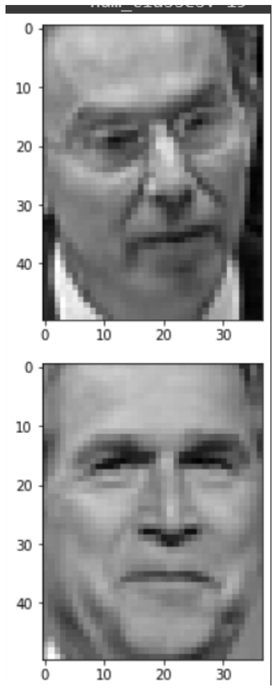
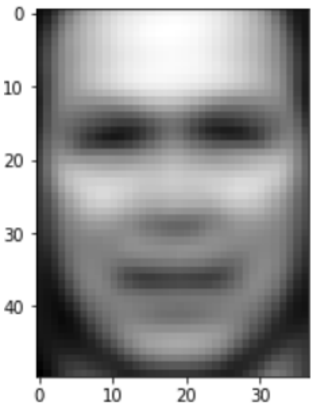
4.1

a.

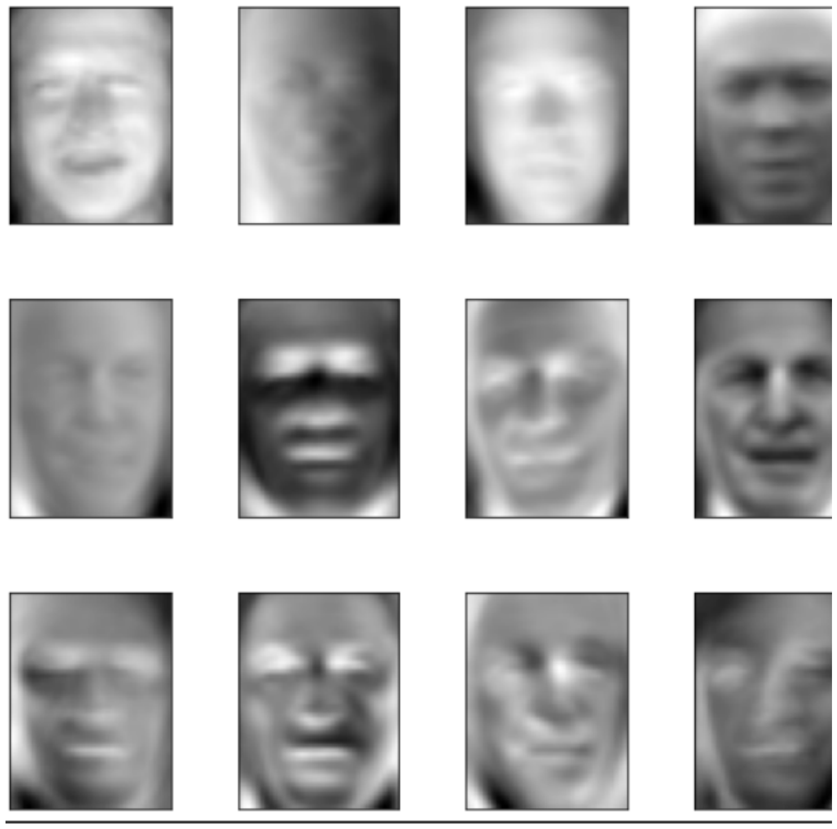




<=mean

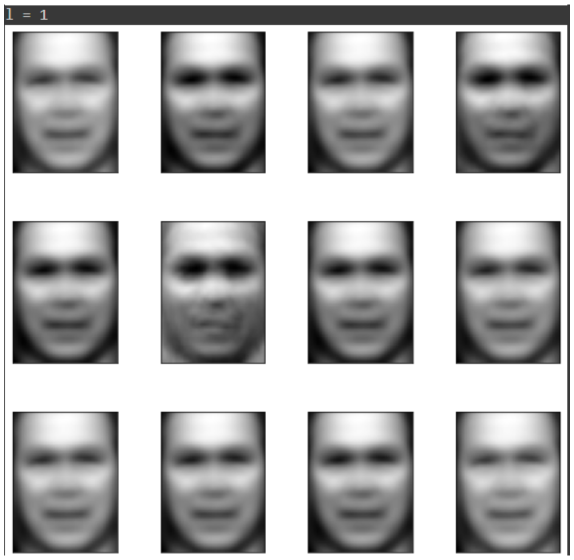
The average face has notable features, such as the nose, mouth, and eyes. However, it lacks distinct features, making it almost look like a mask

