CS156 Assignment 6

Akmarzhan Abylay

December 2020

CS156 - Machine Learning Fashionista

Preprocessing

I am using similar preprocessing methods and an edited dataset as in Assignment 4.

[103]:
```
#importing the libraries
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import train_test_split
from glob import glob
from PIL import Image
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from tensorflow.keras.preprocessing import image


#paths for images
glob_shirts = glob('shirts_mod/*')
glob_jerseys = glob('jersey_mod/*')

size = (45, 45, 3)
```

[104]:
```
def extract(folder, out):
    """Function for extracting and resizing the images."""
    lst = []
    for path in folder:
        img = image.load_img(path, target_size=size)
        img = image.img_to_array(img)
        if len(img.flatten())==np.product(size):
            lst.append((img.flatten(), out))
    return lst
```

[105]:
```
shirts = np.asarray(extract(glob_shirts, 0))
jerseys = np.asarray(extract(glob_jerseys, 1))

#initializing the X and y using both the jerseys and shirts
X = np.append(jerseys[:, 0], shirts[:, 0])
```

```
y = np.append(jerseys[:, 1], shirts[:, 1])

X = np.stack(i for i in X)
y = np.stack(i for i in y)
```

Splitting (Train, Test)

As given in the instructions, I split my dataset into 80% training and 20% testing data. I also scaled it so that eeach feature contributes equally.

[106]:
```
# #scaling the inputs to get better results
sc = StandardScaler()
X = sc.fit_transform(X)

#splitting the dataset into 80% training and 20% testing data with stratify
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.2,␣
 ↪random_state = 0, stratify=y)
```

SVC

Linear SVC

I classified the images using different kernels (i.e., linear, poly, rbf) in SVC. I used `GridSearchCV()` to get the best parameters. Since the accuracy rate is basically `1-error rate`, I will use accuracy as a measurement of performance.

[107]:
```
from sklearn.svm import SVC
from sklearn.model_selection import GridSearchCV

param_grid = {
    'C': [0.001, 0.01, 0.1, 1],
    'tol': [1e-6, 1e-4, 1e-2]}

search = GridSearchCV(SVC(kernel="linear"), param_grid, n_jobs=-1)
search.fit(X_train, y_train)
print("Best parameter (CV score=%0.3f):" % search.best_score_)
print(search.best_params_)
```

```
Best parameter (CV score=0.655):
{'C': 0.001, 'tol': 1e-06}
```

[108]:
```
from sklearn.svm import SVC

svc_lin = SVC(kernel="linear", random_state = 0, C=0.001, tol=1e-6)
svc_lin.fit(X_train, y_train)
```

[109]:
```
from sklearn.model_selection import cross_val_score

def accuracy(clf, data, deep=False):
    """Function for printing the accuracies."""

    acc_train = cross_val_score(estimator = clf, X = X_train, y = y_train, cv =␣
 ↪5)
    acc_test = cross_val_score(estimator = clf, X = X_test, y = y_test, cv = 5)

    print("Model: {} Data\n\nTrain Set Accuracy: {:.2f} %".format(data,␣
 ↪acc_train.mean()*100))
    print("Test Set Accuracy: {:.2f} %".format(acc_test.mean()*100))

    return acc_train.mean()*100, acc_test.mean()*100
```

[110]:
```
acc_train_lin, acc_test_lin = accuracy(svc_lin, "Linear SVC")
```

```
Model: Linear SVC Data

Train Set Accuracy: 65.51 %
Test Set Accuracy: 67.57 %
```

As expected, we would get a pretty bad accuracy rate since the data is not very suitable for linear methods. However, it is still not worse than randomly guessing.

Poly SVM

[6]:
```
from sklearn.svm import SVC
from sklearn.model_selection import GridSearchCV

param_grid = {
    'C': [0.001, 0.01, 0.1, 1],
    'tol': [1e-6, 1e-4, 1e-2]}

search = GridSearchCV(SVC(kernel = "poly", degree = 2), param_grid, n_jobs=-1)
search.fit(X_train, y_train)
print("Best parameter (CV score=%0.3f):" % search.best_score_)
print(search.best_params_)
```

```
Best parameter (CV score=0.728):
{'C': 1, 'tol': 1e-06}
```

[12]:
```
svc_poly = SVC(kernel="poly", degree = 2, random_state = 0, C=1, tol=1e-6,␣
 ↪verbose = 0)
svc_poly.fit(X_train, y_train)

acc_train_poly, acc_test_poly = accuracy(svc_poly, "Poly SVC")
```

```
Model: Poly SVC Data

Train Set Accuracy: 72.79 %
Test Set Accuracy: 69.55 %
```

Polynomial SVC is better than the linear kernel SVC, but it is still doing an "OK" job in classifying.

