Exercise 9

Classification

IAML 2020

Purpose These exercises will introduce you to the foundations of machine learning linear regression and logistic regression (classification) and you will work train, test and evaluate classifiers. You will also make Histogram of Oriented Gradients (HOG) feature descriptors. HOG features will be used to detect objects in images.

Notes

- All exercises are non-mandatory and we have purposefully added (hopefully) enough exercises to keep everyone busy. We mark less important exercises with ^{extra}. If you don't have time to do all exercises, wait until we release the solutions and use them as a guide.
- In the exercises based on script (i.e. `.py`) files, you are expected to implement code at certain points in the file. Although it is typically clear from context which piece of code you have to change, we have included specifier comments of the type `# <Exercise x.x (n)>`, where `x.x` is the exercise number and `(n)` is the letter associated with the specific task.

Exercise 9.1

Evaluating classifiers

In this exercise, you will work with classifier evaluation. Use the file `classification.py` to make your changes. The classification task we model is whether x is smaller (`Class 1`) or larger (`Class 2`) than 0. The script generates random data and adds noise to simulate measurement errors. It then trains a *Linear Regression*-classifier and *Logistic Regression* and plots the results as shown in Figure 1. The *Decision Boundary* is shown with red dashed lines hence separating the data into the two classes (at $y = 0.5$). The figure shows the results of the classification blue dots (`Class 1`) and orange dots (`Class 2`).

Your task is to evaluate the two classifiers using a number of metrics.

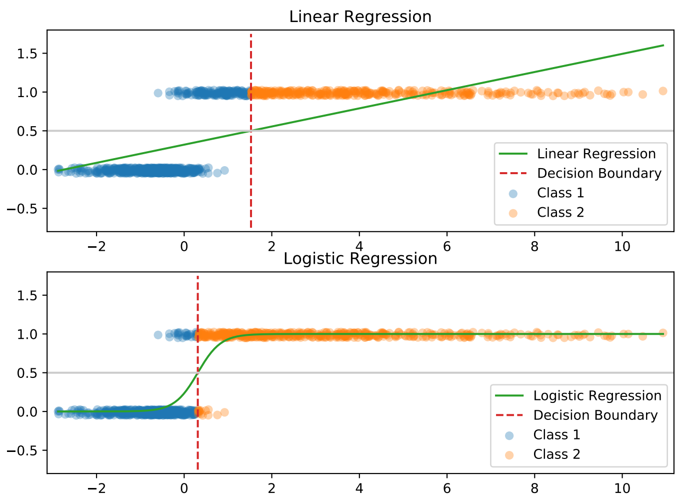

Figure 1: Binary classification using: (top) linear regression; and (bottom) logistic regression.

Table 1 specifies how to obtain different measures of the classifies using *false positives*, *false negatives*, *true positives*, and *true negatives*.

1. Implement the function `root_mean_squared_error(y_true, y_predict)` that calculates the RMSE (root mean squared error) and print the error for both models. This error describes the mean distance between actual and predicted points, thereby giving us a measure of the performance of our classifier. The RMSE can be calculated as follows:

$$RMSE(\hat{y}) = \sqrt{MSE(\hat{y})} = \sqrt{\frac{1}{N}\sum_{i=0}^{N}(\hat{y}_i - y_i)^2} \tag{1}$$

   where $\hat{y}$ is a vector of model predictions and y is a vector of ground truth values. You may already have used the MSE (mean squared error) in *exercise 3.05 (h)*. RMSE is simply the error in the same units as the output y.

Table 1: Confusion matrix or error matrix.

		Prediction			
		Positive	Negative		
Ground Truth	Positive	True Positive (TP)	False Negative (FN)	Recall ($\frac{TP}{TP+FN}$)	False Negative Rate ($\frac{FN}{FN+TP}$)
	Negative	False Positive (FP)	True Negative (TN)	False Positive Rate ($\frac{FP}{FP+TN}$)	True Negative Rate ($\frac{TN}{TN+FP}$)
		Precision ($\frac{TP}{TP+FP}$)	False Omission Rate ($\frac{FN}{FN+TN}$)		
		False Discovery Rate ($\frac{FP}{FP+TP}$)	Negative Predictive Value ($\frac{TN}{TN+FN}$)		

Hint: You can use numpy to operate on the entire vectors at once without having to make an explicit loop. Simply subtract the vectors, square them, and then take the mean and square root.

