Australian National University

Computer Lab 2 Report

ENGN**4528** Computer Vision

by Abhinav Pandey

(U6724645)

Task 1 : Harris Corner Detector

Introduction

A corner can be defined as a point on the junction of two or more edges. Although corners are only a small percentage of the image, these *interest* points contain the most important features of images which are invariant to scale, rotation, and illumination. [1] Determining the corners in an image is an essential step in finding these features. This is where the Harris Corner Detector comes into the *picture*.

The Harris Corner Detection algorithm can be divided into a series of steps :

1. Convert colour image to grayscale
2. Calculate the spatial derivatives of the grayscale image
3. Create the Second Moment Matrix, **M**
4. Compute the Harris Response, **R** (measure of "cornerness") at every pixel.
5. Non-maximum suppression and Thresholding [1]

For Task 1, we implemented Corner Detection (in *Python*) on these four input images :

- Implementation of Non-Maximum suppression:
    - Divide input image into equal-sized non-overlapping tiles of a small size (eg. For image Harris.jpg, window_size =(5, 5))
    - Select the pixel with the highest value for the responsiveness, R
    - Retain only that pixel in all of the surrounding pixels belonging to that window

Fig. 1. Inputs for Corner Detection

4. Function results on the provided test images. Display your results by marking the detected corners on the input images.

Fig. 2. Corner Detection results on Image 1

Fig. 3. Corner Detection results on Image 2

Fig. 4. Corner Detection results on Image 3

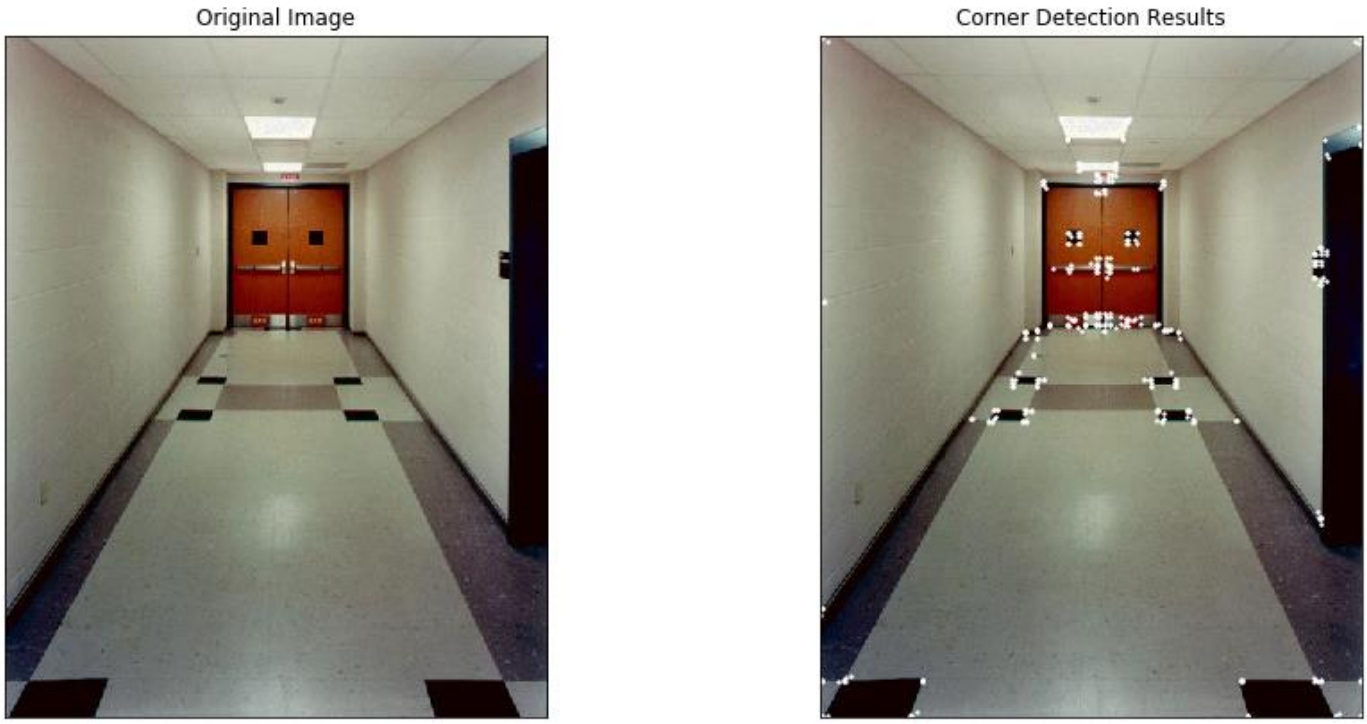


Fig. 5. Corner Detection results on Image 4

5(a). Compare your results with that from python's built-in function

Fig. 6

Harris Corner Detection on Image 1 (thresh = 0.003)

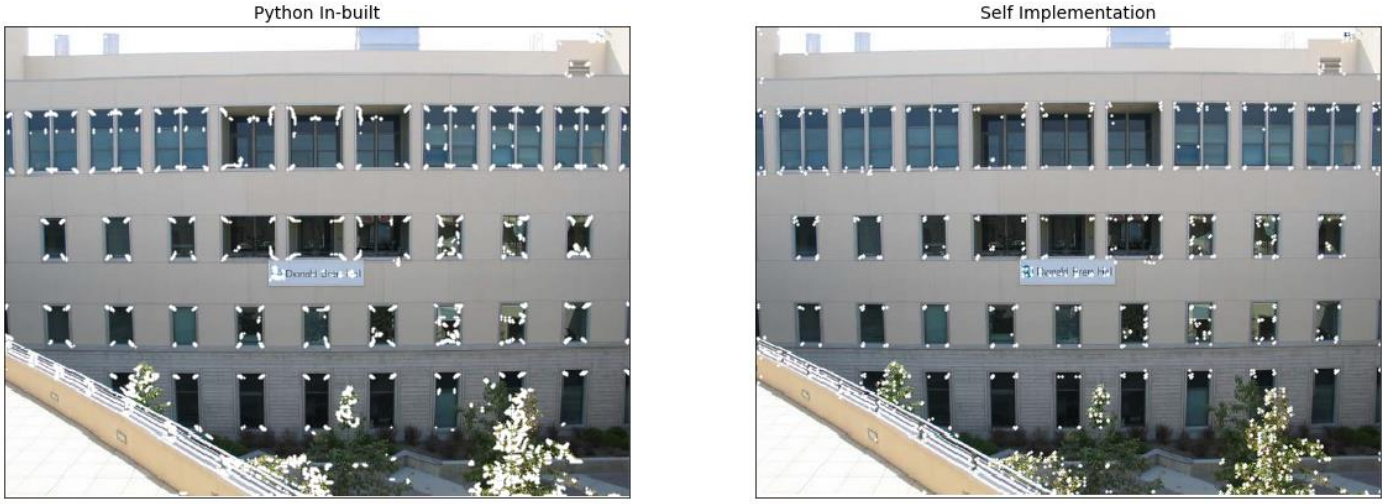


Fig. 7

Harris Corner Detection on Image 2 (thresh = 0.01)

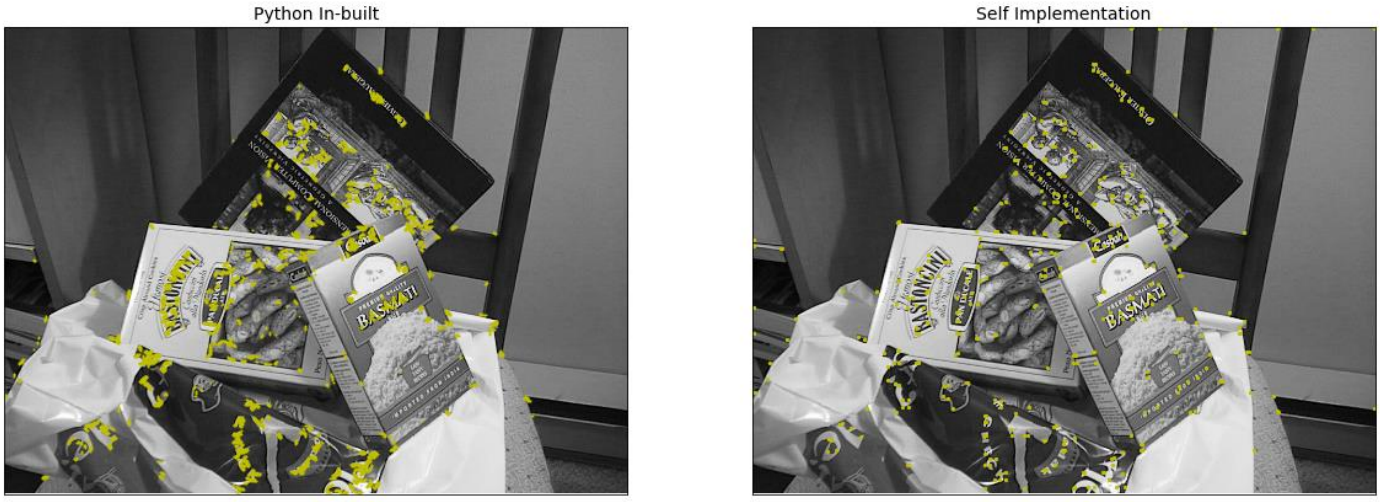
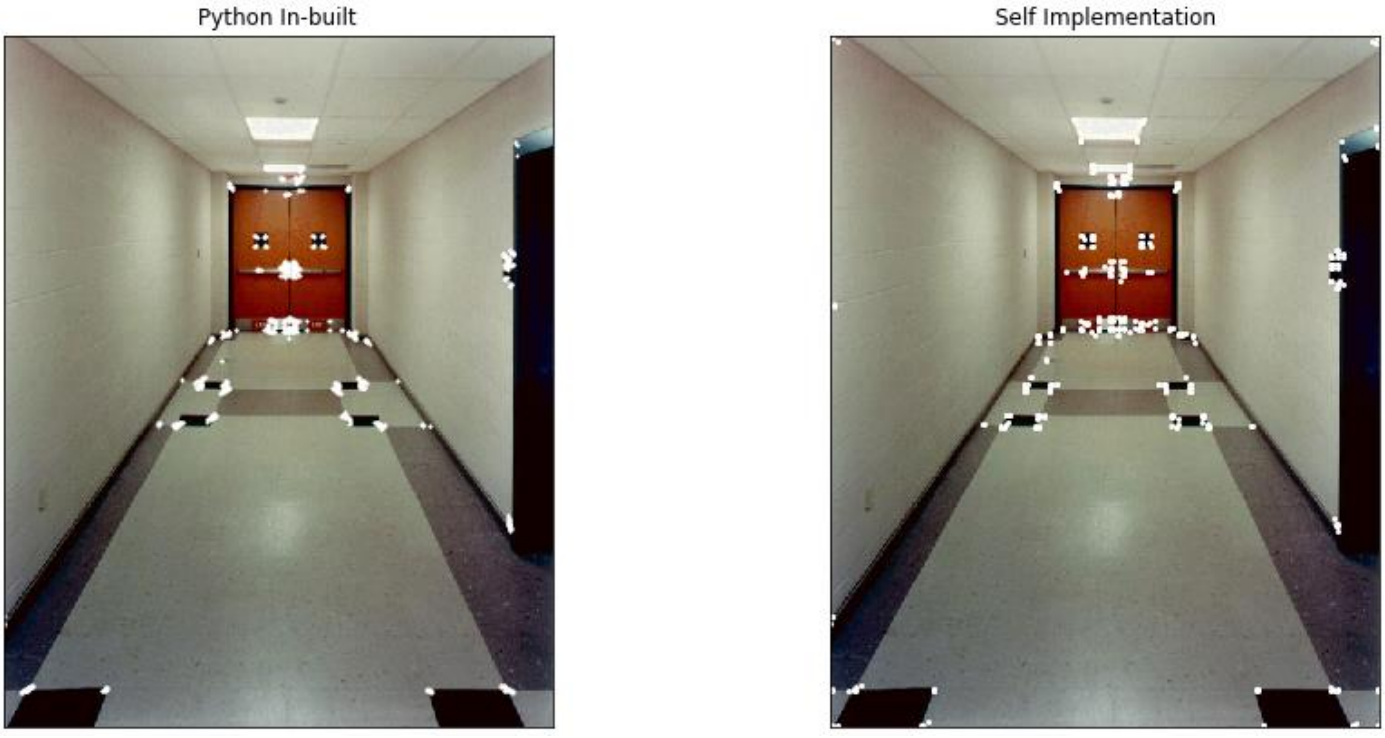


Fig. 8

Harris Corner Detection on Image 3 (thresh = 0.01)

Fig. 9

Harris Corner Detection on Image 4 (thresh = 0.01)



5(b). Discuss the factors that affect the performance of Harris corner detection

The Harris Corner detection results depend on the different parameters that are used in calculating the Responsiveness function and on the selected threshold value. Another factor that affects the corner detection results would be the implementation of non-maximum suppression. For instance, the results would be different when we select the 10 highest values in a subset window of the image compared to when we select only the 2 maximum values.

Task 2 : K-Means Clustering and Color Image Segmentation

K-Means Clustering is an Unsupervised Learning methodology and is a common tool for image segmentation.

In this task, we represent each pixel of an image in a 5D matrix consisting the following the features:

1. The lightness value, *L**
2. The green–red component, a*
3. The yellow-blue component, b*
4. X coordinate of the pixel
5. Y coordinate of the pixel

2.1. For this task, we implemented the K-Means algorithm from scratch. These are the steps for the algorithm –

1. Choose the number of clusters K.
2. Select at random K points, the centroids(not necessarily from your dataset).
3. Assign each data point to the closest centroid → that forms K clusters.
4. Compute and place the new centroid of each cluster.
5. Reassign each data point to the new closest centroid. If any reassignment . took place, go to step 4, otherwise, the model is ready. [3]

2.2. Apply your K-means function to color image segmentation. Please compare segmentation results
(1) using different numbers of clusters
(2) with and without pixel coordinates

(1) Segmentation results for different number of clusters

For this task we will be testing and comparing K-Means image segmentation results on two different images viz. “Peppers” and “M&M”.

Fig. 10 Original Image for Color Segmentation ("Peppers")

Fig. 11 Segmentation results with different number of clusters (Peppers)

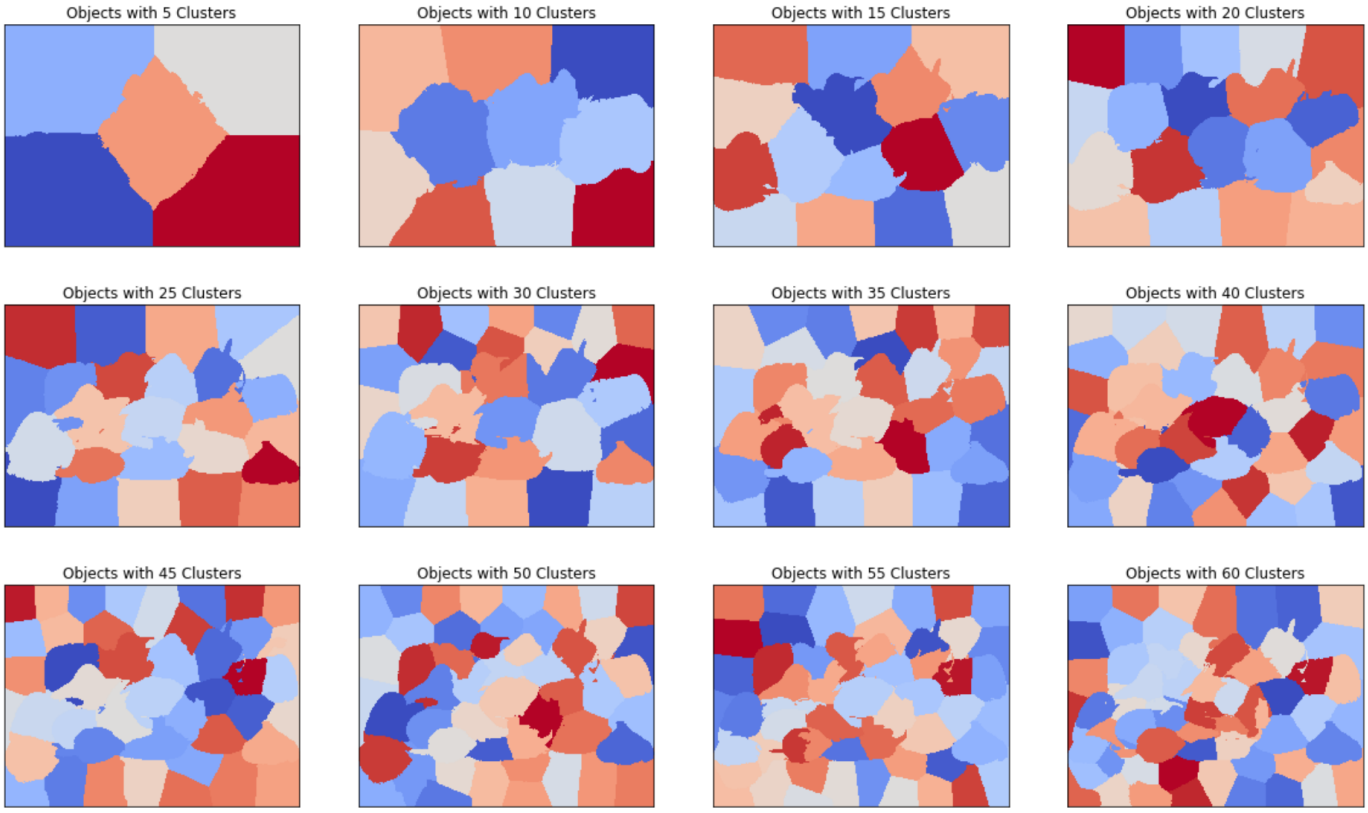
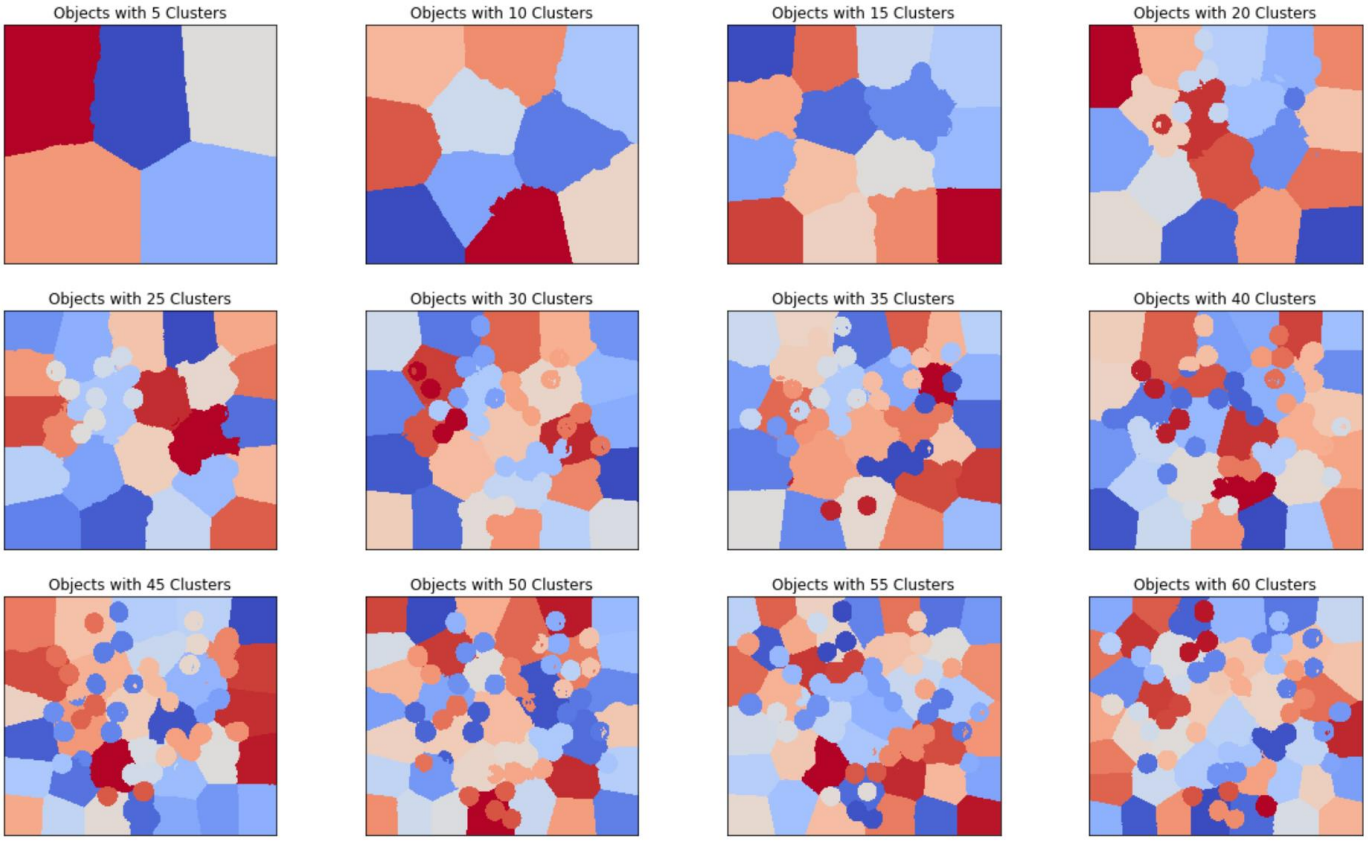


Fig. 12 Original Image for Color Segmentation ("M&M")

Fig. 13 Segmentation results with different number of clusters (M&Ms)



Observations :

- For lower values of k, the clusters are not very helpful in identifying important segments of the image. Since, we're also using the x, y coordinates of each pixel, the clustering algorithm places importance on the position as well as colour features while performing the expectation maximisation algorithm. This leads to insignificant clusters being formed for values of k = 5 or 10.

- We see drastic improvement in segmentation results as k increases. For instance, when k = 25, we can observe that the algorithm is able to correctly identify individual entities (Eg. The garlic on the right, or the tomato on the left)

- However, for much higher values of k, such as k = 60, the segmentation results become noisier, and it becomes very difficult to identify individual entities in the image.

(2) Segmentation results with and without Pixel Coordinates

The image segmentation results differ significantly in the case we do not use x, y coordinates from when we do. This can be easily seen by visually comparing the results in Fig. 14 and Fig. 15.

a) Peppers :

Fig. 14

KMeans without X and Y coordinates

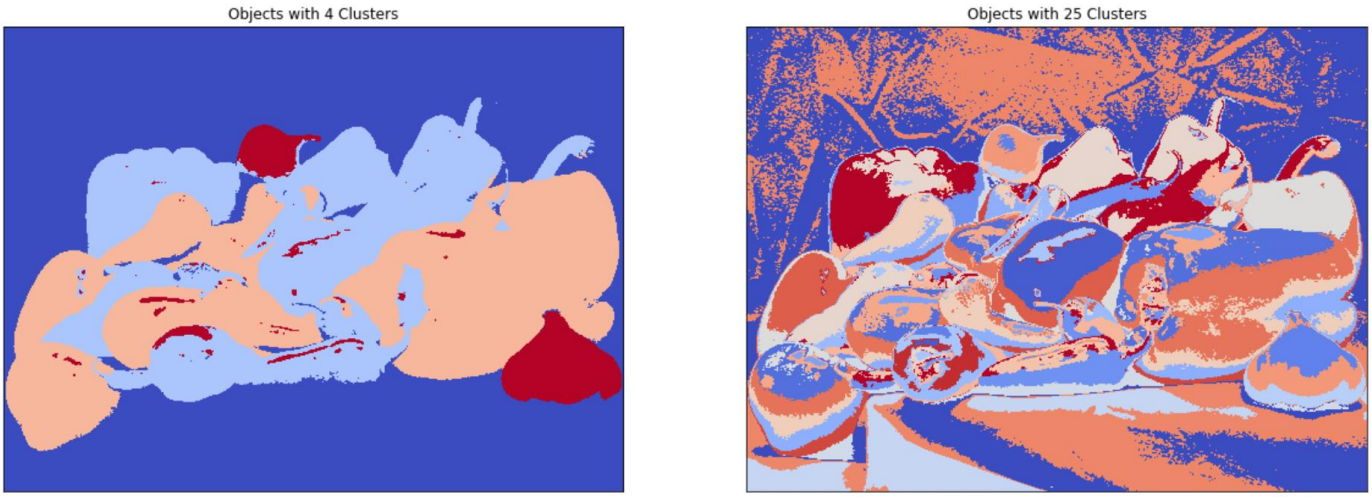
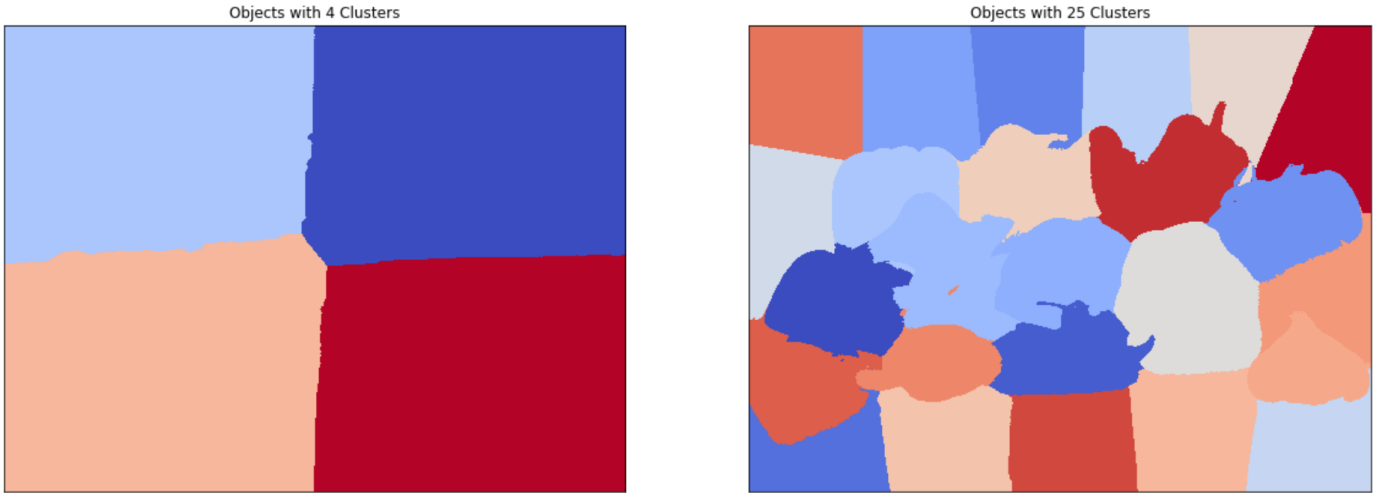


Fig. 15

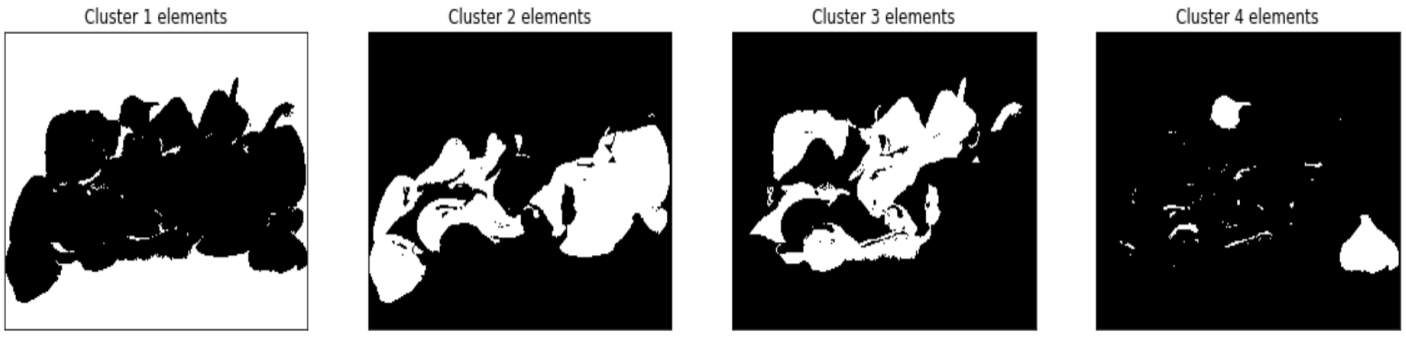
KMeans with X and Y coordinates



Observations :

- When k = 4, the K-Means segmentation results are extremely meaningful when the data does not contain positional information of the pixel (the individual cluster elements can be observed in Fig.16) On the other hand, when the x, y coordinates are included, the clusters that are formed do not add any value in terms of segmentation. It is obvious that those pixels which lie close to one another, regardless of their colour, are clustered together.

Fig. 16 Individual Cluster elements for k = 4 (represented by white regions)



- However, we get more meaningful clusters for k = 25, when the pixel coordinates are included. This can be easily seen by looking closely at Fig. 15. We can identify peppers and garlic, easily. This is not the case in Fig. 14. For instance, the garlic on the right belongs to 3 different clusters.

b) M&Ms:

Fig. 17
KMeans without X and Y coordinates

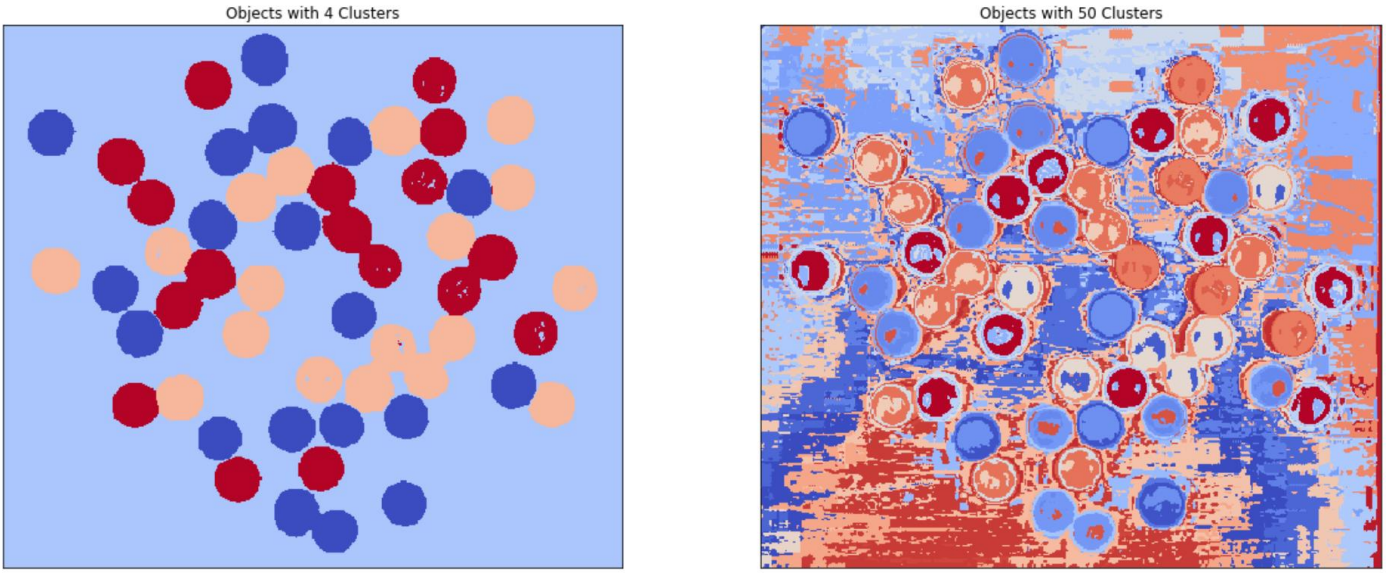
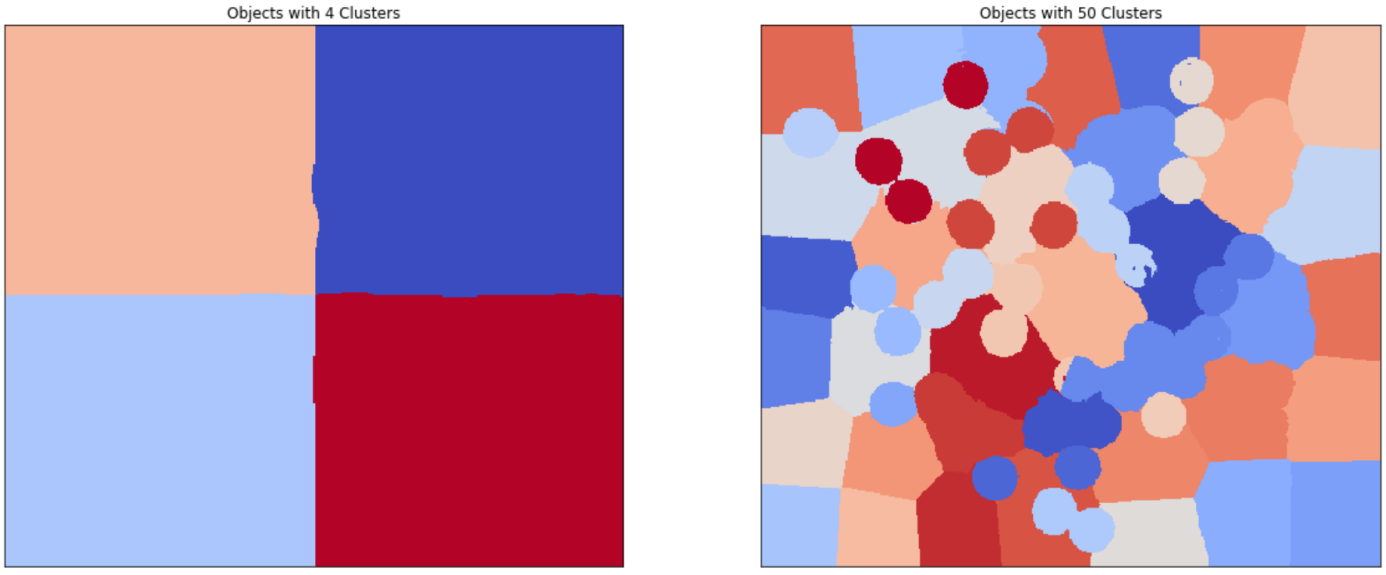


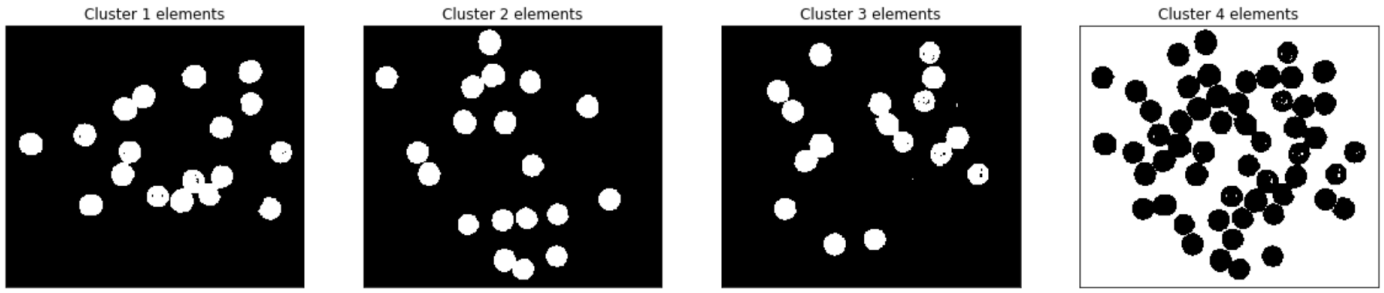
Fig. 18
KMeans with X and Y coordinates



Observations :

- Here, too, when k = 4, the K-Means segmentation results are extremely meaningful when the data does not contain positional information of the pixel (the individual cluster elements can be observed in Fig.19) On the other hand, when the x, y coordinates are included, the clusters that are formed, for k=4, do not add any value in terms of segmentation. This reaffirms our earlier observation that those pixels which lie close to one another, regardless of their colour, are clustered together.

Fig. 19 Individual Cluster elements for k = 4 (represented by white regions)



- On the other hand, in the case when k = 50, the 5D representation of the data forms more meaningful clusters/ image segments as can be observed from the formation of similarly coloured circles in Fig. 18.

(3) Implement K-means++ as a new initialization strategy.

The idea behind the K-means++ algorithm is to find initial centroids such that these centroids are as far away from each other, and therefore much likely to lie in separate clusters right before the K-Means even begins.

The key steps of my implementation of the kmeans++ algorithm are as follows:

1. Select a random datapoint from the data
2. Calculate the distances of all the datapoints to the centroid closest
3. Select the datapoint with the maximum distance as a new centroid
4. Repeat steps 2 and 3 until k centroids are obtained

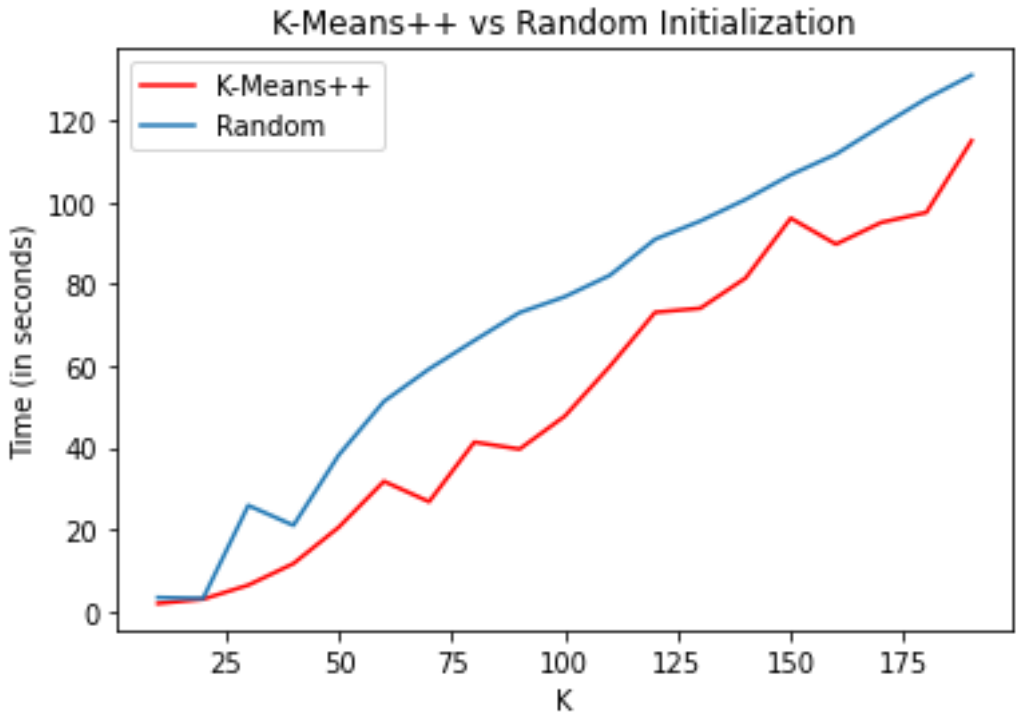
```
# Step 1: Select a random datapoint from the data as a centroid
centroids = data[np.random.randint(low=0, high=nrows),:]
centroids
for iteration in range(k-1):
    # Step 2 : Calculate distance of all datapoints to the closest centroid
    # a) Calculate distance from each centroid
    distance = np.array([dist_from_centroid(data, centroids[i]) for i in range(len(centroids))])
    # b) Store the smallest centroid distance for each datapoint
    distance = distance.min(axis=0)
    # Step 3 : Select the datapoint with maximum distance from initial centroid
    new_centroid = data[distance.argmax()]
    centroids = np.vstack((centroids, new_centroid))
```

Kmeans++ vs Random Initialization for Kmeans Clustering

The following graph(Fig.20.) compares the convergence time of the K-Means algorithm with the Kmeans++ initialization strategy and the Random Initialisation strategy for the **Peppers** image for number of clusters, k in the range 2 to 200.

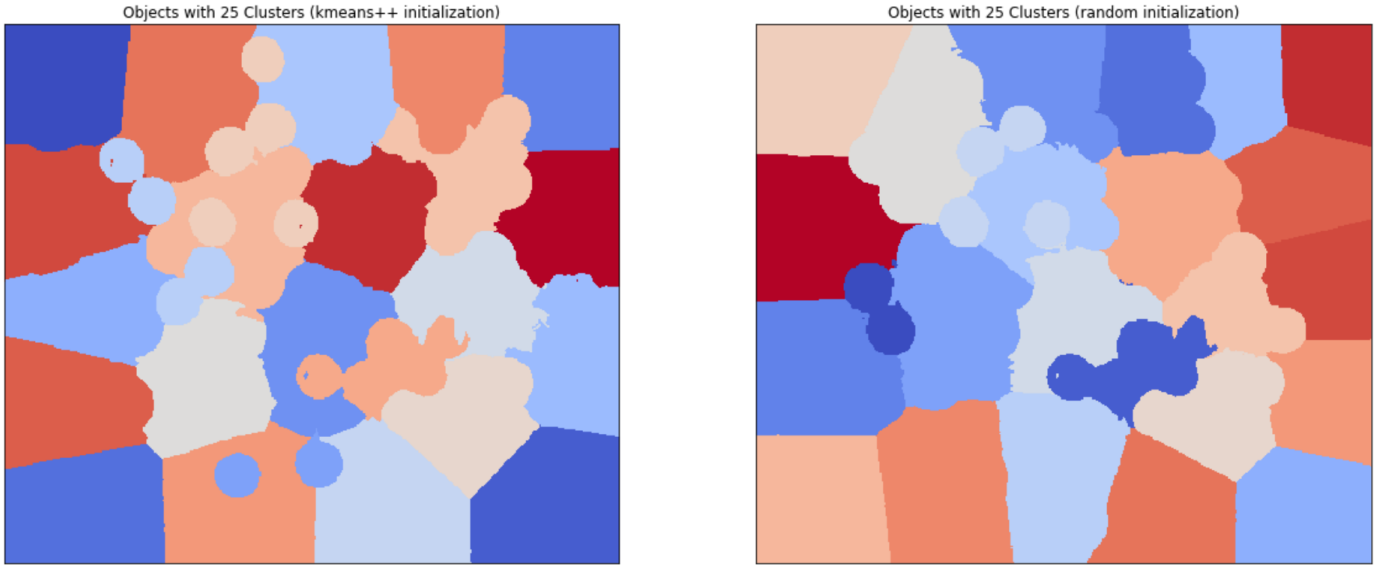
We can see that the Kmeans++ initialization unanimously leads to a faster convergence of the segmentation/clustering algorithm.

Fig. 20 Convergence Time Comparison (Peppers)



Also, we can see from Fig. 21 that the Kmeans++ initialisation strategy leads to a better segmentation result, in fewer iterations (maximum number of iterations permitted = 25).

Fig. 21. Segmentation results with different initialization strategies (max_iteration=25)



Task 3 : Face Recognition using Eigenface

3.1. Why is alignment necessary for Eigen-face?

Eigenfaces refers to an appearance-based approach to face recognition that seeks to capture the variation in a collection of face images and use this information to encode and compare images of individual faces. [2] The eigenface approach for face detection relies on pixel to pixel similarity for dimensionality reduction and feature extraction. Therefore, it performs best when the eyes, nose, lips, etc. in all images are approximately in the same pixel region and have the same orientation.

3.2. Facial Recognition using eigenfaces can be done by following a series of steps :

1. Read and store training in images in a 2D big data matrix. (Each image is represented as a single data point)
    - The dataset contains a total of 135 images of 15 individuals, consisting of 9 images of each of these individuals.

```
# List of filepaths for all training images
im_filepaths = glob.glob("trainingset/*")
# Add all image matrices to a numpy array to create a big data matrix of images
training_images = np.array([plt.imread(image).flatten() for image in im_filepaths])
print(training_images.shape)
>> (135, 45045)
```

Fig. 22. Sample images from the training data

2. Normalizing the data and performing dimensionality reduction (using Principal Component Analysis)

The steps involved are :

- Determine the **mean face** from all training images
- **Normalise the data** by removing common features (i.e. subtracting the mean face from training images)
- Computing a **variance-covariance** matrix using each of the 135 training images
- Computing **eigenvalues and eigenvectors** from the above variance-covariance matrix.

The implementation of these steps are as follows :

- Determine the **mean face** from all training images

```
# Determine the average of all the training face images
avg_face = training_images.mean(axis=0)
```

Fig. 23 Mean Face from Training Images

- **Normalise the data** by removing common features (i.e. subtracting the mean face from training images)

```
# Normalise the training data by subtracting the mean image from each one
normalised_training_images = training_images - avg_face
```

Fig. 24. Training Images after Normalising

3.2.2. Find a faster way to compute eigen values and vectors, explain the reason?

Each training image has a size of **231 x 195** = 45045. To determine the eigenvectors, we would require computing a 45045 x 45045 matrix and 45045 eigenvectors, which is computationally expensive.

A faster way to compute these eigenvectors is to first compute the variance-covariance matrix for all the 135 images and then determine the eigenvalues and eigenvectors of this matrix. Although, these eigenvectors are not of size 45045, they have a direct relationship with the 45045 size eigenvectors and thus can be used in their place to perform dimensionality reduction.

- Computing the **variance-covariance** matrix using each of the 135 training images

```
# Compute the covariance matrix for the normalised images
cov_matrix = np.cov(normalised_training_images)
print(cov_matrix.shape)
>>  (135, 135)
```

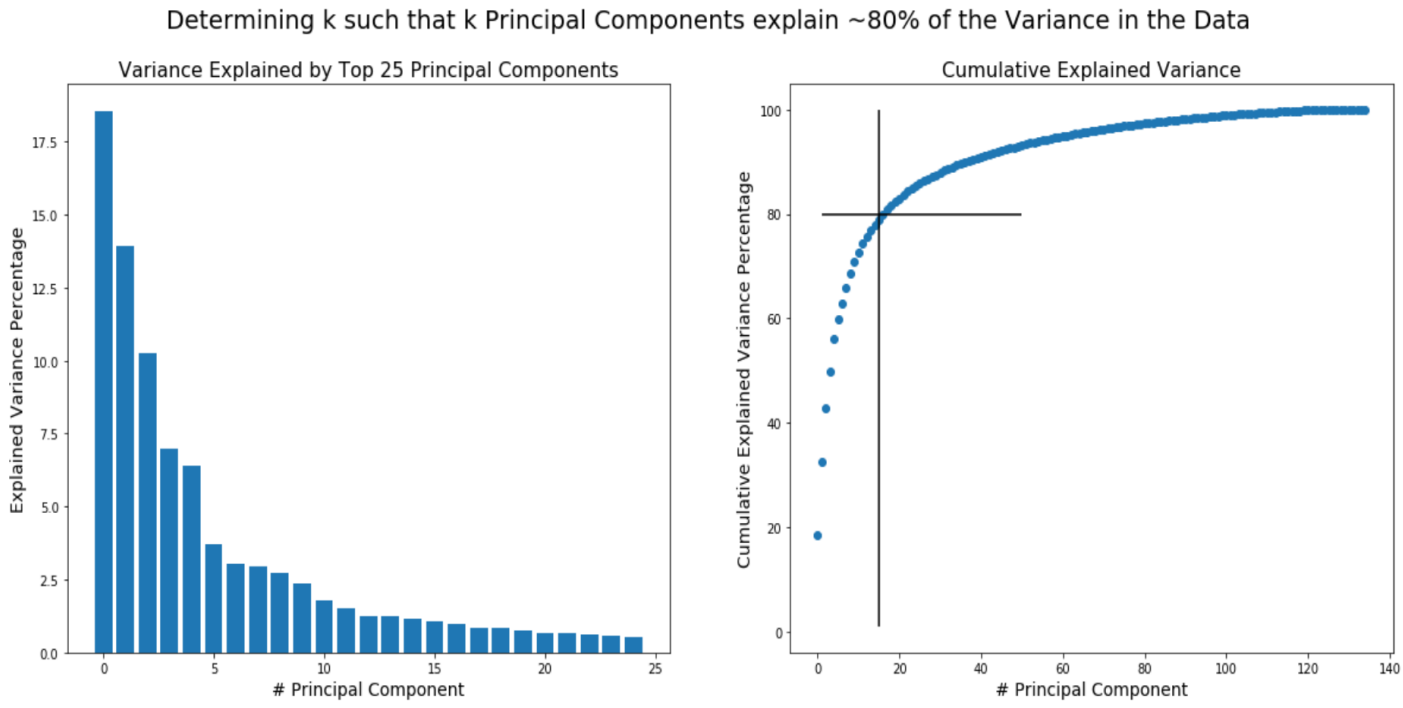
- Computing **eigenvalues and eigenvectors** from the above variance-covariance matrix.

```
# Calculate the eigen-values and eigen-vectors of the covariance matrix
eig_values, eig_vectors = np.linalg.eig(cov_matrix)
# Sort eigenvalues and eigenvectors (in descending order of eigenvalues)
desc_sorted_index = eig_values.argsort()[::-1]
eig_values = eig_values[desc_sorted_index]  # Sorted eigenvalues
eig_vectors = eig_vectors[:, desc_sorted_index] # Corresponding eigenvectors
```

3. Determine the top k principal components and visualize the top-k eigen- faces.

Here, I have selected **k** = 15. As we can see in the figure below, approximately 80% of the variance in the training data can be explained using the first **15 Principal Components** (eigenvectors), which is an acceptable threshold for information retention during dimensionality reduction. In other words, these 15 Principal Components alone capture 80% of the variance in the data. Therefore, we can use just these top 15 features for our Face Detection System and get the approximately the same recognition accuracy, with the added advantage of faster computation.

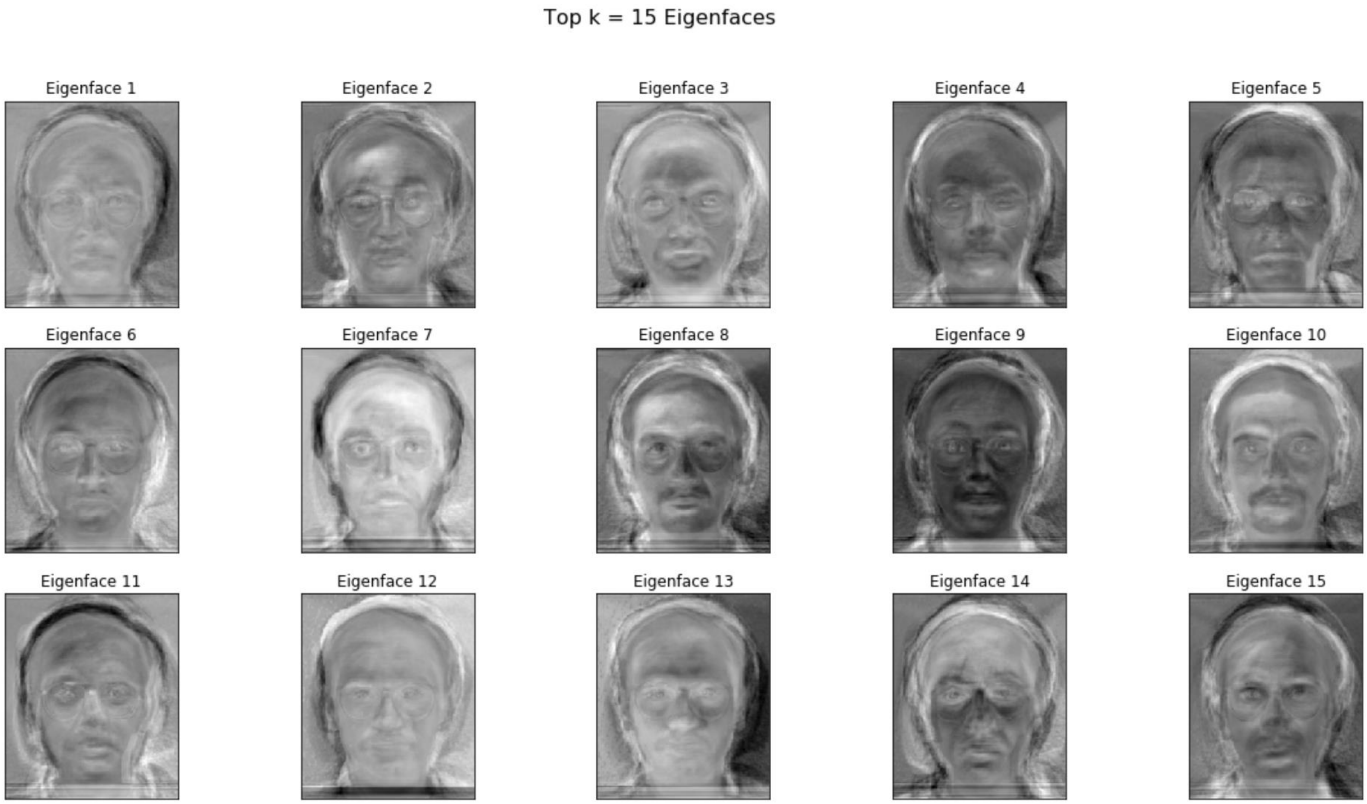
Fig. 25



The top k = 15 eigenfaces can computed by computing the projection of the normalised training images on the basis spanned by the first 15 Principal Components. The code is as follows :

```
# Assign k, where k is the number of Principal Components

k = 15

# Select the first k Principal Components (eigen-vectors)

k_components = eig_vectors[:k, :]

# Determine the eigen-faces using the first k Principal Components

eigen_faces = np.dot(k_components, normalised_training_images)
```

Fig. 26



The final step of the training process is to represent our training data as the weighted sum of the top k =15 eigenfaces. This enables us to create a *model*, which can be used to recognize unseen face images

```
# Project eigen-faces onto the subspace spanned by eigen-vectors

model_weights = np.dot(normalised_training_images, eigen_faces.T)
```

4. **For each of the 10 test images in Yale-Face find out which three face (training) images are the most similar.**
    - **Show these top 3 faces next to the test image.**
    - **Analyse the recognition accuracy.**

Using the above model, we can determine which images in the training data have the most resemblance to each test image. We can do this by :

I. calculating the Euclidean distance of the normalised test image projections to the normalised training image projections on the basis spanned by the **k** Principal Components.
II. Identifying the 3 closest training images for each test image

The code for testing looks like this -

```
# Vectorise test images
test_images = np.array([plt.imread(image).flatten() for image in test_data_filepaths])
# Normalise the test data by removing the mean image features
normalised_test_images = test_images - avg_face
# Project test images on the k Principal Component subspace
test_weights = normalised_test_images @ eigen_faces.T
# Calculate distance of all test images from each training image in the new subspace
dist_from_train_images = np.linalg.norm(test_weights[:,None] - model_weights, axis=2)
# Determine the indexes of 3 "most similar" training images (those having least distances from each test image)
test_preds = dist_from_train_images.argsort(axis=1)
test_preds = [test_preds[i][:3] for i in range(len(test_preds))]
```

The figure (APPENDIX Figure 1(a) and (b) for prediction results can be found at the end of the report.

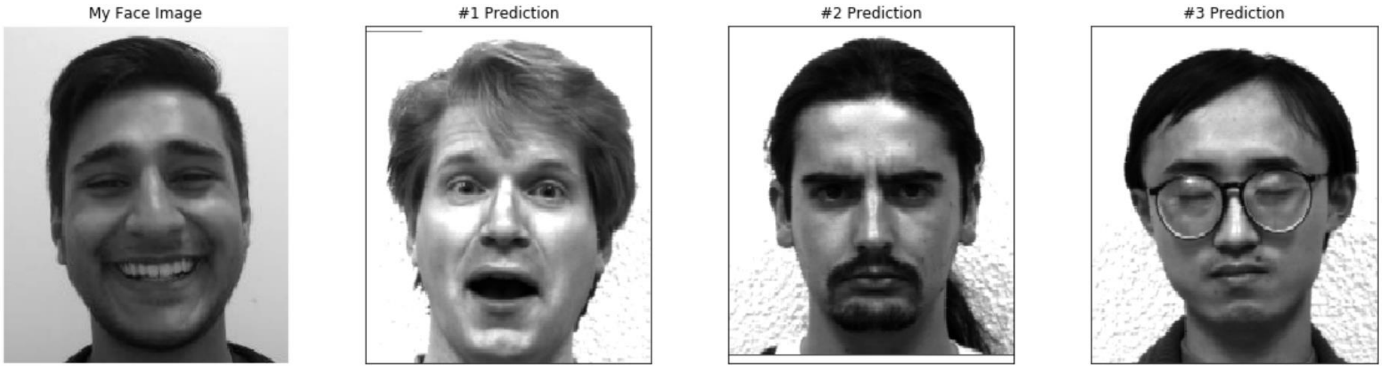
Table. 1. Overview of the recognition accuracy

	# Correct Predictions	# Incorrect Predictions	# Total	Accuracy
1st Prediction	10	0	10	100 %
2nd Prediction	9	1	10	90 %
3rd Prediction	9	1	10	90 %
OVERALL	**28**	**2**	**30**	**93.34 %**

5. Read in one of your own frontal face images. Then run your face recognizer on this new image. Display the top 3 faces in the training folder that are most similar to your own face.

The image below shows the predictions for my own face. We can see that the results are unsatisfactory, which is not surprising since the predictions can only be face images which were included during the training of the face recognition system.

Fig. 27. Predictions for new face image (when system is not trained on similar images)



6. **Repeat the previous experiment by pre-adding the other 9 of your face images into the training. Display the top 3 faces that are the closest to your face.**

Steps 1 to 5 were exactly repeated on an updated dataset (containing 9 new images).

<u>The figure (APPENDIX Figure 2(a) and (b)) for prediction results can be found at the end of the report.</u>

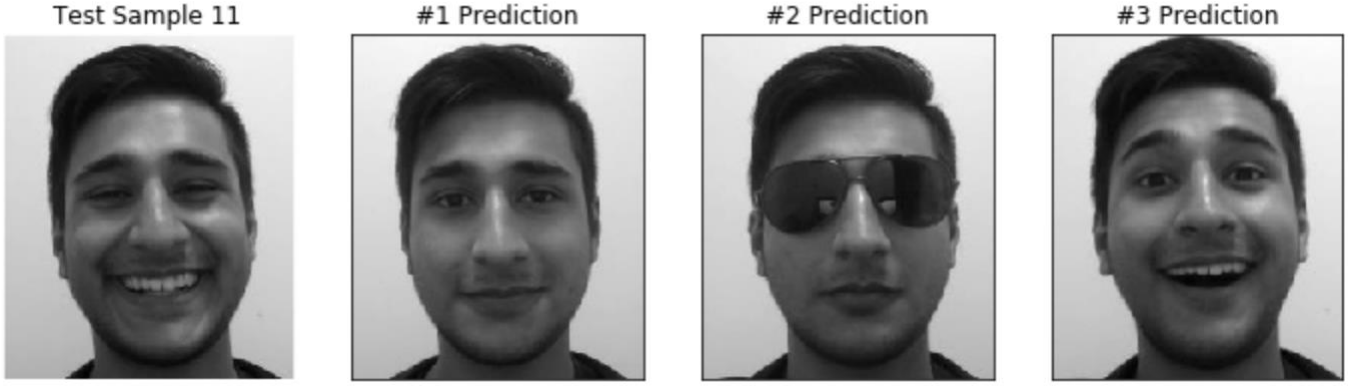
Table 2 contains an overview of the performance of the new face detection system.

Table. 2. Overview of the recognition accuracy (with UPDATED training set)

	# Correct Predictions	# Incorrect Predictions	# Total	Accuracy
1st Prediction	11	0	11	100 %
2nd Prediction	10	1	11	90.91 %
3rd Prediction	9	2	11	81.81 %
OVERALL	**30**	**3**	**33**	**90.91 %**

This new system is correctly able to recognise the new face image with 100% accuracy (for top 3 predictions).

Fig. 28. Predictions for new face image (when system is trained on similar images)



Observations :

- It is interesting to note that the overall recognition accuracy is worse for this new system compared to earlier version. One probable reason for this could be incorrect alignment of the new images compared to the previously existing images in the training set.
- Both the systems provide incorrect 2nd and 3rd predictions for Test Sample 5. The loss of information during PCA might be causing this failure. This is a scenario where an individual has to weigh the trade-off between reduced dimensionality (hence, faster computations) and information loss.

References

1. En.wikipedia.org. 2020. *Harris Corner Detector*. [online] Available at: <https://en.wikipedia.org/wiki/Harris_Corner_Detector> [Accessed 16 May 2020].
2. Sheng Zhang and Matthew Turk (2008) Eigenfaces. Scholarpedia, 3(9):4244.
3. Medium. 2020. *Introduction To Image Segmentation With K-Means Clustering*. [online] Available at: <https://towardsdatascience.com/introduction-to-image-segmentation-with-k-means-clustering-83fd0a9e2fc3> [Accessed 17 May 2020].

APPENDIX

Figure 1. a) Face Recognition Results

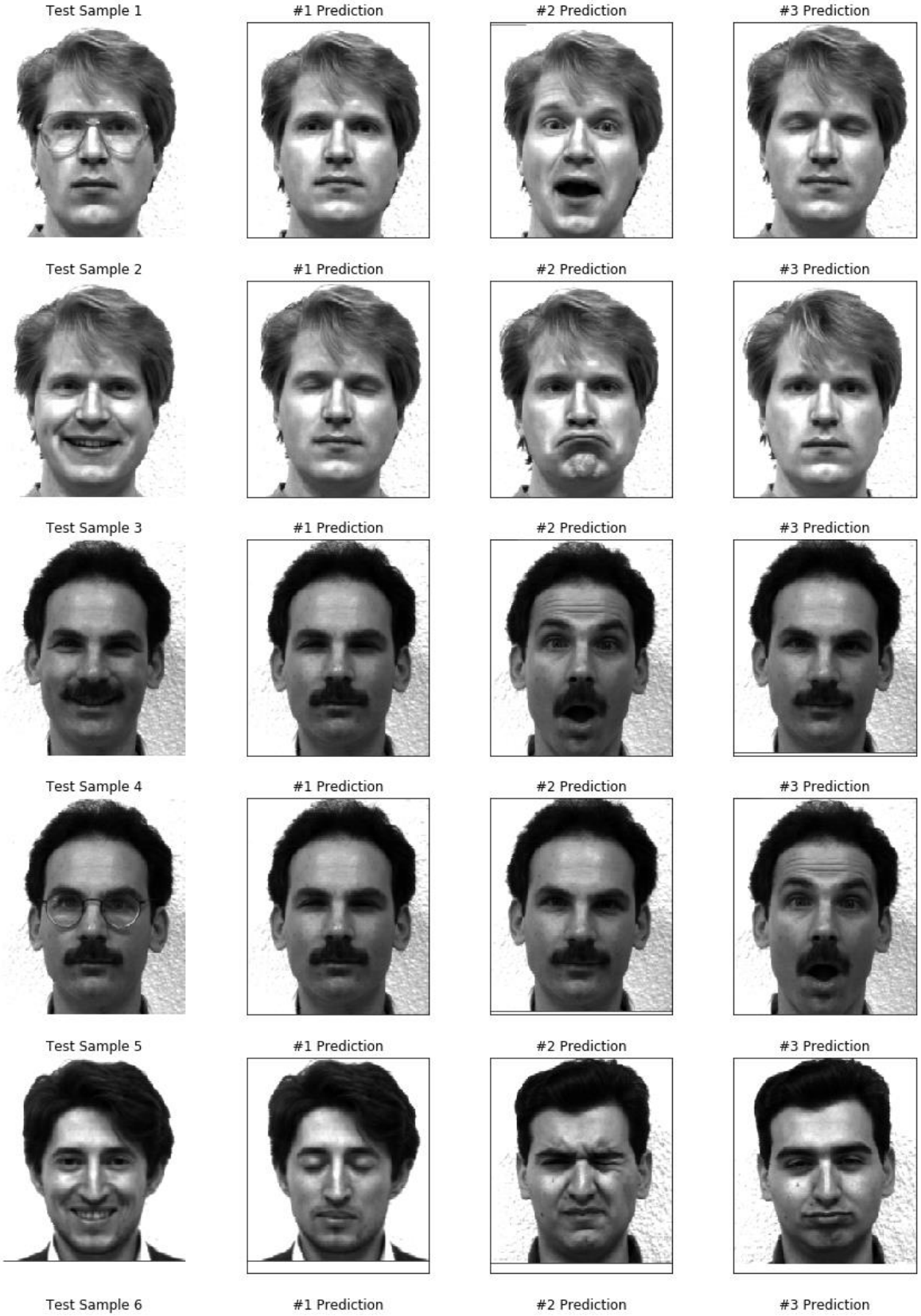


Figure 1. b) Face Recognition Results

Figure 2. a) Face Recognition Results (with updated Training data)

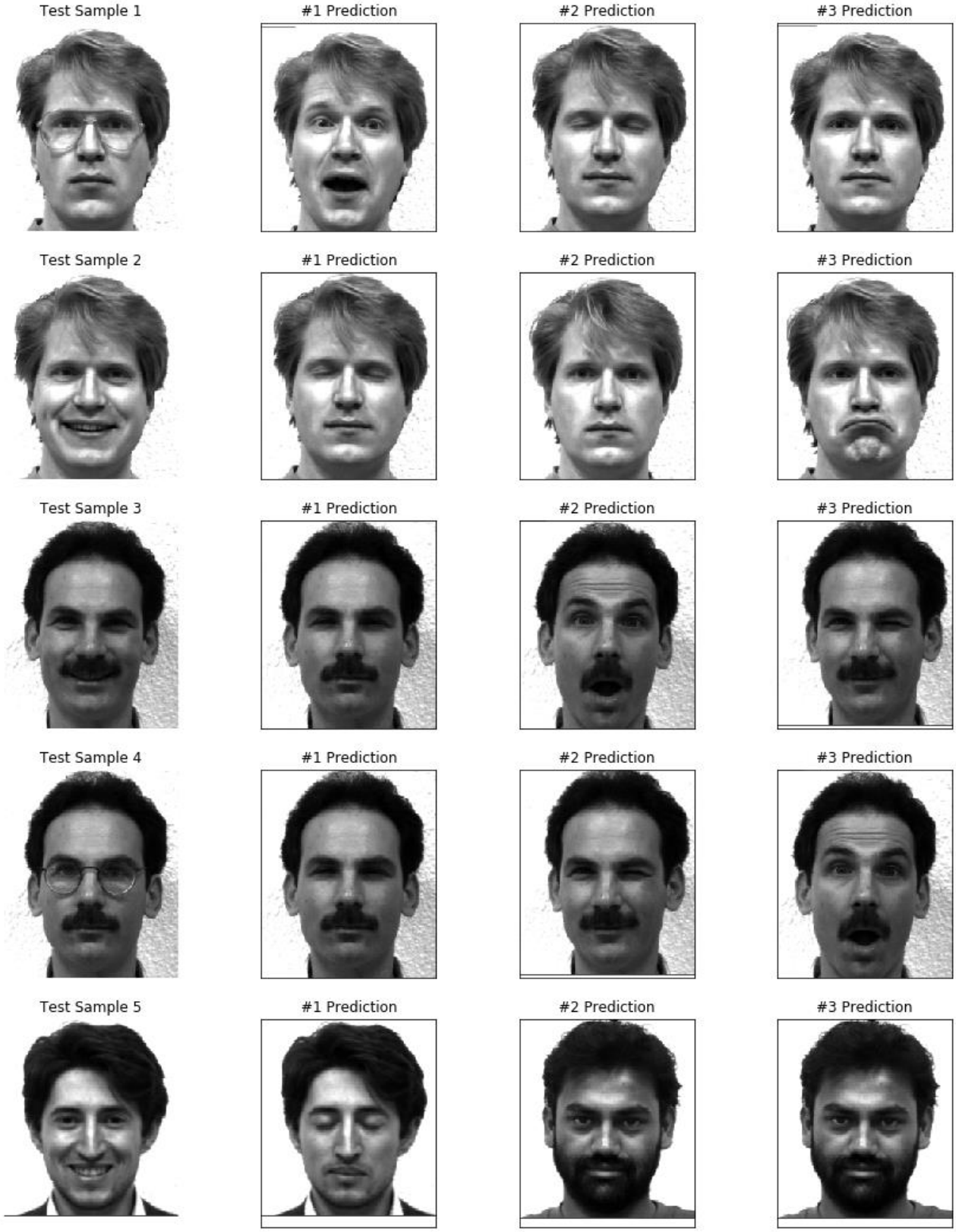


Figure 2. b) Face Recognition Results (with updated Training data)