RBF

[7]:
```
param_grid = {
    'C': [0.001, 0.01, 0.1, 1],
    'tol': [1e-6, 1e-4, 1e-2]}

search = GridSearchCV(SVC(kernel = "rbf"), param_grid, n_jobs=-1)
search.fit(X_train, y_train)
print("Best parameter (CV score=%0.3f):" % search.best_score_)
print(search.best_params_)
```

```
Best parameter (CV score=0.763):
{'C': 1, 'tol': 1e-06}
```

[13]:
```
svc_rbf = SVC(kernel = 'rbf', random_state = 0, C=1, tol=1e-6, verbose = 0)
svc_rbf.fit(X_train, y_train)

acc_train_rbf, acc_test_rbf = accuracy(svc_rbf, "Poly SVC")
```

```
Model: Poly SVC Data

Train Set Accuracy: 76.26 %
Test Set Accuracy: 74.72 %
```

RBF kernel is better than the previous two kernels, since it takes the data into higher dimensions. I would say that 74.7% without any additional preprocessing is pretty good.

Deep Neural Network

Feature Extraction

Below I used two different approaches for transfer learning. In feature extraction, the weights are frozen, so we are basically "extracting the features" of the images using the pre-trained vgg16 model. We then pass it through an additional simple layer to get the classification.

In fine-tuning, we repeat all the steps from feature extraction and then we unfreeze the weights and train it further to fit our specific classification (i.e., jerseys vs shirts) using a small learning rate.

I chose the `binary_crossentropy` loss function since we are doing a binary classification, where the target is either 0 ir 1 (i.e., shirts or jerseys). It minimizes the average difference between the predicted and true probability distributions, which is why it is suitable for our task. The perfect score is 0 and when we train the neural networks, we can see how the value of the loss function gradually decreases.

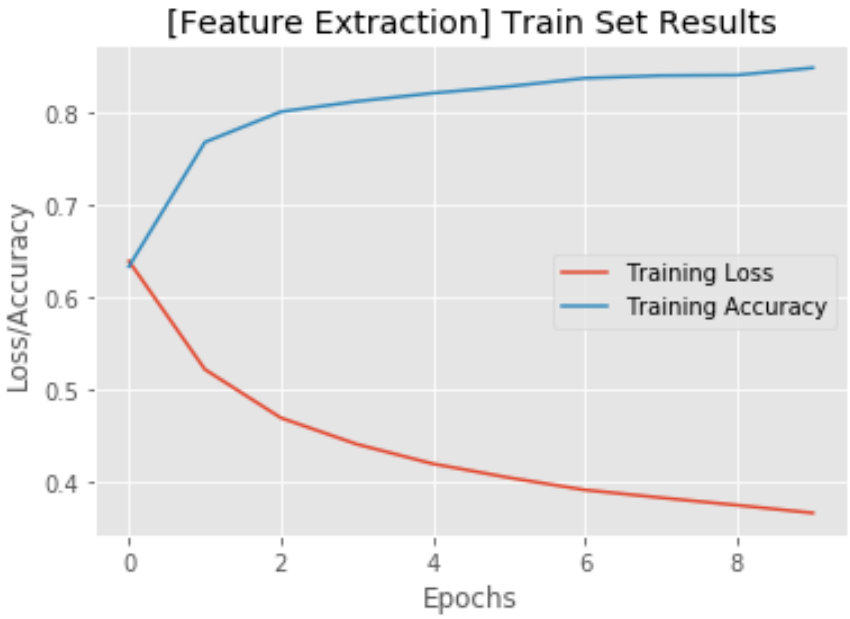
[14]:
```
from tensorflow.keras.applications.vgg16 import VGG16
from tensorflow.keras.layers import Input, Dense, Flatten
from tensorflow.keras.models import Model
```

[83]:
```
#defining the vgg16 model and changing the input dimensions
vgg16 = VGG16(weights="imagenet", include_top=False,␣
 ↪input_tensor=Input(shape=size))

#freezing the layers so that they are not trainable
for layer in vgg16.layers:
    layer.trainable = False

#adding a simple layer for classification as an output
output = vgg16.output
output = Flatten(name="flatten")(output)
output = Dense(1, activation="sigmoid")(output)

#assembling the model
model1 = Model(inputs=vgg16.input, outputs=output)

#compiling the model
model1.compile(loss='binary_crossentropy', optimizer='adam', metrics='accuracy')
```

[84]:
```
N = 10 #number of epochs

#reshaping the input to fit the neural network
X_train = X_train.reshape((len(X_train),size[0], size[1], size[2]))
X_test = X_test.reshape((len(X_test),size[0], size[1], size[2]))

extraction = model1.fit(X_train, y_train, batch_size=20, epochs=N, verbose=0)
```

Below we can see how the accuracy rate and the loss changed with each new iteration.

