FINAL PROJECT COMPUTER VISION

26.01.2024

Work environment

For this project I used the virtual environment “computervision” in Anaconda and jupyter lab.

Slot coordinates detection

First of all, before starting to detect whether the slots of a park in an image were busy or free, I had to face the problem of how to recognize a slot.

The first idea that I’ve come up with was to take, for example at first, a rainy image, where I was sure that there were any cars in the park (but then it could have been done with all, or part of, the dataset given) and by using a Canny, being able to find the lines delimiting all the parking slots of the park image. Then, I would use the defined slots for searching if there were cars in it or not. Unfortunately, as I tried I recognized that it was difficult to find good parameters for finding suitable delimiters for every slot, so I discarded the idea. Only after doing so, I discovered that in the dataset there were “csv”s that displayed the coordinates of every slot in that camera and their width and height. So I used this information by importing the csvs and creating a dataframe.

The first idea for detection

As far as the recognition of a busy or free slot is concerned , the first basic idea (*first_attempt.ipynb*) that I’ve implemented consisted in taking an image, because at first I tried only for an image to make sure that it’ll work, and creating a bounding box by hand, for the same reason as before, so that I would be sure if that particular slot was busy or free. After doing so, I’ve made a SIFT searching for keypoints and descriptors of that particular slot.

Thereafter, I’ve collected in two lists all the examples of busy slots (taken from the dataset *CNRPark-Patches-150x150\A\busy*) and free slots (from dataset *CNRPark-Patches-150x150\A\free*) and for every image in these lists I’ve used a SIFT to find the descriptors. This was useful for implementing the matching between the descriptors of a busy or free image and the ones of the slot. If there was a good match (using Lowe's ratio, an integral part of the SIFT algorithm used to evaluate how reliable and accurate the matches between features extracted from different images are) between the busy image and the slot a counter “busy” would increment of one and the same for the free dataset with counter “free”. After matching all the busy images with the slot and all the free ones, if the “busy” counter was a bigger number than the free one, the slot would be declared as busy otherwise as free.

```
busy counter:  11988
free counter:  6614
the parking slot is BUSY
```

Measure of quality

The main negative aspect of this approach, other than the time inefficiency and the raw method for taking the slot, is that in general free slots have less descriptors than busy ones, so there would be maybe a good detection for busy slots but not the same can be said for the free ones.

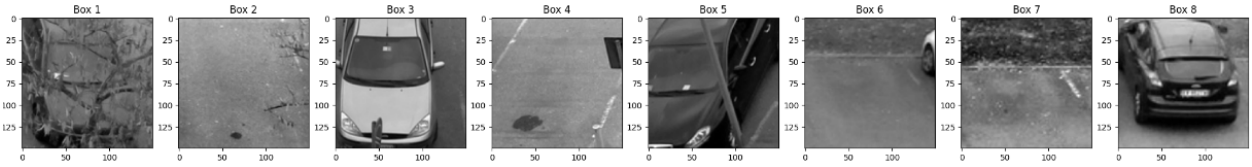
So, for this very reason I’ve decided to change the approach to the problem.

The idea for the project

The idea for the project (*FINAL_PROJECT.ipynb*) is to create a **bag of words** with the main features of an image and use a **machine learning approach** for detecting the accuracy of the correct matching between the slots of an image and its ground truth.

I used four images taken from the dataset, with different weather condition, different cameras and with different number of free and busy slots, to better test the quality of the implementation. By random choice (using *np.random.randint(0, 4)*) I pick an image taken from the list of four with a random index.

Then, I dynamically build bounding boxes for each slot, as said in the first paragraph, from the csv of the specific camera of the picked image.

