

Figure 1: Average Face

**CS 5785 - Applied Machine Learning : Homework 2**  
**authors : Francesco Perera and Claire Opila**

## 1 Summary

For this assignment, we implemented the Eigenface method for recognizing human faces using the Yale Face Database. Using the images present in the database, we trained and built a logistic regression model that utilized the eigenfaces to make predictions. In the second part of this assignment, we trained a Bernoulli Naive Bayes classifier, a Gaussian Naive Bayes Classifier, and a Logistic Regression Classifier on the Yummly data set of recipes. The classifiers used a list of ingredients to predict the cuisine of the recipe. For each data point, the features were one hot encoded as having the presence or absence of an ingredient in the training data set.

## 2 Eigenface for face recognition

### 2.1 Download the Face Dataset

The Yale Face Database was downloaded and unzipped to a folder called faces. This folder contains three more folders (images, train and test). The image folder contains all images. We used the train folder to build the logistic regression classifier and the test one to validate results and scores.

### 2.2 Read Training Data

Using the code snippet provided in the assignment outline, we loaded the feature vectors and labels for the training and test data in appropriate numpy arrays. A sample image from both train and test data was plotted.

### 2.3 Average Face

The average face,  $\mu$ , was calculated by computing the average for each column. This calculation was done by using the mean function in numpy (*i.e* `averageFace = np.mean(x, axis = 0)`). See Figure 1

### 2.4 Mean subtraction

The average face was subtracted from every row in the data matrix ( train and test data). Two sample face images from these new train and test matrices were plotted.



Figure 2: The first 10 eigenfaces

## 2.5 Eigenface

Using the svd method present in numpy, we performed Singular Value Decomposition (SVD) on the training data. The svd method generated three outputs:

1. U: left singular matrix
2. D: diagonal matrix whose elements are singular values of the original matrix
3. V: right singular matrix

We then plotted the first 10 images in the right singular matrix, V.

## 2.6 Low - rank Approximation

The rank - r approximation error was calculated using U, D and V from the svd computation. Specifically, the approximation errors were calculated from  $r = 1$  up to  $r = 200$ . The rank - r approximation was computed with the following formula:

$$X(r) = np.dot(U[:, : r] * D[:, r], V[:, : r])$$

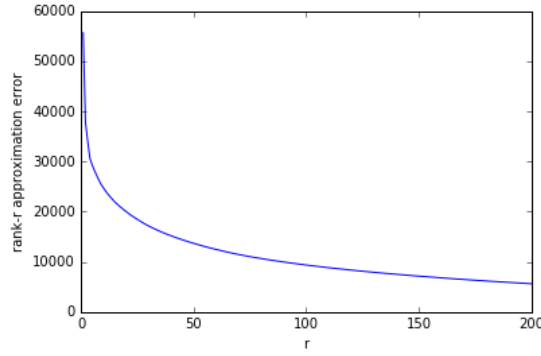


Figure 3: Rank R Approximation

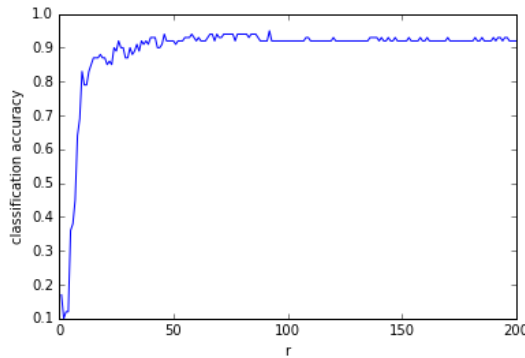


Figure 4: Accuracy

The approximation error was computed by taking the norm of  $X(r)$  and  $X$  for every  $r$ . The norm method in `numpy.linalg` was utilized for the calculation:

$$error = np.linalg.norm(Xr - X)$$

The approximation errors for all  $r$  values were then plotted. Refer to Figure 3.

## 2.7 Eigenface Feature

Using the transpose of the right singular matrix ( $V$ ),  $V^T$ , we computed the matrix of eigenface features by calculating the dot product between the original matrix (train or test matrices) and  $V^T$ . We calculated the dot product by using the dot method in numpy:

$$F = np.dot(x, np.array(v[:, :]).T)$$

where  $F$  is the eigenface matrix,  $x$  is the original train or test matrix and  $v$  is the right singular matrix.

