10907 Pattern Recognition

Lecturers
Prof. Dr. Ivan Dokmanić ⟨ivan.dokmanic@unibas.ch⟩

Tutors
Alexandra Flora Spitzer ⟨alexandra.spitzer@stud.unibas.ch⟩
Roman Fries ⟨r.fries@unibas.ch⟩
Cheng Shi ⟨cheng.shi@unibas.ch⟩
Vinith Kishore ⟨vinith.kishore@unibas.ch⟩
Valentin Debarnot ⟨valentin.debarnot@unibas.ch⟩

Assignment 4

Deadline: 23.11.2023
Total points: 8

Academic integrity You are encouraged to discuss the material with others, but it is essential to document these discussions by writing down the names of the individuals you worked with and any tools that helped you with the assignment. Any instance of cheating will be dealt extremely strictly, potentially resulting in receiving zero credit for the course at a minimum.

FAQ

1. What should I do after uploading my code? Double or triple check! make sure it goes through all the test cases and a score appears. This score is not your final score because you can correct the reported errors and resubmit the code again for a higher score. This is why I highly encourage uploading your code early so there is enough time to debug.
2. What do I do when seeing an error from Gradescope? Let us know as soon as possible. It could be that your implementation take too long to run (timeout error).
3. What to do if the TA makes a mistake grading my solution? If you disagree with the score, you can request a Regrade on Gradescope and we will take a second look as soon as possible.
4. How to use Piazza forum effectively? Open a public thread whenever you have a question regarding the class, homework, Gradescope, etc. You can choose to post anonymously. Some of us will try to answer quickly.

Math

Exercise 1 (FIR Wiener filter 2 points)**.**

In class we derived a denoising Wiener filter which acts directly in the frequency domain. That is conceptually simple and the obtained expression is pleasingly interpretable. On the other hand, to apply this Wiener filter we always need to compute forward and inverse Fourier transforms. That can be a problem on constrained hardware in embedded devices and cameras. Even if it is not a problem, we cannot really control the filter's spatial support (the "size" of the kernel) and we should zero pad to avoid issues with circular convolution when doing the inverse transform.

It is thus often desirable to compute a convolutional Wiener filter which has an impulse response (a kernel) of fixed length. This means that we have to further restrict the class of estimators g discussed in class. We will work with 1D audio signals but the ideas extend to images.

Formally, let $\mathbf{y}$ be a noisy version of the audio signal $\mathbf{x}$ of length N, corrupted with i.i.d. zero-mean Gaussian noise of per-sample variance σ^2. In other words, $y[n] = x[n] + w[n]$ for zero-mean i.i.d. Gaussian noise $w[n]$ with per-sample variance σ^2. To compute the linear MMSE estimator we were solving

$$\min_{\mathbf{H}} \ \mathbb{E} \ \|\mathbf{H}(\mathbf{y} - \mu_Y) + \mu_X - \mathbf{x}\|^2.$$

The expectation averages over the joint distribution of $\mathbf{x}$ and $\mathbf{y}$ (or equivalently over $\mathbf{x}$ and the noise). We then restricted $\mathbf{H}$ to be a convolution and we obtained the Wiener filter in the frequency domain.[1] Now we further restrict $\mathbf{H}$ to be a convolution with a filter $\mathbf{h}$ of fixed size $L + 1$, i.e. $\mathbf{H}\mathbf{y} = \mathbf{y} * \mathbf{h}$. For simplicity we'll assume that $\mu = 0$ but that's easy to revert. (It's definitely true for audio recorded on common hardware which does not let the DC component pass.) The filter size is often much smaller than the signal length, $L \ll N$. We will use the explicit convolution notation and denote the filter

$$\mathbf{h} = \begin{bmatrix} h[0] \\ h[1] \\ \vdots \\ h[L] \end{bmatrix}.$$

The filtered signal is then

$$\widehat{x}[n] = y[n] * h[n] = \sum_{m=0}^{L} h[m] y[n-m].$$

We want to find $\mathbf{h}$ which minimizes the MSE *for each* n (optimality of this follows from certain stationarity assumptions),

$$\mathbb{E} \ [(\widehat{x}[n] - x[n])^2] = \mathbb{E} \left[\left(x[n] - \sum_{m=0}^{L} h[m] y[n-m] \right)^2 \right].$$

1. Starting from the above expression, show that the coefficients of the optimal filter satisfy

$$\mathbb{E}\Big[(x[n] - \widehat{x}[n])\, y[n-\ell] \Big] = \mathbb{E} \left[\left(x[n] - \sum_{m=0}^{L} h[m] y[n-m] \right) y[n-\ell] \right] = 0, \quad \ell = 0, 1, \ldots, L.$$

   Hint: differentiate the expected error with respect to $h[\ell]$.