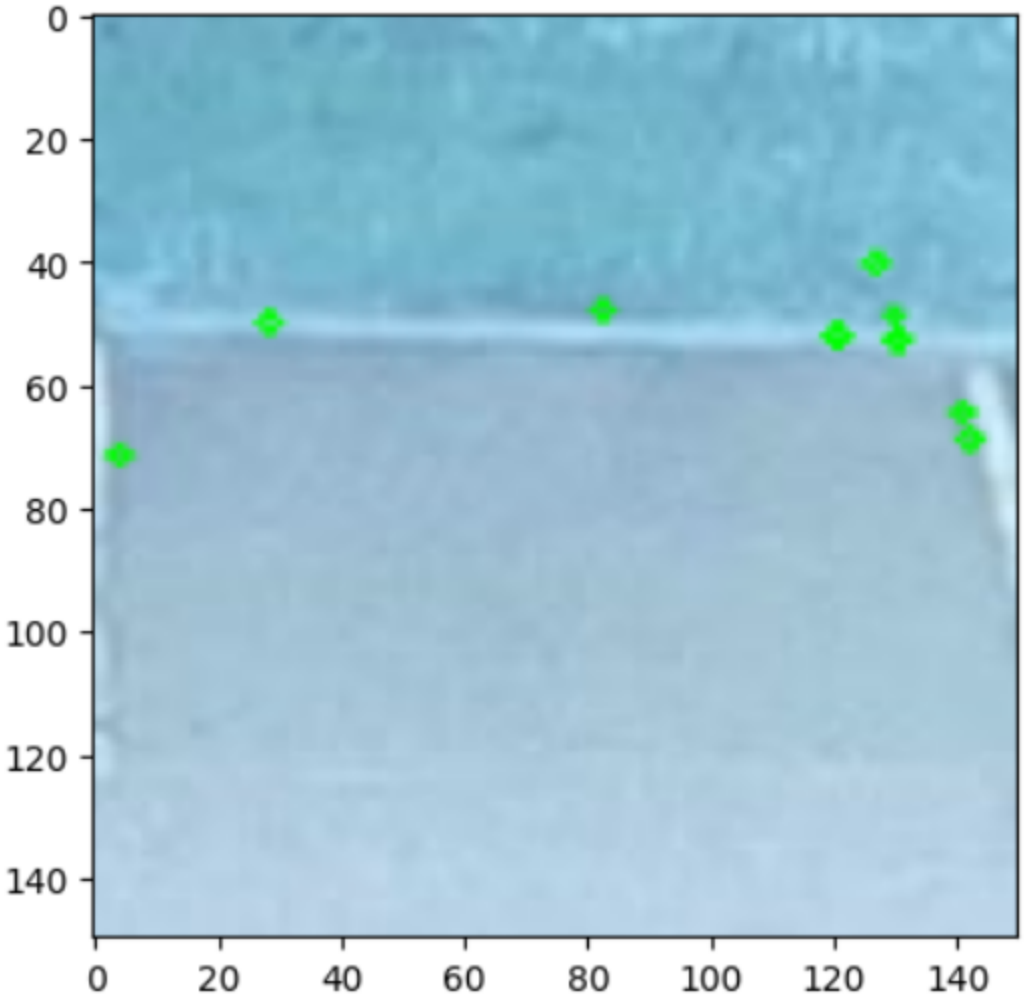


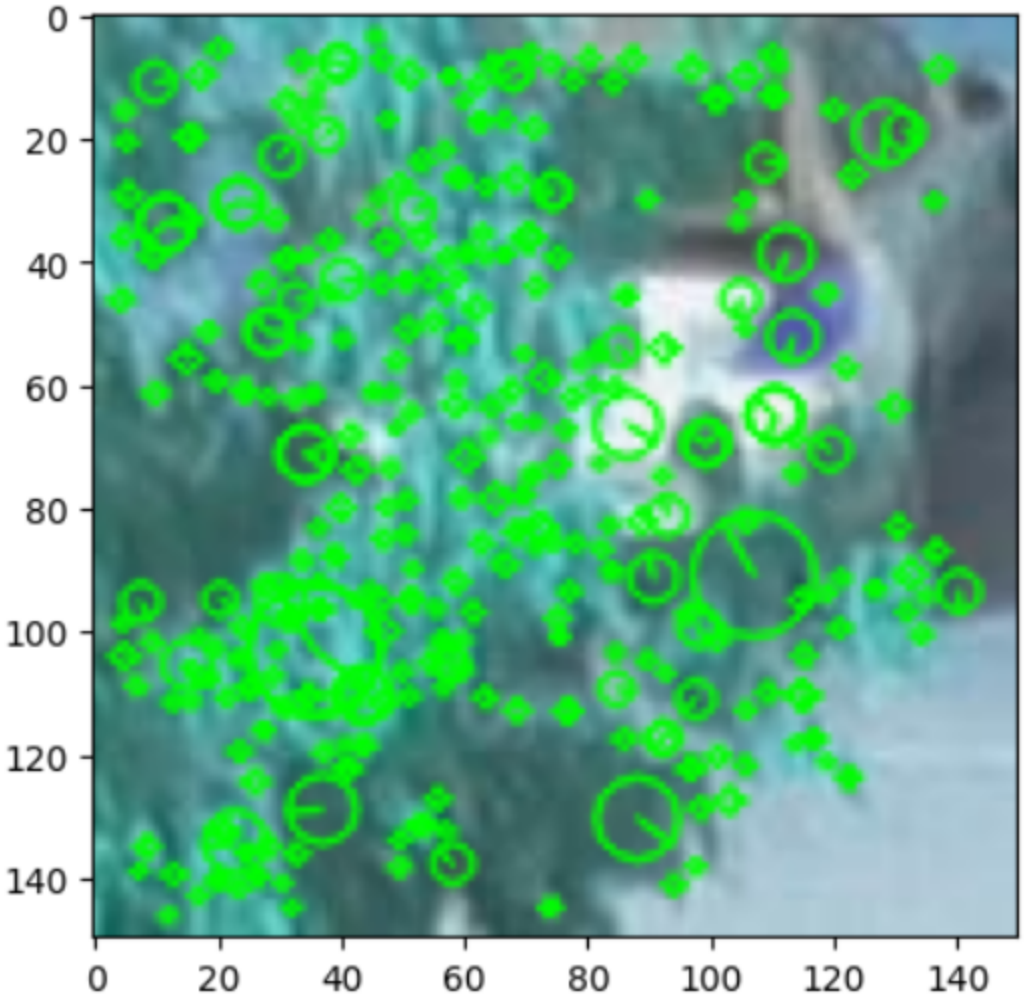
Examples of the first 8 slots of an image

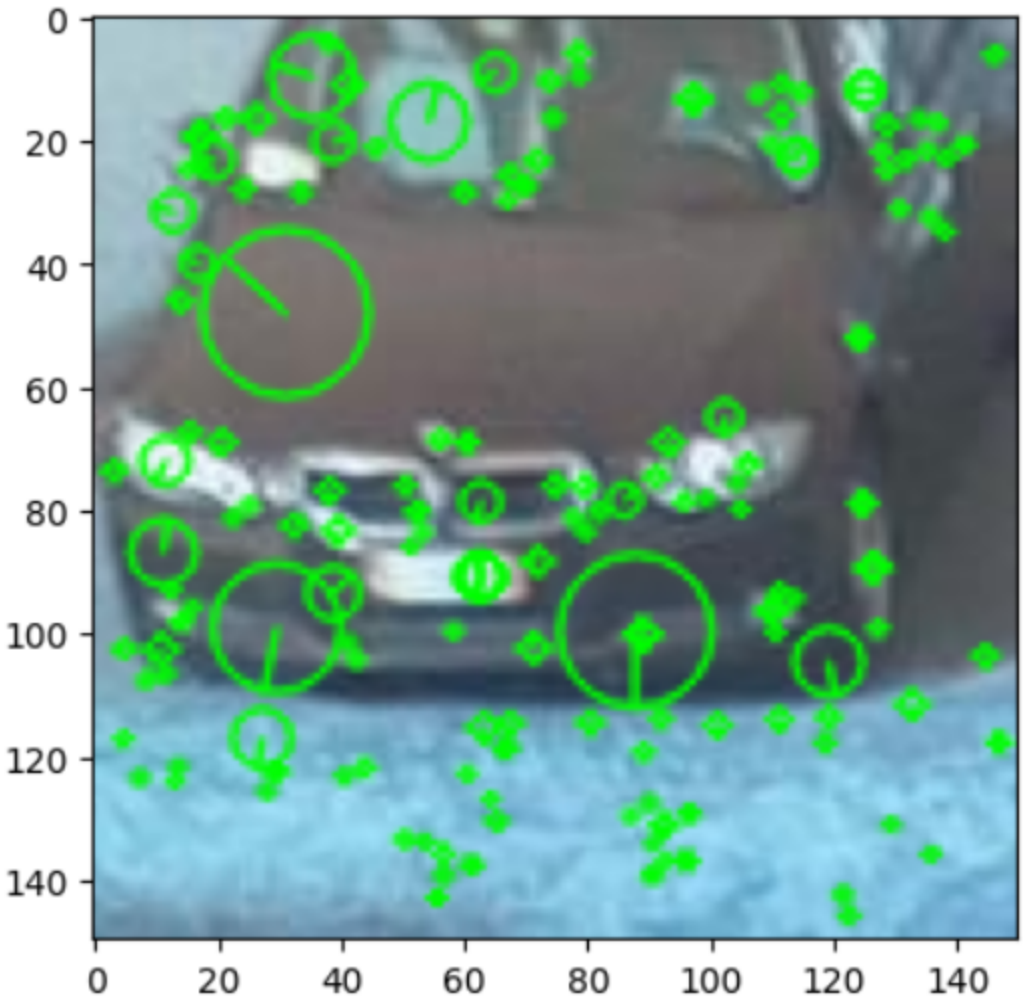
From these slots I build manually the ground truth as a vector of *1s* and *0s* indicating whether a slot is busy (1) or free (0).

