Université Paris Nanterre

Insects or not ?

Classification binaire d'articles Wikipédia sur des espèces d'insectes et non-insectes

Charlotte Schermesser et Léna Gaubert
M1 TAL 2022-2023

UE : Introduction à la fouille de texte

Source gallica.bnf.fr / Bibliothèque nationale de France

Plusieurs insectes différents, Lequeu, Jean Jacques (1757-1826), dessin. (Gallica)

Table des matières

Introduction

La fouille de texte permet de traiter efficacement certaines tâches précises, comme la classification. Cette dernière consiste à attribuer une classe à chaque donné d'entrée, permettant ainsi de trier les données en fonction de la correspondance de leurs caractéristiques à des classes. Elle peut être réalisée par apprentissage automatique, processus pendant lequel une partie d'un corpus déjà étiqueté (chaque donnée est associée à une classe) sera utilisée pour entraîner l'algorithme, tandis que l'autre permettra de le tester l'algorithme sur sa capacité à classifier de nouvelles données.

L'objectif de ce projet est de comparer les performances de plusieurs algorithmes de classification en entraînant des classifieurs par apprentissage automatique sur un corpus de notre choix.

Nous utiliserons le logiciel Weka et la bibliothèque scikit-learn pour réaliser cette tâche. Pour le corpus, nous avons choisi de nous intéresser aux insectes. Plus spécifiquement, à la classification d'animaux en insectes ou non insectes, en prenant des textes sur les insectes ainsi que des textes sur des animaux avec lesquels ils peuvent souvent être confondus, comme les Arachnides. Le corpus constitué pour cette tâche de classification binaire est composé de deux classes, les insectes et les non insectes pour un total de 221 textes.

Constitution du corpus

Les textes de notre corpus sont des articles du wiki fandom "insect wiki", qui réunit des passionnés du sujet et compte de nombreux articles sur les insectes ainsi que sur des animaux souvent confondus avec. Les articles de ce wiki sont sous licence Creative Commons Attribution-ShareAlike (CC BY-SA), une licence libre qui autorise le partage, la distribution et la création d'œuvres dérivées sous la même licence. Le wiki est disponible à ce lien: https://insects.fandom.com/wiki/Insect_Wiki

Le système de catégorisation principal des pages du wiki est l'ordre auquel appartiennent les insectes (par exemple: Coleoptera, Odonata, Lepidoptera). Chaque page d'ordre contient les pages de différents insectes lui appartenant ainsi que des pages de catégories correspondant à des familles de cet ordre.

Pour constituer notre corpus, nous avons pris presque tous les articles de non insectes que sur le wiki, dont les deux catégories principales sont "Arachnids" (araignées, acariens, scorpions etc.) et "Myriapods" (millepattes) pour constituer la classe "non insecte" puis sélectionné quelques pages dans les différents ordres d'insectes (coleoptera, blattodea etc.) afin d'avoir un corpus "insectes" assez divers. Les insectes des ordres Coleoptera, lepidoptera et hemiptera sont plus représentés dans cette classe car leurs ordres comportent plus d'articles, mais nous nous sommes limitées à 20 articles max par ordre pour ne pas qu'un ordre spécifique soit sur représenté.

Nous avons utilisé la page Special:export du wiki pour entrer les catégories du wiki avec les articles qui nous intéressent, retiré les doublons (certaines pages étant dans plusieurs catégories) puis téléchargé deux documents XML contenant les textes des pages sélectionnées. Le premier comporte les pages de non insecte et le second, les pages d'insectes. Ces fichiers XML ont ensuite été transformés en txt avec le script wiki_xml_parsing, que nous avions vu en cours de Machine creativity.

Chaque document txt ainsi obtenu a été transformé en dossier (un par classe) à l'aide du script corpus_split.py

Weka

Une fois les données collectées, il est nécessaire de les préparer afin de les rendre lisibles par weka. Ce logiciel propose une grande variété d'algorithmes d'apprentissage automatique pour la classification.

