

Table des matières

Connaissez-vous l'apprentissage automatique, ou *machine learning* en anglais? Concrètement, il s'agit d'une branche de l'informatique, qui englobe les algorithmes qui apprennent par eux-mêmes. Apprendre par soi-même, qu'est-ce que cela veut dire? Tout simplement qu'au début, un algorithme de d'apprentissage automatique ne sait rien faire; puis, au fur et à mesure qu'il s'entraîne sur des données, il est capable de répondre de plus en plus efficacement à la tâche qu'on lui demande de faire. Un peu comme vous et moi, en quelque sorte.

On peut ainsi classer les algorithmes en deux catégories:

— les algorithmes «classiques», où l'algorithme se contente d'appliquer une série de règles sans apprendre des cas précédents;
— les algorithmes d'apprentissage automatique, où l'algorithme est capable d'effectuer un apprentissage à partir des données déjà vues.

Aujourd'hui, nous allons parler de *machine learning*, et plus particulièrement de la famille des **SVM**.

Mais avant toute chose, un SVM, c'est quoi? Derrière ce nom barbare (qui est l'acronyme de *Support Vector Machines*, soit *machines à vecteurs support* en français, parfois traduit

par *séparateur à vaste marge* pour garder l'acronyme) se cache un algorithme d'apprentissage automatique, et qui est très efficace dans les problèmes de classification. Cela ne vous dit rien? Ça tombe bien, on va justement voir ce que c'est.

Dans un premier temps, nous allons découvrir les grands principes qui sous-tendent les SVM.
Ensuite, nous allons nous plonger dans les mathématiques qui font toute la beauté de la chose! Pour cette partie (qui peut être sautée si les maths ne sont pas votre tasse de thé), vous aurez besoin d'être familier avec les bases de l'algèbre linéaire (espace vectoriel de dimension n, produit scalaire, norme euclidienne).
Enfin, nous reviendrons à du concret, avec les généralisations du SVM, et un exemple pratique pour faire joujou!

Prêt? C'est parti!

1. Problème de classification, SVM : C'est quoi, au juste ?

Avant toute chose, nous allons commencer par établir un cadre à ce que nous allons voir. En particulier, qu'est-ce qu'un problème de classification?

Considérons l'exemple suivant. On se place dans le plan, et l'on dispose de deux catégories: les ronds rouges et les carrés bleus, chacune occupant une région différente du plan. Cependant, la frontière entre ces deux régions n'est pas connue. Ce que l'on veut, c'est que quand on lui présentera un nouveau point dont on ne connaît que la position dans le plan, l'algorithme de classification sera capable de prédire si ce nouveau point est un carré rouge ou un rond bleu.

Voici notre **problème de classification**: pour chaque nouvelle entrée, être capable de déterminer à quelle catégorie cette entrée appartient.
Autrement dit, il faut être capable de trouver la frontière entre les différentes catégories. Si on connaît la frontière, savoir de quel côté de la frontière appartient le point, et donc à quelle catégorie il appartient.

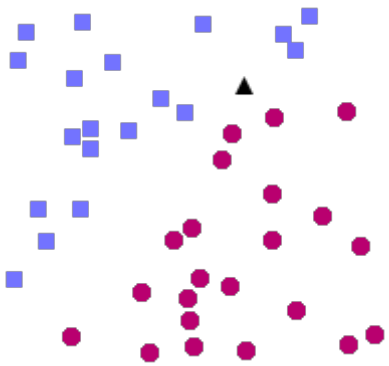


FIGURE 1.1. – Figure 1. L'objet à classer (le triangle noir) est-il un rond rouge ou bien un carré bleu ?

Le SVM est une solution à ce problème de classification[1]. Le SVM appartient à la catégorie des *classificateurs linéaires* (qui utilisent une séparation linéaire des données), et qui dispose de sa méthode à lui pour trouver la frontière entre les catégories.

Pour que le SVM puisse trouver cette frontière, il est nécessaire de lui donner des **données d'entraînement**. En l'occurrence, on donne au SVM un ensemble de points, dont on sait déjà si ce sont des carrés rouges ou des ronds bleus, comme dans la Figure 1. A partir de ces données, le SVM va estimer l'emplacement le plus plausible de la frontière: c'est la période d'**entraînement**, nécessaire à tout algorithme d'apprentissage automatique.

Une fois la phase d'entraînement terminée, le SVM a ainsi trouvé, à partir de données d'entraînement, l'emplacement supposé de la frontière. En quelque sorte, il a «appris» l'emplacement de la frontière grâce aux données d'entraînement. Qui plus est, le SVM est maintenant capable de prédire à quelle catégorie appartient une entrée qu'il n'avait jamais vue avant, et sans intervention humaine (comme c'est le cas avec le triangle noir dans la Figure 2): c'est là tout l'intérêt de l'apprentissage automatique.

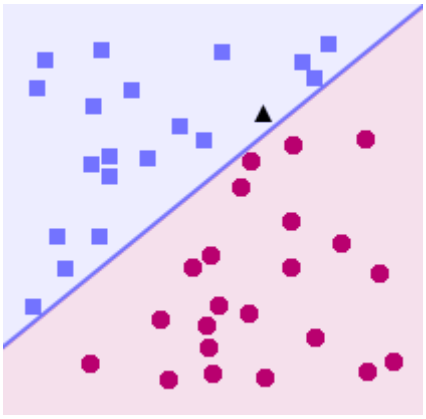


FIGURE 1.2. – Figure 2. Notre SVM, muni des données d'entraînement (les carrés bleus et les ronds rouges déjà indiqués comme tels par l'utilisateur), a tranché : le triangle noir est en fait un carré bleu.

Comme vous pouvez le constater dans la figure ci-dessus, pour notre problème le SVM a choisi une ligne droite comme frontière[2]. C'est parce que, comme on l'a dit, le SVM est un classificateur *linéaire*. Bien sûr, la frontière trouvée n'est pas la seule solution possible, et n'est probablement pas optimale non plus.

Cependant, il est considéré que, étant donné un ensemble de données d'entraînement, les SVM sont des outils qui obtiennent parmi les meilleurs résultats. En fait, il a même été prouvé que dans la catégorie des classificateurs linéaires, les SVM sont ceux qui obtiennent les meilleurs résultats.

Un des autres avantages des SVM, et qu'il est important de noter, est que ces derniers sont très efficaces quand on ne dispose que de peu de données d'entraînement: alors que d'autres algorithmes n'arriveraient pas à généraliser correctement, on observe que les SVM sont beaucoup plus efficaces. Cependant, quand les données sont trop nombreuses, le SVM a tendance à baisser en performance.

Maintenant que nous avons défini notre cadre, nous pouvons partir à l'assaut des SVM. En route!

2. Les SVM, dans les grandes lignes

Bon, nous savons donc que le but, pour un SVM, est d'apprendre à bien placer la frontière entre deux catégories. Mais comment faire? Quand on a un ensemble de points d'entraînement, il existe plusieurs lignes droites qui peuvent séparer nos catégories. La plupart du temps, il y en a une infinité… Alors, laquelle choisir?

