Projet 1 - IFT6285

Fabrice Lamarche, Charles Colella, Alaeddine Mellouli

Novembre 2021

1 Introduction

COLA The Corpus of Linguistic Acceptability: consiste à identifier si une phrase est acceptable ou pas.

SST The Stanford Sentiment Treebank ou la tâche consiste à identifier si la phrase d'entrée est positive ou pas

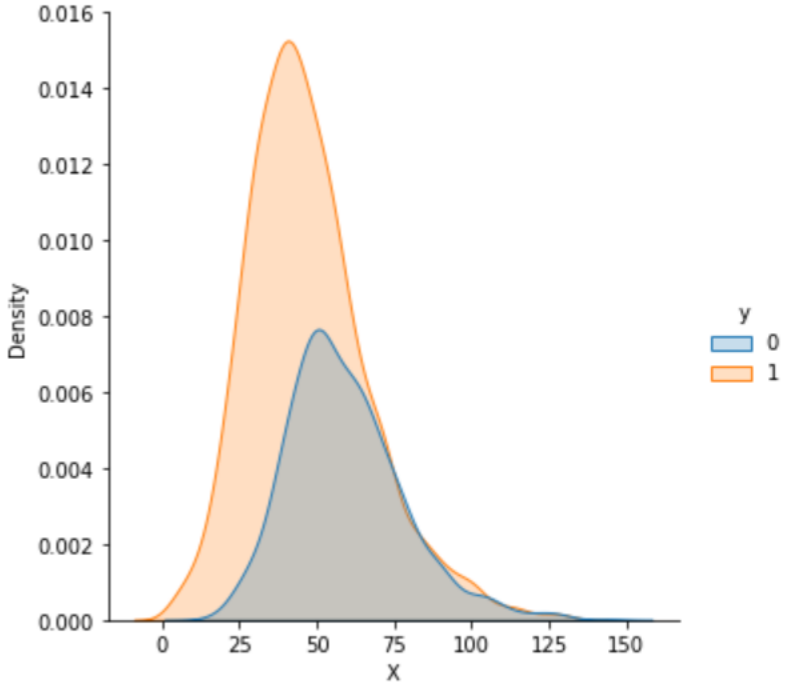
QQP Quora Question Pairs, consiste à identifier si deux questions sont des paraphrases

L'ensemble de données MRPC est constitué de 5801 paires de phrases récoltés sur internet accompagnés d'une évaluation humaine indicant si ces deux phrases sont des paraphrases l'une de l'autre ou non. Cet ensemble est séparé en un corpus d'entrainement de 4076 paires de phrases et un corpus de test de 1725 paires de phrases. L'évaluation du caractère équivalent des paires de phrases à été faite par un groupe de trois juges et à résultée en 3900 paires de phrases évaluées comme équivalente au sens sémantique et 1901 paires non équivalentes. Cette évaluation pourrait etre jugée arbitraire étant donnée que différents juges auraient obtenus différents résultats, néanmoins, la tâche conciste ici à détecter quelle paires de phrases sont sémantiquement équivalente selon ce groupe de juge. Ici, les paires représentant des paraphrases ont comme label 1 et 0 pour les autres.

2 QQP Quora Question Pairs,

2.1 Modèle de base

Pour cette tâche, nous avons utilisé comme baseline un modèle calculant la distance Levenshtein entre les paires de phrases, puis utilisé une régression logistique afin de classifier ces paires comme étant des paraphrases ou non. Comme il y a ici qu'un seul attribut par paire, la régression logistique revient à un régression linéaire. La figure ci-dessous présente la distribution des distances Levenshtein pour chacune des classes.



La métrique utilisée pour quantifier la performance des modèles appliqués à cet ensemble de donné est la précision par classe, soit le nombre de point bien classé de cette classe divisé par le nombre de point total de cette classe. Notre classifieur utilisant une régression logistique présente les précisions suivantes:

Label	0	1
Précision	0.57	0.63

Table 1: Performance de précision du baseline (Régression logistique sur les distances Levenshtein)

Le pourcentage général de paires classés dans le bon groupe est donc de 65% pour notre modèle baseline.

3 Amélioration des performances

Notre première tentative pour améliorer les performances sur cette tâche a été de vectoriser les phrases à l'aide de l'algorithme TF-IDF. La méthode TF-IDF vise à exprimer l'importance d'un mot relativement à un ensemble de textes. Cette aproche se résume en un équillibre entre la fréquence d'un terme dans un corpus de texte (tf : term-frequency) et l'inverse de la fréquence de ce terme dans un document (idf : inverse-document-frequency). Nous avons donc calculé les

plongements avec TF-IDF de la concaténation des paires de phrases séparés par un placeholder, puis avons classifiés ces plongements avec une régression logistique. Les performances de ce modèle sont présentés dans la table X.

Label	0	1
Précision	0.63	0.69

Table 2: Performance de précision du modèle de régression logistique sur les plongements TF-IDF

La précision sur l'ensemble des classe était ici de 69%, donc mieux que le baseline. Ici nous aurions pu explorer la régularisation sur la régression logistique mais nous avons trouvé d'autres méthodes plus performantes par la suite, cette étape n'a donc pas été nécéssaire.

