Assignment # 02 (Part 01)

Introduction to Image Analysis and Machine Learning
(Spring 2017)

April 5, 2017

Purpose: In this part of the assignment, you will be using the Histogram of Oriented Gradients (HOG) descriptor and a Linear SVM to characterize digits from the MNIST handwritten dataset. In second part of the assignment, you will determine user's gaze location on the identified digits using a Head Mounted Eye Tracker (HMET) scenario. For this purpose, an eye video, a scene video and a pre-calculated calibration matrix for these videos is provided in the assignment resources.

Task: Please download exercise resources from learnIT before starting the exercises.

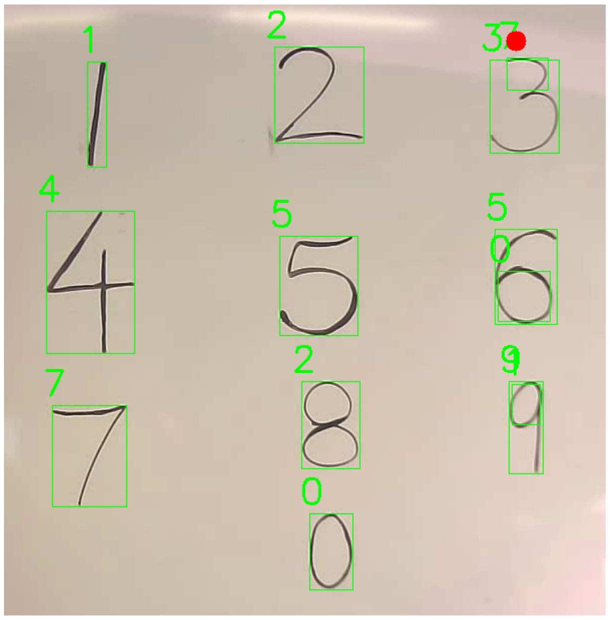

Figure 1: Digit Classification using HOG and SVM. The identified hand written digits are shown in the green bounding boxes while classified digit is written on top of respective boxes. You will see somw mis-classifications as well. The red dot shows estimated gaze position for this particular frame in the scene video.

Warm up - Plot different SVM classifiers in the iris dataset[1]

Purpose of this warm up example is to show the importance of parameters in the classifiers. Depending on the classification problem different parameters of a classifier can be fine tuned or changed to get improved results from given data. More simple examples were presented to you during the exercise session last week, where we implemented linear regression and logistic regression from scratch. We used different parameters such as learning rate, optimization iterations, train-test-split to minimize the error.
Similarly this example from scikit-learn is a comparison of different linear SVM classifiers on a 2D projection of the iris dataset. This data set consists of several attributes/features of the flowers. Only first 2 features of the dataset are considered: `Sepal length` and `Sepal width`.
This example shows how to use SVM classifiers with different kernels. LinearSVC uses the One-vs-All (also known as One-vs-Rest) multiclass reduction while SVC uses the One-vs-One multiclass reduction. **We will use LinearSVC (One-vs-Rest) in this assignment for handwritten digit classification.** Both linear models have linear decision boundaries (intersecting hyperplanes) while the non-linear kernel models (polynomial or Gaussian RBF) have more flexible non-linear decision boundaries with shapes that depend on the kind of kernel and its parameters.

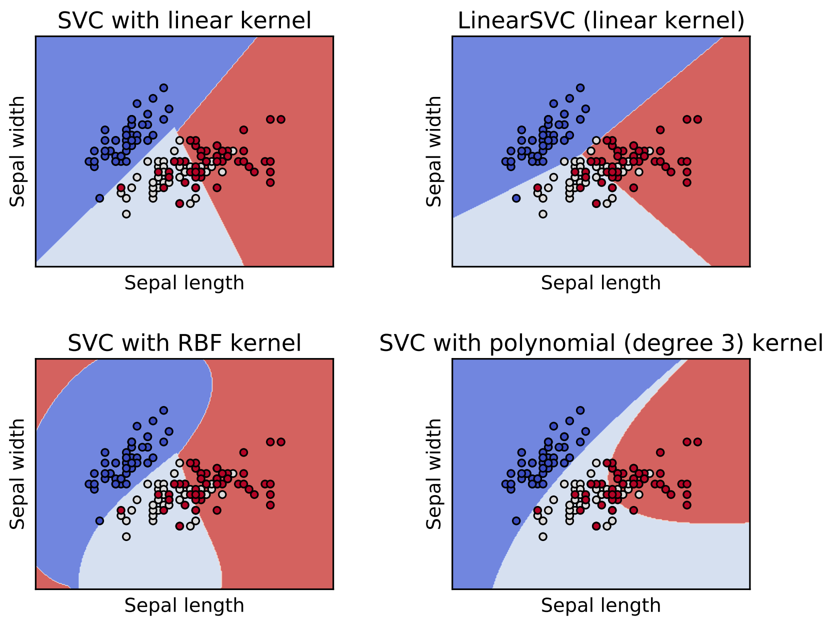

Figure 2:

[1]http://scikit-learn.org/stable/auto_examples/svm/plot_iris.html

Handwritten Digit Classification using HOG Descriptors and Linear Support Vector Machines

Similar to edge orientation histograms, HOG operates on the gradient magnitude representation of an image. HOG computes histograms on a dense grid of uniformly-spaced cells. Furthermore, these cells can also overlap and be contrast normalized to improve the accuracy of the descriptor.
HOG features have been used successfully, as mentioned in the lectures, in many areas of computer vision and machine learning but is especially noteworthy in the detection of people in images.