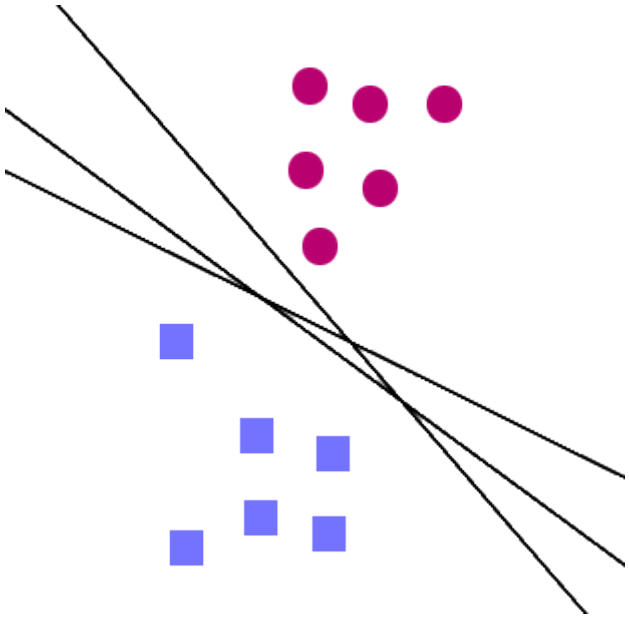
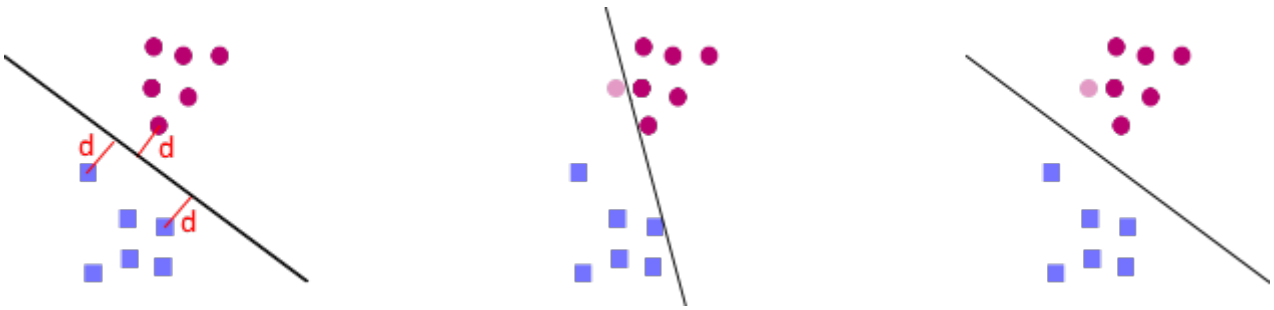


FIGURE 2.3. – Figure 3 : Pour un ensemble de données d'entraînement, il existe plusieurs frontières possibles

Intuitivement, on se dit que si on nous donne un nouveau point, très proche des ronds rouges, alors ce point a de fortes chances d'être un rond rouge lui aussi. Inversement, plus un point est près des carrés bleus, plus il a de chances d'être lui-même un carré bleu. Pour cette raison, un SVM va placer la frontière aussi loin que possible des carrés bleus, mais également aussi loin que possible des ronds rouges.



1. Les SVM peuvent aussi servir à d'autres problèmes d'apprentissage automatique, appelés *problèmes de régression*. Cela dit, nous n'en parlerons pas ici.
2. Il est possible de faire mieux que des frontières linéaires avec les SVM, mais nous verrons ça un peu plus tard, dans la partie suivante.

FIGURE 2.4. – Figure 4 : A gauche, on maximise la distance d entre la frontière et les points d'entraînement. Au centre, une frontière non-optimale, qui passe très près des points d'entraînement. Cette frontière classe de façon erronée le rond rouge clair comme un carré bleu ! Au contraire, à droite, la frontière optimale (qui maximise d) classe bien le rond rouge clair comme un rond rouge.

Comme on le voit dans la figure 4, c'est bien la frontière la plus éloignée de tous les points d'entraînement qui est optimale, on dit qu'elle a la meilleure **capacité de généralisation**. Ainsi, le but d'un SVM est de trouver cette frontière optimale, en maximisant la distance entre les points d'entraînement et la frontière.

Les points d'entraînement les plus proches de la frontière sont appelés **vecteurs support**, et c'est d'eux que les SVM tirent leur nom: SVM signifie *Support Vector Machine*, ou Machines à Vecteur Support en français. Support, parce que ce sont ces points qui «supportent» la frontière. Vecteurs, parce que… on en reparlera plus tard, dans les explications mathématiques (quel teasing !).

Ça, c'est pour les principes de base. En théorie, ça suffit, mais en pratique… En effet, les SVM sont conçus de telle manière que la frontière est forcément, dans notre exemple, une ligne droite. Et ça, c'est loin d'être suffisant pour la plupart des cas.

Considérons l'exemple suivant.

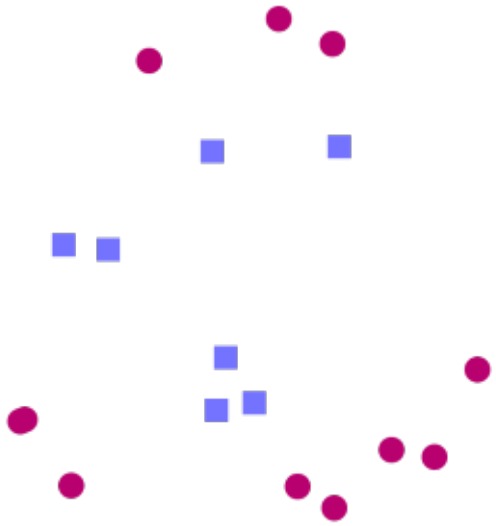


FIGURE 2.5. – Figure 5 : Comment séparer les deux catégories avec une ligne droite ?

Puisque les carrés sont entourés de ronds de toute part, il est impossible de trouver de ligne droite qui soit une frontière: on dit que les données d'entraînement ne sont pas **linéairement séparables**. Cependant, imaginez qu'on arrive à trouver une transformation qui fasse en sorte que notre problème ressemble à ça:

FIGURE 2.6. – Figure 6 : Les mêmes points d'entraînement, après transformation

A partir de là, il est facile de trouver une séparation linéaire. Il suffit donc de trouver une transformation qui va bien pour pouvoir classer les objets. Cette méthode est appelée **kernel trick**, ou astuce du noyau en français. Vous vous en doutez, toute la difficulté est de trouver la bonne transformation…

Bien! Nous savons maintenant quel est le but d'un SVM, et quelles sont les grandes idées qui sous-tendent l'ensemble. Il est maintenant temps de se plonger dans le bain, avec une formalisation du problème. Si les maths ne vous attirent pas, ne partez pas tout de suite! Vous pouvez directement passer à la partie «Cas non-séparable» et les suivantes, où nous allons voir des améliorations du SVM. Dans le cas contraire, suivez le guide.

3. Formalisme des SVM

De façon plus générale que dans les exemples donnés précédemment, les SVM ne se bornent pas à séparer des points dans le plan. Ils peuvent en fait séparer des points dans un espace de dimension quelconque. Par exemple, si on cherche à classer des fleurs par espèce, alors que l'on connaît leur taille, leur nombre de pétales et le diamètre de leur tige, on travaillera en dimension 3.
Un autre exemple est celui de la reconnaissance d'image: une image en noir et blanc de 28*28 pixels contient 784 pixels, et est donc un objet de dimension 784. Il est ainsi courant de travailler dans des espaces de plusieurs milliers de dimensions.

Fondamentalement, un SVM cherchera simplement à trouver un hyperplan qui sépare les deux catégories de notre problème.