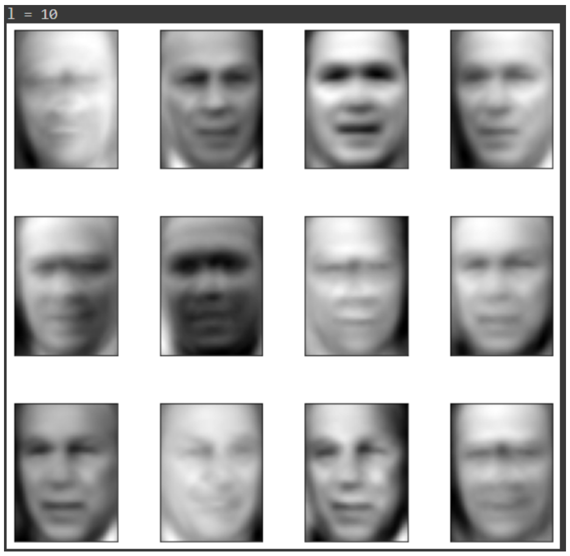
b.

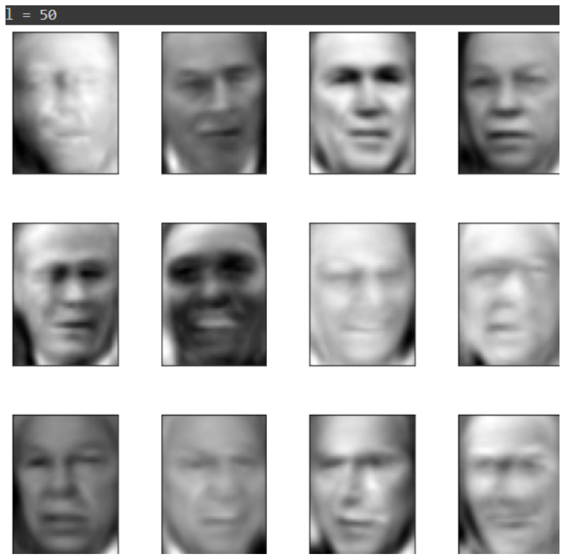


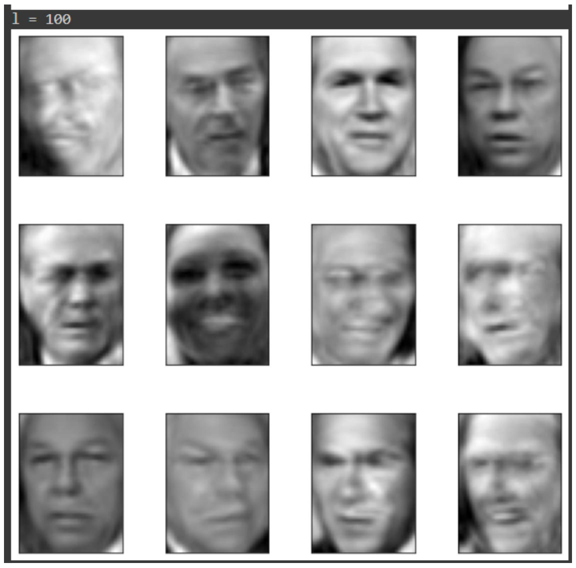
These are the top eigenfaces because they contain a lot of data for face recognition and have the notable features mentioned in a.

