Rapport TC1

David Biard, Samuel Berrien

I. Introduction

Ce projet se basera sur le challenge Kaggle d'Otto disponible à l'adresse suivante https://www.kaggle.com/c/otto-group-product-classification-challenge. Cette compétition consiste à reconnaître la catégories de 200,000 produits chacun représentés par 93 *features*. Il y a au total, 9 catégories de produits.

II. Méthodologie

Afin d'apréhender au mieux la tâche de classification à réaliser, nous avons élaboré une méthodologie qui pourrait être résumée en 3 étapes. La comphéhension et l'analyse des données, l'essai et l'interprétation de différents algorithmes de classification/régression ainsi que le choix et l'optimisation de modèles pertinents.

A. Compréhension et analyse des données

Les données regroupent 200,000 produits étiquetés selon neuf classes. Chaque produit est représenté par 93 *features*. Ces données ainsi récupérées sont fournies en deux ensembles séparés :

- `train.csv` (61877 produits)
- `test.csv` (144368 produits)

A savoir que le fichier `test.csv` ne contient que des exemples sans étiquette. En effet, l'idée et de soumettre nos prédictions sur le site de Kaggle afin d'obtenir un score.

Nous avons dans un premier temps analysé la distribution des classes. On peut facilement se rendre compte que la répartition des exemples au sein des classes dans les données du fichier `train.csv` n'est pas du tout uniforme. Au contraire, on peut observer une grande inégalité au niveau des effectifs comme le montre ce diagramme.

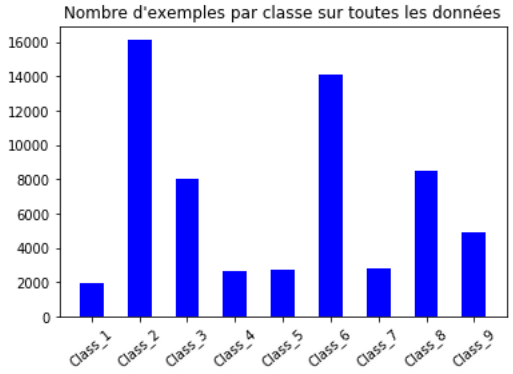

Fig. 1. Répartition des produits en fonction des classes (`train.csv`)

Cette répartition des données implique un tâche de reconnaissance bien plus difficile pour les classes sous-représentées, dû au fait que nous aurons beaucoup moins d'exemples pour entrainer notre modèle.

Egalement, nous avons remarqué que les *features* contiennent énormément de '0'. Cela peut éventuellement impliquer des difficultés de classification pour certains modèles de classification.

Enfin, pour faciliter la suite de notre travail, nous avons séparé l'ensemble `train.csv` en deux sous-ensembles. Le premier (80%) servant à entrainer nos modèles, et le second (20%) faisant office d'ensemble de validation permettant d'optimiser nos algorithmes. Les tests finaux seront directement réalisés en soumettant nos matrices de prédictions sur Kaggle avec l'autre jeu de tests.

B. Essai de différents modèles de classification

Dans un premier temps, nous avons voulu tester plusieurs algortihmes de classification sur ces données. En effet, pour ce faire nous avons utilisé le toolkit *sickit-learn* permettant d'utiliser facilement un bon nombre de *classifiers*. L'idée est donc de passer ces données à plusieurs modèle nous paraissant pertinent et intéressant afin de les comparer en regardant leurs performances sur l'ensemble de validation.

Liste des algorithmes qui ont été testé une première fois sans optimisation (*holdout method*) :

- SVM
- K-NN
- Bayésien Naïf
- Perceptron
- Regression Logistique
- SGD Classifier
- MLP : Multi-Layers-Perceptron
- Random Forest
- XGBoost

Nous avons donc obtenu des taux de précision sur l'ensemble de ces modèles d'apprentissage. Evidemment les performances relevées ont été mesurées sur l'ensemble de validation préalablement construit à l'aide du fichier `train.csv`. Ainsi, ces algortihmes n'auront pas forcément les même résultats sur les données de test de Kaggle. Cependant, nous avons pu sélectionner les méthodes nous paraissant les plus prometteuses afin de les optimiser.

