Big Data sous Python

SVM et réseaux de neurones

Contradictory, My Dear Watson

M2 – Économétrie et statistiques, parcours économétrie appliquée

IAE Nantes – Université de Nantes

Julie Perron

Marina Gourin

Année universitaire 2010-2021

PRÉSENTATION DU SUJET

L'objectif de ce rapport est de présenter notre démarche et nos résultats obtenus en lien avec notre *repo github*. Il présente alors les différentes méthodes de classification SVM (machine de support vectoriel) et ANN (réseau de neurones) suite au cours du même nom.

Notre sujet est le suivant : *Contradictory, My Dear Watson* (cf. Kaggle Competition). Lorsque nous avons 2 phrases, il y a 3 façons de les relier : l'une pourrait entraîner l'autre, l'une pourrait contredire l'autre ou bien, elles pourraient être sans rapport l'une à l'autre. L'objectif est alors de créer un modèle qui attribue un label de 0, 1 ou 2 correspondant respectivement à un lien d'implication, de neutralité ou de contradiction. Pour ce faire, nous disposons d'un jeu d'apprentissage *train* et d'un jeu de validation *test*. Dans chacun des deux échantillons, nous disposons des variables suivantes :

- Un identifiant : ID ;
- Une prémisse : premise, qui est une proposition dont nous déduirons des conséquences ou des conclusions ;
- Une hypothèse : hypothesis ;
- Un label avec les modalités 0, 1 et 2 qui est notre variable à expliquer ;
- Une langue : language, les deux échantillons contiennent une quinzaine de langues différentes ;
- Une abréviation de cette langue : lang_abv.

A noter que nous ne disposons pas de la variable « label » dans notre échantillon test.

