CS 5785: APPLIED MACHINE LEARNING

HOMEWORK 2

September 27, 2017

Sarah Le Cam - sdl83
Yunie Mao - ym224

Cornell Tech

Contents

EIGENFACE FOR FACE RECOGNITION

Summary

We were given a set of black and white pictures of faces with training and testing data files containing the links to those images and corresponding labels identifying the individuals. Using this data, we calculated the mean image and subtracted it from each of the images in our training set. We then performed a Singular-Value Decomposition to find the set of Eigenfaces. Using our Eigenfaces, we computed the Eigenfeatures and the ranked r-dimensional feature vectors for both the training images and test images. Finally, we fitted the Eigenfeatures and labels of our training set to a logistic regression using *scikit-learn*'s logistic regression model. We used this model to find predicted labels for our test data using the Eigenfeatures of the test images and calculated the mean accuracy on the given test data and labels. To visualize the fit of our model, we plotted our classification's accuracy for the first 200 dimensions in the face space of our test data.

Data

We were provided with a set of 640 pictures total of 10 distinct subjects and two files - a testing and a training text file - matching each image to its respective label. The training set contained the image links and label pairings for 540 images and the testing set contained 100. Each image is 50 x 50 pixels black and white photograph. In order to use this data for model fitting, we converted the images into grayscale and stored the pixel data in a matrix.

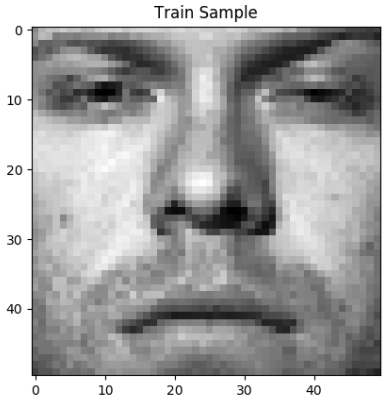
Procedure & Insights

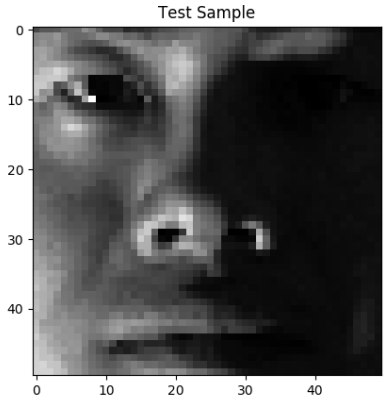
Question 1 (a)

We downloaded and unzipped the faces data file. We then used Anaconda Navigator's Spyder IDE to create a Python project and included our images folder (*faces/images*) and our training and testing data files (*faces/train.txt* & *faces/test.txt*). We generated a Python file (*eigenFaces.py*) and imported the relevant external libraries (NumPy, SciPy, MatPlotLib and sklearn).

Question 1 (b)

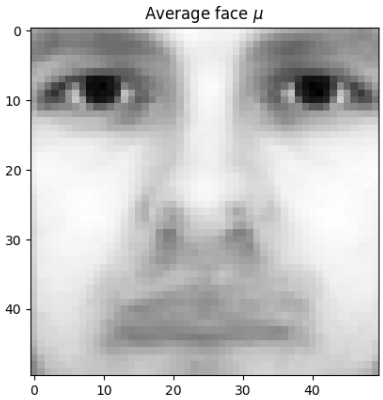
We retrieved each image link from the training and testing datasets using the *split()* function. We then computed the images' greyscale pixel information using MatPlotLib's *imread()* and stored the pixel configurations of the training and testing images in two matrices of size, respectively, 540 x 2500 and 100 x 2500. We then displayed a sample image (the 10^{th} in each dataset) using the pixel information stored in each of these matrices using *imshow()*. We saved these the sample image for the training set as *training_image.png* and that for the testing set as *test_sample.png*. For both the training and testing datasets, we also extracted the labels from the text files into 2 flat arrays of size 540 (training labels) and 100 (testing labels) using the *split()* function.