Nous avons ensuite essayé de classifier les plongements TF-IDF avec un perceptron multicouche. Ici, nous avons constaté un problème de surapprentissage, car la précision avec les paramètres par défault du modèle sur l'ensemble d'entrainement était beacoup plus élevé que sur celui de test. Nous avons donc tenté de régulariser le modèle avec les paramètres α, le terme de pénalisation L2, en arrêtant l'entrainement avant convergence ou en utilisant moins de noeuds et de couches. Ces hyperparamètres ont étés optimisés avec une recherche en grille, et les valeurs optimales sont les suivantes : Le modèle contient une seule couche de 100 noeuds, le terme de pénalisation α est égal à 0.1 et l'entrainent est arrêté dès que la précision n'augmente pas sur 10% des données qui ont étés mise de coté à cette fin. La table X présente les performances de ce modèle.

Label	0	1
Précision	0.60	0.71

Table 3: Performance de précision du modèle perceptron multicouche sur les plongements TF-IDF

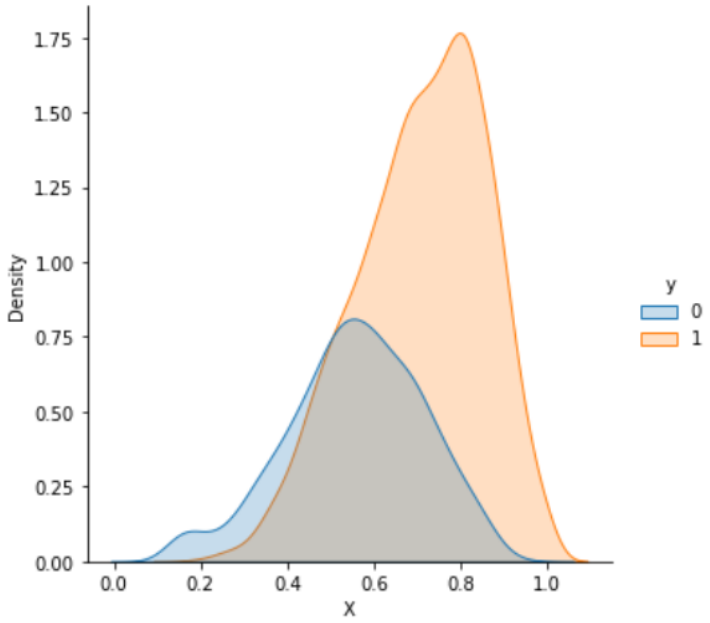
On voit donc que les performances ne sont pas supérieur à celle de la regression logistique. En effet, la précision générale de ce modèle est également de 69%.

Finalement nous avons tenté de calculer la distance cosinus entre les paires de plongements TF-IDF et de classifier ces derniers avec une régression logistique. C'est ce modèle qui a été le plus performant, offrant une précision générale de 72%. Les performances sur les deux classes sont présentés dans la table X.

Ici, aucun hyperparamètres n'ont étés à êtres otpimisés, comme la régression logistique en une seule dimension correspond à une régression linéaire. La distribution des distances cosinus est présenté dans la figure ci-dessous :

Label	0	1
Précision	0.64	0.74

Table 4: Performance de précision de la régression linéaire sur les distances cosinus entre les plongements TF-IDF



3.1 Résultats

Pour cette tâche, notre meilleur modèle est donc une régression logistique sur les distances cosinus entre les paires de plongements TF-IDF. Toutes les librairies utilisés pour ce modèles sont celles de *scikit-learn*. Les performances de précision sur les deux classes sont de 72% et les précisions par classes sont présentés dans la table X.

3.2 Analyse des lacunes

Afin d'analyser les problèmes avec notre meilleur modèle, nous avons générés différents groupes de 10 paires de phrases possédant chacun un critère permettant d'identifier si notre modèle avait une certaine capacité. Le premier groupe est composé de paires de phrases identique à l'exception d'un chiffre qui change donc le sens de la seconde phrase. Notre modèle a classifié chacune de ces paires comme étant des paraphrases. Le deuxième groupe est composé de phrases dont un seul mot a été changé en s'assurant que celui-ci changeait complètement le sens de la phrase. Encore une fois, notre modèle a classifié chacune de ces paires comme étant des paraphrases. Le troisième groupe est composé de paires de phrases ou la seconde est une négation de la première. Ici aussi, toute les paires ont étées identifiées comme des paraphrases. Finalement les deux autres groupes sont composés de paraphrases plus simples que celles proposés dans le jeux de données originales. Ici, notre modèle a correctement identifié 55% des paires de phrases. Nous avons donc

identifié plusieurs lacunes de notre modèle. Celui-ci ne reconnaissait comme paraphrase des paires ou des chiffres avaient étés changés, ou des mots avaient étés remplacés par leur contraire et ou des négations avaient étés rajoutées. Les autres modèles que nous avons étudiés présentaient également ces mêmes problèmes et nous n'avons pas pu trouver de solution à ceux-ci.

