Table des matières

[**Membres de l’équipe projet :** 1](#_Toc85101361)

[Partie algorithmes Machine Learning pour classification en 27 classes à partir du texte des produits 3](#_Toc85101362)

[**Codes livrés sur github pour cette phase:** 3](#_Toc85101363)

[**Caractéristiques de la base de données:** 3](#_Toc85101364)

[**Vectorisation : TfidfVectorizer vs CountVectorizer** 4](#_Toc85101365)

[**Validation croisée : TfidfVectorizer** 5](#_Toc85101366)

[**VotingClassifier** 6](#_Toc85101367)

[**Vectorisation : TfidfVectorizer vs CountVectorizer** 7](#_Toc85101368)

[**VotingClassifier** 8](#_Toc85101369)

[**VotingClassifier** 11](#_Toc85101370)

[**Partie RNN GRU pour classification en 27 classes à partir du texte des produits** 16](#_Toc85101371)

[**Code livré sur github pour cette phase:** 16](#_Toc85101372)

[**Partie ConvNet pour classification en 27 classes à partir des images des produits** 24](#_Toc85101373)

[**Codes livrés sur github pour cette phase:** 24](#_Toc85101374)

[**Caractéristiques de la base de données:** 24](#_Toc85101375)

[**Détermination quelque peu arbitraire d’une taille des images qui seront stockées dans le tenseur 4D d’entrée du réseau de neurones CNN** 26](#_Toc85101376)

[**Quelques cas trés compliqués pour illustrer la difficulté de tirer de l’information différenciatrice de certaines images** 26](#_Toc85101377)

[**Atout d’un réseau à convolution sur un modéle exclusivement dense :** 28](#_Toc85101378)

[**Architecture du réseau par convolution choisi :** 29](#_Toc85101379)

[**Configuration du processus d'entraînement :** 31](#_Toc85101380)

[**Utilisation d’un générateur de données personnalisé pour alimenter l’entrainement du réseau ConvNet :** 33](#_Toc85101381)

[**Résultats pour le ConvNet sans dropout (mécanisme de régularisation pour limiter l’overfitting) et des images données au modèle de taille 150 x 150 pixels en 3 canaux de couleur.** 34](#_Toc85101382)

[**Résultats pour le ConvNet avec dropout de 50% en entrée du classifieur entiérement connecté (dense) et des images données au modèle de taille 150 x 150 pixels en 3 canaux de couleur.** 35](#_Toc85101383)

[**Résultats pour le ConvNet avec dropout de 50% en entrée du classifieur entiérement connecté (dense) et des images données au modèle de taille 80 x 80 pixels en 3 canaux de couleur et en ayant meme prolongé l’entrainement à 15 epochs plutôt que 10.** 37](#_Toc85101384)

[**Partie transfer learning ConvNet pour classification en 27 classes à partir des images des produits** 38](#_Toc85101385)

[**Codes livrés sur github pour cette phase:** 38](#_Toc85101386)

[**Etapes déjà envisagées pour les phases d’amélioration :** 43](#_Toc85101387)

# **Membres de l’équipe projet :**

Haeji YUN,

Mamadou LO,

Christophe Paquet

# Partie algorithmes Machine Learning pour classification en 27 classes à partir du texte des produits

# **Codes livrés sur github pour cette phase:**

<https://github.com/DataScientest-Studio/Rakuten_a_win_win_text_image_classification_in_PY>

<https://github.com/DataScientest-Studio/Rakuten_a_win_win_text_image_classification_in_PY/blob/main/rakuten_text_3009.ipynb>

**A. ALGORITHMES CLASSIQUES**

La prédiction avec les algorithmes classiques pour la patie texte comprend trois grandes parties :

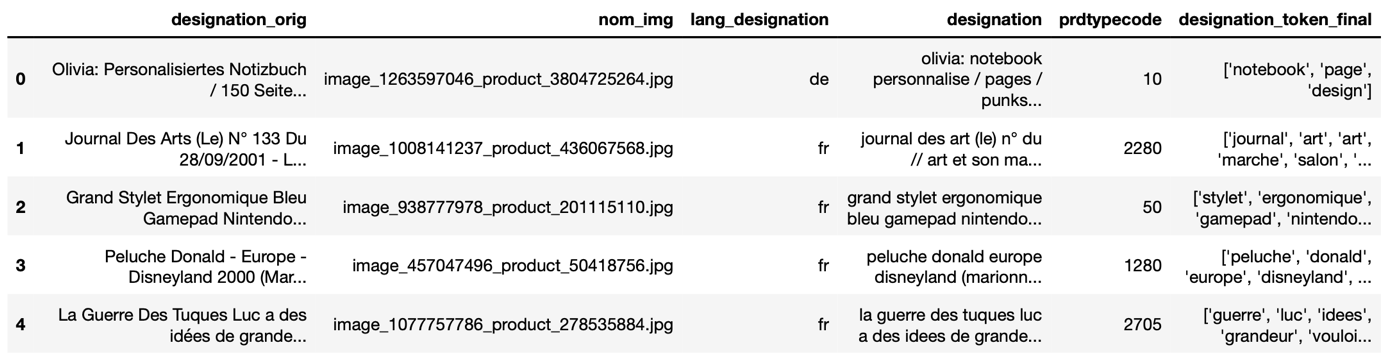
* Comparaison TfidfVectorizer et CountVectorizer
* Comparaison avec et sans la stemmatisation
* Classes mieux prédites et les classes les moins bien prédites

# **Caractéristiques de la base de données:**

Le fichier en entrée est celui issu de la phase d’exploration des données et de préprocessing du texte : X\_train\_rakuten\_afterEDA\_preprocessing.csv

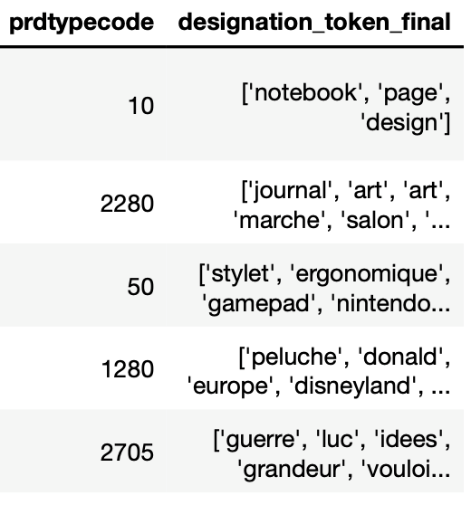
Dimension : (84613, 6)

L’aperçu de la database :



**COMPARAISON TfidfVectorizer et CountVectorizer AVEC LA STEMMATISATION**

Colonnes utilisées pour la modélisation des textes :



# **Vectorisation : TfidfVectorizer vs CountVectorizer**

Trois algorithmes classiques sont utilisés : Régression Logistique, SVM, Random Forest

Les résultats sont assez correctes et similaires pour les trois algorithmes : il y a les classes 10 et 1282 qui restent en-dessous de 0.60 de f1-score.

Les résultats TfidfVectorizer et CountVectorizer restent assez proches sauf pour l’algorithme SVM où TfidfVectorizer est plus performant.

SVM reste le plus performant parmi les trois et son temps d’exécution est particulièrement long avec à l’entour de 30 min.

|  |  |  |
| --- | --- | --- |
|  | **TfidfVectorizer** | **CountVectorizer** |
| Régression Logistique | Temps d'exécution : 4.8e+01ms | Temps d’exécution : 4.1e+01ms |
| SVM | Temps d'exécution : 1.6e+03ms | Temps d'exécution : 1.2e+03ms |
| Random Forest | Temps d'exécution : 4.3e+02ms | Temps d'exécution : 5.7e+02ms |

# **Validation croisée : TfidfVectorizer**

Pour la validation croisée, TfidfVectorizer, qui est plus performant, est gardé

Régression Logistique

* les paramètres : {'C': [0.01, 0.1, 1.0],'penalty': ['l1', 'l2']}
* les meilleurs paramètres : {'C': 1.0, 'penalty': 'l2'}

Les meilleurs paramètres sont les paramètres par défaut, l’algorithme initial est gardé

La validation croisée pour SVM et Random Forest avec les paramètres ci-dessous n’ont pas été abouti à cause de temps d’exécution trop longs (plus de 5 heures..)

SVM

* les paramètres : {'C': [0.01, 0.1, 1.0],'kernel': ['rbf', 'linear','poly'], 'gamma':[0.001,0.1,0.5]}

Random Forest

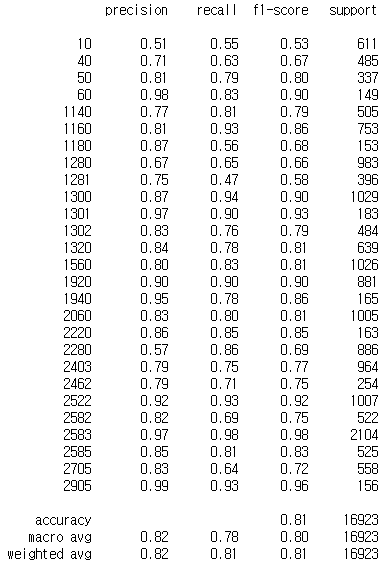
* les paramètres : {'n\_estimators':[20, 100, 200],'criterion':['gini','entropy']}

