CS5785
Applied Machine Learning
HW2
Due Date: 09/28/2016

Siyadong Xiong
sx225@cornell.edu
Teammates: N/A

1 Programming Exercises

1.1 Eigenface for recognition

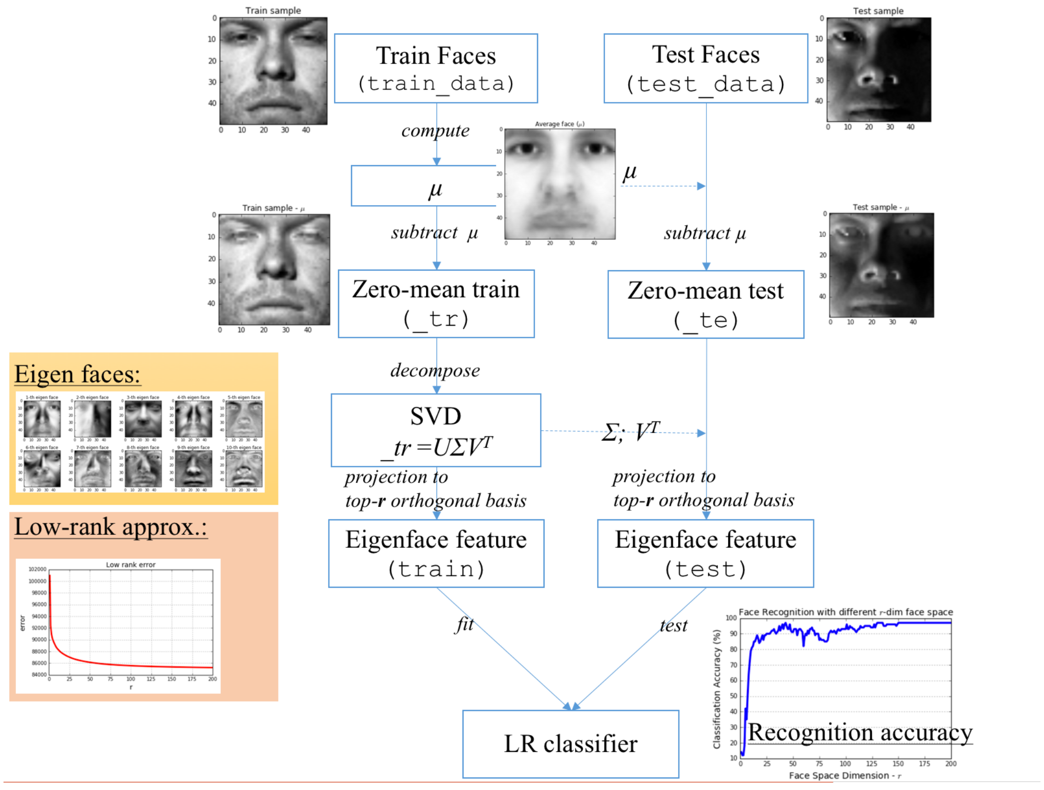


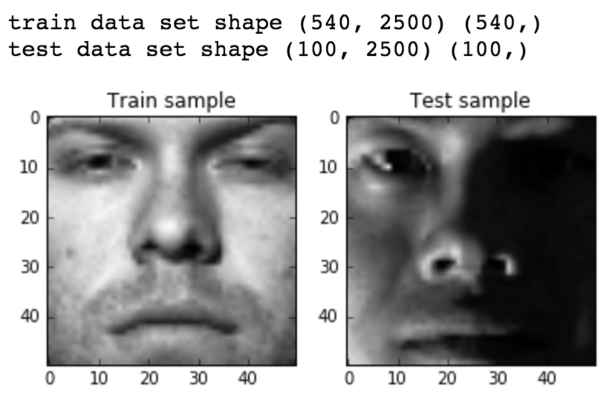
Figure 1: Eigenface for recognition flow chart.

Fig.1 demonstrated the entire work flow of this task and some quick results. Details are as follows.

