UNIVERSITÉ DE MONTPELLIER

PROJET D'ECD

Classification de documents par opinion

Auteurs : Elsa Martel, Axel Depret, Niels Benichou

11 avril 2016

Table des matières

Introduction

Tout d'abord, un jeu de données textuelles nous a été donné. Il s'agit d'un corpus d'à peu près 2000 documents contenant des avis d'internautes sur des films. Chaque document est associé à sa polarité selon l'avis (+1 : positif et -1 : négatif).

Le but de ce projet étant d'extraire des modèles à partir des algorithmes de classification de documents par opinion suivant : NaivesBayes, J48, IBk et SMO.

Nous avons dû suivre plusieurs étapes :

- Faire un prétraitement sur les données,
- Réfléchir aux configurations à appliquer sur les prétraitements,
- Analyser les résultats obtenus.

I. Prétraitements

1. Partie données brutes

a) Lecture du CSV

La lecture des deux fichiers CSV se fait grâce à la méthode lecture qui prend en paramètre une chaîne qui permet de déterminer quel type de traitement l'utilisateur souhaite obtenir. Dans le cas du traitement brut, ce paramètre est b.
Le corps de la méthode lecture se décompose en quatre parties.

Premièrement, deux String permettent de paramétrer les PATH dans le répertoire `"./elements_projet/"` concaténé au nom des fichiers .csv, respectivement dataset.csv et labels.csv .
Puis la String resultat reçoit une en-tête arff `"@RELATION donnees\n@ATTRIBUTE text STRING\n@ATTRIBUTE eval {-1,1}\n@data"` grâce à la méthode header qui ne fait rien d'autre que de renvoyer cette chaîne.

Deuxièmement, les fichiers .csv sont ouverts en lecture puis envoyés dans leurs buffers respectifs. Voici le code pour l'un d'entre eux :

```
InputStream ipsData = new FileInputStream(data);
InputStreamReader ipsrData = new InputStreamReader(ipsData);
BufferedReader brData = new BufferedReader(ipsrData);
```

Troisièmement, en fonction de l'option choisie un traitement va s'effectuer. Dans le cas présent, nous voulons juste le texte brut. Ce traitement est simple, tant qu'il y a des lignes à lire dans les deux fichiers .csv on concatène la ligne de data avec celle de label dans la String `resultat`. A noter cependant qu'un échappement des doubles quotes est effectué pour éviter tout problèmes ou incohérences liés à ces dernières (par exemple un nombre impair de quotes).

```
resultat += "\n\"" + echappementQuotes(ligneData) + "\"," + ligneLabel;
```

Quatrièmement, les buffers sont fermés et on renvoie la String resultat.

b) Construction du fichier ARFF

Dans cette partie, une String est créée avec le PATH du répertoire de sortie des fichiers, `"./fichiers_arff/"`, concaténé au nom du fichier `"Bruts.arff"` dans notre cas. Enfin, on appelle la méthode `ecriture` qui prend en paramètres le contenu du fichier précédemment créé en I.1.a et le nom du fichier que l'on vient de créer dans cette partie. La méthode `ecriture` se compose ainsi :

- Création et ouverture en écriture du fichier
- Création du buffer du fichier
- Création d'un `PrintWriter` sur le buffer
- Écriture du contenu dans le buffer grâce à la méthode `print` du `PrintWriter`
- Fermeture du fichier

Le fichier est alors prêt à être utilisé.

NB : Pour les 3 autres parties, nous avons effectué les prétraitements de cette partie. En outre seule le I.1.a troisièmement change pour le traitement.

2. Partie données traitées

Dans cette partie, il s'agit de présenter les différents types de traitements afin d'éliminer le bruit (ponctuation et stopwords) ou bien de remplacer certaines chaines de caractères pour leur donner du sens (smileys).

a) Ponctuation

Les caractères de ponctuation ont en général pour but de structurer une phrase pour la rendre plus lisible par l'homme, mais pour la machine ils sont source d'erreurs. Même si on pourrait se servir par exemple du ! pour amplifier le sens de la phrase, on préfère les effacer pour éviter les erreurs. De plus, un grand nombre d'autres caractères de ponctuation ne servent à rien comme #, @ …
La liste des caractères est définie comme ceci :

```
public String[] ponctuation = { ",", ".", "!", "[", "]", "-", "?", "_",
"/", "*", "@", "(", ")", ":", "&", ";", "#", "<", ">" };
```

Ensuite lors du traitement, en plus de la méthode `echappementQuotes` on appelle la méthode `traitementPonctuation`. Cette méthode efface chaque caractère qui match avec la String ponctuation en le remplaçant par un espace.

```
Resultat += "\n\""+ echappementQuotes(traitementPonctuation(ligneData))
+ "\"," + ligneLabel;
```

b) Smiley

Les smileys n'ont pas de valeur sémantique pour Weka, il faut donc les remplacer par un mot qui s'approche au mieux de ce qu'a voulu exprimer l'auteur du texte.
Le problème avec les smileys c'est que s'ils comportent des lettres comme ";D" cela peut paraître anodin, mais il ne faut pas confondre " ;D " et " ;De plus " … Dans le deuxième cas, ce n'est pas un smiley mais juste un ; qui n'a pas été espacé du mot suivant.
Pour pallier au problème, nous avons décidé d'utiliser deux hashtables, une pour les mots ne finissant pas par une lettre et une pour les autres… Voici une partie du code :

```
smileySansLettre = new Hashtable();
smileyAvecLettre = new Hashtable();
smileySansLettre.put(":)", "happy");
smileySansLettre.put(":-)", "happy");
smileySansLettre.put("q:)", "happy");
smileySansLettre.put(":(", "sad");
smileySansLettre.put(":-(", "sad");
smileyAvecLettre.put(":D ", "cheerful");
smileyAvecLettre.put(":-D", "cheerful");
smileySansLettre.put(":o)", "happy");
smileySansLettre.put(";)", "happy");
smileySansLettre.put(";-)", "happy");
```

Enfin, on ajoute au traitement I.2.a la méthode `traitementSmiley` qui, pour la ligne passée en paramètre va changer tous les smileys présents dans la hashtable par leurs valeurs dans la hashtable.

```
resultat+="\n\""+echappementQuotes(traitementPonctuation(
   traitementSmiley(ligneData))) + "\","+ ligneLabel;
```

c) Stop list

La stoplist nous permet de supprimer des termes qui n'auront pas d'incidence sur la classification et qui sont également qualifiés de "bruit". En effet, des termes comme "among", "before", "a" et "if" n'apportant pas d'informations sur la polarité, il faut les retirer afin de raffiner les données à traiter.
Nous nous sommes basés sur une liste prise sur le web et nous l'avons personnalisée afin d'y retirer certains termes qui ont une conséquence sur la classification. Nous avons ciblé principalement les mots issus de la négation, de la comparaison et des conjonctions.
Le traitement suit le même schéma que les précédents à ceci près qu'une nouvelle méthode est appelée : `stoplist`, avant d'entrer dans la boucle de traitement. Cette méthode va ouvrir le fichier des stop words et les envoie dans la liste `listeStop`.
Enfin, on ajoute la méthode `traitementStopList` au traitement. Celle-ci convertit la ligne passée en paramètre en lowerCase puis efface tous les mots qui concordent avec la liste créée par la méthode `stoplist`.

```
resultat+="\n\"" +echappementQuotes(traitementStopList(traitementPonctu
ation(traitementSmiley(ligneData))+ "\"," + ligneLabel;
```

3. Partie données lemmatisées

Pour cette partie, nous avons utilisé les fichiers arff créés dans la partie précédente :

- Traitement des smileys et ponctuation
- Traitement des smileys, ponctuation et stop list.

Nous avons ensuite appliqué le stemmer LovinsStemmer sur les 2 fichiers arff chargés dans Weka. Cette option est à rajouter aux configurations. Celle-ci permet de ramener le mot à sa racine.
Par exemple, les mots "am ", " are " et " is " vont être remplacés par "be". Les mots "car", "cars", "car's " et "cars'" vont être transformés en "car".

Le but de la lemmatisation et de la racinisation (stemming) est de ramener le mot dans sa forme de base. Mais il y a une différence entre les 2 principes.

La racinisation (stemming) va couper la fin du mot dans l'espoir d'atteindre cet objectif correctement ; la plupart du temps avec succès, et inclut souvent la suppression des affixes dérivationnels.

La lemmatisation va faire les choses correctement avec l'utilisation d'un vocabulaire, de l'analyse morphologique des mots pour supprimer les terminaisons et pour retourner la base qui est connue comme le lemme.

4. Partie données lemmatisées avec analyse morphosyntaxique

Cette dernière partie se déroule en plusieurs étapes :

- **1ère étape :** Création de deux fichiers CSV (un avec traitement des smileys et de la ponctuation, l'autre avec les mêmes traitements auquel on ajoute le traitement des stopwords) à partir du fichier CSV de base contenant les commentaires.
- **2ème étape :** Création des fichiers correspondants aux commentaires et résultants d'un script shell dans lequel on applique l'outil TreeTagger.
- **3ème étape :** Création des fichiers arff à partir des fichiers précédents où les données sont les lemmes associés aux termes d'origine des commentaires.

a) Etape 1

Pour ce faire, on a élaboré un script java (voir code) et deux méthodes (`lectureEtTraitement` et `ecriture`).
Selon le choix de l'utilisateur (avec ou sans traitement des stopwords), on applique la fonction `lectureEtTraitement` qui va lire le contenu du fichier dataset.csv, effectuer les traitements équivalents à la demande de l'utilisateur et retourner le contenu traité afin qu'il soit écrit dans un nouveau fichier csv (dataset_for_lemmatisation_smileys_ponctuation.csv ou dataset_for_lemmatisation_smileys_ponctuation_stopwords.csv) à l'aide de la méthode `ecriture`.

b) Etape 2

Pour cette étape, une fois les fichiers CSV traités obtenus, nous avons développé un script shell (cf annexe 1) pour appliquer le traitement TreeTagger.
En fonction de la demande de l'utilisateur (lemmatisation avec ou sans les stopwords), on lit le fichier CSV ligne par ligne. Chaque ligne est traitée avec l'outil de lemmatisation TreeTagger et le résultat est redirigé dans un fichier qui est le résultat de la lemmatisation du commentaire (cf annexe 2).

c) Etape 3

Dans cette dernière étape, on va utiliser notre script java de la 1ère étape et la méthode `creationFichierArffLemmatisation`.
On utilise à l'intérieur la méthode `creationContenuArffLemmatisation` afin de créer le contenu du fichier arff avec d'abord l'en-tête puis pour les données, on récupère l'ensemble des fichiers résultants de l'étape 2 et l'on récupère la partie lemme de chaque fichier à l'aide de la méthode `lemmatisationChaine`. Cela nous forme une chaîne de lemmes pour chaque fichier/commentaire. On associe chaque chaîne avec le label d'évaluation (-1,1) comme dans l'étape de création de fichiers arff. On écrit enfin le contenu dans un fichier arff.

d) TreeTagger

TreeTagger nous permet d'effectuer l'étiquetage morphosyntaxique en identifiant pour chaque terme la classe à laquelle il appartient (catégorie grammaticale, genre, nombre, temps…) à partir de son contexte et de donner sa forme canonique (cf annexe 2). L'outil fonctionne sur la base des arbres à décisions où pour un terme, on analyse les termes antérieurs afin de déterminer la classe morphosyntaxique du terme courant.

e) Filtres

Nous avons divisé en deux cas la partie lemmatisation avec analyse morphosyntaxique en appliquant des filtres afin de voir si cela a un impact sur la classification des données.
Pour le premier cas, nous avons appliqué des filtres simples afin de ne pas prendre la ponctuation restante et des termes dont TreeTagger n'arrive pas à trouver le lemme.

Cas de lemmes non trouvés par TreeTagger :	@card@ et <unknown>
Ponctuation :	" et '

TreeTagger a également du mal à mettre sous forme de lemme les contractions négation en anglais. Par exemple :

Terme d'origine	**Traitement TreeTagger**
wouldn't	would MD would n't RB n't

Cela va nous poser problème pour la classification où nous aurons deux formes de négation (not et n't). Pour rectifier ce problème, nous avons remplacé tous les n't par not afin d'unifier la négation en un seul terme.

Concernant le second cas, nous passons par des filtrages de catégories des termes en plus des filtres énoncés précédemment. Le but étant de garder les termes qui sont soit des adjectifs, des verbes ou bien des adverbes. En effet, ces catégories ont en général un poids sémantique conséquent sur la polarité d'une phrase que ce soit les adjectifs (avec bad, stupid, amazing, etc…), les adverbes (d'infériorité, de supériorité ou bien d'égalité) et enfin les verbes (like, love, hate, etc…).
Nous avons donc regroupé les tags qu'utilise TreeTagger pour définir ces catégories et nous avons récupéré les lemmes correspondants dès que le tag était inclus dans l'ensemble défini.