# **VotingClassifier**

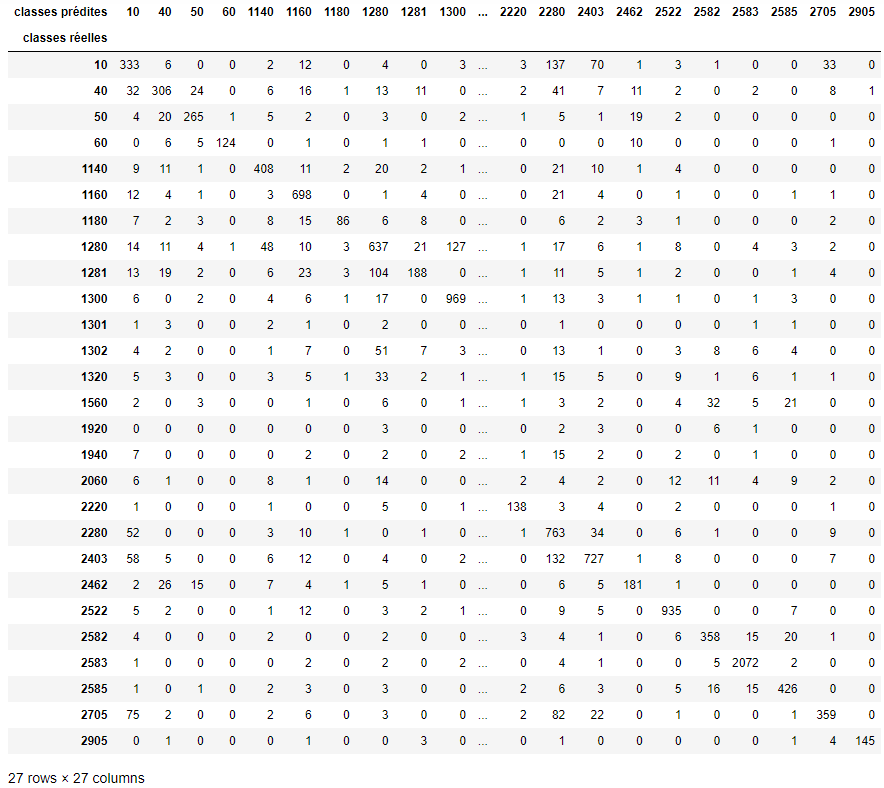
Combinaison des trois algorithmes: Régression Logistique, SVM, Random Forest

VotingClassifier est aussi performant que SVM qui était le plus performant des trois algrithmes classiques choisis, avec un accuray moyen de 0.81. Il y a 20 classes parmi les 27 qui ont f1-score supérieur à 0.70

Le temps d’exécution est légèrement plus rapide que SVM : 1.4e+03ms contre 1.6e+03ms



Sur la matrice de confusion la classification en diagonale est apprente, qui montre la bonne prédiction.

****

**COMPARAISON TfidfVectorizer et CountVectorizer SANS LA STEMMATISATION**

Colonnes utilisées pour la modélisation des textes :



# **Vectorisation : TfidfVectorizer vs CountVectorizer**

Trois algorithmes classiques sont utilisés : Régression Logistique, SVM, Random Forest

Les résultats sont assez correctes et similaires pour les trois algorithmes : il y a les classes 10 et 1281 qui restent en-dessous de 0.60 de f1-score dans tous les cas.

* Les classes 40 et 1180 y s’ajoutent pour La Régression Logistique
* Les classes 40, 1180, 2280 et 2403 y s’ajoutent pour l’algorithme SVM quand CountVectorizer est utilisés :

Les résultats TfidfVectorizer et CountVectorizer restent assez proches dans le cas de Random Forest. Pour la Régression Logisitique et SVM, la performance globale reste similaire mais les classes sont mieux prédites avec TfidfVectorizer.

SVM avec TfidfVectorizer reste le plus performant parmi les trois et son temps d’exécution est particulièrement long avec à l’entour de 30 min.

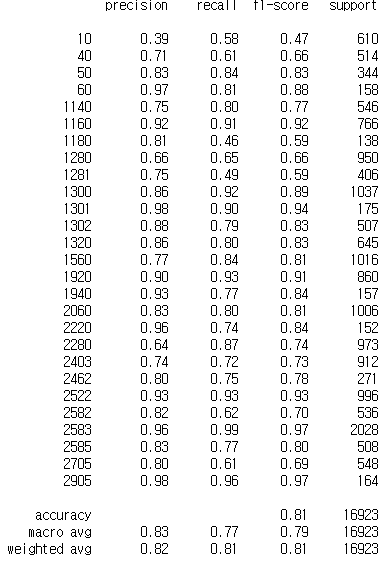
|  |  |  |
| --- | --- | --- |
|  | **TfidfVectorizer** | **CountVectorizer** |
| Régression Logistique | Temps d'exécution : 9.3e+01ms | Temps d’exécution :9.3e+02ms |
| SVM | Temps d'exécution : 1.5e+03ms | Temps d'exécution : 1.2e+03ms |
| Random Forest | Temps d'exécution : 4.2e+02ms | Temps d'exécution : 5.3e+02ms |

# **VotingClassifier**

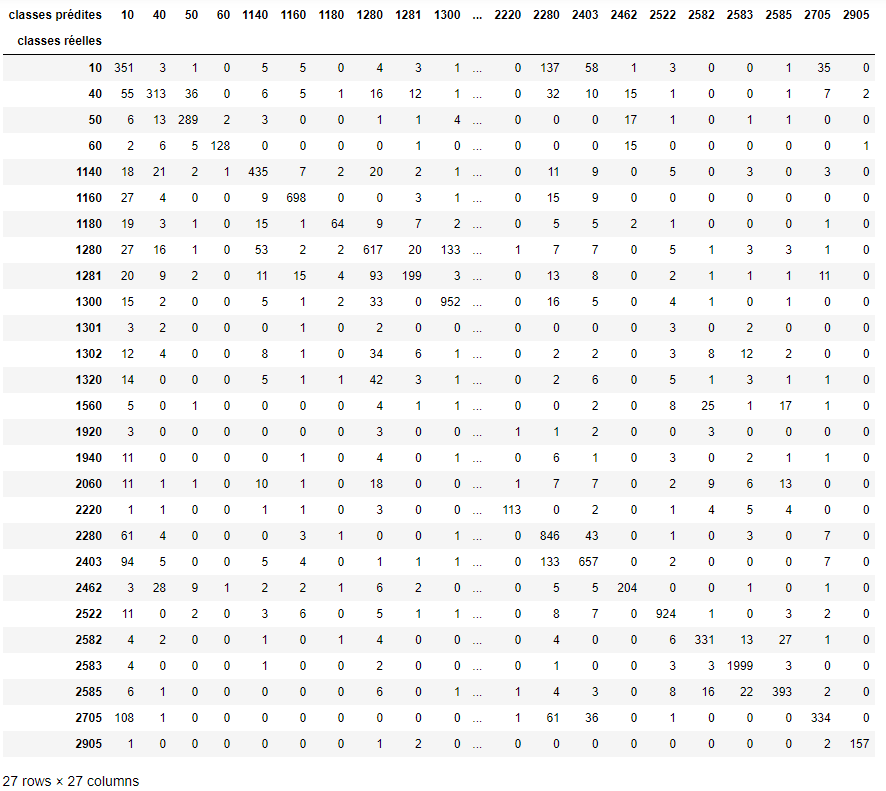
Combinaison des trois algorithmes: Régression Logistique, SVM, Random Forest

VotingClassifier est aussi performant que SVM qui était le plus performant des trois algrithmes classiques choisis, avec un accuray moyen de 0.81. Il y a 21 classes parmi les 27 qui ont f1-score supérieur à 0.70

Le temps d’exécution est légèrement plus long que SVM : 1.9e+03ms contre 1.5e+03ms



Sur la matrice de confusion la classification en diagonale est apprente, qui montre la bonne prédiction.



**COMPARAISON STEMMATISATION VS SANS STEMMATISATION**

**TfidfVectorizer**

Les cas de stemmatisation et sans stemmatisation présentent une performance de niveau similaire : 0.01 plus performant avec la stemmatisation.

Les temps d’exécution restent similiaires aussi sauf dans le cas de la régression logistique où le temps d’exécution avec la stemmatisation est deux fois plus rapide que la sans stemmatisation.

f1-score de chaque classe reste similiare aussi dans les deux cas. Néanmoins il y a deux cas de prédiction de classe qui présente une différence apparente :

* SVM : la prédiction de la classe 1180 est plus performante de 0.10. avec la stemmatisation
* Random Forest : la prédiction de la classe 1160 est plus performante sans stemmatisation de 0.15.

Pour tous les deux cas, Voting Classifier reste le plus performant.

**TfidfVectorizer**

|  |  |  |
| --- | --- | --- |
|  | **STEMMATISATION** | **SANS STEMMATISATION** |
| Régression Logistique | Temps d'exécution : 4.8e+01ms | Temps d'exécution : 9.3e+01ms |
| SVM | Temps d'exécution : 1.6e+03ms | Temps d'exécution : 1.5e+03ms |
| Random Forest | Temps d'exécution : 4.3e+02ms | Temps d'exécution : 4.2e+02ms |
| **VotingClassifier** | Temps d'exécution : 1.4e+03ms | Temps d'exécution : 1.9e+03ms |

**LES CLASSES LES MIEUX/ MOINS BIEN PREDITES**

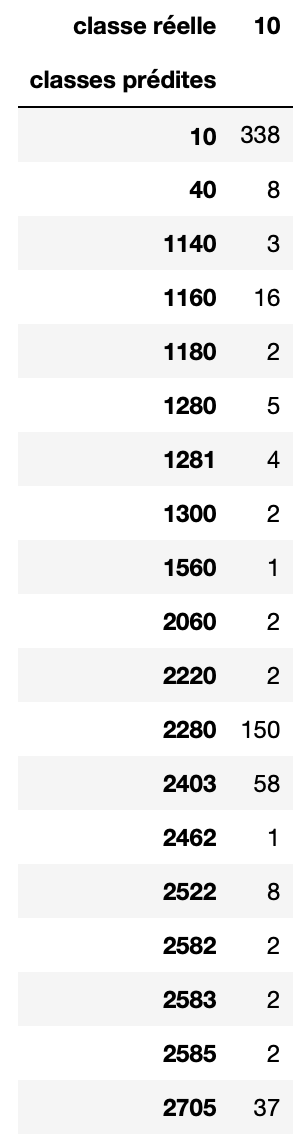
Les classes les mieux prédites et les moins bien prédites restent les même dans les différents algorithmes essayés :

* Les mieux prédites sont les classes 2583 et 2905
* Les moins bien prédites sont les classes 10 et 1281

Les prédictions de la classe 10 :

Les fausses prédictions fréquentes sont les classes 2280, 2403, 2705. On remarque la similitude des classes, toutes liées à la lecture (10 éditions , 2280 magazines, 2403 livres\_par\_lot, 2705 livres)

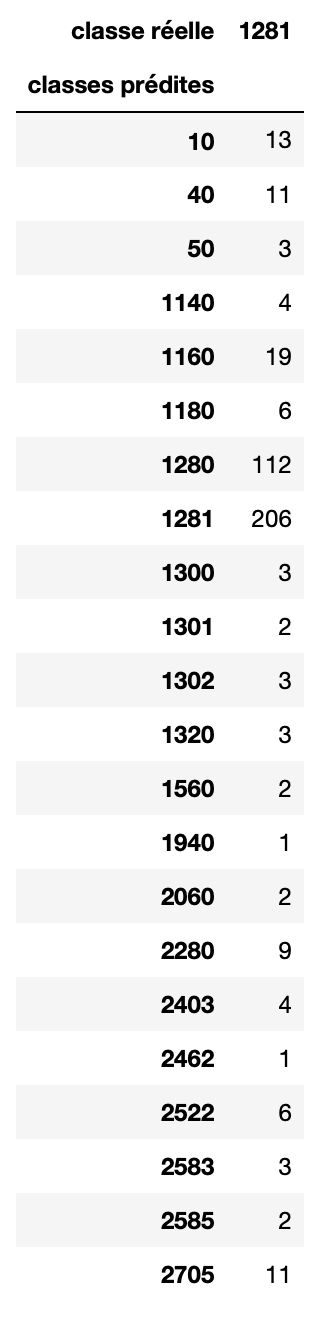
Étonnement, il n’y a aucun mot commun entre les classes de fausses prédictions et la vraie classe.