Learning Goal: After completing these exercises you should be able to:

- Extract HOG features from a sample of the MNIST handwritten digit dataset.
- Train a Linear SVM classifier on these HOG features.
- Extract handwritten digits from custom images.
- Apply the classifier to recognize each of these digit images.

You will be using your pupil detector from assignment 1 to extract pupil center as an eye-feature from given eye video. You will use extracted pupil center to approximate gaze location in the scene video using a pre-calculated calibration matrix. User calibration data is provided in the assignment resources.

Output: Your are required to submit videos (scenario 1 and 2) where you show classified digits bounded inside the boxes as shown in figure 1. You are also required to show the estimated gaze location in the scene video. Based on the gaze location the bounding box of the respective digits must change it's color (you may use euclidean distance to chose the closest digit when gaze point is not exactly into digit's ROIs. Address the questions asked in the exercises appropriately in the relevant parts of your assignment and final report.

Handwriting recognition

Before implementing the digit recognition steps, lets take a look at the project structure:
As you can see, there are two sub-modules in the main `DataAndDescription` module, namely `descriptors` and `utils`. The `descriptors` sub-module stores the implementation of the HOG descriptor, while `utils` stores `datasets.py`, a series of convenience functions that will help us load the MNIST dataset and pre-process digits prior to feature extraction.
Finally, we have two driver scripts, `train.py` and `classifiy.py`, that will be used to train the SVM and classify new data samples, respectively. There exist another script, namely `detect_pupil.py` where you will use your pupil detector from assignment 1 to extract pupil center from given eye videos.

Implementing HOG

The first step in implementing the handwritten digit recognizer is to create the HOG descriptor class.

Task: Open up the `hog.py` file. The constructor of HOG requires four parameters. The `orientations` parameter defines how many gradient orientations will be in each histogram (i.e., the number of bins in the histogram). The `pixelsPerCell` parameter defines the number of pixels that will fall into each cell. When computing the HOG descriptor over an image, the image will be partitioned into multiple cells, each of size `(pixelsPerCell x pixelsPerCell)`. A histogram of gradient magnitudes will be computed for each cell. After storing the arguments for the constructor, we define the `describe` method which requires only an image as input for which the HOG descriptor should be computed. Computing the HOG descriptor itself is handled by the `hog` method of the `feature` sub-package of `scikit-image`. We simply pass in the `number of orientations`, `pixels per cell`, `cells per block`, and whether or not `square-root normalization` should be applied to the image prior to computing the HOG descriptor. Finally, the resulting HOG feature vector is returned to the calling method.

Working with MNIST

In our next step, we need a dataset of digits that we can use to extract HOG features from and train our machine learning model. In this case, we will be using the `MNIST` digit dataset, a classic dataset in computer vision and machine learning literature. The sample of the dataset, that we are going to use, consists of 5,000 data points, each with a feature vector of length 784, corresponding to the 28 x 28 gray-scale pixel intensities of the image. But before we can start working with the digits directly, we need to define some methods to help us manipulate and prepare the dataset for feature extraction. Please go through `dataset.py` for these methods.
In order to load the MNIST dataset from disk, we define the `load_digits` method. This method requires only a single argument, `datasetPath`, which is the path to where the MNIST sample dataset resides on disk. From there, the `genfromtxt` NumPy function loads the dataset from disk and stores it as an unsigned 8-bit NumPy array. The first column of the data matrix contains labels, which is the digit that the image contains. The target will fall in the range [0, 9]. Likewise, all columns after the first one contain the pixel intensities of the image. These are grayscale pixels of the digit image of size M x N and will always fall in the range [0, 255]. Finally, the tuple of data and target are returned to the caller. Next up, we need a method to help us preprocess the digit images.
Everyone has a different writing style. While most of us write digits that lean to the left, some lean to the right. Some of us write digits at varying angles. These various angles can cause confusion for the machine learning models trying to learn representations of each digit 0-9. In order to help fix some of the lean of digits, we define the `deskew` method. This function takes two arguments: the image of the digit that is going to be deskewed, and the width of the image that we will resize it to. The skew is computed based on the moments and then the warping matrix is constructed. This matrix `M` will be used to deskew the image.

