Analyse de sentiment sur Azure Machine Learning

Cédric Dietzi – 02/03/2021

Dans ce post, nous proposons un retour d'expérience sur la réalisation d'un classificateur de sentiment de tweets sur Azure Machine Learning (Azure ML). L'application qui utilisera le classificateur permettra au client de détecter le 'bad buzz' concernant sa société sur les réseaux sociaux.

Le code est disponible à l'adresse [adresse du code].

Pour entrainer le modèle, nous disposons d'un corpus de 1.6 millions de tweets annotés 'positif' ou 'négatif'.

Dans ce document, nous avons conservé le jargon anglais qu'on utilise habituellement sans le traduire pour faciliter la compréhension.

Baselines

Une première baseline est obtenue par l'appel à l'API « Cognitive Service Sentiment Analysis » d'Azure ; une deuxième par le test d'un modèle classique via le Designer drag & drop d'Azure ML dans lequel nous avons implémenté un modèle 1-Gram et une simple régression logistique.

Modèle 1-Gram et régression logistique sur le Designer Azure ML

Les résultats sont les suivants :

	Accuracy
Cognitive Service Sentiment Analysis	73.1 %
Designer ; 1 Gram + Logistic regression	74.9 %

Baselines

On constate que sur un corpus très spécifique comme des tweets, le service d'Azure ML ne fait pas mieux qu'une approche (très) basique.

Peut-on améliorer cette situation de départ ?

Architectures à explorer

Pour une application donnée, il existe généralement des articles passant en revue les différentes approches possibles. Par exemple, un article récent de décembre 2019 pour l'analyse de sentiment dresse un panorama de l'état de l'art : '*Sentiment analysis using deep learning architectures : a review de A Yadav et D K Vishwakarma*'. Ces revues sont par ailleurs de bonnes sources vers les auteurs qui ont travaillé sur le sujet.

On constate que :

1. Les architectures en réseau de neurones sont les plus efficaces
2. Le préprocessing inclus typiquement un nettoyage des noms d'utilisateurs, hashtags et urls et une normalisation par stemming ou lemmatisation.
3. L'encodage des tokens est réalisé par un embedding pré-entrainé word2vec, glove ou fasttext
4. L'extraction de features est automatique et s'appuie sur un réseau convolutif, récurrent ou une combinaison des deux.
5. Pour gérer le risque d'overfitting, des éléments 'dropout' ou de régularisation sont introduits dans l'architecture.

Les architectures à explorer sont donc :

Brique	**Options**
Préprocessing	Stemming ou Lemmatisation
Word embedding	Word2Vec, Glove ou Fasttext (non testé)
Extraction de features	CNN ou LSTM ou CCN + LSTM
Dropout et régularisation	Hyperparamètres à optimiser

Architectures à explorer

CNN ou LSTM ?

L'extraction de features est l'élément conceptuel qui à priori distingue le plus ces architectures. On peut synthétiser les approches comme suit.

Le réseau convolutif (CNN) détermine lors de l'apprentissage un ensemble de motifs qui sont caractéristiques de la positivité ou négativité d'un tweet. Lors de la prédiction, le réseau évalue le niveau de présence de chaque motif dans le tweet et fournit ainsi les features pour la décision. Pour implémenter le CNN nous nous sommes inspirés de « *Convolutional Neural Networks for Sentence Classification de Yoon Kim* »

Le réseau récurrent (LSTM) apprend les motifs globaux caractéristiques de la positivité ou négativité d'un tweet. À la suite de l'apprentissage, il produit des features qui intègrent ces spécificités.

On peut combiner les deux approches. Ci-dessous, un schéma de principe d'une architecture qui combine les deux approches.

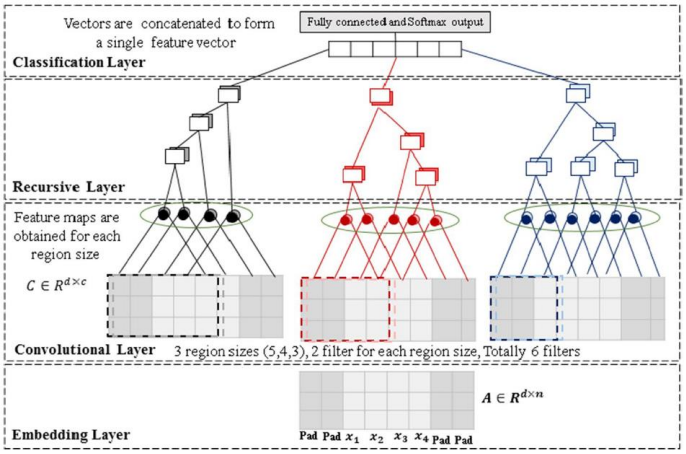


Schéma de principe CNN + LSTM (A robust sentiment analysis method based on sequential combination of convolutional and recursive neural networks - H Sadr)

Les simulations sur Azure ML nous diront quelle architecture retenir.

