02807: Challenge 3 - LSH of videos

Code with output (use while reading report):
https://nbviewer.jupyter.org/gist/anonymous/feb0416c61d9affd0ff0d96c8a18cbd9
Script to verify:
https://gist.github.com/anonymous/79555bcb0b3de140a22a2082d920b647

Analysis of problem:

There are essentially 3 problems that has to be addressed in this challenge:

1. Read the frames of the videos in an efficient manner
2. Hash the resulting frame / frames with a good hash function
3. Cluster these hashes to reconstruct the clusters represented in the videos.

From David's comments, the highest priority is to get a good rand-index, thus for this challenge i optimize the speed but never at the sacrifice of accuracy.

All experiments were run on a laptop with 16gb RAM, SSD and i7-7700HQ 3.8 Ghz. Do note that the 3.8 Ghz is using turbo boost. The duration of the script is too long to sustain turbo boost through the whole process, so the real clock speed is lower.

Structuring the problem:

All videos are extracted to a folder **videos/**

For testing and developing a solution, I took the 100 first true clusters and placed them in a folder **easy/** which made it much faster to test different ideas and solutions.

Final solution is obviously run on all videos.

Reading the frames:

I've done quite some experiments with different packages and ended up with **openCV**, using a combination of threading for fast I/O and multiprocessing for parallelization.

The overall idea is to use the class FileVideoStream, which is a modified version of the one found here:
https://www.pyimagesearch.com/2017/02/06/faster-video-file-fps-with-cv2-videocapture-and-opencv/
The class creates a new thread that reads in the frames from a videofile and puts these into a queue of size 128. The main thread will continuously grap frames from the buffer and process them. The advantage of this approach is, that instead of taking a frame, processing it, taking a frame, processing etc, we will constantly be putting new frames into the buffer and analyzing them. So whenever the thread is waiting to receive the data from the SSD, the other thread can analyse the images in the buffer.

This reduces the latency which comes from I/O, and using multiprocessing I can do 4 of these processes simultaneously, one on each core of my laptop.

The class needed modification as it would stop if a process were reading the frames faster than the que could fill. This is done through some simple logic; if there is no elements in the que but the class is not stopped, sleep the main thread for a bit of time and try to fetch a new frame after it wakes up. The reason for sleeping instead of passing is, that passing just runs full speed and wastes a lot of cycles.

My final solution combines all the frames into a single average frame for the video. First implementation showed severe memory consumption because I stored all the frames of the video before combining them at the end. To combat this, I just add the current frame to a numpy array, **tot_frames** holding the sum of all the pixels values. When I've read all the frames, i divide by the total number of frames to get the mean image for a video.

To further reduce memory impact I decided to return the hash instead of the the average frame, thus I end up with 9700 hashes instead of 9700 images in memory.

Hashing:
The process of hashing is essentially a process of finding proper image descriptors, for many computer-vision tasks, a very solid image descriptor is the SIFT-features. I implemented these, but they were not the best solution and also rather computational intensive.

In the end I used the **perception-hash** from the *imagehash* package. Description is found here: http://www.hackerfactor.com/blog/index.php?/archives/432-Looks-Like-It.html
It resizes the image, reduces color, and uses discrete cosine transform to construct a hash.

The library can both give a hash string and also a boolean array that the hash string is computed from. One can easily find the distance between two hashes by subtraction or perhaps the hamming distance, however for clustering this many samples I prefered to have higher dimensionality so I flatten the boolean array and used this as input to the clustering algorithm. Thus every video ends up with a boolean array of length 32 (8x8 hash):

```
In [11]: hashes[0]

Out[11]: array([ True,  True, False,  True, False, False, False, False,  True,
                 True,  True, False, False, False,  True, False, False,  True,
                 True, False,  True,  True,  True,  True, False,  True,  True,
                False,  True, False,  True,  True, False, False, False,  True,
                 True,  True, False,  True, False,  True,  True, False, False,
                 True, False,  True, False, False, False, False,  True, False,
                False,  True, False,  True,  True,  True, False, False, False,  True], dtype=bool)
```

