Yang, Steven. Assignment 5

April 10, 2022

1 Assignment 5

1.1 Steven H. Yang, Minerva University

1.2 Assignment instructions:

1.2.1 Machine Learning Fashionista 2.0

In this assignment we revisit the dataset from the dimension reduction unit. The pictures of clothing are all originally taken from ImageNet, which is a large dataset containing over a million photos with many different categories. Every year there is a competition to see which techniques perform the best. The winning entry is then open-sourced and made available to all machine learning researchers for further research or to allow the development of novel applications.

Now we want to compare SVMs and deep neural networks.

[1]:
```python
from keras.applications.vgg16 import VGG16
from sklearn import datasets
from keras.models import Model
from sklearn import svm
from sklearn.model_selection import train_test_split
from sklearn.metrics import classification_report
from keras.preprocessing import image
from keras.layers import Dense
from keras.applications.vgg16 import preprocess_input
from sklearn.metrics import accuracy_score
from sklearn.svm import SVC
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
from glob import glob
```

[2]:
```python
# set a path for all images
jerseys = glob('Jersey/*')
shirts = glob('Shirt/*')

jersey_flattened = []
shirt_flattened = []

# for each image path
```

```
for path in jerseys:
    img = image.load_img(path, target_size=(224, 224))
    x = image.img_to_array(img)
    x = np.expand_dims(x, axis=0)
    jersey_flattened.append(x)
jersey_flattened = np.asarray(jersey_flattened).reshape(-1, 224, 224, 3)
jersey_flattened = preprocess_input(jersey_flattened)

# for each image path
for path in shirts:
    img = image.load_img(path, target_size=(224, 224))
    x = image.img_to_array(img)
    x = np.expand_dims(x, axis=0)
    shirt_flattened.append(x)
shirt_flattened = np.asarray(shirt_flattened).reshape(-1, 224, 224, 3)
shirt_flattened = preprocess_input(shirt_flattened)

# Define X and Y variables
X = np.concatenate((jersey_flattened, shirt_flattened))
Y = np.array([0 for i in range(len(jersey_flattened))] + [1 for i in␣
 ↪range(len(shirt_flattened))])

# Split the data, use test size of 0.3 to avoid overfitting.
X_train, X_test, Y_train, Y_test = train_test_split(X, Y, test_size=0.3,␣
 ↪shuffle=True)
print(X_train.shape, X_test.shape, Y_train.shape, Y_test.shape)
```

```
(1961, 224, 224, 3) (841, 224, 224, 3) (1961,) (841,)
```

1. Support Vector Machines

- Train a support vector classifier using each of the following kernels:
  - Linear
  - Poly (degree = 2)
  - RBF
- If you encounter any issues with training time or memory issues, then you may use a reduced dataset, but carefully detail why and how you reduced the dataset. Unnecessarily reducing the dataset will result in reduced grades!
- Report your error rates on the testing dataset for the different kernels.

1.3 Logic behind the Training

Since the SVM classifier will include a lot of noise, there's a brilliant way needed to cancel this noise that was originated from the variance. To do so, I use the L1 regularization parameter into the SVM classifier. This will avoid the images wrongly orientied or classified completely wrong labels.

[3]:
```
n_samples = X_train.shape[0]
X_train_svm= X_train.reshape((n_samples, -1))
```

```
n_samples = X_test.shape[0]
X_test_svm = X_test.reshape((n_samples, -1))

# Training a polynomial kernel
poly2 = svm.SVC(C=0.5, kernel='poly', degree =2)
poly2.fit(X_train_svm, Y_train)

# Training an RBF kernel
rbf = svm.SVC(C=0.5, kernel= 'rbf')
rbf.fit(X_train_svm, Y_train)

# Training a linear kernel
linear = svm.SVC(C=0.5, kernel='linear')
linear.fit(X_train_svm, Y_train)

print('Polynomial - 2D SVM error rate: ', 1 - poly2.score(X_test_svm, Y_test))
print('RBF SVM Accuracy: ', 1 - rbf.score(X_test_svm, Y_test))
print('Linear SVM Accuracy: ', 1 - linear.score(X_test_svm, Y_test))
```

```
Polynomial - 2D SVM error rate:  0.3721759809750297
RBF SVM Accuracy:  0.35315101070154575
Linear SVM Accuracy:  0.41736028537455405
```

[4]:
```
# Training a polynomial kernel with degree of 3
poly3 = svm.SVC(C=0.5, kernel='poly', degree =3)
poly3.fit(X_train_svm, Y_train)

# Training a polynomial kernel with degree of 4
poly4 = svm.SVC(C=0.5, kernel='poly', degree =4)
poly4.fit(X_train_svm, Y_train)

print('Polynomial - 3D SVM error rate: ', 1 - poly3.score(X_test_svm, Y_test))
print('Polynomial - 4D SVM error rate: ', 1 - poly4.score(X_test_svm, Y_test))
```

```
Polynomial - 3D SVM error rate:  0.39120095124851373
Polynomial - 4D SVM error rate:  0.40190249702734837
```

1.4 Error Rate

Based on the models above, the error rates are following: - SVM Degree 2: 37% - RBF SVM: 35% - Linear SVM: 42%

Based on this observation, polynomial SVM may show the better performance than the Lienear SVM. Therefore, trying a polynomial SVM with higer degree will lower the error rate; therefore, I try this in the above code cell.