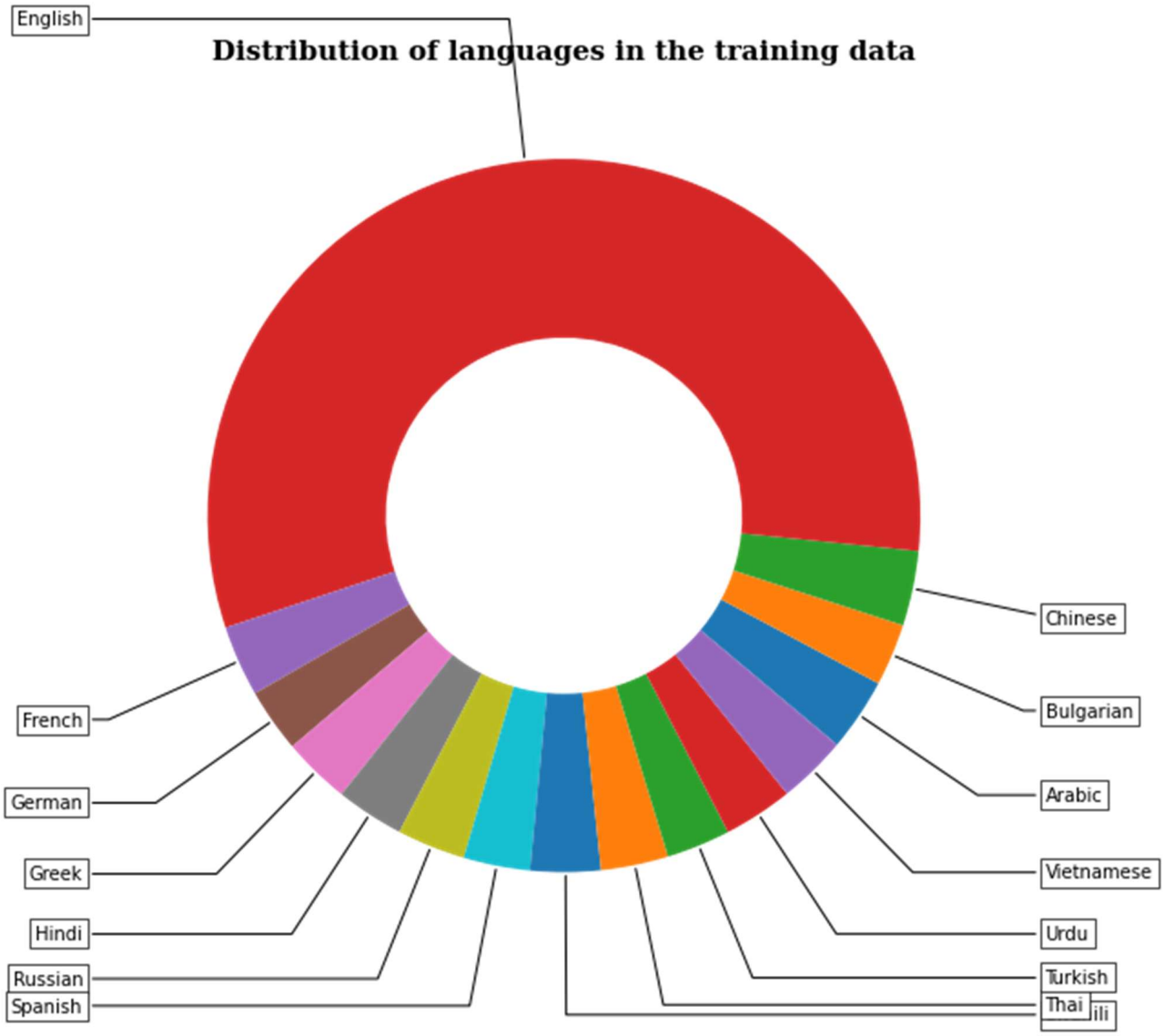
ANALYSE EXPLORATOIRE

Avant de nous lancer dans la construction de classifieurs, nous avons voulu dans un premier temps explorer nos données à travers plusieurs visualisations.

Dans un premier temps, nous avons analysé la forme de notre échantillon d'apprentissage avec la commande $train.head()$. Nous avons également visualisé sa taille avec la fonction $train.shape$. Notre échantillon train dispose alors de 12 120 observations et 6 colonnes.

Afin d'avoir un premier aperçu des différentes langues de notre échantillon, nous avons réalisé un diagramme circulaire permettant d'analyser la distribution des différentes langues de notre base.

Figure 1 – Descriptif des différentes langues de notre échantillon train

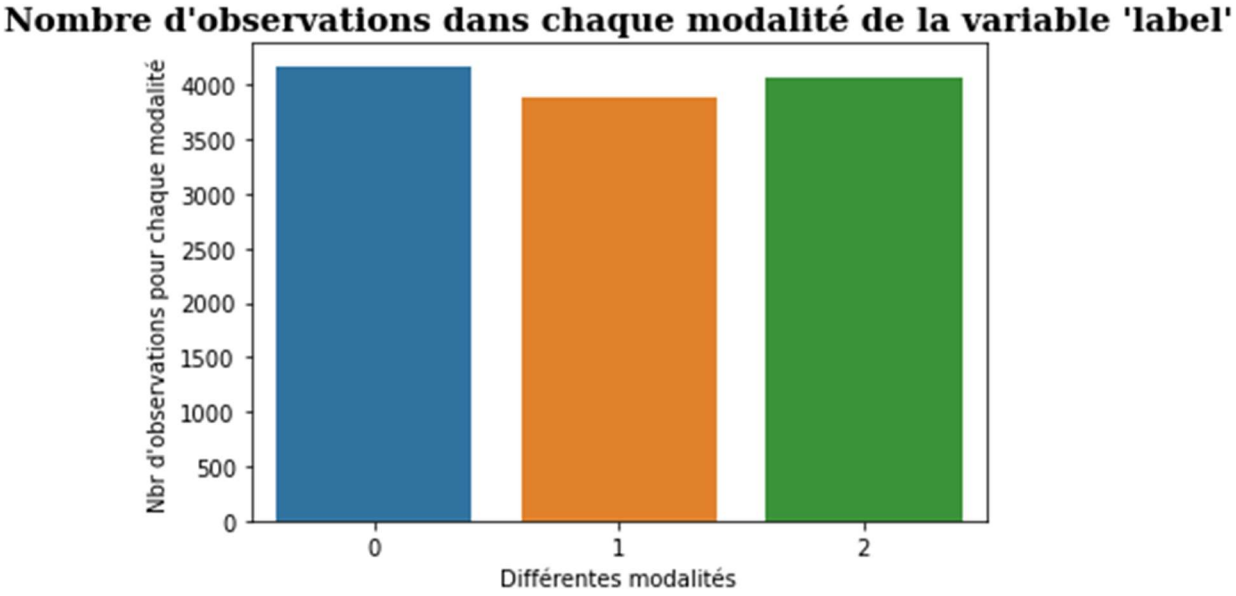


Source : Python

Comme nous pouvions nous y attendre, la langue anglaise est la langue majoritaire de notre base de données avec plus de la moitié des observations (6 870 observations sur 12 120 au total soit plus de 56.68 %).

Dans un deuxième temps, nous avons voulu observer la distribution de notre variable dépendante. Pour ce faire, nous avons réalisé un graphique en bar permettant d'analyser le nombre d'observations pour chacune des modalités de la variable label (**Figure 2**).

Figure 2 - Descriptif des différentes modalités de notre variable à expliquer



Source : Python

Pour rappel, la modalité 0 correspond à une implication entre les deux phrases, 1 à un lien de neutralité et 2 à un lien de contraction. Nous pouvons alors voir que les modalités sont globalement bien réparties. Seule la modalité 1 comporte le moins d'observations (3 880). Les autres possèdent quasiment le même nombre d'observations, 4 176 pour la modalité 0 et 4 064 pour la modalité 2.