Adjectifs	JJ JJR JJS
Adverbes	RB RBR RBS RP WRB
Verbes	VB VBD VBG VBN VBP VBZ

Dans chacun des deux cas précédents et par rapport aux étapes décrites, nous obtenons deux fichiers arff :

- Un fichier ayant subi les traitements ponctuation et smileys
- Un fichier ayant subi les traitements ponctuation et smileys et stopwords

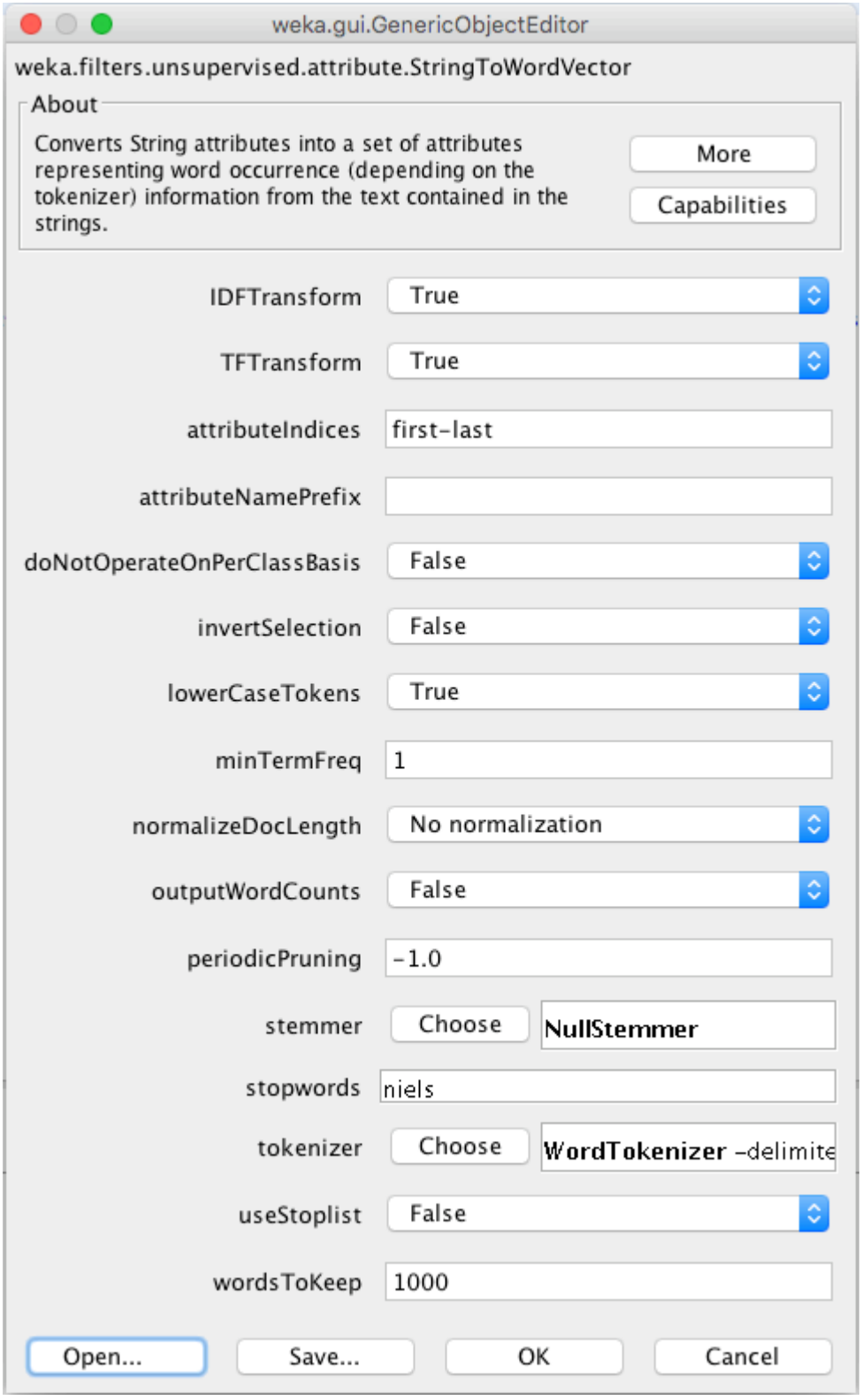
II. Configurations

Pour les quatre configurations, nous utilisons IDFTransform et TFTransform pour évaluer l'importance d'un terme dans le corpus.
IDF est la fréquence d'un mot dans le corpus. Il diminue le poids des mots qui apparaissent souvent dans le corpus.
TF est la fréquence d'un mot dans une phrase.

Nous utilisons l'option lowerCaseToken pour uniformiser le texte et que l'analyse ne considère pas qu'un mot avec une majuscule soit différent de ce mot sans majuscule.

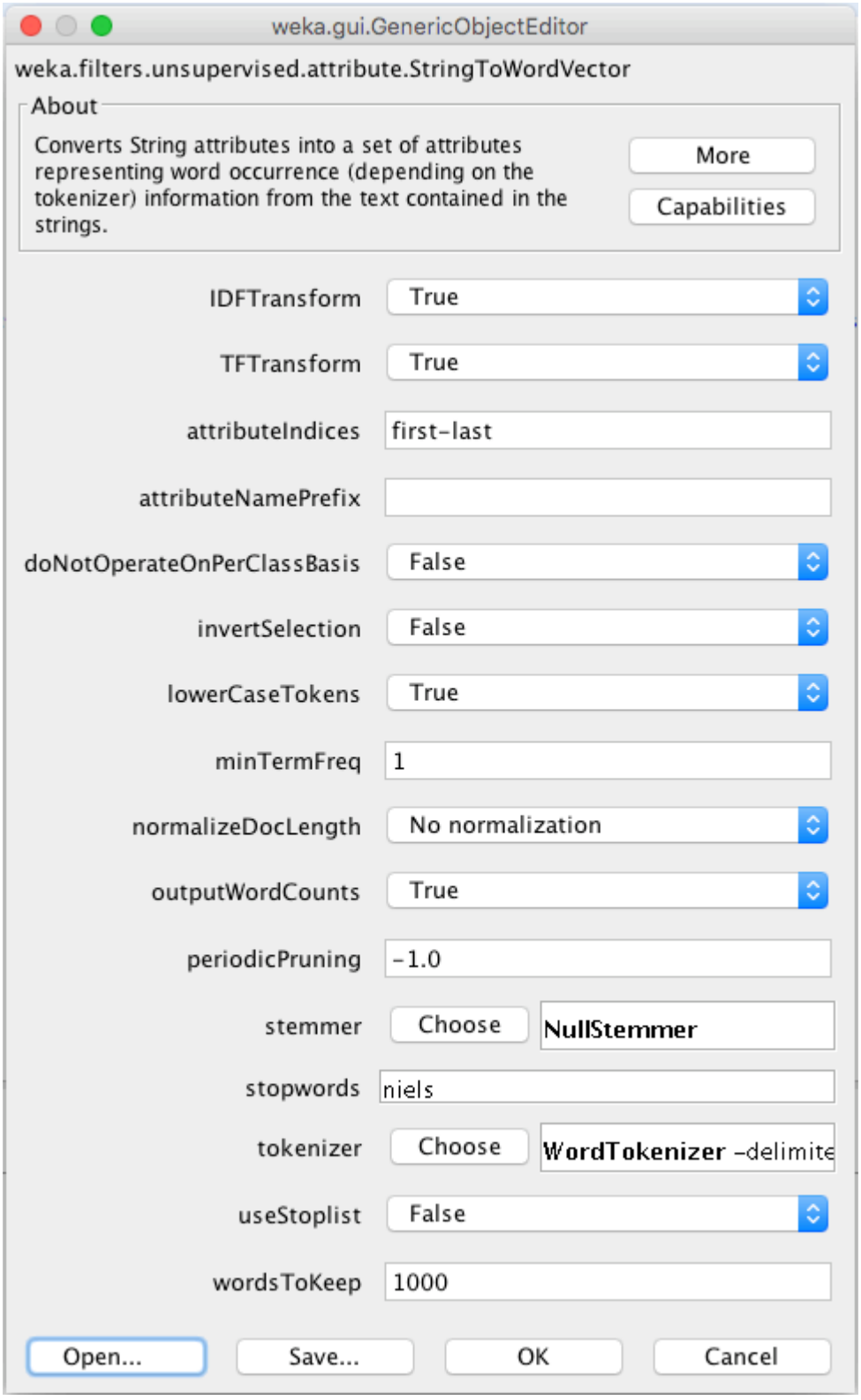
1. Configuration 1

BINAIRE (sans N-gram outputWordCount = false)



2. Configuration 2

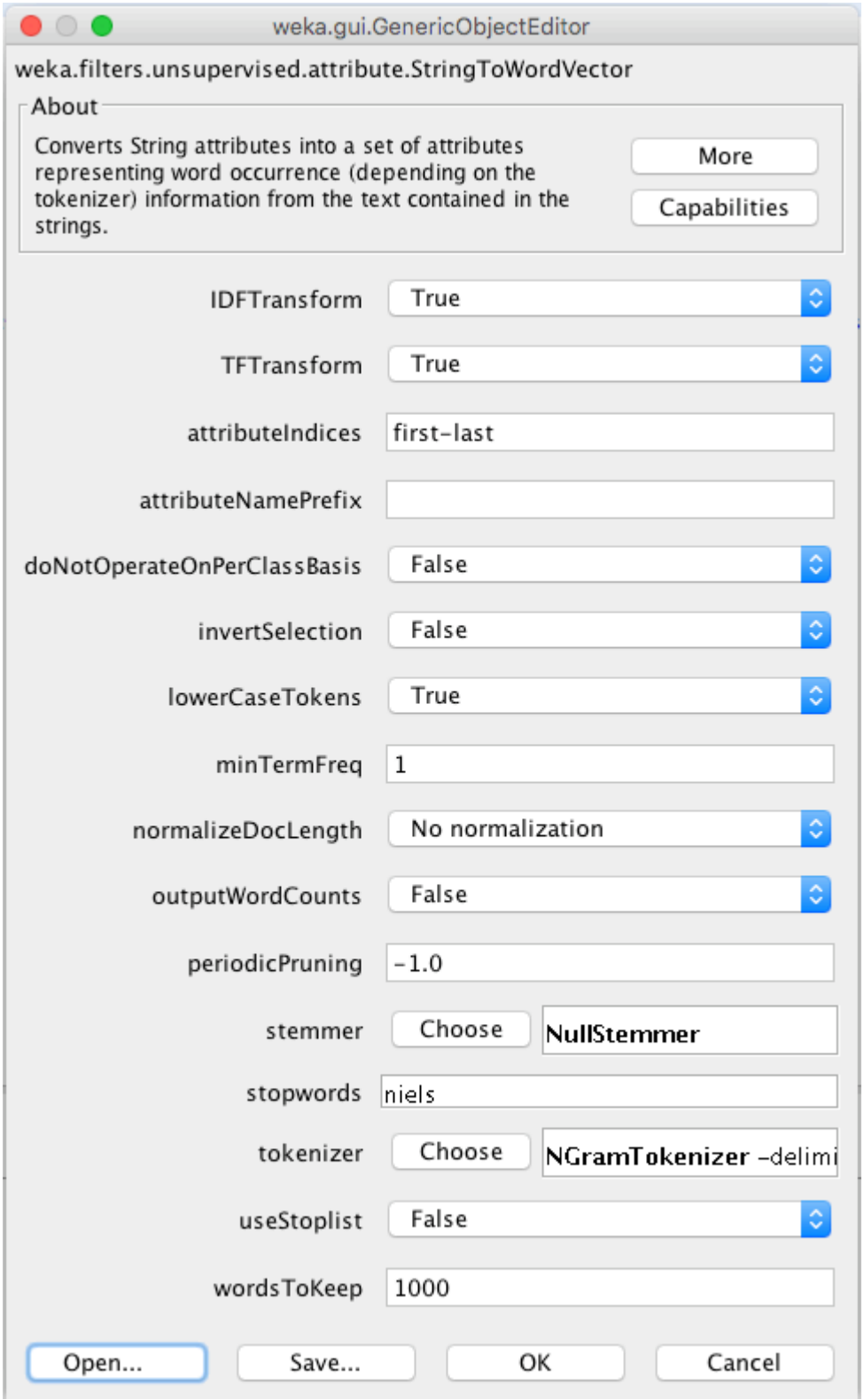
FREQUENTIELLE (sans N-gram outputWordCount = true)



3. Configuration 3

BINAIRE (avec N-gram outputWordCount = false)

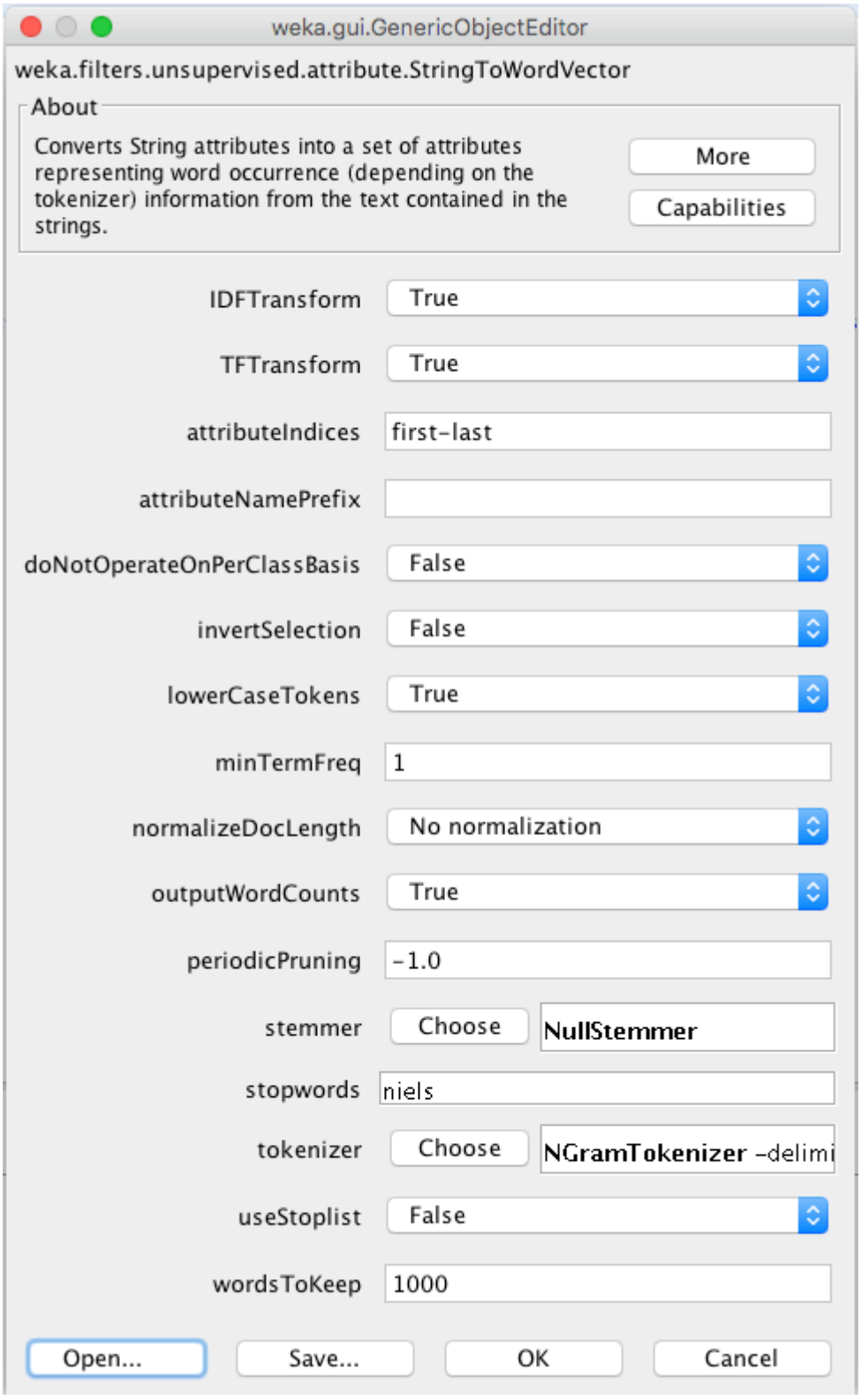
Pour cette configuration nous avons ajouté les N-Grams, nous avons des uni-grams, des bi-grams et des tri-grams.



