Projet de fin d'études: Rapport final

Étude de la popularité d'articles en fonction de leur titre à l'aide de techniques d'apprentissage machine

---

Présenté par:	Mathieu Roy (mathieu.roy.37@gmail.com)
Remis à:	Professeur Émile Knystautas
Sous la supervision de:	Professeur Brahim Chaib-draa
Dans le groupe de:	Dialogue, apprentissage et multiagents
Dans le cadre du cours:	GPH-3001
Dans le cadre du programme de:	Génie Physique
Dans l'établissement de:	L'Université Laval
Date du projet:	Du 13 janvier 2014 au 5 mai 2014
Date de remise du rapport final:	Lundi, le 5 mai 2014

Résumé

Le but de ce projet était de trouver quels caractéristiques d'un titre aide son article à devenir populaire, puis à utiliser ces connaissances afin de faire des prédictions sur la popularité d'idées de titres que nous avons.

Avec la méthode des k voisins les plus proches et en utilisant chaque mots de plus de 2 lettres comme caractéristiques, nous avons obtenu un gain d'information de 11.0%. Avec la méthode de classification naïve bayésienne, nous avons obtenue un gain d'information moyen de 12.0%, mais l'avantage de cette méthode est qu'elle peut donner le niveau de confiance qu'elle a dans la classification d'un certain titre: ainsi le gain d'information peut varier. L'arbre de décisions nous a permis de générer un arbre avec 4 caractéristiques, soit la longueur du titre, son type, le site où il est publié et s'il contient un nombre. En plus de bien pouvoir visualisé les données, celui-ci nous permet d'obtenir un gain d'information de 14%. La régression linéaire nous a permis d'obtenir le plus haut gain d'information, soit de 31.4%. Pour faire celle-ci, nous avons utilisé la longueur du titre et la popularité moyenne des mots le composant.

Les fonctions qui ont été créés peuvent être utiles pour un journaliste qui hésite entre plusieurs titres. Il pourra ainsi choisir celui qui a la plus haute probabilité d'être populaire. Mais il faut garder en tête les limites de cet outil: celui-ci ne peut pas générer des titres, mais seulement estimer si les idées des journalistes seront populaires.

Table des matières

Remerciement

Merci au professeur Brahim Chaib-draa qui a accepté de m’offrir de travailler avec lui pour faire un projet sur l’apprentissage automatique et de m’avoir donner les outils nécessaire à sa réalisation. J’ai confirmé une forte passion pour ce domaine et j’espère pouvoir continuer à travailler dans ce domaine à la maîtrise et sur le marché du travail.

Merci à la communauté de logiciel *open source* qui m’a permis d’avoir accès à Python et plusieurs de ses bibliothèques, telle que NumPy.

Introduction

L'apprentissage automatique constitue un des champs d'étude de l'intelligence artificielle. Comme son nom le dit, l'apprentissage automatique consiste à la création de logiciel qui peuvent apprendre à partir de données. Une définition formelle proposée par Tom Mitchell en 1998 est (traduit de l'anglais; source: Ng, Andrew):

> Un programme informatique est dit d'apprendre d'une expérience E par rapport à une certaine tâche T et une performance mesurée P, si sa performance sur T, tel que mesuré par P, s'améliore avec l'expérience E.

La tâche peut être d'apprendre à reconanître le visage de quelqu'un sur des photos, de mettre en format digital des documents écrits, de recommander des livres et des films en fonction des goûts de quelqu'un, etc. L'apprentissage automatique peut être utilisé dans une grande variété de domaine tant que nous disposons de suffisamment de données et qu'elles sont bien structurées.

Dans le cadre de ce projet, j'utiliserai certains outils de l'apprentissage automatique afin de déterminer qu'est-ce que les titres des articles de nouvelles qui vont virales ont en particulier. Le but est de déterminer qu'est-ce qui incite le plus de personnes à cliquer sur un lien et à partager ou aimer l'article à l'aide de réseaux sociaux. Ceci est souvent important pour les sites web de nouvelles car plus ils ont de trafic, plus il y a de personnes exposées à leur publicités, et donc plus haut sont leurs revenus.

Dans un premier temps, l'étendu du problème à étudier sera clairement défini. Puis, les donnés que nous espérons recueillir sera établies. Par la suite, je couvrirai la théorie de l'apprentissage automatique nécessaire afin d'atteindre notre objectif. Plus spécifiquement, nous couvrirons la méthode des k plus proches voisins, la classification naïve bayésienne, l'analyse fréquentielle, l'arbre de décision et la régression linéaire. Puis, il sera expliqué quels outils ont été d'utilisés et pourquoi. Après, la théorie sera appliquée à la problématique d'intérêt. Les parties importantes du code informatique sera étudiée en profondeur, mais le lecteur intéressé à consulter à l'intégralité du code utilisé se verra référé aux annexes. Finalement, la signifiance des résultats expérimentaux sera discutés.

Problématique

Le but est de trouver quels aspects d'un titre aide à rendre l'article virale. En premier lieu, nous trouverons une façon de quantifier la popularité des articles et nous expliquerons pourquoi ceci est important. Puis, nous spécifirons quelles donnés nous utiliserons. Finalement, nous élaborerons sur la tâche que l'algorithme doit effectuer.

Performance

Nous quantifierons le succès de ces articles à l'aide de leur interaction avec les réseaux sociaux les plus populaire en Amérique du Nord, soit: Facebook, Twitter, Google+, Diggs, Pinterest, LinkedIn, Delicious, StumbleUpon et Reddit (source: Alexa).

Nous définission la popularité d'un article comme étant la somme des quantités suivantes:

- les "j'aime" via Facebook
- les partages sur Facebook
- les commentaires via Facebook
- les tweets via Twitter
- les "+1" via Google+
- les partages sur Diggs
- les "épingles" sur Pinterest
- les partages sur LinkedIn
- les favoris sur Delicious
- les "j'aime" via StumbleUpon
- les partages sur Reddit

Le critère de loin le plus important pour les compagnies de nouvelles consistent à la quantité de personnes qui regardent leurs articles car plus il y en a, plus il y de personnes exposées à leur publicité ce qui consiste généralement à leur plus grande (voir la seule) source de revenu. Afin que beaucoup de personnes regardent leurs articles, il est important que leurs lecteurs interagissent avec les articles à l'aide des réseaux sociaux. Par exemple, le plus qu'il y a de personnes qui partagent l'article, le plus de personnes y seront exposés, ou le plus d'interactions il y a dans les commentaires, le plus les utilisateurs resteront sur le site web pour lire les commentaires. Ainsi, en mesurant la quantité d'interactions entre chaque article et les divers réseaux sociaux, nous obtenons une très bonne mesure du succès de chaque article.

Expérience

L'étude sera limité à des articles de nouvelles puisque différents types d'articles auraient surement des aspects différents qui les rends populaire. De plus, l'étude sera limité à des articles qui se retrouvent sur Internet puisque les charactéristiques de ceux-ci sont plus facilement quantifiables et puisque nous voulons évaluer est l'interaction des utilisateurs avec l'article à l'aide des réseaux sociaux. Nous limiterons les articles à un seul language puisque nous voulons savoir par exemple quels mots sont les plus populaires, donc évaluer plusieurs

languages en même temps viendraient grandement complexifier le problème puisque cela nécessiteraient des traductions afin de comparer les articles entre eux. De plus, cela n'apporterait probablement pas de résultats significativement différent à savoir ce qui rend un article populaire. Puisque 56,1% des sites web utilisent l'anglais et que seulement 3,9% des sites web utilisent le français (source: W3Techs), nous utiliserons des articles en anglais pour faire notre recherche. Ainsi nous pourrons plus facilement avoir beaucoup de donnés et les résultats seront utiles pour une plus grande partie de la population. Finalement, nous prendrons des articles recueillis de 4 sites web, soit Buzzfeed, ViralNova, Upworthy et Wimp, car ils sont très populaire, leurs données sont facilement accessibles et parce qu'ils fonctionnent avec tous les réseaux sociaux qui nous intéressent: il est même possible de commenter les articles avec un compte Facebook. En tous, nous évaluerons 2616 articles. Cela correspond à un nombre suffisament élevé pour pouvoir trouver des corrélatoins intéressantes entre les titres d'articles et leur popularité. De plus, ce nombre n'est pas trop grand ce qui nous permettra de travailler à une vitesse raisonnable à l'aide de la puissance de calcul à laquelle nous avons accès.

Tâche

Le problème consiste à trouver ce qui rend un article populaire. Nous avons défini ce que nous attendons par 'populaire' dans la sous-section "performance" et nous avons exposé la banque d'articles utilisée dans la sous-section "expérience". La tâche tant qu'à elle consiste à utiliser cette expérience pour apprendre à reconnaître les caractéristiques qui rendent les titres populaires. Certaines des caractéristiques qui seront étudiées sont sa longueur, les mots qui les composent, s'ils contiennent des nombres, le type de phrase utilisé, ainsi que le site web sur lequel ils sont publiés. Une fois l'apprentissage effectué, nous serons capable d'utiliser ces connaissances afin de faire des prédictions sur la popularité des titres. Cela nous permettrait d'améliorer la popularité de nos articles.

Nous avons utilisé 4 algorithmes différents pour analyser les titres. Bien qu'ils utilisent tous la même expérience et la même façon de mesurer la performance, leur tâches varient légèrement. La méthode des k plus petits voisins utilise la popularité de chaque mot pour prédire si le titre sera populaire. La classification naïve bayésienne utilise également la popularité de chaque mot, mais avec une approche plus probabilistique: il est donc possible de dire à quel point nous sommes confiant qu'un titre sera populaire. L'arbre des décisions tant qu'à lui sépare les titres en sous-branches dépendemment de leur longueur, leur type, s'il contiennent un nombre et où il sont publier et estime par la suite dans quelle catégorie il appartient. Finalement, la régression linéaire sépare mets les titres dans un graphique où l'axe des 'x' représente la longueur et l'axe 'y' représente la popularité des mots, puis nous traçons une droite qui essai du mieux possible de séparer les titres populaires des titres qui ne le sont pas.

Théorie

Étapes d'une application d'apprentissage automatique

Lorsque nous créons une application d'apprentissage automatique, les étapes générales sont normalement les mêmes. Elles seront donc introduites ici.

Premièrement, il faut recueillir les données. Celle-ci peuvent constituer à des données prises par des appareils de mesures, à des données recueillis sur Internet, etc. Puis, il faut organiser les données dans un format qui pourra être compris par un programme informatique. Par exemple, on peut les quantifier et les mettre dans des listes. Par la suite, il faut analyser les données afin de vérifier si elles ont bien été mesurées et téléchargées par le programme. On peut donc par exemple mettre les données brutes dans un graphique. Une fois cette étape passé, on peut maintenant entraîner l'algorithme. C'est à cette étape que l'apprentissage automatique s'effectue. Elle consiste à donner une série de donner au programme dont le résultat est connu. Par la suite, on test l'algorithme afin d'observer si le programme a bien appris. On donne à nouveau une série de donner au programme, mais cette fois-ci sans le résultat, et nous comparons ce qu'il prédit avec le résultat réel. Nous pouvons ainsi obtenir le taux d'erreur. S'il est trop élevé, nous pouvons retourner aux étapes précédentes afin d'essayer d'améliorer l'algorithme. Lorsque celui-ci est à notre goût, on peut finalement utiliser l'algorithme. C'est-à-dire que nous utiliserons l'algorithme sur des données que nous n'avons réellement pas le résultat afin de faire des prédictions sur celui-ci.

Le taux d'information gagnée par l'algorithme se calcul comme suit

$$information = (taux\ d'erreur\ initial - nouveau\ taux\ d'erreur) \div taux\ d'erreur\ initial$$

Le gain d'information maximal est lorsque le nouveau taux d'erreur est de 0, ainsi le gain est de 100% puisqu'il passe d'un choix purement aléatoire à une décision purement déterministe.

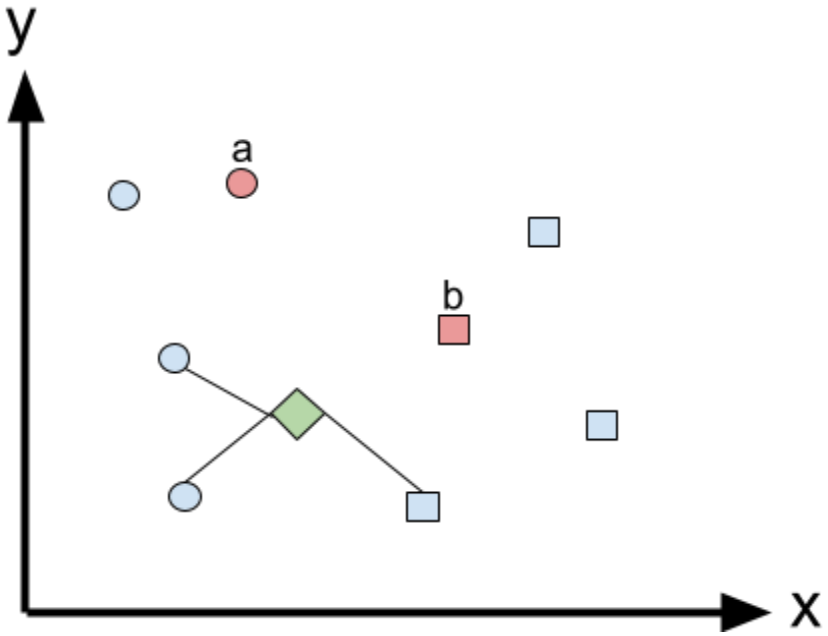
Méthode des k plus proches voisins

Le principal algorithme que nous utiliserons s'appelle la méthode des k plus proches voisins.

Cette méthode peut être utilisé pour quelconques nombre de caractéristiques, mais le nombre de catégories devraient être restreint.

Par exemple, dans un cas simple, nous pourrions avoir deux caractéristiques, soit 'x' et 'y', pouvant prendre des valeurs numériques réelles quelconques. De plus, les données pourraient appartenir à deux catégories, disons les cercles ◯ et les carrés ◰. Ces données peuvent être visualisé dans le graphique 1. On sépare les données en deux groupes: un qui servira à entraîner l'algorithme (qui corresponds à la couleur bleu sur le graphique) et un qui servira à tester l'algorithme (qui corresponds à la couleur rouge sur le graphique). Par la suite, nous pourrons utiliser l'algorithme sur des données ayant une catégorie inconnue (représentées par un losange ◇ vert sur le graphique).

Graphique 1 : Exemple simple de la méthode des k plus proches voisins



L'algorithme consite à prendre un point avec des caractéristiques connues, mais une catégorie inconnue, tel que représente le losange vert sur le graphique 1. Puis à mesurer sa distance 'd' de chaque autre point. Cela se fait simplement avec le théorème de Pythagore, soit

$$d = \sqrt{(y_2 - y_1)^2 + (x_2 - x_1)^2}$$

Si nous avons 'n' caractéristiques où 'n' plus grand que 2, nous devons utiliser le théorème généralisé de Pythagore afin de calculer la distance Euclédienne en n-dimension, soit

$$d = \sqrt{\sum_{i=1}^{n} (a_i - b_i)^2}$$

Par la suite, nous trouvons les 'k' voisins les plus proches de la valeur que nous cherchons à catégoriser, c'est-à-dire les 'k' voisins qui ont donné la plus petite distance 'd'. La plus part du temps, le nombre de voisins 'k' varient entre 3 et 7. Pour cet exemple, nous utiliserons un 'k' de 3. Sur le graphique 1, une ligne droite a été tracée entre la valeur à catégoriser et chacun de ses 3 plus proches voisins. Finalement, il s'agit de déterminer de quelles catégories sont la majorités des voisins proches. Cela correspondra à notre meilleure hypothèse tant qu'à la catégorie de la valeur inconnue. Dans notre exemple, il y a 2 cercles et 1 carré dans les voisins proches de la valeur inconnue. Ainsi, nous supposons que la valeur inconnue appartient à la catégorie des cercles.

Pour savoir comment bien notre algorithme performe, nous prenons une fraction 'f' des valeurs connues pour le tester. Normalement 'f' varie entre 5 et 25%. Pour cet exemple simple, nous utilisons 25% des donnés pour tester notre algorithme (rappel: ces valeurs correspondent aux données en rouge sur le graphique 1). Ainsi, nous prenons le point 'a', sans montrer à l'algoritme que c'est un cercle, et nous lui demandons d'estimer à quelle catégorie cette valeur

appartient. L'algorithme trouvera les 3 voisins les plus proches qui sont dans le groupe des valeurs connues (rappel: ces valeurs correspondent aux données en bleu sur le graphique 1). Ici, cela correspond à 2 cercle et 1 carré, l'algorithme supposera donc que la valeur appartient à la catégorie des cercles. L'algorithme vérifie par la suite avec la réponse. On fait de même pour le point 'b'. Dans cet exemple simple, l'algorithme obtiendrait une taux d'efficacité de 100%. En réalité, le taux d'efficacité sera généralement plus bas que 100%, mais plus haut que 100% divisé par le nombre de catégories possibles.

De plus, cela arrive parfois que certaines caractéristiques ne soient pas tous du même ordre de grandeur, mais qu'elles soit quand même tous aussi important à la détermination de la catégorie. Dans ces cas là, il est important de normaliser les caractéristiques.