RÉALISATION DES CLASSIFIEURS

A. Création d'un sous-échantillon

Afin de nous simplifier la tâche et comme nous avions remarqué la langue anglaise représentait plus de la moitié des observations, nous avons créé un sous-échantillon composé uniquement de la langue anglaise. Grâce à la commande *shape* nous constatons que ce sous-échantillon comporte 6 870 observations et 6 colonnes. Comme notre nouvel échantillon ne contient que des observations de la langue anglaise, nous avons supprimé les variables « *lang_abv* » et « *language* » dont nous n'avions plus besoin.

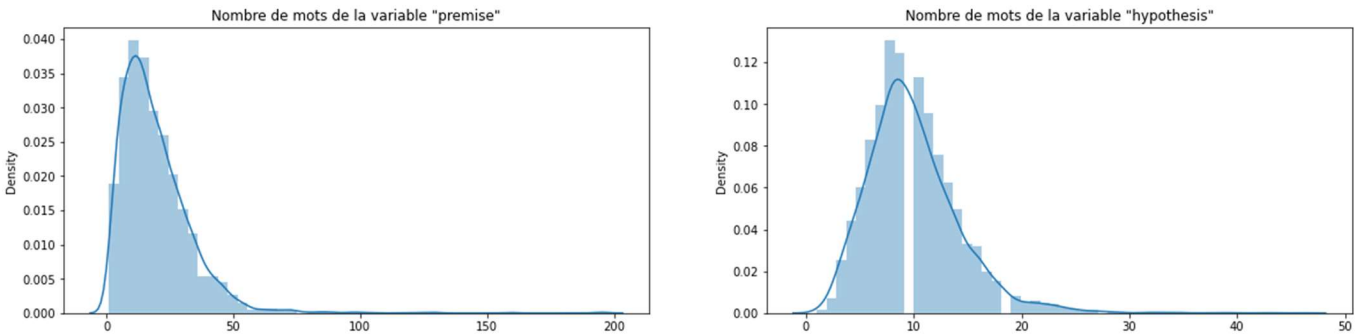
Notre base de données étant composée de variables textuelles, il nous faut donc les transformer en variables numériques afin de pouvoir réaliser des classifieurs. Alors, plusieurs approches vont être appliquées afin de comparer les résultats.

B. Première approche

Dans un premier temps, nous allons tester une approche permettant de transformer nos deux variables textuelles en numériques en comptant le nombre de mots dans les phrases des variables « *premise* » et « *hypothesis* ». Pour ce faire, nous avons créé une fonction *word_count* qui prend en entrée un DataFrame et la colonne concernée. Cette fonction revoie alors la longueur, c'est-à-dire le nombre de mots, d'un texte présent dans une colonne d'un DataFrame.

Afin d'avoir un aperçu visuel du nombre de mots dans les phrases de nos variables « *premise* » et « *hypothesis* » nous avons réalisé l'histogramme suivant.

Figure 3 - Représentation graphique du nombre de mots dans nos deux variables textuelles



Source : Python

Nous pouvons alors voir que la variables « *premise* » comporte principalement des phrases comportant entre 0 et 50 mots avec une moyenne à environ 20 mots. Au contraire, pour la variable « *hypothesis* », nous pouvons voir 3 groupes : des phrases comportant 0 à 8 mots, des phrases de 10 à 18 mots et enfin de 20 à 30. Ce dernier groupe est moins important que les deux précédents.

Afin de créer des classifieurs, il nous faut donc des variables numériques. Ainsi, nous allons créer un nouveau DataFrame comprenant uniquement les observations en langue anglaise et prenant les deux variables comptant le nombre de mots de nos deux variables textuelles. Ce DataFrame comporte alors 4 variables : l'identifiant, le label, premise et hypothesis pour lesquelles le nombre de mots a été compté. La suite de l'analyse se fera sur ce dernier DataFrame.

Nous allons alors préparer nos variables Y et X. La variable Y comprend les observations de la colonne « label » et X comprend les variables explicatives, à savoir « *premise* » et « *hypothesis* ». Ainsi de vérifier la distribution des modalités de notre variable Y, nous avons utilisé la fonction *count*. La modalité 0 de la variable Y représente 35.33% des observations, la modalité 2 représente 33.14% et enfin la modalité 1 représente 31.53%. Nous pouvons alors noter une répartition quasi égale des modalités.

Une fois notre DataFrame constitué, plusieurs modèles vont être réalisés à savoir, un arbre de décision, une forêt aléatoire, une régression logistique, un SVM linéaire, un SVM gaussien et enfin un classifieur MLP.

1. Arbre de décision

Dans un premier temps, nous avons créé un classifieur à partir de la fonction $DecisionTreeClassifier$ et en utilisant les paramètres par défaut. Pour analyser la qualité de ce premier classifieur, nous avons calculé le F1 score. Il peut être interprété comme une moyenne pondérée de la précision et du *recall* (le rapport entre les observations positives correctement prédites et toutes les observations réelles positives). Après application du modèle sur notre échantillon d'apprentissage, nous obtenons un F1 score de 0.5047.

