Homework 2

By: Daniel Speiser and Gideon Glass

September 29, 2015

Contents

1 Part 1: Eigenface for Face Recognition

1.1 Summary

The goal was to implement the Eigenface method for human facial recognition. Using images from the Yale Face database (detailed below), we explored the power of Single Value Decomposition (SVD) and trained a Logistic Regression model to predict the classification of human faces presented from a test data set,

1.2 Data Set

The Yale Face Database contained:

- 640 face images
- 64 images under different lighting conditions per each 10 distinct subjects

1.3 Solution Steps

A. The data was downloaded from the Yale Face Database in ZIP format:

   - images folder - contains all the images for the test and training data
   - train.txt - each line gives an image path and the corresponding subject label
   - test.txt - each line gives an image path and the corresponding subject label

B. Once downloaded, the training data was parsed into matrix X (Fig. 1), a 540 x 2500 dimensional matrix in which each row represents a "flattened" face image of the 540 training images. The test data was parsed in a similar manner, resulting in a matrix of size 100 x 2500.

   A sample image from the train data is displayed below (Fig. 2):

```
def parse_data(data_set_path):
    data, labels = [], []
    for line in open(data_set_path):
        im = misc.imread(line.strip().split()[0])
        data.append(im.reshape(2500,))
        labels.append(line.strip().split()[1])
    data, labels = np.array(data, dtype=float), np.array(labels, dtype=int)
    return (data, labels)
```

Figure 1: 'parse_data' Function

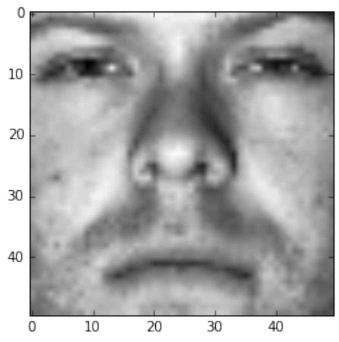


Figure 2: Sample image from train data

C. The *average face* μ was computed from the whole training set by summing up every column in X and dividing by the total number of faces (fig. 3). The resulting image is displayed below in figure 4:

```
# Compute the average face
sum_face = np.sum(train_data, axis=0)
total_faces = len(train_data)
average_face = np.divide(sum_face, total_faces)
```

Figure 3: Compute the average face over the train data set by summing each column of X and dividing by the total number of faces

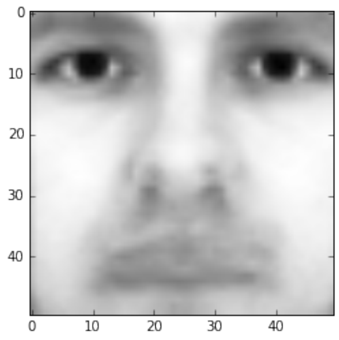


Figure 4: Average face computed from training data set

D. Using the average face computed above, the *mean subtraction* was calculated. By subtracting the average face from every column in matrix X (train data), each resulting row contained a 2500 dimensional vector, flattened image of x_i - μ.

The same computation was then performed on the test data set.

```
# Compute mean subtraction
def get_mean_sub(data_set, average_face):
    mean_sub = []
    for x in data_set:
        mean_sub.append(np.diff([x,average_face], axis=0).flatten())
    return mean_sub
```

Figure 5: 'get_mean_sub' Function

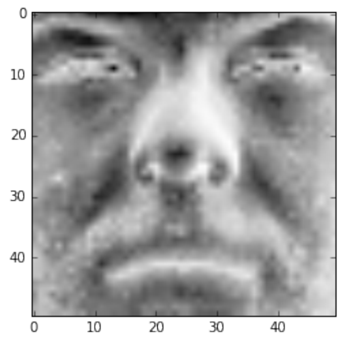


Figure 6: Sample face resulting from mean subtraction computation performed on train data

E. Singular Value Decomposition (SVD) was next performed on training set X, where X = UΣV^T. Each row of V^T, referred to as v_i, has the same dimension as the face image, namely 2500 pixels. Each row v_i is represents the i-th Eigenface.