Classification naïve bayésienne

La formule la plus fondamentale aux probabilités de Bayes est sans doute celle-ci

$$p(s|m) = p(m|s) * p(s) \div p(m)$$

Nous l'utiliserons afin d'estimer la probabilité que titre quelconque devienne populaire. En mots, celle-ci dit que la probabilité qu'un article atteigne la condition de succès 's', soit de devenir viral, sachant qu'il contient le mot 'm' dans son titre égale la fraction des titres d'article viral qui contiennent le mot 'm' fois la fraction d'article viral divisé par la fraction de d'article qui contiennent le mot 'm'. La fraction p(m|s) est calculée en prenant le nombre d'articles qui contient le mot 'm' dans son titre sur le nombre total d'article viral dans notre banque de donnés servait à l'entraînement de l'algorithme. La fraction p(s) est calculée en prenant le nombre d'articles viral divisé par le nombre total d'article dans cette même banque de données. Puis la fraction p(m) est calculée en prenant le nombre d'articles contenant le mot 'm' sur le nombre total d'article dans cette même banque de données. Ainsi, pour chaque mot qui est dans le titre, nous pourrons évaluer si les probabilités que l'article soit viral augment, diminue ou reste similaire.

Puisqu'il y a plusieurs mots dans le titre, l'équation sera en fait

$$p(s|m_1\&m_2\&\ldots\&m_i) = p(m_1\&m_2\&\ldots\&m_i|s) * p(s) \div p(m_1\&m_2\&\ldots\&m_i)$$

C'est-à-dire que nous calculons les probabilités pour l'ensemble des mots composant le titre. Nous considérerons qu'il est plus probable que l'article soit viral si

$$p(s|m_1\&m_2\&\ldots\&m_i) > p(\sim s|m_1\&m_2\&\ldots\&m_i)$$

et qu'il est plus probable de ne pas être viral autrement. Ici, le symbol ~ signifie "not", c'est-à-dire n'appartenent pas à la catégorie devant laquelle il se situe.

Afin d'entraîner notre algorithme, cela nécessite N^M opérations où N est le nombre de d'échantillons et M le nombre de caractéristiques. Lors de l'analyse de texte, le nombre de caractéristiques peut facilement dépasser le milles. À ce moment là, le nombre d'opérations nécessaires devient énormément trop grand pour toute application pratique. C'est là

qu'embarque la notion de "naïveté" bayesienne, car pour réduire le nombre d'opération, nous supposerons que les caractéristiques sont indépendantes, c'est-à-dire que

$$p(m_1 \& m_2 \& \ldots \& m_i) = p(m_1) * p(m_2) * \ldots * p(m_i)$$

Lors de l'analyse d'un texte par exemple, cela signifierait qu'un mot quelconque a autant de chance d'apparaître après un certain mot que tout autre. Nous savons que cela n'est pas vrai: par exemple, un verbe va rarement apparaître après un autre verbe. Toutefois, cela permet d'allégir énormément le nombre d'opérations nécessaires et permet quand même de faire des estimations sur les catégories.Cette simplification apporte deux problèmes que nous devons corriger.

Dès qu'un mot ne ce trouve pas dans une des catégories et donc que sa probabilité égal 0, alors la probabilité total sera aussi de zéro à cause de la multiplication, ce qui ne serait pas le cas si nous n'avions pas fait l'approximation d'indépendence. Pour remédier à ce problème, nous ajoutons 1 au numérateur de la fraction de chaque mot dans chaque catégorie et nous initialisons les dénominateur à 2, comme ça, cela donne une probabilité initiale de 0.5. Cela n'affectera pas les probabilités pour les mots fréquents, mais empêchera qu'il y ait des probabilités de 0. Ceci est désirable car même si un mot n'apparaît pas dans le titre, cela ne peut pas assurer qu'il ne deviendra pas viral: c'est peut-être juste trop rare pour qu'on ait un exemple dans notre banque de données.

L'autre problème est quand multipliant beaucoup de petites valeurs, cela peut mener à un soupassement arithmétique, c'est-à-dire que le programme va approximer la valeur à zéro tellement elle est petite. Pour remédier à ce problème, nous prenons le logarithme de chaque probabilité. Cela change la forme de la courbe, mais l'important est que la relation suivante soit respectée

$$log(p(x_1)) > log(p(x_2)) \; ssi \; p(x_1) > p(x_2)$$

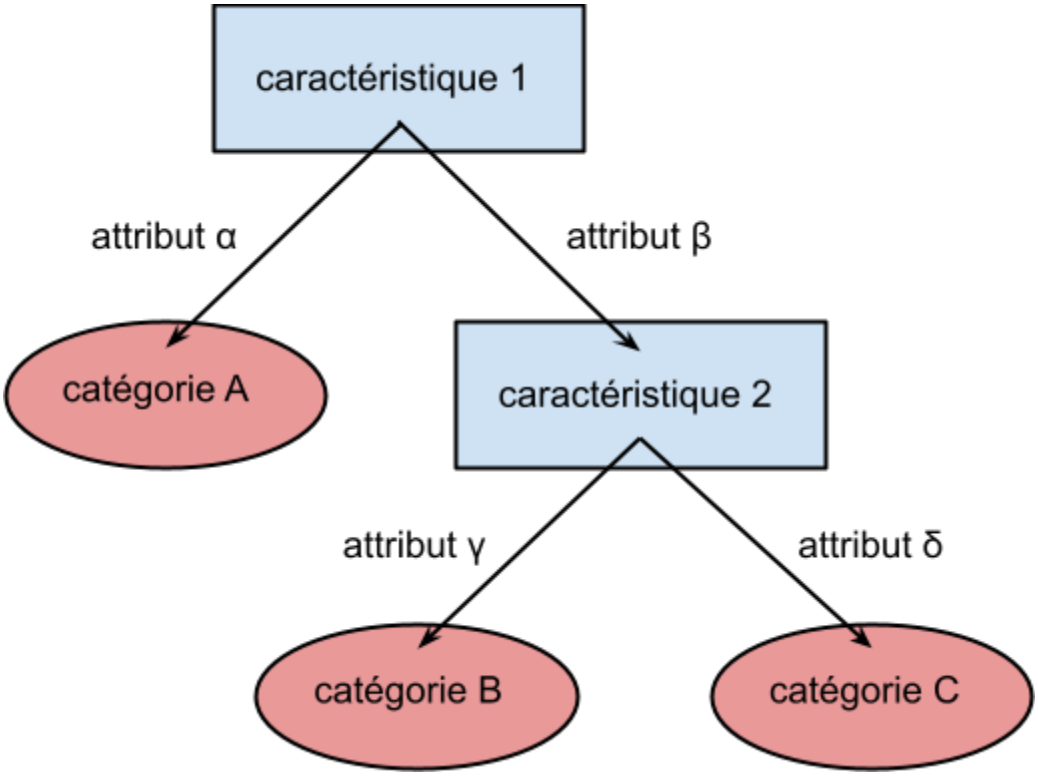
Ainsi, nous pouvons savoir qu'elle probabilité est la plus grande sans connaître cette valeur directement.

Arbre de décision

L'objectif principal d'un arbre de décision est de trouver la catégorie à laquelle appartient une variable avec le moins de réponses possibles pour tous les caractéristiques. Un exemple simple d'arbre est montré à la figure 1. Premièrement, on doit spécifier la caractéristique 1 de la variable, c'est-à-dire trouver si elle a l'attribut α ou β. Si la réponse est α, nous avons terminé et pouvons classifier la variable dans la catégorie A. Autrement, nous devons répondre à une deuxième caractéristique avant de pouvoir classifier la variable.

Un autre avantage de l'arbre de décision est qu'il nous permet de très bien visualiser les données.

Figure 1 : Exemple d'un arbre de décision



Pour construire un arbre de décision, il faut trouver sur quel caractéristique séparer l'arbre en premier. Pour y arriver, nous essayons avec chaque caractéristique, puis nous mesurons celui qui nous donnera les meilleurs résultats pour effectuer la séparation. Par meilleurs résultats, il est attendu que c'est la caractéristique qui organise le plus les données. Il faut donc quantifier ce gain d'information. Nous quantifions l'information que contient une variable xi comme

$$l(x_i) = - log_2(p(x_i))$$

où $p(x_i)$ est la probabilité de choisir cette catégorie. Pour calculer l'entropie de Shannon, ce qui correspond au niveau d'organisation de l'information, on doit calculer l'espérance mathématique de tous les catégories possibles, c'est-à-dire l'information obtenue en moyenne. Ceci est trouvé avec

$$H = - \sum_{i=1}^{n} p(x_i) * log_2(p(x_i))$$

où 'n' est le nombre de catégories.

Une fois que nous avons trouvé la caractéristique qui lorsque séparée permet le d'organiser le mieux les données, nous l'utilisons pour séparer les données. Puis nous refaisons de même pour chaque sous-section avec les caractéristiques restantes. Dès qu'une sous-section est composée en grande partie d'une catégorie spécifique ou lorsqu'il n'y a plus ce caractéristiques

à utiliser pour séparer les données, nous arrêtons de séparer les données de cette sous-section et les catégorisons dans la catégorie à laquelle la plus grande partie des données appartient. Cette méthode correspond à l'algorithme nommé ID3.

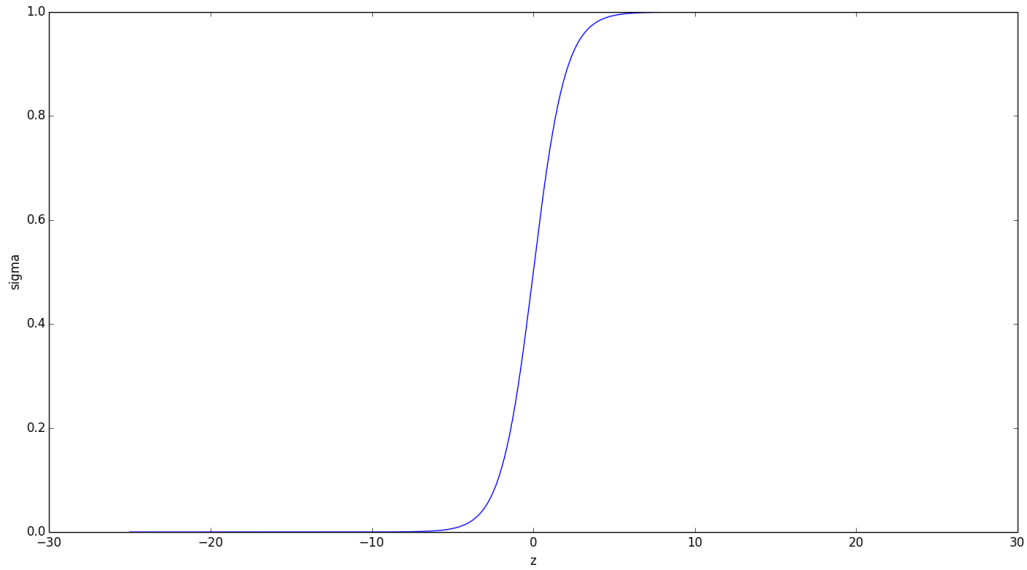
Régression linéaire

Dans le cas où nous avons 2 catégories, nous voulons trouver une fonction qui prend comme terme les diverses caractéristiques et donne 0 ou 1 comme résultat. Nous pourrions donc utiliser la fonction de Heaviside, mais puisque celle-ci est mathématiquement difficile à utiliser, nous utiliserons plutôt la fonction sigmoïde, soit

$$\sigma(z) = 1 \div (1 + e^{-z})$$

Le graphique 2 montre à quoi ressemble cette fonction.

Graphique 2: Fonction sigmoïde



Nous remarquons que pour des valeurs très petites, la fonction s'approche de 0, et pour des valeurs très grandes, la fonction s'approche de 1. Nous allons arrondir tous ce qui est plus grand que 0,5 à 1, et tous ce qui est plus petit que 0,5 à 0. En fait, plus la fonction nous donnera un résultat proche d'un extrême, plus il y a de chances que ce résultat soit bon.

L'entrée de la fonction sigmoïde sera

$$z = w^T x = \sum_{i=1}^{n} w_i * x_i$$

où 'x' est le vecteur des données et 'w' est le vecteur des coefficients. Nous devrons optimiser ce dernier.

Pour s'y faire nous utiliserons l'algorithme du gradient. Son principe consiste à se déplacer dans la direction du gradient, de la plus grande pente, afin de trouver le point maximum. Par exemple, si nous avons deux variables, nous aurions

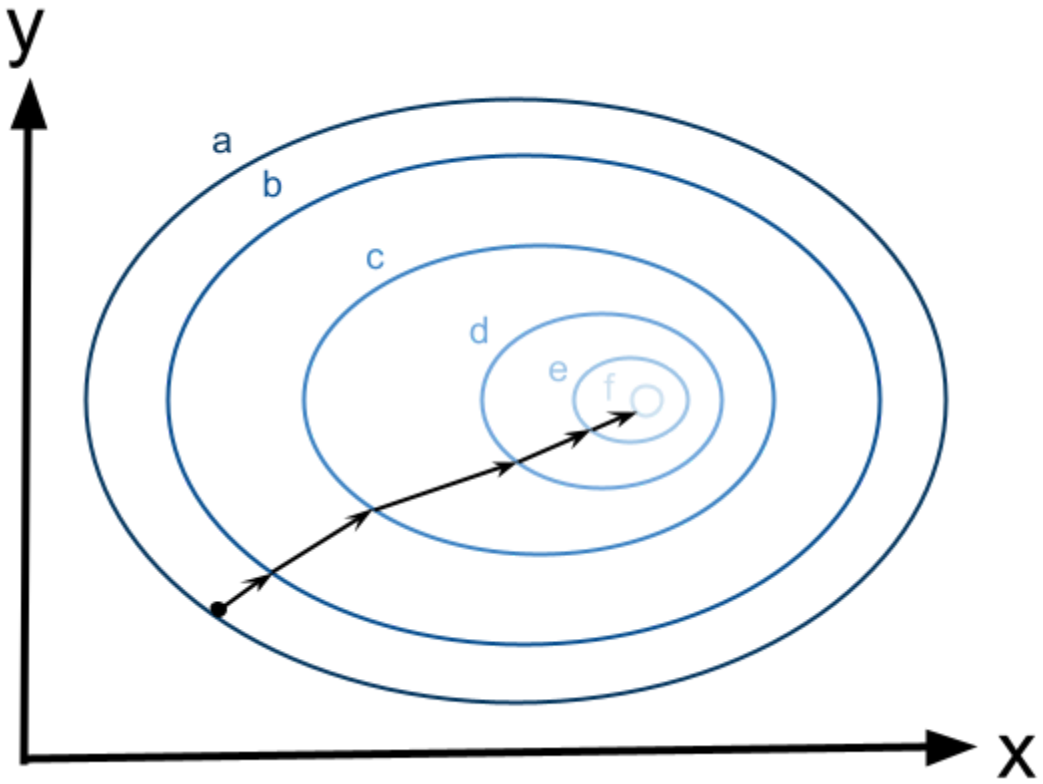
$$\nabla f(x,y) = (\delta(x,y)/\delta x, \delta(x,y)/\delta y)$$

À chaque itération, on s'approche de 'x' en fonction de $\delta(x,y)/\delta x$ et on s'approche de 'y' en fonction de $\delta(x,y)/\delta y$, soit

$$w := w + \alpha * \nabla_w f(w)$$

Le graphique 3 montre un exemple à deux variables où chaque flèche correspond à une itération s'approchant d'un maximum car $f > e > d > c > b > a$.

Graphique 3 : Algorithme du gradient



Cet algorithme est répété jusqu'à temps qu'une condition d'arrêt soit atteinte: soit a nombre préétablie d'étapes ou qu'une certaine marge de tolérance soit atteint. (source: Harrington)

Outils

Hardware

Notre banque de données consiste à 2616 titres d'une longueur moyenne de 62 caractères pris sur un des 4 sites webs étudiés et avec chacun 11 valeurs correspondant à divers réseaux sociaux. Parmis tous ces articles, nous retrouvons 7793 mots différent en tous.

Avec cette quantité d'information, nous pouvons facilement effectuer plusieurs boucles sur l'ensemble des donnés en moins de 10 minutes avec un simple ordinateur portable. Certains algoritmes auraient demandés plus de puissance, mais n'aurait pas significativement améliorer les résultats de l'expérience puisqu'il existe de très bonnes approximations à ceux-ci.

Le projet à donc été effectué sur un simple ordinateur portable avec un processeur Intel® Core™ i3-350M avec une fréquence de 2.27 gigahertz et une mémoire vive de 4,00 gigaoctets.

Software

Le logiciel que nous avons choisi est Python. Il y a plusieurs raisons qui ont mené à ce choix.

Python est un langage de haut niveau et utilise une syntaxe très claire. En effet il a acquis le surnom de pseudo-code exécutable. Ainsi, même si le lecteur n'est pas nécessairement familier avec ce logiciel, celui-ci devrait être en mesure de comprendre la signification du code. De plus, Python peut facilement manipuler le texte. Cela sera donc très utiles puisque nous travaillerons avec 2616 titres.

Python est populaire. Il y a donc beaucoup de matériel facilitant son apprentissage. Par exemple, le livre "Machine Learning in Action" par Peter Harrington n'utilise que ce logiciel. C'est d'ailleurs principalement ce livre que j'ai utilisé pour m'initier à l'apprentissage automatique. J'ai également utilisé le livre "The Python Tutorial" par Guido Van Rossum et Fred L. Drake Jr. afin de m'initier à ce langage de programmation. C'est deux livres ont été écrit pour la version 2.7.5 du logiciel. De plus, cette version est relativement récente et encore très utilisée. C'est donc celle-ci que j'ai utilisée. Sa popularité font également en sorte qu'il y a plusieurs bibliothèques de disponibles.