Afin d'améliorer cette qualité plutôt basse, nous avons modifié les paramètres de notre classifieur. Nous avons fixé la profondeur maximale de l'arbre à 300, le nombre minimum

d'échantillon requis pour fractionner le nœud à 3 et fixé un poids égal à toutes les classes. Cependant, même en changeant les paramètres, le F1 score reste le même.

2. Random Forest

Notre deuxième classifieur est une forêt aléatoire. Pour utiliser ce classifieur, nous avons utilisé la fonction $RandomForestClassifier$ avec les paramètres par défaut dans un premier temps. Comme précédemment, nous avons calculé le F1-score. Ce dernier est un peu plus élevé que le précédent avec 0.5076. Dans un deuxième temps, nous avons modifié les paramètres de *Random Forest*. Nous avons fixé le nombre d'arbres dans la forêt, la profondeur de l'arbre, le nombre minimum d'échantillon et enfin le poids des classes. Seulement, le score F1 n'est pas meilleur que le précédent sur notre échantillon d'apprentissage.

3. La régression logistique

Notre troisième modèle est une régression logistique. Nous avons utilisé la fonction $LogisticRegression$ en fixant le solveur à *liblinear*, c'est-à-dire une classifieur linéaire. Malheureusement pour nous, le choix de ce classifieur n'est pas réellement le bon puisque le F1 score de ce dernier est de 0.3558. Il s'agit pour l'instant, du modèle le moins précis.

4. SVM linéaire

Le but de cet exercice étant d'appliquer les méthodes de classifications vues en cours, nous avons appliqué un SVM linéaire. Dans un premier temps, nous avons fixé le paramètre C, mesurant le degré de pénalité, à 1. Nous avons calculé le score de ce modèle sur nos variables Y et X de notre échantillon d'apprentissage. Ce score s'élève à 0.3553. Même en fixant une pénalité plus importante (0.0001), le score ne s'améliore pas réellement. Ainsi, nous pouvons penser que nos variables explicatives, transformer en nombre de mots par phrase n'est pas la meilleure approche.

5. SVM gaussien

Notre second modèle SVM est un SVM gaussien en fixant le *kernel* égal à « rbf » (*radical basis function*). Dans un premier temps, nous avons fixé le paramètre C à 1. Le score de ce modèle s'établit à 0.4744. En modifiant le paramètre C à $1e^6$, le score s'améliore pour atteindre 0.5080. La matrice de confusion de ce dernier est la suivante :

1306	436	685
622	972	572
669	396	1212

Source : Python

Les valeurs en diagonale sont les valeurs bien classées. Les valeurs en dehors de cette diagonale sont les valeurs mal placées. Ainsi pour la classe 1, 1 121 observations sont mal classées contre 1306 de bien classées. Pour la classe 2, 1194 sont mal placées contre uniquement 972 de bien classées. Enfin, pour la classe 3, 1212 observations sont bien classées et 1065 sont mal prédites par le modèle. Même si la qualité du modèle n'est pas optimale, nous pouvons noter qu'il s'agit de notre meilleur modèle pour l'instant pour cette première approche.

6. MLP classifier

Enfin, notre dernier modèle est un classifieur MLP (*Multi-Layer Perceptron*, un perceptron avec plusieurs couches) utilisé grâce à la fonction $MLPClassifier$. Nous avons fixé le solveur en « lgfgs » qui est un optimiseur de la famille des méthodes de quasi-Newton, un paramètre alpha à 1^{e-5} qui est un paramètre de pénalité et le *hidden_layer_sizes* représentant le nombre de neurones dans la couche cachée. Cependant, le score de ce modèle n'est pas non plus optimal puisqu'il s'établit à 0.3665.

En conclusion de cette première approche, nous pouvons dire que le fait d'avoir transformer nos variables textes en variables numériques en comptant le nombre de mots par phrase n'est pas la meilleure solution puisque les résultats obtenus ne sont pas très significatifs et concluants.

C. Deuxième approche

Puisque l'approche précédente ne nous a pas fourni des résultats satisfaisants, nous allons utiliser une nouvelle approche en reprenant le DataFrame comprenant uniquement la langue anglaise.

Avant de commencer, nous avons vérifié le type de chaque variable en utilisant la fonction *dtypes*. Les variables « *premise* » et « *hypothesis* » étant des variables avec du texte, il est logique que ce soient des variables de types « *object* ». Nous avons donc, pour effectuer une deuxième approche, changer le type de ces deux variables en chaîne de caractère avec la fonction *str*.

