Reconnaissance de la gestuelle dans le langage des signes

Bec Alicia, Carletti Circe, Lam Nelly,
Marandat Lucas, Paoli Antonin, Trupin Charlotte

Avril 2021

https://github.com/nelly-lam/Projet_Reconnaissance_Langue_Des_Signes

Table des matières

1 Introduction

Notre dataset provient de la plateforme "kaggle" (`https://www.kaggle.com/datamunge/sign-language-mnist`). Il a été téléchargé près de 40000 fois. On voit donc qu'il est très utilisé. Pourquoi avons-nous choisi ce dataset en particulier ? L'ensemble du groupe voulait que notre projet ait une vraie utilité dans la vie de tous les jours et qu'il soit intemporel. Le choix s'est donc orienté vers le décodage du langage des signes.
Imaginez une conférence, un meeting, une rencontre où l'intervenant doit utiliser le langage des signes pour communiquer. Alors il suffirait qu'une simple caméra détecte les mouvements de l'intervenant pour que les spectateurs lisent la traduction faite par le programme.
Bien évidemment ce projet n'est qu'une projection de ce qui pourrait être possible. Et dans notre cas, nous ne ferons que la première étape en entraînant un modèle sur ce jeu de données qui représente des images de gestuelle.

2 Preprocessing

Notre dataset se compose de deux fichiers : `sign_mnist_test.csv` et `sign_mnist_train.csv`. Dans les deux fichiers, la première ligne décrit la signification des colonnes, les suivantes décrivent les images (plus de 27000).
La première colonne est dédiée aux labels numérotés de 0 à 24, chacun représentant une lettre de l'alphabet (attention le z et le j ne sont pas représentés), les suivantes sont dédiées aux pixels qui composent les images, allant de 1 à 784). Pour le moment, nous ne travaillons que sur `sign_mnist_train.csv`.

FIGURE 1 – Representation de certaines images de notre fichier

Nous avons décidé de séparer en deux nos données en un ensemble de données d'entraînement de modèle et un ensemble de données dédiée à la validation, le ratio étant 70/30. Nous avons remarqué des doublons d'images que nous avons donc supprimés de nos données sur l'ensemble de validation et l'ensemble de test.

2.1 PCA

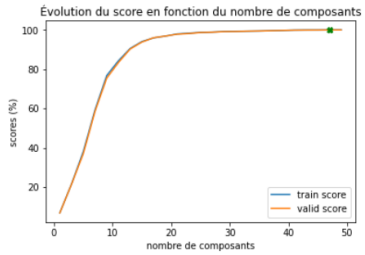
L'analyse en composantes principales, appelée aussi PCA, réduit le nombre de dimensions d'un jeu de données décrit par un grand nombre de variables, ce qui réduit ainsi le temps de calcul.
Afin de déterminer le nombre idéal de composants, nous avons appliqué une PCA en faisant varier ce nombre afin d'optimiser nos données pour l'utilisation du modèle SVC des supports vector machine. Nous avons appliqué le modèle svm (que nous avons choisi car il est plus adapté à notre ensemble de données et plus efficace sur un ensemble de données qui a beaucoup de dimensions) sur la pca, que nous avons fait varier pour plusieurs composants sur l'ensemble de validation et de test. Nous avons aussi calculé la différence en pourcentage qui nous permet de savoir si on a un over/underfitting.