Par exemple, nous avons utilisé la bibliothèques scientifiques NumPy (source: NumPy) qui nous a permis de faire des opérations vectorielles et matricielles. Cela rend d'ailleurs le code encore plus facilement lisible. De plus, puisque cette bibliothèque est compilée en utilisant un langage de plus bas niveau, soit le C, ce qui rend les calculs avec cet outil plus rapide.

Un autre bibliothèque que nous utiliserons est Matplotlib (source: Matplotlib). Celle-ci permet de tracer des graphiques en 2D et 3D. Elle sera utilisé pour créer des arbres de décisions.

Nous avons également un peu utilisé chacune de ces bibliothèques (source: Python):

- pyparsing 2.0.2
- python dateutil 2.2
- pytz 2014.2
- six 1.6.1

Un autre avantage est que Python, ainsi que toutes les bibliothèques utilisées, sont *opensources* et gratuites.

L'édtieur de texte que nous avons utilisé pour écrire les programmes est Bloc-notes 6.1 (source: Microsoft).

Expérimentation

Préparation des données

Nous avons trouvé une série de donnés avec 2616 titres d'articles avec leur source et leur interaction avec les réseaux sociaux (source: Ripenn). Ce fichier était en format .xlsx. Après avoir fait quelques ajustement, nous l'avons enregistré en format .txt en séparant les colonnes par une tab afin de pouvoir le lire à partir de Python.

L'intégralité du code qui sert à préparer les données se retrouve en annexe A, mais nous allons expliquer les parties générales qui s'appliquent pour tous les algorithmes utilisés ici. Nous créons une 'liste' nommée "titles" avec tous les titres du document. Nous enlevons tous les signes de ponctuations et les mots de une ou deux lettres des titres et nous mettons tous les mots restant en lettres minuscules, car ce qui nous intéresse c'est à savoir si un mot apparaît dans un titre et non de comment il est formaté. Nous créons une liste avec la somme des "j'aime", partages, etc. pour chaque titre. Puis, nous créons une liste nommée "binarySuccess" qui dit pour chaque titre si celui-ci est populaire ou pas. Nous allons donc utiliser un critère minimum arbitraire pour considérer un article comme étant viral. La médiane du nombre d'interactions de chaque titres est de 2167, alors que la moyenne est de 19730 interactions. Il y a que 16,3% des données qui sont plus haut que la moyenne alors que (évidemment) 50,0% sont en haut de la médiane. Lors de nos calculs, nous allons considérer un article comme populaire s'il a reçu plus d'interactions que la médiane, et comme non populaire autrement.

Nous séparons la liste de titres en 2 parties: 90% des titres serviront à entraîner les algorithmes et 10% serviront à les tester. En effet, nous avons remarqué que c'est généralement le ratio idéal.

Code 0

```
from os import listdir
from numpy import *
import operator
from math import *
import re
import matplotlib.pyplot as plt
import pickle
```

Méthode des k plus proches voisins

Nous créons une liste nommée "vocabList" qui contient tous les mots différents contenus dans l'ensemble des titres. Puis nous modifions chaque titre en une série de chiffre. Par exemple, si nous avions

$$titles[0] = ["This\ is\ a\ title!\ "]$$

$$titles[1] = ["This\ is\ a\ real\ sentence."]$$

La liste de vocabulaire serait

$$vocabList = ['this', 'title', 'real', 'sentence']$$

Ainsi, les titres deviendraient

$$titlesVec[0] = [1, 1, 0, 0]$$
$$titlesVec[1] = [1, 0, 1, 1]$$

Le nombre représente le nombre de fois que le mot correspondant dans la liste de vocabulaire apparaît dans le titre. Cette liste de titres est nommée "titlesVec". Chaque mot de la liste de vocabulaire sera individuellement considéré comme une caractéristique. Nous normalisons chacune de ces caractéristiques afin qu'ils aient tous le même poids dans l'évaluation de la popularité d'un titre: nous ne voulons pas que les mots souvent utilisés aient un poids plus grand car les mots communs ne sont pas nécessairement ceux qui différencie un titre populaire d'un titre non populaire. La fonction du code 1 effectue cette tâche.

Code 1 : Normalisation des vecteurs des titres

```
def autoNorm(dataSet):
    minVals = dataSet.min(0)
    maxVals = dataSet.max(0)
    ranges = maxVals - minVals
    normDataSet = zeros(shape(dataSet))
    m = dataSet.shape[0]
    normDataSet = dataSet - tile(minVals, (m,1))
    normDataSet = normDataSet/tile(ranges, (m,1))
    return normDataSet, ranges, minVals
```

Nous utilisons 10% des données pour tester l'algorithme. Tel que vu dans la section "Théorie", nous calculons la distance euclidienne entre chacun de ces titres et les 90% autres titres. Puis pour chaque titre, nous prenons les 'k' titres les plus proches et calculons de quelle catégorie est la majorité de ceux-ci. Le résultat correspond à notre prédiction pour la catégorie du titre. Nous comparons ce résultat avec le résultat réel. Ainsi nous pouvons calculer le taux d'erreur. La fonction principale pour cette effectuer cette tâche se retrouve au code 2. Le reste du code se retrouve à l'annexe B.

Code 2 : Méthode des k plus proches voisins

```
def testKNNOnce(k,s):
    titles,charCount,numbers,question,binarySuccess,website = allData()
    ternarySuccessV = ternarySuccess()
    if s==2: success=binarySuccess
    elif s==3: success=ternarySuccessV
    else: print "not supported yet"; return
    vocabList = createVocabList(titles)
    titlesVec = titles2Vec(vocabList,titles)
```

```
    trainTitlesVec, trainClasses, testTitlesVec, testClasses = trainTest(titles,success)
    trainTitlesVec, ranges, minVals = autoNorm(array(trainTitlesVec)) #
    testTitlesVec, ranges, minVals = autoNorm(array(testTitlesVec)) #
    errorCount = 0.0
    numTestVec = len(testTitlesVec)
    for i in range(numTestVec):
        classifierResult = classify0(testTitlesVec[i][:],trainTitlesVec,trainClasses,k)
        if (classifierResult != success[i]): errorCount += 1.0
    return errorCount, numTestVec
#    print "the total error rate is: %f" % (errorCount/float(numTestVec[-1]))
```

Puisque le groupe de titres servant au testage est choisi de façon aléatoire, le taux d'erreur varie légèrement à chaque fois que nous utilisons l'algorithme. Nous exécutons donc l'algorithme 10 fois et prenons la moyenne. Le taux d'erreur obtenu est de 0.445. Sans cette information, en choisissant la catégorie au hasard, nous aurions eu un taux d'erreur de 0.500. Nous avons donc obtenu un gain d'information de $(0.500 - 0.445)/0.500 = 11\%$.

Puisque l'algorithme nous donne un bon gain d'information, nous pouvons l'utiliser pour estimer si nos titres seront populaires. La fonction *useKNN* qui sert à ça se retrouve à l'annexe A. Elle mesure la distance entre le vecteur du titre que nous voulons utiliser et les vecteurs de tous les titres dans notre banque de données. Puis elle retourne la catégorie que la majorité des 3 points voisins sont.

Classification naïve bayésienne

En premier lieu, nous entraînons l'algorithme pour déterminer les probabilités que chaque mots apparaissent dans la catégorie "populaire" et dans la catégorie "pas populaire". La fonction pour effectuer cette tâche se trouve au code 3.

Code 3 : Classification naïve bayésienne - Entraînement

```
def trainBayes2(trainTitlesVec,trainClasses):
    numTrainTitles = len(trainTitlesVec)
    numWords = len(trainTitlesVec[0])
    pBinarySuccess = sum(trainClasses)/float(numTrainTitles)
    p0Num = ones(numWords); p1Num = ones(numWords)      #a ete change en 1
    p0Denom = 2.0; p1Denom = 2.0                        #a ete change a 2.0
    for i in range(numTrainTitles):
        if trainClasses[i] == 1:
            p1Num += trainTitlesVec[i]
            p1Denom += sum(trainTitlesVec[i])
        else:
            p0Num += trainTitlesVec[i]
            p0Denom += sum(trainTitlesVec[i])
    p1Vec = [log(this/p1Denom) for this in p1Num] #a ete change en log
    p0Vec = [log(this/p0Denom) for this in p0Num] #a ete change en log
    return p0Vec,p1Vec,pBinarySuccess
```

Par la suite nous testons l'algorithme. C'est à dire que nous prenons chaque titre dans la liste de tests et nous estimons sa catégorie avec l'algorithme. Puis nous comparons la réponse de l'algorithme avec la vraie réponse. Cela nous permet de trouver le taux d'erreur de l'algorithme. La fonction de base pour effectuer cette tâche se trouve au code 4.

Code 4 : Classification naïve bayésienne - Test

```
def testBayes2Once(titles,binarySuccess):
    trainTitlesVec, trainClasses, testTitlesVec, testClasses = trainTest(titles,binarySuccess)
    p0Vec,p1Vec,pBinarySuccess = trainBayes2(array(trainTitlesVec),array(trainClasses))
    errorCount = 0
    for index in range(len(testTitlesVec)):
        if classifyBayes2(array(testTitlesVec[index]),p0Vec,p1Vec,pBinarySuccess) !=\
            testClasses[index]:
            errorCount += 1
    errorRate = float(errorCount)/len(testTitlesVec)
    return errorRate, pBinarySuccess
```

Puisque le 10% des titres à tester est choisi au hasard, le taux d'erreur varie légèrement à chaque fois que nous testons l'algorithme. Nous créons donc une fonction qui exécutera l'algorithme N fois et retournera le taux d'erreur moyen.

En testant l'algorithme 20 fois, nous obtenons un taux d'erreur moyen de 0,440 alors que sans l'algorithme il aurait été de 0,500. Puisque le taux d'erreur est 6% plus bas, cela veut dire que nous avons 6%/0,5 = 12% plus d'information avec l'algorithme que sans. L'algorithme nous permet donc d'être 6,0% fois plus certain de si notre article sera virale ou pas où virale a été défini comme étant meilleur que la médiane.

Nous avons également fait une deuxième version de l'algorithme où nous enlevions les mots vides (voir annexe C). Un mot vide est un mot sans signification particulière mais servant plutôt à relié grammaticalement les autres mots, appelés mots pleins. Par exemple, les mots français "les" et "il", ou les mots anglais "the" et "it", sont des mots vides. Puisqu'ils sont souvent là plus pour la structure du texte que pour leur signification, nous croyons qu'il n'aide peut-être pas à savoir si un titre est (ou sera) populaire et donc qu'ils ne valent peut-être pas la peine de les indexer. Nous avons donc créé une liste de 706 mots vides afin que l'algorithme puisse déterminé ceux qui le sont. Avec cette version de l'algorithme, en la testant 20 fois, nous obtenons un taux d'erreur moyen de 0,455 alors que sans l'algorithme il aurait été de 0,501. Cet algorithme nous permet donc d'avoir un gain d'information de 9,2%, soit 2,9% de moins que l'algorithme précédent. Ainsi, nous avons perdu de la précision en enlevant les mots vides. Notre hypothèse était donc fausse. Bien qu'il est vrai que les mots vides aient un moins grand impact sur la popularité d'un article, ceux-ci y jouent tout de même un rôle.

L'avantage principal de la classification naïve bayésienne par rapport à la méthode des k voisins les plus proches est son aspect probabilistique. Toutefois, nous n'avons pas encore utilisé celui-ci. Nous avons donc créé un algorithme (qui se trouve à l'annexe B et dont la fonction principale se retrouve également au code 5) qui donne le taux d'erreur en fonction d'un certain niveau de confiance.

Code 5 : Classification naïve bayésienne - Test version 3

```
def testBayes2v3Once(titles,binarySuccess,certainty):
    trainTitlesVec, trainClasses, testTitlesVec, testClasses = \
        trainTest(titles,binarySuccess)
    p0Vec,p1Vec,pBinarySuccess = trainBayes2(array(trainTitlesVec),array(trainClasses))
    errorCountSure = 0; countSure = 0
    errorCountUnsure = 0; countUnsure = 0
    for index in range(len(testTitlesVec)):
        p0,p1,experiment = \
            classifyBayes3(array(testTitlesVec[index]),p0Vec,p1Vec,pBinarySuccess)
        theory = testClasses[index]
        if experiment != theory:
            if experiment == 1:
               if float(p0)/float(p1) > float(certainty): errorCountSure += 1
               else: errorCountUnsure += 1
            elif experiment == 0:
                if float(p1)/float(p0) > float(certainty): errorCountSure += 1
                else: errorCountUnsure +=1
        if experiment == 1:
            if float(p0)/float(p1) > float(certainty): countSure += 1
            else: countUnsure +=1
        elif experiment == 0:
            if float(p1)/float(p0) > float(certainty): countSure += 1
            else: countUnsure +=1
    errorRateSure = float(errorCountSure)/float(countSure)
    errorRateUnsure = float(errorCountUnsure)/float(countUnsure)
    return errorRateSure,errorRateUnsure,pBinarySuccess
```

Le "niveau de certitude" ne dépasse pas le 1.2. Puisque nous avons utilisé des logarithmes dans nos calculs, cette valeur n'a pas vraiment d'unités, mais ce qu'il est important de savoir est que plus il est loin de 1, plus l'algorithme est confiant de sa réponse. Par exemple, nous avons séparé le taux d'erreur lorsqu'il avait un niveau de certitude de plus de 1.05 ou de moins de 1.05. Lorsque l'algorithme était confiant, son taux d'erreur était de 0.375, ce qui représente un gain d'information de 24.8% car sans l'algorithme, son taux d'erreur était de 0.499. Puis, lorsque l'algorithme n'était pas confiant, son taux d'erreur était de 0.474, ce qui représente un gain d'information de 5.0%. Nous voyons donc que son degré de certitude peut varier beaucoup.

Maintenant que nous avons entraîner et tester notre algoritme, nous sommes prêt à l'utiliser. Nous avons donc créé une fonction (voir code 6) qui nous permet d'entrer un titre et celle-ci nous dira si le titre a plus de chances de rendre l'article viral ou pas.

Code 6 : Classification naïve bayésienne - Utilisation

```
def useBayes2():
    titles,charCount,question,binarySuccess,website = allData()
    vocabList = createVocabList(titles)
    titlesVec = titles2Vec(vocabList, titles)
    p0Vec,p1Vec,pBinarySuccess = trainBayes2(array(titlesVec),array(binarySuccess))
    title = str(raw_input("What's the idea for your title?"))
    titleFormated = title.split()
    for index in range(len(titleFormated)):
        titleFormated[index] = titleFormated[index]\
            .strip('-').strip('.').strip(',').strip('?').strip('!').strip("'")
    titleFormated = [low.lower() for low in titleFormated if len(low) > 1]
    titleVec = array(title2Vec(vocabList,titleFormated))
    classResult = classifyBayes2(titleVec,p0Vec,p1Vec,pBinarySuccess)
    if classResult == 0:
        print 'the title of the article will probably not be popular'
    else: print 'the title of the article will probably be popular'
```

Nous pouvons également utiliser les variables *p0* et *p1* pour retourner la certitude avec laquelle nous sommes de la catégorie choisie grâce à la version 3 de l'algorithme servant à l'entraînement.

Arbre de décision

Dans les deux sous-section précédentes, nous avons évalué l'importance des mots du titre dans la popularité d'un article. Nous allons maintenant étudier d'autres aspects du titre qui influence sa popularité.

Une des caractéristiques sera le type de phrase; c'est-à-dire si le titre est de type interrogatif, exclamatif ou déclaratif. En fait, nous observerons simplement si le titre contient un point d'interrogation '?', un point d'exclamation '!' ou ni un ni l'autre.

Une autre caractéristique sera de savoir si le titre contient un ou des nombres sous forme numérique ou écrite en toute lettre. Nous avons donc créé un fichier .txt avec 108 constituant possible d'un nombre en anglais. Ainsi, pour savoir si un titre contient un nombre, nous regardons s'il a un des mots dans ce fichier.

Une autre caractéristique consistera au site web sur lequel l'article est, soit Buzzfeed, Wimp, Upworthy ou ViralNova. En effet, si l'article est sur un site web plus populaire, il a plus de chances d'être lui-même populaire.

Une dernière caractéristique sera la longueur du titre calculée en nombre de caractères incluant les espaces. Puisqu'un arbre de décision ne fonctionne qu'avec des valeurs nominales et que la longueur d'un titre est une valeur continue, nous quantifions cette valeur. Les titres populaires ont une longueur moyenne de 59 caractères et les titres non populaire une une longueur moyenne de 64 caractères.

La fonction principale pour créer l'arbre de décisions est au code 7 (source: Harrington).