Les prédictions de la classe 1281 :

Les fausses prédictions fréquentes sont la classe 1280. On remarque la similitude des deux classes, tous les deux liées au jeu d’enfants (1281 jeux\_enfant , 1280 jouets\_enfant)

Étonnement, il n’y a aucun mot commun entre la classe de fausses prédictions et la vraie classe.

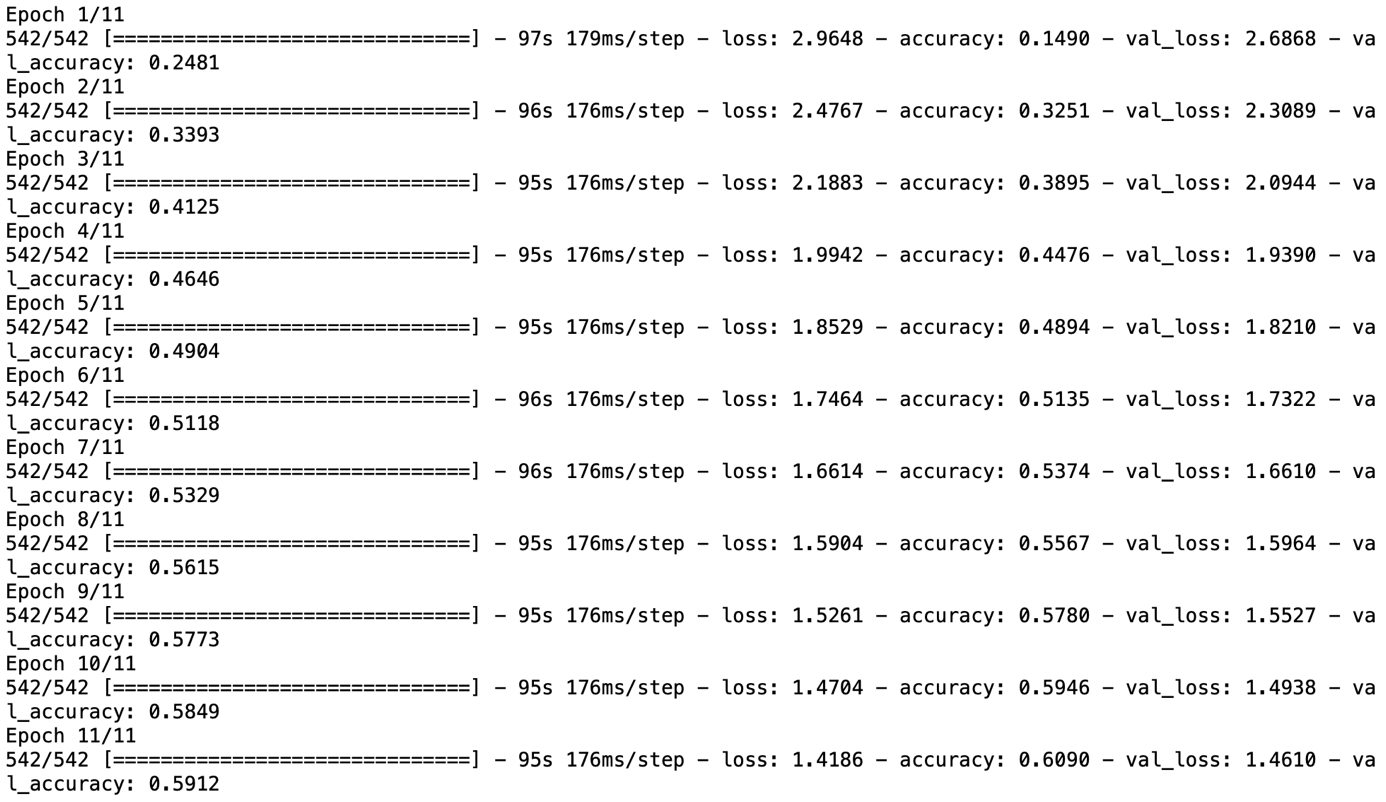


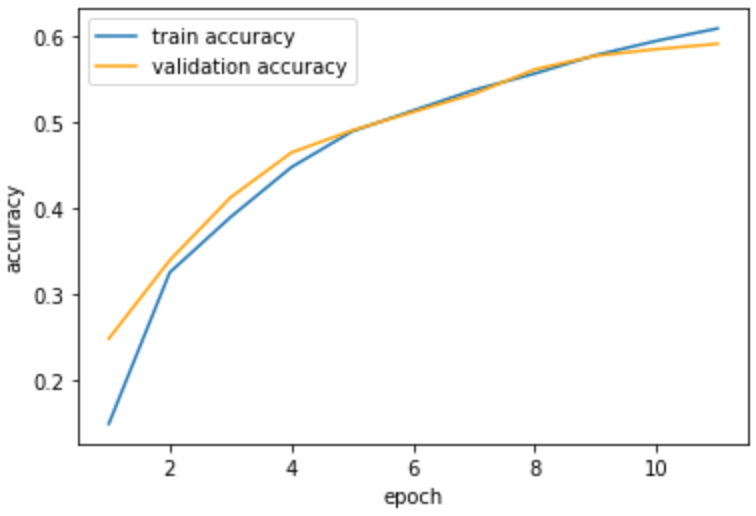
**B. WORD EMBEDDING**

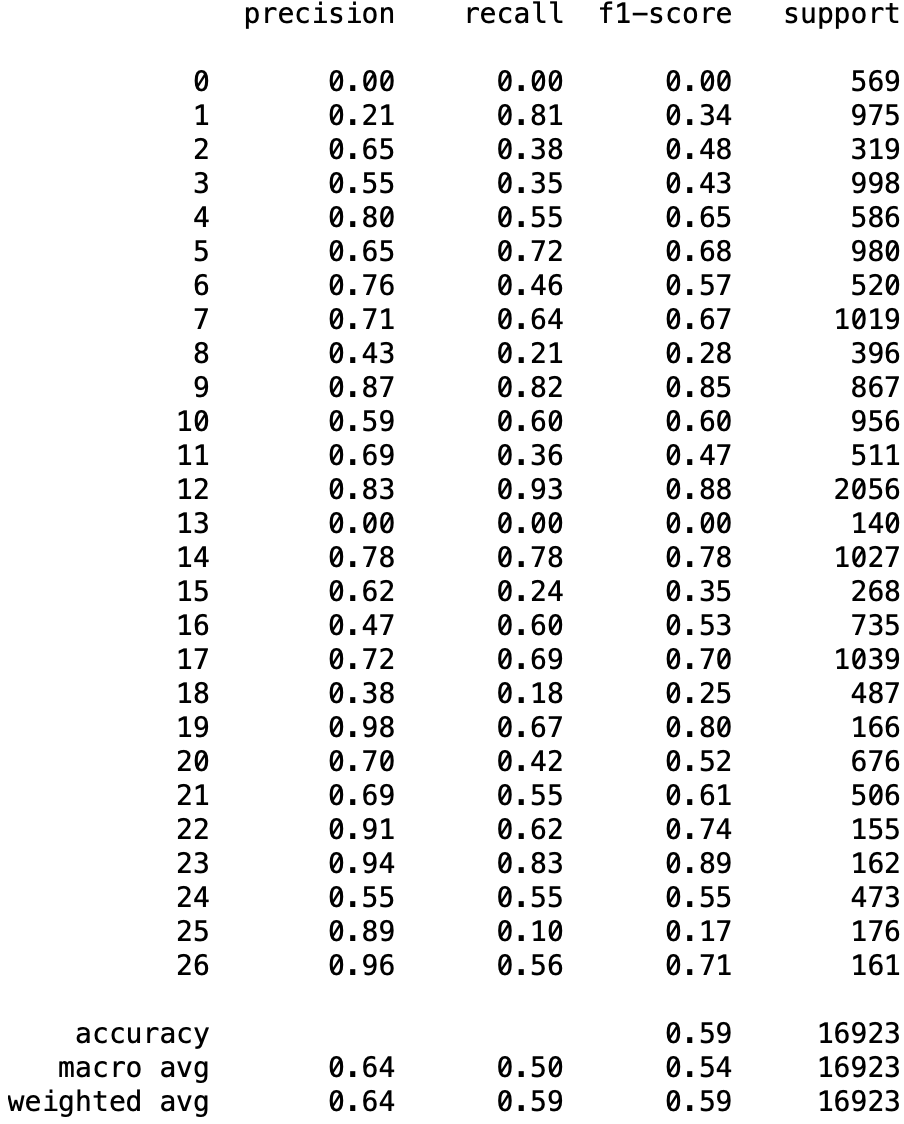
Première tentation avec Word Embedding, avec la taille de vocabulaire à 1000, 11 epochs

Vu le grand nombre de texte de notre data qui est à l’ordre de 50.000, le modèle ne prédit pas correctement avec une précision de 0.6 et il y a une grande disparité de précision entre les classes.

