02807 - Computational Tools for Big Data
Challenge 3

1 Introduction

In this project a set of 9700 small videos were processed and clustered according to similarities based on Locality Sensitive Hashing of the video frames. The video-set contains 970 'true' groups of 10 videos produced by altering brightness,frame rate, adding a black border or deleting parts of the beginning or end of a video. The video-set can be downloaded here.

In the project we use the following Python libraries:

- **imageio** - To read the video sequences
- **imagehash** - To hash the frames from the video sequences
- **OpenCV** - To perform various forms of image processing
- **multiprocessing** - To handle multiple processes when loading the data and extracting features
- **sklearn.cluster** - For clustering the videos.
- **os** - For file handling
- **time** - For timing the running time

The code was tested on a Macbook Pro Retina 15" Early-2013, with an Intel Core i7 2.4 GHz CPU (4 physical cores, 4 virtual cores), 256 GB SSD, 8 GB 1600 MHz DDR3 RAM and a Nvidia Geforce GT 650M GPU with 1GB GDDR5 V-RAM.

The full Python-script of the solution can be found here.

2 Approach

The main idea behind our approach to solve the clustering problem, is to compute a 'perceptual' hash of each brightness-normalized gray-scale frame in a video, convert the hash-string to a number and use this number/feature to update a hashed feature-vector. The hashed feature-vector should then ideally describe the frames of the video independently of the brightness, order of the frames and overall length of the video. The feature-vectors of all the videos are then used to perform the clustering.

In order to compute the perceptual-hash/feature of all (or a limited set of) the frames in a video, we created the helper function ***compute_features(video_name)***, which can be seen in the listing below. The function uses the library ***imageio*** to obtain an iterable reader-object and the resolution for a video with the name *video_name*. The function iterates over all frames in the video, converts them to gray-scale, crops to a predefined size, and normalizes the brightness by using the ***Opencv*** functions *cvtColor()* and *equalizeHist()*. A feature representation of each frame is then computed using the 'perceptual' hash function *average_hash()* from the ***ImageHash*** library. The hash is 'perceptual' in the sense that the general structure or low frequencies of the image are retained when reducing the size of the image and computing the hash. We also experimented with the *phash()* function, that uses more advanced DCT-components to retain the image structure in the hash, but decided to go with *average_hash()* because it was faster and in general performed well enough.The *hash_size* parameter defines the number of characters in the resulting string.

```
def compute_features(video_name):
    video = imageio.get_reader(video_name) # get reader-object for video
    w,h = video.get_meta_data()['size']#get original video-resolution in order to perform crop
    frame_features = [] # list for frame-features
    compute = 4 #only compute feature every 4th frame
    count = 0 # frame count

    for frame in video:
        count +=1
        if count%compute==0:#only compute features for limited amount of frames
            count = 0
            gray = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY) #convert to gray-scale
            crop = crop_image(gray,w,h) #crop image
            eq = cv2.equalizeHist(crop) # normalize brightness by equalization of histogram

            # get feature by computing perceptual hash of frame
            feat = str(ih.average_hash(Image.fromarray(eq),hash_size=4))
            frame_features.append(feat)
    return frame_features
```

In order to obtain the same resolution for all videos and avoid the influence of the black border added to some of the videos, we created the following helper function ***crop_image(im,w,h)***, where *im* is the frame that should be cropped and w and h are the original image-dimensions. The reasoning behind our chosen dimensions can be found in Section 3.

```
def crop_image(im,w,h):
    if h > w:#check for portrait aspect ratio, and define crop image dimensions
        x = int(250);y = int(450)
    else:
        x = int(450);y = int(250)

    x_crop = int(w/2 - x/2);y_crop = int(h/2 - y/2)#define start x,y-coordinate for crop
    crop = im[y_crop:y_crop+y, x_crop:x_crop+x] #extract cropped image
    return crop
```