Code 7 : Arbre de décisions - Entraînement

```
def createTree(dataSet,labels):
    classList = [example[-1] for example in dataSet]
    if classList.count(classList[0]) == len(classList):
        return classList[0]
    if len(dataSet[0]) == 1:
        return majorityCount(classList)
    bestFeat = chooseBestFeatureToSplit(dataSet)
    bestFeatLabel = labels[bestFeat]
    myTree = {bestFeatLabel:{}}
    del(labels[bestFeat])
    featValues = [example[bestFeat] for example in dataSet]
    uniqueVals = set(featValues)
    for value in uniqueVals:
        subLabels = labels[:]
        myTree[bestFeatLabel][value] = createTree(splitDataSet\
            (dataSet, bestFeat, value),subLabels)
    return myTree
```

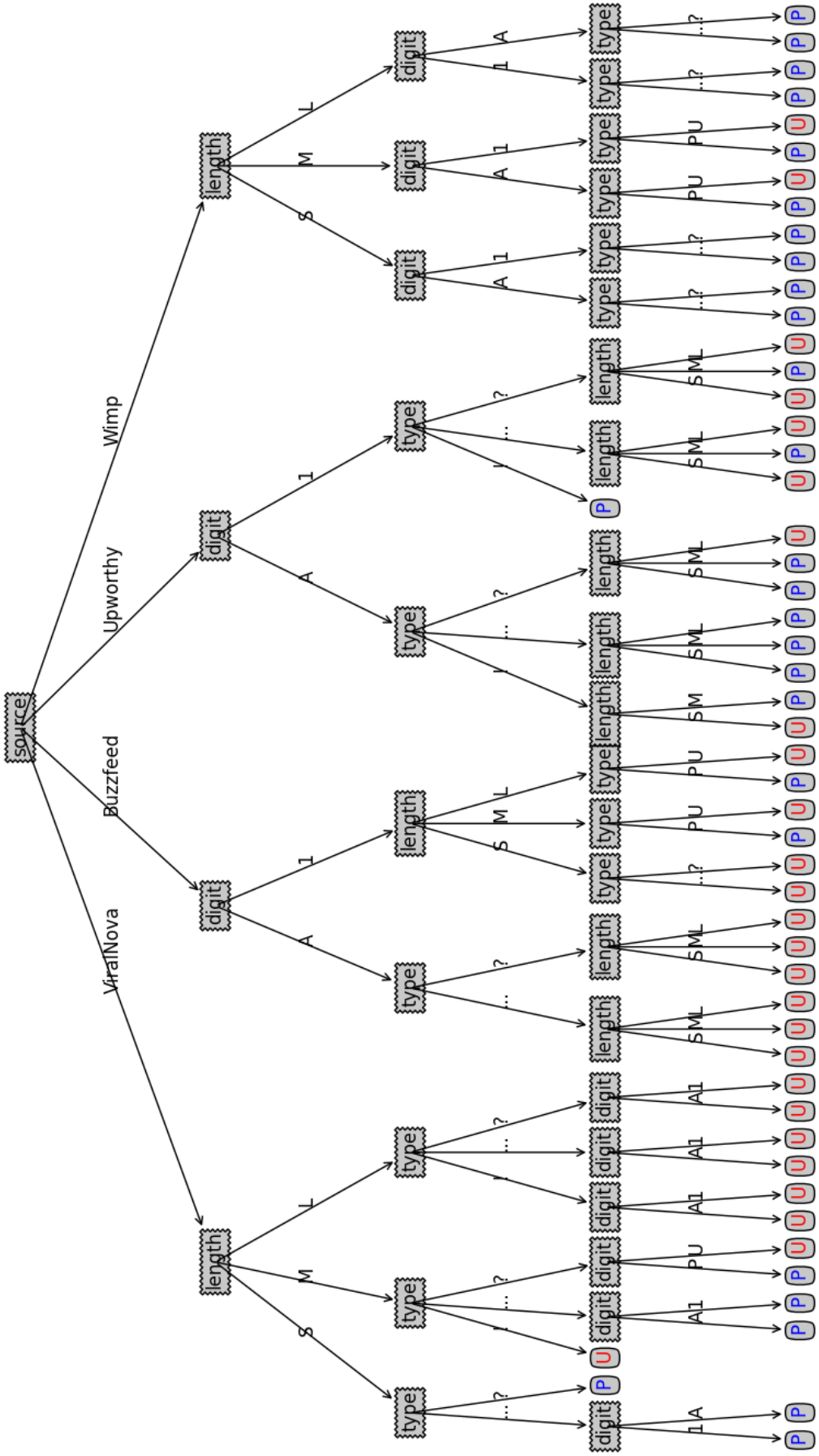
Cette fonction fait appelle à plusieurs autre fonctions qui se retrouvent à l'annexe E. Elle appelle d'abord une fonction pour trouver la meilleure caractéristique sur laquelle splitter. Cette dernière utilise le concept d'entropie de Shannon (vue dans la section "Théorie") afin de déterminer cette caractéristique. Puis elle ajoute la séparation à une séquence de type dictionnaire. Ensuite, pour chaque nouvelle branche de l'arbre, l'algorithme s'appelle elle-même afin de splitter à nouveau chaque section en sous-sections. La deuxième caractéristique utilisée ne sera pas nécessairement la même pour toute les sections de l'arbre car chaque section calcul indépendemment que caractéristique organiserait le mieux son information. Au début de cette fonction, il y a 2 conditions d'arrêt de spécifié: soit que tous les informations restantes sont de la même catégorie et qu'il n'est donc plus nécessaire de splitter l'arbre ou que toute les caractéristiques ont été utilisées.

Nous avons maintenant un arbre de décision qui a été généré, mais Python n'a pas d'outils intégrés permettant de bien visualiser les données. Nous avons donc créé nos propres fonctions à partir de la bibliothèque de Matplotlib. L'intégralité de ces fonctions se retrouve à

l’annexe E. Ce code a fortement été inspiré par du code qui se retrouve au chapitre 3 du livre “Machine Learning in Action” par Peter Harrington. Nous n’allons pas élaborer sur ce code puisqu’il ne sert qu’à visualiser les donner.

Nous utilisons cet algorithme sur nos données et nous obtenons l’arbre de décision montré à la figure 2.

Figure 2 : Arbre de décision pour déterminer les articles très populaires



Légende de la figure 2:
source: site web sur lequel se retrouve l'article
- ViralNova
- Buzzfeed
- Upworthy
- Wimp

length: la longueur du titre
- S: court *(en: small)*
- M: medium
- L: long

type: type de la phrase
- ?: interrogatif
- !: exclamatif
- …: déclaratif

digit: si le titre contient un nombre
- A: le titre ne contient pas de nombres
- 1: le titre contient des nombres

catégorie:
- P: populaire
- U: non populaire

On peut remarquer sur l'arbre de décision que si une sous-branche ne contient plus de données avec un certain attribut d'une caractéristique qui n'a pas encore été utilisé pour séparé les données, cet attribut ne deviendra pas une sous-branche. Par exemple, les titres de Buzzfeed ne contenant pas de chiffres sont tous de type interrogatif ou déclaratif: il n'y a donc pas de sous-branche pour le type exclamatif. Si une branche ne lui reste qu'une caractéristique et que tous ces données ont le même attribut, celle-ci ne pourra pas effectuer de séparation et donc montra, si tel est le cas, qu'il y a des données des deux catégories dans cette sous-branche. La caractéristique du type de phrase pour les titres de Buzzfeed contenant un nombre et étant long en est un exemple.

Il est intéressant de remarquer qu'à 3 endroits sur l'arbre de décisions où une décision est atteint avant d'avoir séparé les données avec les 4 caractéristiques. Si le titre est du site web ViralNova, qu'il est court et qu'il est de type interrogatif, celui-ci sera probablement populaire peu importe s'il contient un nombre. Si le titre est du site web ViralNova, qu'il est de longueur moyenne et qu'il est de type exclamatif, celui-ci ne sera probablement pas populaire peu importe s'il contient un nombre. Puis si le titre est du site web Upworthy, qu'il contient un nombre et qu'il est de type exclamatif, celui-ci sera probablement populaire.

Le reste des branches utilisent les 4 caractéristiques. Il est toutefois intéressant de remarquer que toute les catégories d'une sous branche sont parfois de la même catégorie. Par exemple, tous les sous-branches des titres ViralNova qui sont longs sont dans la catégorie non populaire. Également, tous les sous-branches des les titres de Buzzfeed ne contenant pas de nombre sont

dans la catégorie non populaire. Un autre exemple: tous les sous-branches des titres de Wimp qui sont court ou long sont dans la catégorie populaire.

Par contre, il est parfois nécessaire d'utiliser les 4 caractéristiques pour catégoriser l'information. Par exemple, un titre de Upworthy ne contenant pas de nombre et étant de type interrogatif peut être dans la catégorie populaire ou non populaire dépendemment de sa longueur.

Maintenant que nous pouvons construire un arbre de décisions, nous pouvons tester notre algorithme. Le code 8 montre les 2 principales fonctions pour le testage. L'intégralité du code se retrouve à l'annexe E.

Code 8 : Arbre de décisions - Testage

```
def testTreeOnce():
    titlesVocab,charCount,numbers01,question01,binarySuccess,website = allData()
    vocabList = createVocabList(titlesVocab)

    dataSet, featLabels = treePlotterCharacteristics()
    trainingSetSize = int(floor(0.9*len(dataSet)))
    set = range(len(dataSet))
    testDataSet = []; testClasses = []
    for i in range(len(dataSet)-trainingSetSize):
        randIndex = int(random.uniform(0,len(set)))
        testDataSet.append(dataSet[set[randIndex]])
        testClasses.append(dataSet[set[randIndex]][-1])
        del(set[randIndex])
    trainDataSet=[]; trainClasses = []
    for index in set:
        trainDataSet.append(dataSet[index])
        trainClasses.append(dataSet[index][-1])

    inputTree = createTree(trainDataSet,featLabels)

    index = 0
    errorRate = 0
    for testVec in testDataSet:
        featLabels = ['length','digit','type','source']
        classLabel2 = classify(inputTree,featLabels,testVec)
        if classLabel2 != testClasses[index]:
            errorRate += 1
        index += 1
    errorRate = errorRate/float(index)

    return errorRate

def classify(inputTree,featLabels,testVec):
    classLabel = 0
```

```
    firstStr = inputTree.keys()[0]
    secondDict = inputTree[firstStr]
    featIndex = featLabels.index(firstStr)
    for key in secondDict.keys():
        if testVec[featIndex] == key:
            if type(secondDict[key]).__name__=='dict':
                classLabel = classify(secondDict[key],featLabels,testVec)
            else: classLabel = secondDict[key]
    return classLabel
```

Ce code sépare d'abord les données en 2 groupes: un avec 90% des données qui servira à entraîner l'algorithme et un avec 10% des données qui servira à tester l'algorithme. Puis il créer un arbre de décision avec les données pour l'entraînement à l'aide de la fonction *createTree* (voir code 5). Par la suite, la fonction appelle la fonction *classify* afin de faire passer chaque titre du groupe pour le testage dans l'arbre de décision. La fonctionne retourne la catégorie prédit par l'arbre. Puis cette catégorie est comparée avec la catégorie réelle afin de déterminer si la prédiction était bonne. Nous pouvons ainsi calculer le taux d'erreur. Puisque le groupe servant au testage est choisi aléatoirement, le taux d'erreur varie un peu à chaque fois que nous utilisons l'algorithme. Nous exécutons donc l'algorithme 100 fois et prenons le taux d'erreur moyen. Celui que nous trouvons est de 0,430. Sans l'arbre de décision, le taux d'erreur est de 0,500 puisque la moitié des titres sont populaires. Ainsi, le taux d'information gagné est de $(0,500 - 0,430)/0,500 = 14\%$.

Maintenant que nous savons que notre algorithme fonctionne, nous pouvons utiliser l'algorithme pour prédire les chances qu'un certain titre devienne populaire. La fonction principale pour effectuer cette tâche se trouve au code 9.

Code 9 : Arbre de décision - Utilisation

```
def useTree():
    titles,charCount,numbers01,question01,binarySuccess,website = allData()
    vocabList = createVocabList(titles)
    dataSet,labels = treePlotterCharacteristics()
    inputTree = createTree(dataSet,labels)

    title = str(raw_input("What's the idea for your title?"))

    charCountAll, charCountPopular, charCountUnpopular = charCount2()
    charCount = len(title)
    if charCount < min(charCountPopular,charCountUnpopular):
        length = "S"
    elif charCount > max(charCountPopular,charCountUnpopular):
        length = "L"
    else:
        length = "M"
```

```
    titleFormated = title.split()
    titleFormated = str(titleFormated)
    titleFormated = titleFormated.split('\t')
    titleFormated = list(titleFormated)
    for index in range(len(titleFormated)):
        titleFormated[index] = titleFormated[index]\
            .strip('-').strip('.').strip(',').strip('?').strip('!').strip("'")
    titleFormated = [low.lower() for low in titleFormated if len(low) > 2]
    numbersFile = open("D:\\Python27\\MathieuRoy\\numbers.txt")
    numbersList = numbersFile.read()
    numbersList = numbersList.split('\n')
    numbersList = str(numbersList)
    numbersFormated = []
    for index in range(len(numbersList)):
        numbersFormated.extend(numbersList[index].split('\t'))
    numbers = "A"
    for number in numbersFormated:
        if number in title:
            numbers = "1"
            break

    interrogativeMark = "?"
    exclamatoryMark = "!"
    if interrogativeMark in title:
        phraseType = "?"
    elif exclamatoryMark in title:
        phraseType = "!"
    else:
        phraseType = "..."

    website = str(raw_input("What's the website of your article?"))

    featLabels = ['length','digit','type','source']
    testVec = [length, numbers, phraseType, website]
    print featLabels
    print testVec

    classLabel2 = classify(inputTree,featLabels,testVec)
    return classLabel2
```

Le code initialise demande à l'utilisateur d'entrée son idée de titre. Par la suite, il trouve sa longueur, s'il contient un nombre et son type. Puis il demande à l'utilisateur sur quel site web il va publier son article. Par la suite, l'algorithme utilise la fonction *classify* (voir code 9) pour prédire dans quelle catégorie le titre appartient, c'est-à-dire s'il sera populaire ou pas.

Analyse fréquentielle

Nous voulons trouver quels sont les mots les plus populaires dans les articles populaires et dans les articles pas populaires. Cela sera intéressant à connaître en soit et en plus nous aidera dans la section suivante pour effectuer une régression linéaire.

Pour s'y faire, nous utilisons plusieurs fonctions qui se retrouvent à l'annexe D. Toutefois, nous étudierons la fonction principale qui est celle du code 10.

Premièrement, nous séparons les titres dans deux listes: une pour les titres populaires et l'autre pour les titres non populaire. Puis nous utilisons la fonction *mostFreq* (voir annexe D), afin de déterminer le nombre de fois que chaque mot apparaît dans la liste des titres populaires, puis dans la liste des titres non populaires. Nous créons également des listes pour les mots vides et les mots pleins spécifiquement. Puis on élimine les mots qui apparaissent moins de 4 fois puisque ceux-ci ne seront pas assez fréquent pour faire une comparaison significative. On divise ensuite la liste des mots dans les titres populaires par le nombre total de mots dans tous les titres populaires. On fait de même pour les titres non populaires. Cela nous permet de savoir quel pourcentage des mots sont un mot spécifique. On divise ensuite le pourcentage pour chaque mot dans les titres populaires par celui dans les titres non populaires. Cela nous permet de savoir combien de fois plus souvent un mot apparaît dans les titres populaire en comparaisons aux titres non populaires. Afin de mieux visualiser les résultats, nous les classons du plus grand ratio au plus petit.

Code 10 : Calcul des fréquences relatives

```
def relativeFreqOnce(titles,binarySuccess,vocabList):
    titlesPopular = []; titlesUnpopular = []; allWords = []
    totalNumberWordsPopular = 0.0; totalNumberWordsUnpopular = 0.0
    for index in range(len(titles)):
        if binarySuccess[index] == 1:
            titlesPopular.append(titles[index])
            totalNumberWordsPopular += len(titles[index])
        else:
            titlesUnpopular.append(titles[index])
            totalNumberWordsUnpopular += len(titles[index])
    numberPopular,stopWordsNumberPopular,otherWordsNumberPopular\
        = mostFreq(titlesPopular)
    numberUnpopular,stopWordsNumberUnpopular,otherWordsNumberUnpopular\
        = mostFreq(titlesUnpopular)

    N = 4 # on ne garde que les mots apparraissant au moins N fois
    indexPopular = 0; indexUnpopular = 0
    numberPopularN = {}; numberUnpopularN = {}
    for word in numberPopular:
        if word[1] < N: break
        else: indexPopular += 1
```

```
    numberPopularN = numberPopular[0:indexPopular]
    for word in numberUnpopular:
        if word[1] < N: break
        else: indexUnpopular += 1
    numberUnpopularN = numberUnpopular[0:indexUnpopular]

    freqPopular = [(transfer[0],transfer[1]/totalNumberWordsPopular)\
        for transfer in numberPopularN]
    freqUnpopular = [(transfer[0],transfer[1]/totalNumberWordsUnpopular)\
        for transfer in numberUnpopularN]

    relativeFreq = {}
    for word in vocabList:
        popularIndex = 0
        for index in range(len(freqPopular)):
            if word in freqPopular[index]:
                popularIndex = index
                break
            else: popularIndex = "a"
        unpopularIndex = 0
        for index in range(len(freqUnpopular)):
            if word in freqUnpopular[index]:
                unpopularIndex = index
                break
            else: unpopularIndex = "a"
        if (popularIndex != "a") & (unpopularIndex != "a"):
            relativeFreq[word] =\
                freqPopular[popularIndex][1]/freqUnpopular[unpopularIndex][1]
    relativeFreq = sorted(relativeFreq.iteritems(),\
        key=operator.itemgetter(1),reverse=True)

    return relativeFreq
```

Le tableau 1 montre les mots vides et pleins avec le plus haut ratio.