4 SST: The Stanford Sentiment Treebank

4.1 Introduction

Dans ce document, on utilise des modèles neuronaux pour faire une prédiction sur des phrases du corpus de développement de **The Stanford Sentiment Treebank** afin d'identifier si une phrase est positive ou négative.

5 Analyse I : Un système de base

5.1 Préparation des données

Avant de créer et d'entrainer notre modèle de base, il faut tout d'abord préparer les données. Pour mieux gérer la mémoire, on utilise les fonctions *cache()* et *prefetch()* pour optimiser le canal (pipeline) de notre Input. Et puis on prépare 3 sets différents de donnée :

- **train dataset :** On utilise les phrases du corpus *train* pour entrainer notre modèle. On obtient 59127 phrases pour l'entrainement.
- **validation dataset :** On utilise 872 phrases pour la validation pendant l'entrainement.
- **test dataset :** On utilise 872 phrases pour évaluer notre modèle après l'entrainement a la dernière étape.

5.2 Architecture

Notre système de base est constitué tout d'abord d'une couche pour vectoriser note input et créer un vocabulaire de 40000 entrées. Puis, on réduit les dimensions de l'entrée à l'aide d'une couche *Embedding* pour retourner des vecteurs de taille 16. On utilise après une couche *dropout* pour éviter l'overfitting sur nos phrases suivie d'une couche de *global pooling* à une dimension. À la fin, on prend une couche *fcc (fully connected layer)* avec une fonction d'activation *sigmoid* pour avoir notre classement a l'output.
Ce modèle est représenté dans la figure suivante.

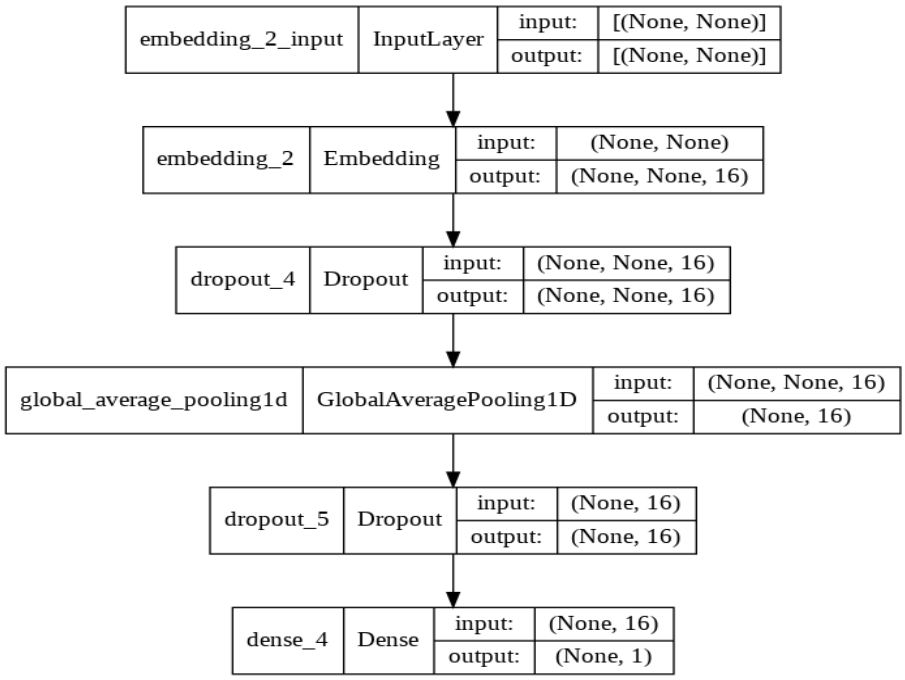


Figure 1: Architecture du modèle de base

5.3 Entrainement

Pour l'entraînement, on choisit la perte binaire comme fonction *BinaryCrossentropy()* et on entraine le modèle pour 10 époques avec un optimiseur *Adam* avec la précision binaire *BinaryAccuracy* comme métrique.
Ce modèle prend environ **17s** par époque et environ **3mn** pour s'entrainer sur *Google Colab*.
On peut examiner l'historique de l'entrainement de ce modèle de base à travers les figures suivantes :

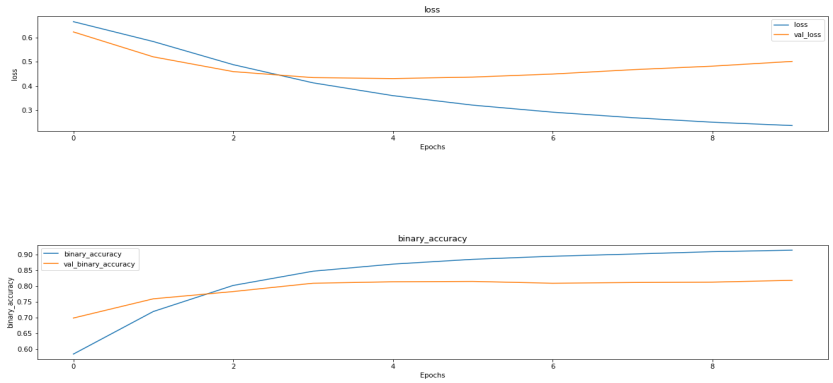


Figure 2: Courbe perte (1) et précision (2) en fonction du nombre d'époques durant l'entrainement de système de base

5.4 Évaluation

Sur notre set de test, ce modèle simple achève une perte de **0.29** et une précision binaire de **89.56%** sur le *test dataset*. Ce résultat est vraiment impressionnant en considérant la simplicité de ce modèle et le temps court mis pour l'entrainer.

