# [Entraînez un modèle prédictif linéaire](https://openclassrooms.com/fr/courses/4444646-entrainez-un-modele-predictif-lineaire)

Table des matières

[Entraînez un modèle prédictif linéaire 1](#_Toc57814120)

[**Trouvez une combinaison linéaire de variables qui approxime leurs étiquettes** 3](#_Toc57814121)

[**Maximisation de la vraisemblance** 3](#_Toc57814122)

[**Méthode des moindres carrés** 6](#_Toc57814123)

[**Interprétation** 7](#_Toc57814124)

[**Conclusion** 7](#_Toc57814125)

[Contrôlez la complexité de votre modèle 9](#_Toc57814126)

[Limitations de la régression linéaire 9](#_Toc57814127)

[Régularisation 9](#_Toc57814128)

[Choix du coefficient de régularisation 9](#_Toc57814129)

[Réduisez l’amplitude des poids affectés à vos variables 10](#_Toc57814130)

[Régularisation de Tykhonov 10](#_Toc57814131)

[Solution de la régression ridge 10](#_Toc57814132)

[Chemin de régularisation 12](#_Toc57814133)

[Conclusion 13](#_Toc57814134)

[Réduisez le nombre de variables utilisées par votre modèle 14](#_Toc57814135)

[Lasso 14](#_Toc57814136)

[Interprétation géométrique 14](#_Toc57814137)

[Chemin de régularisation 16](#_Toc57814138)

[Elastic Net 17](#_Toc57814139)

[Conclusion 18](#_Toc57814140)

[TP - Comparez le comportement du lasso et de la régression ridge 19](#_Toc57814141)

[Chargeons le jeu de données 19](#_Toc57814142)

[La baseline : une régression classique 19](#_Toc57814143)

[Application de la régression ridge 20](#_Toc57814144)

[Application du Lasso 23](#_Toc57814145)

[Conclusion 25](#_Toc57814146)

[Partie 1 26](#_Toc57814147)

[Compétences évaluées 26](#_Toc57814148)

[ Question 1 26](#_Toc57814149)

[ Question 2 26](#_Toc57814150)

[ Question 3 27](#_Toc57814151)

[ Question 4 28](#_Toc57814152)

[ Question 5 28](#_Toc57814153)

[ Question 6 29](#_Toc57814154)

[ Question 7 29](#_Toc57814155)

[ Question 8 29](#_Toc57814156)

[ Question 9 31](#_Toc57814157)

[Prédisez linéairement la probabilité de l’appartenance d’un point à une classe 32](#_Toc57814158)

[Prédire linéairement l'appartenance à une classe 32](#_Toc57814159)

[Solution de la régression logistique 34](#_Toc57814160)

[Régularisation 35](#_Toc57814161)

[Conclusion 35](#_Toc57814162)

[**Maximisez la marge de séparation entre vos classes** 36](#_Toc57814163)

[**Cas linéairement séparable** 36](#_Toc57814164)

[**Cas non-linéairement séparable** 43](#_Toc57814165)

[**Et la régression dans tout ça ?** 49](#_Toc57814166)

[**Conclusion** 50](#_Toc57814167)

[Classifiez vos données en plus de deux classes 51](#_Toc57814168)

[One-versus-rest 51](#_Toc57814169)

[One-versus-one 52](#_Toc57814170)

[SVM multi-classe 54](#_Toc57814171)

[Conclusion 55](#_Toc57814172)

[TP - Entraînez une régression logistique et une SVM linéaire 56](#_Toc57814173)

[Chargeons le jeu de données 56](#_Toc57814174)

[Appliquez la régression logistique 57](#_Toc57814175)

[Appliquez une SVM Linéaire 60](#_Toc57814176)

[Conclusion 62](#_Toc57814177)

[Entraînez-vous à classer automatiquement des feuilles d’arbres 63](#_Toc57814178)

[À vous de jouer 63](#_Toc57814179)

[Vérifiez votre travail 63](#_Toc57814180)

Vous avez découvert le concept d’apprentissage supervisé dans le cours [Initiez-vous au machine learning](https://openclassrooms.com/courses/initiez-vous-au-machine-learning). Dans [ce cours](https://openclassrooms.com/courses/evaluez-et-ameliorez-les-performances-d-un-modele-de-machine-learning), vous avez appris à évaluer un modèle de classification ou de régression…

Il est maintenant temps de **découvrir les algorithmes classiques du machine learning supervisé**. Dans ce cours, vous apprendrez à maîtriser les algorithmes dont la fonction de décision est une combinaison linéaire des variables.

Vous découvrirez en particulier la **régression linéaire** et la **régression logistique**. Vous apprendrez à contrôler les poids affectés à chacune des variables pour éviter le sur-apprentissage ou construire des modèles parcimonieux. Vous comprendrez le fonctionnement des **machines à vecteurs de support (SVM)**. Enfin, vous saurez résoudre des problèmes de classification à plus de deux classes.

Suivez ce cours pour développer des modèles linéaires prédictifs sur vos données !

**Objectifs pédagogiques :**

* Entraîner un **algorithme linéaire de classification** binaire ou multiclasse ou de régression,
* **Contrôler les poids affectés aux différentes variables** par un mécanisme de régularisation,
* Comprendre les points communs et différences entre **SVM** et **régression linéaire/logistique.**

**Prérequis :**

Ce cours de Data Science se situe au croisement des mathématiques et de l'informatique. Pour en profiter pleinement, n'hésitez pas à vous rafraîchir la mémoire, avant ou pendant le cours, sur :

* **Python** pour le calcul numérique que nous utiliserons dans la partie TP du cours (librairie numpy et création de graphes avec pyplot),
* Quelques notions d'**algèbre linéaire**, telles que manipulation de vecteurs, multiplications de matrices, normes,
* Quelques notions de **probabilités et statistiques**, telles que distribution de loi de probabilité et variance.

**Trouvez une combinaison linéaire de variables qui approxime leurs étiquettes**

- sortie console -

Nous allons dans ce cours nous attaquer à des problèmes *supervisés* que nous allons résoudre à l'aide de modèles *linéaires.*Autrement dit, nous allons essayer d'approximer les étiquettes de nos données grâce à une *combinaison linéaire*des variables utilisées pour représenter lesdites données. Dans ce chapitre, nous allons travailler sur des problèmes de *régression* : nos étiquettes sont des nombres réels (et non pas des catégories).

Un exemple de problème que l'on pourrait chercher à résoudre ici est celui de prédire le nombre d'utilisateurs connectés sur OpenClassrooms (notre étiquette réelle) comme une combinaison linéaire de variables telle que l'heure et le jour de la semaine, à partir des données de l'année passée (nos observations étiquetées).

Prenons des données, **n** points en **p** dimensions, représentés par la matrice , avec leurs étiquettes à valeurs réelles, représentées par un vecteur .

Le but de la **régression linéaire**est de trouver une fonction *linéaire* qui permette de prédire l'étiquette  du i-ème point à partir du vecteur . Autrement dit, f(x) peut s'écrire sous la forme .

Comment trouver les valeurs des coefficients  ?

**Maximisation de la vraisemblance**

Nous allons pour ce faire nous placer dans un contexte *bayésien.*Tout de suite les grands mots ! En fait, cela veut dire que nous allons réfléchir en termes de *probabilités*et chercher les valeurs de β0, β1, …, βp pour lesquelles il est le plus *vraisemblable*d'observer les données {X,y} qui nous sont données.

Nous allons commencer par supposer que les vecteurs  sont les réalisations d'une variable aléatoire **p**-dimensionnelle **x** et que les nombres réels  sont les réalisations d'une variable aléatoire y.

Pour cela, nous allons avoir besoin de poser quelques hypothèses :

- la première, *l'hypothèse de linéarité*, est que nous pouvons approximer, pour chacun des points de notre jeu de données, son étiquette par une combinaison linéaire de la valeur des variables qui le décrivent : . Le terme de bruit  ϵi représente la différence entre la combinaison linéaire des variables et la vraie étiquette.

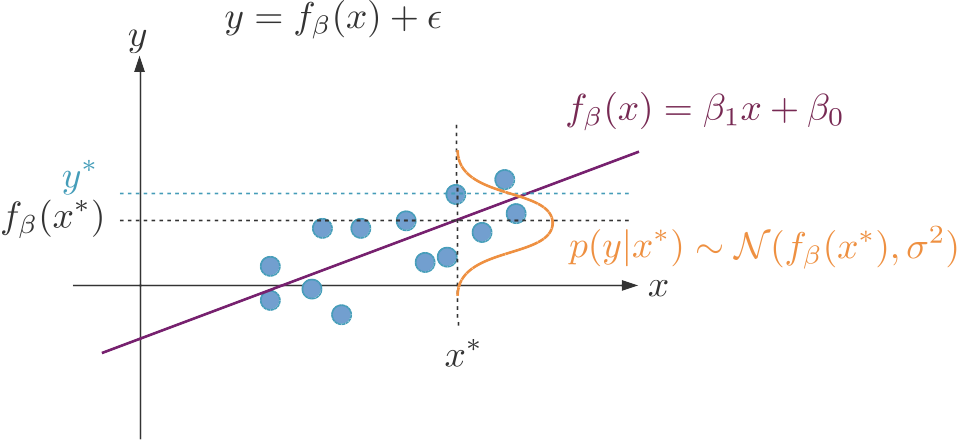
- la deuxième, *l'hypothèse de normalité*, est que le bruit est *gaussien*, ou *normalement distribué*: les ϵi sont la réalisation d'une variable aléatoire gaussienne, centrée en 0 et d'écart-type σ :  

- la troisième, *l'hypothèse d'indépendance*, est que les n couples  sont des réalisations *indépendantes* les unes des autres des variables x et y.

On utilise souvent l'expression**indépendants et identiquement distribués**, abrégé «**i.i.d.** » (« *independent and identically distributed* » en anglais) pour qualifier le fait que des points soient des réalisations *indépendantes* de la *même* distribution (ici, cela veut dire que les paramètres seront les mêmes pour tous les points).

Appelons β le vecteur de dimensions (p+1) β=(β0,β1,…,βp). Ce vecteur *paramétrise*notre modèle : une fois que nous avons trouvé la valeur de β, notre modèle est entièrement spécifié et nous pouvons l'utiliser pour faire des prédictions.

 Selon nos hypothèses, la probabilité de y étant donné x et β : p(y|x,β) suit une distribution normale, centrée en notre prédiction, et d'écart-type σ.

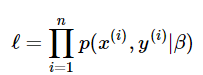


L'erreur entre la prédiction et la vraie valeur de l'étiquette vient d'une distribution gaussienne (ici avec une seule variable).

**Vraisemblance**

La **vraisemblance**, que nous allons noter ℓ (pour son nom anglais, « *likelihood*»), c'est la probabilité d'observer nos n observations étant donné β.

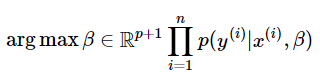
Grâce à l'hypothèse d'indépendance, nous pouvons dire que cette probabilité est le produit des probabilités d'observer chacune des observations :



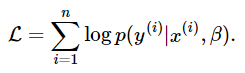
Par définition de la probabilité conditionnelle,



 Nous cherchons donc à trouver β qui maximise cette quantité. Comme   ne dépend pas de β, on peut l'ignorer et nous cherchons donc à résoudre le problème suivant :

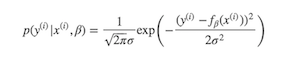


Pour simplifier les calculs, nous allons maximiser non la vraisemblance, mais son *logarithme*: comme le logarithme est une fonction monotone strictement croissante, les deux sont équivalents. Nous cherchons donc à **maximiser le log de vraisemblance**, à savoir

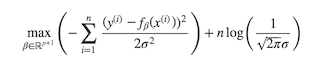


En anglais, on appelle ça « *log-likelihood maximisation*».

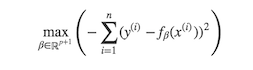
Grâce à nos hypothèses, nous connaissons la forme de  : une gaussienne, centrée en la prédiction et d'écart-type σ. Nous pouvons donc écrire



Nous pouvons prendre le log de cette expression, et trouver le maximum du log de vraisemblance est donc équivalent à



En éliminant de cette équation les termes qui ne dépendent pas de β, nous trouvons que maximiser le log de la vraisemblance est équivalent à résoudre

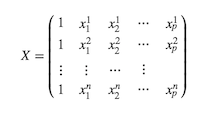


**Méthode des moindres carrés**

Regardons de plus près le résultat auquel nous sommes arrivés : nous cherchons à minimiser la somme des carrés des différences entre chaque étiquette et sa valeur prédite... autrement dit, maximiser la vraisemblance est équivalent à *minimiser la somme des carrés des erreurs* !

Cette façon de trouver les coefficients d'une régression linéaire s'appelle la **méthode des moindres carrés** et est connue depuis Gauss et Legendre, qui faisaient donc déjà du machine learning au début du XIXème siècle 😉

 Mais comment résout-on ce problème en pratique ? Pour cela, nous allons passer à une écriture sous forme matricielle. Nous allons commencer par ajouter une colonne de 1 à la matrice de données X : désormais, j'appelle X  qui a la forme suivante :



 Cela me permet d'écrire  comme , et donc sous forme matricielle pour toutes les observations comme le produit Xβ. Ainsi, minimiser la somme des carrés des erreurs est équivalent à



Il s'agit désormais de minimiser une forme quadratique convexe, ce qui peut se faire en annulant son gradient en β :



Le gradient d'une forme quadratique se calcule de manière analogue à la dérivée d'un polynôme de degré 2 :



Nous cherchons donc à trouver β tel que



 Si la matrice est inversible, alors la solution s'écrit



 Sinon, la solution n'est pas unique, et il faudra utiliser à la place de  un **pseudo-inverse** de , à savoir une matrice  **A** telle que que l'on peut trouver grâce à un algorithme de calcul de pseudo-inverse comme celui de Moore-Penrose.

Pour en savoir plus sur les pseudo-inverses, consulter par exemple la [**page Wikipedia**](https://www.wikiwand.com/fr/Pseudo-inverse) correspondante.

**Interprétation**

Un des grands avantages de la régression linéaire est qu'elle produit un modèle **interprétable**, au sens où il nous permet de comprendre sur quoi se base la prédiction.

Plus précisément, si je rajoute une unité à la valeur de la variable **x1**, alors son étiquette augmente de **β1**. Plus **|β1|** est grande, plus la variable **x1** a un effet important sur la prédiction.