Tableau 1: Mots avec le plus haut ratio populaire / non populaire

Rang (#)	Mots Vides	Ratio	Mots Pleins	Ratio
1	stop	2,355	baby	4,503
2	i've	2,295	cat	4,348
3	saw	2,019	closer	2,989
4	its	1,993	absolutely	2,826
5	took	1,902	cry	2,795

6	then	1,963	cutest	2,717
7	actually	1,812	note	2,717
8	which	1,812	left	2,484
9	you've	1,739	eyes	2,446
10	without	1,739	water	2,174

Le tableau 2 montre les mots avec le plus bas ratio.

Tableau 2: Mots avec le plus bas ratio populaire / non populaire

Rang (#)	Mots Vides	Ratio	Mots Plein	Ratio
1	there	0,320	wrong	0,290
2	than	0,322	facebook	0,311
3	what's	0,435	death	0,362
4	still	0,435	winter	0,395
5	too	0,448	secret	0,418
6	want	0,450	adorable	0,423
7	give	0,453	dead	0,435
8	nothing	0,466	olympics	0,466
9	very	0,466	history	0,483
10	happens	0,512	guess	0,483

Google+ vs Facebook

Tant qu'à avoir créer cette fonction, nous allons satisfaire notre curiosité: nous allons comparer les préférences des utilisateurs de Facebook par rapport aux utilisateur de Google+. Nous trouvons le ratio entre les "j'aime" de Facebook et les "+1" de Google+

$$ratio = (\#\, j'aime)/(\#\ + 1)$$

Si le nombre de "+1" est de 0, nous le remplaçons par 0,5 pour éviter les divisions par 0 et qui donnerait un ratio infini ce qui n'est pas réaliste car en fait, s'il y a 0 "+1" c'est surement juste parce que l'article n'a vraiment pas été populaire.

En utilisant du code de la sous-section "Fréquence" et en ajoutant la fonction *comparisonData* (voir annexe H), nous arrivons à trouver les mots qui sont plus populaires sur Facebook et ceux qui le sont plus sur Google+. Le tableau 3 montre les mots populaires sur Facebook.

Tableau 3 : Mots plein plus populaires sur un réseau en particulier

Rang (#)	Facebook	Fréquence relative	Google+	Fréquence relative
1	gay	3,463	inside	4,764
2	cat	3,463	thought	3,638
3	letter	3,463	idea	3,465
4	note	2,886	photos	3,032
5	baby	2,624	crazy	3,032
6	america	2,597	signs	2,490
7	kitter	2,309	brilliant	2,166
8	art	2,309	completely	2,104
9	story	2,309	abandoned	2,079
10	water	2,309	whoa	1,949
11	white	2,309	wrong	1,906
12	dog	2,261	pictures	1,906
13	car	2,116	couple	1,877
14	police	2,020	adorable	1,877
15	olympic	2,202	wow	1,877
16	women	1,847	hilarious	1,840
17	wrote	1,847	til	1,829
18	eyes	1,847	unbelievable	1,829
19	greatest	1,847	animal	1,819
20	wife	1,814	food	1,732

De plus, Facebook est 1,732 fois plus populaire sur Google+ et Google est 1,732 fois plus populaire sur Facebook.

Régression linéaire

Nous choisissons 2 caractéristiques reliées au titre, soit la popularité de ces mots et sa longueur. Nous définissons la popularité de ses mots comme

$$P_{titre} = 1/N * \sum_{mot=1}^{N} P_{mot}$$

où 'N' est le nombre de mot dans le titre et P_{mot} est la popularité de chaque mot tel que défini dans la sous-section "Fréquence" (voir les tableaux 1 et 2 par exemple). Puis nous définissons la longueur d'un titre comme le nombre de caractères qu'il contient incluant les espaces. Le code servant à mettre ces données dans des listes se trouve à l'annexe F.

Le code 8 montre l'algorithme servant à trouver les coefficents 'w' tel que défini dans la section "Théorie".

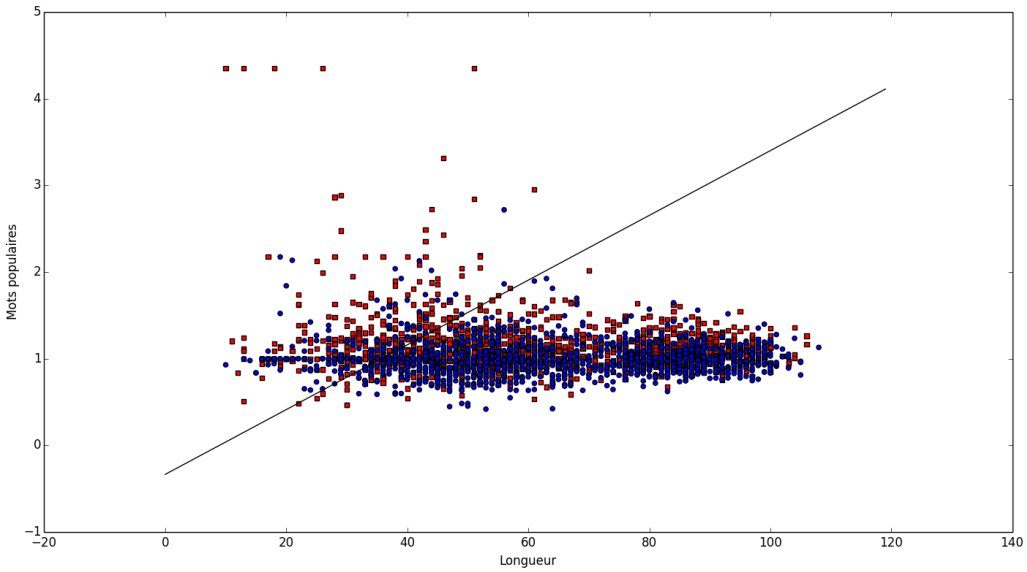
Code 8 : Coefficients de 'w'

```
def gradAscent(dataMatIn,labelMatIn):
    dataMat = mat(dataMatIn)
    labelMat = mat(labelMatIn).transpose()
    m,n = shape(dataMat)
    alpha = 0.001
    maxCycles = 500
    weights = ones((n,1))
    for k in range(maxCycles):
        multi = dataMat*weights
        h = sigmoid(multi)
        error = (labelMat - h)
        weights = weights + alpha * dataMat.transpose() * error
    return weights
```

La constante 'alpha' est petite afin de s'approcher tranquillement du maximum au lieu d'osciller autour de celui-ci. Nous avons mis la constante 'maxCycles' à 500 car cela laisse plus qu'assez d'itérations à l'algorithme pour s'approcher du maximum.

Nous mettons chaque titre avec la régression linéaire résultante dans le graphique 4. L'intégralité du code ayant servit à produire ce graphique se retrouve à l'annexe F.

Graphique 4 : Régression linéaire de la popularité moyenne des mots en fonction de la longueur du titre



Légende graphique 4
rouge: titre populaire
bleu: titre non populaire

Les coefficients trouvés sont: $w_0 = 24.99$, $w_1 = -2.79$ *et* $w_2 = 74.57$. Ainsi, l'équation de la régression linéaire est

$$y = (-w_0 - w_1 * x)/w_2 = -0.335 + 0.0374 * x$$

Si le 'y' du titre est plus grand que celui trouvé à l'aide de l'équation précédente, le titre sera catégorisé comme populaire; autrement il sera catégorisé comme non populaire.

Maintenant que l'algorithme pour l'entraînement est fait, nous pouvons la tester. Nous créons donc une fonction qui sépare nos données en deux: 90% pour l'entraînement et 10% pour le testage. Nous trouvons les coefficients 'w' avec le 90% des données d'entraînement, puis nous utilisons ceux-ci pour prédire dans quelle catégorie appartienne chacun des titres dans les données servant au testage. Nous comparons la prédiction avec la vrai valeur à l'aide de la fonction au code 9. Nous pouvons donc calculer le taux d'erreur. Puisque les données servant à l'entraînement sont prises aléatoirement, le taux d'erreur variera à chaque fois. Ainsi, nous exécutons cette fonction 100 fois et prenons la moyenne des taux d'erreur. L'intégralité du code ayant été utilisé pour trouver ce taux se retrouve à l'annexe G.

Code 9 : Régression linéaire - Classification

```
def classifyVector(inX, weights):
    prob = sigmoid(sum(inX*weights))
    if prob > 0.5: return 1.0
    else: return 0.0
```

Nous obtenons un taux d'erreur de 0.343. Cela représente un gain d'information de $(0.500 - 0.343)/0.500 = 31.4\%$. Notre taux d'erreur est relativement très bas. Nous pouvons donc utiliser cet algorithme pour prédire si un titre sera populaire ou pas. Nous créons une fonction *useRegres* qui demande à l'utilisateur d'entrer son idée de titre. Puis la fonction calcule la popularité moyenne de ses mots ainsi que la longueur totale du titre. Il utilise toute notre banque de données pour entraîner l'algorithme. Puis il se sert des coefficients 'w' trouvés pour faire une prédiction sur la catégorie du titre que l'utilisateur a entrée. Encore une fois, l'intégralité du code se retrouve à l'annexe F.

Discussion

Nous avons trouvé un taux d'erreur ainsi qu'un gain d'information pour chaque algorithme. Le tableau 3 récapitule cette information.

Tableau 3 : Taux d'erreur et gain d'information

	Taux d'erreur	**Gain d'information**
k voisins les plus proches	0.445	11.0%
Classification bayésienne	0.440	12.0%
Arbre de décisions	0.430	14.0%
Régression linéaire	0.343	31.4%

Le gain d'information a été calculé par rapport au taux d'erreur si nous choisissions aléatoirement, qui était donc de 0.500.

La méthode des k voisins les plus proches est a probablement été la plus facile à implémenter, mais elle prends beaucoup plus de temps à s'exécuter. De plus c'est l'algorithme qui donne le plus faible gain d'information.

La classification naïve bayésienne est tant qu'à elle un peu plus rapide, mais reste tout de même lent. De plus, bien que son gain d'information moyen soit de 12.0%, un aspect intéressant de celui-ci est que sont gain d'information peut varier de 0% jusqu'à des valeurs très hautes: un gain d'information de 50% n'est pas hors de porté! Ainsi, en plus d'estimer une catégorie, il peut dire à quel point il est confiant que celle-ci soit la bonne. C'est un avantage de cette méthode.

L'algorithme de l'arbre de décisions prend un peu de temps à généré son arbre; par contre, une fois celui-ci généré, il peut classer des titres très rapidement. Son gain d'information est un peu plus grand que les deux fonctions précédentes, mais pas de beaucoup. Toutefois, l'avantage significatif de cet algorithme est qu'il rend les informations facilement compréhensible par un être humain contrairement aux 2 algorithmes précédent.

La régression linéaire est à peu prêt aussi rapide que l'arbre de décisions: ce qui est bon. En plus, nous avons obtenus un gain d'information de 31.4% avec celle-ci, ce qui est excellent! Cet algorithme permet aussi de mettre l'information sous forme visuelle compréhensible pour l'humain.

Il est important de noter que la comparaison des gains d'information calculés ici à ses limites à nous permettre de comparer la performance des algorithmes car nous n'avons pas utilisé la même information, c'est-à-dire les mêmes caractéristiques, pour tous les algorithmes.

Il est a noté que les sites web que nous avons étudiés sont très populaires: ils reçoivent entre 4 et 13 millions de visite par mois (source: Ripenn). La plus part de ces sites de nouvelles utilisent déjà des techniques d'optimisation pour leurs titres. Par exemple, Wimp demande à ses employés de générer 25 titres pour chaque article et puis dans choisir 2 sur lesquels ils effectuent une technique de marketing appelé 'test A/B'. Donc en prenant ça en compte, un gain d'information de 10 à 30% est très bon.

Si nous avions analysé des sites web n'utilisant pas déjà des algorithmes d'optimisation, notre gain d'information aurait probablement été considérablement plus grand. Par exemple, il y a déjà 30% de leurs titres qui contiennent un nombre. Ils se sont donc surement rendu compte que ça l'attirait plus de lecteur d'utiliser un nombre dans le titre et ont déjà optimisé pour ce critère.

Il y a également des aspects d'un titre qui sont beaucoup plus difficile à étudier à l'aide d'apprentissage automatique. Par exemple, en lisant les titres les plus populaires, nous nous sommes rendu compte qu'un titre avait plus de chances d'être populaire s'il avait un aspect émotionnel où beaucoup de personnes pouvaient se sentir concernées par exemple, ou si le titre en disait assez pour piquer la curiosité du lecteur, mais pas trop afin de l'inciter à cliquer sur le lien. Un exemple de titre qui rassemble ces 2 aspects et qui a été très populaire est "This Guy's Wife Got Cancer, So He Did Something Unforgettable. The Last 3 Photos Destroyed Me." (source: Rippen). De plus, ce titre utilise "this guy" au lieu de "someone" parce que ça aide le lecteur à imaginer quelqu'un de spécifique et donc cela apporte une composante émotionnelle plus forte (source: Buffer). Finalement, la dernière phrase du titre incite le lecteur à lire l'article jusqu'au bout.

Pour améliorer l'algorithme, nous pourrions également considérer les groupes de 2 mots dans notre vocabulaire au lieu de seulement les mots individuels. De plus, il pourrait être intéressant d'agrandir la banque de titres étudiés afin d'avoir des résultats encore plus solide au niveau statistique.

Il faut également tenir en compte de d'autres aspects que le titre pour rendre un article populaire. Par exemple, il faut écrire un bon article pour inciter les personnes à le partager. Un bon titre sert surtout à inciter les gens à cliquer sur le lien, mais pas nécessairement à le partager.

Malgré les limites de l'approche d'apprentissage automatique, cet approche peut tout de même nous donner un gain d'information dans les 30% juste en analysant le titre, ce qui est très bon!

Conclusion

Pour terminer, nous avons remarqué que de travailler avec plusieurs méthode d'apprentissage automatique fonctionne bien afin de faire des prédictions. Nous ne recommandons pas de choisir seulement une de ces algorithmes pour faire de tels analyses, mais plutôt une combinaisons de ceux-ci car ils se complète bien. Certains sont plus facile à implémenter, tel la méthode des k voisins les plus proches, d'autres ont l'avantage de pouvoir donner un intervalle de confiance de sa catégorisation, tel la classification naïve bayésienne, d'autres sont plus précis tel la régression linéaire, et d'autres sont plus facile à visualiser par exemple, tel l'arbre de décisions.

Nous avons remarqué que le choix des mots utilisés dans le titre, ainsi que sa longueur, son type et s'il contient des nombres, sont des caractéristiques qui ont une corrélation significative avec la popularité de l'article. Nous avons choisi 10% des articles au hasard afin de tester nos algorithmes. Si nous avions estimé la catégorie de ces titres totalement au hasard, nous aurions obtenu un taux d'erreur d'environ 50% puisque 50% des titres sont classés comme populaires et 50% comme non populaire. Mais en analysant différentes caractéristiques des titres à l'aide de 4 algorithmes différents, nous avons réussi à baisser le taux d'erreur de 5 à 16%!

Pour améliorer ce taux d'erreur, nous pourrions essayer de différentier les mots par leur signification car certains mots ont plusieurs significations et d'autres ont plusieurs synonymes. Nous pourrions également analyser le corps du texte de l'article avec des méthodes similaires.

Les outils qui ont été créés peuvent être utile pour un journaliste qui hésite entre plusieurs titres pour la rédaction de son titre. Il pourra choisir celui qui a la plus haute probabilité d'être populaire.

Un bon titre incite les lecteurs à cliquer sur le lien pour lire l'article, mais afin qu'il le partage, il est nécessaire que l'article soit bon aussi. Il est important de savoir ce que les outils décrits dans cet ouvrage peut apporter, mais il est également important de reconnaître leur limite.

Références

Nouvelles

BuzzFeed. http://www.buzzfeed.com/.
ViralNova. http://viralnova.com/.
UpWorthy. http://www.upworthy.com/.
Wimp. http://www.wimp.com/.

Réseaux sociaux

Facebook. https://www.facebook.com/.
Twitter. https://twitter.com/.
Google+. https://plus.google.com/.
Pinterest. http://www.pinterest.com/.
LinkedIn. http://www.linkedin.com/.
Delicious. https://delicious.com/.
StumbleUpon. https://www.stumbleupon.com/.
Reddit. http://www.reddit.com/.
Digg. http://digg.com/.
Alexa. "Alexa Top 500 Sites". http://www.alexa.com/. 30 June 2009. Repéré en novembre 2011 par Wikipédia.

