final_zty

October 20, 2021

MH4510 Project: Brain Tumor Classification

Models included: KNN, Support Vector Machine, Random Forest

By: The *Learning Machines*

[ ]:
```
# import packages
import numpy as np
import os
import cv2
import json

from sklearn import preprocessing
from sklearn.decomposition import PCA

from sklearn.neighbors import KNeighborsClassifier
from sklearn.svm import SVC
from sklearn.ensemble import RandomForestClassifier

from sklearn.model_selection import GridSearchCV, cross_val_score
from sklearn.metrics import confusion_matrix, accuracy_score, f1_score
```

[ ]:
```
# define data directory
data_dir = 'C://Users//zhang//Desktop//BTC//data//'
train_dir = data_dir+ 'Training/'
test_dir = data_dir + 'Testing/'
os.makedirs('./logs', exist_ok=True)
```

[ ]:
```
# self-defined data loading function
def read_data(dir: str):
    classes = {'no_tumor':0,'glioma_tumor':1,'pituitary_tumor':
↪2,'meningioma_tumor':3}
    images = []  #Img info
    labels = []  #labels
    for cls in classes:
        pth = dir + cls
        for j in os.listdir(pth):
            img = cv2.imread(pth + '/' + j, 0)  #Read data in grey mode`
            img = cv2.resize(img,(224,224))     #Same size as all models
```

```
            images.append(img)
            labels.append(classes[cls])

    np.unique(labels)
    images = np.array(images)
    labels = np.array(labels)

    return images, labels
```

[ ]:
```
# read training/validation data
train_images, train_labels = read_data(train_dir)
print("Successfully loaded", len(train_images), "images, and",␣
 ↪len(train_labels), "corresponding labels for training.")

# read test data
test_images, test_labels = read_data(test_dir)
print("Successfully loaded", len(test_images), "images, and", len(test_labels),␣
 ↪"corresponding labels for testing.")
```

```
Successfully loaded 2870 images, and 2870 corresponding labels for training.
Successfully loaded 394 images, and 394 corresponding labels for testing.
```

Data Preprocessing

[ ]:
```
# flatten data
train_images = train_images.reshape(len(train_images),-1)
test_images = test_images.reshape(len(test_images),-1)

# normalization (for KNN and SVM)
train_images_std, test_images_std = map(preprocessing.StandardScaler().
 ↪fit_transform, (train_images, test_images))
```

Model 1: KNN

[ ]:
```
# model tuning
opt_pca = 0
opt_score = 0
runlog = open("./logs/knn_runlog.txt", "a")   #run log for parity check
runlog.write("KNN Model Tuning Run Log\n\n")

# looping all pca parameters
for var_ratio in np.arange(0.9, 0.96, step = 0.01):
    runlog.write("current ratio: %2f\r\n" %var_ratio)

    # apply pca to training data
    train_images_r = PCA(n_components = var_ratio, whiten = True, random_state␣
 ↪= 15).fit_transform(train_images_std)
```

```
    # exhaustive search for optimal parameters
    knn = KNeighborsClassifier()
    hyper_knn = dict(
        n_neighbors = range(1, 3),
        p = [1, 2]
    )

    knn_grid_search = GridSearchCV(
        estimator = knn,
        param_grid = hyper_knn,
        scoring = 'f1_weighted',
        cv = 5
    ).fit(train_images_r, train_labels)

    # update run log
    runlog.write(json.dumps(knn_grid_search.best_params_))
    runlog.write("\nScore: %f\r\n\n" %knn_grid_search.best_score_)

    #update optimal model globally
    if knn_grid_search.best_score_ > opt_score:
        opt_score = knn_grid_search.best_score_
        opt_pca = var_ratio
        knn_opt = knn_grid_search.best_estimator_
        knn_opt_param = knn_grid_search.best_params_

runlog.close()
```