   SVD was performed on the data (Fig. 7) using the Scipy library. The first 10 eigenfaces are displayed below in figure 8:

```
# Compute Eigenfaces
u, s, v = sp.linalg.svd(np.asmatrix(train_mean_sub))

# Display first 10 Eigenfaces
for eig in v[:10]: display_image(eig)
```

Figure 7: Performing SVD using built in functionality from Scipy library.

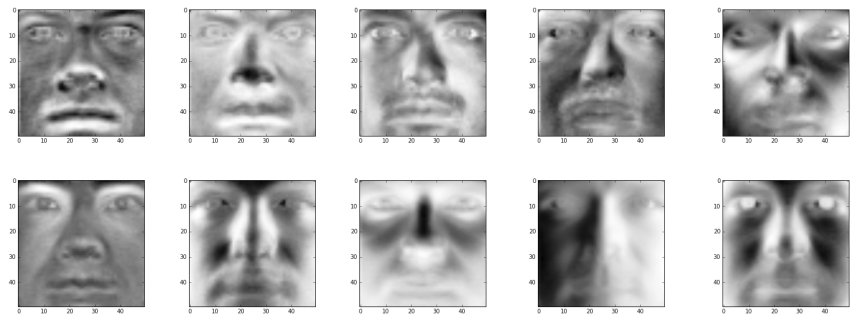


Figure 8: First 10 Eigenfaces as computed above using SVD.

F. The *rank-r approximation* of our data was then calculated as follows:

$$\hat{X}_r = U[:, :r]\Sigma[:r, :r]V^T[:r, :]$$

After calculating $\hat{X}_r$ (Fig. 9), the *rank-r approximation error* was then plotted as a function of r (r = 1,2,...200). The rank-r approximation error was calculated using the Frobenius Norm of a matrix utilizing the built in numpy.norm() function (Fig. 9).

The result are plotted below on figure 10:

```
def rank_r_approx(u, s, v, r):
    sigma = sp.linalg.diagsvd(s, len(u), len(v)) # Reconstruct sigma from singular value
    x = (u[:,:r]).dot(sigma[:r,:r]).dot(v[:r,:])
    return x

def compute_approx_error(x, xr):
    return np.linalg.norm(np.subtract(x, xr))
```

Figure 9: Calculating the rank r approximation error

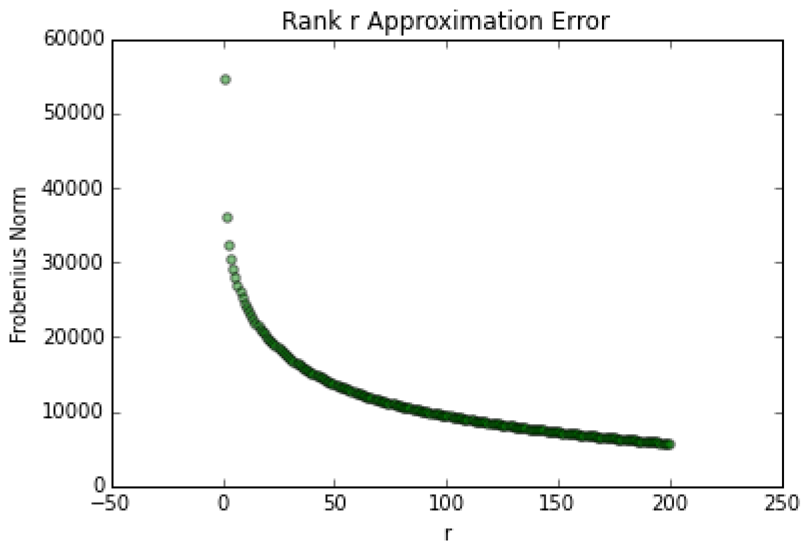


Figure 10: Rank r approximation error as a function of r

G. Next, the r-dimensional Eigenface feature vector, F, was computed. This vector was computed by multiplying X by the transpose of the first r rows of V^T as demonstrated by the function in figure 11.