Cette interprétation n'est valide que si les variables décrivant nos données sont *indépendantes*. En effet, sinon, le fait de modifier **x1** va entraîner des modifications sur les variables *corrélées*à **x1** et l'effet sur la valeur prédite n'est plus aussi direct.

Par ailleurs, si les variables sont corrélées, la matrice X ne sera pas de rang colonne plein et donc ne sera pas inversible, ce qui veut dire qu'il n'y aura pas de solution unique. En particulier, les coefficients des variables corrélées seront dépendants les uns des autres, et une modification de la valeur de l'un peut être compensée par une variation dans la valeur des autres. Cela signifie que, dans le cas où les variables sont corrélées, la régression linéaire est **instable**: plusieurs solutions sont possibles.

**Conclusion**

* On apprend les coefficients d'une régression linéaire en **maximisant le log de la vraisemblance**, où, de manière équivalente si l'on suppose l'erreur normalement distribuée et centrée en zéro, en **minimisant la somme des carrés des erreurs**.
* Cette méthode s'appelle la **méthode des moindres carrés**.
* Si la matrice  est inversible, la régression linéaire admet une solution *unique*et *explicite.*
* Sinon, on peut calculer une solution grâce à un algorithme de calcul de *pseudo-inverse*, mais cette solution n'est pas unique.

Dans scikit-learn, la régression linéaire est implémentée comme [LinearRegression](http://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LinearRegression.html) dans le module linear\_model.

## Contrôlez la complexité de votre modèle

### Limitations de la régression linéaire

Dans le chapitre précédent, nous avons vu que la régression linéaire souffre de quelques inconvénients **quand les variables sont corrélées** : la solution n'est pas unique et les coefficients ont une grande variabilité, et l'interprétation est plus difficile.

Une autre situation dans laquelle la solution n'est pas unique est celui où **le nombre de variables est plus grand que celui d'observations.**Dans ce cas, la matrice  n'est pas inversible :  X ne peut pas être de rang colonne plein en ayant plus de colonnes (variables) que de lignes (observations).

Il y a donc un risque de **sur-apprentissage.**

### Régularisation

Pour limiter le sur-apprentissage, on peut utiliser une technique, la **régularisation**, qui consiste à contrôler simultanément **l'erreur du modèle sur le jeu d'entraînement**et **la complexité du modèle.** Souvenez-vous, plus un modèle est « complexe », plus il est susceptible de sur-apprendre. L'idée ici va être de ne pas minimiser seulement l'erreur du modèle, mais une nouvelle **fonction objective** qui est la somme d'un terme d'erreur et d'un terme mesurant la complexité du modèle. Dans le cas de la régression linéaire, il s'agit donc de résoudre



Le **régularisateur**, qui mesure la complexité du modèle, est une fonction des poids β du modèle.

Dans les chapitres suivants, vous verrez quelques exemples concrets de régularisateurs.

### Choix du coefficient de régularisation

Quel que soit le régularisateur utilisé, notre algorithme d'apprentissage contient maintenant un **hyperparamètre**, le **coefficient de régularisation λ**qui contrôle l'importance relative du terme d'erreur et du terme de régularisation.

Plus **λ** est grand, plus le terme de régularisation est important. Plus il est petit, est plus l'erreur est importante ; s'il est suffisamment faible (et en particulier s'il est égal à zéro), on retrouvera la solution de la régression linéaire non-régularisée. Quelle valeur donner à cet hyperparamètre ? En général, c'est une question que l'on réglera en utilisant une **validation croisée** (cf. le [chapitre correspondant](https://openclassrooms.com/courses/evaluez-et-ameliorez-les-performances-d-un-modele-de-machine-learning/mettez-en-place-un-cadre-de-validation-croisee) du cours « Évaluez et améliorez les performances d'un modèle de machine learning »).

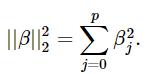
## Réduisez l’amplitude des poids affectés à vos variables

Dans le chapitre précédent, nous avons vu que la régularisation permet d'éviter le sur-apprentissage d'une régression linéaire en rajoutant à la fonction objective (la somme des carrés des erreurs) un terme de régularisation qui mesure la complexité du modèle.

### Régularisation de Tykhonov

 La régularisation de Tykhonov est un cas particulier de régularisation, dans lequel on utilise pour régulariser la régression linéaire **le carré de la norme**du vecteur de poids β.

Plus précisément, il s'agit de la **norme**ℓ2, ou **norme euclidienne**, c'est-à-dire



 Nous avons donc maintenant le problème d'optimisation suivant :



 que l'on peut aussi écrire



Le modèle linéaire que l'on apprend en résolvant cette équation est appelé **régression ridge**, ou ridge regression en anglais.

Pour ceux qui en auraient entendu parler, cela rappelle le « weight decay » (dégradation / modération des pondérations) utilisées dans les réseaux de neurones.

### Solution de la régression ridge

La fonction à minimiser est encore une fois une forme quadratique. Comme dans le cas de la méthode des moindres carrés (régression linéaire non régularisée), nous pouvons résoudre ce problème d'optimisation en annulant le gradient en β de la fonction objective :



Encore une fois, il s'agit du gradient d'une forme quadratique, qui se calcule de manière analogue à la dérivée d'un polynôme de degré 2 :

−2X⊤(y−Xβ)+2λβ=0−2X⊤(y−Xβ)+2λβ=0

 Ce qui nous donne



 Or si λ>0, la matrice  est toujours inversible !

La régression ridge admet donc toujours une unique solution explicite :



#### Importance de la standardisation des variables

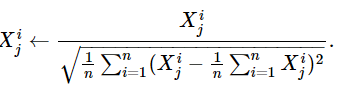
Que se passe-t-il si l'on multiplie la variable **x1** par une constante **α** ? Regardons tout d'abord le cas de la régression linéaire sans régularisation. La solution de la régression linéaire vérifie  .

Maintenant, si j'appelle  la matrice obtenue en remplaçant **x1** par **αx1** dans **X**, la solution  de la régression linéaire qui explique **y** comme une fonction linéaire de αx1, x2 ,…, xp vérifie .

On a donc , ce dont on peut conclure que β. Ainsi, multiplier une variable par une constante équivaut à diviser le poids correspondant par la même constante.

Dans le cas de la régression ridge, cependant, l'effet de la multiplication de **x1** par une constante dépend aussi du terme de régularisation , et est bien moins clair.

Pour cette raison, l'échelle de la plage des valeurs prises par les différentes variables a un impact sur le résultat de la régression ridge (mais pas sur celui de la régression non régularisée). Pour cette raison, je vous recommande fortement de **standardiser** les variables avant d'apprendre une régression ridge. Il s'agit de faire en sorte que chaque variable ait un écart-type de 1, en leur appliquant la transformation suivante :



 Cette transformation est implémentée dans [sklearn.preprocessing.StandardScaler](http://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.StandardScaler.html#sklearn.preprocessing.StandardScaler).

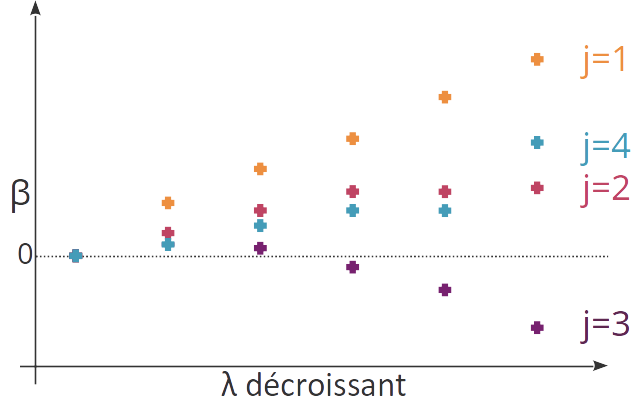
### 

### Chemin de régularisation

Comment la solution de la régression ridge évolue-t-elle en fonction du coefficient de régularisation **λ** ? Si **λ** est très grand, alors la régularisation prend le dessus, le terme d'erreur n'importe plus et la solution est **β=0**. À l'inverse, si **λ** est très faible, le terme de régularisation n'est plus utilisé, et on retrouve la solution de la régression linéaire non régularisée.

On peut visualiser l'effet que la valeur du coefficient de régularisation a sur chaque variable en affichant la valeur des différents coefficients **βj** en fonction de **λ** quand **λ** décroît. Le résultat s'appelle le **chemin de régularisation.**

En pratique, on choisira la valeur optimale du coefficient de régularisation entre ces deux extrêmes (solution nulle et solution de la régression non régularisée) par validation croisée.



Exemple de chemin de régularisation pour 4 variables.

Un des effets de la régression ridge, que l'on peut observer sur le schéma entre les variables j=2 et j=4, est de **grouper**les variables corrélées, en leur affectant des coefficients similaires. Là où la régression non régularisée peut « répartir » le poids affecté à un groupe de variable corrélées de plusieurs façons, la régression régularisée les répartit de manière homogène.

### Conclusion

* La norme ℓ2 du vecteur de poids peut être utilisée comme terme de régularisation de la régression linéaire.
* Cela s'appelle la **régularisation de Tykhonov**, ou **régression ridge.**
* La régression ridge admet toujours une solution analytique unique.
* La régression ridge permet **d'éviter le surapprentissage**en **restraignant l'amplitude des poids.**
* La régression ridge a un effet de **sélection groupée** : les variables corrélées ont le même coefficient.

La régression ridge est implémentée dans scikit-learn : [linear\_model.Ridge.](http://scikit-learn.org/stable/modules/generated/sklearn.linear_model.Ridge.html#sklearn.linear_model.Ridge)

[linear\_model.RidgeCV](http://scikit-learn.org/stable/modules/generated/sklearn.linear_model.RidgeCV.html#sklearn.linear_model.RidgeCV)permet de déterminer la valeur optimale du coefficient de régularisation par validation croisée.

## Réduisez le nombre de variables utilisées par votre modèle

### 

### Lasso

La régression ridge nous permet de réduire l'amplitude des coefficients d'une régression linéaire et d'éviter le sur-apprentissage. Cependant, on peut souhaiter pousser les choses plus loin, et annuler certains coefficients. Les variables qui auront un coefficient égal à zéro ne feront plus partie du **modèle,** qui en sera simplifié d'autant. Un tel modèle, avec beaucoup de coefficients nuls, est appelé un modèle **parcimonieux**(ou « sparse » en anglais).

Il s'agit donc d'une méthode de **sélection de variables** et de **réduction de dimension supervisée** : les variables qui ne sont pas nécessaires à la prédiction de l'étiquette sont éliminées.

Pour arriver à cela, il suffit de remplacer le terme de régularisation de la régression ridge, autrement dit la norme **ℓ2** de **β**, par la norme **ℓ1** de ce vecteur, c'est-à-dire : .

Nous avons donc maintenant le problème d'optimisation suivant :



C'est ce qu'on appelle le **lasso**, pour Least Absolute Shrinkage and Selection Operator. « Absolute » pour l'utilisation de la norme 1, « shrinkage » (réduction) parce qu'on contraint les coordonnées de ββ à avoir des valeurs faibles, et « selection » parce qu'on va tellement les réduire que certaines seront nulles.

Cet algorithme est aussi connu sous le nom de **poursuite de base** (en anglais « basis pursuit ») dans le domaine du traitement du signal.

Contrairement à la régression ridge, ce problème d'optimisation n'a pas de solution explicite ! Heureusement, on peut utiliser un algorithme du gradient pour le résoudre.

Le lasso n'a pas nécessairement une solution unique, car le problème n'est pas strictement convexe. Cependant, la valeur de Xβ qui minimise la fonction objective, elle, est unique. De plus, deux solutions du même lasso ont nécessairement le même signe sur leur **support** (c'est à dire leurs coefficients non-nuls), autrement dit, les coefficients qui ne valent zéro dans aucune de ces deux solutions ne peuvent pas avoir de signes opposés. Ainsi, on saura à partir d'une unique solution si l'effet d'une variable sur la prédiction est positif ou négatif.

### Interprétation géométrique

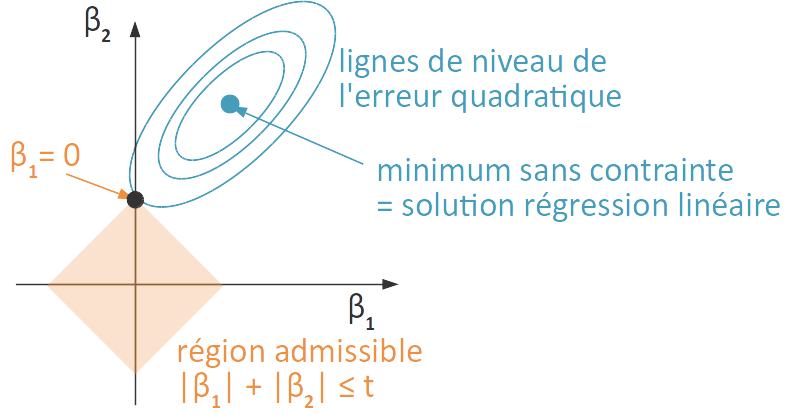
Mais pourquoi donc l'utilisation de la norme **ℓ1** force-t-elle certains coefficients à 0, contrairement à la norme **ℓ2** ?

Il se trouve que, pour une valeur de  donnée, il existe un  unique tel que le lasso est équivalent à résoudre le problème suivant :



Pour les curieux, cette équivalence s'obtient par dualité et selon les conditions de Karush-Kuhn-Tucker.

 Géométriquement, cela veut dire que, sauf dans le cas où la solution sans contrainte (c'est-à-dire non pénalisée, autrement dit la solution de la régression linéaire obtenue par la méthode des moindre carrés) vérifie aussi la contrainte, la solution est un point situé à l'intersection d'une ligne de niveau du terme d'erreur  et de la région admissible, c'est-à-dire la région de où la contrainte est vérifiée. Comme le terme d'erreur est quadratique en **β**, la ligne de niveau est une ellipsoïde. Par ailleurs, la région admissible est une boule **ℓ1** de rayon **t**, autrement dit, un hypercube. Comme cette boule a des « coins », l'ellipsoïde est susceptible de la rencontrer sur un de ces coins, là où une ou plusieurs coordonnées sont nulles.