Précisions sur les hyperplans
Dans un espace vectoriel de dimension finie n, un hyperplan est un sous-espace vectoriel de dimension $n-1$. Ainsi, dans un espace de dimension 2 un hyperplan sera une droite, dans un espace de dimension 3 un hyperplan sera un plan, etc.
Soit un espace vectoriel E de dimension n. L'équation caractéristique d'un hyperplan est de la forme $w_1x_1 + w_2x_2 + \cdots + w_nx_n = 0$, où $w_1, \ldots, w_n$ sont des scalaires. Par définition, tout vecteur $x = \begin{pmatrix} x_1 \\ \vdots \\ x_n \end{pmatrix} \in E$ vérifiant l'équation appartient à l'hyperplan. Par exemple,

en dimension 2, $ax + by = 0$ est bien l'équation caractéristique d'une droite vectorielle (qui passe par l'origine).
De plus, un hyperplan sépare complètement l'espace vectoriel en deux parties distinctes. Ainsi, une ligne droite sépare le plan en deux régions distinctes; et en dimension 3, c'est le plan qui joue le rôle de ce séparateur: il y a un demi-espace «d'un côté» du plan, et un autre demi-espace «de l'autre côté» du plan. Avec un hyperplan, il est donc possible de diviser notre espace vectoriel en deux catégories distinctes: tout pile ce qu'il nous faut pour notre problème de classification!

Comme vous pouvez le constater, un hyperplan vectoriel passe toujours par 0. C'est pour cette raison qu'on utilisera un **hyperplan affine**, qui n'a pas quant à lui obligation de passer par l'origine.

Ainsi, si l'on se place dans $\mathbb{R}^n$, pendant son entraînement le SVM calculera un hyperplan vectoriel d'équation $w_1 \cdot x_1 + w_2 \cdot x_2 + \cdots + w_n \cdot x_n = 0$, ainsi qu'un scalaire (un nombre réel) b. C'est ce scalaire b qui va nous permettre de travailler avec un hyperplan affine, comme nous allons le voir.

Le vecteur $w = \begin{pmatrix} w_1 \\ \vdots \\ w_n \end{pmatrix}$ est appelé **vecteur de poids**, le scalaire b est appelé **biais**.

Une fois l'entraînement terminé, pour classer une nouvelle entrée $x = \begin{pmatrix} a_1 \\ \vdots \\ a_n \end{pmatrix} \in \mathbb{R}^n$, le SVM regardera le signe de:

$$h(x) = w_1 a_1 + \cdots + w_n a_n + b = \sum_{i=1}^{n} w_i \cdot a_i + b = w^\top \cdot x + b$$

Si $h(x)$ est positif ou nul, alors x est d'un côté de l'hyperplan affine et appartient à la première catégorie, sinon x est de l'autre côté de l'hyperplan, et donc appartient à la seconde catégorie.

En résumé, on souhaite savoir, pour un point x, s'il se trouve d'un côté ou de l'autre de l'hyperplan. La fonction h nous permet de répondre à cette question, grâce à la classification suivante:

$$\begin{cases} h(x) \geq 0 \implies x \in \text{catégorie 1} \\ h(x) < 0 \implies x \in \text{catégorie 2} \end{cases}$$

4. Entraînement des SVM

Ainsi, étant donnés un hyperplan de vecteur de poids w, et de biais b, nous pouvons calculer si un point x_k appartient à telle ou telle catégorie, grâce au signe de $h(x_k)$.

En particulier, supposons que l'on assigne à tout point x_k un label l_k qui vaut 1 si x_k appartient à la première catégorie, et -1 si x_k appartient à la seconde catégorie. Alors, si le SVM est correctement entraîné, on a toujours $l_k h(x_k) \geq 0$, c'est-à-dire $l_k(w^\top \cdot x_k + b) \geq 0$.
Le but d'un SVM, lors de l'entraînement, est donc de trouver un vecteur de poids w et un biais b tels que, pour tout x_k de label l_k appartenant aux données d'entraînement, $l_k(w^\top \cdot x_k + b) \geq 0$.
Autrement dit, de trouver un hyperplan séparateur entre les deux catégories.

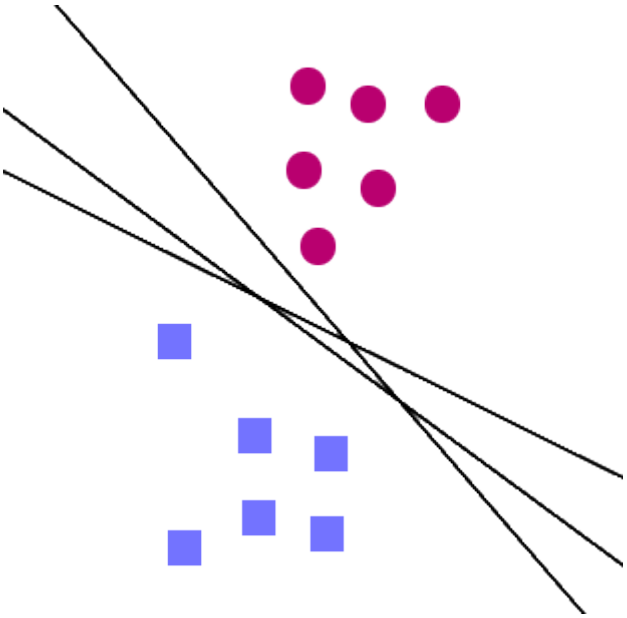


FIGURE 4.7. – Figure 7 : Quel hyperplan choisir ?

Comme on l'a vu précédemment, on choisira l'hyplerplan qui maximise la **marge**, c'est-à-dire la distance minimale entre les vecteurs d'entraînement et l'hyperplan. De tels vecteurs situés à la distance minimale sont appelés **vecteurs supports**.

i

C'est de là que vient le nom des SVM. En effet, on peut prouver que l'hyperplan donné par un SVM ne dépend que des vecteurs supports, c'est donc tout naturellement qu'on a appelé cette structure les *Support Vectors Machines*, c'est-à-dire les machines à vecteur support.

4.1. Calcul de la marge

Si l'on prend un point $x_k \in \mathbb{R}^n$, on peut prouver que sa distance à l'hyperplan de vecteur support w et de biais b est donnée par[3]:

$$\frac{l_k(w^\top \cdot x_k + b)}{\|w\|}$$

où $\|w\|$ désigne la norme euclidienne de w. La marge d'un hyperplan de paramètres (w, b) par rapport à un ensemble de points (x_k) est donc $\min_k \frac{l_k(w^\top \cdot x_k + b)}{\|w\|}$. Pour rappel, la marge est la distance minimale de l'hyperplan à un des points d'entraînement.

4.2. Maximisation de la marge

On veut trouver l'hyperplan de support w et de biais b qui permettent de maximiser cette marge, c'est-à-dire qu'on veut trouver un hyperplan avec la plus grande marge possible. Cela permettra, intuitivement, d'être tolérant face aux petites variations.
Cette intuition est justifiée: en 1995, le Russe Vladimir Vapnik a prouvé que cette maximisation produit un hyperplan optimal, c'est-à-dire qui donnera lieu au moins d'erreurs possible (on parle de *capacité de généralisation maximale*).

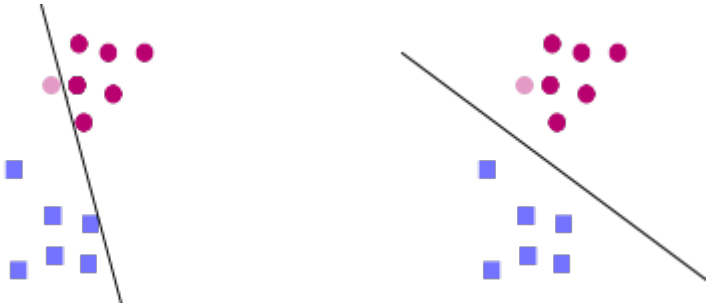


FIGURE 4.8. – Figure 8 : La capacité de généralisation est meilleure à droite (avec l'hyperplan optimal) qu'à gauche

Puisque l'on cherche l'hyperplan qui maximise la marge, on cherche l'unique hyperplan dont les paramètres (w, b) sont donnés par la formule: $\arg\max_{w,b} \min_k \frac{l_k(w^\top \cdot x_k + b)}{\|w\|}$.

