

Classification supervisée de documents

1. Introduction

La classification automatique supervisée de document devient nécessaire à cause du volume de documents échangés et stockés sur support électronique. A la différence de la classification non supervisée où l'ordinateur doit découvrir lui-même des groupes de documents, la classification supervisée suppose qu'il existe déjà une classification de documents. C'est le cas par exemple d'une bibliothèque ou d'un moteur de recherche comme Yahoo !. Le but est alors de classer automatiquement un nouveau document.

Comme les documents sont nombreux ou que leur nombre augmente sans cesse, il serait difficile de programmer à l'avance des règles de décision pour déterminer la classe d'un nouveau document. Même si cela était possible, ces règles devraient être régulièrement modifiées par l'utilisateur pour qu'elles reflètent la réalité actuelle. Nous présentons donc des méthodes d'apprentissage qui, à partir de documents déjà classés, permettent de classer de nouveaux documents. Nous nous intéressons donc ici aux algorithmes d'apprentissage supervisés, c'est à dire où les réponses du programme sont fixées à l'avance (la hiérarchie de Yahoo ou la catalogue de la bibliothèque). De façon simple, le but de l'algorithme est de découvrir pourquoi chaque document d'exemple a été rangé dans telle ou telle classe, afin de prédire la classe de nouveaux documents à ranger dans le futur.

La plupart des algorithmes d'apprentissage supervisés tentent donc de trouver un **modèle** – une fonction mathématique - qui explique le lien entre des données d'entrée et les classes de sortie. Ces jeux d'exemples sont donc utilisés par l'algorithme. Dans le cas de la classification de documents, on fournit donc à la machine des exemples sous la forme (Document, Classe). Cette méthode de raisonnement est appelée **inductive** car on induit de la connaissance (le modèle) à partir des données d'entrée (les Documents) et des sorties (leurs Catégories). Grâce à ce modèle, on peut alors déduire les classes de nouvelles données : le modèle est utilisé pour prédire. Le modèle est bon s'il permet de bien prédire.

Une méthode ne cherche pas à calculer de modèle : c'est le cas du raisonnement à partir d'exemples (K plus proches voisins, Category-based Search).

Il existe de nombreuses méthodes d'apprentissage supervisé :

- K plus proches voisins (et ses variantes : Category-based Search et Cluster-based Search)
- arbres de décisions (<http://www.dbmsmag.com/9807m05.html>)
- Naïve Bayes (ou encore Simple Bayes)
- réseaux de neurones
- machines à support de vecteurs (ou SVM)
- Programmation génétique

Vous trouverez une très bonne documentation dans le livre de Tom Mitchell « Machine Learning » (ISBN 0071154671).

Nous décrivons succinctement ces différentes méthodes et détaillons les plus utilisées pour la classification supervisée de documents : K plus proches voisins, arbres de décision et Naïve Bayes. Parmi toutes les méthodes, Naïve Bayes l'emporte souvent dans les expériences menées jusqu'ici par les équipes de recherche. Pourtant, très récemment (1999), une nouvelle technique appelée Support Vector Machine (SVM) semble donner de meilleurs résultats.

1. K plus proches voisins et ses améliorations

Plus connus en anglais sous le nom K-nearest neighbor (K-NN) [Weiss and Kulikowski 1990], ou encore Memory Based Reasoning [Stanfill and Waltz 1986]

Cette méthode diffère des traditionnelles méthodes d'apprentissage car aucun modèle n'est induit à partir des exemples. Les données restent telles quelles : elles sont simplement stockées en mémoire.

