Homework 2

Hong Gan hg383@cornell.edu
Zhen Liu zl557@cornell.edu

September 28, 2016

1 Eigenface for face recognition

1.1 Download The Face Dataset

In this exercise, the task is implement Eigenface method for recognizing human faces. Image is from Yale Face Database B[1]. It takes use of the knowledge of Single Value Decomposition (SVD). Eigenface on Wikipedia[2] provide information about how SVD is applied in face recognition. The dataset downloaded have a set of training data and a set of testing data. In training data, there are 540 images, each has 50 * 50 pixels. I flatted the 50*50 into 2500*1 vector. So the matrix form of training data has a dimension of 540*2500. Test data has 100 image. The dimension of matrix of testing data is (100, 2500).

1.2 Display a face

The 10^{th} face in training data was chosen, reshape it to 50*50 pixels image and plot it with matplotlib imshow function. The image and the code for generate the figure:

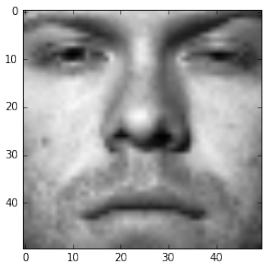


Figure 1: Face from training data

```
plt.imshow(train_data[9,:].reshape(50,50), cmap = cm.Greys_r)
```

[1] http://cs5785-cornell-tech-16fall.github.io/data/faces.zip
[2] https://en.wikipedia.org/wiki/Eigenface

1.3 Average Face

Eigenface process start from the average of all face images. So let's computed the average face first.

```
col_mean = np.mean(train_data, axis=0)
```

In this snippet, axis is set to 0, which tells the function to compute average value in each column of the training matrix. *imshow* is used to plot the average face directly should have a gray scale.

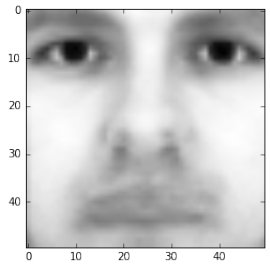


Figure 2: Average Face

1.4 Mean Subtraction

Next task is to subtract average face from every training image. The 10^{th} face image is chosen again as X and the result shows below.

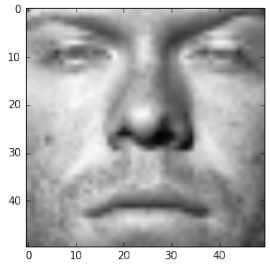


Figure 3: Mean Subtraction

The code generate the image:

```
aver = train_data - np.tile(col_mean, (540,1))
plt.imshow(aver[9,:].reshape(50,50), cmap = cm.Greys_r)
plt.show()
```

1.5 Eigenface

SVD function in numpy lib is used to perform SVD on the training set to complete the task.

```
# Eigenface
U, S, V = np.linalg.svd(train_data, full_matrices=True)
# each row of V has same dimension 2500 as the face image
print(U.shape, S.shape, V.shape)
plt.figure(figsize=(50,50))
for i in range(10):
    plt.imshow(V[i,:].reshape(50,50), cmap = cm.Greys_r)
    plt.subplot(5,2,i+1)
```

The return value V in the code is already transpose of V.

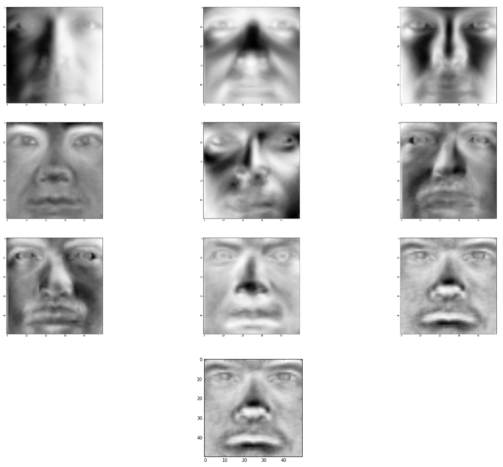


Figure 4: Mean Subtraction

1.6 Low-rank Approximation

.A little bit of intuition is as r increase, the rank-r approximation error is going to be decrease.

