RAPPORT PROJET DE SESSION

ANALYSE DE SENTIMENT : IMDB REVIEWS DATASET

Lucas Gonthier - Matricule : 21 155 376

Adrien Ingarao - Matricule : 21 099 794

IFT 712 - Techniques d'apprentissage

Université de Sherbrooke

Automne 2021

Table des matières

ABSTRACT

Le traitement automatique des langues (*natural language processing*) est un domaine de recherche majeur en apprentissage automatique. Dans le cadre du cours IFT-712 de l'université de Sherbrooke, nous allons étudier un sous-ensemble de ce domaine, l'analyse de sentiment. L'objectif de ce projet est de comparer plusieurs modèles d'apprentissage automatique sur un jeu de données d'une taille de 50 000 commentaires de films. Nous faisons dans ce projet, la revue de certaines techniques basiques de *NLP*.

Keywords Analyse de Sentiment · Prétraitement · Apprentissage Automatique · NLP

1 Présentation du sujet

1.1 Description des données

Le jeu de données représente 50 000 commentaires en anglais de films provenant de la base de données de film IMDb. Ce jeu de données a été construit principalement à des fins de recherches. Le jeu de données est également disponible sur **Kaggle** (https://www.kaggle.com/lakshmi25npathi/imdb-dataset-of-50k-movie-reviews).
Les commentaires ont été classés en positif ou négatif. Les données sont balancées, il y a autant de positif que de négatif. L'objectif est de déterminer si un commentaire est négatif ou positif en se basant sur son contenu textuel. Ce type de classification binaire s'appelle l'analyse de sentiment (sentiment analysis).

	review	sentiment
0	One of the other reviewers has mentioned that ...	positive
1	A wonderful little production. The...	positive
2	I thought this was a wonderful way to spend ti...	positive
3	Basically there's a family where a little boy ...	negative
4	Petter Mattei's "Love in the Time of Money" is...	positive

Figure 1: Exemple des données au format colonne

1.2 Analyse des sentiments

Comme son nom l'indique l'analyse des sentiments a pour objectif de déterminer si le sentiment dégagé par un texte est positif ou négatif. Le sentiment dégagé par un texte dépend d'une multitude de facteurs tels que le contexte, la langue utilisée et bien d'autres. Bien que l'analyse soit simple de compréhension, son utilisation permet de résoudre des problèmes complexes. Par exemple, de nombreuses études montrent qu'il y a une forte corrélation entre les tweets et l'évolution des valeurs boursières. Dans notre cas, l'analyse des sentiments des commentaires de film pourrait aider à prédire le box-office d'un film par exemple.

2 Gestion de projet & Design

2.1 Gestion de projet

Dans le but d'avoir une gestion de projet en accord avec les bonnes pratiques, nous avons utilisé le gestionaire de version **Git** avec le service web **Github**. Lors du développement de notre projet, nous avons adopté la pratique d'une branche par fonctionnalité et nous avons essayé de nous y conformer au maximum. Lorsqu'une fonctionnalité était terminée et validée par un membre du groupe alors nous procédions au merge sur la branche main. De plus, les commits étaient fait régulièrement, toujours dans le but d'avoir une bonne gestion de projet.
Le code du projet est disponible à l'adresse suivante : https://github.com/lugonthier/IMDB_reviews.
Pour se servir du projet, le fichier **README.md** du projet fournit les explications. Egalement, en annexe A de ce rapport se trouve des explications pour faire fonctionner une simple validation croisée.
Aussi voici les noms d'utilisateurs git de ce projet :

- **Adrien Ingarao**
  - inga2603 (compte gitlab universitaire)
- **Lucas Gonthier**
  - lugonthier (compte github personnel)
  - gonl3002 (compte gitlab universitaire)

2.2 Design

2.2.1 MLflow

Lorsque l'on fait des experimentations en apprentissage automatique, il est récurrent de ne plus savoir où l'on en est dans nos experimentations et notre démarche. L'utilisation des notebooks accentue ce désordre si on ne les utilise pas de façon rigoureuse. C'est pourquoi nous avons décidé d'utiliser le projet open source **MLflow**. Ce projet est une plateforme pour gérer le cycle de vie de l'apprentissage automatique. Nous en avons fait une utilisation à des fins d'experimentations.
Plus particulièrement, nous avons utilisé la partie **Tracking** de MLflow. Cela nous permet d'exécuter nos scripts, les résultats sont alors enregistrés automatiquement et sont disponibles dans une hiérarchie de dossier(dans le dossier **mlruns**) ou alors via une interface utilisateur (en local ou en ligne). Un exemple des résultats sous interface utilisateur est disponible en annexe B.
Les résultats peuvent inclure ce que l'on veut, tels que les hyperparamètres du modèle et les scores sur différentes métriques. Par la suite, depuis l'interface, il est possible de télécharger les résultats au format csv, ce que nous avons fait. A chaque étape où nous faisons des experimentations, nous téléchargons les résultats puis nous les analysons dans des notebooks.

2.2.2 Structures des modèles

Dans cette même vision d'exigeance, nous avons divisé nos modèles en plusieurs entités sous forme de classe Python. Dans cette sous-section, lorsque l'on parle de modèle, nous faisons référence à un ensemble comprenant les étapes de prétraiement et le modèle d'apprentissage automatique.

Les modèles sont donc divisés en 3 entités :

- La classe **TextPreprocessor** : Cette classe s'occupe de tout le prétraitement du texte sauf de la vectorisasion. La description de son champ d'actions est présente à la sous-section 5.1.
- La classe **Vectorizer** : Cette classe est celle qui va s'occuper de la vectorisation du texte. Elle est globale et peut être utilisée pour accéder à plusieurs types de vectorisation. Les différentes vectorisations utilisées sont décrites à la sous-section 5.2
- Le modèle d'apprentissage automatique, provenant de *scikit-learn*. C'est un modèle classique d'apprentissage automatique, par exemple une régression logistique. Les modèles que nous utilisons sont décrits à la section 6.

Par la suite ces 3 entités sont concaténés dans une **pipeline**. Nous utilisons la pipeline de *scikit-learn*. L'utilisation de la pipeline est décrite à la section 5.3.
Et pour finir cette pipeline est donnée à une autre classe **Experiment**. Cette classe basée sur *MLflow*, va nous servir à entraîner et évaluer le modèle (toute la pipeline) et à enregister les résultats via *MLflow*.

2.2.3 Scripts

Pour lancer nos experimentations, nous avons créé 3 scripts :

- **validation.py** ce script permet de :
  - Faire une validation croisée.
- **size_evaluation.py** ce script permet de :
  - Evaluer l'impact du nombre de dimensions.
  - Evaluer l'impact du nombre de données d'entraînement.
- **hyperparameter_search.py** ce script permet de:
  - Faire une recherche d'hyperparamètres.

Pour chaque script éxécuté, nous devons passer certains arguments comme le modèle que l'on souhaite expérimenter.

3 Démarche scientifique

Afin d'avoir des résultats de classification satisfaisant, nous avons mis une démarche en place, qui bien sûr sera susceptible de changer selon les résultats :
Etape 1 : Afin d'avoir une première idée, nous allons appliquer une première validation croisée sans prétraitement approfondi.
Etape 2 : Nous allons essayer de trouver le meilleur prétraitement possible selon les modèles.
Etape 3 : Nous allons évaluer les performances des modèles selon le nombre de dimensions. En effet, notre méthode de prétraitement que nous expliquons dans la partie 5 génère beaucoup de dimensions. Premièrement, les modèles peuvent performer différemment selon le nombre de dimensions. Aussi, s'il est possible de réduire le nombre de dimensions, cela permettra d'avoir des modèles plus rapides et donc d'effectuer l'étape de recherche d'hyperparamètres plus rapidement.
Etape 4 : Nous ferons une recherche d'hyperparamètres pour chacun des modèles.
Etape 5 : Nous ferons une nouvelle validation croisée avec exactement les mêmes données (composition des **folders**) qu'à l'étape 1 et 2 afin de comparer l'évolution des performances des modèles.
Etape 6 : Enfin, nous pourrons tester nos modèles sur le jeu de données test que nous avons laissé de côté.

4 Gestion des données

En ce qui concerne la gestion des données, nous avons séparé les données en 2 ensembles de données :

- Un ensemble d'entraînement.
- Un ensemble de test.