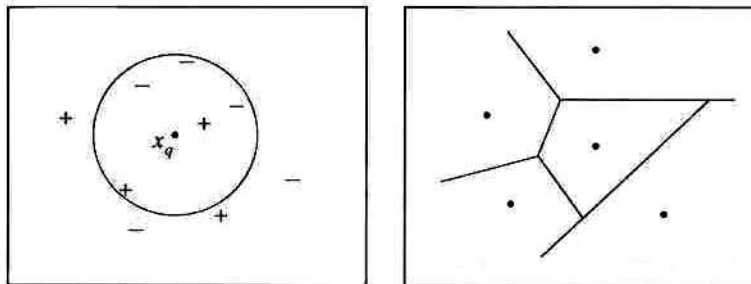
Pour prédire la classe d'un nouveau cas (où ranger un nouveau document ?), l'algorithme cherche les K plus proches voisins de ce nouveau cas et prédit (s'il faut choisir) la réponse la plus fréquente de ces K plus proches voisins. La méthode utilise donc deux paramètres : le nombre K et la fonction de similarité pour comparer le nouveau cas aux cas déjà classés.

$$d(x_i, x_j) \equiv \sqrt{\sum_{r=1}^n (a_r(x_i) - a_r(x_j))^2}$$

Ces valeurs sont arbitraires mais importantes car des résultats très différents résultent de leurs choix. Notez aussi que, si le temps d'apprentissage est inexistant puisque les données sont stockées telles quelles, la classification d'un nouveau cas est par contre coûteuse puisqu'il faut comparer ce cas à tous les exemples déjà classés.

Voici un exemple. On doit classer le nouveau document x_q . Si on choisit $K = 1$, x_q sera classé +. Si $K = 5$, le même x_q sera classé -. On voit donc que le choix de K est très important dans le résultat final !

Sur la figure de droite, on a représenté les exemples par des points. Chaque surface autour d'un exemple montre les positions possibles de nouveaux cas à classer où le résultat de la classification sera la classe de l'exemple si $K=1$. Cette figure est aussi connue sous le nom de **diagramme de Voronoi**.



Dans KNN de base, on choisit la classe majoritairement représentée par les K plus proches voisins. Une autre solution est de pondérer la contribution de chaque K plus proche voisin en fonction de sa distance avec le nouveau cas à classer.

Notez qu'avec cette méthode de pondération, on pourrait utiliser les N exemples au lieu des K plus proches voisins : en effet, plus un exemple sera éloigné du nouveau cas à classer et moins sa classe contribuera au résultat final ! Le seul désavantage est la perte de temps...

Les expériences menées avec les KNN montrent qu'ils résistent bien aux données bruitées. Par contre, ils requièrent de nombreux exemples.

Un problème rencontré avec le KNN de base est qu'il utilise tous les attributs d'un cas pour calculer la similarité avec un nouveau cas à classer. Prenons l'exemple d'un document qui contient 2000 mots. Contrairement aux arbres de décision où l'on ne teste qu'un attribut à

chaque noeud de l'arbre, la fonction de similarité des KNN utilise à chaque fois tous les attributs. Le problème principal n'est pas la perte de temps : imaginons que seulement 2 mots parmi les 2000 mots permettent de décider la classe d'un document (par exemple Web et HTML iront dans la catégorie « nouvelles technologie – internet »). La méthode kNN représente chaque cas par un point dans un espace à N dimensions (ici 2000 puisqu'on a 2000 mots différents). Les 2 mots discriminants vont être « noyés » dans les 2000 mots, c'est à dire qu'ils ne vont pas forcément se retrouver « proches » dans l'espace à 2000 dimensions.

Pour remédier à ce problème, on pondère l'importance de chacun des attributs. Mais comment choisir ces poids ? On utilise la méthode de validation croisée (cross-validation) : 2/3 des exemples sont utilisés pour trouver les poids, et les 1/3 restant valident les poids choisis.

Un inconvénient majeur de kNN reste le temps qu'il met pour classer un nouveau cas : il faut calculer la similarité entre K (ou même les N) exemples et le nouveau cas, puis décider quelle classe choisir (soit par majorité, soit en fonction de pondération selon la distance de chaque exemple avec le nouveau cas).