Logiciels

Python. https://www.python.org/
NumPy. http://www.numpy.org/.
Matplotlib. http://matplotlib.org/.
Python. PyParsing 2.0.2. https://pypi.python.org/pypi/pyparsing.
Python. Python Dateutil 2.2. https://pypi.python.org/pypi/python-dateutil.
Python. Pytz 2014.2. https://pypi.python.org/pypi/pytz/2014.2.
Python. Six 1.6.1. https://pypi.python.org/pypi/six.
Microsoft. Bloc-note. http://office.microsoft.com/fr-CA/?CTT=97

Information

Ripenn. 7 Things Marketers Can Learn From 2,616 Viral Headlines. Repéré en avril 2014. http://www.ripenn.com/blog/7-things-marketers-can-learn-from-2616-viral-headlines/#bonus.
Ranks NL. English stopwords list. Repéré en avril 2014. http://www.ranks.nl/stopwords.
http://web.cs.swarthmore.edu/~meeden/cs63/f11/ml-intro.pdf
W3Techs. Usage of content languages for websites. Repéré en mai 2014. http://w3techs.com/technologies/overview/content_language/all.
Buffer. How to Write The Perfect Headline: The Top Words Used in Viral Headlines. http://blog.bufferapp.com/the-most-popular-words-in-most-viral-headlines.
Harrington, Peter. Machine Learning in Action. ISBN: 9781617290183. Manning. Avril 2012. 384 pages. http://www.manning.com/pharrington/

Bibliographie

Guido Van Rossum et Fred L. Drake Jr. The Python Tutorial (2.7.5). ebshelf Inc. Septembre 2013. 143 pages.

Ng, Andrew. Machine Learning. Standford via Coursera.
https://class.coursera.org/ml-003/lecture

Annexes

Annexe A : Préparer les données

```
################ FORMATING THE DATA ################

def allData(): #meta fonction pour formater tous les donnees
    titles,likeShare,website = importData()
    charCountV = charCount(titles)
    numbersV = numbers()
    binarySuccessV = binarySuccess(success(likeShare))
    questionV = question(titles)
    titlesV = []
    for title in titles:
        titleFormated = title.split()
        for index in range(len(titleFormated)):
            titleFormated[index] =
titleFormated[index].strip('-').strip('.').strip(',').strip('?').strip('!').strip("'")
        titleFormated = [low.lower() for low in titleFormated if len(low) > 2]
        titlesV.append(titleFormated)
    return titlesV,charCountV,numbersV,questionV,binarySuccessV,website

def importData():
    #charCount, number, question, sexualOrientation, WORDS, likeShare
    rawData = open("D:\\Python27\\MathieuRoy\\title6.txt")
    rawDataRead = rawData.read() #car readlines() ne fonctionne pas
    rawDataReadSplit = rawDataRead.split('\n')
    numberOfLines = len(rawDataReadSplit)
    numberOfColums = 20
    titles = []
    likeShare = zeros((numberOfLines,11))
    website = []
    index = 0
    for line in rawDataReadSplit:
        line = line.strip() # enleve les espaces au debut et a la fin du string
        listFromLine = line.split('\t') #split un string au \t (ie. une tab) et les enleve
        titles.extend(listFromLine[0:1])
        likeShare[index] = listFromLine[7:18]
        website.append(listFromLine[-1])
        index += 1
    return titles, likeShare, website

def success(likeShare):
    success = []
    index = 0
    for line in likeShare:
        success.append(sum(line))
        index += 1
```

```
    return success

def binarySuccess(success):
    average = sum(success)/len(success)
    medianD = median(success)
    binarySuccess = []
    for number in success:
        if number >  medianD : # average: #
            binarySuccess.append(1) #popular
        else:
            binarySuccess.append(0) #unpopular
    #print average, medianD
    return binarySuccess

def ternarySuccess():
    titles,likeShare,website = importData()
    successV = success(likeShare)
    successV.sort()
    first = int(1/float(3)*len(successV))
    second = int(2/float(3)*len(successV))
    ternarySuccess = []
    for number in successV:
        if number < successV[first]: #666
            ternarySuccess.append(0) #not popular
        elif number < successV[second]: #4950
            ternarySuccess.append(1) #popular
        else:
            ternarySuccess.append(2) #very popular
    return ternarySuccess

def median(mylist):
    sorts = sorted(mylist)
    length = len(sorts)
    if not length % 2:
        return (sorts[length / 2] + sorts[length / 2 - 1]) / 2.0
    return sorts[length / 2 - 0.5]

def charCount(titles):
    charCount = []
    for title in titles:
        charCount.append(len(title))
    return charCount

def question(titles):
    questionMark = "?"
    question = []
    for index in range(len(titles)):
        if questionMark in titles[index]:
            question.append(1)
        else:
```

```
            question.append(0)
    return question

def numbers():
    rawData = open("D:\\Python27\\MathieuRoy\\title6.txt")
    rawDataRead = rawData.read() #car readlines() ne fonctionne pas
    rawDataReadSplit = rawDataRead.split('\n')
    titles = []
    index = 0
    for line in rawDataReadSplit:
        line = line.strip() # enleve les espaces au debut et a la fin du string
        listFromLine = line.split('\t') #split un string au \t (ie. une tab) et les enleve
        titles.extend(listFromLine[0:1])
        index += 1

    numbersFile = open("D:\\Python27\\MathieuRoy\\numbers.txt")
    numbersList = numbersFile.read()
    numbersList = numbersList.split('\n')
    numbersFormated = []
    for index in range(len(numbersList)):
        numbersFormated.extend(numbersList[index].split('\t'))
    numbers = []
    for title in titles:
        isOne = 0
        for number in numbersFormated:
            if number in title:
                numbers.append(1)
                isOne = 1
                break
        if isOne == 0:
            numbers.append(0)
    return numbers


############### FORMATING MORE DATA ###############

#creer un vecteur train avec ces classes et un vecteur test avec ces classes
def trainTest(titles,success):
    vocabList = createVocabList(titles)
    #pourrait ajouter une variable pour le ratio et mettre 0.9 par defaut
    trainingSetSize = int(floor(0.9*len(titles)))
    set = range(len(titles))
    testTitlesVec = []; testClasses = []
    for i in range(len(titles)-trainingSetSize):
        randIndex = int(random.uniform(0,len(set)))
        testTitlesVec.append(title2Vec(vocabList,titles[set[randIndex]]))
        testClasses.append(success[set[randIndex]])
        del(set[randIndex])
    trainTitlesVec=[]; trainClasses = []
    for index in set:
```

```
        trainTitlesVec.append(title2Vec(vocabList, titles[index]))
        trainClasses.append(success[index])
    return trainTitlesVec, trainClasses, testTitlesVec, testClasses

def createVocabList(titles):
    vocabSet = set([]) #creer un set vide
    for title in titles:
        vocabSet = vocabSet | set(title) #union de 2 sets
    vocabList = list(vocabSet)
    return vocabList

def title2Vec(vocabList,title):
    titleVec = [0]*len(vocabList)
#    words = title.split()
#    words = title.lower()
    for word in title:
        if word in vocabList:
            titleVec[vocabList.index(word)] = 1 #si le mot apparait dans le titre (le nombre
de fois n'est pas important)
        else: print "the word: %s is not in my vocabulary" % word
    return titleVec

def titles2Vec(vocabList,titles):
    titlesVec = []
    for title in titles:
        title2vec = title2Vec(vocabList,title)
        titlesVec.append(title2vec)
    return titlesVec
```

Annexe B : Classification naïve bayésienne

```
############### KNN ###############

def testKNN(x=2,k=3,s=2):
    x=x-1
    errorCountTotal = 0
    for i in range(x):
        errorCount, numTestVec = testKNNOnce(k,s)
        errorCountTotal+=errorCount
    return errorCountTotal/float(numTestVec)/float(x)

def testKNNOnce(k,s):
    titles,charCount,numbers,question,binarySuccess,website = allData()
    ternarySuccessV = ternarySuccess()
    if s==2: success=binarySuccess
    elif s==3: success=ternarySuccessV
    else: print "not supported yet"; return
    vocabList = createVocabList(titles)
    titlesVec = titles2Vec(vocabList,titles)
    trainTitlesVec, trainClasses, testTitlesVec, testClasses = trainTest(titles,success)
    trainTitlesVec, ranges, minVals = autoNorm(array(trainTitlesVec)) #
    testTitlesVec, ranges, minVals = autoNorm(array(testTitlesVec)) #
    errorCount = 0.0
    numTestVec = len(testTitlesVec)
    for i in range(numTestVec):
        classifierResult = classify0(testTitlesVec[i][:],trainTitlesVec,trainClasses,k)
        #print "the classifier came back with: %d, the real answer is: %d"\
        #    % (classifierResult, testClasses[i])
        if (classifierResult != success[i]): errorCount += 1.0
        #print "so far the total error rate is: %f" % (errorCount/float(i+0.001))
    return errorCount, numTestVec
    print "the total error rate is: %f" % (errorCount/float(numTestVec[-1]))

def classify0(titleVec, titlesVec, binarySuccess, k):
    titlesSize = len(titlesVec)
    title = array(titleVec); titlesVec = array(titlesVec)
    diffMat = tile(titleVec, (titlesSize,1)) - titlesVec
    sqDiffMat = diffMat**2
    sqDistances = sqDiffMat.sum(axis=1)
    distances = sqDistances**0.5
    sortedDistIndicies = distances.argsort()
    classCount={}
    for i in range(k):
        voteIlabel = binarySuccess[sortedDistIndicies[i]]
        classCount[voteIlabel] = classCount.get(voteIlabel,0) + 1
    sortedClassCount = sorted(classCount.iteritems(),key=operator.itemgetter(1), reverse=True)
    return sortedClassCount[0][0]
```

```
#en fait peut-etre que les valeurs qui apparraissent plus souvent devrait avoir plus de poids?
#Finalement non, cela semble vraiment utile!
def autoNorm(dataSet):
    minVals = dataSet.min(0)
    maxVals = dataSet.max(0)
    ranges = maxVals - minVals
    normDataSet = zeros(shape(dataSet))
    m = dataSet.shape[0]
    normDataSet = dataSet - tile(minVals, (m,1))
    normDataSet = normDataSet/tile(ranges, (m,1))
    return normDataSet, ranges, minVals

def useKNN():
    titles,charCount,numbers,question,binarySuccess,website = allData()
    vocabList = createVocabList(titles)
    titlesVec = titles2Vec(vocabList,titles)
    resultList = ['Unpopular','Popular']
    title = str(raw_input("What's the idea for your title?"))
    titleFormated = title.split()
#.split('.').split(',').split('?').split('!').split("'").strip('?')
    titleFormated = [low.lower() for low in titleFormated if len(low) > 1]
    titleVec = title2Vec(vocabList,titleFormated)
    normDataSet, ranges, minVals = autoNorm(array(titlesVec))
    classifierResult = classify0((titleVec-minVals)/ranges,normDataSet,binarySuccess,3)
    print "Your title will probably be: ",resultList[classifierResult]
```

Annexe C : Classification naïve bayésienne

```
############### BAYES ###############

def testBayes2(number):
    titles,charCount,numbers,question,binarySuccess,website = allData()
    errorRateT = 0; pBinarySuccessT = 0
    for i in range(number):
        errorRate, pBinarySuccess = testBayes2Once(titles,binarySuccess)
        errorRateT += errorRate
        pBinarySuccessT += pBinarySuccess
    errorRate = errorRateT/float(number)
    pBinarySuccess = pBinarySuccessT/float(number)
    print 'le taux derreur est: ', errorRate
    print 'en choisissant au hasard on aurait obtenu: ', pBinarySuccess

def testBayes2Once(titles,binarySuccess):
    trainTitlesVec, trainClasses, testTitlesVec, testClasses = trainTest(titles,binarySuccess)
    p0Vec,p1Vec,pBinarySuccess = trainBayes2(array(trainTitlesVec),array(trainClasses))
    errorCount = 0
    for index in range(len(testTitlesVec)):
        experiment = classifyBayes2(array(testTitlesVec[index]),p0Vec,p1Vec,pBinarySuccess)
        theory = testClasses[index]
        if experiment != theory:
            errorCount += 1
    errorRate = float(errorCount)/float(len(testTitlesVec))
    return errorRate, pBinarySuccess

#Dans la version 2, nous enlevons les mots vides
def testBayes2v2(number):
    titles,charCount,numbers,question,binarySuccess,website = allData()
    errorRateT = 0; pBinarySuccessT = 0
    for i in range(number):
        errorRate, pBinarySuccess, vocabList, p0V, p1V =
testBayes2v2Once(titles,binarySuccess)
        errorRateT += errorRate
        pBinarySuccessT += pBinarySuccess
    errorRate = errorRateT/float(number)
    pBinarySuccess = pBinarySuccessT/float(number)
    print 'le taux derreur est: ', errorRate
    print 'en choisissant au hasard on aurait obtenu: ', pBinarySuccess
    return vocabList,p0V,p1V

def testBayes2v2Once(titles, binarySuccess):
    N = 30
    vocabList = createVocabList(titles)
    stopWords = open("D:\\Python27\\MathieuRoy\\stopWords.txt")
    stopWords = stopWords.read()    #car readlines() ne fonctionne pas
    stopWords = stopWords.split('\n')       #split les 'enter'
```

```
    titlesPopular = []; titlesUnpopular = []; allWords = []
    for index in range(len(titles)):
        if binarySuccess[index] == 1:
            titlesPopular.append(titles[index])
        else:
            titlesUnpopular.append(titles[index])
        allWords.extend(titles[index])
    sortedFreq,sortedStopWordsFreq,sortedSignWordsFreq = mostFreq(allWords)
    for pairW in sortedFreq:
        if (pairW[0] in stopWords) & (pairW[0] in vocabList): vocabList.remove(pairW[0])
    trainTitlesVec, trainClasses, testTitlesVec, testClasses = trainTest(titles,binarySuccess)
    p0V,p1V,pBinarySuccess = trainBayes2(array(trainTitlesVec),array(trainClasses))
    errorCount = 0
    for index in range(len(testTitlesVec)):
        testTitleVec = testTitlesVec[index]
        if classifyBinarySuccess(array(testTitleVec),p0V,p1V,pBinarySuccess) !=
testClasses[index]:
            errorCount += 1
    errorRate = float(errorCount)/len(testTitlesVec)
    return errorRate, pBinarySuccess, vocabList, p0V, p1V

def trainBayes2(trainTitlesVec,trainClasses):
    numTrainTitles = len(trainTitlesVec)
    numWords = len(trainTitlesVec[0])
    pBinarySuccess = sum(trainClasses)/float(numTrainTitles)
    p0Num = ones(numWords); p1Num = ones(numWords)      #a ete change en 1
    p0Denom = 2.0; p1Denom = 2.0                        #a ete change a 2.0
    for i in range(numTrainTitles):
        if trainClasses[i] == 1:
            p1Num += trainTitlesVec[i]
            p1Denom += sum(trainTitlesVec[i])
        else:
            p0Num += trainTitlesVec[i]
            p0Denom += sum(trainTitlesVec[i])
    p1Vec = [log(this/p1Denom) for this in p1Num] #a ete change en log
    p0Vec = [log(this/p0Denom) for this in p0Num] #a ete change en log
    return p0Vec,p1Vec,pBinarySuccess
#chance que mot X se retrouve dans une des categories binaire ET pourcentage des titres qui se
retrouvent dans une des categories binaires

def classifyBayes2(titleVec, p0Vec, p1Vec, pBinarySuccess):
    p1 = sum(titleVec * p1Vec) + log(pBinarySuccess)
    p0 = sum(titleVec * p0Vec) + log(1.0 - pBinarySuccess)
    if p1 > p0:
        return 1
    else:
        return 0

def useBayes2():
    titles,charCount,numbers,question,binarySuccess,website = allData()
```

```
    vocabList = createVocabList(titles)
    titlesVec = titles2Vec(vocabList, titles)
    p0Vec,p1Vec,pBinarySuccess = trainBayes2(array(titlesVec),array(binarySuccess))
    for time in range(25):
        title = str(raw_input("What's the idea for your title?"))
        titleFormated = title.split()
        for index in range(len(titleFormated)):
            titleFormated[index] = titleFormated[index]\
                .strip('-').strip('.').strip(',').strip('?').strip('!').strip("'")
        titleFormated = [low.lower() for low in titleFormated if len(low) > 2]
        titleVec = array(title2Vec(vocabList,titleFormated))
        p0,p1,classResult = classifyBayes2(titleVec,p0Vec,p1Vec,pBinarySuccess)
        if classResult == 0:
            print 'the title will be UNPOPULAR with a probability of', p1/float(p0)
        else: print 'the title will be POPULAR with a probability of', p0/float(p1)
#        again = str(raw_input("Do you want to try again?"))
#        yes = "yesYesYESyes!Yes!YES!1sureSureOkokOK"
#        if again not in yes :
#            break
#    return classResult

def testBayes2v3(number):
    titles,charCount,numbers,question,binarySuccess,website = allData()
    errorRateSureT = 0; errorRateUnsureT = 0; pBinarySuccessT = 0
    for i in range(number):
        errorRateSure,errorRateUnsure,pBinarySuccess = testBayes2v3Once(titles,binarySuccess)
        errorRateSureT += errorRateSure
        errorRateUnsureT += errorRateUnsure
        pBinarySuccessT += pBinarySuccess
    errorRateSureT = errorRateSureT/float(number)
    errorRateUnsureT = errorRateUnsureT/float(number)
    pBinarySuccess = pBinarySuccessT/float(number)
    print 'le taux derreur est quand on est confiant: ', errorRateSureT
    print 'le taux derreur est quand on nest PAS confiant: ', errorRateUnsureT
    print 'en choisissant au hasard on aurait obtenu: ', pBinarySuccess

def testBayes2v3Once(titles,binarySuccess,certainty):
    trainTitlesVec, trainClasses, testTitlesVec, testClasses = \
        trainTest(titles,binarySuccess)
    p0Vec,p1Vec,pBinarySuccess = trainBayes2(array(trainTitlesVec),array(trainClasses))
    errorCountSure = 0; countSure = 0
    errorCountUnsure = 0; countUnsure = 0
    for index in range(len(testTitlesVec)):
        p0,p1,experiment = \
            classifyBayes3(array(testTitlesVec[index]),p0Vec,p1Vec,pBinarySuccess)
        theory = testClasses[index]
        if experiment != theory:
            if experiment == 1:
               if float(p0)/float(p1) > float(certainty): errorCountSure += 1
               else: errorCountUnsure += 1
```

```
            elif experiment == 0:
                if float(p1)/float(p0) > float(certainty): errorCountSure += 1
                else: errorCountUnsure +=1
        if experiment == 1:
            if float(p0)/float(p1) > float(certainty): countSure += 1
            else: countUnsure +=1
        elif experiment == 0:
            if float(p1)/float(p0) > float(certainty): countSure += 1
            else: countUnsure +=1
    errorRateSure = float(errorCountSure)/float(countSure)
    errorRateUnsure = float(errorCountUnsure)/float(countUnsure)
    return errorRateSure,errorRateUnsure,pBinarySuccess

def classifyBayes3(titleVec, p0Vec, p1Vec, pBinarySuccess):
    p1 = sum(titleVec * p1Vec) + log(pBinarySuccess)
    p0 = sum(titleVec * p0Vec) + log(1.0 - pBinarySuccess)
    if p1 > p0:
        return p0, p1, 1
    else:
        return p0, p1, 0
```

