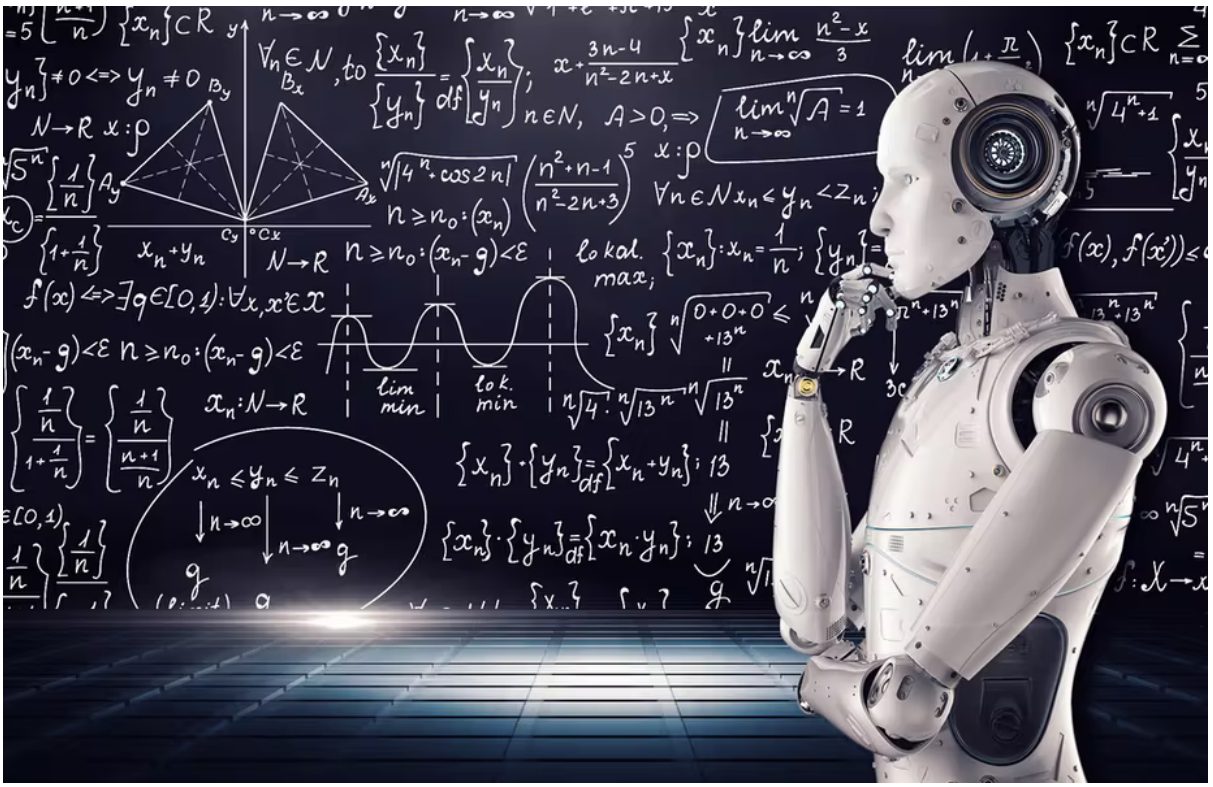


“L'apprentissage profond est une partie de l'apprentissage automatique qui concerne les algorithmes inspirés de la structure et du fonctionnement du cerveau, appelés réseaux neuronaux artificiels.

L'apprentissage profond est une technologie clé pour les voitures sans conducteur, qui leur permet de reconnaître un panneau stop ou de distinguer un piéton d'un lampadaire.

L'apprentissage profond fait l'objet d'une attention particulière ces derniers temps, et ce pour une bonne raison. Il permet d'obtenir des résultats qui n'étaient pas possibles auparavant.”

Je ne pouvais pas commencer cet article sans donner la parole à une IA ! Ces premières lignes, qui définissent le Deep Learning, proviennent en effet d’un algorithme de génération de texte basé sur le modèle GPT13 proposé par la société Copy.ai.