After completion, we will have a list **res_cv** containing tuples (video_path, multiprocess_result). I extract the hashes and the proper names of the video (removing the path and file-type), using the following code:

```
hashes = []
names = []
for result in res_cv: #Loop over results from process
    hashes.append(result[1].get())
    name = result[0]
    name = name.replace(path, '')
    name = name.replace('.mp4', '')
    names.append(name)

names = np.array(names)
```

Clustering the hashes

The clustering step is quite difficult, if done wrong one can easily think that the hashes are the problem. Indeed it is hard to find algorithms that keep the same cluster sizes - that is not normally how you use clustering as it is more used to capture the structure of the data. Thus using standard clustering techniques, the separation needs to be very perfect to get decent results.

I considered rolling my own clustering algorithm, combining the pairs with smallest distance if they did not belong to a cluster of size larger than 10. This is quite like hierarchical clustering and fortunately enough my hashes had good enough separation to use **Agglomerative Clustering** from sklearn. I use the *ward*-linkage which is the best choice for the majority of problems in my experience.

The clusters are then created as sets, by finding the labels and the corresponding video names for the data.

I also try to use Kmeans. The main computation time is in the reading of the videos, thus the loss of doing both is negligible.

Results:

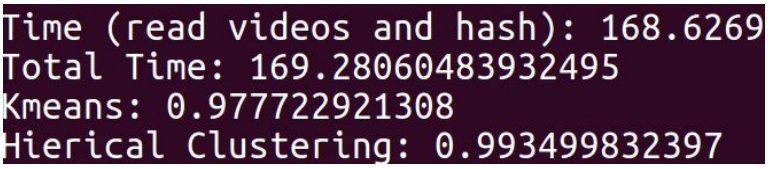
The total computation time was **1398** seconds or 23.3 minutes.
1366 seconds were used on reading frames and computing hashes (the computation of hashes were instant compared to the time for reading the frames).
32 seconds were used to construct both Kmeans and Agglomerative Clustering with 970 clusters including testing.

Rand_index: 0.97

In [5]:
```
from sklearn.cluster import KMeans
from sklearn.cluster import AgglomerativeClustering

X = np.array(np.array(hashes))
if path == "easy/":
    num_clusters = 100
else:
    num_clusters = 970
kmeans = KMeans(n_clusters=num_clusters).fit(X)
agg = AgglomerativeClustering(n_clusters=num_clusters
                              , linkage="ward").fit(X)

clusters_km = []
clusters_ag = []
for cluster in range(num_clusters):
    clusters_km.append(set(names[kmeans.labels_ == cluster]))
    clusters_ag.append(set(names[agg.labels_ == cluster]))

print("Total Time:", time.time()-start)

if path == "easy/":
    print("Kmeans:", rand_index_small(clusters_km))
    print("Agglomerative Clustering:", rand_index_small(clusters_ag))
else:
    print("Kmeans:", rand_index(clusters_km))
    print("Agglomerative Clustering:", rand_index(clusters_ag))
```

```
Total Time: 1398.166345834732
Kmeans: 0.906092856067
Agglomerative Clustering: 0.970307574513
```

For the easy data-set with only 1000 videos (total time is after clustering and calculating rand-index):



Discussion:
I have tried quite a few tricks to speed up the script, only taking the first 30 frames and sampling the frames at a smaller resolution. However each time we throw some accuracy away and since that is the main judging criteria I didn't want to sacrifice any.

Using a few tricks I could get a rand-index of 0.96 for about 700 seconds computation time and 0.83 with 300 seconds.

I was quite surprised that the SIFT-features didn't perform better (around 0.5 rand_index), but on the other hand I could not do a full bag of SIFT-features because there we simply too many for all the videos.

I could not find any efficient way to skip frames, if one could take only every 5. Frame, the result would probably not change much but time could perhaps be much faster. The bottleneck is however reading in the frames, and placing the reader in another position of the video takes longer than reading in 5 frames sequentially.