WILLIAM D'ANJOU 20188213
VANESSA THIBAULT-SOUCY 20126808
20 Décembre 2022

TP2 - 3335

Traitement des données

Comme tout bon programme qui demande l'analyse de données contenue dans un fichier externe, nous avons tout d'abord procédé à la lecture du des données textuelles contenues dans le fichier "data.txt". Ces données proviennent du fichier demandé par l'énoncé du TP, mais pour des raisons pratiques, nous avons décidé de le renommer. La méthode utilisée pour cette étape préliminaire dans le code est rudimentaire et consiste principalement à lire chacune des lignes du fichier, à ignorer les caractères "=" et les lignes qui contiennent deux caractères "$", et à diviser chaque ligne en mots et les traiter selon les cas. Durant cette étape, nous localisons la séquence de caractère correspondant au mot "interest" ou une de ses variantes, comme "interested" et autres. Lorsque le mot est localisé, son index est mémorisé et une nouvelle liste est créée contenant les 2 mots avant et après, s'il existe, incluant le type de ces mots. Nous utiliserons plus tard ces deux caractéristiques afin d'entraîner nos différents modèles. De plus, nous plaçons le sens du mot "interested" en première position de la liste afin d'y avoir accès facilement pour l'entraînement et la validation de nos modèles.

Ensuite, nous utilisons la classe "CountVectorizer" de la librairie scikit-learn pour transformer les listes contenant nos informations en vecteurs de caractéristiques, avec une fonction de tokenization qui divise les mots en tokens. Les vecteurs sont convertis en tableaux numpy et stockés dans la variable "X". Le code enregistre également les étiquettes dans une liste "na".
Nous effectuons, ensuite, une division de notre jeu de données afin d'obtenir un jeu d'entraînement et un jeu de test avec lequel nous évaluerons nos modèles.

Nos classifieurs

En ce qui concerne nos modèles de classification sur le jeu de données d'entraînement: Naive Bayes (à la fois gaussien et multinomial), un classificateur de forêts aléatoires, un réseau de neurones multi-couches (MLP) et une SVM linéaire. Pour chaque modèle, il mesure la performance en utilisant l'accuracy sur le jeu de données de test.
Enfin, le code effectue des comparaisons et des analyses sur les résultats obtenus avec chaque modèle. Il compare la performance des différents algorithmes en termes d'accuracy, et analyse l'impact de différentes options sur la performance, telles que le stemming, le nombre de neurones cachés et la taille de fenêtre pour la désambiguïsation.

Méthode d'évaluation / Évaluation des performances

Afin d'évaluer les performances de nos différents algorithmes, nous avons choisi différentes méthodes selon l'algorithme employé. Lorsque nous mentionnons faire usage de différentes méthodes d'échantillonnage, nous référons à utiliser respectivement 20%, 50%, et 80% de nos échantillons de base afin de s'en servir comme échantillons de test, le reste servant alors à l'entraînement de nos modèles.

- Naïve-Bayes

  Le classifieur naïf de Bayes est un modèle de classification probabiliste qui utilise le théorème de bayes pour prédire le sens du mots "interest" d'un échantillons en utilisant des information sur les probabilités de chaque classes et les probabilités conditionnelles des caractéristiques de l'échantillon données. Le classifieur de bayes suppose que les caractéristiques des mots sont indépendantes les unes des autres. Ceci est une hypothèse simplificatrice mais qui peut donner de bon résultats dépendamment des situations. Pour entraîner l'algorithme, nous lui avons fourni un jeu de données annotées, pour lesquelles il peut calculer les probabilité de chaque sens possible, et ensuite chaque sens sachant les caractéristique observée. L'algorithme peut ensuite appliquer les probabilités qu'il a appris sur un jeu de test afin d'évaluer les performances de cet algorithme. Pour ce faire, nous avons simplement fait rouler l'algorithme sur nos données et nous comparons par la suite le nombre de résultats obtenus avec les résultats espérés. Puisque l'appel à cet algorithme ne possède pas de paramètre permettant une approche personnalisée, nous nous contentons d'appliquer l'algorithme tel quel aux données brutes. Cependant, nous avons fait rouler l'algorithme un certain nombre de fois sur nos échantillons afin de calculer une certaine moyenne de rendement. De plus, puisqu'il n'y a pas de paramètre à changer, nous avons simplement utilisé différents pourcentage d'échantillons de

tests sur l'algorithme afin de voir les différentes performances selon la division de notre échantillon. Nous avons donc pu observer des cas de sous-apprentissage, ainsi que de sur-apprentissage, selon les résultats obtenus par la courbe de validation.