2. For each ℓ between 0 and L, the above expression yields a linear equation for the filter coefficients. We thus obtain $L + 1$ linear equations in $L + 1$ unknowns which can be succinctly written as

$$\boldsymbol{R}\boldsymbol{h} = \boldsymbol{r}.$$

[1] If you are reading this statement before Friday, then it is anti-causal.

Find the $(L+1) \times (L+1)$ matrix $\boldsymbol{R}$ and the vector $\boldsymbol{r}$. *Hint: you should see some covariances (expectations of products) pop up.* Armed with these expressions, we'll easily implement the filter in code in Exercise 6.

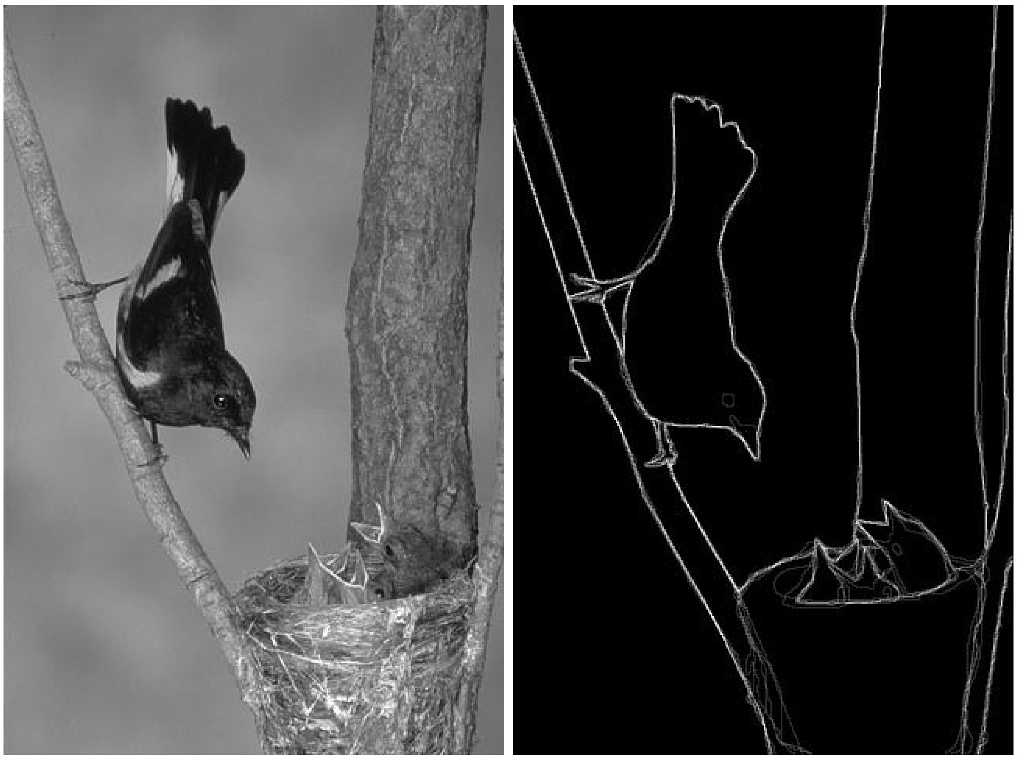

Figure 1: An example image and the corresponding human-marked boundaries.

Coding

Exercise 2 (Boundary Detection - 3 + 1 + 2 points)**.**

Boundary classification/detection aims at identifying the boundary of large scale objects in an image, unlike edge detection, which mainly concerns changes in low-level features of an image, such as brightness or color. Figure 1 illustrates an example from the Berkeley Segmentation Dataset[2]. In this exercise, you will build various components required to create an accurate boundary detector. This assignment can be broadly divided into three parts

1. Feature Extraction
2. Data Preprocessing
3. Training and Evaluation

Intuitively, the neighborhood of a pixel contains information that is useful to discriminate whether the pixel is a boundary pixel or not. There has been much progress in designing such feature extractors. You will implement three gradient-based feature extractors presented in [1]. These represent brightness, color, and texture information from the image.