Les données ont été séparées à l'aide de la fonction **train_test_split** de *scikit-learn*. Afin d'effectuer des validations croisées, les données d'entraînement ont été étiquetées **train** ou **test** pour chaque **folder**.

	review	sentiment	fold_5	fold_4	fold_3	fold_2	fold_1
0	That's what I kept asking myself during the ma...	negative	train	test	train	train	train
1	I did not watch the entire movie. I could not ...	negative	train	train	train	test	train
2	A touching love story reminiscent of In the M...	positive	train	train	train	test	train
3	This latter-day Fulci schlocker is a totally a...	negative	train	train	train	test	train
4	First of all, I firmly believe that Norwegian ...	negative	train	test	train	train	train

Figure 2: Exemple des données séparées en train/test pour chaque folder

Le choix de décider l'appartenance de chaque donnée à un **folder** à ce stade, nous permet de pouvoir comparer les résultats de validation croisée de chaque modèle (et chaque version des modèles) sur les **mêmes** données. De plus, cela permet de satisfaire le critère de **reproductibilité**, une condition essentielle dans le processus d'amélioration perpétuelle des connaissances scientifiques.

5 Prétraitement

La plupart des algorithmes d'apprentissage ne fonctionnent pas bien (parfois ne prennent pas en charge) les données textuelles. Il est donc nécessaire de les transformer afin de pouvoir les fournir à de tels algorithmes d'apprentissage. De plus, il est préférable (nécessaire) de transformer les données afin d'en faire ressortir le plus d'informations possibles. Notre approche est basée sur la transformation du texte en tokens. Ces tokens correspondent aux mots et n-grams (suite de n mots). Les tokens constitueront les dimensions de nos données.

5.1 Prétraitement du texte

Avant de transformer le texte en tokens, nous avons nettoyé le texte.

5.1.1 Ponctuation

Chaque texte s'est vu retirer la ponctuation. En effet, si parfois la ponctuation peut servir et apporter de l'information (en particulier du sens), nous avons réalisé qu'elle n'apportait pas d'informations discriminantes à nos algorithmes.

5.1.2 Majuscule vs Minuscule

La plupart des méthodes de NLP (Natural Language Processing), en particulier celles que nous utilisons ici, se basent sur la fréquence des mots. Par conséquent, il semble évident que « The » et « the » doivent compter pour le même mot. Il est alors naturel de vouloir mettre tous les caractères en minuscule. Cependant, il peut être pertinent de garder certaines majuscules. En effet, de nombreux noms propres sont aussi des mots communs. Par exemple, si un commentaire cite un acteur ou un personnage portant le nom de famille « Good », on ne voudrait pas qu'il soit confondu avec le mot « good ». Malheureusement, de telles distinctions sont souvent compliquées, il est nécessaire d'extraire au maximum le sens du texte. C'est pourquoi, nous avons décidé dans un premier temps, de transformer tous les caractères en minuscules, en supposant que l'impact des noms propres sera négligeable. Si jamais, il arrivait que les modèles ne soient pas assez performants, nous prendrons en considération un moyen de conserver les majuscules utiles.

5.1.3 Stop words

Les « stop words » sont tous les mots très répétitifs, ils sont les plus communs. Par exemple :

- « Le », « la », « à » sont des stop words de la langue française.
- « a », « the », « in » sont des stop words de la langue anglaise.

Les stop words sont parfois retirés (souvent) lors de la phase de prétraitement. Cependant, dans les cas d'analyse des sentiments, ils peuvent apporter de l'information. C'est pourquoi, nous avons décidé d'essayer les 2 cas, avec et sans stop words.

5.1.4 Normalisation

Par normalisation, nous entendons les techniques telles que « **lemmatization** » et « **stemming** ».

Le **stemming**, est une technique de normalisation du vocabulaire. Le but est d'éliminer certaines différences qui n'apportent pas ou peu d'informations et qui empêchent les mots d'être considérés ensemble. Par exemple, les différentes pluralités à la fin des mots ou encore les verbes sous différentes formes. Par exemple, les mots « cat » et « cats » devraient être considérés ensemble. L'objectif est donc d'extraire la racine (stem, d'où le terme **stemming**) des mots. Cette technique est particulièrement compliquée à implémenter. Heureusement, la bibliothèque *nltk* fournit des stemmers nous permettant d'utiliser facilement cette technique.

La **lemmatization**, est une technique qui va un peu plus loin que le stemming. La différence est que la **lemmatization** prend en considération le sens des mots et permet d'associer ensemble des mots qui ont un sens significativement proche. Par exemple, les mots « better » et « good » pourraient être considérés ensemble. Cette technique encore plus complexe à implémenter est également disponible dans la bibliothèque *nltk*.

Nous avons en particulier utilisé deux techniques présentent dans *nltk*, un stemmer, **PorterStemmer** et un lemmatizer, **WordNetLemmarizer**. Notre stratégie est de tester si le stemmer, le lemmatizer ou ne rien appliquer est le plus efficace.

Toutes les techniques de prétraitement citées précédemment, sont particulièrement importantes. Elles permettent notamment de réduire le nombre de tokens et donc la dimension. Elles permettent également de réduire le surapprentissage et parfois de réduire le temps d'entraînement. Et surtout cela peut augmenter les performances des modèles.

5.2 Vectorisation

Après avoir transformé le texte, comme nous l'avons expliqué dans la sous-section précédente, il faut mettre le texte sous une forme compréhensible pour les algorithmes d'apprentissage. Notre stratégie est de « vectoriser » le texte grâce aux tokens. Chaque token correspond à une dimension. Les valeurs qui sont affectées aux variables pour chaque objet dépend de la technique utilisée. Il en existe des particulièrement connues et fonctionnant très bien avec la plupart des modèles.

5.2.1 Count vectorizer

Cette technique est particulièrement simple, elle consiste à transformer les données textuelles en matrice de tokens avec pour valeur le nombre d'occurrences de token dans le texte.

Par exemple, si on considère les textes suivant :

- Texte 0 : « The last Star Wars movie was amazing. I hope the next movie will be great too. »
- Texte 1 : « He always plays in great movie. »

Alors, la matrice correspondante sera :

	always	amazing	be	great	he	hope	in	last	movie	next	plays	star	the	too	wars	was	will
0	0	1	1	1	0	1	0	1	2	1	0	1	2	1	1	1	1
1	1	0	0	1	1	0	1	0	1	0	1	0	0	0	0	0	0

Figure 3: Exemple de vectoriation avec le Count Vectorizer

5.2.2 TF-IDF Vectorizer

Tf-idf (ou term-frequency times inverse document-frequency), est une technique ayant pour but de pondérer les tokens. Plus les tokens apportent peu d'information, plus ils auront un poids faible.

La formule pour calculer la matrice de Tf-idf est la suivante :

$$tf\text{–}idf(t,d) = tf(t,d).idf(t) \quad avec \begin{cases} tf(t,d) \text{ la fréquence du token } t \text{ dans le document } d. \\ idf(t) = log\frac{1+n}{1+df(t)} + 1 \\ n \text{ le nombre de texte dans tout l'ensemble.} \\ df(t) \text{ le nombre de document qui contiennent le token t.} \end{cases} \tag{1}$$

5.3 Pipeline

Une pipeline est une technique qui va venir encapsuler tout le processus, du prétraitement jusqu'au modèle d'apprentissage. Une pipeline est composée de N étapes, N-1 transformateurs et 1 dernière étape qui est le modèle.

Figure 4: image de *scikit-learn* décrivant une pipeline

Au-delà du côté pratique qui permet de tout exécuter d'un coup, la pipeline est essentielle pour certaines techniques telles que la validation croisée. En effet, cela va permettre d'éviter les fuites de statistiques sur les données de test et de garantir que les mêmes échantillons sont utilisés pour former les transformateurs et les

modèles. Si l'on applique les transformateurs sur tout l'ensemble de données, alors on peut s'attendre à une fuite de données (**data leakage**).

6 Modèles

6.1 Régression Logistique

Le premier modèle que nous avons choisi est la régression logistique. Ce modèle est un classifieur linéaire qui se compose d'un unique neurone.

La sortie du réseau est le résultat d'un produit scalaire entre la couche d'entrée et les poids combinés à une fonction d'activation (une sigmoïde dans notre cas). Ainsi la sortie d'un réseau logistique entraîné peut être associé à la probabilité d'appartenance d'un exemple à une classe.

Pour un réseau logistique on utilise comme fonction de coût une entropie croisée avec un terme de régularisation pour éviter le surapprentissage. Cette fonction de coût provient d'un maximum a posteriori:

$$argmin_W - \Sigma_{n=0}^{N} t_n ln(y_W(\vec{x}_n)) + (1 - t_n) ln(1 - y_W(\vec{x}_n)) + \lambda \Sigma_{i=1}^{d} (w_i)^2 \tag{2}$$

L'entraînement d'un réseau logistique se base sur une solution itérative : une descente de gradient.

Ce modèle nécessite en théorie d'avoir des données linéairement séparables pour obtenir de bonnes performances. Le choix de ce modèle va nous permettre de mesurer son efficacité sur des données textuelles.

