Assignment 3: PCA & LDA

Ash

2018/12/06

Preparing Data

Since the original dataset is quite messy: the images are of different sizes, there are images with only shirt/jersey with background and there are images with people wearing shirt/jersey, images are taken from different angles and some of which does not show the entire shirt/jersey,... we will will perform analysis on two selections of the dataset. The first selection would be 200 each for shirt and jersey (400 in total) selected randomly from the original dataset to judge the general performance of the classifier we are buidling. The second selection would be around 100 each for shirt and jersey (215 in total) hand-picked to have a clear, distinguishable background to visualize the variances in the principle components and the eigenshirt/eigenjersey. Data will be load into Python using OpenCV and turned into numpy ndarray that has the dimension $(N, 200, 200, 3)$, where N is the number of data in the array, 200 is the pixel height and width of the images (images that has bigger or smaller dimension will be interpolated using the cubic interpolation of OpenCV) and 3 is the number of color channel (RGB).

```
1
2 '''
3 Dumping shirts and jerseys data for later usage
4 '''
5 import cv2
6 import glob
7 import numpy as np
8
9
10 #--Dumping 200 manually chosen clean background shirts and jerseys--#
11 path_s = '/Users/ash/Downloads/cs156_3/s_100/*JPEG'
12 path_j = '/Users/ash/Downloads/cs156_3/j_100/*JPEG'
13
14 #Using glob to iteratively parsing the files with JPEG name in a folder,
       glob
15 # will close the file after parsing automatically
16 files_s = glob.glob(path_s)
17 files_j = glob.glob(path_j)
18
19 shirts, jerseys = np.zeros(shape=(106, 200, 200, 3)), np.zeros(shape=(109,
       200, 200, 3))
```

```
20
21 shape_0 = []
22 shape_1 = []
23
24 #Reading the parsed images using OpenCV, interpolated them to be 200x200
       and put them into a ndarray
25 count_s = 0
26 for _ in files_s:
27     img = cv2.imread(_)
28     shape_0.append(img.shape[0])
29     shape_1.append(img.shape[1])
30     dim = (200,200)
31     resized = cv2.resize(img, dim, interpolation=cv2.INTER_CUBIC)
32     shirts[count_s,:] = resized
33     count_s +=1
34
35 #Priting out the average size of the image to see whether 200x200
       interpolation is reasonable
36 print('Average size of shirt images:', (np.mean(shape_0),np.mean(shape_1)))
37
38
39 #Doing the same thing for jerseys
40 shape_0 = []
41 shape_1 = []
42
43 count_j = 0
44 for _ in files_j:
45     img = cv2.imread(_)
46     shape_0.append(img.shape[0])
47     shape_1.append(img.shape[1])
48     dim = (200,200)
49     resized = cv2.resize(img, dim, interpolation=cv2.INTER_CUBIC)
50     jerseys[count_j,:] = resized
51     count_j +=1
52
53 print('Average size of jersey images:', (np.mean(shape_0),np.mean(shape_1))
       )
54
55 #Dumping the ndarrays to local machine for future usage
56 shirts.dump('/Users/ash/Downloads/cs156_3/shirts_100.data')
57 jerseys.dump('/Users/ash/Downloads/cs156_3/jerseys_100.data')
58
59
60 #__Choosing and dumping 2000 programatically chosen shirts and jerseys__#
61
62 #Doing the same thing as above, just with 200 randomly chosen images
       instead of 100 pre-selected ones
63 path_s = '/Users/ash/Downloads/cs156_3/shirt_original/*JPEG'
64 path_j = '/Users/ash/Downloads/cs156_3/jersey_original/*JPEG'
65
66 files_s = glob.glob(path_s)
67 files_j = glob.glob(path_j)
68
69 chosen_s = []
```

```
chosen_j = []

for _ in np.random.choice(len(files_s),size=200, replace=False):
    chosen_s.append(files_s[_])

for _ in np.random.choice(len(files_j),size=200, replace=False):
    chosen_j.append(files_j[_])

shirts, jerseys = np.zeros(shape=(200, 200, 200, 3)), np.zeros(shape=(200,
    200, 200, 3))

count_s = 0
for _ in chosen_s:
    img = cv2.imread(_)
    dim = (200,200)
    resized = cv2.resize(img, dim, interpolation=cv2.INTER_CUBIC)
    shirts[count_s,:] = resized
    count_s +=1

count_j = 0
for _ in chosen_j:
    img = cv2.imread(_)
    dim = (200,200)
    resized = cv2.resize(img, dim, interpolation=cv2.INTER_CUBIC)
    jerseys[count_j,:] = resized
    count_j +=1

shirts.dump('/Users/ash/Downloads/cs156_3/shirts_200.data')
jerseys.dump('/Users/ash/Downloads/cs156_3/jerseys_200.data')


-->Output:

Average size of shirt images: (404.4622641509434, 388.4339622641509)
Average size of jersey images: (384.42201834862385, 385.42201834862385)
```

200x200 resizing seems reasonable since bigger images will resulted in a bigger dataset, which is hardly to analyze given the constraints on computational power.