4.3. Normalisation et résolution du problème

Même si on peut montrer que l'hyperplan optimal est unique, il existe plusieurs couples (w, b) qui décrivent ce même hyperplan[4]. On décide de ne considérer que l'unique paramétrage (w, b) tel que les vecteurs support x_s vérifient $l_s(w^\top \cdot x_s + b) = 1$. Par conséquent, $\forall k,\ l_k(w^\top \cdot x_k + b) \geq 1$, et l'égalité est atteinte si x_k est un vecteur support. Dit autrement, cette normalisation sur w et b permet de garantir que la marge $\min_k \frac{l_k(w^\top \cdot x_k + b)}{\|w\|}$ est alors de $ \frac 1{|w|}$.

Le problème d'optimisation de la marge $\arg\max_{w,b} \min_k \frac{l_k(w^\top \cdot x_k + b)}{\|w\|}$ se simplifie ainsi en $\arg\max_{w,b} \frac{1}{\|w\|}$, tout en gardant en tête nos hypothèses de normalisation, à savoir $\forall k,\ l_k(w^\top \cdot x_k + b) \geq 1$.

On se retrouve donc avec le problème suivant: $\begin{cases} \text{maximiser} & \frac{1}{\|w\|} \\ \text{sous les contraintes} & \forall k,\ l_k(w^\top \cdot x_k + b) \geq 1 \end{cases}$

Que l'on peut reformuler de la façon suivante: $\begin{cases} \text{minimiser} & \|w\| \\ \text{sous les contraintes} & \forall k,\ l_k(w^\top \cdot x_k + b) \geq 1 \end{cases}$

Que, pour des raisons pratiques, on reformule à nouveau: $\begin{cases} \text{minimiser} & \frac{\|w\|^2}{2} \\ \text{sous les contraintes} & \forall k,\ l_k(w^\top \cdot x_k + b) \geq 1 \end{cases}$

Ce genre de problème est appelé **problème d'optimisation quadratique**, et il existe de nombreuses méthodes pour le résoudre. Dans le cas présent, on utilise la méthode des multiplicateurs de Lagrange, que le lecteur intéressé pourra consulter sur ce PDF (en anglais).

Cette résolution donnera une valeur optimale pour w, mais rien pour b. Pour retrouver b, il suffit de se rappeler que pour les vecteurs support, $l_s(w^\top \cdot x_s + b) = 1$. On en déduit donc que b est tel que $\min_k l_k(w^\top \cdot x_k + b) = 1$.

Notre SVM est à présent entraîné, et nous pouvons l'utiliser pour classer des données qu'il n'avait jusqu'à présent jamais vues (en utilisant notre fonction de classification h). Félicitations! Vous savez désormais comment fonctionne un SVM, comment l'entraîner, et comment s'en servir.

5. Systèmes non linéaires ; astuce du noyau

Abordons maintenant le problème des données non linéairement séparables. Pour rappel, des données sont non linéairement séparables quand il n'existe pas d'hyperplan capable de séparer correctement les deux catégories. Ce qui, d'ailleurs, arrive quasiment tout le temps en pratique.

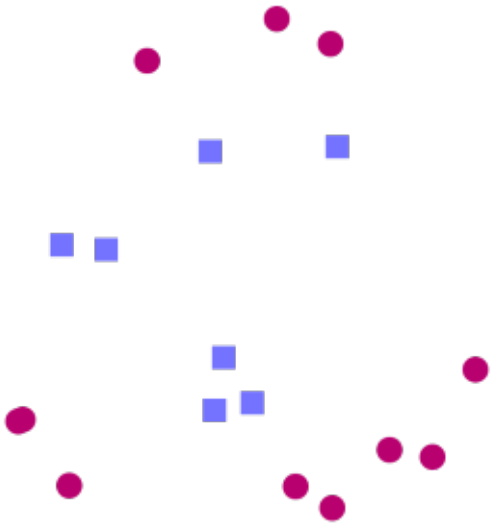


FIGURE 5.9. – Figure 9 : Données d'entraînement non linéairement séparables

5.1. Plongée en dimension supérieure

Pour contourner le problème, l'idée est donc la suivante: il est impossible de séparer linéairement les données dans notre espace vectoriel? Qu'importe! Essayons dans un autre espace. De façon générale, il est courant ne pas pouvoir séparer les données parce que l'espace est de trop petite

3. La distance d'un point x à un hyperplan w, avec un biais b, est donnée par la formule $\frac{|w^\top \cdot x + b|}{\|w\|}$. Or, montre facilement que pour un hyperplan séparateur, $\forall k$, $l_k(w^\top \cdot x_k + b) = |w^\top \cdot x_k + b|$, d'où la formule.

4. En effet, vous pouvez vérifier par vous-mêmes que si un hyperplan H est décrit par les paramètres (w, b), alors H est également décrit par les paramètres $(\alpha w, \alpha b)$, pour tout réel non nul α.

dimension[5]. Si l'on arrivait à transposer les données dans un espace de plus grande dimension, on arriverait peut-être à trouver un hyperplan séparateur.

Je sens que j'en ai perdu quelques-uns en cours de route.
Considérons l'exemple suivant, qui vous aidera à mieux comprendre. Je souhaite construire un SVM qui, à partir de la taille d'un individu, me dit si cet individu est un jeune adolescent (entre 12 et 16 ans, carrés bleus) ou non (ronds rouges).

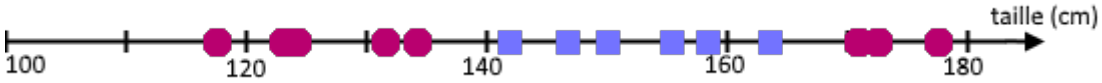


FIGURE 5.10. – Figure 10. Entre 10 et 12 ans, on mesure entre 120 et 140 cm ; entre 12 et 16 ans, entre 140 et 165 cm ; entre 16 et 18 ans on mesure plus de 165 cm. Peut-on trouver un hyperplan qui sépare les jeunes de 12-16 ans des autres ?

On constate que l'on ne peut pas trouver d'hyperplan séparateur (ici, on est en dimension 1, un hyperplan séparateur est donc un simple point). Ce qui est embêtant: si l'on ne peut pas trouver d'hyperplan séparateur, notre SVM ne sera pas capable de s'entraîner, et encore moins de classer de nouvelles entrées…

On essaie donc de trouver un nouvel espace, généralement de dimension supérieure, dans lequel on peut projeter nos valeurs d'entraînement, et dans lequel on pourra trouver un séparateur linéaire.

Dans cet exemple, je vous propose d'utiliser la projection $\varphi : x \mapsto \left(\frac{x-150}{10}, \left(\frac{x-150}{10}\right)^2\right)$. On passe donc de l'espace vectoriel $\mathbb{R}$, de dimension 1, à l'espace vectoriel $\mathbb{R}^2$, de dimension 2.