6.2 Arbre de Décision

Le second modèle que nous avons choisi est un arbre de décision. L'objectif du modèle consiste à prédire la classe d'une donnée en se basant sur des règles de décisions apprises sur les données d'entraînement. Ces règles de décisions raisonnent sur les valeurs des variables des données. On dit qu'il s'agit d'un classifieur sans paramètres.

Les arbres de décision sont construits en divisant l'arbre du sommet vers les feuilles en choisissant à chaque étape une des variables d'entrées qui réalise le meilleur partitionnement de la base d'entraînement. Le choix de la variable de séparation sur un nœud selon un critère à maximiser. Les critères les plus souvent utilisés sont le critère de Gini et l'entropie de Shannon.

Nous avons choisi ce modèle pour différentes raisons. Les arbres de décision sont connus pour avoir tendance à créer des arbres très complexes qui ont tendance à mal généraliser, néanmoins il existe des stratégies pour éviter ce problème (pré-élagage etc..). Il serait donc intéressant d'observer les performances du modèle en fonction de différentes stratégies. Enfin, les arbres de décisions n'émettent pas d'hypothèse sur les données.

6.3 Naive Bayes Multinomial

Le troisième modèle que nous avons choisi est une implémentation de l'algorithme "Naive Bayes" pour des données distribuées selon une loi multinomiale. Pour rappel l'algorithme "Nayes Bayes" fait une hypothèse d'indépendance entre chaque paire de variables d'une donnée :

$$P(x_i|y, x_1, ..., x_{i-1}, x_{i+1}, ..., x_n) = P(x_i|y) \tag{3}$$

La distribution des données est paramétrisée par les vecteurs $\theta_y = (\theta_{y_1}, .., \theta_{y_n})$ pour chaque classe y, où n est le nombre de dimensions, et θ_{y_i} la probabilité $P(x_i|y)$ qu'une variable i appartienne à un exemple dont la classe est y. Dans notre cas c'est la probabilité qu'un mot appartienne à un commentaire (review) dont la classe est positive ou négative. Ce paramètre θ_y est estimé à l'aide d'un maximum de vraisemblance un peu modifié.

Ce modèle est très utilisé en analyse textuelle, il serait donc intéréssant d'observer ses performances sur nos données.

6.4 Forêt aléatoire

Le quatrième modèle que nous avonc choisi est une méthode d'ensemble d'arbre de décision. Le principe consiste à créer un certains nombres d'arbres de décision puis de les entrainer sur des sous ensembles de la base d'entrainement . Ces sous ensembles sont souvent construits en utilisant des techniques d'inférences comme "bootstrap". Après entrainement, la prédiction d'une forêt aléatoire consiste à faire un simple vote majoritaire.

Les forêts aléatoires sont souvent utilisées pour palier aux problèmes que rencontrent les arbres de décision simples. En effet, en combinant de l'aléatoire et un vote majoritaire, les forêts aléatoires permettent de diminuer le surapprentissage qui est un problème récurrent des arbres de décision simple. On peut donc voir ce modèle comme une amélioration de notre arbre de décision, on s'attend alors à avoir de meilleures performances en généralisation. Il serait donc intéréssant de comparer leurs performances.

6.5 SVM linéaire

Le cinquième modèle que nous avons choisi est une Machine à Vecteurs de Support (SVM) linéaire. Ce modèle se base la notion de marge et de noyau pour classifier au mieux les données. En effet, les SVM cherchent à trouver les paramètres d'un hyperplan (cas linéaire) qui maximisent la marge avec une contrainte. Dans notre cas le noyau de notre modèle est linéaire.

Nous avons choisi ce modèle pour diffèrentes raisons. En effet, après vectorisation de nos données, le nombre de dimensions correspond à l'étendu du vocabulaire utilisé dans les 50 000 commentaires. C'est un très grand nombre. Or les SVM sont en général efficaces dans des espaces à grandes dimensions. De plus, tout comme la régression logistique, un SVM linéaire fait l'hypothèse que les données sont linéairement séparables. Il serait donc intéréssant d'observer les performances du modèle sur des données textuelles.

6.6 Perceptron multicouche

Pour notre sixième modèle nous avonc choisi un perceptron multicouche. Il s'agit d'un réseau de neurones organisé en plusieurs couches. Une couche d'entrée, un certain nombre de couches cachées puis une couche de sortie. Le réseau possède des poids w associés à chaque neurones des couches cachées.

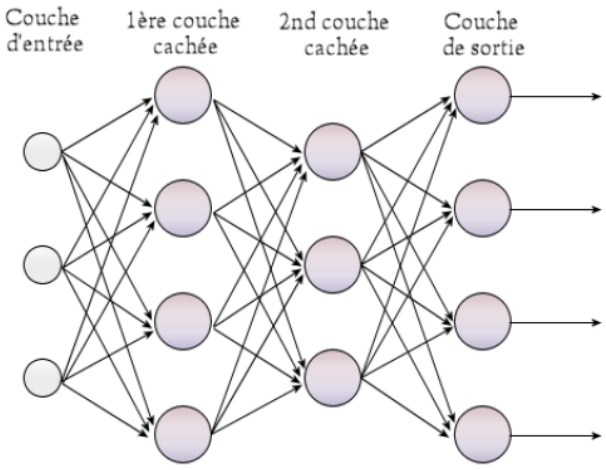

Figure 5: Exemple de Perceptron multicouche

L'objectif du modèle est d'associer à un exemple passé en entrée une classe. Le modèle utilise le mécanisme de la propagation avant pour calculer le score d'un exemple. Ce score est passé à une fonction d'activation, souvent softmax ou sigmoid (sigmoid dans notre cas). Ainsi la sortie d'un réseau de neurones peut être vu comme une distribution de probabilitées conditonnelles. La classe de l'exemple est associée à la probabilité la plus élevée.

L'entrainement du modèle consiste à apprendre les poids w en minimisant une fonction de coût. Pour cela on utilise une solution itérative : une descente de gradient. Cette dernière utilise le mécanisme de retropropagation pour mettre à jours les poids.

Nous avons choisi le perceptron multicouche pour diverses raisons. Notamment car il s'agit de l'ancêtre des réseaux de neurones modernes, il s'agit donc de la première étape indispensable si l'on souhaite s'intéresser aux réseaux de neurones actuels.

7 Evaluation des modèles

7.1 Métriques

Pour évaluer nos modèles, nous pouvons nous baser sur différentes mesures comme le **rappel** (recall), la **précision** (precision), le **score F1** et bien d'autres. Cependant, nos données sont balancées, c'est-à-dire qu'il y a autant de données pour chacune des 2 classes. De plus, nous avons fait en sorte d'avoir également des données balancées au sein de chaque **folder** pour la validation croisée, afin de garder une certaine intégrité vis à vis de la répartition des classes. C'est pourquoi, au vu de l'état de nos données, la **justesse** (aussi appelée **accuracy**) semble être un choix judicieux.

La justesse correspond simplement au nombre total de données bien prédites divisé par le nombre de données totales.

Par conséquent, nous utiliserons la justesse comme mesure principale pour évaluer nos modèles. Cependant, nous utilisons également d'autres métriques lorsque la justesse n'est plus suffisante ou pour tout simplement évaluer nos modèles sous un autre angle.

Nous précisons que nous ne faisons que des validations croisées. Ainsi, afin d'alléger les explications, parfois, nous emploieront les termes **justesse de validation** et **justesse d'entraînement** pour désigner respectivement la **justesse moyenne de validation** et la **justesse moyenne d'entraînement** lors d'une validation croisée.

7.2 Premières validations croisées

Afin d'avoir une idée des performances de nos modèles, nous avons effectué deux premières validation croisées avec 5 folds comme décrit dans la partie 4.

Pour cette première validation croisée :

- Nous n'avons pas effectué de prétraitement complexe tel que la normalisation.
- Nous avons choisi d'utiliser le count vectorizer.
- Nous avons pris les hyperparamètres par défaut de *scikit-learn* hormis pour les modèles basés sur les arbres (Arbre de décision et Forêt aléatoire) pour lesquels nous avons fixé la profondeur maximale afin d'éviter un surapprentissage.

On remarque que tous les modèles ont une justesse de validation **supérieure à 80**% hormis l'arbre de décision qui a une justesse de 72.89%. Cependant, on constate que le perceptron multicouche (mlp) et le SVM linéaire (linearsvc) ont une justesse de d'entraînement de 1, pour une justesse de validation de respectivement 89% et 87.07%. Cela témoigne d'un **surapprentissage**. On peut également noter un léger écart de justesses pour la régression logistique avec une justesse d'entraînement de 97.94% pour une validation de 88.7%.

Pour cette deuxième validation croisée, nous avons pris les mêmes conditions que la première, nous avons seulement changé le vectoriseur en prenant le TF-IDF vectorizer. Les résultats sont globalement similaires. On peut tout de même noter l'écart réduit entre la justesse d'entraînement et de validation pour la régression logistique, par rapport à la première validation croisée. Dans la suite, nous ne considérerons que le **Tfidf Vectorizer**. D'après nos recherches, il semble plus robuste et permet d'extraire des informations plus discriminantes.