## 2.8 Face Recognition

Using the eigenface matrices  $F$  and  $F_{test}$  for train data and test data, we built the logistic regression model and calculated the mean accuracy score for all  $r$ 's ( $r = 1, 2, \dots, 200$ ).

The accuracy score was then plotted (refer to Figure 4). The plot shows that as  $r$  increases, the mean accuracy settles to approximately 0.9.

# 3 What's Cooking?

## 3.1 Reading the Training Data

The data was downloaded from the Kaggle site, and read in from the JSON format using Pandas. Looking at the data, there is a column of labels or cuisines, and a column of ingredients. Each instance of a recipe contains an array of ingredients.

### 3.2 Analyzing the Data

There are 20 different cuisine types, and a total of 39,774 different rows. Below is a table of the frequency counts of the different recipes by cuisine .

Cuisine	Instance Counts
Brazilian	467
British	804
Cajun Creole	1546
Chinese	2673
Filipino	755
french	2646
greek	1175
Indian	3003
Irish	667
Italian	7838
Jamaican	526
Japanese	1423
Korean	830
Mexican	6438
Moroccan	821
Russian	489
Southern Us	4320
Spanish	989
Thai	1539
Vietnamese	825

As we can see, Italian recipes and Mexican recipes appear with the highest frequency.

If we take a look at the actual ingredients, and we put them into a dictionary, we get 6714 different unique ingredients. If we examine the ingredients even further, we see that the ingredients suffer from casing and phrasing issues. For example, there might be 'Romaine' and 'romaine', and phrases like "Kraft Zesty Italian Dressing." So, now we go back and clean the ingredients so as to break apart any phrases into component words. 'Romaine Lettuce' becomes 'romaine', 'lettuce.' We also convert everything to lowercase, and drop any words that are 'a', 'the', 'of.' We then create a new dictionary from this and we find the number of unique ingredients to be about half as large at 3186. We find the most frequent ingredient to be salt.

### 3.3 One Hot Encoding

Now that we have reduced the unique ingredients to a dictionary, we can more easily one hot encode each ingredient. First, we must create a new dictionary that maps each ingredient to a numerical value. The new dictionary has an ingredient as the key, and a number as the value. We created a new list of ingredients by replacing each ingredient with its numerical representation.

Now that we have this numerical representation, we can one hot encode it. First, we make a blank matrix of zeroes using numpy, that has the following dimensions: the number of row examples X the number of unique ingredients. We then go through the numerical encoded ingredient list, and for each row, we use each of these numbers to index to a specific column in the blank matrix of zeros, and set it to 1. Code below:

```
## next we create an empty numpy matrix of zeroes. We will use this later to set specific index to 1

blank_data = np.empty((len(factorized_list), len(factorized_consolidated_list)+1))

##Now we loop through the entire factorized ingredient matrix represented with numerical values
```

```

## and set that index location in the corresponding row to 1
s = 0
while s < len(factorized_list):
    for item in factorized_list[s]:
        blank_data[s][item] = 1
    s+=1

```

Next, we factorize the cuisine labels to numerical representation using pandas built in function `pd.factorize`. With the one hot encoded matrix, and our factorized labels we can now train our algorithms.

### 3.4 Naive Bayes Algorithm

If we look at Naive Bayes, there is a drastic difference in performance. Bernoulli Naive Bayes almost performs as well as Logistic Regression with a performance of .71. Gaussian Naive Bayes on the other hand, performs with a very poor accuracy of .26. A possible explanation is that the Bernoulli model of the data's distribution more closely matches the Yummly data set, and Gaussian distributions do not do a particularly good job of modeling the data. To elaborate on this, Gaussian is meant for a normal distribution of continuous features, while Bernoulli is better suited for discrete counts, like a bag of words, or the count of particular words. Additionally, Bernoulli Naive Bayes by its very nature is designed to only deal with binary values. Since our data is one hot encoded, signifying the presence of absence of a word, it is engineered more for a Bernoulli classifier than a Gaussian classifier. Code below:

```

##Gaussian Naive Bayes
gnb = GaussianNB()
scores_gnb = cross_validation.cross_val_score(gnb, X, Y, cv=3)
scores_gnb.mean()

##Bernouli Naive Bayes
bnb = BernoulliNB()
scores_bnb = cross_validation.cross_val_score(bnb, X, Y, cv=3)
scores_bnb.mean()

```

### 3.5 Logistic Regression

Using the same dataset, and 3 fold cross validation, we find an accuracy of .781 for Logistic regression.

```

scores = cross_validation.cross_val_score(LogisticRegression(), X, Y, cv=3)
scores.mean()

```

### 3.6 Training the Classifier

Next, we trained the logistic regression classifier on the entire data set of X with the below code:

```

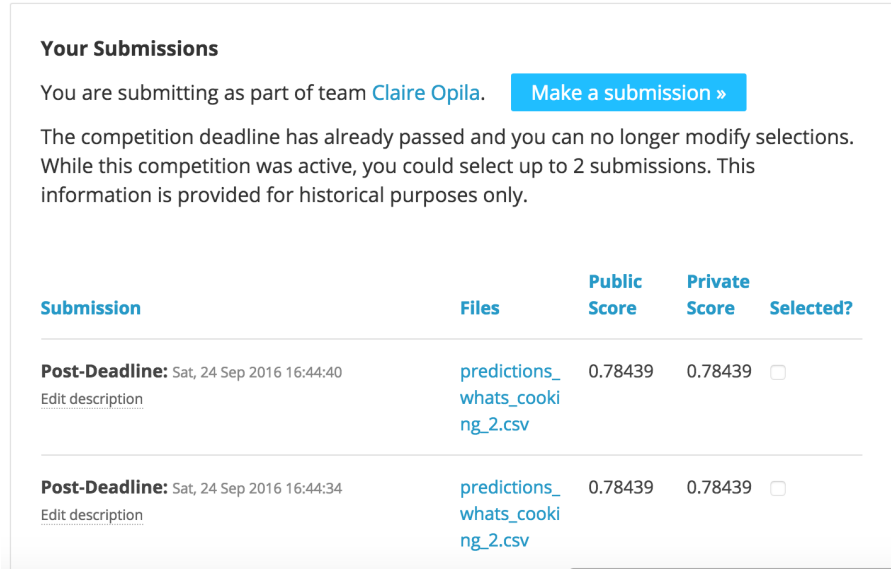
#Train the model on the full data set
clf_full = LogisticRegression()
clf_full.fit(X , Y)

```

### 3.7 Predicting the Cuisines of the Test Data

In order to predict the cuisines of a recipe using the ingredients, we had to clean the test data in the same format that we cleaned the training data. So, we break apart any phrases into component words. For example, 'Romaine Lettuce' becomes 'romaine', 'lettuce.' We also convert everything to lowercase, and drop any words that are 'a', 'the', 'of.' When we map the ingredients to numerical representations. We use the dictionary created off of the train data so that the classifier will know

how to treat the ingredients. For any ingredients that are not in the dictionary from the training data, we drop them, that is we do not include them as features. We then one hot encode these numerical representations as we did the train data. Using this tactic, we predict the cuisines of the recipes in the test data. Our algorithm outputs a list of numerical labels that we map back to the string cuisine label for the Kaggle submission using a dictionary. Our highest performing submission performs with an accuracy of .784.



**Your Submissions**

You are submitting as part of team [Claire Opila](#). [Make a submission »](#)

The competition deadline has already passed and you can no longer modify selections. While this competition was active, you could select up to 2 submissions. This information is provided for historical purposes only.

Submission	Files	Public Score	Private Score	Selected?
<b>Post-Deadline:</b> Sat, 24 Sep 2016 16:44:40 <a href="#">Edit description</a>	<a href="#">predictions_</a> <a href="#">whats_cooki</a> <a href="#">ng_2.csv</a>	0.78439	0.78439	<input type="checkbox"/>
<b>Post-Deadline:</b> Sat, 24 Sep 2016 16:44:34 <a href="#">Edit description</a>	<a href="#">predictions_</a> <a href="#">whats_cooki</a> <a href="#">ng_2.csv</a>	0.78439	0.78439	<input type="checkbox"/>

Figure 5: Kaggle Submission and Score

## 4 Written Exercises

### 4.1

Using the Lagrange multiplier:

$$L(a) = a^T B a - \lambda(a^T W a - 1) = 0$$

$$\frac{dL}{da} = a^T (B + B^T) - \lambda a^T (W + W^T) = 0$$

$$a^T (B + B^T) = \lambda a^T (W + W^T)$$

if B and W are positive definite.  $B = B^T$  and  $W = W^T$  then:

$$a^T (2B^T) = \lambda a^T (2W^T)$$

$$a^T B^T = \lambda a^T W^T$$

$$(Ba)^T = \lambda (Wa)^T$$

Because  $B = B^T$  and  $W = W^T$ , then  $(Ba)^T = Ba$  and  $(Wa)^T = Wa$

$$Ba = \lambda Wa$$

We obtain

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

which is in the form of a standard eigenvalue problem.

## 4.2

a)

The LDA rule classified to class 2 if:

$$\delta(x_1) < \delta(x_2)$$

Replacing that in our equation:

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

We Get

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

b)

Given least squares of:

$$\sum_{i=1}^N (y_i - \beta_o - \beta^T x_i) \geq 0$$

We prove that:

$$\hat{\beta}$$

satisfies:

$$[(N-2) \sum + N \sum_B] \beta = N(\hat{u}_2 - \hat{u}_1)$$

We know that:

$$Y = a_1 U_1 + a_2 U_2$$

$$X^T U_i = N_i \hat{u}_i$$

$$X^T y = a_1 N_1 \hat{u}_1 + a_2 N_2 \hat{u}_2$$

Matrix Form:

$$(Y - \beta_o - X\beta)^T (y - \beta_o - X\beta) \geq 0$$

Minimizing:

$$2X^T X\beta - 2X^T y + 2\beta_o X^T = 0$$

$$2N\beta_o - 2(y - X\beta) = 0$$

$$\beta_o = \frac{Y - X\beta}{N} = 0$$

$$X^T X\beta - X^T y + \frac{(y - X\beta)}{N} X^T = 0$$

$$(X^T X - \frac{XX^T}{N})\beta = X^T y - \frac{yX^T}{N}$$

$$X^T y - \frac{yX^T}{N} = a_i N_i \hat{\mu}_1 + a_2 N_2 \hat{\mu}_2 - \frac{1}{N} (N_1 \hat{\mu}_1 + N_2 \hat{\mu}_2) (a_1 N_1 + N_2 \mu_2)$$

$$= \frac{N_1 N_2}{N} (t_1 - t_2) (\mu_2 - \mu_1)$$

$$\begin{aligned}
X^T X &= (N-2) \sum + N_1 \hat{\mu}_1 \hat{\mu}_1^T + N_2 \hat{\mu}_2 \hat{\mu}_2^T \\
X^T X - \frac{1}{N} X^T X &= (N-2) \sum + \frac{N_1 N_2}{N} \sum_{\beta} \\
[(N-2)] \sum + \frac{N_1 N_2}{N} \sum_{\beta} &= \frac{N_1 N_2}{N} (a_1 - a_2) (\hat{\mu}_1 - \hat{\mu}_2)
\end{aligned}$$

if

$$a_1 = \frac{-N}{N_1}, a_2 = \frac{N}{N_2}$$

Then:

$$[(N-2) \sum + \frac{N_1 N_2}{N} \sum_{\beta}] \beta = N(-\mu_1 + \mu_2)$$

c)

$$\begin{aligned}
\sum_{\beta} \beta &= (\hat{\mu}_2 - \hat{\mu}_1)(\hat{\mu}_2 - \hat{\mu}_1)^T \hat{\beta} = \lambda(\hat{\mu}_2 - \hat{\mu}_1), \text{ where } \lambda \in \mathbb{R} \\
&\epsilon \beta
\end{aligned}$$

is a linear combination in the same direction as

$$\mu_2 - \mu_1$$

d)