On se retrouve donc avec les données d'entraînement suivantes:

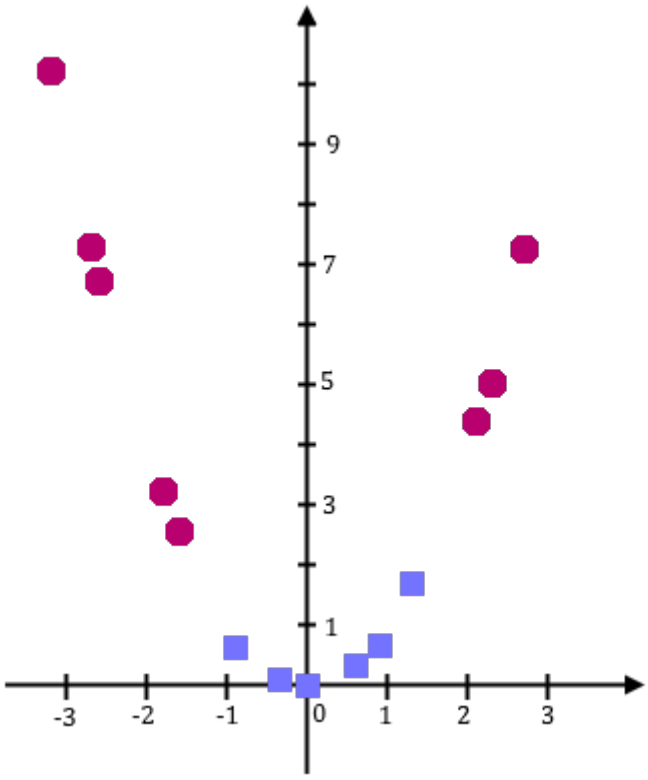


FIGURE 5.11. – Figure 11. Les données d'entraînement, après projection

On voit que les données sont alors linéairement séparables: notre SVM va pouvoir fonctionner!

Plus formellement, l'idée de cette redescription du problème est de considérer que l'espace actuel, appelé *espace de description*, est de dimension trop petite; alors que si on plongeait les données dans un espace de dimension supérieure, appelé *espace de redescription*, les données seraient linéairement séparables.

Bien sûr, il ne suffit pas d'ajouter des dimensions à l'espace de description, cela ne résoudrait d'ailleurs rien. Il faut au contraire redistribuer les points depuis l'espace de description vers l'espace de redescription, à l'aide d'une fonction φ, nécessairement non-linéaire.

On définit l'opération de redescription des points x de E vers E' par l'opération: $\begin{array}{rcl} \varphi : E & \to & E' \\ x & \mapsto & \varphi(x) \end{array}$

Dans ce nouvel espace E', on va tenter d'entraîner le SVM, comme nous l'aurions fait dans l'espace E. Si les données y sont linéairement séparables, c'est gagné! Par la suite, si l'on veut classer x, il suffira de classer $\varphi(x)$: on obtient ainsi un SVM fonctionnel.

5.2. L'astuce du noyau pour simplifier les calculs

Malheureusement, plus on se plonge dans une grande dimension, plus les calculs sont longs. Alors, si on se plonge en dimension infinie… Néanmoins, il est possible de simplifier les calculs. Quand on pose le problème d'optimisation quadratique dans l'espace E', on s'aperçoit que les seules apparitions de φ sont de la forme $\varphi(x_i)^\top \cdot \varphi(x_j)$. De même dans l'expression de la fonction de classification h'. Par conséquent, il n'y a pas besoin de connaître expressément E', ni même φ: il suffit de connaître toutes les valeurs $\varphi(x_i)^\top \cdot \varphi(x_j)$, qui ne dépendent donc que des x_i.

On appelle donc **fonction noyau** la fonction $K : E \times E \to \mathbb{R}$ définie de la façon suivante: $K(x, x') = \varphi(x)^\top \cdot \varphi(x')$. A ce moment, le calcul de l'hyperplan séparateur dans E' ne nécessite ni la connaissance de E', ni de φ, mais seulement de K.

Par exemple, supposons que E soit l'espace de description, de dimension n; et que l'espace de redescription E' soit de dimension n^2. Le calcul de $K(x_i, x_j)$ se fait en $O(n)$, tandis que le calcul de $\varphi(x_i)^\top \cdot \varphi(x_j)$, qui donne le même résultat, se fait en $O(n^2)$, d'où un avantage en temps de calcul immédiat.

C'est de ce constat qu'arrive le **kernel trick**, ou astuce du noyau. Puisque l'on n'a besoin que de connaître K, pourquoi ne pas utiliser un K quelconque, qui correspond à une redescription $E \to E'$ quelconque, et de résoudre le problème de séparation linéaire sans même se soucier de l'espace de redescription dans lequel on évolue?
Grâce au théorème de Mercer , on sait qu'une condition suffisante pour qu'une fonction K soit une fonction noyau est que K soit continue, symétrique et semi-définie positive.

Noyau symétrique semi-défini positif
Une fonction φ est dite symétrique si et seulement si, $\forall x, y,\ \varphi(x, y) = \varphi(y, x)$ (dans le cas où φ est à deux variables).

Une fonction symétrique est dite semi-définie positive si et seulement si, pour tout ensemble fini $(x_1, \ldots, x_n)$, et pour tous réels $(c_1, \ldots, c_n)$, $\sum_{i,j} c_i c_j K(x_i, x_j) \geq 0$.

Par exemple, le produit scalaire usuel $\langle a, b\rangle \mapsto \sum_{i=0}^{N} a_i b_i$ est semi-défini positif, car il est symétrique et $\forall (x_1, \ldots, x_n), \forall (c_1, \ldots, c_n), \sum_{i=0}^{n}\sum_{j=0}^{n} c_i c_j \langle x_i, x_j\rangle = \left\|\sum_{i=0}^{n} c_i x_i\right\|^2 \geq 0$.

Ainsi, il est possible d'utiliser n'importe quelle fonction noyau afin de réaliser une redescription dans un espace de dimension supérieure. La fonction noyau étant définie sur l'espace de description E (et non sur l'espace de redescription E', de plus grande dimension), les calculs sont beaucoup plus rapides.

Voici une liste non exhaustive de noyaux couramment utilisés.

- Le noyau polynomial, $K(x, x') = (\alpha x^\top \cdot x' + \lambda)^d$
- Le noyau gaussien, $K(x, x') = \exp\left(-\frac{\|x-x'\|^2}{2\sigma^2}\right)$
- Le noyau laplacien, $K(x, x') = \exp\left(-\frac{\|x-x'\|}{\sigma}\right)$
- Le noyau rationnel, $K(x, x') = 1 - \frac{\|x'-x\|^2}{\|x'-x\|^2+\sigma}$

Chacun possède ses avantages et ses inconvénients, qui sont décrits dans cet article (en anglais). C'est donc à l'utilisateur de choisir le noyau, et les paramètres de ce noyau, qui correspond le mieux à son problème. Et comment choisir son noyau, me demanderez-vous? Eh bien… Je n'ai pas de méthode à vous donner, ça dépend énormément de votre jeu de données. Avec l'expérience, certains noyaux deviennent plus intuitifs que d'autres dans certains problèmes.

Pour résumer, voici les quatre grandes idées à la base du *kernel trick*:

- Les données décrites dans l'espace d'entrée E sont projetées dans un espace de redescription E'.
- Des régularités linéaires sont cherchées dans cet espace E'.
- Les algorithmes de recherche n'ont pas besoin de connaître les coordonnées des projections des données dans E', mais seulement leurs produits scalaires.
- Ces produits scalaires peuvent être calculés efficacement grâce à l'utilisation de fonctions noyau.

6. Cas non-séparable

Nous savons ainsi que quand les données ne sont pas linéairement séparables, on peut toujours recourir au kernel trick, qui avec un peu de chance, nous permettra de séparer linéairement nos données. Est-ce toujours possible de trouver un bon noyau?