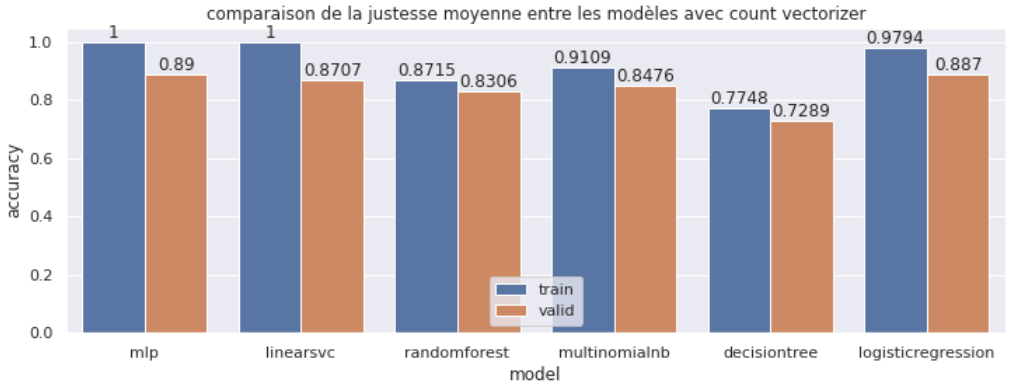

Figure 6: moyennes des justesses sur une validation croisée avec un Count Vectorizer

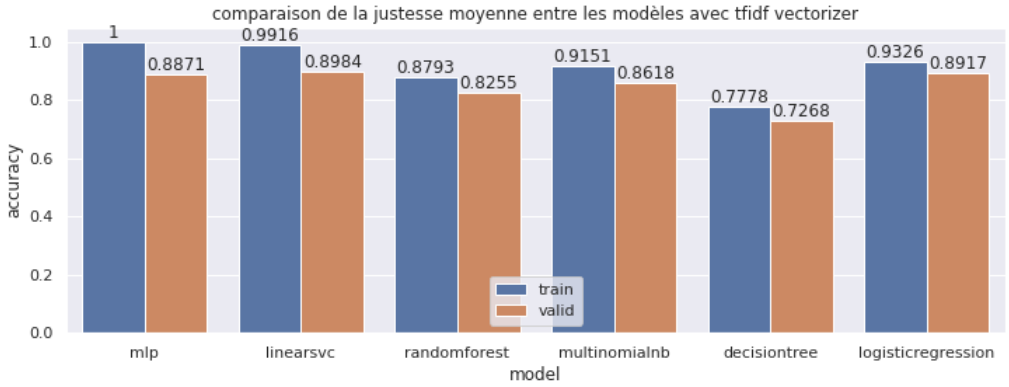

Figure 7: moyennes des justesses sur une validation croisée avec un TF-IDF Vectorizer

7.3 Recherche du meilleur prétraitement

En ce qui concerne le prétraitement, hormis la vectorisation du texte, nous avons retiré la ponctuation et passé tous les caractères en minuscules. En revanche, pour choisir entre garder les stop words et appliquer une quelconque normalisation (ou aucune), cela est beaucoup moins intuitif. C'est pourquoi nous avons émis l'idée d'essayer toutes les combinaisons de prétraitement possible. Nous pouvons choisir de garder ou non les stop words, nous avons 3 choix de normalisation (aucune, le WordNetLemmatizer, le PorterStemmer). Cela fait donc 2 x 3 = 6 combinaisons.
Idéalement, nous pourrions inclure cette recherche avec la recherche d'hyperparamètres. En effet, cela assurerait une meilleure fiabilité (en terme de performance) quant aux choix de prétraitement vis-à-vis de chaque modèle et de chaque combinaison d'hyperparamètres. Cependant, cela signifie multiplier par 6 la recherche d'hyperparamètres pour chaque modèle. Pour certains modèles comme la Régression Logistique cela est envisageable mais beaucoup moins pour les modèles tel que le Perceptron Multicouche. C'est pourquoi avant d'éventuellement mettre en place cette stratégie nous avons essayé les 6 combinaisons sur les modèles avec des hyperparamètres fixés.

Les résultats des 6 combinaisons sont extrêmement similaires, parfois même identiques pour certains modèles.

Ci-dessous les résultats des 2 meilleures combinaisons :
Malgré l'efficacité théorique de techniques comme le stemming ou la lemmatization, en pratique nous n'avons pas de meilleurs résultats évidents par rapport à l'application d'aucune technique de normalisation (telle qu'à la sous partie 7.2).

	model	train accuracy	validation accuracy
0	mlp	1.0000	0.8764
1	linearsvc	0.9877	0.8908
2	randomforest	0.8789	0.8292
3	multinomialnb	0.9100	0.8599
4	decisiontree	0.7864	0.7340
5	logisticregression	0.9301	0.8898

Figure 8: Moyennes des justesses sur validation croisée avec PorterStemmer et sans stop words

	model	train accuracy	validation accuracy
0	mlp	0.9999	0.8830
1	linearsvc	0.9921	0.8942
2	randomforest	0.8788	0.8298
3	multinomialnb	0.9177	0.8641
4	decisiontree	0.7783	0.7329
5	logisticregression	0.9348	0.8909

Figure 9: Moyennes des justesses sur validation croisée avec WordNetLemmatizer et sans stop words

Nous émettons certaines hypothèses pour justifier un manque de performance en utilisant les techniques de normalisation :

- La première est que nous étudions des commentaires de films. La plupart de ces commentaires sont courts.
- Jusqu'à présent nous utilisions tous les mots des données d'entraînements (sauf lorsqu'on retire les stop words), et ceux même s'il n'y a qu'une occurrence. Par conséquent, le nombre de dimensions est très grand et atteint plus de 100 000. L'application d'une normalisation pourrait se montrer insignifiante face à autant de dimensions.

Dans la suite, nous enlevons les stop words, cela permet de réduire un peu le nombre de dimensions. Aussi, nous utilisons le PorterStemmer qui permettrait éventuellement de réduire les dimensions. De plus, le PorterStemmer est beaucoup moins coûteux en temps de calcul que le WordNetLemmatizer. La prochaine étape pourrait nous éclairer quant à la véracité de notre deuxième hypothèse.

7.4 Evaluation des performances selon le nombre de dimensions

L'étude des performances des modèles selon le nombre de dimensions peut répondre à plusieurs problématiques.
Premièrement, une partie des algorithmes d'apprentissage automatique deviennent de plus en plus lent lorsque le nombre de dimensions augmente. Réduire le nombre de dimensions permettrait de rendre plus rapide certains procédés tels que la recherche d'hyperparamètres. Aussi, pour des modèles destinés à être deployés en production, le temps d'entraînement ou de prediction peut être un enjeu majeur selon l'utilisation que l'on veut en faire.
Une grande dimensionnalité peut aussi empêcher de converger vers une bonne solution. En effet, certaines dimensions pourraient n'être que du bruit. Cela pourrait causer un sous apprentissage ou un surapprentissage. Ce problème est connu sous le nom de **malédition de la dimensionnalité** (*the curse of dimenstionality*).

Afin d'étudier l'impact de la dimensionnalité sur les performances des modèles (en terme de **justesse** et de **généralisation**), nous avons effectué une validation croisée (Avec les mêmes 5 **folds** que précédemment) pour chaque nombre de dimension différent sélectionné. Plus particulièrement, nous avons pris un nombre de dimensionalité suivant une suite géométrique partant de **500 dimensions** avec une **raison de 2** (en s'arretant avant 100 000 dimensions). Par conséquent, les modèles ont été évalués sur les dimensions suivantes : **500**, **1000**, **2000**, **4000**, **8000**, **16000**, **32000**, **64000**.
Pour sélectionner les dimensions que l'on prend parmi toutes celles disponibles, nous utilisons le paramètre **max_features** du TF-IDF Vectoriser. Cela a pour effet de ne retenir que les **max_features** mots les plus fréquents parmi tous les textes. (Pour rappel, les mots sont nos dimensions).

Ci-dessous 2 graphes de l'évolution de la **justesse** en fonction du nombre de dimensions. (Nous avons séparé les résultats en 2 graphes pour une meilleure lisibilité). Egalement, en annexe D les résultats sont sous la forme de tableau, avec les valeurs exactes.

Le graphe de la **figure 10**, montre les résultats des modèles suivant : **Arbre de Décision** (decisiontree), **SVM Linéaire** (linearsvc), **Régression Logistique** (logisticregression).