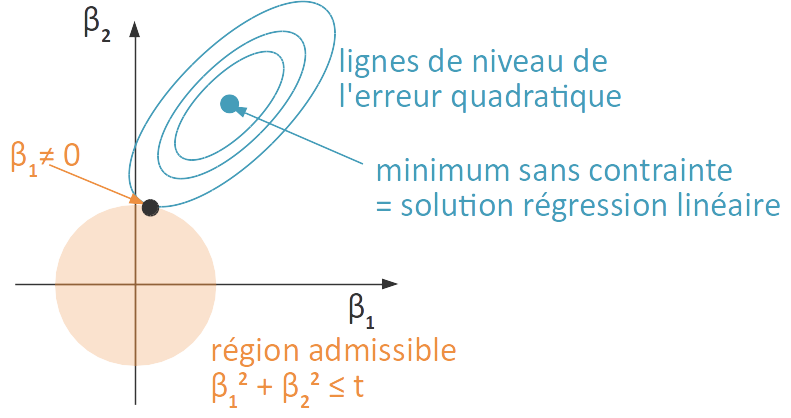
Solution du lasso.

#### Comparaison avec la régression ridge

Qu'en est-il de la régression ridge ? On peut démontrer une équivalence similaire, et la régression ridge est équivalent à résoudre



On s'intéresse donc maintenant à l'intersection d'une courbe de niveau de l'erreur avec la boule **ℓ2** de rayon **t**, autrement dit la boule euclidienne « ronde » dont on a plus l'habitude. Il n'y a ici aucune raison que cette intersection se fasse à un endroit où une ou plusieurs coordonnées s’annulent...

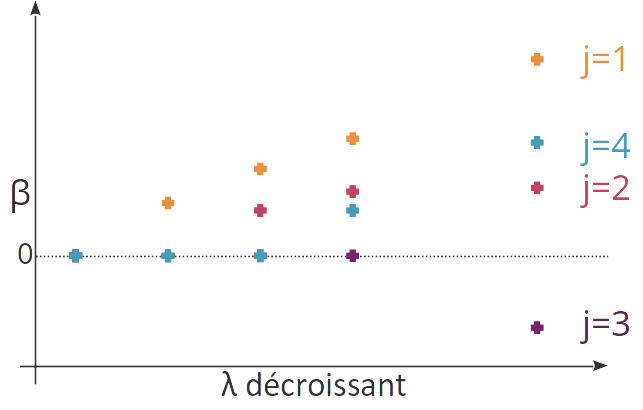


Solution de la régression ridge.

### 

### Chemin de régularisation

Comme pour la régression ridge, on peut visualiser les poids affectés aux différentes variables selon la valeur du coefficient de régularisation **λ**. Quand **λ** est grand, le modèle est très parcimonieux (**β=0** si **λ** est suffisamment grand). Plus il diminue, plus de variables vont avoir un coefficient non nul : on voit les variables entrer une à une (ou petit groupe par petit groupe) dans le modèle.



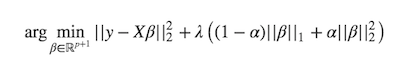
Exemple de chemin de régularisation du lasso.

Si plusieurs variables corrélées contribuent à la prédiction de l'étiquette, le lasso va avoir tendance à choisir une seule d'entre elles (affectant un poids de 0 aux autres), plutôt que de répartir les poids équitablement comme la régression ridge. C'est ainsi qu'on arrive à avoir des modèles très parcimonieux. Cependant, laquelle de ces variables est choisie est aléatoire, et peut changer si l'on répète la procédure d'optimisation. Le lasso a donc tendance à être instable.

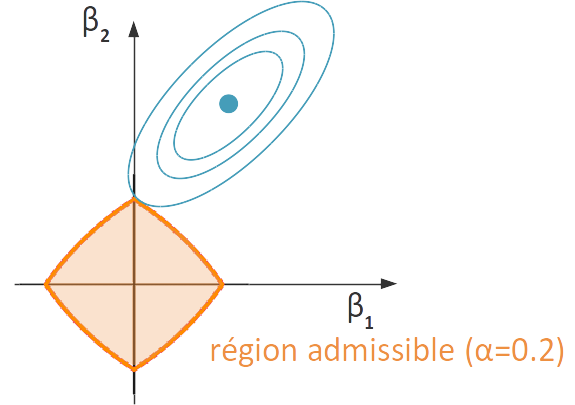
### Elastic Net

Si je résume, l'utilisation de la norme **ℓ2** nous permet d'éviter le sur-apprentissage avec une solution unique, et l'utilisation de la norme **ℓ1** d'avoir un modèle parcimonieux mais instable. Ne peut-on pas avoir un peu de beurre et d'argent du beurre à la fois ?

Eh bien si ! C'est ce que permet la méthode **elastic net**, qui combine les deux termes de régularisation en un. On formule ainsi le problème suivant :



La région admissible a une forme intermédiaire entre celles des boules de norme **ℓ1** et de norme **ℓ2**, et la solution est parcimonieuse mais moins que pour le lasso, car les variables corrélées entre elles qui sont pertinentes sont toutes sélectionnées (et reçoivent le même coefficient), comme avec la régression ridge.



Solution de l'elastic net.

C'est un peu louche, non, de pouvoir avoir le beurre et l'argent du beurre ? Le prix à payer en est le fait d'avoir maintenant non pas un seul mais deux hyperparamètres à sélectionner, **λ** et **α**, ce qui demande plus de ressources computationnelles...

### Conclusion

* Le **lasso** utilise la norme **ℓ1** du vecteur **β** comme régularisateur pour obtenir un modèle **parcimonieux**.
* Le lasso peut donc être utilisé comme un algorithme de **réduction de dimension supervisée.**
* Le lasso n'a pas de solution explicite, ni nécessairement unique.
* L**'elastic net** combine les normes **ℓ1** et **ℓ2** pour obtenir une solution moins parcimonieuse que le lasso, mais plus stable et dans laquelle toutes les variables corrélées pertinentes pour la prédiction de l'étiquette sont sélectionnées et reçoivent un poids identique.

## TP - Comparez le comportement du lasso et de la régression ridge

Dans ce chapitre, nous allons mettre en pratique les deux algorithmes de régularisation de la régression linéaire étudiés précédemment : la régression ridge et le lasso. Pour cela, nous allons appliquer et comparer les résultats de ces deux algorithmes sur un jeu de données sur le cancer de la prostate.

### Chargeons le jeu de données

Le jeu de données est téléchargable [**ici**](https://s3-eu-west-1.amazonaws.com/static.oc-static.com/prod/courses/files/Parcours_data_scientist/entrainez-un-modele-predictif-lineaire/TP_1_prostate_dataset.txt).

C’est un jeu de données assez classiques relativement petit, avec 97 observations et 9 variables. Nous allons ici définir comme target **y** la quantité d’expression de l’antigène qui est associée à la détection de ce cancer (la colonne lpsa du dataset). Les autres variables sont des constantes associées, dont le détail est [disponible ici](https://rafalab.github.io/pages/649/prostate.html)

Première étape, charger le jeu de données à l'aide de pandas :

import pandas as pd

raw\_data = pd.read\_csv('prostate.data.txt', delimiter='\t')

On récupère nos variables explicatives pour la régression, qu'on place dans une matrice X et notre variable expliquée **y** à part. On ne récupère pas la dernière colonne du dataset qui est un booléen associé à la présence du cancer, car on ne traite pas les variables discrètes dans ce TP - il nous sera utile pour le TP classification en revanche 😉

X\_train = raw\_data.iloc[:60,1:-3]

y\_train = raw\_data.iloc[:60,-2]

X\_test = raw\_data.iloc[60:,1:-3]

y\_test = raw\_data.iloc[60:,-2]

Ok on peut maintenant effectuer nos régressions ! 💪

### La baseline : une régression classique

La première étape est d'effectuer une régression linéaire classique afin de récupérer une erreur baseline, qu'on souhaite améliorer à l'aide des techniques de régularisation.

from sklearn import linear\_model

import numpy as np

# On crée un modèle de régression linéaire

lr = linear\_model.LinearRegression()

# On entraîne ce modèle sur les données d'entrainement

lr.fit(X\_train,y\_train)

# On récupère l'erreur de norme 2 sur le jeu de données test comme baseline

baseline\_error = np.mean((lr.predict(X\_test) - y\_test) \*\* 2)

print(baseline\_error)

On obtient l'erreur quadratique ci-dessous

2.8641499657014458

### 

### Application de la régression ridge

Comme vu dans le chapitre sur la régression ridge, on doit trouver un coefficient de régularisation adapté. Pour rappel, l'objectif est de biaiser un peu la prédiction, afin de diminuer l'erreur standard.   
  
On appelle ce coefficient alpha, on va en tester un certain nombre afin de trouver celui qui est optimal

n\_alphas = 200

alphas = np.logspace(-5, 5, n\_alphas)

On peut maintenant tester toute les régressions ridges avec les différentes valeur de l'hyperparamètre **α**. On récupère les poids des différents coefficients de la régression associées ainsi que l'erreur quadratique.

from sklearn.linear\_model import Ridge

ridge = linear\_model.Ridge()

coefs = []

errors = []

for a in alphas:

ridge.set\_params(alpha=a)

ridge.fit(X\_train, y\_train)

coefs.append(ridge.coef\_)

errors.append([baseline\_error, np.mean((ridge.predict(X\_test) - y\_test) \*\* 2)])

On peut afficher l'évolution de la valeur des différents poids associés aux paramètres

import matplotlib.pyplot as plt

ax = plt.gca()

ax.plot(alphas, coefs)

ax.set\_xscale('log')

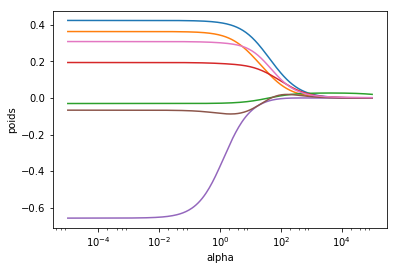
plt.xlabel('alpha')

plt.ylabel('weights')

plt.title('Ridge coefficients as a function of the regularization')

plt.axis('tight')

plt.show()



Les poids des différents paramètres en fonction de alpha

Comme on peut le voir (et comme c'était prévu), la valeur de alpha diminue les poids de tous les paramètres de la régression. Etudions maintenant la valeur de l'erreur quadratique

ax = plt.gca()

ax.plot(alphas, errors)

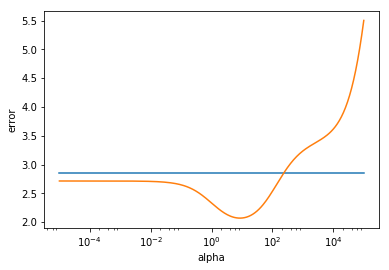
ax.set\_xscale('log')

plt.xlabel('alpha')

plt.ylabel('error')

plt.axis('tight')

plt.show()



Erreur en fonction d'alpha

Comme on peut le voir, la régularisation diminue l'erreur sur le jeu de données test. Vers alpha=10, le minimum semble se trouver pour la régression ridge. On peut récupérer la valeur minimum :

min(errors)

2.0722706805123288

C'est déjà mieux 🙂

Pour effectuer une cross validation de la régression ridge, vous pouvez utiliser la fonction sklearn.linear\_model.RidgeCVqui effectuer une recherche automatique des hyperparamètres. J'ai ici effectué une recherche manuelle pour le TP.

Comparons maintenant avec le Lasso !

### 

### Application du Lasso

On teste aussi un certain nombre d'hyperparamètres pour appliquer le lasso

n\_alphas = 300

alphas = np.logspace(-5, 1, n\_alphas)

lasso = linear\_model.Lasso(fit\_intercept=False)

coefs = []

errors = []

for a in alphas:

lasso.set\_params(alpha=a)

lasso.fit(X\_train, y\_train)

coefs.append(lasso.coef\_)

errors.append([baseline\_error, np.mean((lasso.predict(X\_test) - y\_test) \*\* 2)])

On peut maintenant afficher les résultats

ax = plt.gca()

ax.plot(alphas, coefs)

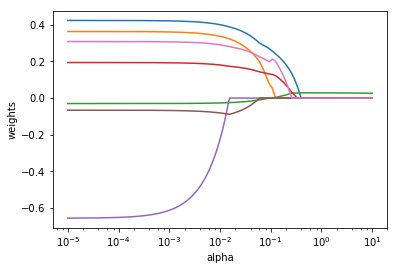
ax.set\_xscale('log')

plt.xlabel('alpha')

plt.ylabel('weights')

plt.axis('tight')

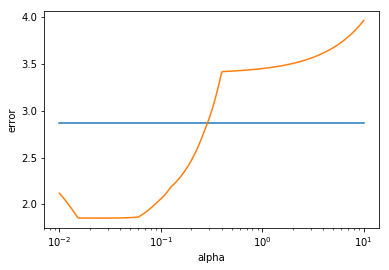
plt.show()



Les poids en fonction de alpha

Comme on peut le voir, le lasso permet de supprimer des variables en mettant leur poids à zéro. C'est le cas si deux variables sont corrélées. L'une sera sélectionnée par le Lasso, l'autre supprimée. C'est aussi son avantage par rapport à une régression ridge qui ne fera pas de sélection de variables.

On peut observer maintenant le comportement de l'erreur.



L'erreur en fonction de la valeur alpha

Le minimum est atteint à

1.8531561201728328

On fait encore mieux qu'avec la régression ridge! En effet comme vu dans les chapitres précédents, le lasso a pour avantage de pouvoir sélectionner un sous-ensemble des variables explicatives afin de permettre une meilleure généralisation.

De même, la fonction sklearn.linear\_model.LassoCVpermet d'effectuer une recherche des hyperparamètres de manière automatisée

### Conclusion

Avec ce TP vous avez pu mettre en oeuvre de manière pratique la régression ridge et le lasso. La régularisation fonctionne bien dans les deux cas, et permet de réduire l'erreur totale du modèle à l'aide de l'ajout du biais qui quantifie la complexité. L'un diminue grandement l'influence de certaines variables sur le modèle tandis que l'autre peut directement les supprimer (mettre leur poids à zéro), et donc d'être parcimonieux.

# Partie 1

Bravo ! Vous avez réussi cet exercice !