   F was computed for both the train and test data.

```
def eigdenface_feature(x, v, r):
    vt = np.transpose(v[:r,:])
    f = x.dot(vt)
    return f
```

Figure 11: 'eigenface_feature' function

H. Using F and F_{test} computed above, we trained a Logistic Regression model to fit and predict human face images. The function below (Fig.12) implements Linear Regression from sklearn library, fitting our model with F from our training data and the corresponding training labels. Then, F_{test} was used for prediction of classification.

   Classification was performed for F and F_{test} feature vectors over r = 1,2...200.

   The classification accuracy was recorded for each r and each plotted as a function of r = 1,2,...200 (Fig. 13).

   As evident from the Figure 13, prediction accuracy improved r increased. For example, when r = 10, classification accuracy was approximately 79 percent. When r reached r = 200, the accuracy reached approximately 97 percent.

```
def classify(f_train, train_labels, f_test, test_labels, r):
    # Fit the model using logistic regression
    lr = linear_model.LogisticRegression(random_state=1).fit(f_train, train_labels)

    # Predict using test set
    predicted_values = lr.predict(f_test)
    accuracy = (predicted_values == test_labels).sum() / float(len(test_labels))
    return accuracy

accuracy = {}
for r in range(1,201):
    accuracy[r] = classify(f_train[r], train_labels, f_test[r], test_labels, r)
```

Figure 12: Classification of the test data for r = 1,2,...200

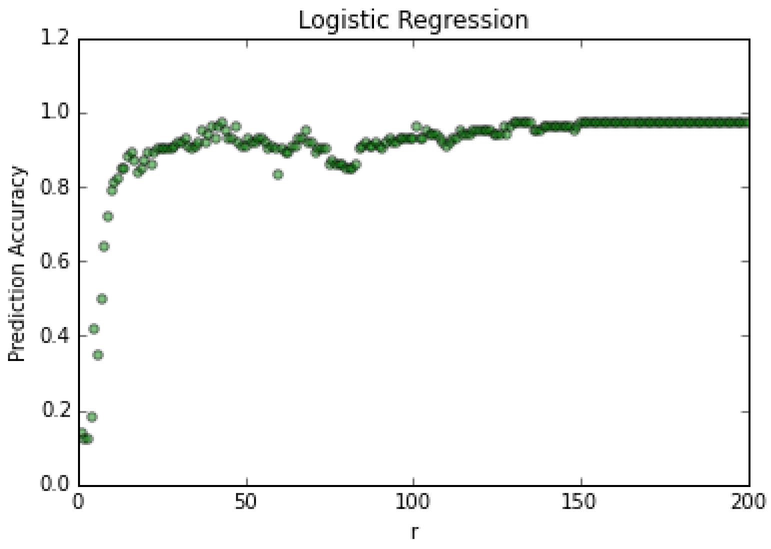


Figure 13: Classification accuracy as a function of r = 1,2,...200

2 Part 2: What's Cooking?

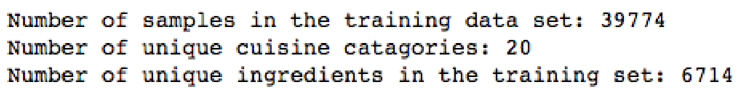
2.1 Summary

The goal was to predict what type of cuisine is cooking based on the ingredients used in the recipe. This was done by creating binary feature vectors of each cuisine, and comparing ingredients found in each labeled recipe with ingredients overall. We performed the classification and predicted the accuracy using several different methods, namely Gaussian Prior Naïve Bayes, Bernoulli Prior Naïve Bayes, and Logistic Regression. We then used the method with the highest accuracy after cross validation (logistic regression) to predict the cuisines in the test set, and submitted our predictions to Kaggle.