4. Configuration 4

FREQUENTIELLE (avec N-gram outputWordCount = true)

Nous avons comme pour la configuration 3 des n-grams.



III. Présentation des algorithmes

1. Arbres de décisions J48

L'algorithme J48 est à base d'arbre de décision permettant une fonction de classement représentée par cet arbre en partant de la racine et en allant vers les feuilles.
Les nœuds internes d'un arbre de décision représentent les différents attributs, les branches entre les nœuds indiquent les valeurs possibles que ces attributs peuvent avoir dans les échantillons observés, alors que les nœuds terminaux ou feuilles indiquent le résultat final de classification de la variable dépendante.

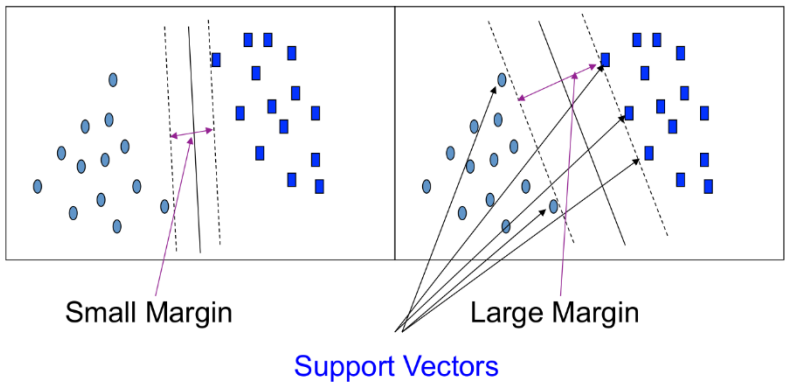
2. Naïve Bayes

L'algorithme de Naïve Bayes utilise la théorie des probabilités pour classifier les données.

Cette méthode de classification est basée sur le théorème de Bayes avec une forte indépendance des hypothèses :

$$P(C \mid A_1 A_2 \mathbf{K}\ A_n) = \frac{P(A_1 A_2 \mathbf{K}\ A_n \mid C)P(C)}{P(A_1 A_2 \mathbf{K}\ A_n)}$$

3. Machines à support de vecteurs SMO

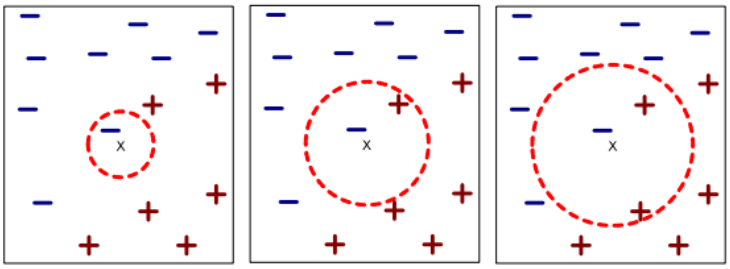


Les SVM (Support Vector Machine) sont des classificateurs qui reposent sur deux idées clés :

- La notion de marge maximale. La marge est la distance entre la frontière de séparation et les échantillons les plus proches. Ces derniers sont appelés vecteurs supports. Dans les SVM, la frontière de séparation est choisie comme celle qui maximise la marge. Le problème est de trouver la frontière séparatrice optimale, à partir d'un ensemble d'apprentissage.
- Transformer l'espace de représentation des données d'entrées en un espace de plus grande dimension, dans lequel il est probable qu'il existe une ligne séparatrice linéaire.

L'algorithme SMO (Sequential Minimal Optimisation) nous permet d'implémenter ces deux concepts.

4. K plus proches voisins



(a) 1-nearest neighbor (b) 2-nearest neighbor (c) 3-nearest neighbor

La recherche des K plus proches voisins nécessite trois conditions :

- L'ensemble des textes,
- La mesure de distance pour calculer la distance entre les textes,
- Une valeur k qui est le nombre de plus proches voisins à retrouver.

Pour classifier les textes non catégorisés :

- Calcul de la distance par rapport aux autres textes,
- Identifier les k plus proches voisins,
- Utiliser les noms des catégories des textes les plus proches pour déterminer la catégorie des textes non catégorisés.

IV. Analyse des résultats

Les graphes ci-dessous représentent le pourcentage de données correctement classifiées.

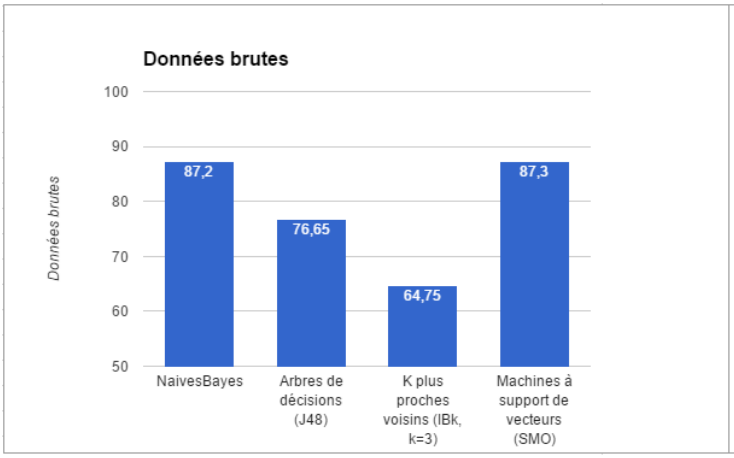
Dans les graphes, la légende "avec stop words" veut dire que nous enlevons les stop words du fichier arff créé.
La légende "sans stop words" veut dire que nous gardons les stop words.

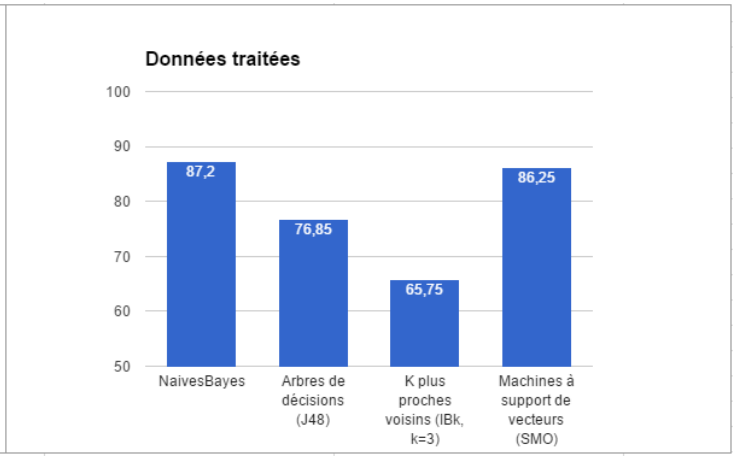
1. Binaire

a) Sans n-gram

➢ Comparaison : brutes et traitées

Cross-validation : 10





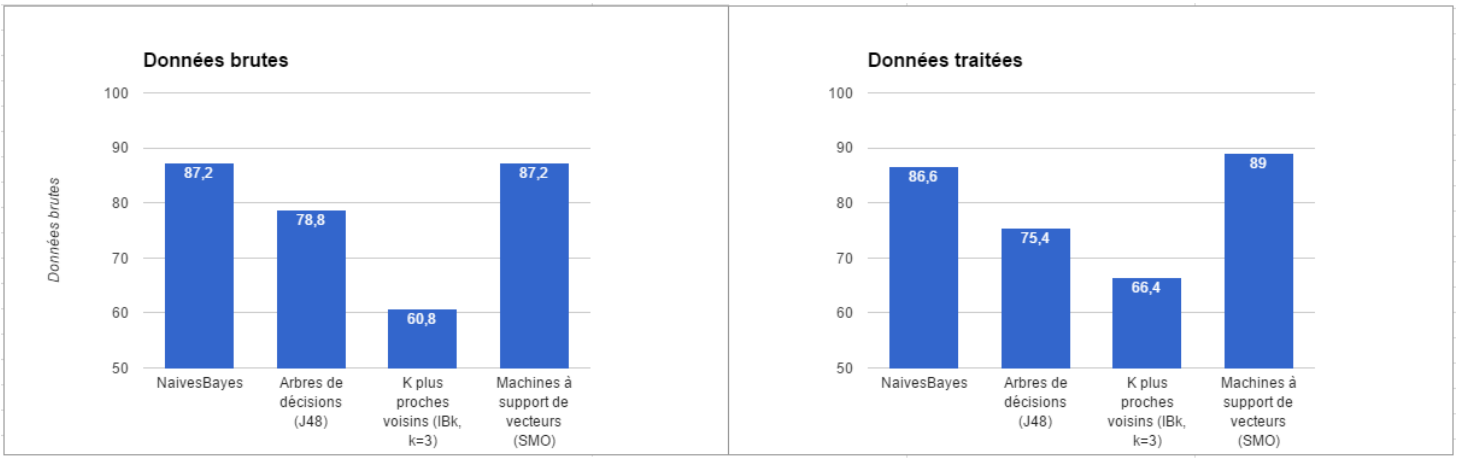
Naives Bayes :
Nous pouvons remarquer qu'il n'y a pas de changements car il est de base robuste contre le bruit et les attributs non pertinents. Le fait d'avoir retiré la ponctuation n'a pas changé le pourcentage de correctement classifié.

J48 :
Une très légère augmentation de 0,2% est à constater. L'algorithme n'est pas/peu influencé par la présence de mots parasites.

IBk :
Nous nous apercevons que les données traitées ont permis d'avoir un meilleur résultat lors de la classification. Cela est dû au fait que nous avons enlevé des mots et amélioré la qualité grâce au remplacement des smileys. Cet algorithme est sensible au bruit.

SMO :
Comme on peut le voir, les données brutes permettent un meilleur classement. Le fait de retirer des termes et/ou d'entrainer des faibles quantités de données rend moins efficace la classification via l'algorithme SMO.

Percent-split : 75%



Naives Bayes :
On peut remarquer un baisse faible (inférieur à 1%). On peut faire la remarque inverse faite par rapport à SMO dans le cas du cross-validation : Le fait de retirer des termes et/ou d'entrainer une plus grosse partie des données rend moins efficace la classification via l'algorithme Naives Bayes.

J48 :
Une baisse non négligeable de plus de 3% est observé. On peut en déduire la même remarque que sur Naives Bayes énoncé précédemment.

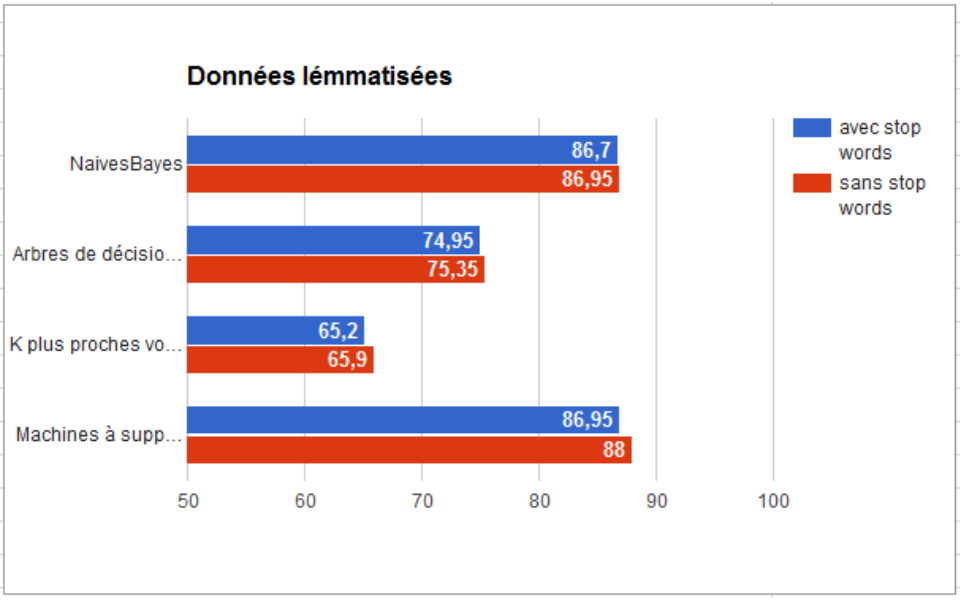
IBk :
On aperçoit une augmentation significative de 5,6%, ce qui est meilleur que celle observée avec la cross-validation. Prendre un plus grand jeu de données pour l'entrainement permet d'avoir une meilleure classification.