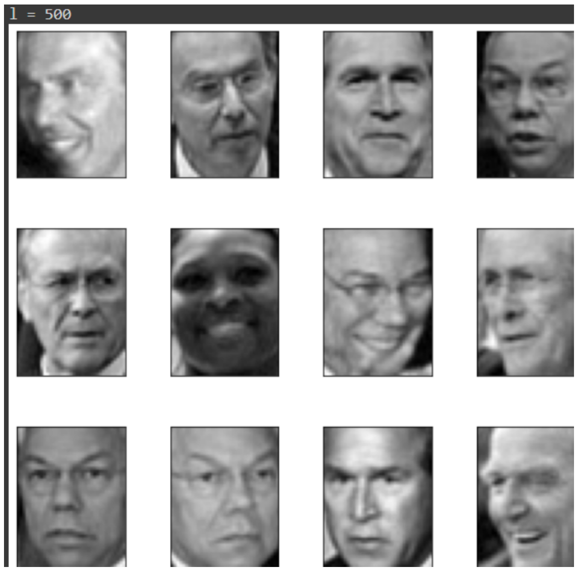
c.

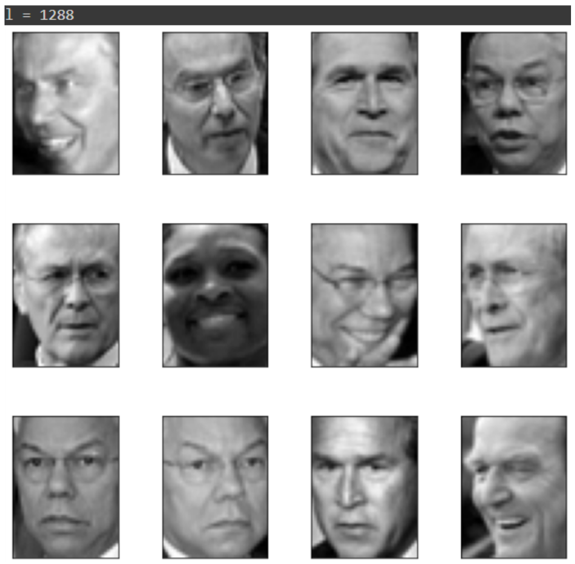








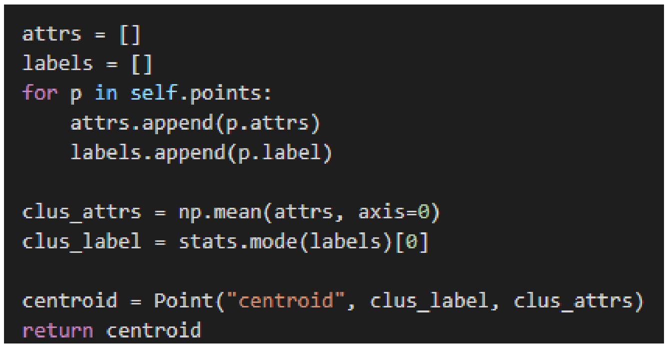




As the number of components increases, the resulting image becomes clearer and the face is identifiable. With the lower l_values, the images are identical to the average face and indistinguishable.