5. En fait, on peut montrer qu'avec des données aléatoires, quand on travaille en dimension N, il devient de plus en plus dur de trouver un hyperplan séparateur quand on dépasse N valeurs d'entraînement. C'est le *Cover's theorem* (en anglais)

Malheureusement, la réponse est non. Il existe des ensembles qui sont non séparables, peu importe les transformations qu'on lui fasse subir. Pour s'en convaincre, reprenons l'exemple précédent, avec les tailles des jeunes ados. D'un côté, René, 14 ans, est un ado dans toute sa splendeur, et mesure 1m50. De l'autre côté, Lucile n'est pas bien grande: malgré ses 20 ans, elle ne mesure que 1m50.
Dans un tel cas, il est impossible de différencier René de Lucile à partir de leur taille: aucun SVM ne pourra les séparer; le problème est donc non séparable.

De plus, dans la plupart des cas en pratique, on n'arrive pas à trouver de kernel trick qui permette de séparer parfaitement les données. Et puis, on ne va pas non plus s'amuser à tous les tester un par un…

Pire encore, si l'ensemble de données d'entraînement n'est pas linéairement séparable, alors le problème d'optimisation quadratique sur lequel repose le SVM n'a pas de solution. Par conséquent, le SVM ne peut pas choisir d'hyperplan séparateur (de frontière), et donc ne marchera pas du tout. Et comme, dans la vraie vie, les données d'entraînement sont rarement linéairement séparables, même après kernel trick, on n'est pas sortis de l'auberge…

Mais ne vous inquétez pas! Il est possible d'adapter le SVM, en rajoutant une notion de **variable ressort** (*slack variable*), qui autorise les erreurs de classification. Chaque erreur aura un coût, et le SVM tente alors de trouver l'hyperplan séparateur qui minimise le coût associé aux erreurs de classification, tout en maximisant la marge comme avant. Ainsi, notre SVM est capable de retourner un hyperplan séparateur qui marche (à peu près), que les données d'entraînement soient linéairement séparables ou non[6].

En fait, dans la pratique, tout le monde utilise des SVM qui autorisent les erreurs de classifications: c'est littéralement impossible de travailler autrement en apprentissage automatique, que ce soit avec les SVM ou d'autres outils.

Je vous passe les détails mathématiques cette fois-ci. Si vous êtes curieux, vous pouvez regarder comment ça marche à partir de la page 19 de ce cours de Stanford (en anglais).

7. Tester l'entraînement d'un SVM

Maintenant que votre SVM est entraîné avec toutes les données d'entraînement, comment faire pour être sûr que votre SVM ne vous raconte pas de salades (c'est-à-dire, comment savoir si le SVM a bien été entraîné et ne commet pas trop d'erreurs)?

La solution est de séparer aléatoirement ses données en deux groupes distincts.

Le premier groupe, appelé **ensemble d'entraînement**, sera utilisé pour entraîner notre SVM. Le second groupe, appelé **ensemble de test**, n'est pas utilisé lors de l'entraînement. On va en fait tester notre SVM sur chacune des entrées de l'ensemble de test: combien d'entrées on été correctement classées, une catégorie est-elle mieux reconnue que l'autre…?

6. Un SVM utilisant des *slack variables* peut tout à fait faire des erreurs de classification sur un ensemble de données d'entraînement qui est linéairement séparable (ça arrive quand le gain sur la marge est supérieur au coût engendré par les erreurs de classification). C'est particulièrement pratique pour ne pas être trop affecté par les valeurs extrêmes ou aberrantes (appelées *outliers* en anglais).

L'ensemble d'entraînement regroupe typiquement 80% de nos données, et l'ensemble de test contient les 20% restants. Il est capital que les données de test n'aient pas été utilisées pour entraîner le SVM, cela fausserait le résultat.

Pour plus de vérification, on peut utiliser une méthode supplémentaire, qui s'appelle la **validation croisée**. Ce tutoriel se voulant, originellement, être court (courage, c'est bientôt fini 🍊), je vous redirige vers l'article Wikipédia . Plus d'informations sur la version anglaise .

8. SVM multiclasse

Enfin, jusque ici, nous n'avons vu que des SVM permettant de classer les données en deux catégories. Mais que se passe-t-il si nous avons plus que deux catégories? Par exemple, supposons que je veuille entraîner un SVM à distinguer les abeilles, les guêpes et les frelons (savoir ce qui vient de me piquer ne va pas diminuer ma douleur, mais c'est toujours bon à savoir 🍊), j'ai besoin d'un SVM qui sache distinguer trois classes différentes.

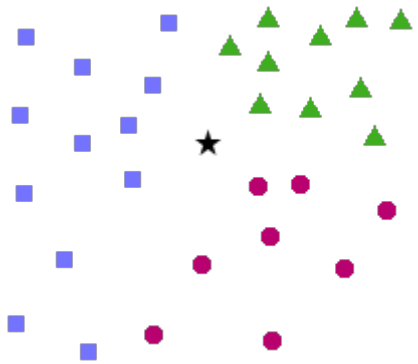


FIGURE 8.12. – Figure 12 : L'étoile noire est-elle une abeille (carré bleu), une guêpe (triangle vert), ou un frelon (rond rouge) ?

Si l'on reprend la définition rigoureuse de la frontière d'un SVM (un hyperplan qui sépare l'espace vectoriel en deux), on est bien embêtés: un hyperplan qui sépare l'espace vectoriel en trois (ou plus), ça n'existe pas… Mais comme bien souvent, les chercheurs qui se sont penchés sur le sujet ont trouvé des parades.

Il existe ainsi plusieurs méthodes d'adaptation des SVM aux problèmes multiclasses. Je ne vais parler que des approches *one-vs-one* et *one-vs-all*, parce que se sont les deux seules que je connais, et pour autant que je sache elles sont largement utilisées par la communauté.

! Ces deux méthodes ne sont pas équivalentes: comme on va le voir, elles peuvent donner des résultats différents pour une même entrée, et avec les mêmes données d'entraînement.

8.1. One vs one

Dans cette approche, on va créer des «voteurs»: chaque voteur V_{ij} détermine si mon entrée x a plus de chances d'appartenir à la catégorie i ou à la catégorie j. Ainsi, un voteur V_{ij} est un SVM qui s'entraîne sur les données de catégorie i et j uniquement.

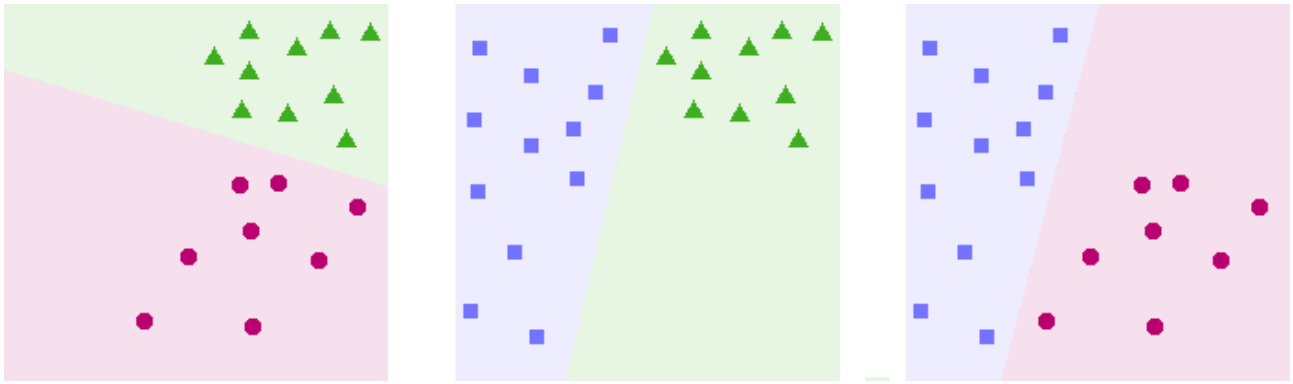


FIGURE 8.13. – Figure 13 : Les trois voteurs (SVM) de mon problème. A gauche, le voteur frelon-guêpe, au centre le voteur guêpe-abeille, à droite le voteur abeille-frelon.