### Compétences évaluées

* Comprendre les points communs et différences entre lasso et régression ridge
* Contrôler les poids affectés aux différentes variables par un mécanisme de régularisation en régression
* Entraîner un algorithme linéaire de régression

### Question 1

**Afin d’effectuer une maximisation de la vraisemblance pour trouver les coefficients**ββ**d’une régression linéaire, il est nécessaire d’avoir les hypothèse préalables suivantes :**

*Attention, plusieurs réponses sont possibles.*

* + 

Linéarité des données

* + 

Normalité du bruit

* + 

Les réalisations x(i) sont indépendantes de y(i)

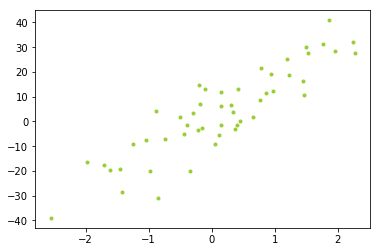
* + 

Les observations sont iid

*Les paires de données*(x,y)*doivent être indépendantes les unes des autres, mais on essaie justement de trouver un lien entre x et y !*

### Question 2

**Quel est le vecteur de paramètres optimal**β**d’une régression linéaire effectuée sur les observations ci-dessous?**

****

* + 

β=(15,15)

* + 

β=(5,−15)

* + 

β=(0,15)

*La pente n'est clairement pas négative, il n'y a pas d'ordonnée à l'origine*

### Question 3

**Quel est le vecteur de paramètres optimal**β**d’une régression linéaire effectuée sur les observations ci-dessous?**

|  |  |
| --- | --- |
| X | y |
| -0.78768 | -34.59703199 |
| -1.51760513 | -30.79543532 |
| 0.74416271 | 19.31018182 |
| -0.62288928 | -19.44809959 |

* + 

52

* + 

13

* + 

26

* + 

32

*Il suffit d'appliquer la formule :*

### Question 4

**La régularisation du modèle est nécessaire lorsque :**

*Attention, plusieurs réponses sont possibles.*

* + 

Il y a trop d’observations par rapport au nombre de features

* + 

Il y a trop de features par rapport au nombre d’observations

* + 

Des observations sont redondantes dans le dataset

* + 

Des features sont corrélées entre elles

*Lorsque les features sont corrélées entre elles, la régression linéaire devient instable (la solution n'est plus unique et les poids ont une plus grande variabilité).  De plus, lorsque le nombre de features est plus grand que le nombre d'observations, la matrice  n'est plus inversible*

### Question 5

**Comment choisir au mieux l’hyperparamètres**λ**?**

* + 

Par validation croisée avec recherche sur grille

* + 

En observant l’évolution de l’erreur en fonction de lambda

* + 

Par le principe de parcimonie

### Question 6

**Le paramètre de régularisation a pour effet de :**

*Attention, plusieurs réponses sont possibles.*

* + 

Diminuer l’importance des poids

* + 

Augmenter l’importance des poids

* + 

Donner autant d’importance aux variables corrélées

* + 

Mieux distinguer les variables corrélées les unes des autres

### Question 7

**La différence principale entre l’utilisation de l’algorithme du lasso et de la régression ridge est :**

*Attention, plusieurs réponses sont possibles.*

* + 

que la régression ridge élimine certaines features du modèle

* + 

que la régression ridge prend plus de temps que le lasso

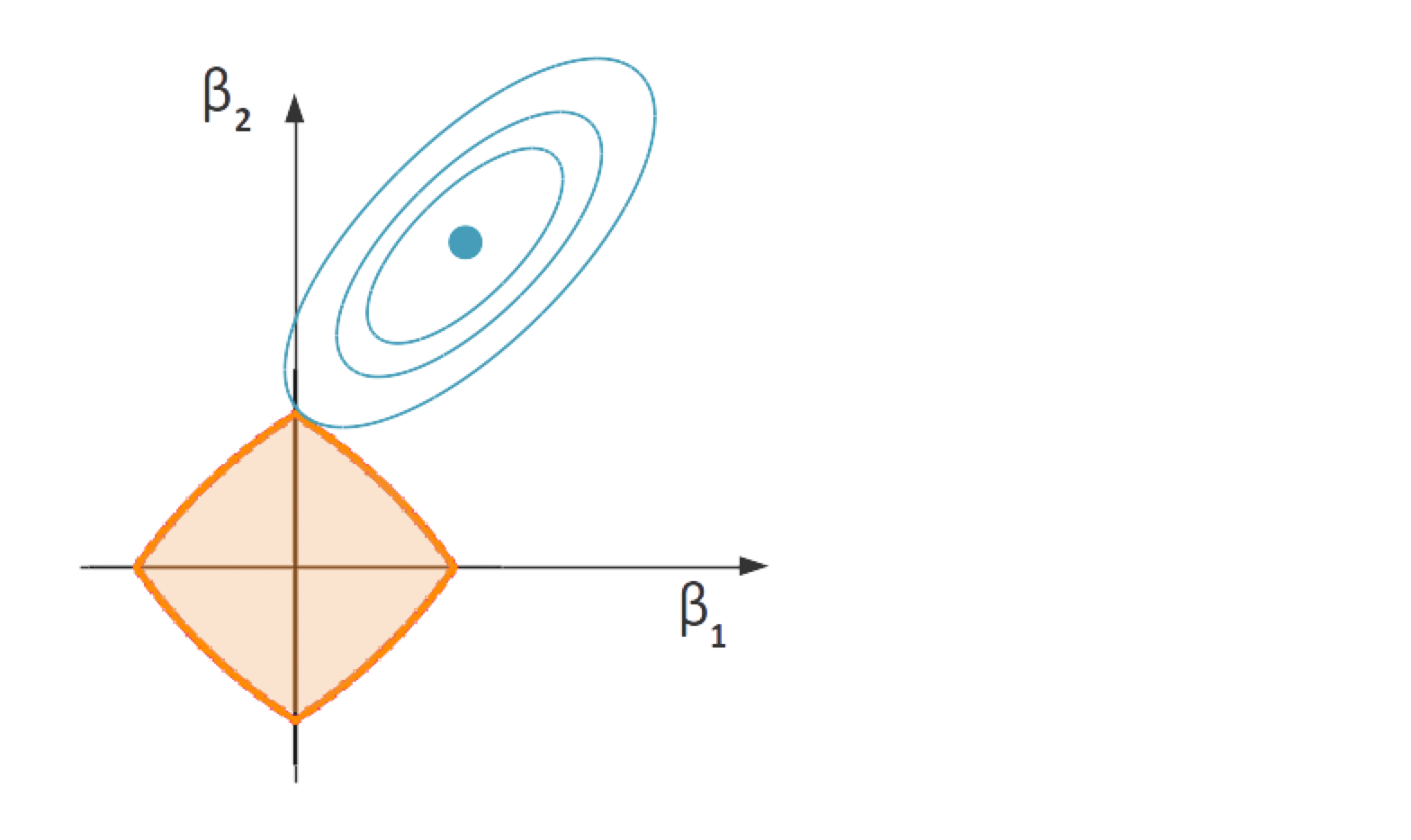
* + 

que le lasso sélectionne un sous-ensemble des features lorsque certaines variables sont corrélées

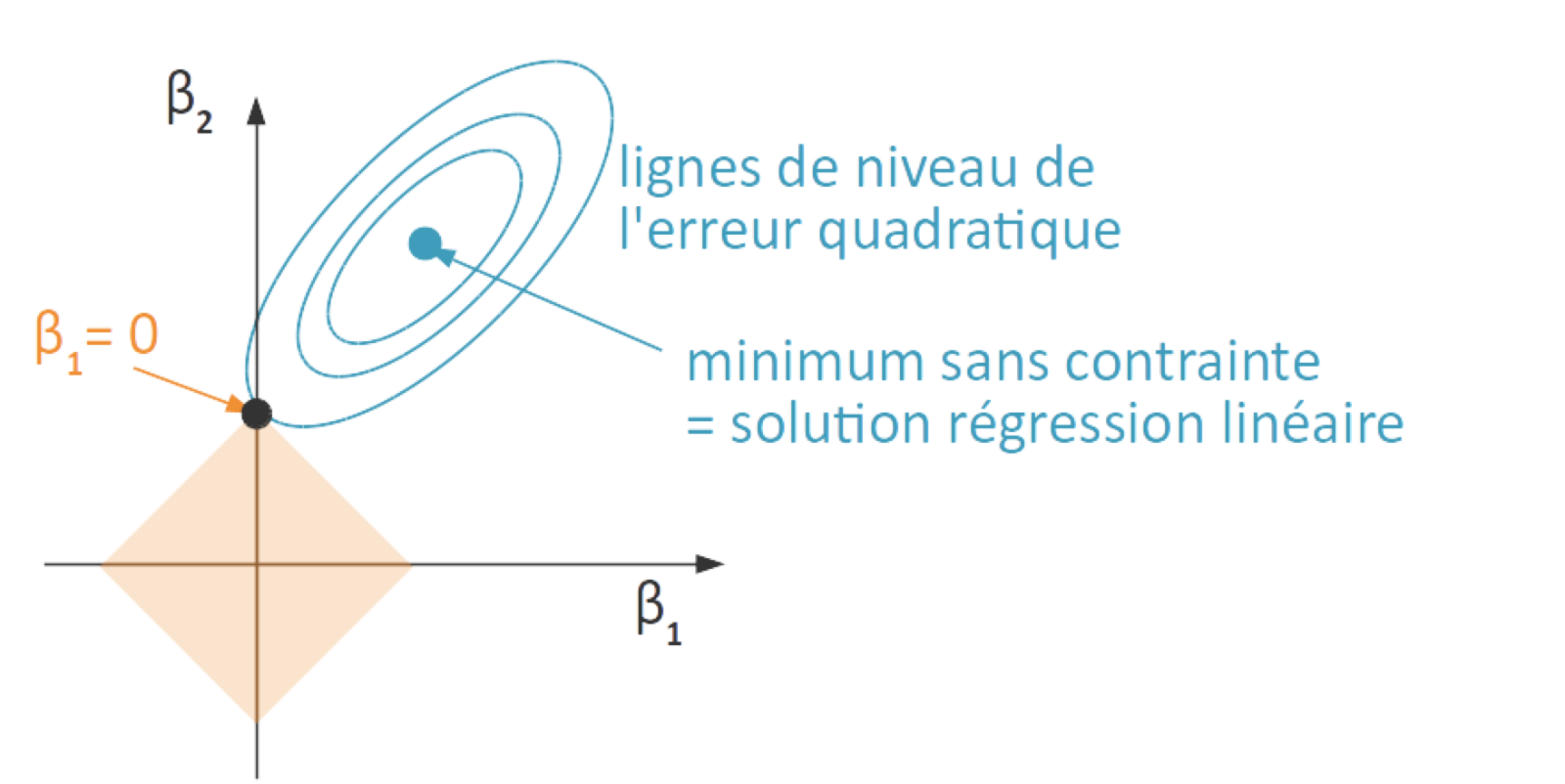
### Question 8

**Lequel de ces schémas représente la régression ridge**

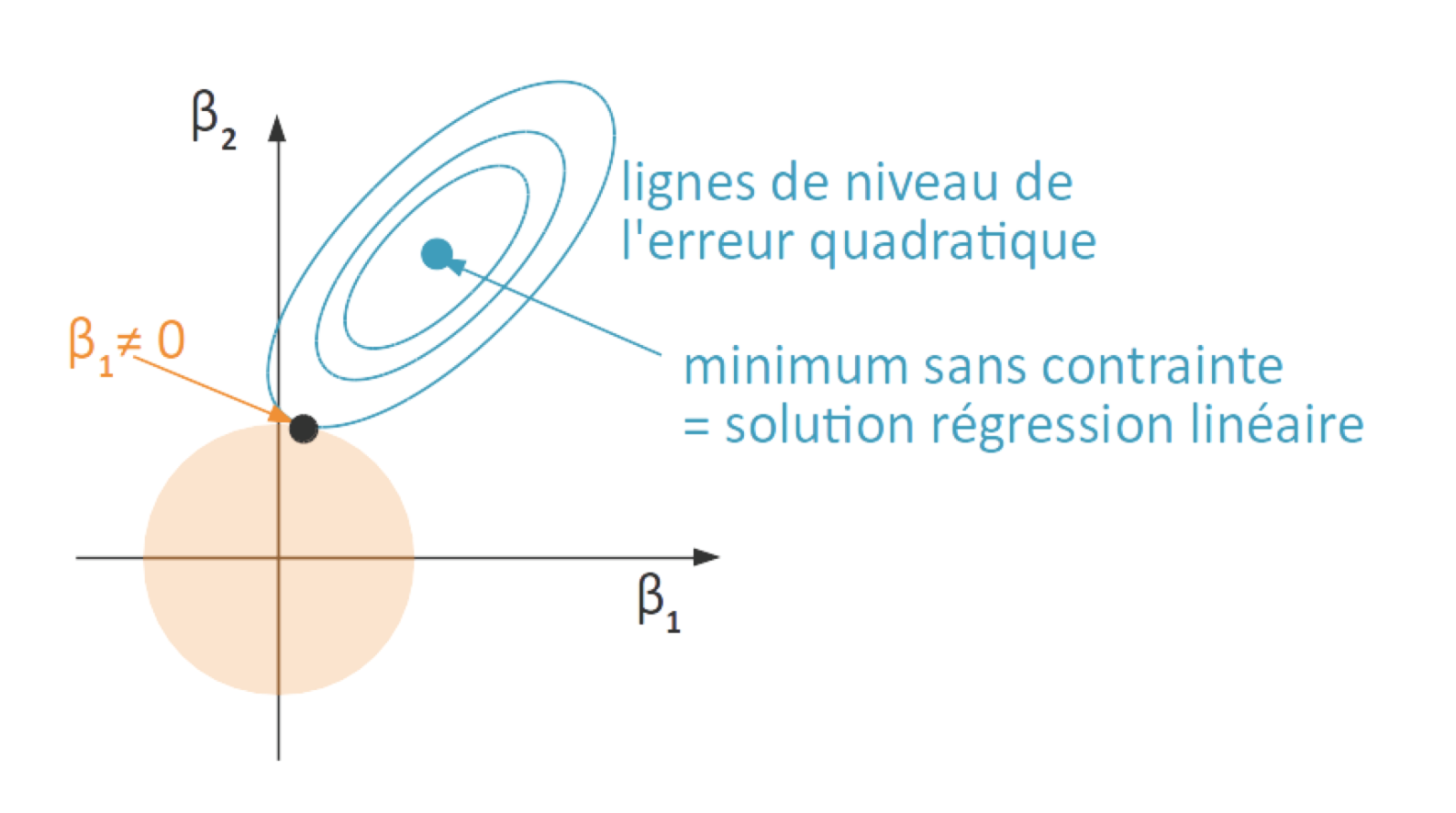
* + 



* + 



* + 



*La régression ridge possède un paramètre de régularisation quadratique et est donc représenté sur ces schémas par un cercle.*

### Question 9

**Le lasso est :**

* + 

Moins stable que la régression ridge

* + 

Plus stable que la régression ridge

*Le lasso est moins stable car il élimine des variables du modèle*

## Prédisez linéairement la probabilité de l’appartenance d’un point à une classe

### 

### Prédire linéairement l'appartenance à une classe

Nous allons maintenant nous intéresser à des problèmes de **classification binaire.** Nos données sont toujours **n** points en **p** dimensions, représentés par la matrice , mais leurs étiquettes, représentées par un vecteur , représentent maintenant l'appartenance (1) ou non (0) à une classe.