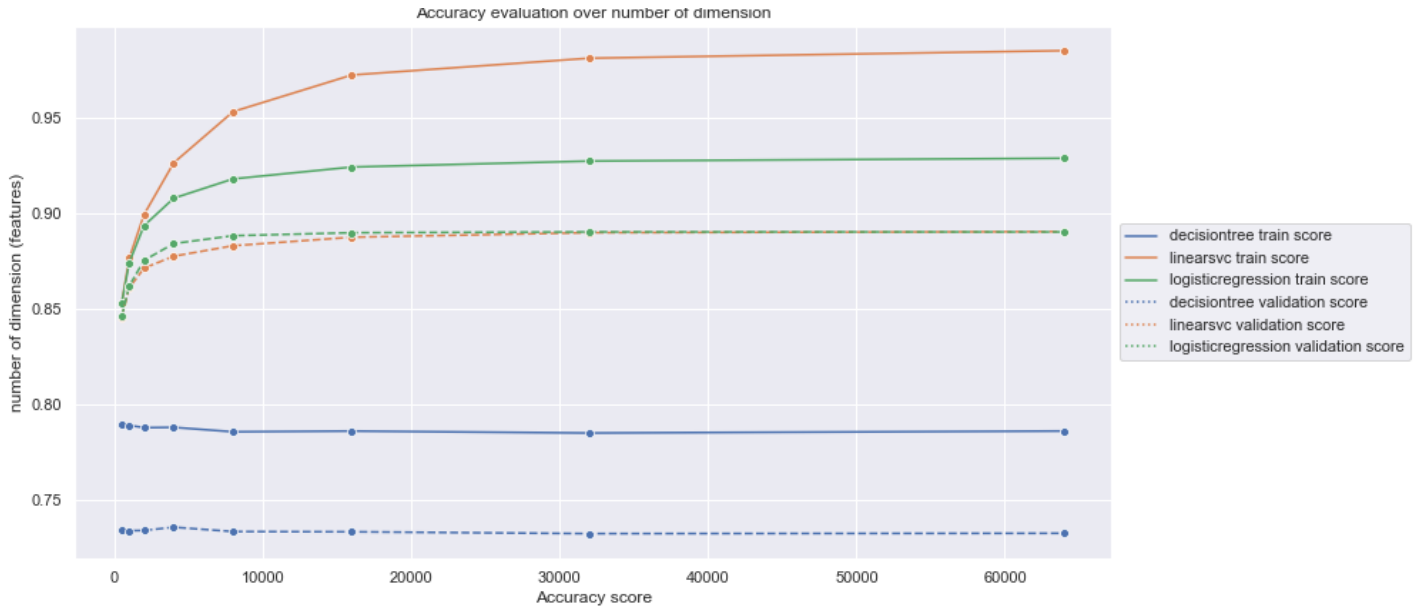

Figure 10: moyennes des justesses sur une validation croisée avec un TF-IDF Vectorizer

Sur ce premier graphe, on peut faire plusieurs remarques :

- **Arbre de Décision** :
  - Les 2 courbes (bleues) augmentent à la même allure sans creuser d'écart jusqu'à 4000 dimensions. Puis la courbe d'entraînement (train) continue d'augmenter alors que la courbe de validation décroit doucement.
  - On va donc prendre **4000** dimensions pour ce modèle.
- **Régression Logistique** :
  - Au début, à 500 dimensions, il y a un très faible écart entre les courbes d'entraînement et de validation. Puis l'écart augmente jusqu'à 16 000 dimensions puis se stabilise.
  - Les 2 courbes continuent de croître jusqu'à 32 000 dimensions. Par la suite, la courbe de validation décroit alors que celle d'entraînement augmente.
  - On va donc prendre **32 000** dimensions pour ce modèle.
- **SVM Linéaire**
  - L'écart est faible avec 500 dimensions. Puis l'écart se creuse jusqu'à ce que la courbe d'entraînement sature après 8000 dimensions à plus de **95.34%** tandis que la validation est alors de **88.31%**.
  - On va donc prendre **8 000** dimensions pour ce modèle.

Le graphe de la **figure 11** montre les résultats des modèles suivant : **Perceptron Multicouche** (mlp), **Naive Bayes Multinomial** (multinomialnb), **Forêt Aléatoire** (randomforest).

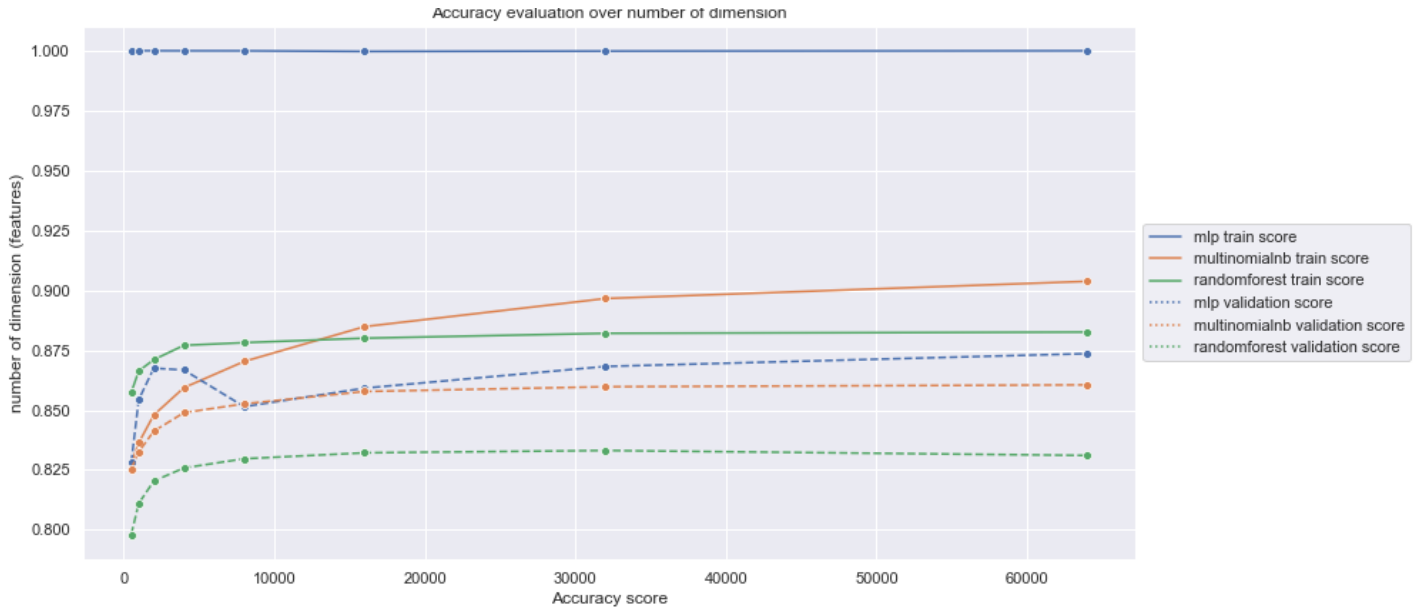

Figure 11: moyennes des justesses sur une validation croisée avec un TF-IDF Vectorizer

Sur ce deuxième graphe, on peut faire les remarques suivantes :

- **Perceptron Multicouche**
  - On remarque que la courbe d'entraînement sature à 100% pour n'importe quel nombre de dimension. Il faut trouver un autre moyen de réduire ce qui s'apparente à du surapprentissage.
  - La courbe de validation après un pic autour de **1 000** dimensions, augmente progressivement sans s'arrêter.
  - Le choix du nombre de dimensions est compliqué car l'étude pour ce modèle n'a pas apporté beaucoup d'informations. Les 3 meilleurs résultats de validation sont **86.75%**, **86.82%** et **87.36%** pour respectivement 2000, 32 000 et 64 000 dimensions.
  - Nous faisons le choix de 2000 car le temps d'entraînement moyen est de 100 secondes seulement, il est 3 fois plus important pour 32 000 dimensions et plus, cela serait handicapant pour la recherche d'hyperparamètre.
- **Naive Bayes Multinomial**
  - L'écart entre les courbes est faible au début puis augmente au fur et à mesure que le nombre de dimensions augmente.
  - Cependant, les courbes continuent d'augmenter et l'écart reste negligeable, **90.37%** pour l'entraînement et **86.05%** à 64 000 de dimensions.
  - On va donc conserver le **maximum** de dimensions (environ 100 000 dimensions pour 40 000 données d'entraînement). On suppose qu'avec plus de dimensions, les courbes continueront d'augmenter.
- **Forêt Aléatoire**
  - L'écart entre les courbes restent assez constant jusqu'à 32 000 dimensions. Après 32 000 dimensions, l'écart augmente.
  - Cela est principalement dû au fait que la courbe de validation décroît.
  - On va donc prendre **32 000** dimensions.