Now, it begins the training part for the machine learning algorithm. I create lists for each example of busy slot (still taken from the dataset *CNRPark-Patches-150x150\A\busy*) and free slot (*CNRPark-Patches-150x150\A\free*) and another list to collect altogether the two, since a mix of busy and free slots examples will be used to create a codebook of centroids after implementing kmeans.

I convert all these images to gray and by using a *xfeatures2d.SIFT_create()* and *extractor.detectAndCompute* I find keypoints and descriptors for each image.


0
20
40
60
80
100
120
140
0
20
40
60
80
100
120
140


0
20
40
60
80
100
120
140
0
20
40
60
80
100
120
140



Furthermore by using *np.random.choice* I take only 1000 samples from each list of busy, free or mixed images and collect separately all of their descriptors.

By implementing the *kmeans* algorithm of 30 clusters using the list of mixed images, I create a codebook where are contained all the centroids of the clusters.

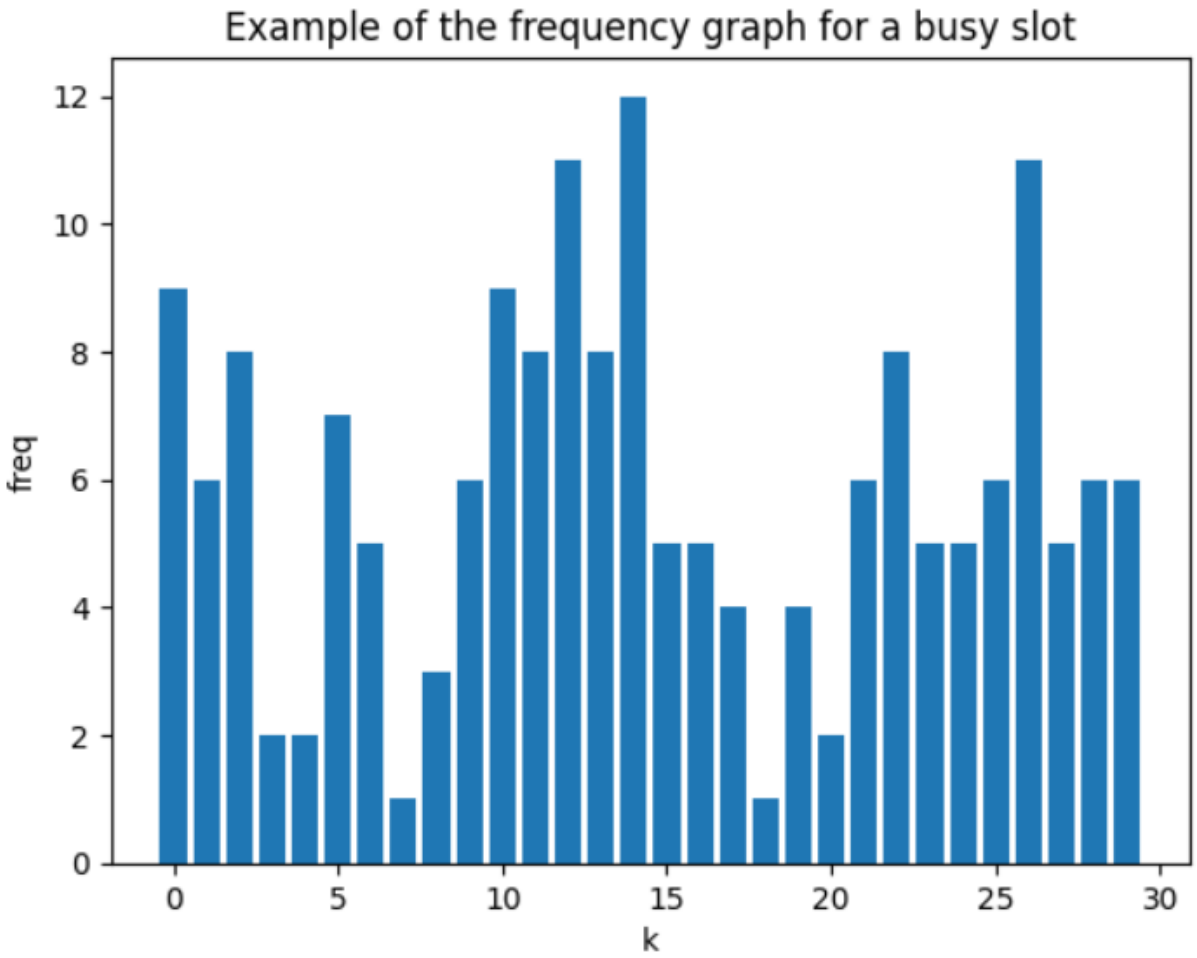
```
BF [[12.464795   7.230557   9.359871  ...  4.059825   3.83341    9.415094 ]
 [15.563461  15.067307  15.976442  ...  8.943269   7.960577   8.746154 ]
 [25.911993  13.024178  10.968085  ...  6.32205    7.695358  36.063347 ]
 ...
 [20.818092  18.983374  16.15697   ... 34.51296   26.52714   15.93643  ]
 [80.75074   18.245127   4.7897224 ...  6.6993504  4.145895   8.926757 ]
 [18.89394   20.87344   23.910427  ... 15.547683  15.902852  19.319073 ]]
```