SMO :
Contrairement à l'observation faite sur le cross-validation, le fait d'appliquer SMO sur un plus gros jeu de données permet une amélioration d'environ 2%.

Conclusion :
Nous pouvons en conclure que le changement d'entrainement (cross-validation, percent-split) va influencer certains algorithmes. Comme par exemple, SMO qui a permis d'obtenir une meilleure classification lors des données traitées. Mais l'algorithme J48 en percent-split a perdu.
SMO et Naives Bayes donnent les meilleurs résultats contrairement à IBk qui nous donne les plus mauvais résultats.

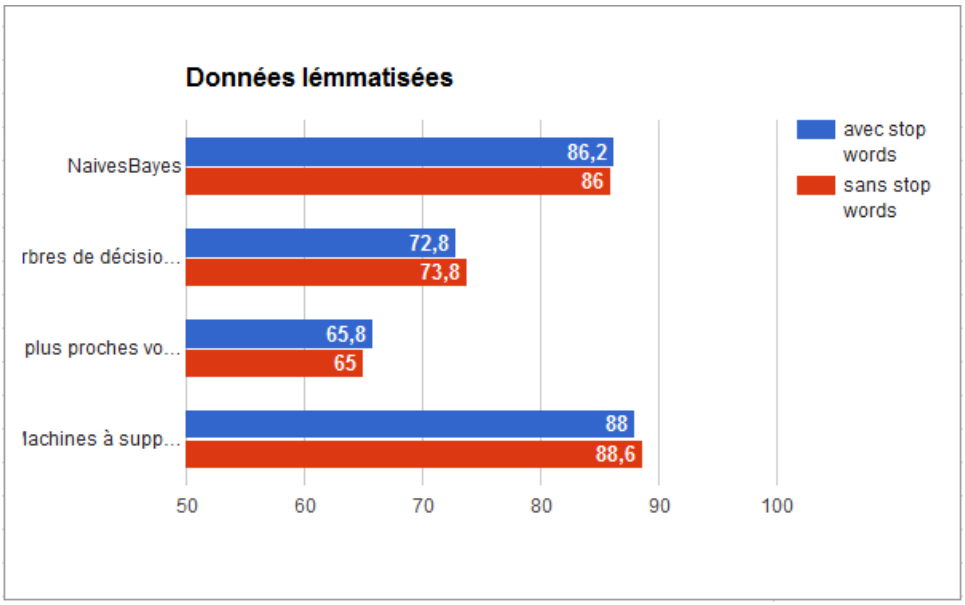
- Comparaison : lemmatisée

Cross-validation : 10



Nous pouvons remarquer que la lemmatisation sans les stop words est un peu meilleure que celle avec les stop words.
Le fait d'enlever certains stop words génère une perte de la précision.

Percent-split : 75%

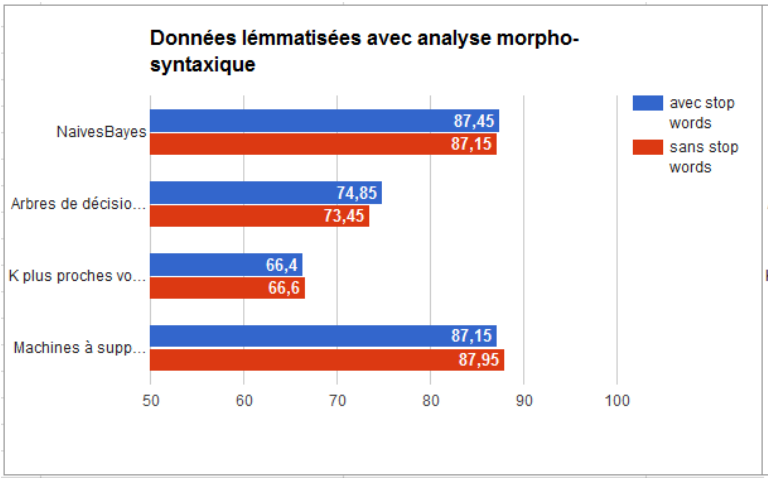


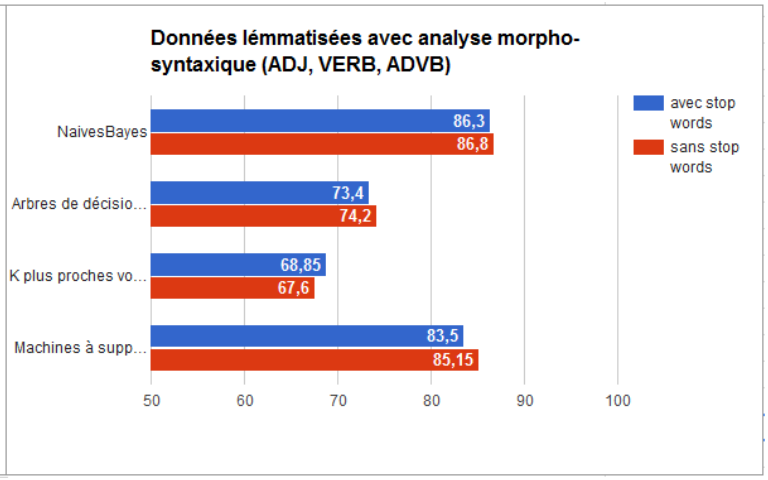
Contrairement à ce qui a été dit précédemment, la moitié des algorithmes sont meilleurs avec les stop words. Le fait d'enlever certains termes via la stop-list personnalisée, qui ont un poids important pour les algorithmes J48 et SMO dans la classification, provoque une diminution des résultats.
Cela ne pose pas de problème pour Naives bayes et IBk qui ont de sensibles augmentations.

Conclusion :
Naives Bayes et SMO nous donnent encore les meilleurs résultats.

- Comparaison : lemmatisée avec analyse morphosyntaxique

Cross-validation : 10





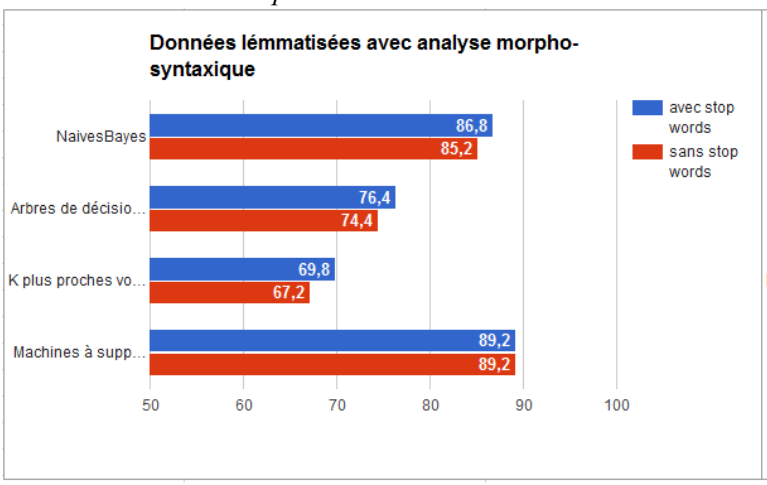
Naives Bayes :
Nous nous apercevons que l'analyse morphosyntaxique sans enlever des catégories permet d'avoir une meilleure classification. Cela est dû au fait que nous enlevons des éléments nécessaires à la classification.
Nous avons une légère inversion de la variation sans stop words et avec stop words. Comme on ne garde que les catégories : adjectifs, adverbes et verbes il y a sûrement des termes dans les catégories que nous enlevons qui doivent influencer la classification.

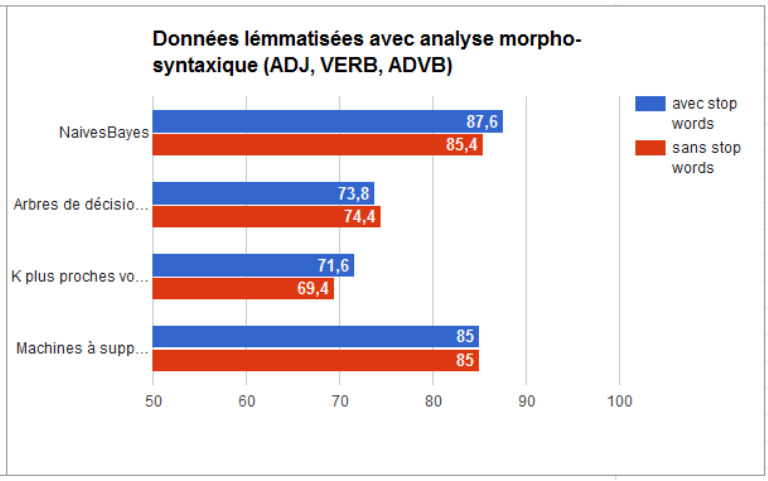
J48 :
Même remarque que pour Naives Bayes.

IBk :
Pour IBk, on observe l'inverse par rapport à J48 et Naives Bayes. Ceci s'explique par le fait que nous avons beaucoup moins de termes parasites.
Dans ce cas, nous obtenons un meilleur résultat lorsque nous appliquons l'analyse morphosyntaxique en gardant les verbes, adverbes et adjectifs (plus de 2% avec les stop words et 1% sans les stop words dans les résultats).
Dans le 2éme graphe, l'analyse morphosyntaxique avec filtre sur le fichier qui ne contient plus les stop words permet une meilleure classification.

SMO :
Même remarque que pour Naives Bayes. L'analyse sans utiliser les filtres lors de l'analyse morphosyntaxique est meilleure. On a une différence de 2%.

Percent-split : 75%





Naives Bayes :
Contrairement au cross-validation, l'entrainement sur un plus gros jeu de données permet une augmentation du pourcentage des données correctement classifiés (0,8 % avec stop words et 0.2% sans stop words). Les effets du filtrage des catégories se font ressentir.

J48 :
Il n'y a pas de changement entre les deux analyses morphosyntaxiques sans stop words. Mais nous observons une perte de plus de 2% pour l'analyse morphosyntaxique avec filtre des catégories.

IBk :
Nous pouvons faire la même remarque que pour le cross-validation.

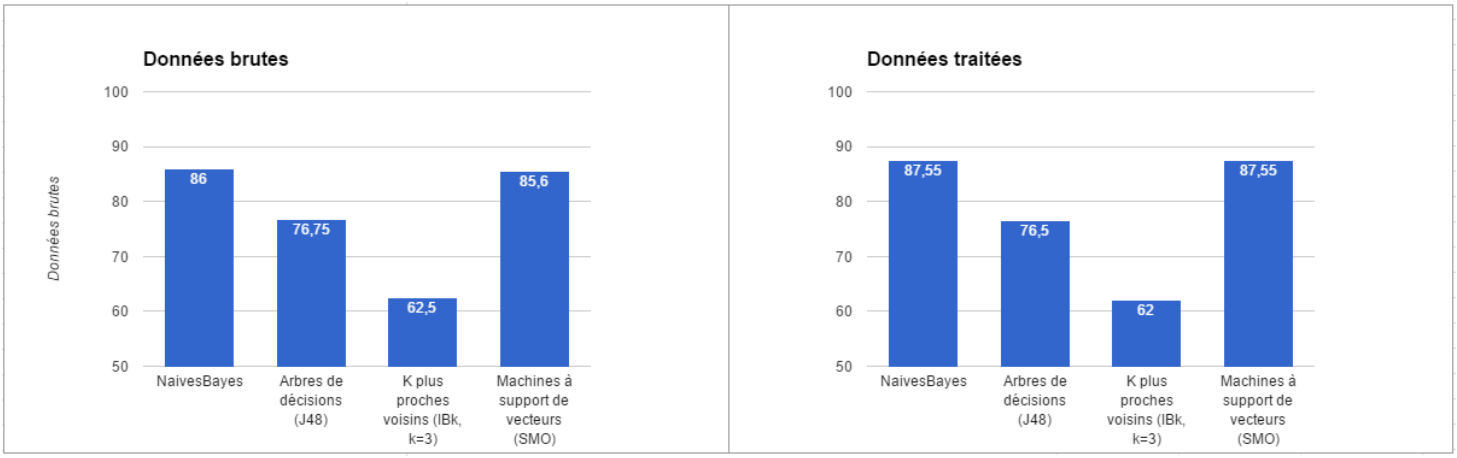
SMO :
Nous pouvons faire la même remarque que pour le cross-validation.
L'analyse morphosyntaxique sans filtre permet une augmentation de 4,2% par rapport à la l'analyse morphosyntaxique avec les filtres. Le fait d'avoir retiré un grand nombre de termes avec le filtrage des catégories peut être à l'origine de cette remarque.