The helper function ***create_vector(features)*** is given a list of frame-features *features* and returns the hashed feature-vector *vec*. The variable *N_buck* describes the number of buckets used for the hashing i.e. the size of the feature-vector.

```
def create_vector(features):
    # Compute hashed feature-vector
    N_buck = 700 # number of feature-hash buckets
    vec = np.zeros(N_buck)#initialize feature vector

    for feat in features: # for each feature in feature-list
        vec[hash(feat)%N_buck]+=1 #increase feature-count at hashed index
    return vec
```

The helper function ***create_feature_list(file_chunk)*** computes and returns the list-pair *(names,features)* of video names and corresponding hashed feature-vectors for all file-names in the list *file_chunk*.

```
def compute_feature_list(file_chunk):
    names = []
    features = []

```

```
    for f in file_chunk: # iterate over all files 'f' in chunk
        if f.endswith('.mp4'): #avoid .DS_store files on macOS
            names.append(os.path.splitext(f)[0])#add video-name to list
            feats = compute_features('videos/'+f)#compute features
            v = create_vector(feats)# compute hashed feature vector
            features.append(v) # create list of feature for all frame in video
    return (names, features)
```

In order to utilize parallel processing of the videos, we created the function ***get_file_chunks(files,n_cores)*** to compute equally sized lists of file-names for each available cpu-core to process. The function takes a list of file-names *files* and the number of available cpu-cores *ncores* as input and returns a list of file-name-lists. All lists will contain an equal amount of file-names except if the total number of files is not divisible by the number of cpu-cores, then the last list will contain an amount equal to the remainder.

```
def get_file_chunks(files,n_cores):
    chunk_size = int(len(files)/cores) #number of files in each chunk
    #list comprehension iterating over all file-indices with chunk-size step
    chunks = [files[i:i+chunk_size] for i in range(0,len(files),chunk_size)]
    return chunks
```

In the main function of our script we use the *Pool()* function from the ***multiprocessing*** library to manage the parallel processing and the Spectral Clustering method from ***sklearn.cluster*** to cluster the videos. The most important parts are shown with explaining comments in the listing below.

```
if __name__ == '__main__':
    files = os.listdir('/Path/to/videos') # get list of all file-names in directory
    # split files into chunks/lists of equal size for each core to process
    cores = mp.cpu_count()# available number of cpu cores
    file_chunks = get_file_chunks(files,cores) # list of lists with filenames

    # Deploy processing pool, computing feature vector for each video
    # results is a list of feature-vectors, one for each video in each file-chunk
    pool = mp.Pool(cores) #initialize multiprocessing pool
    results = pool.map(compute_feature_list,file_chunks)
    pool.close();pool.join() # wait for completion of all processes

    #concatenate results from different processes
    features = [];vid_names = []
    for res in results:
        for i in range(len(res[0])):
            vid_names.append(res[0][i])
            features.append(res[1][i])
    X = np.array(features) # array of feature vectors

    # compute clustering of videos
    Nclusters = 970
    cls = SpectralClustering(n_clusters = Nclusters,eigen_solver='arpack',
        affinity="nearest_neighbors", n_init=20,
        assign_labels = 'discretize',n_jobs=-1).fit_predict(X) # n_jobs =-1 uses all available cores
    clusters = [set() for i in range(Nclusters)] #initialize empty cluster-structure
    # create cluster structure needed for computing rand-index
    for i in range(cls.size):
        clusters[cls[i]].add(vid_names[i])
    r_i = rand_index(clusters)
    print('Adjusted Rand-index: '+str(r_i))
```

3 Testing

A subset of 300 videos from 30 true video-groups was created for testing, to avoid testing on the entire data set every time the code was updated. The test results from the small set, can be seen in Table 1. Various methods were implemented with different levels of success. Initially the videos were clustered based on hash of a mean image from a video sequence. However this proved to be a naive method with low precision. It only yielded an adjusted rand index of 0.601, for only 300 videos.