Ensuite, pour transformer ces variables en chiffres, nous avons utilisé un système d'encodage avec la fonction $LabelEnconder( )$ qui va nous permettre de transformer ces variables textuelles en leur associant un score. Nous disposons ainsi d'un nouveau DataFrame comprenant 6 colonnes : l'identifiant, premise, hypothesis, label, le score de la variable *premise* et le score de la variable *hypothesis*. Par soucis de simplicité et pour éviter des temps de calculs plutôt long, nous avons seulement sélectionné les 150 premières lignes de ce DataFrame.

Comme précédemment, nous avons créé nos variables Y et X. La variable Y prend en compte la colonne « label » du DataFrame et X prend les deux nouvelles colonnes créées, à savoir le score associé aux variables « *premise* » et « *hypothesis* ».

A l'inverse de l'approche précédente, nous avons créé deux échantillons, un échantillon *train* et un *test*, à partir de ce nouveau DataFrame en utilisant la fonction $train\_test\_split$ à partir du package $sklearn$. Nous avons fixé la taille de notre échantillon *test* à 30% de notre DataFrame initial.

Pour cette deuxième approche tous les modèles testés pour la première approche ne seront pas appliqués. Nous appliquerons dans un premier temps un SVM linéaire puis un SVM avec la fonction $GridSearch$.

1. SVM linéaire

Notre premier modèle est un SVM linéaire en utilisant les paramètres par défaut. Nous avons seulement fixé le degré du polynôme à 3. Afin de vérifier la qualité du modèle, nous avons calculé l'accuracy score permettant de calculer la précision du sous-échantillon. L'ensemble des valeurs prédites par le modèle doit correspondre au valeurs réellement observées (*X_valid*). Dans notre cas, nous obtenons un score égal à 33.33% ce qui est plutôt mauvais. Seulement 33.33% des valeurs prédites sont réellement observées.

Nous avons également créé un tableau montrant les principales mesures de classification avec la fonction $classification\_report$. Ce dernier indique la précision, le *recall*, le F1-score et le support qui est le nombre d'occurrence de chaque classe dans l'échantillon test.

Nous pouvons noter, en observant le tableau ci-dessous, que la qualité de ce modèle est plutôt mauvaise puisqu'aucun des scores ne dépassent 0.5 qui est un score moyen. Seule la catégorie 0 peut être plus au moins acceptée.

Figure 4 - Classification report du SVM linéaire

	precision	recall	f1-score	support
0	0.47	0.42	0.44	19
1	0.25	0.08	0.12	13
2	0.25	0.46	0.32	13
accuracy			0.33	45
macro avg	0.32	0.32	0.30	45
weighted avg	0.34	0.33	0.32	45

Source : Python

2. SVM avec GridSearch

Afin d'améliorer le modèle précédent, nous avons effectué un deuxième modèle SVM avec GridSearch. Pour ce faire, nous allons créer plusieurs listes de paramètres. L'approche GridSearch va alors considérer de manière exhaustive toutes ces combinaisons possibles de ces paramètres. Nous avons créé une liste de valeur pour le paramètre C, $gamma$ et $kernel$.

Après un temps de calcul assez conséquent, les meilleurs paramètres de ce nouveau modèle sont les suivants :

$$"C" = 10$$

$$"gamma" = 0.001$$

$$"kernel" = "rbf"$$

La matrice de confusion associée à ce modèle est plutôt très mauvaise puisque beaucoup des valeurs prédites par le modèle ne correspondent pas aux valeurs réellement observées. Nous avons également construit le tableau de classification comme le modèle précédent et malheureusement pour nous, ce modèle n'est pas non plus optimal puisque beaucoup des scores sont assez faibles.

Figure 5 - Classification report du SVM linéaire avec GridSearch

	precision	recall	f1-score	support
0	1.00	0.05	0.10	19
1	0.00	0.00	0.00	13
2	0.30	1.00	0.46	13
accuracy			0.31	45
macro avg	0.43	0.35	0.19	45
weighted avg	0.51	0.31	0.17	45

Source : Python

Pour conclure sur cette deuxième approche, nous avons essayé de créer deux classifieurs. Malheureusement pour nous, ces deux derniers ne nous ont pas fournis de résultats satisfaisants puisque les scores étaient en général inférieurs à 0.5. Ainsi, cette deuxième approche permettant de transformer les variables textes en numériques grâce à un encodage n'est pas la meilleure solution pour traiter ce sujet.

D. Troisième approche

Puisque les deux premières approches que nous avons utilisées pour transformer nos variables explicatives en texte, nous allons tenter une nouvelle approche afin de transformer ces variables. Dans cette nouvelle approche, nous allons créer une nouvelle variable « sentence » qui est le regroupement des deux variables textes, « *hypothesis* » et « *premise* ». Comme pour l'approche précédente, nous avons transformé cette nouvelle variable en chaîne de caractère (str). Nous disposons alors d'un DataFrame composé de 5 variables.

