# Modèles neuronaux pour le traitement des langues

Matthieu Labeau

LIMSI-CNRS

27 janvier, 2016

# Plan



Limsi

# Rappels d'Algèbre linéaire

# Scalaires, vecteurs, matrices et tenseurs

- Un scalaire est un simple nombre, naturel ou réel, usuellement noté en italique : $s \in \mathbb{R}$.
- Un vecteur est un tableau de scalaires indexés, noté :

$$\mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} \in \mathbb{R}^n$$

On peut identifier les vecteurs a des points dans l'espace, chaque élément étant une coordonée. La taille du vecteur est la dimension de cet espace (ici, $n$).

# Scalaires, vecteurs, matrices et tenseurs

- Une matrice est un tableau à deux axes, dont chaque élément est identifié par 2 indices, notée :

$$\mathbf{A} = \begin{bmatrix} a_{1,1} & \dots & a_{1,n} \\ \vdots & & \vdots \\ a_{n,1} & \dots & a_{n,n} \end{bmatrix} \in \mathbb{R}^{n \times n}$$

  On note la $i$-ème ligne de la matrice $\mathbf{A}_{i,:}$, et sa $j$-ème colonne $\mathbf{A}_{:,j}$. Transposer une matrice revient à inverser ses 2 axes :

$$\begin{bmatrix} a_{1,1} & a_{1,2} & a_{1,3} \\ a_{2,1} & a_{2,2} & a_{2,3} \end{bmatrix} \Rightarrow \begin{bmatrix} a_{1,1} & a_{2,1} \\ a_{1,2} & a_{2,2} \\ a_{1,3} & a_{2,3} \end{bmatrix}$$

- Dans certains cas, on aura besoin d'un tableau à plus de 2 axes : c'est un tenseur, que l'on notera aussi $\mathbf{A}$.

# Multiplication de matrices

- L'opération la plus important que l'on réalisera avec des matrices sera de les multiplier. Pour obtenir la matrice **C** en multipliant **A** et **B**, **A** doit avoir autant de colonnes que **B** a de lignes. Pour des matrices de tailles $n \times m$ et $m \times p$, on obtient **C** suivant la formule :

$$c_{i,j} = \sum_{k=1}^{m} a_{i,k} b_{k,j}$$

- Le produit scalaire entre deux vecteur **x** et **y** est un cas particulier du produit de matrices : on l'écrit $\mathbf{x}^T \mathbf{y}$
- Le produit de matrices élément par élément est noté $\mathbf{A} \odot \mathbf{B}$ et s'appelle produit de Hadamard.

# Norme

- Les normes sont des fonctions qui renvoient des valeurs non-négatives : elles sont des mesures de distance entre points.
- On utilisera principalement la norme $L^p$ :

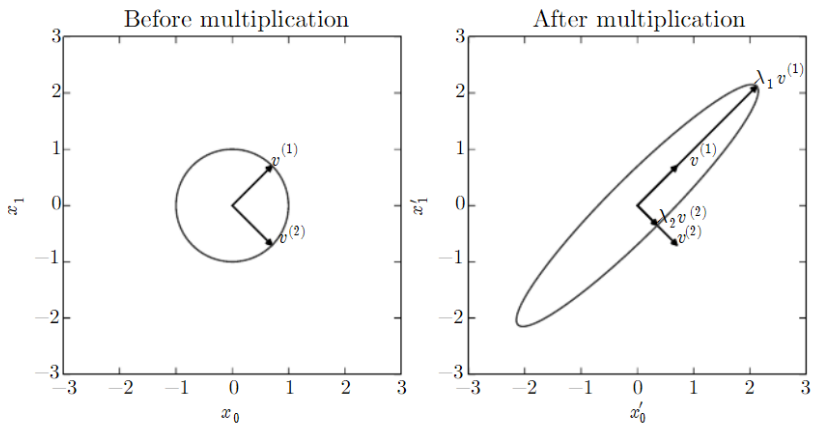
$$\|\mathbf{x}\|_p = \left( \sum_i |x_i|^p \right)^{\frac{1}{p}}$$

notamment la norme $L^2$ (euclidienne).

- Cependant, pour différencier des éléments proches de zéro, on utilisera la norme $L^1$ car la pente de la norme euclidienne décroit autour de zéro.