Pour classer une entrée on retournera tout simplement la catégorie qui aura remporté le plus de duels. Ici, comme on le voit dans la figure 14, la catégorie «frelon» est celle qui remporte le plus de duels: j'ai donc été piqué par un frelon. Outch!

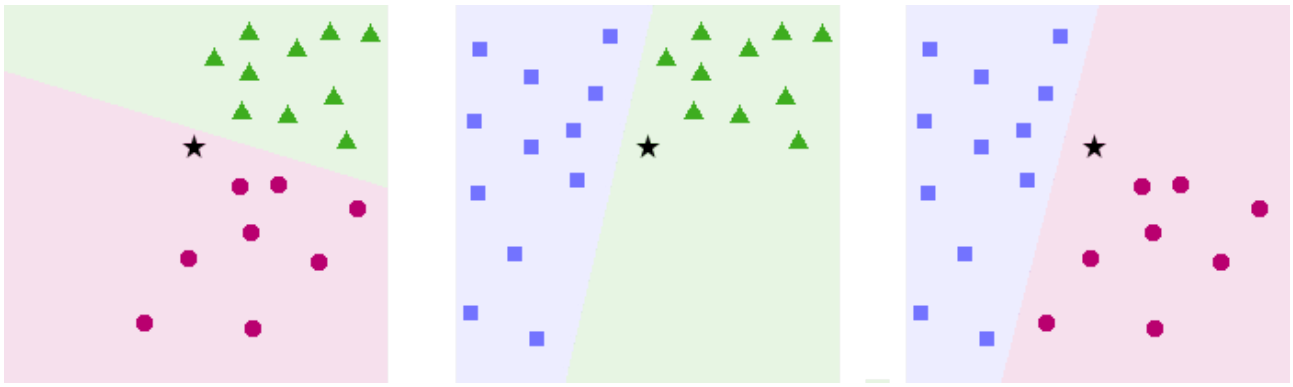


FIGURE 8.14. – Figure 14 : l'étoile est plus frelon que guêpe, plus guêpe qu'abeille et plus frelon qu'abeille.

L'inconvénient de cette méthode est que le nombre de voteurs est globalement proportionnel au carré du nombre de catégories: pour 10 catégories, ce sont 45 voteurs qu'il faut créer, mais pour 20 catégories, il en faudra 190, d'où un temps de calcul de plus en plus élevé.

8.2. One vs all

L'approche *one-vs-all* consiste à créer un SVM par catégorie. Dans notre exemple, un SVM sera ainsi spécialisé dans la reconnaissance des abeilles, un autre dans la reconnaissance des guêpes, et un autre dans les frelons.

Pour entraîner mon SVM spécialisé en reconnaissance d'abeilles, je crée deux catégories: la catégorie «abeille», qui contient toutes les entrées d'abeilles, et la catégorie «pas abeille», qui contient toutes les autres entrées. Je fais de même pour mes SVM spécialisés dans les autres catégories.

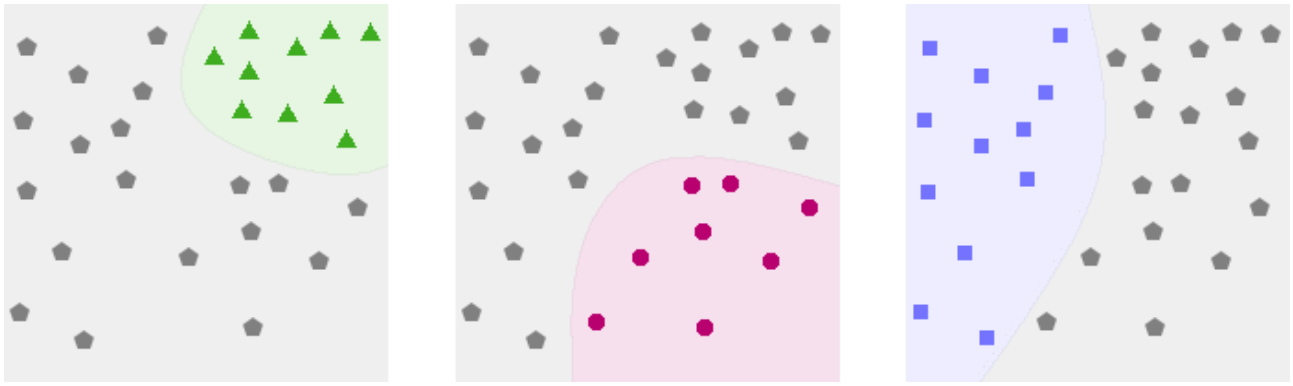


FIGURE 8.15. – Figure 15 : A gauche, le SVM spécialisé dans la reconnaissance des guêpes, au centre dans la reconnaissance des frelons et à droite des abeilles (on suppose qu'on a trouvé un kernel trick approprié à chaque fois, afin d'obtenir une frontière non linéaire)

Pour classer une nouvelle entrée, on regarde à quelle catégorie la nouvelle entrée est le plus probable d'appartenir.

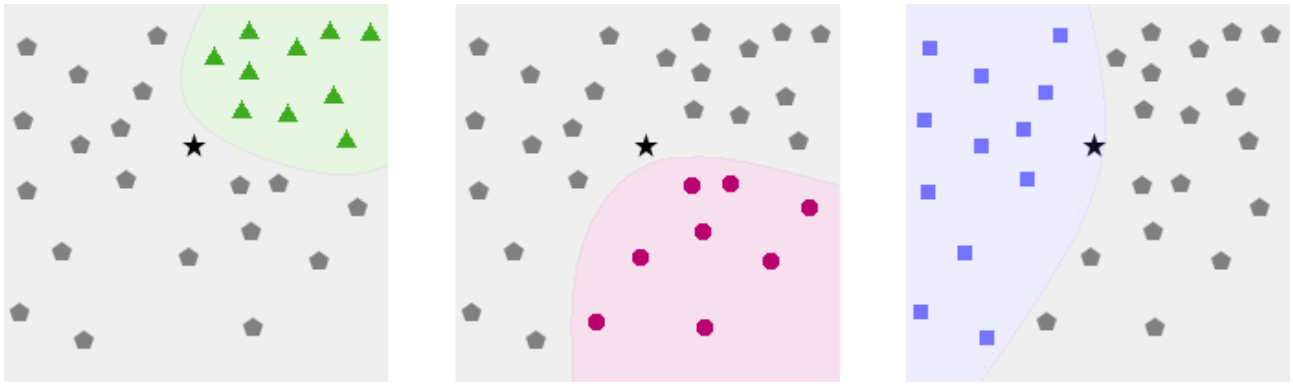


FIGURE 8.16. – Figure 16 : à gauche, probablement pas une guêpe. Au centre, probablement pas un frelon. A droite, probablement une abeille

Dans notre exemple, Figure 16, on voit que l'insecte m'ayant piqué est une abeille. Pôv' bête. 🍊

Il arrive que plusieurs SVM aient un résultat positif (par exemple, probablement une abeille, et probablement une guêpe). Dans ce cas-là, on prend celui qui est le plus certain de son résultat (c'est-à-dire celui pour lequel l'étoile est le plus éloignée de la frontière). De même, quand tous les résultats sont négatifs, on prend alors la catégorie du SVM pour lequel l'entrée est le plus près possible de la frontière (c'est-à-dire, celui pour lequel on est le moins sûr que le résultat est négatif).
En termes plus mathématiques, si l'on nomme h_i les différentes fonctions de classification des SVM associés à nos catégories, l'entrée x appartiendra à la catégorie j, avec $j = \underset{i}{\operatorname{argmax}}\, h_i(x)$.