2.2 Solution Steps

A. The data set was downloaded, and then parsed using the pandas Python library.

B. The data sets are given in .json format, and consisted of several components. The training set has ids, cuisine labels, and the ingredients involved in each individual recipe. The test set has the same format, minus the cuisine labels (which is what we will be predicting). The number of samples, unique cuisines, and unique ingredients, along with the code and determined this can be found in figures 14 and 15 respectively.



```
Number of samples in the training data set: 39774
Number of unique cuisine catagories: 20
Number of unique ingredients in the training set: 6714
```

Figure 14: Number of Samples, Cuisines, and Ingredients

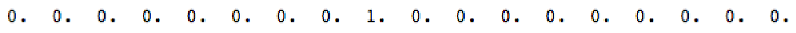
```
ingredients_train = np.array(train_data['ingredients']) # save training ingredients
ingredients_test = np.array(test_data['ingredients'])
# save testing ingredients
cuisines_train = np.array(train_data['cuisine'])
# save training cuisines
# find unique cuisines and ingredients
unique_cuisine_train = np.unique(cuisines_train) # finds unique elements. can also be done using set
# uses itertools chain method to append all sub-lists from a list of lists, and then find unique ele
unique_ingredients_train = np.unique(list(it.chain.from_iterable(ingredients_train)))

print "Number of samples in the training data set: {0}".format(len(train_data))
print "Number of unique cuisine catagories: {0}".format(len(unique_cuisine_train))
print "Number of unique ingredients in the training set: {0}".format(len(unique_ingredients_train))
```

Figure 15: Code to Calculate Part B Numbers

C. Here we represented each dish by a binary ingredient feature vector. A *small* portion of a sample recipe's feature vector can be found in figure

16, and the code that generated it (along with the rest of part C) can be found in figure 17.



```
0.  0.  0.  0.  0.  0.  0.  0.  1.  0.  0.  0.  0.  0.  0.  0.  0.  0.
```

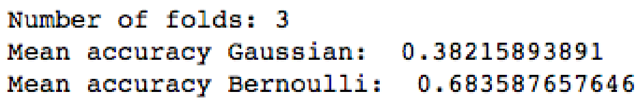
Figure 16: Snippet of a Binary Feature Vector

```
def binary_feature_vectorization(unique_ingredients, ingredient_lists):
    feature_vectors = [] # list to contain all feature vectors
    for ingredients in ingredient_lists:
        # create vector of length 'd', with each element instantiated to '0'
        bin_feat_vect= np.zeros(len(unique_ingredients))
        for ingredient in ingredients: # for each ingredient
            if ingredient in unique_ingredients: # if it is in the recipe
                # set this ingredient to '1', indicating its presence in the recipe
                bin_feat_vect[np.where(unique_ingredients == ingredient)[0][0]] = 1
        feature_vectors.append(bin_feat_vect) # add to list
    return np.array(feature_vectors)

feature_vectors_train = binary_feature_vectorization(unique_ingredients_train, ingredients_train)
feature_vectors_test = binary_feature_vectorization(unique_ingredients_train, ingredients_test)
```

Figure 17: Binary Feature Vectorization Method

D. Using the Naïve Bayes Classifier to perform 3 fold cross-validation on the training set (using both Gaussian and Bernoulli distribution prior assumptions), we attained a low prediction accuracy. The percentages and code that calculated these percentages can be found in figures 18 and 19 respectively.



```
Number of folds: 3
Mean accuracy Gaussian:  0.38215893891
Mean accuracy Bernoulli:  0.683587657646
```

Figure 18: Gaussian and Bernoulli Accuracies

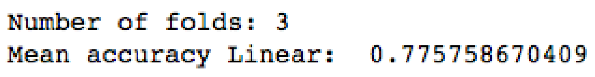
```python
# use 3 different classifiers to fit the data, and perform 3 fold cross validation on each
gnb = nb.GaussianNB().fit(feature_vectors_train, cuisines_train)
bnb = nb.BernoulliNB().fit(feature_vectors_train, cuisines_train)
lr = lm.LogisticRegression().fit(feature_vectors_train, cuisines_train)
# cross validation, k=3
scores_gauss = cv.cross_val_score(gnb, feature_vectors_train, cuisines_train, cv=3)
scores_bernoulli = cv.cross_val_score(bnb, feature_vectors_train, cuisines_train, cv=3)
scores_linear = cv.cross_val_score(lr, feature_vectors_train, cuisines_train, cv=3)

print "Number of folds: 3"
print "Mean accuracy Gaussian: ", scores_gauss.mean()
print "Mean accuracy Bernoulli: ", scores_bernoulli.mean()
print "Mean accuracy Linear: ", scores_linear.mean()
```

Figure 19: Classifier Fitting and Cross Validation

E. As seen in figure 18, the Bernoulli Distribution Prior Assumption had an accuracy over 30% greater than when using the Gaussian assumption. This is largely due to the fact that our data is binary and discrete, the ideal case for Bernoulli Distribution. Additionally the data may not be evenly distributed, which could explain the incredibly poor performance of the Gaussian Distribution Assumption.