7.5 Recherche d'hyperparamètres

L'intérêt de trouver les meilleurs hyperparamètres est de maximiser une métrique (ou minimiser une fonction de perte). C'est pourquoi dans l'idéal, nous pourrions inclure, en plus des hypeparamètres des modèles, les paramètres de toute la pipeline. En effet, les performances dépendent d'une multitude de facteurs. Par exemple, un modèle pourrait être plus performant avec une certaine normalisation alors qu'un autre non. Il en va de même pour le choix du vectoriseur et de bien d'autres choix de prétraitements. Malheureusement, les paramètres (de la pipeline) étant trop nombreux, cela est bien trop coûteux en temps de calcul. De plus, nos modèles sont plutôt performants sans recherche d'hyperparamètres. Nous allons donc nous concentrer exclusivement sur les hyperparamètres des modèles.
Pour la recherche d'hyperparamètres, nous avons utilisé l'implémentation *Grid Search* de *scikit-learn*, en utilisant une validation croisée avec 5 folders.

Ci-dessous sont les hyperparamètres pour chaque modèle que nous avons considérer pour notre recherche :

Régression logistique

- **C** : est l'inverse de la force de la régularisation. Plus la valeur est petite, plus la régularisation est forte.
  - Les valeurs testées sont : 0.001, 0.01, 0.1, 1, 10, 100
- **solver** : est l'algorithme d'optimisation. En plus de *lbfgs*, nous utiliserons les algorithmes *sag* et *saga* qui sont les plus adaptés pour les grands jeux de données.
  - Les valeurs testées sont : *lbfgs*, *sag*, *saga*
- Après **réalisation**, nous obtenons pour la meilleure combinaison :
  - Une justesse de validation moyenne de **89.07%**.
  - Avec **C** = 1, **solver** = *lbfgs*

Arbre de Décision

- **criterion** : est la fonction utilisée pour mesurer la qualité du fractionnement.
  - Les valeurs testées sont : *gini*, *entropy*
- **splitter** : est la stratégie utilisée pour le fractionnement.
  - Les valeurs testées sont : *best*, *random*
- **max_depth**: est la profondeur maximum de l'arbre.
  - Les valeurs testées sont : 10, 11, 12, 13, 14
- **max_features** : le nombre maximum de caractéristiques à considérer lorsqu'on cherche le meilleur fractionnement.
  - Les valeurs testées sont : *None*, *auto*, *sqrt*
- Après **réalisation**, nous obtenons pour la meilleure combinaison :
  - Une justesse de validation moyenne de **73.55%**.
  - Avec **criterion** = *gini*, **max_depth** = 14, **max_features** = *None*, **splitter** = *best*

Naive Bayes Multinomial

- **alpha** : est le paramètre de lissage additif. Si alpha = 0, cela est équivalent au lissage de Laplace. Si alpha est strictement inférieur à 1, c'est un lissage de Lidstone.
  - Les valeurs testées sont : 0.25, 0.5, 0.75, 1
- Après **réalisation**, nous obtenons pour la meilleure combinaison :
  - Une justesse de validation moyenne de **86.04%**.
  - Avec **alpha** = 0.75

Forêt aléatoire

En plus des mêmes hyperparamètres que l'arbre de décision, nous prenons en considération :

- **n_estimators** : est le nombre d'arbre souhaité.
  - Les valeurs testées sont : 50, 150, 250, 350
- Pour les valeurs des autres hyperparamètres, nous prenons les mêmes que ceux de l'arbre de décision.
- Après **réalisation**, nous obtenons pour la meilleure combinaison :
  - Une justesse de validation moyenne de **84.51%**.
  - Avec **criterion** = *gini*, **max_depth** = 14, **n_estimators** = 350, **max_features** = *sqrt*

SVM Linéaire

- **C** : tout comme pour la régression logistique, c'est un paramètre de régularisation. La force de régularisation est inversement proportionnelle à C.
  - Les valeurs testées sont : 0.001, 0.01, 1, 10, 100
- **loss** : est la fonction de perte utilisée.
  - Les valeurs testées sont : *hinge*, *squared_hinge*
- Après **réalisation**, nous obtenons pour la meilleure combinaison :
  - Une justesse de validation moyenne de **88.97%**.
  - Avec **C** = 1, **loss** = *hinge*

Perceptron multicouche

- **hidden_layer_sizes** : est la forme que l'on veut pour le réseau.
  - Les valeurs testées sont : (50,), (50, 100, 50), (50, 100, 200, 100, 50)
- **solver** : est l'algorithme utilisé pour l'optimisation des poids.
  - Les valeurs testées sont : *lbfgs*, *adam*
- **alpha** : est un terme de régularisation (L2).
  - Les valeurs testées sont : 0.0001, 0.001, 0.01
- **learning_rate** : est la manière dont le taux d'apprentissage évolue.
  - Les valeurs testées sont : *constant*, *adaptive*
- Après **réalisation**, nous obtenons pour la meilleure combinaison :
  - Une justesse de validation moyenne de **87.18%**.
  - Avec **alpha** = 0.001, **hidden_layer_sizes** = (50, 100, 200, 100, 50), **learning_rate** = *adaptative*, **solver** = *lbfgs*

Les recherches d'hyperparamètres réalisées ont été faites sur l'ensemble d'entraînement, mais les folders, ne sont pas les mêmes que ceux utilisés lors des autres validations croisées des sous-sections 7.2 à 7.4. Il serait donc intéressant d'effectuer une validation croisée pour chaque modèle avec les hyperparamètres trouvés afin de constater les évolutions de performance.

7.6 Validation croisée post recherche d'hyperparamètres

Afin d'évaluer nos hypothèses sur le nombre de dimensions et les résultats de la recherche d'hyperparamètres, nous réalisons une comparaison entre les résultats de la figure 8 et une nouvelle validation croisée. Comme expliqué à la sous-section précédente, pour comparer les résultats nous utilisons les mêmes données pour chaque **folder** lors de la validation croisée. Nous pouvons constater les résultats sur la figure 12 ci-dessous :

	model	train accuracy before tuning	train accuracy post tuning	train trend	valid accuracy before tuning	valid accuracy post tuning	valid trend
0	decisiontree	0.7864	0.8087	↗	0.7340	0.7358	↗
1	linearsvc	0.9877	0.9531	↘	0.8908	0.8941	↗
2	logisticregression	0.9301	0.9275	↘	0.8898	0.8904	↗
3	mlp	1.0000	0.8910	↘	0.8764	0.8739	↘
4	multinomialnb	0.9100	0.9157	↗	0.8599	0.8604	↗
5	randomforest	0.8789	0.9037	↗	0.8292	0.8428	↗

Figure 12: Comparaison des validations croisées avant-après sélection du nombre de dimensions et recherche d'hyperparamètres.

La première chose que l'on remarque est l'augmentation de la justesse (moyenne) de validation pour tous les modèles à l'exception du Perceptron Multicouche. Cependant, les augmentations restent relativement faibles, de **0.06%** à **1.36%**.

En ce qui concerne le Perceptron Muticouche, la justesse de validation est passée de **87.84%** à **87.39%** soit une perte de **0.25%**. Cependant, la justesse moyenne d'entraînement est passée de **100%** à **89%**. On a ainsi pu conserver une bonne justesse moyenne de validation tout en supprimant le surapprentissage.

Les justesses moyennes d'entraînement des modèles SVM Linéaire et Régression Logistique ont légèrement diminuées. Cependant, cela n'est pas forcémment une mauvaise chose. En effet, les justesses de validation ont augmentées, en restant inférieures aux justesses d'entraînement, réduisant ainsi l'écart des justesses validation/entraînement.

Quant aux 3 autres modèles, Arbre de Décision, Naive Bayes Multinomial, Forêt Aléatoire, les justesses d'entraînement et de validation ont toutes deux **augmentées** tout en conservant un **écart convenable**.

Nous avons pu voir à travers cette analyse que nos modèles ont diminués l'écart entre les justesses d'entraînement et de validation. Cependant, la recherche d'hyperparamètres n'a pas montré une évolution significative des performances des modèles. La raison la plus probable est notre sélection du nombre de dimensions. En effet, nous avons grandement réduit le nombre de dimensions de plusieurs modèles pour différentes raisons dont la volonté d'éviter le surapprentissage. Il est possible que nous ayons été trop stricts dans la sélection du nombre de dimensions.

8 Performances sur les données tests

Maintenant que nous avons sélectionnés la configuration de nos modèles (et du reste de la pipeline), nous pouvons réaliser le test final sur les données que nous avons laissés de côté depuis le début. Ce test final va pourvoir confirmer le bon déroulement notre démarche scientifique ou au contraire remettre en question notre démarche. Après execution des prédictions sur les données de test, voici les résultats de justesse pour chaque modèle :

- ***Arbre de Décision*** : 74.70%
- ***Régression Logistique*** : 88.39%
- ***SVM Linéaire*** : 89.03%
- ***Perceptron Multicouche*** : 87.24%
- ***Naive Bayes Multinomial*** : 86.34%
- ***Forêt aléatoire*** : 84.23%

Nous pouvons d'abord constater que les modèles sont plutôt performants, ils sont tous au-delà de 80% hormis l'Arbre de Décision qui est à 74.70%. Nous remarquons également que les résultats diffèrent très peu des résultats de validation de la figure 12 cela traduit une bonne généralisation (et fiabilité) de nos modèles.