Unfortunately, higher degree polynomial did not really show me the lower error rates. I proved that my hypothesis is wrong in this case. However, it is still valid that the polynomial SVMs perfrom better than linear SVM. Therefore, in this case, it is better to use either polynomial SVM with 2 degree or RBF SVM based on their performances. Hypothetically, I suspect the data is Gaussian distributed because of the RBF error rate.

2. Deep Neural Networks Using Keras load the VGG16 network. This is the convolutional neural network which won ImageNet 2014, and the accompanying paper is available here, if you want to read more about it. Keras code to perform this step is available here, under the heading "Extract features with VGG16." - Perform transfer learning using VGG16. - What loss function did you choose, and why? - What performance do you achieve on your test set and how does this compare to the performance you were originally able to achieve with the linear methods? - (optional) If you want, you can also perform a "fine-tuning" step. In this step we unfreeze the weights and then perform a few more iterations of gradient descent. This fine tuning can help the network specialize its performance in the particular task that it is needed for. Now, measure the new performance on your test set and compare it to the performance from the previous step.

[11]:
```
# define the VGG model and freeze the convolutional layers
vgg = VGG16(weights='imagenet', include_top=True)
vgg.trainable = False
output = Dense(1, activation='sigmoid')(vgg.layers[-2].output)

# define the keras model
model = Model(inputs = vgg.input, outputs = output)
# compile the keras model
model.compile(loss='binary_crossentropy', optimizer='adam',␣
 ↪metrics=['accuracy'])
# fit the keras model on the dataset
model.fit(X_train, Y_train, epochs=10, batch_size=10)
```

```
Epoch 1/10
197/197 [==============================] - 355s 2s/step - loss: 0.6209 -
accuracy: 0.7292
Epoch 2/10
197/197 [==============================] - 364s 2s/step - loss: 0.4912 -
accuracy: 0.7879
Epoch 3/10
197/197 [==============================] - 376s 2s/step - loss: 0.3802 -
accuracy: 0.8225
Epoch 4/10
197/197 [==============================] - 343s 2s/step - loss: 0.3621 -
accuracy: 0.8394
Epoch 5/10
197/197 [==============================] - 361s 2s/step - loss: 0.3223 -
accuracy: 0.8649
Epoch 6/10
197/197 [==============================] - 369s 2s/step - loss: 0.2781 -
accuracy: 0.8807
Epoch 7/10
197/197 [==============================] - 379s 2s/step - loss: 0.2914 -
accuracy: 0.8730
```

```
Epoch 8/10
197/197 [==============================] - 389s 2s/step - loss: 0.2292 -
accuracy: 0.9128
Epoch 9/10
197/197 [==============================] - 366s 2s/step - loss: 0.2068 -
accuracy: 0.9148
Epoch 10/10
197/197 [==============================] - 334s 2s/step - loss: 0.1947 -
accuracy: 0.9220
```

[11]:
```
<keras.callbacks.History at 0x7f900d1164f0>
```

[12]:
```python
predictions = (model.predict(X_test) > 0.5).astype(int)

# Error is 1 - accuracy
print(1 - accuracy_score(predictions, Y_test))
```

```
0.26040428061831156
```

1.5 Error Rate

I applied the binary cross entropy as my loss function. It is because the model will only include two labeling of the classes and the goal of the model is maximizign the entroy between two classes. I considered other two functions: cateogrical cross entropy and Poisson loss function. However, given that the data itself will be classified binarily, categorical cross entropy may result in overfitting while the Poisson loss function will have a significant preconception of the data distribution.

I applied the sigmoid activation for the VGG16 model. This will help my investigation to see if the error rates are lower than the error rates that were observed in the previous models above. This activation will filter out the most significant noises from the data had been filtered already in VGG16, so it will get us a lower error rate. Therefore, the modeling would be much easier as I get rid out off the noise that negatively contributes to the fit.

As a result, the error rate showed 0.26. This means that the error was significantly less than anyother models that we tried above. Thanks to the processing of the noise data, I was able to get the great accuracy for this time!

1.6 Conclusion

For everything I tried above, no matter which kernal I use for the SVM, neural training method with VGG16 performed much better. It is because by appling binary cross entropy and sigmoid activation, I avoid any viable way to perform necessary feature extraction; therefore, the noise variation within the images will be cancelled so the result gets better. The similar idea can be applied to the SVM models by appling an LDA model and then apply the SVM model after. However, this is not the greatest method to try because there maybe incorrectly labelled images still exist. This will result the wrongly oriented images in the training set's LDA transformation's Eigenimages; therefore the performance would be not good enough as much as we observed.

Therefore, the neural network model is much preferred in this dataset and classification.