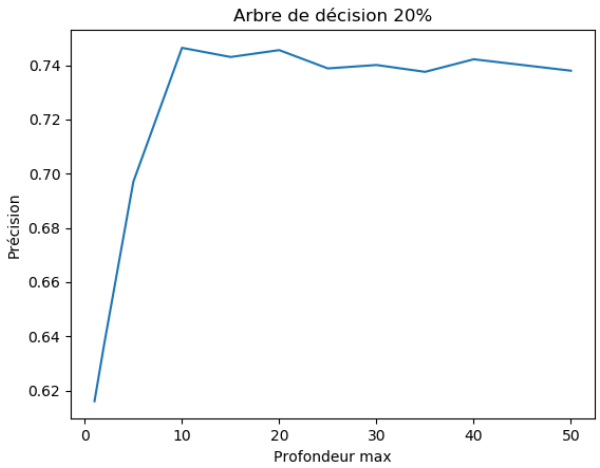
Résultats		
20%	50%	80%
0.41350	0.44764	0.41003

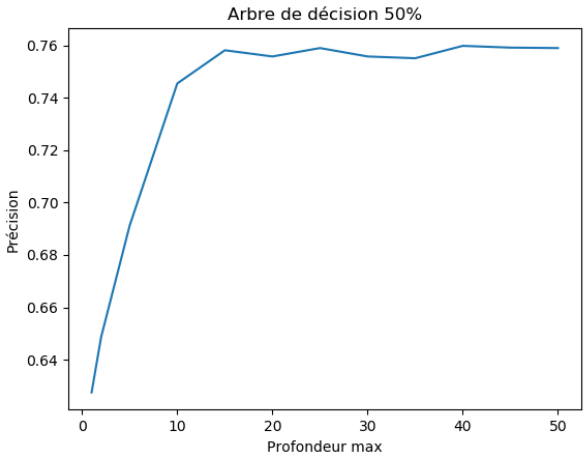
À la vue de ces résultats, on peut constater qu'utiliser 50% de nos données en tant qu'échantillon de tests donne de meilleurs résultats par rapport aux 20% et 80%. L'un montrant très probablement un sous-entraînement et l'autre un sur-entraînement. Cela dit, malgré le fait que 50% donne le meilleur résultat, il faut tout de même réaliser que sa performance n'est pas très bonne. En effet, on a un taux de succès inférieur à 50% ce qui montre clairement que le choix de Naïve-Bayes pour notre problème n'est peut-être pas la bonne approche à prendre pour le résoudre.

- Arbre de décision

  Un arbre de décision est un modèle de classification qui permet de prédire la classe d'un échantillon en se basant sur une série de tests sur les caractéristiques de l'échantillon. L'arbre de décision est construit en commençant par une racine qui représente toutes les données d'entraînement, puis en divisant récursivement les données en sous-ensembles plus petits en fonction de tests sur les caractéristiques de chaque mot entourant « interest ». Chaque nœud de l'arbre représente un test sur une caractéristique et chaque branche représente la réponse au test (par exemple, "caractéristique A = NN"). Les feuilles de l'arbre représentent les classes prédits pour chaque sous-ensemble de données. Pour l'entraînement, l'algorithme utilise un jeu de données d'entraînement annoté avec le sens obtenu et évalue le poids de chaque caractéristique afin de déterminer lequel prendre comme racine. En construisant l'arbre de décision, l'algorithme divise les données de façon récursive selon les caractéristiques rencontrées. Une fois l'arbre de décision entraîné, nous pouvons ensuite vérifier sa performance avec un jeu de tests. Afin d'évaluer cette performance d'un arbre de décision, nous avons choisi de jouer le paramètre de

la profondeur maximal que l'appel de l'algorithme pouvait avoir, pour constater les différents résultats possibles. Nous comparons par la suite les résultats obtenus avec ceux espérés pour en tirer un taux de succès, soit la performance de cet algorithme selon les paramètres choisis. Nous avons également évalué la différence de performance selon différents formats d'échantillons pour nos données d'entraînement et de test afin de voir si les résultats varient. Nous avons ensuite comparé les graphiques obtenus par l'algorithme et avons choisi nos paramètres en fonction des meilleurs résultats, tout en tenant compte des risques de sur et sous apprentissage.