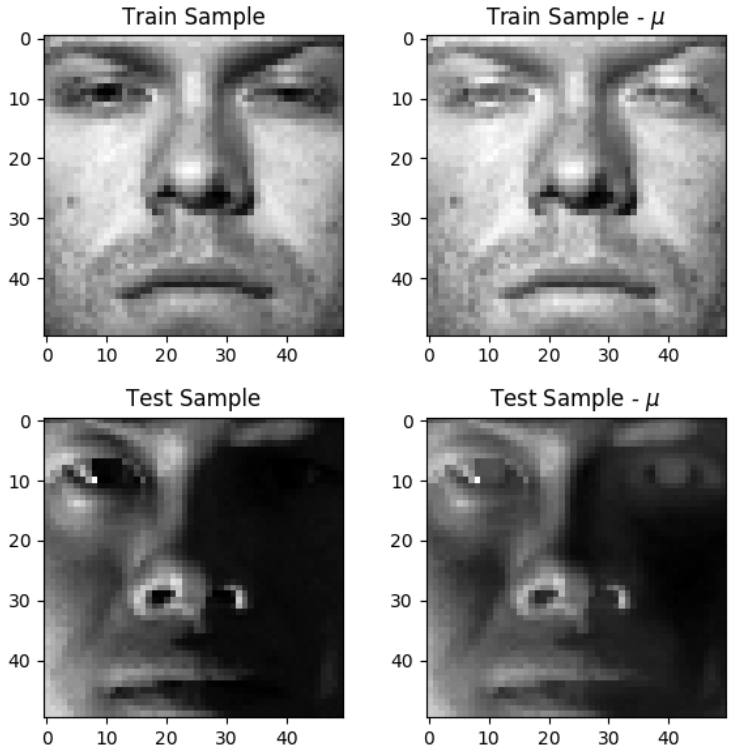
Question 1 (c)

Using the NumPy's *mean()* function along the vertical axis of the training dataset pixel matrix, we found the average face μ and displayed it using the MatPlotLib *imshow()* function. We saved the image as *average_image.png*.



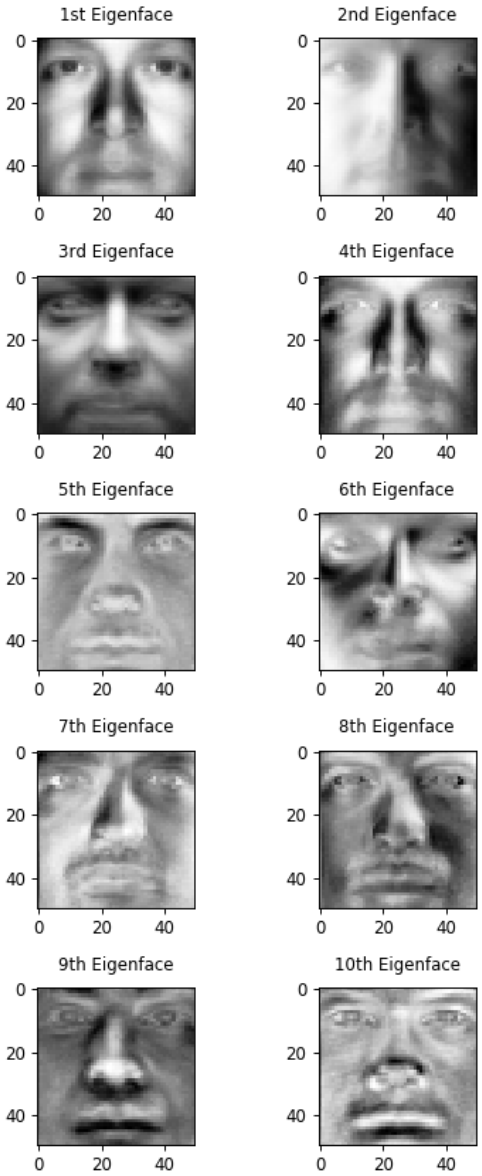
Question 1 (d)

We then subtracted our average face μ's pixel values from those of every image in the training and testing matrices to form new adjusted matrices. These new matrices indicate distance from the mean, allowing us to centralize our data. Again, we displayed a sample (the 10^{th} in each matrix) from each of the new adjusted datasets (see *original_and_adjusted_images.png*).