Les textes du corpus ont été transformés en différents documents .arff (corpus_weka.arff, et corpus_boolean_weka.arff) avec le script vectorisation.py afin d'être exploitables sur Weka.

Options de tests:

Cross validation: Le dataset est divisé en n parties (folds) de taille égale (on peut changer n, le nombre de parties, dans les paramètres). A chaque itération (n fois), n-1 folds sont utilisés pour l'apprentissage et le fold restant sert à tester l'algorithme ainsi entraîné. Les résultats du test sont une moyenne des résultats de chaque itération.

Percentage split: Le dataset sera divisé en deux ensembles, dont la taille dépend du pourcentage spécifié. L'un des ensemble servira d'entraînement tandis que l'autre sera l'ensemble de test.

supplied test set: l'utilisateur peut fournir un ensemble de test en plus du dataset déjà fourni qui servira alors entièrement pour l'apprentissage.

L'ensemble des données étant assez petit, on utilisera ici la Cross validation et le document corpus_boolean_weka.arff . Chaque algorithme disponible sur weka est testé avec les paramètres par défaut, mais nous ne les mentionnerons pas tous.

Famille Bayes

Cette famille d'algorithmes est basée sur la règle de Bayes.

$$P(A \mid B) = \frac{P(B \mid A)P(A)}{P(B)}$$

C'est avec l'algorithme NaiveBayesMultinomial Updateable que les meilleurs résultats sont obtenus avec 95,47% d'articles correctement classifiés. On voit qu'il classe très bien les non insectes mais il est difficilement interprétable à cause du nombre de mots utilisés pour classifier.

a	**b**	**←classified as**
101	8	**a = insects**
2	110	**b = non_insects**

Précision	0,956
Rappel	0,955
F Mesure	0,955

Famille Rules

Jrip

En utilisant Jrip en cross validation avec les paramètres de prédéfinis, on obtient 88,23 % de pages correctement classifiées (195 pages sur un total de 221.)

L'un des avantages de Jrip est qu'il utilise des règles de classifications simples et compréhensibles. Ces règles suivent le format: IF (condition) THEN (classe) (taux de précision/taux d'erreur). Ici les articles ont été classés selon les règles suivantes:

(categoryinsecta >= 1) => xClasse=insects (70.0/0.0)

(wings >= 1) => xClasse=insects (16.0/1.0)

(categorylepidoptera >= 1) => xClasse=insects (4.0/0.0)

(categoryhymenoptera >= 1) => xClasse=insects (5.0/0.0)

(how >= 1) => xClasse=insects (3.0/0.0)

=> xClasse=non_insects (123.0/12.0)

a	**b**	**←classified as**
91	18	**a = insects**
8	104	**b = non insects**

On voit que les non-insectes sont plutôt bien classés (ils ne répondent pour la plupart à aucune des règles qui permettraient de les classer en insectes), tandis que les insectes le sont moins.

Précision	0.885
Rappel	0.882
F Mesure	0.882

Zero R

Tous les documents sont classés « non insectes », la classe majoritaire. Pendant l'entraînement, il détermine la classe majoritaire et prédit cette classe pour toutes les

nouvelles données, ce qui donne dans notre cas un résultat proche de 50 % de fichiers correctement classifiés (50,68%) nos deux classes étant à peu près équilibrées. Changer les paramètres n'a pas d'impact sur le résultat.

OneR

L'algorithme OneR trouve une règle simple qui utilise une seule variable pour trier toutes les instances du jeu de données. Ici le critère décisif pour la classification est la catégorie 'insecta'.

categoryinsecta:

< 0.5 -> non_insects

>= 0.5 -> insects

(182/221 instances correct)

a	**b**	**←classified as**
70	39	**a = insects**
0	112	**b = non_insects**

On peut voir que les non insectes sont parfaitement classés, mais les insectes le sont beaucoup moins. Cela peut être dû au fait qu'aucun non-insecte n'appartient à la catégorie insecta (une catégorie du wiki qui correspond à la classe d'animaux invertébrés qui réunit tous les insectes), mais que beaucoup d'insectes ne sont pas non plus dans cette catégorie sur le wiki.