	score d'entrainement	score de validation	différence
1 composants	6.97%	6.88%	0.09%
3 composants	21.65%	21.53%	0.12%
5 composants	38.06%	37.01%	1.05%
7 composants	59.27%	58.52%	0.75%
9 composants	76.74%	75.42%	1.32%
11 composants	84.27%	83.39%	0.88%
13 composants	90.53%	90.25%	0.28%
15 composants	94.03%	93.78%	0.25%
17 composants	95.92%	95.99%	-0.07%
19 composants	96.82%	96.84%	-0.02%
21 composants	97.94%	97.81%	0.12%
23 composants	98.30%	98.14%	0.16%
25 composants	98.68%	98.63%	0.05%
27 composants	98.85%	98.81%	0.04%
29 composants	99.13%	99.09%	0.04%
31 composants	99.21%	99.11%	0.10%
33 composants	99.32%	99.32%	-0.00%
35 composants	99.43%	99.45%	-0.02%
37 composants	99.57%	99.56%	0.01%
39 composants	99.70%	99.71%	-0.01%
41 composants	99.89%	99.83%	0.06%
43 composants	99.95%	99.92%	0.03%
45 composants	99.92%	99.90%	0.02%
47 composants	99.98%	99.95%	0.03%
49 composants	99.98%	99.95%	0.03%

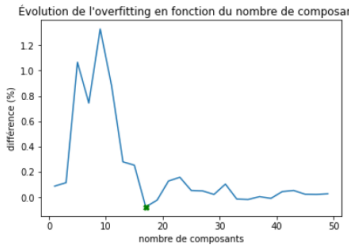
FIGURE 2 – Tableau des scores

2.2 Le meilleur nombre de composant

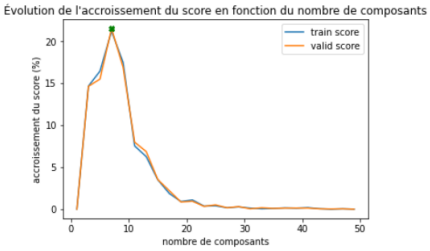
Nous avons relevé plusieurs critères pour choisir au mieux le composant le plus optimal.



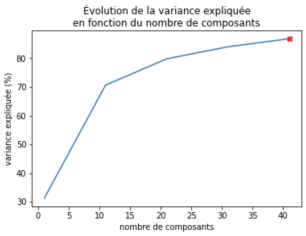
Critère du plus haut score : Un score élevé signifie que le modèle prédit bien les résultats. La courbe représentant les scores obtenus avec l'entraînement des données d'apprentissage `X_train` (train score) croît constamment. Ici le score maximal est obtenu avec 47 composants.



Critère de l'overfitting le plus bas : Un modèle sur-entraîné sur les données d'apprentissage se généralise mal car il prend trop en compte les caractéristiques des données d'apprentissage quand il s'entraîne sur des nouvelles. Ici le nombre de composants qui minimise l'overfitting est 17.



Critère de l'accroissement du score : L'idée est de choisir un nombre de composants faible pour accélérer les calculs des modèles, tout en garantissant un score élevé. Ici le nombre de composants qui permet d'obtenir rapidement un score élevé est 7.



Critère de la variance expliquée des données : L'idée est de choisir le nombre de composants qui maximise la variance expliquée des données. La variance croît constamment, en effet au plus nous conservons de composants au plus la variance expliquée se rapproche de la variance des données initiales. Ici le nombre de composants qui maximise la variance expliquée est 41.

Pour choisir le nombre de composants optimal, une classe CompRanker comprenant deux fonctions a été implémentée. Le but de cette classe va être de mettre à jour, dans un tableau, la performance de chaque composant sous la forme d'un score. Pour cela nous utilisons deux fonctions : `update_rank()` qui met à jour les scores pour chaque nombre de composants, dans le tableau des scores `n_comps_score` en incluant les résultats de l'overfitting, de la variance et de l'accroissement des scores, et `get_best_n_comps()` qui trie dans l'ordre décroissant les valeurs du tableau des scores et forme des tuples (nombre de composants $\times$ scores).

```
class CompRanker:
def __init__(self, n):
    self.n_comps_score = [0 for i in range(n)]

def update_rank(self, data, reverse=False):
    for i, x in enumerate(data):
        self.n_comps_score[i] += -x if reverse else x

def get_best_n_comps(self, nb_comp_range):
    scores = sorted(enumerate(self.n_comps_score),key=lambda x: x[1],reverse=True)
    formated_ranks = [(nb_comp_range[i], score) for i, score in scores]
    return tuple(zip(*formated_ranks))
```

Afin que notre code soit le plus efficace possible, nous avons décidé d'appliquer la pca et de calculer les meilleurs scores via les trois critères cités ci-dessus sur une liste de nombres de composants, liste étant choisie et modifiée en trois temps.