# Décomposition en valeurs propres

- On appelle les vecteurs propres et valeurs propres d'une matrices $\mathbf{A}$ les vecteurs $\mathbf{v}$ et scalaires $\lambda$ tels que $\mathbf{Av} = \lambda\mathbf{v}$.
- On peut décomposer certaines matrices en vecteurs propres en l'écrivant $\mathbf{A} = \mathbf{V}\Lambda\mathbf{V}^{-1}$.
- On l'interprête géométriquement en disant qu'une valeur propre déformera l'espace dans la direction de son vecteur propre associé :

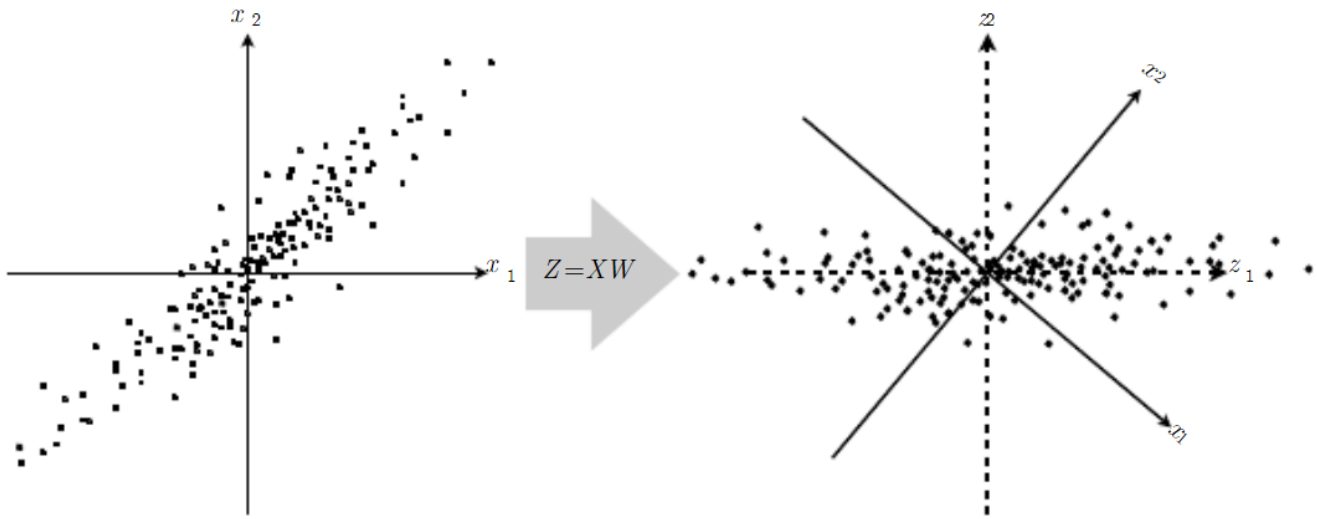


# Problème

- Supposons que l'on possède une liste de $m$ vecteurs $(\mathbf{x}^1, ..., \mathbf{x}^m)$, que l'on veut compresser. On tolère une perte d'information, mais on la voudrait aussi petite que possible.
- L'idée est de réduire la dimension de ces points : il nous faut trouver les dimensions qui contiennent le plus d'information sur l'ensemble des vecteurs.
- Il est possible d'écrire le problème comme une minimisation de l'erreur de reconstruction lorsqu'on cherche à decoder des données qui ont été encodées avec la contrainte d'une réduction de dimension.
- Il est aussi possible de le voir comme la recherche des dimensions ou les données ont le plus de covariance (ou elles sont le plus étendues) : ce sont les dimensions qui contiennent le plus d'information sur les données.

# Exemple

- Une fois que l'on a trouvé les deux principales directions pour nos données, on peut choisir de les représenter dans ce nouveau système de coordonnées.

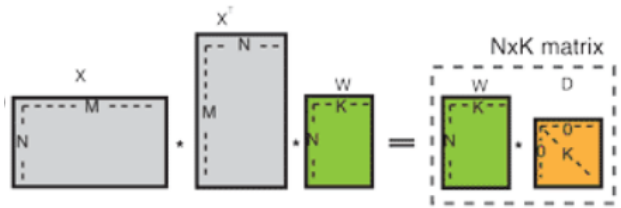


# Dérivation

- Il y a plusieurs méthodes pour obtenir les composantes principales d'une série de vecteurs. On peut notamment se servir de la décomposition en valeurs propres.
- On écrit la liste de vecteurs sous la forme d'une matrice $\mathbf{X}$, que l'on standardise (c'est-à-dire, soustraire la moyenne et diviser par la variance).
- On veut obtenir une matrice de taille $N \times M$. En appliquant la décomposition en valeurs propres à $\mathbf{X}\mathbf{X}^T$, selon l'équation :

$$\mathbf{X}\mathbf{X}^T = \mathbf{W}\mathbf{D}\mathbf{W}^T$$

on peut isoler le terme qui nous intéresse.