4.2

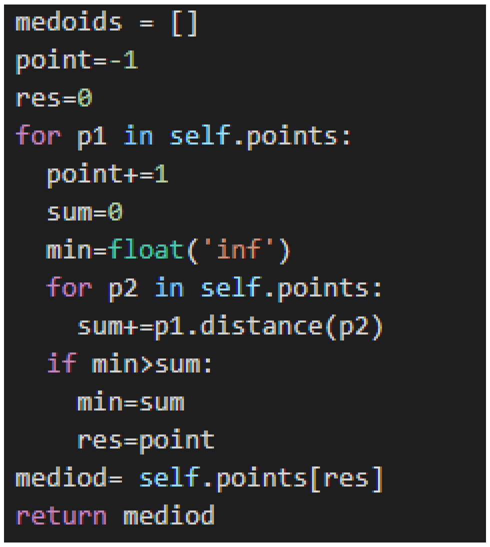
a. This is a bad idea because the minimum possible value of J is min J(c,μ,k)=0, which is when $\mu_c(i)=x^{(i)}$ and $c_i=i$. This makes it so that no information is gained when including a variable k.
b. centroid



```
attrs = []
labels = []
for p in self.points:
    attrs.append(p.attrs)
    labels.append(p.label)

clus_attrs = np.mean(attrs, axis=0)
clus_label = stats.mode(labels)[0]

centroid = Point("centroid", clus_label, clus_attrs)
return centroid
```

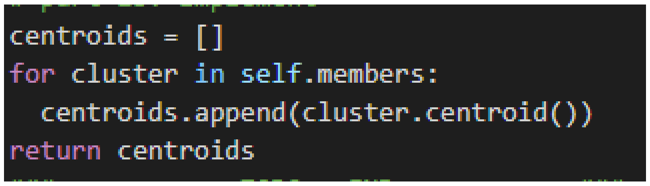
medoid

d.



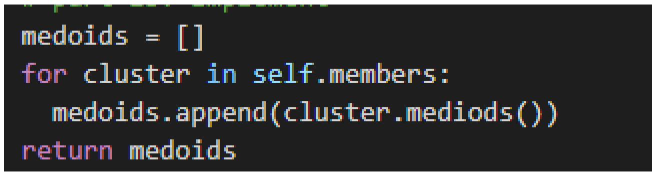
```
medoids = []
point=-1
res=0
for p1 in self.points:
  point+=1
  sum=0
  min=float('inf')
  for p2 in self.points:
    sum+=p1.distance(p2)
  if min>sum:
    min=sum
    res=point
mediod= self.points[res]
return mediod
```

centroids



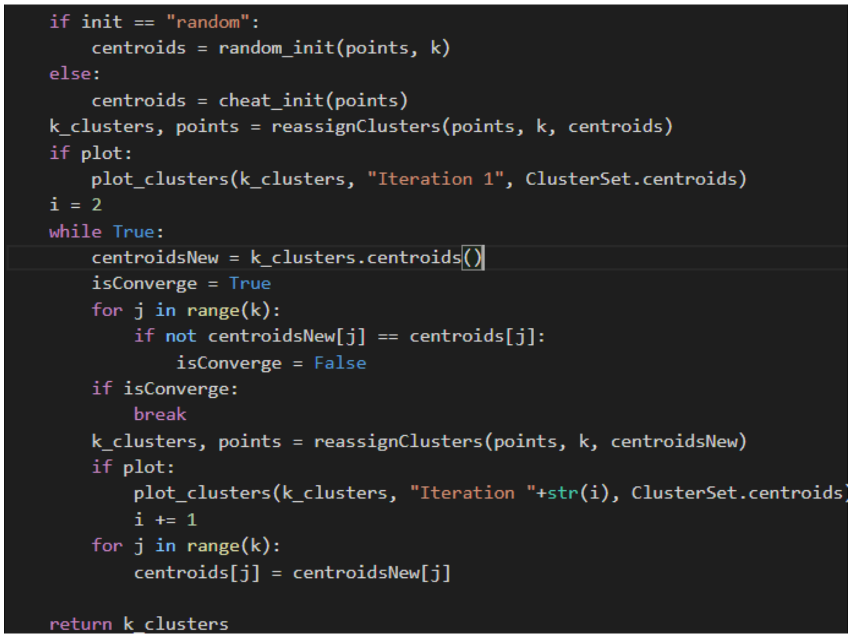
```
centroids = []
for cluster in self.members:
  centroids.append(cluster.centroid())
return centroids
```