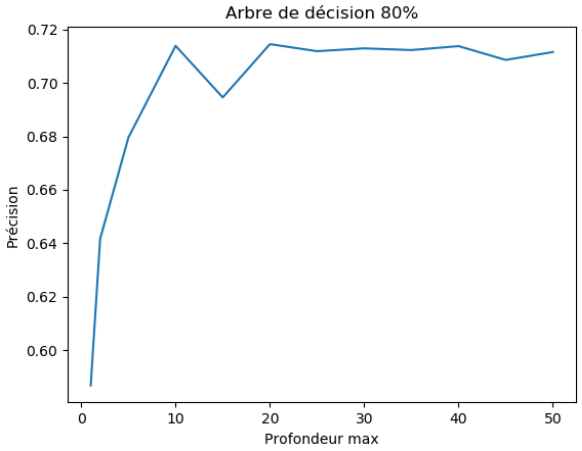


Tableau des résultats

Arbre de décision 50%	
1	0.61603
2	0.63713
5	0.69705
10	0.74641
15	0.74304
20	0.74557
25	0.73882
30	0.74008
35	0.73755
40	0.74219
45	0.74008
50	0.73797

Arbre de décision 20%	
1	0.62753
2	0.64865
5	0.69122
10	0.74544
15	0.75811
20	0.75574
25	0.75895
30	0.75574
35	0.75507
40	0.75980
45	0.75912
50	0.75895

Arbre de décision 80%	
1	0.58681
2	0.64169
5	0.67958
10	0.71388
15	0.69456
20	0.71451
25	0.71187
30	0.71293
35	0.71230
40	0.71377
45	0.70860
50	0.71156

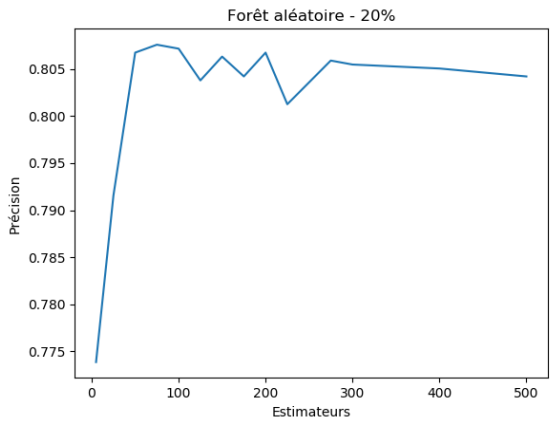
La précision d'un arbre de décision avoisine le 75% ce qui est nettement supérieur à notre tentative de résoudre le problème avec Naïve-Bayes. On remarque rapidement qu'on n'obtient pas de meilleure performance au-delà de 20 comme profondeur maximale à notre arbre. Ce constat peut également être fait peu importe la proportion de nos échantillons en tant que "test" ou bien que résultat espéré. La performance augmente cependant très rapidement lorsque notre profondeur est très petite, comme quoi notre arbre a besoin de suffisamment de choix afin de nous donner un résultat avec une certaine certitude.

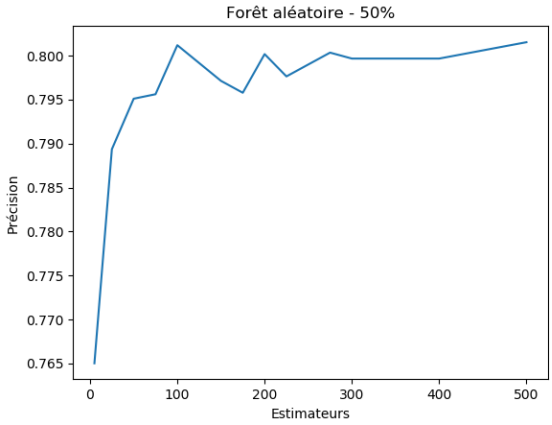
- Forêt aléatoire

  Une forêt aléatoire est un modèle de classification qui consiste en un ensemble d'arbres de décision construits de façon indépendante et entraînés sur des sous-ensembles aléatoires sur l'ensemble des données d'entraînement. Pour la phase d'entraînement du modèle, on tire aléatoirement un sous-ensemble de