Il faut augmenter la taille de vocabulaire pour les prochains entraînements.

****

****

****

# **Partie RNN GRU pour classification en 27 classes à partir du texte des produits**

# **Code livré sur github pour cette phase:**

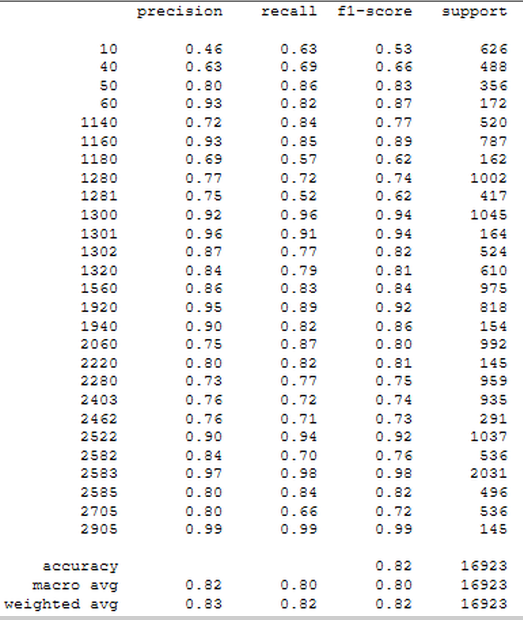
<https://github.com/DataScientest-Studio/Rakuten_a_win_win_text_image_classification_in_PY>

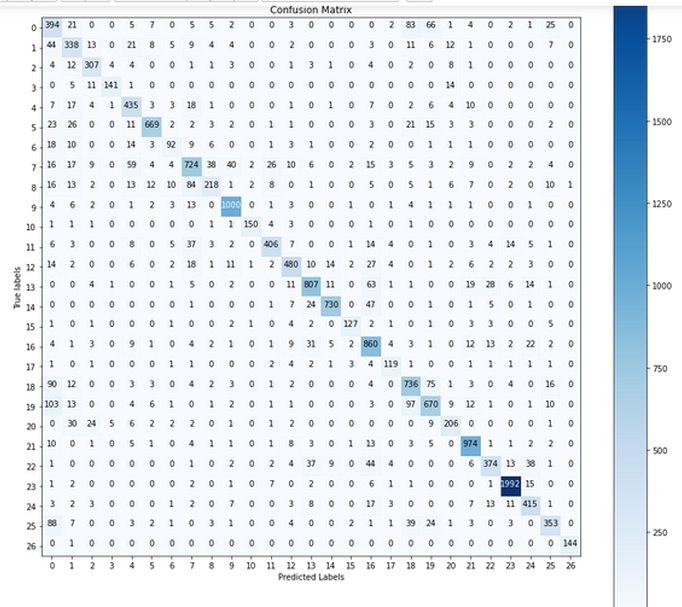
Rakuten\_project\_processing\_text\_141021.ipynb

Le Meilleur modéle trouvé par experimentation est le model5.

Le rapport de classification sklearn et la matrice de confusion ont été générés à partir du modéle restitué qui utilisait la sauvegarde

Rakuten\_text\_embedding\_200words\_640\_GRU\_640\_dense\_64\_27\_after\_3epochs.h5





# Pour la classe 0 (editions) , les prédictions erronées se retrouvent sur la classe 18 (jeux\_video) et 19 (console\_de\_jeu) , et un peu sur la classe 25 (articles\_nourriture) aussi.

# Pour la classe 1 (magazines) , les prédictions erronées se sont portées sur la classe 0 (editions)

# Pour la classe 7 (mobilier\_interieur) , erreurs sur la classe 4 (livres) , 8 (jeux\_enfants) et 9 (literie)

# Pour la classe 11 (figurine) , erreurs sur la classe 7 (mobilier\_interieur)

# Pour la classe18 (jeux\_video) , erreurs sur la classe 0 (editions) et 19 (console\_de\_jeu)

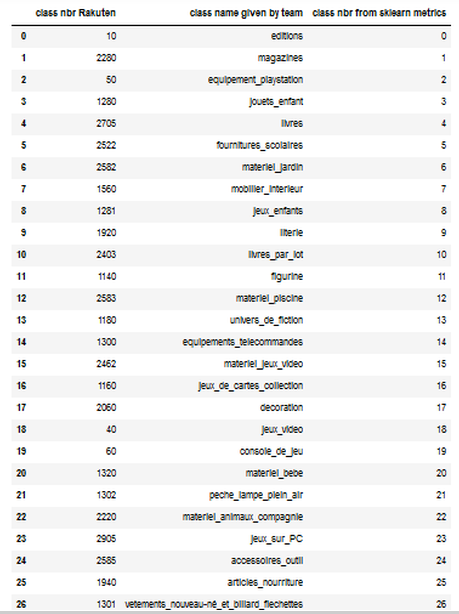
# Pour la classe 19 (console\_de\_jeu) , erreurs sur la classe 0 (editions) et 18 (jeux\_video)

# Pour la classe 13 (univers\_de\_fiction) , erreurs vers la classe 16 (jeux\_de\_cartes\_collection)

# Pour la classe 22 (materiel\_animaux\_compagnie) , erreurs vers la classe 13 (univers\_de\_fiction) , 16 (jeux\_de\_cartes\_collection) et 24 (accessoires\_outil)

# Pour la classe 25 (articles\_nourriture) , erreurs vers la classe 0 (editions)

Note : le réseau Dense impose des classes comprises entre 0 et 26 donc une table de correspondante existe avec les numéros des classes utilisés par Rakuten.et ceux don’t la probabilité est donnée en sortie



Voici toute la démarche itérative de recherche de ce Meilleur modéle :

Le préprocessing consiste à préparer le texte pour qu’il soit encapsulé dans un tenseur 2D et traité par la première couche input de chaque modéle :

**first raw text of test set :**

[happy father day canape lire accueil decoration coussin festival taie oreiller couverture happy father day canape lit decoration pillow festival coussin case specification couvertur parfait place canape cafe bibliothequ livre party club magasin materiau forme lin place amovible lavable type tai oreiller deviation mesure comprendre forfait forfait tai oreiller]

**text converted into integers list:**

[1285, 27138, 2584, 90, 978, 304, 10, 8, 811, 60, 7, 49, 1285, 27138, 2584, 90, 106, 10, 821, 811, 8, 436, 83, 1486, 54, 84, 90, 293, 941, 81, 416, 677, 616, 18, 34, 226, 84, 287, 252, 22, 211, 7, 693, 32, 102, 48, 48, 211, 7]

first tensor :

[ 1285 27138 2584 90 978 304 10 8 811 60 7 49

1285 27138 2584 90 106 10 821 811 8 436 83 1486

54 84 90 293 941 81 416 677 616 18 34 226

84 287 252 22 211 7 693 32 102 48 48 211

7 0 0 0 0 0 0 0 0 0 0 0

0 0 0 0 0 0 0 0 0 0 0 0

0 0 0 0 0 0 0 0 0 0 0 0

0 0 0 0 0 0 0 0 0 0 0 0

0 0 0 0 0 0 0 0 0 0 0 0

0 0 0 0 0 0 0 0 0 0 0 0

0 0 0 0 0 0 0 0 0 0 0 0

0 0 0 0 0 0 0 0 0 0 0 0

0 0 0 0 0 0 0 0 0 0 0 0

0 0 0 0 0 0 0 0 0 0 0 0

0 0 0 0 0 0 0 0 0 0 0 0

0 0 0 0 0 0 0 0 0 0 0 0

0 0 0 0 0 0 0 0 0 0 0 0

0 0 0 0 0 0 0 0 0 0 0 0

0 0 0 0 0 0 0 0 0 0 0 0

0 0 0 0 0 0 0 0 0 0 0 0

0 0 0 0 0 0 0 0 0 0 0 0

0 0 0 0 0 0 0 0 0 0 0 0

0 0 0 0 0 0 0 0 0 0 0 0

0 0 0 0 0 0 0 0 0 0 0 0

0 0 0 0 0 0 0 0 0 0 0 0

0 0 0 0 0 0 0 0 0 0 0 0

0 0 0 0 0 0 0 0 0 0 0 0

0 0 0 0 0 0 0 0 0 0 0 0

0 0 0 0 0 0 0 0 0 0 0 0

1. 0]

Note : aucun modéle n’a une couche embedding préentrainée donc celle s’entraine progressivement sur les textes qui lui sont présentés

**Le modèle de nom « model »** :

il est composé d’une couche embedding qui recoit un tenseur de 350 mots et qui va générer un vecteur dense de taille 512 et son dictionnaire de mots est égal au nombre de mots trouvé par l’objet de la classe Tokenizer.

Puis, il y a le classificateur composé de deux couches dense, l’une de 64 neurones et l’autre finale de 27 avec un dropout intermédiaire de 20%

Comme nous verrons dans toutes les tentatives, la performance sur la précision est autour de 79 – 80 % ce qui est déjà le cas pour ce modèle très simple.

**Le modèle de nom « model1 »** : (ce qui est testé : ajout d’une couche LSTM)

il est composé d’une couche embedding qui recoit un tenseur de 350 mots et qui va générer un vecteur dense de taille 512 et son dictionnaire de mots est égal au nombre de mots trouvé par l’objet de la classe Tokenizer.

Puis, on intégre une couche RNN LSTM avec 512 unités

Puis, il y a le classificateur composé de deux couches dense, l’une de 64 neurones et l’autre finale de 27 avec un dropout intermédiaire de 20%

Conclusion de l’essai : Pas de gain

**Le modèle de nom « model2»** : (ce qui est testé : diminution de la taille du vecteur dense de sortie de la couche embedding)

il est composé d’une couche embedding qui recoit un tenseur de 350 mots et qui va générer un vecteur dense de taille 128 et son dictionnaire de mots est égal au nombre de mots trouvé par l’objet de la classe Tokenizer.