Gradient Based Extractor: The underlying method in all the feature extraction processes involves observing local changes in the image features. At location (x, y) in an image, consider a circular patch with of radius r that divides the patch into two parts along the diameter at orientation θ. The gradient function $G(x, y, \theta, r)$ compares the features between the two halves of the disk. If there is a significant difference between the features in the two halves, it indicates the presence of some discontinuity, whether it's a boundary or something else. The difference is compared in terms of the feature distribution between the two halves. To compare the distribution, assume that the image feature is discretized into n levels (n bins in the histogram). We

[2]`https://www2.eecs.berkeley.edu/Research/Projects/CS/vision/grouping/segbench/`

define the distance between the histograms g, h in the two halves as:

$$\chi^2(g,h) = \frac{1}{2}\sum_{i=1}^{n} \frac{(g_i - h_i)^2}{g_i + h_i}. \tag{1}$$

Observe that this metric assumes no relation across the levels. However, for certain features such as brightness and color gradient, there is a relationship between the discrete levels. This relationship can be obtained using the function `colorsim()` in the script `feature.py`. The relationship between the n levels is represented as a matrix D of size $n \times n$. The distance, accounting for this relationship, is calculated as follows:

$$\chi_D^2(g,h) = (g-h)^{\mathrm{T}} D (g-h) \tag{2}$$

The two halves can be at different orientations depending on the variable θ. Your task is to first complete the functions in `gradient_feature.py`, which are further used in other feature extractors. The functions to be completed are:

1. `compute_histogram_gradient()`: Compute the gradient using the discretized image feature for all the pixels. To complete the function efficiently, follow these steps:
   (a) Generate a filter consisting of a circular mask of radius r. Thus, your filter has a size of $(2r+1) \times (2r+1)$ with 1's inside the disk and 0 outside.
   (b) Generate two masks from the circular mask, namely the left and right masks, at orientation θ. For reference, Figure 2 shows the left and the right masks oriented at four different angles θ, $\theta = 0°$, $\theta = 45°$, $\theta = 90°$ and $\theta = 135°$. We use these masks to estimate the histograms.
   (c) The image feature is of size $H \times W$ and is discretized into n levels, each pixel location will have one of the n values (the values can be integers $\{0, \ldots, n-1\}$ indicating the features or the discretized feature itself). Now, obtain a binary image from the feature image, such that the value at location (x, y) is 1 if the feature image contains the i^{th} feature value, else 0. Using convolution, you can obtain the number of times this value has occurred around the neighborhood of the pixel located at (x, y). You can use the left and right masks to obtain the number of times the feature occurs on the two sides. Repeat this process for all the n discrete values and store the filtered output. This gives us the histogram of the features.
   (d) Using the counts from the filtered output, compute the distance (gradient feature) using Equation (1) if the variable `similarity_map` is 'None', else use Equation (2) along with similarity map.

2. `cgmo()`: Compute `n_orient` number of equally spaced angles between $[0, \pi)$ used to generate the left and right masks. Use the function `compute_histogram_gradient()` to generate the features and return them.

The `GradientFeature` class provides us with the underlying functions used to generate the brightness, color, and texture gradient features.

Feature Extraction: Now, complete the functions in the file `feature.py`, which output the required gradient-based features for a given image. Note that the class `FeatureExtractor` in the file is initialized with the variable `n_bins`, which indicates the number of discretization levels, and `texton_features`, which contains a pre-computed texture filter (Leung-Malik filter bank) and the corresponding centroids used to obtain texture features (texton maps) from the image. The functions to be completed are:

1. `brigthness_gradient()`: This function takes the image as input and performs the following:
   (a) Convert the image to grayscale by averaging across the channels. If the image is already grayscale, no need to perform this operation.