2. Evaluate the RMSE for both the linear and logistic classifiers (at the bottom of `classification.py`).

3. Implement the function `calculate_metrics(y_true, y_predict)` which calculates the four central metrics (**TP**, **FN**, **FP**, **TN**) as well as **Precision** and **Recall** metrics from Table 1 and returns them in a dictionary.
Hint: The function `confusion_matrix`[1] from the `sklearn` library can be used to calculate the confusion matrix (the central 2×2 matrix from Table 1). Use these values to derive the rest of the metrics.

[1] documentation link

4. Make the function `calculate_metrics` also return the accuracy:

$$accuracy = \frac{TP + TN}{TP + TN + FP + FN}.$$

5. Evaluate both *Linear Regression* and *Logistic Regression* classifiers through `calculate_metrics`.

6. What happens with *Linear Regression* and *Logistic Regression* if there is one large outlier in the data?

7. **Use a training and test dataset:**$^{(Extra)}$ In the previous exercise, you used the entire dataset of training and prediction.

   - Split the dataset into training and test data (e.g. 70% training data and 30% test data)
   - Calculate and compare the accuracy of the classifier for the training data, test data and the entire dataset.

Exercise 9.2

Histogram of Oriented Gradients (HOG)

In this exercise, you will use OpenCV to calculate HOG features. HOG features have successfully been used for various types of object detection via classification. We will use it next week for detection of pedestrians. Use the script `hog_descriptor.py` to answer this exercise.

Given an image (region), HOG features are calculated through the following steps:

1. Preprocessing / smoothing
2. Calculate image gradients for each pixel
3. Subdivide the image into $N \times N$ cells, and for each cell calculate a histogram of edge magnitudes and orientations
4. Normalize the cells across larger regions incorporating multiple cells (`blocks`).
5. Concatenate the histograms of each block into one feature vector.

Training data

Select one of these images in the folder `inputs` `peopleXY.jpg` . to answer the exercise and test your code.

1. **Select a Region of Interest (ROI):** The code to select a ROI in a Matplotlib window is provided in the script. It allows you to select a ROI with a fixed aspect ratio (1:2). Select a person in the input image, as shown in Figure 2.

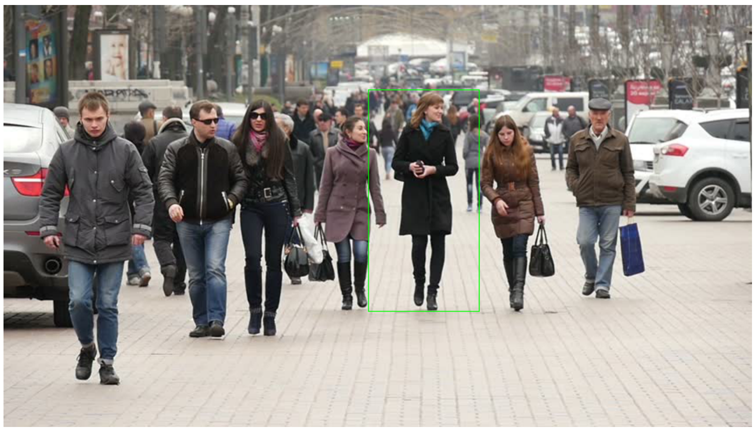

Figure 2: Image from the dataset.

   The image in in Figure 2, has a resolution of 852×480 pixels and the ROI has 126×252 pixels. The script crops the input image and resizes the ROI to 64×128 (i.e. variable `resized` in the code).

2. Select a person in the input image, and use the resized image in the following steps to calculate the HOG descriptor.

3. **Calculate the gradient:** Implement the function `computeGradients()` in which you will compute the gradient of the resized image. Use a Sobel filter to calculate the image gradient and gradient magnitude of the ROI. Use the function `cv2.Sobel()` [2] to detect the gradients through the following equations for horizontal (G_x) and vertical (G_y) gradients:

[2] An OpenCV function to calculate the first, second, third, or mixed image derivatives using an extended Sobel operator.

$$G_x = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix}, G_y = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix}.$$

Now that you have the gradient images, you can compute the final gradient magnitude $\|G\|$ image through:

$$\|G\| = \sqrt{G_x^2 + G_y^2},$$

and the orientation θ of the gradient for each pixel in the resized image through:

$$\theta = \arctan G_y, G_x.$$

Hint: Use the function `cv2.cartToPolar()` [3] to compute both magnitude $\|G\|$ and orientation θ. Figure 3 shows the edges (`magnitude`) detected using Sobel filters.