Entrainement

Hyperparamètres

Les hyperparamètres ont été choisis comme suit :

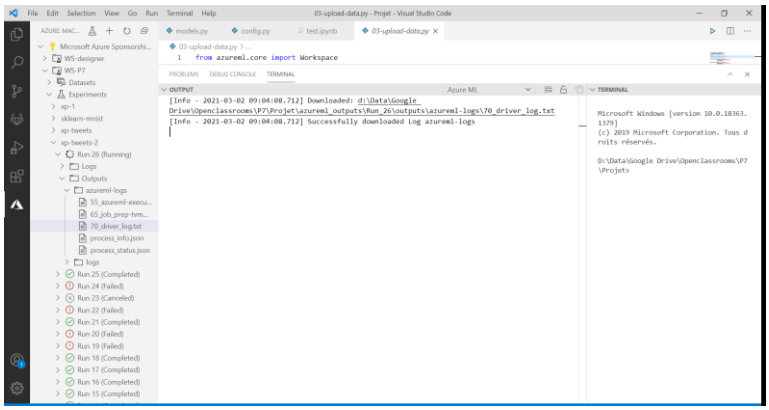
- Les poids du dropout et de la régularisation sont recherchés par validation croisée dans les plages [0.0, 0.2, 0.5] et [0.0, 0.0001, 0.01, 1.0] respectivement.
- Le learning rate est fixé à 0.01 mais on permet son ajustement dynamique via la fonction `ReduceLROnPlateau`.
- L'algorithme d'optimisation est Adam.
- À la vue du nombre d'exemples disponibles pour l'entrainement, le nombre d'epochs est fixé à 1.
- La taille des batchs est fixée à 256 pour accélérer l'apprentissage.

Difficulté particulière : Keras et GridSearchCV

Keras n'a pas de fonction de validation croisée et le wrapper qui permet d'utiliser GridsearchCV de scikit-learn empêche des fonctionnalités de Keras comme `ModelCheckpoint` de fonctionner. En deux mots, l'intégration de ces deux univers est trop complexe à notre goût et nous avons préféré réécrire une fonctionnalité similaire à GridSearchCV. L'implémentation de cette fonctionnalité est décrite dans le schéma d'architecture logicielle plus bas.

Architecture logicielle et intégration dans Azure ML

Nous avons travaillé dans l'éditeur VS Code qui offre une interface de gestion des ressources Azure ML, ce qui facilite le travail.