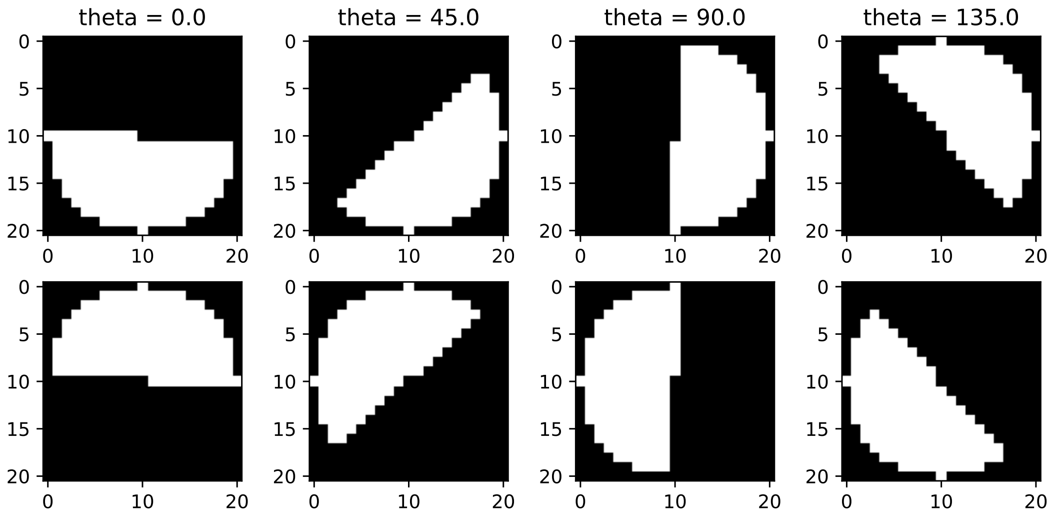

Figure 2: Left and Right masks at different orientations

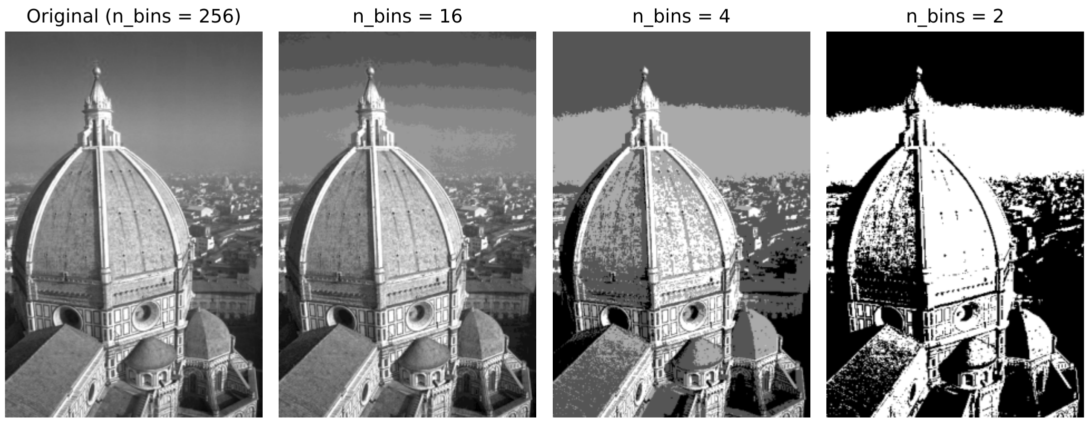

Figure 3: Discretized image at different levels

(b) Digitize the image into `n_bins` discrete levels. You can use the `np.digitize` function to do this using uniform levels to discretize the image. Figure 3 shows the discretization of the image at different levels.

(c) The radius is given in term of the percentage of the diagonal length of the image. Convert this number to the number of pixels. This is done by multiplying the radius with the diagonal length of the image and converting it into an integer.

(d) Compute the similarity between the levels using the function `colorsim` which takes the updated radius parameter as input.

(e) Use the function `cgmo` from the class `GradientFeature` to obtain the brightness gradient feature at the given number of orientations (`n_orient`).

2. `color_gradient`: Computes the color feature. Use the function `brigthness_gradient()` for each color channel. If the image is already grayscale, return the output from the `brigthness_gradient()` function.

3. `texture_gradient()`: Uses the Leung-Malik filter to obtain texture features, which are then used to obtain the texture gradient output. Thus, the function should:

(a) Convert the image to grayscale by averaging across the channels.

(b) The radius is given in term of the percentage of the diagonal length of the image. Convert this number to the number of pixels.

(c) Extract the filters and the corresponding centroids from the dictionary (`self.texton_features`). The centroids are used to discretize the filtered outputs so that they are suitable for the function `GradientFeature.cgmo()`. The dictionary contains two keys: 'fb' which corresponds to the filters, and 'tex' which corresponds to the centroids. Extract these filters and centroids.

(d) Use the function `extract_filters` in the file to convert the filters present in the variable to a list of filters.