6 Analyse II : Meilleur système trouvé

6.1 Architecture

Notre modèle est constitué tout d'abord d'une couche pour vectoriser note input et créer un vocabulaire de 40000 entrées. Puis, on réduit les dimensions de l'entrée à l'aide d'une couche *Embedding* pour retourner des vecteurs de taille 16.
On utilise après deux couches *LSTM* bidirectionnelles et empilées avec un nombre d'unités cachées à optimiser dans la prochaine étape. Pour les deux on utilise deux couches *dropout* à 50% pour éviter l'overfitting sur nos phrases. Après on utilise une succession de 3 *fcc (fully connected layer)*, les deux premiers avec une fonction d'activation *relu* et la dernière avec une fonction d'activation *sigmoid* pour avoir notre prédiction entre 0 et 1 a l'output. Ces couches sont tous entrelacé d'une couche *dropout* à 30% et 20% respectivement.
À la fin, on définit un optimiseur *Adam* avec un *'learning rate* à optimiser dans la prochaine étape et *Binary Crossentropy* comme fonction de perte. Ce modèle est mieux représenté dans la figure suivante :

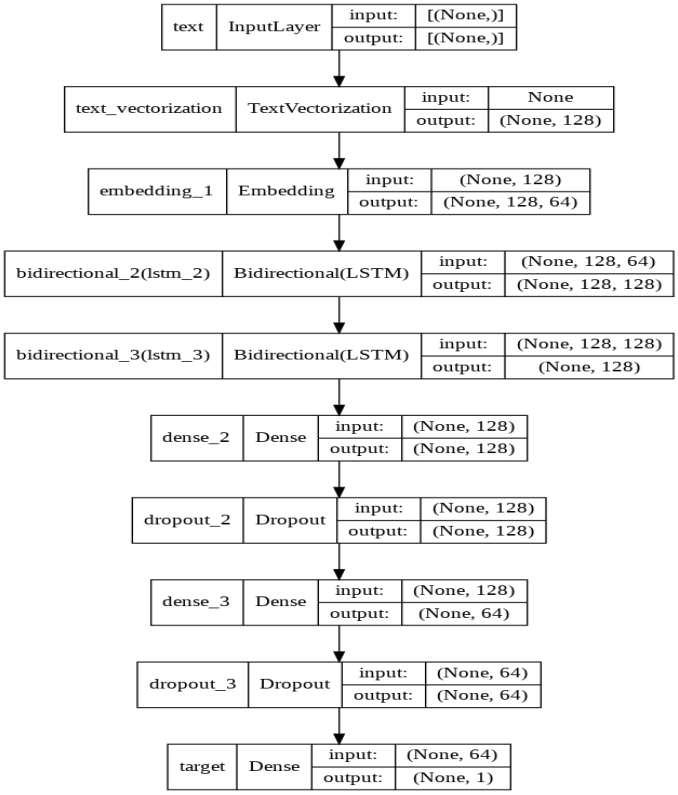


Figure 3: Architecture du modèle complexe

6.2 Recherche de la configuration optimale

Après avoir défini notre modèle, on passe à la recherche de la configuration parfaite à l'aide d'un *BayesianOptimization* de *Keras* a 7 essais. Pour notre unité cache des deux *LSTM*, on définit un intervalle entre 32 et 128 pour la recherche. Et pour le *'learning rate* de notre optimiseur, on limite la recherche sur ces 3 options : [1e-3, 1e-3, 1e-4].
Puis, on lance la recherche avec 10 époques et en utilisant un callback *'early stop* avec une patience de 3 pour réduire le temps de la recherche.
Cette recherche prend en moyenne **750s** par époque et environ **125mn** par essai sur *Google Colab*.
Le résultat de la recherche est **128** pour les unités cachées des deux *LSTM* et **1e-4** pour le *'learning rate*. // Mais due à la restriction de Colab pour les temps d'entrainement, on a dû diminuer la taille de notre modèle pour pouvoir l'entrainer plus rapidement. C'est pourquoi on va prendre **64** pour les unités cachées des deux *LSTM*.

Model's Hyper Parameters (lr: learning rate, hu: LSTM's hidden units)	*Validation Loss (5 epochs)*
(hu = 16) and (lr = 1e-4)	0.4391
(hu = 16) and (lr = 1e-3)	0.4621
(hu = 16) and (lr = 1e-2)	0.4497
(hu = 32) and (lr = 1e-4)	0.4240
(hu = 64) and (lr = 1e-4)	**0.4051**

6.3 Entrainement

On entraine ce modèle sur 10 époques avec un optimiseur *Adam* avec la précision binaire *Binary-Accuracy* comme métrique. Ce modèle prend environ **125mn** pour s'entrainer sur *Google Colab* et donne à la fin une précision binaire de **95,61%** sur le *train dataset* et **81,08%** sur le *validation dataset*.