Comme précédemment, nous avons découpé notre échantillon en deux : un échantillon *train* et *test*. Notre variable Y étant la variable « label » du DataFrame et X la variable « sentence ».
Premièrement, nous avons décomposé notre nouvelle variable X_train que nous avons stocké dans une liste *sentence*. Ensuite, nous avons utilisé la fonction $word2vec$ qui utilise des techniques d'apprentissage en profondeur et est basé sur les réseaux de neurones pour convertir les mots en vecteurs. Nous avons défini la dimension des vecteurs de mots à 300, la distance maximale entre le mot réel et le mot prédit dans une phrase à 20, nous avons ignoré les mots dont la fréquence totale est inférieure à 2 et le nombre de cœur de travail à 1. Cette spécification $word2vec$ nous a permis de créer un corpus composé de 4809 mots.

Après quelques visualisations de notre corpus et après avoir afficher les 5 premières coordonnées du vecteur associé au mot « people », nous avons, pour chaque phrase, fait la somme des vecteurs associés au mots qui la composent ou pas si le mot n'est pas présent dans le corpus.

Pour terminer cette troisième approche, nous avons effectué une régression logistique. Nous avons fixé le solveur à « lbfgs » et un maximum d'itérations à 2000. Seulement la

performance de ce modèle n'est pas très satisfaisante encore une fois puisque que nous obtenons un score associé égal à 0.3212.

E. Quatrième approche

Les précédentes approches n'étant pas concluantes en termes de performance et de qualité de classification, nous avons décidé de combiner les deux méthodes de transformation de texte utilisées précédemment.

Pour ce faire, nous avons créé un nouveau Dataframe contenant, d'une part, les colonnes « premise », « hypothesis » pour lesquelles le nombre de mots avait été compté pour chaque ligne, et d'autre part, les colonnes « code_premise » et « code_hypothesis » pour lesquelles nous avions attribué un score pour chaque ligne avec le Label Encoder.

Avant de passer à l'estimation des différents modèles de classification, nous avons décidé de diminuer la taille du jeu de données, en ne sélectionnant que les 500 premières lignes du DataFrame, afin de limiter un temps de convergence des modèles estimés trop long.

Par la suite, nous avons défini notre X et notre Y et avons procédé à l'échantillonnage, toujours en se basant sur la répartition suivante : 30% pour l'échantillon test et 70% pour l'échantillon d'apprentissage.

Etant en possession de tous les éléments nécessaires, nous sommes passés à l'estimation de plusieurs algorithmes de classification : un arbre de décision, deux forêts aléatoires, une régression logistique, deux SVM, un perceptron « multicouches » et un ANN.

1. Arbre de décision

Nous avons donc débuté les estimations par un arbre de décision. Nous avons décidé, dans un premier temps, de garder les paramètres de défaut afin d'avoir un aperçu de la pertinence de ce classifieur pour notre jeu de données.

Après spécification du modèle, nous l'avons appliqué à l'échantillon test. Nous avons ainsi obtenu un F1-score de 1 pour l'échantillon d'apprentissage et un F1-score de 0.3079 pour l'échantillon test.

Etant confrontés à un sur-apprentissage pour l'échantillon d'apprentissage, nous avons décidé de modifier les paramètres de l'arbre de décision :

- Une profondeur maximale de l'arbre de 50,
- Un nombre minimum pour fractionner le nœud de 3,

- Un poids identique pour toutes les classes.

Une fois que le modèle s'est entrainé sur l'échantillon train, nous l'avons appliqué à l'échantillon test et calculé le F1-score. Nous avons pu constater que la performance du classifieur est quasiment identique à celle de l'arbre de décision estimé avec les paramètres de défaut. Notons également que le F1-score pour l'échantillon d'apprentissage a légèrement diminué : nous passons de 1 à 0.9315.

2. Random Forest

Ensuite nous avons testé la performance et la qualité d'un classifieur de type Random Forest. Nous avons procédé en plusieurs étapes pour fixer les paramètres de l'algorithme : dans un premier temps, nous avons garder les paramètres de défaut, et dans un deuxième temps, nous avons utilisé GridSearch pour chercher les paramètres optimaux.

a) Avec paramètres de défaut

Nous avons donc appliqué le Random Forest en conservant les paramètres à défaut. Nous avons, comme pour les classifieurs précédents, calculé le F1-score. Etant une nouvelle fois confronté à un sur-apprentissage pour l'échantillon train, nous avons décidé de modifier les paramètres. Notons toutefois que ce type de classifieur n'est pas très performant sur l'échantillon test puisque le F1-score est de 31.99%.

Par la suite, nous avons donc fixé manuellement le nombre d'arbres dans la forêt, la profondeur de l'arbre, le nombre minimum d'échantillon et enfin le poids des classes. Si le F1-score a diminué pour l'échantillon d'apprentissage (82.07%), la précision de l'algorithme, en termes de classification, reste médiocre.

b) Avec GridSearch

N'obtenant pour l'instant pas des résultats satisfaisants avec Random Forest, nous avons décidé d'avoir recours à GridSearch pour obtenir les paramètres optimaux.

