CV Assignment 3 Report

The aim of this assignment is to classify 25 different categories of food items using various computer vision and machine learning techniques.

Part 1:

Before we go into linear algebra or clustering, we use a vanilla machine learning classifier. The classifier we use for this work is a support vector machine. We do not actually implement this from scratch, but we use svm_multiclass from Cornell University, and we convert our inputs and feed them into it. We manually parse the output generated to create a confusion matrix.

The baseline method is in SVM.h header, and it follows the same structure as that of NearestNeighbor.h. It has the train() and test() methods, and to add to that, we add another method called format(), which modifies the image data into the classifier format. Apart from that, instead of using the confusion matrix given to us, we create a new python file called confusion.py, which takes the predicted outputs as a parameter and returns a 25-class confusion matrix.

Execute Code:

Compile the code using make.

For training the baseline method: ./a3 train baseline

For testing this method: ./a3 test baseline

The results generated by this method are given below:

The SVM classifier uses a linear model with the value of c 1.0.

For color images:

Size(size * size * 3)-d	Number of Iterations	Time Taken(seconds)	Accuracy (%)
40	874	406.10	17.6
60	898	927.18	20.8
75	875	1377.09	20
90	906	2089.43	19.6

Here, size should be interpreted as size * size * 3 (for 3-color channels), so the size of 40 becomes a 4800-d vector, and a size of 60 becomes a 10800-d vector. The best performance from our tests was when we used the size of 60, i.e., each image is represented in a 10800-d feature space. The number of iterations and time taken for training are given in table above. When the size of 60 was used, it classified 50 images correct out of 250 on the test set. To make things easier for testing for you, we are initializing the size to 10. You can change it in SVM.h variable '`size`'. (Training: 2 minutes, testing: 10 seconds, accuracy 10%).

For Gray-scale images:

Size (size * size)-d	Number of Iterations	Time(seconds)	Accuracy(%)
40	57	15.44	10
60	65	32.02	9.6
75	86	65.79	9.6
90	98	96.6	10
120	119	164.5	10

Here, size should be just interpreted as size * size, and therefore, a size of 40 makes us represent the image as a 1600-d vector. Does color play a role in the baseline method? Looking at accuracies, we must accept that it does. In the previous table, even though we used a size of 40 (4800-d), we got an accuracy of 17.6%. But for grayscale, even if we used a representation of size 120 (14400-d), we are not getting good performance (10%) on the test set.

Part 2:

Part 2.1 :

Principal Component Analysis:

The idea behind using this is because eigenfaces was considered the first best face recognition methodology, and we wanted it to try whether it works for foods or not. The code is initialized to size 5 to make testing for you easier. (Training: 35 seconds, testing: 9 seconds, accuracy: 3.2%).

In this method, we first store our data in a matrix of size (size * size * 3, total number of training files). For example, if the size is 20, the shape of this initial matrix becomes (1200, 1250). Before we jump into eigen decomposition, we need to find the covariance matrix for our data. This results in a matrix of shape (1200, 1200). Now we calculate eigen decomposition using Python Numpy's implementation of eig(). We integrated this python module into our classifier header (PCA.h). This returns 1200 eigen values and 1200 eigen vectors. All these are not important or necessary for the task. We first use half-and-half method and we tested for three different number of principal components. Before that, once we get our eigen vectors, we select *x* components, say half, 600, which reduces it (600, 1200), i.e., 600 vectors each with 1200-d dimensions. Once we have them, we multiply that to our original data matrix, i.e., (600, 1200) multiplied with (1200, 1250), gives a reduced dimensionality of (600, 1250). We continue the SVM routine on this reduce dimensionality measure. The results and observations on eigen values and vectors are discussed below, but before that, if you want to test the implementation, please follow the procedure below.

Compile the code using make.

To train the data: ./a3 train eigen

To test the data: ./a3 test eigen

Please note that if you want to change the number of principal components, you need to change them manually from PCA.h file.

Results:

We tested this methodology of two different sizes (20 and 40) or (1200-d and 4800-d) vectors.

For size 20:

Number of PCs	Accuracy(%)
Elbow (95)	3.6
Quarter (300)	3.2
Half (600)	5.6
Three – Quarters (900)	4.4

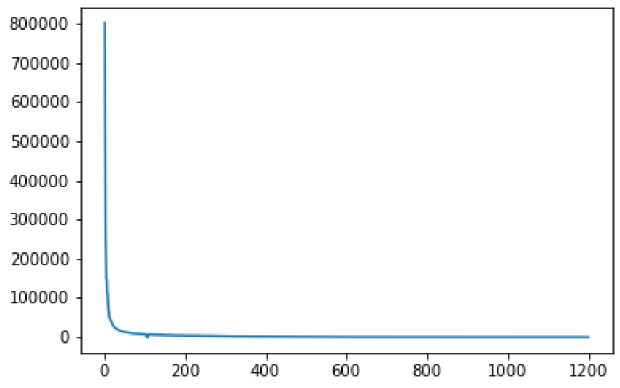
For 40:

Number of PCs	Accuracy (%)
Elbow (80)	3.2
Quarter (1200)	4.0
Half (2400)	3.6
Three – Quarters (3600)	3.2

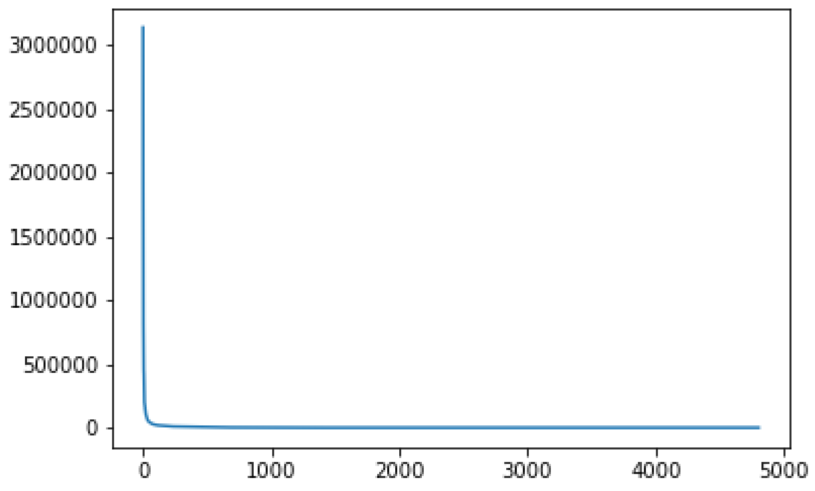
In the above tables, Elbow is defined as the point of sudden drop in eigen values for a size. We believe that since the elbow for 40 is less than 20, that might be the reason for the dip in accuracy.

The question is, how fast do these eigen values change? The answers are illustrated in the plots below.

For size 20: (The elbow is somewhere around 95)

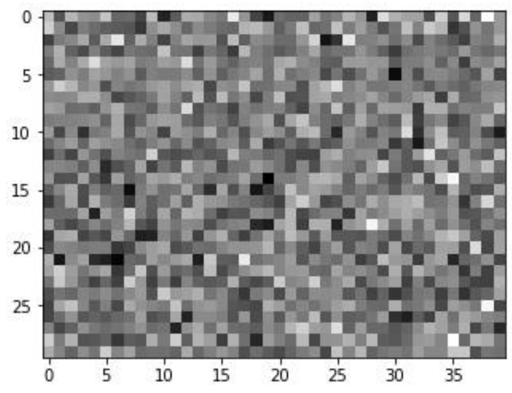


For size 40: (Elbow is around 80)

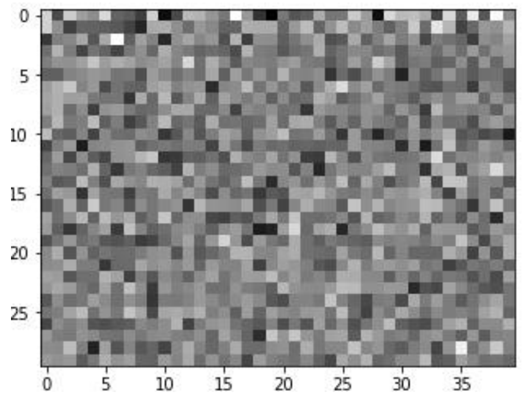


Now that we noticed how these eigen-values change over, now it is time to check who do these values represent. Since we use only small sizes (time constraints), we might not be able to see the food items, but we might be able to see some very low-level patterns.

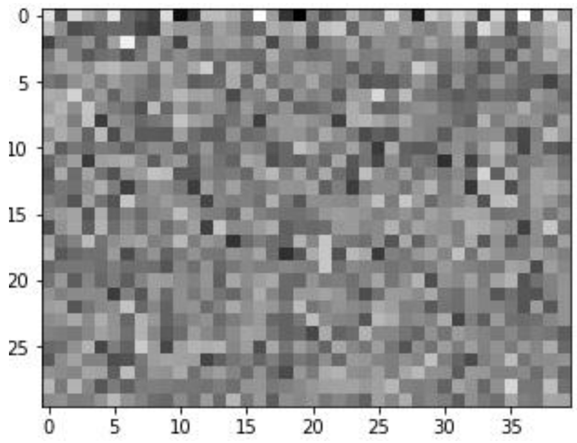
For size 20, since each eigen-vector will have 1200-dimensions, to represent that as an image, we reshape the 1200-d vector in to a rectangle of size 30 * 40. The represents that the top three eigen vectors carry are shown below.



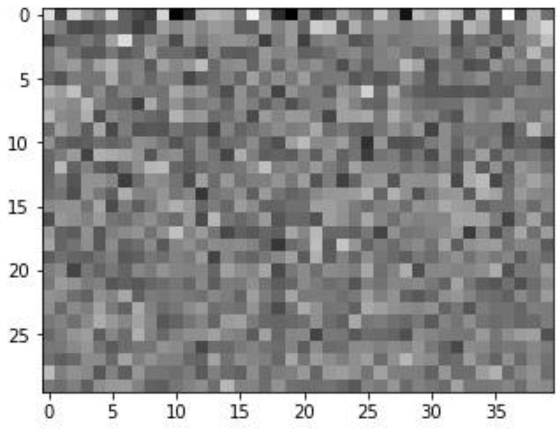
(One)



(Two)



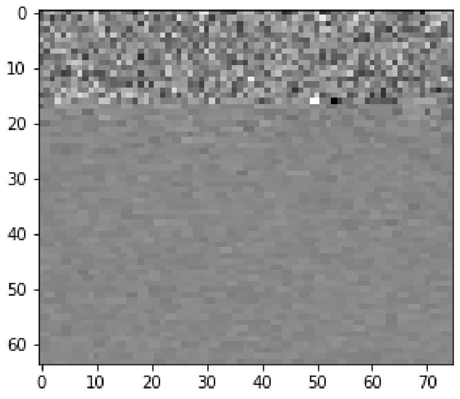
(Three)

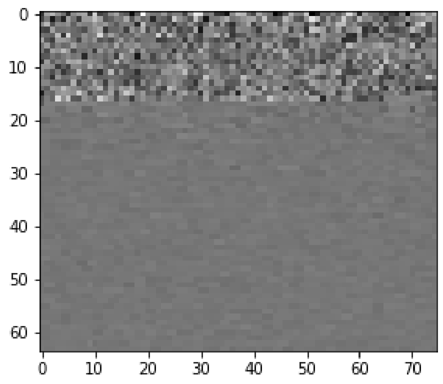


(Four)

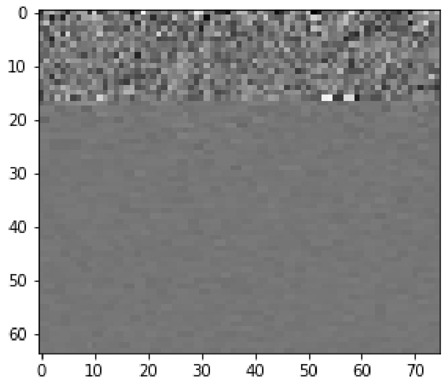
The above images look like some patterns, but they are not highly interpretable, and we believe the reason might be the reduced dimensionality.

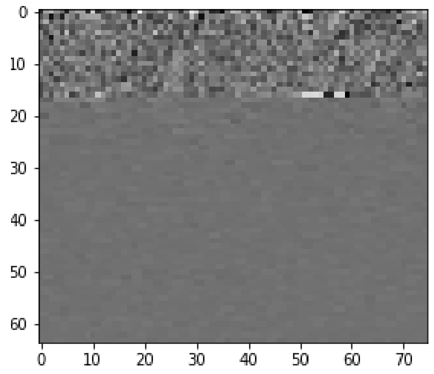
For size 40, each eigen vector will have 4800-d, and therefore we converted them into a rectangle of size 64 * 75.





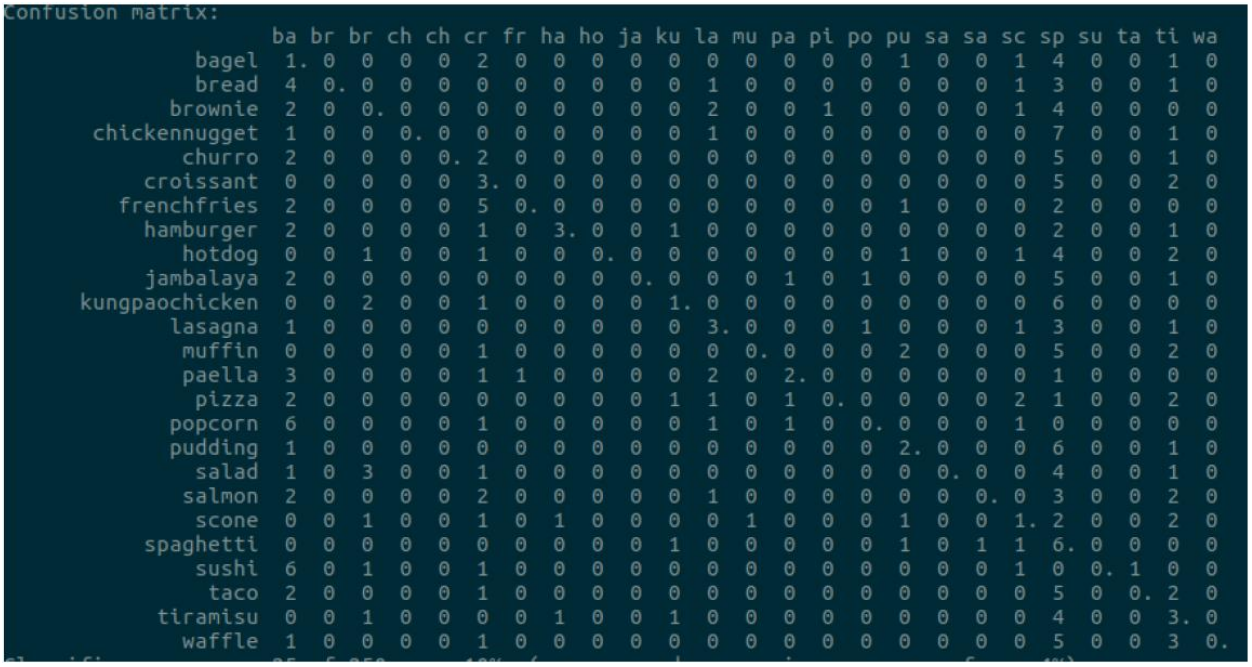
(One) (Two)


(Three)


(Four)

Again, since the dimensionality is very less, the eigen foods are almost un-interpretable. On a positive note, there is some change compared to size 20. Size 20 looked like random values distributed on a rectangle, but when it comes to size 40, we can that through the progression of eigen-vectors, there are subtle changes, that, may be for higher dimensions could be the ones learning more rich representations or patterns.

Part 2.2:
To Execute Code:
Compile the code using make.
• For training the haar method: ./a3 train haar
• For testing this method: ./a3 test haar
The results generated by this method are given below:



```
Confusion matrix:
                ba br br ch ch cr fr ha ho ja ku la mu pa pi po pu sa sa sc sp su ta ti wa
          bagel  1. 0  0  0  0  2  0  0  0  0  0  0  0  0  0  0  1  0  0  1  4  0  0  1  0
          bread  4  0. 0  0  0  0  0  0  0  0  0  1  0  0  0  0  0  0  0  1  3  0  0  1  0
        brownie  2  0  0. 0  0  0  0  0  0  0  0  2  0  0  1  0  0  0  0  1  4  0  0  0  0
   chickennugget 1  0  0  0. 0  0  0  0  0  0  0  1  0  0  0  0  0  0  0  0  7  0  0  1  0
          churro 2  0  0  0  0. 2  0  0  0  0  0  0  0  0  0  0  0  0  0  0  5  0  0  1  0
       croissant 0  0  0  0  0  3. 0  0  0  0  0  0  0  0  0  0  0  0  0  0  5  0  0  2  0
     frenchfries 2  0  0  0  0  5  0. 0  0  0  0  0  0  0  0  0  1  0  0  0  2  0  0  0  0
       hamburger 2  0  0  0  0  1  0  3. 0  0  1  0  0  0  0  0  0  0  0  0  2  0  0  1  0
          hotdog 0  0  1  0  0  1  0  0  0. 0  0  0  0  0  0  0  1  0  0  1  4  0  0  2  0
       jambalaya 2  0  0  0  0  0  0  0  0  0. 0  0  0  1  0  1  0  0  0  0  5  0  0  1  0
   kungpaochicken 0 0  2  0  0  1  0  0  0  0  1. 0  0  0  0  0  0  0  0  0  6  0  0  0  0
         lasagna 1  0  0  0  0  0  0  0  0  0  0  3. 0  0  0  1  0  0  0  1  3  0  0  1  0
          muffin 0  0  0  0  0  1  0  0  0  0  0  0  0. 0  0  0  2  0  0  0  5  0  0  2  0
          paella 3  0  0  0  0  1  1  0  0  0  0  2  0  2. 0  0  0  0  0  0  1  0  0  0  0
           pizza 2  0  0  0  0  0  0  0  0  0  1  1  0  1  0. 0  0  0  0  2  1  0  0  2  0
         popcorn 6  0  0  0  0  1  0  0  0  0  0  1  0  1  0  0. 0  0  0  1  0  0  0  0  0
         pudding 1  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  2. 0  0  0  6  0  0  1  0
           salad 1  0  3  0  0  1  0  0  0  0  0  0  0  0  0  0  0  0. 0  0  4  0  0  1  0
          salmon 2  0  0  0  0  2  0  0  0  0  0  1  0  0  0  0  0  0  0. 0  3  0  0  2  0
           scone 0  0  1  0  0  1  0  1  0  0  0  0  1  0  0  0  1  0  0  1. 2  0  0  2  0
       spaghetti 0  0  0  0  0  0  0  0  0  0  1  0  0  0  0  0  1  0  1  1  6. 0  0  0  0
           sushi 6  0  1  0  0  1  0  0  0  0  0  0  0  0  0  0  0  0  0  1  0  0. 1  0  0
            taco 2  0  0  0  0  1  0  0  0  0  0  0  0  0  0  0  0  0  0  0  5  0  0. 2  0
        tiramisu 0  0  1  0  0  0  0  1  0  0  1  0  0  0  0  0  0  0  0  0  4  0  0  3. 0
          waffle 1  0  0  0  0  1  0  0  0  0  0  0  0  0  0  0  0  0  0  0  5  0  0  3  0.
```

We generate few filers randomly and few that capture the shape of the food items, in place of generating all filters randomly. We normalized the input images and conducted a unit vector normalization on the feature vectors. We also included colors into classification. Even with these parameters we were unable to see an effective accuracy from the model.
Best accuracy observed is 10%.
The confusion matrix of the classifier indicates that many of the images are getting misclassified as spaghetti and croissant. This is understandable from the color and general texture of these food items.

Part 2.3:

The Bag of Words model is used to group features that represent a similar entity together. To do that, we need information about our current image. Information (or) in the sense of computer vision, interest points of an image can be retrieved by SIFT. When SIFT is applied on an image, it returns hundreds or thousands of points, each of which is a 128-d vector. All these points, irrespective of the class they belong, are taken into one set. For the given training set, there are more than three million SIFT points. Now that we have our SIFT vectors, we apply k-means clustering on it. We use OpenCV's implementation of k-means algorithm, with maximum of 20 iterations and only one attempt. Once we have all our means, we save them in OpenCV's StorageFile. Our work is only half-done. We assume that each cluster represents one specific feature of any image. Now that we have some *n* number of clusters, we need to find the histogram of each image. For example, if there are 500 clusters, we use a 500-d histogram to represent our image. Each dimenision, in vague words, represents the normalized value of occurrence of each specific entity. Once we have our 500-d representations for all the images, we continue with our regular SVM routine.

Execution Code:

Compile the code with make

For training using Bag of Words: ./a3 train bow

For testing using Bag of Words: ./a3 test bow

To change the size of clusters, you have to open Bagofwords.h and change the number in the variable '`clusters`'.

This takes a long time to run, since we have 3349128 SIFT points. Therefore, we provided the already trained models for you. If you hit on train, it will run for more than two hours, to collect SIFT points, do clustering, and SVM. Therefore, we recommend to use our pre-trained model for testing. Training takes more than three hours for predefined clusters (500), but we have provided the models for you. Just test the model using the structure given below, it should take around 9 minutes (Accuracy : 12%).

Results:

Clusters	SIFT Points Considered	Accuracy
50	349128	4%
500	1022000	12%

We wanted to try mode clusters and combinations, but training was taking too much time. But for the two models we tested, the 500 cluster one is giving good results compared to the entirety of PCA. But if we had to compare the baseline and other models (PCA and BOW), with the current observations, we would prefer vanilla SVM over PCA and BOW (for time and performance).

Q3

How to run:

1. make overfeat

2. ./a3 train deep
   ./a3 test deep

Procedure -

1. Firstly the data is downloaded using download_weights.py

2. We use image size for scaling image to small to the size 231x231

3. As given in q3, we use overfeat to detect features

4. Then we apply SVM on the images

5. We use svm_classify & svm_multiclass_learn to compare features and to get a final result

6. We get an accuracy of around 70%

Comparison with part 1 and 2–

As opposed to part 1 and part 2, the pretrained network gave a much higher accuracy and worked exceedingly well with SVM

References:

1) Cornell University SVM multi-class
2) OpenCV
3) OverFeat