Annexe D : Analyse fréquentielle

```
############### FREQUENCY ###############

def mostFreq(titles):
    vocabList = createVocabList(titles)
    allWords = []
    for title in titles:
        allWords.extend(title)
    freqDict = {}
    for token in vocabList:
       freqDict[token]=allWords.count(token)
    number = sorted(freqDict.iteritems(), key=operator.itemgetter(1),reverse=True)
    stopWords = open("D:\\Python27\\MathieuRoy\\stopWords.txt")
    stopWords = stopWords.read()     #car readlines() ne fonctionne pas
    stopWords = stopWords.split('\n')       #split les 'enter'
    stopWordsNumber = []
    otherWordsNumber = []
    for wordFreq in number:
        if wordFreq[0] in stopWords:
            stopWordsNumber.append(wordFreq)
        else:
            otherWordsNumber.append(wordFreq)
    return number,stopWordsNumber,otherWordsNumber

def relativeFreq():
    titles,charCount,numbers,question,binarySuccess,website = allData()
    vocabList = createVocabList(titles)

    relativeFreqTotal = relativeFreqOnce(titles,binarySuccess,vocabList)

    stopWords = open("D:\\Python27\\MathieuRoy\\stopWords.txt")
    stopWords = stopWords.read()     #car readlines() ne fonctionne pas
    stopWords = stopWords.split('\n')       #split les 'enter'
    otherWords = []
    for word in vocabList:
        if word not in stopWords:
            otherWords.append(word)

    relativeFreqStopWords = relativeFreqOnce(titles,binarySuccess,stopWords)
    relativeFreqOtherWords = relativeFreqOnce(titles,binarySuccess,otherWords)

    return relativeFreqTotal, relativeFreqStopWords, relativeFreqOtherWords

def relativeFreqOnce(titles,binarySuccess,vocabList):
    titlesPopular = []; titlesUnpopular = []; allWords = []
    totalNumberWordsPopular = 0.0; totalNumberWordsUnpopular = 0.0
    for index in range(len(titles)):
        if binarySuccess[index] == 1:
```

```
            titlesPopular.append(titles[index])
            totalNumberWordsPopular += len(titles[index])
        else:
            titlesUnpopular.append(titles[index])
            totalNumberWordsUnpopular += len(titles[index])
    numberPopular,stopWordsNumberPopular,otherWordsNumberPopular\
        = mostFreq(titlesPopular)
    numberUnpopular,stopWordsNumberUnpopular,otherWordsNumberUnpopular\
        = mostFreq(titlesUnpopular)

    N = 4 # on ne garde que les mots apparraissant au moins N fois
    indexPopular = 0; indexUnpopular = 0
    numberPopularN = {}; numberUnpopularN = {}
    for word in numberPopular:
        if word[1] < N: break
        else: indexPopular += 1
    numberPopularN = numberPopular[0:indexPopular]
    for word in numberUnpopular:
        if word[1] < N: break
        else: indexUnpopular += 1
    numberUnpopularN = numberUnpopular[0:indexUnpopular]

    freqPopular = [(transfer[0],transfer[1]/totalNumberWordsPopular)\
        for transfer in numberPopularN]
    freqUnpopular = [(transfer[0],transfer[1]/totalNumberWordsUnpopular)\
        for transfer in numberUnpopularN]

    relativeFreq = {}
    for word in vocabList:
        popularIndex = 0
        for index in range(len(freqPopular)):
            if word in freqPopular[index]:
                popularIndex = index
                break
            else: popularIndex = "a"
        unpopularIndex = 0
        for index in range(len(freqUnpopular)):
            if word in freqUnpopular[index]:
                unpopularIndex = index
                break
            else: unpopularIndex = "a"
        if (popularIndex != "a") & (unpopularIndex != "a"):
            relativeFreq[word] =\
                freqPopular[popularIndex][1]/freqUnpopular[unpopularIndex][1]
    relativeFreq = sorted(relativeFreq.iteritems(),\
        key=operator.itemgetter(1),reverse=True)

    return relativeFreq

def getTopWords():
```

```
    vocabList,p0V,p1V = testBayes2v2(1)
    topPopular = []; topUnpopular = []
    for i in range(len(p0V)):
        if p0V[i] > -6.0 : topPopular.append((vocabList[i],p0V[i]))
        if p1V[i] > -6.0 : topUnpopular.append((vocabList[i],p1V[i]))
    sortedPopular = sorted(topPopular, key=lambda pair: pair[1], reverse=True)
    print "**********POPULAR**********"
    for item in sortedPopular:
        print item[0]
    sortedUnpopular = sorted(topUnpopular, key=lambda pair: pair[1], reverse=True)
    print "**********UNPOPULAR**********"
    for item in sortedUnpopular:
        print item[0]

def storeWordsFreq(wordsFreq):
    fw=open("D:\\LiberT\\Desktop\\PROJET\\mots.txt",'w')
    pickle.dump(wordsFreq,fw)
    fw.close()

def retrieveWordsFreq():
    fr = open("D:\\LiberT\\Desktop\\PROJET\\mots.txt")
    wordsFreq = pickle.load(fr)
    fr.close()
    return wordsFreq


################ FACEBOOK VS GOOGLE+ ################


def comparisonData():
    rawData = open("D:\\Python27\\MathieuRoy\\title6.txt")
    rawDataRead = rawData.read() #car readlines() ne fonctionne pas
    rawDataReadSplit = rawDataRead.split('\n')
    numberOfLines = len(rawDataReadSplit)
    numberOfColums = 20
    titles = []
    likeShare = zeros((numberOfLines,11))
    comparison = []
    website = []
    index = 0
    for line in rawDataReadSplit:
        line = line.strip() # enleve les espaces au debut et a la fin du string
        listFromLine = line.split('\t') #split un string au \t (ie. une tab) et les enleve
        titles.extend(listFromLine[0:1])
        likeShare[index] = listFromLine[7:18]
        listStrF = str(listFromLine[7:8])
        listStrG = str(listFromLine[12:13])
        listF = float(listStrF.strip("[").strip("]").strip("'"))
        listG = float(listStrG.strip("[").strip("]").strip("'"))
        if listG == 0: listG = 0.5
```

```
        comparison.append(listF/listG)
        website.append(listFromLine[-1])
        index += 1
    titlesFormated = []
    for title in titles:
        titleFormated = title.split()
        for index in range(len(titleFormated)):
            titleFormated[index] =
titleFormated[index].strip('-').strip('.').strip(',').strip('?').strip('!').strip("'")
        titleFormated = [low.lower() for low in titleFormated if len(low) > 2]
        titlesFormated.append(titleFormated)
    medianNetwork = median(comparison)
    binaryNetwork = []
    for number in comparison:
        if number > medianNetwork:
            binaryNetwork.append(1) #Facebook
        else:
            binaryNetwork.append(0) #Google+

    vocabList = createVocabList(titlesFormated)
    relativeFreqTotal = relativeFreqOnce(titlesFormated,binaryNetwork,vocabList)
    stopWords = open("D:\\Python27\\MathieuRoy\\stopWords.txt")
    stopWords = stopWords.read()    #car readlines() ne fonctionne pas
    stopWords = stopWords.split('\n')       #split les 'enter'
    otherWords = []
    for word in vocabList:
        if word not in stopWords:
            otherWords.append(word)

    relativeFreqStopWords = relativeFreqOnce(titlesFormated,binaryNetwork,stopWords)
    relativeFreqOtherWords = relativeFreqOnce(titlesFormated,binaryNetwork,otherWords)

    return relativeFreqTotal, relativeFreqStopWords, relativeFreqOtherWords
```