Après avoir testé toutes les combinaisons possibles de paramètres, GridSearch a conclu que les meilleurs paramètres étaient :

$$"max\_depth" = 20$$

$$"n\_estimators" = 10$$

Nous avons donc réestimé le Random Forest en tenant compte du choix de GridSearch pour les paramètres. Nous avons ainsi obtenu un F1-score de 34.64% pour l'échantillon tets.

3. Régression logistique

Dans un troisième temps, nous avons estimé une régression logistique. Notre troisième modèle est une régression logistique. A l'issue de cette modélisation, nous avons obtenu le meilleur F1-score pour l'échantillon test, à savoir 37.31%.

4. SVM

Le prochain classifieur qui a été utilisé pour cette approche, a été le SVM : d'une part, un SVM avec kernel linéaire, et d'autre part, un SVM avec kernel gaussien.

a) Avec kernel linéaire

Dans un premier temps, nous avons appliqué un SVM avec les paramètres de défaut (degré de pénalité fixé à 1) sur l'échantillon test et avons obtenu un score égal à 0.38.

Voulant améliorer ce score, nous avons décidé de fixer une pénalité plus importante en indiquant 0.1 pour le paramètre « *C* ». Nous avons ainsi pu noter que la précision en termes de classification s'était légèrement améliorée en augmentant la pénalité.

Par ailleurs, nous avons obtenu un taux moyen de prédiction en cross-validation de 33.33%.

b) Avec kernel gaussien

Ensuite, nous avons décidé d'utiliser une autre fonction noyau : la gaussienne. Nous avons donc suivi la même procédure que précédemment : premièrement, nous avons conservé les paramètres de défaut, puis avons modifié manuellement ces derniers.

Dans le premier cas, nous avons obtenu un score de précision égal à 35.33%. Nous pouvons d'ores et déjà constater que le SVM linéaire, avec les paramètres de défaut, offre une meilleure précision en termes de classification.

Dans le deuxième cas, nous avons fixé le paramètre « C » à 0.1 et le degré de la fonction noyau à 2. Après réestimation, nous avons obtenu un score de précision légèrement inférieur au précédent, à savoir 34.67%.

5. MLP Classifier

Le cinquième classifieur à avoir été utilisé pour cette approche est le perceptron multicouches. Pour ce faire, nous avons choisi un solveur de type « lbfgs » et avons fixé le paramètre alpha ainsi que le « hidden_layer ».

Après avoir entrainé le classifieur sur l'échantillon train, nous l'avons appliqué à l'échantillon test et avons obtenu un score de précision de 34.67.

Nous pouvons constater que ce classifieur a une précision en termes de classification, identique à celle du SVM gaussien avec paramètres fixés manuellement.

6. ANN et Tensorflow

Nous avons terminé cette section par l'application d'un réseau de neurones artificiels (ANN) sur le jeu de données. Pour ce faire, nous avons utilisé keras.Sequencial de Tensorflow. Nous avons donc tout d'abord spécifier le modèle Keras à l'aide d'une liste de couches « layer.Dense », et avons choisi une fonction d'activation linéaire redressée (« relu »). Ensuite, nous avons spécifié l'optimiseur, en choisissant comme métrique la précision (« Accuracy »).

Nous avons par le suite entrainé le modèle sur l'échantillon d'apprentissage en fixant la valeur du paramètre « echop » à 10. Après application du modèle sur l'échantillon test, nous avons obtenu un score de précision de 31.33%.

F. Cinquième approche

Dans cette dernière approche, nous avons décidé de nous concentrer sur la classification des « contradictions » (modalité 2 de la variable label). Nous avons, de ce fait, créé un nouveau Dataframe en incorporant une nouvelle variable : « contradiction ».

Cette variable est de type binaire et a été construite à partir de la variable « label » :

$$\begin{cases} 0 \ si \ label \ \neq 2 \\ 1 \ si \ label = 3 \end{cases}$$

Afin d'utiliser ce jeu de données pour la classification, nous avons eu recours à deux méthodes pour de transformation textuelle. Dans un premier temps, nous avons utilisé CountVectorizer de Scikit-learn qui permet de transformer une phrase en un vecteur de nombre, et dans un deuxième temps, nous avons utilisé Pipeline Scikit-Learn, qui permet également de vectoriser des phrases.

Concernant les classifieurs utilisés dans cette approche, nous avons fait de le choix d'estimer un SVM linéaire et un SVM à partir des paramètres obtenus avec GridSearch.

Par ailleurs, nous avons diminuer la taille du jeu de données et avons sélectionné les 1000 premières lignes.

1. SVM linéaire

Nous avons donc estimé un SVM linéaire à partir des vecteurs obtenus par les deux méthodes de traitement textuel. Comparons les résultats obtenus selon la méthode de vectorisation utilisée.