Puis, on intégre une couche RNN LSTM avec 128 unités

Puis, il y a le classificateur composé de deux couches dense, l’une de 64 neurones et l’autre finale de 27 avec un dropout intermédiaire de 20%

Conclusion de l’essai : Pas de gain et meme perte de perf

**Le modèle de nom « model3»** : (ce qui est testé : diminution de la taille du tenseur d’entrée de la couche embedding à 200 mots)

il est composé d’une couche embedding qui recoit un tenseur de 200 mots et qui va générer un vecteur dense de taille 128 et son dictionnaire de mots est égal au nombre de mots trouvé par l’objet de la classe Tokenizer.

Puis, on intégre une couche RNN LSTM avec 128 unités

Puis, il y a le classificateur composé de deux couches dense, l’une de 64 neurones et l’autre finale de 27 avec un dropout intermédiaire de 20%

Conclusion de l’essai : retour à une perf de 79% donc je garde cette taille de 200 mots

**Le modèle de nom « model4»** : (ce qui est testé : augmentation de la taille du vecteur dense de sortie de la couche embedding)

il est composé d’une couche embedding qui recoit un tenseur de 200 mots et qui va générer un vecteur dense de taille 640 et son dictionnaire de mots est égal au nombre de mots trouvé par l’objet de la classe Tokenizer.

Puis, on intégre une couche RNN LSTM avec 640 unités

Puis, il y a le classificateur composé de deux couches dense, l’une de 64 neurones et l’autre finale de 27 avec un dropout intermédiaire de 20%

Conclusion de l’essai : On passe pour la première fois légèrement au dessus de 80% donc je garde cette taille de vecteur dense de 640

**Le modèle de nom « model5»** : (ce qui est testé : on passe d’une couche RNN LSTM à une couche RNN GRU)

il est composé d’une couche embedding qui recoit un tenseur de 200 mots et qui va générer un vecteur dense de taille 640 et son dictionnaire de mots est égal au nombre de mots trouvé par l’objet de la classe Tokenizer.

Puis, on intégre une couche RNN GRU avec 640 unités

Puis, il y a le classificateur composé de deux couches dense, l’une de 64 neurones et l’autre finale de 27 avec un dropout intermédiaire de 20%

Conclusion de l’essai : bingo, on avoisine une performance de 82% et malgré des tentatives ultérieures, ce modèle ne sera pas surpassé.

**Le modèle de nom « model6 »** : (ce qui est testé : revenir au modèle le plus simple avec les deux décisions de taille intégrées 200 mots par tenseur et un vecteur dense de 640 )

il est composé d’une couche embedding qui recoit un tenseur de 200 mots et qui va générer un vecteur dense de taille 640 et son dictionnaire de mots est égal au nombre de mots trouvé par l’objet de la classe Tokenizer.

Puis, il y a le classificateur composé de deux couches dense, l’une de 64 neurones et l’autre finale de 27 avec un dropout intermédiaire de 20%

Comme nous verrons dans toutes les tentatives, la performance sur la précision est autour de 79 – 80 % ce qui est déjà le cas pour ce modèle très simple.

Conclusion de l’essai : gain toutefois minime par rapport à model et inférieur à model5

**Le modèle de nom « model7»** : (ce qui est testé : utilisation de couches ConvNet1D)

il est composé d’une couche embedding qui recoit un tenseur de 200 mots et qui va générer un vecteur dense de taille 640 et son dictionnaire de mots est égal au nombre de mots trouvé par l’objet de la classe Tokenizer.

Puis, on intégre deux couches Conv1D avec 32 filtres générés et une fenetre de convolution de 9 et au milieu une couche MaxPooling1D qui réduit de 5 (concentre,synthétise)

Puis, il y a le classificateur composé de deux couches dense, l’une de 64 neurones et l’autre finale de 27 avec un dropout intermédiaire de 20%

Conclusion de l’essai : perf de 78,5% donc dans le bas des modéles expérimentés

**Le modèle de nom « model8»** : (ce qui est testé : augmentation à 64 du nombre de filtres générés par chacune des couches ConvNet1D)

il est composé d’une couche embedding qui recoit un tenseur de 200 mots et qui va générer un vecteur dense de taille 640 et son dictionnaire de mots est égal au nombre de mots trouvé par l’objet de la classe Tokenizer.

Puis, on intégre deux couches Conv1D avec 64 filtres générés et une fenetre de convolution de 9 et au milieu une couche MaxPooling1D qui réduit de 5 (concentre,synthétise)

Puis, il y a le classificateur composé de deux couches dense, l’une de 64 neurones et l’autre finale de 27 avec un dropout intermédiaire de 20%

Conclusion de l’essai : perf de 79 % donc dans le bas des modéles expérimentés

**Le modèle de nom « model9»** : (ce qui est testé : augmentation à 96 du nombre de filtres générés par chacune des couches ConvNet1D , passage à 7 mots de la fenetre de convolution et à 4 pour la couche MaxPooling1D)

il est composé d’une couche embedding qui recoit un tenseur de 200 mots et qui va générer un vecteur dense de taille 640 et son dictionnaire de mots est égal au nombre de mots trouvé par l’objet de la classe Tokenizer.

Puis, on intégre deux couches Conv1D avec 96 filtres générés et une fenetre de convolution de 7 et au milieu une couche MaxPooling1D qui réduit de 4 (concentre,synthétise)

Puis, il y a le classificateur composé de deux couches dense, l’une de 64 neurones et l’autre finale de 27 avec un dropout intermédiaire de 20%

Conclusion de l’essai : perf de toujours 79 % donc dans le bas des modéles expérimentés

Améliorations / optimisations diverses en fin de notebook qui devront etre utilisées de base durant l’exercice du métier de Data Scientist pour le gain temps et cout d’exécution

1. Utilisation du callbacks EarlyStopping sur le model5
2. Utilisation du callback TensorBoard sur le model5

# **Partie ConvNet pour classification en 27 classes à partir des images des produits**

# **Codes livrés sur github pour cette phase:**

<https://github.com/DataScientest-Studio/Rakuten_a_win_win_text_image_classification_in_PY>

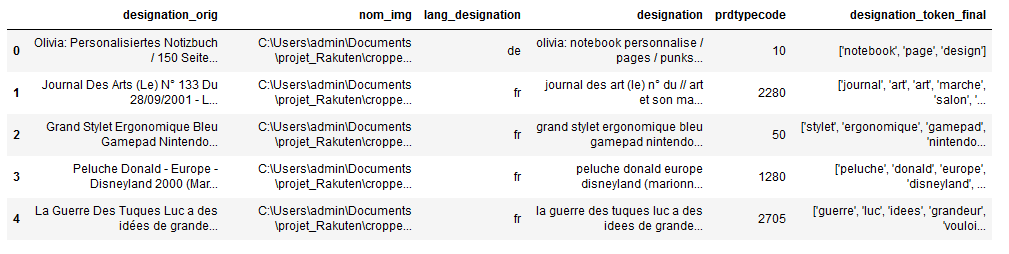
Rakuten\_project\_processing\_baseline\_images160921\_gpu.ipynb

# **Caractéristiques de la base de données:**

Le fichier en entrée est celui issu de la phase d’exploration des données et de préprocessing du texte : X\_train\_rakuten\_afterEDA\_preprocessing.csv

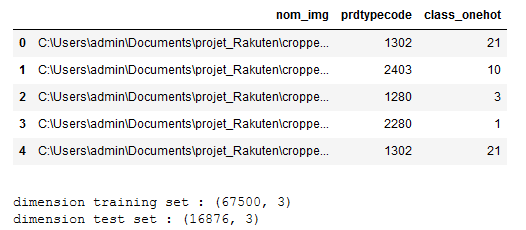
Dimension : (84613, 6)

Début des données de la database :



Note : les images utilisées ne sont plus celles originelles téléchargées depuis le challenge Rakuten mais celles retraitées pour supprimer les bords blancs et donc garder la partie utile qui contient vraiment l’information des produits à classer.

Colonnes spécifiquement utilisées pour la modélisation des images avec le réseau ConvNet :



Raison de la suppression de 237 échantillons:

Des fichiers image avec une taille inférieure à 2 kb sont considérés comme inexploitables, assimilés à du bruit et non de l’information qui pourrait etre utile au calcul des poids du réseau de neurones.

The file C:\Users\admin\Documents\projet\_Rakuten\cropped\_image\_train\\image\_953088054\_product\_216795580.jpg has a too small size of 942 bytes

The file C:\Users\admin\Documents\projet\_Rakuten\cropped\_image\_train\\image\_1142089742\_product\_884747735.jpg has a too small size of 0 bytes **(\*)**

number of deleted rows due to image size smaller than 2K : 237

new shape of dataset after preprocessing : (84376, 6)

(\*) A l’origine, il n’y avait pas de fichiers image de taille nulle sur le site du challenge Rakuten MAIS l’opération de cadrage (cropping) qui fonctionne avec un seuil codé à 192 en l’occurence (valeur sur l’échelle des gris) a eu comme léger effet de bord de produire 113 fichiers de taille nulle.

voici deux exemples d’images et on comprend la raison de la non detection des objets presents dans l’image:





# **Détermination quelque peu arbitraire d’une taille des images qui seront stockées dans le tenseur 4D d’entrée du réseau de neurones CNN**