1.7 Eigenface Feature and Face Recognition

For this step, eigenface feature is used to perform face recognition on the original dataset. The top r eigenfaces span an r-dimensional linear subspace of the original image space called face space, whose origin is the average face , and whose axes are the eigenfaces. Therefore, using the top r eigenfaces, we can represent a 2500-dimensional face image z as an r-dimensional feature vector.

The $540 * r$ feature matrix as input and train with logistic regression and test on test data. And plot the classification accuracy with r from 1 to 200

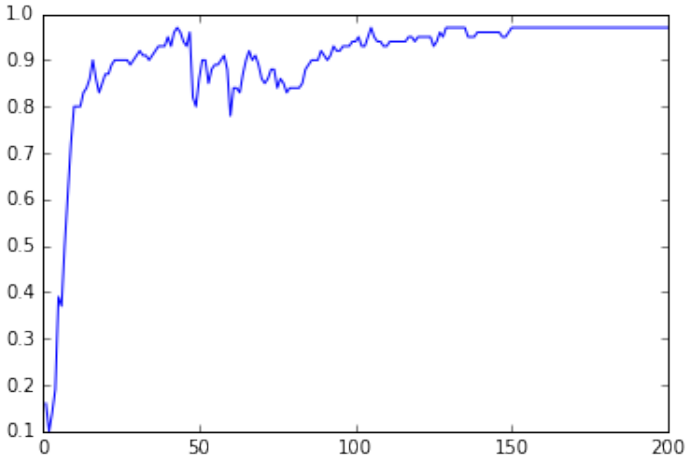


Figure 5: Classification Accuracy Plot

Code that generate the plot is attached below:

```
def subtract_avar(X):
col_mean = np.mean(X, axis=0)
return X - col_mean

def r_dimension_feature_matrix(r, X, V):
    F = np.dot(X, (V[:r,:]).T)
    return F
from sklearn import linear_model
avar = subtract_avar(train_data)
U, S, V = np.linalg.svd(avar, full_matrices=True)
x, y = [], []
for i in range(1,201):
    F_train = r_dimension_feature_matrix(i, \
                                         subtract_avar(train_data), \
                                         V)
```

```
        F_test  = r_dimension_feature_matrix(i, \
                                              subtract_avar(test_data), \
                                              V)
        logistic = linear_model.LogisticRegression()
        logistic.fit(F_train, train_labels)
        x.append(i)
        y.append(logistic.score(F_test, test_labels))
    plt.plot(x,y)
```

1.8 Summary

After doing this programming exercise, I felt completely interested in face recognition and computer vision. And I really saw the power the single value decomposition. It can compress data while maintain the most valuable information. Feature dimension has reduced from original 2500 to r. according to the result, with 200 feature dimension, the program accuracy has already reached over 95%. We really dont need to train with the whole training data. This method is great to decompose some matrix that is really big. What come to my mind is something like data like move rate or something, with the power of SVD, we can largely increase the performance of recommendation system.

2 Whats Cooking?

2.1 Download the training and test data

The data is downloaded from Kaggle[1]. It contains a training set, a testing set, and a sample submission. Training set and testing set data are *json* files. python json lib is imported to process those files.

2.2 About the data

In training data, there are totally 6714 ingredients, and totally 20 types of cuisine. A variable called *ingredientsTable* is used to keep all the unique ingredients appearing in the training set.

Also during the data processing, I realized that there are some ingredients only contain in the testing data. So that an extra table *ingredientsTestOnly* is made to store those ingredients just in case.

2.3 Represent each dish by a binary ingredient feature vector

Because we already have the *ingredientsTable* which have all unique ingredients appearing in the training set, so that we could construct a vector $n * d$ to represent the training data set. Each row has length of d represent a dish, and where $x_i = 1$ if the dish contains

[1]https://www.kaggle.com/c/whats-cooking

ingredient i and $x_i = 0$ otherwise. Total of n dishes contains in this matrix, where n is the number of dishes.