Pour remédier à cet inconvénient, une autre méthode appelée **Category-Based Search** [Iwayama 95] (ou encore simple nearest centroid, voir <http://www.qucis.queensu.ca/achallc97/papers/p025.html>) a été conçue. Elle consiste à représenter tous les documents rangés dans une catégorie par un cas unique (par exemple la moyenne des documents associés à une catégorie). Pour classer un nouveau document, on cherche le représentant le plus proche du document à classer. On peut alors modifier le représentant de cette catégorie si on veut prendre en compte ce nouveau document comme un nouvel exemple. On gagne en rapidité puisqu'on ne compare plus tous les documents 2 à 2 avec le nouveau document à classer, mais uniquement le nouveau document avec le représentant de chaque catégorie.

Une autre méthode, **Cluster-based search** [Salton and McGill 1983], améliore encore la version de base de KNN. Au lieu de représenter les documents d'une même catégorie, on utilise un algorithme de classification non supervisée (clustering) qui regroupe les documents par similarité. Quand on doit classer un nouveau document, on le compare aux représentants de chaque classe automatiquement découverte. L'intérêt par rapport à la méthode précédente (Category-based Search) est qu'un document peut être rangé dans plusieurs catégories, puisque les documents rangés automatiquement dans un cluster n'ont pas forcément la même catégorie au départ. On peut imaginer de donner le résultat sous forme des fréquences décroissantes des catégories.

On le voit, la méthode kNN a été très enrichie. Nous n'avons pas mentionné en détail les fonctions de similarité qui utilisent des formules de régression pour pondérer les exemples en fonction de leur distance avec le nouveau cas à classer. Vous trouverez encore une fois ces informations dans le livre de Tom Mitchell.

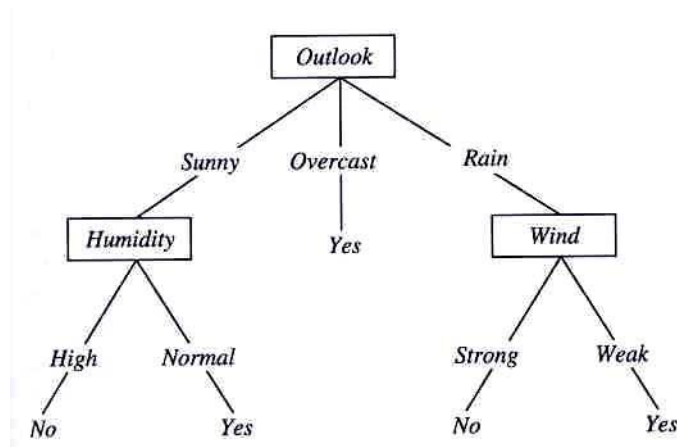
2. Arbres de décisions (<http://www.dbmsmag.com/9807m05.html>)

Les arbres de décision sont les plus populaires des méthodes d'apprentissage. Les algorithmes connus sont ID3 (Quinlan 1986) et C4.5 (Quinlan 1993). Ils sont également populaires pour la classification de documents.

Comme toute méthode d'apprentissage supervisée, les arbres de décision utilisent des exemples. Si l'on doit classer des documents dans des catégories, il faut construire un arbre de

décision par catégorie. Pour déterminer à quelle(s) catégorie(s) appartient un nouveau document, on utilise l'arbre de décision de chaque catégorie auquel on soumet le document à classer. Chaque arbre répond Oui ou Non (il prend une décision).

Concrètement, chaque nœud d'un arbre de décision contient un test (un IF...THEN) et les feuilles ont les valeurs Oui ou Non. Chaque test regarde la valeur d'un attribut de chaque exemple. En effet, on suppose qu'un exemple est un ensemble d'attributs/valeurs. Pour des documents, chaque attribut peut être un mot, et la valeur sera par exemple 0 ou 1 selon que ce mot appartient ou non au document.



Pour construire l'arbre de décision, il faut trouver quel attribut tester à chaque nœud. C'est un processus récursif. Pour déterminer quel attribut tester à chaque étape, on utilise un calcul statistique qui détermine dans quelle mesure cet attribut sépare bien les exemples Oui/Non. On crée alors un nœud contenant ce test, et on crée autant de descendants que de valeurs possibles pour ce test.