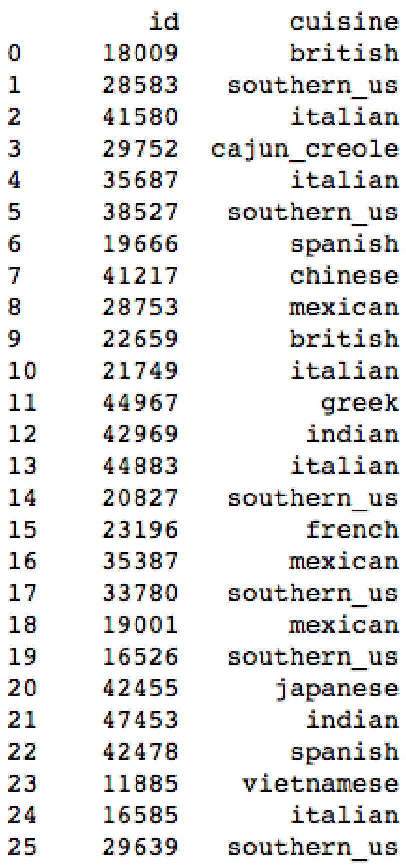
F. Using a Logistic Regression Model classifier, and after performing 3 fold cross-validation, we attained an average accuracy rating greater than either Naive Bayes method. The accuracy can be seen below in figure 20, and the code which calculated the mean accuracy can be found above in figure 19.



```
Number of folds: 3
Mean accuracy Linear:  0.775758670409
```

Figure 20: Gaussian and Bernoulli Accuracies

H. After training our best classifier (logistic regression model) with the training data, we generated cuisine labels on the test set. We then saved the predictions, submitted the results to Kaggle, and received an accuracy of 78.329%. A snippet of the submission .csv file, and the code that generated it can be seen in figures 21 and 22 respectively.



```
         id        cuisine
0     18009        british
1     28583    southern_us
2     41580        italian
3     29752   cajun_creole
4     35687        italian
5     38527    southern_us
6     19666        spanish
7     41217        chinese
8     28753        mexican
9     22659        british
10    21749        italian
11    44967          greek
12    42969         indian
13    44883        italian
14    20827    southern_us
15    23196         french
16    35387        mexican
17    33780    southern_us
18    19001        mexican
19    16526    southern_us
20    42455       japanese
21    47453         indian
22    42478        spanish
23    11885     vietnamese
24    16585        italian
25    29639    southern_us
```

Figure 21: Submission .CSV Snippet

```
# predict test cuisine labels using fitted training data
test_predict = lr.predict(feature_vectors_test)
# store predictions in a dataframe, and save it to a csv file in the required column format
submission = pd.DataFrame({'id': test_data['id'], 'cuisine': test_predict})[['id', 'cuisine']]
submission.to_csv('cuisine_submission.csv', index=False)
```

Figure 22: Predictions and Submission File Code

3 Part 3: Hand Written Problems

3.1 Eigenvalue Problem

Show how to solve the generalized eigenvalue problem max $a^T Ba$ subject to $a^T Wa = 1$ by transforming to a standard eigenvalue problem. (HTF Exercise 4.1)

By applying the Lagrange multiplier, we define:

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

By taking the derivative w.r.t a, we compute:

$$\frac{d\mathcal{L}}{da} = (B + B^T)a - \lambda(W + W^T)a$$

Setting the above to 0, we obtain:

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

This represents the standard eigenvalue problem.