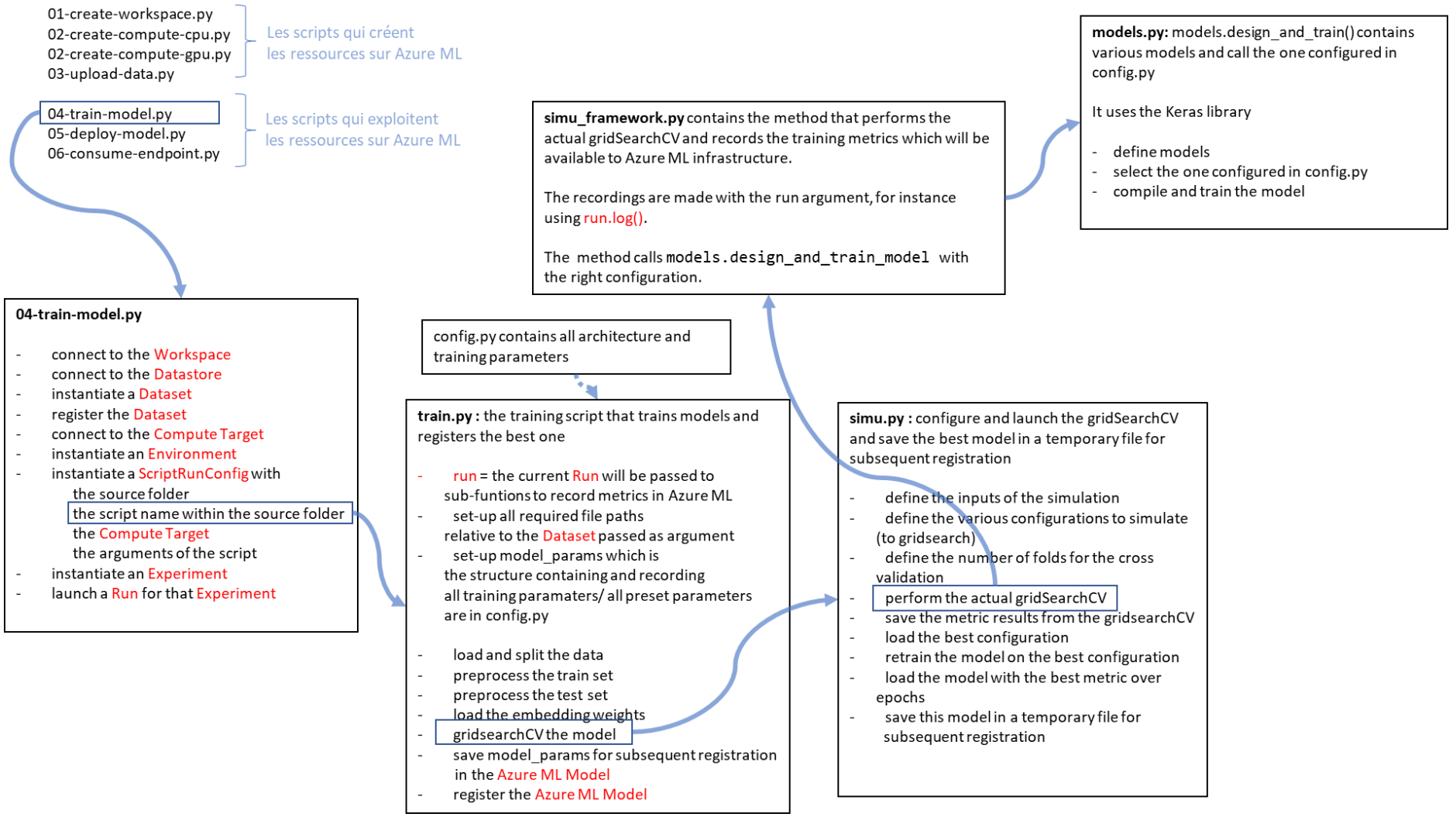


Interface Azure ML sur VS Code

Nous avons géré les différentes étapes nécessaires au processus d'intégration dans Azure ML via des scripts dans VS Code.

- 01-create-workspace: crée un workspace sur Azure ML
- 02-create-compute-cpu ou -gpu: configure une ressource de calcul cpu ou gpu sur Azure ML
- 03-upload-data : upload les jeux de données et la structure de répertoires nécessaires à l'entrainement du modèle
- 04-train-model.py : lance un run (un entrainement) sur la ressource de calcul cpu ou gpu
- 05-deploy-model : enregistre le modèle entrainé pour être consommé comme un service API
- 06-consume-endpoint : script de test de l'appel au service

L'architecture logicielle du projet est décrite dans le schéma ci-dessous. Les codes sont disponibles à [adresse des codes]. Les éléments spécifiques à l'intégration dans Azure ML sont en rouge.



Architecture logicielle

Focus sur l'entrainement

Les paramètres d'entrainement sont entièrement configurés dans le fichier **config.py**. On y définie par exemple l'architecture à entrainer ou les plages d'hyperparamètres à rechercher.

L'entrainement lui-même est déclenché par un appel au script **04-train-model.py**. Ce script indique à Azure ML quelles sont les ressources à utiliser. Il charge dans Azure ML l'environnement et les modules pythons qui décrivent les algorithmes de traitement. Enfin, il indique par quel module commencer. En l'occurrence **train.py**.

train.py peut être vu comme le pipeline de traitement à effectuer à l'entrainement. A noter : la classe run instanciée dans ce script est passée aux différentes fonctions appelées et permet d'enregistrer des résultats de mesure dans Azure ML. Ces mesures sont disponibles en cours de simulation dans l'interface web d'Azure ML pour suivre son déroulement.