Précision	0,869
Rappel	0,824
F mesure	0,817

Famille des arbres

Les algorithmes de cette famille construisent un arbre de décision pour résoudre les problèmes de classification.

J48

J48 crée un arbre dont les nœuds internes sont les tests effectués sur des caractéristiques et ces tests permettent de déterminer les classes finales représentées par les **feuilles** de l'arbre (dans le cas présent, insect et non_insect). 86,42% des articles sont correctement classifiés

categoryinsecta <= 0

| categoryhymenoptera <= 0

| | wings <= 0

| | | nest <= 0

| | | | antennae <= 0: **non_insects** (113.0/9.0)

| | | | antennae > 0

| | | | | bodies <= 0: **insects** (6.0/1.0)

| | | | | bodies > 0: **non_insects** (3.0)

| | | nest > 0

| | | | == <= 0: **insects** (6.0/1.0)

| | | | == > 0: **non_insects** (2.0)

| | wings > 0: **insects** (12.0/1.0)

| categoryhymenoptera > 0: **insects** (9.0)

categoryinsecta > 0: **insects** (70.0)

Number of Leaves : 8

Size of the tree : 15

La représentation de l'arbre permet une bonne lisibilité et compréhension des critères de classification de J48. On peut voir que certaines caractéristiques typiquement attribuées aux insectes ne sont pas de bons discriminants, comme les antennes par exemple, qui sont aussi présentes chez certains myriapoda.

a	**b**	**←classified as**
91	18	**a = insects**

12	100	**b = non insects**

Précision	0.865
Rappel	0.864
F Mesure	0.864

LMT

L'algorithme LMT (Logistic Model Trees) combine arbre de décisions et modèles linéaires généralisés. Chaque feuille de l'arbre de décision construit à partir des données d'entraînement est remplacée par un modèle linéaire généralisé qui vient affiner les prédictions. Il classe correctement 94,12% des articles du wiki.

Pour chaque classe, on a une liste d'attributs avec les poids correspondants. Plus le poids est grand, plus l'attribut détermine la prédiction de cette classe.

a	**b**	**←classified as**
97	12	**a = insects**
1	111	**b = non_insects**

Précision	0,946
Rappel	0,941
F mesure	0,941

Les non insectes sont très bien classés mais les insectes moins.

Scikit learn

Scikit learn est une bibliothèque python open source de machine learning. Elle propose différents outils pour l'analyse de données et peut-être utilisée pour différentes tâches de machine learning, notamment pour la classification. Dans le cadre de notre problème de classification binaire, nous avons donc testé un des algorithmes de la famille des machine à vecteurs de support (MVS, ou *SVM* en anglais) : la classification à vecteurs de support linéaire (*Linear SVC* en anglais,

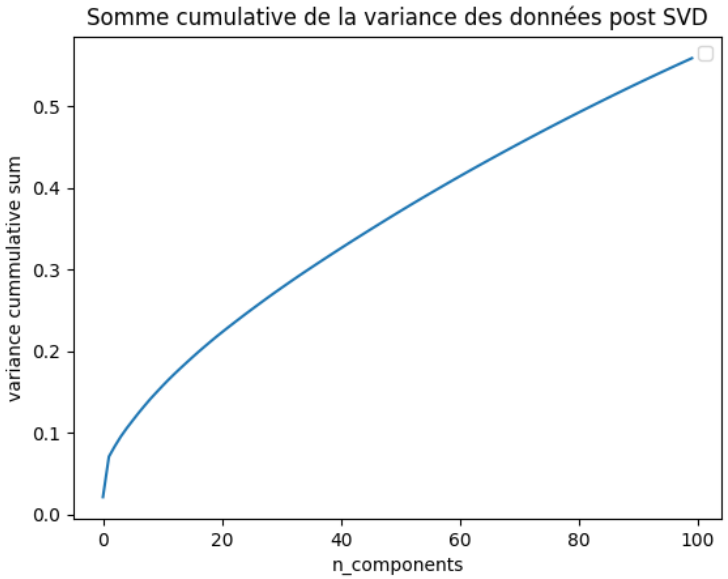
c’est ainsi qu’elle est nommé par le module scikit learn). Ce classifieur étant un séparateur binaire, il nous semblait le plus adapté à nos données.