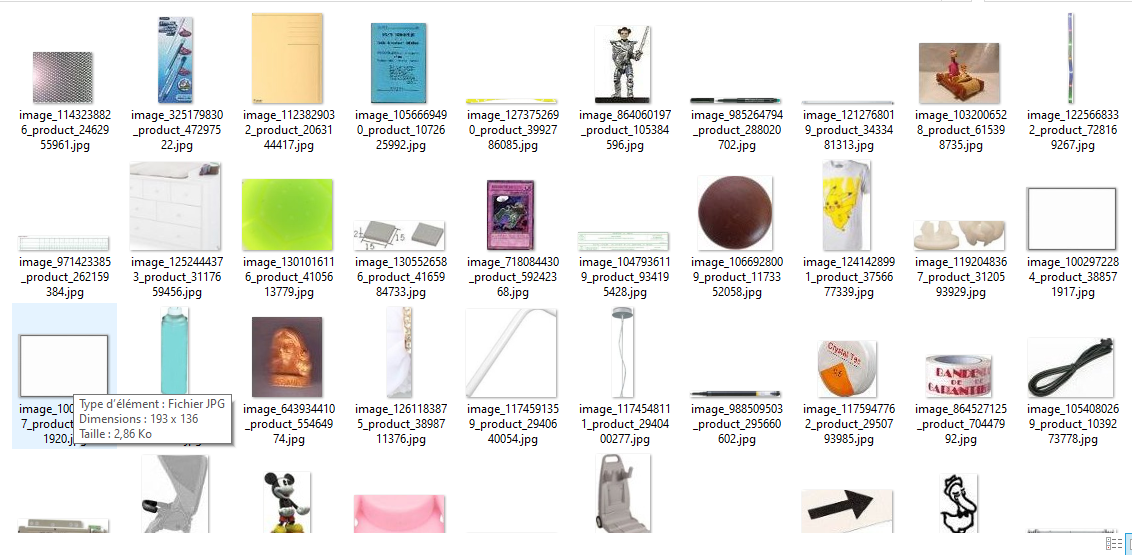
amount of images with size less than 50 kb : 45726

amount of images with size higher than 50 kb : 38650

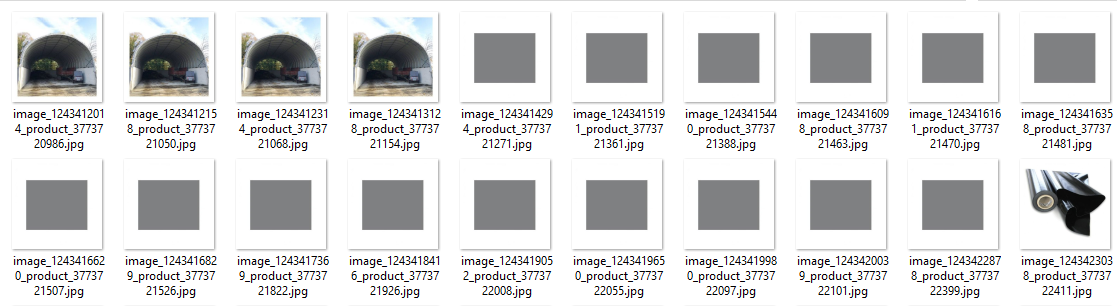
**Remember : 67.5 kb is the image size choosen as input to the ConvNet that is 150 \* 150 pixels with 3 color channels**

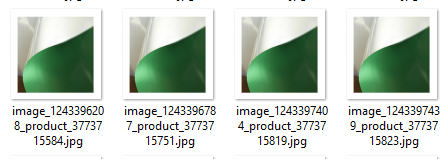
amount of original images with size quite small , less than 10 kb : 3095

# **Quelques cas trés compliqués pour illustrer la difficulté de tirer de l’information différenciatrice de certaines images**



Zoom sur les trés petites images et des motifs uniformes (deux dans la première colonne par exemple)





Quelques séries d’images pour illustrer que ,sans contexte de description textuelle, pour nous aider, nous serions incapable de déduire ce qui est représenté sur ces images et à fortiori donc de les classer

Sans approfondir, on comprend que la performance de la vision par ordinateur que l’on va implémenter avec différentes architectures de réseaux de neurones sera limitée par cette proportion d’images qui portent peu de features sur lesquelles s’appuyer pour modéliser une classification.

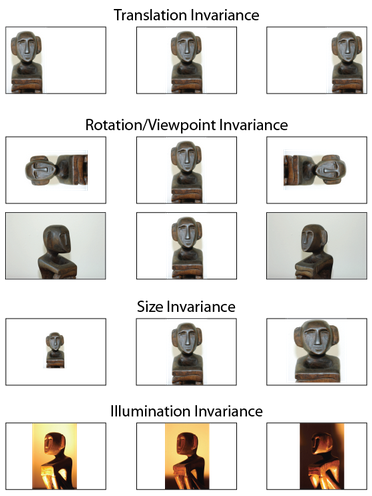
# **Atout d’un réseau à convolution sur un modéle exclusivement dense :**

Les couches Dense apprennent des motifs globaux dans l’image (par exemple des motifs d’un impliquant tous les pixels d’un chiffre du fameux jeu de données MNIST) alors que les couches de convolution apprennent des motifs locaux (ceux dans la fenetre de convolution = filtre = kernel = carte de caractéristiques = feature map)

Et un ConvNet a deux propriétés trés interessantes:

Un motif appris quelque part dans l’image sera reconnu s’il se trouve à d’autres endroits de celle ci. (acquisition de la connaissance invariante par translation)

Cette propriété donne l’avantage d’avoir besoin de moins d’exemples d’apprentissage pour apprendre des représentations qui ont une capacité de généralisation (notre but)



Images extraites de cet article =>

<https://qastack.fr/stats/208936/what-is-translation-invariance-in-computer-vision-and-convolutional-neural-netwo>

Autre avantage trés important est la capacité à apprendre des hierarchies spatiales de motifs c’est à dire que la première couche apprend comme déjà mentionné des motifs hyper locaux de type bords par exemple , la deuxième couche apprendra de plus grand motifs issus des caractéristiques de la première couche et ainsi de suite.

Cela permet aux ConvNet d’apprendre efficacement des concepts visuels de plus en plus complexes ET abstraits.(concept de plus haut niveau = un chat ici ci dessous 😊 )

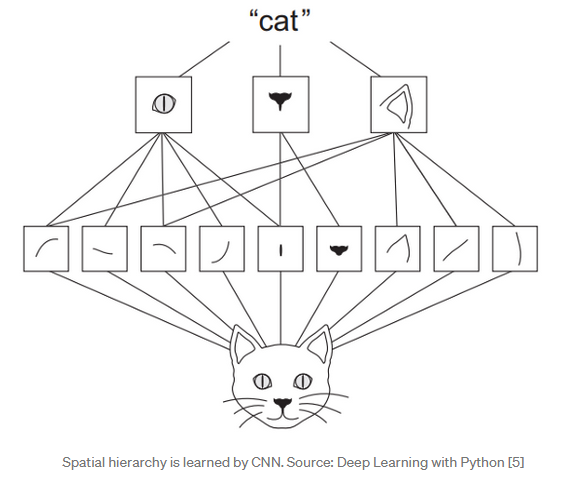


Image extraite de cet article =>

<https://medium.com/@data_datum/deep-learning-in-5-minutes-part-1-convolutional-neural-networks-13ea4e0b486f>

# **Architecture du réseau par convolution choisi :**

On va alimenter ce réseau avec des images de taille 150 sur 150 pixels et 3 canaux de couleurs.

Donc, cela veut dire en amont la présence d’une phase de génération des images à partir de celles téléchargées puis prétraitées (cropped) et de taille variée avec convertion de leur format JPEG en grilles RGB de pixels.

Il faut aussi pour que la backpropagation durant laquelle le calcul des poids est effectué par l’optimisation de la descente de gradient se passe au mieux , il faut travailler avec de petites valeurs donc on va aussi diviser par 255 les valeurs des pixels et passer en floating, type plus adapté pour le calcul :-)

Model: "sequential\_1"

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Layer (type) Output Shape Param #

=================================================================

conv2d\_1 (Conv2D) (None, 148, 148, 32) 896

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

max\_pooling2d\_1 (MaxPooling2 (None, 74, 74, 32) 0

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

conv2d\_2 (Conv2D) (None, 72, 72, 64) 18496

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

max\_pooling2d\_2 (MaxPooling2 (None, 36, 36, 64) 0

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

conv2d\_3 (Conv2D) (None, 34, 34, 128) 73856

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

max\_pooling2d\_3 (MaxPooling2 (None, 17, 17, 128) 0

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

conv2d\_4 (Conv2D) (None, 15, 15, 128) 147584

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

max\_pooling2d\_4 (MaxPooling2 (None, 7, 7, 128) 0

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

flatten\_1 (Flatten) (None, 6272) 0

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

dense\_1 (Dense) (None, 512) 3211776

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

dense\_2 (Dense) (None, 27) 13851

=================================================================

Total params: 3,466,459

Trainable params: 3,466,459

Non-trainable params: 0

Ce réseau ConvNet est constitué de 4 blocs ou modules (conv2d-ReLU , max\_pooling2d)

, d’une couche Flatten qui sert à fabriquer le vecteur d’entrée dans le réseau complétement connecté (Dense) avec une activation softmax pour la dernière couche dense de sortie (probabilité pour chacune des 27 classes et dont la somme est égale à 100%) .

Ces deux couches dense vont rendre la fonction de classification en se basant sur les features qui ont été dégagées de l’image par le reseau ConvNet.

Note : le stride est gardé à sa valeur par défaut de 1 pour les couches de convolution (distance entre deux fenetres successives d’application du patch de convolution)

Note : le padding est gardé à sa valeur par défaut de “valid” (concerne l’effet de bord du patch de convolution 3 sur 3 (kernel\_size) qui ne peut pas se centrer sur les cases de l’image du contour de celle ci) ce qui induit le -2 sur les deux dimensions

Remarque : La profondeur des cartes de caractéristiques (feature maps) augmente progressivement avec le réseau (de 32 à 128 filtres) tandis que la taille des cartes de caractéristiques diminue (de 148 x 148 à 7 x 7)

Cette logique est presque toujours utilisée dans la configuration des ConvNet.