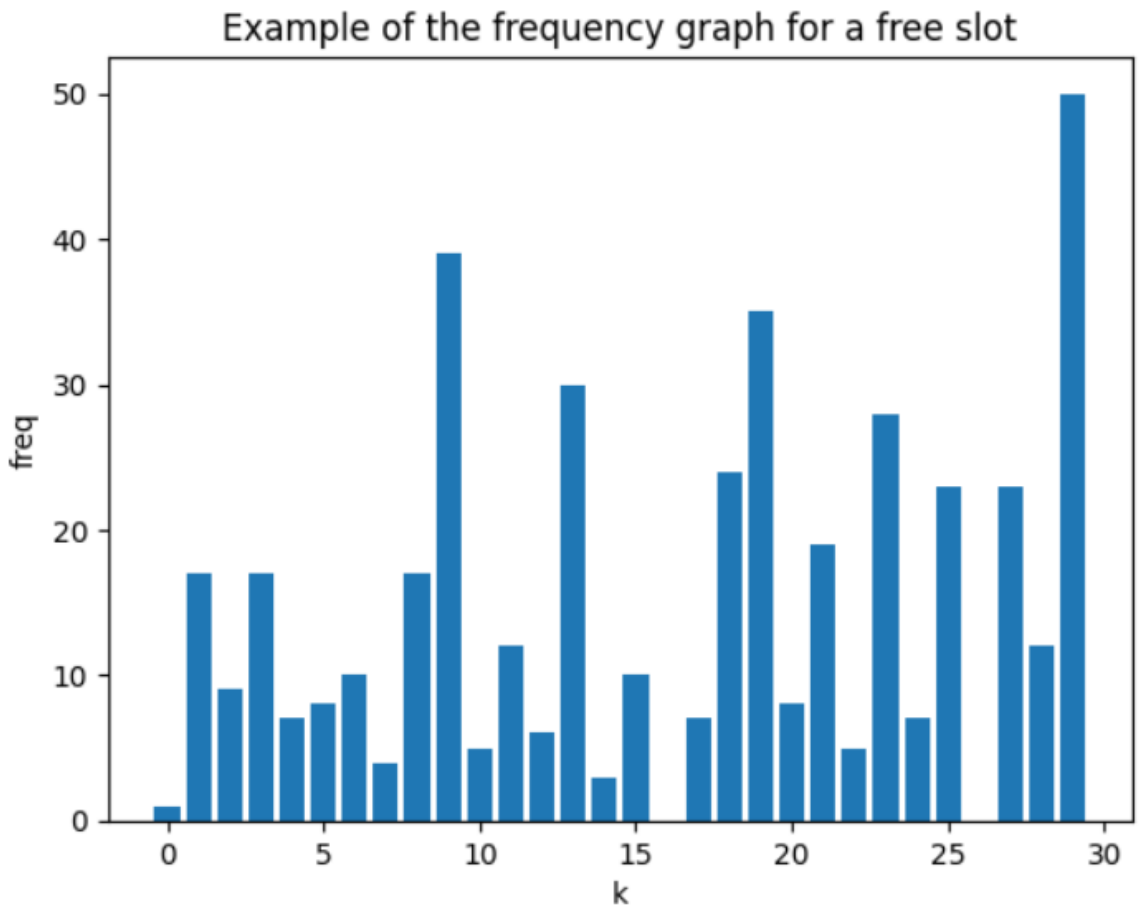
Now that I have the centroids, using *scipy.cluster.vq* I can do the vector quantization and identify if a feature of a busy image or free image belongs to which centroid. So, now I have two lists of visual words, one where are contained all the vectors of every busy slot and another one for free slots separately. From the vectors of visual words I create the frequency vectors, to express the frequency of how many features in that particular slot belongs to each centroids.

```
[[2. 3. 1. ... 6. 1. 2.]
 [8. 1. 0. ... 6. 5. 2.]
 [9. 6. 8. ... 5. 6. 6.]
 ...
 [2. 5. 5. ... 2. 8. 1.]
 [1. 3. 1. ... 0. 1. 1.]
 [0. 1. 1. ... 0. 0. 0.]]
```