1.1.1 Imported Libraries & Load face data

As usual, we firstly imported Python scientific/vis libraries including NumPy, Scipy, matplotlib, and sklearn. Then we loaded face data with help of the given code in the assignment description. We afterwards showed samples of both train and test sets in Fig.2.

```python
%matplotlib inline
from scipy import misc
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.cm as cm
from sklearn.linear_model import LogisticRegression

...

plt.subplot(1, 2, 1)
plt.imshow(train_data[10, :].reshape(50,50), cmap = cm.Greys_r)
plt.title('Train sample')
plt.subplot(1, 2, 2)
plt.imshow(test_data[10, :].reshape(50,50), cmap = cm.Greys_r)
plt.title('Test sample')
plt.show()
```



```
train data set shape (540, 2500) (540,)
test data set shape (100, 2500) (100,)
```

Figure 2: Face samples.

1.1.2 Zero-mean faces, SVD and Eigenfaces

In my opinion, the reason why we need to subtract mean at every dimension is that as we would project the face data to lower-dimensional space, we need to find a set of orthogonal basis on which the face data have most energy. The distance to mean is used to measure how many energy lying on one direction. We used

```
mu = np.apply_along_axis(np.mean, 0, train_data)
plt.imshow(mu.reshape(50,50), cmap = cm.Greys_r)
plt.title('Average face ($\mu$)')
plt.show()

_tr = train_data - mu
_te = test_data - mu
```

to demonstrate the average face (in Fig.3) and get the deviation matrix of train and test to the average face (in Fig.4). Fig.

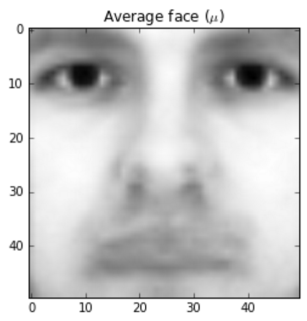


Figure 3: *mu* face.

We then conducted SVD decomposition on zero-mean train set and showed top-10 singular values eigenfaces and least-5 ones in Fig.5. Obviously, eigenfaces with high singular values contained more information while lower ones contained less.

```
u, s, v = np.linalg.svd(_tr)
```

1.1.3 Low rank approximation

Since singular vectors with very small singular values contained less energy on its direction, we could simply abandon these directions, i.e., approximate the original image with lower rank U, Σ, and V^T. We use

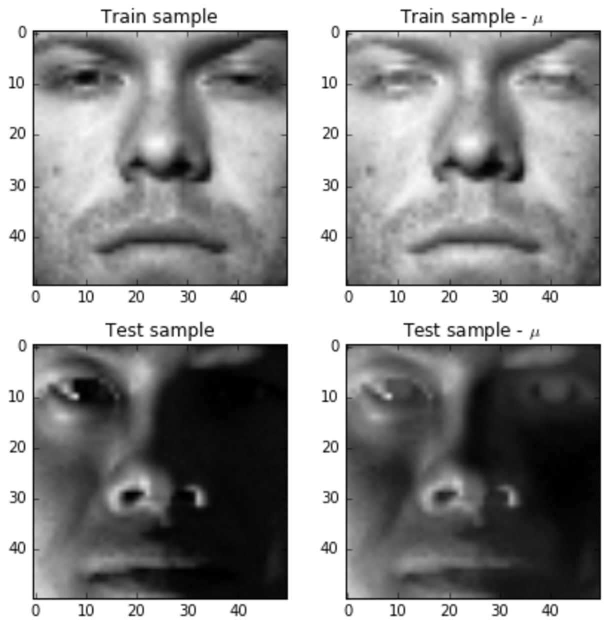


Figure 4: Sample faces subtracted by mu.

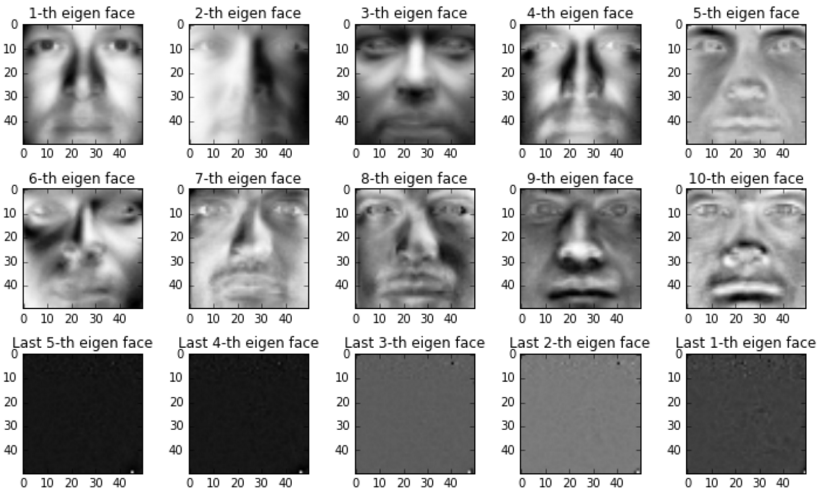


Figure 5: Eigenfaces.

```python
def gen_Xr(r):
    return u[:, :r].dot(np.diag(s[:r]).dot(v[:r, :]))

low_rank_err = []
for r in xrange(1, 201):
    low_rank_err.append(np.linalg.norm(gen_Xr(r) - train_data,
        ord='fro'))
```