Une bonne introduction selon moi à cet article qui traitera 3 approches différentes, dans le domaine du traitement naturel du langage (NLP), permettant de détecter le sentiment associé à un texte :

1. L'approche “***API sur étagère***”, qui consiste à faire appel à l’API du service cognitif de Microsoft Azure.
2. L’approche “Modèle simple”, consistant à créer un modèle par simple glissé/déposé à partir du concepteur de Microsoft Azure Studio.
3. L’approche “Modèle avancé”, consacrée à l’élaboration puis l’optimisation d’un modèle de Deep Learning.

Contexte & Environnement technique

Nous disposons d'un jeu de données contenant 1 600 000 tweets collectés entre juin et avril 2009. Notre objectif est de prédire le sentiment positif ou négatif associé à chaque tweet, dans le but de pouvoir détecter d'éventuelles Bad Buzz.

Afin d'obtenir des résultats dans des temps raisonnables, nous utiliserons 0.1% de ce jeu de données soit 1600 tweets avec une parité stricte dans les labels (les sentiments associés aux textes : 0 pour un sentiment négatif, 1 pour un sentiment positif).

		Classe réelle	
		-	+
Classe prédite	-	True Negatives *(vrais négatifs)*	False Negatives *(faux négatifs)*
	+	False Positives *(faux positifs)*	True Positives *(vrais positifs)*

Ce projet fait donc partie de la famille de la classification binaire, nous utiliserons par conséquent les métriques adaptées à cette problématique pour évaluer et comparer nos modèle : Précision et Rappel, accuracy et temps de calcul.

Dans le domaine de l'intelligence artificielle, le matériel technique est fondamental, car les résultats obtenus peuvent en effet varier du simple au triple en fonction des appareils, des packages ou encore du système d'exploitation installé.
J'essaie donc d'être le plus précis possible, tout au long de cet article sur l'aspect de l'environnement technique de ce projet.

L'ensemble des résultats présentés par la suite ont été obtenus à partir d'un environnement technique basé sur le langage de programmation Python dans sa version 3.8.12.

Mon système d'exploitation macOS Monterey version 12.2, puce M1 Apple silicone (à noter que de nombreuses bibliothèques ne sont pas encore totalement compatibles aux nouvelle puce ARM d'Appel, mais que de nombreux tutoriels existent et expliquent comment contourner ces difficultés).

Commençons à présent !

Approche “*API sur étagère*”

Microsoft Azure est la plateforme des services cloud proposée par Microsoft depuis février 2010, (anciennement Windows Azure).

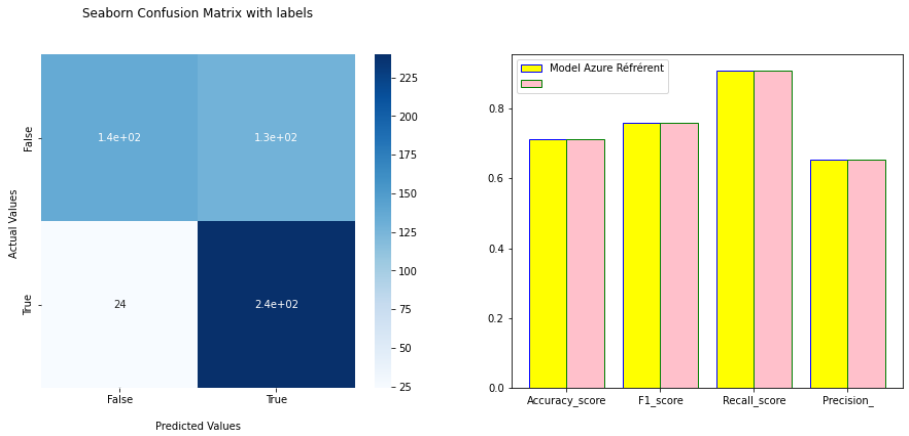
Dans notre cas, nous ferons appel au service cognitif qui consiste à se connecter à l’API dédiée à l’analyse des sentiments. Pour cela, il faut nécessairement avoir un compte Microsoft Azure, générer sur son espace en ligne le groupe de ressource contenant l’API puis se connecter à celle-ci via une clé et un point de terminaison. (Plus de détails ici).

```
def get_sentiment_from_Azure(x):

    def authenticate_client(key,endpoint):
        ta_credential = AzureKeyCredential(key)
        text_analytics_client = TextAnalyticsClient(
                endpoint=endpoint,
                credential=ta_credential)
        return text_analytics_client

    key = "key##########################"
    endpoint = "https://detectionsentiment.cognitiveservices.azure.com/"
    client = authenticate_client(key,endpoint)

    return client.analyze_sentiment([x])[0].confidence_scores.values()
```

La bibliothèque azure-ai-text analytics dans sa dernière version 5.1.0 est également nécessaire.