2.4 Nave Bayes Classifier

sklearn lib is imported to perform the 3 fold cross-validation on the training set, and your average classification accuracy is reported as a measurement of those algorithms.

```
all_folds = cross_validation.KFold(len(train_X), n_folds=3)

for train, test in all_folds:
    gnb = GaussianNB()
    print 'Nave Bayes Classifier, Gaussian distribution {}'.format( \
        gnb.fit(train_X[train], train_Y[train]) \
        .score(train_X[test], train_Y[test]))
    bnb = BernoulliNB()
    print 'Nave Bayes Classifier, Bernoulli distribution {}'.format( \
        bnb.fit(train_X[train], train_Y[train]) \
        .score(train_X[test], train_Y[test]))
```

2.5 Result

From the result, we could see that Bernoulli prior report better in terms of cross-validation accuracy as almost as twice much as Gaussian prior.

```
=============Run 1=============
Nave Bayes Classifier, Gaussian distribution 0.379016442902
Nave Bayes Classifier, Bernoulli distribution 0.684190677327
===========Run 1 End===========
=============Run 2=============
Nave Bayes Classifier, Gaussian distribution 0.382938603108
Nave Bayes Classifier, Bernoulli distribution 0.679514255544
===========Run 2 End===========
=============Run 3=============
Nave Bayes Classifier, Gaussian distribution 0.377583345904
Nave Bayes Classifier, Bernoulli distribution 0.686906019007
===========Run 3 End===========
```

The differences in the result might caused by the characteristic of those two different type of distributions. For the Gaussian distribution, the probability density reach its highest level in the middle of the the range, and Bernoulli distribution have it probability density in either 0 or 1, not anywhere between them. Because in our task, the X value is either 0 or 1, so that Bernoulli distribution would fit the situation better. And result supported the idea.

2.6 Logistic Regression Model

For the Logistic Regression Model, sklearn lib is used to perform 3 fold cross-validation on the training set.

```
for train, test in all_folds:
print 'Logistic Regression Model {}'.format( \
linear_model.LogisticRegression() \
        .fit(train_X[train], train_Y[train]) \
        .score(train_X[test], train_Y[test]))
```

The result is reported as below

```
Logistic Regression Model 0.775833459044
Logistic Regression Model 0.772137577312
Logistic Regression Model 0.77869965304
```

For the result, we could see that Logistic Regression Model report even slightly better than the Nave Bayes Classifier with Bernoulli distribution. Among all 3 algorithm performed on this dataset, Logistic Regression Model provided the best result in cross-validation.

2.7 Submit results to Kaggle and Summary

After did all tasks above, logistic regression model is decided to use as the model for final submission. After trained with the entire training dataset, test dataset is used to make the prediction, and output formed with related data ID *test_Y*. The final file data is csv, and Kaggle reported a score of 0.78339 as a result.

One thing is noticeable is that there are numbers of ingredients that either basically the same thing but with different name, or they have different name but very close to each other. A idea to improve the overall performance of the result is to reduce the duplicate of ingredients with different names.

3 Written Exercise

3.1 HTF EX 4.1

From Generalized to Standard Eigenvalue Problem To max $a^T Ba$ subject to $a^T Wa = 1$ Use Lagrange Multiplier,

$$L(\lambda) = a^T Ba - \lambda(a^T Wa - 1)$$

Take derivative at a,

$$\frac{\partial L}{\partial a} = (B + B^T)a - \lambda(W + W^T)a = 0$$
$$\lambda(W + W^T)a = (B + B^T)a$$
$$(W + W^T)^{-1}(B + B^T)a = \lambda a$$

so this has turn into standard Eigenvalue problem: has form as Aa=λa

3.2 HTF EX 4.2

(a) we have linear discriminant function as follow:

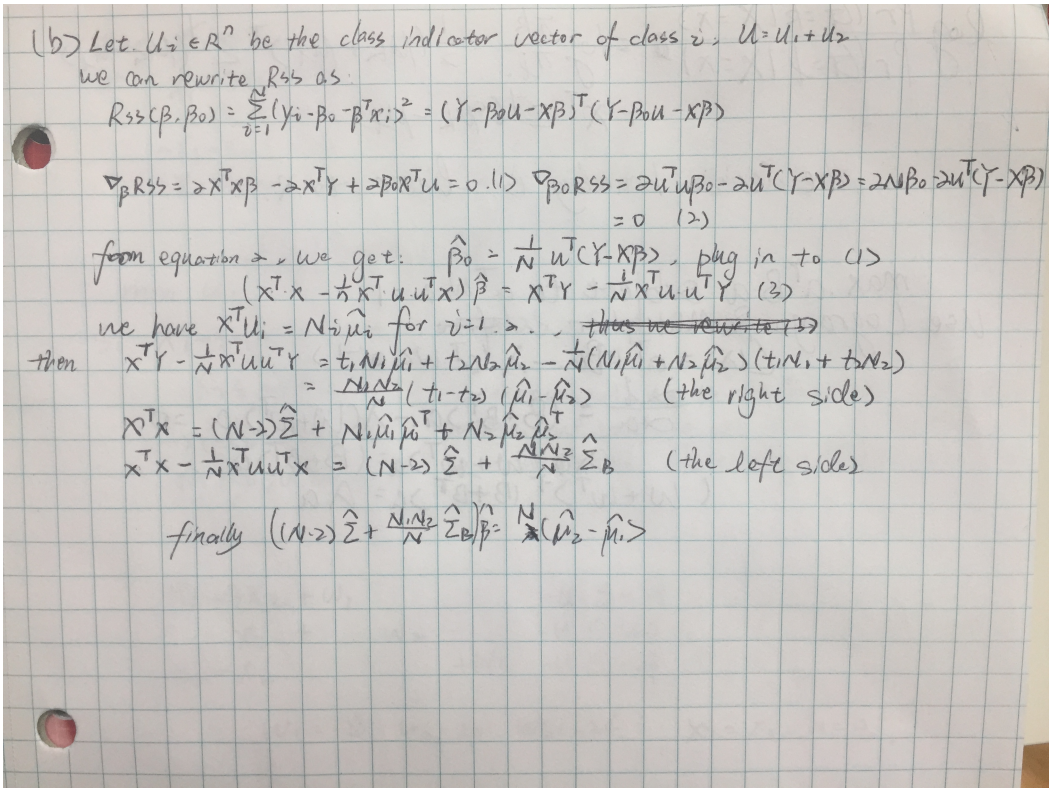
$$\delta_k(x) = x^T\Sigma^{-1}\mu_k - \frac{1}{2}\mu_k^T\Sigma^{-1}\mu_k + log\pi_k$$

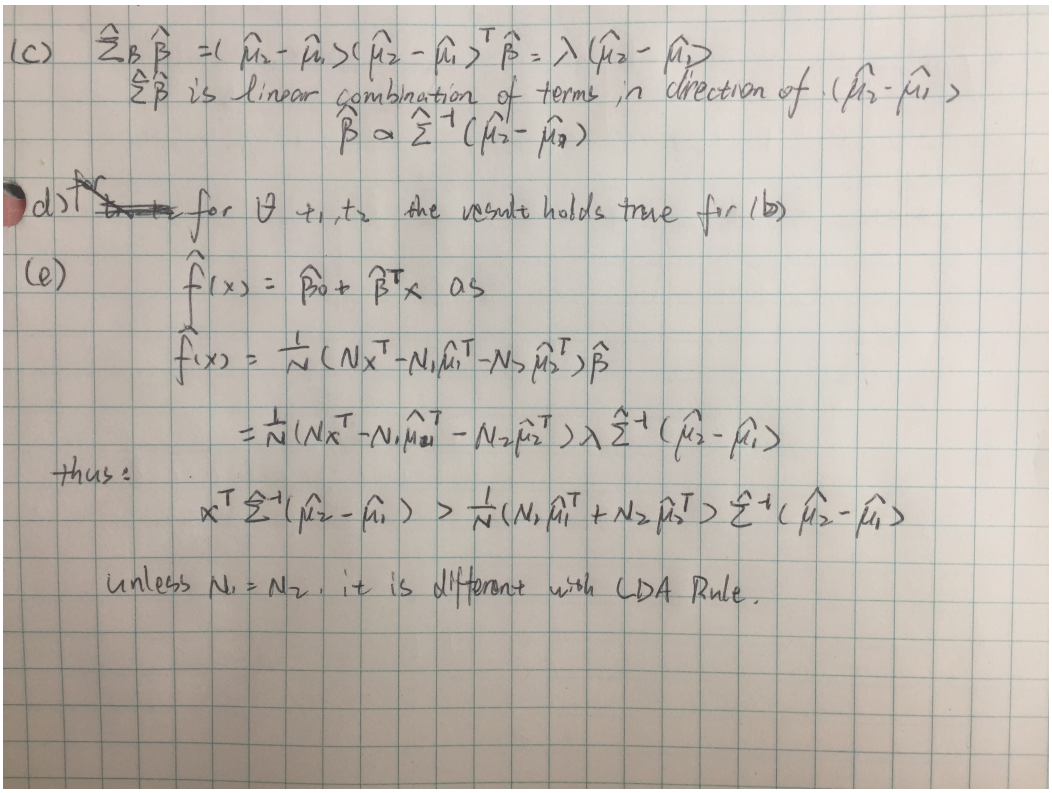
we classify to class two, which requires:

$$\delta_1(x) < \delta_2(x)$$

then we have:

$$x^T\hat{\Sigma}^{-1}(\hat{\mu_1} - \hat{\mu_2}) > \frac{1}{2}(\hat{\mu_1} + \hat{\mu_2})^T\hat{\Sigma}^{-1}(\hat{\mu_1} - \hat{\mu_2}) - log(N_2/N_1)$$





3.3 RLU EX 11.3.1

Please see the attached ipython code / result in the following pages.

Written 3

September 28, 2016

In [1]:
```
# Due to a problem with printing out the matrix,
# I switched from numpy to sympy as the following thread suggested.
# (Can't get accurate number due to the 17 digits limitation)
# stackoverflow.com/questions/22865245/numpy-possible-better-formatting
import sympy as sy
import numpy as np
sy.init_printing(use_latex='mathjax')
```

Exercise 11.3.1 : In Fig. 11.11 is a matrix M. It has rank 2, as you can see by observing that the first column plus the third column minus twice the second column equals 0.

In [2]:
```
M = sy.Matrix([[1, 2, 3], [3, 4, 5], [5, 4, 3], [0, 2, 4],[1, 3, 5]])
```

(a) Compute the matrices MTM and MMT

In [4]:
```
# for M^TM:
MTM = M.transpose()*M
MTM
```

Out[4]:

$$\begin{bmatrix}36 & 37 & 38\\37 & 49 & 61\\38 & 61 & 84\end{bmatrix}$$

In [5]:
```
# for MM^T
MMT = M*M.transpose()
MMT
```

Out[5]:

$$\begin{bmatrix}14 & 26 & 22 & 16 & 22\\26 & 50 & 46 & 28 & 40\\22 & 46 & 50 & 20 & 32\\16 & 28 & 20 & 20 & 26\\22 & 40 & 32 & 26 & 35\end{bmatrix}$$

(b) Find the eigenvalues for your matrices of part (a).

In [33]:
```
# eigenvalues
MTM.eigenvals()
```

Out[33]:

$$\left\{0 : 1, \quad -\frac{\sqrt{19081}}{2} + \frac{169}{2} : 1, \quad \frac{\sqrt{19081}}{2} + \frac{169}{2} : 1\right\}$$

In [34]:
```
# eigenvector
MMT.eigenvals()
```

Out[34]:

$$\left\{0 : 3, \quad -\frac{\sqrt{19081}}{2} + \frac{169}{2} : 1, \quad \frac{\sqrt{19081}}{2} + \frac{169}{2} : 1\right\}$$

(c) Find the eigenvectors for the matrices of part (a).

In [35]:
```
MTM.eigenvects()
```

Out[35]:

$$\left[\left(0, \quad 1, \quad \left[\begin{bmatrix}1\\-2\\1\end{bmatrix}\right]\right), \quad \left(-\frac{\sqrt{19081}}{2} + \frac{169}{2}, \quad 1, \quad \left[\begin{bmatrix}-\frac{38}{-\frac{97}{2}+\frac{\sqrt{19081}}{2}} + \frac{-\frac{52022}{-\frac{97}{2}+\frac{\sqrt{19081}}{2}}+2257}{\left(-\frac{304471}{4836}+\frac{1049\sqrt{19081}}{4836}\right)\left(-\frac{97}{2}+\frac{\sqrt{19081}}{2}\right)}\\ -\frac{\frac{1406}{-\frac{97}{2}+\frac{\sqrt{19081}}{2}}+61}{-\frac{304471}{4836}+\frac{1049\sqrt{19081}}{4836}}\\ 1\end{bmatrix}\right]\right), \right.$$