3.2 LDA and Linear Least Squares

Suppose we have features $x \in \mathbb{R}^P$, a two-class response, with class sizes N_1, N_2, and the target coded as $-N/N_1, N/N_2$.

(a) Show that the LDA rule classifies to class 2 if

$$x^T\hat{\sum}^{-1}(\hat{\mu}_2 - \hat{\mu}_1) > \frac{1}{2}\hat{\mu}_2^T\hat{\sum}^{-1}\hat{\mu}_2 - \frac{1}{2}\hat{\mu}_1^T\hat{\sum}^{-1}\hat{\mu}_1 + \log\left(\frac{N_1}{N}\right) - \log\left(\frac{N_2}{N}\right)$$

and class 1 otherwise.

Since we have a binary classifier (two classes), we say $k = 2$.

Additionally we are given that:

$$\delta_k(x) = x^T\hat{\sum}^{-1}\hat{\mu}_k - \frac{1}{2}\hat{\mu}_k^T\hat{\sum}^{-1}\hat{\mu}_k + \log(\frac{N_k}{N})$$

And that:

$$\delta 2(x) > \delta 1(x)$$

We can substitute in each $\delta_i(x)$ where $i = 1, 2$, which gives us the following inequality:

$$x^T\hat{\sum}^{-1}\hat{\mu}_2 - \frac{1}{2}\hat{\mu}_2^T\hat{\sum}^{-1}\hat{\mu}_2 + \log(\frac{N_2}{N}) > x^T\hat{\sum}^{-1}\hat{\mu}_1 - \frac{1}{2}\hat{\mu}_1^T\hat{\sum}^{-1}\hat{\mu}_1 + \log(\frac{N_1}{N})$$

We can then subtract the R.H.S. of the inequality, giving us this simplification:

$$x^T\hat{\sum}^{-1}\hat{\mu}_2 - \frac{1}{2}\hat{\mu}_2^T\hat{\sum}^{-1}\hat{\mu}_2 + \log(\frac{N_2}{N}) - \left(x^T\hat{\sum}^{-1}\hat{\mu}_1 - \frac{1}{2}\hat{\mu}_1^T\hat{\sum}^{-1}\hat{\mu}_1 + \log(\frac{N_1}{N})\right) > 0$$

$$x^T\hat{\sum}^{-1}\hat{\mu}_2 - \frac{1}{2}\hat{\mu}_2^T\hat{\sum}^{-1}\hat{\mu}_2 + \log(\frac{N_2}{N}) - x^T\hat{\sum}^{-1}\hat{\mu}_1 + \frac{1}{2}\hat{\mu}_1^T\hat{\sum}^{-1}\hat{\mu}_1 - \log(\frac{N_1}{N}) > 0$$

$$x^T\hat{\sum}^{-1}(\hat{\mu}_2 - \hat{\mu}_1) - \frac{1}{2}\hat{\mu}_2^T\hat{\sum}^{-1}\hat{\mu}_2 + \log(\frac{N_2}{N}) + \frac{1}{2}\hat{\mu}_1^T\hat{\sum}^{-1}\hat{\mu}_1 - \log(\frac{N_1}{N}) > 0$$

Giving us the inequality above which we were seeking:

$$x^T\hat{\sum}^{-1}(\hat{\mu}_2 - \hat{\mu}_1) > \frac{1}{2}\hat{\mu}_2^T\hat{\sum}^{-1}\hat{\mu}_2 - \frac{1}{2}\hat{\mu}_1^T\hat{\sum}^{-1}\hat{\mu}_1 + \log\left(\frac{N_1}{N}\right) - \log\left(\frac{N_2}{N}\right)$$

(b) Given the minimization of the least squares criterion:

$$\begin{aligned}\sum_{i=1}^{N}(y_i - \beta_0 - \beta^T x_i)^2 &= (y_i - \beta_0 - \beta^T x_i)(y_i - \beta_0 - \beta^T x_i) \\ &= y_i^2 - 2y_i\beta_0 - 2y_i\beta^T x_i + \beta_0^2 + 2\beta_0\beta^T x_i + (\beta^T x_i)^2\end{aligned}$$

If we simplify this expression:
...?