Une fois l’ensemble connecté l’utilisateur peut envoyer du texte sous forme de chaîne de caractère, l’API retourne quant à elle la probabilité des sentiments associés : Neutre, Positif et Négatif. Nous appliquons ensuite une régression linéaire qui permet de séparer nos données en deux classes.

L’obtention de nos données s’est effectué en 5 minutes et 34 secondes, l’instance de calcul étant située en France Central. A noter qu’aucun traitement sur notre jeu de données n’a été apporté.



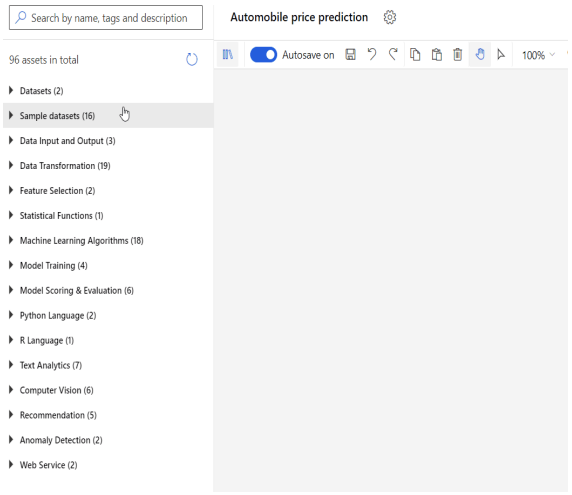
Sur la figure de gauche, nous pouvons observer la matrice de confusion qui permet de visualiser la proportion entre les valeurs prédites positives ou négatives avec les valeurs réelles. Ainsi, on peut constater que le modèle prédit très bien les tweets positifs et se trompe très peu lorsqu'il prédit un tweet positif bien que la part de prédictions erronées sur les tweets positifs soit non négligeable.

Sur la figure de droite, nous observons les scores d’accuracy, la proportion de bonnes réponses, qui est de 70%, de rappel, la proportion de tweet vrai-positif par rapport à l’ensemble des prédiction, de l’ordre de 90% et enfin le score de précision, la proportion de vrai positif sur l'ensemble des positifs prédits, 65%.

Nous observons donc de très bons scores pour cette première approche. Nous utiliserons ces résultats comme point de comparaison par la suite.

Approche “*Modèle sur mesure simple*”

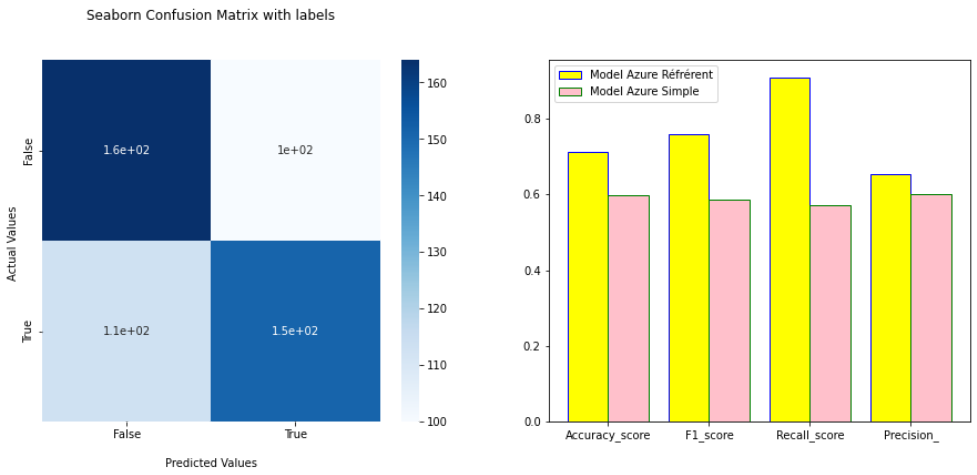
Nous l’avons vu, l’API du service cognitif d’Azure offre de très bons résultats, voyons à présent si les autres services de Microsoft Azure peuvent en faire autant !