Par exemple, les **p** variables peuvent représenter le niveau de gris des  pp pixels d'une image, et on peut disposer de  nn images étiquetées en fonction de si elles représentent ou non un panda. Comment créer un modèle linéaire qui prédise si une image représente ou non un panda ?

Nous savons déjà (voir le chapitre [Trouvez une combinaison linéaire de variables qui approxime leurs étiquettes](https://openclassrooms.com/courses/entrainez-un-modele-predictif-lineaire/titre-de-votre-premier-chapitre-111) de ce cours) résoudre des problèmes de **régression linéaire.** Peut-on utiliser cette base pour résoudre ce problème ?

#### Application directe de la régression linéaire

Dans un problème de régression, les étiquettes  sont à valeurs dans . 0 et 1 sont des éléments de . Peut-on utiliser directement une régression linéaire ?

Le problème de cette approche est que beaucoup de points de coordonnées différentes doivent avoir la même étiquette exactement :   pour tous les points  positifs. Une fonction linéaire n'est pas le bon outil pour cela…

#### Transformation d'une fonction linéaire en probabilité

Très bien, s'il faut que les points positifs puissent avoir des valeurs différentes les unes des autres, et pareil pour les points négatifs, peut-on essayer de prédire un nombre proche de 1 pour les points positifs, et un nombre proche de 0 pour les points négatifs ?

D'accord, mais « proches » et « différents » comment ? Cela ressemble de plus en plus à une **probabilité**: et si l'on essayait plutôt de prédire ?

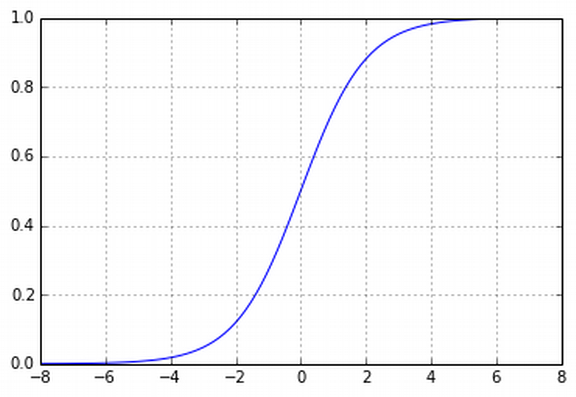
Il nous reste encore quelques problèmes :

* Une probabilité est comprise entre 0 et 1. Une fonction linéaire, à moins d'être constante, va nous sortir des nombres entre −∞ et + ∞ .
* Les probabilités ne se comportent pas linéairement : on voudrait une fonction qui varie peu près de 0 et près de 1, et plus fortement près de 0.5.

Pourquoi est-ce que je dis que les probabilités ne se comportent pas linéairement ? Imaginons que je sois absolument convaincue qu'un point est positif, parce qu'il est très loin de tous les points négatifs (comparativement aux autres points positifs). J'aimerais prédire une probabilité très proche de 1, disons 0.999. Imaginons maintenant que je déplace légèrement mon point. Il va rester très loin de tous les points négatifs. Ma probabilité ne devrait pas changer de beaucoup.

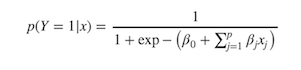
Par contre, prenons un point situé entre les points positifs et les points négatifs. S'il est un peu plus proche des positifs, j'aimerais prédire une probabilité supérieure à 0.5. Si maintenant j'effectue exactement la même translation que précédemment. La prédiction, si elle est linéaire, va bouger d'autant que précédemment. Mais cette translation peut amener mon point plus près des négatifs, auquel cas je voudrais que ma prédiction devienne inférieure à 0.5, et change plus radicalement que précédemment quand sa valeur était proche de 1.

Pour résoudre ces problèmes, nous allons utiliser une **transformation logistique** : au lieu de prédire  directement comme la valeur d'une fonction linéaire en x, nous allons composer cette fonction linéaire avec la fonction logistique : .



Fonction logistique.

Nous avons donc le modèle suivant :

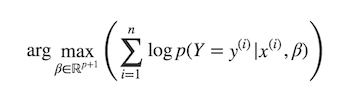


C'est ce qu'on appelle une **régression logistique.**

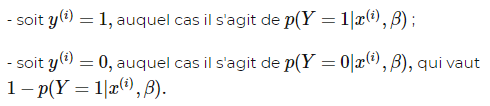
Oui, on a bien appelé régression logistique un modèle qui permet de faire de la classification...

### Solution de la régression logistique

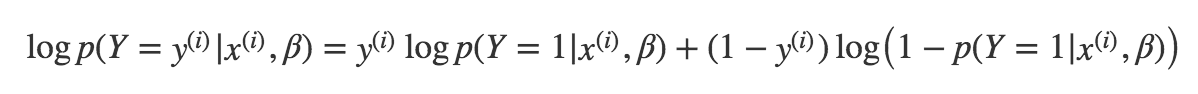
Comme la régression linéaire, la régression logistique peut s'apprendre par maximum de vraisemblance (voir la [section correspondante](https://openclassrooms.com/courses/4444646/4444653?status=draft#/id/r-4521330) du chapitre « Trouvez une combinaison linéaire de variables qui approxime leurs étiquettes » de ce cours). Souvenez-vous, nous allons chercher à maximiser le logarithme de la vraisemblance, à savoir résoudre



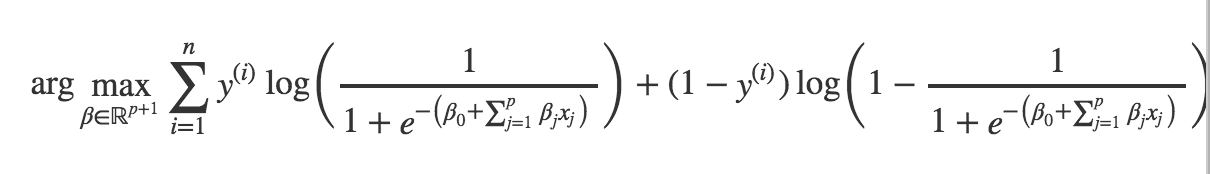
Comment calculer  ? Il y a deux cas possibles :

- 

Autrement dit :



Il ne nous reste plus qu'à remplacer  par sa valeur, à savoir la transformation logistique d'une combinaison linéaire des variables, et on obtient le problème suivant :



Contrairement au cas de la régression linéaire, le gradient de la fonction objective (celle que l'on cherche à maximiser) n'a pas de forme explicite et nous ne pouvons donc pas trouver de solution analytique. Cependant, la fonction objective est concave et nous pouvons utiliser la méthode du gradient.

La régression logistique est implémentée dans scikit-learn : [sklearn.linear\_model.LogisticRegression](http://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LogisticRegression.html).

### Régularisation

Nous avons vu dans les chapitres précédents que l'on peut utiliser la régularisation pour contrôler les coefficients d'une régression linéaire, et éviter le sur-apprentissage ou créer des modèles parcimonieux. Les mêmes concepts s'appliquent à la régression logistique, à la différence que la régression logistique régularisée par la norme **ℓ2** n'admet pas de solution explicite.

On pourra donc utiliser :

* La régression logistique avec régularisation **ℓ2** pour éviter le sur-apprentissage (dans scikit-learn, c'est même l'implémentation par défaut de la régression logistique) ;
* La régression logistique avec régularisation **ℓ1** pour obtenir un modèle parcimonieux (dans scikit-learn, il suffit d'utiliser l'option'penalty'=l1).

### Conclusion

* La **régression logistique** modélise la probabilité qu'une observation appartienne à la classe positive comme une transformation logistique d'une combinaison linéaire des variables.
* Les coefficients d'une régression logistique s'apprennent par maximisation de vraisemblance, mais il n'existe pas de solution explicite.
* La vraisemblance est convexe, et de nombreux solveurs peuvent être utilisés pour trouver une solution numérique.
* Les concepts de **régularisation** **ℓ1** et **ℓ2** s'appliquent aussi à la régression logistique.

**Maximisez la marge de séparation entre vos classes**

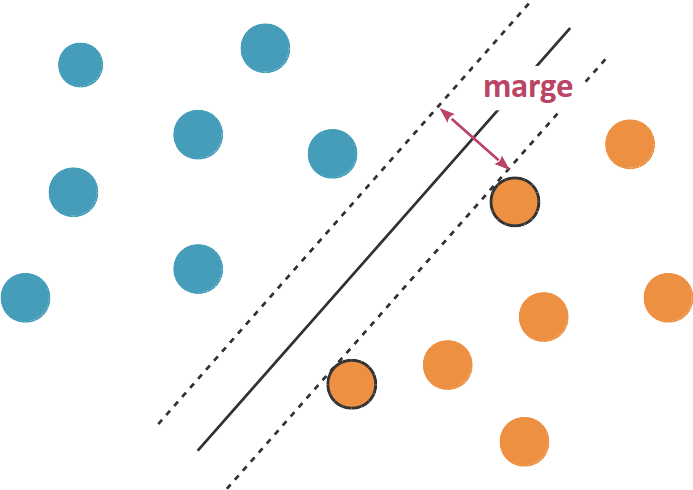
**Cas linéairement séparable**

Vous avez déjà vu comment transformer la régression linéaire pour prédire la probabilité de l'appartenance d'un point à la classe positive. Mais puisqu'on parle de modèles linéaires, peut-on essayer de trouver un hyperplan (en deux dimensions, une droite) qui sépare « au mieux » les deux classes ?

Comment définir « au mieux » ? Commençons par considérer le cas **linéairement séparable**, autrement dit celui dans lequel il existe au moins un hyperplan  tel que tous les points d'une classe soient d'un côté de  et tous les points de l'autre classe soient de l'autre côté de .

**Marge d'un hyperplan séparateur**

En fait, si les données sont linéairement séparables, il existe généralement une infinité d'hyperplans séparateurs qui classifient correctement les données. (le schéma ci-dessous, en dimension 2, devrait vous en convaincre). Pour formaliser lequel parmi ces multiples hyperplans nous convient le mieux, nous allons définir **la marge** d'un hyperplan séparateur  comme deux fois la distance de  au point du jeu de données qui en est le plus proche. (Il peut y avoir *plusieurs* points qui sont les plus proches de , comme sur le schéma ci-dessous.)



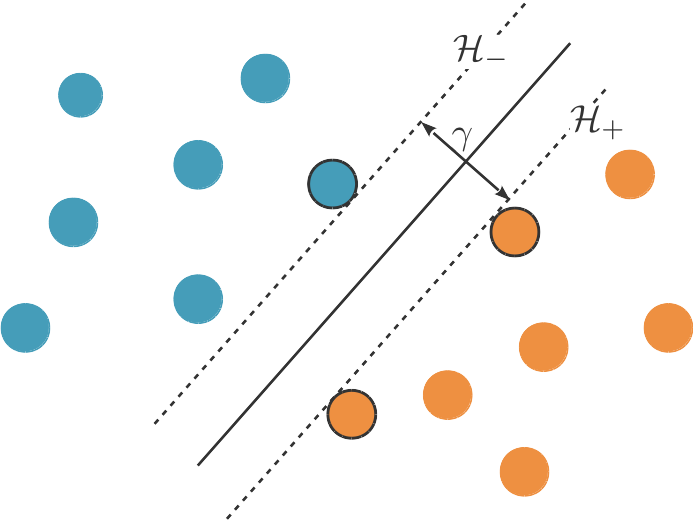
La marge est le double de la distance entre l'hyperplan séparateur et les observations les plus proches (points cerclés de noir).

Comment utiliser ce concept de marge pour choisir un hyperplan séparateur ? L'idée derrière les SVMs est de chercher un hyperplan de **marge maximale.**Un hyperplan de marge maximale se situe à mi-distance entre les points positifs et les points négatifs dont il est le plus proche (voir illustration ci-dessous). En effet, si  est plus proche des points positifs que des points négatifs, on peut le déplacer vers les points négatifs et augmenter sa marge.

Un hyperplan séparateur  de marge maximale **γ** définit une zone, un « *no man's land* » en quelque sorte, délimitée par deux hyperplans et  parallèles à  , situés chacun à une distance **γ/2** de , et entre lesquels il n'y a aucun point du jeu de données. Ce *no man's land*est souvent appelé la **zone d'indécision** car on manque d'information pour classifier les points dans cette zone (notre décision de positionner la **frontière de décision**, c'est-à-dire l'hyperplan séparateur, au milieu de cette zone est basée sur un choix de modélisation et non pas sur les données).

Le(s) point(s) positif(s) le(s) plus proche(s) de  sont situés sur , et de même, le(s) point(s) négatif(s) le(s) plus proche(s) de  sont situés sur . Ainsi, ces points supportent (ou, disons,*soutiennent*)  et    et s'appellent les **vecteurs de support**, d'où le nom de **machine à vecteurs de support**ou **Support Vector Machine (SVM)**de l'algorithme que nous allons découvrir aujourd'hui.

Certains auteurs français utilisent le nom de **Séparatrice à Vaste Marge**pour respecter les initiales SVM. Cependant, les SVM ne sont pas les seuls classifieurs à large marge, qui comptent par exemple parmi eux l'algorithme AdaBoost.



Séparatrice à marge maximale. Les observations cerclées de noirs sont les vecteurs de support.

**Formulation du SVM**

Très bien, mais comment trouver un hyperplan séparateur à marge maximale ?