[ ]:

```
# presenting the optimal model
print("KNN:\nOptimal pca parameter:\n", opt_pca, "Optimal parameters\n",␣
 ↪knn_opt_param)
test_images_knn = PCA(n_components = opt_pca, whiten = True, random_state = 15).
 ↪fit(train_images_std).transform(test_images_std)
```

```
KNN:
Optimal pca parameter:
 0.9 Optimal parameters
 {'n_neighbors': 1, 'p': 1}
```

[ ]:

```
# predicting the Test set results
pred_knn = knn_opt.predict(test_images_knn)
f1_knn = f1_score(test_labels, pred_knn, average='weighted')
acc_knn = accuracy_score(test_labels, pred_knn)
cm_knn = confusion_matrix(test_labels, pred_knn)

print("Weighted f1 score:\n", f1_knn, "\nAcuracy:\n", acc_knn, "\nConfusion␣
 ↪matrix:\n", cm_knn)
```

```
Weighted f1 score:
```

```
 0.7740644848070707
Acuracy:
 0.7944162436548223
Confusion matrix:
 [[105   0   0   0]
 [ 13  80   3   4]
 [  4  49  21   0]
 [  2   6   0 107]]
```

Model 2: Multiclass Support Vector Machine

```
[ ]: # model tuning
opt_pca = 0
opt_score = 0
runlog = open("./logs/svm_runlog.txt", "a")   #run log for parity check

# looping all pca parameters
for var_ratio in np.arange(0.9, 0.96, step = 0.01):
    runlog.write("current ratio: %2f\r\n" %var_ratio)

    # apply pca to training data
    train_images_r = PCA(n_components = var_ratio, whiten = True, random_state␣
 ↪= 15).fit_transform(train_images_std)

    # exhaustive search for optimal parameters
    svm = SVC()
    hyper_svm = dict(
        C = range(2, 5),
        kernel = ['poly', 'rbf', 'sigmoid'],
        degree = [4, 5]
    )

    svm_grid_search = GridSearchCV(
        estimator = svm,
        param_grid = hyper_svm,
        scoring = 'f1_weighted',
        cv = 5
    ).fit(train_images_r, train_labels)

    # update run log
    runlog.write(json.dumps(svm_grid_search.best_params_))
    runlog.write("\nScore: %f\r\n\n" %svm_grid_search.best_score_)

    #update optimal model globally
    if svm_grid_search.best_score_ > opt_score:
        opt_score = svm_grid_search.best_score_
        opt_pca = var_ratio
        svm_opt = svm_grid_search.best_estimator_
```

```
        svm_opt_param = svm_grid_search.best_params_

runlog.close()
```

[ ]:
```
# presenting the optimal model
print("Multiclass Support Vector Machine:\nOptimal pca parameter:\n", opt_pca,␣
 ↪"Optimal parameters\n", svm_opt_param)
test_images_svm = PCA(n_components = opt_pca, whiten = True, random_state = 15).
 ↪fit(train_images_std).transform(test_images_std)
```

```
Multiclass Support Vector Machine:
Optimal pca parameter:
 0.9 Optimal parameters
 {'C': 3, 'degree': 4, 'kernel': 'rbf'}
```

[ ]:
```
# predicting the Test set results
pred_svm = svm_opt.predict(test_images_svm)
f1_svm = f1_score(test_labels, pred_svm, average='weighted')
acc_svm = accuracy_score(test_labels, pred_svm)
cm_svm = confusion_matrix(test_labels, pred_svm)

print("Weighted f1 score:\n", f1_svm, "\nAcuracy:\n", acc_svm, "\nConfusion␣
 ↪matrix:\n", cm_svm)
```

```
Weighted f1 score:
 0.7075553342136243
Acuracy:
 0.7106598984771574
Confusion matrix:
 [[ 80  21   2   2]
 [ 16  51  16  17]
 [  0  15  46  13]
 [  1  11   0 103]]
```