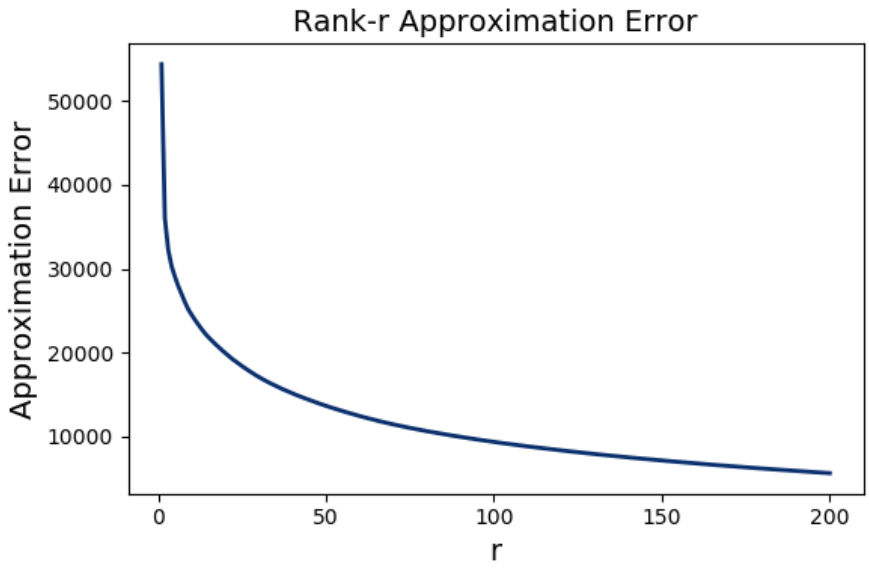
Question 1 (e)

We performed a Singular Value Decomposition (SVD): $X = U\Sigma V^T$ where X is the matrix representation of the adjusted training set. Using NumPy's *linalg.svd()* function, we computed U (the left-singular vector matrix), Σ (the covariance matrix), and V^T (the transpose of the left-singular vectors of X). We then displayed the top ten Eigenfaces from V^T as images in grayscale using *imshow()* (see *first_ten_eigenfaces.png*).



Question 1 (f)

We generated a helper function to compute the rank-r approximation of our adjusted training data by taking the first r columns of U, the first r elements of Σ and the first r rows of V^T. We then computed the low-rank approximation error of our adjusted training data to the rank-r approximation for r = 1, 2, ..., 200 and plotted the results as a function of the value of r (see *low_rank_approximation_err.png*). As the plot shows, as r increases the approximation error decreases exponentially. A value of only 200 for r corresponds to a relatively low approximation error.



Question 1 (g)