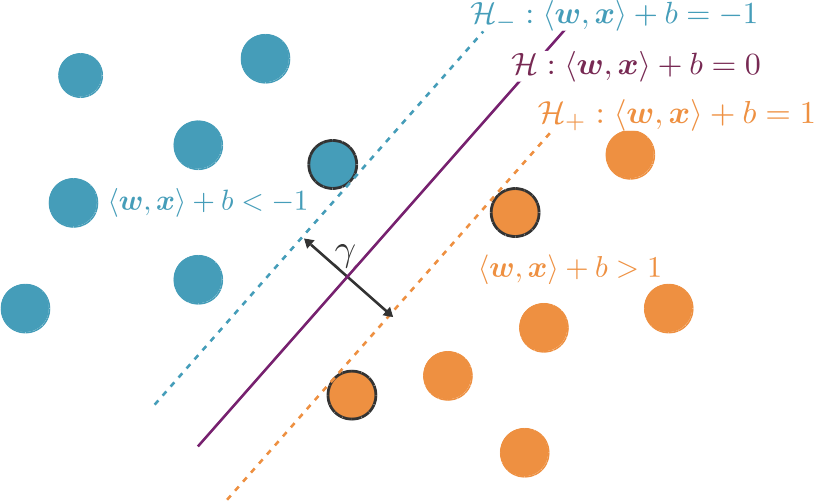
Considérons nos données : n points en p dimensions, représentés par la matrice , d'étiquettes représentées par un vecteur .

J'utilise ici -1 et non pas 0 pour la classe négative pour simplifier les calculs.

L'équation d'un hyperplan en dimension p est paramétrisée par les coordonnées du vecteur normal à cet hyperplan,  ainsi que par un scalaire . Nous pouvons ainsi poser que l'équation de notre hyperplan séparateur de marge maximale est .

Ici **⟨w,x⟩** désigne le **produit scalaire** entre **w** et **x**. Ainsi,  est d'équation . Les **wj** correspondent aux **βj** utilisés précédemment et **b** correspond à **β0**. Je prends ici les notations traditionnellement utilisées pour les SVM car vous les retrouverez ailleurs.

Les hyperplans  et  sont parallèles à  et ont donc le même vecteur normal **w**. De plus, ils sont équidistants par rapport à , et leurs équations peuvent donc  et  ou **K** est une constante. Comme nous pouvons arbitrairement multiplier **w**, **b** et **K** par n'importe quel scalaire pour obtenir les mêmes équations, nous pouvons fixer **K=1**, et ainsi avoir  et .



Nous pouvons maintenant utiliser ces équations pour deux choses :

- Calculer la marge **γ**, autrement dit la distance entre  et . Quelques manipulations vous permettront d'arriver à .

- Garantir que nos données soient correctement classifiées : les points positifs doivent tous être à l'extérieur de , et les points négatifs doivent tous être à l'extérieur de .  Plus précisément, nous cherchons à ce que tous les points d'étiquette  vérifient , et que tous les points d'étiquette  vérifient . On peut combiner ces deux conditions en une seule : .

Nous avons donc le problème d'optimisation suivant :



 Pour simplifier les calculs, nous allons choisir de non pas maximiser , mais de minimiser , ce qui est équivalent. Nous obtenons ainsi le problème suivant :



Il s'agit d'un problème d'optimisation quadratique sous contrainte, qui peut être résolu avec un certain nombre de solveurs numériques. Cependant nous verrons plus bas que l'on peut le reformuler d'une façon qui nous donnera une solution plus facilement interprétable.

**Prédire avec une SVM**

D'accord, mais avant ça, comment utiliser une SVM pour faire des prédictions ?

Supposons notre problème d'optimisation résolu. Nous avons donc trouvé et  qui minimisent la fonction objective (correspondant à l'inverse de la marge) sous contrainte que tous les points du jeu de données soient du bon côté de la zone d'indécision.

Nous allons maintenant étiqueter comme *positif*  un point qui est du même côté de  que les points positifs du jeu d'entraînement, autrement dit **x** pour lequel . À l'inverse, nous prédisons comme *négatif* un point **x** situé de l'autre côté de l'hyperplan séparateur, autrement dit pour lequel 

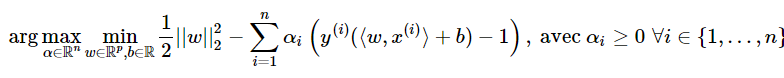
Nous utilisons donc directement l'hyperplan séparateur comme **fonction de décision :**



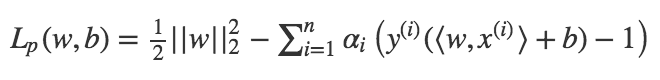
Si cette fonction est positive, le point est étiqueté positif, et inversement, si elle est négative, le point est étiqueté négatif.

**Formulation duale et interprétation géométrique**

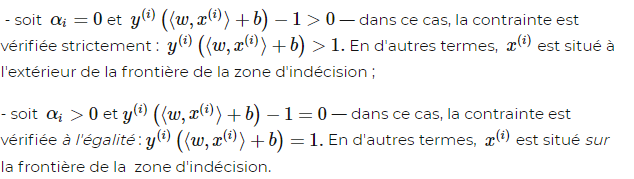
Selon le vocabulaire en vigueur dans le domaine de l'optimisation sous contraintes, le problème d'optimisation que nous avons posé plus haut s'appelle un **problème primal.**Par une technique connue sous le nom des **multiplicateurs de Lagrange**, ce problème primal peut être reformulé en introduisant un scalaire **αi** (appelé, sans surprise, un multiplicateur de Lagrange) pour chacune de nos **n** contraintes ; il est équivalent à



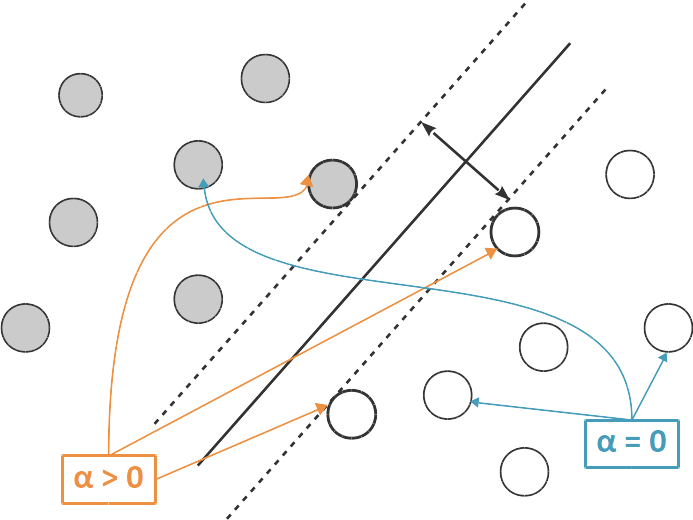
La fonction suivante s'appelle le **lagrangien** du problème.



De plus les coefficients **αi** vérifient les conditions, dites de **Karush-Kuhn-Tucker** (ou **KKT**), que pour tout i∈{1,…,n},

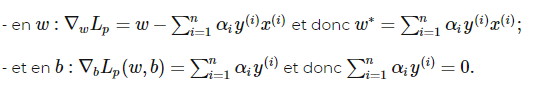


Cela signifie que les points x(i) pour lesquels αi>0 sont les vecteurs de support, et ceux pour lesquels αi=0 sont ceux situés loin de et .

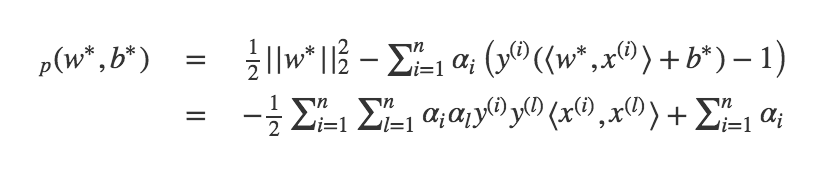


Les vecteurs de support sont ceux pour lesquels  αα est non nul.

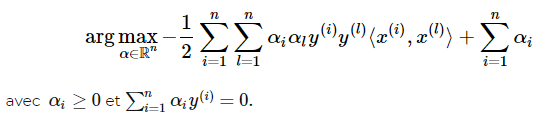
 Appelons  une solution de la minimisation du lagrangien. Celle-ci peut s'obtenir en annulant le gradient du lagrangien



Et donc



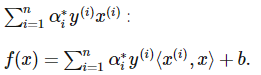
Le problème d'optimisation devient donc



Cette nouvelle formulation s'appelle la **forme duale**du SVM. On peut donc trouver les coefficients du SVM en résolvant ce problème d'optimisation plutôt que le primal.

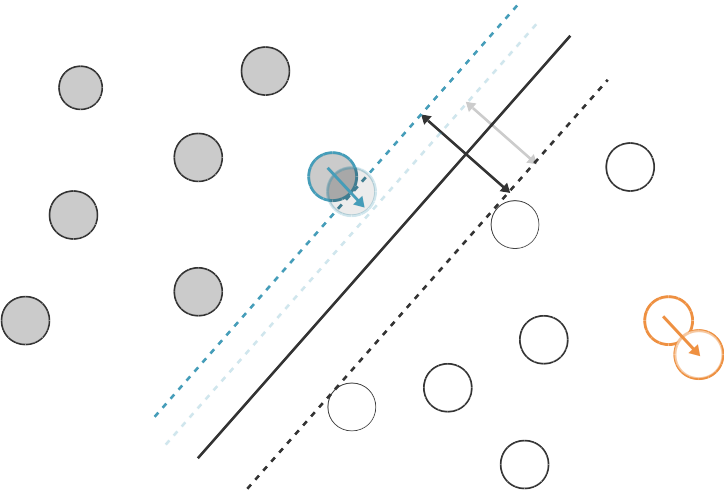
Mais on aurait pu résoudre le primal directement, non ? Quel est l'intérêt d'avoir fait tous ces calculs ?

Une fois que nous avons trouvé la solution  du problème dual, nous pouvons réécrire notre fonction de décision en remplaçant  par sa valeur



Nous avons donc écrit notre fonction de décision **en termes de produits scalaires entre le point à étiqueter et les points du jeu d'entraînement.** Cela jouera un rôle important dans le fait de pouvoir étendre les SVM à des problèmes de classification non linéaire, comme nous le verrons dans un autre cours.

Par ailleurs, souvenez-vous de notre interprétation géométrique plus haut : seuls certains points auront un coefficient  et ces points sont les vecteurs de support. **La solution ne dépend donc pas des points qui ne sont pas vecteurs de supports,** pour lesquels  Et c'est logique, car si l'on déplace un peu ces points, ils restent loin de la zone d'indécision. et l'hyperplan séparateur ne changera pas. Par contre, si l'on déplace un vecteur de support, il pourra entraîner avec lui  (ou s'il s'agit d'un point négatif), ce qui déplacera .



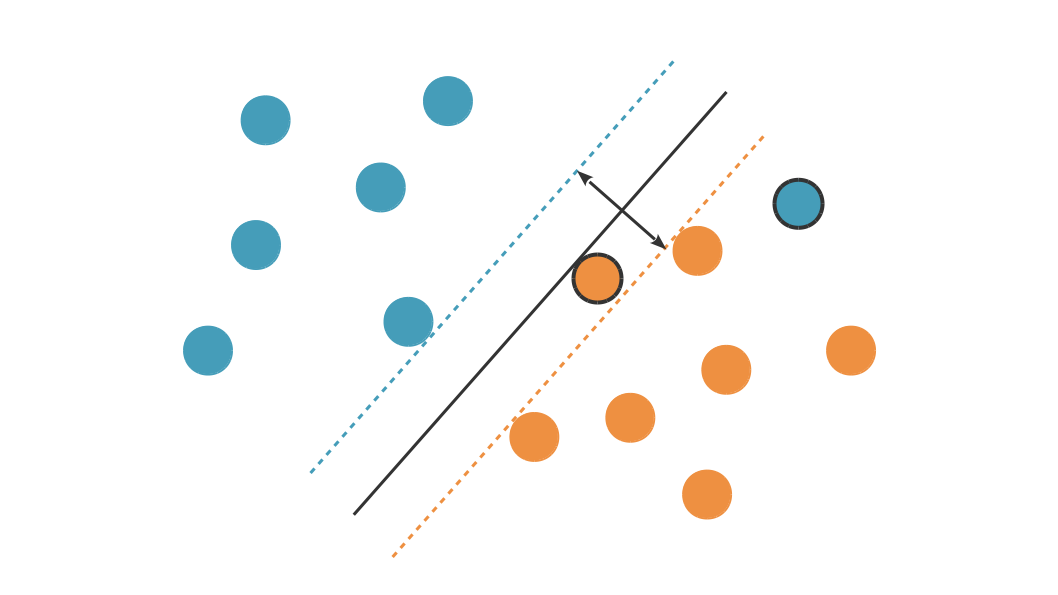
Déplacer le point orange, qui n'est pas un vecteur de support, ne change pas la solution. Déplacer le point bleu, qui est vecteur de support, modifie l'hyperplan séparateur.

Ainsi, notre fonction de décision peut être calculée sur la base du produit scalaire entre le point à étiqueter et les vecteurs de support uniquement.

Remarquons que le primal est un problème d'optimisation en p dimensions, alors que le dual est un problème d'optimisation en n dimensions. Si p≪n, alors résoudre le primal sera plus efficace. À l'inverse, si l'on a peu d'échantillons et beaucoup de variables (n≪p), résoudre le dual sera plus efficace.

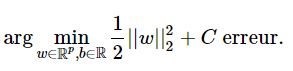
**Cas non-linéairement séparable**

Malheureusement, dans la plupart des cas, les données ne sont pas linéairement séparables. Il va donc falloir que nous acceptions de faire des erreurs, autrement dit que certains points de notre jeu d'entraînement se retrouvent *du mauvais côté de la frontière de la zone d'indécision.*



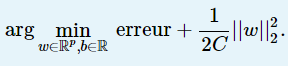
Le point orange cerclé de noir est du bon côté de l'hyperplan séparateur, mais du mauvais côté de H- (la ligne orange en pointillés). Le point bleu cerclé de noir est du mauvais côté de l'hyperplan séparateur.

Plus la marge est grande, plus nous somme susceptibles d'avoir d'erreurs. Nous allons donc formuler un nouveau problème : minimiser la marge *et*l'erreur simultanément :



L'hyperparamètre C sert à quantifier l'importance relative du terme d'erreur et du terme de marge.