Pré-traitement des données

Le pré-traitement de données est ici nécessaire. Pour ce faire, nous avons tout d’abord réalisé un Data Frame grâce à Pandas, dans lequel nous avons stocké toutes nos données. Celui-ci fut réalisé grâce au script create_database.py qui à partir de l’arborescence de notre dossier corpus, réalise le Data Frame comprenant la classe de l’article (label : insects, non-insects), le numéro de l’article dans notre corpus (documentID), le nom du fichier (file_name), et enfin le contenu textuel du fichier (text). Une fois le Data Frame créé, on le sauvegarde au format .csv, afin de l’utiliser dans nos scripts suivants.

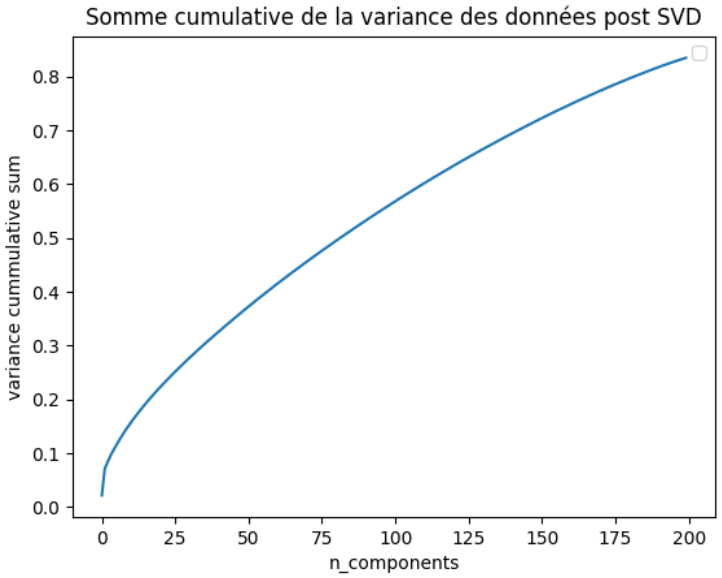
Ainsi, on nettoie et homogénéise le contenu textuel des articles en opérant directement sur la colonne ‘text’ du Data Frame obtenu : on retire les sauts de ligne, les caractères spéciaux, et les hyperliens. Certains des articles récoltés ici étant relativement courts, nous avons jugé qu’il était peut-être plus judicieux de ne pas opérer sur les stopwords avec NLTK (avec le recul, c’était peut-être une erreur !).

Suite à ce nettoyage, on applique donc le TF-IDF, qui retourne nos données vectorisées. Cependant, un nouveau problème apparaît : les vecteurs retournés présentent un trop grand nombre de dimensions, ce qui risque de poser problème lors de l’apprentissage. Ainsi, on s’intéresse à la réduction de dimension : on réalise celle-ci en appliquant une Décomposition en Valeurs Singulières (SVD en anglais, la fonction utilisée ici est TruncatedSVD). Derrière cette réduction de données, on souhaite également augmenter l’interprétabilité de nos données tout en conservant un maximum d'informations. On réduit premièrement le nombre de composantes (n_components) à 100. Si on s’intéresse ensuite à la variance des données d’entraînement transformées par une projection de chaque composante, on obtient le graphe suivant :



On interprète ce graphique de la façon suivante : seulement un peu plus de 50% de la variance des données transformées sont stockées dans les 100 premières composantes. On en conclut qu'on perd donc beaucoup d'informations à travers cette transformation. On suppose qu'en augmentant le nombre de composantes, on parviendra à éviter ce problème.