Microsoft Azure Machine Learning Studio est défini comme la ressource de niveau supérieur pour le Machine Learning, c’est un environnement de développement permettant d’accéder divers ressource telques des Notebooks, des base de données, des instances de calculs et en ce qui nous concerne un outil appelé “Concepteur”.

Cet outil permet via de simple glisser/déposer de concevoir les processus de traitement de données nécessaire en Machin Learning (nettoyage, pré-processing, entraînement d’un modèle, évaluation des score).

Voici la ma configuration retenue : **1.** Import Data, **2.** Select Columns in data : “text” et “Label”, **3.** Preprocess Text, **4.** Extract N-Gram Features from Text, **5.** Split Data, 6 MultiCass Boosted Decision Tree, **7.** Train Model, **8.** Score Model, **9.** Evaluate Model.
L'entraînement a pris 12 minutes et 34 secondes avec l’instance de calcul Standard_D11_v2 (2 cœurs, 14 Go de RAM, 100 Go de disque (0.23$/h).



Nous obtenons des scores corrects mais inférieurs au modèle de référence :

- Accuracy : 60%
- Rappel : 57%
- Précision : 65%

Ce modèle prédit très bien les tweets négatifs et se trompe très peu sur les prédictions de tweets positifs.

Approche "*Modèle sur mesure Avancé*"

La dernière approche que nous observerons traitera de la conception d'un modèle de Deep Learning que nous déploierons sur les serveurs de Microsoft Azure.

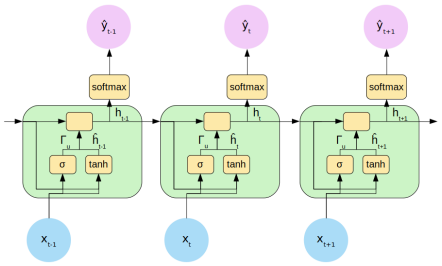
Ce type de démarche se déroule en plusieurs étape afin de fournir à l'algotyme des données qu'il pourra traiter, différentes configurations ont été testé afin de vous présenter celle qui apporte les meilleurs résultats :

1. **Prétraitement du texte :** Le corpus, l'ensemble des textes du jeu de données, a été nettoyés des éléments n'étant ni des caractères alphabétiques ni de la ponctuation, les mots du corpus, ou tokens, ont subit une lemmatisation, qui consiste à donner la forme neutre canonique du mot, puis une stemmatisation qui consiste à conserver la racine du mot. Voici un exemple d'un pré-traitement :

```
initial :  @kenichan i dived many times for the ball. managed to save 50%  the rest go out of bounds
final : i dive mani time for the ball . manag to save % the rest go out of bound

Exemple de lemmization : managed --> manage
Exemple de Stemming : managed --> manag
```

2. **Word Embedding :** Cela consiste à transformer notre corpus en matrice de nombre réel qui sera ensuite comprise par notre modèle de deep learning, la taille de la matrice pouvant évoluer en fonction du nombre de mot contenu dans le vocabulaire du corpus, la longueur arbitraire des tweets et enfin la la taille de l'espace dans lequel nous projetons la représentation de nos tokens. Les méthodes de plongement de mot (ou word Embedding) Word2Vec et Fasttext ont été testées, mais c'est la couche d'Embedding def Keras qui apporte les meilleures performances, car elle offre l'avantage d'avoir reçu un entraînement sur un très large corpus.

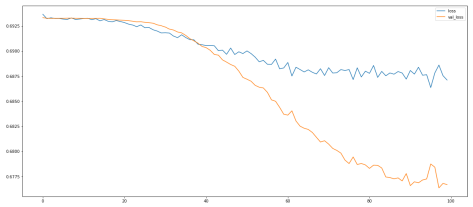
3. **Construction du modèle :** Nous utilisons la bibliothèque keras (installation pour mac M1 ici) pour l'empilement de nos couches de neurone. L'architecture Long Short-Term Memory (LSTM) a été retenue pour notre modèle, c'est une architecture basée sur les réseau de neurone récurrent (RNN) le plus adapté aux données séquentielles car son mécanisme permet de conserver ou d'oublier de l'information via un mécanisme de porte.