Conclusion :
Dans tous les cas, avec cette configuration binaire sans N-gram, nous pouvons remarquer que les quatre algorithmes sont impactés positivement par le percent-split à 75%.
Naives Bayes et SMO nous donnent une nouvelle fois les meilleurs résultats.

b) Avec n-gram

➢ Comparaison : brutes et traitées

Cross-validation : 10



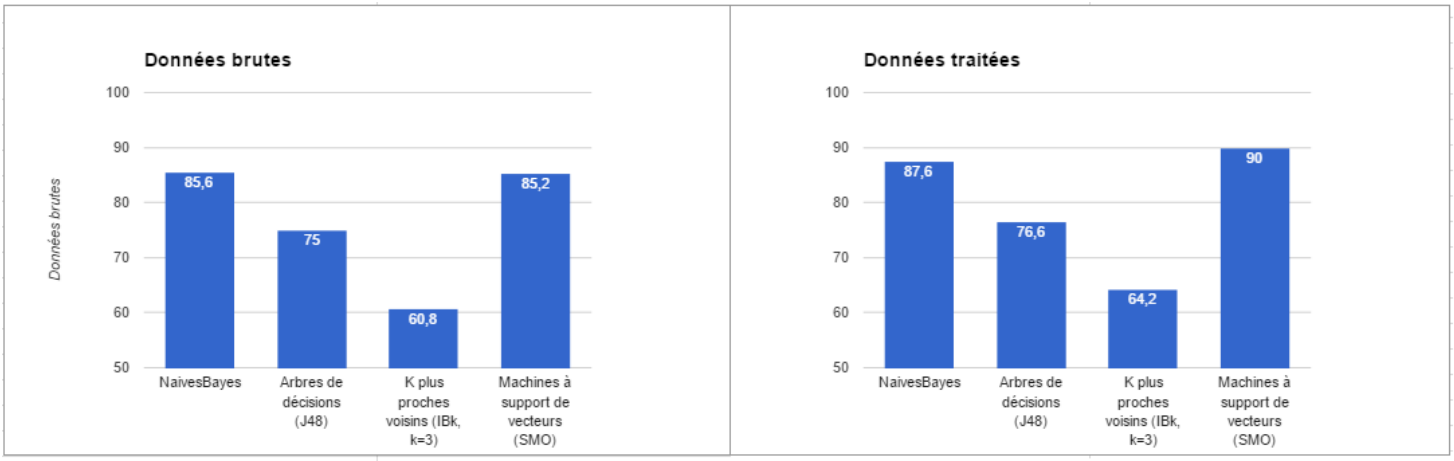
Naives Bayes :
Nous pouvons remarquer que le traitement des données avec n-grams permet d'avoir une hausse de plus de 1,5%. Le fait d'avoir enlevé des termes non-significatifs pour la polarité et donc d'avoir des ensembles de 1, 2 et 3 termes plus cohérents influe sur la classification.

J48 :
Une très faible baisse est à noter. Les n-grams n'ont quasiment pas d'effet sur l'amélioration de la classification entre brutes et traitées.

IBk :
Contrairement au cas sans utilisation des n-grams, on obtient ici une faible baisse. IBk, malgré des données "nettoyées", ne parvient pas à mieux classifier avec des ensembles de mots en attribut.

SMO :
Nous obtenons une augmentation pour les données traitées (environ 2%). Nous pouvons réitérer la même remarque que sur l'algorithme Naives Bayes.

Percent-split : 75%



Naives Bayes :
Comme précédemment dans cross-validation, nous obtenons une variation croissante quasi-identique.

J48 :
Les données traitées permettent d'avoir une meilleure classification due aux n-grams et à l'entrainement de 75% des données.

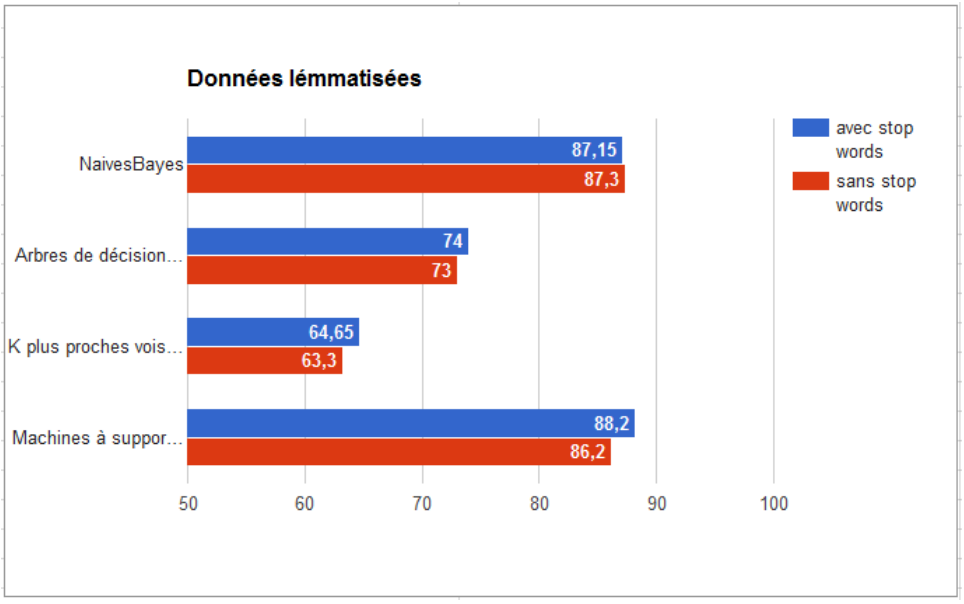
IBk :
Les données traitées permettent une hausse de 3%. En utilisant une plus grande quantité de données entrainées pour l'algorithme IBk, celui-ci arrive à améliorer la classification entre brutes et traitées.

SMO :
Cette fois-ci, nous obtenons une très forte augmentation (5%) et nous atteignons pour la première fois le seuil des 90% correctement classifié. Même remarque que sur J48 et IBk : plus on a une grande quantité de données entrainées, mieux sera l'amélioration de la prédiction.

Conclusion :
Pour les données traitées, lorsque nous utilisons percent-split pour classifier, nous avons des résultats en hausse par rapport aux données brutes.
Le contraire se produit lorsqu'on applique le cross-validation.
SMO et Naives Bayes sont les plus efficaces.

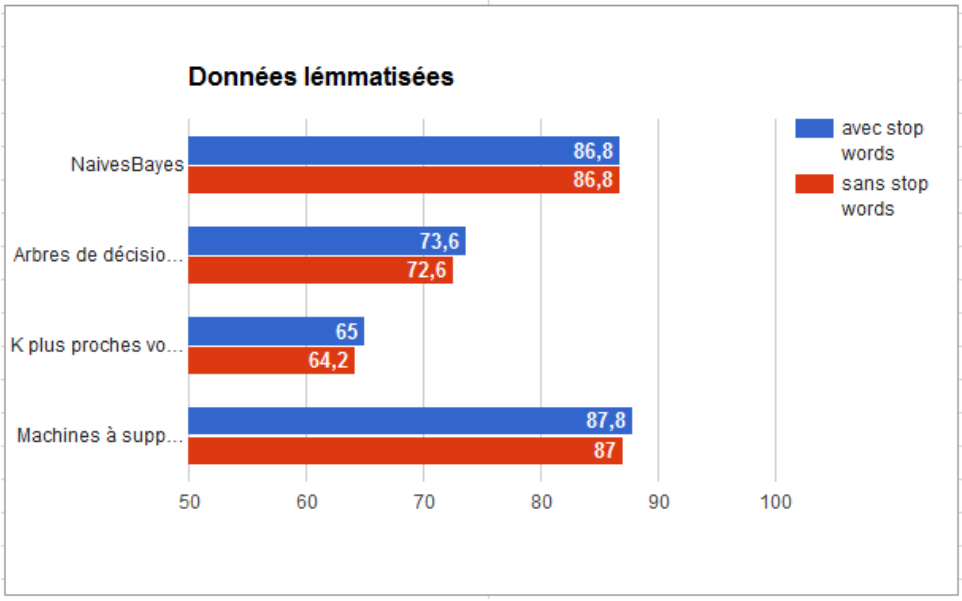
- Comparaison : lemmatisée

Cross-validation : 10



Nous pouvons remarquer que les données avec stop words permettent d'avoir une meilleure classification avec des taux de croissance compris entre 1 et 2 % hormis pour Naives Bayes. La baisse de résultats peut être expliquée par la diminution du nombre de mots due à la lemmatisation et donc de la qualité de la classification.

Percent-split : 75%

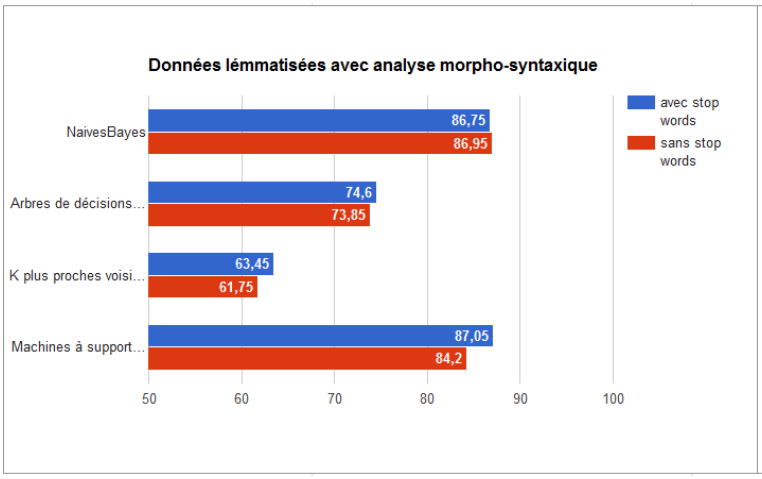


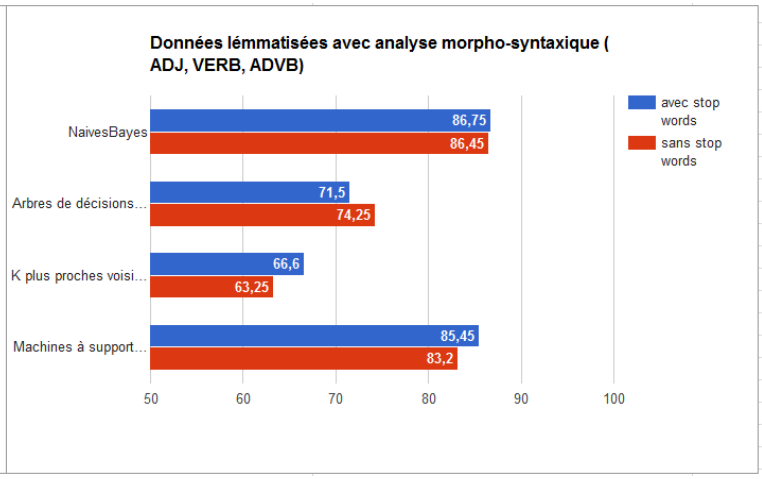
Globalement, nous obtenons de meilleurs résultats en retirant les stop words.

Conclusion :
Mis à part pour IBk, les résultats des données lemmatisées sont meilleurs avec le cross-validation 10. SMO et Naives Bayes nous donnent les meilleurs résultats.

- Comparaison : lemmatisée avec analyse morphosyntaxique

Cross-validation : 10





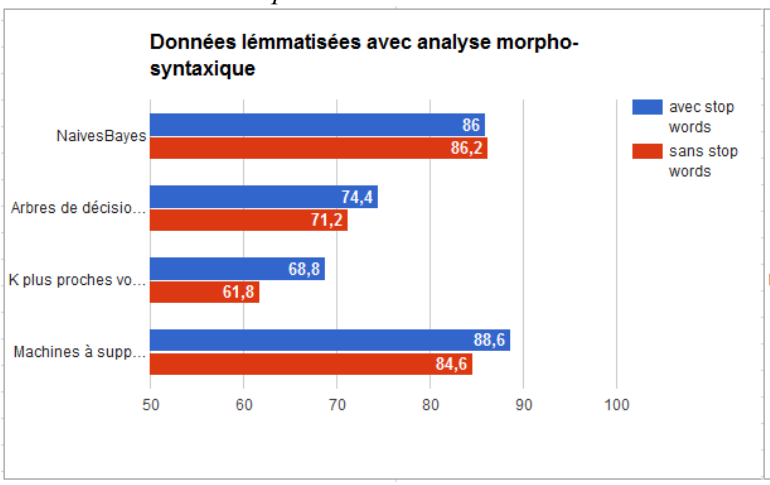
Naives Bayes :
Pour les deux analyses avec N-grams, Naives Bayes est très peu sensible aux changements.

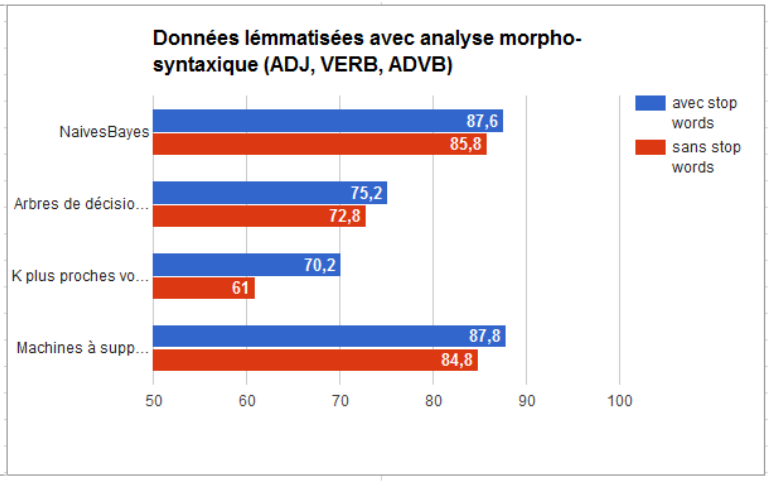
J48 :
Nous pouvons observer une perte de plus de 3% pour l'analyse morphosyntaxique avec filtres avec stop words.

IBk :
Des catégories étant supprimées dans le deuxième cas, les plus proches voisins sont potentiellement plus pertinents.

SMO :
L'écart entre les pourcentages correctement classifiés avec et sans stop words sont quasi-constants (plus de 2%) avec cependant, une petite baisse de performance dans le second graphe.

Percent-split : 75%





Naives Bayes :
Nous pouvons remarquer qu'au niveau du premier graphe, il y a une légère régression de 0,2% quand on utilise la stop-list.
Alors que pour le second graphe, on peut observer une hausse de 1,8 %.
La version sans stop-list contient des termes non-significatifs pour la polarité qui, malgré la lemmatisation via Treetagger, restent tel que "why", "there", "together" et "only". Ceci explique peut-être l'inversion de la tendance des histogrammes entre le graphe 1 et 2.