medoids



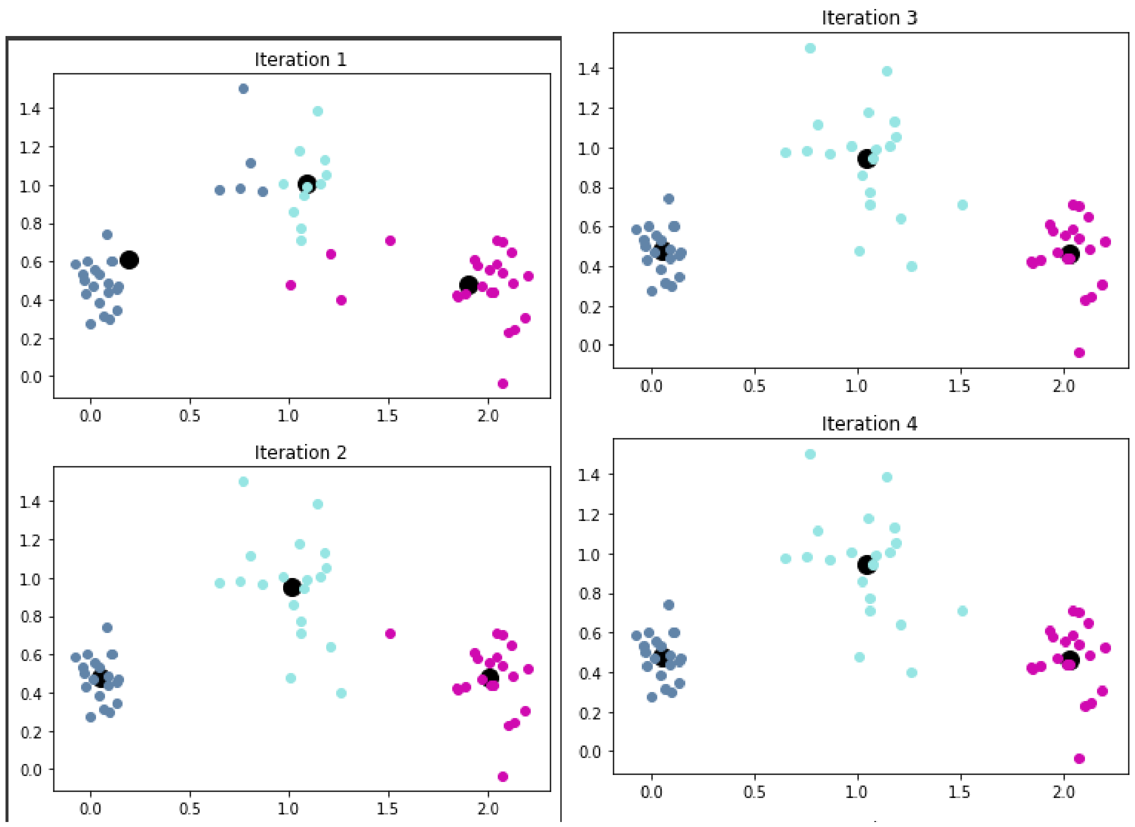
```
medoids = []
for cluster in self.members:
  medoids.append(cluster.mediods())
return medoids
```

c.