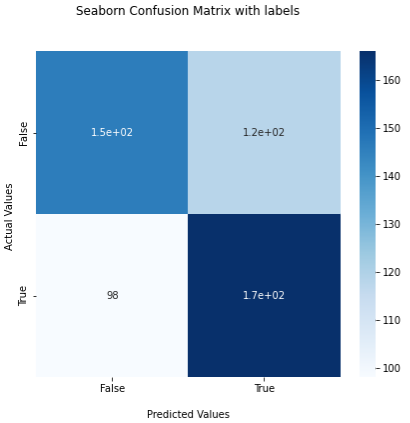

```python
def my_LSTM5(len_dict):
    model = Sequential()
    model.add(Embedding(input_dim=len_dict, output_dim=12, input_length=32))
    model.add(Bidirectional(LSTM(units=8, return_sequences=True)))
    model.add(Bidirectional(LSTM(units=8, return_sequences=False)))
    model.add(Dense(units=1, activation='sigmoid'))
    return model
```

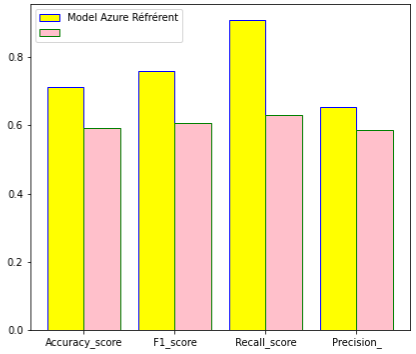
Notre modèle possède donc une couche séquentielle permettant d'initialiser le modèle, notre couche d'embedding keras, deux couches de 8 nœuds LSTM bidirectionnelles permettant de lire nos données de droite à gauche ainsi que de gauche à droite, et enfin une fonction d'activation sigmoid permettant de faire de la classification binaire.

4. **<u>Entraînement et optimisation du modèle :</u>** Enfin, pour l'entraînement de notre modèle, nous séparons notre jeu de données en deux : entraînement et validation, puis nous lançons l'apprentissage sur 100 epoch, (nombre de fois que l'algorithme verra les données), avec un callback à 10, c'est-à-dire que pour 10 itérations successives si le modèle ne progresse plus l'apprentissage s'arrête. Le temps d'entraînement était de 132 secondes.



5. **<u>Résultat et déploiement :</u>** Cette configuration offre elle aussi des performances inférieures au modèle de référence avec une accuracy de 0.58, un score de rappel de 0.69 et un score de précision égale à 0.65. La matrice de confusion nous révèle que le modèle fait très peu d'erreurs sur ses prédictions de tweets négatifs.





<u>Conclusion</u>

	Accuracy_score	F1_score	Recall_score	Precision_	Learing_time
Model Azure Réfrérent	0.71	0.76	0.91	0.65	331.04
Model Azure Simple	0.6	0.59	0.57	0.65	0.2
LSTM wt KERAS	0.58	0.62	0.69	0.56	132.41

Au vu de nos résultats, nous pouvons comparer ces trois approches et tirer une conclusion sur leurs avantages et inconvénients.

<u>API sur étagère :</u> Cette approche détient les meilleures performances et est la plus simple à mettre en place et permet d'obtenir les résultats le plus rapidement.

<u>Modèle sur mesure simple :</u> Bien qu'elle possède de bons scores, cette approche est la moins pratique à mettre en place, l'appel au modèle, le coût de l'utilisation du service et la relative faible étendue des possibilités d'optimisation des modèles en sont les raisons.

<u>Modèle sur mesure avancé :</u> Cette approche est la plus longue à mettre en œuvre, car de nombreuses stratégies d'optimisation et de prétraitement peuvent être mises en place. Selon moi, lorsque cette approche retenue, elle doit avoir comme objectif de se rapprocher voir dépasser les performances de l'approche API sur étagère avant d'être déployé afin de justifier le temps consacré à son perfectionnement !