# **Configuration du processus d'entraînement :**

model1.compile(loss='categorical\_crossentropy', # loss function multiclass

optimizer='adam', # optimisation algorithm

metrics=['accuracy']) # evaluation metrics

When fitting a neural network for classification, Keras provide the following three different types of cross entropy loss function:

* [**binary\_crossentropy**](https://keras.io/api/losses/probabilistic_losses/#binarycrossentropy-class): Used as a loss function for binary classification model. The binary\_crossentropy function computes the cross-entropy loss between true labels and predicted labels.
* [**categorical\_crossentropy**](https://keras.io/api/losses/probabilistic_losses/#categoricalcrossentropy-class): Used as a loss function for multi-class classification model where there are two or more output labels. The output label is assigned one-hot category encoding value in form of 0s and 1. The output label, if present in integer form, is converted into categorical encoding using keras.utils to\_categorical method.
* [**sparse\_categorical\_crossentropy**](https://keras.io/api/losses/probabilistic_losses/#sparsecategoricalcrossentropy-class): Used as a loss function for multi-class classification model where the output label is assigned integer value (0, 1, 2, 3…). This loss function is mathematically same as the categorical\_crossentropy.It just has a different interface.

categorical\_crossentropy ( cce ) produces a one-hot array containing the probable match for each category,



predicted probabilities of the 27 classes :

[[3.8844178e-04 3.8118571e-05 2.1597662e-04 2.4368903e-03 1.0566720e-02

2.2207780e-06 8.5158455e-05 3.4877907e-05 3.8836138e-06 2.1526153e-05

1.4997761e-04 1.0777811e-05 1.2414291e-06 2.1438068e-06 5.5146246e-04

3.0622954e-04 3.1338951e-07 1.3116463e-05 1.6205518e-04 9.8474771e-01

2.8827615e-06 3.2690837e-06 1.3000564e-06 5.5394008e-05 1.6227957e-04

1.4581930e-09 3.6069574e-05]]

Petit exercice pour pouvoir afficher de facon compréhensible la classe prédite avec la probabilité la plus élevée =>

On demande à numpy l’indice de l’élément du tableau qui a la plus grande valeur et ensuite on utilise celui ci pour prendre dans les deux listes (numéro et nom de la classe) 😊

predicted product class for this image : 60 console\_de\_jeu

real Rakuten class : 60 console\_de\_jeu

name\_class = ['editions','magazines',"equipement\_playstation","jouets\_enfant","livres","fournitures\_scolaires","materiel\_jardin","mobilier\_interieur","jeux\_enfants","literie","livres\_par\_lot","figurine","materiel\_piscine","univers\_de\_fiction","equipements\_telecommandes","materiel\_jeux\_video","jeux\_de\_cartes\_collection","decoration","jeux\_video","console\_de\_jeu","materiel\_bebe","peche\_lampe\_plein\_air","materiel\_animaux\_compagnie","jeux\_sur\_PC","accessoires\_outil","articles\_nourriture","vetements\_nouveau-né\_et\_billard\_flechettes"]

list\_class = [10, 2280 , 50 ,1280 ,2705, 2522, 2582, 1560, 1281, 1920, 2403, 1140, 2583, 1180, 1300 ,2462, 1160, 2060 , 40, 60 ,1320 ,1302 ,2220 ,2905, 2585, 1940 ,1301]

list\_class\_onehot = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26]

sparse\_categorical\_crossentropy ( scce ) produces a category index of the most likely matching category.

# **Utilisation d’un générateur de données personnalisé pour alimenter l’entrainement du réseau ConvNet :**

traingen = CustomDataGen(df\_train,

X\_col={'nom\_img':'nom\_img'},

y\_col={'class\_onehot': 'class\_onehot'},

batch\_size=32, input\_size=(150, 150, 3))

valgen = CustomDataGen(df\_test,

X\_col={'nom\_img':'nom\_img'},

y\_col={'class\_onehot': 'class\_onehot'},

batch\_size=32, input\_size=(150, 150, 3))

history = model.fit(traingen,

validation\_data=valgen,

epochs=10)

Explications :

la méthode fit gére l’incrémentation d’un index de progression qui est passé à la méthode \_\_getitem\_\_ de la classe CustomDataGen(tf.keras.utils.Sequence) qui hérite elle meme de la classe tensorflow-keras Sequence.

traingen sera en charge de générer un tenseur 4D (\*) qui contient le batch suivant de 32 images stockées sous forme de matrice 3D (150,150,3). Par contre, je n’ai pas implémenté la méthode pour effectuer un mélange (shuffle) entre 2 epochs des lignes du dataset df\_train , mécanisme qui a pour but comme d’autres de lutter contre l’overfitting.

(\*) Rappel : un tenseur est un conteneur de données.

Tenseur 0D = scalaire ou une constante

Tenseur 1D = vecteurs

Tenseur 2D = matrices (batch de textes encodés pour le modèle embedding + RNN GRU)

Tenseur 3D (une image définie par sa largeur, hauteur et de 1 à 3 canaux de couleur)

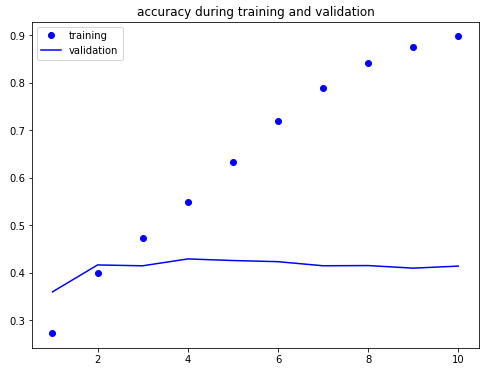
et tenseurs haute dimension qui contiennent des tenseurs de dimension inférieures

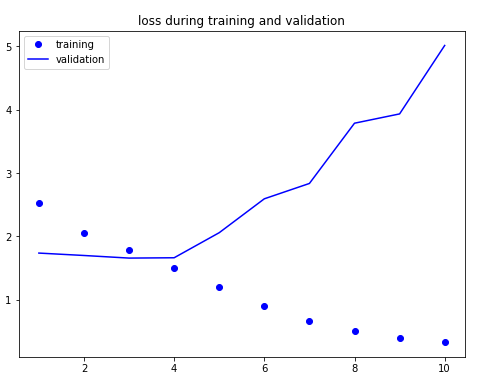
tenseur 4D = batch d’images dans notre cas de modèles ConvNet.

# **Résultats pour le ConvNet sans dropout (mécanisme de régularisation pour limiter l’overfitting) et des images données au modèle de taille 150 x 150 pixels en 3 canaux de couleur.**

ON constate l’effet typique du surentrainement des réseaux de neurons avec la précision qui monte linéairement (avec comme cible 100%) , ici 90% au bout de 10 epochs.

Par contre, coté généralisation (jeu de validation), la précision est stabilisée à 42% et le plateau est meme atteint déjà au bout de 2 epochs et la fonction loss prend la mauvaise direction (augmentation) durant l’epoch 5

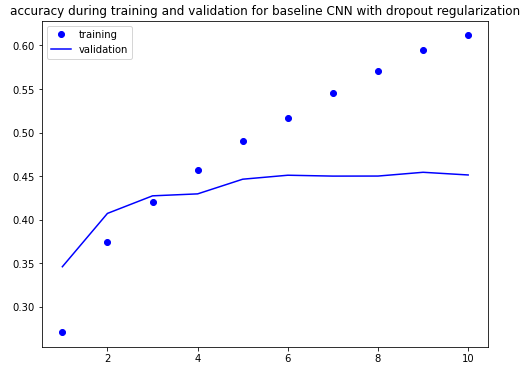


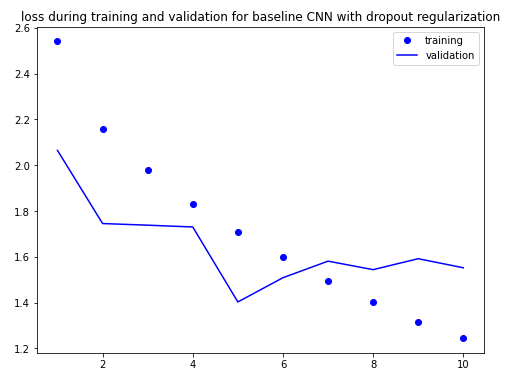


# **Résultats pour le ConvNet avec dropout de 50% en entrée du classifieur entiérement connecté (dense) et des images données au modèle de taille 150 x 150 pixels en 3 canaux de couleur.**

Toujours l’effet typique du surentrainement des réseaux de neurones mais un effet visible du dropout important de 50% des features avec la précision qui monte linéairement mais avec une pente beaucoup moins importante car on est seulement à 60% de précision au bout de 10 epochs à comparer au 90% sans la présence de cette suppression aléatoire

Par contre, coté généralisation (jeu de validation), la précision est stabilisée à 45% quand meme 3% de mieux.



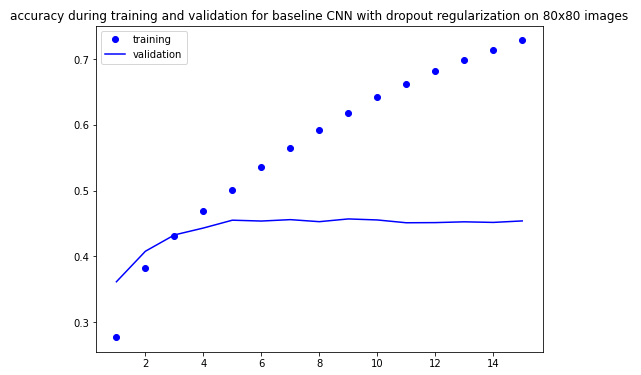


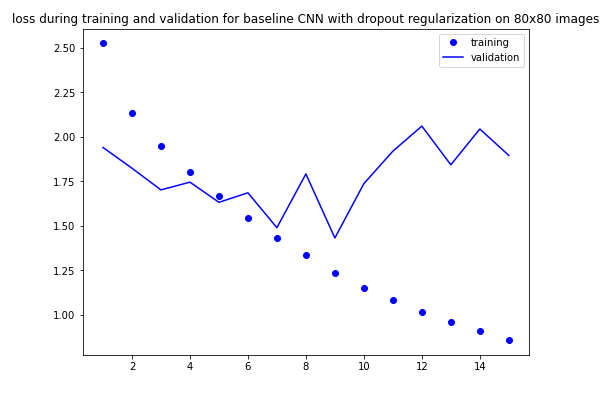
On est encore loin de courbes qui se suivent idéalement pour démontrer la capacité de ce modèle ConvNet à généraliser la classification

# **Résultats pour le ConvNet avec dropout de 50% en entrée du classifieur entiérement connecté (dense) et des images données au modèle de taille 80 x 80 pixels en 3 canaux de couleur et en ayant meme prolongé l’entrainement à 15 epochs plutôt que 10.**

L’utilisation d’images de taille 80 sur 80 pixels n’a pas amélioré la précision moyenne du modèle sur sa généralisation (courbe de validation) comparée à l’emploi d’ images converties en taille 150 sur 150.