Loading Data

First, let's look at the clear background, pre-selected dataset. From this point on, unless otherwise indicated, the analysis will be performed on the 215 clear background dataset.

```
#Loading relevant packages
import numpy as np
import matplotlib.pyplot as plt
from sklearn.decomposition import PCA
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis
```

```
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score, f1_score, roc_auc_score,
    confusion_matrix
from sklearn.model_selection import cross_val_score, train_test_split

#Loading and labeling dumped images
'''
215 images
'''
shirts = np.load('/Users/ash/Downloads/cs156_3/shirts_100.data')
jerseys = np.load('/Users/ash/Downloads/cs156_3/jerseys_100.data')

# '''
# 400 images
# '''
# shirts = np.load('/Users/ash/Downloads/cs156_3/shirts_200.data')
# jerseys = np.load('/Users/ash/Downloads/cs156_3/jerseys_200.data')

data = np.concatenate((shirts,jerseys), axis=0)
data = data/255
data = data.reshape(data.shape[0], 200*200,3)
label = np.zeros(shape=(data.shape[0], 1))
label[:shirts.shape[0],:] = 1

#Splitting to train/test set with 80/20 ratio, while also reshaping to a
    two dimensional ndarray for compatibility with sklearn
X_train, X_test, y_train, y_test = train_test_split(data.reshape(data.shape
    [0], data.shape[1]*data.shape[2]),label, test_size=0.2, random_state
    =25)

print(X_train.shape)
print(y_train.shape)
print(X_test.shape)
print(y_test.shape)

-->Output:

(172, 120000)
(172, 1)
(43, 120000)
(43, 1)
```

Randomly choosing some data in the training set to visualize to make sure we were dumping and loading the data correctly, we can see that in figure 1 the shirts and jerseys are in very clean background (since they have been hand-picked). Interpolation seems to work well, since we can see the images without any glitches in the pixels.

```
choices = np.random.choice(X_train.shape[0], size=20, replace=False)

count=1
plt.figure(figsize=(12,6))
for _ in choices:
    plt.subplot(5,4,count)
    plt.imshow(X_train[_].reshape(200,200,3))
```

```
    plt.axis('off')
    count +=1
plt.show()

-->Output:

Figure 1
```

Figure 1: Some Shirt/Jersey From The Training Set For The Clean Dataset

Logistic Regression On Original Data

Training a logistic regression on l-2 norm regularization for the training data and evaluate the scores by two-fold cross-validation. Since we have a small training set, choosing a bigger k for k-fold cross-validation will likely return not very good results because the classifier does not have enough training example to learn from, so we choose 2. We also choose ridge regression instead of lasso (l-1 norm regularization) because rigde regression encourages shrinkages in the coefficients, while lasso also encourages feature selection by promoting unimportant feature to have 0 for the coefficient, and since by looking at the example we see there are much variability in the position of the "important" aspects of the shirt/jersey, and they are not really concentrated in a region of pixels so selecting pixels via the lasso seems unnecessary.

```
#1. Train a simple logistic regression on the unprojected data

model_1 = LogisticRegression(penalty='l2', solver='lbfgs')
```

```
accs = cross_val_score(model_1,X_train, y_train.reshape(-1,), scoring='
    accuracy', cv=2)

f1s = cross_val_score(model_1,X_train, y_train.reshape(-1,), scoring='f1',
    cv=2)

aucs = cross_val_score(model_1,X_train, y_train.reshape(-1,), scoring='
    roc_auc', cv=2)

print("Acc:", accs)
print("F1:", f1s)
print("ROC AUC:", aucs)

--->Output:

Acc: [0.73563218 0.72941176]
F1: [0.72289157 0.73563218]
ROC AUC: [0.76825397 0.80210643]
```

Accuracy scores show that the classifier performed okay with around 70% accuracy, and had some balance between the precision and recall judging by the F1 score. This seems reasonable given the limited amount of data we have. Let's see if the results generalize well on the test set:

```
#Fitting the model on the whole training set and evaluate its performance
model_1.fit(X_train, y_train.reshape(-1,))
preds = model_1.predict(X_train)
print('Train set accuracy:', accuracy_score(y_train, preds))
print('Train set f1:', f1_score(y_train, preds))
print('Train set ROC AUC:', roc_auc_score(y_train, preds))

print('##DIAGNOSTICS INFO##-Interations till convergence:', model_1.n_iter_
    )

#Evaluating the model performance on the test set
test_preds = model_1.predict(X_test)
print('Test set accuracy:', accuracy_score(y_test, test_preds))
print('Test set f1:', f1_score(y_test, test_preds))
print('Test set ROC AUC:', roc_auc_score(y_test, test_preds))

--->Output:
Train set accuracy: 1.0
Train set f1: 1.0
Train set ROC AUC: 1.0
##DIAGNOSTICS INFO##-Interations till convergence: [100]

Test set accuracy: 0.5581395348837209
Test set f1: 0.5581395348837209
Test set ROC AUC: 0.5608695652173914
```

We can clearly see that the model is overfitting even with regularization. This is hardly surprising since we have $200 \times 200 \times 3 = 120000$ dimension feature space, and with only around 170 training examples the classifier can easily overfit the training set. A reasonable next step if we want to improve this simple model is using grid-search or

similar technique to tune the classifier parameters, maybe focusing on the regularization weight. But we can see also see that the solver had convergence at 100 iterations, which is the maximum allowed number of iteration, which means the solver had a hard time navigate around the enormous feature space. Given this, it's probably better to look at some dimensionality reduction techniques rather than tune this simple model.