Nous pourrions écrire ce problème comme

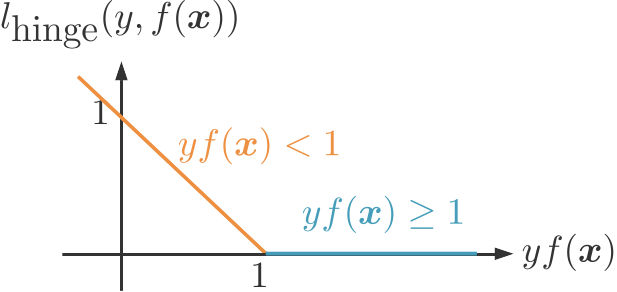


 Il s'agit donc bien d'une **régularisation ℓ2**, et le coefficient C vaut **1/(2λ)** par rapport à ce que nous avons pu écrire pour la régression ridge ou la régression logistique régularisée. L'erreur que nous allons utiliser, par contre, est différente de celle que nous avons utilisée pour la régression logistique régularisée.

**Perte hinge**

Comment mesurer l'erreur de notre classifieur ? Rappelez-vous, nous souhaitons que  soit du bon côté de la frontière de la zone d'indécision, autrement dit, que  Si c'est le cas, nous voulons que notre fonction d'erreur vaille 0. Sinon, nous allons définir une erreur qui est d'autant plus grande que  s'éloigne de la valeur 1, autrement dit, que  s'éloigne de la frontière ( s'il est positif ou  s'il est négatif).

Cette erreur s'appelle la **perte hinge**, ou « ***hinge loss*** » en anglais. « *Hinge* » signifie « charnière » en anglais, et correspond à la forme « en coude » de la perte hinge.



Perte hinge.

La perte hinge peut s'écrire de manière analytique de la fonction suivante :

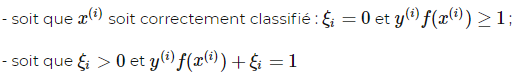


**SVM à marge souple**

Nous pouvons maintenant formuler notre problème d'optimisation :

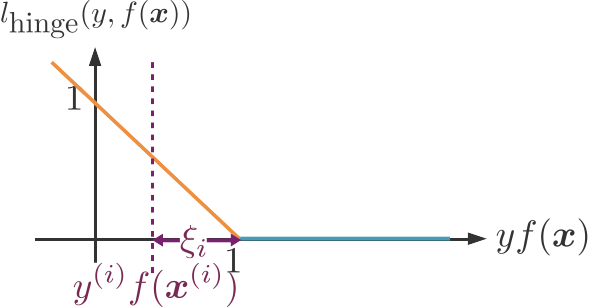


Malheureusement cette nouvelle fonction objective ne peut pas être minimisée analytiquement. Nous allons donc introduire n **variables d'ajustement ξi**, appelées ***slack variables*** en anglais, qui vont mesurer la distance entre ) et 1. Pour chaque point du jeu de données, la perte hinge vaut donc **ξi** et l'on souhaite



On peut reformuler ces deux possibilités en deux contraintes :





*La variable d'ajustement***ξi***est la distance entre***yf(x)***et sa valeur souhaitée de 1.*

Nous aboutissons donc à la formulation suivante :



avec pour tout 

C'est la forme primale de ce qu'on appelle le**SVM à marge souple,** ou ***soft-margin SVM*** en anglais.

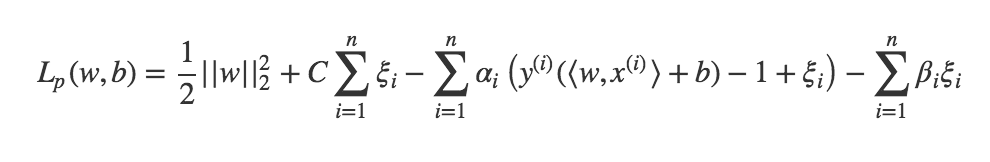
**Formulation duale**

Avec les mêmes outils d'optimisation sous contrainte que pour le cas linéairement séparable, nous pouvons dériver la**formulation duale** du SVM à marge souple et les conditions **KKT** correspondantes.

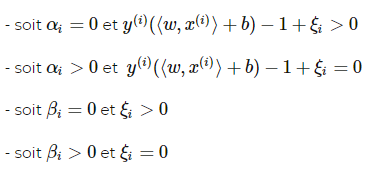
Nous introduisons ainsi deux**multiplicateurs de Lagrange** par observation : αiαi pour la première contrainte, et βiβi pour la deuxième. Le primal devient équivalent à



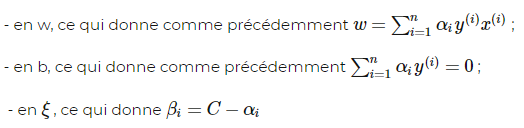
Le **lagrangien** vaut ici



 Les **conditions de Karush-Kuhn-Tucker** sont les suivantes :



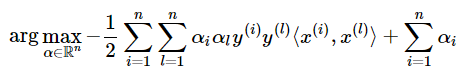
La minimisation du lagrangien se fait en minimisant son gradient :



Cette dernière équation nous permet de lier β à α, et le problème de maximisation en α et β devient un problème de maximisation en αα seulement.

De plus, la contrainte βi≥0 nous donne αi≤C.

La f**orme duale du SVM à marge souple** est donc :

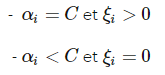


 avec 

**La seule différence avec le cas linéairement séparable** est la contrainte αi≤C !

**Interprétation géométrique**

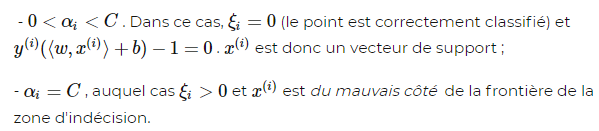
Revenons aux conditions de KKT. Les deux dernières d'entre elles peuvent se réécrire

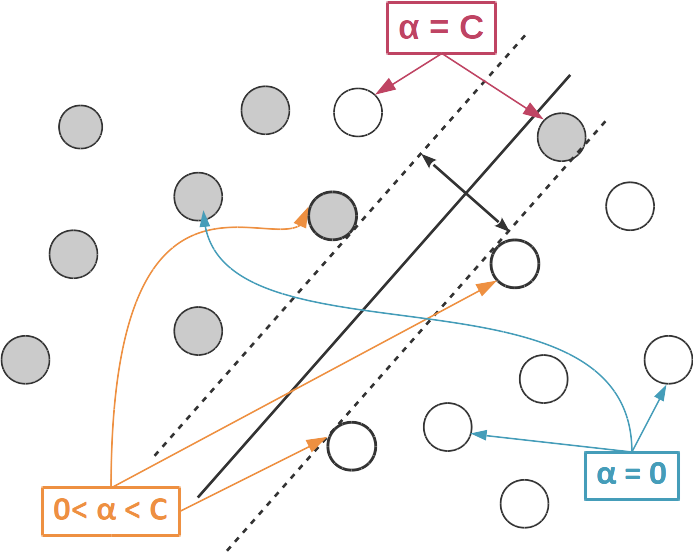


En prenant toutes les conditions de KKT en considération, on arrive donc à 3 cas possibles pour αi :

 - αi=0. Dans ce cas, 

Comme précédemment pour le cas linéairement séparable,  est à l'extérieur de la zone d'indécision  (frontière exclue) ;





*Les points qui vérifient*α=C*sont ceux pour lesquels on a fait une erreur.*

La SVM, aussi appelée **SVC** pour « *Support Vector Classification* », est implémentée dans scikit-learn : [sklearn.svm.LinearSVC](http://scikit-learn.org/stable/modules/generated/sklearn.svm.LinearSVC.html).

Par défaut, quand on appelle LinearSVC, c'est le problème dual (en n dimensions) qui est résolu. Pour résoudre le problème primal (en p dimensions, et donc plus efficace quand on a peu de variables), il faudra utiliser l'option 'dual=False'.

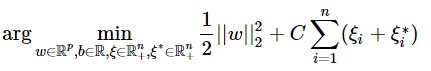
**Et la régression dans tout ça ?**

Les SVM peuvent aussi être utilisées pour des problèmes de régression. On cherche toujours à minimiser , mais les contraintes sont maintenant données par une fonction de perte différente : étant donné un hyperparamètre , nous souhaitons que la différence entre l'étiquette  et sa valeur prédite  soit, en valeur absolue, inférieure à ϵ . Nous séparons cette condition en deux :

Si la différence est positive, nous souhaitons que . Si elle est négative, cette condition sera naturellement vérifiée puisque ϵ>0.

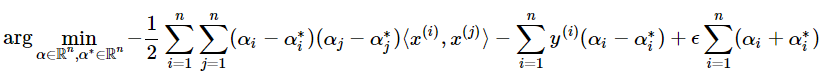
Si la différence est négative, nous souhaitons que .

Comme cela ne pourra pas toujours être le cas, nous introduisons deux variables d'ajustement par observation dans nos données :  pour ajuster la première condition, et  pour ajuster la deuxième. Le primal s'écrit donc :



 sous les conditions que  et  .

Le dual s'écrit



 avec .

La régression SVM, aussi appelée SVR pour Support Vector Regression, est implémentée dans scikit-learn : [sklearn.svm.LinearSVR](http://scikit-learn.org/stable/modules/generated/sklearn.svm.LinearSVR.html). Encore une fois, une implémentation duale et une implémentation primale sont disponibles via l'option 'dual' et il vaut généralement mieux choisir 'dual=False' quand on a beaucoup de données et 'dual=True' quand on a beaucoup de variables.

**Conclusion**

* Les **SVM (Support Vector Machines)**, aussi appelées en français **Machines à Vecteurs de Support**et parfois **Séparatrices à Vaste Marge**, cherchent à séparer *linéairement* les données.
* La version **primale**résout un problème d'optimisation à p variables et est donc préférable si on a moins de variables que d'échantillons.
* À l'inverse, la version **duale**résout un problème d'optimisation à n variables et est donc préférable si on a moins d'échantillons que de variables.
* Les**vecteurs de support** sont les points du jeu de données qui sont les plus proches de l'hyperplan séparateur.
* La **fonction de décision** peut s'exprimer uniquement en fonction du produit scalaire du point à étiqueter avec les vecteurs de support.

Dans un prochain cours, nous verrons comment utiliser cette méthode pour créer des classifieurs  non linéaires.

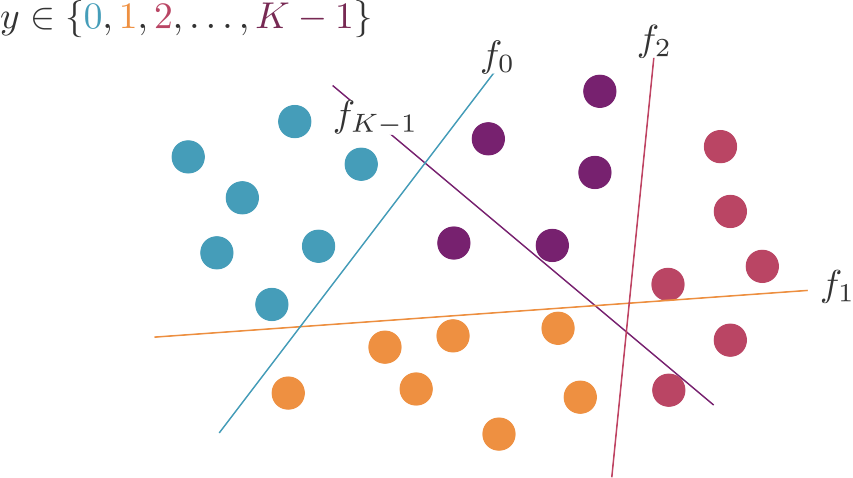
## Classifiez vos données en plus de deux classes

Supposons maintenant que nous ayons plus de deux classes. Par exemple, un problème de classification d'images entre « voiture », « vélo »,  « bus de ville », « piéton », « scooter ». Comment utiliser les classifieurs binaires que nous savons construire pour résoudre un tel problème ?

Nos données sont toujours n points en p dimensions, représentés par la matrice , et leurs étiquettes, . Nous avons donc K classes.

### One-versus-rest

Une première approche consiste à créer un classifieur fk pour chacune des classes, qui sépare les points de cette classe de tous les autres points.



Chaque droite sépare (parfois avec des erreurs) les points d'une classe de tous les autres.

Cette méthode est appelée **one-versus-rest** (« une contre le reste », **OVR** en abrégé) ou **one-versus-all** (« une contre toutes », **OVA** en abrégé).

Nous avons maintenant K classifieurs. Comment obtenir une seule prédiction finale ?

Supposons que nos classifieurs soient des régressions logistiques. Alors , il est logique de prédire que x appartient à la classe pour laquelle cette prédiction est la plus élevée. Dans le cas des SVM,  indique la distance entre x et l'hyperplan qui sépare la classe k des autres. Plus cette valeur est élevée, plus nous sommes convaincus que x appartient à la classe k.

Nous allons donc simplement prédire **la classe pour laquelle la fonction de décision retourne la valeur la plus élevée :**



#### Complexité algorithmique

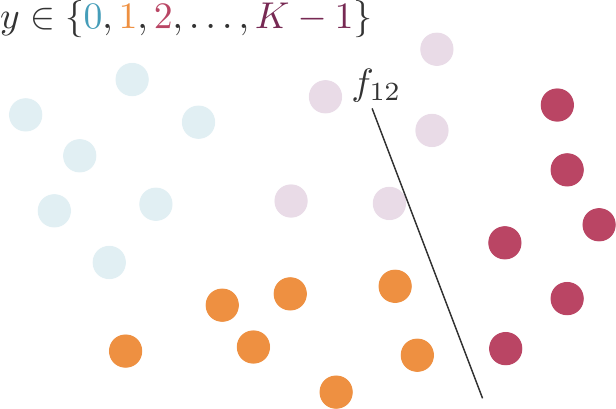
Pour entraîner notre algorithme, nous avons dû entraîner K classifieurs sur n points chacun. Si j'appelle  la complexité algorithmique du classifieur de base, la complexité de l'entraînement de notre OVR est donc .

En ce qui concerne la prédiction, il faut faire K prédictions, puis en prendre le maximum. Nous avons donc une complexité de , où  est la complexité de la prédiction d'un de nos classifieurs de base.

OVR est implémenté dans scikit-learn : [sklearn.multiclass.OneVsRestClassifier.](http://scikit-learn.org/stable/modules/generated/sklearn.multiclass.OneVsRestClassifier.html)

### One-versus-one

Une autre façon de faire consiste à calculer K(K-1) classifieurs, chacun séparant une classe d'une autre en ignorant les autres classes (et les observations qui leur appartiennent).



On sépare la classe 1 de la classe 2 en ignorant les autres classes.

Cette méthode est appelée **one-versus-one** (« une contre une », **OVO** en abrégé).

Comment fait-on une prédiction maintenant, avec K(K-1) classifieurs ?