J48 :
Les résultats dans le second graphe sont améliorés dans les deux cas (avec et sans stop words). Nous pouvons aussi constater que dans chaque graphe, l'écart avec/sans stop words n'est pas négligeable (plus de 2 %). Ce sont donc les différentes filtres (stop words et catégories) qui ont apporté plus de précision à J48.

IBk :
L'analyse avec stop words est bien meilleure que celle sans les stop words. On enregistre même les meilleurs écarts (7% pour le premier graphe et plus de 9% pour le second).
On voit aussi une net amélioration avec les n-grams où l'on franchit la barre des 70%. Le bruit est en grande partie traité ce qui joue sur l'efficacité de cet algorithme.

SMO :
Ne garder que certaines catégories n'impact pas vraiment pas sur une amélioration de la classification.
On observe une baisse de moins d'1% dans le cas où l'on a utilisé une stop-list.
Cet algorithme a l'air sensible à la quantité de termes présents dans les données.

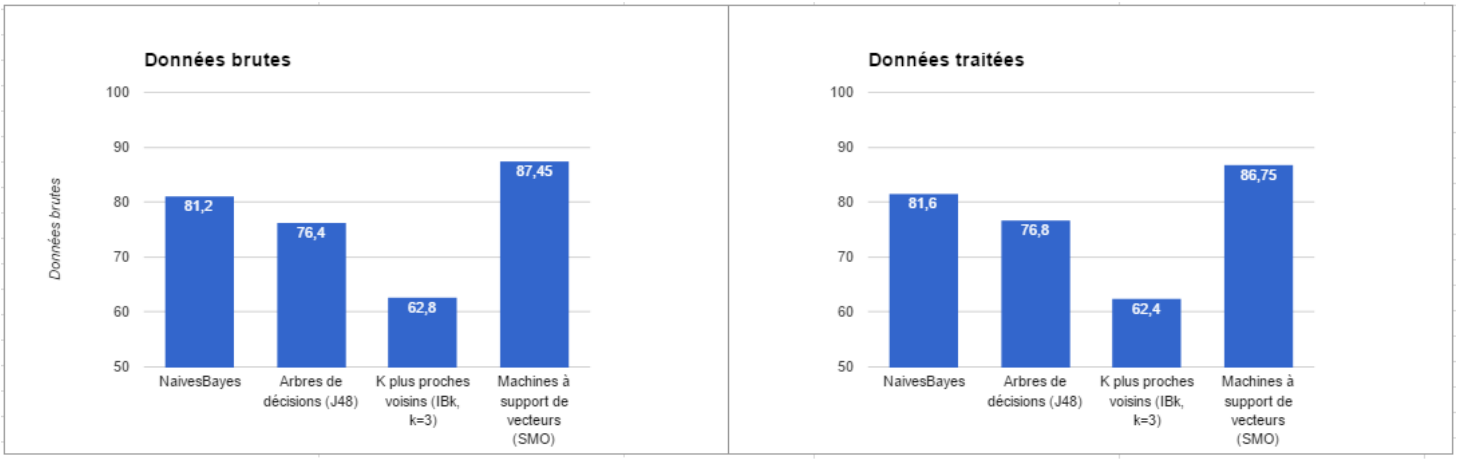
Conclusion :
Avec les quatre algorithmes nous voyons que le percent-split 75% permet une amélioration (notable dans le cas d'IBK). Mais les résultats sans stop words sont un peu meilleurs en cross-validation.
L'entrainement d'une plus grande portion de données nous montre l'intérêt d'enlever des termes (via la stop-list et le filtrage des catégories) dans l'optique d'améliorer la classification.
Naives Bayes et SMO sont les deux meilleurs algorithmes pour cette configuration binaire avec N-gram.

2. Fréquentielle

a) Sans n-gram

➢ Comparaison : brutes et traitées

Cross-validation : 10



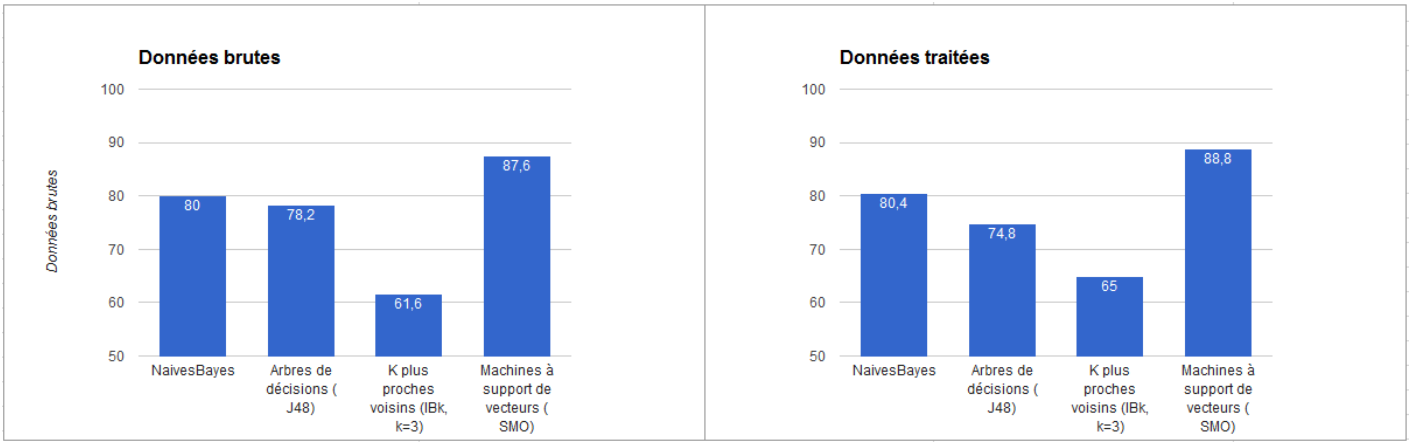
Naives Bayes :
Nous pouvons constater que les résultats sont proches.
La fréquence des termes influe négativement sur la classification avec cet algorithme. Cela pourrait concerner les termes parasites qui amplifie le bruit.

J48 :
Même remarque que celle faite sur les données binaires.

IBk :
Comme pour l'analyse binaire avec N-grams, nous obtenons une légère baisse avec les données traitées.

SMO :
Là encore, nous avons une baisse (0,7% dans ce cas) sur les données traitées. En se basant sur les résultats ci-dessous du percent-split 75%, nous en concluons que la "petite" taille du jeu de données entrainée en cross-validation 10 ne favorise pas l'amélioration de la classification avec les données traitées.

Percent-split : 75%



Naives Bayes :
Même remarque que pour le Naives Bayes en cross-validation. Nous pouvons noter en plus une perte de plus de 1% en percent-split par rapport au cross-validation

J48 :
Les données brutes sont en hausse de presque 4 % par rapport au résultat des données traitées, et presque 2% par rapport au données brutes en cross validation.

IBk :
Sur les données traitées, IBk est en hausse de 3,4% car il est très sensible au bruit.

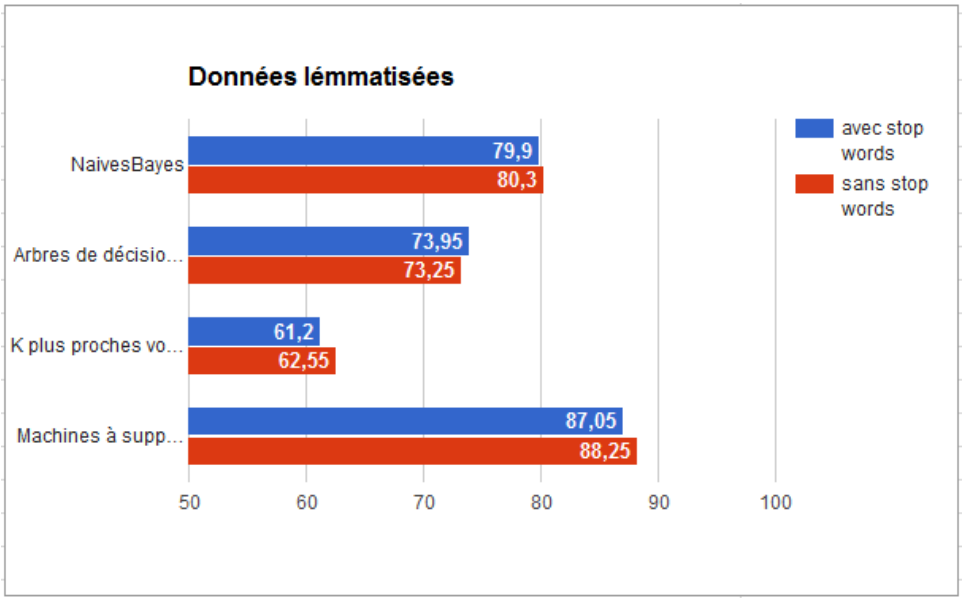
SMO :
Les données traitées permettent une meilleure classification (1%). Il est aussi globalement plus performant dans ce test en percent-split.

Conclusion :
Nous pouvons constater une nouvelle fois que le percent-split à 75% permet d'avoir les meilleurs résultats, saufs pour Naives Bayes qui reste meilleur en cross-validation 10.

Celui-ci donne de moins bons résultats par rapport aux données binaires. SMO reste l'algorithme qui arrive à correctement classifier les commentaires

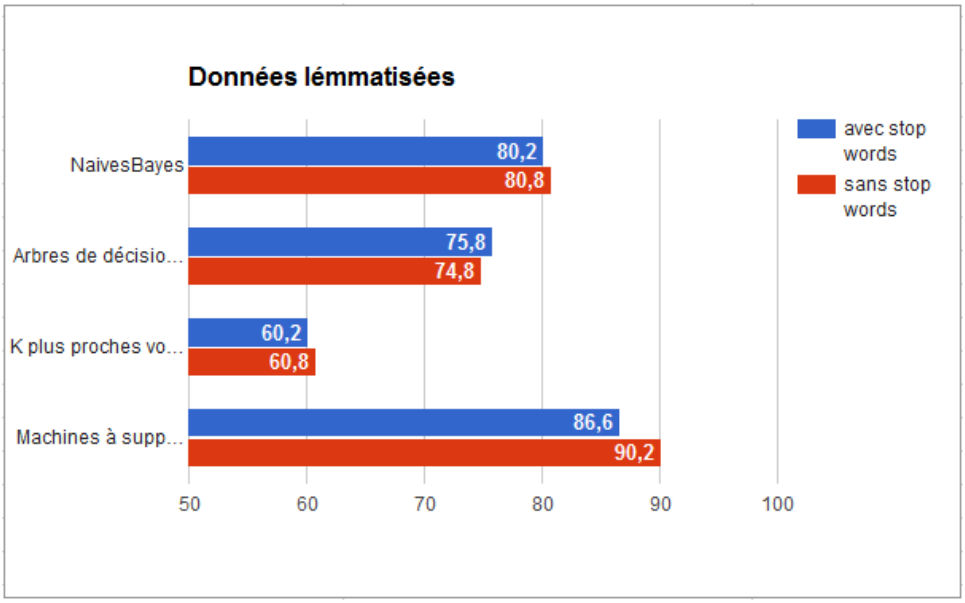
- Comparaison : lemmatisée

Cross-validation : 10



Les stop words ne sont bénéfiques que pour J48, on obtient toutefois un score très élevé avec SMO.

Percent-split : 75%



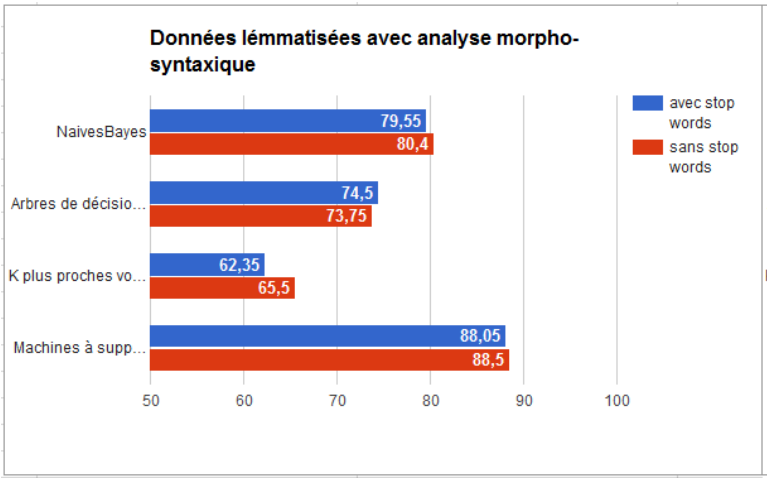
Les résultats en percent-split ont la même tendance que précédemment, mais sont plus élevés. A noter que l'algorithme machines à support vectoriel SMO passe à nouveau la barre des 90%.

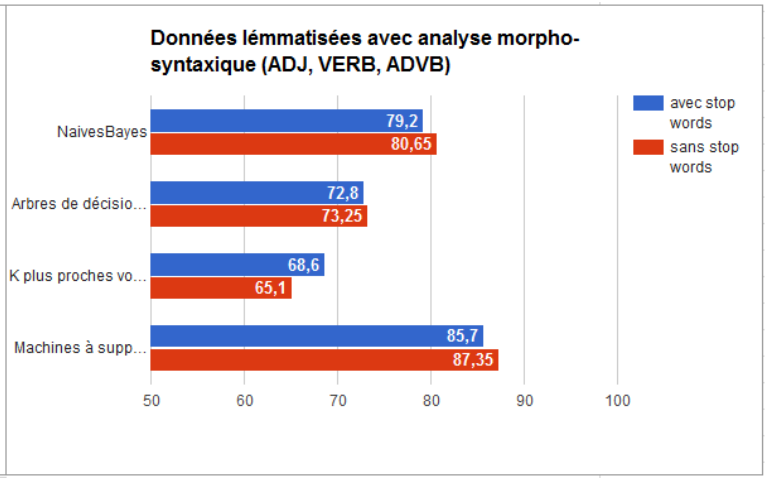
Conclusion :
Nous notons une baisse des performances dans la plupart des cas avec les stop words. Seul J48 en profite car cela lui permet d'affiner son arbre de décision.