[85]:
```
plt.style.use("ggplot")

#plotting
epoch = np.arange(0, N)
plt.plot(epoch, extraction.history["loss"], label="Training Loss")
plt.plot(epoch, extraction.history["accuracy"], label="Training Accuracy")
plt.title("[Feature Extraction] Train Set Results")
plt.xlabel("Epochs")
plt.ylabel("Loss/Accuracy")
plt.legend(loc="best")
```



[86]:
```
from sklearn.metrics import accuracy_score

def nn_acc(clf, data):
    """Function for calculating the accuracies."""

    _, acc_train = clf.evaluate(X_train, y_train, verbose=0)
    _, acc_test = clf.evaluate(X_test, y_test, verbose=0)

    print("Model: {}\n\nTrain Set Accuracy: {:.2f} %".format(data,␣
 ↪acc_train*100))
    print("Test Set Accuracy: {:.2f} %".format(acc_test*100))

    return acc_train, acc_test
```

[87]:
```
acc_train_feat, acc_test_feat = nn_acc(model1, "Feature Extraction")
```

```
Model: Feature Extraction

Train Set Accuracy: 85.39 %
Test Set Accuracy: 81.88 %
```

Fine Tuning

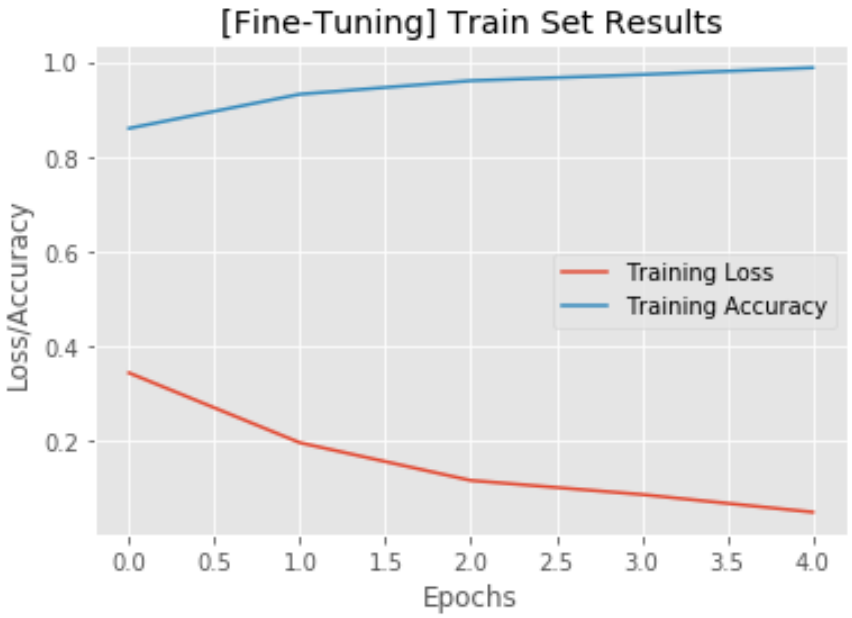
[90]:
```
from tensorflow.keras.optimizers import Adam

#unfreezing the layers
```

```
for layer in model1.layers:
        layer.trainable = True

#performing a few more iterations of gradient descent with a small learning rate
model1.compile(loss='binary_crossentropy', optimizer=Adam(lr=1e-5),␣
 ↪metrics='accuracy')
fine_tuning = model1.fit(X_train, y_train, epochs=5, batch_size=30, verbose=0)
```

A similar accuracy/loss change plot as we iterate more.

[92]:
```
#plotting
epoch = np.arange(0, 5)

plt.plot(epoch, fine_tuning.history["loss"], label="Training Loss")
plt.plot(epoch, fine_tuning.history["accuracy"], label="Training Accuracy")
plt.title("[Fine-Tuning] Train Set Results")
plt.xlabel("Epochs")
plt.ylabel("Loss/Accuracy")
plt.legend(loc="best")
```

[92]:
```
<matplotlib.legend.Legend at 0x7fae01e83990>
```



[95]:
```
acc_train_fine, acc_test_fine = nn_acc(model1, "Fine Tuning")
```

```
Model: Fine Tuning

Train Set Accuracy: 99.44 %
```

```
Test Set Accuracy: 84.12 %
```

Report

Below I made a comparison of all test and train set accuracies.