données d'entraînement (avec remplacement) et on l'utilise pour entraîner un arbre de décision parmi les arbres de la forêt. Ainsi, chaque arbre est entraîné sur une version différente des données d'entraînement, ce qui permet à la forêt aléatoire de capturer une variété de combinaisons de caractéristiques des données, ce qui permet de ralentir l'apparition de sur-apprentissage sur les données. Une fois la forêt aléatoire entraînée, le modèle peut ensuite être utilisé pour prédire le sens du mot "interest" dans un échantillon donné en utilisant chacun des arbres de la forêt pour faire une prédiction et puis faire la moyenne des prédictions de tous les arbres pour en obtenir une prédiction finale du sens.

Pour évaluer notre forêt aléatoire, nous avons modifier le nombre d'estimateurs afin de choisir la quantité optimale pour notre problème. On peut voir le nombre d'estimateurs comme le nombre d'arbres présents dans notre forêt. En plus de faire l'essai de différents nombre d'arbres, nous avons également testé l'algorithme avec des échantillons de tailles différentes afin d'observer les différences de performances entre chacun de nos essais.





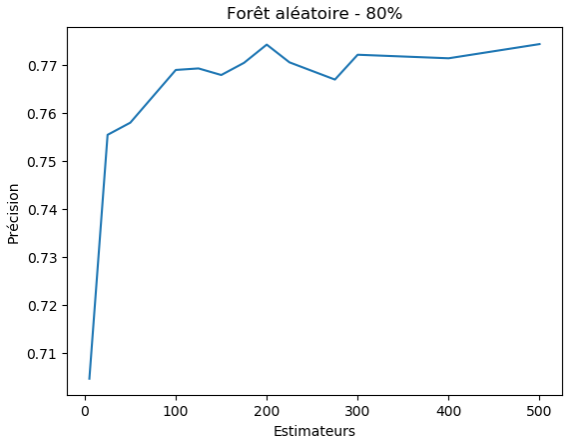


Tableau des résultats

Forêt aléatoire 20%	
5	0.77384
25	0.79156
50	0.80675
100	0.80717
125	0.80380
150	0.80633
175	0.80422
200	0.80675
225	0.80127
275	0.80591
300	0.80549
400	0.80506
500	0.80422

Forêt aléatoire 50%	
5	0.76503
25	0.78936
50	0.79510
100	0.80118
125	0.79916
150	0.79713
175	0.79578
200	0.80017
225	0.79764
275	0.80034
300	0.79966
400	0.79966
500	0.80152

Forêt aléatoire 80%	
5	0.70470
25	0.75546
50	0.75799
100	0.76897
125	0.76929
150	0.76792
175	0.77045
200	0.77425
225	0.77055
275	0.76697
300	0.77214
400	0.77140
500	0.77435