Exemple : si on teste la présence d'un mot, les valeurs possibles sont Présent/Abscent. A chaque fois, on aura donc deux descendants pour chaque nœud.

On répète ce processus en associant à chaque descendant le reste des exemples qui satisfont le test du prédécesseur.

ID3(*Examples*, *Target_attribute*, *Attributes*)

Examples are the training examples. *Target_attribute* is the attribute whose value is to be predicted by the tree. *Attributes* is a list of other attributes that may be tested by the learned decision tree. Returns a decision tree that correctly classifies the given *Examples*.

- Create a *Root* node for the tree
 - If all *Examples* are positive, Return the single-node tree *Root*, with label = +
 - If all *Examples* are negative, Return the single-node tree *Root*, with label = -
 - If *Attributes* is empty, Return the single-node tree *Root*, with label = most common value of *Target_attribute* in *Examples*
 - Otherwise Begin
 - $A \leftarrow$ the attribute from *Attributes* that best* classifies *Examples*
 - The decision attribute for *Root* $\leftarrow A$
 - For each possible value, v_i , of A ,
 - Add a new tree branch below *Root*, corresponding to the test $A = v_i$
 - Let $Examples_{v_i}$ be the subset of *Examples* that have value v_i for A
 - If $Examples_{v_i}$ is empty
 - Then below this new branch add a leaf node with label = most common value of *Target_attribute* in *Examples*
 - Else below this new branch add the subtree
ID3($Examples_{v_i}$, *Target_attribute*, $Attributes - \{A\}$)
 - End
 - Return *Root*
-

Il existe de nombreuses variantes pour construire des arbres de décision. ID3 utilise la mesure statistique appelée *Information Gain*.

Pour calculer cette valeur, on utilise un second calcul, celui de l'entropie de la classification. L'entropie est définie par :

$$\text{Entropie}(S) = -p/N \log_2 p/N - n/N \log_2 n/N$$

Où S est l'ensemble des exemples (samples), de taille N ,
 p (positive example) est le nombre d'exemples classés Oui,
et n (negative example) est le nombre d'exemples classés Non dans l'ensemble S des N exemples.

Attention : on définit $0 \log 0 = 0$

L'entropie permet de mesurer l'homogénéité des exemples. Si l'entropie vaut 0, alors tous les exemples appartiennent à la même classe (par exemple Oui). Si l'entropie vaut 1, alors c'est qu'il y a autant d'exemples positifs que d'exemples négatifs.

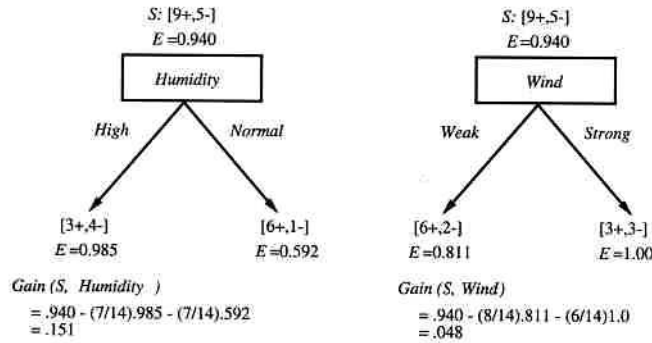
La mesure appelée Information Gain calcule la réduction attendue de l'entropie des exemples si un attribut particulier est utilisé. L'algorithme de classification calcule cette valeur pour chaque attribut et choisit alors celui qui réduit le plus l'entropie, c'est à dire celui qui permettra le plus nettement possible de séparer les exemples qui restent.

$$\text{Information Gain}(S,A) = \text{Entropie}(S) - \text{Somme sur les valeurs de } A \text{ de } (|S_v| * \text{Entropie}(S_v) / |S|)$$