Logistic Regression On PCA-Projected Data

Using PCA, we can plot the total explained variance for the projection of data on figure 2.

```
#2. Use PCA to choose and project data to a lower dimension, then run
    logistic regression

#Initiating the PCA
pca_rgb = PCA(n_components=data.shape[0])
pca_rgb.fit(data.reshape(data.shape[0], data.shape[1]*data.shape[2]))
cum_sum = np.cumsum(pca_rgb.explained_variance_ratio_)

#Plotting explained variance ratio
plt.figure(figsize=(12,8))
plt.scatter([_ for _ in range(data.shape[0])], cum_sum)
plt.xlabel('Number of component')
plt.ylabel('Explained variance')
plt.show()
```

The explained variance ratio starts to flat out at around 100 components, so we will project the data into its 100 principle components for dimension reduction and then train a new logistic regressor to differentiate between shirts and jerseys.

```
print('Using 30 first components, containing %f of explained variance...' %
    cum_sum[30])

#Projecting the data using PCA with 30 components
pca_rgb = PCA(n_components=30)
X_train_re = pca_rgb.fit_transform(X_train)
X_test_re = pca_rgb.transform(X_test)

model_1_re = LogisticRegression(penalty='l2', solver='lbfgs')

model_1_re.fit(X_train_re, y_train.reshape(-1,))
preds_re = model_1_re.predict(X_train_re)
print('Train set accuracy:', accuracy_score(y_train_re, preds_re))
print('Train set f1:', f1_score(y_train_re, preds_re))
print('Train set ROC AUC:', roc_auc_score(y_train_re, preds_re))

print('##DIAGNOSTICS INFO##-Interations till convergence:', model_1_re.
    n_iter_)

test_preds_re = model_1_re.predict(X_test_re)
print('Test set accuracy:', accuracy_score(y_test_re, test_preds_re))
print('Test set f1:', f1_score(y_test_re, test_preds_re))
```

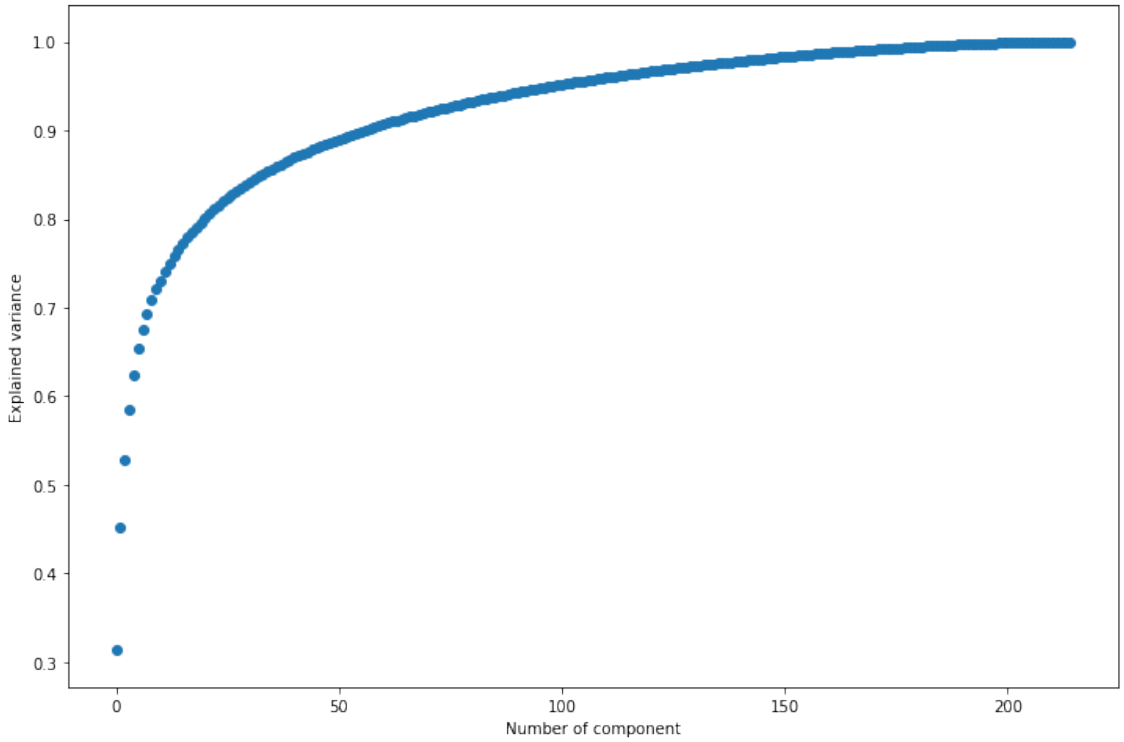


Figure 2: Explained Variance Ratio For PCA Projection Of The Clean Dataset

```
print('Test set ROC AUC:', roc_auc_score(y_test_re, test_preds_re))

--->Output:

Using 30 first components, containing 0.841679 of explained variance...
Train set accuracy: 0.8081395348837209
Train set f1: 0.7950310559006212
Train set ROC AUC: 0.8068904832814404
##DIAGNOSTICS INFO##-Interations till convergence: [75]

Test set accuracy: 0.7441860465116279
Test set f1: 0.7555555555555555
Test set ROC AUC: 0.7445652173913043
```