# Application

- On va s'intéresser plus tard à l'analyse de sentiment, dans notre cas la classification de critiques de films selon leur avis (positif ou négatif.)
- D'abord, on va chercher une manière de représenter chaque critique comme un vecteur de réels.
- Ensuite, on s'intéressera aux dimensions principales de l'ensemble des vecteurs représentant les critiques, en appliquant l'algorithme aux données.

# Apprentissage

- Lorsqu'on parle d'apprentissage dans le contexte de machines, on parle d'améliorer le résultat d'une expérience qui met en place une tâche particulière. Ce résultat est mesuré à l'aire d'une mesure de performance spécifique.
- Ces tâches, performances et expériences sont les trois composantes d'un algorithme d'apprentissage automatique.

# Tâches classiques

- Classification : On demande au programme de classer l'entrée dans une des $k$ catégories possible. L'algorithme doit donc en général apprendre une fonction $f : \mathbb{R}^n \rightarrow 1, ..., k$ qu'on peut ensuite appliquer à n'importe quelle entrée. Il est aussi possible d'apprendre une distribution de probabilité multinomiale sur les classes.
- Régression : On demande au programme d'associer une valeur numérique à l'entrée. L'algorithme doit donc apprendre une fonction $f : \mathbb{R}^n \rightarrow \mathbb{R}$
- Prédiction structurée : Catégorie très large, qui inclut les tâches ou la sortie demandée possède d'importantes dépendances entre ses éléments. Par exemple, le parsing - associer à une phrase un arbre qui décrit sa structure grammaticale.
- Traduction : Il s'agit d'associer à une séquence dans un langage une séquence dans un autre langage.

# Performance

- Le choix de la mesure de performance est souvent spécifique à la tâche associée : pour les tâches de classification, on utilisera la précision, c'est à dire la proportion d'examples pour lesquels le modèle appris à produit la bonne classe.
- Le choix de la performance peut paraître évident, mais il est souvent difficile à faire, car il est difficile de traduire en termes mesurables le comportement désiré d'un système.
- Lors d'une prédiction structurée, est ce qu'une erreur invalide toute la séquence ? Lors d'une régression, pénalise t'on différement les petites des grosses erreurs ?

# Expérience

- Selon l'expérience qu'ils utilisent, on décrira certains algorithmes comme *supervisés* ou *non-supervisés*.
- Chaque algorithme utilise un ensemble de données (dans notre cas, le plus souvent un corpus) décomposé en *exemples*. Il représentera les exemples à l'aide de *caractéristiques* (ou *features*).
- Lorsqu'il n'est pas évident, le choix de ces features est souvent l'enjeu principal associé à la tâche : quelle est la représentation des données qui conviendra le mieux au problème ?

# Supervisé vs Non-supervisé

- On décrira comme supervisés les algorithmes qui s'occupent de données ou, à chaque exemple, est associé une sortie désirée. Dans la classification, on cherche à obtenir la sortie désirée à l'aide des *features*. Si on note cette sortie $y$ et l'exemple associé $\mathbf{x}$, l'algorithme cherchera à apprendre $p(y|\mathbf{x})$.
- L'apprentissage non-supervisé essaie de déduire des informations qui peuvent nous sembler intéressantes à partir des *features*. Par exemple, regrouper des exemples en clusters, ou découvrir la distribution de probabilité qui les a générés. L'algorithme cherche ici à apprendre $p(\mathbf{x})$ ou certaines de ses propriétés.
- Les lignes entre apprentissage supervisé et non-supervisé ne sont pas claires. On a par exemple montré qu'il était équivalent d'écrire, pour un modèle de langue, qu'on cherchait à apprendre la probabilité d'une séquence $p(\mathbf{w})$ ou à choisir le mot qui suit en fonction d'un contexte $p(w_n|w_1, ..., w_{n-1})$.

# Problème

On commence avec le modèle le plus simple possible : un modèle linéaire. Il supposera que la sortie $y$ peut être exprimée comme fonction linéaire de l'entrée $\mathbf{x}$, c'est à dire comme fonction d'une somme pondérée :

$$y = f(\mathbf{x}^T \mathbf{w}) = f(\sum_i x_i w_i)$$

Le vecteur $\mathbf{w}$, pour weights, contiendra les poids, qui sont ici les paramètres du modèle. Ils contrôlent son comportement : si le poids associé à une *feature* est nul, celle-ci n'aura aucune influence sur la sortie.