[3] An OpenCV function to calculate the magnitude and angle of 2D vectors.

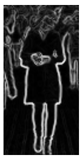
Figure 3: The gradients of the resized image.

4. **Define the number of cells:** The resized image is divided into $N \times N$ cells and a histogram of gradients is calculated for each $N \times N$ cells, as shown in Figure 4. By default the number of pixels per cell is 64 pixels, i.e. a 8×8 cells. We have provided a slider in which you can change the value of the variable `pixels_per_cell`.

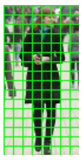
Figure 4: The resized image divided by 8×16 cells.

5. **Calculate the histogram:**$^{(Extra)}$ The code to display a polar histogram (e.g of edge orientations) is given in the exercise file (in Figure 5. The figure also contains a slider which can be used to change the value of variable `bins` that specifies the number of bins of the histogram. You have to change the script so it shows the histogram of the resized image.

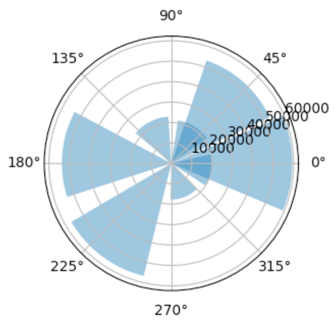

Figure 5: The histogram of egde magnitude and orientation of the entire resized image.

The gradient magnitude is sensitive to overall lighting which, in turn, can influence the feature descriptor. To account for light variations, it is useful to normalize the histogram by scaling the intensity of each cell to the interval of [0, 1].

6. **Define the cells per block:** A block is combined by a group of cells, e.g. 2×2 or 3×3 pixels, as shown in Figure 6. We have provided a slider in which you can change the value of the variable `cells_per_block`.

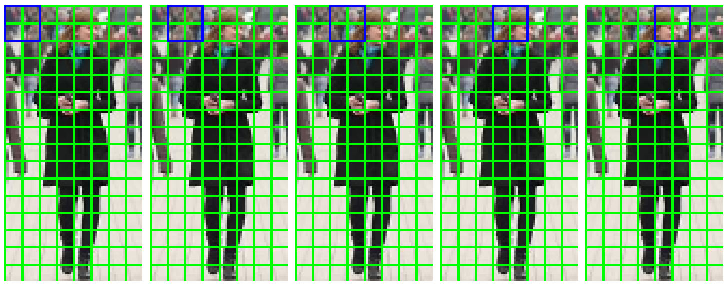
Figure 6: Five first steps of the histogram normalization using a 2×2 block size.

7. **Calculate the HOG descriptor:** After block normalization concatenate the resulting histograms into the final HOG feature descriptor (vector). Use the function `skimage.feature.hog()`[4] to compute the feature descriptor vector from the resized image. Use the command:

```
(H, hog) = feature.hog(image, orientations=bins,
    ↪ pixels_per_cell=(pixels_per_cell, pixels_per_cell),
    ↪ cells_per_block=(cells_per_block, cells_per_block),
    ↪ block_norm="L1-sqrt", visualise=True, feature_vector=
    ↪ True)
```

[4] An scikit-image function to extract the Histogram of Oriented Gradients (HOG) for a given image.

Figure 7 shows an example of HOG descriptor. The function `skimage.feature.hog()` also returns the feature descriptor vector (`H`), in which its size is equals to the number of: Bins $\times$ Block Columns $\times$ Block Rows $\times$ Cells in the Block.

Figure 7: Histogram of Oriented Gradients (HOG) for a given image.

8. **Test the HOG descriptor:** Test your implementation by selecting multiple people from the images available in our dataset (`peopleXY.jpg`). You have to compare (visually) the HOG feature of people and non-people images.