Ainsi, pour n_components=200, on obtient :



Avec 200 composantes, on récupère donc bien un plus grand pourcentage de la variance. Cependant, nous avons pu remarquer que les modèles entraînés étaient nettement moins performants suite à l'application d'une SVD avec 200 composantes. Il s'agit ici d'opter pour un compromis : finalement on appliquera à nos données une SVD en ne conservant que 100 composantes.

LinearSVC

Présentation de l'algorithme

L'apprentissage par un SVM consiste à établir un hyperplan entre les points que représentent nos données, de façon à les grouper en classe. Cet apprentissage repose sur les vecteurs de support : c'est la séparation entre ces deux vecteurs qu'on cherche à maximiser.

La fonction LinearSVC implémentée par scikit learn permet d'aller au delà de la fonction SVM : tandis que SVM permet d'être paramétrée de façon à effectuer une classification linéaire (avec le paramètre kernel="linear"), LinearSVC vient avec toute une panoplie de paramètres permettant à l'utilisateur une meilleure flexibilité quant au choix de la loss, et de la pénalité (penalty). Tandis que le choix de la loss aura un impact sur la mesure de la qualité de nos résultats, la pénalité (penalty) impose à nos résultats des contraintes, de façon à éviter l'overfitting. Cette dernière, qu'on ajoute à la loss, opère en pénalisant les coefficients de régressions les plus importants : il s'agit de réduire le nombre de paramètres et de "simplifier" le modèle, pour ainsi obtenir de meilleures prédictions et être plus performant. La fonction LinearSVC propose deux pénalités différentes :

- L1 : une pénalité L1 est égale à la valeur absolue de l'amplitude des coefficients,
- L2 : une pénalité L2 est égale au carré de l'amplitude des coefficients.

Par défaut, la fonction LinearSVC ajoute à sa loss une pénalité L2. Nous avons conservé ce paramètre par défaut. Les fonctions loss proposées sont 'hinge' et 'squared_hinge', toutes deux des alternatives à la cross-entropy pour les problèmes de classifications binaires. Par défaut, on choisit la fonction 'hinge' définie ainsi : $l(y) = max(0, 1 - t \cdot y)$, avec $t = \pm 1$, la classe désirée en sortie, et $y = w \cdot x + b$ où (w, b) sont les paramètres de l'hyperplan, et x l'entrée.

Implémentation en Python

L'implémentation du modèle LinearSVC est très intuitive : c'est là la puissance de Scikit learn. On sépare tout d'abord nos données de façon à avoir un jeu pour l'entraînement et un autre pour tester le modèle.

```
X_train, X_test, y_train, y_test = train_test_split(X, y,
test_size=0.30)
```

On définit notre modèle de classification binaire grâce à la fonction LinearSVC mentionnée précédemment :

```
clf = LinearSVC(C=1.0, tol=0.001, random_state=42, loss="hinge")
```

Enfin, on peut entraîner le modèle sur nos données, avant de pouvoir finalement prédire les résultats sur notre jeu de test.

```
clf = clf.fit(X_train, y_train)
y_pred = clf.predict(X_test)
```

Des paramètres optimaux ?

Afin de déterminer quels seront les paramètres optimaux pour le modèle qu'on cherche à implémenter, Scikit learn propose une méthode particulièrement pratique : la HalvingGridSearchCV. Cette fonction prend en argument un dictionnaire de paramètres que l'utilisateur cherche à varier, et le modèle intéressé. Bien qu'elle puisse être coûteuse en temps de calcul et ressource, elle présente l'avantage de renvoyer le modèle le plus performant sur nos données.