We can see that with only 30 principle components the classifier was able to generalize much better, which means there many of the 120000 pixels are essentially not informative to the classifier with this classification task. This is understandable since much of the original pixels are background along with not-useful information such as the colors of the shirt/jersey. However, in figure 3 we can see that the train/test accuracy is very unstable, indicating that the model is still not too good at distinguishing between shirts and jerseys. In this case, we probably should turn into a different classifier, like SVM, to model the differences between jersey and shirts.

```
#Training a classifier on each of the first 50 components of the PCA, then evaluate the classifier on the trainingset and the testset
train_acc, test_acc = [], []
for _ in range(1,51):
    pca_rgb = PCA(n_components=_)
    reduced_data = pca_rgb.fit_transform(data.reshape(data.shape[0], data.shape[1]*data.shape[2]))
    X_train_re, X_test_re, y_train_re, y_test_re = train_test_split(reduced_data, label, test_size=0.2, random_state=25)
    model_1_re = LogisticRegression(penalty='l2', solver='lbfgs')
    model_1_re.fit(X_train_re, y_train_re.reshape(-1,))
    preds_re = model_1_re.predict(X_train_re)
    train_acc.append(accuracy_score(y_train_re, preds_re))
    test_preds_re = model_1_re.predict(X_test_re)
    test_acc.append(accuracy_score(y_test_re, test_preds_re))

x = [_ for _ in range(50)]
plt.figure(figsize=(12,8))
plt.plot(x, train_acc, label='train', linestyle='--')
plt.plot(x, test_acc, label='test', linestyle='--' )
plt.plot(x, cum_sum[:50], label='explained_variance')
plt.legend()
plt.xlabel('Number of component')
plt.ylabel('Accuracy/Explained variance ')
plt.show()

--->Output:

Figure 3
```

It is also interesting to look at the PCA principle components of the shirts to see the most "common" feature of the shirts (in other words the shirt that retains the most variance, which is the most general of all the shirts) (so called eigen-shirt). We can see in figure 4.1 that the eigenshirt (correspond to the first principle component where most of the variance is) the shape of the shirt is very clear, and with this much variance in the first component we were able to distinguish the shape of the shirt very clearly. In figure 4.2, we can see that the 2nd, 3rd and so on next component also contains some information on the "shirtness" of a shirt, but for example the third and fourth row in fugure 4.2 indicates that these components contain uninformative information such as the shape of the arms, or the patterns on the shirts. We can see similar things with the eigenjersey in figure 5: in 5.1 we can clearly the most distinguishable feature of the shirt and the jersey is the shape of its arm, and like with the shirt the 2nd, 3rd,... components of the jerseys (figure 5.2) contains useless information (in terms of the classification task at hand) like the stripe patterns (there were indeed many sport jerseys with stripes in the dataset).

```
#2.5 Eigenshirt and eigenjersey (R,B,G and RBG)

##Shirts
plt.figure(figsize=(6,6))
```