# Classification

- On va chercher maintenant à faire de la classification binaire, c'est à dire à décider à laquelle de deux catégories l'entrée appartient.
- Formellement, notre modèle va être défini comme la fonction suivante :

$$f_w(\mathbf{x}) = \begin{cases} 1 & \text{si} \quad \mathbf{x}^T\mathbf{w} > 0 \\ 0 & \text{si} \quad \mathbf{x}^T\mathbf{w} < 0 \end{cases}$$

  et on cherchera à apprendre les meilleurs poids $\mathbf{w}$ possible.
- On travaille avec un modèle linéaire : si on se place avec des données en dimmension deux, cela revient à trouver la ligne qui séparera avec le moins d'erreurs possibles deux ensembles de points.

# Perceptron

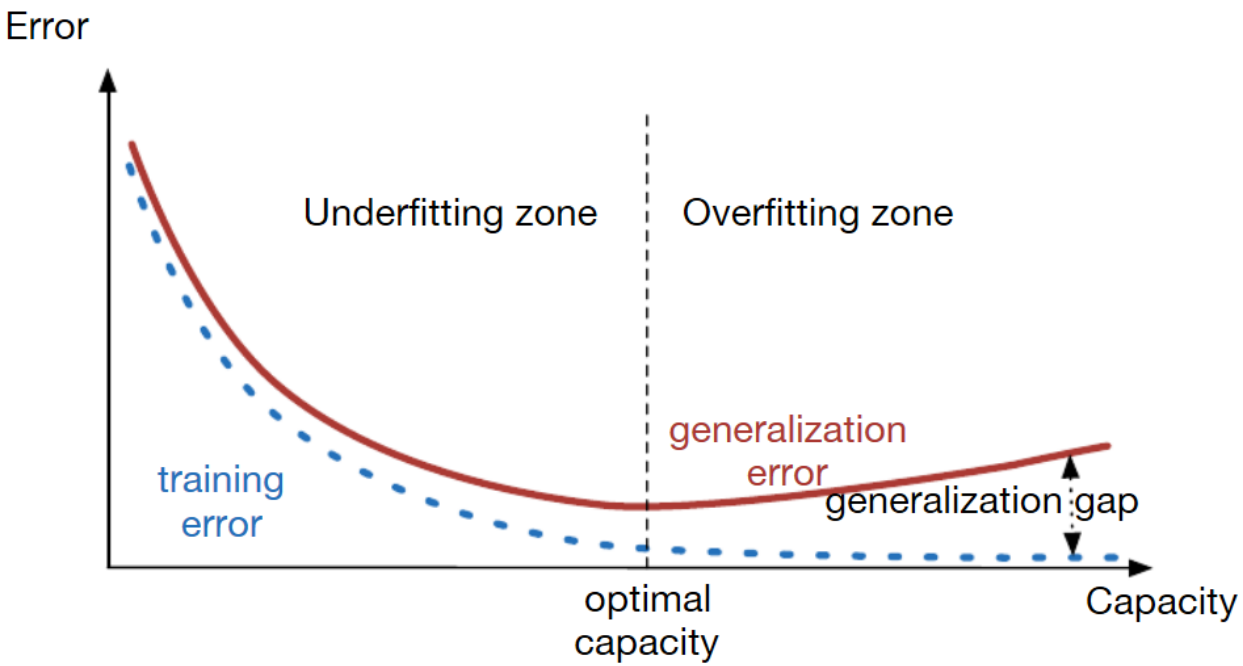
- On appelle ce modèle un perceptron.
- On peut simplement l'entrainer avec la procédure suivante :
  - Initialiser $\mathbf{w}$
  - Pour $I$ itérations :
  - Pour chaque exemple $(y, \mathbf{x})$ :
  - $y_{predicted} = f_{\mathbf{w}}(\mathbf{x})$
  - Si $y_{predicted} \neq y$ :
  - Mise à jour des poids : $\mathbf{w} \leftarrow \mathbf{w} + \frac{y\mathbf{x}}{i}$
- On peut interpréter la mise à jour comme le fait de déplacer la ligne de séparation de l'autre coté du point qui a posé problème.
- On va appliquer cette procédure à l'entraînement d'un modèle qui décidera si une critique est positive ou négative.