Aussi, annexe E, nous pouvons voir la matrice de confusion sur les données tests, pour chaque modèle. A l'aide de ces graphes, nous voyons que l'Arbre de Décision n'a pas un bon équilibre FP/FN, c'est-à-dire qu'il classifie 2 fois plus mal les données de la classe 0 (sentiment négative) que ceux de la classe 1 (sentiment positif). En revanche les autres modèles ont plutôt un bon équilibre FP/FN. Enfin, on peut observer que l'Arbre de Décision généralise moins bien que la Fôret aléatoire ce qui est cohérent avec nos hypothèses émises plus tôt.

9 Pistes d'amélioration

Pour améliorer nos résultats, il existe une variété de moyen pour y parvenir. Nous pourrions envisager d'améliorer notre phase de prétraitement. Par exemple, nous pourrions utiliser un correcteur de faute, la librairie *TextBlob* nous permet de faire cela.
Une autre piste serait d'obtenir plus de données. En effet, fournir plus de données permet souvent d'aider les modèles à généraliser. Nous pourrions ainsi être moins stricts dans la régularisation ainsi que le réduction du nombre de dimensions.
Nous pourrions également utiliser des techniques plus récentes qui sont beaucoup utilisées comme le **Word Embeddings** ou l'**apprentissage profond**. Une approche beaucoup utilisée est l'**apprentissage par transfert** (*transfert learning*). Cette technique consiste à tirer partie de l'experience acquise en résolvant un problème (assez similaire) et à l'appliquer à un autre problème.

10 Conclusion

A travers ce projet, nous avons explorés plusieurs modèles d'apprentissage automatique ainsi que plusieurs méthodes de traitement du language naturel. Nous avons pu voir que le choix du prétraitement, en particulier le nombre de dimensions, pouvait impacter le modèle ainsi que sa capacité à bien généraliser. Nous avons également réussi à prévenir un surapprentissage sur certains modèles. Finalement, ce dernier test de la section 8 est venu affirmer notre démarche scientifique.
Sur les données d'entraînement, sur une validation croisée après recherche d'hyperparamètres, notre meilleur modèle est le SVM linéaire avec **89.41%** de justesse moyenne. Egalement, sur le jeu de test final, le SVM linéaire obtient le meilleur résultat avec **89.03%**.

Annexes

Appendix A Tutoriel pour faire fonctionner une validation croisée

Nous supposons ici que le projet a déjà été cloné et que vous vous trouvez à la racine du projet.

- **Etape 1** : Installer les dépendances.
  Pour cela, effectuer la commande : **pip install -r requirements.txt**
- **Etape 2** : Lancer *MLflow*.
  Pour cela, effectuer la commande dans un terminal (à la racine du projet) : **mlflow ui**
- **Etape 3** : Lancer la validation croisée.
  Dans un autre terminal effectuer la commande (toujours à la racine du projet) : **python expermimentation/validation.py 1 2 2 3000 0 1 1 LogisticRegressionTest**.
  Cette commande lancera une validation croisée pour une régression logistique avec certaines configuration de prétraitement. Pour essayer d'autres modèles et configurations, vous pouvez vous référer au fichier **README.md** du projet.
- **Etape 4** : Voir les résultats.
  Les résultats basiques (justesse) peuvent être visibles dans le terminal. Cependant, les résultats sont disponibles à l'adresse `http://localhost:5000` grâce à l'interface utilisateur de *MLflow*.

Appendix B Interface utilisateur de MLflow

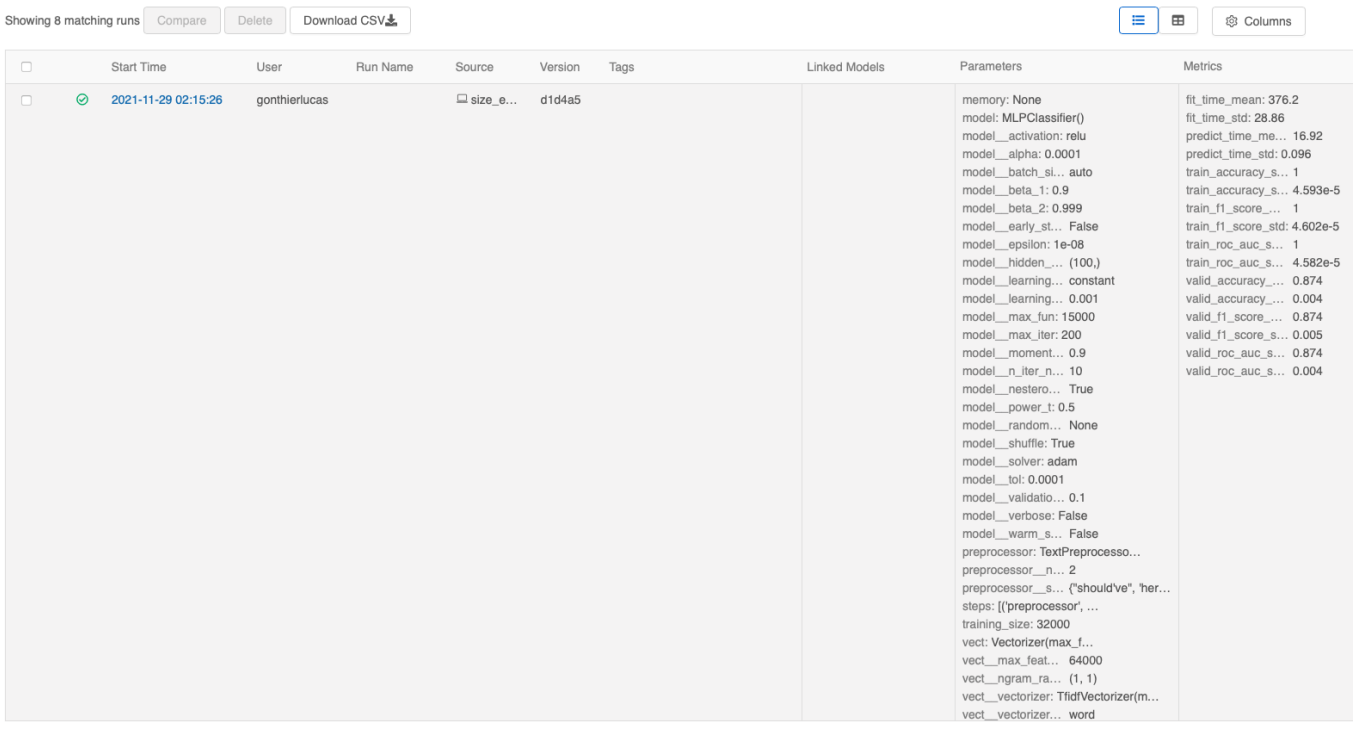

Figure 13: Exemple de résultats d'une validation croisée dans l'inteface utilisateur de MLflow

Appendix C Structure du projet

IMDB_reviews/
|__ data/
|__ ensembling/
|__ experiment/
|__ experimentation/
|__ model/
|__ mlruns/
|__ result_analysis/

- **data** : Est le dossier où sont stockés les données originales et séparés en folder.

- **ensembling** : est le dossier où est implémentée une classe Ensembling pour prendre en charge des modèles d'ensemble.

- **experiment** : est le dossier où est implémenté la classe Experiment décrite à la sous-section 2.2.2.

- **model** : est le dossier où sont les modèles utilisés pour le projet sous forme de dictionnaire (clé, valeur) avec comme clé le nom de la classe du modèle et comme valeur le modèle.

- **mlruns** : est le dossier créé par *MLflow* pour stocker les résultats. S'il n'est pas présent, il sera créé à la première utilisation du script **validation.py**.

- **result_analysis** : est contenant des sous-dossiers pour chaque étape de notre démarche. Dans ces sous dossiers se trouvent les résultats sous format csv et un notebook où les résultats sont analysés.