The method was slightly improved using a standard crop for every video, to avoid the black border added to certain videos. Looking through 10 videos from a category, the one with no border, and the one with the biggest black bars were found. 2 matching frames from the video, can be seen in Figure 1. It can be seen that the left screenshot, has no border. It is the only one in its category without any border, and it seems to be the original. The one on the right, has the largest black border of the 10 videos, and it has lower brightness. To improve the feature extraction, all videos were cropped equally to a size of (450,250) eliminating the black border from interfering with the hashing. The crop on top of the mean image method,

Figure 1: Left: No crop video Right: Full crop

yielded an adjusted rand index of 0.780. However this was only on the small set of 300 videos, which meant that for the full set of 9700 videos, the adjusted rand index would be significantly lower.

The final method was using cropped and brightness normalized frames from the video sequence as features as described in the previous section. This yielded an adjusted rand index of 1.0 from the small set, and thus it was chosen to be the method to try on the full set. When testing the method of using frames as features

Method	Adjusted Rand Index	Running Time	Settings
Cropped frames as features	1.0	39 s	Hash_size of 4 for each frame, Every 12th frame read
Mean Image and crop	0.780	62 s	Every 4th frame read
Mean Image	0.601	59 s	Every 4th frame read

Table 1: Test of video hashing methods on a small subset of 300 videos

on the full set of 9700 test videos, various hash sizes for each frame, and frame skips were tested. The test results can be seen in Table 2.

A frame skip efficiency test was done using every frame and every 4th frame, using the same hash size of 4 for each frame. As expected, using every frame yielded a better result, with an adjusted rand index of 0.95422. Using every 4th frame resulted in an adjusted rand index of 0.94371. So a relative difference in adjusted rand index of 1.1%, but for a significant running time trade-off, with a relative difference of 36.5%.

Due to limited testing time, and running the tests on an old consumer laptop, we chose to test on every 4th frame, as the difference in adjusted rand index was minor, compared to the time trade-off.

Then testing of using different hash sizes for each frames was performed. The results for this can also be seen in Table 2 sorted for descending adjusted rand index. It can be seen that the optimal hash size for our implementation, is a hash size of 5 for each frame, using every frame.

Settings	Adjusted Rand Index	Running Time
Hash_size of 5, Every frame	0.98131	2157 s
Hash_size of 5, Every 4th frame	0.97944	1565 s
Hash_size of 6, Every 4th frame	0.97940	1558 s
Hash_size of 6, Every frame	0.97734	2208 s
Hash_size of 7, Every 4th frame	0.95618	1607 s
Hash_size of 4, Every frame	0.94522	2127 s
Hash_size of 4, Every 4th frame	0.94371	1558 s
Hash_size of 8, Every 4th frame	0.78665	1674 s

Table 2: Test of video hashing on 9700 test videos

4 Results

Our **optimal implementation got an adjusted rand index of 0.98131** with a **total running time of 2157 seconds, or 35 minutes and 57 seconds**, running on a consumer laptop as specified in the introduction. The Terminal output from the solution can be seen in this gist.

Our optimal method uses feature-hashing with a bucket-number of 700, to create a feature-vector for each video, where the features are the 'perceptual' hashes of cropped and brightness-normalized gray-scale video-frames. It uses a hash size of 5, and every frame in each video are used. It is not the optimal solution in terms of running time, but it has the highest adjusted rand index. Taking the time trade-off into consideration, skipping a few frames between each feature can be done without much loss as seen in Section 3.

5 Discussion

As testing on our laptops was slow and inefficient, we used DTU HPC for some of the testing. However we were not successful in finding a suitable setup for the HPC, as the running time was as slow as on our laptops, despite using 40 cores. This gave us time constraints in terms of testing additional settings, and additional features hashed for each video.

Given more time, we would have tested different amounts of frames skipped between each frame used as feature, and we would have added additional features to hash for each video. As explained in the former sections, initially a method of hashing a mean image of each video was implemented as well, however with little success.

This could have been added as an additional features for each video, and other methods such as finding the dominant colour, etc. could have been added as well, for increased accuracy, but with a significant increase in complexity.

6 Conclusion

The results of making our own locality sensitive hash function for hashing small videos looks promising. We focused on reaching a good balance between accuracy (adjusted rand index) and complexity (running time). Low complexity was important to us due to testing on our resource constrained hardware (laptops), while maintaining a decent accuracy.