# Généralisation

- La difficultée principale, en apprentissage automatique, est d'obtenir de bonnes performances sur de nouvelles données, que le modèle n'a pas encore vues. Le but est donc d'être capable de *généraliser*.
- Jusqu'ici, on a parlé d'améliorer la performance sur les données avec lesquelles on entraînait le modèle, c'est à dire réduire l'erreur d'apprentissage. C'est de l'optimisation : ce qui fait la différence avec l'apprentissage, c'est que l'on cherche aussi à réduire l'erreur de généralisation.
- On testera donc notre modèle sur des données différentes, mais similaires : les données de test. Si elles partagent la même distribution sous-jacente, notre modèle devrait être capable de généraliser.

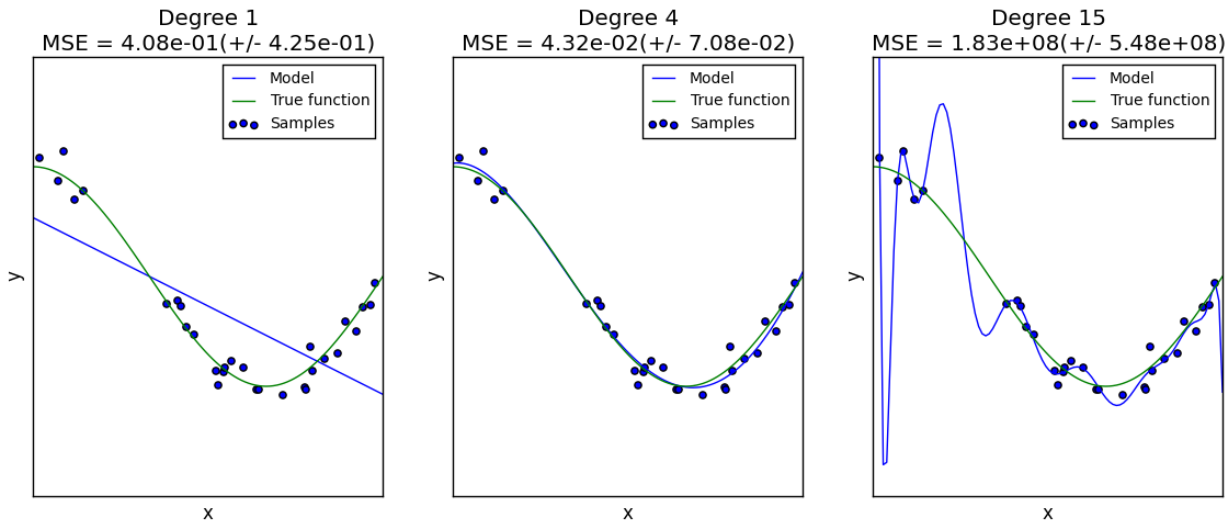
# Sur-apprentissage

- Si notre modèle n'apprend pas assez, on parlera d'*underfitting*.
- Si il apprend 'trop', c'est à dire qu'il correspond tellement bien aux données d'entraînement qu'il commet plus d'erreurs sur les données de test, on parlera d'*overfitting*.



# Capacité

- On peut contrôler la capacité de notre modèle à apprendre avec le nombre de features qu'il prendra en compte, et donc son nombre de paramètres.
- La capacité d'un modèle est notre premier levier pour éviter l'under- ou l'over-fitting.



# Régularisation

- Il est important, en apprentissage automatique, d'accorder la capacité du modèle à la complexité des données.
- On peut forcer notre modèle à préférer un certain type de fonction, en ajoutant une contrainte à ses paramètres. C'est la *régularisation*.
- La régularisation est une modification du calcul de performance, qui affectera l'erreur de généralisation, sans affecter l'erreur d'apprentissage.

# Validation

- Pour certains algorithmes, des paramètres sont extérieurs à la phase d'apprentissage, parce qu'ils sont trop difficiles à optimiser, ou à inclure dans l'expérience : ce sont les hyperparamètres.
- On gardera à l'écart une partie des données d'entraînement pour évaluer notre choix d'hyperparamètres : ce seront les données de validation.
- Si on manque de données, on pourra avoir recours à la cross-validation.