Cela valide cette taille qui est de plus celle d’entrée du modéle préentrainé VGG16 😊





# **Partie transfer learning ConvNet pour classification en 27 classes à partir des images des produits**

# **Codes livrés sur github pour cette phase:**

<https://github.com/DataScientest-Studio/Rakuten_a_win_win_text_image_classification_in_PY>

Rakuten\_project\_processing\_images\_transfer\_learning\_141021.ipynb

Note : il y a une contrainte qui est de respecter le format d’entrée du modèle préentrainé utilisé

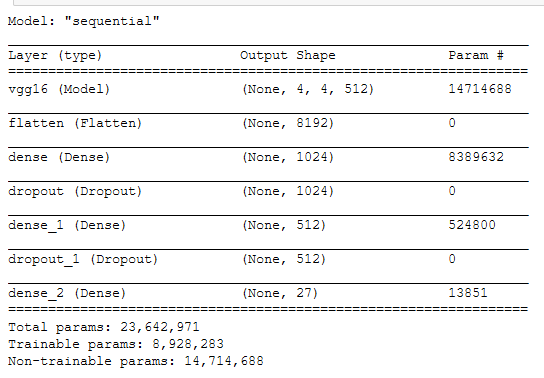
Dans le cas VGG16 , image de taille (150,150,3)

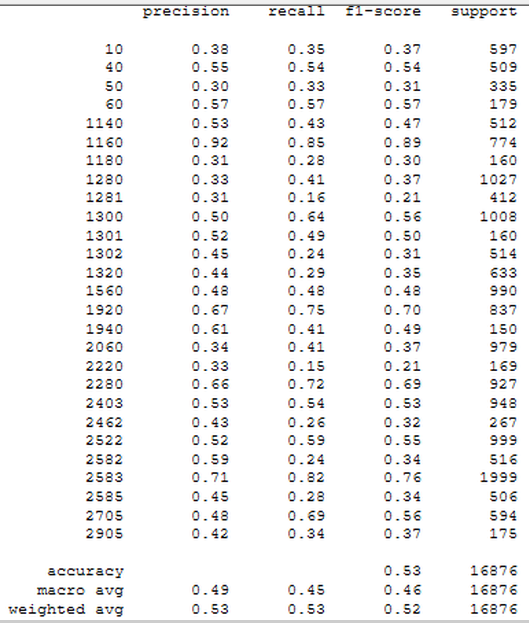
Le modèle de nom model3 va se reveler le meilleur de tous les essais itératifs menés meme si 52,5 % de performance de précision n’est pas réjouissant , juste la symbolique des 50 % grapillée (gain de 7,5 % , Merci VGG16 au passage , la technique du transfer learning utile !! ) 😊

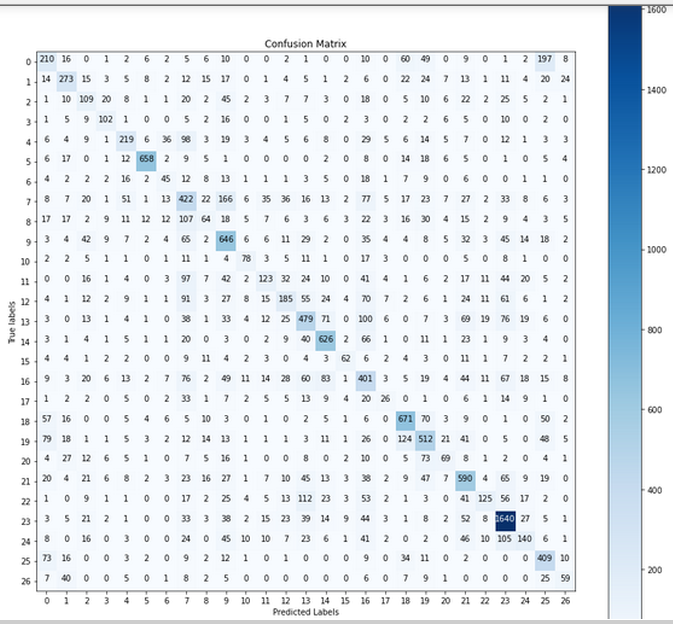
Note : Pour le modèle VGG16, on garde intégralement les poids hérités de l’entrainement de celui-ci sur la base de données de référence imagenet

**#5 modules,blocks inheritated from VGG16**

model3.summary()







# comments

# real class 0 (editions) , lot of predictions on class 25 (articles\_nourriture) , 18 (jeux\_video) and 19 (console\_de\_jeu)

# real class 4 (livres) , lot of predictions on class 7 (mobilier\_interieur)

# real class 7 (mobilier\_interieur) , lot of predictions on class 9 (literie)

# real class 8 (jeux\_enfants) , lot of predictions on class 7 (mobilier\_interieur)

# real class 11 (figurine) , lot of predictions on class 7 (mobilier\_interieur)

# real class 12 (materiel\_piscine) , lot of predictions on class 7 (jouets\_enfant) and 16 (jeux\_de\_cartes\_collection)

# real class 13 (univers\_de\_fiction) , lot of predictions on class 16 (jeux\_de\_cartes\_collection)

# real class 16 (jeux\_de\_cartes\_collection), lot of predictions on several classes 7,13,14 and 23

# real class 22 (materiel\_animaux\_compagnie) , lot of predictions on class 13 (univers\_de\_fiction)

# real class 24 (accessoires\_outil) , lot of predictions on class 23 (jeux\_sur\_PC)

En utilisant un seuil par exemple de 10% de la précision obtenue par classe, on peut afficher

ce genre d’informations utiles pour savoir les classes confondues au niveau des images analysées :

La classe 0 (10 editions) a souvent été pris pour la classe 18 (40 jeux\_video)

La classe 0 (10 editions) a souvent été pris pour la classe 19 (60 console\_de\_jeu)

La classe 0 (10 editions) a souvent été pris pour la classe 25 (1940 articles\_nourriture)

La classe 8 (1281 jeux\_enfants) a souvent été pris pour la classe 7 (1560 mobilier\_interieur)

La classe 8 (1281 jeux\_enfants) a souvent été pris pour la classe 19 (60 console\_de\_jeu)

La classe 22 (50 materiel\_animaux\_compagnie) a souvent été pris pour la classe 13 (1180 univers\_de\_fiction)

La classe 22 (50 materiel\_animaux\_compagnie) a souvent été pris pour la classe 16 (1160 jeux\_de\_cartes\_collection)

La classe 22 (50 materiel\_animaux\_compagnie) a souvent été pris pour la classe 23 (2905 jeux\_sur\_PC)

**Le modèle de nom « model4»** : (ce qui est testé : permettre la mise à jour des poids des cinq dernières couches du modèle VGG16 pour essayer de spécialiser ceux-ci sur notre jeu d’images d’entrainement)

Conclusion de l’essai : performance en berne à 45% de précision, on perd totalement le gain d’utiliser le modèle VGG16

**Le modèle de nom « model5»** : (ce qui est testé : on prend cette fois ci le modèle préentrainé Xception qui est dans le Top 5 Accuracy score de la documentation keras available applications)

Conclusion de l’essai : prometteur et pourtant plus mauvaise configuration vite à oublier ☹

**Le modèle de nom « model7»** : (ce qui est testé : le modèle préentrainé Xception a une sortie dans un format très grand ce qui induisant dans le réseau dense du model5 , 50 millions de paramètres (poids) à ajuster donc ici, on réduit le nombre de neurones du réseau dense)

Conclusion de l’essai : pire que le model 5 donc configuration out  ☹

**Le modèle de nom « model6»** : (ce qui est testé : on revient à utiliser le modèle VGG16 et on augmente à 3 couches le réseau dense qui sert à classifier)

Conclusion de l’essai : performance identique au model3 mais plus couteux et beaucoup plus long en epochs donc pas le meilleur compromis

# **Etapes déjà envisagées pour les phases d’amélioration :**

Etape 1 : Utiliser l’expérience accumulée dans le modéle préentrainé VGG16 (transfer learning) en maintenant inchangés ses poids mis au point sur des databases images significatives.(fait)

Etape 2 : Libérer le bloc 5 de convolution du réseau VGG16 afin de permettre le réglage de précision des poids des trois couches de convolution le constituant.(fait)

Etape 3 :

Analyser par échantillonnage le top 3 des classes les moins bien classées et constater la nature des classes prédites erronées.(fait)

On pourra s’aider d’une matrice de confusion également pour voir les classes « communicantes / confondues »

Remarque : Il est souvent judicieux de travailler sur les points faibles car le gain espéré est meilleur si une solution est trouvée bien sur.

Etape 4 (faisabilité d’abord à étudier car ne doit pas etre systématique) (abandonné car le jeu de données est déjà bien étoffé ET surtout la limite de performance tient en grande partie à la couverture / le recouvrement entre certaines classes utilisées par Rakuten : pas évident déjà pour nous de classer manuellement les produits ) :

Pourquoi pas essayer d’introduire dans la classe générateur de données qui est du code propriétaire une augmentation aléatoire de données ciblée sur ces classes. Par contre, déclencher ce mécanisme sur les classes qui sont des photos de texte ne parait pas judicieux / adapté donc il faudra estimer si cette contrainte restrictive ne va pas engrendrer une machine à gaz ,(code trop lourd pas rapport au gain potentiel)

Etape 5 : si le modèle réussit à dépasser la barre des 50% de précision sur l’ensemble des 27 classes alors une technique ensembliste pourra etre envisagée.(fait en algo ML)

Etape 6 : on peut afficher une prédiction d’un produit avec les 3 classes qui ont obtenues la meilleure probabilité

Etape x,y : introduire un modèle de réseau de neurones récurrent baseline pour créer une première classification en 27 classes basée sur le texte.(fait)

Etape XL : code sous le sapin de Noel : classification multimodale qui associerait un modèle CNN (partie images) et un modèle ML ou DL (partie textes) en utilisant le modèle fonctionnelle de tensorflow / keras : le Graal et peut etre au passage un petit respect de la part de notre chef de projet 😊

Etape finale : mettre en forme avec streamlit notre projet