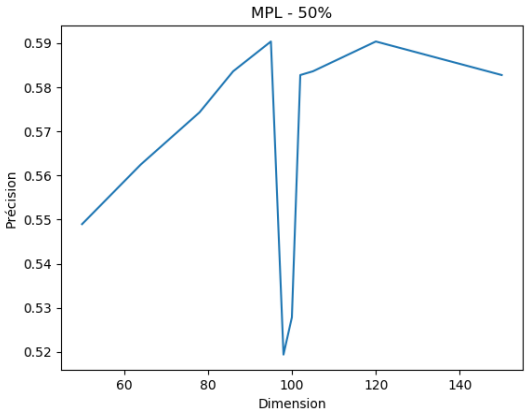
- SVM

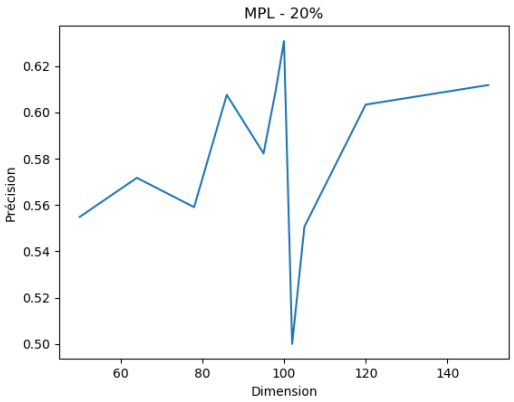
  Un support vector machine (SVM) est un modèle de classification qui utilise l'analyse des données pour trouver un hyperplan de séparation entre les différents sens dans l'espace des caractéristiques. L'objectif d'un SVM est de trouver un hyperplan qui sépare le mieux les différents sens possible pour le mot “interest” tout en maximisant la marge de séparation entre elles. Tout comme le classificateur *Naïve-Bayes* de sklearn, le SVM ne possède pas vraiment de paramètres personnalisables intéressants pour faire différentes analyses plus poussées. Nous nous sommes donc arrêtés sur la simple différence entre la génération de nos échantillons afin d’observer les différences de performances.

Résultats		
20%	50%	80%
0.64979	0.64611	0.61741

- MLP

  Un perceptron multicouche (aussi connu sous le nom de réseau de neurones à propagation avant) est un modèle de classification qui utilise une série de couches de neurones pour apprendre à prédire une classe à partir de caractéristiques données. Un perceptron multicouche est un type de réseau de neurones qui est constitué de plusieurs couches de neurones : une couche d'entrée, une ou plusieurs couches cachées et une couche de sortie. Chaque couche est constituée de plusieurs neurones, qui sont connectés aux neurones de la couche suivante par des poids. Le perceptron multicouche apprend en ajustant ces poids durant l'entraînement afin de minimiser l'erreur de test. Afin de trouver la configuration optimale pour notre MLP, nous avons choisi d’essayer différentes configuration de neurones lors de l’appel de l’algorithme. Par soucis de temps, nous avons sélectionné des configurations “carrées” afin que ce soit plus rapide à générer, sachant que notre problème ne nécessitait pas un type de perceptron plus complexe. Cette décision peut avoir des conséquences sur nos résultats, mais notre compréhension de l’algorithme de sklearn ne nous permet pas de faire une analyse plus détaillée de nos résultats. Nous avons également testé ces configurations avec des échantillons de tailles différentes par rapport aux échantillons de tests, ce qui nous a donné des résultats particuliers avec du sur apprentissage et du sous-apprentissage dépendamment des expériences.


MPL - 50%
Précision
Dimension
0.59
0.58
0.57
0.56
0.55
0.54
0.53
0.52
60
80
100
120
140


MPL - 20%
Précision
Dimension
0.62
0.60
0.58
0.56
0.54
0.52
0.50
60
80
100
120
140

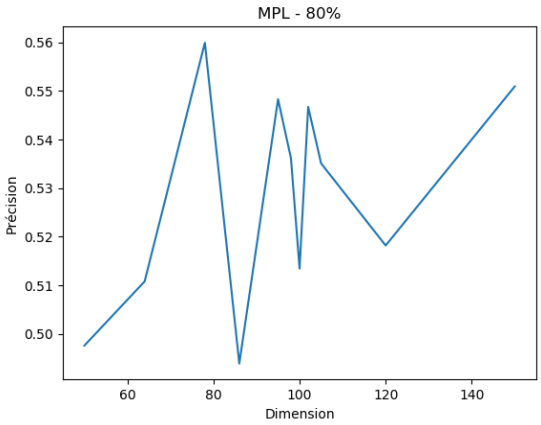

MPL - 80%
Précision
Dimension
0.56
0.55
0.54
0.53
0.52
0.51
0.50
60
80
100
120
140

Tableau des résultats

MLP 20%	
50	0.55485
64	0.57173
78	0.55907
86	0.60759
95	0.58228
98	0.60970
100	0.63080
102	0.50000
105	0.55063
120	0.60338
150	0.61181

MLP 50%	
50	0.54899
64	0.5625
78	0.57432
86	0.58361
95	0.59037
98	0.51943
100	0.52787
102	0.58277
105	0.58361
120	0.59037
150	0.58277

MLP 80%	
50	0.49763
64	0.51082
78	0.55989
86	0.49393
95	0.54828
98	0.53615
100	0.51346
102	0.5467
105	0.53509
120	0.51821
150	0.55092