Appendix D Résultats de l'évolution de la justesse en fonction du nombre de dimensions

	model	vect__max_features	train_accuracy_score_mean	valid_accuracy_score_mean
0	decisiontree	64000	0.786031	0.732475
1	decisiontree	32000	0.785000	0.732300
2	decisiontree	16000	0.786019	0.733250
3	decisiontree	8000	0.785644	0.733375
4	decisiontree	4000	0.787963	0.735650
5	decisiontree	2000	0.787806	0.733950
6	decisiontree	1000	0.789037	0.733825
7	decisiontree	500	0.789269	0.734150
0	linearsvc	64000	0.985331	0.890525
1	linearsvc	32000	0.981363	0.889950
2	linearsvc	16000	0.972575	0.887575
3	linearsvc	8000	0.953363	0.883125
4	linearsvc	4000	0.926556	0.877625
5	linearsvc	2000	0.899494	0.871400
6	linearsvc	1000	0.876563	0.861025
7	linearsvc	500	0.853356	0.845825
0	logisticregression	64000	0.928950	0.890350
1	logisticregression	32000	0.927475	0.890375
2	logisticregression	16000	0.924350	0.889925
3	logisticregression	8000	0.918138	0.888375
4	logisticregression	4000	0.908013	0.884250
5	logisticregression	2000	0.893625	0.875475
6	logisticregression	1000	0.874275	0.862025
7	logisticregression	500	0.853313	0.846050
0	mlp	64000	0.999962	0.873575
1	mlp	32000	0.999838	0.868225
2	mlp	16000	0.999694	0.859175
3	mlp	8000	0.999950	0.851450
4	mlp	4000	0.999963	0.866725
5	mlp	2000	0.999981	0.867500
6	mlp	1000	1.000000	0.854375
7	mlp	500	1.000000	0.828075
0	multinomialnb	64000	0.903744	0.860525
1	multinomialnb	32000	0.896575	0.859750
2	multinomialnb	16000	0.884844	0.857775
3	multinomialnb	8000	0.870300	0.852625
4	multinomialnb	4000	0.859350	0.848925
5	multinomialnb	2000	0.847963	0.841200
6	multinomialnb	1000	0.836400	0.832600
7	multinomialnb	500	0.827613	0.825475
0	randomforest	64000	0.882550	0.831075
1	randomforest	32000	0.882050	0.833150
2	randomforest	16000	0.880012	0.832175
3	randomforest	8000	0.878144	0.829700
4	randomforest	4000	0.877031	0.825850
5	randomforest	2000	0.871050	0.820400
6	randomforest	1000	0.866331	0.811050
7	randomforest	500	0.857338	0.798100

Figure 14: Résultats de l'évolution de la justesse en fonction du nombre de dimensions

Appendix E Matrices de confusion des modèles sur le jeu de données tests

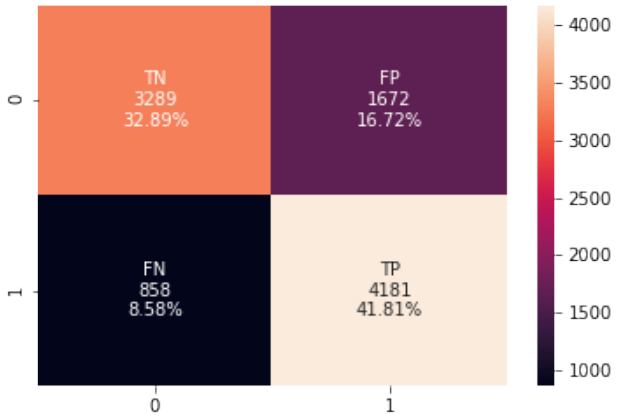

Figure 15: Matrice de confusion de l'Arbre de Décision sur le jeu de données test

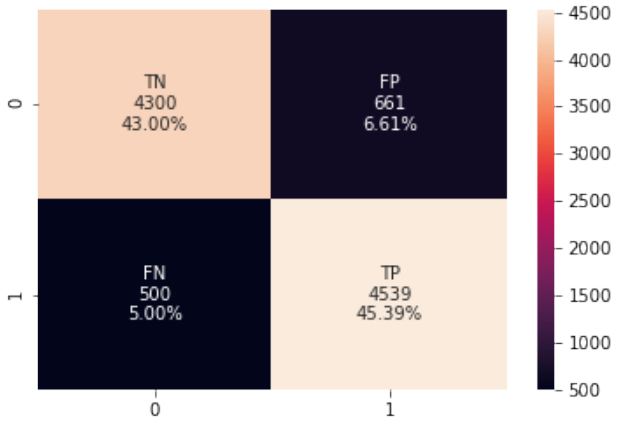

Figure 16: Matrice de confusion de la Régression Logistique sur le jeu de données test

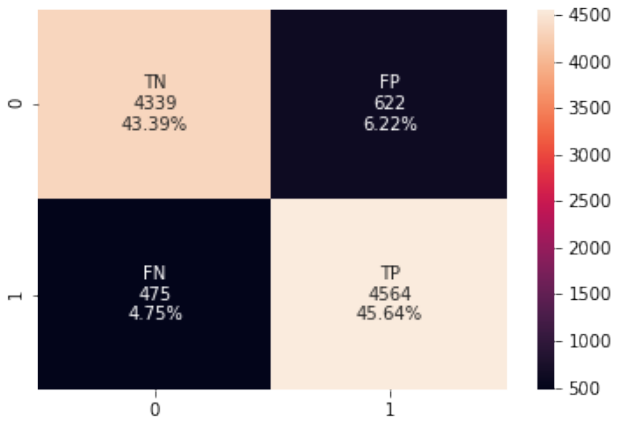

Figure 17: Matrice de confusion du SVM Linéaire sur le jeu de données test

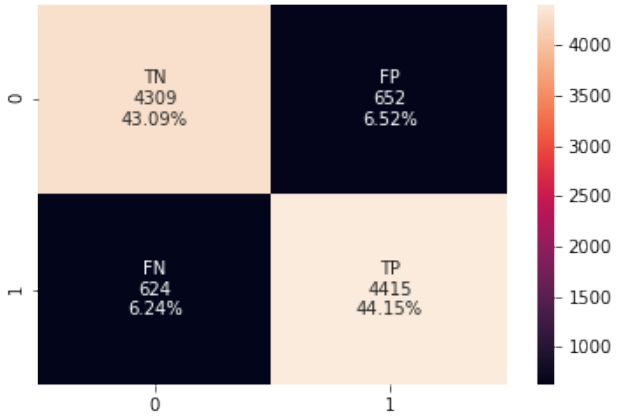

Figure 18: Matrice de confusion du Perceptron Multicouche sur le jeu de données test

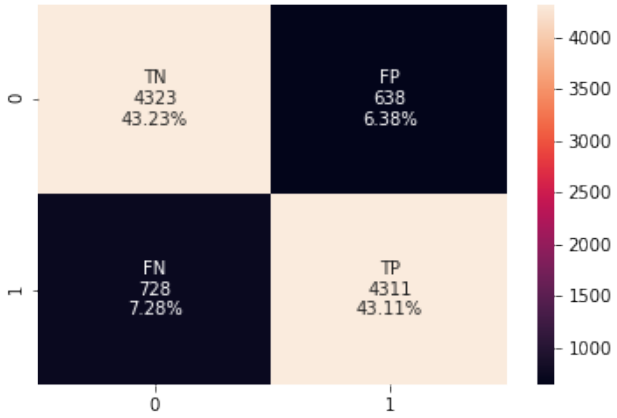

Figure 19: Matrice de confusion du Naive Bayes Multinomial sur le jeu de données test

Figure 20: Matrice de confusion de la Forêt Aléatoire sur le jeu de données test

11 Bibliographie

Papiers & Livres

[1] *Learning Word Vectors for Sentiment Analysis*. Andrew L. Maas, Raymond E. Daly, Peter T. Pham, Dan Huang, Andrew Y. Ng, et Christopher Potts, 2011.

[2] Normalization, TF-IDF vectors, *Natural Language Processing In Action*. Hobson Lane, Cole Howard, Hannes Max Hapke, 2019, pages 30-113.

[3] End-To-End Machine Learning Project, the curse of dimensionality, *Hands-on Machine Learning with Scikit-Learn, Keras & TensorFlow, Second Edition*. Aurélien Géron, 2019, pages 35-220.

[4] Bias-Variance Tradeoff, *Machine Learning Engineering*. Andriy Burkov, 2020, pages 155-160.

[5] Natural Language Processing, Deep Learning For Natural Language Processing, *Artificial Intelligence A Modern Approach, Fourth Edition*. Stuart Russel, Peter Norvig, 2020, pages 823-878.

Ressources internets

[6] MLflow Tracking : https://mlflow.org/docs/latest/tracking.html

[7] Scikit-Learn : https://scikit-learn.org/stable/

[8] Scikit-Learn Pipeline : https://scikit-learn.org/stable/modules/generated/sklearn.pipeline.Pipeline.html

[9] Reproductibilité : https://fr.wikipedia.org/wiki/Reproductibilité

[10] Bibliothèque Python, nltk : https://www.nltk.org

[11] Bibliothèque Python, TextBlob : https://textblob.readthedocs.io/en/dev/

[12] Image Perceptron multicouche : https://fr.wikipedia.org/wiki/Perceptron_multicouche#/media/Fichier:Perceptron_4layers.png