(e) Use the function `filterbank_output()` to generate a filtered output. Note that this function uses `Filter.convolve2d_fft` from the previous assignment. Make sure to include this file.

(f) Using the centroids and image features, compute the cluster to which the feature at location (x, y) belongs. You can use the function `pairwise_distance()` which is already present in the file to help you with the discretization process. Now you should have an image feature of size $H \times W$ with values at each location as integers corresponding to the index of the centroid it belongs to.

(g) Use `GradientFeature.cgmo()` function to obtain the texture gradients. The input is the discretized texture image feature.

(h) (Optional) `extract_features()`: This function is used to combine different features. You can either experiment with the parameters used to generate the features. For example, if you choose to combine brightness gradient and texture gradient, each with 4 orientations and a radius of 0.01, then the output for an image is of size $H \times W \times 3$, which can be expanded to $H \times W \times 8$. This provides additional features that can be used to train the classifier.

Submit the outputs for the test image (`img_test`) already loaded in the script as a PDF file along with scripts. The expected outputs for the test image is provided as '.npy' files.

Data Preprocessing: This includes generating the labels and sampling the corresponding pixel-level features from the image in a format appropriate for training a logistic regression classifier. You need to use the logistic regression classifier from the previous assignment. Make sure to copy it to your current working directory.

Your task is to complete the functions in the file `data_loader.py`, which loads the boundary map and the image. The class `BSDS300Loader` is initialized with three variables:

1. `image_path`: Contains the path where the images are stored.
2. `seg_path`: Contains the path to the folder with the binary boundaries of the image.
3. `image_list`: Contains the names of the files in the image folder as integers.

Your task is to complete the following functions in the class:

1. `load_data()`: Given an integer, you need to load the image using `plt.imread` and then load the segmentation maps using the function `load_seg()`, which you will complete next.
2. `load_seg()`: Given the integer corresponding to the image file and the image shape, you need to check all the folders with human annotations and find the one corresponding to the image. Note that folders correspond to human subjects, and each image will have segmentations done by multiple human subjects. Your task is to load these segmentation maps in the form of a binary image, where 1 corresponds to a boundary and 0 to the other. Combine the segmentations for an image from different human subjects using unions. That is, a location (x, y) in the image is a boundary if it is considered a boundary by at least one of the human subjects. Carefully go through the documentation regarding the segmentation map provided in the link to convert the segmentation map to a binary image.

3. (Optional) `load_segmentation()`: Convert the `.seg` file to a binary image. However, we will evaluate the previous function directly rather than this function.

The next task is to sample the appropriate pixel-level features at location (x, y) and its corresponding segmentation labels. Note that since the features are computed by looking at the neighborhood, it is highly likely that we might get erroneous predictions for pixels very close to the boundary. Thus, while preparing the dataset, we choose pixels either on the boundary or at least a certain distance apart, which is defined by the variable `buffer_size` in the file `sample.py`. You can use the function `sample_data` in the file `sample.py` to generate such samples. It takes as input the image features and the corresponding image segmentation map and returns the data as variables `X`, `y`.

Training and Evaluation: Now we put all the above steps together and train a classifier to detect boundaries. We have provided you with a skeleton code by loading the required libraries in the file `boundary.py`. You can use `Metrics.py` from the assignment 2 to compute the precision recall curve. Make sure to copy it in your current working directory. We have provided you with necessary train and test path and an initial set of hyper-parameters we used. However you are free to change this or add some new ones. You are task here is to:

1. Choose a small set of images from the training set (> 5).
2. Generate samples form the set of images using the function `train_loader.load_data()`
3. Compute the mean and standard deviation and normalize the features.
4. Generate similar samples from the test data.
5. Normalize the test data using mean and standard deviation from the training data.
6. Train a logistic regression classifier.
7. Plot the precision recall curve.

Report the precision recall curve, accuracy and the hyper parameters used as a PDF file along with scripts.

Note:

1. You need to submit the following files:
   (a) `boundary.py`
   (b) `data_loader.py`
   (c) `feature.py`
   (d) `filter.py`
   (e) `gradient_feature.py`
   (f) `logistic.py`
   (g) `metrics.py`
2. Some of the scripts are manually graded while some of the scipts are automatically graded using the Autograder.

References

[1] David R Martin, Charless C Fowlkes, and Jitendra Malik. Learning to detect natural image boundaries using local brightness, color, and texture cues. *IEEE transactions on pattern analysis and machine intelligence*, 26(5):530–549, 2004.