Stack of frequency vectors

While doing so, I also fill a list of labels with *1s* for each element of the *visual_words_B* list and with *0s* for elements in *visual_words_F*, in such a way as to use it for the training algorithm.


Example of the frequency graph for a busy slot
freq
k
0
2
4
6
8
10
12
0
5
10
15
20
25
30

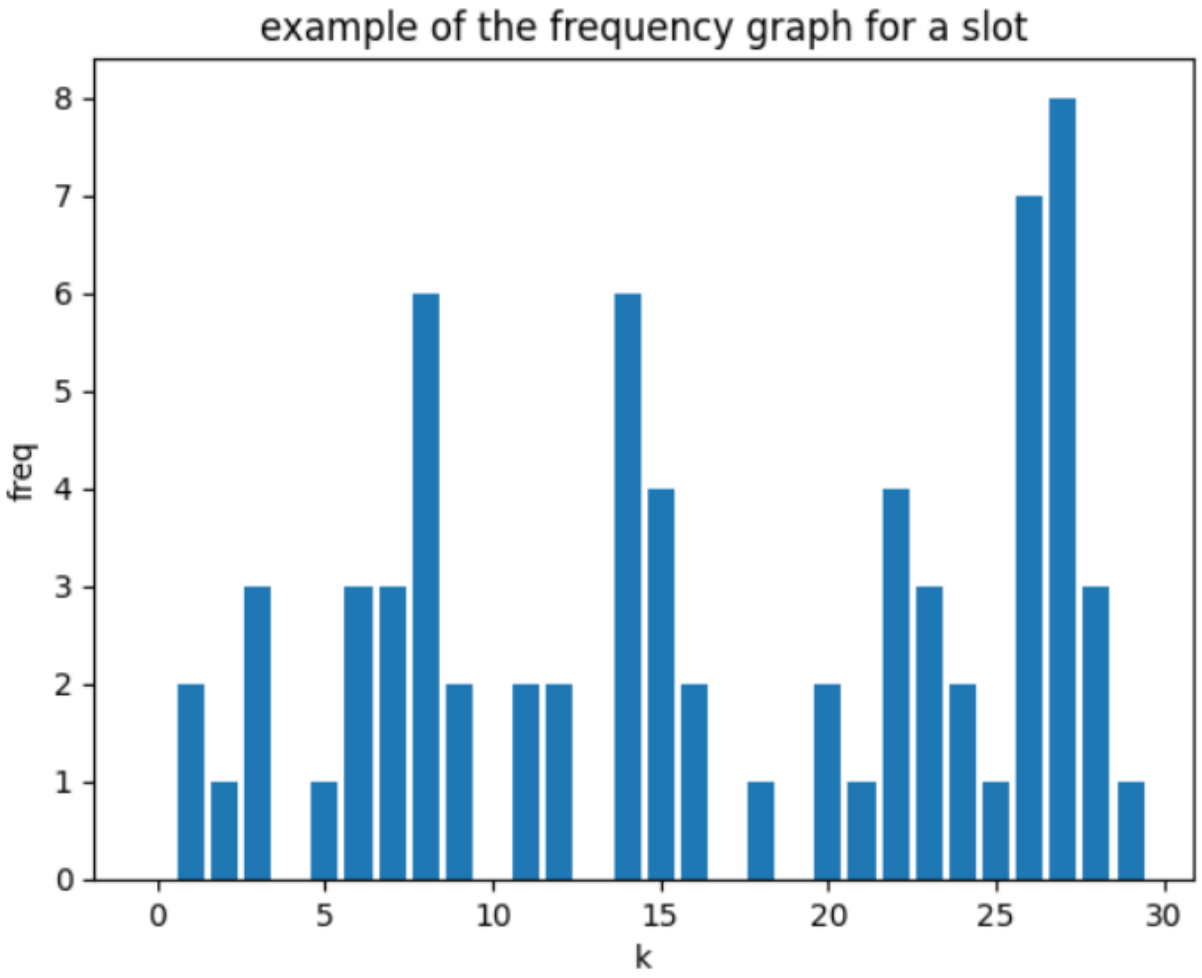


The graphs indicate the frequency of each feature in each of the 30 clusters of the kmeans algorithm

Now, by using *train_data, test_data, train_labels, test_labels = train_test_split(freq_vectors_BF, labels, test_size=0.2, random_state=42)* I split the frequency vectors of the mixed images in a 80% for the training part and a 20% for the test part of the machine learning algorithm. I use a SVC classifier and by doing a *.fit* I start the training for the classifier and then it makes a prediction (*.predict*). To evaluate the quality of the performance, when it finishes it gives an accuracy score (*test_score = accuracy_score(predictions, test_labels)*).

```
Accuracy on test data:  0.9571068124474348
```

To conclude, after finishing the training of the classifier I build a *visual_words_S* list also for the slots of the picked image that I want to classify and a *frequency_vectors_S.*



And now I can make predictions with *frequency_vectors_S* (*predictions = svc_classifier.predict(frequency_vectors_S*). It gives the accuracy score and a confusion matrix, which tells the number of true positives (instances that belong to the positive class and are correctly predicted as positive), true negatives (instances that belong to the negative class and are correctly predicted as negative), false positives (instances that belong to the negative class but are incorrectly predicted as positive), false negatives (instances that belong to the positive class but are incorrectly predicted as negative) found.

```
predictions:  [0 0 1 0 1 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 1 1 1 0 1 1
 1 1 1 1 1 1]
ground truth:  [1, 1, 0, 1, 0, 1, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 0, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1]
Accuracy: 0.7674418604651163
Confusion matrix:
[[ 2  5]
 [ 5 31]]
```

```
predictions:  [0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
37
ground truth:  [0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
37
Accuracy: 1.0
Confusion matrix:
[[36  0]
 [ 0  1]]
```

```
predictions:  [1 0 1 1 1 1 1 1 0 1 1 0 1 1 1 1 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 0 1 1 0
 0 0 0 0 1 0 0 0 1]
46
ground truth:  [1, 0, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
46
Accuracy: 0.7608695652173914
Confusion matrix:
[[ 3 11]
 [ 0 32]]
```

```
predictions:  [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0]
54
ground truth:  [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0]
54
Accuracy: 1.0
Confusion matrix:
[[51  0]
 [ 0  3]]
```

Measure of quality

The accuracy given by the machine learning algorithm are pretty good especially the one of the training (circa 96%). As far as the accuracy score for the slots is concerned, it swings from 77% to 100% for these four particular images, but the number of images that can be taken can be far more higher than this and by using more images there will be a better visualization of the average accuracy score.

