**《Computer Vision Foundations and Applications》Homework 3**

1. **Requirements**

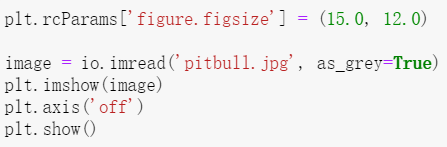
**Attention: Please hand in hw3\_release’s hw3.ipynb file with all the outputs.**

This assignment covers: **image compression using SVD**, **kNN methods for image recognition**. **PCA and LDA to improve kNN.**



**Part 1: Image Compression**

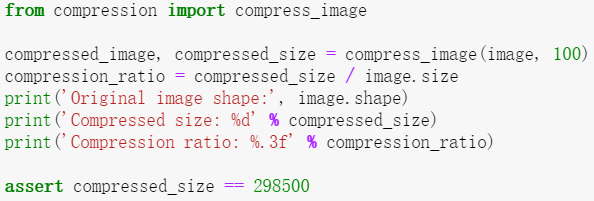
Image compression is used to reduce the cost of storage and transmission of images (or videos). One lossy compression method is to apply Singular Value Decomposition (SVD) to an image, and only keep the top n singular values.



Let's implement image compression using SVD.

We first compute the SVD of the image, and as seen in class we keep the n largest singular values and singular vectors to reconstruct the image.

Implement function compress\_image in compression.py.





**Face Dataset**

We will use a dataset of faces of celebrities. Download the dataset using the following command:

sh get\_dataset.sh

The face dataset for CS131 assignment. The directory containing the dataset has the following structure:

faces/

train/

angelina jolie/

anne hathaway/

...

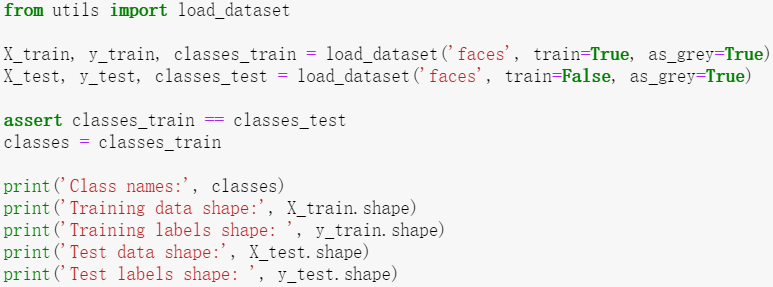
test/

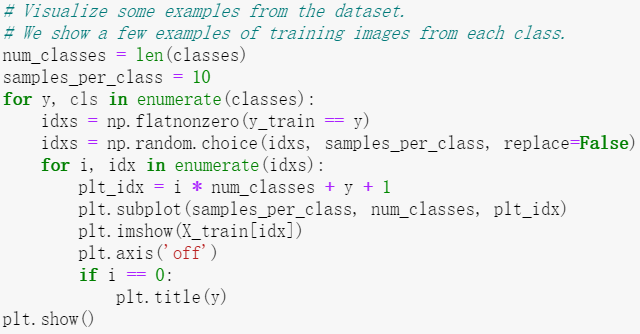
angelina jolie/

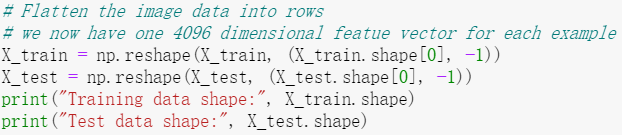
anne hathaway/

...

Each class has 50 training images and 10 testing images.







**Part2: k-Nearest Neighbor**

We're now going to try to classify the test images using the k-nearest neighbors algorithm on the **raw features of the images** (i.e. the pixel values themselves). We will see later how we can use kNN on better features.