Où S = l'ensemble des exemples,

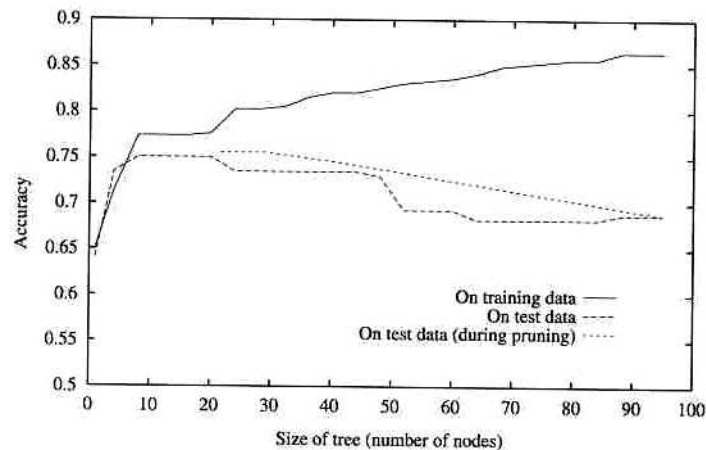
A est l'attribut utilisé,

S_v = le sous-ensemble de S dont l'attribut A a la valeur v



L'algorithme tel que nous l'avons décrit est celui utilisé par ID3. Dans cette version, un problème important apparaît : l'apprentissage par cœur (overfitting). L'arbre de décision classe trop bien les exemples, mais est mauvais pour généraliser, c'est-à-dire qu'il prédit mal la classification (Oui /Non) de nouvelles instances (documents).

L'apprentissage par cœur peut survenir lorsque les exemples sont bruités ou qu'il y a peu d'exemples. Dans ce dernier cas par exemple, il se peut qu'un arbre de décision soit construit avec des tests sur des attributs qui séparent bien les exemples, mais que ces mêmes tests ne soient pas bons pour classer de nouvelles instances. De façon générale, plus l'arbre de décision est profond, et plus le risque d'apprentissage par cœur augmente. Le problème est donc de trouver la profondeur idéale.



Pour estimer cette profondeur idéale, on divise l'ensemble des exemples en 2/3 pour les construire l'arbre de décision, et 1/3 pour valider l'arbre. La procédure de validation est la suivante : après avoir construit un arbre de décision complet, qui risque donc l'overfitting, on enlève successivement des nœuds de cet arbre. Ce nœud devient une feuille dont la valeur est la valeur prépondérante dans les feuilles situées jusqu'alors sous ce nœud. On valide alors ce nouvel arbre amputé d'un nœud (pruned tree) grâce aux exemples de validation (1/3) : si le nouvel arbre ne classe pas plus mal les exemples de validation que l'arbre précédent l'amputation, alors on le garde. On continue à amputer des nœuds de l'arbre de décision tant que le nouvel arbre continue à classer de mieux en mieux les exemples de validation.

Mais cette méthode a un inconvénient lorsque les exemples sont limités : on n'utilise en effet que les 2/3 des exemples de départ pour construire l'arbre, puisqu'on réserve 1/3 pour le valider ensuite. De nombreuses autres méthodes ont été proposées, notamment celle utilisée dans C4.5. On écrit l'arbre de décision sous forme de règles IF... THEN et on élimine progressivement des

conditions dans les IF. Ob n'élimine une condition que si elle ne décroît pas la capacité de l'arbre à classer les exemples.

D'autres extensions de l'algorithme ID3 permettent aussi d'utiliser des exemples où les valeurs des attributs sont continues. C'est le cas par exemple dans un document où on utilise la mesure TFIDF au lieu du simple test mot Présent/Absent. Pour traiter ces valeurs continues, ID3 les rend discrètes en séparant l'intervalle des valeurs continues en plusieurs intervalles, chacun définissant alors une valeur discrète.