Pour prédire, nous allons utiliser un **vote de la majorité** : la classe prédite est celle retournée par le plus grand nombre de classifieurs.

Nous n'avons pas besoin de calculer K(K-1) classifieurs, mais seulement   : une fois que nous avons séparé la classe k de la classe l nous savons séparer la classe l de la classe k, autrement dit,  fkl=−flk.

#### Complexité algorithmique

Nous devons maintenant construire  classifieurs plutôt que K. Nos temps de calcul vont augmenter ! Eh bien, tout dépend de . Nous entraînons certes  classifieurs, mais chacun d'entre eux est entraîné sur  points seulement.

OVO est implémenté dans scikit-learn : [sklearn.multiclass.OneVsOneClassifier.](http://scikit-learn.org/stable/modules/generated/sklearn.multiclass.OneVsOneClassifier.html)

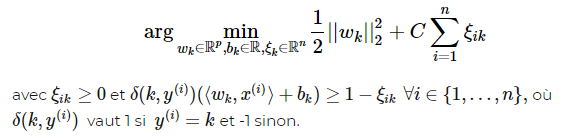
Les principes OVR et OVO peuvent s'appliquer à tous les classifieurs, et pas seulement aux classifieurs linéaires !

### SVM multi-classe

Regardons maintenant de plus près le cas des SVM.

Appliquons l'algorithme OVR dans un contexte multi classe. Nous allons construire l'un après l'autre K hyperplans séparateurs d'équations  . Chacun de ces hyperplans vérifie

argminwk∈Rp,bk∈R,ξk∈Rn12||wk||22+C∑i=1nξikarg⁡minwk∈Rp,bk∈R,ξk∈Rn12||wk||22+C∑i=1nξik



Nous avons donc K problèmes d'optimisation à 2n contraintes à résoudre. Chacune des contraintes dit que chaque point  doit être au pire à une distance ξi du « mauvais côté » de la frontière de la zone d'indécision entre sa classe et l'union des autres classes.

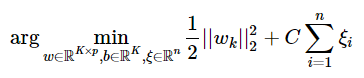
Mais nous pouvons changer la contrainte pour chercher à découvrir les K hyperplans simultanément, de sorte que chaque point du jeu d'entraînement soit (1) du bon côté de l'hyperplan séparateur de sa classe, d'équation  et plus loin de cet hyperplan séparateur que de tous les autres, définis par . Idéalement, nous voudrions donc que pour chaque point ,



Comme cela va certainement ne pas être systématiquement possible, on peut réintroduire ici les variables d'ajustement et imposer



 Nous pouvons donc formuler le SVM multi-classe comme le problème suivant, qui utilise n variables au lieu de   :



 avec .

Comme dans OVR, la prédiction se fait selon la fonction de décision maximale :



Cette méthode a été proposée par**Crammer et Singer** en 2002. Elle est implémentée dans scikit-learn via l'option multi\_class="crammer\_singer" de  [sklearn.svm.LinearSVC](http://scikit-learn.org/stable/modules/generated/sklearn.svm.LinearSVC.html). Par défaut, LinearSVC gère le multi-classe avec une approche OVR.

### Conclusion

* L'approche **one-versus-rest** de la classification multi-classes consiste à créer K classifieurs binaires qui séparent chaque classe k de l'union des autres classes. On prédit alors la classe pour laquelle la fonction de décision est **maximale.**
* L'approche **one-versus-one** consiste à créer K(K-1) classifieurs binaires qui séparent chacun une classe d'une autre, en ignorant tous les autres points. On utilise alors un **vote de la majorité** pour prédire. Cette approche crée plus de classifieurs, mais chacun est entraîné sur moins d'observations.
* On peut créer des**SVM multiclasse** en optimisant simultanément K classifieurs binaires (méthode de **Crammer et Singer**).

Nous verrons aussi dans un prochain cours qu'il existe plusieurs algorithmes non-linéaires qui sont naturellement multi-classes.

## TP - Entraînez une régression logistique et une SVM linéaire

Dans ce chapitre, nous allons comparer les performances des deux modèles de classification binaire étudiés précédemment : la régression logistique et le SVM Linéaire.

### 

### Chargeons le jeu de données

Le jeu de données est téléchargable [**ici**](https://s3-eu-west-1.amazonaws.com/static.oc-static.com/prod/courses/files/Parcours_data_scientist/entrainez-un-modele-predictif-lineaire/TP_2_datset_mushrooms.csv).

C'est encore un jeu de données assez utilisé comme exemple, qui contient des entrées correspondants à des caractéristiques de champignons (surface, couleur, etc), ainsi qu'un indicateur 'class' de comestibilité ('p' pour poisonous, 'e' pour edible).

import pandas as pd

raw\_data = pd.read\_csv('mushrooms.csv')

Première chose, encoder nos données en données chiffres plutôt que des lettres

from sklearn.preprocessing import LabelEncoder

labelencoder=LabelEncoder()

for col in raw\_data.columns:

raw\_data[col] = labelencoder.fit\_transform(raw\_data[col])

raw\_data.head()

On sépare notre jeu de donnée de manière classique entre données d'entraînement et données de test.

from sklearn.model\_selection import train\_test\_split

# On récupère les features d'un côté...

X = raw\_data.iloc[:,1:23]

# et les labels de l'autre

y = raw\_data.iloc[:,0]

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.33)

Maintenant qu'on a préparé notre jeu de données, on peut tester les modèles de classification !

### 

### Appliquez la régression logistique

Pour rappel, la régression logistique peut avoir un paramètre de régularisation de la même manière que la régression linéaire, de norme 1 ou 2.

Observons dans un premier temps la performance de la régression logistique classique

import numpy as np

from sklearn.linear\_model import LogisticRegression

from sklearn.metrics import roc\_curve, auc

lr = LogisticRegression(solver = 'liblinear')

lr.fit(X\_train,y\_train)

# On récupère la prédiction de la valeur positive

y\_prob = lr.predict\_proba(X\_test)[:,1]

# On créé un vecteur de prédiction à partir du vecteur de probabilités

y\_pred = np.where(y\_prob > 0.5, 1, 0)

false\_positive\_rate, true\_positive\_rate, thresholds = roc\_curve(y\_test, y\_prob)

roc\_auc = auc(false\_positive\_rate, true\_positive\_rate)

print(roc\_auc)

La sortie donne une aire sous la courbe de 0.982 (pas forcément exactement la même pour vous). Ce qui est déjà correct.

On peut afficher cette courbe

import matplotlib.pyplot as plt

plt.figure(figsize=(10,10))

plt.title('Receiver Operating Characteristic')

plt.plot(false\_positive\_rate,true\_positive\_rate, color='red',label = 'AUC = %0.2f' % roc\_auc)

plt.legend(loc = 'lower right')

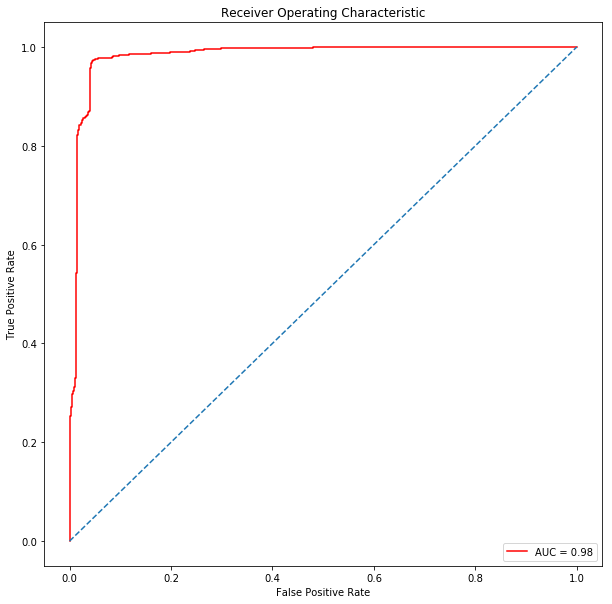
plt.plot([0, 1], [0, 1],linestyle='--')

plt.axis('tight')

plt.ylabel('True Positive Rate')

plt.xlabel('False Positive Rate')

plt.show()



On peut maintenant essayer de tuner cette régression logistique en testant plusieurs paramètres, à l'aide de la fonction GridSearchCV  qui effectue aussi en passant une validation croisée.

from sklearn.model\_selection import cross\_val\_score

from sklearn.model\_selection import GridSearchCV

lr = LogisticRegression(solver = 'liblinear')

params = {'C': np.logspace(-3, 3, 7) , 'penalty':['l1','l2'] }

lr\_gs = GridSearchCV(lr, params, cv=10)

lr\_gs.fit(X\_train, y\_train)

print(lr\_gs.best\_params\_)

{'penalty': 'l2', 'C': 1000.0}

Comme pour la régression linéaire, la différence entre un paramètre de régularisation de norme 1 ou 2 est que celui de norme 1 effectue en même temps une sélection des variables (parmi celles qui sont corrélées) en mettant leur poids à zéro.

On peut évaluer notre modèle de la même manière que pour la régression logistique classique, avec une AUC :

# On récupère la prédiction de la valeur positive

y\_prob = lr.predict\_proba(X\_test)[:,1]

# On créé un vecteur de prédiction à partir du vecteur de probabilités

y\_pred = np.where(y\_prob > 0.5, 1, 0)

false\_positive\_rate, true\_positive\_rate, thresholds = roc\_curve(y\_test, y\_prob)

roc\_auc = auc(false\_positive\_rate, true\_positive\_rate)

print(roc\_auc)

0.9824

On est déjà sur une classification très efficace

### 

### Appliquez une SVM Linéaire

On peut maintenant tester les performances du SVM Linéaire.

from sklearn.svm import LinearSVC

from sklearn.model\_selection import GridSearchCV

from sklearn.metrics import roc\_curve, auc

svm = LinearSVC()

params = { 'C': np.logspace(-3, 3, 7) }

gs\_svm = GridSearchCV(lr, params, cv=10)

gs\_svm.fit(X\_train, y\_train)

print(gs\_svm.best\_params\_)

On peut maintenant passer à l'évaluation

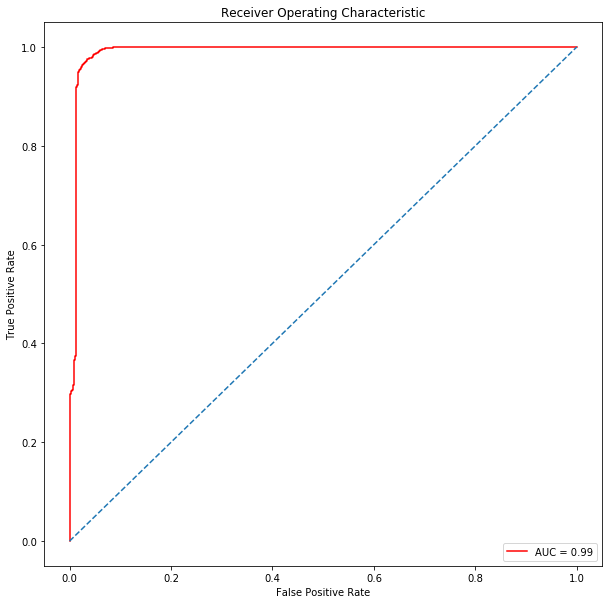
from sklearn.metrics import roc\_curve, auc

false\_positive\_rate, true\_positive\_rate, thresholds = roc\_curve(y\_test, y\_prob)

roc\_auc = auc(false\_positive\_rate, true\_positive\_rate)

print(roc\_auc)

Le SVM Linéaire possède des performances similaires à la régression logistique. Comme on le verra dans un autre cours, son intérêt réside surtout dans l'usage de kernels qui permet de passer dans des espaces non-linéaires à l'aide de la forme duale.



### Conclusion

Les données sont relativement simples à modéliser avec des modèles linéaires classiques. En y ajoutant une pénalisation, on arrive à un niveau de précision tout à fait acceptable. Lorsque l'on a besoin de modéliser des effets non linéaires plus complexes, les modèles linéaires que l'on vient de voir peuvent ne pas suffire : on fait alors appel à d'autres approches que vous approfondirez dans les autres cours.

## Entraînez-vous à classer automatiquement des feuilles d’arbres

### À vous de jouer

Pour vous entraîner, réalisez cet exercice étape par étape. Une fois terminé, vous pouvez comparer votre travail avec les pistes que je vous propose.

Vous allez effectuer une classification multi classe comme vous l'avez vu dans le chapitre précédent.  
Le jeu de données que vous allez utiliser est un dataset de feuilles d’arbres. L’objectif est de les catégoriser par espèce d’arbre à partir de leur caractéristiques. Vous pouvez télécharger le dataset à [ce lien](https://s3-eu-west-1.amazonaws.com/static.oc-static.com/prod/courses/files/Parcours_data_scientist/entrainez-un-modele-predictif-lineaire/Dataset_feuilles_1.csv) et le [test-set](https://s3-eu-west-1.amazonaws.com/static.oc-static.com/prod/courses/files/Parcours_data_scientist/entrainez-un-modele-predictif-lineaire/test.csv.zip). Par ailleurs, le dataset original se trouve ici : <https://www.kaggle.com/c/leaf-classification/data>

**Votre Mission**

Comme dit plus haut votre objectif sera de déterminer quelle est l’espèce de l’arbre à laquelle appartient la feuille.

Les caractéristiques extraites des images des feuilles sont essentiellement 3 vecteurs de dimension 64 (margin, shape & texture), dont la description du dataset détaillée se trouve ici : <https://www.kaggle.com/c/leaf-classification/data>

Utilisez bien l’ensemble des notions vues dans cette section (choix des hyperparamètres, régularisation) afin de pouvoir obtenir les meilleurs performances de classification possible.

Vous devrez donc :

* Créer une baseline de performances avec le K-NN
* Utiliser le SVM multiclasse avec différents paramètres et l’optimiser
* Une critique et visualisation des performances des modèles sur ce jeu de données
* Une sélection d’un modèle final à partir des performances

### Vérifiez votre travail

Alors, vous êtes allé au bout ? Suivez le guide pour vérifier votre travail !

Vérifiez que les éléments suivants sont présents :

* Une baseline de performances avec l’algorithme du k-NN est présente ;
* le SVM multiclasse  est utilisée avec différents paramètres et l’optimiser à l’aide d’une cross validation sur C, loss & penalty ;
* les visualisation et critique des performances sont constructives et permettent de choisir le modèle optimal sans hésiter.