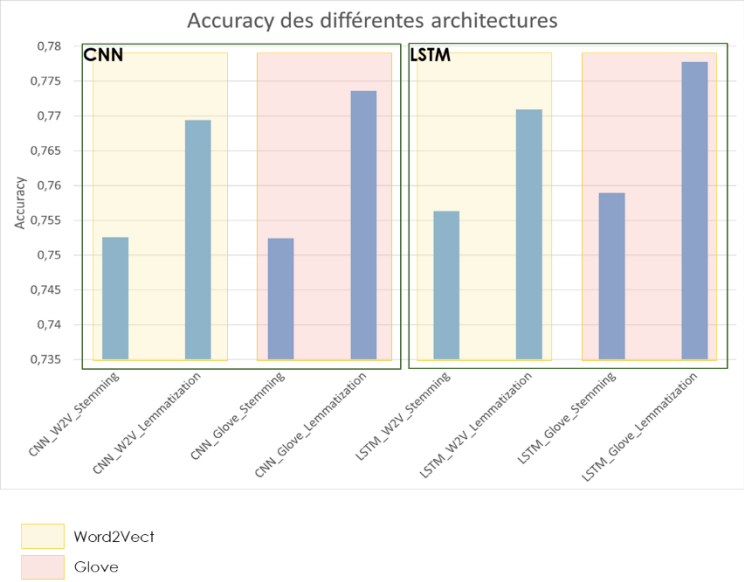
La validation croisée est gérée par deux modules. **simu.py** définie les configurations d'hyperparamètres à parcourir et **simu_framework.py** boucle dessus et, pour chacune d'elles, appelle **model.py**.

model.py configure le modèle qu'on a choisi dans **config.py** et l'entraine.

Résultats

Toutes les expériences sont d'abord réalisées sur les mêmes 80 000 exemples d'entrainement, 10 0000 de validation et 10 000 de test. Les plages de recherche des hyperparamètres sont epochs = [1], batch_size = [256], learning_rate = [0.01], dropout_rate = [0.0, 0.2, 0.5], l2_reg = [0.0, 0.0001, 0.01, 1.0]. Le nombre de 'folds' de validation croisée est 3.

Sans entrainement de l'embedding



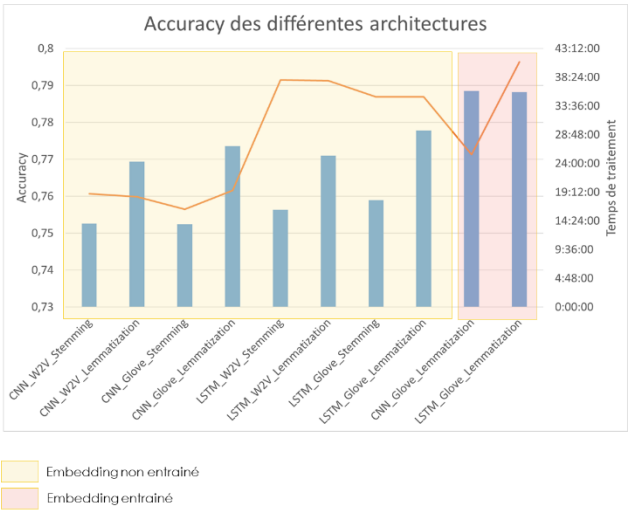
Accuracy sans entrainement de l'embedding

La lemmatisation et l'embedding Glove donnent de meilleurs accuracies que le stemming et Word2Vec.

sur CNN : 77,36%

sur LSTM : 77,77%

Avec entrainement de l'embedding



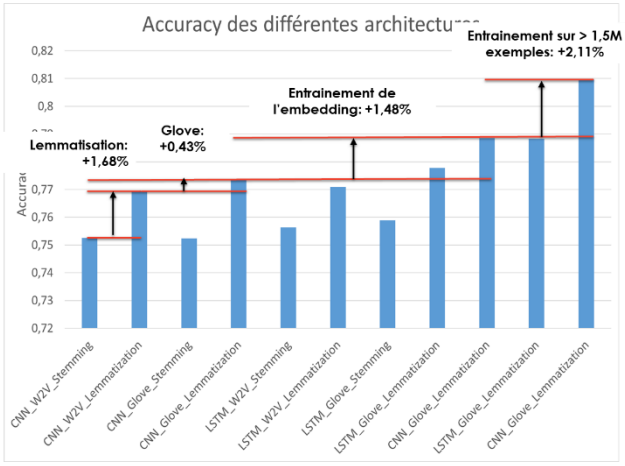
Accuracy avec entrainement de l'embedding

L'entrainement de l'embedding améliore de +1% les résultats. Les deux architectures CNN ou LSTM obtiennent des performances comparables mais LSTM est 60% plus lent.

CNN = 78.84%

LSTM = 78.82%

Entrainement sur 1.52 millions d'exemples



Résultat de l'architecture Lemmatisation + Glove + CNN entrainée sur 1.6 M d'exemples : 81% d'accuracy.

Annexe : fichier de résultats

Runs.xlsx