Rapport projet langue maternelle

Étudiants : Maxime VINCENT, Abderrahim BENMELOUKA

Objectifs du projet

Nous avons un *dataset* de documents correspondants à des rédactions de candidats au TOEFL (Test of English as a Foreign Language). Le but est de retrouver leur langue maternelle sachant que ce ne sont pas des Anglais natifs.

Pour cela, nous pouvons utiliser les techniques classiques du NLP. Mais le but de ce projet n'est pas seulement de trouver un modèle performant, il faut avant tout trouver des méthodes permettant d'améliorer le score en amont et en aval du modèle.

Métrique de score

Comme nous avons affaire à un problème de classification **multi-classe** nous utilisons la **macro-f1**. Notez que la valeur de cette métrique est très proche de l'accuracy simple dans le cas où les classes sont équilibrées ce qui est le cas avec notre *dataset*.

Nous découpons notre *dataset* en trois parties : *train*, *valid* et *test*. En prenant 600 samples par langue pour le *train*, 200 pour *valid* et 100 pour *test*. Nous n'avons utilisé le *dataset* de test qu'une seule fois dans la dernière cellule de 06-pipelines.ipynb.

Architecture du code

Tout le code source est disponible dans le dossier **src**. Vous y trouverez les *notebooks* commentés expliquant notre démarche, ainsi que les fichiers python pour encapsuler les traitements réalisés dans des fonctions et des classes.

Fichier	Rôle
01-baseline.ipynb	Entraîner et évaluer le modèle de base.
02-chars.ipynb	Analyse et nettoyage des caractères.
03-features.ipynb	Analyse et création des features.
04-cascade.ipynb	Composition de modèles pour séparer les classes proches.
05-pipelines.ipynb	Recherche de la meilleure stratégie d'apprentissage.
estimators.py	Encapsulation des méthodes à utiliser dans les pipelines.
utils.py	Fonctions utiles comme le chargement des données.
predict_data.py	Script de prédiction du *dataset* data.txt non-étiqueté.

Le *notebook* principal du projet est **pipelines.ipynb**. Son but est d'avoir une vue synthétique de toutes les stratégies réalisées pour déterminer la meilleure.

Premier modèle

Notre *baseline* utilise simplement un sac de mots comme vectorisation et un modèle de *support vector machine* linéaire comme modèle (LinearSVC). Elle atteint un score déjà honorable de 0,66 sachant la difficulté du problème.

Vectorisation des documents

Nous avons essayé trois vectorisations classiques du NLP pour transformer les documents textuels en vecteurs de flottants.

- Bag of Words
- TF-IDF
- Words Embedding

Il s'est finalement avéré que **TF-IDF** est le meilleur choix de vectorisation pour le problème. Elle a permis de gagner 12 points de macro-f1 par rapport à la *baseline*, en veillant à ne pas passer en minuscules le texte et en utilisant des unigrammes et bigrammes. Le *bag of words* ne fonctionne pas aussi bien avec notre modèle qui préfère des nombres flottants. De plus, il est probable que les *words embedding* résument trop l'information ce qui est un problème pour détecter certains signaux.

Tokenization des documents

Nous avons essayé plusieurs *tokenizers* pour découper les mots des documents.

- Découpage selon les espaces
- Tokenizer NLTK
- Tokenizer Gensim

Le meilleur *tokenizer* s'avère être celui de NLTK. Il présente l'avantage de considérer la ponctuation comme des *tokens* ce qui peut être intéressant dans notre cas. Cependant, cette étape n'a pas une grande influence sur le score.

Construction des features

On suppose qu'ajouter des *features* peut aider le modèle à obtenir un meilleur score. Le tout c'est d'en trouver qui apportent de l'information et qui ne sont pas redondantes par rapport à la vectorisation.

Nous avons implémenté les *features* suivantes :

- Nombre de mots
- Longueur moyenne des phrases en nombre de caractères et de mots
- Richesse du vocabulaire en proportion et en nombre de mots différents
- Nombre de consonnes doublées
- Nombre de contractions

Chacune de ces *features* est calculée pour chaque document. Nous les avons ensuite discrétisées sous forme de vecteurs *one-hot* pour gommer les éventuels effets de distributions non-linéaires.

Malheureusement, si l'on concatène toutes ces *features* entre elles en ajoutant la vectorisation par TF-IDF le score du modèle baisse. C'est un signe que ces *features* n'apportent pas d'information utile pour aider le modèle, et au contraire qu'elles ajoutent du bruit.

Optimisation des hyperparamètres

Nous avons essayé d'optimiser les hyperparamètres de notre modèle LinearSVC en utilisant une grille de recherche et également manuellement. Cette étape laborieuse n'a permis de glaner au plus que 1 point de score.

Nous avons également remarqué que le modèle a tendance à sur-apprendre sur l'ensemble d'apprentissage mais que le score sur l'ensemble de validation ne semble pas tant impacté que cela. Ce comportement est assez étrange.

Sélection du modèle

Nous avons essayé différents modèles de classification et il s'est avéré que le **LinearSVC** en plus d'être rapide généralise le mieux. Toutefois, nous ne sommes pas parvenus à dépasser le plafond de **0,80** de score.

Cascade de modèles

Nous avons travaillé jusque là en amont du modèle, puis sur le choix du modèle. Nous travaillons à présent en aval du modèle. Pour cela, nous avons observé la matrice de confusion obtenue à l'issue de l'évaluation sur l'ensemble de validation.

Certaines paires de classes sont très proches et le modèle a tendance à se tromper entre les deux. C'est par exemple le cas de (HIN, TEL), (KOR, JPN) ou (FRE, SPA). Nous avons donc construit et entraîné des sous-modèles en ayant au préalable créé des **méta-étiquettes** pour ces paires de classes. L'idée est d'utiliser des **modèles spécialisés** pour différencier les classes proches.

Malheureusement, en pratique les modèles en cascade n'ont pas amélioré le score, ils l'ont même plombé ! Il est probable que le fait d'avoir créé des méta-étiquettes pour regrouper les classes entre elles, a joué en la défaveur du modèle. Le résultat est assez surprenant.

Pipeline finale

Notre pipeline finale a obtenu **0,80** de score sur l'ensemble de test. Nous n'avons pas utilisé les *features*, ni les modèles en cascade. Nous avons seulement utilisé la vectorisation par **TF-IDF**. Le modèle final est donc très simple et c'est en cela que la *Machine Learning* est particulièrement frustrant. Nous avons passé notre temps à implémenter des choses compliquées qui n'ont finalement rien apporté…