D'autres versions d'ID3 utilisent des mesures différentes de l'information gain pour choisir quel attribut tester à chaque récursion. Voir pour cela **Machine Learning de Tom Mitchell, page 73**.

3. Naïve Bayes (ou Simple Bayes)

Nommés d'après le théorème de Bayes, ces méthodes sont qualifiées de "Naive" ou "Simple" car elles supposent l'indépendance des variables. L'idée est d'utiliser des conditions de probabilité observées dans les données. On calcule la probabilité de chaque classe parmi les exemples. Ce sont les "prior probabilities". Par exemple, si la classe "informatique" revient 2 fois sur les 5 documents donnés en exemple, sa "prior probability" sera de 2/5. En plus des "prior probas", l'algorithme calcule les fréquences d'apparition de chaque variable d'entrée avec celles de sortie. Pour classer des documents, les variables d'entrée sont les mots présents dans l'ensemble des documents. A chaque mot on calcule le nombre de fois qu'il apparaît dans les documents classés dans une classe donnée. On calcule cette fréquence pour chaque classe.

LEARN_NAIVE_BAYES_TEXT(*Examples*, *V*)

Examples is a set of text documents along with their target values. *V* is the set of all possible target values. This function learns the probability terms $P(w_k|v_j)$, describing the probability that a randomly drawn word from a document in class v_j will be the English word w_k . It also learns the class prior probabilities $P(v_j)$.

1. collect all words, punctuation, and other tokens that occur in *Examples*

- *Vocabulary* ← the set of all distinct words and other tokens occurring in any text document from *Examples*

2. calculate the required $P(v_j)$ and $P(w_k|v_j)$ probability terms

- For each target value v_j in *V* do
 - $docs_j$ ← the subset of documents from *Examples* for which the target value is v_j
 - $P(v_j) \leftarrow \frac{|docs_j|}{|Examples|}$
 - $Text_j$ ← a single document created by concatenating all members of $docs_j$
 - n ← total number of distinct word positions in $Text_j$
 - for each word w_k in *Vocabulary*
 - n_k ← number of times word w_k occurs in $Text_j$
 - $P(w_k|v_j) \leftarrow \frac{n_k+1}{n+|Vocabulary|}$

CLASSIFY_NAIVE_BAYES_TEXT(*Doc*)

Return the estimated target value for the document *Doc*. a_i denotes the word found in the i th position within *Doc*.

- *positions* ← all word positions in *Doc* that contain tokens found in *Vocabulary*
- Return v_{NB} , where

$$v_{NB} = \underset{v_j \in V}{\operatorname{argmax}} P(v_j) \prod_{i \in \text{positions}} P(a_i | v_j)$$

Une variante des Naive Bayes sont les réseaux Bayésiens : dans ce modèle, on ne suppose plus que les variables sont toutes indépendantes, et on autorise certaines à être liées. Cela alourdit considérablement les calculs et les résultats n'augmentent pas de façon significative.

4. Réseaux de neurones

Les réseaux de neurones sont utilisés pour leur capacité à apprendre à partir d'exemples bruités comme les caméras ou les micros (reconnaissance de forme ou de son). Mais ils sont aussi utilisables pour des problèmes où les méthodes symboliques (arbres de décisions) sont souvent utilisées. Leur performance est alors équivalente.

Les réseaux de neurone sont appropriés lorsque le temps d'apprentissage n'est pas essentiel : ce temps est en effet souvent très supérieur à d'autres méthodes comme les arbres de décision. Par contre, la classification d'un nouveau cas (par exemple un document) est très rapide.

Enfin, les réseaux de neurones sont appropriés si la compréhension de la fonction apprise par le réseau n'est pas essentielle. Avec un arbre de décision, l'opérateur humain peut toujours visualiser l'arbre et « comprendre » comment la machine décide. Avec un réseau de neurone, des techniques de visualisation existent, mais elles demandent généralement plus d'expertise que l'analyse d'un arbre de décision (qui peut être visualisé sous forme de règles).