Dans la première sélection, nous choisissons des nombres de composants variant de 1 à 41 avec un pas de 10, afin d'avoir une idée générale de la zone où le nombre de composants permet le meilleur score. Après les exécutions, nous obtenons le meilleur nombre de composants corrélé avec le meilleur score obtenu.

Dans la deuxième sélection, nous nous appuyons sur le meilleur nombre de composants renvoyé par la première sélection pour le choix des nouveaux nombres de composants : nous diminuons la taille de la liste en faisant un encadrement du meilleur nombre avec un pas de 3.

Dans la troisième sélection, nous suivons le même procédé que la deuxième en changeant le pas à 1.
En réduisant au fur et à mesure le choix des nombres de composants, nous affinons les meilleurs scores obtenus. Ainsi le verdict est tombé, en récupérant le score le plus élevé le nombre de composant optimal est 19.

3 Modèle

3.1 Comparaison des modèles

Après avoir preprocessé les données, nous avons cherché quel modèle était le plus adapté. Pour cela nous avons décidé d'en tester quatre différents sur l'ensemble d'entraînement.
Nous avons tout d'abord créé la fonction `comparison_of_models` pour comparer les différents modèles.

```
def comparison_of_models(X_train, y_train, X_valid, y_valid, model_dict):
    tab_result = pd.DataFrame(columns=["CVScore",
                                       "(+/-)",
                                       "Score_training",
                                       "Score_validation"])
    for model_name, model in model_dict.items():
        model.fit(X_train, y_train)
        CVScore = cross_val_score(model, X_train, y_train, cv=5)
        train_score = model.score(X_train, y_train)
        valid_score = model.score(X_valid, y_valid)
        tab_result.loc[model_name] = np.array([CVScore.mean(),
                                               CVScore.std(),
                                               train_score,
                                               valid_score])
    return tab_result
```

	CVScore	(+/-)	Score_training	Score_validation
Svc	0.94833	0.002929	0.967843	0.964789
NuSvc	0.81304	0.006155	0.829743	0.817023
DecisionTree	0.92965	0.004114	1.000000	0.945847
LinearSvc	0.00000	0.000000	0.420127	0.423142

FIGURE 3 – Score avec PCA

	CVScore	(+/-)	Score_training	Score_validation
Svc	0.999376	0.000265	0.999948	1.000000
NuSvc	0.923041	0.002370	0.934905	0.930670
DecisionTree	0.845406	0.005408	1.000000	0.865226
LinearSvc	0.000000	0.000000	0.278073	0.267606

FIGURE 4 – Score sans PCA

En fonction du nombre de composants (section preprocessing). La PCA compresse les données, nous permettant de réduire le bruit de chaque image de notre data set. En réduisant drastiquement le nombre de dimensions (de 784 à 19) et donc la complexité de notre modèle, nous avons conservé un très bon score de validation similaire à celui obtenu sans PCA. La PCA nous a donc bel et bien permis de réduire à l'essentiel les informations de notre data set.

Après comparaison, nous pouvons déduire que le modèle SVC nous donne les meilleurs résultats. Nous allons donc maintenant chercher à optimiser les hyper-paramètres de ce modèle.

3.2 Choix des hyper-paramètres

Pour déterminer les hyper-paramètres de notre modèle, nous avons utilisé la fonction GridSearchCV qui permet de déterminer la meilleure combinaison des hyper-paramètres possible, données en arguments.