Annexe E : Création d'un arbre de décision

```
################ TREE PLOTTER CHARACTERISTICS ################

def treePlotCharacteristics():
    titles,charCount,numbers01,question01,binarySuccess,website = allData()
    charCountAll, charCountPopular, charCountUnpopular = charCount2()

    length = []
    for index in range(len(charCount)):
        if charCount[index] < min(charCountPopular,charCountUnpopular):
            length.append("S")
        elif charCount[index] > max(charCountPopular,charCountUnpopular):
            length.append("L")
        else:
            length.append("M")

    numbers = []
    for index in range(len(numbers01)):
        if numbers01[index] == 1:
            numbers.append("1")
        elif numbers01[index] == 0:
            numbers.append("A")

    rawData = open("D:\\Python27\\MathieuRoy\\title6.txt")
    rawDataRead = rawData.read() #car readlines() ne fonctionne pas
    rawDataReadSplit = rawDataRead.split('\n')
    titlesRaw = []
    index = 0
    for line in rawDataReadSplit:
        line = line.strip() # enleve les espaces au debut et a la fin du string
        listFromLine = line.split('\t') #split un string au \t (ie. une tab) et les enleve
        titlesRaw.extend(listFromLine[0:1])
        index += 1
    phraseType = [] #Declarative, Interrogative or Exclamatory
    interrogativeMark = "?"
    exclamatoryMark = "!"
    for index in range(len(titlesRaw)):
        if interrogativeMark in titlesRaw[index]:
            phraseType.append("?")
        elif exclamatoryMark in titlesRaw[index]:
            phraseType.append("!")
        else:
            phraseType.append("...")

    labels = ['length','digit','type','source']

    dataSet = range(len(binarySuccess))
    for index in range(len(binarySuccess)):
```

```
        if binarySuccess[index] == 1:
            dataSet[index] =
[length[index],numbers[index],phraseType[index],website[index],"P"]
        elif binarySuccess[index] == 0:
            dataSet[index] =
[length[index],numbers[index],phraseType[index],website[index],"U"]

    return dataSet, labels, #length, numbers, phraseType

def charCount2():
    titles,charCount,numbers01,question01,binarySuccess,website = allData()
    charCountPopular = 0; charCountUnpopular = 0
    for index in range(len(binarySuccess)):
        if binarySuccess[index] == 1:
            charCountPopular += charCount[index]
        else:
            charCountUnpopular += charCount[index]
    charCountPopular = charCountPopular/sum(binarySuccess)
    charCountUnpopular = charCountUnpopular/(len(binarySuccess)-sum(binarySuccess))
    charCountAll = sum(charCount)/len(charCount)

    shortPopular = 0; mediumPopular = 0; longPopular = 0;
    shortUnpopular = 0; mediumUnpopular = 0; longUnpopular = 0;
    length = []
    for index in range(len(binarySuccess)):
        if binarySuccess[index] == 1:
            if charCount[index] < min(charCountPopular,charCountUnpopular):
                shortPopular += 1
            elif charCount[index] > max(charCountPopular,charCountUnpopular):
                longPopular += 1
            else:
                mediumPopular += 1
        else:
            if charCount[index] < min(charCountPopular,charCountUnpopular):
                shortUnpopular += 1
            elif charCount[index] > max(charCountPopular,charCountUnpopular):
                longUnpopular += 1
            else:
                mediumUnpopular += 1
    shortPopular = shortPopular/float(sum(binarySuccess))
    mediumPopular = mediumPopular/float(sum(binarySuccess))
    longPopular = longPopular/float(sum(binarySuccess))
    shortUnpopular = shortUnpopular/float((len(binarySuccess)-sum(binarySuccess)))
    mediumUnpopular = mediumUnpopular/float((len(binarySuccess)-sum(binarySuccess)))
    longUnpopular = longUnpopular/float((len(binarySuccess)-sum(binarySuccess)))

    #print shortPopular, mediumPopular, longPopular
    #print shortUnpopular, mediumUnpopular, longUnpopular
    return charCountAll, charCountPopular, charCountUnpopular
```

```
def numbers2():
    titles,charCount,numbers01,question01,binarySuccess,website = allData()
    numbersPopular = 0; numbersUnpopular = 0
    for index in range(len(binarySuccess)):
        if binarySuccess[index] == 1:
            numbersPopular += numbers01[index]
        else:
            numbersUnpopular += numbers01[index]
    numbersPopular = numbersPopular/float(sum(binarySuccess))
    numbersUnpopular = numbersUnpopular/float((len(binarySuccess)-sum(binarySuccess)))
    numbersAll = sum(numbers01)/float(len(numbers01))
    return numbersAll, numbersPopular, numbersUnpopular

def phraseType2():
    titles,charCount,numbers01,question01,binarySuccess,website = allData()

    rawData = open("D:\\Python27\\MathieuRoy\\title6.txt")
    rawDataRead = rawData.read() #car readlines() ne fonctionne pas
    rawDataReadSplit = rawDataRead.split('\n')
    titlesRaw = []
    index = 0
    for line in rawDataReadSplit:
        line = line.strip() # enleve les espaces au debut et a la fin du string
        listFromLine = line.split('\t') #split un string au \t (ie. une tab) et les enleve
        titlesRaw.extend(listFromLine[0:1])
        index += 1
    phraseType = [] #Declarative, Interrogative or Exclamatory
    interrogativeMark = "?"
    exclamatoryMark = "!"
    for index in range(len(titlesRaw)):
        if interrogativeMark in titlesRaw[index]:
            phraseType.append("interrogative")
        elif exclamatoryMark in titlesRaw[index]:
            phraseType.append("exclamatory")
        else:
            phraseType.append("declarative")

    declarativePopular = 0; interrogativePopular = 0; exclamatoryPopular = 0;
    declarativeUnpopular = 0; interrogativeUnpopular = 0; exclamatoryUnpopular = 0;
    length = []
    for index in range(len(binarySuccess)):
        if binarySuccess[index] == 1:
            if phraseType[index] == "declarative":
                declarativePopular += 1
            elif phraseType[index] == "exclamatory":
                exclamatoryPopular += 1
            elif phraseType[index] == "interrogative":
                interrogativePopular += 1
        else:
            if phraseType[index] == "declarative":
```

```
                declarativeUnpopular += 1
            elif phraseType[index] == "exclamatory":
                exclamatoryUnpopular += 1
            elif phraseType[index] == "interrogative":
                interrogativeUnpopular += 1
#   print exclamatoryPopular, exclamatoryUnpopular

    declarativePopular = declarativePopular/float(sum(binarySuccess))
    interrogativePopular = interrogativePopular/float(sum(binarySuccess))
    exclamatoryPopular = interrogativePopular/float(sum(binarySuccess))
    declarativeUnpopular = declarativeUnpopular/float((len(binarySuccess)-sum(binarySuccess)))
    interrogativeUnpopular =
interrogativeUnpopular/float((len(binarySuccess)-sum(binarySuccess)))
    exclamatoryUnpopular =
interrogativeUnpopular/float((len(binarySuccess)-sum(binarySuccess)))

    print declarativePopular, interrogativePopular, exclamatoryPopular
    print declarativeUnpopular, interrogativeUnpopular, exclamatoryUnpopular

def website2():
    titles,charCount,numbers01,question01,binarySuccess,website = allData()
    websiteSet = set(website)
    websiteSet = list(websiteSet)
    BuzzfeedPopular = 0; WimpPopular = 0; UpworthyPopular = 0; ViralNovaPopular = 0;
    BuzzfeedUnpopular = 0; WimpUnpopular = 0; UpworthyUnpopular = 0; ViralNovaUnpopular = 0;
    for index in range(len(website)):
        if binarySuccess[index] == 1:
            if website[index] == "Buzzfeed":
                BuzzfeedPopular += 1
            elif website[index] == "Wimp":
                WimpPopular += 1
            elif website[index] == "Upworthy":
                UpworthyPopular += 1
            elif website[index] == "ViralNova":
                ViralNovaPopular += 1
        elif binarySuccess[index] == 0:
            if website[index] == "Buzzfeed":
                BuzzfeedUnpopular += 1
            elif website[index] == "Wimp":
                WimpUnpopular += 1
            elif website[index] == "Upworthy":
                UpworthyUnpopular += 1
            elif website[index] == "ViralNova":
                ViralNovaUnpopular += 1
    BuzzfeedPopular = BuzzfeedPopular/float(sum(binarySuccess))
    WimpPopular = WimpPopular/float(sum(binarySuccess))
    UpworthyPopular = UpworthyPopular/float(sum(binarySuccess))
    ViralNovaPopular = ViralNovaPopular/float(sum(binarySuccess))
    BuzzfeedUnpopular = BuzzfeedUnpopular/float(len(binarySuccess)-sum(binarySuccess))
    WimpUnpopular = WimpUnpopular/float(len(binarySuccess)-sum(binarySuccess))
```

```
    UpworthyUnpopular = UpworthyUnpopular/float(len(binarySuccess)-sum(binarySuccess))
    ViralNovaUnpopular = ViralNovaUnpopular/float(len(binarySuccess)-sum(binarySuccess))
    print BuzzfeedPopular, WimpPopular, UpworthyPopular, ViralNovaPopular
    print BuzzfeedUnpopular, WimpUnpopular, UpworthyUnpopular, ViralNovaUnpopular



############### TREES ###############

def calcShannonEnt(dataSet):
    numEntries = len(dataSet)
    labelCounts = {}
    for featVec in dataSet:
        currentLabel = featVec[-1]
        if currentLabel not in labelCounts.keys():
            labelCounts[currentLabel] = 0
        labelCounts[currentLabel] += 1
    shannonEnt = 0.0
    for key in labelCounts:
        prob = float(labelCounts[key])/numEntries
        shannonEnt -= prob * log(prob,2)
    return shannonEnt

def splitDataSet(dataSet, axis, value):
    retDataSet = []
    for featVec in dataSet:
        if featVec[axis] == value:
            reducedFeatVec = featVec[:axis]
            reducedFeatVec.extend(featVec[axis+1:])
            retDataSet.append(reducedFeatVec)
    return retDataSet

def chooseBestFeatureToSplit(dataSet):
    numFeatures = len(dataSet[0]) - 1
    baseEntropy = calcShannonEnt(dataSet)
    bestInfoGain = 0.0; bestFeature = -1
    for i in range(numFeatures):
        featList = [example[i] for example in dataSet]
        uniqueVals = set(featList)
        newEntropy = 0.0
        for value in uniqueVals:
            subDataSet = splitDataSet(dataSet, i, value)
            prob = len(subDataSet)/float(len(dataSet))
            newEntropy += prob * calcShannonEnt(subDataSet)
        infoGain = baseEntropy - newEntropy
        if (infoGain > bestInfoGain):
            bestInfoGain = infoGain
            bestFeature = i
    return bestFeature
```

```
def majorityCount(classList):
    classCount={}
    for vote in classList:
        if vote not in classCount.keys(): classCount[vote] = 0
        classCount[vote] += 1
    sortedClassCount = sorted(classCount.iteritems(),
        key=operator.itemgetter(1), reverse=True)
    return sortedClassCount[0][0]

def createTree(dataSet,labels):
    classList = [example[-1] for example in dataSet]
    if classList.count(classList[0]) == len(classList):
        return classList[0]
    if len(dataSet[0]) == 1:
        return majorityCount(classList)
    bestFeat = chooseBestFeatureToSplit(dataSet)
    bestFeatLabel = labels[bestFeat]
    myTree = {bestFeatLabel:{}}
    del(labels[bestFeat])
    featValues = [example[bestFeat] for example in dataSet]
    uniqueVals = set(featValues)
    for value in uniqueVals:
        subLabels = labels[:]
        myTree[bestFeatLabel][value] = createTree(splitDataSet\
            (dataSet, bestFeat, value),subLabels)
    return myTree

def testTree(N):
    errorRate = 0
    for i in range(N):
        errorRate += testTreeOnce()
    errorRate = errorRate/float(N)
    return errorRate

def testTreeOnce():
    titlesVocab,charCount,numbers01,question01,binarySuccess,website = allData()
    vocabList = createVocabList(titlesVocab)

    dataSet, featLabels = treePlotterCharacteristics()
    trainingSetSize = int(floor(0.9*len(dataSet)))
    set = range(len(dataSet))
    testDataSet = []; testClasses = []
    for i in range(len(dataSet)-trainingSetSize):
        randIndex = int(random.uniform(0,len(set)))
        testDataSet.append(dataSet[set[randIndex]])
        testClasses.append(dataSet[set[randIndex]][-1])
        del(set[randIndex])
    trainDataSet=[]; trainClasses = []
    for index in set:
```

```
        trainDataSet.append(dataSet[index])
        trainClasses.append(dataSet[index][-1])

    inputTree = createTree(trainDataSet,featLabels)

    index = 0
    errorRate = 0
    for testVec in testDataSet:
        featLabels = ['length','digit','type','source']
        classLabel2 = classify(inputTree,featLabels,testVec)
        if classLabel2 != testClasses[index]:
            errorRate += 1
        index += 1
    errorRate = errorRate/float(index)

    return errorRate

def useTree():
    titles,charCount,numbers01,question01,binarySuccess,website = allData()
    vocabList = createVocabList(titles)
    dataSet,labels = treePlotterCharacteristics()
    inputTree = createTree(dataSet,labels)

    title = str(raw_input("What's the idea for your title?"))

    charCountAll, charCountPopular, charCountUnpopular = charCount2()
    charCount = len(title)
    if charCount < min(charCountPopular,charCountUnpopular):
        length = "S"
    elif charCount > max(charCountPopular,charCountUnpopular):
        length = "L"
    else:
        length = "M"

    titleFormated = title.split()
    titleFormated = str(titleFormated)
    titleFormated = titleFormated.split('\t')
    titleFormated = list(titleFormated)
    for index in range(len(titleFormated)):
        titleFormated[index] = titleFormated[index]\
            .strip('-').strip('.').strip(',').strip('?').strip('!').strip("'")
    titleFormated = [low.lower() for low in titleFormated if len(low) > 2]
    numbersFile = open("D:\\Python27\\MathieuRoy\\numbers.txt")
    numbersList = numbersFile.read()
    numbersList = numbersList.split('\n')
    numbersList = str(numbersList)
    numbersFormated = []
    for index in range(len(numbersList)):
        numbersFormated.extend(numbersList[index].split('\t'))
    numbers = "A"
```

```
    for number in numbersFormated:
        if number in title:
            numbers = "1"
            break

    interrogativeMark = "?"
    exclamatoryMark = "!"
    if interrogativeMark in title:
        phraseType = "?"
    elif exclamatoryMark in title:
        phraseType = "!"
    else:
        phraseType = "..."

    website = str(raw_input("What's the website of your article?"))

    featLabels = ['length','digit','type','source']
    testVec = [length, numbers, phraseType, website]
    print featLabels
    print testVec

    classLabel2 = classify(inputTree,featLabels,testVec)
    return classLabel2

def classify(inputTree,featLabels,testVec):
    classLabel = 0
    firstStr = inputTree.keys()[0]
    secondDict = inputTree[firstStr]
    featIndex = featLabels.index(firstStr)
    for key in secondDict.keys():
        if testVec[featIndex] == key:
            if type(secondDict[key]).__name__=='dict':
                classLabel = classify(secondDict[key],featLabels,testVec)
            else: classLabel = secondDict[key]
    return classLabel

def storeTree(inputTree,filename):
    fw = open(filename,'w')
    pickle.dump(inputTree,fw)
    fw.close()

def grabTree(filename):
    fr = open(filename)
    return pickle.load(fr)



############### TREE PLOTTER ###############

decisionNode = dict(boxstyle="sawtooth", fc="0.8")
```

```
leafNode = dict(boxstyle="round4", fc="0.8")
arrow_args = dict(arrowstyle="<-")

def plotNode(nodeTxt, centerPt, parentPt, nodeType):
    if nodeTxt == "P":
        createPlot.ax1.annotate(nodeTxt, xy=parentPt,
        xycoords='axes fraction',
        xytext=centerPt, textcoords='axes fraction',
        va="center", ha="center", bbox=nodeType, arrowprops=arrow_args,color='b')
    elif nodeTxt == "U":
        createPlot.ax1.annotate(nodeTxt, xy=parentPt,
        xycoords='axes fraction',
        xytext=centerPt, textcoords='axes fraction',
        va="center", ha="center", bbox=nodeType, arrowprops=arrow_args,color='r')
    else:
        createPlot.ax1.annotate(nodeTxt, xy=parentPt,
        xycoords='axes fraction',
        xytext=centerPt, textcoords='axes fraction',
        va="center", ha="center", bbox=nodeType, arrowprops=arrow_args)

#set_color='red'

def createPlot(inTree):
    fig = plt.figure(1, facecolor='white')
    fig.clf()
    axprops = dict(xticks=[], yticks=[])
    createPlot.ax1 = plt.subplot(111, frameon=False, **axprops)
    plotTree.totalW = float(getNumLeafs(inTree))
    plotTree.totalD = float(getTreeDepth(inTree))
    plotTree.xOff = -0.5/plotTree.totalW; plotTree.yOff = 1.0;
    plotTree(inTree, (0.5,1.0), '')
    plt.show()

def getNumLeafs(myTree):
    numLeafs = 0
    firstStr = myTree.keys()[0]
    secondDict = myTree[firstStr]
    for key in secondDict.keys():
        if type(secondDict[key]).__name__=='dict':
            numLeafs += getNumLeafs(secondDict[key])
        else: numLeafs +=1
    return numLeafs

def getTreeDepth(myTree):
    maxDepth = 0
    firstStr = myTree.keys()[0]
    secondDict = myTree[firstStr]
    for key in secondDict.keys():
        if type(secondDict[key]).__name__=='dict':
            thisDepth = 1 + getTreeDepth(secondDict[key])
```

```
        else: thisDepth = 1
        if thisDepth > maxDepth: maxDepth = thisDepth
    return maxDepth

def plotMidText(cntrPt, parentPt, txtString):
    xMid = (parentPt[0]-cntrPt[0])/2.0 + cntrPt[0]
    yMid = (parentPt[1]-cntrPt[1])/2.0 + cntrPt[1]
    createPlot.ax1.text(xMid, yMid, txtString)

def plotTree(myTree, parentPt, nodeTxt):
    numLeafs = getNumLeafs(myTree)
    getTreeDepth(myTree) #
    firstStr = myTree.keys()[0]
    cntrPt = (plotTree.xOff + (1.0 + float(numLeafs))/2.0/plotTree.totalW,\
        plotTree.yOff)
    plotMidText(cntrPt, parentPt, nodeTxt)
    plotNode(firstStr, cntrPt, parentPt, decisionNode)
    secondDict = myTree[firstStr]
    plotTree.yOff = plotTree.yOff - 1.0/plotTree.totalD
    for key in secondDict.keys():
        if type(secondDict[key]).__name__=='dict':
            plotTree(secondDict[key],cntrPt,str(key))
        else:
            plotTree.xOff = plotTree.xOff + 1.0/plotTree.totalW
            plotNode(secondDict[key], (plotTree.xOff, plotTree.yOff),
                cntrPt, leafNode)
            plotMidText((plotTree.xOff, plotTree.yOff), cntrPt, str(key))
    plotTree.yOff = plotTree.yOff + 1.0/plotTree.totalD

#def myTreeTest():
#    return myTreeTest = {'charCount': {'medium': {'phraseType': {'...': {'numbers': {'01':
'Ukn', 'ab': 'Ukn'}}, '???': {'numbers':{'ab': 'Ukn', '01': 'Viral'}}, '!!!': {'numbers':
{'ab': 'Ukn', '01': 'Ukn'}}}}, 'long': {'phraseType': {'...': {'numbers': {'ab': 'Ukn','01':
'Ukn'}}, '???': {'numbers': {'ab': 'Ukn', '01': 'Ukn'}}, '!!!': {'numbers': {'ab':'Ukn', '01':
'Ukn'}}}}}, 'short': {'numbers': {'ab': {'phraseType': {'...': 'Viral', '???': 'Viral', '!!!':
'Ukn'}}, '01': {'phraseType': {'...':'Viral','???':'Ukn','!!!':'Viral'}}}}}}
```

Annexe F : Régression linéaire

```
############### LINEAR REGRESSION ###############
#avec le nombre de mot populaire et la longueur du titre


def regresData(N=0):
    titles,charCount,numbers01,question01,binarySuccess,website = allData()
    wordsFreq = retrieveWordsFreq()

    popularityAvg = []
    for title in titles:
        popularity = 0
        numberOfWords = 0
        for words in wordsFreq:
            if words[0] in title:
                popularity += words[1]
                numberOfWords += 1
        if numberOfWords == 0:
            popularityAvg.append(1)
        else:
            popularityAvg.append(popularity/float(numberOfWords))

    dataMat = []; labelMat = []
    for index in range(len(titles)-N):
        dataMat.append([1.0,float(charCount[index]),float(popularityAvg[index])])
        labelMat.append(int(binarySuccess[index]))
    return dataMat,labelMat

from numpy import *

def sigmoid(inX):
    return 1.0/(1+exp(-inX))

def gradAscent(dataMatIn,labelMatIn):
    dataMat = mat(dataMatIn)
    labelMat = mat(labelMatIn).transpose()
    m,n = shape(dataMat)
    alpha = 0.001
    maxCycles = 500
    weights = ones((n,1))
    for k in range(maxCycles):
        multi = dataMat*weights
        h = sigmoid(multi)
        error = (labelMat - h)
        weights = weights + alpha * dataMat.transpose() * error
    return weights

def plotBestFit(dataMatIn,labelMatIn):
```

```
    import matplotlib.pyplot as plt
    dataMat,labelMat=regresData(dataMatIn,labelMatIn)
    weights = gradAscent() #
    dataArr = array(dataMat)
    n = shape(dataArr)[0]
    xcord1 = []; ycord1 = []
    xcord2 = []; ycord2 = []
    for i in range(n):
        if int(labelMat[i])== 1:
            xcord1.append(dataArr[i,1]); ycord1.append(dataArr[i,2])
        else:
            xcord2.append(dataArr[i,1]); ycord2.append(dataArr[i,2])
    fig = plt.figure()
    ax = fig.add_subplot(111)
    ax.scatter(xcord1, ycord1, s=20, c='red', marker='s')
    ax.scatter(xcord2, ycord2, s=20, c='blue')
    x = arange(0.0, 120.0, 1.0)
    y = (-weights[0]-weights[1]*x)/weights[2]

    ys = str(y)
    ys = list(ys.strip("[").strip("]").split())
    yl = []
    for number in ys:
        yl.append(float(number))

    x = list(x);
    ax.plot(x, yl,c='black')
    plt.xlabel('Longueur'); plt.ylabel('Mots populaires');
    plt.show()

#def stocGradAscent0(dataMatrix, classLabels):
#        weights = [(float(weights[h]) + alpha * error * float(dataMatrix[i][h])) for h in
range(len(weights))]
#        for h in range(len(weights)):
#            weights[h] = weights[h] + alpha * error * dataMatrix[randIndex][h]


def classifyVector(inX, weights):
    prob = sigmoid(sum(inX*weights))
    if prob > 0.5: return 1.0
    else: return 0.0

def testRegresOnce():
    dataMat,labelMat=regresData()
    trainingSet = []; trainingLabels = []
    trainingSetSize = int(floor(0.9*len(dataMat)))
    set = range(len(dataMat))
    testDataMat = []; testClasses = []
    for i in range(len(dataMat)-trainingSetSize):
        randIndex = int(random.uniform(0,len(set)))
```

```
        testDataMat.append(dataMat[set[randIndex]])
        testClasses.append(labelMat[set[randIndex]])
        del(set[randIndex])
    trainDataMat=[]; trainClasses = []
    for index in set:
        trainDataMat.append(dataMat[index])
        trainClasses.append(labelMat[index])
    trainWeights = gradAscent(trainDataMat,trainClasses)

    errorCount = 0; numTestVec = 0.0
    for features in trainDataMat:
        numTestVec += 1.0
        lineArr =[]
        for i in range(3):
            lineArr.append(float(features[i]))
        lineArr = array(lineArr)
        testAnswer = classifyVector(lineArr, trainWeights)
        if int(testAnswer) != int(trainClasses[int(numTestVec)-1]):
            errorCount += 1
    errorRate = (float(errorCount)/numTestVec)
    print "the error rate of this test is: %f" % errorRate
    return errorRate

def testRegres(N):
    numTests = N;
    errorSum=0.0
    for k in range(numTests):
        errorSum += colicTest()
    print "after %d iterations the average error rate is: %f" % (numTests,
errorSum/float(numTests))


def useRegres():
    titles,charCounts,numbers01,question01,binarySuccess,website = allData()

    title = str(raw_input("What's the idea for your title?"))
    charCount = len(title)

    titleFormated = title.split()
    for index in range(len(titleFormated)):
        titleFormated[index] =
titleFormated[index].strip('-').strip('.').strip(',').strip('?').strip('!').strip("'")
    titleFormated = [low.lower() for low in titleFormated if len(low) > 2]

    wordsFreq = retrieveWordsFreq()
    popularity = 0
    numberOfWords = 0
    for words in wordsFreq:
        if words[0] in titleFormated:
            popularity += words[1]
```

```
        numberOfWords += 1
if numberOfWords == 0:
    popularityAvg = 1
else:
    popularityAvg = popularity/float(numberOfWords)

lineArr = [1.0,float(charCount),float(popularityAvg)]
dataMatIn,labelMatIn = regresData()
weights = gradAscent(dataMatIn,labelMatIn)
answer = classifyVector(lineArr, weights)
if answer == 1:
        print "the title will probably be popular"
else:
        print "the title will probably NOT be popular"
```