Historiquement, la version la plus simple d'un réseau de neurone est le perceptron. Les perceptrons sont capables d'apprendre des fonctions linéairement séparables comme AND, OR, NAND, NOR. Le perceptron ne peut par contre pas apprendre le XOR. C'est d'ailleurs une critique adressée en 1969 par Minsky et Papert et les perceptrons ont été oubliés pendant quelques années...

Concrètement, un perceptron est donné en sortie un Oui/Non. En entrée, il prend un vecteur de dimension N (par exemple le vecteur d'un document), et en interne, il contient N poids (ou coefficients). En plus, on lui fournit un seuil. Simplement, le calcul est le suivant :

Si $w_0 + w_1 x_1 + w_2 x_2 + \dots + w_n x_n > 0$ **alors** le perceptron répond 1
Sinon le perceptron répond -1

L'apprentissage consiste à déterminer les poids w_i pour que les exemples soient bien classés. On le comprend, il faut un perceptron par catégorie puisqu'un perceptron n'est capable de répondre que Oui/Non. On commence avec des poids pris au hasard et on soumet les exemples en entrée du perceptron. On modifie à chaque fois les poids pour que le perceptron classe bien cet exemple. Il existe de nombreuses formules pour modifier les poids w_i , la première étant :

$w_i = w_i + n(t-o)x_i$

Où n est une constante positive qui est le taux d'apprentissage (0.1 puis diminue...),
 t est la réponse que le perceptron aurait dû donner,
 o est la réponse actuelle du perceptron.

Cette méthode de modification des poids fonctionne que si les exemples sont linéairement séparables. Une autre formule, connue sous le nom de Règle Delta (« Delta Rule » ou encore « Gradient Descent ») permet d'éliminer cette limitation.

La méthode suppose le calcul de l'erreur E du perceptron sur l'ensemble des exemples. A chaque itération, il faut modifier les poids w_i pour faire diminuer cette erreur. Pour calculer comment modifier les w_i , on calcule la dérivée de l'erreur E par rapport à chaque w_i (le **gradient**). Cette dérivée est facilement calculée par la formule suivante (non démontrée !) :

$$\Delta w_i = n * \text{Somme sur les exemple d (td-od)}x_{id}$$

Ensuite, comme pour la méthode précédente, chaque w_i est modifié avec la formule :

$$w_i = w_i + \Delta w_i$$

Il existe des variante de cette méthode pour améliorer la rapidité des calculs et s'assurer qu'on trouve bien un minimum dans l'erreur E (il peut y avoir plusieurs minimums locaux !). L'une d'elle s'appelle « incremental gradient descent », qui ne sera pas détaillée ici. Voir le livre « Machine Learning » de Tom Mitchell.

Après le perceptron et sa chute après la critique de Minsky et Papert, c'est l'invention de l'algorithme de BackPropagation en 1985 pour entraîner des réseaux de neurone à plusieurs couches qui a marqué la résurgence des réseaux de neurones. Ces réseaux de neurones sont « universels » : ils ne cherchent pas une fonction qui puisse modéliser l'ensemble des exemples donnés, mais plusieurs fonctions peuvent être apprises. Ils peuvent traiter naturellement des problèmes non séparables linéairement et sont rapides pour classer un nouveau cas. Le modèle construit est représenté par le réseau lui-même, et pour cette raison on associe souvent l'image de boîte noire à un réseau de neurones.

Pour apprendre des fonctions non séparables linéairement, il faut changer la fonction utilisée dans chaque neurone. Actuellement, on remplace la combinaison linéaire du perceptron par une fonction sigmoïde :

$$\text{Sigmoid} = 1 / (1 + e^{-\text{net}}) \text{ avec } \text{net} = w_1x_1 + w_2x_2 + \dots + w_nx_n$$

A la différence du perceptron, ce type d'unité ne répond pas Oui/Non mais une valeur entre 0 et 1.