6.4 Évaluation

Ce modèle relativement complexe achève une perte de **0.26** et une précision binaire de **92,43%** sur le *test dataset*.
En comparant avec notre modèle de base, ce modèle a achevé une augmentation de précision de **2,87%**. Ce modèle est meilleur en prédiction, mais il prend aussi environ **41 fois** plus de temps pour s'entrainer, ce qui fait que cette augmentation est à la fin pas très impressionnante.
Ce résultat montre qu'en effet, notre modèle de base était dès le début un peu complexe et a pu capturer les relations entre les données et les labels ce qui fait que pour cette tache, ce classificateur est considéré bon.

6.5 Génération des données

Dans cette partie, on analyse les phrases (quel que soit avec un label positif ou négatif) ou notre modèle a fait une fausse prédiction, puis on essaie de capturer cette erreur et de générer des nouvelles phrases à partir de ces erreurs.
On constate d'abord que pour certaine phrase, le label est dès le départ faux comme **"find it diverting"** qui est synonyme d'amusant, mais qui est étiqueté comme négatif. Après, on observe que notre modèle est fautif quand il y a des oppositions dans les phrases, par exemple : **"far from annoying"** ou **"funny enough, but nothing new"**.
Aussi, notre modèle parfois ne peut pas bien comprendre la phrase si elle introduit des termes

péjoratifs comme **"viewers don't have to worry about being subjected to foul and offensive jokes"**.
Finalement, le modèle ne peut pas capturer certaines références culturelles ou sociales, comme **"classic hollywood"** qui est plutôt péjoratif et pas mélioratif. Une solution serait d'introduire plus d'exemples de ce genre pour que le modèles puisse capturer ces motifs.
À la fin, on teste notre modèle sur ces 50 nouvelles phrases générées : Il achève une perte de **1,76** et une précision binaire de **40,00%**. Ce résultat est surprenant, du fait que toutes les phrases sont très proches des phrases qui sont mal classifié par notre modèle. (générées avec des synonymes, constructions syntaxiques similaires, etc.)

7 CoLa : Corpus of Linguistic Acceptability

7.1 Introduction

Le Corpus of Linguistic Acceptability (CoLA) dans sa forme complète se compose de 10657 phrases de 23 publications linguistiques, annotées de manière experte pour l'acceptabilité (grammaticalité) par leurs auteurs originaux. La version publique que nous utiliserons contient 9594 phrases appartenant à des ensembles de formation et de développement, et exclut 1063 phrases appartenant à un ensemble de tests retenus

7.2 Première étude des données

Chaque ligne des fichiers .tsv du corpus se compose de 4 colonnes séparées par des tabulations dans l'odre ci-dessous :

- Le code représentant la source de la phrase
- L'étiquette de jugement d'acceptabilité (0 = inacceptable, 1 = acceptable)
- Le jugement d'acceptabilité tel qu'il a été initialement noté par l'auteur
- La phrase

Nous utiliserons le corpus tokenized-in_domain_train.tsv (8551 lignes) pour la phase d'entrainement et le corpus tokenized-in_domain_dev.tsv (527 lignes) pour la phase de test.

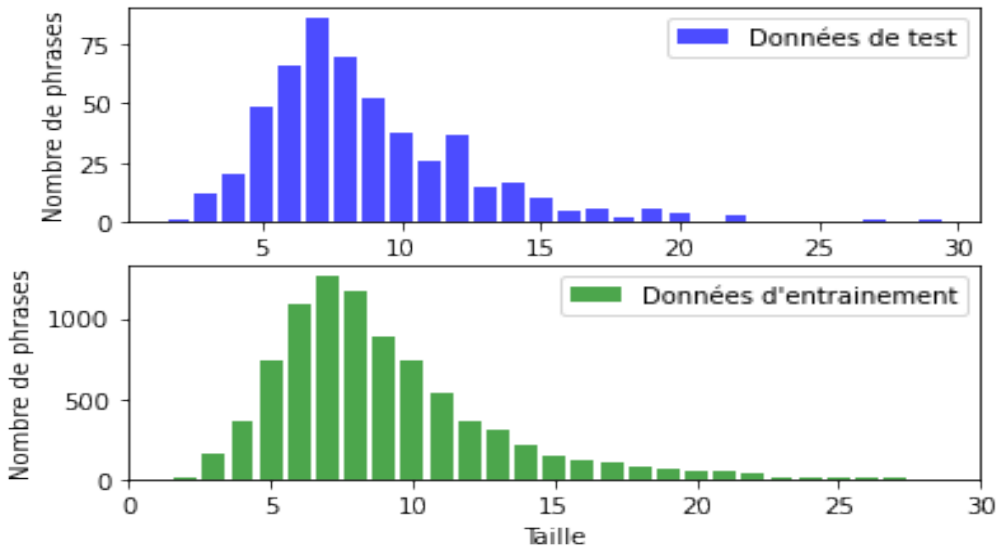


Figure 4: Nombre de phrases en fonction de la taille de celles-ci

Nous utiliserons l'étiquette du jugement d'accessibilité afin d'entrainer un modele et d'être en mesure de determiner si une phrase est correcte ou non. la répartition des étiquettes correspond à la figure ci-dessous :