Par curiosité, nous avons donc testé cette méthode ! Cependant, après exécution du programme, les résultats obtenus par le modèle estimé le meilleur (best_estimator_) sur nos données, n'étaient pas aussi élevés que nous le souhaitions.

```
# paramètres variés
params = {
    "penalty":["l2"],
    "loss":["squared_hinge", "hinge"],
    "max_iter": np.arange(400, 1200, 100),
    "C": np.arange(0.1, 0.5, 0.05)
}
```

```
search = HalvingGridSearchCV(clf, params, resource='n_samples',
                             factor=2,
```

```
                                    random_state=0,
                                   n_jobs=-1).fit(X, y)
```

Le modèle finalement retenu par la fonction search :

```
print(search.best_estimator_)
>>> LinearSVC(C=0.20000000000000004, max_iter=1100)
```

Ce dernier nous permet d'obtenir le résultat ci-dessous :

```
accuracy = accuracy_score(y_test, y_pred)
print(accuracy)
>>> 0.8656716417910447
```

On note cependant quelques problèmes : en passant l'entièreté de nos données dans l'entraînement des modèles de search, on se retrouve à faire prédire au modèle des résultats sur lesquels il s'est déjà entraîné. Par ailleurs, la gestion du paramètre C (paramètre de régularisation), nous semblait plus délicate.

Finalement, nous sommes donc restées sur le modèle présenté précédemment (voir plus haut, Implémentation en python).

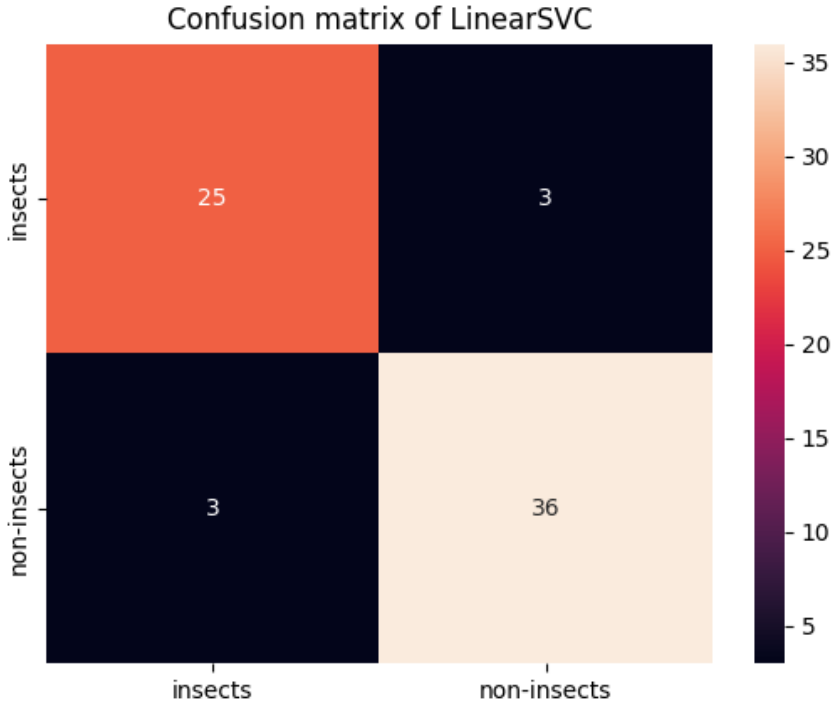
Résultats

On mesure les performances de notre modèle sur les prédictions grâce à l'accuracy, que l'on va mesurer pour 15 data split différents (c'est-à-dire qu'on exécute les lignes de code présentées dans Implémentation en python 15 fois). On calcule finalement la moyenne et l'écart-type de ces 15 accuracies :

- moyenne : $\overline{a} = 0.900$
- écart-type : $\sigma_a = 0.035$
- variance : $V(a) = 0.001$

Bien que la dispersion des résultats d'un data split à un autre soit inévitable, le modèle effectue des prédictions avec une précision moyenne de 0.90. Nous ne sommes pas parvenues à obtenir de résultats plus élevés pour ce modèle de SVM, mais les résultats tirés du modèle le plus "précis", restent néanmoins satisfaisants.