In [43]:
```
# Ehhhhhhh...
# Let's try something else..
# ( First two numbers are eigenvalue, last col is eigenvector)
# http://docs.sympy.org/latest/modules/evalf.html
# Calculate them as number instead of equation
# With only eigenvector [2]. [0] and [1] are eigenvalues, skip
temp = MTM.eigenvects()
eigvectMTM = []
for x in temp:
    eigvectMTM.append(x[2][0].evalf()/x[2][0].evalf().norm())
```

In [44]:
```
eigvectMTM
```

Out[44]:

$$\left[\begin{bmatrix}0.408248290463863\\-0.816496580927726\\0.408248290463863\end{bmatrix}, \quad \begin{bmatrix}-0.815978481555022\\-0.12588456422607\\0.564209353102882\end{bmatrix}, \quad \begin{bmatrix}0.409282849594866\\0.56345932401811\\0.717635798441355\end{bmatrix}\right]$$

```
In [46]: # Better, try MM^T
         temp = MMT.eigenvects()
         eigvectMMT = []
         for x in temp:
             eigvectMMT.append(x[2][0].evalf()/x[2][0].evalf().norm())

In [47]: eigvectMMT
```

Out[47]:

$$\left[\begin{bmatrix} 0.784464540552736 \\ -0.588348405414552 \\ 0.196116135138184 \\ 0 \\ 0 \end{bmatrix}, \begin{bmatrix} 0.159063930284883 \\ -0.0332003042935722 \\ -0.73585663402025 \\ 0.510392095148223 \\ 0.414259977858995 \end{bmatrix}, \begin{bmatrix} 0.297695678025794 \\ 0.570508561088988 \\ 0.520742971163787 \\ 0.322578472988394 \\ 0.458984914519991 \end{bmatrix}\right]$$

(d) Find the SVD for the original matrix M from parts (b) and (c). Note that there are only two nonzero eigenvalues, so your matrix Σ should have only two singular values, while U and V have only two columns.

```
In [93]: # Nonzero eigenvalues only
         # U = MM^T
         U = eigvectMMT[2].row_join(eigvectMMT[1])
         # V = M^TM
         V = eigvectMTM[2].row_join(eigvectMTM[1])
```

Becuase according to (11.6) $M^TMV = V\Sigma^2$, so that to get Σ, Σ is the diagonal matrix whose entries are the corresponding eigenvalues, then $M^TM = \Sigma^2$, thus $\Sigma = \sqrt{M^TM}$

```
In [128]: temp = MTM.eigenvects()
          r = sy.diag((temp[2][0]**0.5).evalf(), (temp[1][0]**0.5).evalf())

In [129]: U*r*V.transpose()
```

Out[129]:

$$\begin{bmatrix} 1.0 & 2.0 & 3.0 \\ 3.0 & 4.0 & 5.0 \\ 5.0 & 4.0 & 3.0 \\ 4.44089209850063 \cdot 10^{-16} & 2.0 & 4.0 \\ 1.0 & 3.0 & 5.0 \end{bmatrix}$$

(e) Set your smaller singular value to 0 and compute the one-dimensional approximation to the matrix M from Fig. 11.11.

```
In [130]: r_e = r
          r_e[3] = 0

In [131]: r_e
```

Out[131]:

$$\begin{bmatrix} 12.3922151554901 & 0 \\ 0 & 0 \end{bmatrix}$$

In [132]:
```
U*r_e*V.transpose()
```

Out[132]:

$$\begin{bmatrix} 1.50988900017444 & 2.07866280305371 & 2.64743660593297 \\ 2.89357442676402 & 3.98358126199138 & 5.07358809721875 \\ 2.64116727959428 & 3.63609257375932 & 4.63101786792435 \\ 1.63609257375932 & 2.25240714716974 & 2.86872172058017 \\ 2.3279352870541 & 3.20486637663858 & 4.08179746622305 \end{bmatrix}$$

(f) How much of the energy of the original singular values is retained by the one-dimensional approximation?

$$\frac{12.3922151554901^2}{12.3922151554901^2+3.92848616391112^2}$$

In [146]:
```
r = sy.diag((temp[2][0]**0.5).evalf(), (temp[1][0]**0.5).evalf())
r[0]**2/float(r[0]**2+r[3]**2)
```

Out[146]:

$$0.908680452425793$$