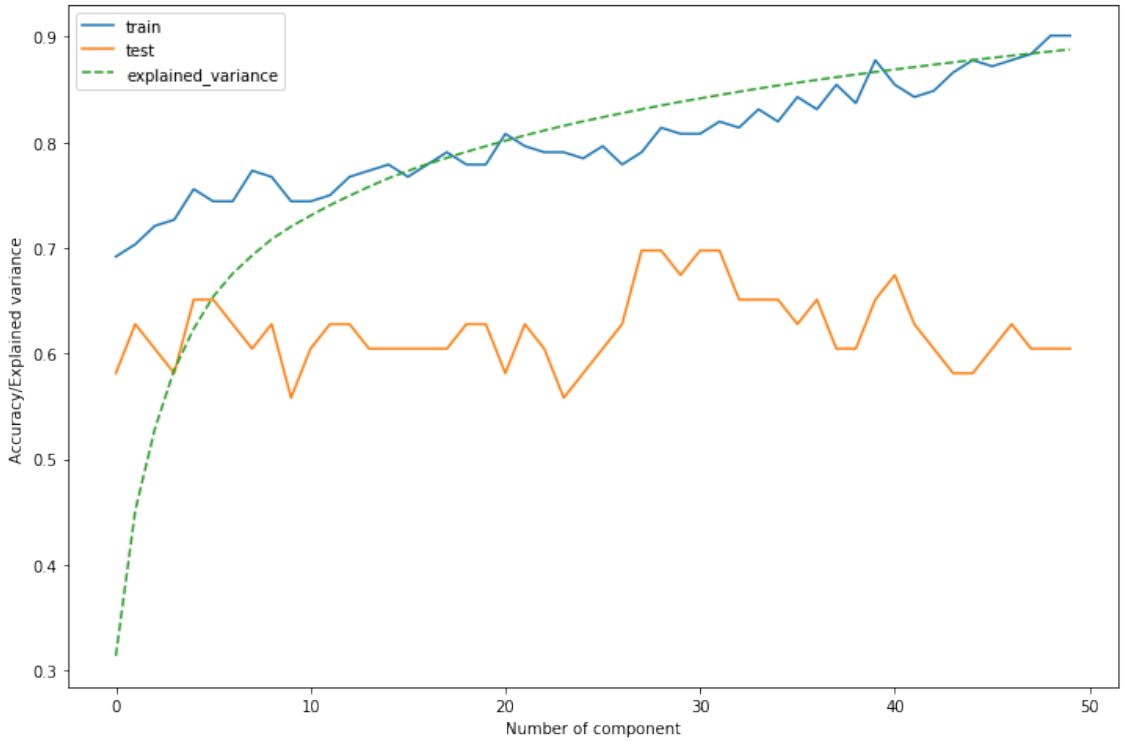
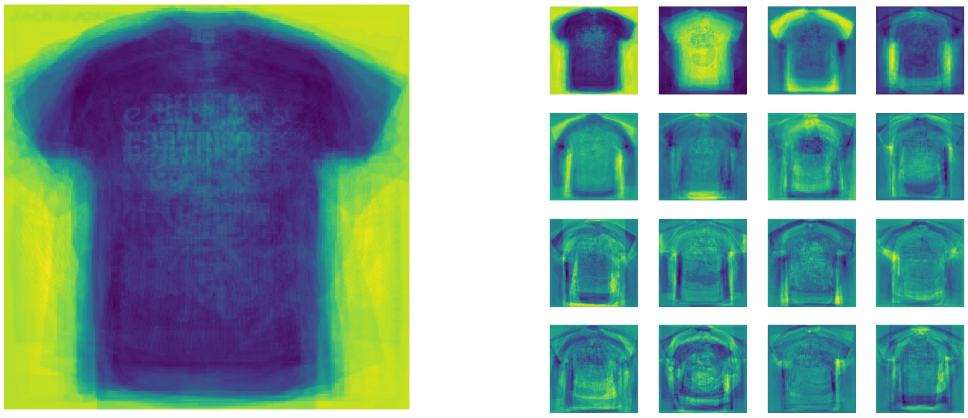


Figure 3: Train/Test Accuracy For PCA-Projected Data Using Logistic Regression

```
decomposer_r = PCA(n_components=30)
decomposer_g = PCA(n_components=30)
decomposer_b = PCA(n_components=30)
eigen = shirts.reshape(shirts.shape[0], 200*200,3)/255
decomposer_r.fit(eigen[:,:,0])
decomposer_g.fit(eigen[:,:,1])
decomposer_b.fit(eigen[:,:,2])

plt.imshow(decomposer_r.components_[0].reshape(200,200))
plt.imshow(decomposer_g.components_[0].reshape(200,200))
plt.imshow(decomposer_b.components_[0].reshape(200,200))
plt.show()

count = 1
plt.figure(figsize=(12,12))
for _ in range(16):
    plt.subplot(4,4,count)
    plt.imshow(decomposer_r.components_[_].reshape(200,200))
    plt.imshow(decomposer_g.components_[_].reshape(200,200))
    plt.imshow(decomposer_b.components_[_].reshape(200,200))
    plt.axis('off')
    count+=1
plt.show()

```



(a) First Principle Component

(b) First 16 Principle Components

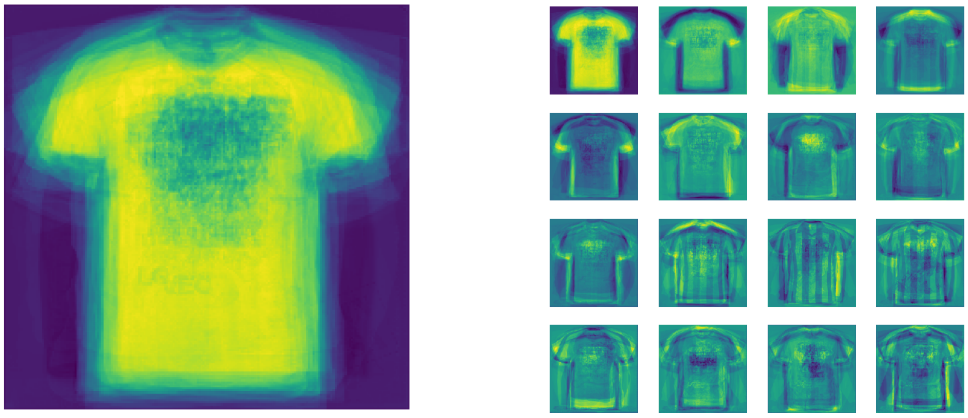
Figure 4: Eigenshirt

```
—>Output:

Figure 4.1 and 4.2
```