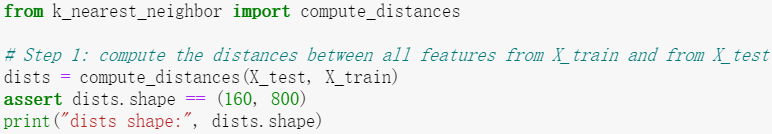
Here are the steps that we will follow:

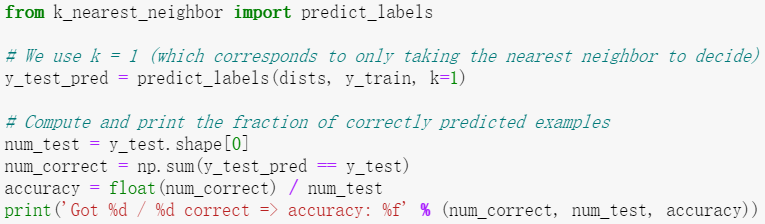
1.We compute the L2 distances between every element of X\_test and every element of X\_train in compute\_distances.

2.We split the dataset into 5 folds for cross-validation in split\_folds.

3.For each fold, and for different values of k, we predict the labels and measure accuracy.

4.Using the best k found through cross-validation, we measure accuracy on the test set.





**Cross-Validation**

We don't know the best value for our parameter k. There is no theory on how to choose an optimal k, and the way to choose it is through cross-validation.

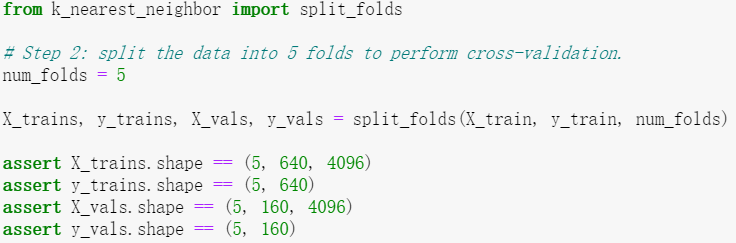
We **cannot** compute any metric on the test set to choose the best k, because we want our final test accuracy to reflect a real use case. This real use case would be a setting where we have new examples come and we classify them on the go. There is no way to check the accuracy beforehand on that set of test examples to determine k.

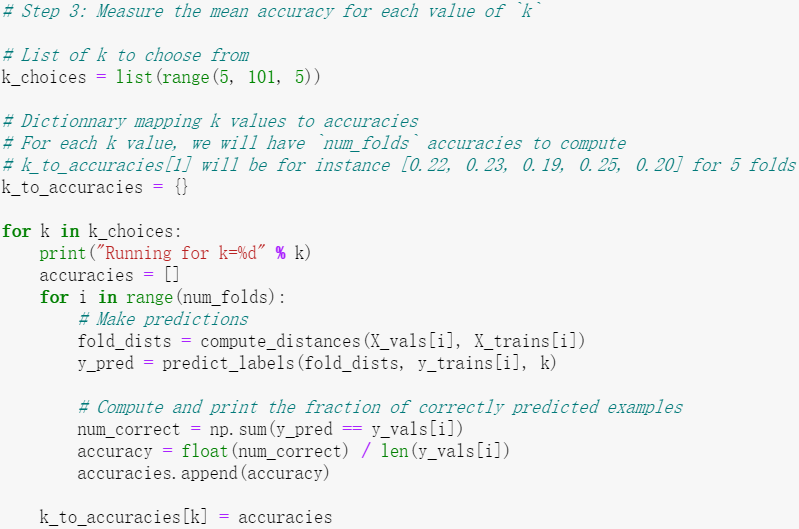
Cross-validation will make use split the data into different fold (5 here).

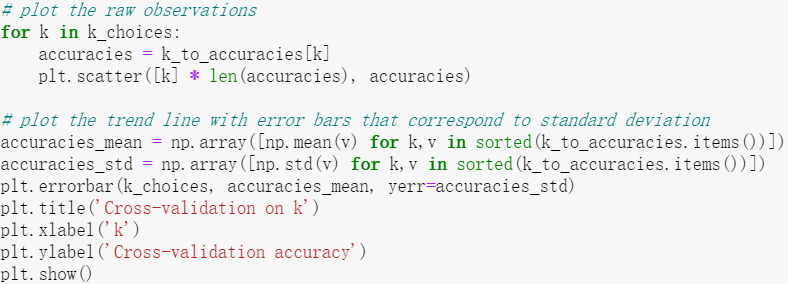
For each fold, if we have a total of 5 folds we will have:

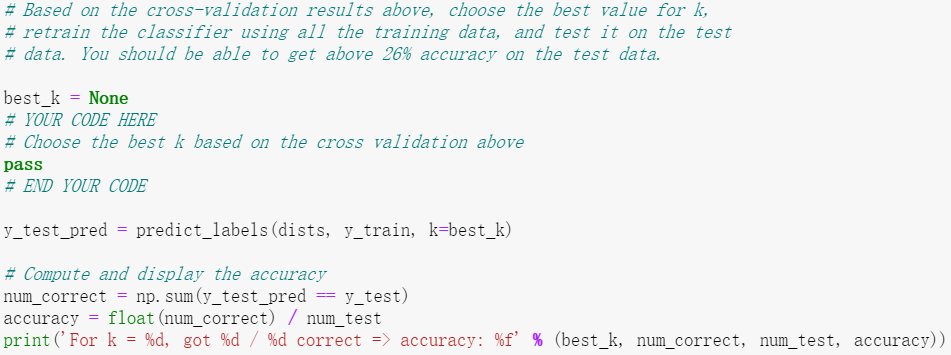
* 80% of the data as training data
* 20% of the data as validation data

We will compute the accuracy on the validation accuracy for each fold, and use the mean of these 5 accuracies to determine the best parameter k.









**Part3: PCA**

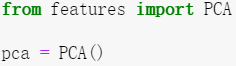
Principal Component Analysis (PCA) is a simple yet popular and useful linear transformation technique that is used in numerous applications, such as stock market predictions, the analysis of gene expression data, and many more. In this tutorial, we will see that PCA is not just a "black box", and we are going to unravel its internals in 3 basic steps.

**Introduction**

The sheer size of data in the modern age is not only a challenge for computer hardware but also a main bottleneck for the performance of many machine learning algorithms. The main goal of a PCA analysis is to identify patterns in data; PCA aims to detect the correlation between variables. If a strong correlation between variables exists, the attempt to reduce the dimensionality only makes sense. In a nutshell, this is what PCA is all about: Finding the directions of maximum variance in high-dimensional data and project it onto a smaller dimensional subspace while retaining most of the information.

**A Summary of the PCA Approach**

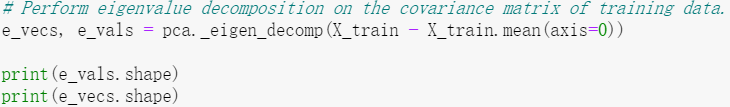
* Standardize the data.
* Obtain the Eigenvectors and Eigenvalues from the covariance matrix or correlation matrix, or perform Singular Vector Decomposition.
* Sort eigenvalues in descending order and choose the 𝑘 eigenvectors that correspond to the 𝑘 largest eigenvalues where 𝑘 is the number of dimensions of the new feature subspace ( 𝑘≤𝑑 ).
* Construct the projection matrix 𝐖 from the selected 𝑘 eigenvectors.
* Transform the original dataset 𝐗 via 𝐖 to obtain a 𝑘 -dimensional feature subspace Y.



**3.1 Eigendecomposition**

The eigenvectors and eigenvalues of a covariance (or correlation) matrix represent the "core" of a PCA: The eigenvectors (principal components) determine the directions of the new feature space, and the eigenvalues determine their magnitude. In other words, the eigenvalues explain the variance of the data along the new feature axes.

Implement \_eigen\_decomp in features.py.



**3.2 Singular Value Decomposition**

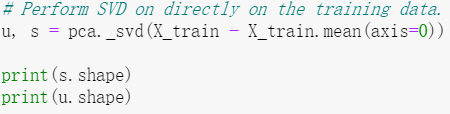
Doing an eigendecomposition of the covariance matrix is very expensive, especially when the number of features (D = 4096 here) gets very high.

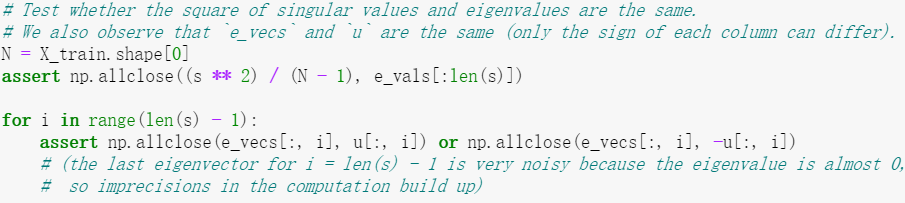
To obtain the same eigenvalues and eigenvectors in a more efficient way, we can use Singular Value Decomposition (SVD). If we perform SVD on matrix 𝑋 , we obtain 𝑈 , 𝑆 and 𝑉 such that:

* the columns of 𝑈 are the eigenvectors of
* the columns of are the eigenvectors of
* the values of 𝑆 are the square roots of the eigenvalues of (or )

Therefore, we can find out the top k eigenvectors of the covariance matrix 𝑋𝑇𝑋 using SVD.

Implement \_svd in features.py.



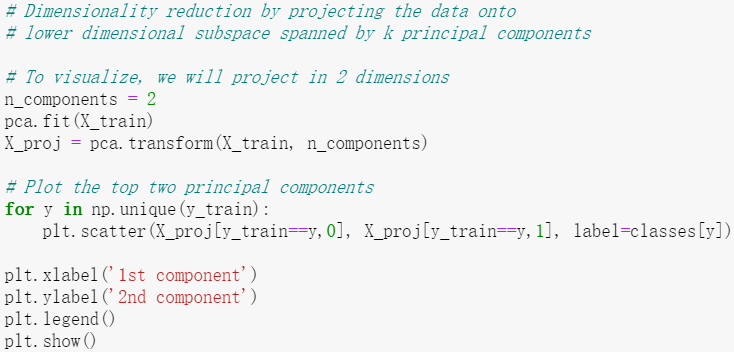


**3.3 Dimensionality Reduction**

The top 𝑘 principal components explain most of the variance of the underlying data.

By projecting our initial data (the images) onto the subspace spanned by the top k principal components, we can reduce the dimension of our inputs while keeping most of the information.

In the example below, we can see that **using the first two components in PCA is not enough** to allow us to see pattern in the data. All the classes seem placed at random in the 2D plane.

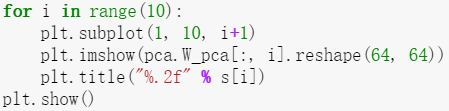


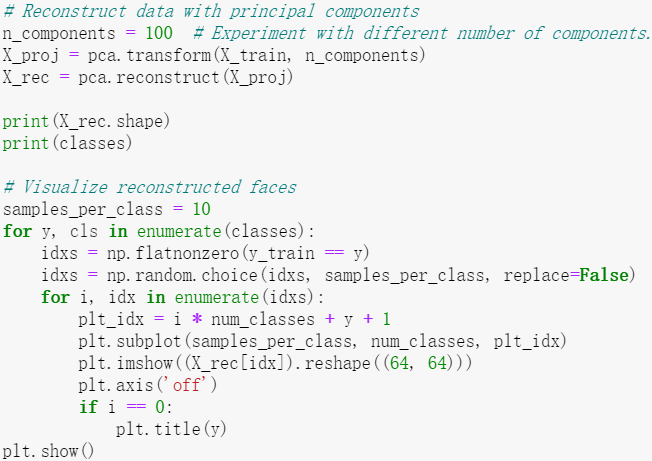
**3.4 Visualizing Eigenfaces**

The columns of the PCA projection matrix pca.W\_pca represent the eigenvectors of .

We can visualize the biggest singular values as well as the corresponding vectors to get a sense of what the PCA algorithm is extracting.

If we visualize the top 10 eigenfaces, we can see tht the algorithm focuses on the different shades of the faces. For instance, in face n°2 the light seems to come from the left.





**Written Question 1**

Question: Consider a dataset of 𝑁 face images, each with shape (ℎ,𝑤) . Then, we need 𝑂(𝑁ℎ𝑤) of memory to store the data. Suppose we perform dimensionality reduction on the dataset with 𝑝 principal components, and use the components as bases to represent images. Calculate how much memory we need to store the images and the matrix used to get back to the original space.

Said in another way, how much memory does storing the compressed images and the uncompresser cost.

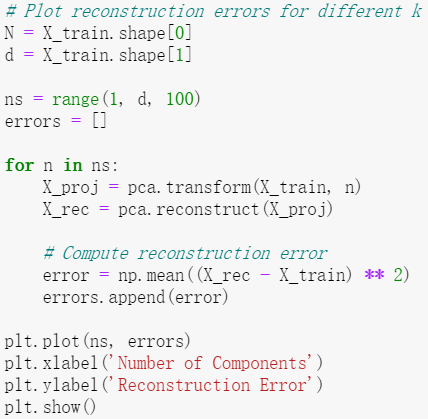
**Answer: TODO**

**3.5 - Reconstruction error and captured variance**

We can plot the reconstruction error with respect to the dimension of the projected space.

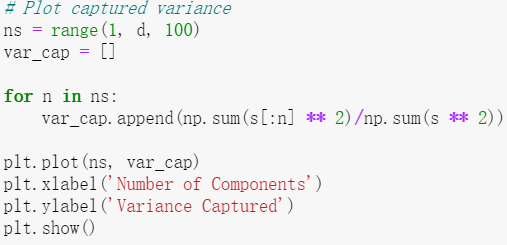
The reconstruction gets better with more components.

We can see in the plot that the inflexion point is around dimension 200 or 300. This means that using this number of components is a good compromise between good reconstruction and low dimension.



We can do the same process to see how much variance is captured by the projection.

Again, we see that the inflexion point is around 200 or 300 dimensions.



**3.6 - kNN with PCA**

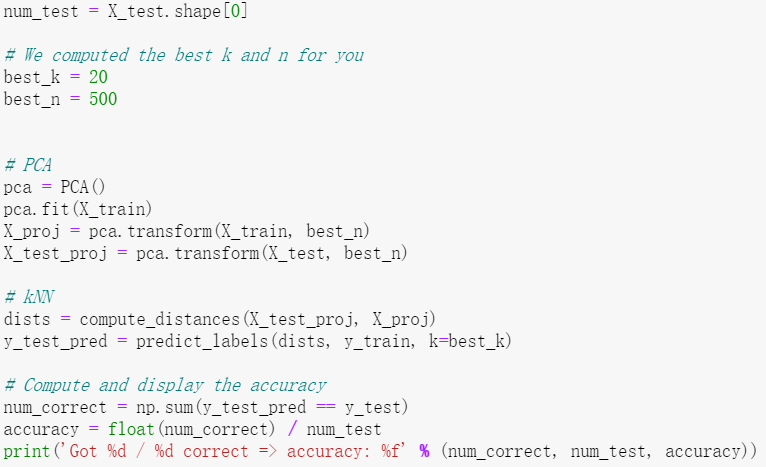
Performing kNN on raw features (the pixels of the image) does not yield very good results.

Computing the distance between images in the image space is not a very good metric for actual proximity of images.

For instance, an image of person A with a dark background will be close to an image of B with a dark background, although these people are not the same.

Using a technique like PCA can help discover the real interesting features and perform kNN on them could give better accuracy.

However, we observe here that PCA doesn't really help to disentangle the features and obtain useful distance metrics between the different classes. We basically obtain the same performance as with raw features.



**Written Question 2**

Question: Given a dataset that consists of images of the Hoover tower, your task is to learn a classifier to detect the Hoover tower in new images. You implement PCA to reduce the dimensionality of your data, but find that your performance in detecting the Hoover tower significantly drops in comparison to your method on the original input data. A sample of your input training images is shown below. Why is the performance suffering?

Hoover Tower Training Set

**Answer: TODO**

**Part4: Fisherface: Linear Discriminant Analysis**

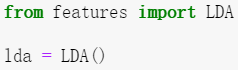
LDA is a linear transformation method like PCA, but with a different goal.

The main difference is that LDA takes information from the labels of the examples to maximize the separation of the different classes in the transformed space.

Therefore, LDA is **not totally unsupervised** since it requires labels. PCA is **fully unsupervised**.

In summary:

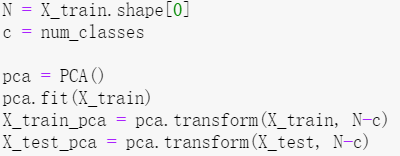
* PCA perserves maximum variance in the projected space.
* LDA preserves discrimination between classes in the project space. We want to maximize scatter between classes and minimize scatter intra class.



**4.1 - Dimensionality Reduction via PCA**

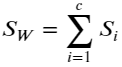
To apply LDA, we need 𝐷<𝑁 . Since in our dataset, 𝑁=800 and 𝐷=4096 , we first need to reduce the number of dimensions of the images using PCA.

More information at: <http://www.scholarpedia.org/article/Fisherfaces>



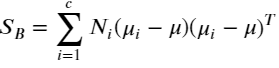
**4.2 - Scatter matrices**

We first need to compute the within-class scatter matrix:

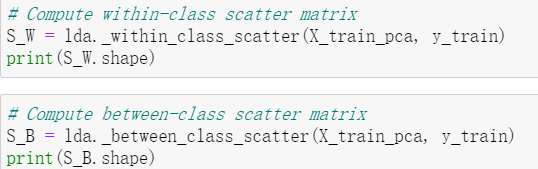


where  is the scatter of class 𝑖 .

We then need to compute the between-class scatter matrix:

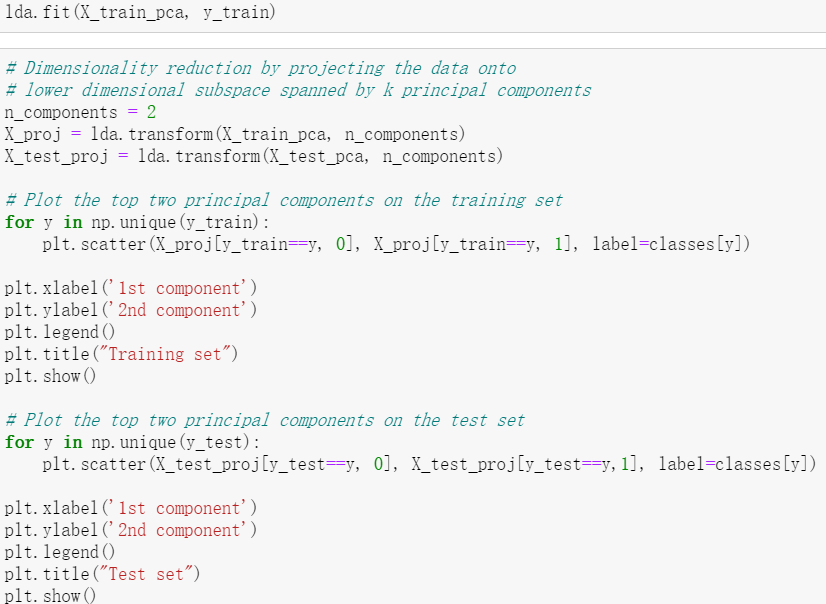


where 𝑁𝑖 is the number of examples in class 𝑖 .



**4.3 - Solving generalized Eigenvalue problem**

Implement methods fit and transform of the LDA class.



**4.4 - kNN with LDA**

Thanks to having the information from the labels, LDA gives a discriminant space where the classes are far apart from each other.

This should help kNN a lot, as the job should just be to find the obvious 10 clusters.

However, as we've seen in the previous plot (section 4.3), the training data gets clustered pretty well, but the test data isn't as nicely clustered as the training data (overfitting?).

Perform cross validation following the code below (you can change the values of **k\_choices** and **n\_choices** to search). Using the best result from cross validation, obtain the test accuracy.

num\_folds = 5

X\_trains, y\_trains, X\_vals, y\_vals = split\_folds(X\_train, y\_train, num\_folds)

k\_choices = [1, 5, 10, 20]

n\_choices = [5, 10, 20, 50, 100, 200, 500]

# n\_k\_to\_accuracies[(n, k)] should be a list of length num\_folds giving the different

# accuracy values that we found when using that value of n and k.

n\_k\_to\_accuracies = defaultdict(list)

for i in range(num\_folds):

# Fit PCA

pca = PCA()

pca.fit(X\_trains[i])

N = len(X\_trains[i])

X\_train\_pca = pca.transform(X\_trains[i], N-c)

X\_val\_pca = pca.transform(X\_vals[i], N-c)

# Fit LDA

lda = LDA()

lda.fit(X\_train\_pca, y\_trains[i])

for n in n\_choices:

X\_train\_proj = lda.transform(X\_train\_pca, n)

X\_val\_proj = lda.transform(X\_val\_pca, n)

dists = compute\_distances(X\_val\_proj, X\_train\_proj)

for k in k\_choices:

y\_pred = predict\_labels(dists, y\_trains[i], k=k)

# Compute and print the fraction of correctly predicted examples

num\_correct = np.sum(y\_pred == y\_vals[i])

accuracy = float(num\_correct) / len(y\_vals[i])

n\_k\_to\_accuracies[(n, k)].append(accuracy)

for n in n\_choices:

print()

for k in k\_choices:

accuracies = n\_k\_to\_accuracies[(n, k)]

print("For n=%d, k=%d: average accuracy is %f" % (n, k, np.mean(accuracies)))

# Based on the cross-validation results above, choose the best value for k,

# retrain the classifier using all the training data, and test it on the test

# data. You should be able to get above 40% accuracy on the test data.

best\_k = None

best\_n = None

# YOUR CODE HERE

# Choose the best k based on the cross validation above

pass

# END YOUR CODE

N = len(X\_train)

# Fit PCA

pca = PCA()

pca.fit(X\_train)

X\_train\_pca = pca.transform(X\_train, N-c)

X\_test\_pca = pca.transform(X\_test, N-c)

# Fit LDA

lda = LDA()

lda.fit(X\_train\_pca, y\_train)

# Project using LDA

X\_train\_proj = lda.transform(X\_train\_pca, best\_n)

X\_test\_proj = lda.transform(X\_test\_pca, best\_n)

dists = compute\_distances(X\_test\_proj, X\_train\_proj)

y\_test\_pred = predict\_labels(dists, y\_train, k=best\_k)

# Compute and display the accuracy

num\_correct = np.sum(y\_test\_pred == y\_test)

accuracy = float(num\_correct) / num\_test

print("For k=%d and n=%d" % (best\_k, best\_n))

print('Got %d / %d correct => accuracy: %f' % (num\_correct, num\_test, accuracy))