to generate lower rank approximations and compute their Frobenius Norm to original train set. Low rank errors are shown in Fig.6. The error descend fast with rather a small subset of singular vectors.

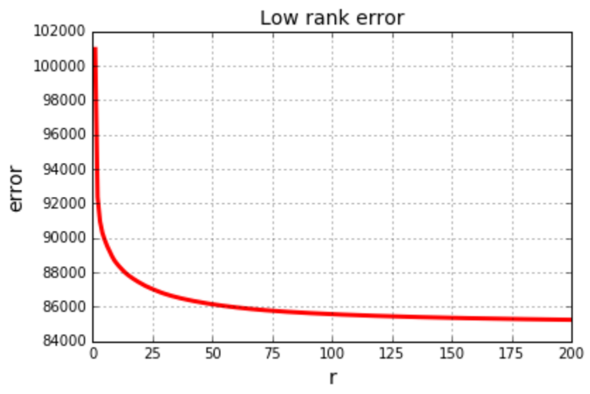


Figure 6: Low rank error with different r.

1.1.4 Eigenface features and classification

As Sec.1.1.3 mentioned, the error is quite small with image projected to a rather small subset of singular vectors. This leads to the idea that we could use it to reduce dimensions when conducting classification. Projecting original images to top-r singular vectors, we comprise the eigenface features.

```python
def gen_eigenFace_feature(V, X, r=1):
    """
        V: p * p
        X: N * p
        r: rank
    """
    return X.dot(V[:r, :].T)

```

```python
def get_eigen_face_feature(v, train, test, r=1):
    return gen_eigenFace_feature(v, train, r),\
                gen_eigenFace_feature(v, test, r)

accu = []
for r in xrange(1, 201):
    tr, te = get_eigen_face_feature(v, _tr, _te, r)
    cls = LogisticRegression()
    cls.fit(tr, train_labels)
    accu.append(cls.score(te, test_labels))
```

Picking different value of r, we fit LR models with projected features and test the classifier on test set. The result is shown in Fig.7. With r above 30, we reached acceptable classification (recognition) results. This means that we could reduced the original 2500 dimensional features to 30 dimensions.

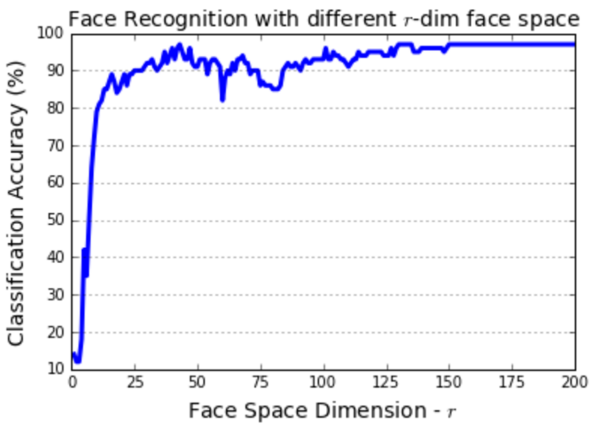


Figure 7: Classification accuracy with different r.

1.2 What's cooking

1.2.1 Imported libraries

In this section, we used the following scientific libraries.

```python
%matplotlib inline
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import matplotlib.cm as cm
```

```

from sklearn.linear_model import LogisticRegression
from sklearn.naive_bayes import GaussianNB, BernoulliNB
from sklearn.feature_extraction.text import CountVectorizer
from sklearn.preprocessing import LabelEncoder
```

1.2.2 Data set properties

Reading from json to DataFrame and analyzing the basic properties of train set,

```
train_data = pd.read_json("train.json")
print "[INFO] N is %d." % train_data.shape[0]

from itertools import chain
all_igdt = set(chain.from_iterable(train_data.ingredients))
print "[INFO] In total %d ingredients." % len(all_igdt)
```

we know that there are 39774 dishes and 6714 ingredients in total.

```
[INFO] N is 39774.
[INFO] In total 6714 ingredients.
```