# Estimation du maximum de vraisemblance

- Pendant l'entraînement, on cherche à estimer les meilleures paramètres pour approximer une fonction des données $(\mathbf{x}^1, ..., \mathbf{x}^m)$.
- Notant les paramètres du modèle $\text{argmin}_\theta$, on veut obtenir les meilleures valeurs de $p_{model}(\mathbf{x}, \text{argmin}_\theta)$ possibles.
- L'estimateur du *maximum de vraisemblance* se définit alors comme celui qui nous donnera les paramètres :

$$\theta_{ML} = \text{argmax}_\theta \prod_{i=1}^{m} p_{model}(\mathbf{x}^i, \theta)$$

# Estimation du maximum de vraisemblance

- Ainsi, maximiser la vraisemblance revient à minimiser moins la log-vraisemblance. C'est la *fonction objectif* associée à notre tâche :

$$\theta_{ML} = \text{argmin}_\theta - \sum_{i=1}^{m} \log p_{model}(\mathbf{x}^i, \theta) = \text{argmin}_\theta C(\theta)$$

- Il est relativement facile de démontrer que cela revient à minimiser la divergence de Keiblack-Lubler entre le modèle est les données : on minimise la distance entre deux distributions de probabilités.
- Choisir cet objectif permet d'assurer certaines propriétés, comme celui d'être la meilleure estimation asymptotiquement (quand $m$ devient très grand).

# Gradient

- Même si il est parfois possible calculer ce minimum directement, en général, le moyen le plus simple de minimiser une fonction dépendant de nombreuses variables est de le faire de manière itérative avec une *descente de gradient*.
- On calcule le gradient de la fonction objectif par rapport aux paramètres, et on l'utilise pour les mettre à jour :

$$\theta^{k+1} = \theta^k - \lambda \frac{\partial C(\theta^k)}{\partial \theta^k}$$

- On appelle $\lambda$ le *taux d'apprentissage*, ou *learning rate*. C'est un hyperparamètre dans la plupart des algorithmes qui utilisent la descente de gradient, et il existe de nombreuses stratégies pour le choisir et le faire évoluer.

# Objectif

- Reprenons notre tâche-exemple : attribuer à un ensemble de critiques de films, leur classe (positive ou négative). On dispose des données suivantes : 1000 textes positifs, 1000 textes négatifs.
- Supposons que l'on veuille reprendre un classifieur linéaire, mais apprendre cette fois la probabilité pour chaque critique d'être positive ou négative.
- On utilisera pour cela un modèle log-linéaire : il s'agit d'utiliser un modèle linéaire pour donner un score à chaque exemple, puis de le transformer en probabilité à l'aide de la fonction logistique.

# Régression logistique

- On écrivait tout à l'heure, pour le perceptron :

- On décrit maintenant une approche similaire, mais probabiliste. Puisque la distribution est binomiale, pour un exemple $x$,

$$p(y|\mathbf{x}) = \begin{cases} h_\theta(\mathbf{x}) & \text{si} \quad y = 1 \\ 1 - h_\theta(\mathbf{x}) & \text{si} \quad y = 0 \end{cases}$$

- Ici, $h_\theta$ calcule un score associé à l'exemple et lui applique la fonction logistique pour obtenir une valeur entre 0 et 1 :

$$h_\theta(\mathbf{x}) = \frac{1}{1 + \exp(-\theta^T \mathbf{x})}$$

# Algorithme

- On peut réecrire la probabilité :

$$p(y|\mathbf{x}) = h_\theta(\mathbf{x})^y(1 - h_\theta(\mathbf{x}))^{1-y}$$

- On cherche à ce que la probabilité obtenue soit proche de la classe voulue : notre fonction objectif est donc la suivante :

$$C(\theta) = \sum_{i=1}^{m} \left[ y_i log h_\theta(\mathbf{x}^i) + (1 - y_i) log 1 - h_\theta(\mathbf{x}^i) \right]$$

- On calcule son gradient :

$$\frac{\partial C(\theta_j)}{\partial \theta_j} = \sum_{i=1}^{m} \left[ y_i x_j^i - h_\theta(\mathbf{x}^i) x_j^i \right]$$

# Expériences

- On peut maintenant appliquer notre algorithme, qui pour chaque exemple d'entraînement appliquera la descente de gradient aux paramètres.
- Il faut maintenant préparer les données : partager entre entraînement et test, choisir les *features*.
- On peut comparer les résultats au perceptron. Y-a t'il une amélioration si on choisit nos *features* à l'aide de l'analyse en composantes principales ?

# Références


- Deep Learning Book, Ian Goodfellow, Yoshua Bengio and Aaron Courville
- Introduction to Machine Learning with Scikit-Learn
- An introduction to PCA