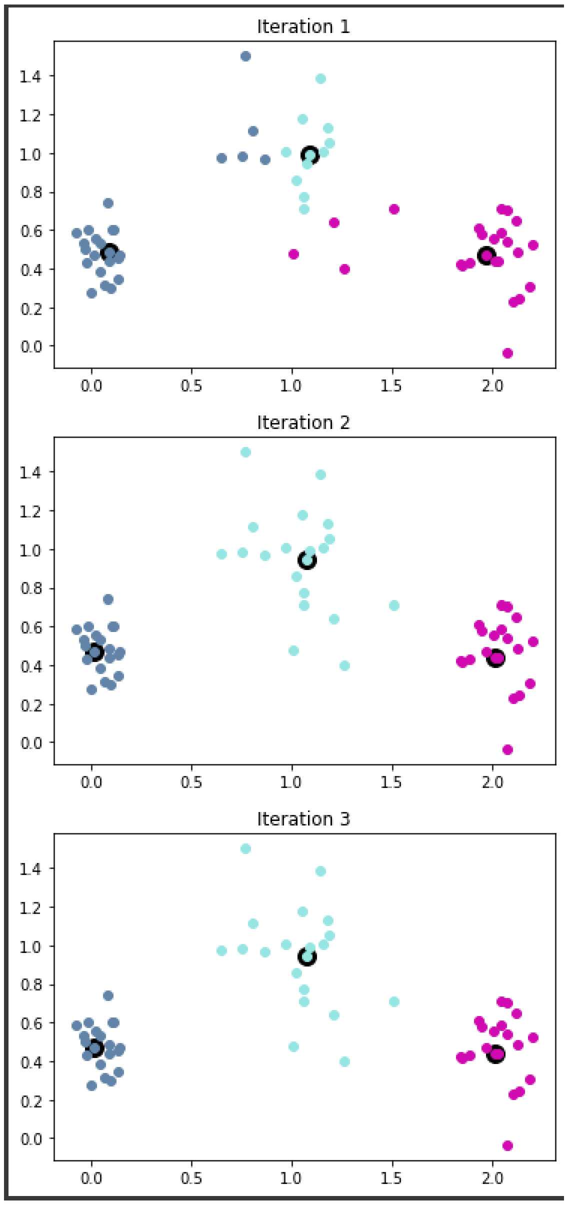
```
    if init == "random":
        centroids = random_init(points, k)
    else:
        centroids = cheat_init(points)
    k_clusters, points = reassignClusters(points, k, centroids)
    if plot:
        plot_clusters(k_clusters, "Iteration 1", ClusterSet.centroids)
    i = 2
    while True:
        centroidsNew = k_clusters.centroids()
        isConverge = True
        for j in range(k):
            if not centroidsNew[j] == centroids[j]:
                isConverge = False
        if isConverge:
            break
        k_clusters, points = reassignClusters(points, k, centroidsNew)
        if plot:
            plot_clusters(k_clusters, "Iteration "+str(i), ClusterSet.centroids
            i += 1
        for j in range(k):
            centroids[j] = centroidsNew[j]

    return k_clusters
```

d.



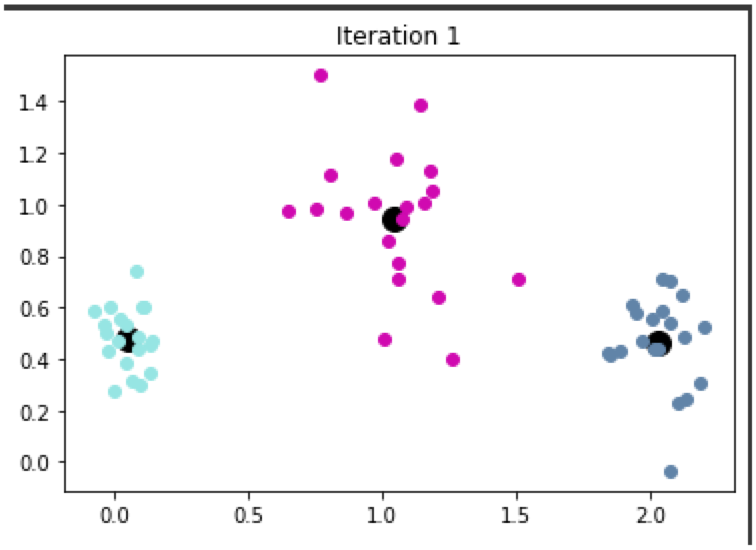
We see that as the iterations increase we are able to find a better center and properly label the data.

e.