[111]:
```
accuracies = {'Linear SVC':[acc_train_lin, acc_test_lin],
              'Poly SVC':[acc_train_poly, acc_test_poly],
              'RBF SVC':[acc_train_rbf, acc_test_rbf],
              'NN [Feature Extraction]':[acc_train_feat, acc_test_feat],
              'NN [Fine-Tuning]':[acc_train_fine, acc_test_fine]}

pd.DataFrame(accuracies, index=['Accuracy (train)', 'Accuracy (test)']).round(2)
```

[111]:

	Linear SVC	Poly SVC	RBF SVC	NN [Feature Extraction]	NN [Fine-Tuning]
Accuracy (train)	65.51	72.79	76.26	0.85	0.99
Accuracy (test)	67.57	69.55	74.72	0.82	0.84

As mentioned, the "rbf" kernel was the best with 74.7% test set accuracy among the SVC classifiers. The linear kernel SVC performed the worst with around 67% (test). The "poly" kernel was in the middle with about 69.5% test set accuracy rate. The train set accuracy was higher than the test set accuracy for "poly" and "rbf" SVC classifiers, but they weren't drastically different, which might mean that they were overfitting by a bit. However, it also suggests that the SVC method wasn't really suitable for our task as the images weren't easily linearly separated, even in higher dimensions.

Neural Networks (NN) performed much better than any of the SVCs. For Feature Extraction, we just used the pre-trained VGG16 model, and it performed quite well, with a train set accuracy of 85.4% and a test set accuracy of around 81.9%. There is quite a gap between the train and test set accuracies, which might mean that the model overfitted too much on the training data and didn't pick up the general trends. The same thing happened for fine-tuning, but it was more notable. As mentioned, in fine-tuning, we unfroze the model's weights and ran a few more iterations for optimizing for our specific task. It gave us better results overall, but it overfitted hugely, as we got a train set accuracy of 99.4%, which is almost perfect, and a test set accuracy of around 84.1%. This is good enough given that we had a resized set of images (i.e., (45, 45, 3) - very low resolution) and our dataset was still a little noisy (e.g., there are still some misclassified images left; there are zoomed-in and zoomed-out photos, which include more objects than a single jersey/shirt).

I played around with the NN, and for (225, 225, 3) resized images, I got a test set accuracy of approximately 89.3% for feature extraction and almost 92% for fine-tuning. I personally felt like fine-tuning was superior since it gave the best test and train set accuracy rates, and the extra fine-tuning step also helped the NN suit the specific task of classifying jerseys and shirts better. It is also very convenient as we build upon some existing model and appropriate it to perform well in different circumstances. This is why I think the neural network where I performed fine-tuning was the best.

Extra

Noise tolerance

I also tried out different test images to see how well the NN will perform. I considered the best model (i.e., NN with fine-tuning). The pictures I chose were noisy, but I just wanted to check how well the NN will perform on images that aren't very clear (e.g., very zoomed-in pictures, shirts wrapped in plastic, several folded shirts in one shot, zoomed out pictures with faces). I just chose 5 photos of shirts and 5 jerseys from the internet that seemed confusing to me (i.e., the model never saw them) to see whether the NN will classify them correctly.

It did relatively well as it correctly classified 7 out of 10 pictures. I used the same method for predicting and assessing the accuracy as above (which is why I didn't include the additional code for that). It ended up misclassifying only the images which had extreme noise in them, such as the shirt in a plastic wrap, two people with the same shirt in one picture, and a collection of different jerseys on the floor. In the beginning, I wasn't even sure whether the shirt in the plastic wrap was a shirt at all. I would say that the model tolerates some level of noise but is bad for extreme cases and particularly noisy data. We need more data and a better model (e.g., more complicated and that also prevents overfitting; we could use Dropout layers) for better performance.

Testing on original dataset

As mentioned, I fixed some of the misclassified images and deleted images that didn't resemble shirts nor jerseys. I was curious to see whether this affected the performance of my model. I ran the same code lines but for the original dataset and obtained the best testing accuracy of around 62% for the SVC method overall. The RBF kernel turned out to be the best, once again, but the test set accuracy wasn't as high as on the edited data. I got a test set accuracy of around 70% for the feature extraction NN, which is not very good but still ok. For the fine-tuning NN, I got a testing accuracy of about 71%.

I also performed a difference of proportions (in %) test to see whether the difference between the test set accuracy for the fine-tuning NN on original data was significant compared to the same model on the fixed dataset. I wanted to assess whether fixing the dataset changed anything. Using *this* resource, for 561 test points that I assessed, it showed me that the p-value was much smaller than 0.0001 (threshold was 0.05), which means that fixing the dataset made a huge difference. There are still some misclassified shirts and jerseys left in the dataset that I didn't pick up, but I guess that if the datasets were perfect (which is very rare), our results would have been even better.