L'inconvénient de cette méthode est que, quand on commence à avoir beaucoup de données, on ne dispose (par exemple) que de 10 données «abeilles» contre 100 «pas abeilles». Quand le déséquilibre entre la quantité de données dans les deux catégories est trop fort, un SVM obtient de moins bons résultats.

9. A vous de jouer !

9.1. Un SVM en action

Trève de théorie!

Je vous propose de tester un SVM, un vrai. Pour cela, voici un JSFiddle dont le code a été honteusement pompé sur la page d'accueil de la bibliothèque LibSVM .

!(https://jsfiddle.net/xjgvnuw7/1/)

Le SVM avec les paramètres que j'ai mis par défaut est un SVM avec un noyau linéaire (pas de kernel trick), qui autorise les erreurs (le coût est le nombre situé après `-c` dans la barre de texte). Il peut reconnaître trois classes différentes, avec une classification à la one-vs-one.
Cliquez sur la grille pour rajouter des points, cliquez sur le bouton «Changer de couleur» pour rajouter des points d'une autres couleur (rouge, jaune, bleu). La documentation de la barre de paramètres est donnée sur le site (en anglais), dont voici un copier-coller:

Contenu masqué n°1

9.2. Et dans la vraie vie, alors ?

Concrètement, voici des exemples de ce que peut faire un SVM.

— Classer des fleurs suivant leur espèce, en fonction de la longueur et largeur de leurs pétales et sépales (c'est en fait un des *datasets* typiques du machine learning, appelé *iris dataset*);
— En médecine, les SVM sont utilisés pour reconnaître, par exemple, la diffusion dans le cerveau d'altérations liées aux AVC (en anglais) ;
— Reconnaissance de caractères sur des plaques d'immatriculation (en anglais) ;
— En médecine encore, identification des gènes responsables de certains types de cancer (en anglais) ;

— etc.

Comme nous en avions parlé, les SVM sont très efficaces quand le nombre de données d'entraînement est faible, en particulier en comparaison avec d'autres méthodes d'apprentissage automatique. Par exemple, les réseaux de neurones (outils d'apprentissage profond, ou *deep learning*, très à la mode) sont très performants, mais ont besoin d'une grosse quantité de données d'entraînement (quelques milliers à plusieurs millions voire plus, suivant la complexité du réseau de neurones). C'est donc par exemple dans le cas où le nombre de données d'entraînement est faible que l'on va favoriser les SVM aux réseaux de neurones.

9.3. Un SVM à la maison

Si vous voulez vous aussi jouer avec un SVM, vous pouvez bien sûr le coder vous-même, mais, à moins que vous n'adoriez résoudre algorithmiquement des problèmes d'optimisation quadratique, il vaut mieux utiliser une bibliothèque. 🍊

— En Python 2 et Python 3, vous pouvez vous servir de **scikit**, plus précisément de `sklearn.svm` .
— En Java, vous pouvez utiliser **LIBSVM** , qui a également été portée en Matlab, Python, Ruby, Perl, NodeJS, bindée en PHP…
— En Lua et en C, vous pouvez utiliser **Torch**, avec le module `torch-svm` , etc.

Par ailleurs, je vous recommande la lecture de ce guide d'utilisation des SVM (en anglais), qui vous permettra d'éviter certains écueils, et donc d'améliorer la pertinence de votre SVM.

Pfiou!

Ca y est, vous savez comment fonctionne un SVM. Vous savez aussi comment les rendre plus performants grâce au *kernel trick*, et vous savez comment les adapter pour classer plus que deux catégories.

Il reste encore de nombreuses choses à apprendre sur les SVM: optimiser les paramètres du kernel choisi, utiliser les SVM pour faire de la régression, connaître la différence entre C-SVM et nu-SVM[7], optimiser le calcul de h, etc., mais je pense que vous en avez assez pour l'instant. 🍊 Qui plus est, vous en savez bien assez pour voler de vos propres ailes, maintenant!

Merci à ache, backmachine, elegance, gbdivers, Holosmos, Hugo, SpaceFox, Vayel pour tous leurs retours pendant la période de bêta, qui fut très constructive. Merci également au gentil modo artragis. Merci enfin à Gabbro, pour tous ses retours pendant la validation du tutoriel, dont il s'est occupé.

Ici s'achève ce tutoriel. Bon *machine learning* à vous!

10. Sources, liens pour aller plus loin

10.1. Documents orientés théorie

— Support-Vector Network (en anglais): un des articles fondateurs des SVM, par Cortes et Vapnik, de 1995.
— A Gentle Introduction to Support Vector Machines in Biomedicine (en anglais): un document très détaillé et pédagogue.
— Machines à Vecteur Support , un document traitant d'autres aspects intéressants des SVM.
— Lagrange Multipliers Tutorial in the Context of Support Vector Machines (en anglais): un document traitant de la résolution du problème d'optimisation quadratique des SVM.
— Support Vector Machines (en anglais): un cours traitant des SVM de l'université de Stanford.
— History of SVMs (en anglais): récit de l'invention des SVM, au fil des décennies.
— Machines à Vecteur de Support : l'article Wikipédia sur les SVM, histoire de faire dans l'originalité.

10.2. Documents orientés pratique

— A Practical Guide to Support Vector Classification (en anglais): un mode d'emploi des SVM, qui détaille notamment les écueils classiques à éviter.
— Why use SVM ? (en anglais): un article comparant rapidement le SVM à d'autres algorithmes de machine learning.
— Kernel Functions for Machine Learning Applications (en anglais): une large liste de fonctions noyaux, avec leurs caractéristiques.

Contenu masqué

Contenu masqué n°1

```
options:
-s svm_type : set type of SVM (default 0)
        0 -- C-SVC
        1 -- nu-SVC
        2 -- one-class SVM
        3 -- epsilon-SVR
        4 -- nu-SVR
-t kernel_type : set type of kernel function (default 2)
        0 -- linear: u'*v
        1 -- polynomial: (gamma*u'*v + coef0)^degree
        2 -- radial basis function: exp(-gamma*|u-v|^2)
```

7. Pour faire bref, ce sont différentes méthodes pour autoriser les erreurs de classification, c.f. la partie «Cas non-séparable».

Contenu masqué

```
        3 -- sigmoid: tanh(gamma*u'*v + coef0)
-d degree : set degree in kernel function (default 3)
-g gamma : set gamma in kernel function (default 1/num_features)
-r coef0 : set coef0 in kernel function (default 0)
-c cost : set the parameter C of C-SVC, epsilon-SVR, and nu-SVR
    (default 1)
-n nu : set the parameter nu of nu-SVC, one-class SVM, and nu-SVR
    (default 0.5)
-p epsilon : set the epsilon in loss function of epsilon-SVR
    (default 0.1)
-m cachesize : set cache memory size in MB (default 100)
-e epsilon : set tolerance of termination criterion (default 0.001)
-h shrinking: whether to use the shrinking heuristics, 0 or 1
    (default 1)
-b probability_estimates: whether to train a SVC or SVR model for
    probability estimates, 0 or 1 (default 0)
-wi weight: set the parameter C of class i to weight*C, for C-SVC
    (default 1)

The k in the -g option means the number of attributes in the input
    data.
```

Retourner au texte.