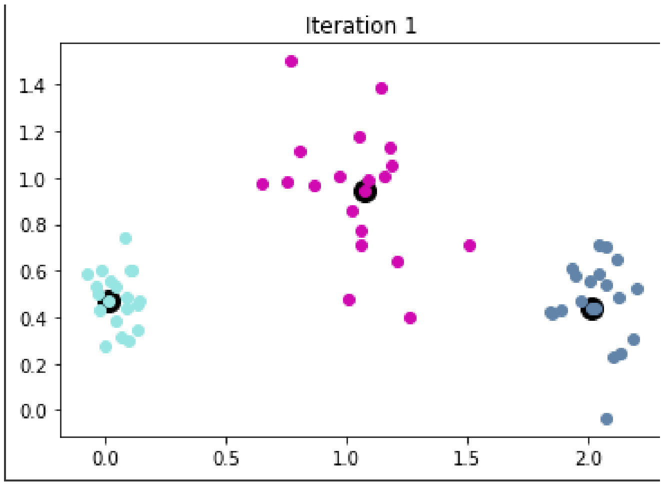


The kmedoid code is exactly the same as the kmean one, but with medoids instead of centroid. We see that the Kmedoid looks similar to the kmeans graph, but the medoid instead of a centroid.

f. kmeans:



Kmedoid:



It appears that both algorithms converge in 1 iteration, which make them much faster than just randomizing.

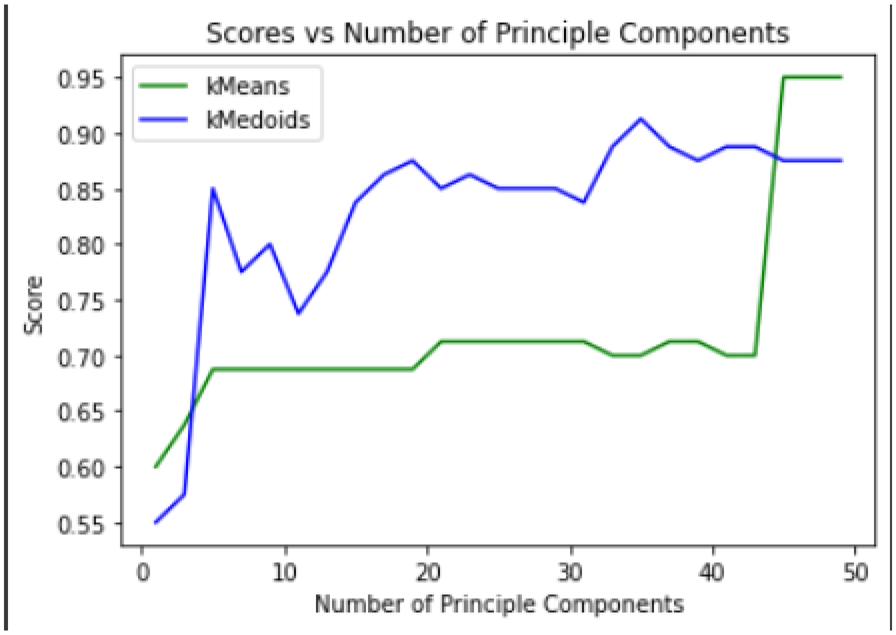
4.3)

a.

	Avg purity	Min purity	Max purity	Avg time	Min time	Max time
k-means	0.61375	0.525	0.725	0.444	0.255	0.694
k-med	0.602	0.5125	0.65625	0.419	0.2065	0.686

It appears that they perform fairly similarly, with k-medoid slightly being faster.

b.



We see that the score and the number of principle components generally increase together, with the kMeans alg being better in the long run. This makes sense since more components means more data, allowing the algorithms to perform better.

c. I used a loop to go over all the faces, then looked at faces that had scores that were high and scores that were low. This created the following result: