Rapport de projet de reconnaissance d'images

Lucas David & Théo Legras

1 Implémentation et utilisation du classificateur à distance minimum (DMIN)

On choisi d'implémenter ce classificateur sous forme d'une classe, ceci étant le plus courant et le plus pratique pour encapsuler les comportements et stocker les données nécessaires (extrait du fichier **`dmin.py`**) :

```python
import numpy as np

class DMIN:
  def __init__(self):
    self.data = np.array([])
    self.label = np.array([])
    self.n_samples = 0

  def fit(self, data, label):
    self.data = data
    self.label = label
    self.n_samples = len(set(label))
    return self

  def predict(self, data):
    return [self.label[np.argmin(np.sum(np.subtract(self.data, data[iterator])
     ** 2, axis=1))] for iterator in range(0, len(data))]

  def score(self, data, label):
    return np.count_nonzero(self.predict(data) == label) / len(data)
```

L'utilisation se résumera à l'instanciation de `DMIN` à l'appel de `DMIN.fit` et selon l'usage l'appel de `DMIN.predict` et `DMIN.score`. En particulier, on peut donc déterminer le taux de réussite via la fonction membre `DMIN.score(<données à tester>, <labels correspondants>)`.

Dans le cas de nos données de développement, on obtient un score de 68,80% pour une exécution de 96,45 secondes. Il est toujours intéressant de noter que si on teste l'ensemble d'entraînement, on obtient le score parfait... On verra plus tard que ce n'est pas le cas de tous les algorithmes car cela peut être un indicateur d'*overfitting* (du surapprentissage ou de la surinterprétation), c'est-à-dire correspond trop étroitement aux données.

2 Utilisation de l'analyse en composantes principales (PCA) et application à DMIN

L'utilisation de l'Implémentation de la PCA (`sklearn.decompostion.PCA`) est plutôt simple. On peut choisir via le paramêtre `n_components` le nombre de dimensions à garder, si $0 \leq$ `n_components` < 1, on indique la proportion des données à garder en variance (%).

Nous nous choisissons de faire nos tests en modulant en variance plus qu'en nombre de dimensions car la notion de variance peut être mise en parallèle avec la perte de précision à postériori de la PCA. De plus cela permet d'écarter les cas de réductions dans des nombres dimensions proches (e.g. passer de 760 dims. à 700 dims. ne signifie pas grand chose alors que de 100 dims. à 40 dims. à un impact visible). Effectivement, selon le modèle on n'obtiendra pas la même courbe "Variance des données par rapport aux nombres de dimensions" et ce n'est généralement pas linéaire.

Globalement, nous appliquons dans les grandes ligne la PCA et DMIN sur les données réduites comme suit :

```
 1 import numpy as np
 2 from sklearn.decompostion import PCA
 3
 4 X = np.load('data/trn_img.npy')
 5 Y = np.load('data/trn_lbl.npy')
 6 devX = np.load('data/dev_img.npy')
 7 devY = np.load('data/dev_lbl.npy')
 8
 9 pca = PCA(n_components=0.5) # Ici on garde 50% de la variance des données.
10 reducedX = pca.fit_transform(X) # On colle au modèle et on transforme X.
11 print('Dimensions: {}'.format(reducedX)) # On peut afficher le nombre de
       dimension en valeur.
12 reducedDevX = pca.transform(reducedDevX) # On transforme devX.
13
14 dmin = DMIN() # On utilise DMIN.
15 dmin.fit(reducedX, Y)
16 print('Score: {}'.format(dmin.score(reducedDevX, devY)))
```

En terme de vitesse d'exécution initialiser la PCA et l'appliquer se fait en temps très raisonnable et permet de réduire drastiquement le temps d'exécution de DMIN.

n (%)	n (dims)	Execution time PCA	Score DMIN	Execution time DMIN
0,05	1	0,855s	22,60%	0,149s
0,1	1	0,809s	22,60%	0,157s
0,15	1	0,776s	22,60%	0,149s
0,2	1	0,779s	22,60%	0,155s
0,25	1	0,775s	22,60%	0,158s
0,3	2	0,791s	45,18%	0,207s
0,35	2	0,806s	45,18%	0,212s
0,4	2	0,772s	45,18%	0,21s
0,45	2	0,781s	45,18%	0,209s
0,5	3	0,780s	56,30%	0,256s
0,55	4	0,800s	65,44%	0,314s
0,6	5	0,821s	68,64%	0,329s
0,65	6	0,844s	71,42%	0,379s
0,7	9	0,814s	76,28%	0,501s
0,75	14	0,790s	78,12%	0,687s
0,8	24	0,806s	80,50%	1,076s
0,85	42	0,804s	81,84%	1,837s
0,9	82	0,807s	82,56%	4,129s
0,95	182	0,821s	82,36%	11,542s
1	784	–	68,80%	96,458s