A.Martin et T.Stérin

Rapport Projet de Vision

Notre technique

- Nous **thresholdons** les images de la base de données pour se ramener en images binaires.

- Nous utilisons les moments d'ordre deux pour mettre les images dans des positions et dimensions "standards" : on redimensionne et rotatione les images de telles sorte à avoir les vecteurs de **PCA** colinéaires aux vecteurs de la base canonique.

- Puis, nous extrayons le **contour** grâce aux routines d'openCV (plus performantes que ce que nous avions codé maison). Et en calculons le codage de **freeman** associé.

- Notre métrique est alors la **distance d'édition de Levenshtein** pour comparer ces codages de contours.
  On réalise un plus proche voisin pour classifier les entrées.

Pour calculer la similarité entre deux images, on fait usage de la manière dont notre algorithme les aurait classifiées.
En effet, on pourrait simplement utiliser la distance d'édition entre les deux contours mais à ce moment là on ne savait pas comment la normaliser.
Ainsi notre mesure de similarité n'à du sens que lorsque l'on compare deux images qui sont proches d'images de la BDD.

Langage et dépendances

Nous avons codé le projet en python3. Vous avez besoin des paquets:
- OpenCV pour python (cv2): sudo pip3 install opencv-python
- numpy: sudo pip3 install numpy
- Levenshtein (distance d'édition) : sudo pip3 install python-levenshtein

Architecture du projet

- `img.py` : primitives de calcul des moments, d'extraction des composantes principales, de rotations etc...

- `freeman.py` : extraction des contours, calcul du code de freeman et distance d'édition
- `l2_dist.py` : classe utilitaire pour norme L2
- `classify.py` : script de cross-validation
- `main.py` : script principal

Utilisation pratique

Nos programmes sont compatibles avec vos scripts de tests mais nous avons dû modifier une seule chose qui était problématique : votre facteur de scaling était parfois beaucoup trop petit et menait à des images dégénérées pour l'extraction des composantes principales (voir `tmp_crash.pgm`). Nous avons donc thresholdé ce facteur (l.38, `scriptClassification.sh`).
Voici des lignes de commandes types pour avoir les résultats de nos programmes:
- `python3 classify.py` : lance la mesure de cross validation pour notre technique, environs 5 exemples par classes sont retirés de la BDD et servent à mesurer le taux de succès en classif.
- `python3 main.py -classify <img>` : retourne le score de similarité d' pour chaque classe dans l'ordre du fichier `classes.csv`
- `python3 main.py -similarity <img1> <img2>` : retourne le score de similarité entre et

Pour fonctionner correctement les programmes ont besoin de la présence des deux fichiers de dump .pkl dans le même répertoire.

Nos Résultats

En mesure de cross-validation sur la base de données nous réalisons un score d'environ **85%**. La différence avec vos scripts est que l'on conserve une partie uniquement pour les tests. Cette mesure classifie des images **étrangères** à la BDD. Le problème est en un sens **plus dur** mais nous n'ajoutons pas de bruit.
Pour vos scripts:
- `scriptClassification.sh`: **197/200** succès (sans bruit)
- `scriptClassification.sh`: **191/200** succès (bruit constant: 0.7)

Comparaison avec la norme L2

Une technique plus simple que la notre consiste, après l'étape de normalisation en composantes principales, de simplement réaliser un plus proche voisin en utilisant la métrique L2 usuelle sur les vecteurs images obtenus.

Cette méthode est un peu moins bonne en cross-validation avec un score de **80%** environ.
Cependant elle fonctionne tout aussi bien avec vos scripts et ce sûrement car elle n'est confrontée qu'à des images de la base de données (même tournées, même bruitées).
On obtient les scores :
- `scriptClassification.sh`: **196/200** succès (sans bruit)
- `scriptClassification.sh`: **195/200** succès (bruit constant: 0.7)