Logistic Regression On LDA-Projected Data

Using the LDA class in the Scikit-learn package, we can perform the same analysis with the PCA and figure out the train/test scores.

```
#3. Use LDA to choose and project data to a lower dimension, then display
    the mixture model results

lda_1 = LinearDiscriminantAnalysis()
lda_1.fit(X_train, y_train.reshape(-1,))

train_preds = lda_1.predict(X_train)
test_preds = lda_1.predict(X_test)

print('Train set accuracy:', accuracy_score(y_train, train_preds))
print('Train set f1:', f1_score(y_train, train_preds))
print('Train set ROC AUC:', roc_auc_score(y_train, train_preds))

print('Test set accuracy:', accuracy_score(y_test, test_preds))
print('Test set f1:', f1_score(y_test, test_preds))
print('Test set ROC AUC:', roc_auc_score(y_test, test_preds))

—->Output:

Train set accuracy: 0.9767441860465116
Train set f1: 0.9759036144578314
Train set ROC AUC: 0.9767158521727359
Test set accuracy: 0.6511627906976745
```



(a) First Principle Component

(b) First 16 Principle Components

Figure 5: Eigenjersey

```
Test set f1: 0.6511627906976744
Test set ROC AUC: 0.6543478260869565
```

The LDA did perform well on the training set but not so much on the test set, so it overfitted, albeit not as bad as using the original 120000 pixels. The LDA by definition found the best dimension that if we project the data down to it's best linearly distinguishable, so in this case we probably either don't have enough data or the decision boundary between the shirts and the jerseys are truly non-linear.

Similar Analysis For 400 Randomly Chosen Shirts And Jerseys

We can perform the same analysis for randomly chosen shirts and jerseys (some examples are plotted in figure 6, where we can see the data is much, much more unclean and the color is off probably because upon reshaping we mixed up the color channels). The test set accuracy for unprojected data is around 62.5%, for the PCA-projected data is around 57.5% and for the LDA-projected data is around 58.0%. The decreased performance from unprojected to projected data is probably because with there are much more unuseful variance in this unclean set of data, and when the data is projected into a lower dimension the first components contain uninformative variance. We can clearly see this in figure 7: there are so much uninformative variance that even the first principle component of the eigenshirt/jersey is blurred, which means there are probably no useful information in the most varied direction of the data.

Discussion

After the analysis, if forced to choose between these simple models we will go with the PCA-projected because of two reasons: first, we can see that there are potential for

Figure 6: Train/Test Accuracy For PCA-Projected Data Using Logistic Regression

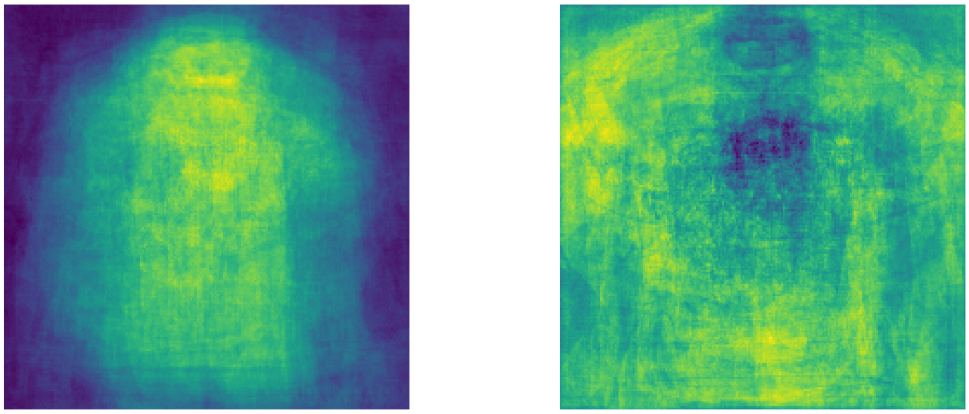
generalizable results when using different numbers of principle components, and thus this can be tuned using cross-validation making it possible for this model to get better and second, when the data is projected we can see that with only about ¡50 components the shirt/jersey is quite distinguishable with only around 200 examples, so collecting more data ought to make the model better.

Side note: I'm actually surprised that the model can perform $> 50\%$ accuracy since I can't even differentiate between a shirt and a jersey sometimes with my eyes.

RBF Linear Model

Since using data from the whole year is quite messy and hard to visualize we can either choose one day (with every-5-minute data) or the mean of the whole year for sake of visualization of the RBFs in action. The data for one date (01/25/2016) was chosen.

```
1 import numpy as np
2 import matplotlib.pyplot as plt
3 from sklearn import linear_model
4
5 years = range(2011, 2017)
6 files = ['/Users/ash/Downloads/yosemite-temperatures/yosemite_village/
    CRNS0101-05-%d-CA_Yosemite_Village_12_W.txt'
7         % y for y in years]
```



(a) Shirt (b) Jersey

Figure 7: First Principle Component - Uncleaned Data

```
usecols = [1, 2, 8]

data = [np.loadtxt(f, usecols=usecols) for f in files]
data = np.vstack(data)
# Map from HHmm to an integer
data[:, 1] = np.floor_divide(data[:, 1], 100) * 60 + np.mod(data[:, 1],
    100)
valid = (data[:, 2] > -1000) & (data[:, 0] == 20160125)
print(data.shape)

---->Output:

(631296, 3)
```