This is valid because a1, a2 were arbitrary and distinct.

e)

$$\begin{aligned}
\hat{\beta}_o &= -\frac{1}{N} (N_1 \hat{\mu}_1^T + N_2 \hat{\mu}_2^T) \hat{\beta} \\
f(x) &= \hat{\beta}_o + \hat{\beta}^T x
\end{aligned}$$

if:

$$\begin{aligned}
&\beta - \beta^T \\
f(x) &= -\frac{1}{N} (N_1 \hat{\mu}_1^T + N_2 \hat{\mu}_2^T - N x^T) \hat{\beta} \\
f(x) &= -\frac{1}{N} (N_1 \hat{\mu}_1^T + N_2 \hat{\mu}_2^T - N x^T) \lambda \sum_{\beta}^{-1} (\hat{\mu}_2 - \hat{\mu}_1) \\
&f(x) > 0 \\
&(-\frac{1}{N} N_1 \mu_1^T - \frac{1}{N} N_2 \mu_2^T) \lambda \sum_{\beta}^{-1} (\hat{\mu}_2 - \hat{\mu}_1) + X^T \lambda \sum_{\beta}^{-1} (\mu_2 - \mu_1) > 0 \\
&X^T \sum_{\beta} (\hat{\mu}_2 - \hat{\mu}_1) > \frac{1}{N} (N_1 \hat{\mu}_1^T + N_2 \hat{\mu}_2^T) \sum_{\beta}^{-1} (\mu_2 - \mu_1)
\end{aligned}$$

This only = LDA if N1 = N2



### 11.3.1

(a)

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

$$M^T M = \begin{bmatrix} 36 & 37 & 38 \\ 37 & 49 & 61 \\ 38 & 61 & 84 \end{bmatrix}$$

(b)and(c)

Eigenvalues for  $MM^T$  and  $M^T M$  are the following:

$$\lambda_1 \approx 1.53566996e + 02$$

$$\lambda_2 \approx 1.54330035e + 01$$

Normalized eigenvectors for  $MM^T$  and  $M^T M$  are the following:

$$MM^T: v_1 \approx (0.297696, 0.570509, 0.520742, 0.322578, 0.458985)$$

$$v_2 \approx (-0.15906393, 0.0332003, 0.73585663, -0.5103921, -0.41425998)$$

\* The other original eigenvectors( $v_3, v_4, v_5$ ) from  $MM^T$  were dropped because their eigenvalues were very close to 0 and thus dropped.

$M^T M$ :

$$v_1 \approx (-0.40928285, -0.56345932) \quad v_2 \approx (-0.81597848, -0.12588456)$$

$$v_3 \approx (0.40824829, -0.81649658)$$

\* The third column from  $M^T M$  because its corresponding eigenvalue was very close to 0.

(d)

$$U = \begin{bmatrix} 0.29769568 & -0.15906393 \\ 0.57050856 & 0.0332003 \\ 0.52074297 & 0.73585663 \\ 0.32257847 & -0.5103921 \\ 0.45898491 & -0.41425998 \end{bmatrix}$$

$$V = \begin{bmatrix} 0.409283 & -0.815978 & 0.408248 \\ 0.56346 & -0.125885 & -0.816497 \\ 0.717636 & 0.56421 & 0.408248 \end{bmatrix}$$

$$D = \begin{bmatrix} 12.39221516 & 0 & 0 \\ 0 & 3.92848616 & 0 \\ 0 & 0 & 0 \end{bmatrix} \text{ (square root of eigenvalues)}$$

(e)

$$Aprox = \begin{bmatrix} -1.509889 & -3.01023347 & 1.5060724 \\ -2.89357443 & -5.76885757 & 2.88626023 \\ -2.64116728 & -5.26563883 & 2.63449111 \\ -1.63609257 & -3.26184284 & 1.63195696 \\ -2.32793529 & -4.64115489 & 2.32205088 \end{bmatrix} \text{ (Approximation with first eigenvalue)}$$

(f)

$$\text{energy} = \frac{12.39221516^2}{12.39221516^2 + 3.92848616^2} = 0.9087$$