Using `LabelEncoder` to encode labels,

```
_labels = train_data.cuisine
label_enc = LabelEncoder()
y = label_enc.fit_transform(_labels)

assert len(label_enc.classes_) == len(set(_labels))
assert y.shape[0] == train_data.shape[0]

print "[INFO] In total %d labels" % len(label_enc.classes_)
print label_enc.classes_
```

we know that there are 20 types of cuisine in total, including

```
[INFO] In total 20 labels
[u'brazilian' u'british' u'cajun_creole' u'chinese'
   u'filipino' u'french', u'greek' u'indian' u'irish'
   u'italian' u'jamaican' u'japanese' u'korean', u'mexican'
   u'moroccan' u'russian' u'southern_us' u'spanish' u'thai',
u'vietnamese']
```

1.2.3 Ingredients feature vectorization

We use `CountVectorizer` to encode ingredients feature. It is a bit ugly when vectorizing, but I did not come up with more elegant approaches. It is noteworthy that we cannot simply use comma to separate different ingredients since there exist commas in ingredients strings.

```
_igdt_str_list = map(lambda r: "sepearate".join(r),
    train_data.ingredients)
assert len(_igdt_str_list) == train_data.shape[0]

enc = CountVectorizer(vocabulary=all_igdt, tokenizer=lambda x
    : x.split('sepearate'))
X = enc.fit_transform(_igdt_str_list)

assert X.shape == (train_data.shape[0], len(all_igdt))
```

The transformed feature matrix X is represented by sparse matrix (csr) which makes sense.

1.2.4 Classification: Gaussian Naïve Bayes, Bernoulli Naïve Bayes and LR

We respectively utilized Naïve Bayes under Gaussian and Bernoulli prior assumption, and Logistic Regression models to do classification.

```
tr = []
te = []
for tr_idx, te_idx in cross_validation.KFold(X.shape[0],
    n_folds=3):
    tr.append(tr_idx)
    te.append(te_idx)

cls_gaussian_nb = GaussianNB()
cls_gaussian_nb.fit(X[tr[0]].toarray(), y[tr[0]])
print cls_gaussian_nb.score(X[te[0]].toarray(), y[te[0]])
cls_bernoulli_nb = BernoulliNB()
cls_bernoulli_nb.fit(X[tr[0]], y[tr[0]])
print cls_bernoulli_nb.score(X[te[0]], y[te[0]])
```

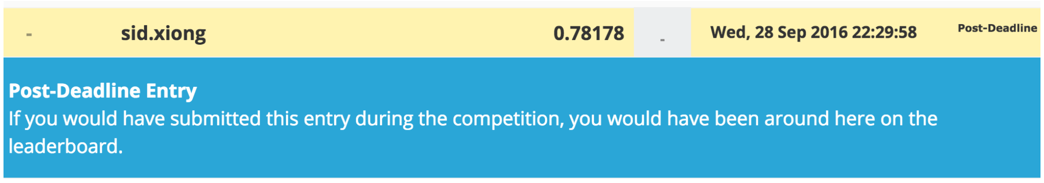
```python
cls_lr = LogisticRegression()
cls_lr.fit(X[tr[0]], y[tr[0]])
print cls_lr.score(X[te[0]], y[te[0]])

....
```

The classification results for Gaussian-NB, Bernoulli-NB, and LR are $\sim$ 37%, $\sim$ 70%, and $\sim$ 78%, respectively. Details are shown in `.ipynb`. Compared Gaussian-NB with Bernoulli-NB, the latter performs much better. The reason is apparent: we vectorize the ingredients features using binary encoding, which means that for each feature, there are only two status, i.e., 1 and 0, indicating whether it exists. Hence, Bernoulli prior assumption is more reasonable than Gaussian one.

Naïve Bayes models, however, assume that features are independent so that their joint pdf are 0. It doesn't make sense in this problem because intuitively speaking in every cuisine the ingredients have rather high correlations. We observed that LR model performs better than Bernoulli-NB, which verifies our interpretation.

1.2.5 Submit to Kaggle

We utilized the LR model and get the predictions on test set, and then submitted to Kaggle, as shown in Fig.8.



- sid.xiong 0.78178 - Wed, 28 Sep 2016 22:29:58 Post-Deadline

Post-Deadline Entry
If you would have submitted this entry during the competition, you would have been around here on the leaderboard.

Figure 8: Submission to Kaggle result.

2 Written Exercises

HTF 4.1

From Generalized to Standard Eigenvalue Problem Max/min optimization problem subject to equality constraints could be solved by Lagrange multiplier[1]. We formulate the problem as

$$\begin{aligned} &\underset{a}{\operatorname{argmax}} f(a) \\ &s.t. \\ &h(a) = a^T \mathbf{W} a - 1 = 0, \end{aligned} \tag{1}$$

where $f(a) = a^T \mathbf{B} a$. Hence, the $\mathcal{L}$ is

$$\mathcal{L}(a, \lambda) = a^T \mathbf{B} a + \lambda (a^T \mathbf{W} a - 1), \tag{2}$$

where λ is the Lagrange multiplier. Setting the derivative of $\mathcal{L}$ over a, which is given as

$$\frac{\partial \mathcal{L}}{\partial a} = 2\mathbf{B}a + 2\lambda \mathbf{W} a = 0, \tag{3}$$

to 0, we could derive that

$$\mathbf{W}^{-1}\mathbf{B}a = \lambda a. \tag{4}$$

This indicates that the optimal a is the eigen vector of $\mathbf{W}^{-1}\mathbf{B}$ with eigenvalue of λ.

HTF 4.2

(a) We could rewrite the condition to

$$log(\frac{N_2}{N_1}) - \frac{1}{2}\left(\hat{\mu_2} + \hat{\mu_1}\right)^T \hat{\Sigma}^{-1}\left(\hat{\mu_2} - \hat{\mu_1}\right) + x^T \hat{\Sigma}^{-1}\left(\hat{\mu_2} - \hat{\mu_1}\right) > 0. \tag{5}$$

Apparently the left side of Eq.5 is literally the *log-odds* in LDA, given that

$$\frac{\pi_{y_2}}{\pi_{y_1}} = \frac{N_2}{N_1} \tag{6}$$

based on given target encoding, and $\hat{\mu}$ is the sample mean that is approximate to the mean of corresponding random variable, i.e., μ. Hence, we have

$$log\frac{Pr(G = y_2 | X = x)}{Pr(G = y_1 | X = x)} > 0, \tag{7}$$

which will rule classifies to class 2.

(b) If we extend x with its interception, we can rewrite the least square criterion to matrix format, as

$$f(\beta) = (y - \mathbf{X}\beta)^T (y - \mathbf{X}\beta). \tag{8}$$

Taking f's gradient on β and setting it to 0, as

$$\frac{\partial f}{\partial \beta} = 2\mathbf{X}^T (\mathbf{y} - \mathbf{X}\beta) = 0, \tag{9}$$

we could get the optimal β. So,

$$\mathbf{X}^T\mathbf{X}\beta = \mathbf{X}^T\mathbf{y}. \tag{10}$$

It is noteworthy that x and y now have $p+1$ dimensions since we put interception in them, but we will focus on original p dimensions.

The right side of Eq.10 can be derived as

$$\mathbf{X}^T\mathbf{y} = \begin{bmatrix} \sum_{i=1}^{N} y_i \\ \sum_{i=1}^{N} y_i\mathbf{x}_i \end{bmatrix}, \tag{11}$$

in which

$$\begin{aligned} \sum_{i=1}^{N} y_i\mathbf{x}_i &= -\frac{N}{N_1}(\sum_{y_i=y_1} \mathbf{x}_i) + \frac{N}{N_2}(\sum_{y_i=y_2} \mathbf{x}_i) \\ &= N(\hat{\mu}_2 - \hat{\mu}_1) \end{aligned} \tag{12}$$

The coefficient of the left side of Eq.10 can be derived as

$$\mathbf{X}^T\mathbf{X} = \begin{bmatrix} 1 & \sum_i \mathbf{x}_i^T \\ \sum_i \mathbf{x}_i & \mathbf{X}_p^T\mathbf{X}_p \end{bmatrix}. \tag{13}$$

We only care about the right-down corner $p * p$ matrix.