À l'aide du module Seaborn, on visualise la matrice de confusion du modèle présentant la meilleure accuracy ($a = 0.955$) :



Encore une fois, Scikit learn se montre très pratique grâce à sa fonction *classification_report*, du sous-module *metrics*. Cette dernière nous permet finalement d'obtenir la grille de mesures suivante :

	precision	recall	f1-score
insects	0.91	1.00	0.95
non-insects	1.00	0.92	0.96
accuracy			0.96
macro avg	0.95	0.96	0.95
weighted avg	0.96	0.96	0.96

Pour ce modèle, seulement 3 articles "insects" sont prédits "non-insects" : de la même façon, 3 articles "non-insects", sont prédits "non-insects". Cependant, le nombre d'articles "insects" et "non-insects" étant inégal dans ce jeu de test (28 "insects" contre 39 "non-insects"), on en conclut que le modèle parvient un peu mieux à classer les articles "non-insects", que les "insects". Malgré cela, la diagonale (noire) reste plutôt bien minimisée.

Il est évident que ces résultats auraient pu être optimisés, notamment avec un meilleur pré-traitement (peut-être aurait-il fallu davantage jouer avec les paramètres de la fonction TF-IDF, revoir la réduction de données…), ou bien encore avec un

corpus plus important (malheureusement, nous ne disposions ici que de 221 articles). Par ailleurs, la taille inégale des articles pourrait également avoir une influence sur les résultats présentés ici.

Conclusion

Ce projet fut l'occasion pour nous de nous familiariser avec de nombreux algorithmes de classification proposés par Weka ainsi que les procédures à suivre lorsqu'on cherche à implémenter un classifieur en Python, avec Scikit learn. Si l'algorithme NaiveBayesMultinomial Updateable obtient ici les meilleurs résultats, il est évident que nous aurions pu améliorer ces derniers avec un corpus plus important (et plus homogène !), et peut-être aussi en mettant plus l'accent sur le pré-traitement des données avant que celles-ci soient vectorisées pour Weka et sklearn.

La méthode implémentée avec scikit learn, bien qu'intéressante quant aux techniques sur lesquelles elle repose, nous donne finalement moins d'informations sur la classification de nos articles. Peut-être aurait-il fallu opter pour une autre fonction, ou bien davantage regarder les attributs de LinearSVC afin de mieux comprendre quels termes étaient jugés importants.

Ceci étant dit, les algorithmes que nous avons testé sur Weka nous ont permis de discerner des critères de classification des insectes auxquels nous n'avions pas pensé. Par exemple, contrairement à ce que l'on pourrait penser, la présence d'antennes ne suffit pas pour déterminer si l'espèce considérée appartient à la classe des insectes ou non ! En revanche, dans le contexte du wiki, la présence d'ailes (wings) implique nécessairement que celle-ci appartient à la classe des insectes.

De ce projet, nous tirons donc des conclusions intéressantes sur la classification des insectes, ainsi que de nouvelles connaissances sur les algorithmes de classification.

Ressources

WekaManual-3-8-3 https://waikato.github.io/weka-wiki/documentation/
insect wiki https://insects.fandom.com/wiki/Insect_Wiki
Regularization: Simple definitions, L1 & L2 penalties (https://www.statisticshowto.com/regularization/)
Scikit learn documentation: Linear SVC (https://scikit-learn.org/stable/modules/generated/sklearn.svm.LinearSVC.html)