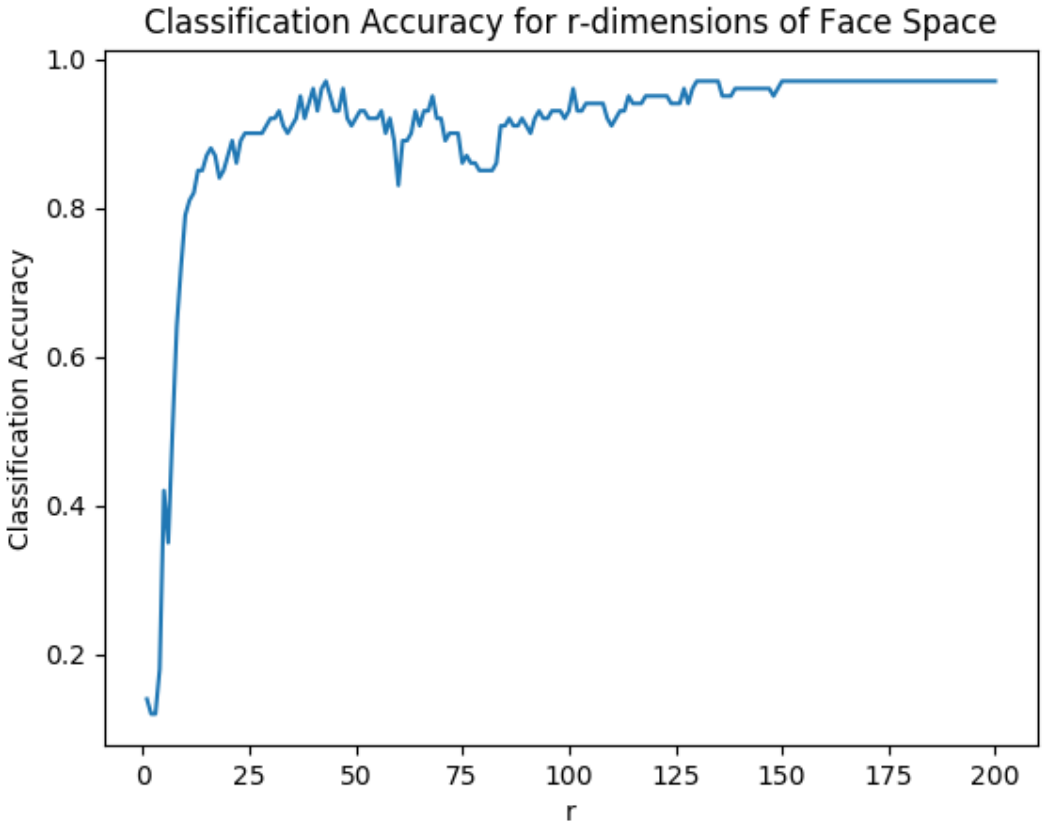
Since the first r Eigenfaces span the r-dimensional subspace of the original image space (*face space*), we can represent a 2500-dimensional face image as an r-dimensional feature vector and thereby reduce the dimensions prior to classification. To compute the r-dimensional feature matrices for the training and test images, we multiplied these images by the transpose of the first r rows of V^T.

Question 1 (h)

Using the function we generated in 1(g), we extracted the Eigenfeatures for our training and test data for r = 10. We then fitted our training Eigenfeatures into a logistic regression model provided by scikit-learn and used the model to generate predicted labels for the test data. We

then computed the classification accuracy rate on the test data given the Eigenfeatures and labels. We achieved a classification accuracy of 79%.

To show the classification rate on the test data as a function of r, we generated the Eigenfeatures, trained a logistic regression model, and computed the accuracy rate on our test set for the first 200 values of r. The following plot (*face_recognition_classification_accuracy.png*) shows the classification accuracy for varying r-dimensions. We can see that for values of $r > 30$, we achieve a classification accuracy of 90%.



WHAT'S COOKING?

Summary

We were given descriptive information of many different dishes for training and testing purposes. Our goal was to find the best possible method for classifying recipes by cuisine when given their respective ingredients. We first transformed the data into a usable numeric matrix, then performed multiple classification attempts using different methods. We found that the logistic regression model worked best.

Data

The Kaggle competition provided us with training and testing json files containing information describing recipes. Both datasets contained recipe identifiers and ingredients lists. The training dataset included an addition cuisine field. The training file included 39,774 dishes with 20 categories and 6,714 unique ingredients. The testing file included 9,944 dishes with 4,484 unique ingredients.

