Polytech Paris-Sud

Rapport de projet de reconnaissance d'images

Lucas David
Théo Legras

25 Mars 2019

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 paramètre `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.

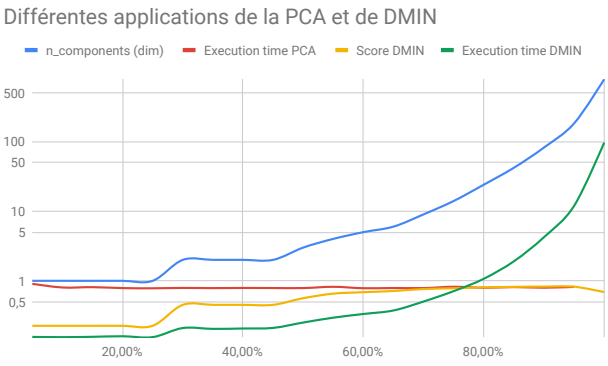
Globalement, nous appliquons dans les grandes ligne la PCA et DMIN sur les données réduites comme suit :

```
import numpy as np
from sklearn.decompostion import PCA

X = np.load('data/trn_img.npy')
Y = np.load('data/trn_lbl.npy')
devX = np.load('data/dev_img.npy')
devY = np.load('data/dev_lbl.npy')

pca = PCA(n_components=0.5) # Ici on garde 50% de la variance des données.
reducedX = pca.fit_transform(X) # On colle au modèle et on transforme X.
print('Dimensions: {}'.format(reducedX)) # On peut afficher le nombre de
    dimension en valeur.
reducedDevX = pca.transform(reducedDevX) # On transforme devX.

dmin = DMIN() # On utilise DMIN.
dmin.fit(reducedX, Y)
print('Score: {}'.format(dmin.score(reducedDevX, devY)))
```

Comme nous pouvons le voir, nous appliquons la transformation à la fois sur les données d'entraînement et sur les données de développement sinon ce ne sera pas cohérent pour notre classificateur, cependant les étiquettes n'ont pas besoin d'être modifié étant donnée qu'elles ne sont pas concernées par la transformation.

De plus, il est important de dire que la PCA a un temps d'exécution très faible de l'ordre de la seconde, ce qui rend son utilisation vraiment profitable et ce quelque soit le nombre de dimensions résultant de la réduction.

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



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. De plus, on remarquera que le fait d'appliquer la PCA améliore un peu le score lorsque l'on garde jusqu'à 50% de la variance des données, on peut se douter que le fait de retirer certaines dimensions à diminuer en quelques sortes le bruit de l'image néfaste dans le cas de l'algorithme DMIN.

3 Choix des implémentations et utilisations des classificateurs et paramètres

3.1 Support Vector Machines (SVM)

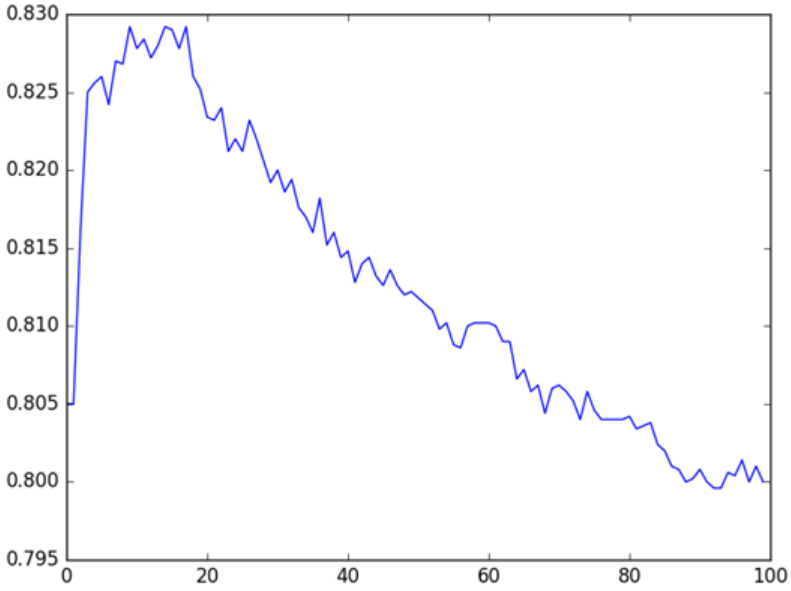
Dans un premier temps, l'utilisation de la SVM rest trivial grâce à l'implémentation des classificateurs qui reprend les même fonctions `fit`, `predict` et `score`.

On utilisera d'ailleurs `sklearn.svm.SVC` (les autres classificateurs n'apportant pas de fonctionnalité indispensable d'après la documentation du site). Pour la SVM on utilisera un gamma égale à "scale" au lieu de "auto" car "auto" donne des résultats qui sont bien inférieurs à ceux que l'on obtient avec "scale" et parfois même incompréhensible surtout avec l'usage de la PCA (de plus "auto" ne sera plus le paramètre par défaut qui sera remplacé par "scale" dans la prochaine mise à jour de scikit-learn).