In figure 8, we are fitting a linear regression without any regularization onto the data, and we can clearly see that there is inherent non-linearity in the variability of the temperatures so a linear regression model with linear basis function is not a good choice. In figure 8, using 250 radial basis function we were able to fit the variability quite well. The MSE and MAE are clearly lower with RBFs, since we are are fitting the non-linearities of the data with every troughs. However, this is when no regularization is apply, so it runs the risk of overfit. This is clearly shown when we are using 20, 40, and so on RBFs to fit the data: the more RBFs we use, the more it fit the variability of the training data.

```
#Fitting a simple linear model with linear basis function
x_train = data[valid, 1].reshape(-1, 1)
y_train = data[valid, 2]
regr = linear_model.LinearRegression()
regr.fit(x_train, y_train)

#Fitting a simple linear model with RBFs
n_rbf = 250
```

```
rbf_centers = np.linspace(0,1435,n_rbf).reshape(-1,1)

from sklearn.metrics.pairwise import rbf_kernel
from sklearn.linear_model import Ridge

sigma = 1

rbf_X_train = rbf_kernel(x_train, rbf_centers, gamma=1 / sigma)
regr_rbf = linear_model.LinearRegression()
regr_rbf.fit(rbf_X_train, y_train)

y_true = y_train
x_true = x_train

plot_y_1 = regr.predict(x_true)

rbf_X = rbf_kernel(x_true, rbf_centers)
plot_y_2 = regr_rbf.predict(rbf_X)

plt.figure(figsize=(12,8))
plt.scatter(x_true, y_true)
plt.plot(x_true, plot_y_1)
plt.plot(x_true, plot_y_2)
plt.show()

--->Output:

Figure 8
```

```
from sklearn.metrics import mean_squared_error, mean_absolute_error

print('MSE on linear regression:', mean_squared_error(y_true, plot_y_1))
print('MSE on RBF linear regression:', mean_squared_error(y_true, plot_y_2)
    )
print('MAE on linear regression:', mean_absolute_error(y_true, plot_y_1))
print('MAE on RBF linear regression:', mean_absolute_error(y_true, plot_y_2
    ))

--->Output:

MSE on linear regression: 0.5984321706449398
MSE on RBF linear regression: 0.18659228579023052
MAE on linear regression: 0.5906459941700257
MAE on RBF linear regression: 0.1428475666963053
```

```
plt.figure(figsize=(20,12))
for _ in [2,4,6,8,10,12,14,16,18,20,22,24,]:
    n_rbf = _*10
    rbf_centers = np.linspace(0,1435,n_rbf).reshape(-1,1)
    sigma = 1
    rbf_X_train = rbf_kernel(x_train, rbf_centers, gamma=1 / sigma)
    regr_rbf = linear_model.LinearRegression()
    regr_rbf.fit(rbf_X_train, y_train)
    rbf_X = rbf_kernel(x_true, rbf_centers)
```