(c) Since, from our previous problem (b) we attain the fact that

$$\hat{\sum}_B = (\hat{\mu}_2 - \hat{\mu}_1)(\hat{\mu}_2 - \hat{\mu}_1)^T$$

and can show that the linear combination $\hat{\sum}_B\beta$ is in the direction of $(\hat{\mu}_2 - \hat{\mu}_1)$ since:

$$\hat{\sum}_B\beta = (\hat{\mu}_2 - \hat{\mu}_1)(\hat{\mu}_2 - \hat{\mu}_1)^T\beta = \lambda(\hat{\mu}_2 - \hat{\mu}_1)$$

is true for a scalar $\lambda \in \mathbb{R}$. Additionally, since $\hat{\sum}_B\beta$ is a linear combination, we know that $\hat{\beta} \propto \hat{\sum}^{-1}(\hat{\mu}_2 - \hat{\mu}_1)$, and therefore the least squares regression coefficient is identical to the LDA coefficient, up to a scalar multiple.

(d) To show that the above holds for any (distinct) coding of the two classes, we can look at any arbitrary coding such that $x \neq y$ where x is in class 1, and y is in class 2. Since these classes have fixed sizes N_1 and N_2, and the target is coded as $-\frac{N}{N_1}$, $-\frac{N}{N_2}$, the above holds.

(e) ???????????

3.3 SVD Problems

3.3.1 (A) Compute the matrices M^TM and MM^T:

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

$$M^TM = \begin{bmatrix} 1 & 3 & 5 & 0 & 1 \\ 2 & 4 & 4 & 2 & 3 \\ 3 & 5 & 3 & 4 & 5 \end{bmatrix} \begin{bmatrix} 1 & 2 & 3 \\ 3 & 4 & 5 \\ 5 & 4 & 3 \\ 0 & 2 & 4 \\ 1 & 3 & 5 \end{bmatrix} = \begin{bmatrix} 36 & 37 & 38 \\ 37 & 49 & 61 \\ 38 & 61 & 84 \end{bmatrix}$$

$$MM^T = \begin{bmatrix} 1 & 2 & 3 \\ 3 & 4 & 5 \\ 5 & 4 & 3 \\ 0 & 2 & 4 \\ 1 & 3 & 5 \end{bmatrix} \begin{bmatrix} 1 & 3 & 5 & 0 & 1 \\ 2 & 4 & 4 & 2 & 3 \\ 3 & 5 & 3 & 4 & 5 \end{bmatrix} = \begin{bmatrix} 14 & 26 & 22 & 16 & 22 \\ 26 & 50 & 46 & 28 & 40 \\ 22 & 46 & 50 & 20 & 32 \\ 16 & 28 & 20 & 20 & 26 \\ 22 & 40 & 32 & 26 & 35 \end{bmatrix}$$

3.3.2 (B) Find the eigenvalues for your matrices of part (a)

(Calculated using an online eigenvalue/vector pair calculator)

MM^T Eigenvalues = [0 0 0 15.4330 153.5670]
M^TM Eigenvalues = [0 15.4330 153.5670]

3.3.3 (C) Find the eigenvectors for your matrices of part (a)

$$M^TM \text{ Eigenvectors} = \begin{bmatrix} -0.482 & -0.8160 & 0.4093 \\ 0.8165 & -0.1259 & 0.5636 \\ -0.4082 & 0.5642 & 0.7176 \end{bmatrix}$$

$$MM^T \text{ Eigenvectors} = \begin{bmatrix} -0.1606 & 0.9197 & 0.1205 & 0.1591 & 0.2977 \\ -0.5131 & -0.3397 & 0.5429 & -0.0332 & 0.5705 \\ 0.1889 & 0.0424 & -0.3871 & -0.7359 & 0.5207 \\ -0.3239 & -0.1560 & -0.7115 & 0.5104 & 0.3226 \\ 0.7552 & -.01128 & 0.1862 & 0.4143 & 0.4590 \end{bmatrix}$$

3.3.4 (D) Find the SVD for the original matrix M from parts (b) and (c).