Phase	Entrainement	Test
Classe 0	2528	160
Classe 1	6023	367

Table 5: Repartition des classes durant l'entrainement et le test

7.3 Systeme de base

Nous utiliserons un classificateur binaire pour séparer nos données. Une premiere idée serait d'encoder les phrases d'entrainement et de test en utilisant des nombres différents pour chaque mot du vocabulaire (encodage à chaud). Ces étapes de travail peuvent être résumées avec la figure suivante :

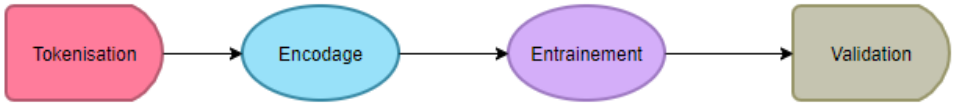


Figure 5: Workflow

La taille totale du vocabulaire est de 1328 mots pour l'entrainement. La particularité des données utilisées pour le test est qu'il n'existe pas de mots qui n'appartiennent pas aux vocabulaire, ce qui évite de mettre en place des procédures dans le cas de mots inconnus.

Initialisation : $i = 0$
Pour tous les mots ω **du vocabulaire** A **:**
  Si $\omega \notin A$ **:**
    $A(\omega) = i$
    $i = i + 1$

Figure : Algorithme d'encodage des mots

Pour l'entrainement le système de classification deux possibilités ont été testées avec sklearn :

- Support Vector Machine (SVM)
- Logistic Regression

SVM est un modèle utilisé à la fois pour les problèmes de classification et de régression, cet algorithme utilise l'astuce du noyau (kernel trick) pour trouver le meilleur séparateur de ligne. L'astuce est de reconsidérer le problème dans un espace de dimension supérieure, éventuellement de dimension infinie. Dans ce nouvel espace, il est alors très probable qu'il existe une séparation linéaire.

Pour la régression logistique, ce n'est pas la réponse binaire (phrase correcte/incorrecte) qui est directement modélisée, mais la probabilité de réalisation d'une des deux modalités (la phrase est correcte par exemple).
Cette probabilité de réalisation ne peut pas être modélisée par une droite car celle-ci conduirait à des valeurs 0 ou 1. Cette probabilité est donc modélisée avec des fonctions non linéaires comme la fonction sigmoïde.

7.4 Résultats

En appliquant les deux algorithmes, on obtient les résultats ci-dessous :

Algorithme	SVM	RL (Poly d5)	RL (RBF)	RL (Sigmoïde)
Accuracy	66.8	67.2	67.4	67.2

Figure : Résultats pour le système de base

La regression a été appliquée une fois en utilisant une fonction de base radiale, un sigmoïde et un polynome de degrés 5. On remarque que toutes ces méthodes donnent à peu près le même résultat. En utilisant les figures ci-dessus, on peut constater deux choses :

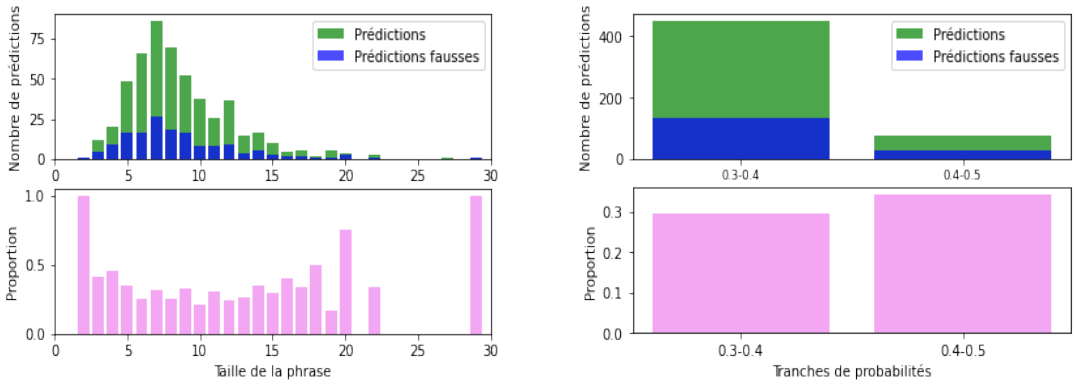


Figure 6: Résultats pour la régression logistique avec fonction de base radiale

- Tout d'abord la taille des phrases ne semble pas corrélée eux résultats de prédiction. Bien que la proportion de prédictions fausses pour les phrases très longues et très courtes est plus importante, ce phénomène est probablement dû aux faibles nombres de données de test de longueur extreme.
- Ensuite, on remarque que l'ensemble des probabilités données en guise de résultat sont proches 0.5, ce qui implique que le modèle utilisé est toujours hesitant dans ces résultats. Cela n'est pas bon signe car l'accuracy calculée a des chances d'être extremement corrélée aux données de test.

