Computer Vision - Exercise 2
Object Recognition

In this exercise, we will implement:

- A K-means clustering algorithm that partitions an image into multiple segments.
- A bag-of-words image classifier that decides whether a test image contains a car (back view) or not.

The data is contained in the `data` folder. Your task is to complete missing parts of the provided code as described in this handout and in the code itself.

1 Environment Setup

For setting up the Python environment, we recommend using miniconda. Please install the latest miniconda version for your OS from the following link https://docs.conda.io/en/latest/miniconda.html. Once the miniconda is installed, you can run the following commands to create and activate the environment:

```
conda create -n ex2 python=3.10

conda activate ex2

pip install scikit-learn opencv-python matplotlib tqdm
```

2 K-means Clustering (40%)

Your task is to fill the missing code (all the `#TODO`s) in `kmeans.py` script.

2.1 Initialization (5%)

Choose a value for K (number of clusters) and randomly initialize K centroids from the pixel values.

2.2 Point Distance Calculation (5%)

Implement a function to calculate Euclidean distance between two points (used to assign points to the nearest centroid).

2.3 K-means Iteration (20%)

Implement the K-means algorithm. They should iteratively:

- Assign each pixel to the closest centroid.
- Recompute the centroids based on the mean of the assigned pixels.

- Stop when centroids no longer change (convergence) or after a maximum number of iterations.

2.4 Experiment with Different Values of K (10%)

To understand how the number of clusters affects the segmentation quality and computational performance,

- Run the K-means algorithm with different values of K, such as 2, 3, 5, 10, etc.
- Visualize the segmented image for each value of K and compare the results.

Please report your experimental observations (*e.g.*, what happens to the segmented image as K increases and how does different values of K affect the computational cost?).

3 Bag-of-words Classifier (60%)

You need to fill the missing code (all the `#TODO`s) in `bow.py` script.

3.1 Local Feature Extraction (20%)

3.1.1 Feature detection - feature points on a grid (5%)

Implement a very simple local feature point detector: points on a grid. Within the function `grid_points`, compute a regular grid that fits the given input image, leaving a border of 8 pixels in each image dimension. The parameters `nPointsX` $= 10$ and `nPointsY` $= 10$ define the granularity of the grid in the x and y dimensions. The function shall return `nPointsX` $\cdot$ `nPointsY` 2D grid points. The function to be written takes the following form:

```
vPoints = grid_points(img, nPointsX, nPointsY, border)
```

3.1.2 Feature description - histogram of oriented gradients (15%)

Next, we describe each feature (grid point) by a local descriptor. We will use the well-known histogram of oriented gradients (HOG) descriptor. Implement the function `descriptors_hog` which should compute one 128-dimensional descriptor for each of the N 2-dimensional grid points (contained in the $N \times 2$ input argument `vPoints`). The descriptor is defined over a 4×4 set of cells placed around the grid point i (see Fig. 1).

For each cell (containing `cellWidth` $\times$ `cellHeight` pixels, which we will choose to be `cellWidth` $=$ `cellHeight` $= 4$), create an 8-bin histogram over the orientation of the gradient at each pixel within the cell. Then concatenate the 16 resulting histograms (one for each cell) to produce a 128 dimensional vector for every grid point. Finally, the function should return a $N \times 128$ dimensional feature matrix. The function to be written takes the following form:

```
descriptors = descriptors_hog(img, vPoints, cellWidth, cellHeight)
```

3.2 Codebook Construction (15%)

In order to capture the appearance variability within an object category, we first need to construct a visual vocabulary (or *appearance codebook*). For this we cluster the large set of all local descriptors from the training images into a small number of visual words (which form the entries of the codebook). For this, you will use the K-means clustering algorithm, where the basic steps are elaborated

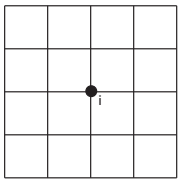

Figure 1: Grid point *i* and surrounding cells.

in Section 2. In this exercise, we can use the K-means algorithm in the `sklearn` library, which is implemented more efficiently.

Fill the function `create_codebook` which takes as input the name of a directory, loads each image in it in turn, computes grid points for it, extracts descriptors around each grid point, and clusters all the descriptors with K-means. It then returns the found cluster centers. Make sure to use all descriptors from all training images in the directory. The function to be written takes the following form:

```
vCenters = create_codebook(nameDir, k, numiter)
```

3.3 Bag-of-words Vector Encoding (15%)

Using the appearance codebook created at the previous step we will represent each image as a histogram of visual words. This representation, also known as the **bag-of-words** representation, records which visual words are present in the image.

3.3.1 Bag-of-Words histogram (10%)

Fill the function `bow_histogram` that computes a bag-of-words histogram corresponding to a given image. The function should take as input a set of descriptors extracted from one image and the codebook of cluster centers. To compute the histogram to be returned, assign the descriptors to the cluster centers and count how many descriptors are assigned to each cluster (i.e. to each 'visual word'). The function to be written takes the following form:

```
bowHist = bow_histogram(vFeatures, vCenters)
```

3.3.2 Processing a directory with training examples (5%)

Fill the function `create_bow_histograms` that reads in all training examples (images) from a given directory and computes a bag-of-words histogram for each. The output should be a matrix having the number of rows equal to the number of training examples and number of columns equal to the size of the codebook (number of cluster centers). The function to be written takes the following form:

```
vBoW = create_bow_histograms(nameDir, vCenters)
```

Using this function we process all positive training examples from the directory **cars-training-pos** and all negative training examples from the directory **cars-training-neg**. We collect the resulting histograms in the rows of the matrices `vBoWPos` and `vBoWNeg`.

3.4 Nearest Neighbor Classification (10%)

We build a bag-of-words image classifier using the nearest neighbor principle. For classifying a new test image, we compute its bag-of-words histogram, and assign it to the category of its nearest-neighbor training histogram. Fill the function `bow_recognition_nearest` that compares the bag-of-words histogram of a test image to the positive and negative training examples and returns the label 1 (car) or 0 (no car), based on the label of the nearest-neighbour. The function to be written takes the following form:

```
label = bow_recognition_nearest(histogram, vBoWPos, vBoWNeg)
```

4 Hand-in

Please hand in 1) all the code (excluding the `data` folder); 2) a short pdf report to explain your implementations and your results. For the bag-of-words classifier, please include the test accuracy of running `bow.py`.

Please zip all files (excluding the `data` folder) into one single file for submission.