More general approach

Since I recognize that taking only four images can be narrow, I implemented also the idea for taking all (or at least an high percentage of images from the dataset) dynamically (*proCV.ipynb*).

The training part remains the same as before and with the same accuracy score. What it changes is that now I take from every weather condition and every camera and data, 50 (or a variable number as I prefer) images and I collect them in a dictionary that has the path of the image as a key and the matrix of the image as the value.

```
all_images = dict()
collection = []

weather = ["SUNNY", "OVERCAST", "RAINY"]
id_camera = range(1,10)

for w in weather:
    for id_cam in id_camera:
        imgs_path = "D:\CNR-EXT_FULL_IMAGE_1000x750\FULL_IMAGE_1000x750\%s\*\camera%d\*.jpg" % (w,id_cam)
        col = imread_collection(imgs_path)
        col = col[:50]
        collection.extend(col)
        for im,path in zip(col,col.files):
            all_images.update({path: im})
```

Now, I repeat the idea for finding the slots' coordinates as before, comparing, with the parsing of a string, the number of camera used for each image.

```
slots = dict()
for id_cam in range(1,10):
    camera_path = "D:\CNR-EXT_FULL_IMAGE_1000x750\camera%d.csv" % (id_cam)
    df = pd.read_csv(camera_path, sep = ";")

    l = camera_path.split("\\")
    camera_num = l[2][6]

    for path in list(all_images.keys()):
        s = path.split("\\")
        camera_path_num = s[5][6]

        if camera_path_num == camera_num:
            img = all_images[path]
            slots[path] = []
            x = df['X'][0]
            for i in range(0, df.shape[0]):
                x = int((df['X'][i])/2.592)
                w = int((df['X'][i] + df['W'][i])/2.592)
                y = int((df['Y'][i])/2.592)
                h = int((df['Y'][i] + df['H'][i])/2.592)
                bbox = img[y:h, x:w]
                bbox = cv.resize(bbox, (150, 150), interpolation=cv.INTER_LINEAR)
                slots[path].append(bbox)
```

Finally, the other change from the previous case is the implementation of the ground truth. Now, I don't have to create it manually but using the "*all.txt*" file in the dataset I can control if every slot connected to the path of its image is 1 (busy) or 0 (free). Moreover, since the slots of every image in the file were not in order, I thought of using as cameras in the slot detection part, the csvs modified from the lowest slot id to the highest and compare it with the ones in the ground truth.

Also in this case I used the parsing of strings to do the comparison.

```
import collections

slo = dict()
ground_tr = dict()
with open('all.txt', 'r') as file:
    lines = file.readlines()
    for line in lines:
        lines_split = line.split("/")
        weather = lines_split[0]
        date = lines_split[1]
        camera = lines_split[2]

        for path in list(all_images.keys()):
            s = path.split("\\")
            weather_path = s[3]
            date_path = s[4]
            camera_path = s[5]
            hp = s[6]
            h_p = hp.split(".")
            hour_path = h_p[0]
            hour_path = hour_path[-4:-2]+"."+hour_path[-2:]

            if weather_path == weather and date_path == date and camera_path == camera and hour_path in line:
                slot_id = line.split(".jpg")
                slot_id = slot_id[0][-3:]
                s = line.split(" ")
                bf = s[1]
                bf = bf.split("\n")
                bf = bf[0]

                slo[slot_id] = bf
            ground_truth = collections.OrderedDict(sorted(slo.items())).values()
            ground_tr[path] = ground_truth
```

The rest of the procedure remains pretty much the same as paragraph two.

Measure of quality

Unfortunately, this very last approach gives me problems in the use of the ground truth, even though I cannot find the issue in the idea I've implemented. In any case, as seen in the main approach (second paragraph) the algorithm works and gives good accuracy scores so I think that if the ground truth in this case would be implemented better it should work as it works with the four images.

References

For the implementation of the bag of words approach I referred to
https://www.pinecone.io/learn/series/image-search/bag-of-visual-words/

Whereas for the kmeans and training part of the classifier SVC I used the laboratories of the course of Machine Learning of professor D. Santoro that I’ve attended last year.