L'algorithme SMO nous donne les meilleurs résultats.

- Comparaison : lemmatisée avec analyse morphosyntaxique

Cross-validation : 10





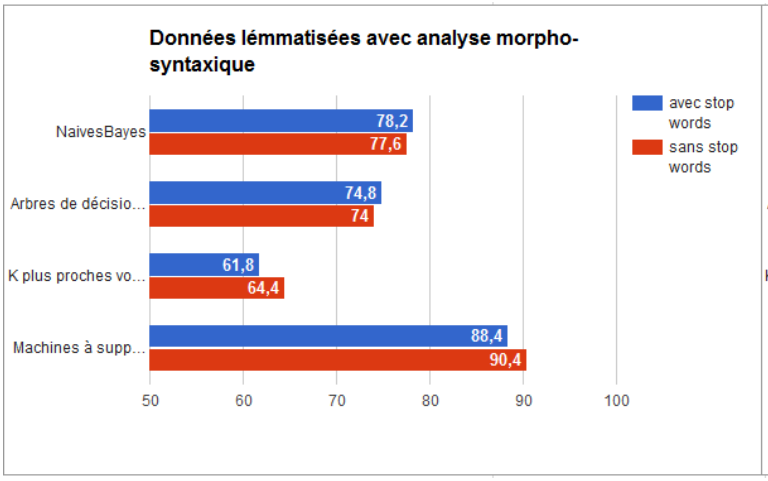
Naives Bayes :
Les résultats sont relativement similaires, le filtre sur les catégories n'apporte pas vraiment de changements.

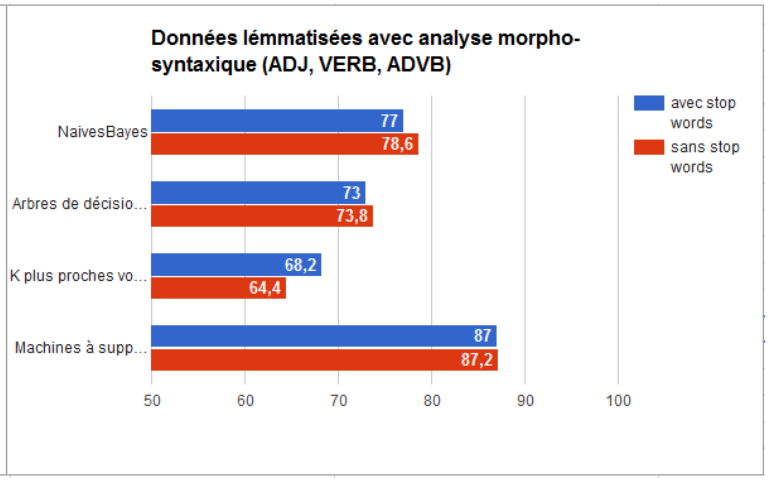
J48 :
Dans ce cas, J48 est meilleur sans filtre de catégories.
Pour le graphe 2, cette régression s'accentue plus pour les données où l'on a appliqué la stop-list.
L'application de filtres enlève des mots décisifs dans la construction de l'arbre de décision.

IBk :
IBk fait un bond de près de 6% pour les commentaires traités avec une stop-list car comme dit antérieurement, cet algorithme est très sensible au bruit.

SMO :
Les meilleurs résultats sont dans le cas de la lemmatisation sans filtre sur les catégories, la baisse se justifie par des catégories supprimés contenant des termes qui ont du poids pour la classification de SMO.

Percent-split : 75%





Naives Bayes :
Le percent-split n'apporte cette fois pas de meilleurs résultats que le cross-validation. L'utilisation du filtre de catégories est bénéfique uniquement sans stop words avec 1% de gagné.

J48 :
Même commentaire que pour Naives Bayes.

IBk :
A nouveau, nous observons très nettement qu'IBk est sensible aux termes parasites qui amplifient le bruit via les données fréquentielles. Ainsi il gagne plus de 6% en utilisant le filtre de catégories et des stop words.

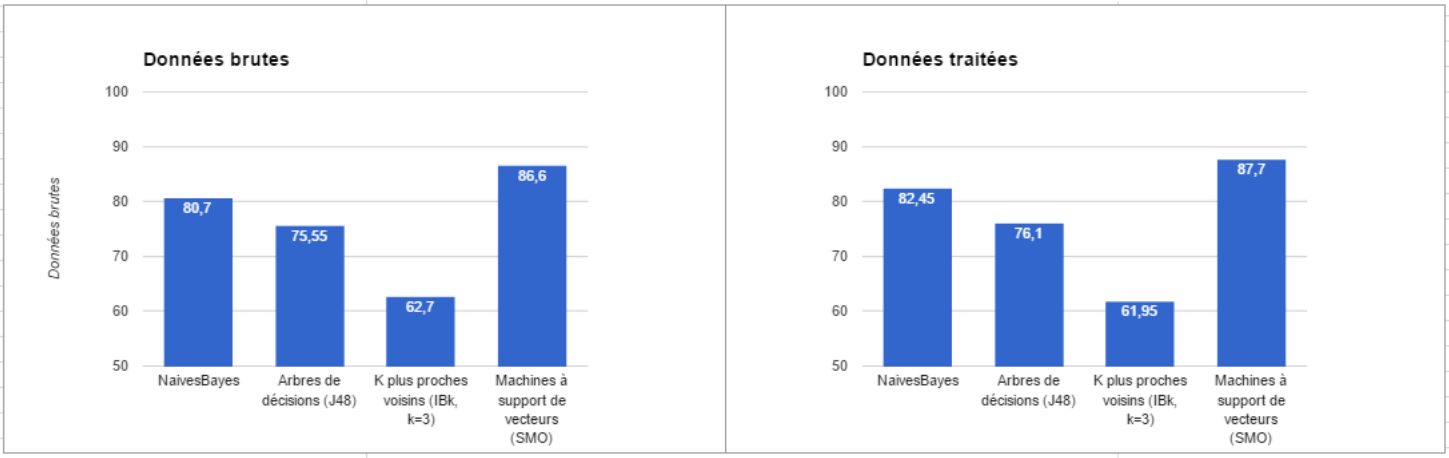
SMO :
Le seuil des 90% est une nouvelle fois dépassé. Bien que le score soit déjà élevé.
Nous pouvons réitérer la même remarque que précédemment.

Conclusion :
Les résultats en cross-validation 10 et en percent-split 75% sont assez similaires. La machine à support de vecteur est encore une fois bien plus performante que les autres. A noter plus globalement que l'utilisation des stop words n'a pas engendré d'amélioration du résultat.

b) Avec n-gram

➢ Comparaison : brutes et traitées

Cross-validation : 10



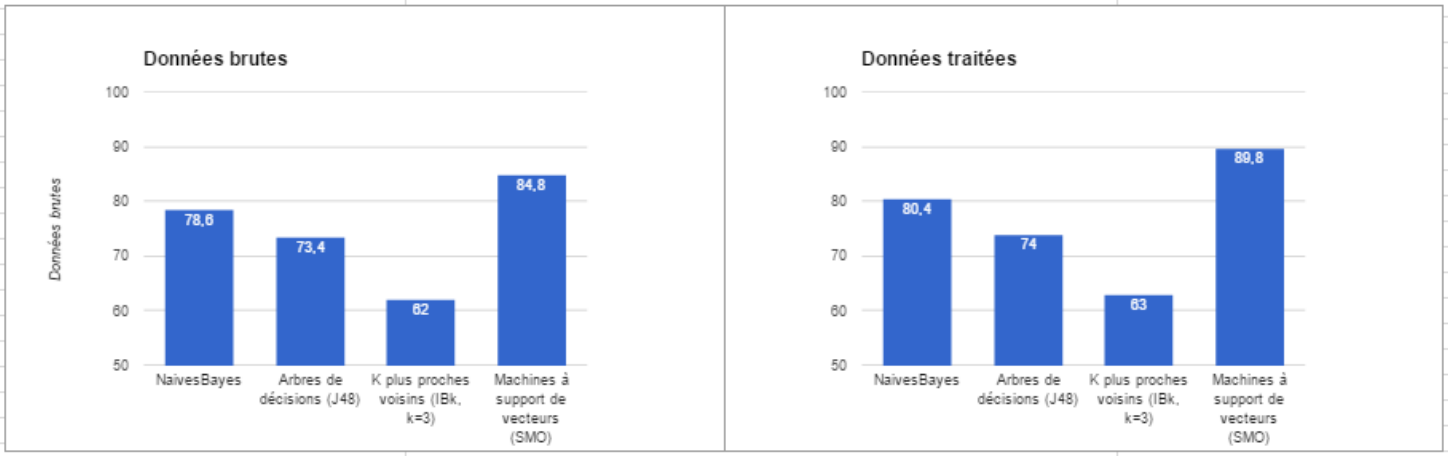
Naives Bayes :
Il y a une augmentation d'environ 2% pour le pourcentage de données correctement classifié. On peut voir ce même résultat dans la partie 1.b.

J48 :
Dans ce cas, Les données traitées permettent de voir une petite augmentation d'environ 0.5%. Nous pouvons en déduire que l'analyse fréquentielle associée au n-grams améliore l'analyse.

IBk :
Comme ce qui avait été constaté en binaire pour les n-grams, on observe une faible perte de moins de 1%.

SMO :
L'algorithme SMO sur les données traitées permet de gagner 1,1%. Nous pouvons en conclure la même remarque que pour le partie 1.b.

Percent-split : 75%



Naives Bayes :
Une augmentation d'environ 2% est remarquée. Nous pouvons en conclure pareil que pour le cross-validation.

J48 :
Nous observons une hausse de plus de 0.5%. Les n-grams appliqués aux données traitées permettent une variation croissante plus grande par rapport au cross-validation (de 0,05 % on passe à 0,6%) néanmoins, le fait d'appliquer un gros jeu de données ne permet pas d'obtenir un meilleur résultat.

IBk :
Les données traitées donnent une augmentation de 1%. Le fait d'appliquer une grande quantité de données à celle à entrainer permet une amélioration.

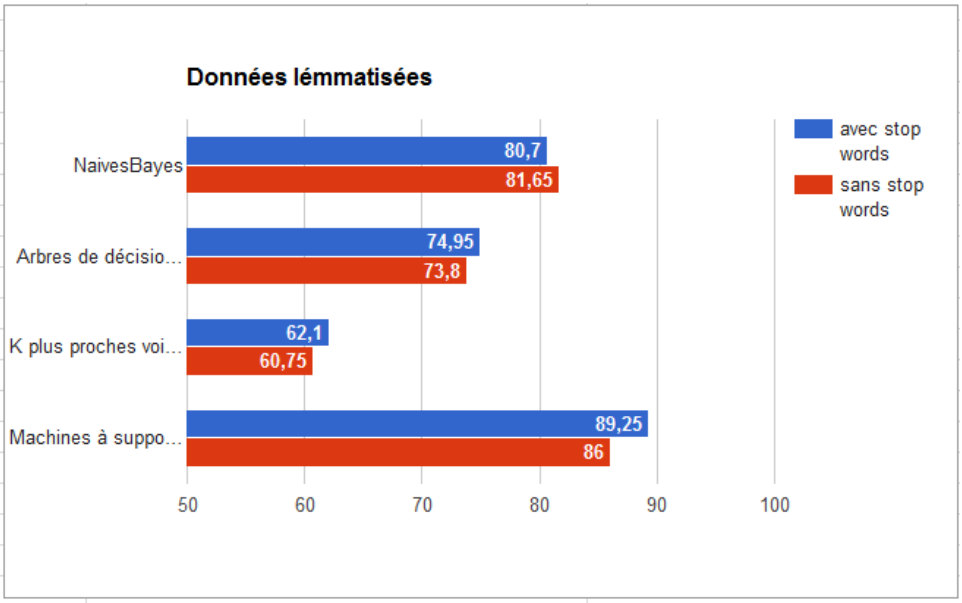
SMO :
Nous obtenons une très forte amélioration (5%). Nous pouvons en conclure la même remarque que précédemment et que 1.b.

Conclusion :
Au niveau des données brutes, le cross-validation donne de meilleurs résultats.
Au niveau des données traitées, le cross-validation est meilleur pour les algorithmes Naives Bayes et J48 alors que le percent-split est amélioré pour les algorithmes IBk et SMO.

Ce dernier reste l'algorithme qui donne le plus de satisfaction pour la classification.