On obtient une exécution d'apprentissage et de prédiction de 39,329 secondes ce qui reste très raisonnable et intéressant par rapport aux autres classificateurs surtout avec le score sur les données de développement 86,10%.

3.2 Le plus proche voisin

Dans un premier temps, l'utilisation des plus proche voisin reste trivial grâce à l'implémentation de `sklearn.neighbors.KNeighborsClassifier` qui reprend les même fonctions `fit`, `predict` et `score`.



Dans la figure ci-dessus, on peut voir les performances du classificateur `sklearn.neighbors.KNeighborsClassifier` avec l'algorithme brute en fonction du nombre k de voisins. On peut voir qu'au début cette performance augmente, et est maximum aux alentours de 5 à 10 (5 étant le nombre par défaut), puis la valeur décroit. Cette décroissance est due au faite que lorsque l'on prends trop de voisins en compte on augmente le risque d'erreurs pour les points placer proche des plans qui sépare les classes entre elles (néanmoins la réussite reste aux alentours de 80% même avec 100 voisins).

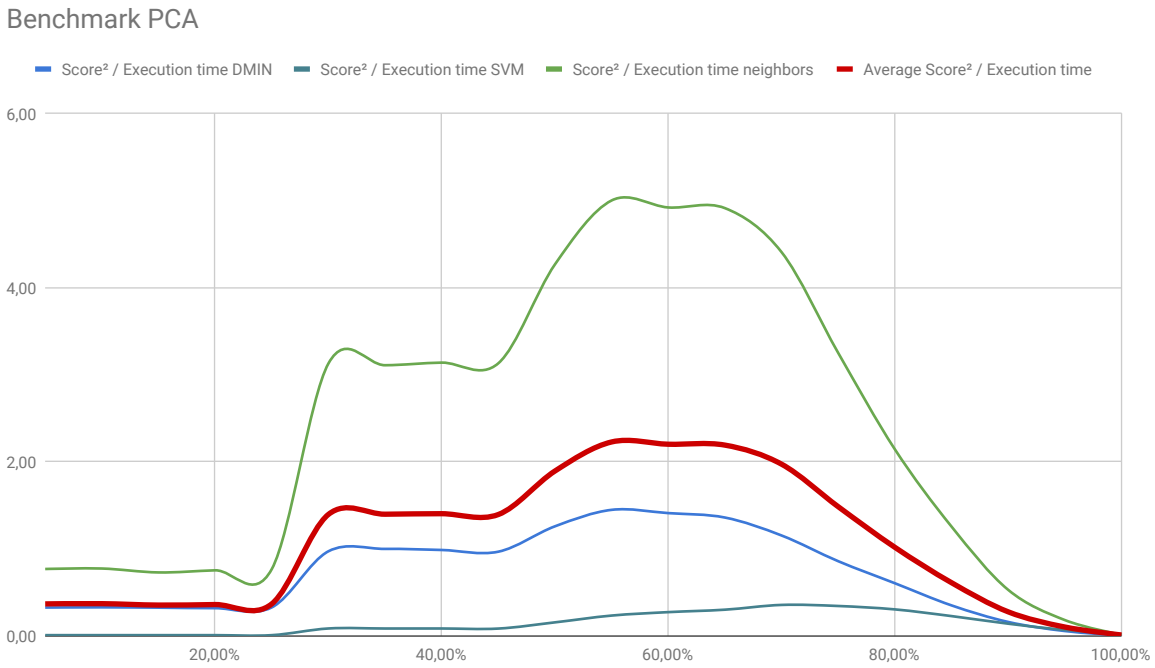
3.3 Bien paramétrer notre PCA

Pour commencer notre analyse nous avons choisis d'appliquer la PCA à différentes variance sur différentes méthodes proposer afin d'en analyser les résultats et de proposer un segment d'utilisation optimal de la PCA sur ce modèle. Pour ce faire, nous avons créer une fonction générique `BenchmarkPCA` permettant de tester les classificateurs fournies en paramètre et d'en extraire les temps d'exécutions et scores correspondants.

```
1 def BenchmarkPCA(csvfile, n_range, algorithms):
2   csvwriter = csv.writer(csvfile, delimiter=';')
3   first_row = ['n_components (%)', 'n_components (dims)', 'Execution time PCA']
4   for algorithm in algorithms:
5     first_row.extend(['Score {}'.format(algorithm), 'Execution time {}'.format(
      algorithm)])
6   csvwriter.writerow(first_row)
7
8   for n in n_range: # On fait une PCA pour chaque valeur
9     print('PCA with n_components={}:'.format(n))
10    start = time.time()
11    pca = PCA(n_components=n)
12    reducedX = pca.fit_transform(X)
13    reducedDevX = pca.transform(devX)
14    end = time.time()
15    print('PCA both fit and transform (on training and dev data) processed in {}
      sec...'.format(end - start))
16    print('\tn_components={} reduce 784 dimensions to {} dimensions.\n'.format(n
      , np.shape(reducedX)[1]))
17    csvrow = [n, np.shape(reducedX)[1], end - start]
18    # On test chaque algorithm de la liste
19    for algorithm in algorithms:
20      print('Testing {} algorithm:'.format(algorithm))
21      start = time.time()
22      if algorithm == SVC:
23        # Le mode par défaut dans la prochaine version est avec gamma='scale' et
       on voit la différence.
24        algorithm_instance = algorithm(gamma='scale')
25      else:
26        algorithm_instance = algorithm()
27      algorithm_instance.fit(reducedX, Y)
28      algorithm_score = algorithm_instance.score(reducedDevX, devY)
29      end = time.time()
30      print('Score: {}'.format(algorithm_score))
31      print('Execution time: {}\n'.format(end - start))
32      csvrow.extend([algorithm_score, end - start])
33    csvwriter.writerow(csvrow)
```