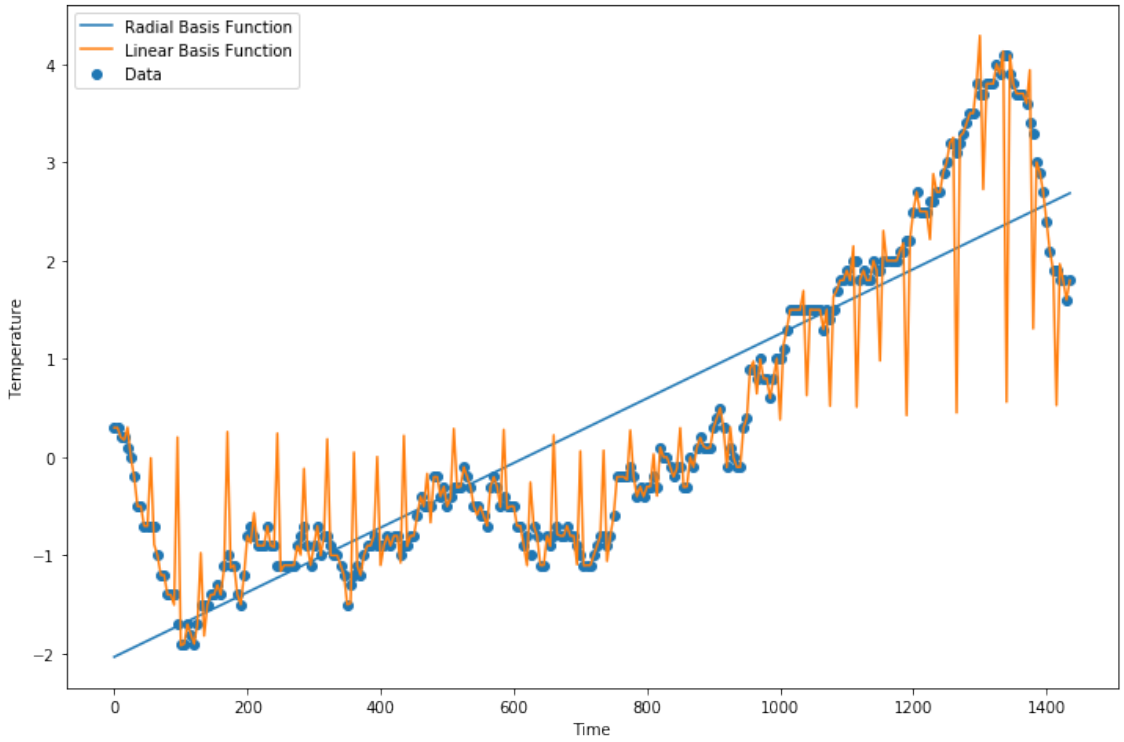


Figure 8: Fitting Linear Models In Temperature Data On 01/25/2016

```
    plot_y_2 = regr_rbf.predict(rbf_X)
    plt.subplot(4,3,_/2)
    plt.scatter(x_true, y_true)
    plt.plot(x_true, plot_y_1)
    plt.plot(x_true, plot_y_2)
    plt.title(_*20)
plt.show()
```

The overfit can be mitigated by using Lasso or Ridge regularization. In figure 10.1 we can see that using ridge regression with different shrinkage parameter α the RBFs are becoming smoother and smoother, and in 10.2 with lasso regression the increase of α increasingly selected the most prominent troughs of the data to model, using fewer and fewer RBFs, which means it is performing a kind of feature selection for the RBFs to prevent overfitting. We can also see that both regularization methods took care to not affect the MSE too much, since beside the regularization terms we are still minimizing the OLS of the data.

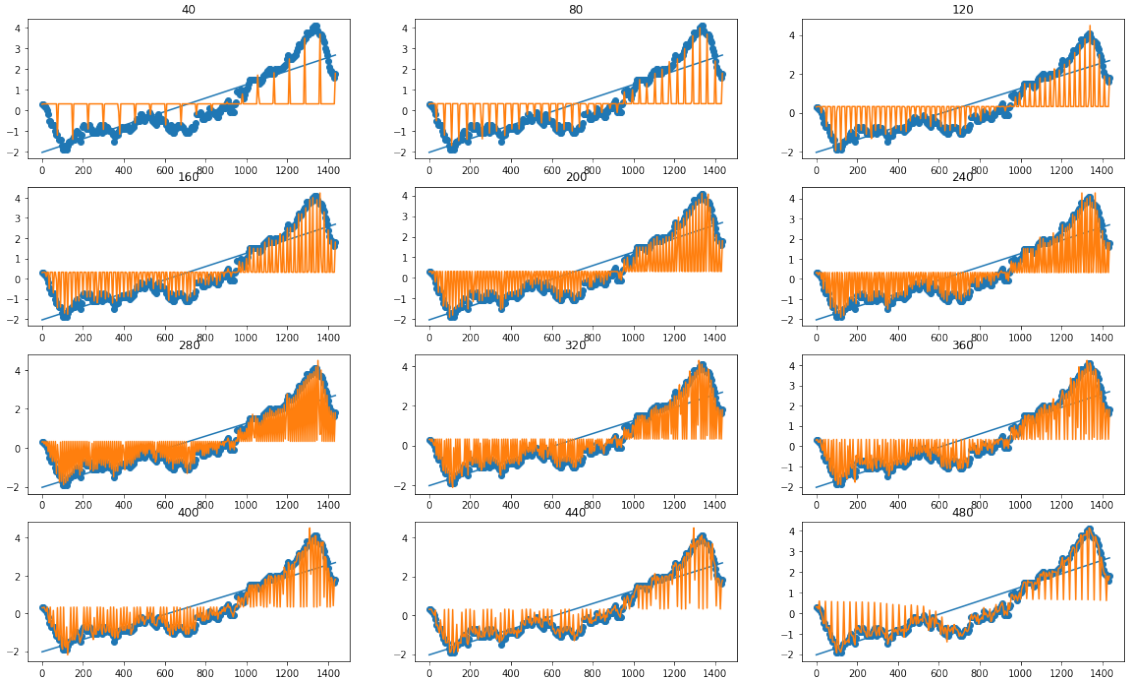
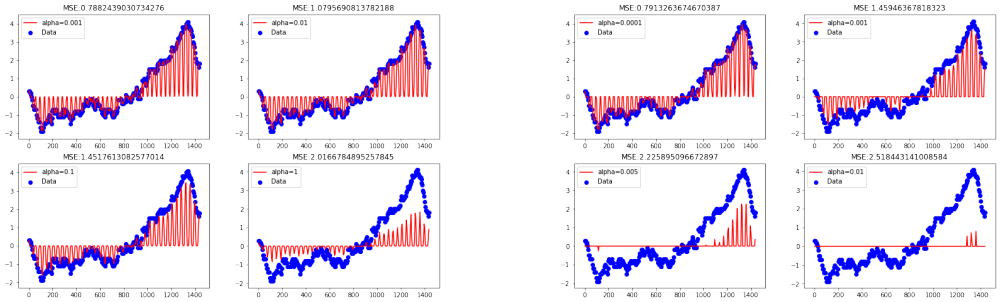


Figure 9: Varying The Number Of RBFs Used Means Overfitting The Data



(a) Ridge (b) Lasso

Figure 10: Regularization on RBFs