- Comparaison : lemmatisée

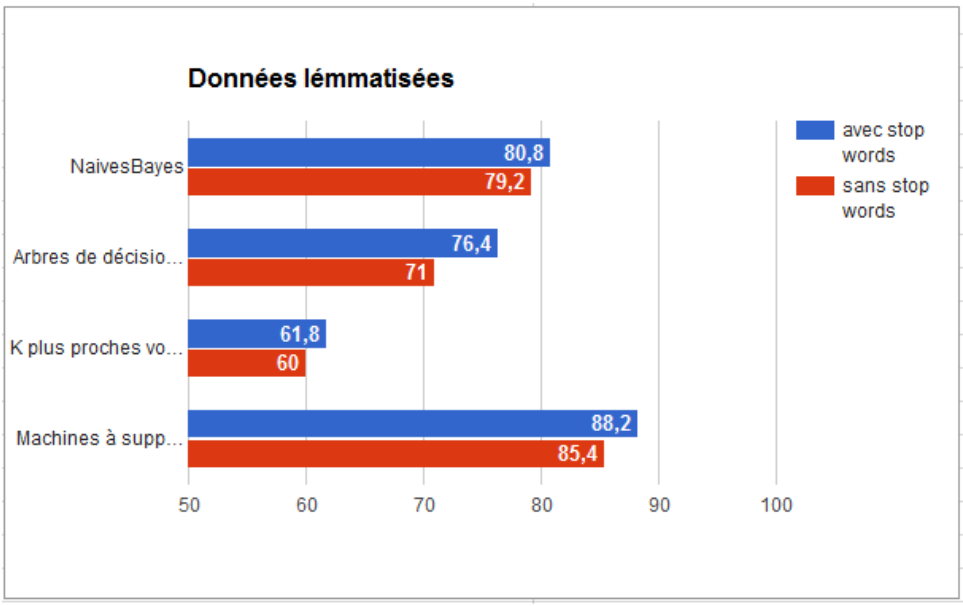
Cross-validation : 10



Pour l'algorithme Naives Bayes, nous voyons que le fait de garder les stop words permet d'avoir une meilleure classification (hausse de presque 1%).
Contrairement aux autres algorithmes qui ont de meilleurs résultats lorsque les stop words ont été supprimés. Surtout pour SMO, l'amélioration est importante de 3,25%.

Les tendances des histogrammes sont semblables à celles faites pour les n-grams en binaire (partie 1.b).

Percent-split : 75%

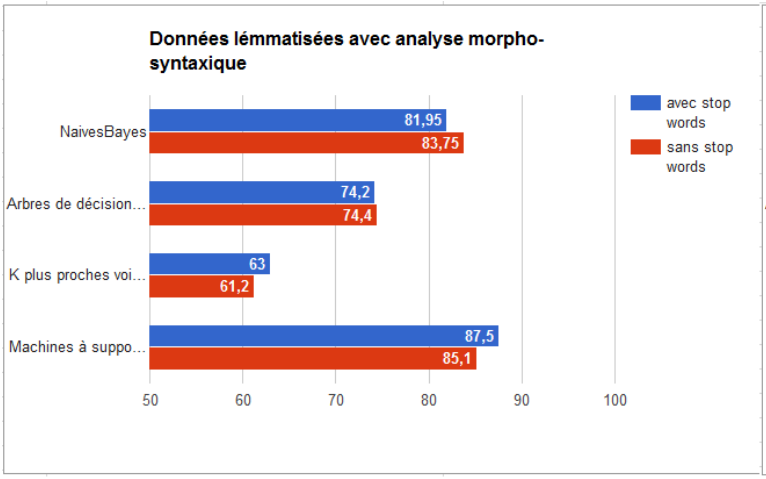


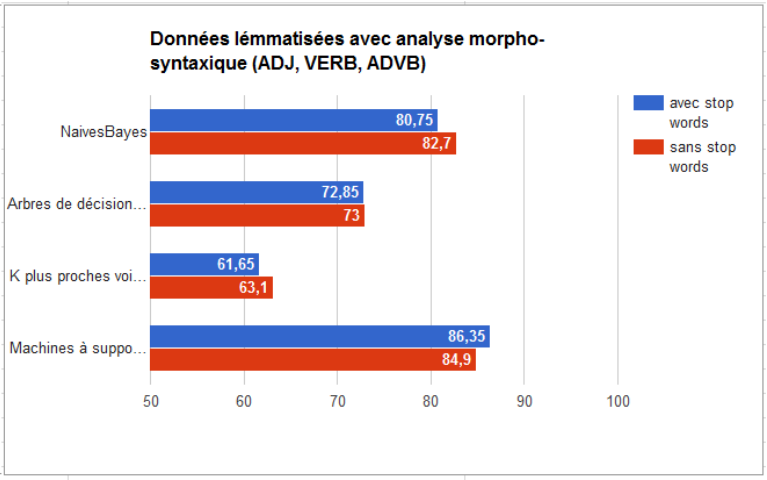
Dans le cas de percent-split, tous les algorithmes classifient mieux lorsque nous appliquons la stop-list.
Une augmentation de 5,4% est observée pour J48.

Conclusion :
La lemmatisation en supprimant les stop words améliore globalement les résultats.
Le meilleur algorithme reste SMO.

- Comparaison : lemmatisée avec analyse morphosyntaxique

Cross-validation : 10





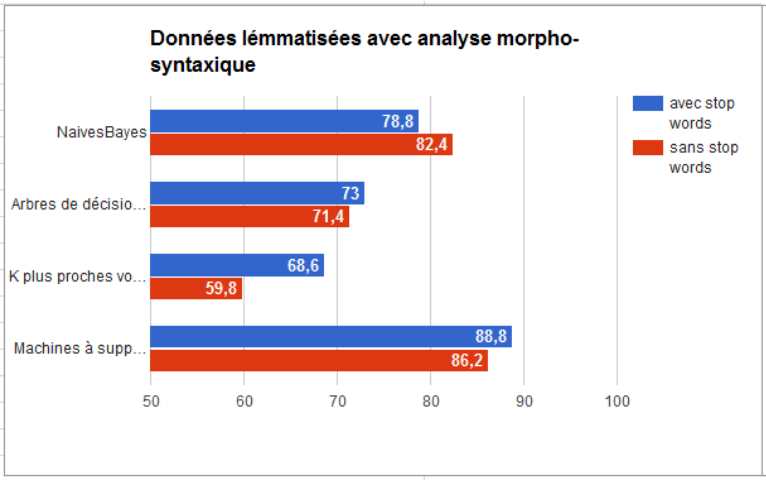
Naives Bayes :
L'analyse morphosyntaxique sans utilisation de filtre permet d'obtenir de meilleures classifications.
Pour les 2 graphes, l'analyse sans stop words est meilleure.
A noter que les résultats sont moins bons qu'en binaire (baisse de plus de 6%).

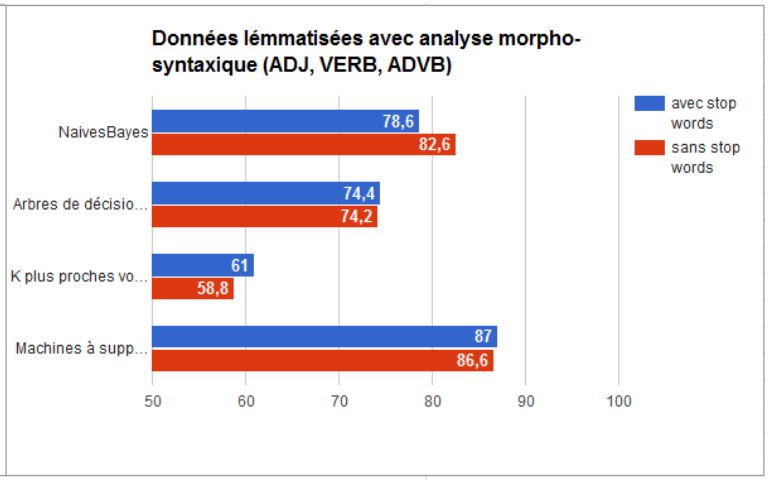
J48 :
Les pourcentages de données correctement classifiées avec et sans filtre de stop words sont sensiblement égaux.
Les résultats sont meilleurs dans le premier graphe (1,4% d'écart par rapport au second). Cela est dû au fait que nous gardons des mots nécessaires à la réalisation de l'arbre de décision.

IBk :
Dans le premier graphe, nous obtenons un meilleur résultat par rapport au 2eme car dans ce dernier, nous enlevons trop de mots importants entre le filtre de catégories et celui de stop words.
De plus, IBk est fortement impacté par l'analyse fréquentielle avec filtre de catégories.

SMO :
Pour SMO, le fait de ne pas appliquer le filtrage de catégories spécifiques donne des résultats plus élevés. On peut en conclure que nous enlevons trop de termes lors de l'utilisation des filtres.
Les résultats sont légèrement mieux qu'en binaire.

Percent-split : 75%





Naives Bayes :
Les résultats sont similaires pour les 2 graphes lors de la suppression et du maintien des stop words. Dans le graphe 2, la version sans stop-list est plus efficace pour classifier tandis qu'on avait le contraire avec les données binaires. Naives Bayes est affecté par l'analyse fréquentielle avec n-grams car il est sensible à la quantité de données. Les possibilités de construire des n-grams sont en effet réduites par les multiples filtrages.

J48 :
Filtrer les catégories fait doubler la progression du cas sans utilisation de la stop-list par rapport au cas avec stop words. Le fait de ne garder que les adjectifs, les verbes et les adverbes permet d'avoir un meilleur arbre de décision.

IBk :
Les données lemmatisées sans filtres donnent une amélioration des résultats : une hausse de 7,6% avec stop words et 1% sans stop words. Même remarque que pour Naives Bayes.

SMO :
Nous pouvons faire la même remarque que celle faite avec le cross-validation.

Conclusion :
Pour l'analyse morphosyntaxique sans filtres, le percent-split permet des améliorations dans la classification. Alors que pour l'analyse morphosyntaxique avec filtres, pour les Naives Bayes et IBk, le cross-validation est mieux alors que pour J48 et SMO c'est l'inverse.
La classification de données binaires reste généralement meilleure que celle avec des données fréquentielles. Peut être que la diminution du nombre de mots provoque une baisse de la possibilité de faire des n-grams. D'autant plus que la fréquence d'apparition d'un ensemble de termes (n-gram) est plus basse que celle d'un terme seul.

SMO est toujours celui qui classe le mieux les données.

Conclusion

La grande variation des résultats en fonction des filtres (catégories et/ou stop words) et des configurations (binaire ou fréquentielle, avec ou sans n-gram), met en avant le fait que ces algorithmes (surtout IBk) sont assez sensibles aux types de configurations et de données. D'où l'intérêt de bien choisir son algorithme en fonction de ses avantages/inconvénients.
Nous pouvons noter cependant que la machine à support de vecteurs nous a toujours apporté les meilleurs pourcentages de classification.
Avec Naives Bayes qui le suit de près, ces deux algorithmes sont donc très polyvalents.

Pour chaque algorithme nous concluons que :

SMO est le plus performant lors de ces tests. Nous pouvons aussi remarquer qu'il nous donne de meilleurs résultats sur des données raffinées. Il est très peu impacté par le bruit, et il est efficace lorsqu'il y a beaucoup de mots dans les textes.

Naive Bayes arrive en deuxième position, proche de SMO, et ce, globalement quel que soit le type d'analyse. De plus, il est moins efficient en fréquentielle qu'en binaire (baisse d'environ 10%). Il est donc très sensible au nombre de mots du fait qu'il utilise des formules probabilistes.

J48 est lui aussi assez stable dans la classification quelque soit les configurations et les données. Nous observons quelques variations lors de l'utilisation de filtres (stop words et/ou catégories), ce qui montre bien qu'il est important d'utiliser des filtres adaptés à la construction de l'arbre de décision de J48.

IBk nous a toujours donné des résultats en retrait par rapport aux autres, avec toutefois une forte hausse en percent-split avec N-grams et filtre de stop words. Nous en concluons que l'algorithme des K plus proches voisins est très sensible aux bruits. Cet algorithme n'est pas adapté à nos tests et ne nous permet malheureusement pas de lui trouver des atouts.

Nous allons faire un classement des 3 plus hautes valeurs obtenues.
C'est l'algorithme SMO qui nous les a donné :

- 90,4% en percent-split en fréquentielle sans n-grams sur les données lemmatisées avec analyse morphosyntaxique sans filtre en gardant les stop words.
- 90,2% en percent-split en fréquentielle sans n-grams sur les données lemmatisées en gardant les stop words.
- 90% en percent-split en binaire avec n-grams sur les données traitées.

Annexes

```
#!/bin/bash

# ** Script pour lemmatisation : créer un fichier par commentaire
contenant le résultat de la lemmatisation **

# Création du dossier contenant l'ensemble des fchiers lemmatisés
directoryLemmatisation="../donnees_lemmatisees/";
mkdir -p $directoryLemmatisation;

directoryDataLemmatisation="";
pathDataToLemmatisation="";

if [ $1 = "1" ]
then
    directoryDataLemmatisation="ponctuation_smileys/";
    pathDataToLemmatisation="../elements_projet/dataset_for_lemmatisati
on_smileys_ponctuation.csv";
else
    directoryDataLemmatisation="ponctuation_smileys_stopwords/";
    pathDataToLemmatisation="../elements_projet/dataset_for_lemmatisati
on_smileys_ponctuation_stopwords.csv";
fi

mkdir -p $directoryLemmatisation$directoryDataLemmatisation;


# Indice unique pour le nom de chaque fichier
indice=1;

# Tant qu'on lit des lignes du fichier CSV (ponctuation + smileys
traités)
while IFS='' read -r line || [[ -n "$line" ]]; do
        echo "Traitement du commentaire n°$indice";

        # Traitement du commentaire avec treetagger et resultat
redirigé dans un fichier texte
        echo $line | tree-tagger-english >
$directoryLemmatisation$directoryDataLemmatisation$indice.txt;

        # Incrémentation de l'indice
        indice=$((indice+1));
done < $pathDataToLemmatisation
```

Annexe 1 : Script shell pour la lemmatisation avec TreeTagger

```
80    CD    @card@
's    POS    's
movies    NNS    movie
Well    RB    well
the    DT    the
only    JJ    only
thing    NN    thing
the    DT    the
movie    NN    movie
is    VBZ    be
worth    JJ    worth
is    VBZ    be
of    IN    of
sexy    JJ    sexy
Kareena    NP    <unknown>
who    WP    who
looked    VBD    look
```

Annexe 2 : Exemple de fichier traité avec TreetTagger (mot d'origine, catégorie, lemme)