Model 3: Random Forest

[ ]:
```
# phase 1: model tuning by setting default pca as 0.9
pca_90 = PCA(0.9, whiten=True, random_state=15).fit(train_images)
train_images_rf, test_images_rf = map(pca_90.transform, (train_images,␣
 ↪test_images))

# tuning by exhausive grid search
rf = RandomForestClassifier(oob_score = True, random_state = 15)

hyper_rf = dict(
    n_estimators = range(800, 850, 10),
    #min_samples_split = [2, 3, 5],
    #min_samples_leaf = [1, 2, 5],
```

```
    #min_impurity_decrease = [0, 0.01, 0.02]
)

rf_grid_search = GridSearchCV(
    estimator = rf,
    param_grid = hyper_rf,
    scoring = 'f1_weighted',
    cv = 5
).fit(train_images_rf, train_labels)
```

[ ]:
```
# presenting the optimal model
print("Random Forest:\nOptimal parameters under 90% pca:\n", rf_grid_search.
 ↪best_params_)
rf_opt = rf_grid_search.best_estimator_
```

```
Random Forest:
Optimal parameters under 90% pca:
 {'n_estimators': 800}
```

[ ]:
```
# predicting the Test set results
pred_rf = rf_opt.predict(test_images_rf)

f1_rf = f1_score(test_labels, pred_rf, average='weighted')
acc_rf = accuracy_score(test_labels, pred_rf)
cm_rf = confusion_matrix(test_labels, pred_rf)
print("Weighted f1 score:\n", f1_rf, "\nAcuracy:\n", acc_rf, "\nConfusion␣
 ↪matrix:\n", cm_rf)
```

```
Weighted f1 score:
 0.739012863556102
Acuracy:
 0.7868020304568528
Confusion matrix:
 [[105   0   0   0]
 [  5  20  13  62]
 [  0   2  71   1]
 [  1   0   0 114]]
```

[ ]:
```
# phase 2: pca tuning
opt_pca = 0
opt_cv = 0
runlog = open("./logs/rf_runlog.txt", "a")   #run log for parity check

# looping all pca parameters
for var_ratio in np.arange(0.9, 0.96, step = 0.01):
    runlog.write("current ratio: %2f\r\n" %var_ratio)

    # apply pca to training data
```

```
    train_images_r = PCA(n_components = var_ratio, whiten = True, random_state␣
↪= 15).fit_transform(train_images)

    rf = RandomForestClassifier(820, oob_score = True, random_state = 15)
    rf_cv_scores = cross_val_score(
        rf,
        train_images_r,
        train_labels,
        cv=5,
        scoring='f1_weighted'
    )
    rf_cv = sum(rf_cv_scores)/5

    runlog.write("cv score = %f\r\n\n" %rf_cv)

    if(rf_cv > opt_cv):
        opt_cv = rf_cv
        opt_pca = var_ratio

runlog.close()
```

```
pca =  0.9

0.631490671270384
pca =  0.91

0.622681945120038
pca =  0.92

0.6215295523605574
pca =  0.93

0.6161921487014592
pca =  0.9400000000000001

0.6222707391913734
pca =  0.9500000000000001

0.616517841466945
```

[ ]:

```
# presenting the optimal pca portion
print("Optimal pca parameter:\n", opt_pca)
train_images_rf, test_images_rf = map(PCA(n_components = opt_pca, whiten =␣
↪True, random_state = 15).fit(train_images).transform, (train_images,␣
↪test_images))
```

```
Optimal pca parameter:
 0.9
```

[ ]:
```
# predicting the Test set results
pred_rf = RandomForestClassifier(820, oob_score = True, random_state = 15).
 ↪fit(train_images_rf, train_labels).predict(test_images_rf)

f1_rf = f1_score(test_labels, pred_rf, average='weighted')
acc_rf = accuracy_score(test_labels, pred_rf)
cm_rf = confusion_matrix(test_labels, pred_rf)
print("Weighted f1 score:\n", f1_rf, "\nAcuracy:\n", acc_rf, "\nConfusion␣
 ↪matrix:\n", cm_rf)
```

```
Weighted f1 score:
 0.7331348683246979
Acuracy:
 0.7791878172588832
Confusion matrix:
 [[105   0   0   0]
 [  4  20  13  63]
 [  0   2  68   4]
 [  1   0   0 114]]
```