C. Choix/Optimisation de certains modèles

Dans un premier temps nous avons voulu effectuer des test en gardant des modèles relativement simples. En effet, notre première approche fût de simplement optimiser les modèles les plus performants dans ceux listés précédemment. La phase d'optimisation consiste à faire varier certains hyper-paramètres du modèle. Cela à permis de gagner en performance. Voici les

résultats des *classifiers* les plus performants une fois optimisés à l'aide de la classe `GridSearchCV()` fournie par *scikit-learn* :

modèle	Acc	Log-Loss
SVM	0.71	0.72
K-NN	0.79	2.30
MLP	0.78	0.58
Random Forest	0.81	0.57
XGBoost	0.81	0.48

A savoir que dans la suite de notre étude, pour la présentation de nos prochains résultats, nous avons utilisé également une autre méthode d'optimisation. Celle-ci n'est pas relative aux hyper-paramètres du modèle mais est plutôt en rapport avec la normalisation des résultats générés par nos modèles. En effet, le système de calcul de *score* que nous fournit Kaggle est basé sur des probabilités d'appartenance à chacune des classes. Ainsi, nos modèles vont générer une matrice de résultats qui, pour chaque produit de test, va produire 9 probabilités d'appartenances, à savoir une par classe. L'idée de cette optimisation est d'apporter un étalonnage des probabilités avec une régression isotonique ou sigmoid, afin que les valeurs retournées par nos modèles s'apparentent plus à de réelles probabilités plutôt qu'à de simples valeurs entre 0 et 1. La classe permettant de réaliser cela est `CalibratedClassifierCV()` disponible avec *sickit-learn*.

III. Résultat et soumission Kaggle

Les modèles sélectionnés ci-dessus ont donc été testé sur les données de test de Kaggle :

modèle	Score
SVM	0.68
K-NN	0.60
MLP	0.55
Random Forest	0.49
XGBoost	0.48

Comme précisé précedemment, tous ces modèles ont été "calibrés" grâce à la méthode décrite dans la partie précédente. En pratique, cette méthode implique une amélioration des performances sur un bon nombre d'algorithmes d'apprentissages que nous avons pu tester.

Nous avons également eu l'idée de mélanger les prédictions probabilitstes de certains modèles, notamment les plus performants du tableau ci-dessus. L'idée du mélange est d'entrainer chaque modèle, de récupérer leurs matrices de probabilités et d'effectuer une moyenne pondérée entre elles. La pondération de chaque matrice dépend du mélange de *classifiers* réalisé.

Les résultats sont les suivants :

Modèles(Pondération)	Score
MLP(0.2) + Random Forest(0.8)	0.49
Random Forest(0.5) + XGBoost(0.5)	0.46
MLP(0.2) + Random Forest(0.4) + XGBoost(0.4)	0.48

IV. Conclusion

Tout d'abord on se rend compte que certains types de modèles fonctionnent mieux que d'autres, en effet, des algorithmes comme *Random Forest* et *XGBoost* qui utilise des arbres comme sous-modèles ont de meilleurs résultats que des algorithmes tels que des réseaux de neurones comme *MLP*. Nous pensons que les limites de *MLP* viennent du fait que les données sont creuses, ce qui ne facilite pas la tâche à des *classifiers* se basant sur énormément de produit matrices-vecteurs.

De plus, ce n'est pas en mélangeant un maximum de modèles performants que l'on obtiendra forcément de meilleurs résultats (voir table de résultats partie précédentes).

Enfin, une méthode qui aurait pu être tentée pour améliorer nos performances, aurait été d'effectuer un pré-traitement sur les données dans le but de mieux uniformiser le nombre d'exemples par classe. Nous aurions pu par exemple créer de nouveaux exemples "bruités" pour les classes faiblement représentées. À tester dans une prochaine aventure...