8 Amélioration du modèle de base

8.1 Word2Vec

Une premiere idée serait de modifier la méthode d'encodage qui ne prend pas en compte le contexte des mots dans les phrases. A la place d'utiliser un encodage à chaud nous utiliserons une technique de plongement lexical (Word2Vec) afin de capturer le maximum d'informations.

En utilisant la librairie gensim nous avons été capable de représenter les mots dans un espace de dimension k que nous ferons varier.

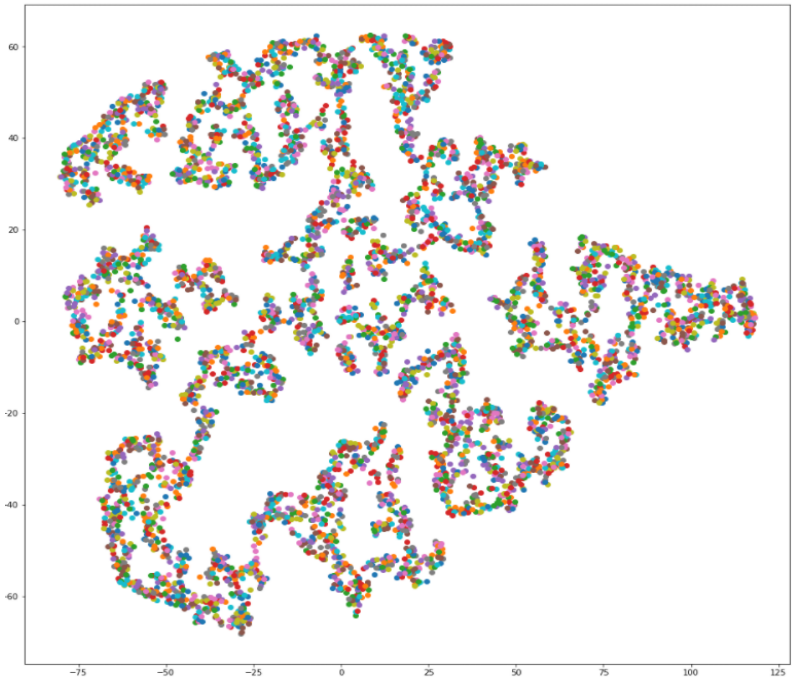


Figure 7: Représentation du vocabulaire pour $k = 2$ en utilisant sklearn.TSNE

En utilisant cette méthode on obtient les résultats ci-dessous :

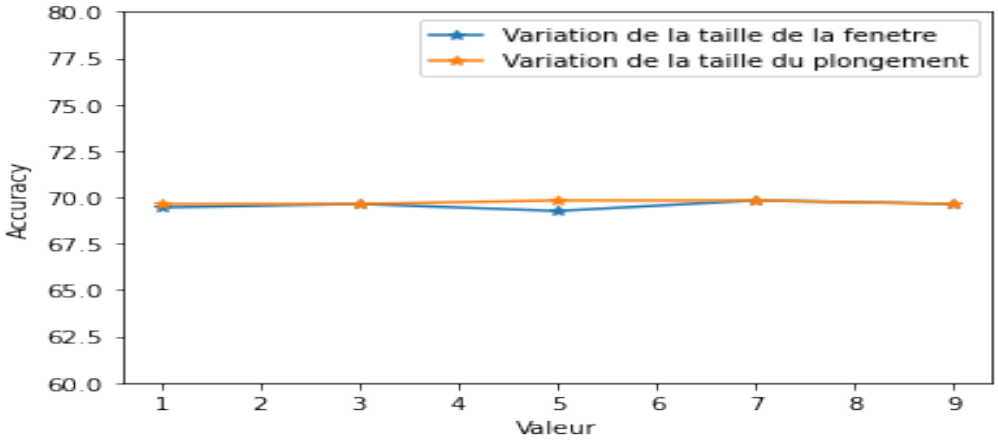


Figure 8: Accuracy en fonction de la taille de la fenêtre et du nombre de dimensions

Sur la figure ci-dessus, on constate une légere amélioration du résultat. Les deux paramètres ne semblent toutefois pas directement corrélées au résultat.

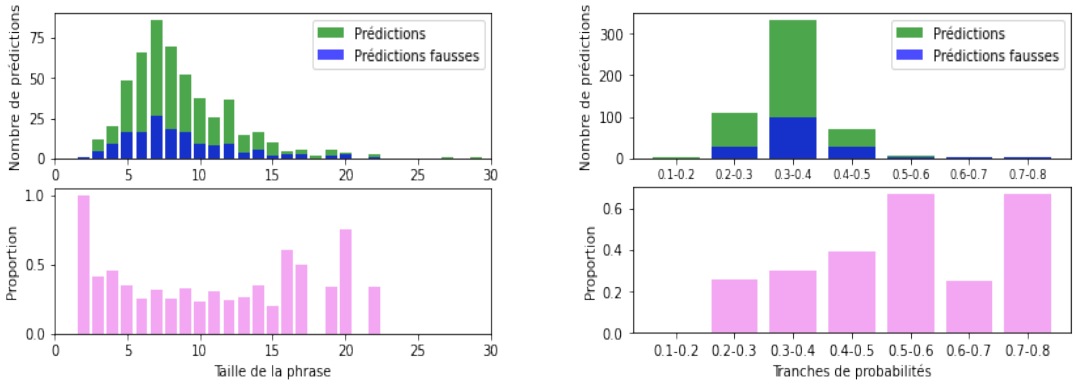


Figure 9: Résultats pour la régression logistique avec fonction de base radiale et encodeur Word2Vec