$$M = U\Sigma V^T$$

$$\begin{bmatrix} 0.2977 & 0.1591 \\ 0.5705 & -0.0332 \\ 0.5207 & -0.7359 \\ 0.3226 & 0.5104 \\ 0.4590 & 0.4143 \end{bmatrix} \begin{bmatrix} \sqrt{153.5670} & 0 \\ 0 & \sqrt{15.4330} \end{bmatrix} \begin{bmatrix} 0.4093 & 0.5632 & 0.7176 \\ -0.8160 & -0.1259 & 0.5642 \end{bmatrix}$$

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

From our SVD components in $M = U\Sigma V^T$, we take our previously calculated Σ and set the smaller singular value to 0. This results in the following matrix (which we'll call Σ'):

$$\Sigma' = \begin{bmatrix} \sqrt{153.5670} & 0 \\ 0 & 0 \end{bmatrix}$$

We can then multiply out the SVD components in $M = U\Sigma' V^T$ to get the one-dimensional approximation to the matrix M:

$$\begin{bmatrix} 0.2977 & 0.1591 \\ 0.5705 & -0.0332 \\ 0.5207 & -0.7359 \\ 0.3226 & 0.5104 \\ 0.4590 & 0.4143 \end{bmatrix} \begin{bmatrix} \sqrt{153.5670} & 0 \\ 0 & 0 \end{bmatrix} \begin{bmatrix} 0.4093 & 0.5632 & 0.7176 \\ -0.8160 & -0.1259 & 0.5642 \end{bmatrix}$$

$$=$$

$$\begin{bmatrix} 3.68916249481 & 0 \\ 7.06975882865 & 0 \\ 6.452626506710001 & 0 \\ 3.99772865578 & 0 \\ 5.6880268227 & 0 \end{bmatrix} \begin{bmatrix} 0.4093 & 0.5632 & 0.7176 \\ -0.8160 & -0.1259 & 0.5642 \end{bmatrix}$$

$$=$$

$$\begin{bmatrix} 1.509974209125733 & 2.077736317076992 & 2.647343006275656 \\ 2.8936522885664453 & 3.9816881722956805 & 5.073258935439241 \\ 2.641060029196403 & 3.6341192485790725 & 4.630404781215097 \\ 1.636270338810754 & 2.251520778935296 & 2.868770083387728 \\ 2.32810937853111 & 3.2034967065446405 & 4.08172804796952 \end{bmatrix}$$

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

The total energy of the original Σ can be found by squaring each of the individual singular values, and adding them:

$$\begin{bmatrix} \sqrt{153.5670} & 0 \\ 0 & \sqrt{15.4330} \end{bmatrix} \Rightarrow \sqrt{153.5670}^2 + sqrt[]15.4330^2 = 153.5670 + 15.4330 = 169$$

The energy of found in Σ' can be found in a similar fashion:

$$\begin{bmatrix} \sqrt{153.5670} & 0 \\ 0 & 0 \end{bmatrix} \Rightarrow \sqrt{153.5670}^2 = 153.5670$$

The amount of retained energy is just the fraction of the energy from Σ' over the total energy:

$$\frac{153.5760}{169} = 0.9086804734 = 90.86804734\%$$

References

[1] Matplotlib Plotting (Histograms, pixel images, etc): http://matplotlib.org/

[2] NumPy Arrays (reshape, etc.): http://www.numpy.org/

[3] Pandas Data Parser (read_csv, to_csv): http://pandas.pydata.org/

[4] Scikit-learn Cross Validation, KFolds, and Linear Regression: http://scikit-learn.org/

[5] SciPy Spatial Euclidean Distance: http://www.scipy.org/

[6] Matrix Multiplication Calculator: http://matrix.reshish.com/multiplication.php

[7] Eigenvalue/Eigenvector Pair Calculator: http://www.mathportal.org/calculators/matrices-calculators/matrix-calculator.php

[8] SVD Calculator: http://comnuan.com/cmnn01004/

[9] Lagrange Multiplier: https://en.wikipedia.org/wiki/Lagrange_multiplier