Training the Classifier

We are now ready to train the digit recognition model to recognize digits. We will use the `LinearSVC` model from `scikit-learn` to train a Support Vector Machine (SVM). We will also import the HOG image descriptor and dataset utility functions. Finally, `argparse` will be used to parse command line arguments, and `joblib` will be used to dump the trained model to file. Please go through `train.py` to get an idea that how we will train the classifier. The `train.py` script requires two command line arguments: `--dataset`, which is the path to the MNIST dataset residing on disk, and `--model`, the output path to the trained LinearSVC. The dataset consisting of the images and targets is loaded from disk. The data list is used to hold the HOG descriptors for each image. Next, we instantiate the HOG descriptor using `18 orientations` for the gradient magnitude histogram, `(10 x 10)` pixels for each cell, and `(1 x 1)` cell per block. Finally, by setting `normalize=True`, we indicate that the square-root of the pixel intensities should be computed prior to constructing the gradient magnitude histograms. We then start looping over the digit images. The HOG feature vector is computed for the pre-processed image by calling the `describe` method. Finally, the data matrix is updated with the HOG feature vector.
In the following lines, we instantiate the `LinearSVC`. The model is then trained using the data matrix and targets. We then dump the model to disk for later use using `joblib`.

Recognizing Digits

Now that the model has been trained, we can use it to classify digits in images or video frames. Please go through `classify.py`, in order to understand the classification process. Just as in the training phase, we require the usage of HOG for the image descriptor and dataset for pre-processing utility functions. The `argparse` package will once again be utilized to parse command line arguments, and `joblib` will load the trained LinearSVC from disk.
The `classify.py` driver script will require three command line arguments. The first, `--model`, is the path to where the serialized model is stored. The second, `--eye_video`, is the path to the eye video. The third, `--scene_video`, is the path to the scene video. (You may also use, `--image`, path to the image that contains digits for testing your code when you are at digit classification stage. In the next stage, trained `LinearSVC` is loaded from disk. Then, the HOG descriptor is instantiated with the exact same parameters as during the training phase.
The first step is to load the image off disk and convert it to gray-scale. From there, the image is blurred using Gaussian smoothing and the Canny edge detector is applied to find edges in the image. Finally, we find contours in the edge map and sort them. Each of these contours represents a digit in an image that needs to be classified. Given these digit regions, we now need to pre-process them.
We start looping over each of the contours. A bounding box for each contour is computed using the `cv2.boundingRect` function, which returns the starting `(x, y)-coordinates` of the bounding box, followed by the `width` and `height` of the box. We then check to see if the width and height of the bounding box are at least seven pixels wide and twenty pixels tall (try to experiment with different proportions if you have mis-classifications). The ROI of the digit is extracted from the gray-scale image using `NumPy` array slicing. The ROI now holds

the digit that is going to be classified. But first, we need to apply some more pre-processing steps. The first step is to apply Otsu's thresholding method to segment the foreground (the digit) from the background (the background the digit was written on).

Exercise 2.1.1: Now, when you have thresholded digit ROIs, perform following tasks to classify them.

1. Compute the HOG feature vector of the thresholded ROI. (Estimated time: 10-15 minutes)
2. Classify the digit ROI based on the HOG feature vector. (Hint: use `predict` method of the digit recognition model which is already learned at the training stage.) (Estimated time: 10-15 minutes)
3. Draw a bounding box surrounding the current digit ROI and then draw predicted digit on top of the bounding rectangle. (Estimated time: 10-15 minutes)
4. You see a lot of mis-classifications in the scene video around the original digits to be classified. Use some easy rules to remove these false positives from the list of contours. (Estimated time: 10-15 minutes)
5. Record final output video for the report. (Estimated time: 5 minutes)

Gaze Point Estimation

So far, you have segmented and classified the digits in the scene video using HOG descriptors and LinearSVM classifier. You might have noticed that these videos (eye and scene videos) are recorded using Head Mounted Eye Tracker (HMET). The subject was looking (Gazing) at different digits while videos was being recorded. Next up, we will estimate subject's gaze position on different digits in the scene video. We will be using 1). the eye-video for tracking and localization of `pupil center` which will be used as an eye-feature during the gaze estimation stage, 2). the calibration matrix for mapping pupil center to the scene image and 3). the scene video where we already have our classified digits. The function `DetectPupil()` in `detect_pupil.py` implements this functionality and returns pupil center coordinates `(PupilX, PupilY)`. In the next sage, we will use estimated pupil center and given calibration matrix to estimate gaze position the scene video. We will use `pickle.load(fileStream)` function to load the calibration matrix from file. We need to convert the pupil center to homogeneous coordinates before multiplying it to the calibration matrix. The returned gaze-point in the scene video is in normalized form. We will have to multiply it with image resolution to get the image coordinates in the scene video frames. The function `calculate_gaze_point()` implements this functionality in `classify.py` and returns the estimated gaze point.

Exercise 2.1.2:

1. Implement `DetectPupil()` function in `detect_pupil.py`. Use you pupil detector that you have already developed during assignment 1. In order to make your task easier, all

the code for creating ROI around eye is provided in `classify.py`. It should be helpful for reliable pupil detection with fewer rules. The `DetectPupil()` function takes this ROI as an input image. (Estimated time: 30-40 minutes)

2. Once you have gaze location in the scene image, display only those digits where user is looking at. Change the color of the bounding box as well. (Estimated time: 15 minutes)

3. Record the final video for your report. (Estimated time: 5 minutes)