Bien qu'il existe désormais des phrases pour lesquelles le modèle arrive plus à trancher on constate toujours une faible répartition des probabilitées. D'autre part il existe aussi certains cas ou la proabilité est très favorable envers une classe alors que cette classe n'est pas la bonne. Quelque chose n'est donc intrinséquement pas bon dans prise la décision.
De plus en analysant les phrases problématiques, la plupart des erreurs sont des fautes de grammaire et non de placement de mots. Nous en déduisons que la seule maniere d'améliorer le modèle est peut-être de changer de features.

8.2 Grammaire + Word2Vec

Une nouvelle méthode est la prise en compte de la grammaire en tant que features. Pour effectuer cette tâche nous avons utilisé la libraire spacy.

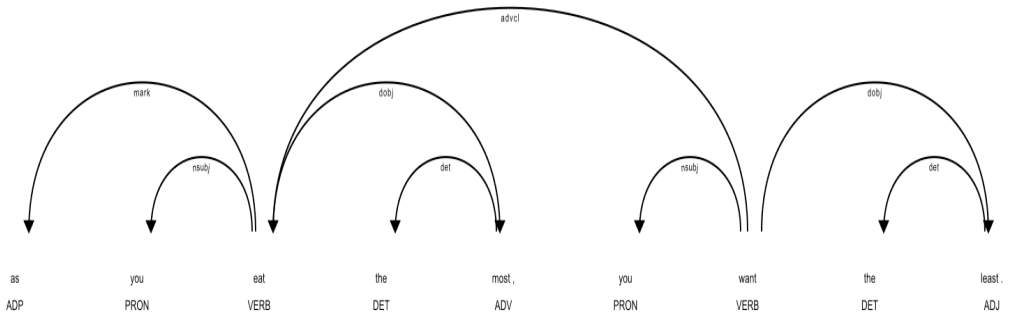


Figure 10: Exemple de phrase problématique analysée avec spacy

La grammaire des phrases est donc encodée avec spacy puis plongée dans un espace de dimension 1 (car la taille du vocabulaire est de seulement 17). L'utilité d'utiliser un Word2Vec avec la grammaire est d'être en mesure de capturer les liens entres les différents élements de grammaire.

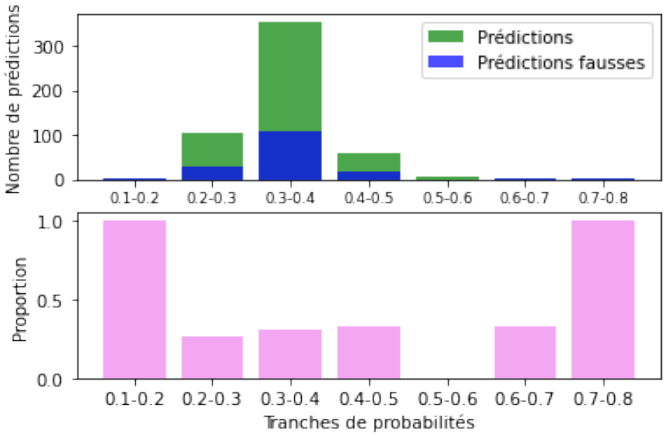


Figure 11: Résultat du test en utilisant l'analyse grammaticale

En utilisant une fenetre de longueur2, une seule dimension de plongement et une regression logistique avec fonction de base radiale en guise de classificateur on obtient une accuracy de 69.3, ce qui est moins bien que le modèle précédent. Sur la figure on constate une répartition des probabilités très similaire au modèle précédent. En analysant plus finement les phrases sur lesquels le modèle s'est trompé, on constate que sur 160 erreurs seulement 3% sont des erreurs en désignant la phrase comme incorrecte. Toutes les autres erreurs concernent l'autre classe (prédiction phrase correcte pour des phrases incorrectes).

D'autre part, on constate que dans les données d'entrainement 2528 correspondent à la classe 0 et 6023 correspond à la classe 1. Une des deux classes ne constitue donc que 29.5% des échantillons de test. Changer le taux d'apprentissage pour agrandir la mémoire n'as pas eu d'effet. Ainsi deux méthodes ont été proposées :

- réduire le nombre d'échantillons de classe 1, puis mélanger le nombre total d'échantillons
- utiliser un modèle neuronal type LSTM

La premiere méthode a été testée mais sans réelle succès (diminution de l'accuracy).