Voici une description des différents hyper-paramètres que nous avons donné en arguments :
c : contrôle le compromis entre une frontière de décision lisse et la classification correcte des données i.e. la distance entre la ligne de décision/séparation et les données mal prédites.
kernel : linéaire (séparation des données sur la même dimension que l'ensemble de données), rbf (comparaison des données sur une dimension infinie), polynomiale (séparation des données sur une dimension étendue - définie par l'hyperparamètre "degree" - du set de données).
degree (hyper-paramètre pour un noyau polynomial) : degrés de notre polynôme.
coef0 (hyper-paramètre pour un noyau polynomial) : coefficient de notre polynôme.

```
Les différentes configurations possibles dans la recherche des hyper-paramètres
configuration 0: {'C': 1, 'coef0': 0.0, 'degree': 2}
configuration 1: {'C': 1, 'coef0': 0.0, 'degree': 3}
configuration 2: {'C': 1, 'coef0': 0.0, 'degree': 4}
configuration 3: {'C': 1, 'coef0': 1.0, 'degree': 2}
configuration 4: {'C': 1, 'coef0': 1.0, 'degree': 3}
configuration 5: {'C': 1, 'coef0': 1.0, 'degree': 4}
configuration 6: {'C': 1, 'coef0': 2.0, 'degree': 2}
configuration 7: {'C': 1, 'coef0': 2.0, 'degree': 3}
configuration 8: {'C': 1, 'coef0': 2.0, 'degree': 4}
configuration 9: {'C': 3, 'coef0': 0.0, 'degree': 2}
configuration 10: {'C': 3, 'coef0': 0.0, 'degree': 3}
configuration 11: {'C': 3, 'coef0': 0.0, 'degree': 4}
configuration 12: {'C': 3, 'coef0': 1.0, 'degree': 2}
configuration 13: {'C': 3, 'coef0': 1.0, 'degree': 3}
configuration 14: {'C': 3, 'coef0': 1.0, 'degree': 4}
configuration 15: {'C': 3, 'coef0': 2.0, 'degree': 2}
configuration 16: {'C': 3, 'coef0': 2.0, 'degree': 3}
configuration 17: {'C': 3, 'coef0': 2.0, 'degree': 4}
```

FIGURE 5 – Configurations

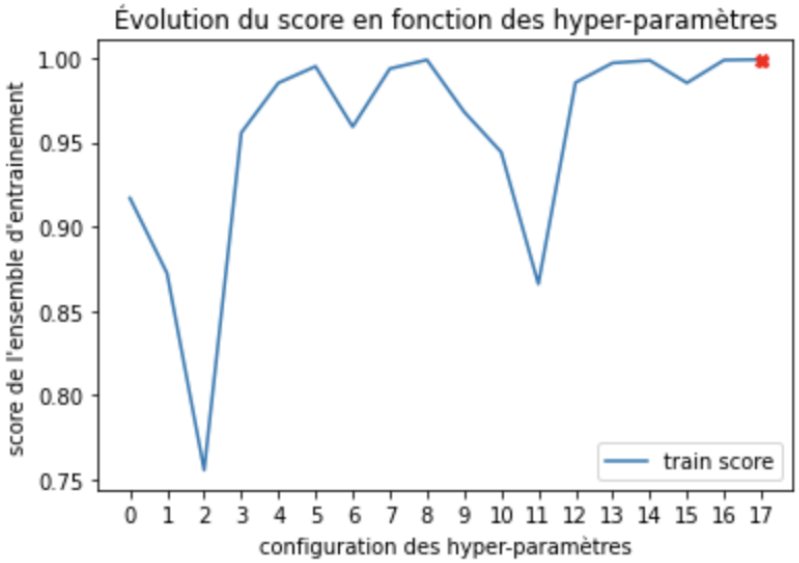


FIGURE 6 – Hyper-paramètres

En abscisse nous avons nos différentes combinaisons possibles et en ordonnée, les scores obtenus. Nous pouvons voir que les meilleurs hyper-paramètres sont : C=3, coef0=2.0, degree=4 avec un kernel = "poly", décidé en amont de la recherche.

4 Test final

Avec le modèle SVC et les hyper-paramètres trouvés précédemment, sur l'ensemble de test nous avons obtenu le score de 0.83.