Procedure & Insights

Question 2 (a)

We joined the Kaggle "What's Cooking?" competition and downloaded the training (*train.json*) and testing (*test.json*) data files. We then used Anaconda Navigator's Spyder IDE to create a Python project and included these files. We generated a Python file (*cooking.py*) and imported the relevant external libraries (NumPy, Pandas, IterTools, MatPlotLib and sklearn).

Question 2 (b)

We used Pandas' DataFrame to import the json data and find the number of distinct dishes and cuisines. The training file includes 39,774 dishes spanning 20 categories. We then used an iterable function to extract the ingredient information from the lists contained in each data object. There were 6,714 unique ingredients included in total in the training set.

Question 2 (c)

To set up our training set for classification, we generated an n x d matrix, where n is the number of dish samples and d is the total number of unique ingredients for both the training (39,774 x 6,714 matrix) and testing (39,774 x 6,714 matrix) datasets. We represented each dish as a binary ingredient vector x, where $x_i = 1$ if the dish contains ingredient i and $x_i = 0$ otherwise. This allows us to have a numeric representation of the dish composition for model fitting and predictions based on the non-numerical databases. To generate these matrices, we used an encoder and scikit-learn's *CountVectorizer()* function to generate a map of the ingredients to their frequency in each dish.

Question 2 (d)

Using scikit-learn's Naive Bayes Classifier, we performed a 3 fold cross-validation on the training data with the Bernoulli distribution prior and the Gaussian distribution prior assumptions. We achieved an average accuracy rate of 68.2% using the Bernoulli Naive Bayes Classifier and 36.9% using the Gaussian Naive Bayes Classifier.

Question 2 (e)

The Bernoulli Naive Bayes Classifier performed much better than the Gaussian Naive Bayes Classifier. This makes sense because we represented each dish as a binary ingredient vector x in our training data. The Bernoulli Naive Bayes best fits our assumptions because it describes

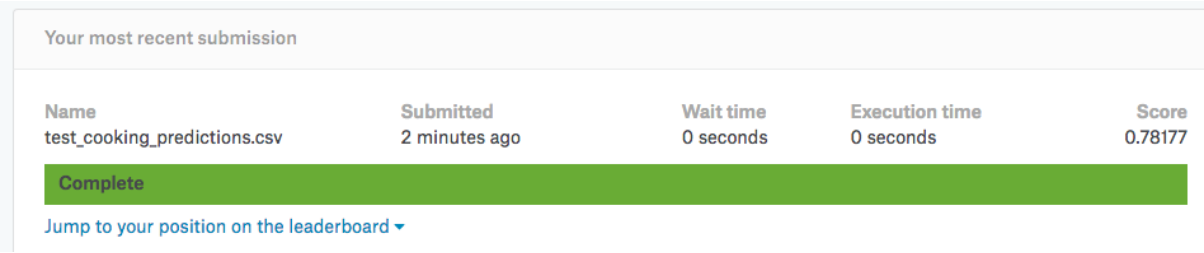
whether or not an ingredient was found in a dish while the Gaussian Naive Bayes Classifier is generally used to describe continuous data that is normally distributed.

Question 2 (f)

We performed a 3 fold cross-validation on the training data using scikit-learn's Logistic Regression model. We achieved an average classification accuracy of 77.5%.

Question 2 (g)

Based on the results from 1(d) and 1(f), the Logistic Regression model had the best accuracy performance over our training data. Using this model, we classified the cuisines based on the dish ingredients in our test data. We generated a csv file that contained the list of dish ids and their corresponding predicted cuisine labels. After submitting our results to Kaggle, we received an accuracy rate of 78.177%.



Written Exercises

Question 1

Maximize: $a^T B a$
Subject to: $a^T W a = 1$

Apply Lagrange Multiplier:
$l(\lambda) = a^T B a – \lambda (a^T W a - 1)$
$\partial l/\partial a = (B + B^T) a – \lambda (W + W^T) a = 0$
$= (B + B^T) a = \lambda (W + W^T) a$
$= (W + W^T)^{-1} (B + B^T) a = \lambda a$

Because matrices B and W are symmetric, we reduce this to
$W^{-1} B a = \lambda a$
This becomes a standard eigenvalue problem $T(a) = \lambda a$ where we apply transformation of $(W^{-1} B)$ over a.

Question 2

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

$\delta_1(x) = x^T \Sigma^{-1} \mu_1 - \frac{1}{2} \mu_1^T \Sigma^{-1} \mu_1 + \log \Pi_1$

$\delta_2(x) = x^T \Sigma^{-1} \mu_2 - \frac{1}{2} \mu_2^T \Sigma^{-1} \mu_2 + \log \Pi_2$

When $\delta_2(x) > \delta_1(x)$, the LDA rule classifies x to class 2 if:

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

$$= \frac{1}{2} \hat{\mu}_2^T \hat{\Sigma}^{-1} \hat{\mu}_2 - \frac{1}{2} \hat{\mu}_1^T \hat{\Sigma}^{-1} \mu_1 + \log\left(\frac{N_1}{N}\right) - \log\left(\frac{N_2}{N}\right)$$

$$= \frac{1}{2} (\hat{\mu}_2 - \hat{\mu}_1)^T \hat{\Sigma}^{-1} (\hat{\mu}_2 - \hat{\mu}_1) - \log\left(\frac{N_2}{N_1}\right)$$

c) $\hat{\Sigma}_B \beta = \frac{N_1 N_2}{N^2} (\hat{\mu}_2 - \hat{\mu}_1)(\hat{\mu}_2 - \hat{\mu}_1)^T \beta$ (b)

Because $(\hat{\mu}_2 - \hat{\mu}_1)^T \beta$ is a scalar and $\frac{N_1 N_2}{N^2}$ is a constant, $\Sigma_B \beta$ must be a vector in the direction of $(\hat{\mu}_2 - \hat{\mu}_1)$.

Let $S = \frac{N_1 N_2}{N^2} (\hat{\mu}_2 - \hat{\mu}_1)^T \beta$ so

$$\hat{\Sigma}_B \beta = S (\hat{\mu}_2 - \hat{\mu}_1)$$

~~There~~ We have:

$$[(N-2)\hat{\Sigma} + N\hat{\Sigma}_B]\beta = N(\hat{\mu}_2 - \hat{\mu}_1)$$

$$(N-2)\hat{\Sigma}\beta + N\hat{\Sigma}_B\beta = N(\hat{\mu}_2 - \hat{\mu}_1)$$

$$(N-2)\hat{\Sigma}\beta + N \cdot S(\hat{\mu}_2 - \hat{\mu}_1) = N(\hat{\mu}_2 - \hat{\mu}_1)$$

$$\hat{\Sigma}\beta = \frac{N}{N-2}(1-S)(\hat{\mu}_2 - \hat{\mu}_1)$$

Since $\frac{N}{N-2}$ and $(1-S)$ are scalars, we can rewrite:

$$D = \frac{N}{N-2}(1-S)$$

$$\hat{\Sigma}\beta = D(\hat{\mu}_2 - \hat{\mu}_1)$$

$$\beta = D\Sigma^{-1}(\mu_2 - \mu_1) \Rightarrow \beta \propto \Sigma^{-1}(\mu_2 - \mu_1)$$

Question 3

(a)
We computed MM^T and M^TM using numpy's built-in functions for matrix operations.
m = np.array([[1,0,3],[3,7,2],[2,-2,8],[0,-1,1],[5,8,7]])
m.dot(m.transpose()) produces:

$$MM^T = \begin{bmatrix} 10 & 9 & 26 & 3 & 26 \\ 9 & 62 & 8 & -5 & 85 \\ 26 & 8 & 72 & 10 & 50 \\ 3 & -5 & 10 & 2 & -1 \\ 29 & 85 & 50 & -1 & 138 \end{bmatrix}$$

m.transpose().dot(m) produces:

$$M^TM = \begin{bmatrix} 39 & 57 & 60 \\ 57 & 118 & 53 \\ 60 & 53 & 127 \end{bmatrix}$$

(b)
We computed the eigenvalues and eigenvectors using numpy's linalg.eig function.
eigenvalues, eigenvectors = np.linalg.eig(m.dot(m.transpose()))

Eigenvalues for MM^T: 214.67 and 69.32

Eigenvalues for M^TM: 214.67 and 69.32

(c)
Eigenvectors for MM^T:

$$\begin{bmatrix} -0.16492942 \\ -0.47164732 \\ -0.33647055 \\ -0.00330585 \\ -0.79820031 \end{bmatrix} \text{ and } \begin{bmatrix} 0.24497323 \\ -0.45330644 \\ 0.82943965 \\ 0.16974659 \\ -0.13310656 \end{bmatrix}$$

Eigenvectors for M^TM

$$\begin{bmatrix} 0.42615127 \\ 0.61500884 \\ 0.66344497 \end{bmatrix} \text{ and } \begin{bmatrix} -0.01460404 \\ -0.72859799 \\ 0.68478587 \end{bmatrix}$$

(d) SVD

$M = U \sum V^T$

U is the eigenvectors of MM^T, and V is eigenvectors of M^TM.

$$\sum = \begin{bmatrix} 14.65 & 0 \\ 0 & 8.32 \end{bmatrix}$$

$$U = \begin{bmatrix} -0.16492942 & 0.24497323 \\ -0.47164732 & -0.45330644 \\ -0.33647055 & 0.82943965 \\ -0.00330585 & 0.16974659 \\ -0.79820031 & -0.13310656 \end{bmatrix}$$

$$V^T = \begin{bmatrix} 0.42615127 & 0.61500884 & 0.66344497 \\ -0.01460404 & -0.72859799 & 0.68478587 \end{bmatrix}$$

(e) $\sum = 14.65$

$M = U[:,1] \sum V^T[1,:]$

$$M = \begin{bmatrix} -0.16492942 \\ -0.47164732 \\ -0.33647055 \\ -0.00330585 \\ -0.79820031 \end{bmatrix} [14.65] [0.42615127 \quad 0.61500884 \quad 0.66344497]$$

$$= \begin{bmatrix} -1.02967352 & -1.4859942 & -1.60302635 \\ -2.94454898 & -4.24948552 & -4.58416142 \\ -2.10062471 & -3.03155911 & -3.27031502 \\ [-0.02063881 & -0.02978531 & -0.03213111 \\ -4.98325721 & -7.19168861 & -7.75808302 \end{bmatrix}$$

SOURCES & EXTERNAL LIBRARIES

Stéfan van der Walt, S. Chris Colbert and Gaël Varoquaux. *The NumPy Array: A Structure for Efficient Numerical Computation*, Computing in Science & Engineering, 13, 22-30 (2011), DOI:10.1109/MCSE.2011.37

John D. Hunter. *Matplotlib: A 2D Graphics Environment*, Computing in Science & Engineering, 9, 90-95 (2007), DOI:10.1109/MCSE.2007.55

Jones E, Oliphant E, Peterson P, et al. *SciPy: Open Source Scientific Tools for Python*, 2001-, `http://www.scipy.org/`

Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, Jake Vanderplas, Alexandre Passos, David Cournapeau, Matthieu Brucher, Matthieu Perrot, Édouard Duchesnay. *Scikit-learn: Machine Learning in Python*, Journal of Machine Learning Research, 12, 2825-2830 (2011)

Wes McKinney. *Data Structures for Statistical Computing in Python*, Proceedings of the 9th Python in Science Conference, 51-56 (2010)

What's Cooking? | Kaggle, www.kaggle.com/c/whats-cooking.