À partir des résultats de l'algorithme dessus, on obtient le tableau de valeur et son graphe associé :

n (%)	$\frac{\text{Score}^2}{\text{Exec. time DMIN}}$	Score SVC	Exec. time SVC	$\frac{\text{Score}^2}{\text{Exec. time SVC}}$	Score neighbors	Exec. time neighbors	$\frac{\text{Score}^2}{\text{Execution time neighbors}}$	Avg $\frac{\text{Score}^2}{\text{Execution time}}$
5,00%	0,33	31,94%	3,440s	0,01	25,18%	0,082s	0,77	0,37
10,00%	0,33	31,94%	3,421s	0,01	25,18%	0,082s	0,77	0,37
15,00%	0,33	31,94%	3,467s	0,01	25,18%	0,087s	0,73	0,35
20,00%	0,32	31,94%	3,432s	0,01	25,18%	0,084s	0,75	0,36
25,00%	0,33	31,94%	3,439s	0,01	25,18%	0,083s	0,77	0,37
30,00%	0,97	54,58%	1,886s	0,09	51,38%	0,084s	3,13	1,40
35,00%	1,00	54,58%	1,896s	0,09	51,38%	0,085s	3,11	1,40
40,00%	0,99	54,58%	1,891s	0,09	51,38%	0,084s	3,14	1,40
45,00%	0,97	54,58%	1,909s	0,09	51,38%	0,084s	3,13	1,40
50,00%	1,26	63,52%	1,637s	0,16	61,58%	0,089s	4,27	1,90
55,00%	1,45	70,22%	1,476s	0,23	68,96%	0,095s	5,00	2,23
60,00%	1,41	73,18%	1,430s	0,27	72,18%	0,106s	4,92	2,20
65,00%	1,36	75,06%	1,400s	0,30	74,72%	0,114s	4,91	2,19
70,00%	1,15	80,20%	1,446s	0,36	79,06%	0,142s	4,41	1,97
75,00%	0,86	82,44%	1,627s	0,34	81,28%	0,203s	3,25	1,48
80,00%	0,61	84,34%	1,965s	0,31	82,32%	0,316s	2,14	1,02
85,00%	0,35	85,38%	2,725s	0,23	83,64%	0,557s	1,26	0,61
90,00%	0,16	86,18%	4,532s	0,14	83,70%	1,322s	0,53	0,28
95,00%	0,06	86,28%	9,333s	0,07	83,36%	3,800s	0,18	0,10
100%	0,00	86,10%	39,329s	0,02	82,98%	51,104s	0,01	0,01



Sur le graphe ci-dessus, nous avons tracé le ratio entre le score au carré et le temps d'exécution. Le score est au carré pour une meilleure visibilité des courbes et car nous apportons une importance supplémentaire à la qualité des prédictions (on pourrait d'ailleurs passer le score au cube ou bien le contraire en fonction de nos besoins). On peut remarquer que la courbe explicite un segment optimale entre 55% et 70% de variance des données en moyenne (et globalement pour la majorité des classificateurs testé). Néanmoins comme l'on sais que la SVM est le classificateur le plus optimal pour notre modèle (et celui que nous allons sélectionner), nous avons choisi de nous baser sur une PCA à 80% : cela permet de garder plus de données significatives sans perdre tant de temps par rapport aux autre classificateurs.

3.4 Utilisation de la SVM pour nos données de test

Comme le montre la précédente partie, la SVM reste supérieur quelque soit la PCA appliqué que se soit en précision des prédictions que en temps d'exécution. En effet, avec la bonne maitrise du compromis entre rapidité et efficacité la SVM reste supérieur en tout point par rapport aux autres classificateurs sur notre modèle. C'est pour quoi nous avons généré notre fichier comprenant les étiquettes résultant de la prédiction à l'aide cette méthode.