Based on page 109 of book *HTF*, we get that

$$\begin{aligned} (N-2)\hat{\Sigma} &= \sum_{k\in\{y_1,y_2\}} \sum_{y_i=k} \left(\mathbf{x}_i^T\mathbf{x}_i - 2\mu_1^T\mathbf{x}_i + \mu_1^T\mu_1\right) \\ &= \sum_{y_i=y_1} \mathbf{x}_i^T\mathbf{x}_i + \sum_{y_i=y_2} \mathbf{x}_i^T\mathbf{x}_i - 2N_1\mu_1^T\mu_1 - 2N_2\mu_2^T\mu_2 + N_1\mu_1^T\mu_1 + N_2\mu_2^T\mu_2. \end{aligned} \tag{14}$$

Hence,

$$\begin{aligned}(N-2)\hat{\Sigma}+N\hat{\Sigma}_B &= \sum_{y_i=y_1}\mathbf{x}_i^T\mathbf{x}_i+\sum_{y_i=y_2}\mathbf{x}_i^T\mathbf{x}_i-N_1\mu_1^T\mu_1-N_2\mu_2^T\mu_2+\frac{N_1N_2}{N}\left(\mu_1^T\mu_1+\mu_2^T\mu_2-2\mu_1^T\mu_2\right)\\ &= (I\ give\ up).\end{aligned} \tag{15}$$

rest parts Solutions could be found on the Internet[2].

RLU 11.3.1

$$\mathbf{M}=\begin{bmatrix}1&2&3\\3&4&5\\5&4&3\\0&2&4\\1&3&5\end{bmatrix}$$

Compute M^TM and MM^T

$$M^TM=\begin{bmatrix}14&26&22&16&22\\26&50&46&28&40\\22&46&50&20&32\\16&28&20&20&26\\22&40&32&26&35\end{bmatrix},\ MM^T=\begin{bmatrix}36&37&38\\37&49&61\\38&61&84\end{bmatrix}. \tag{16}$$

Compute eigenvalues and eigenvectors of M^TM and MM^T As $rank(M)=2$, there are only two eigenvalues for M^TM and MM^T.

For M^TM, the eigenvalues are $\{153.57, 15.43\}$, respectively, and the corresponding eigenvectors are

$$\begin{aligned}v_1^1&=[-.41,-.56,-.72]^T\\ v_2^1&=[-.82,-.13,.56]^T.\end{aligned}$$

For MM^T, the eigenvalues are $\{153.57, 15.43\}$, respectively, and the eigenvectors are

$$\begin{aligned}v_1^2&=[.30,.57,.52,.32,.46]^T\\ v_2^2&=[-.16,.03,.73,-.51,-.41]^T.\end{aligned}$$

Obviously, eigenvalues of these two square matrix are identical.

SVD For SVD decomposition

$$\mathbf{M} = \mathbf{U}\Sigma\mathbf{V}^T, \tag{17}$$

$\mathbf{U}$ consists of MM^T's eigenvectors and $\mathbf{V}$ consists of M^TM's eigenvectors. Σ is a diagonal matrix with every element as the square root of eigenvalues. Therefore, we could get the SVD result as

$$\begin{aligned} U &= [v_1^2, v_2^2] \\ \Sigma &= diag\{\sqrt{153.57}, \sqrt{15.43}\} \\ V &= [v_1^1, v_2^1]. \end{aligned}$$

One-dim approx of M We could compute the one-dimensional approximation of M if we only retain the largest eigenvalue.

$$\begin{aligned} M' &= U[:, :1]\sqrt{153.57}\; V^T[:1, :] \\ &= \begin{bmatrix} 1.60841961304 & 0.248137918654 & -1.11214377573 \\ -0.335714203019 & -0.0517920963687 & 0.232130010281 \\ -7.4408210612 & -1.14792796372 & 5.14496513376 \\ 5.16097304213 & 0.796205852332 & -3.56856133747 \\ 4.1889061341 & 0.64624084482 & -2.89642444446 \end{bmatrix}. \end{aligned}$$

Engergy Eigenvalues indicate the energy distribution on eigenvectors. So the proportion of energy retained is given by

$$\alpha = \frac{153.57}{153.57 + 15.43} = 90.87\%. \tag{18}$$

3 Attachments Details

`hw2_face.ipynb` and `hw2_cooking.ipynb` in `hw2_sidxiong.zip`

References

[1] *https://en.wikipedia.org/wiki/Lagrange_multiplier*

[2] *http://waxworksmath.com/Authors/G_M/Hastie/WriteUp/weatherwax_epstein_hastie_solutio*