Pour ce faire, nous avons avons créé un tableau montrant les principales mesures de classification. Nous pouvons noter, en observant les **Figures 6** et **7**, que la qualité des classifieurs est plutôt satisfaisante et est nettement plus élevée que dans le cas où on essayait de classifier la variable « label ».

Figure 6 - Classification report du SVM linéaire (CountVectorizer)

	precision	recall	f1-score	support
0	0.64	0.68	0.66	201
1	0.27	0.24	0.26	99
accuracy			0.53	300
macro avg	0.46	0.46	0.46	300
weighted avg	0.52	0.53	0.53	300

Figure 7 - Classification report du SVM linéaire (Pipeline)

	precision	recall	f1-score	support
0	0.70	0.88	0.78	211
1	0.29	0.11	0.16	89
accuracy			0.65	300
macro avg	0.49	0.50	0.47	300
weighted avg	0.58	0.65	0.60	300

Source : Python

De plus, nous pouvons constater que la vectorisation par Pipeline permet d'obtenir un meilleur score de précision, en termes de classification, qu'avec le CountVectorizer.

2. SVM avec GridSearch

Afin d'améliorer le classifieur précédent, nous avons eu recours à GridSearch pour trouver les paramètres optimaux. Après convergence, nous avons obtenu les paramètres suivants :

$$"C" = 0.1$$

$$"gamma" = 1$$

$$"kernel" = "linear"$$

Par la suite, nous avons réestimé les classifieurs avec les paramètres retenus par GridSearch et avons construit les tableaux de classification suivants :

Figure 8 - Classification report du SVM avec GridSearch (CountVectorizer)

	precision	recall	f1-score	support
0	1.00	0.70	0.83	300
1	0.00	0.00	0.00	0
accuracy			0.70	300
macro avg	0.50	0.35	0.41	300
weighted avg	1.00	0.70	0.83	300

Figure 9 - Classification report du SVM avec GridSearch (Pipeline)

	precision	recall	f1-score	support
0	1.00	0.70	0.83	300
1	0.00	0.00	0.00	0
accuracy			0.70	300
macro avg	0.50	0.35	0.41	300
weighted avg	1.00	0.70	0.83	300

Source : Python

Nous pouvons constater que quelle que soit la méthode de vectorisation, nous obtenons les mêmes résultats. De plus, nous pouvons noter que dans 100% des cas, le classifieur affecte les phrases de l'échantillon test à la classe 0.

Nous pouvons donc conclure pour cette approche, qu'un SVM linéaire avec les paramètres de défaut, offre une meilleure performance et une meilleure précision qu'un SVM, dont les paramètres ont été obtenus avec GridSearch.

CONCLUSION

Pour conclure sur cette étude, nous pouvons dire que nous avons effectué plusieurs approches afin de transformer nos variables de type texte en variables numériques. Quatre approches ont été menées afin de créer plusieurs classifieurs permettant par la suite de prédire si deux phrases ont un lien d'implication, de neutralité, ou bien de contradiction.

Les différentes approches mises en œuvre ont fonctionné pour transformer nos variables en numériques, cependant les modèles réalisés nous ont donné des résultats non satisfaisants. En effet, plusieurs modèles ont été exécuté (régression linéaire, Random Forest, SVM linéaire et gaussien, MLP classifier etc.), cependant, comme nous pouvions nous y attendre, ces modèles ne sont de bonne qualité. Effectivement, beaucoup d'entre-eux possèdent un score inférieur à 0.5. Ainsi ces modèles ne peuvent pas vraiment être utilisés pour réaliser des prédictions sur le lien entre deux phrases.

Notre manque de connaissances en ce qui concerne les données textuelles nous a alors posé un certain problème. En effet, nous ne savions pas exactement comment traiter ce genre d'information.

Néanmoins, nous avons tenté quelques approches, en nous inspirant de ce qui avait été pu faire sur des données textuelles[1], afin d'appliquer nos connaissances sur les SVM et ANN vues en cours.

Le choix du sujet étant, par conséquent, un peu au-dessus de nos compétences, nous avons obtenu des classifieurs que nous pouvons qualifier de médiocres. Malgré cela, ce sujet nous a permis de découvrir plusieurs techniques de transformation de phrases en chiffres. De plus, cette étude nous a aidé à élargir nos connaissances en Machine Learning et plus précisément en réseaux de neurones.

[1] Dupre Xavier, « Classification de phrases avec word2vec »,
http://www.xavierdupre.fr/app/papierstat/helpsphinx/notebooks/text_sentiment_wordvec.html
Gunjit Bedi,"A guide to Text Classification(NLP) using SVM and Naive Bayes with Python",
https://medium.com/@bedigunjit/simple-guide-to-text-classification-nlp-using-svm-and-naive-bayes-with-python-421db3a72d34
Uzsoy Ana Sofia, "Tutorial Notebook
Contradictory, My Dear Watson", https://www.kaggle.com/anasofiauzsoy/tutorial-notebook