En utilisant cette nouvelle unité comme base d'un neurone, on utilise alors des réseaux de neurone avec plusieurs neurones. Dans leur version la plus simple, ils sont constitués de deux niveaux : un niveau d'entrée et un de sortie. Pour la classification de documents, chaque neurone de sortie représente par exemple une catégorie, tandis que les neurones d'entrée sont encore une fois utilisés pour représenter le vecteur de chaque document.

L'algorithme le plus utilisé aujourd'hui pour « apprendre » les poids de tels réseaux est l'algorithme de « backpropagation ». Le calcul de l'erreur est similaire au précédent utilisé dans le perceptron, sauf qu'il somme tous les neurones. Voir Tom Mitchell « Machine Learning ».

5. Machines à support de vecteurs (ou SVM)

Cette technique - initiée par Vapnik - tente de séparer linéairement les exemples positifs des exemples négatifs dans l'ensemble des exemples. Chaque exemple doit être représenté par un vecteur de dimension n . La méthode cherche alors l'hyperplan qui sépare les exemples positifs des exemples négatifs, en garantissant que la marge entre le plus proche des positifs et des négatifs soit maximale. Intuitivement, cela garantit un bon niveau de généralisation car de nouveaux exemples pourront ne pas être trop similaires à ceux utilisés pour trouver l'hyperplan mais être tout de même situés franchement d'un côté ou l'autre de la frontière. L'efficacité des SVM est supérieure à celle de toutes les autres méthodes sur la classification de textes. Son efficacité est aussi très bonne pour la reconnaissance de formes. Un autre intérêt est la sélection de Vecteurs Supports qui représentent les vecteurs discriminant grâce auxquels est déterminé l'hyperplan. Les exemples utilisés lors de la recherche de l'hyperplan ne sont alors plus utiles et

seuls ces vecteurs supports sont utilisés pour classer un nouveau cas. Cela en fait une méthode très rapide.

6. Programmation génétique

C'est une méthode générale qui peut être utilisée après n'importe quelle méthode précédente, par exemple avec les arbres de décisions. En entrée, un algorithme génétique reçoit une population de classifieurs non optimaux. Le but du programme génétique est de produire un classifieur plus optimal que chacun de ceux de la population d'origine. D'une façon simple, cela consiste à extraire les meilleures parties de chaque classifieur d'origine et de les mettre ensemble pour produire un nouveau classifieur. Cela suppose de pouvoir comparer l'efficacité d'un classifieur. Un résultat important de la méthode est qu'après chaque itération on obtient un classifieur meilleur qu'avant. On peut donc arrêter les itérations à tout moment, même si le résultat n'est pas l'optimum.

7. Conclusion

Il existe donc une quantité importante de méthodes pour la classification de documents. Toutes sont issues des recherches sur l'apprentissage (« Machine Learning »). Comme pour la classification non supervisée, l'étape de représentation des documents est essentielle. On a vu que la plupart des méthodes nécessitent de représenter chaque document sous la forme d'un vecteur (type attribut/valeur).

Aussi, appliquées à la classification de documents, les méthodes peuvent se révéler très lentes puisqu'il est courant de traiter plusieurs centaines (voire milliers) de mots pour chaque document.

Contrairement à la classification non supervisée, la classification supervisée peut mesurer l'importance de chaque mot pour classer de nouveaux documents. Par exemple, une mesure (« information gain ») calcule la typicalité d'un terme. Plus un mot est lié à une catégorie et pas aux autres, et plus il est important : si un nouveau document le contient, ce mot sera très discriminant. De nombreuses mesures semblables ont été mises au point.

Enfin, à l'inverse de la classification non supervisée, il est ici simple d'évaluer les résultats d'une classification. Parmi les N exemples de documents classés, on utilise une partie des documents pour l'entraînement, et le reste pour le test. Pendant la phase de test, on soumet chaque document à l'algorithme de classification et on regarde simplement si la machine trouve la bonne classe. Bien sûr, le résultat de ce test n'est en rien garanti lorsque la machine aura à classer de nouveaux documents !